NEW YORK INSTITUTE OF TECHNOLOGY

College of Engineering & Computing Sciences

CPU Design

VLSI Simulation

Team 4: Adina Tsibulevskiy; Emilio Santana-Ferro;
Mikhail Smirnov; Roy Reshef; Yordan Nikolov

EENG 661 - M01 Introduction to VLSI Design

Final Project

Spring 2024

I. Introduction

A 24-bit Central Processing Unit (CPU) is designed using Verilog to execute from a predetermined set of sixteen instructions that will output a 24-bit binary output. The CPU itself is made up of five components: Instruction Fetch, Instruction Decode, Arithmetic Logic Unit (ALU), Write-Back, and Memory. The ID, ALU, and Memory are instantiated as separate modules while the Write-Back is integrated into the Register File, and the Instruction Fetch is inside the CPU module. The memory has blocks designated for instructions as well as data. The design also includes memory-mapped peripherals, which function externally to the CPU that are able to interact with the inputs and outputs as well as have a designated block of memory. The two peripherals implemented in the system are a 24-bit timer and a 16-bit integer divider. The designed CPU is able to read and write data to and from memory, perform arithmetic and logic operations, and can interact with peripherals, and is simulated using testbenches.

II. Design

A. CPU

The CPU executes the 24-bit instructions that are structured in one of the three layouts shown in Fig. 1. Instructions that use data from registers will fall under the first structure. Instructions that use data and a value will fall under the second structure. Instructions that store and load data to and from the memory will be formatted like the third opcode structure.

The sixteen instructions programmed for the CPU design are listed in Table 1. The instructions are handled by the Instruction Decode (ID) module. Each instruction is assigned to a value from binary 0 through 15, followed by its destination and what inputs the instruction is pulling from. Fig. 3 shows the simulated pinout of the CPU design. The CPU is driven by its inputs listed as the reset signal, clock signal, instructions, and input data. The output ports are utilized to communicate with the peripherals.

