Behavior Model of a 32-Bit Processor Using Verilog

Abdurrahman Mohammad

San Jose State University Computer Science Department

San Jose State University Santa Clara, United States

abdurrahman.mohammad@sjsu.edu

Abstract – This paper covers the design of an ALU, a 32x32 register file, a state machine, and a control unit. It also covers the simulation of the instruction set "CS 147DV" designed by Kaushik Patra involving the components mentioned in addition to a predefined 32x64 memory. The language used to model this processor is Verilog. This report explains how the listed components work together to implement "CS 147DV" and extensive testing.

I. Introduction

The purpose of this project is "to implement a behavioral model, using Verilog, of a processor supporting CS 147DV" (proj_02_guidelines). To do this, we must design the essential components involved in this processor and specify in the control unit the data path of each instruction in CS 147DV. The design of each component is elaborated in this report. The implementation of the three different types of instructions will also be explained as well as extensive testing of each instruction in CS 147DV.

II. General Information

Tools used:

1. An adequate computer
2. Windows 10 by Microsoft
3. ModelSim by Mentor Graphics
4. Started code and guidelines provided by SJSU professor Kaushik Patra

III. Components

The components involved in our processor are as such:

1. ALU: performs the arithmetic and logical operations in the processor
2. Memory: stores a large block of data, including instructions and stack data
3. Register File: stores a short block of quick, convenient, and easily accessible data
4. State Machine: transitions through the and manages the five states involved in the instruction cycle
5. Control Unit: generates the control signals involved in running the instructions
6. Clock: synchronizes the components to an electric rhythm of on and off pulses
7. Processor: contains the above components

IV. ALU Design and Testing

A. ALU Specifications

The function of an ALU (Arithmetic Logic Unit) is to perform the arithmetic and logical operations involved in executing the instructions in CS 147DV. Our ALU supports the following operations:

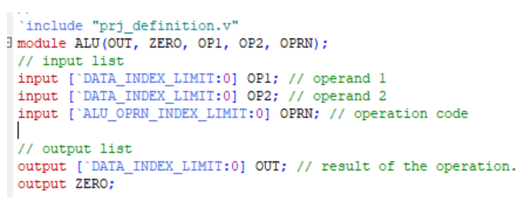
1. Addition
2. Subtraction
3. Multiplication
4. Bitwise AND
5. Bitwise OR
6. Bitwise NOR
7. Set less than
8. Shift left
9. Shift right

In addition to these operations, our ALU also supports the ZERO flag which is set when the output is 0.

B. ALU Design

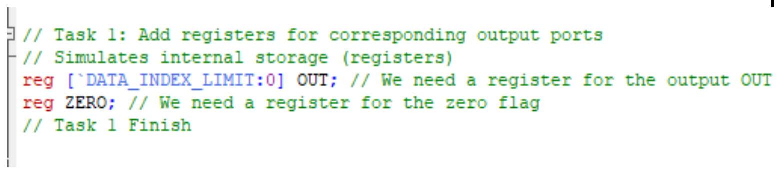
The steps involved in the ALU deign are as follows:

1. Create ports for input and output



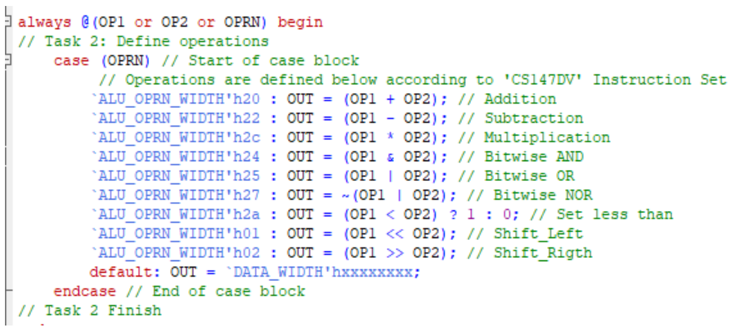
```
`include "prj_definition.v"
module ALU(OUT, ZERO, OP1, OP2, OPRN);
// input list
input [`DATA_INDEX_LIMIT:0] OP1; // operand 1
input [`DATA_INDEX_LIMIT:0] OP2; // operand 2
input [`ALU_OPRN_INDEX_LIMIT:0] OPRN; // operation code

// output list
output [`DATA_INDEX_LIMIT:0] OUT; // result of the operation.
output ZERO;
```

2. Add registers for corresponding output ports



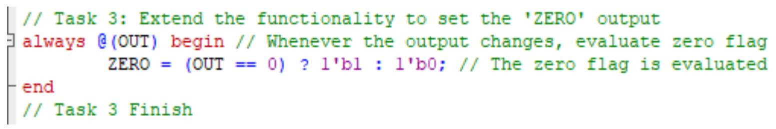
```
// Task 1: Add registers for corresponding output ports
// Simulates internal storage (registers)
reg [`DATA_INDEX_LIMIT:0] OUT; // We need a register for the output OUT
reg ZERO; // We need a register for the zero flag
// Task 1 Finish
```

3. Define the ALU functions corresponding to CS 146DV function codes



```
always @(OP1 or OP2 or OPRN) begin
// Task 2: Define operations
    case (OPRN) // Start of case block
         // Operations are defined below according to 'CS147DV' Instruction Set
        `ALU_OPRN_WIDTH'h20 : OUT = (OP1 + OP2); // Addition
        `ALU_OPRN_WIDTH'h22 : OUT = (OP1 - OP2); // Subtraction
        `ALU_OPRN_WIDTH'h2c : OUT = (OP1 * OP2); // Multiplication
        `ALU_OPRN_WIDTH'h24 : OUT = (OP1 & OP2); // Bitwise AND
        `ALU_OPRN_WIDTH'h25 : OUT = (OP1 | OP2); // Bitwise OR
        `ALU_OPRN_WIDTH'h27 : OUT = ~(OP1 | OP2); // Bitwise NOR
        `ALU_OPRN_WIDTH'h2a : OUT = (OP1 < OP2) ? 1 : 0; // Set less than
        `ALU_OPRN_WIDTH'h01 : OUT = (OP1 << OP2); // Shift_Left
        `ALU_OPRN_WIDTH'h02 : OUT = (OP1 >> OP2); // Shift_Rigth
        default: OUT = `DATA_WIDTH'hxxxxxxxx;
    endcase // End of case block
// Task 2 Finish
```

4. Add the ZERO flag capability

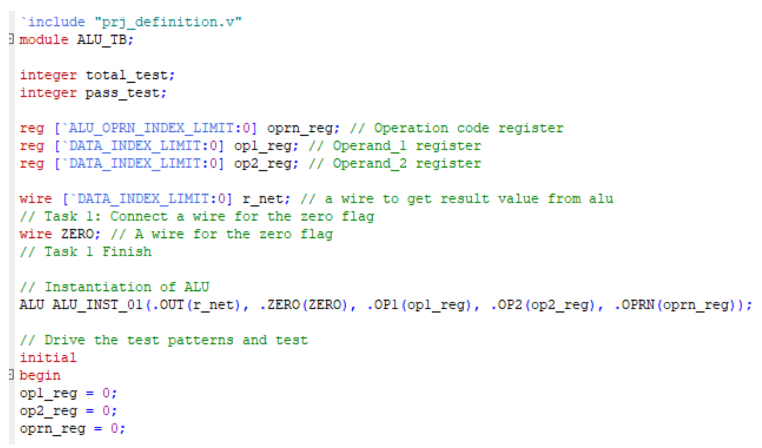


```
// Task 3: Extend the functionality to set the 'ZERO' output
always @(OUT) begin // Whenever the output changes, evaluate zero flag
        ZERO = (OUT == 0) ? 1'b1 : 1'b0; // The zero flag is evaluated
end
// Task 3 Finish
```

C. ALU Testing

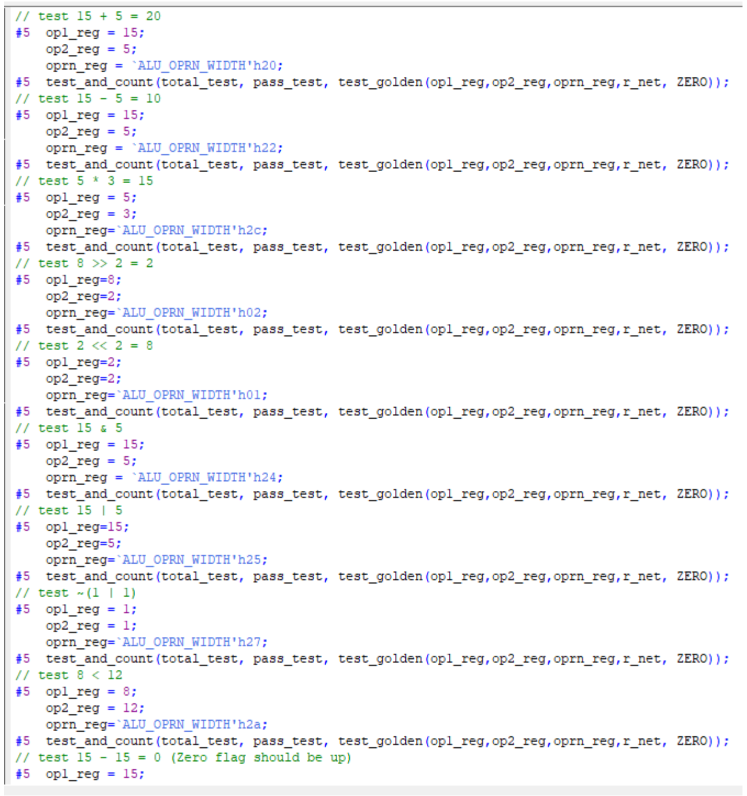
To test the ALU, create a test bench encompassing all the defined functions. Here is the test bench design:

1. Make a test bench file and instantiate an ALU. Create adequate input and output ports and add registers for ALU inputs. Add a wire to get the result and zero flag from ALU.



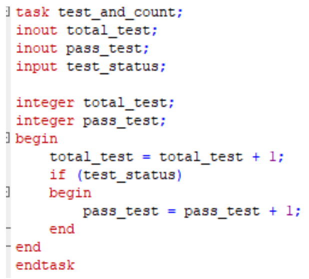
```
`include "prj_definition.v"
module ALU_TB;

integer total_test;
integer pass_test;

reg [`ALU_OPRN_INDEX_LIMIT:0] oprn_reg; // Operation code register
reg [`DATA_INDEX_LIMIT:0] op1_reg; // Operand_1 register
reg [`DATA_INDEX_LIMIT:0] op2_reg; // Operand_2 register

wire [`DATA_INDEX_LIMIT:0] r_net; // a wire to get result value from alu
// Task 1: Connect a wire for the zero flag
wire ZERO; // A wire for the zero flag
// Task 1 Finish

// Instantiation of ALU
ALU ALU_INST_01(.OUT(r_net), .ZERO(ZERO), .OP1(op1_reg), .OP2(op2_reg), .OPRN(oprn_reg));

// Drive the test patterns and test
initial
begin
op1_reg = 0;
op2_reg = 0;
oprn_reg = 0;
```

