Computer Science / Computer Engineering 142

Advanced Computer Organization

Term Project: Phase Two

Team: 5

By: Ethan Kinyon & Andrew Enright

12/05/2017

TR 5:30pm

Professor: Behnam Arad

Table of Contents

Objective

The purpose of the term project is to design and simulate a datapath and control unit for a pipelined system, which will be able to execute the following assembly instruction.

Function	Syntax	opcode	op1	op2	funct. Code	type	Operation
Signed addition	add op1, op2	1111	reg	reg	0000	A	op1 = op1 + op2
Signed subtraction	sub op1, op2	1111	reg	reg	0001	A	op1 = op1 - op2
Signed multiplication	mul op1, op2	1111	reg	reg	0100	A	op1 = op1 * op2 op1: Product (lower half) R15: Product (upper half)
Signed division	div op1, op2	1111	reg	reg	0101	A	op1: 16-bit quotient R15: 16-bit remainder
Move	mv op1, op2	1111	reg	reg.	0111	A	op1 <= op2
SWAP	swp op1, op2	1111	reg	reg	1000	A	op1 <= op2 op2 <= op1
AND immediate	andi op1, op2	1000	reg	Immd.	N/A	C	op1 = op1 & {8'b0, constant}
OR immediate	or op1, op2	1001	reg	Immd.	N/A	C	op1 = op1 | {8'b0, constant}
Load byte unsigned	lbu op1, immd (op2)	1010	reg	reg	N/A	B	op1 = {8'b0, Mem [immd + op2] } (sign extend immd)
Store byte	sb op1, immd (op2)	1011	reg	reg	N/A	B	Mem [immd + op2](7:0) = op1(7:0) (sign extend immd)
Load	lw op1, immd (op2)	1100	reg	reg	N/A	B	op1 = Mem [immd + op2] (sign extend immd)
Store	sw op1, immd (op2)	1101	reg	reg	N/A	B	Mem [immd + op2] = op1 (sign extend immd)
Branch on less than	blt op1, op2	0101	reg	immd.	N/A	C	if (op1 < R15) then PC = PC + op2 (sign extend op2 & shift left)
Branch on greater than	bgt op1, op2	0100	reg	immd.	N/A	C	if (op1 > R15) then PC = PC + op2 (sign extend op2 & shift left)

Function	Syntax	opcode	op1	op2	funct. Code	type	Operation
Branch on equal	beq op1, op2	0110	reg	immd.	N/A	C	if (op1 = R15) then PC = PC + op2 (sign extend op2 & shift left)
jump	jmp op1	0001	off-set	-------	N/A	D	pc = pc + op1 (sign extend op1& shift left)
halt	Halt	0000	-----	------	N/A	D	halt program execution

Status Report

Complete this form by typing the requested information and include the completed form in your report after TOC. Gray cells will be filled by the instructor.

Name	*%Contribution*	*Grade*
Ethan Kinyon	40	
Andrew Enright	60	

Please do not write in the first table

Project Report/Presentation	*20%*	/200
Functionality of the individual component	*40%*	/400
Functionality of the overall design	*25%*	/250
Design Approach	*5%*	/50
Total Points		/900

A: List all the instructions that were implemented correctly and verified by the assembly program on your system:

Instructions	*State any issue regarding the instruction.*
Signed Addition	No issue
Signed Subtraction	No issue
signed multiplication	No issue
signed division	No issue
Move	Not in ALU
Swap	Not in ALU
AND Immediate	No issue
OR Immediate	No issue
Load byte unsigned	Will output whole word, not just byte
Store Byte	No issues
Load	No issues
Store	No Issues
branch on less than	Not working
branch on greater than	Not working
branch on equal to	Not working
Jump	No issue
Halt	No issue

B: Fill out the next table:

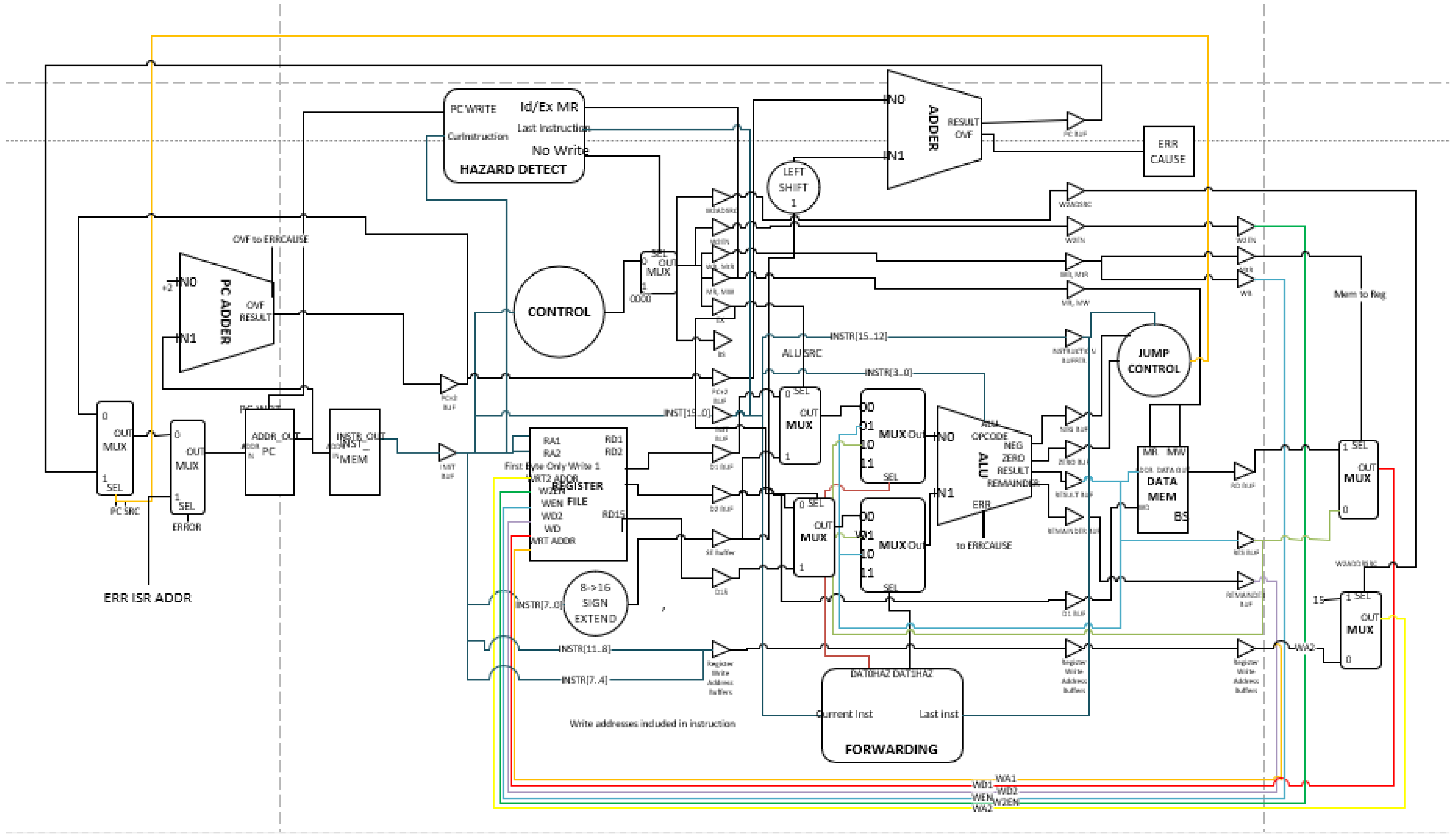
Individual Components	Does your system have this component?	List the student who designed and verified the block	Does it work?	List problems with the component, if any.
ALU	Yes	Ethan	Yes	NA
ALU control unit	No	NA	NA	NA
Memory Unit	Yes	Ethan	Yes	NA
Register File/Other Reg	Yes	Andrew	Yes	NA
PC	Yes	Ethan	Yes	NA
Instruction Registers	Yes	Ethan	Yes	NA
Multiplexors	Yes	Andrew	Yes	NA
Exception handler 1. Unknown opcode 2. Arithm. Overflow	Yes	Andrew	Yes	NA
Control Units 1. Main 2. Forwarding 3. lw hazard detection	Yes	Andrew	Yes	NA

How many stages to you have in your pipeline? 5

C: State any issues regarding the overall operation of the datapath? Be specific.

NA

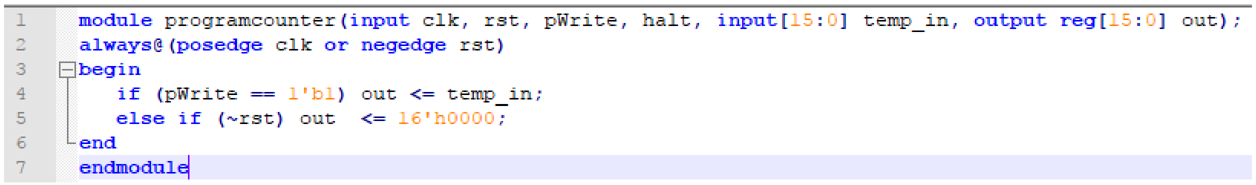
Datapath



HAZARD DETECT
PC WRITE
Id/Ex MR
Last Instruction
No Write
CurInstruction
ADDER
RESULT
OVF
IN0
IN1
ERR CAUSE
LEFT SHIFT 1
PC ADDER
OVF to ERRCAUSE
CONTROL
MUX
PC
INSTR_MEM
REGISTER FILE
RA1
RA2
RD1
RD2
RD15
WA2
WEN
WD
WRT ADDR
First Byte Only Write 1
8->16 SIGN EXTEND
ALU
OPCODE
NEG
ZERO
RESULT
REMAINDER
ERR
to ERRCAUSE
ALU SRC
INSTR[15..12]
INSTR[3..0]
INSTR[7..0]
INSTR[11..8]
INSTR[7..4]
INST[15..0]
JUMP CONTROL
DATA MEM
MR
MW
BS
Mem to Reg
FORWARDING
DATAHAZ DAT1HAZ
Current Inst
Last inst
Write addresses included in instruction
ERR ISR ADDR
PC SRC
ERROR
SEL
OUT
WA1
WD1
WD2
WEN
W2EN
WA2
15

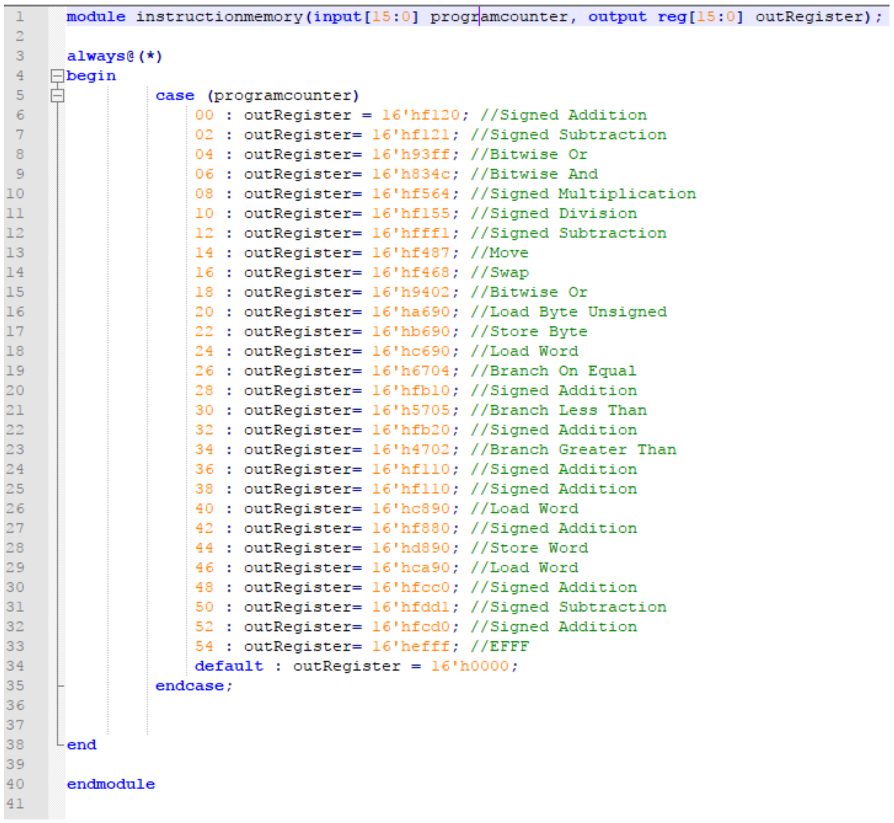
Stage One: Instruction Fetch

Program Counter:



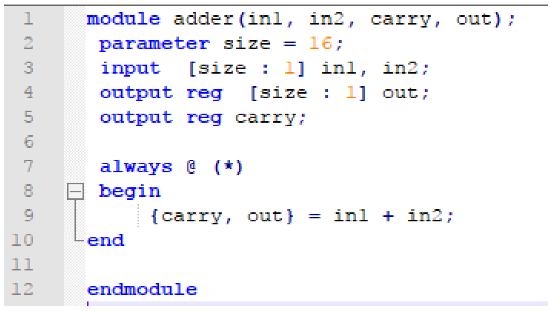
```verilog
module programcounter(input clk, rst, pWrite, halt, input[15:0] temp_in, output reg[15:0] out);
always@(posedge clk or negedge rst)
begin
    if (pWrite == 1'b1) out <= temp_in;
    else if (~rst) out  <= 16'h0000;
end
endmodule
```

Instruction Memory:



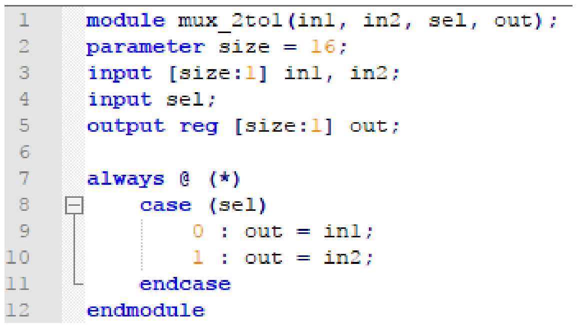
```verilog
module instructionmemory(input[15:0] programcounter, output reg[15:0] outRegister);

always@(*)
begin
        case (programcounter)
            00 : outRegister = 16'hf120; //Signed Addition
            02 : outRegister= 16'hf121; //Signed Subtraction
            04 : outRegister= 16'h93ff; //Bitwise Or
            06 : outRegister= 16'h834c; //Bitwise And
            08 : outRegister= 16'hf564; //Signed Multiplication
            10 : outRegister= 16'hf155; //Signed Division
            12 : outRegister= 16'hfff1; //Signed Subtraction
            14 : outRegister= 16'hf487; //Move
            16 : outRegister= 16'hf468; //Swap
            18 : outRegister= 16'h9402; //Bitwise Or
            20 : outRegister= 16'ha690; //Load Byte Unsigned
            22 : outRegister= 16'hb690; //Store Byte
            24 : outRegister= 16'hc690; //Load Word
            26 : outRegister= 16'h6704; //Branch On Equal
            28 : outRegister= 16'hfb10; //Signed Addition
            30 : outRegister= 16'h5705; //Branch Less Than
            32 : outRegister= 16'hfb20; //Signed Addition
            34 : outRegister= 16'h4702; //Branch Greater Than
            36 : outRegister= 16'hf110; //Signed Addition
            38 : outRegister= 16'hf110; //Signed Addition
            40 : outRegister= 16'hc890; //Load Word
            42 : outRegister= 16'hf880; //Signed Addition
            44 : outRegister= 16'hd890; //Store Word
            46 : outRegister= 16'hca90; //Load Word
            48 : outRegister= 16'hfcc0; //Signed Addition
            50 : outRegister= 16'hfdd1; //Signed Subtraction
            52 : outRegister= 16'hfcd0; //Signed Addition
            54 : outRegister= 16'hefff; //EFFF
            default : outRegister = 16'h0000;
        endcase;


end

endmodule
```

Adder:



```
module adder(in1, in2, carry, out);
 parameter size = 16;
  input   [size : 1] in1, in2;
  output reg  [size : 1] out;
  output reg carry;

  always @ (*)
 begin
     {carry, out} = in1 + in2;
end

endmodule
```

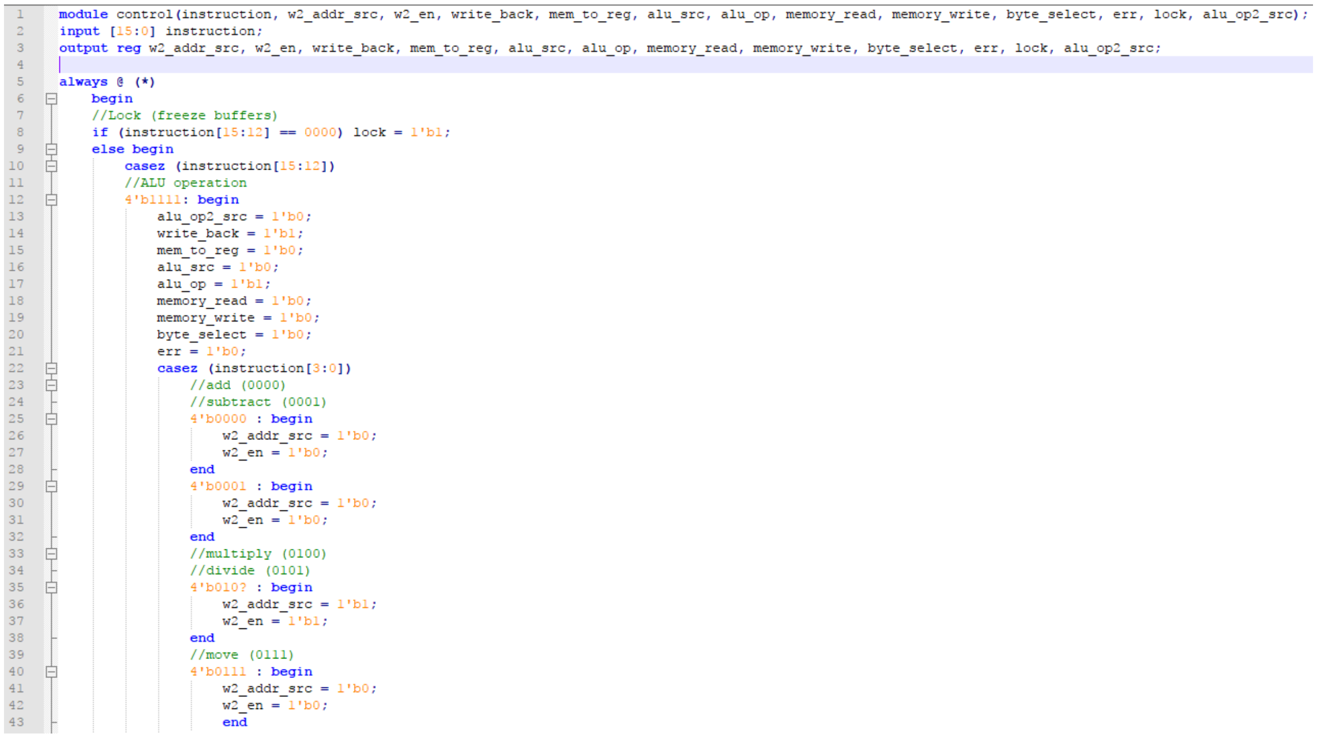
Multiplexor 2 to 1:



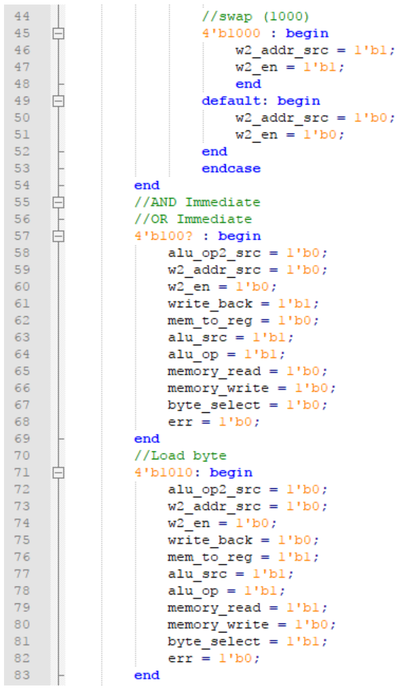
```
module mux_2to1(in1, in2, sel, out);
parameter size = 16;
input [size:1] in1, in2;
input sel;
output reg [size:1] out;

always @ (*)
    case (sel)
        0 : out = in1;
        1 : out = in2;
    endcase
endmodule
```

Stage Two: Instruction Decode

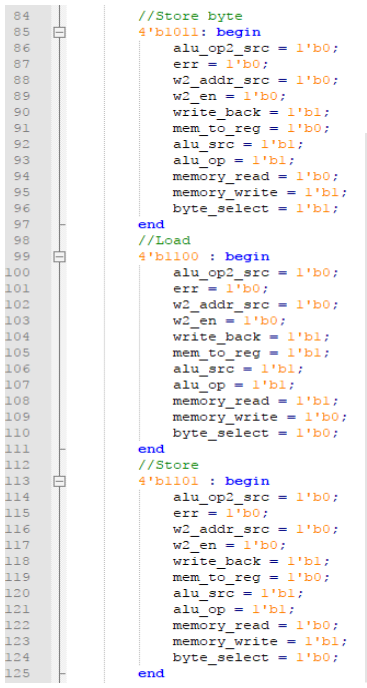
Control Unit:



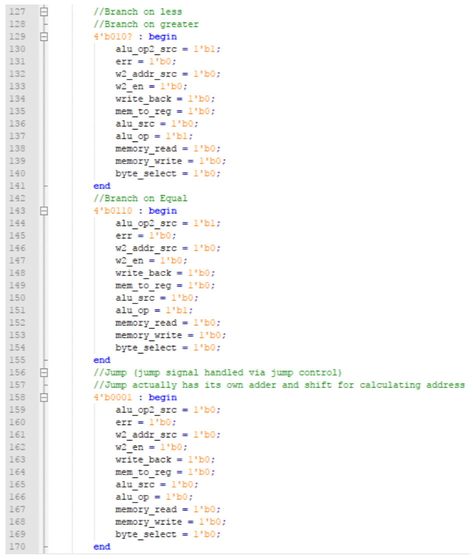
```verilog
module control(instruction, w2_addr_src, w2_en, write_back, mem_to_reg, alu_src, alu_op, memory_read, memory_write, byte_select, err, lock, alu_op2_src);
input [15:0] instruction;
output reg w2_addr_src, w2_en, write_back, mem_to_reg, alu_src, alu_op, memory_read, memory_write, byte_select, err, lock, alu_op2_src;

always @ (*)
    begin
    //Lock (freeze buffers)
    if (instruction[15:12] == 0000) lock = 1'b1;
    else begin
        casez (instruction[15:12])
        //ALU operation
        4'b1111: begin
            alu_op2_src = 1'b0;
            write_back = 1'b1;
            mem_to_reg = 1'b0;
            alu_src = 1'b0;
            alu_op = 1'b1;
            memory_read = 1'b0;
            memory_write = 1'b0;
            byte_select = 1'b0;
            err = 1'b0;
            casez (instruction[3:0])
                //add (0000)
                //subtract (0001)
                4'b0000 : begin
                    w2_addr_src = 1'b0;
                    w2_en = 1'b0;
                end
                4'b0001 : begin
                    w2_addr_src = 1'b0;
                    w2_en = 1'b0;
                end
                //multiply (0100)
                //divide (0101)
                4'b010? : begin
                    w2_addr_src = 1'b1;
                    w2_en = 1'b1;
                end
                //move (0111)
                4'b0111 : begin
                    w2_addr_src = 1'b0;
                    w2_en = 1'b0;
                    end
```



```verilog
                //swap (1000)
                4'b1000 : begin
                    w2_addr_src = 1'b1;
                    w2_en = 1'b1;
                    end
                default: begin
                    w2_addr_src = 1'b0;
                    w2_en = 1'b0;
                end
                endcase
        end
        //AND Immediate
        //OR Immediate
        4'b100? : begin
            alu_op2_src = 1'b0;
            w2_addr_src = 1'b0;
            w2_en = 1'b0;
            write_back = 1'b1;
            mem_to_reg = 1'b0;
            alu_src = 1'b1;
            alu_op = 1'b1;
            memory_read = 1'b0;
            memory_write = 1'b0;
            byte_select = 1'b0;
            err = 1'b0;
        end
        //Load byte
        4'b1010: begin
            alu_op2_src = 1'b0;
            w2_addr_src = 1'b0;
            w2_en = 1'b0;
            write_back = 1'b1;
            mem_to_reg = 1'b1;
            alu_src = 1'b1;
            alu_op = 1'b1;
            memory_read = 1'b1;
            memory_write = 1'b0;
            byte_select = 1'b1;
            err = 1'b0;
        end
```

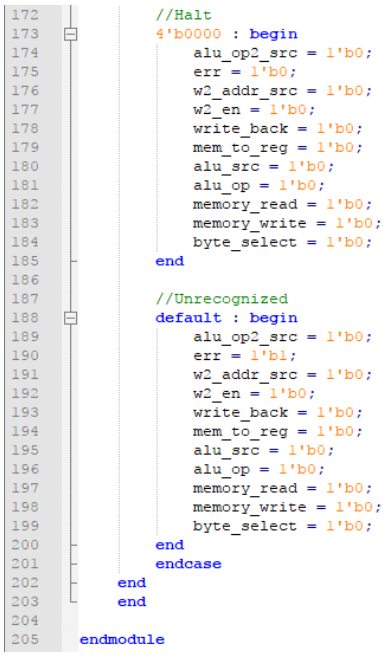


```verilog
        //Store byte
        4'b1011: begin
            alu_op2_src = 1'b0;
            err = 1'b0;
            w2_addr_src = 1'b0;
            w2_en = 1'b0;
            write_back = 1'b1;
            mem_to_reg = 1'b0;
            alu_src = 1'b1;
            alu_op = 1'b1;
            memory_read = 1'b0;
            memory_write = 1'b1;
            byte_select = 1'b1;
        end
        //Load
        4'b1100 : begin
            alu_op2_src = 1'b0;
            err = 1'b0;
            w2_addr_src = 1'b0;
            w2_en = 1'b0;
            write_back = 1'b1;
            mem_to_reg = 1'b1;
            alu_src = 1'b1;
            alu_op = 1'b1;
            memory_read = 1'b1;
            memory_write = 1'b0;
            byte_select = 1'b0;
        end
        //Store
        4'b1101 : begin
            alu_op2_src = 1'b0;
            err = 1'b0;
            w2_addr_src = 1'b0;
            w2_en = 1'b0;
            write_back = 1'b1;
            mem_to_reg = 1'b0;
            alu_src = 1'b1;
            alu_op = 1'b1;
            memory_read = 1'b0;
            memory_write = 1'b1;
            byte_select = 1'b0;
        end
```