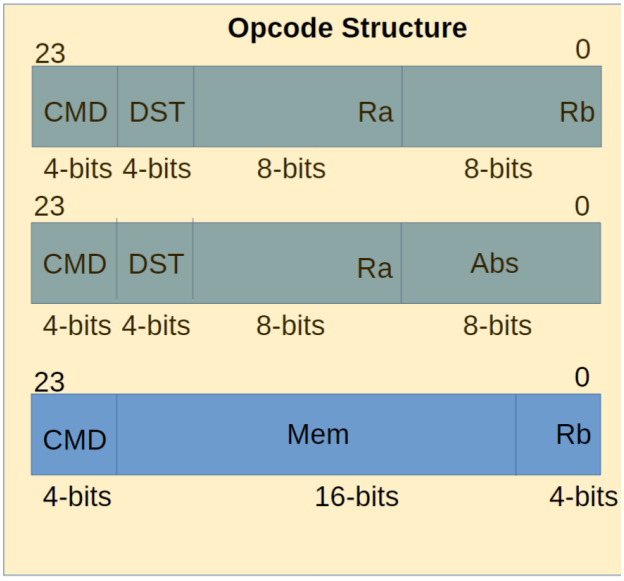

Figure 1: Opcode Structure

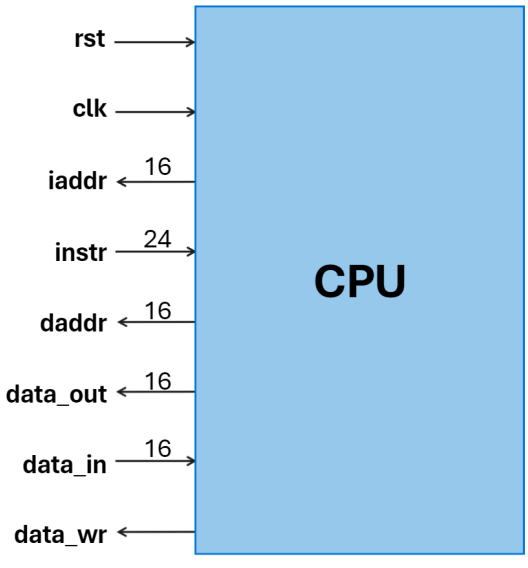

Figure 2: CPU Pinout

The Instruction Decode module in Appendix A takes in a 4-bit command value and maps it to the corresponding instruction listed in Table 1. The binary values are matched to the ALU Control module in Appendix B, which assigns the instructions to a binary value from 0 to 15. The Write-Back module is integrated amongst the other modules. The memory module in Appendix C loads and stores data. The *load* instruction takes data from the memory block and loads into a register. The *store* instruction takes data from a register and stores it into a line in the data memory block. The *jump* instruction allows for the program counter to be moved manually. The Register module, Appendix D, functions as internal, temporary memory that stores values from memory to be used by the operating blocks of the CPU. The Memory module, Appendix E, functions as the permanent memory of the CPU system. Data and instructions can be read from and written to in the memory blocks. Instructions are stored in the reserved instruction block which is also mapped to by the peripherals in the system.

Instruction	Opcode	Description
ADDRR Ra, Rb, Ry	0000_RyRaRb	Ry ← Ra + Rb
SUBRR Ra, Rb, Ry	0001_RyRaRb	Ry ← Ra - Rb
MULRR Ra, Rb, Ry	0010_RyRaRb	Ry ← Ra × Rb
XOR Ra, Rb, Ry	0011_RyRaRb	Ry ← Ra^Rb
INV Ra, Ry	0100_RyRaXX	Ry ← ~Ra
AND Ra, Rb, Ry	0101_RyRaRb	Ry ← Ra & Rb
OR Ra, Rb, Ry	0110_RyRaRb	Ry ← Ra | Rb
JMP	0111_XXRaXX	PC ← Ra
NOP	1000_XXXXXX	No Operation (Idle)
RESERVED	1001_XXXXXX	N/A
LD R, Mem	1010_MMMMRb	Rb ← [Mem]
ST Rb, Mem	1011_MMMMRb	[Mem] ← R
ADDRA Ra, #Imm, Ry	1100_RyRaIM	Ry ← Ra + #Imm
MULRA Ra, #Imm, Ry	1101_RyRaIM	Ry ← Ra × #Imm
RESERVED	1110_XXXXXX	N/A
RESERVED	1111_XXXXXX	N/A

Table 1: Instruction Set

B. Memory-Mapped Peripherals

1. 24-bit Countdown Timer

```
module twofourCounter(
    input wire clk,
    input wire rst,
    input wire [23:0] given_value,
    output reg [23:0] count
);

// Reset the counter
always @(posedge clk or posedge rst) begin
    if (rst)
        count <= given_value;
    else if (count > 0)
        count <= count - 1;
end

endmodule
```

The first peripheral is a 24-bit countdown timer that counts down from a given value, limited to 24-bits, down to zero. The second peripheral is a 16-bit integer divider that uses two registers to store its output, one for the quotient and one for the remainder. The peripherals are memory mapped into a designated block in the memory to be utilized by the CPU.

2. 16-bit Integer Divider

```
module divider(divisor, dividend, remainder, result);
input [15:0] divisor, dividend;
output reg [15:0] result, remainder;
// Variables
integer i;
reg [15:0] divisor_temp, dividend_temp;
reg [15:0] temp;
always @(divisor or dividend)
begin
      divisor_temp = divisor;
      dividend_temp = dividend;
      temp = 0;
      for(i = 0;i < 8;i = i + 1)
      begin
            temp = {temp[6:0], dividend_temp[7]};
            dividend_temp[7:1] = dividend_temp[6:0];
            /* Substract the Divisor Register from the Remainder
Register place the result in remainder register (temp variable here!)
            */
            temp = temp - divisor_temp;
            // Compare the Sign of Remainder Register (temp)
            if(temp[7] == 1)
            begin
            /*Restore original value by adding the Divisor Register to
Remainder Register and placing the sum in Remainder Register Shift
Quotient by 1 and Add 0 to last bit.
            */
                  dividend_temp[0] = 0;
                  temp = temp + divisor_temp;
            end
            else
            begin
            /*
                  * Shift Quotient to left. Set right most bit to 1.
            */
                  dividend_temp[0] = 1;
            end
      end
```