2. Test all the functions



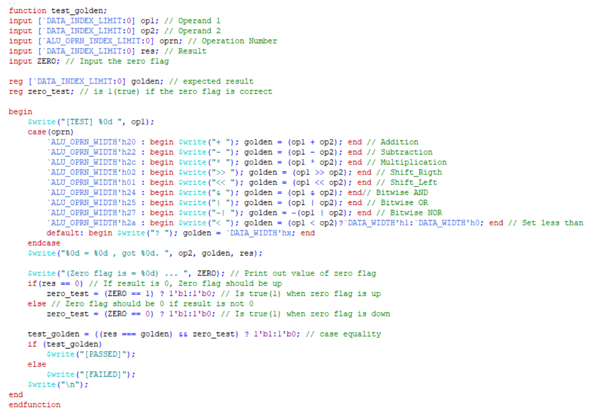
```
// test 15 + 5 = 20
#5  op1_reg = 15;
    op2_reg = 5;
    oprn_reg = `ALU_OPRN_WIDTH'h20;
#5  test_and_count(total_test, pass_test, test_golden(op1_reg,op2_reg,oprn_reg,r_net, ZERO));
// test 15 - 5 = 10
#5  op1_reg = 15;
    op2_reg = 5;
    oprn_reg = `ALU_OPRN_WIDTH'h22;
#5  test_and_count(total_test, pass_test, test_golden(op1_reg,op2_reg,oprn_reg,r_net, ZERO));
// test 5 * 3 = 15
#5  op1_reg = 5;
    op2_reg = 3;
    oprn_reg=`ALU_OPRN_WIDTH'h2c;
#5  test_and_count(total_test, pass_test, test_golden(op1_reg,op2_reg,oprn_reg,r_net, ZERO));
// test 8 >> 2 = 2
#5  op1_reg=8;
    op2_reg=2;
    oprn_reg=`ALU_OPRN_WIDTH'h02;
#5  test_and_count(total_test, pass_test, test_golden(op1_reg,op2_reg,oprn_reg,r_net, ZERO));
// test 2 << 2 = 8
#5  op1_reg=2;
    op2_reg=2;
    oprn_reg=`ALU_OPRN_WIDTH'h01;
#5  test_and_count(total_test, pass_test, test_golden(op1_reg,op2_reg,oprn_reg,r_net, ZERO));
// test 15 & 5
#5  op1_reg = 15;
    op2_reg = 5;
    oprn_reg = `ALU_OPRN_WIDTH'h24;
#5  test_and_count(total_test, pass_test, test_golden(op1_reg,op2_reg,oprn_reg,r_net, ZERO));
// test 15 | 5
#5  op1_reg=15;
    op2_reg=5;
    oprn_reg=`ALU_OPRN_WIDTH'h25;
#5  test_and_count(total_test, pass_test, test_golden(op1_reg,op2_reg,oprn_reg,r_net, ZERO));
// test ~(1 | 1)
#5  op1_reg = 1;
    op2_reg = 1;
    oprn_reg=`ALU_OPRN_WIDTH'h27;
#5  test_and_count(total_test, pass_test, test_golden(op1_reg,op2_reg,oprn_reg,r_net, ZERO));
// test 8 < 12
#5  op1_reg = 8;
    op2_reg = 12;
    oprn_reg=`ALU_OPRN_WIDTH'h2a;
#5  test_and_count(total_test, pass_test, test_golden(op1_reg,op2_reg,oprn_reg,r_net, ZERO));
// test 15 - 15 = 0 (Zero flag should be up)
#5  op1_reg = 15;
```

3. Create a task to keep track of the passes and failures.



```
task test_and_count;
inout total_test;
inout pass_test;
input test_status;

integer total_test;
integer pass_test;
begin
    total_test = total_test + 1;
    if (test_status)
    begin
        pass_test = pass_test + 1;
    end
end
endtask
```

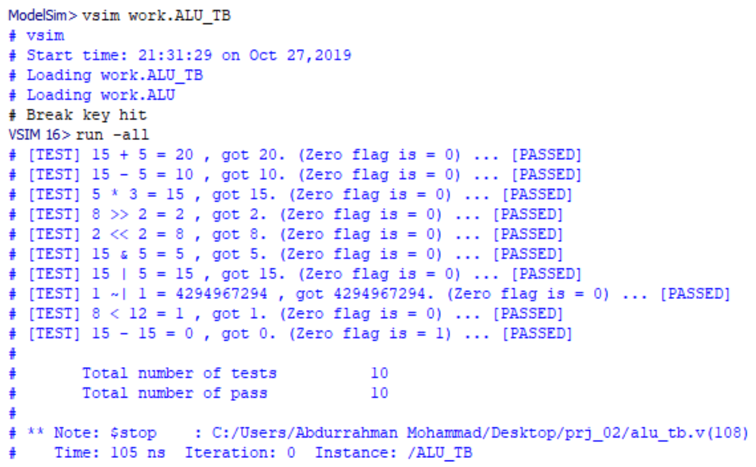
4. Create a function to evaluate the results from the ALU with the actual results of the operations



```
function test_golden;
input [`DATA_INDEX_LIMIT:0] op1; // Operand 1
input [`DATA_INDEX_LIMIT:0] op2; // Operand 2
input [`ALU_OPRN_INDEX_LIMIT:0] oprn; // Operation Number
input [`DATA_INDEX_LIMIT:0] res; // Result
input ZERO; // Input the zero flag

reg [`DATA_INDEX_LIMIT:0] golden; // expected result
reg zero_test; // is 1(true) if the zero flag is correct

begin
    $write("[TEST] %0d ", op1);
    case(oprn)
        `ALU_OPRN_WIDTH'h20 : begin $write("+ "); golden = (op1 + op2); end // Addition
        `ALU_OPRN_WIDTH'h22 : begin $write("- "); golden = (op1 - op2); end // Subtraction
        `ALU_OPRN_WIDTH'h2c : begin $write("* "); golden = (op1 * op2); end // Multiplication
        `ALU_OPRN_WIDTH'h02 : begin $write(">> "); golden = (op1 >> op2); end // Shift_Rigth
        `ALU_OPRN_WIDTH'h01 : begin $write("<< "); golden = (op1 << op2); end // Shift_Left
        `ALU_OPRN_WIDTH'h24 : begin $write("& "); golden = (op1 & op2); end// Bitwise AND
        `ALU_OPRN_WIDTH'h25 : begin $write("| "); golden = (op1 | op2); end // Bitwise OR
        `ALU_OPRN_WIDTH'h27 : begin $write("~| "); golden = ~(op1 | op2); end // Bitwise NOR
        `ALU_OPRN_WIDTH'h2a : begin $write("< "); golden = (op1 < op2)?`DATA_WIDTH'h1:`DATA_WIDTH'h0; end // Set less than
       default: begin $write("? "); golden = `DATA_WIDTH'hx; end
    endcase
    $write("%0d = %0d , got %0d. ", op2, golden, res);

    $write("(Zero flag is = %0d) ... ", ZERO); // Print out value of zero flag
    if(res == 0) // If result is 0, Zero flag should be up
        zero_test = (ZERO == 1) ? 1'b1:1'b0; // Is true(1) when zero flag is up
    else // Zero flag should be 0 if result is not 0
        zero_test = (ZERO == 0) ? 1'b1:1'b0; // Is true(1) when zero flag is down

    test_golden = ((res === golden) && zero_test) ? 1'b1:1'b0; // case equality
    if (test_golden)
        $write("[PASSED]");
    else
        $write("[FAILED]");
    $write("\n");
end
endfunction
```

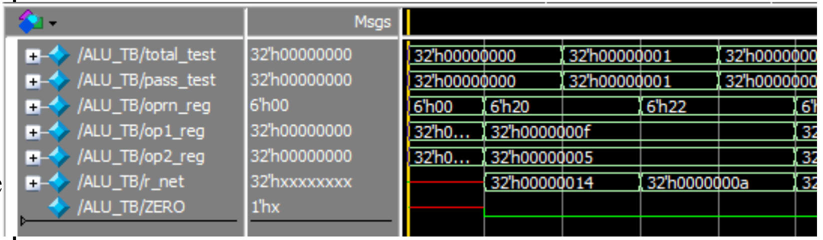
5. When you run the ALU test bench, you should get something like this:

Console output:



```
ModelSim> vsim work.ALU_TB
# vsim
# Start time: 21:31:29 on Oct 27,2019
# Loading work.ALU_TB
# Loading work.ALU
# Break key hit
VSIM 16> run -all
# [TEST] 15 + 5 = 20 , got 20. (Zero flag is = 0) ... [PASSED]
# [TEST] 15 - 5 = 10 , got 10. (Zero flag is = 0) ... [PASSED]
# [TEST] 5 * 3 = 15 , got 15. (Zero flag is = 0) ... [PASSED]
# [TEST] 8 >> 2 = 2 , got 2. (Zero flag is = 0) ... [PASSED]
# [TEST] 2 << 2 = 8 , got 8. (Zero flag is = 0) ... [PASSED]
# [TEST] 15 & 5 = 5 , got 5. (Zero flag is = 0) ... [PASSED]
# [TEST] 15 | 5 = 15 , got 15. (Zero flag is = 0) ... [PASSED]
# [TEST] 1 ~| 1 = 4294967294 , got 4294967294. (Zero flag is = 0) ... [PASSED]
# [TEST] 8 < 12 = 1 , got 1. (Zero flag is = 0) ... [PASSED]
# [TEST] 15 - 15 = 0 , got 0. (Zero flag is = 1) ... [PASSED]
#
#       Total number of tests          10
#       Total number of pass           10
#
# ** Note: $stop    : C:/Users/Abdurrahman Mohammad/Desktop/prj_02/alu_tb.v(108)
#    Time: 105 ns  Iteration: 0  Instance: /ALU_TB
```

Waveform output (cropped):



V. Memory Design

To design the memory module, you must:

1. Define the input and output ports.

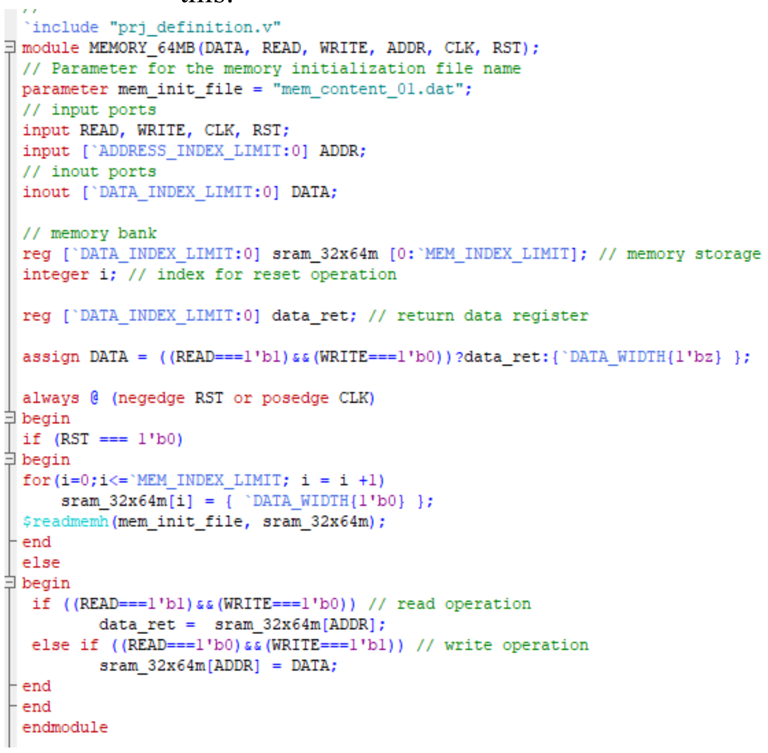
a. Input ports: READ, WRITE, CLK (clock), RST (reset signal)

b. Inout ports: ADDR (address in memory to access or modify), DATA (data located at address ADDR)

2. Create 64 count 32-bit registers to get your 32x64 memory file.

3. Add a return register to return data
4. When reset signal RST = 1, set all memory address data to 0.
5. You may have an option to preload memory from a file
6. Let the user of memory read data only when READ = 1 and WRITE = 0
7. Let the user write data only when READ = 0 and WRITE = 1

Overall, your memory design should look like this:



```
//
`include "prj_definition.v"
module MEMORY_64MB(DATA, READ, WRITE, ADDR, CLK, RST);
// Parameter for the memory initialization file name
parameter mem_init_file = "mem_content_01.dat";
// input ports
input READ, WRITE, CLK, RST;
input [`ADDRESS_INDEX_LIMIT:0] ADDR;
// inout ports
inout [`DATA_INDEX_LIMIT:0] DATA;

// memory bank
reg [`DATA_INDEX_LIMIT:0] sram_32x64m [0:`MEM_INDEX_LIMIT]; // memory storage
integer i; // index for reset operation

reg [`DATA_INDEX_LIMIT:0] data_ret; // return data register

assign DATA = ((READ===1'b1)&&(WRITE===1'b0))?data_ret:{`DATA_WIDTH{1'bz} };

always @ (negedge RST or posedge CLK)
begin
if (RST === 1'b0)
begin
for(i=0;i<=`MEM_INDEX_LIMIT; i = i +1)
    sram_32x64m[i] = { `DATA_WIDTH{1'b0} };
$readmemh(mem_init_file, sram_32x64m);
end
else
begin
 if ((READ===1'b1)&&(WRITE===1'b0)) // read operation
        data_ret =  sram_32x64m[ADDR];
 else if ((READ===1'b0)&&(WRITE===1'b1)) // write operation
        sram_32x64m[ADDR] = DATA;
end
end
endmodule
```

VI. Register File Design and Testing

A. Register File Design

The design of the register file is very similar to the design of out memory.