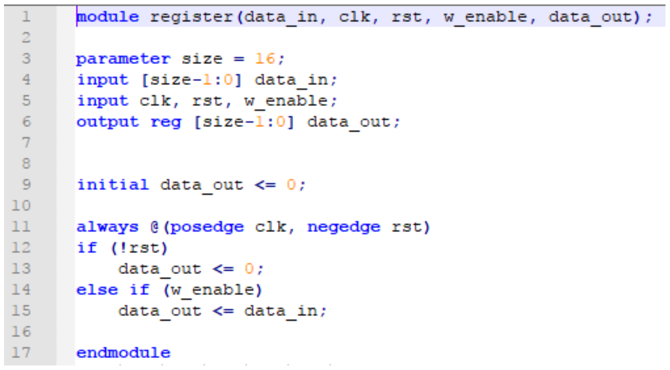
```verilog
        //Branch on less
        //Branch on greater
        4'b010? : begin
            alu_op2_src = 1'b1;
            err = 1'b0;
            w2_addr_src = 1'b0;
            w2_en = 1'b0;
            write_back = 1'b0;
            mem_to_reg = 1'b0;
            alu_src = 1'b0;
            alu_op = 1'b1;
            memory_read = 1'b0;
            memory_write = 1'b0;
            byte_select = 1'b0;
        end
        //Branch on Equal
        4'b0110 : begin
            alu_op2_src = 1'b1;
            err = 1'b0;
            w2_addr_src = 1'b0;
            w2_en = 1'b0;
            write_back = 1'b0;
            mem_to_reg = 1'b0;
            alu_src = 1'b0;
            alu_op = 1'b1;
            memory_read = 1'b0;
            memory_write = 1'b0;
            byte_select = 1'b0;
        end
        //Jump (jump signal handled via jump control)
        //Jump actually has its own adder and shift for calculating address
        4'b0001 : begin
            alu_op2_src = 1'b0;
            err = 1'b0;
            w2_addr_src = 1'b0;
            w2_en = 1'b0;
            write_back = 1'b0;
            mem_to_reg = 1'b0;
            alu_src = 1'b0;
            alu_op = 1'b0;
            memory_read = 1'b0;
            memory_write = 1'b0;
            byte_select = 1'b0;
        end
```

Control Unit cont:



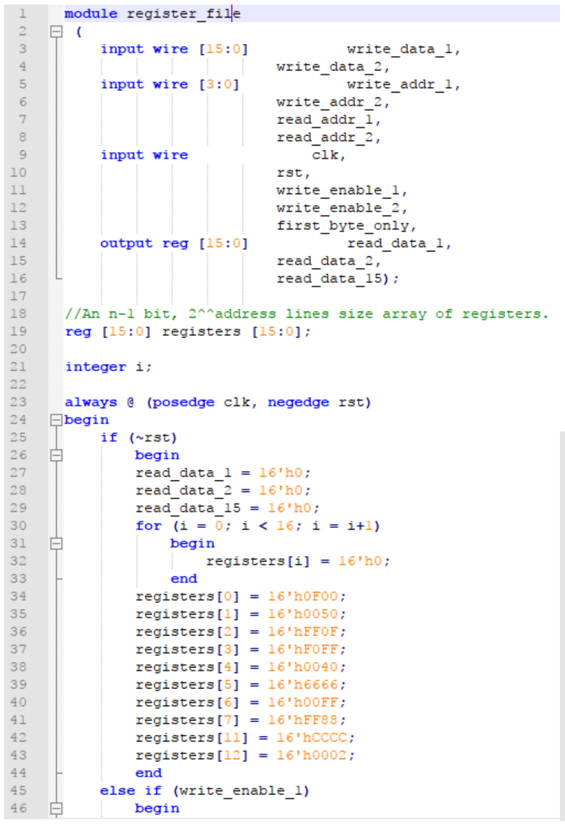
```
        //Halt
        4'b0000 : begin
            alu_op2_src = 1'b0;
            err = 1'b0;
            w2_addr_src = 1'b0;
            w2_en = 1'b0;
            write_back = 1'b0;
            mem_to_reg = 1'b0;
            alu_src = 1'b0;
            alu_op = 1'b0;
            memory_read = 1'b0;
            memory_write = 1'b0;
            byte_select = 1'b0;
        end

        //Unrecognized
        default : begin
            alu_op2_src = 1'b0;
            err = 1'b1;
            w2_addr_src = 1'b0;
            w2_en = 1'b0;
            write_back = 1'b0;
            mem_to_reg = 1'b0;
            alu_src = 1'b0;
            alu_op = 1'b0;
            memory_read = 1'b0;
            memory_write = 1'b0;
            byte_select = 1'b0;
        end
        endcase
    end
    end

endmodule
```

Register File:

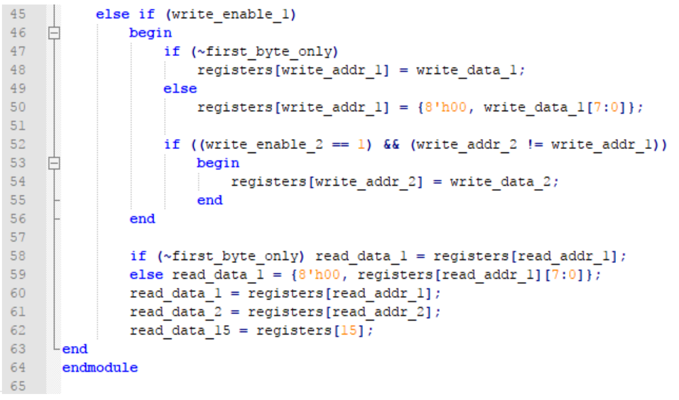


```
module register(data_in, clk, rst, w_enable, data_out);

parameter size = 16;
input [size-1:0] data_in;
input clk, rst, w_enable;
output reg [size-1:0] data_out;


initial data_out <= 0;

always @(posedge clk, negedge rst)
if (!rst)
    data_out <= 0;
else if (w_enable)
    data_out <= data_in;

endmodule
```

Register File cont:

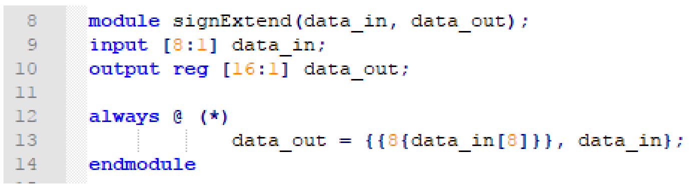


```verilog
module register_file
 (
    input wire [15:0]           write_data_1,
                        write_data_2,
    input wire [3:0]            write_addr_1,
                        write_addr_2,
                        read_addr_1,
                        read_addr_2,
    input wire                      clk,
                        rst,
                        write_enable_1,
                        write_enable_2,
                        first_byte_only,
    output reg [15:0]           read_data_1,
                        read_data_2,
                        read_data_15);

//An n-1 bit, 2^^address lines size array of registers.
reg [15:0] registers [15:0];

integer i;

always @ (posedge clk, negedge rst)
begin
    if (~rst)
        begin
        read_data_1 = 16'h0;
        read_data_2 = 16'h0;
        read_data_15 = 16'h0;
        for (i = 0; i < 16; i = i+1)
            begin
                registers[i] = 16'h0;
            end
        registers[0] = 16'h0F00;
        registers[1] = 16'h0050;
        registers[2] = 16'hFF0F;
        registers[3] = 16'hF0FF;
        registers[4] = 16'h0040;
        registers[5] = 16'h6666;
        registers[6] = 16'h00FF;
        registers[7] = 16'hFF88;
        registers[11] = 16'hCCCC;
        registers[12] = 16'h0002;
        end
    else if (write_enable_1)
        begin
```



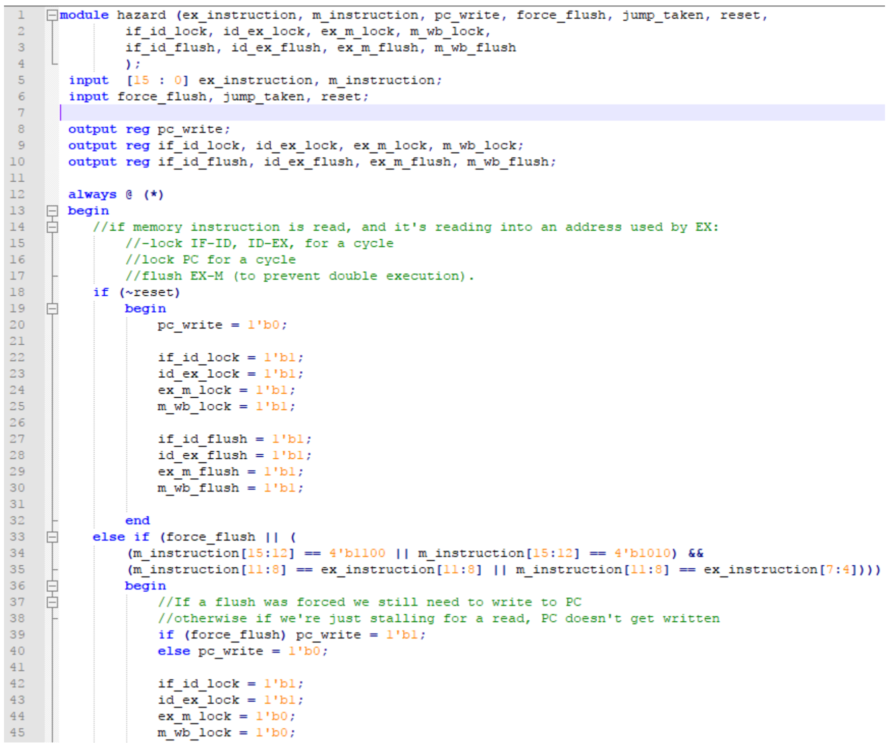
```verilog
    else if (write_enable_1)
        begin
            if (~first_byte_only)
                registers[write_addr_1] = write_data_1;
            else
                registers[write_addr_1] = {8'h00, write_data_1[7:0]};

            if ((write_enable_2 == 1) && (write_addr_2 != write_addr_1))
                begin
                    registers[write_addr_2] = write_data_2;
                end
        end

        if (~first_byte_only) read_data_1 = registers[read_addr_1];
        else read_data_1 = {8'h00, registers[read_addr_1][7:0]};
        read_data_1 = registers[read_addr_1];
        read_data_2 = registers[read_addr_2];
        read_data_15 = registers[15];
end
endmodule
```

Sign Extension:

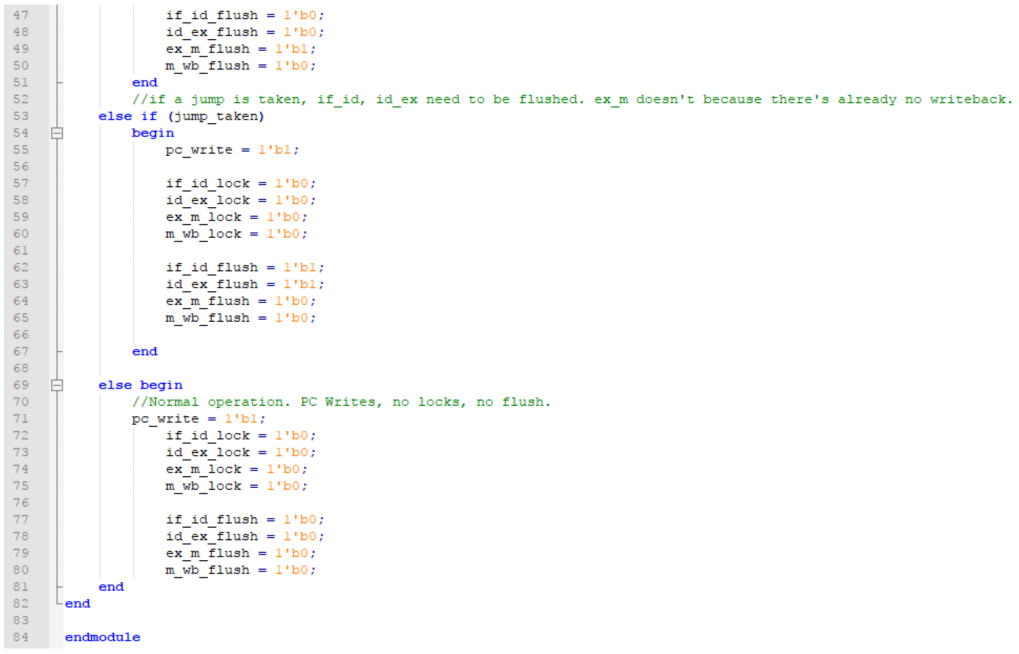


```verilog
module signExtend(data_in, data_out);
input [8:1] data_in;
output reg [16:1] data_out;

always @ (*)
            data_out = {{8{data_in[8]}}, data_in};
endmodule
```

Hazard Detection Unit:



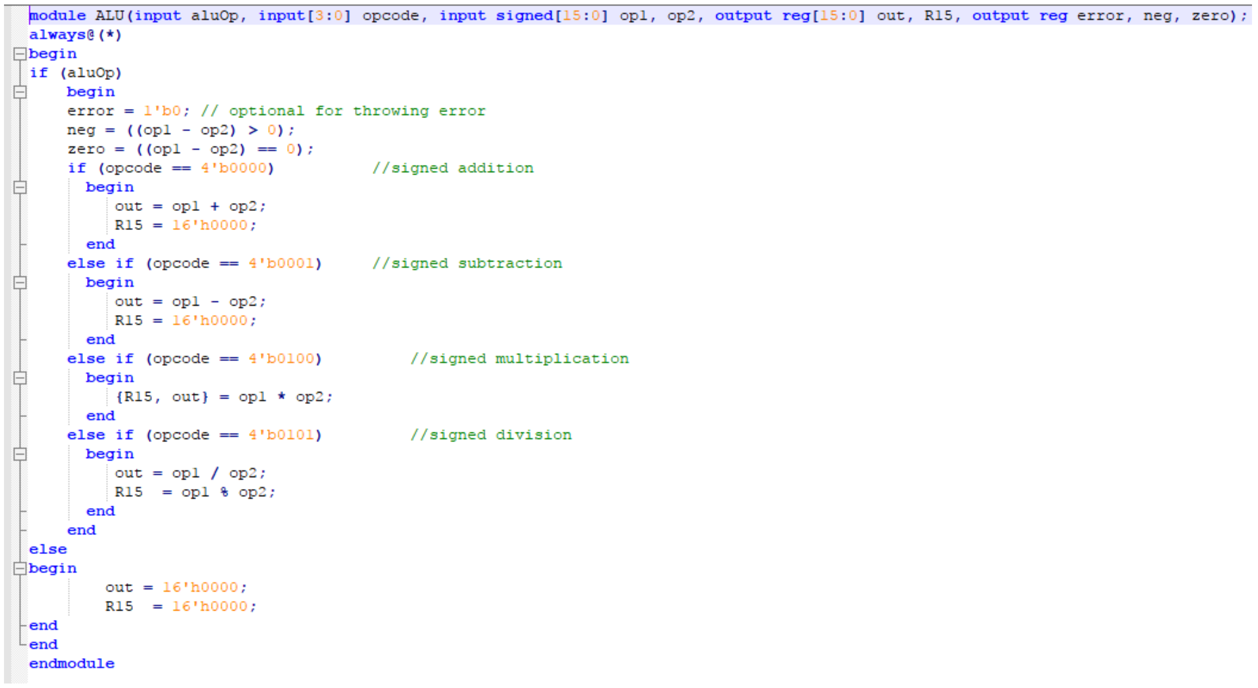
```
module hazard (ex_instruction, m_instruction, pc_write, force_flush, jump_taken, reset,
        if_id_lock, id_ex_lock, ex_m_lock, m_wb_lock,
        if_id_flush, id_ex_flush, ex_m_flush, m_wb_flush
        );
 input  [15 : 0] ex_instruction, m_instruction;
 input force_flush, jump_taken, reset;

 output reg pc_write;
 output reg if_id_lock, id_ex_lock, ex_m_lock, m_wb_lock;
 output reg if_id_flush, id_ex_flush, ex_m_flush, m_wb_flush;

 always @ (*)
 begin
    //if memory instruction is read, and it's reading into an address used by EX:
        //-lock IF-ID, ID-EX, for a cycle
        //lock PC for a cycle
        //flush EX-M (to prevent double execution).
    if (~reset)
        begin
            pc_write = 1'b0;

            if_id_lock = 1'b1;
            id_ex_lock = 1'b1;
            ex_m_lock = 1'b1;
            m_wb_lock = 1'b1;

            if_id_flush = 1'b1;
            id_ex_flush = 1'b1;
            ex_m_flush = 1'b1;
            m_wb_flush = 1'b1;

        end
    else if (force_flush || (
        (m_instruction[15:12] == 4'b1100 || m_instruction[15:12] == 4'b1010) &&
        (m_instruction[11:8] == ex_instruction[11:8] || m_instruction[11:8] == ex_instruction[7:4])))
        begin
            //If a flush was forced we still need to write to PC
            //otherwise if we're just stalling for a read, PC doesn't get written
            if (force_flush) pc_write = 1'b1;
            else pc_write = 1'b0;

            if_id_lock = 1'b1;
            id_ex_lock = 1'b1;
            ex_m_lock = 1'b0;
            m_wb_lock = 1'b0;
```



```
            if_id_flush = 1'b0;
            id_ex_flush = 1'b0;
            ex_m_flush = 1'b1;
            m_wb_flush = 1'b0;
        end
        //if a jump is taken, if_id, id_ex need to be flushed. ex_m doesn't because there's already no writeback.
    else if (jump_taken)
        begin
            pc_write = 1'b1;

            if_id_lock = 1'b0;
            id_ex_lock = 1'b0;
            ex_m_lock = 1'b0;
            m_wb_lock = 1'b0;

            if_id_flush = 1'b1;
            id_ex_flush = 1'b1;
            ex_m_flush = 1'b0;
            m_wb_flush = 1'b0;

        end

    else begin
        //Normal operation. PC Writes, no locks, no flush.
        pc_write = 1'b1;
            if_id_lock = 1'b0;
            id_ex_lock = 1'b0;
            ex_m_lock = 1'b0;
            m_wb_lock = 1'b0;

            if_id_flush = 1'b0;
            id_ex_flush = 1'b0;
            ex_m_flush = 1'b0;
            m_wb_flush = 1'b0;
    end
end

endmodule
```

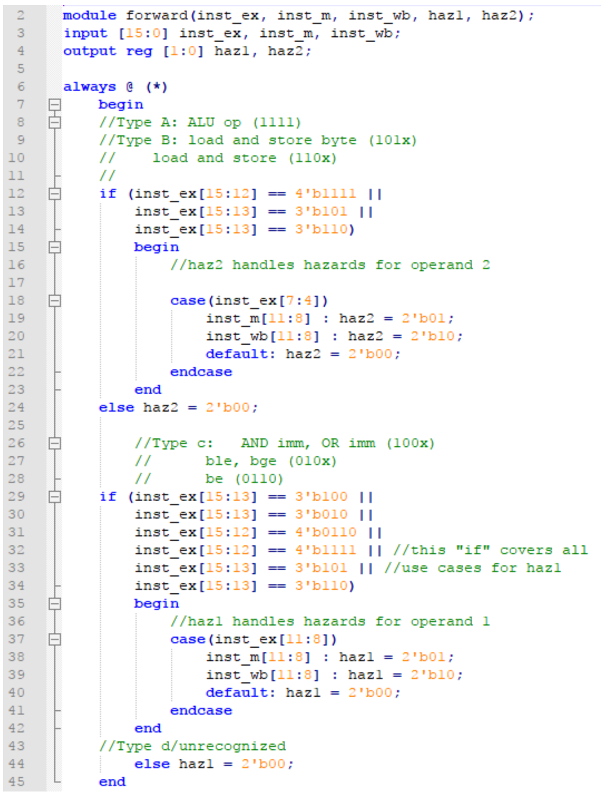
Stage Three: Execution

Arithmetic Logical Unit (ALU):



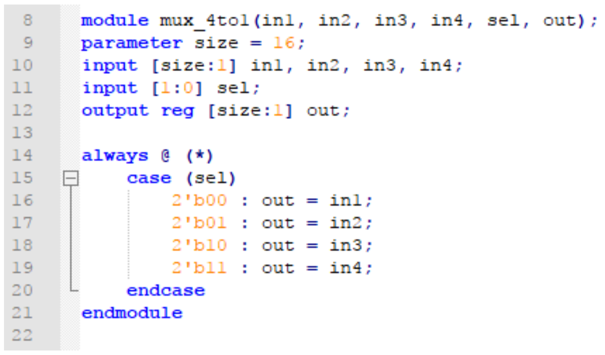
```
module ALU(input aluOp, input[3:0] opcode, input signed[15:0] op1, op2, output reg[15:0] out, R15, output reg error, neg, zero);
always@(*)
begin
if (aluOp)
    begin
    error = 1'b0; // optional for throwing error
    neg = ((op1 - op2) > 0);
    zero = ((op1 - op2) == 0);
    if (opcode == 4'b0000)          //signed addition
      begin
        out = op1 + op2;
        R15 = 16'h0000;
      end
    else if (opcode == 4'b0001)     //signed subtraction
      begin
        out = op1 - op2;
        R15 = 16'h0000;
      end
    else if (opcode == 4'b0100)         //signed multiplication
      begin
        {R15, out} = op1 * op2;
      end
    else if (opcode == 4'b0101)         //signed division
      begin
        out = op1 / op2;
        R15  = op1 % op2;
      end
    end
else
begin
        out = 16'h0000;
        R15  = 16'h0000;
end
end
endmodule
```

Forwarding Unit:



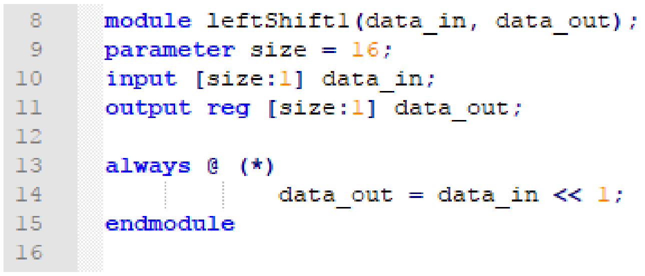
```
module forward(inst_ex, inst_m, inst_wb, haz1, haz2);
input [15:0] inst_ex, inst_m, inst_wb;
output reg [1:0] haz1, haz2;

always @ (*)
    begin
    //Type A: ALU op (1111)
    //Type B: load and store byte (101x)
    //    load and store (110x)
    //
    if (inst_ex[15:12] == 4'b1111 ||
        inst_ex[15:13] == 3'b101 ||
        inst_ex[15:13] == 3'b110)
        begin
            //haz2 handles hazards for operand 2

            case(inst_ex[7:4])
                inst_m[11:8] : haz2 = 2'b01;
                inst_wb[11:8] : haz2 = 2'b10;
                default: haz2 = 2'b00;
            endcase
        end
    else haz2 = 2'b00;

        //Type c:   AND imm, OR imm (100x)
        //      ble, bge (010x)
        //      be (0110)
    if (inst_ex[15:13] == 3'b100 ||
        inst_ex[15:13] == 3'b010 ||
        inst_ex[15:12] == 4'b0110 ||
        inst_ex[15:12] == 4'b1111 || //this "if" covers all
        inst_ex[15:13] == 3'b101 || //use cases for haz1
        inst_ex[15:13] == 3'b110)
        begin
            //haz1 handles hazards for operand 1
            case(inst_ex[11:8])
                inst_m[11:8] : haz1 = 2'b01;
                inst_wb[11:8] : haz1 = 2'b10;
                default: haz1 = 2'b00;
            endcase
        end
    //Type d/unrecognized
        else haz1 = 2'b00;
    end
```

Multiplexor 4 to 1:



```
module mux_4to1(in1, in2, in3, in4, sel, out);
parameter size = 16;
input [size:1] in1, in2, in3, in4;
input [1:0] sel;
output reg [size:1] out;

always @ (*)
    case (sel)
        2'b00 : out = in1;
        2'b01 : out = in2;
        2'b10 : out = in3;
        2'b11 : out = in4;
    endcase
endmodule
```

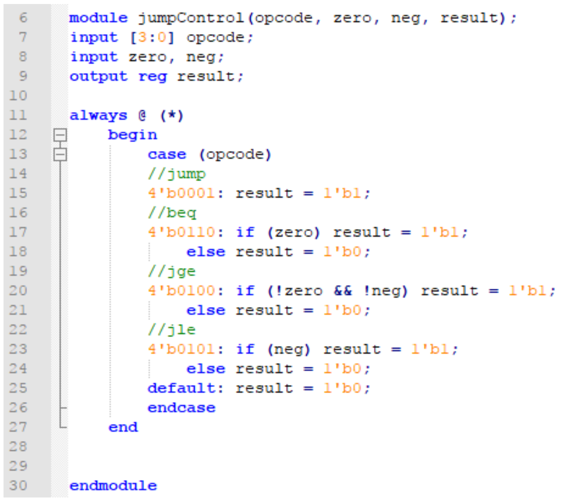
Left Shift:



```
module leftShift1(data_in, data_out);
parameter size = 16;
input [size:1] data_in;
output reg [size:1] data_out;

always @ (*)
            data_out = data_in << 1;
endmodule
```

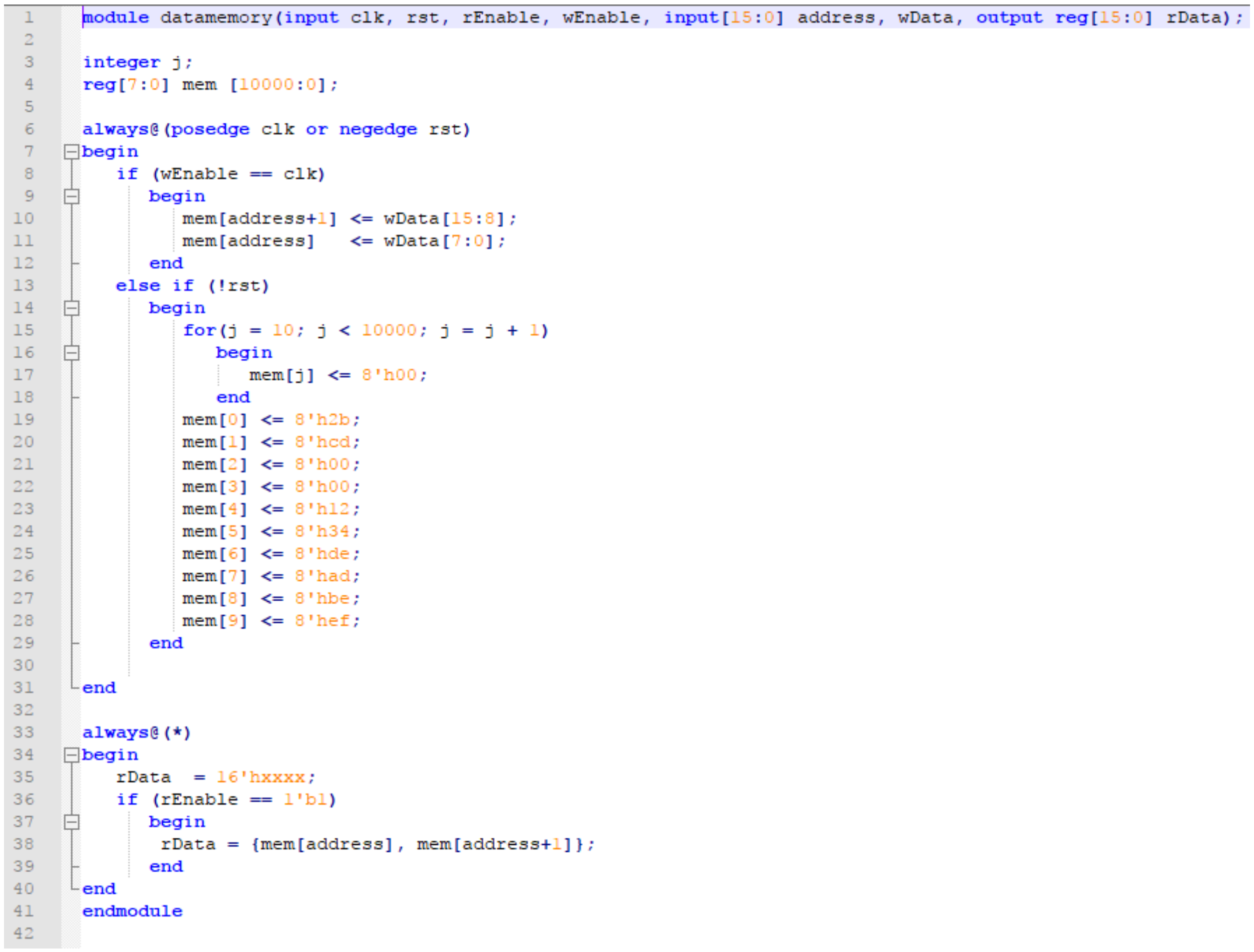
Stage Four: Memory

Jump Control:



```
module jumpControl(opcode, zero, neg, result);
input [3:0] opcode;
input zero, neg;
output reg result;

always @ (*)
    begin
        case (opcode)
        //jump
        4'b0001: result = 1'b1;
        //beq
        4'b0110: if (zero) result = 1'b1;
            else result = 1'b0;
        //jge
        4'b0100: if (!zero && !neg) result = 1'b1;
            else result = 1'b0;
        //jle
        4'b0101: if (neg) result = 1'b1;
            else result = 1'b0;
        default: result = 1'b0;
        endcase
    end


endmodule
```