C. Testing and Results

1. ALU Testbench

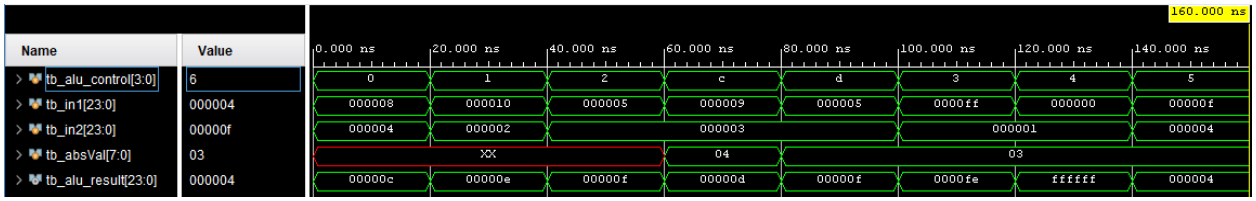

Figure 1: ALU Testbench Result Demonstrating All Nine Arithmetic/Logic Operations

2. Instruction Decode Testbench

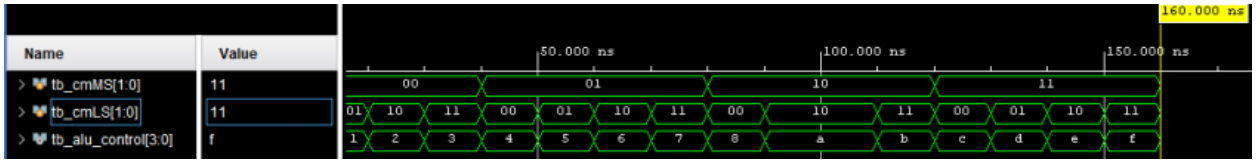

Figure 2: ID Testbench Result Decoding Every Type of Instruction Sent to ALU

3. Memory (Instruction Memory and Data Memory Blocks) Testbench

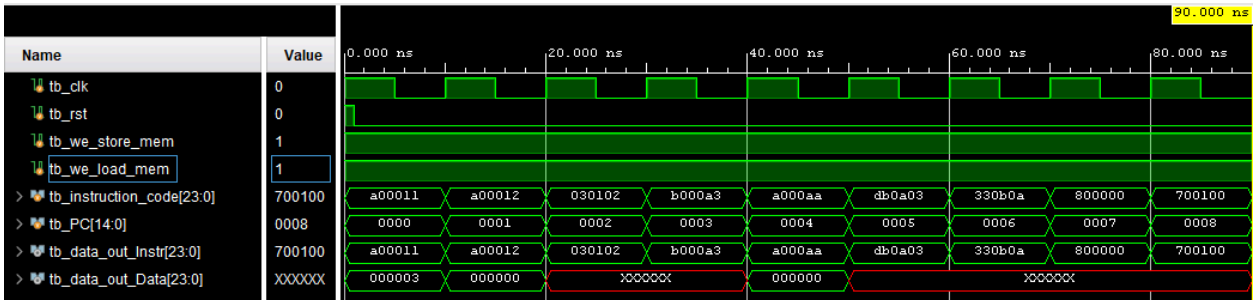

Figure 3: Memory Module Result Loading Instruction Opcodes from Instruction Memory and Block Data From Data Memory Block