1. Define input ports

   READ – is 1 for read operation

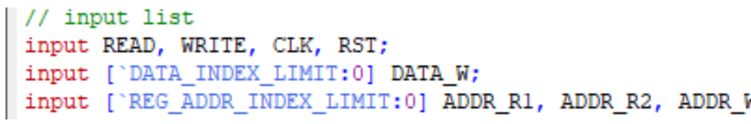
   Write – is 1 for write operation

   CLK – clock signal

   RST – reset signal to reset register file

   DATA_W – data to store

   ADDR_W – the address to store above

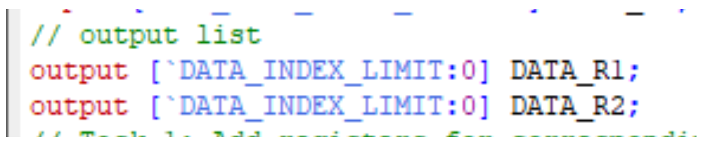


```
// input list
input READ, WRITE, CLK, RST;
input [`DATA_INDEX_LIMIT:0] DATA_W;
input [`REG_ADDR_INDEX_LIMIT:0] ADDR_R1, ADDR_R2, ADDR_
```

2. Define output ports

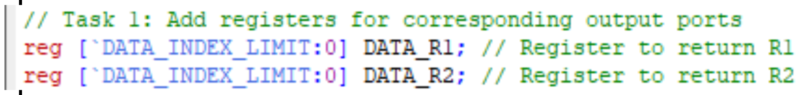
   ADDR_R1 – an address to read data

   ADDR_R2 – an address to read data



```
// output list
output [`DATA_INDEX_LIMIT:0] DATA_R1;
output [`DATA_INDEX_LIMIT:0] DATA_R2;
```

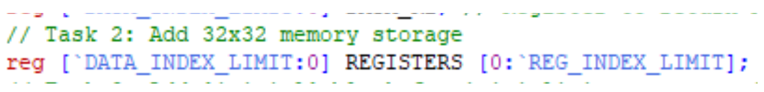
3. Add registers for corresponding output



```
// Task 1: Add registers for corresponding output ports
reg [`DATA_INDEX_LIMIT:0] DATA_R1; // Register to return R1
reg [`DATA_INDEX_LIMIT:0] DATA_R2; // Register to return R2
```

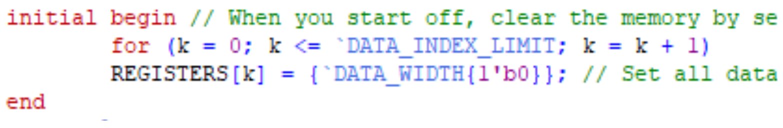
4. Define a 32x32 block of storage

   Add 32 count 32-bit registers



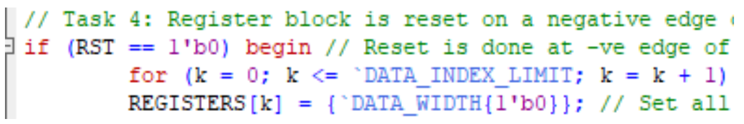
```
// Task 2: Add 32x32 memory storage
reg [`DATA_INDEX_LIMIT:0] REGISTERS [0:`REG_INDEX_LIMIT];
```

5. Define the initialization of the address values when the register file is



```
initial begin // When you start off, clear the memory by se
        for (k = 0; k <= `DATA_INDEX_LIMIT; k = k + 1)
        REGISTERS[k] = {`DATA_WIDTH{1'b0}}; // Set all data
end
```

6. If RST = 1, set all vales to 0.



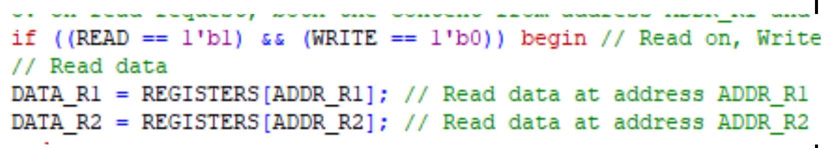
```
// Task 4: Register block is reset on a negative edge
if (RST == 1'b0) begin // Reset is done at -ve edge of
        for (k = 0; k <= `DATA_INDEX_LIMIT; k = k + 1)
        REGISTERS[k] = {`DATA_WIDTH{1'b0}}; // Set all
```

7. If WRITE = 0 and READ = 1

   DATA_R1 = REGISTERS[ADDR_R1]

   DATA_R2 = REGISTERS[ADDR_R2]

   This is the read operation

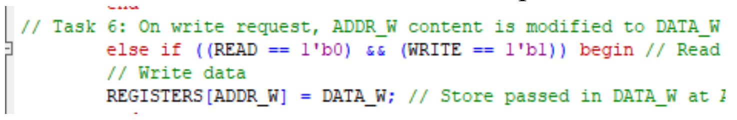


```
if ((READ == 1'b1) && (WRITE == 1'b0)) begin // Read on, Write
// Read data
DATA_R1 = REGISTERS[ADDR_R1]; // Read data at address ADDR_R1
DATA_R2 = REGISTERS[ADDR_R2]; // Read data at address ADDR_R2
```

8. IF WRITE = 1and READ = 0, return data at address ADDR_W

   REGISTERS[ADDR_W] = DATA_W

   This is the write operation



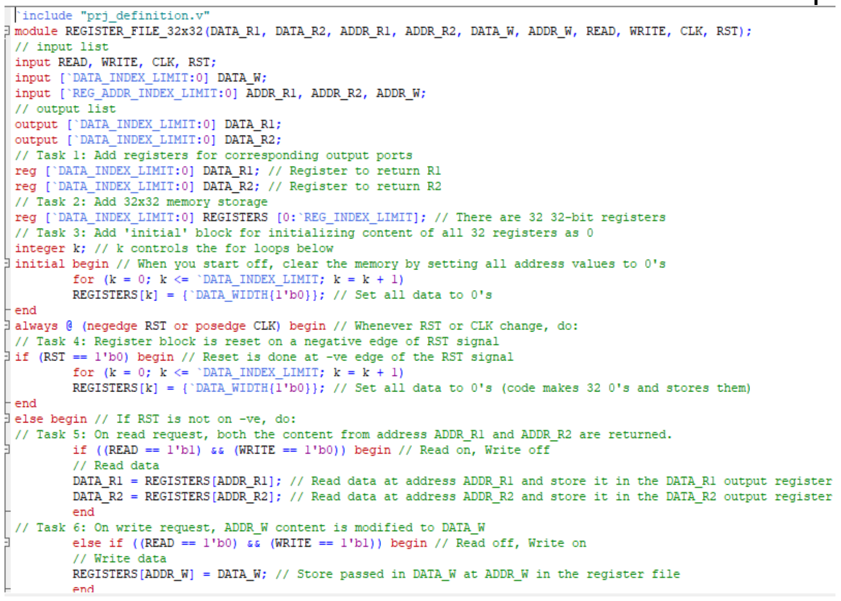
```
// Task 6: On write request, ADDR_W content is modified to DATA_W
        else if ((READ == 1'b0) && (WRITE == 1'b1)) begin // Read
        // Write data
        REGISTERS[ADDR_W] = DATA_W; // Store passed in DATA_W at
```

9. Do not handle Read and Write b

   READ = 1 and WRITE = 1

   READ = 0 and WRITE = 0

10. You should get something like this:

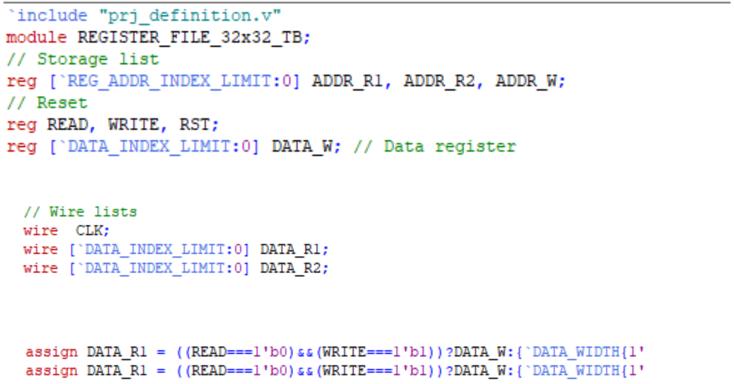


```
`include "prj_definition.v"
module REGISTER_FILE_32x32(DATA_R1, DATA_R2, ADDR_R1, ADDR_R2, DATA_W, ADDR_W, READ, WRITE, CLK, RST);
// input list
input READ, WRITE, CLK, RST;
input [`DATA_INDEX_LIMIT:0] DATA_W;
input [`REG_ADDR_INDEX_LIMIT:0] ADDR_R1, ADDR_R2, ADDR_W;
// output list
output [`DATA_INDEX_LIMIT:0] DATA_R1;
output [`DATA_INDEX_LIMIT:0] DATA_R2;
// Task 1: Add registers for corresponding output ports
reg [`DATA_INDEX_LIMIT:0] DATA_R1; // Register to return R1
reg [`DATA_INDEX_LIMIT:0] DATA_R2; // Register to return R2
// Task 2: Add 32x32 memory storage
reg [`DATA_INDEX_LIMIT:0] REGISTERS [0:`REG_INDEX_LIMIT]; // There are 32 32-bit registers
// Task 3: Add 'initial' block for initializing content of all 32 registers as 0
integer k; // k controls the for loops below
initial begin // When you start off, clear the memory by setting all address values to 0's
        for (k = 0; k <= `DATA_INDEX_LIMIT; k = k + 1)
        REGISTERS[k] = {`DATA_WIDTH{1'b0}}; // Set all data to 0's
end
always @ (negedge RST or posedge CLK) begin // Whenever RST or CLK change, do:
// Task 4: Register block is reset on a negative edge of RST signal
if (RST == 1'b0) begin // Reset is done at -ve edge of the RST signal
        for (k = 0; k <= `DATA_INDEX_LIMIT; k = k + 1)
        REGISTERS[k] = {`DATA_WIDTH{1'b0}}; // Set all data to 0's (code makes 32 0's and stores them)
end
else begin // If RST is not on -ve, do:
// Task 5: On read request, both the content from address ADDR_R1 and ADDR_R2 are returned.
        if ((READ == 1'b1) && (WRITE == 1'b0)) begin // Read on, Write off
        // Read data
        DATA_R1 = REGISTERS[ADDR_R1]; // Read data at address ADDR_R1 and store it in the DATA_R1 output register
        DATA_R2 = REGISTERS[ADDR_R2]; // Read data at address ADDR_R2 and store it in the DATA_R2 output register
        end
// Task 6: On write request, ADDR_W content is modified to DATA_W
        else if ((READ == 1'b0) && (WRITE == 1'b1)) begin // Read off, Write on
        // Write data
        REGISTERS[ADDR_W] = DATA_W; // Store passed in DATA_W at ADDR_W in the register file
        end
```

B. Register File Testing

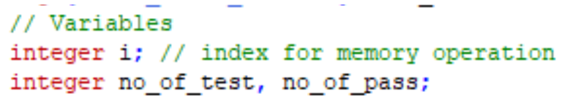
To test the register file, do as such:

1. Create a register file test bench and instantiate a register file
2. Create registers for ADDR_R1, ADDR_R2, ADDR_W, READ, WRITE, RST, and DATA_W. For CLK, DATA_R1, and DATA_R2, create wires. Assign DATA_R1 and DATA_R2