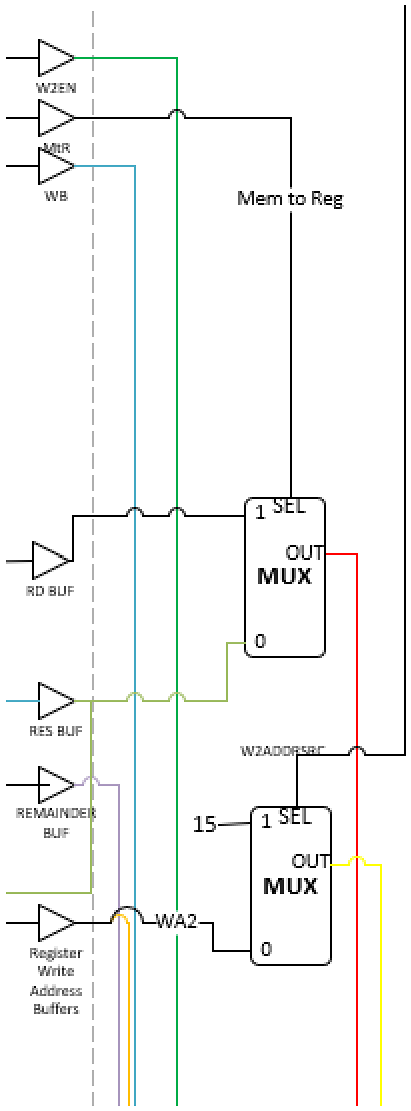
Data Memory:



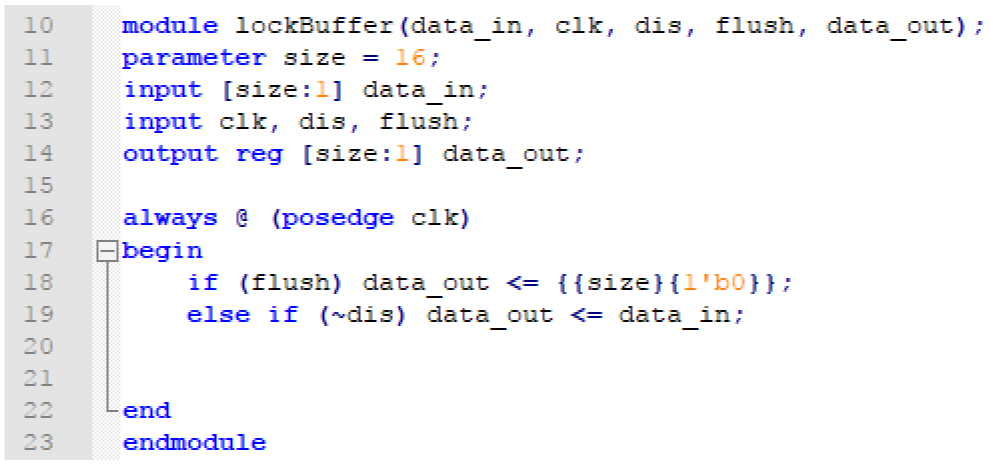
```
module datamemory(input clk, rst, rEnable, wEnable, input[15:0] address, wData, output reg[15:0] rData);

integer j;
reg[7:0] mem [10000:0];

always@(posedge clk or negedge rst)
begin
    if (wEnable == clk)
        begin
            mem[address+1]  <= wData[15:8];
            mem[address]    <= wData[7:0];
        end
    else if (!rst)
        begin
            for(j = 10; j < 10000; j = j + 1)
                begin
                    mem[j] <= 8'h00;
                end
            mem[0] <= 8'h2b;
            mem[1] <= 8'hcd;
            mem[2] <= 8'h00;
            mem[3] <= 8'h00;
            mem[4] <= 8'h12;
            mem[5] <= 8'h34;
            mem[6] <= 8'hde;
            mem[7] <= 8'had;
            mem[8] <= 8'hbe;
            mem[9] <= 8'hef;
        end

end

always@(*)
begin
    rData  = 16'hxxxx;
    if (rEnable == 1'b1)
        begin
         rData = {mem[address], mem[address+1]};
        end
end
endmodule
```

Stage Five: Write Back

The last section is the Write Back stage. This section is composed of two (2 to 1) Multiplexors that allow memory to write to a register, as well as writing to an address source.



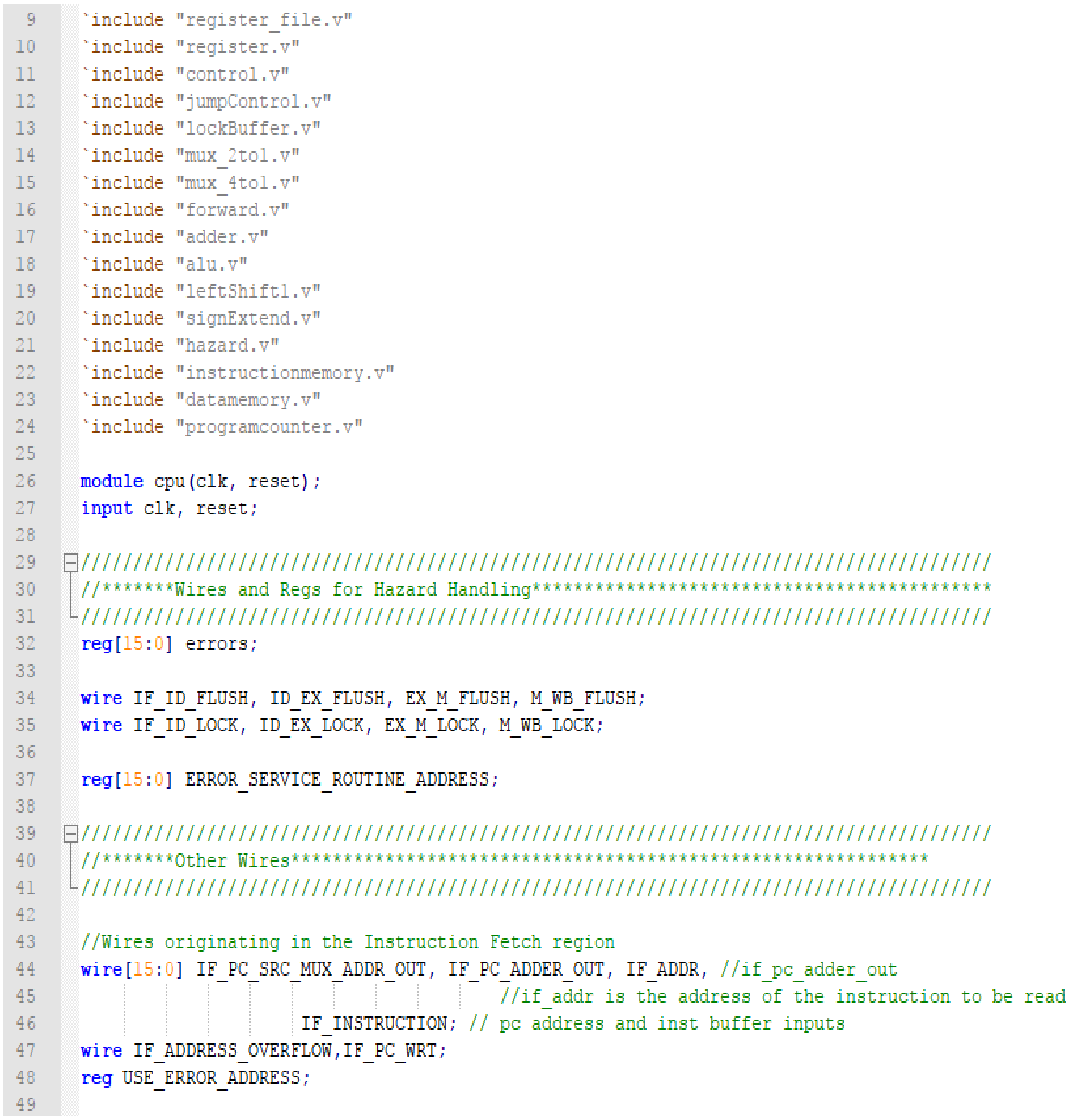
Lock Buffer Module



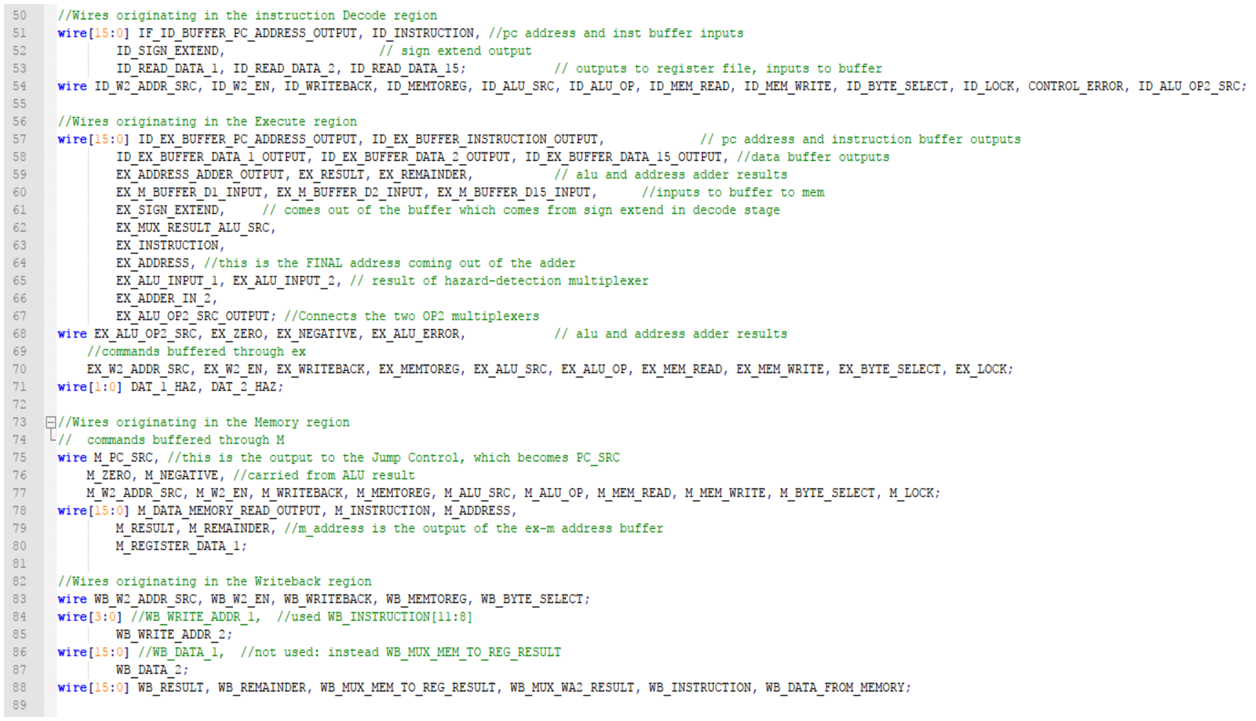
```
module lockBuffer(data_in, clk, dis, flush, data_out);
parameter size = 16;
input [size:1] data_in;
input clk, dis, flush;
output reg [size:1] data_out;

always @ (posedge clk)
begin
    if (flush) data_out <= {{size}{1'b0}};
    else if (~dis) data_out <= data_in;


end
endmodule
```

This module is a simple lock buffer used in the cpu top level design. The buffer designs are implemented in the cpu Verilog code