4. Integer Divider

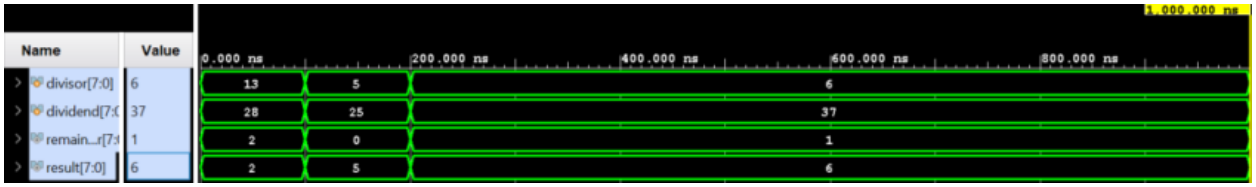

Figure 4: Integer Divider Testbench Results

5. 24-bit Timer that counts down from a given value to zero.

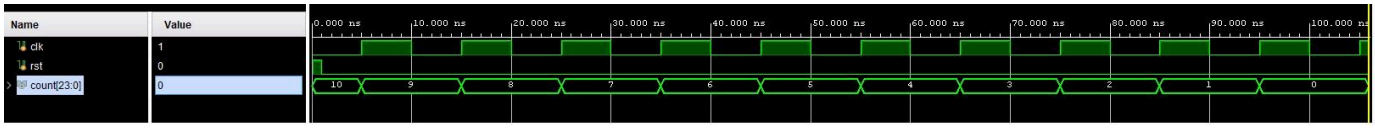

Figure 5: 24-Bit Timer Counting Down Testbench Result

III. Conclusion

The CPU is designed to execute every instruction from the table. Each instruction takes two clock cycles to execute and is 24 bits long. The instructions are designated with a 4-bit value at the front of the opcode. The CPU module consists of instruction fetch, instruction decode, an ALU, write-back functionality, and memory for external storage in addition to its registers.The 24-bit timer and 16-bit integer divider are two peripherals memory-mapped to the instruction block. The testbenches verify the expected outputs of the modules and overall system. The main program loaded in the memory block of the memory module consists of the opcodes that correspond to the following instructions shown below.

[0x00]: LD R1 <- DATA_MEM[1]
[0x01]: LD R2 <- DATA_MEM[2]
[0x02]: AddRR R3 <- R1 + R2
[0x03]: ST R3 -> MEM[10]
[0x04]: LD R10 <- MEM[10]
[0x05]: MULRA R11 <- R10 * 3
[0x06]: XOR R3 <- R11 ^ R10
[0x07]: NOP
[0x08]: JMP R1 (R1 = line 3)

This program first loads predefined values (3 and 7 respectively) from memory into registers R1 and R2, which then get added and stored into register R3. The resulting sum is stored in the data memory block of line 10 and is then loaded into register R10. Next, the value of R10 is multiplied by 3 and stored in R11. Following this instruction, XOR is performed from the data stored in R11 and R10, in which the result is stored in R3. Finally, a NOP is executed and the JMP instruction resets the program back to memory address line 0x03.

