`timescale 1ns / 1ps

module cpu(

input clk

);

wire [4:0] ID\_EX\_Reg\_rd, ID\_EX\_Rs1\_Addr, ID\_EX\_Rs2\_Addr, EX\_MEM\_Rs2\_Addr, EX\_MEM\_Reg\_rd, MEM\_WB\_Reg\_rd;

wire [31:0] EX\_MEM\_nextPC, EX\_MEM\_JumpTo, EX\_MEM\_ALUResult, EX\_MEM\_data\_2, Mem\_WriteData;

wire [31:0] MEM\_WB\_nextPC, MEM\_WB\_Read\_Mem, MEM\_WB\_ALUResult;

wire [3:0] ALUControl, ID\_EX\_ALUInstruct;

wire [2:0] EX\_MEM\_Funct3;//FOR LB

wire [1:0] ALUOp, MemtoReg, ID\_EX\_ALUOp, ID\_EX\_MemtoReg, EX\_MEM\_MemtoReg, MEM\_WB\_MemtoReg;

wire Branch, isBranch, MemRead, MemWrite, ALUSrc, isJump, RegWrite, isZero;

wire IF\_Flush, ID\_EX\_MemRead, ID\_EX\_MemWrite, ID\_EX\_ALUSrc, ID\_EX\_isJump, ID\_EX\_RegWrite;

wire EX\_MEM\_Branch, EX\_MEM\_MemRead, EX\_MEM\_MemWrite, EX\_MEM\_RegWrite, MEM\_WB\_RegWrite, MEM\_WB\_MemRead;//MEM\_WB\_MEMREAD NOT NECESSARY

wire [31:0] mux2\_out, mux3\_out; //FOR COM FORWARD

wire PC\_write, IF\_ID\_write, Mux\_Con; //HAZARD

wire [1:0] ForwardA, ForwardB; //ALU FORWARD

wire Forward\_Com1, Forward\_Com2, Forward\_MemSrc;//COM FORWARD AND MEM FORWARD

wire [9:0] allControlIn, allControlOut;//CONTROL HAZARD

wire [31:0] Instruct, Imm, Imm\_shift, data\_1, data\_2, MUX\_ALU, loadPC, JumpPC, W\_data;

wire [31:0] ALUResult, Read\_Mem, JumpTo, ALUInput1, ALUInput2;

wire [31:0] IF\_ID\_nextPC, IF\_ID\_currentPC, IF\_ID\_Instruct;

wire [31:0] ID\_EX\_nextPC, ID\_EX\_currentPC, ID\_EX\_data\_1, ID\_EX\_data\_2, ID\_EX\_Imm, EX\_MEM\_Imm, MEM\_WB\_Imm;

reg [31:0] PC;

initial PC = 0;

always @ (posedge clk)

if (PC\_write == 1'b1) PC <= loadPC;

Hazard HD (

.IF\_ID\_r1 (IF\_ID\_Instruct[19:15]),

.IF\_ID\_r2 (IF\_ID\_Instruct[24:20]),

.ID\_EX\_RegRd (ID\_EX\_Reg\_rd),

.EX\_MEM\_RegRd (EX\_MEM\_Reg\_rd),

.ID\_EX\_MemRead (ID\_EX\_MemRead),

.EX\_MEM\_MemRead (EX\_MEM\_MemRead),

.ID\_branch (allControlIn[5]),

.ID\_MemWrite (allControlIn[3]),

.ID\_EX\_RegWrite (ID\_EX\_RegWrite),

.PC\_write (PC\_write),

.IF\_ID\_write (IF\_ID\_write),

.Mux\_Control (Mux\_Con)

);