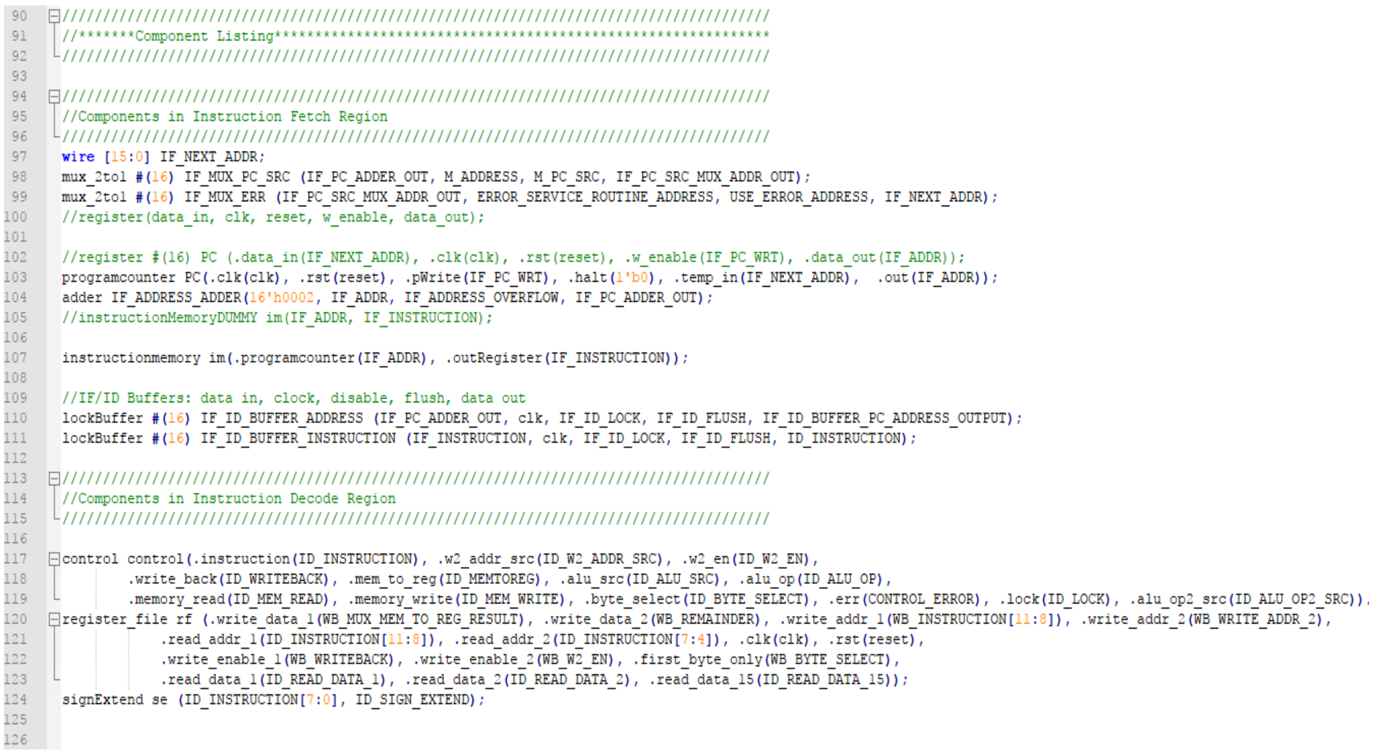
CPU / Top Design



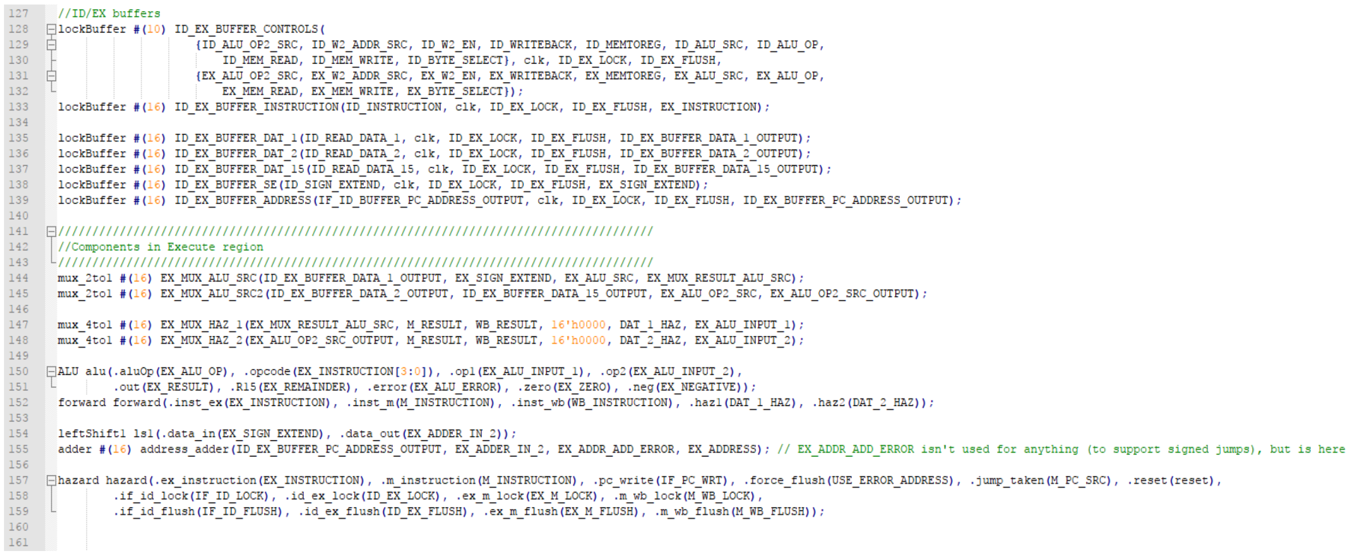
```verilog
`include "register_file.v"
`include "register.v"
`include "control.v"
`include "jumpControl.v"
`include "lockBuffer.v"
`include "mux_2to1.v"
`include "mux_4to1.v"
`include "forward.v"
`include "adder.v"
`include "alu.v"
`include "leftShift1.v"
`include "signExtend.v"
`include "hazard.v"
`include "instructionmemory.v"
`include "datamemory.v"
`include "programcounter.v"

module cpu(clk, reset);
input clk, reset;

/////////////////////////////////////////////////////////////////////////////////////
//*******Wires and Regs for Hazard Handling*******************************************
/////////////////////////////////////////////////////////////////////////////////////
reg[15:0] errors;

wire IF_ID_FLUSH, ID_EX_FLUSH, EX_M_FLUSH, M_WB_FLUSH;
wire IF_ID_LOCK, ID_EX_LOCK, EX_M_LOCK, M_WB_LOCK;

reg[15:0] ERROR_SERVICE_ROUTINE_ADDRESS;

/////////////////////////////////////////////////////////////////////////////////////
//*******Other Wires*****************************************************************
/////////////////////////////////////////////////////////////////////////////////////

//Wires originating in the Instruction Fetch region
wire[15:0] IF_PC_SRC_MUX_ADDR_OUT, IF_PC_ADDER_OUT, IF_ADDR, //if_pc_adder_out
                                        //if_addr is the address of the instruction to be read
                     IF_INSTRUCTION; // pc address and inst buffer inputs
wire IF_ADDRESS_OVERFLOW,IF_PC_WRT;
reg USE_ERROR_ADDRESS;

```



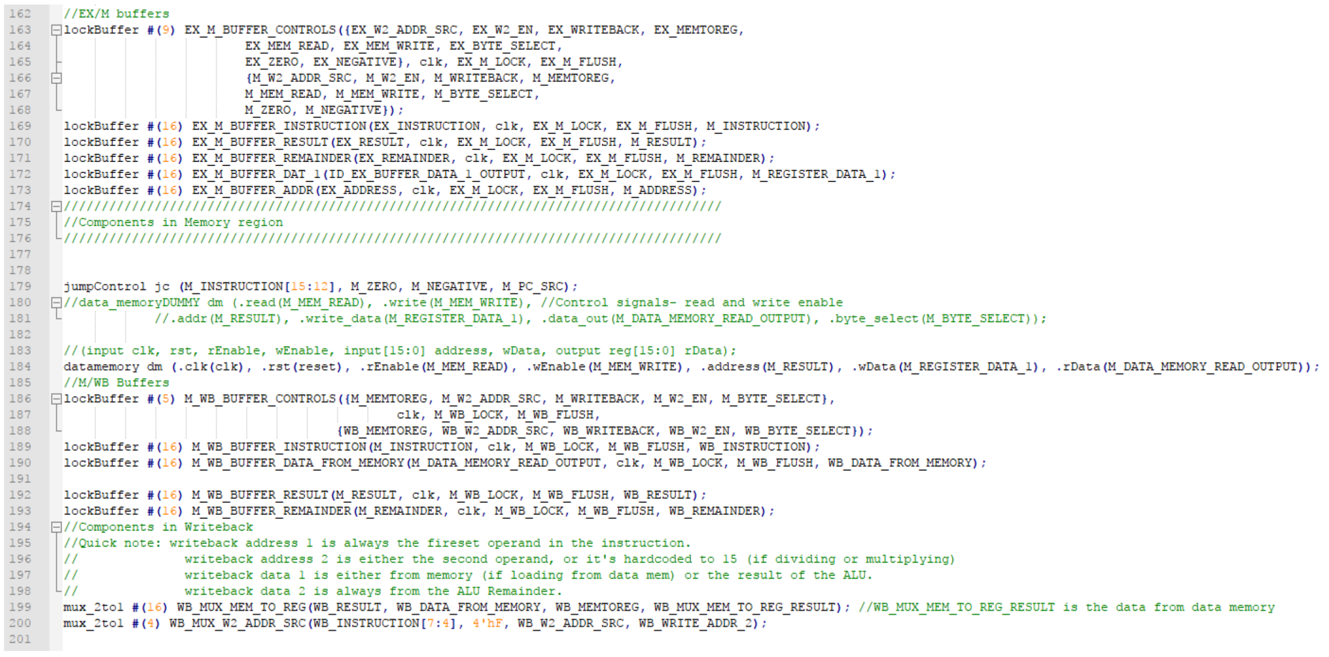
```
//Wires originating in the instruction Decode region
wire[15:0] IF_ID_BUFFER_PC_ADDRESS_OUTPUT, ID_INSTRUCTION, //pc address and inst buffer inputs
        ID_SIGN_EXTEND,                     // sign extend output
        ID_READ_DATA_1, ID_READ_DATA_2, ID_READ_DATA_15;            // outputs to register file, inputs to buffer
wire ID_W2_ADDR_SRC, ID_W2_EN, ID_WRITEBACK, ID_MEMTOREG, ID_ALU_SRC, ID_ALU_OP, ID_MEM_READ, ID_MEM_WRITE, ID_BYTE_SELECT, ID_LOCK, CONTROL_ERROR, ID_ALU_OP2_SRC;

//Wires originating in the Execute region
wire[15:0] ID_EX_BUFFER_PC_ADDRESS_OUTPUT, ID_EX_BUFFER_INSTRUCTION_OUTPUT,             // pc address and instruction buffer outputs
        ID_EX_BUFFER_DATA_1_OUTPUT, ID_EX_BUFFER_DATA_2_OUTPUT, ID_EX_BUFFER_DATA_15_OUTPUT, //data buffer outputs
        EX_ADDRESS_ADDER_OUTPUT, EX_RESULT, EX_REMAINDER,           // alu and address adder results
        EX_M_BUFFER_D1_INPUT, EX_M_BUFFER_D2_INPUT, EX_M_BUFFER_D15_INPUT,      //inputs to buffer to mem
        EX_SIGN_EXTEND,     // comes out of the buffer which comes from sign extend in decode stage
        EX_MUX_RESULT_ALU_SRC,
        EX_INSTRUCTION,
        EX_ADDRESS, //this is the FINAL address coming out of the adder
        EX_ALU_INPUT_1, EX_ALU_INPUT_2, // result of hazard-detection multiplexer
        EX_ADDER_IN_2,
        EX_ALU_OP2_SRC_OUTPUT; //Connects the two OP2 multiplexers
wire EX_ALU_OP2_SRC, EX_ZERO, EX_NEGATIVE, EX_ALU_ERROR,            // alu and address adder results
    //commands buffered through ex
    EX_W2_ADDR_SRC, EX_W2_EN, EX_WRITEBACK, EX_MEMTOREG, EX_ALU_SRC, EX_ALU_OP, EX_MEM_READ, EX_MEM_WRITE, EX_BYTE_SELECT, EX_LOCK;
wire[1:0] DAT_1_HAZ, DAT_2_HAZ;

//Wires originating in the Memory region
//  commands buffered through M
wire M_PC_SRC, //this is the output to the Jump Control, which becomes PC_SRC
    M_ZERO, M_NEGATIVE, //carried from ALU result
    M_W2_ADDR_SRC, M_W2_EN, M_WRITEBACK, M_MEMTOREG, M_ALU_SRC, M_ALU_OP, M_MEM_READ, M_MEM_WRITE, M_BYTE_SELECT, M_LOCK;
wire[15:0] M_DATA_MEMORY_READ_OUTPUT, M_INSTRUCTION, M_ADDRESS,
        M_RESULT, M_REMAINDER, //m_address is the output of the ex-m address buffer
        M_REGISTER_DATA_1;

//Wires originating in the Writeback region
wire WB_W2_ADDR_SRC, WB_W2_EN, WB_WRITEBACK, WB_MEMTOREG, WB_BYTE_SELECT;
wire[3:0] //WB_WRITE_ADDR_1,  //used WB_INSTRUCTION[11:8]
        WB_WRITE_ADDR_2;
wire[15:0] //WB_DATA_1,  //not used: instead WB_MUX_MEM_TO_REG_RESULT
        WB_DATA_2;
wire[15:0] WB_RESULT, WB_REMAINDER, WB_MUX_MEM_TO_REG_RESULT, WB_MUX_WA2_RESULT, WB_INSTRUCTION, WB_DATA_FROM_MEMORY;