IV. Appendix

A. Instruction Decode Module

```
module Instruction_Decode(
    input [1:0] cmMS, // 2 MSBs of instruction portion of Opcode
    input [1:0] cmLS, // 2 LSBs of instruction portion of Opcode
    output reg [3:0] alu_control,
    output reg regwrite_control
);
    always @(cmMS or cmLS)
    begin
```

```
        regwrite_control = 1;
        case (cmMS)
            0: begin
                if(cmLS == 0)
                    alu_control = 4'd0; // ADDRR Ry <- Ra + Rb
                else if(cmLS == 1)
                    alu_control = 4'd1; // SUBRR Ry <- Ra - Rb
                  else if(cmLS == 2)
                    alu_control = 4'd2; // MULRR Ry <- Ra X Rb
                  else if(cmLS == 3)
                    alu_control = 4'd3; // XORRR Ry <- Ra ^ Rb
                end
              1: begin
                if(cmLS == 0)
                    alu_control = 4'd4; // INV Ra Ry <- ~Ra
                else if(cmLS == 1)
                    alu_control = 4'd5; // ANDRR Ry <- Ra & Rb
                  else if(cmLS == 2)
                    alu_control = 4'd6; // ORRR Ry <- Ra | Rb
                  else if(cmLS == 3)
                    alu_control = 4'd7; // JMP PC <- Ra
               end
           2: begin
                if(cmLS == 0)
                    alu_control = 4'd8; // NOP
                else if(cmLS == 1)
                    alu_control = 4'd9; // RESERVED
                  else if(cmLS == 2)
                    alu_control = 4'd10; // LD Rb <- [Mem]
                  else if(cmLS == 3)
                    alu_control = 4'd11; // ST [Mem] <- Rb
                end
           3: begin
                if(cmLS == 0)
                    alu_control = 4'd12; // ADDRA Ry <- Ra + #Imm
                else if(cmLS == 1)
                    alu_control = 4'd13; // MULRA Ry <- Ra X #Imm
                  else if(cmLS == 2)
                    alu_control = 4'd14; // RESERVED
                  else if(cmLS == 3)
                    alu_control = 4'd15; // RESERVED
                end
            endcase
    end
endmodule
```

B. ALU Module

```
/*
ALU Control lines | Function
        0000    ADDRR Ry <- Ra + Rb
        0001    SUBRR Ry <- Ra - Rb
        0010    MULRR Ry <- Ra X Rb
        0011    XORRR Ry <- Ra ^ Rb
        0100    INV Ra Ry <- ~Ra
        0101    ANDRR Ry <- Ra & Rb
        0110    ORRR Ry <- Ra | Rb
        0111    XORRR PC <- Ra
        1000    NOP
        1001    RESERVED
        1010    LD Rb <- [Mem]
        1011    ST [Mem] <- Rb
        1100    ADDRA Ry <- Ra + #Imm
        1101    MULRA Ry <- Ra X #Imm
        1110    RESERVED
        1111    RESERVED
*/
module ALU (
    input [23:0] in1,in2,
    input [7:0] absVal
    input[3:0] alu_control,
    output reg [31:0] alu_result,
);
    always @(*)
    begin
      // Operating based on control input
        case(alu_control)

     //Register Operation
        4'b0000: alu_result = in1+in2;      //Ry <- Ra + Rb
        4'b0001: alu_result = in1-in2;      //Ry <- Ra - Rb
        4'b0010: alu_result = in1*in2;      //Ry <- Ra X Rb
        4'b0011: alu_result = in1^in2;      //Ry <- Ra ^ Rb
        4'b0100: alu_result = ~in1;         //Ry <- ~Ra
        4'b0101: alu_result = in1&in2;      //Ry <- Ra & Rb
```

```
        4'b0110: alu_result = in1|in2;      //Ry <- Ra | Rb

      //Absolute
        4'b1100: alu_result = in1 + absVal; //Ry <- Ra + #Imm
        4'b1101: alu_result = in1 * absVal; //Ry <- Ra X #Imm

      //Jump
      //Delegated to Instruction Fetch inside CPU (4'b0111)

      //Memory
     //Delegated to a direct flag-based line between the two
     //LD Rb <- [Mem] (4'b1010)
     //ST [Mem] <- Rb (4'b1011)

      //Reserved or NOP
        //NOP (4'b1000)
        //RESERVED (4'b1001),(4'b1110),(4'b1111)

        endcase
    end
endmodule
```