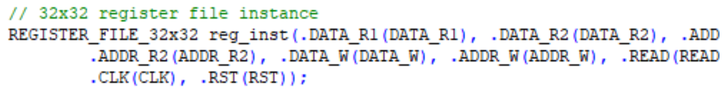
```
`include "prj_definition.v"
module REGISTER_FILE_32x32_TB;
// Storage list
reg [`REG_ADDR_INDEX_LIMIT:0] ADDR_R1, ADDR_R2, ADDR_W;
// Reset
reg READ, WRITE, RST;
reg [`DATA_INDEX_LIMIT:0] DATA_W; // Data register


  // Wire lists
  wire  CLK;
  wire [`DATA_INDEX_LIMIT:0] DATA_R1;
  wire [`DATA_INDEX_LIMIT:0] DATA_R2;


  assign DATA_R1 = ((READ===1'b0)&&(WRITE===1'b1))?DATA_W:{`DATA_WIDTH{1'
  assign DATA_R1 = ((READ===1'b0)&&(WRITE===1'b1))?DATA_W:{`DATA_WIDTH{1'
```

3. Create variables to keep track of test cases and loop counter



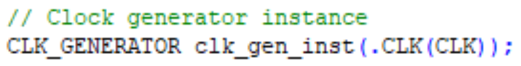
```
// Variables
integer i; // index for memory operation
integer no_of_test, no_of_pass;
```

4. When you instantiate a register file, connect all the registers and wires to the corresponding input and output ports of the register file



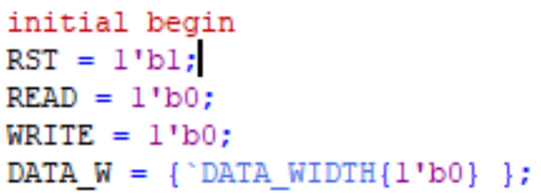
```
// 32x32 register file instance
REGISTER_FILE_32x32 reg_inst(.DATA_R1(DATA_R1), .DATA_R2(DATA_R2), .ADD
        .ADDR_R2(ADDR_R2), .DATA_W(DATA_W), .ADDR_W(ADDR_W), .READ(READ
        .CLK(CLK), .RST(RST));
```

5. Make a clock instance to synchronize your register file



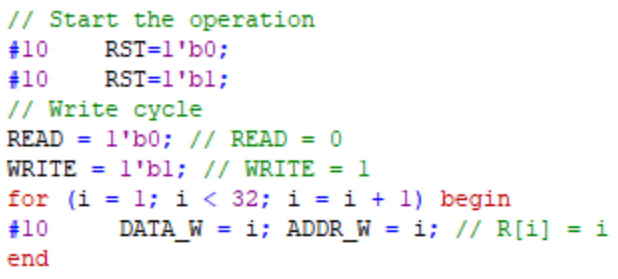
```
// Clock generator instance
CLK_GENERATOR clk_gen_inst(.CLK(CLK));
```

6. Initialize the register file. Flicker RST from 1 to 0 and back to 1 and set READ and WRITE to 0



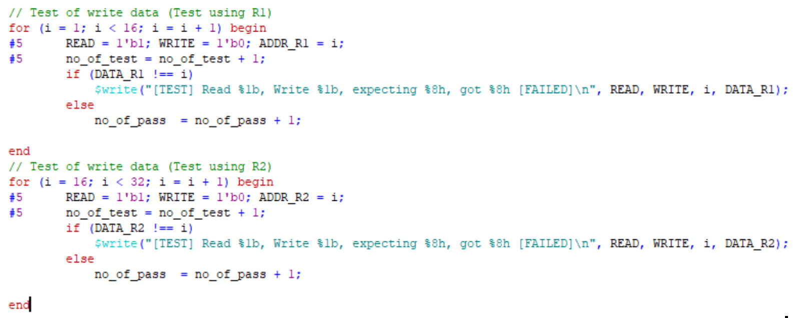
```
initial begin
RST = 1'b1;
READ = 1'b0;
WRITE = 1'b0;
DATA_W = {`DATA_WIDTH{1'b0} };
```

7. Create a FOR loop and write the count variable i to the iith address in the register file



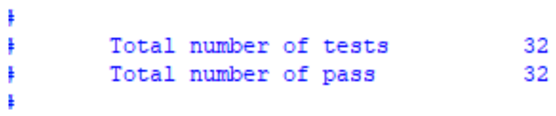
```
// Start the operation
#10     RST=1'b0;
#10     RST=1'b1;
// Write cycle
READ = 1'b0; // READ = 0
WRITE = 1'b1; // WRITE = 1
for (i = 1; i < 32; i = i + 1) begin
#10      DATA_W = i; ADDR_W = i; // R[i] = i
end
```

8. Create another FOR loop to read the data at index i and compare it with the current value of i.



```
// Test of write data (Test using R1)
for (i = 1; i < 16; i = i + 1) begin
#5      READ = 1'b1; WRITE = 1'b0; ADDR_R1 = i;
#5      no_of_test = no_of_test + 1;
        if (DATA_R1 !== i)
            $write("[TEST] Read %1b, Write %1b, expecting %8h, got %8h [FAILED]\n", READ, WRITE, i, DATA_R1);
        else
            no_of_pass  = no_of_pass + 1;

end
// Test of write data (Test using R2)
for (i = 16; i < 32; i = i + 1) begin
#5      READ = 1'b1; WRITE = 1'b0; ADDR_R2 = i;
#5      no_of_test = no_of_test + 1;
        if (DATA_R2 !== i)
            $write("[TEST] Read %1b, Write %1b, expecting %8h, got %8h [FAILED]\n", READ, WRITE, i, DATA_R2);
        else
            no_of_pass  = no_of_pass + 1;

end
```

9. You may print out the number of cases and the number of cases passed

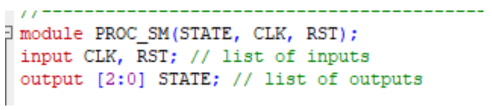


```
#
#       Total number of tests           32
#       Total number of pass            32
#
```

VII. State Machine Design

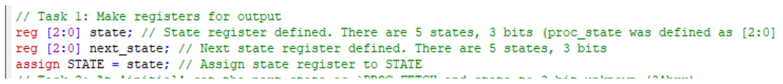
Designing the state machine is fairly easy as it requires only a few steps. The purpose of the state machine is to alternate between the five steps involved in the instruction cycle: instruction fetch, instruction decode/register fetch, execute, memory access, and write back. The purpose of our design should be to cycle through these stages in the order of the list above by determining the next state and transitioning to it. Here is the design:

1. Define input CLK and RST and output STATE



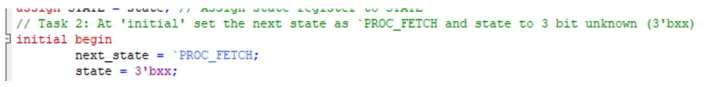
```
//-------------------------------------------
module PROC_SM(STATE, CLK, RST);
input CLK, RST; // list of inputs
output [2:0] STATE; // list of outputs
```

2. Add registers for output and assign state register to output STATE



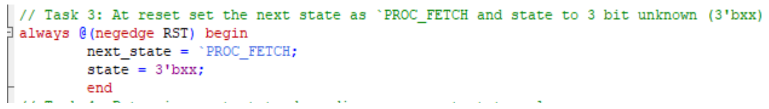
```
// Task 1: Make registers for output
reg [2:0] state; // State register defined. There are 5 states, 3 bits (proc_state was defined as [2:0]
reg [2:0] next_state; // Next state register defined. There are 5 states, 3 bits
assign STATE = state; // Assign state register to STATE
```

3. Initialize the state machine with program fetch, the first stage in the instruction cycle. Reset the state to 3 bit unknown (3’bxx).



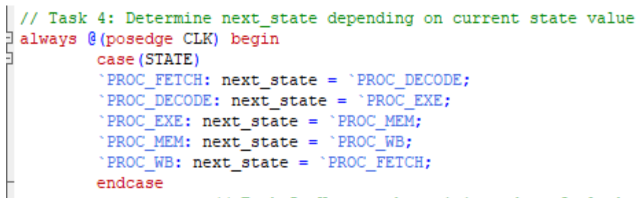
```
// Task 2: At 'initial' set the next state as `PROC_FETCH and state to 3 bit unknown (3'bxx)
initial begin
        next_state = `PROC_FETCH;
        state = 3'bxx;
```

4. At the negative edge of the RST reset signal, set the next state as program fetch and reset the state to 3 bit unknown (3’bxx)



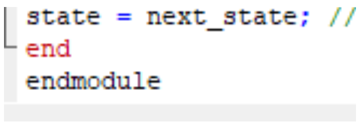
```
// Task 3: At reset set the next state as `PROC_FETCH and state to 3 bit unknown (3'bxx)
always @(negedge RST) begin
        next_state = `PROC_FETCH;
        state = 3'bxx;
        end
```

5. On the positive edge of the clock, determine the next state depending on the current state



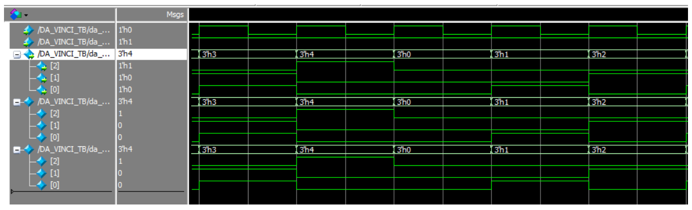
```
// Task 4: Determine next_state depending on current state value
always @(posedge CLK) begin
        case(STATE)
        `PROC_FETCH: next_state = `PROC_DECODE;
        `PROC_DECODE: next_state = `PROC_EXE;
        `PROC_EXE: next_state = `PROC_MEM;
        `PROC_MEM: next_state = `PROC_WB;
        `PROC_WB: next_state = `PROC_FETCH;
        endcase
```

6. Upon determining the next state, transition state to its next state value



```
state = next_state; //
end
endmodule
```

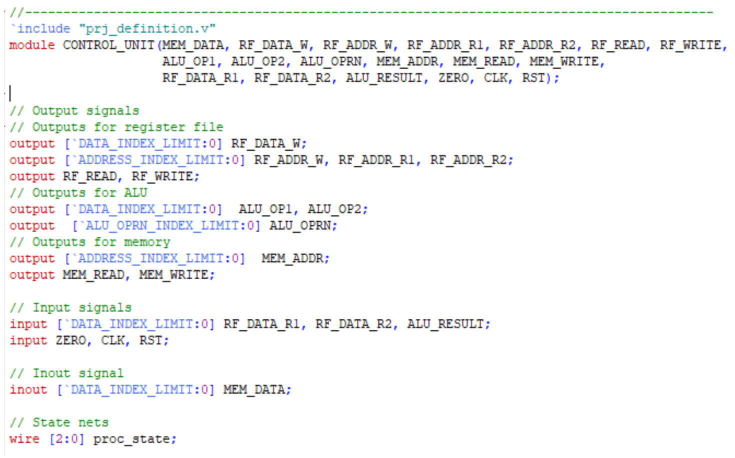
7. Upon running the state machine by simulating control unit, you should get the cropped waveform output as such:



VIII. Control Unit Design

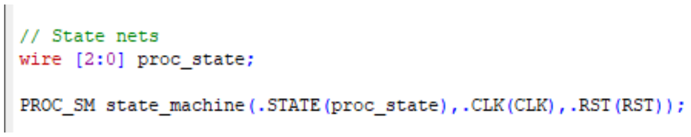
The purpose of the control unit is to generate the control signals which are responsible for controlling our ALU, memory, and register file. The control unit unifies the use of these three components and exchanges data between them. The control unit read the instructions in memory and performs the operations involved in the five states of the instruction cycle. The design is for the control unit (CU) is as such:

1. Define input and output as such:



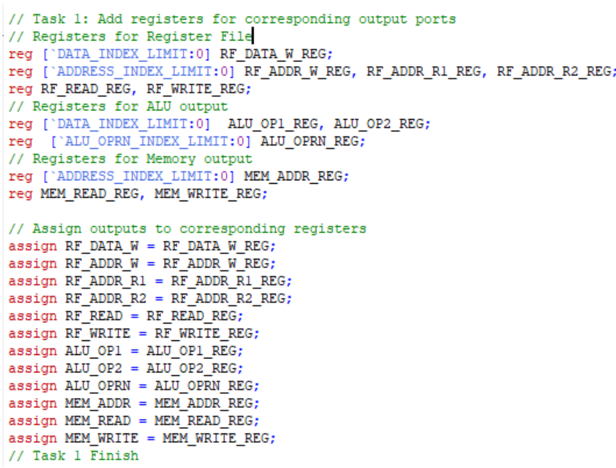
```
//------------------------------------------------------------------------------------------
`include "prj_definition.v"
module CONTROL_UNIT(MEM_DATA, RF_DATA_W, RF_ADDR_W, RF_ADDR_R1, RF_ADDR_R2, RF_READ, RF_WRITE,
                    ALU_OP1, ALU_OP2, ALU_OPRN, MEM_ADDR, MEM_READ, MEM_WRITE,
                    RF_DATA_R1, RF_DATA_R2, ALU_RESULT, ZERO, CLK, RST);

// Output signals
// Outputs for register file
output [`DATA_INDEX_LIMIT:0] RF_DATA_W;
output [`ADDRESS_INDEX_LIMIT:0] RF_ADDR_W, RF_ADDR_R1, RF_ADDR_R2;
output RF_READ, RF_WRITE;
// Outputs for ALU
output [`DATA_INDEX_LIMIT:0]  ALU_OP1, ALU_OP2;
output  [`ALU_OPRN_INDEX_LIMIT:0] ALU_OPRN;
// Outputs for memory
output [`ADDRESS_INDEX_LIMIT:0]  MEM_ADDR;
output MEM_READ, MEM_WRITE;

// Input signals
input [`DATA_INDEX_LIMIT:0] RF_DATA_R1, RF_DATA_R2, ALU_RESULT;
input ZERO, CLK, RST;

// Inout signal
inout [`DATA_INDEX_LIMIT:0] MEM_DATA;

// State nets
wire [2:0] proc_state;
```

2. Instantiate the state machine and add wire to connect program state



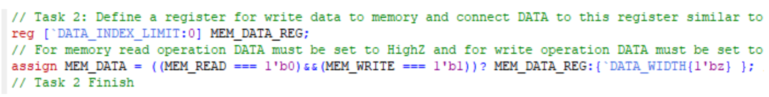
```
// State nets
wire [2:0] proc_state;

PROC_SM state_machine(.STATE(proc_state),.CLK(CLK),.RST(RST));
```

3. Add registers and wire for the corresponding output ports and assign them:



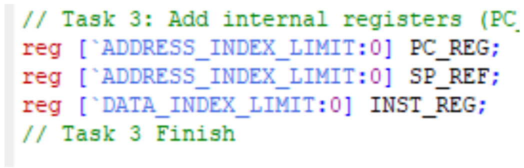
```
// Task 1: Add registers for corresponding output ports
// Registers for Register File
reg [`DATA_INDEX_LIMIT:0] RF_DATA_W_REG;
reg [`ADDRESS_INDEX_LIMIT:0] RF_ADDR_W_REG, RF_ADDR_R1_REG, RF_ADDR_R2_REG;
reg RF_READ_REG, RF_WRITE_REG;
// Registers for ALU output
reg [`DATA_INDEX_LIMIT:0]  ALU_OP1_REG, ALU_OP2_REG;
reg  [`ALU_OPRN_INDEX_LIMIT:0] ALU_OPRN_REG;
// Registers for Memory output
reg [`ADDRESS_INDEX_LIMIT:0] MEM_ADDR_REG;
reg MEM_READ_REG, MEM_WRITE_REG;

// Assign outputs to corresponding registers
assign RF_DATA_W = RF_DATA_W_REG;
assign RF_ADDR_W = RF_ADDR_W_REG;
assign RF_ADDR_R1 = RF_ADDR_R1_REG;
assign RF_ADDR_R2 = RF_ADDR_R2_REG;
assign RF_READ = RF_READ_REG;
assign RF_WRITE = RF_WRITE_REG;
assign ALU_OP1 = ALU_OP1_REG;
assign ALU_OP2 = ALU_OP2_REG;
assign ALU_OPRN = ALU_OPRN_REG;
assign MEM_ADDR = MEM_ADDR_REG;
assign MEM_READ = MEM_READ_REG;
assign MEM_WRITE = MEM_WRITE_REG;
// Task 1 Finish
```

4. Define a register for writing data to memory and connect MEM_DATA to this register



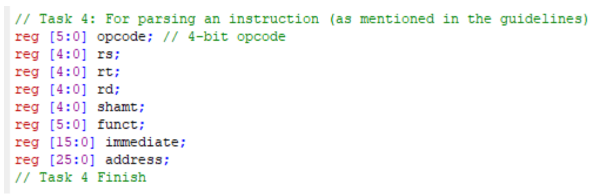
```
// Task 2: Define a register for write data to memory and connect DATA to this register similar to
reg [`DATA_INDEX_LIMIT:0] MEM_DATA_REG;
// For memory read operation DATA must be set to HighZ and for write operation DATA must be set to
assign MEM_DATA = ((MEM_READ === 1'b0)&&(MEM_WRITE === 1'b1))? MEM_DATA_REG:{`DATA_WIDTH{1'bz} };
// Task 2 Finish
```

5. Add internal registers for program counter (PC), stack pointer (SP), and instruction register (INST)



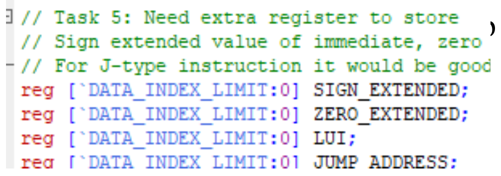
```
// Task 3: Add internal registers (PC
reg [`ADDRESS_INDEX_LIMIT:0] PC_REG;
reg [`ADDRESS_INDEX_LIMIT:0] SP_REF;
reg [`DATA_INDEX_LIMIT:0] INST_REG;
// Task 3 Finish
```

6. Create registers to help parse each instruction



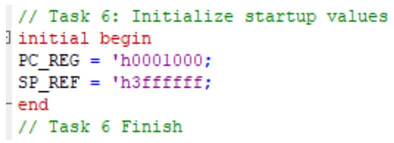
```
// Task 4: For parsing an instruction (as mentioned in the guidelines)
reg [5:0] opcode; // 4-bit opcode
reg [4:0] rs;
reg [4:0] rt;
reg [4:0] rd;
reg [4:0] shamt;
reg [5:0] funct;
reg [15:0] immediate;
reg [25:0] address;
// Task 4 Finish
```

7. Calculate sign extended value of immediate, zero extended value of



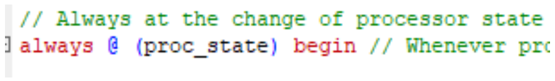
```
// Task 5: Need extra register to store
// Sign extended value of immediate, zero
// For J-type instruction it would be good
reg [`DATA_INDEX_LIMIT:0] SIGN_EXTENDED;
reg [`DATA_INDEX_LIMIT:0] ZERO_EXTENDED;
reg [`DATA_INDEX_LIMIT:0] LUI;
reg [`DATA_INDEX_LIMIT:0] JUMP_ADDRESS;
```

8. Initialize the startup values of the internal registers PC and SP



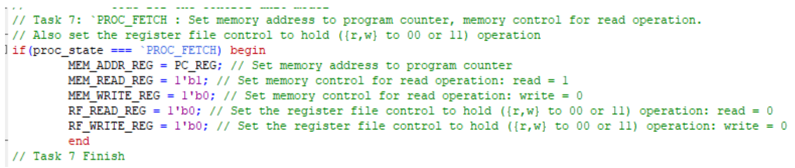
```
// Task 6: Initialize startup values
initial begin
PC_REG = 'h0001000;
SP_REF = 'h3ffffff;
end
// Task 6 Finish
```

9. Whenever the state changes, transition to the next cycle and perform the operations responsible for that cycle.



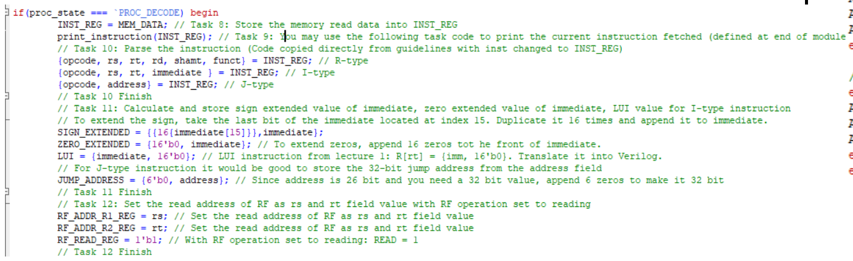
```
// Always at the change of processor state
always @ (proc_state) begin // Whenever pro
```

10. **Instruction fetch**: set the memory address to program counter. Set memory control for read operation



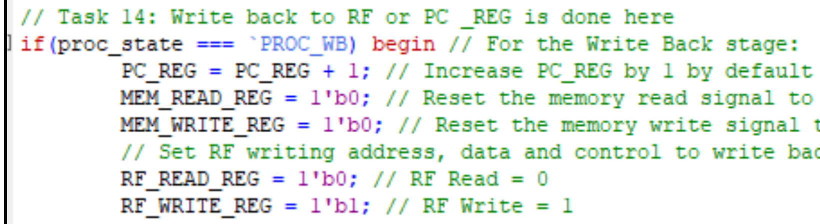
```
// Task 7: `PROC_FETCH : Set memory address to program counter, memory control for read operation.
// Also set the register file control to hold ({r,w} to 00 or 11) operation
if(proc_state === `PROC_FETCH) begin
        MEM_ADDR_REG = PC_REG; // Set memory address to program counter
        MEM_READ_REG = 1'b1; // Set memory control for read operation: read = 1
        MEM_WRITE_REG = 1'b0; // Set memory control for read operation: write = 0
        RF_READ_REG = 1'b0; // Set the register file control to hold ({r,w} to 00 or 11) operation: read = 0
        RF_WRITE_REG = 1'b0; // Set the register file control to hold ({r,w} to 00 or 11) operation: write = 0
        end
// Task 7 Finish
```

11. **Instruction decode/Register Fetch**: Store the memory read data into INST_REG. You may print the instructions. Parse the instructions. Calculate the extra values for sign and zero extended. Set the read address of the register file (addresses of R1 and R2) as rs and rt field. Set register file read.



```
if(proc_state === `PROC_DECODE) begin
        INST_REG = MEM_DATA; // Task 8: Store the memory read data into INST_REG
        print_instruction(INST_REG); // Task 9: You may use the following task code to print the current instruction fetched (defined at end of module
        // Task 10: Parse the instruction (Code copied directly from guidelines with inst changed to INST_REG)
        {opcode, rs, rt, rd, shamt, funct} = INST_REG; // R-type
        {opcode, rs, rt, immediate } = INST_REG; // I-type
        {opcode, address} = INST_REG; // J-type
        // Task 10 Finish
        // Task 11: Calculate and store sign extended value of immediate, zero extended value of immediate, LUI value for I-type instruction
        // To extend the sign, take the last bit of the immediate located at index 15. Duplicate it 16 times and append it to immediate.
        SIGN_EXTENDED = {{16{immediate[15]}},immediate};
        ZERO_EXTENDED = {16'b0, immediate}; // To extend zeros, append 16 zeros tot he front of immediate.
        LUI = {immediate, 16'b0}; // LUI instruction from lecture 1: R[rt] = {imm, 16'b0}. Translate it into Verilog.
        // For J-type instruction it would be good to store the 32-bit jump address from the address field
        JUMP_ADDRESS = {6'b0, address}; // Since address is 26 bit and you need a 32 bit value, append 6 zeros to make it 32 bit
        // Task 11 Finish
        // Task 12: Set the read address of RF as rs and rt field value with RF operation set to reading
        RF_ADDR_R1_REG = rs; // Set the read address of RF as rs and rt field value
        RF_ADDR_R2_REG = rt; // Set the read address of RF as rs and rt field value
        RF_READ_REG = 1'b1; // With RF operation set to reading: READ = 1
        // Task 12 Finish
```

13. **Write Back**: Before the three types are defined in WB, PC must be incremented, memory read and write should be 0, and register file read = 0 and write = 1.