```



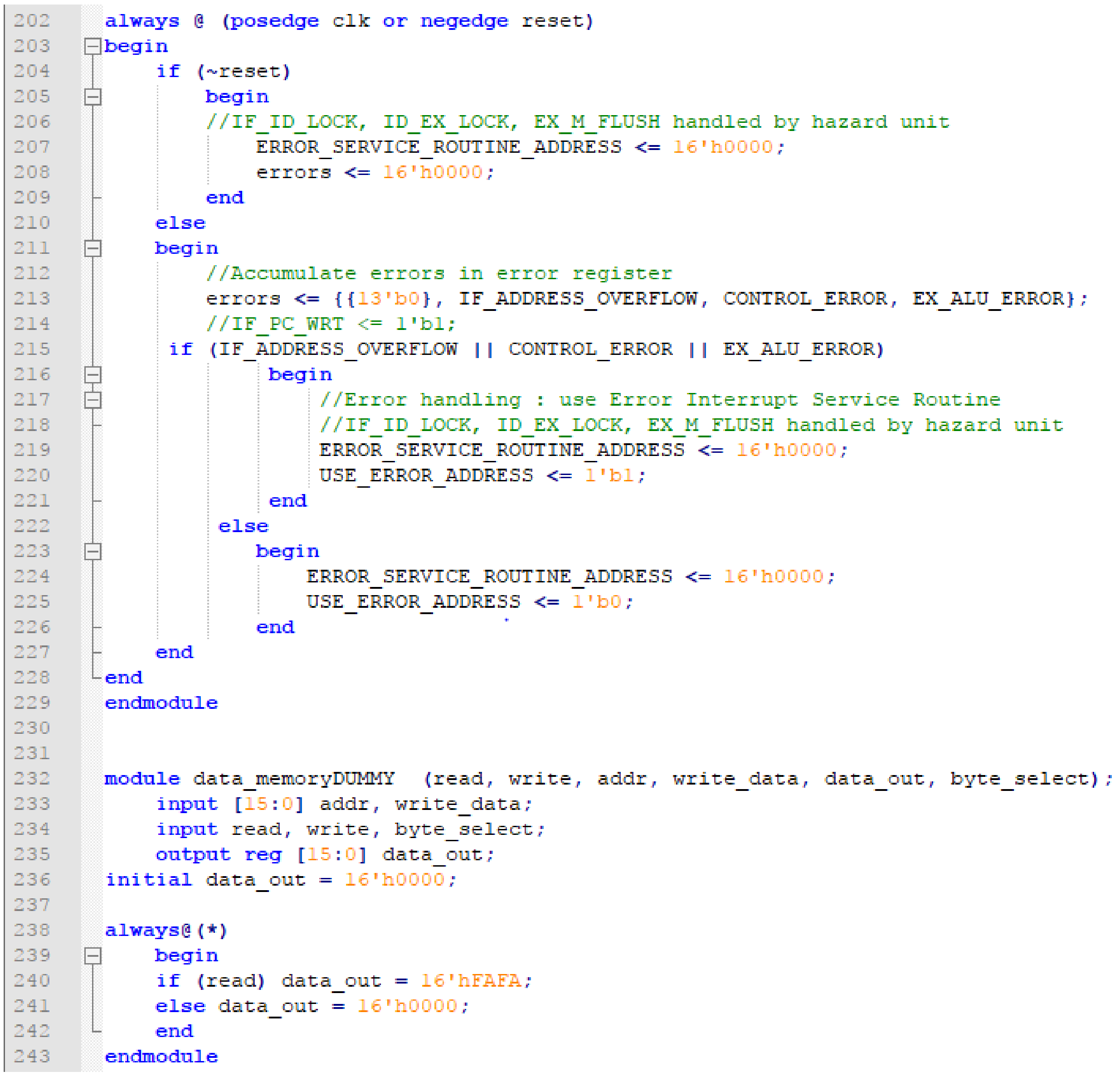
```
///////////////////////////////////////////////////////////////////////////////////////
//*******Component Listing*************************************************************
///////////////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////////////
//Components in Instruction Fetch Region
///////////////////////////////////////////////////////////////////////////////////////
wire [15:0] IF_NEXT_ADDR;
mux_2tol #(16) IF_MUX_PC_SRC (IF_PC_ADDER_OUT, M_ADDRESS, M_PC_SRC, IF_PC_SRC_MUX_ADDR_OUT);
mux_2tol #(16) IF_MUX_ERR (IF_PC_SRC_MUX_ADDR_OUT, ERROR_SERVICE_ROUTINE_ADDRESS, USE_ERROR_ADDRESS, IF_NEXT_ADDR);
//register(data_in, clk, reset, w_enable, data_out);

//register #(16) PC (.data_in(IF_NEXT_ADDR), .clk(clk), .rst(reset), .w_enable(IF_PC_WRT), .data_out(IF_ADDR));
programcounter PC(.clk(clk), .rst(reset), .pWrite(IF_PC_WRT), .halt(1'b0), .temp_in(IF_NEXT_ADDR),  .out(IF_ADDR));
adder IF_ADDRESS_ADDER(16'h0002, IF_ADDR, IF_ADDRESS_OVERFLOW, IF_PC_ADDER_OUT);
//instructionMemoryDUMMY im(IF_ADDR, IF_INSTRUCTION);

instructionmemory im(.programcounter(IF_ADDR), .outRegister(IF_INSTRUCTION));

//IF/ID Buffers: data in, clock, disable, flush, data out
lockBuffer #(16) IF_ID_BUFFER_ADDRESS (IF_PC_ADDER_OUT, clk, IF_ID_LOCK, IF_ID_FLUSH, IF_ID_BUFFER_PC_ADDRESS_OUTPUT);
lockBuffer #(16) IF_ID_BUFFER_INSTRUCTION (IF_INSTRUCTION, clk, IF_ID_LOCK, IF_ID_FLUSH, ID_INSTRUCTION);

///////////////////////////////////////////////////////////////////////////////////////
//Components in Instruction Decode Region
///////////////////////////////////////////////////////////////////////////////////////

control control(.instruction(ID_INSTRUCTION), .w2_addr_src(ID_W2_ADDR_SRC), .w2_en(ID_W2_EN),
        .write_back(ID_WRITEBACK), .mem_to_reg(ID_MEMTOREG), .alu_src(ID_ALU_SRC), .alu_op(ID_ALU_OP),
        .memory_read(ID_MEM_READ), .memory_write(ID_MEM_WRITE), .byte_select(ID_BYTE_SELECT), .err(CONTROL_ERROR), .lock(ID_LOCK), .alu_op2_src(ID_ALU_OP2_SRC));
register_file rf (.write_data_1(WB_MUX_MEM_TO_REG_RESULT), .write_data_2(WB_REMAINDER), .write_addr_1(WB_INSTRUCTION[11:8]), .write_addr_2(WB_WRITE_ADDR_2),
            .read_addr_1(ID_INSTRUCTION[11:8]), .read_addr_2(ID_INSTRUCTION[7:4]), .clk(clk), .rst(reset),
            .write_enable_1(WB_WRITEBACK), .write_enable_2(WB_W2_EN), .first_byte_only(WB_BYTE_SELECT),
            .read_data_1(ID_READ_DATA_1), .read_data_2(ID_READ_DATA_2), .read_data_15(ID_READ_DATA_15));
signExtend se (ID_INSTRUCTION[7:0], ID_SIGN_EXTEND);


```



```
//ID/EX buffers
lockBuffer #(10) ID_EX_BUFFER_CONTROLS(
                    {ID_ALU_OP2_SRC, ID_W2_ADDR_SRC, ID_W2_EN, ID_WRITEBACK, ID_MEMTOREG, ID_ALU_SRC, ID_ALU_OP,
                        ID_MEM_READ, ID_MEM_WRITE, ID_BYTE_SELECT}, clk, ID_EX_LOCK, ID_EX_FLUSH,
                    {EX_ALU_OP2_SRC, EX_W2_ADDR_SRC, EX_W2_EN, EX_WRITEBACK, EX_MEMTOREG, EX_ALU_SRC, EX_ALU_OP,
                        EX_MEM_READ, EX_MEM_WRITE, EX_BYTE_SELECT});
lockBuffer #(16) ID_EX_BUFFER_INSTRUCTION(ID_INSTRUCTION, clk, ID_EX_LOCK, ID_EX_FLUSH, EX_INSTRUCTION);

lockBuffer #(16) ID_EX_BUFFER_DAT_1(ID_READ_DATA_1, clk, ID_EX_LOCK, ID_EX_FLUSH, ID_EX_BUFFER_DATA_1_OUTPUT);
lockBuffer #(16) ID_EX_BUFFER_DAT_2(ID_READ_DATA_2, clk, ID_EX_LOCK, ID_EX_FLUSH, ID_EX_BUFFER_DATA_2_OUTPUT);
lockBuffer #(16) ID_EX_BUFFER_DAT_15(ID_READ_DATA_15, clk, ID_EX_LOCK, ID_EX_FLUSH, ID_EX_BUFFER_DATA_15_OUTPUT);
lockBuffer #(16) ID_EX_BUFFER_SE(ID_SIGN_EXTEND, clk, ID_EX_LOCK, ID_EX_FLUSH, EX_SIGN_EXTEND);
lockBuffer #(16) ID_EX_BUFFER_ADDRESS(IF_ID_BUFFER_PC_ADDRESS_OUTPUT, clk, ID_EX_LOCK, ID_EX_FLUSH, ID_EX_BUFFER_PC_ADDRESS_OUTPUT);

/////////////////////////////////////////////////////////////////////////////////////////
//Components in Execute region
/////////////////////////////////////////////////////////////////////////////////////////
mux_2tol #(16) EX_MUX_ALU_SRC(ID_EX_BUFFER_DATA_1_OUTPUT, EX_SIGN_EXTEND, EX_ALU_SRC, EX_MUX_RESULT_ALU_SRC);
mux_2tol #(16) EX_MUX_ALU_SRC2(ID_EX_BUFFER_DATA_2_OUTPUT, ID_EX_BUFFER_DATA_15_OUTPUT, EX_ALU_OP2_SRC, EX_ALU_OP2_SRC_OUTPUT);

mux_4tol #(16) EX_MUX_HAZ_1(EX_MUX_RESULT_ALU_SRC, M_RESULT, WB_RESULT, 16'h0000, DAT_1_HAZ, EX_ALU_INPUT_1);
mux_4tol #(16) EX_MUX_HAZ_2(EX_ALU_OP2_SRC_OUTPUT, M_RESULT, WB_RESULT, 16'h0000, DAT_2_HAZ, EX_ALU_INPUT_2);

ALU alu(.aluOp(EX_ALU_OP), .opcode(EX_INSTRUCTION[3:0]), .op1(EX_ALU_INPUT_1), .op2(EX_ALU_INPUT_2),
        .out(EX_RESULT), .R15(EX_REMAINDER), .error(EX_ALU_ERROR), .zero(EX_ZERO), .neg(EX_NEGATIVE));
forward forward(.inst_ex(EX_INSTRUCTION), .inst_m(M_INSTRUCTION), .inst_wb(WB_INSTRUCTION), .haz1(DAT_1_HAZ), .haz2(DAT_2_HAZ));

leftShift1 ls1(.data_in(EX_SIGN_EXTEND), .data_out(EX_ADDER_IN_2));
adder #(16) address_adder(ID_EX_BUFFER_PC_ADDRESS_OUTPUT, EX_ADDER_IN_2, EX_ADDR_ADD_ERROR, EX_ADDRESS); // EX_ADDR_ADD_ERROR isn't used for anything (to support signed jumps), but is here

hazard hazard(.ex_instruction(EX_INSTRUCTION), .m_instruction(M_INSTRUCTION), .pc_write(IF_PC_WRT), .force_flush(USE_ERROR_ADDRESS), .jump_taken(M_PC_SRC), .reset(reset),
        .if_id_lock(IF_ID_LOCK), .id_ex_lock(ID_EX_LOCK), .ex_m_lock(EX_M_LOCK), .m_wb_lock(M_WB_LOCK),
        .if_id_flush(IF_ID_FLUSH), .id_ex_flush(ID_EX_FLUSH), .ex_m_flush(EX_M_FLUSH), .m_wb_flush(M_WB_FLUSH));