C. Memory Module

```
module memory(
    input [23:0] instruction_code, // Used for reading or writing to
data memory address
    input [23:0] data_in_mem, // Data to be written to data memory
address
     input[14:0] mem_PC, // Program counter (used for instruction
memory address)
     input mem_clk,
     input mem_rst,
    input we_store_mem, // Storing occurs when flag is up (1) at
posedge clk cycle
     input we_load_mem,    // Loading occurs when flag is up (1) at
posedge clk cycle
    output [23:0] data_out_Instr, // Output of opcode stored in
Instruction Memory
    output[23:0] data_out_Data // Output of data stored in Data
```

```
Memory
);

parameter INSTR_MEM_DEPTH = 32_768; // 32K instructions (represented
by 2^15 address lines)
parameter DATA_MEM_DEPTH = 16_384; // 16K 24-bit words (represented
by 2^14 address lines)
parameter DATA_WIDTH = 24;

// Instruction memory instantiation using Block RAM
reg [DATA_WIDTH-1:0] instruction_mem [0:INSTR_MEM_DEPTH-1];




always @(posedge mem_rst) begin
    // Opcodes of program to be executed
    // LD R1 <- DATA_MEM[1]:
    instruction_mem[0] = 24'b1010_0000_0000_0000_0001_0001;
    // LD R2 <- DATA_MEM[2]:
    instruction_mem[1] = 24'b1010_0000_0000_0000_0001_0010;
    // AddRR R3 <- R1 + R2:
    instruction_mem[2] = 24'b0000_0011_0000_0001_0000_0010;
    // ST R3 -> MEM[10]:
    instruction_mem[3] = 24'b1011_0000_0000_0000_1010_0011;
    // LD R10 <- MEM[10]:
    instruction_mem[4] = 24'b1010_0000_0000_0000_1010_1010;
    // MULRA R11 <- R10 * 3:
    instruction_mem[5] = 24'b1101_1011_0000_1010_0000_0011;
    // XOR R3 <- R11 ^ R10:
    instruction_mem[6] = 24'b0011_0011_0000_1011_0000_1010;
    // NOP:
    instruction_mem[7] = 24'b1000_0000_0000_0000_0000_0000;
    // JMP R1 (R1 = line 3):
    instruction_mem[8] = 24'b0111_0000_0000_0001_0000_0000;

end
```

```
// Data memory instantiation using Block RAM
reg [DATA_WIDTH-1:0] data_mem [0:DATA_MEM_DEPTH-1];



always @(posedge mem_rst) begin
    // Hardcoded values in memory
    data_mem[1] = 24'b0000_0000_0000_0000_0000_0011; // Dummy value
initialized as "3" for R1
    data_mem[2] = 24'b0000_0000_0000_0000_0000_0111; // Dummy value
initialized as "7" for R2
end

// Instruction memory read
assign data_out_Instr = instruction_mem[mem_PC]; // Reading from
16-bit memory address in opcode

// Data load from Memory to Register
always @ (posedge mem_clk) begin
     if (we_load_mem) begin
        data_out_Data <= data_mem[instruction_code[19:4]];
           //data_out_Data should go to register
     end
end
//Data store from Register to Memory
always @ (posedge mem_clk) begin
     if (we_store_mem) begin
        data_mem[instruction_code[19:4]] <= data_in_mem;
           //data_in_mem should be the register data
     end
end

endmodule
```