Forwarding FU (

.ID\_EX\_r1 (ID\_EX\_Rs1\_Addr),

.ID\_EX\_r2 (ID\_EX\_Rs2\_Addr),

.IF\_ID\_r1 (IF\_ID\_Instruct[19:15]),

.IF\_ID\_r2 (IF\_ID\_Instruct[24:20]),

.EX\_MEM\_r2 (EX\_MEM\_Rs2\_Addr),

.MEM\_WB\_RegRd (MEM\_WB\_Reg\_rd),

.EX\_MEM\_RegRd (EX\_MEM\_Reg\_rd),

.ID\_EX\_RegRd (ID\_EX\_Reg\_rd),

.MEM\_WB\_RegWrite (MEM\_WB\_RegWrite),

.EX\_MEM\_RegWrite (EX\_MEM\_RegWrite),

.EX\_MEM\_MemWrite (EX\_MEM\_MemWrite),

.MEM\_WB\_MemRead (MEM\_WB\_MemRead),

.EX\_MEM\_MemRead (EX\_MEM\_MemRead),

.ID\_EX\_MemRead (ID\_EX\_MemRead),

.ID\_EX\_MemWrite (ID\_EX\_MemWrite),

.ID\_EX\_RegWrite (ID\_EX\_RegWrite),

.IF\_ID\_branch (Branch),

.Forwarding\_A (ForwardA),

.Forwarding\_B (ForwardB),

.Forwarding\_Com1 (Forward\_Com1),

.Forwarding\_Com2 (Forward\_Com2),

.MemSrc (Forward\_MemSrc)

);

Instruction\_Memory InstructionMem (

.instruct\_out (Instruct),

.addr (PC)

);

Control Control (

.opcode (IF\_ID\_Instruct[6:0]),

.allControl (allControlIn)

);

Register\_File RegisterFile (

.W\_en (MEM\_WB\_RegWrite),

.W\_data (W\_data),

.R\_addr\_1 (IF\_ID\_Instruct[19:15]),

.R\_addr\_2 (IF\_ID\_Instruct[24:20]),

.W\_addr (MEM\_WB\_Reg\_rd),

.R\_data\_1 (data\_1),

.R\_data\_2 (data\_2)

);

Comparator Compare (

.func ({IF\_ID\_Instruct[2], IF\_ID\_Instruct[14:12]}),

.in1 (mux2\_out),

.in2 (mux3\_out),

.isZero (isZero)

);

Immediate\_Generator ImmGen (

.In (IF\_ID\_Instruct),

.Out (Imm)

);

ALU ALU (

.data1 (ALUInput1),

.data2 (ALUInput2),

.sel (ALUControl),

.result (ALUResult)

);

ALU\_Control ALU\_Control (

.ALU\_op (ID\_EX\_ALUOp),

.instruct (ID\_EX\_ALUInstruct),

.ALU\_sel (ALUControl)

);

Data\_Memory DataMem (

.W\_en (EX\_MEM\_MemWrite),

.R\_en (EX\_MEM\_MemRead),

.Funct3 (EX\_MEM\_Funct3),

.addr (EX\_MEM\_ALUResult),

.W\_data (Mem\_WriteData),

.R\_data\_out (Read\_Mem)

);

IF\_ID\_State\_Reg IF\_ID (

.clock (clk),

.IF\_Flush (IF\_Flush),

.IF\_ID\_Write (IF\_ID\_write),

.currentPC (PC),

.nextPC (PC+32'h00000004),

.Instruct (Instruct),

.currentPC\_out (IF\_ID\_currentPC),

.nextPC\_out (IF\_ID\_nextPC),

.Instruct\_out (IF\_ID\_Instruct)

);