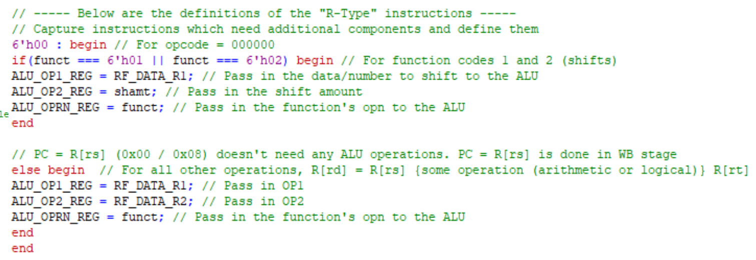
```
// Task 14: Write back to RF or PC _REG is done here
if(proc_state === `PROC_WB) begin // For the Write Back stage:
        PC_REG = PC_REG + 1; // Increase PC_REG by 1 by default
        MEM_READ_REG = 1'b0; // Reset the memory read signal to
        MEM_WRITE_REG = 1'b0; // Reset the memory write signal t
        // Set RF writing address, data and control to write bac
        RF_READ_REG = 1'b0; // RF Read = 0
        RF_WRITE_REG = 1'b1; // RF Write = 1
```

14. EXE, MEM, and WB states will be further elaborated in the upcoming topics.

IX. R-Type Instruction Implementation

The R-Type instructions are primarily arithmetic and logical (except jr) and involve the registers and ALU. The shift instructions sll and srl involve a shift amount and the jr instruction involves the program counter PC.

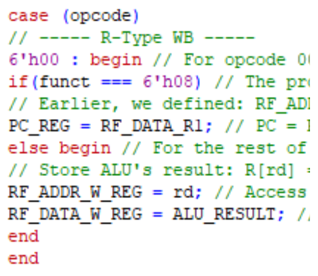
EXE: Capture the instructions that require additional components (shifts). Define them separate from the other instructions. All the instructions involve the ALU so pass in the two operand and their ALU function code (R[rs and R[rt] for all except shift and PC). For shift, OP1 = R[rs] and OP2 = shift amount. Instruction jr is not involved.



```
// ----- Below are the definitions of the "R-Type" instructions -----
// Capture instructions which need additional components and define them
6'h00 : begin // For opcode = 000000
if(funct === 6'h01 || funct === 6'h02) begin // For function codes 1 and 2 (shifts)
ALU_OP1_REG = RF_DATA_R1; // Pass in the data/number to shift to the ALU
ALU_OP2_REG = shamt; // Pass in the shift amount
ALU_OPRN_REG = funct; // Pass in the function's opn to the ALU
end

// PC = R[rs] (0x00 / 0x08) doesn't need any ALU operations. PC = R[rs] is done in WB stage
else begin  // For all other operations, R[rd] = R[rs] {some operation (arithmetic or logical)} R[rt]
ALU_OP1_REG = RF_DATA_R1; // Pass in OP1
ALU_OP2_REG = RF_DATA_R2; // Pass in OP2
ALU_OPRN_REG = funct; // Pass in the function's opn to the ALU
end
end
```

MEM: Not involved

WB: Retrieve the ALU result and assign it to R[rs]. For jr, let PC = R[rs]

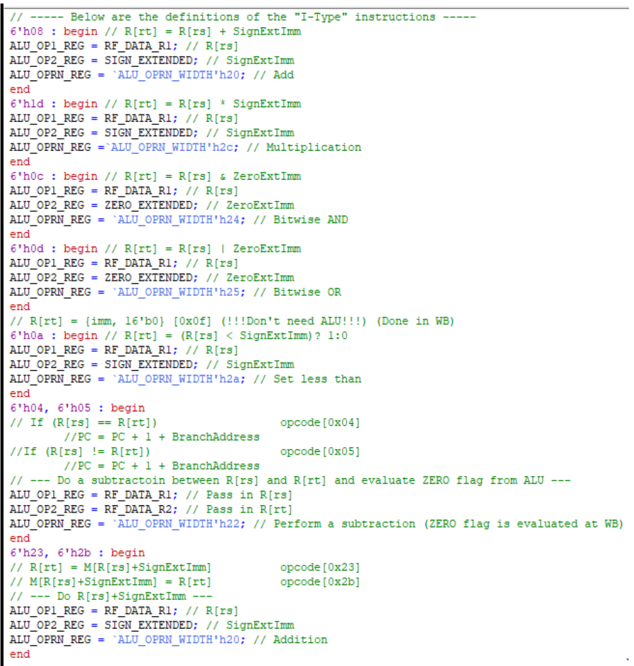


```
case (opcode)
// ----- R-Type WB -----
6'h00 : begin // For opcode 0
if(funct === 6'h08) // The pr
// Earlier, we defined: RF_ADI
PC_REG = RF_DATA_R1; // PC = I
else begin // For the rest of
// Store ALU's result: R[rd] :
RF_ADDR_W_REG = rd; // Access
RF_DATA_W_REG = ALU_RESULT; /.
end
end
```

X. I-Type Instruction Implementation

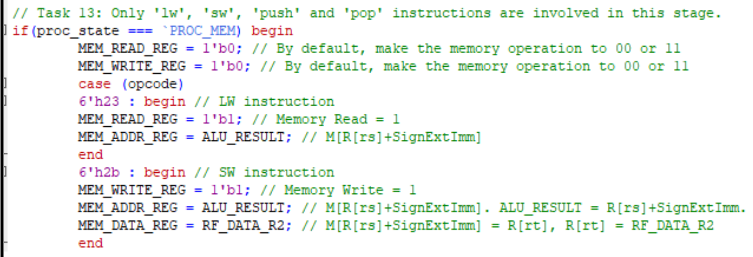
The R-Type instructions are a mix of arithmetic and logical instructions involving a register and an immediate value. Many instructions require the sign-extended, zero-extended, jump-address, and lui which we have precalculated. Store word and load word are involved in the memory stage unlike the rest of the instructions.

EXE: For addi, muli, andi, ori, and slti, pass in the two operands R[rs] and/or sign-extended/zero-extended (depending on the instruction) along with the corresponding ALU function code into the ALU. For beq and bne, OP1 = R[rs] and OP2 = R[rt] and do a subtraction in the ALU. For lw and sw, add R[rs] with sign-extended in ALU



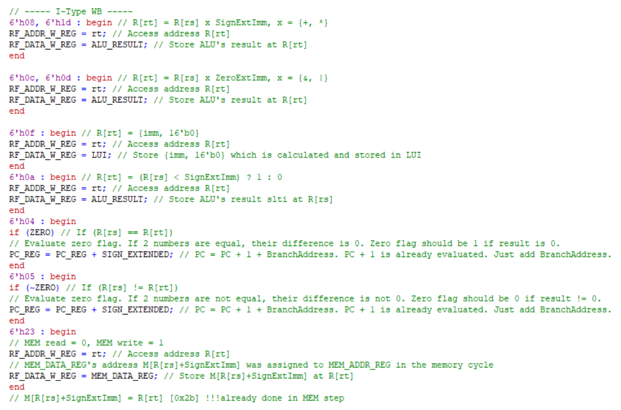
```
// ----- Below are the definitions of the "I-Type" instructions -----
6'h08 : begin // R[rt] = R[rs] + SignExtImm
ALU_OP1_REG = RF_DATA_R1; // R[rs]
ALU_OP2_REG = SIGN_EXTENDED; // SignExtImm
ALU_OPRN_REG = `ALU_OPRN_WIDTH'h20; // Add
end
6'h1d : begin // R[rt] = R[rs] * SignExtImm
ALU_OP1_REG = RF_DATA_R1; // R[rs]
ALU_OP2_REG = SIGN_EXTENDED; // SignExtImm
ALU_OPRN_REG =`ALU_OPRN_WIDTH'h2c; // Multiplication
end
6'h0c : begin // R[rt] = R[rs] & ZeroExtImm
ALU_OP1_REG = RF_DATA_R1; // R[rs]
ALU_OP2_REG = ZERO_EXTENDED; // ZeroExtImm
ALU_OPRN_REG = `ALU_OPRN_WIDTH'h24; // Bitwise AND
end
6'h0d : begin // R[rt] = R[rs] | ZeroExtImm
ALU_OP1_REG = RF_DATA_R1; // R[rs]
ALU_OP2_REG = ZERO_EXTENDED; // ZeroExtImm
ALU_OPRN_REG = `ALU_OPRN_WIDTH'h25; // Bitwise OR
end
// R[rt] = {imm, 16'b0} [0x0f] (!!!Don't need ALU!!!) (Done in WB)
6'h0a : begin // R[rt] = (R[rs] < SignExtImm)? 1:0
ALU_OP1_REG = RF_DATA_R1; // R[rs]
ALU_OP2_REG = SIGN_EXTENDED; // SignExtImm
ALU_OPRN_REG = `ALU_OPRN_WIDTH'h2a; // Set less than
end
6'h04, 6'h05 : begin
// If (R[rs] == R[rt])                    opcode[0x04]
        //PC = PC + 1 + BranchAddress
//If (R[rs] != R[rt])                     opcode[0x05]
        //PC = PC + 1 + BranchAddress
// --- Do a subtractoin between R[rs] and R[rt] and evaluate ZERO flag from ALU ---
ALU_OP1_REG = RF_DATA_R1; // Pass in R[rs]
ALU_OP2_REG = RF_DATA_R2; // Pass in R[rt]
ALU_OPRN_REG = `ALU_OPRN_WIDTH'h22; // Perform a subtraction (ZERO flag is evaluated at WB)
end
6'h23, 6'h2b : begin
// R[rt] = M[R[rs]+SignExtImm]            opcode[0x23]
// M[R[rs]+SignExtImm] = R[rt]            opcode[0x2b]
// --- Do R[rs]+SignExtImm ---
ALU_OP1_REG = RF_DATA_R1; // R[rs]
ALU_OP2_REG = SIGN_EXTENDED; // SignExtImm
ALU_OPRN_REG = `ALU_OPRN_WIDTH'h20; // Addition
end
```

MEM: For lw, read the data at address R[rs]+sign-extended calculated by ALU and store the value in the temporary memory register. For sw, Let the data in the memory index calculated by the ALU equal R[rt].



```
// Task 13: Only 'lw', 'sw', 'push' and 'pop' instructions are involved in this stage.
if(proc_state === `PROC_MEM) begin
        MEM_READ_REG = 1'b0; // By default, make the memory operation to 00 or 11
        MEM_WRITE_REG = 1'b0; // By default, make the memory operation to 00 or 11
        case (opcode)
        6'h23 : begin // LW instruction
        MEM_READ_REG = 1'b1; // Memory Read = 1
        MEM_ADDR_REG = ALU_RESULT; // M[R[rs]+SignExtImm]
        end
        6'h2b : begin // SW instruction
        MEM_WRITE_REG = 1'b1; // Memory Write = 1
        MEM_ADDR_REG = ALU_RESULT; // M[R[rs]+SignExtImm]. ALU_RESULT = R[rs]+SignExtImm.
        MEM_DATA_REG = RF_DATA_R2; // M[R[rs]+SignExtImm] = R[rt], R[rt] = RF_DATA_R2
        end
```