D. Register Module

```
module REG_FILE #(
    parameter SIZE = 24, // 24-bit sized registers
    parameter DEPTH = 16 // 16 registers (4 bit address)
    )
    (
     input [4-1:0] reg_import_to_address, //Address for Memory to
Register Operation
    input [4-1:0] reg_read_addr1, // Address of register Ra that will
be read
    input [4-1:0] reg_read_addr2, // Address of register Rb that will
be read
    input [4-1:0] reg_write_addr, // Address of register Ry that will
be written to
    input [SIZE-1:0] reg_data_in, // Input data to be written
    input reg_clk, //Global clk
    input reg_rst, //Global rst
    input reg_we, // Write to Register enable
     input mem_to_reg_we, //Write from Memory to Register
     input reg_to_mem_we, //Write to Register from Memory
    output [SIZE-1:0] reg_data_out1, // Output data read from
register Ra
    output [SIZE-1:0] reg_data_out2 // Output data read from register
Rb
     output [SIZE-1:0] reg_data_out3 //Output data read from select
register
    );

    reg [SIZE-1:0] reg_data [DEPTH-1:0]; // Register memory locations
(16 registers of 24 bits each)

    // Dummy values for registers on reset
    always @(posedge reg_rst) begin
        reg_data[0]  = {24{1'b0}};
        reg_data[1]  = {24{1'b0}};
        reg_data[2]  = {24{1'b0}};
        reg_data[3]  = {24{1'b0}};
        reg_data[4]  = {24{1'b0}};
        reg_data[5]  = {24{1'b0}};
```

```
        reg_data[6]  = {24{1'b0}};
        reg_data[7]  = {24{1'b0}};
        reg_data[8]  = {24{1'b0}};
        reg_data[9]  = {24{1'b0}};
        reg_data[10] = {24{1'b0}};
        reg_data[11] = {24{1'b0}};
        reg_data[12] = {24{1'b0}};
        reg_data[13] = {24{1'b0}};
        reg_data[14] = {24{1'b0}};
        reg_data[15] = {24{1'b0}};
    end


    // Read data stored in Ra and Rb
    // (reg_read_addr is the register index)
    assign reg_data_out1 = reg_data[reg_read_addr1];
    assign reg_data_out2 = reg_data[reg_read_addr2];

    // Write Logic
    // At each clock pulse, write data to reg
    always @(posedge reg_clk) begin
        if(reg_we) begin
            //reg_write_addr is the register index
            reg_data[reg_write_addr] <= reg_data_in;
        end
    end
      always @(posedge reg_clk) begin
        if(mem_to_reg_we) begin
            //mem_to_reg_write_addr is the register index
            reg_data[reg_read_addr2] <= reg_import_to_address;
        end
    end
           always @(posedge reg_clk) begin
        if(reg_to_mem_we) begin
            //reg_to_mem_write_addr is the register index
            reg_data_out3 <= reg_data[reg_read_addr2];
        end
    end
endmodule
```