ID\_EX\_State\_Reg ID\_EX (

.clock (clk),

.RegWrite (RegWrite),

.MemtoReg (MemtoReg),

.MemRead (MemRead),

.MemWrite (MemWrite),

.Jump (isJump),

.ALUSrc (ALUSrc),

.ALUOp (ALUOp),

.currentPC (IF\_ID\_currentPC),

.nextPC (IF\_ID\_nextPC),

.Reg\_rs1 (data\_1),

.Reg\_rs2 (data\_2),

.Reg\_rs1\_addr (IF\_ID\_Instruct[19:15]),

.Reg\_rs2\_addr (IF\_ID\_Instruct[24:20]),

.Imm\_Gen (Imm),

.Reg\_rd (IF\_ID\_Instruct[11:7]),

.ALU\_Instruct ({IF\_ID\_Instruct[30],IF\_ID\_Instruct[14:12]}),

.RegWrite\_out (ID\_EX\_RegWrite),

.MemtoReg\_out (ID\_EX\_MemtoReg),

.MemRead\_out (ID\_EX\_MemRead),

.MemWrite\_out (ID\_EX\_MemWrite),

.Jump\_out (ID\_EX\_isJump),

.ALUSrc\_out (ID\_EX\_ALUSrc),

.ALUOp\_out (ID\_EX\_ALUOp),

.currentPC\_out (ID\_EX\_currentPC),

.nextPC\_out (ID\_EX\_nextPC),

.Reg\_rs1\_out (ID\_EX\_data\_1),

.Reg\_rs2\_out (ID\_EX\_data\_2),

.Reg\_rs1\_addr\_out (ID\_EX\_Rs1\_Addr),

.Reg\_rs2\_addr\_out (ID\_EX\_Rs2\_Addr),

.Imm\_Gen\_out (ID\_EX\_Imm),

.Reg\_rd\_out (ID\_EX\_Reg\_rd),

.ALU\_Instruct\_out (ID\_EX\_ALUInstruct)

);

EX\_MEM\_State\_Reg EX\_MEM (

.clock (clk),

.RegWrite (ID\_EX\_RegWrite),

.MemtoReg (ID\_EX\_MemtoReg),

.MemRead (ID\_EX\_MemRead),

.MemWrite (ID\_EX\_MemWrite),

.imm (ID\_EX\_Imm),

.nextPC (ID\_EX\_nextPC),

.ALUResult (ALUResult),

.Reg\_rs2 (MUX\_ALU),

.Funct3 (ID\_EX\_ALUInstruct[2:0]),

.Reg\_rd (ID\_EX\_Reg\_rd),

.Reg\_rs2\_addr (ID\_EX\_Rs2\_Addr),

.RegWrite\_out (EX\_MEM\_RegWrite),

.MemtoReg\_out (EX\_MEM\_MemtoReg),

.MemRead\_out (EX\_MEM\_MemRead),

.MemWrite\_out (EX\_MEM\_MemWrite),

.imm\_out (EX\_MEM\_Imm),

.nextPC\_out (EX\_MEM\_nextPC),

.ALUResult\_out (EX\_MEM\_ALUResult),

.Reg\_rs2\_out (EX\_MEM\_data\_2),

.Funct3\_out (EX\_MEM\_Funct3),

.Reg\_rd\_out (EX\_MEM\_Reg\_rd),

.Reg\_rs2\_addr\_out (EX\_MEM\_Rs2\_Addr)

);

MEM\_WB\_State\_Reg MEM\_WB (

.clock (clk),

.MemRead (EX\_MEM\_MemRead),

.RegWrite (EX\_MEM\_RegWrite),

.MemtoReg (EX\_MEM\_MemtoReg),

.imm (EX\_MEM\_Imm),

.nextPC (EX\_MEM\_nextPC),

.ReadData (Read\_Mem),

.ALUResult (EX\_MEM\_ALUResult),

.Reg\_rd (EX\_MEM\_Reg\_rd),

.MemRead\_out (MEM\_WB\_MemRead),

.RegWrite\_out (MEM\_WB\_RegWrite),

.MemtoReg\_out (MEM\_WB\_MemtoReg),

.imm\_out (MEM\_WB\_Imm),

.nextPC\_out (MEM\_WB\_nextPC),

.ReadData\_out (MEM\_WB\_Read\_Mem),

.ALUResult\_out (MEM\_WB\_ALUResult),

.Reg\_rd\_out (MEM\_WB\_Reg\_rd)

);

and (isBranch, Branch, isZero);

or (IF\_Flush, isBranch, isJump);

assign Imm\_shift = {Imm[30:0],1'b0};

assign RegWrite = allControlOut[0];

assign isJump = allControlOut[1];

assign ALUSrc = allControlOut[2];

assign MemWrite = allControlOut[3];

assign MemRead = allControlOut[4];

assign Branch = allControlOut[5];

assign MemtoReg = allControlOut[7:6];

assign ALUOp = allControlOut[9:8];