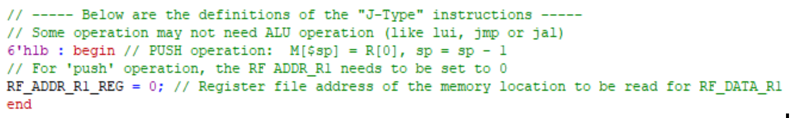
WB: Let R[rt] equal the ALU result for For addi, muli, andi, and ori. Let R[rt] = LUI (precalculated) value for lui instruction. For beq and bne, evaluate the zero flag. If zero flag is set for beq, add the jump-address to PC. If the zero flag in not set in bne, add jump-address to PC. For sw, transfer the data in the memory data register into R[rt].



```
// ----- I-Type WB -----
6'h08, 6'h1d : begin // R[rt] = R[rs] x SignExtImm, x = {+, *}
RF_ADDR_W_REG = rt; // Access address R[rt]
RF_DATA_W_REG = ALU_RESULT; // Store ALU's result at R[rt]
end

6'h0c, 6'h0d : begin // R[rt] = R[rs] x ZeroExtImm, x = {&, |}
RF_ADDR_W_REG = rt; // Access address R[rt]
RF_DATA_W_REG = ALU_RESULT; // Store ALU's result at R[rt]
end

6'h0f : begin // R[rt] = {imm, 16'b0}
RF_ADDR_W_REG = rt; // Access address R[rt]
RF_DATA_W_REG = LUI; // Store {imm, 16'b0} which is calculated and stored in LUI
end
6'h0a : begin // R[rt] = (R[rs] < SignExtImm) ? 1 : 0
RF_ADDR_W_REG = rt; // Access address R[rt]
RF_DATA_W_REG = ALU_RESULT; // Store ALU's result slti at R[rs]
end
6'h04 : begin
if (ZERO) // If (R[rs] == R[rt])
// Evaluate zero flag. If 2 numbers are equal, their difference is 0. Zero flag should be 1 if result is 0.
PC_REG = PC_REG + SIGN_EXTENDED; // PC = PC + 1 + BranchAddress. PC + 1 is already evaluated. Just add BranchAddress.
end
6'h05 : begin
if (~ZERO) // If (R[rs] != R[rt])
// Evaluate zero flag. If 2 numbers are not equal, their difference is not 0. Zero flag should be 0 if result != 0.
PC_REG = PC_REG + SIGN_EXTENDED; // PC = PC + 1 + BranchAddress. PC + 1 is already evaluated. Just add BranchAddress.
end
6'h23 : begin
// MEM read = 0, MEM write = 1
RF_ADDR_W_REG = rt; // Access address R[rt]
// MEM_DATA_REG's address M[R[rs]+SignExtImm] was assigned to MEM_ADDR_REG in the memory cycle
RF_DATA_W_REG = MEM_DATA_REG; // Store M[R[rs]+SignExtImm] at R[rt]
end
// M[R[rs]+SignExtImm] = R[rt] [0x2b] !!!already done in MEM step
```

XI. J-Type Instruction Implementation

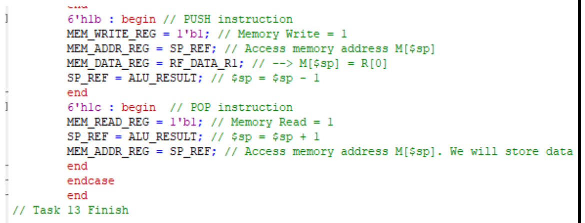
The J-Type instruction are 4: jmp, jal, push, and pop. The first two involve PC and the last two involve the stack and SP.



```
// ----- Below are the definitions of the "J-Type" instructions -----
// Some operation may not need ALU operation (like lui, jmp or jal)
6'h1b : begin // PUSH operation:  M[$sp] = R[0], sp = sp - 1
// For 'push' operation, the RF ADDR_R1 needs to be set to 0
RF_ADDR_R1_REG = 0; // Register file address of the memory location to be read for RF_DATA_R1
end
```

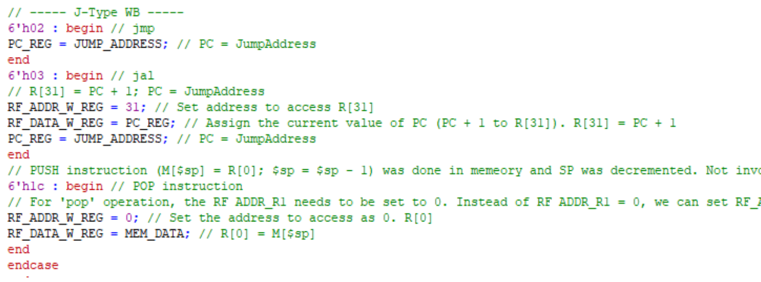
EXE: For push and pop, pass in the value of SP and 1. Subtract for push, add for pop.

MEM: For push, write to memory at index SP (M[SP]) the data at R[0]. Let SP equal the ALU result (SP = SP – 1). For pop, let SP equal ALU result (SP = SP + 1). Set memory access address to the new SP.



```
        6'h1b : begin // PUSH instruction
        MEM_WRITE_REG = 1'b1; // Memory Write = 1
        MEM_ADDR_REG = SP_REF; // Access memory address M[$sp]
        MEM_DATA_REG = RF_DATA_R1; // --> M[$sp] = R[0]
        SP_REF = ALU_RESULT; // $sp = $sp - 1
        end
        6'h1c : begin  // POP instruction
        MEM_READ_REG = 1'b1; // Memory Read = 1
        SP_REF = ALU_RESULT; // $sp = $sp + 1
        MEM_ADDR_REG = SP_REF; // Access memory address M[$sp]. We will store data
        end
        endcase
        end
// Task 13 Finish
```

WB: For jmp, PC = jump-address. For jal, R[31] is the incremented PC and PC is then assigned the jump-address. For pop, write at register file R[0] the data stored at the address pointed by the incremented SP at MEM state.



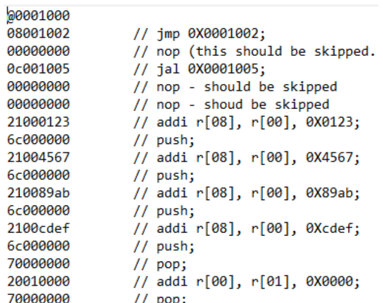
```
// ----- J-Type WB -----
6'h02 : begin // jmp
PC_REG = JUMP_ADDRESS; // PC = JumpAddress
end
6'h03 : begin // jal
// R[31] = PC + 1; PC = JumpAddress
RF_ADDR_W_REG = 31; // Set address to access R[31]
RF_DATA_W_REG = PC_REG; // Assign the current value of PC (PC + 1 to R[31]). R[31] = PC + 1
PC_REG = JUMP_ADDRESS; // PC = JumpAddress
end
// PUSH instruction (M[$sp] = R[0]; $sp = $sp - 1) was done in memeory and SP was decremented. Not invo
6'h1c : begin // POP instruction
// For 'pop' operation, the RF ADDR_R1 needs to be set to 0. Instead of RF ADDR_R1 = 0, we can set RF_A
RF_ADDR_W_REG = 0; // Set the address to access as 0. R[0]
RF_DATA_W_REG = MEM_DATA; // R[0] = M[$sp]
end
endcase
```

XII. System Testing

To test if your design works, handwrite design a short assembly program and convert it into hex following the format of CS 147DV. Store your program in a .dat file and load it in "da_vinci_tb" by changing the filename to your test file's filename in line 38. Also change the filename in line 51 to match the memory dump file to its load file's name and modify the address range to dump. Compare the dump file with the expected output of the designed program (the designer should know what the program does and which memory and register addresses are modified). If the output does not match, doublecheck your assembly program. Afterward, trace your EXE, MEM, or WB for that instruction and try to find and fix any errors.

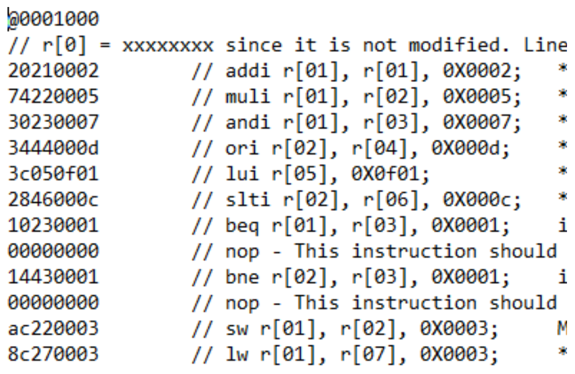
Creating Test Programs

1. R-Type testing: Create a program testing all the R-Type instructions.



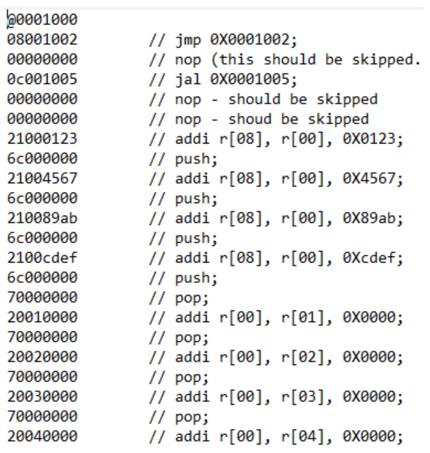
```
@0001000
08001002        // jmp 0X0001002;
00000000        // nop (this should be skipped.
0c001005        // jal 0X0001005;
00000000        // nop - should be skipped
00000000        // nop - shoud be skipped
21000123        // addi r[08], r[00], 0X0123;
6c000000        // push;
21004567        // addi r[08], r[00], 0X4567;
6c000000        // push;
210089ab        // addi r[08], r[00], 0X89ab;
6c000000        // push;
2100cdef        // addi r[08], r[00], 0Xcdef;
6c000000        // push;
70000000        // pop;
20010000        // addi r[00], r[01], 0X0000;
70000000        // pop;
```

2. I-Type testing: Create a program testing all the I-Type instructions.



```
@0001000
// r[0] = xxxxxxxx since it is not modified. Line
20210002        // addi r[01], r[01], 0X0002;    *
74220005        // muli r[01], r[02], 0X0005;    *
30230007        // andi r[01], r[03], 0X0007;    *
3444000d        // ori r[02], r[04], 0X000d;     *
3c050f01        // lui r[05], 0X0f01;            *
2846000c        // slti r[02], r[06], 0X000c;    *
10230001        // beq r[01], r[03], 0X0001;     i
00000000        // nop - This instruction should
14430001        // bne r[02], r[03], 0X0001;     i
00000000        // nop - This instruction should
ac220003        // sw r[01], r[02], 0X0003;      M
8c270003        // lw r[01], r[07], 0X0003;      *
```

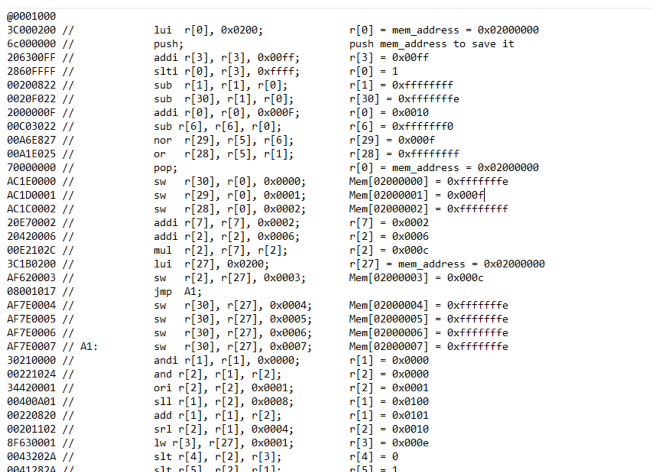
3. J-Type testing: Create a program testing all the J-Type instructions.



```
@0001000
08001002        // jmp 0X0001002;
00000000        // nop (this should be skipped.
0c001005        // jal 0X0001005;
00000000        // nop - should be skipped
00000000        // nop - shoud be skipped
21000123        // addi r[08], r[00], 0X0123;
6c000000        // push;
21004567        // addi r[08], r[00], 0X4567;
6c000000        // push;
210089ab        // addi r[08], r[00], 0X89ab;
6c000000        // push;
2100cdef        // addi r[08], r[00], 0Xcdef;
6c000000        // push;
70000000        // pop;
20010000        // addi r[00], r[01], 0X0000;
70000000        // pop;
20020000        // addi r[00], r[02], 0X0000;
70000000        // pop;
20030000        // addi r[00], r[03], 0X0000;
70000000        // pop;
20040000        // addi r[00], r[04], 0X0000;
```

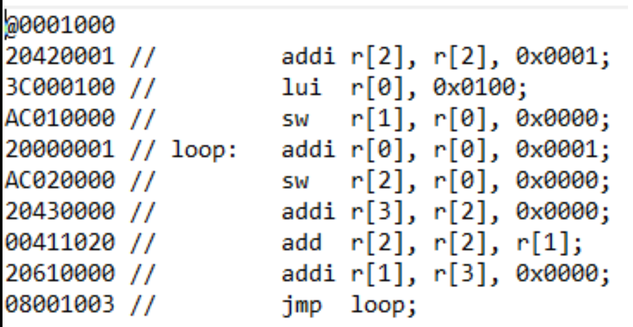
4. Systemwide testing: Create a program with a mix of all three instruction types. You may also use fibbonacci and RevFib by Professor Kaushik Patra to test.