E. CPU Module

```
module CPU_24b(
    input clk,
    input rst,
    input [23:0] instr,
    input [15:0] data_in,
    output reg [15:0] iaddr,
    output reg [15:0] daddr,
    output [15:0] data_out,
    output data_wr // bit for if data was written
    );

    // instr opcode structure:
    // RR type: [ instr[23:20] instr[19:16] instr[15:8] instr[7:0] ]
    //               CMD            DST          Ra          Rb

    // RA type: [ instr[23:20] instr[19:16] instr[15:8] instr[7:0] ]
    //                  CMD         DST         Ra          Abs

    // Mem type: [ instr[23:20] instr[19:4] instr[3:0] ]
    //                  CMD         Mem         Rb

    /*Instruction opcodes
    parameter OP_ADDRR = 4'b0000;
    parameter OP_SUBRR = 4'b0001;
    parameter OP_MULRR = 4'b0010;
    parameter OP_XOR   = 4'b0011;
    parameter OP_INV   = 4'b0100;
    parameter OP_AND   = 4'b0101;
    parameter OP_OR    = 4'b0110;
    parameter OP_JMP   = 4'b0111;
    parameter OP_NOP   = 4'b1000;
    parameter OP_LD    = 4'b1010;
    parameter OP_ST    = 4'b1011;
    parameter OP_ADDRA = 4'b1100;
    parameter OP_MULRA = 4'b1101;
    */
    // Reserved opcodes
    //parameter OP_RESERVED = 4'b1001;
```

```
    //parameter OP_RESERVED = 4'b1110;
    //parameter OP_RESERVED = 4'b1111;

    // Datapath Module

////////////////////////////////////////////////////////////////////////
     always @(negedge clk or posedge rst)
          begin
               if(~rst) begin
               //Reset signals by setting all flags down (<=)
               we <= 0;
               PC_update_flag <= 0;
               we_load_Mem <= 0;
               we_store_Mem <= 0;
               end


               else if (alu_control_temp == 4'd10) begin
               //Load  Memory to Register flag enabled
               we_load_Mem <= 1;

               end

               else if (alu_control_temp == 4'd11) begin
               ///Store  Register to Memory flag enabled
               we_store_Mem <= 1;
               end

               else if (alu_control_temp == 4'd7) begin
               //JMP instruction by enabling the JMP flag
               PC_update_flag <= 1;
               end

               else if (alu_control_temp < 4'd7 ||
 alu_control_temp == 4'd12 || alu_control_temp == 4'd13) begin
               //Register based arithmetic instructions enabled
               we <= 1;
               end
```

```
                else /*if (alu_control_temp == 4'd8 ||
alu_control_temp == 4'd9
                || alu_control_temp == 4'd14 || alu_control_temp ==
4'd15) */ begin
                //Execute a NOP instruction by skiping a negedge
cycle or 2 posedge cycles
                //Disable all writing
                we <=0;
                PC_update_flag <= 0;
                we_load_Mem <= 0;
                we_store_Mem <= 0;

                end
            end

    // Register File with integrated Write Back (WB) Instantiation
////////////////////////////////////////////////////////////////////
    reg we; // Write enable

    REG_FILE reg_file_0(
              .reg_import_to_address(goes_to_reg_address), //Data to
be stored in the register
            .reg_read_addr1(instr[15-4:8]), // Address of Ra
            .reg_read_addr2(instr[7-4:0]), // Address of Rb
            .reg_write_addr(instr[19:16]), // Address of Ry
            .reg_data_in(alu_result_temp), // 24 bit data for writing
to Ry
            .reg_clk(clk), //Global clk
            .reg_rst(rst), //Global rst
            .reg_we(we), //Write flag for ALU arithmetic
               .mem_to_reg_we(we_load_Mem), // LD flag
               .reg_to_mem_we(we_store_Mem) // ST flag
            .reg_data_out1(data_out1), // Read data of Ra (instantly)
            .reg_data_out2(data_out2), // Read data of Rb (instantly)
               .reg_data_out3(data_out3) //Read data of select Reg for
Store
    );
```

```
// Memory
//////////////////////////////////////////////////////////////////

            reg we_store_Mem; //Initiating the ST flag
            reg we_load_Mem;  //Initiating the LD flag

    memory mem0(
            .instruction_code(instr[19:4]),//Memory index from op code
            .data_in_mem(data_out3), //
            .mem_PC(PC), //Connect PC from IF to Memory
            .mem_clk(clk),//Global clk connect
            .mem_rst(rst),//Global rst connect
            .we_store_mem(we_store_Mem), //Flag for ST instruction
            .we_load_mem(we_load_Mem), //Flag for LD instruction
            .data_out_Instr(instr), //Load op code
            .data_out_Data(goes_to_reg_address) // Mem data to LD Reg
     );


    // Instruction Fetch Unit

//////////////////////////////////////////////////////////////////

            // Program Counter
            reg [23:0] PC;

    always @(posedge clk, posedge rst) begin
        if(rst) begin
            PC <= 0;
        end
           else if (PC_update_flag) begin
           PC <= data_out1; // Replace PC with Ra for JMP
           end
        else begin
            PC <= PC + 1; //Proceed to the next word
        end
    end
```

```
    // Instruction Decoder / Control Unit (Generates control signals
for ALU) Initialization

////////////////////////////////////////////////////////////////////
    reg [3:0] alu_control_temp; //ALU control connector variable

    Instruction_Decode ID(
        .cmMS(instr[23:22]), //Two most significant bits of CMD
        .cmLS(instr[21:20]), //Two least significant bits of CMD
        .alu_control(alu_control_temp), //ALU control signal
    );


    // ALU Instantiation

////////////////////////////////////////////////////////////////////

    ALU alu0(
        .in1(data_out1), // Data of Ra
        .in2(data_out2), // Data of Rb
        .absVal(instr[7:0]), //Address from #abs case opp code
        .alu_control(instr[23:20]), //Control signal from CMD
        .alu_result(data_out) //Calculated ALU result
    );

endmodule
```