MUX\_4to1 Mux\_1(.data1(PC+32'h00000004), .data2(IF\_ID\_currentPC+Imm\_shift), .data3(data\_1+Imm), .data4(IF\_ID\_currentPC+Imm\_shift),.sel({isJump,isBranch}), .result(loadPC)); //judge pc

MUX\_2to1 Mux\_2(.data1(data\_1), .data2(EX\_MEM\_ALUResult), .sel(Forward\_Com1), .result(mux2\_out)); //comparator

MUX\_2to1 Mux\_3(.data1(data\_2), .data2(EX\_MEM\_ALUResult), .sel(Forward\_Com2), .result(mux3\_out)); //comparator

MUX\_4to1 Mux\_4(.data1(MEM\_WB\_Read\_Mem), .data2(MEM\_WB\_nextPC), .data3(MEM\_WB\_Imm), .data4(MEM\_WB\_ALUResult), .sel(MEM\_WB\_MemtoReg), .result(W\_data)); //judge write data

MUX\_2to1 Mux\_5(.data1(10'b0000000000), .data2(allControlIn), .sel(Mux\_Con), .result(allControlOut)); //control hazard

MUX\_4to1 Mux\_6(.data1(ID\_EX\_data\_1), .data2(W\_data), .data3(EX\_MEM\_ALUResult), .data4(EX\_MEM\_ALUResult), .sel(ForwardA), .result(ALUInput1)); //ALU forA source1

MUX\_4to1 Mux\_7(.data1(ID\_EX\_data\_2), .data2(W\_data), .data3(EX\_MEM\_ALUResult), .data4(EX\_MEM\_ALUResult), .sel(ForwardB), .result(MUX\_ALU)); //ALU forB

MUX\_2to1 Mux\_8(.data1(MUX\_ALU), .data2(ID\_EX\_Imm), .sel(ID\_EX\_ALUSrc), .result(ALUInput2)); //ALU source2

MUX\_2to1 Mux\_9(.data1(EX\_MEM\_data\_2), .data2(W\_data), .sel(Forward\_MemSrc), .result(Mem\_WriteData)); // MEM for

endmodule

module Register\_File

(

input W\_en,

input [31:0] W\_data,

input [4:0] R\_addr\_1, R\_addr\_2, W\_addr,

output [31:0] R\_data\_1, R\_data\_2

);

reg [31:0] data [31:0];

integer i;

initial begin

for(i=0; i<32; i=i+1)

data[i] <= 0;

end

always @ (\*) begin

if (W\_en && W\_addr!= 0) begin

data[W\_addr] = W\_data;

end

end

assign R\_data\_1 = data[R\_addr\_1];

assign R\_data\_2 = data[R\_addr\_2];

endmodule

module Comparator #

(

parameter BEQ = 4'b0000, parameter BNE = 4'b0001, parameter BLT = 4'b0100, parameter BGE = 4'b0101

)

(

input [3:0] func,

input [31:0] in1, in2,

output reg isZero

);

initial isZero = 1'b1;

always @ (\*) begin

case (func)

BEQ: isZero = (in1==in2)?1'b1:1'b0;

BNE: isZero = (in1==in2)?1'b0:1'b1;

BLT: isZero = ($signed(in1) < $signed(in2))?1'b1:1'b0;

BGE: isZero = ($signed(in1) < $signed(in2))?1'b0:1'b1;

default: isZero = 1'b1;

endcase

end

endmodule

module ALU #

(

parameter ADD = 4'b0010, parameter SUB = 4'b1000, parameter BNE = 4'b1001,

parameter XOR = 4'b0100, parameter OR = 4'b0110, parameter AND = 4'b0111, parameter BLT = 4'b1100,

parameter SLL = 4'b0001, parameter SRL = 4'b0101, parameter SRA = 4'b1101, parameter BGE = 4'b1110

)(

input [3:0] sel,

input [31:0] data1, data2,

//output reg Zero,

output reg [31:0] result

);

always @ (\*) begin

case (sel)