A test of the whole system:



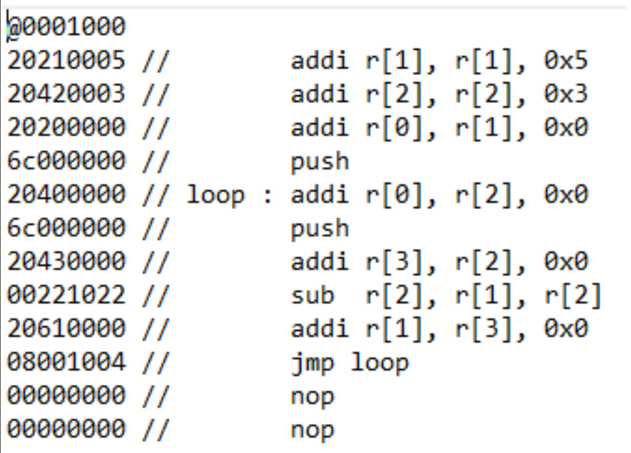
```
@0001000
3C000200 //          lui  r[0], 0x0200;             r[0] = mem_address = 0x02000000
6c000000 //          push;                          push mem_address to save it
206300FF //          addi r[3], r[3], 0x00ff;       r[3] = 0x00ff
2860FFFF //          slti r[0], r[3], 0xffff;       r[0] = 1
00200822 //          sub  r[1], r[1], r[0];         r[1] = 0xffffffff
0020F022 //          sub  r[30], r[1], r[0];        r[30] = 0xfffffffe
2000000F //          addi r[0], r[0], 0x000F;       r[0] = 0x0010
00C03022 //          sub r[6], r[6], r[0];          r[6] = 0xfffffff0
00A6E827 //          nor  r[29], r[5], r[6];        r[29] = 0x000f
00A1E025 //          or   r[28], r[5], r[1];        r[28] = 0xffffffff
70000000 //          pop;                           r[0] = mem_address = 0x02000000
AC1E0000 //          sw   r[30], r[0], 0x0000;      Mem[02000000] = 0xfffffffe
AC1D0001 //          sw   r[29], r[0], 0x0001;      Mem[02000001] = 0x000f
AC1C0002 //          sw   r[28], r[0], 0x0002;      Mem[02000002] = 0xffffffff
20E70002 //          addi r[7], r[7], 0x0002;       r[7] = 0x0002
20420006 //          addi r[2], r[2], 0x0006;       r[2] = 0x0006
00E2102C //          mul  r[2], r[7], r[2];         r[2] = 0x000c
3C1B0200 //          lui  r[27], 0x0200;            r[27] = mem_address = 0x02000000
AF620003 //          sw   r[2], r[27], 0x0003;      Mem[02000003] = 0x000c
08001017 //          jmp  A1;
AF7E0004 //          sw   r[30], r[27], 0x0004;     Mem[02000004] = 0xfffffffe
AF7E0005 //          sw   r[30], r[27], 0x0005;     Mem[02000005] = 0xfffffffe
AF7E0006 //          sw   r[30], r[27], 0x0006;     Mem[02000006] = 0xfffffffe
AF7E0007 // A1:      sw   r[30], r[27], 0x0007;     Mem[02000007] = 0xfffffffe
30210000 //          andi r[1], r[1], 0x0000;       r[1] = 0x0000
00221024 //          and r[2], r[1], r[2];          r[2] = 0x0000
34420001 //          ori r[2], r[2], 0x0001;        r[2] = 0x0001
00400A01 //          sll r[1], r[2], 0x0008;        r[1] = 0x0100
00220820 //          add r[1], r[1], r[2];          r[1] = 0x0101
00201102 //          srl r[2], r[1], 0x0004;        r[2] = 0x0010
8F630001 //          lw r[3], r[27], 0x0001;        r[3] = 0x000e
0043202A //          slt r[4], r[2], r[3];          r[4] = 0
0041282A //          slt r[5], r[2], r[1];          r[5] = 1
```

"Fibonacci" system test:



```
@0001000
20420001 //          addi r[2], r[2], 0x0001;
3C000100 //          lui  r[0], 0x0100;
AC010000 //          sw   r[1], r[0], 0x0000;
20000001 // loop:    addi r[0], r[0], 0x0001;
AC020000 //          sw   r[2], r[0], 0x0000;
20430000 //          addi r[3], r[2], 0x0000;
00411020 //          add  r[2], r[2], r[1];
20610000 //          addi r[1], r[3], 0x0000;
08001003 //          jmp  loop;
```

"RevFib" system test:

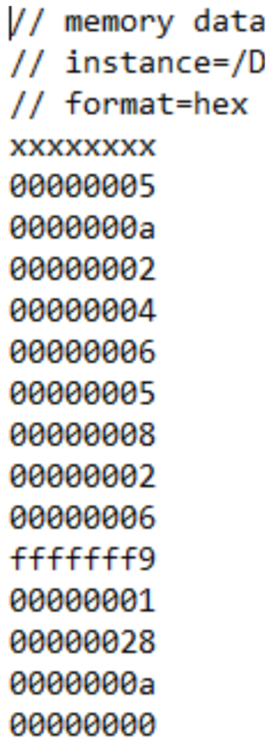


```
@0001000
20210005 //          addi r[1], r[1], 0x5
20420003 //          addi r[2], r[2], 0x3
20200000 //          addi r[0], r[1], 0x0
6c000000 //          push
20400000 // loop :   addi r[0], r[2], 0x0
6c000000 //          push
20430000 //          addi r[3], r[2], 0x0
00221022 //          sub  r[2], r[1], r[2]
20610000 //          addi r[1], r[3], 0x0
08001004 //          jmp loop
00000000 //          nop
00000000 //          nop
```

Evaluating Output

1. Compare your R-Type output in the memory dump file with your program's expected values.

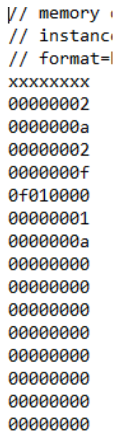
Solution to my R-Type instructions test:



```
// memory data
// instance=/D
// format=hex
xxxxxxxx
00000005
0000000a
00000002
00000004
00000006
00000005
00000008
00000002
00000006
fffffff9
00000001
00000028
0000000a
00000000
```

2. Compare your I-Type output in the memory dump file with your program’s expected values.

Solution to my R-Type instructions test:

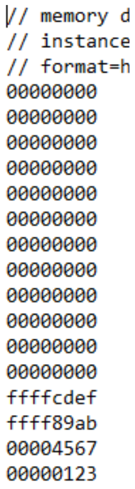


```
// memory 
// instanc
// format=
xxxxxxxx
00000002
0000000a
00000002
0000000f
0f010000
00000001
0000000a
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
```

3. Compare your J-Type output in the memory dump file with your program’s expected values.

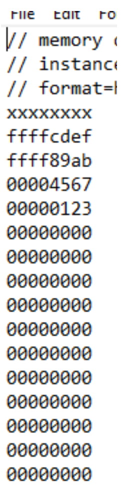
Solution to my R-Type instructions test:

Memory:



```
// memory d
// instance
// format=h
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
ffffcdef
ffff89ab
00004567
00000123
```

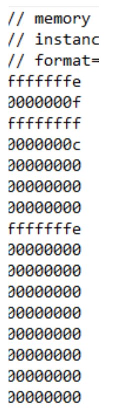
Registers:



```
// memory 
// instanc
// format=
xxxxxxxx
ffffcdef
ffff89ab
00004567
00000123
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
```

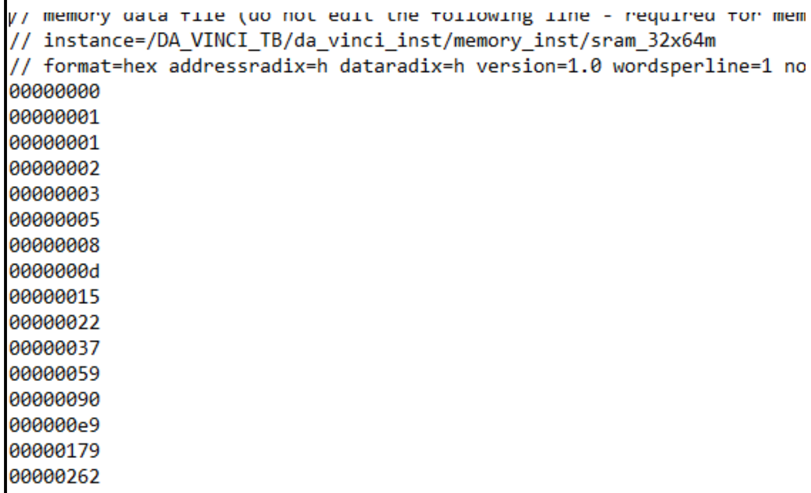
4. Compare your Test_System output in the memory dump file with your program’s expected values.

Solution to Test_System:



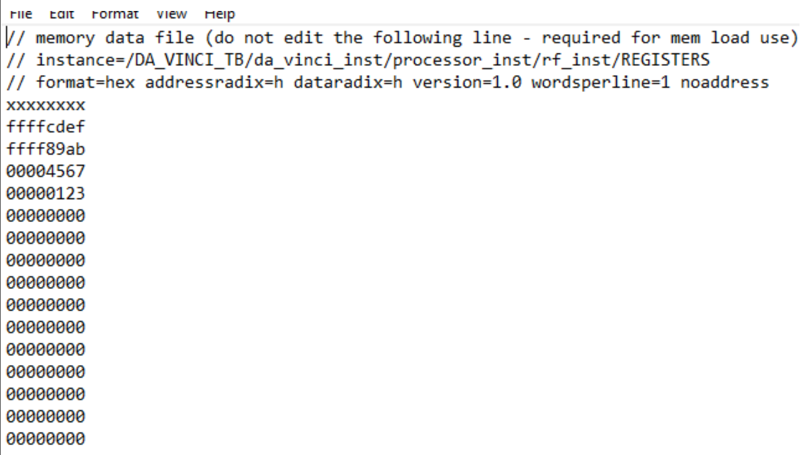
```
// memory
// instanc
// format=
fffffffe
0000000f
ffffffff
0000000c
00000000
00000000
00000000
fffffffe
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
```

5. Solution for fibbonacci.dat



```
// memory data file (do not edit the following line - required for mem
// instance=/DA_VINCI_TB/da_vinci_inst/memory_inst/sram_32x64m
// format=hex addressradix=h dataradix=h version=1.0 wordsperline=1 no
00000000
00000001
00000001
00000002
00000003
00000005
00000008
0000000d
00000015
00000022
00000037
00000059
00000090
000000e9
00000179
00000262
```

6. Solution for RevFib.dat



```
// memory data file (do not edit the following line - required for mem load use)
// instance=/DA_VINCI_TB/da_vinci_inst/processor_inst/rf_inst/REGISTERS
// format=hex addressradix=h dataradix=h version=1.0 wordsperline=1 noaddress
xxxxxxxx
ffffcdef
ffff89ab
00004567
00000123
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
```

XIII. Conclusion

By completing this project, I have demonstrated how an ALU is designed, how computer memory is designed, how a register file is designed, how a state machine is designed, and how a control unit is designed. I have learned in depth how these components function individually and in unison with the help of the control unit. I have learned how a microprocessor works and how assembly language instructions are implemented. Also, I have been familiarized with the Verilog language and have gained experience using the ModelSim application. In conclusion, I have learned how to create a behavioral model for CS 147DV in Verilog and virtually simulate a microprocessor.