```



```
//EX/M buffers
lockBuffer #(9) EX_M_BUFFER_CONTROLS({EX_W2_ADDR_SRC, EX_W2_EN, EX_WRITEBACK, EX_MEMTOREG,
                        EX_MEM_READ, EX_MEM_WRITE, EX_BYTE_SELECT,
                        EX_ZERO, EX_NEGATIVE}, clk, EX_M_LOCK, EX_M_FLUSH,
                        {M_W2_ADDR_SRC, M_W2_EN, M_WRITEBACK, M_MEMTOREG,
                        M_MEM_READ, M_MEM_WRITE, M_BYTE_SELECT,
                        M_ZERO, M_NEGATIVE});
lockBuffer #(16) EX_M_BUFFER_INSTRUCTION(EX_INSTRUCTION, clk, EX_M_LOCK, EX_M_FLUSH, M_INSTRUCTION);
lockBuffer #(16) EX_M_BUFFER_RESULT(EX_RESULT, clk, EX_M_LOCK, EX_M_FLUSH, M_RESULT);
lockBuffer #(16) EX_M_BUFFER_REMAINDER(EX_REMAINDER, clk, EX_M_LOCK, EX_M_FLUSH, M_REMAINDER);
lockBuffer #(16) EX_M_BUFFER_DAT_1(ID_EX_BUFFER_DATA_1_OUTPUT, clk, EX_M_LOCK, EX_M_FLUSH, M_REGISTER_DATA_1);
lockBuffer #(16) EX_M_BUFFER_ADDR(EX_ADDRESS, clk, EX_M_LOCK, EX_M_FLUSH, M_ADDRESS);
/////////////////////////////////////////////////////////////////////////////////////////
//Components in Memory region
/////////////////////////////////////////////////////////////////////////////////////////


jumpControl jc (M_INSTRUCTION[15:12], M_ZERO, M_NEGATIVE, M_PC_SRC);
//data_memoryDUMMY dm (.read(M_MEM_READ), .write(M_MEM_WRITE), //Control signals- read and write enable
            //.addr(M_RESULT), .write_data(M_REGISTER_DATA_1), .data_out(M_DATA_MEMORY_READ_OUTPUT), .byte_select(M_BYTE_SELECT));

//(input clk, rst, rEnable, wEnable, input[15:0] address, wData, output reg[15:0] rData);
datamemory dm (.clk(clk), .rst(reset), .rEnable(M_MEM_READ), .wEnable(M_MEM_WRITE), .address(M_RESULT), .wData(M_REGISTER_DATA_1), .rData(M_DATA_MEMORY_READ_OUTPUT));
//M/WB Buffers
lockBuffer #(5) M_WB_BUFFER_CONTROLS({M_MEMTOREG, M_W2_ADDR_SRC, M_WRITEBACK, M_W2_EN, M_BYTE_SELECT},
                                clk, M_WB_LOCK, M_WB_FLUSH,
                        {WB_MEMTOREG, WB_W2_ADDR_SRC, WB_WRITEBACK, WB_W2_EN, WB_BYTE_SELECT});
lockBuffer #(16) M_WB_BUFFER_INSTRUCTION(M_INSTRUCTION, clk, M_WB_LOCK, M_WB_FLUSH, WB_INSTRUCTION);
lockBuffer #(16) M_WB_BUFFER_DATA_FROM_MEMORY(M_DATA_MEMORY_READ_OUTPUT, clk, M_WB_LOCK, M_WB_FLUSH, WB_DATA_FROM_MEMORY);

lockBuffer #(16) M_WB_BUFFER_RESULT(M_RESULT, clk, M_WB_LOCK, M_WB_FLUSH, WB_RESULT);
lockBuffer #(16) M_WB_BUFFER_REMAINDER(M_REMAINDER, clk, M_WB_LOCK, M_WB_FLUSH, WB_REMAINDER);
//Components in Writeback
//Quick note: writeback address 1 is always the fireset operand in the instruction.
//              writeback address 2 is either the second operand, or it's hardcoded to 15 (if dividing or multiplying)
//              writeback data 1 is either from memory (if loading from data mem) or the result of the ALU.
//              writeback data 2 is always from the ALU Remainder.
mux_2tol #(16) WB_MUX_MEM_TO_REG(WB_RESULT, WB_DATA_FROM_MEMORY, WB_MEMTOREG, WB_MUX_MEM_TO_REG_RESULT); //WB_MUX_MEM_TO_REG_RESULT is the data from data memory
mux_2tol #(4) WB_MUX_W2_ADDR_SRC(WB_INSTRUCTION[7:4], 4'hF, WB_W2_ADDR_SRC, WB_WRITE_ADDR_2);

```



```verilog
always @ (posedge clk or negedge reset)
begin
    if (~reset)
        begin
        //IF_ID_LOCK, ID_EX_LOCK, EX_M_FLUSH handled by hazard unit
            ERROR_SERVICE_ROUTINE_ADDRESS <= 16'h0000;
            errors <= 16'h0000;
        end
    else
    begin
        //Accumulate errors in error register
        errors <= {{13'b0}, IF_ADDRESS_OVERFLOW, CONTROL_ERROR, EX_ALU_ERROR};
        //IF_PC_WRT <= 1'b1;
     if (IF_ADDRESS_OVERFLOW || CONTROL_ERROR || EX_ALU_ERROR)
             begin
                 //Error handling : use Error Interrupt Service Routine
                 //IF_ID_LOCK, ID_EX_LOCK, EX_M_FLUSH handled by hazard unit
                 ERROR_SERVICE_ROUTINE_ADDRESS <= 16'h0000;
                 USE_ERROR_ADDRESS <= 1'b1;
             end
         else
            begin
                ERROR_SERVICE_ROUTINE_ADDRESS <= 16'h0000;
                USE_ERROR_ADDRESS <= 1'b0;
            end
    end
end
endmodule


module data_memoryDUMMY  (read, write, addr, write_data, data_out, byte_select);
    input [15:0] addr, write_data;
    input read, write, byte_select;
    output reg [15:0] data_out;
initial data_out = 16'h0000;

always@(*)
    begin
    if (read) data_out = 16'hFAFA;
    else data_out = 16'h0000;
    end
endmodule
```

Truth Tables / Signals

Control Unit Truth Table

Command	Opcode	W2 ADSRC	W2EN	WB	MtR	ALU SRC	ALU OP (INSTR[3..0]	MR	MW	BS
Signed Add	1111	X	0	1	0	0	0000	0	0	0
Signed Subtract	1111	X	0	1	0	0	0001	0	0	0
Signed Multiply	1111	1	1	1	0	0	0100	0	0	0
Signed Division	1111	1	1	1	0	0	0101	0	0	0
Move	1111	X	0	1	0	0	0111	0	0	0
Swap	1111	0	1	1	0	0	1000	0	0	0
AND Immediate	1000	X	0	1	0	1	x	0	0	0
OR immediate	1001	X	0	1	0	1	x	0	0	0
Load Byte Unsigned	1010	X	0	1	1	1	x	1	0	1
Store Byte	1011	X	0	0	0	1	x	0	1	1
Load	1100	X	0	1	1	1	x	1	0	0
Store	1101	X	0	0	0	1	x	0	1	0
Branch on Less	0101	X	0	0	0	0	x	0	0	0
Branch on Greater	0100	X	0	0	0	0	x	0	0	0
Branch on Equal	0110	X	0	0	0	0	x	0	0	0
Jump	0001	X	0	0	0	0	x	0	0	0
Halt	0000	X	0	0	0	0	x	0	0	0

Jump Control Truth Table

Command	NEG (from ALU)	ZERO (from ALU)	PC SRC (out)
Jump Less Than	0	0	0
Jump Less Than	1	X	1
Jump Equal	x	0	0
Jump Equal	0	1	1
Jump Greater Than	0	0	1
Jump Greater Than	1	X	0
Jump	X	X	1

Forwarding Unit Truth Table

(Current command is C, Command in Memory is M, command in Writeback is WB)

00: Data from Register File

01: Data from M

10: Data from WB

11: Not connected

Command	**DAT1HAZ**	**DAT2HAZ**
C-Op1 != M-Op1 or WB-Op1 C-Op2 != M-Op1 or WB-Op1	00	00
C-Op1 != M-Op1 or WB-Op1 C-Op2 = M-Op1	00	01
C-Op1 != M-Op1 or WB-Op1 C-Op2 = WB-Op2	00	10
C-Op1 = M-Op1 C-Op2 != M-Op1 or WB-Op1	01	00

C-Op1 = M-Op1
C-Op2 = M-Op1
	01	01

C-Op1 = M-Op1
C-Op2 = WB-Op1
	01	10

C-Op1 = WB-Op1 C-Op2 != M-Op1 or WB-Op1	10	00

C-Op1 = WB-Op1
C-Op2 = M-Op1
	10	01

C-Op1 = WB-Op1
C-Op2 = WB-Op1
	10	10

Instruction Hazard Detection Truth Table

(Current command is C, Command in Memory is M)

Instruction Hazard unit's job is to create a bubble (prevent instructions from moving) through the IF/ID, ID/EX buffers in case of a load word or load byte.

Condition	**No Write**	**PC Wrt**

M= LW or M=LB
And(C-Op1 or C-Op2 = M-Op1)
	1	0

Jump Taken (from Jump Control)	1	1

Signal Names

W2 ADSRC: Source of Write Address 2.

0: Instruction[7..3]

1: Hardcoded 15

W2 EN: Write into Write Address 2?

0: Don't write

1: Write ALU Remainder into register file, address 2. ALU Remainder is used for Swap and Move.

WB: Write into Write Address 1?

0: Don't write

1: Write into register file, Address 1

MtR: Memory to Register

0: Write ALU Result into Register

1: Write data from memory into Register

ALU SRC: Source of ALU Data. Overridden by data forwarding!

0: Use data from RD1 for ALU 1

1: Use data from Sign Extended lower half of instruction for ALU 1.

MR: Read data from memory?

0: Read data from memory

1: Do not read from memory

MW: Write data into memory?

0: Do not write into memory at address ADDR

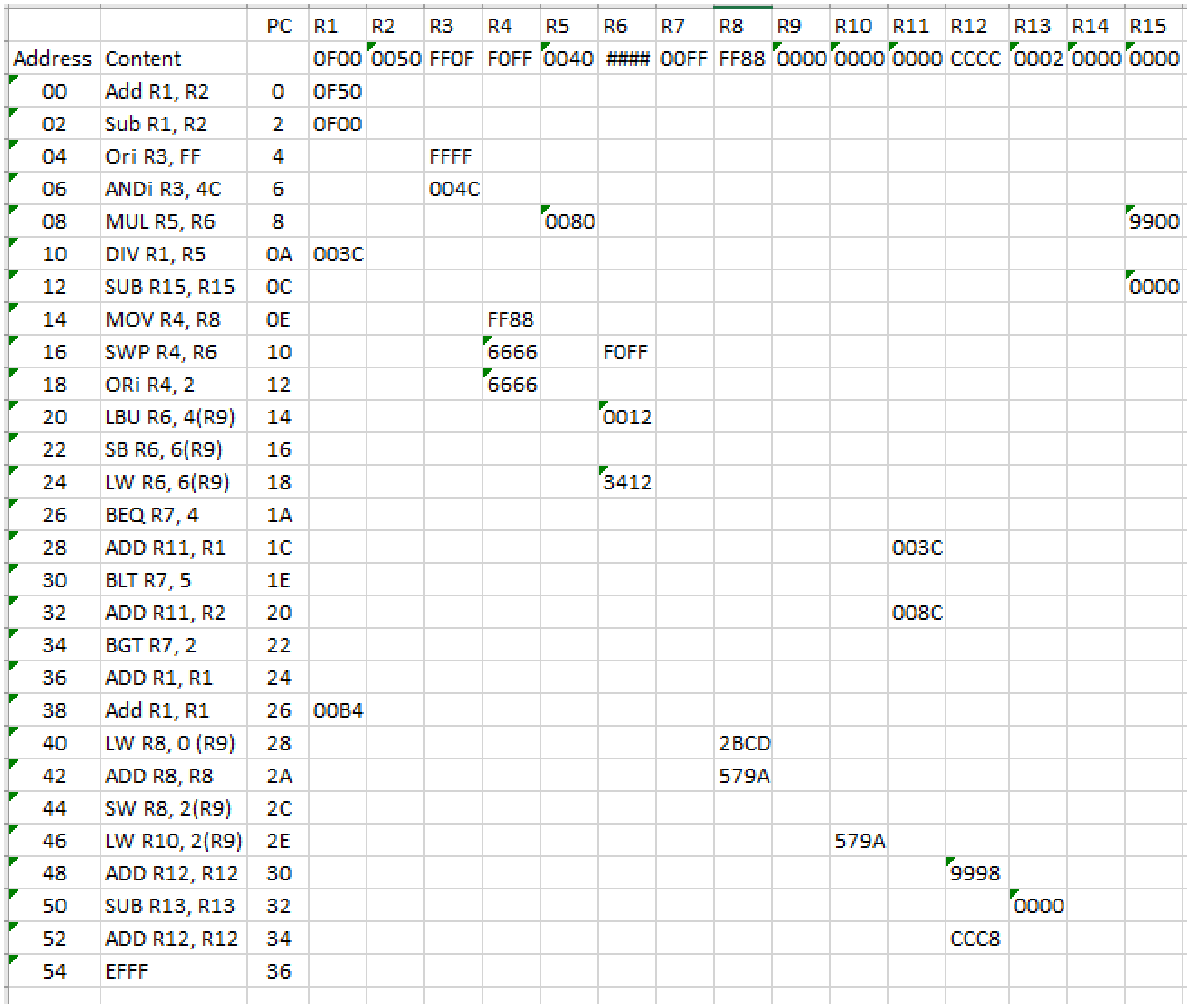
1: Write into memory at address ADDR

BS: Byte select

0: Read and write entire 16-bit word

1: Read and write half word

Test Assembly



		PC	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15
Address	Content		0F00	0050	FF0F	F0FF	0040	####	00FF	FF88	0000	0000	0000	CCCC	0002	0000	0000
00	Add R1, R2	0	0F50														
02	Sub R1, R2	2	0F00														
04	Ori R3, FF	4			FFFF												
06	ANDi R3, 4C	6			004C												
08	MUL R5, R6	8					0080										9900
10	DIV R1, R5	0A	003C														
12	SUB R15, R15	0C															0000
14	MOV R4, R8	0E				FF88											
16	SWP R4, R6	10				6666		F0FF									
18	ORi R4, 2	12				6666											
20	LBU R6, 4(R9)	14						0012									
22	SB R6, 6(R9)	16															
24	LW R6, 6(R9)	18						3412									
26	BEQ R7, 4	1A															
28	ADD R11, R1	1C											003C				
30	BLT R7, 5	1E															
32	ADD R11, R2	20											008C				
34	BGT R7, 2	22															
36	ADD R1, R1	24															
38	Add R1, R1	26	00B4														
40	LW R8, 0 (R9)	28								2BCD							
42	ADD R8, R8	2A								579A							
44	SW R8, 2(R9)	2C															
46	LW R10, 2(R9)	2E										579A					
48	ADD R12, R12	30												9998			
50	SUB R13, R13	32													0000		
52	ADD R12, R12	34												CCC8			
54	EFFF	36															