ADD:

begin

result = data1 + data2; //add

//Zero = 1'b1; //jal & jalr

end

SUB:

begin

if (data1 == data2) begin

//Zero = 1;

result = 0;

end

else begin

//Zero = 0;

result = data1 - data2; //sub

end

end

BNE:

begin

if (data1 == data2) begin

//Zero = 0;

result = 0;

end

else begin

//Zero = 1;

result = data1 - data2; //sub

end

end

BLT:

begin

if ($signed(data1) < $signed(data2)) begin

//Zero = 1;

result = data1 - data2;

end

else begin

//Zero = 0;

result = data1 - data2;

end

end

BGE:

begin

if ($signed(data1) < $signed(data2)) begin

//Zero = 0;

result = data1 - data2;

end

else begin

//Zero = 1;

result = data1 - data2;

end

end

XOR: result = data1 ^ data2; //xor

OR: result = data1 | data2; //or

AND: result = data1 & data2; //and

SLL: result = data1 <<data2; //sll

SRL: result = data1 >>data2; //srl

SRA: result = $signed(data1>>data2); //sra

default: begin

result = 0;

//Zero = 0;

end

endcase

end

endmodule

module ALU\_Control

(

input [1:0] ALU\_op,

input [3:0] instruct,

output reg [3:0] ALU\_sel

);

always @ (\*) begin

case (ALU\_op)

2'b00:

begin

ALU\_sel = 4'b0010; //Addition

end

2'b01:

begin

if (instruct[2:0] == 3'b101)

begin

ALU\_sel = 4'b1110;

end //bge

else

begin

ALU\_sel = {1'b1, instruct[2:0]};

end //beq,bne,blt

end

2'b10:

begin

if (instruct == 0)

begin

ALU\_sel = 4'b0010;

end //add

else

begin

ALU\_sel = instruct;

end //aub,and,or,xor,sll,srl,sra

end

2'b11:

begin

if (instruct[2:0] == 0)

begin

ALU\_sel = 4'b0010;

end //addi

else

begin

ALU\_sel = {1'b0, instruct[2:0]};

end //andi,slli,srli

end

default:ALU\_sel = 4'b0000;

endcase

end

endmodule

module Immediate\_Generator

(

input [31:0] In,

output reg [31:0] Out

);

always @ (\*) begin

case (In[6:0])

7'b0000011: Out = {{20{In[31]}}, In[31:20]};

7'b0001111: Out = {{20{In[31]}}, In[31:20]};

7'b0010011: Out = {{20{In[31]}}, In[31:20]};

7'b0100011: Out = {{20{In[31]}}, In[31:25], In[11:7]};

7'b1100011: Out = {{20{In[31]}}, In[31], In[7], In[30:25], In[11:8]};

7'b1100111: Out = {{20{In[31]}}, In[31:20]};

7'b1101111: Out = {{20{In[31]}}, In[31], In[19:12], In[20], In[30:21]};

default: Out = 0;

endcase

end

endmodule

module Control //rewrite for convenience

(

input [6:0] opcode,

output reg [9:0] allControl

// output reg [1:0] ALUOp, MemtoReg,

// output reg Branch, MemRead, MemWrite, ALUSrc, Jump, RegWrite

);

initial begin

allControl <= 0;

end

always @ (opcode) begin

case (opcode)

//I-type,load

7'b0000011: allControl <= 10'b0000010101;

7'b0010011: allControl <= 10'b1111000101;

7'b0100011: allControl <= 10'b0011001100;

7'b1100011: allControl <= 10'b0111100000;

7'b1100111: allControl <= 10'b0001000111;

7'b1101111: allControl <= 10'b0001100111;

7'b0110011: allControl <= 10'b1011000001;

default: allControl <= 0;

endcase

end

endmodule

module MUX\_2to1

(

input sel,

input [31:0] data1, data2,

output reg [31:0] result

);

always @ (\*) begin

case (sel)

1'b0: result = data1;

1'b1: result = data2;

default: result = 0;

endcase

end

endmodule

module MUX\_4to1

(

input [1:0] sel,

input [31:0] data1, data2, data3, data4,

output reg [31:0] result

);

always @ (\*) begin

case (sel)

2'b00: result = data1;

2'b01: result = data2;

2'b10: result = data3;

2'b11: result = data4;

default: result = 0;

endcase

end

endmodule

module Instruction\_Memory (

input [31:0] addr,

output reg [31:0] instruct\_out

);

reg [31:0] Instruction [127:0];

initial begin

Instruction[0] = 32'b00111001100100000000001100010011;

Instruction[1] = 32'b00000000011000000010001000100011;

Instruction[2] = 32'b00000000010000000000001010000011;

Instruction[3] = 32'b00000000010100000010000000100011;

Instruction[4] = 32'b00000010000000110000000001100011;

Instruction[5] = 32'b00000000000000000010111000000011;

Instruction[6] = 32'b00000001110000101001110001100011;

Instruction[7] = 32'b00000001110000101000001110110011;

Instruction[8] = 32'b00000001110000111111001100110011;

Instruction[9] = 32'b00000000000000111111001100010011;

Instruction[10] = 32'b01000000000000110000001010110011;

Instruction[11] = 32'b00000000011000101101010001100011;

Instruction[12] = 32'b00000000000000000000001110110011;

Instruction[13] = 32'b00000000110000000000000011101111;

Instruction[14] = 32'b00000001010000000000000011101111;

Instruction[15] = 32'b00000000000000000000111000110011;

Instruction[16] = 32'b00000000011111100110111000110011;

Instruction[17] = 32'b00000000000000001000000001100111;

Instruction[18] = 32'b00000100100000000000001100010011;

Instruction[19] = 32'b00001010110000000000001010010011;

end

always @ (\*) begin

instruct\_out = Instruction[(addr>>2)];

end

endmodule

module Data\_Memory #

(

parameter SW = 3'b010,parameter SB = 3'b000, parameter LBU = 3'b100,parameter LW = 3'b010,parameter LB = 3'b000

)(

input W\_en, R\_en,

input [2:0] Funct3,

input [31:0] addr, W\_data,

output reg [31:0] R\_data\_out

);

reg [7:0] data[127:0];

//Write to Data Memory

always @ (\*) begin

if (W\_en) begin

case (Funct3)

SW:

begin //save word

data[addr] = W\_data[7:0];

data[addr+1] = W\_data[15:8];

data[addr+2] = W\_data[23:16];

data[addr+3] = W\_data[31:24];

end

SB:

begin //save byte

data[addr] = W\_data[7:0];

end

default: data[addr] = data[addr];

endcase

end

end

always @ (\*) begin

if (R\_en) begin

case (Funct3)

LW:

begin //load word

R\_data\_out = {data[addr+3], data[addr+2], data[addr+1], data[addr]};

end

LB:

begin //load byte

R\_data\_out = {{24{data[addr][7]}}, data[addr]};

end

LBU:

begin //load byte unsigned

R\_data\_out = {{24{1'b0}}, data[addr]};

end

default: R\_data\_out = R\_data\_out;

endcase

end

end

endmodule

module IF\_ID\_State\_Reg

(

input clock,

input IF\_Flush, // new control

input IF\_ID\_Write, // new control

input [31:0] currentPC,

input [31:0] nextPC,

input [31:0] Instruct,

output reg [31:0] currentPC\_out,

output reg [31:0] nextPC\_out,

output reg [31:0] Instruct\_out

);

initial begin

currentPC\_out = 0;

nextPC\_out = 0;

Instruct\_out = 0;

end

always @ (posedge clock) begin

if (IF\_Flush == 1'b1) begin

currentPC\_out = 0;

nextPC\_out = 0;

Instruct\_out = 0;

end

else if (IF\_ID\_Write == 1'b1) begin

currentPC\_out = currentPC;

nextPC\_out = nextPC;

Instruct\_out = Instruct;

end

end

endmodule

module ID\_EX\_State\_Reg

(

input clock,

input RegWrite,

input [1:0] MemtoReg,

input MemRead,

input MemWrite,

input Jump,

input ALUSrc,

input [1:0] ALUOp,

input [31:0] currentPC,

input [31:0] nextPC,

input [31:0] Reg\_rs1,

input [31:0] Reg\_rs2,

input [4:0] Reg\_rs1\_addr,

input [4:0] Reg\_rs2\_addr,

input [31:0] Imm\_Gen,

input [4:0] Reg\_rd,

input [3:0] ALU\_Instruct,

output reg RegWrite\_out,

output reg [1:0] MemtoReg\_out,

output reg MemRead\_out,

output reg MemWrite\_out,

output reg Jump\_out,

output reg ALUSrc\_out,

output reg [1:0] ALUOp\_out,

output reg [31:0] currentPC\_out,

output reg [31:0] nextPC\_out,

output reg [31:0] Reg\_rs1\_out,

output reg [31:0] Reg\_rs2\_out,

output reg [4:0] Reg\_rs1\_addr\_out, //new for forward

output reg [4:0] Reg\_rs2\_addr\_out, //new for forward

output reg [31:0] Imm\_Gen\_out,

output reg [4:0] Reg\_rd\_out,

output reg [3:0] ALU\_Instruct\_out

);

initial begin //Bug\_06

RegWrite\_out = 0;

MemtoReg\_out = 0;

MemRead\_out = 0;

MemWrite\_out = 0;

Jump\_out = 0;

ALUSrc\_out = 0;

ALUOp\_out = 0;

currentPC\_out = 0;

nextPC\_out = 0;

Reg\_rs1\_out = 0;

Reg\_rs2\_out = 0;

Imm\_Gen\_out = 0;

Reg\_rd\_out = 0;

ALU\_Instruct\_out = 0;

Reg\_rs1\_addr\_out = 0;

Reg\_rs2\_addr\_out = 0;

end

always @ (posedge clock) begin

RegWrite\_out = RegWrite;

MemtoReg\_out = MemtoReg;

MemRead\_out = MemRead;

MemWrite\_out = MemWrite;

Jump\_out = Jump;

ALUSrc\_out = ALUSrc;

ALUOp\_out = ALUOp;

currentPC\_out = currentPC;

nextPC\_out = nextPC;

Reg\_rs1\_out = Reg\_rs1;

Reg\_rs2\_out = Reg\_rs2;

Reg\_rs1\_addr\_out = Reg\_rs1\_addr; //new for forward

Reg\_rs2\_addr\_out = Reg\_rs2\_addr; //new for forward

Reg\_rd\_out = Reg\_rd;

Imm\_Gen\_out = Imm\_Gen;

ALU\_Instruct\_out = ALU\_Instruct;

end

endmodule

module EX\_MEM\_State\_Reg

(

input clock,

input RegWrite,

input [1:0] MemtoReg,

input MemRead,

input MemWrite,

input [31:0] imm,

input [31:0] nextPC,

input [31:0] ALUResult,

input [31:0] Reg\_rs2,

input [2:0] Funct3,

input [4:0] Reg\_rd,

input [4:0] Reg\_rs2\_addr, //new for forward

output reg RegWrite\_out,

output reg [1:0] MemtoReg\_out,

output reg MemRead\_out,

output reg MemWrite\_out,

output reg [31:0] imm\_out,

output reg [31:0] nextPC\_out,

output reg [31:0] ALUResult\_out,

output reg [31:0] Reg\_rs2\_out,

output reg [2:0] Funct3\_out,

output reg [4:0] Reg\_rd\_out,

output reg [4:0] Reg\_rs2\_addr\_out

);

initial begin //Bug\_07

RegWrite\_out = 0;

MemtoReg\_out = 0;

MemRead\_out = 0;

MemWrite\_out = 0;

nextPC\_out = 0;

ALUResult\_out = 0;

Reg\_rs2\_out = 0;

Funct3\_out = 0;

Reg\_rd\_out = 0;

imm\_out = 0;

end

always @ (posedge clock) begin

RegWrite\_out = RegWrite;

MemtoReg\_out = MemtoReg;

MemRead\_out = MemRead;

MemWrite\_out = MemWrite;

imm\_out = imm;

nextPC\_out = nextPC;

ALUResult\_out = ALUResult;

Reg\_rs2\_out = Reg\_rs2;

Funct3\_out = Funct3;

Reg\_rd\_out = Reg\_rd;

Reg\_rs2\_addr\_out = Reg\_rs2\_addr;

end

endmodule

module MEM\_WB\_State\_Reg

(

input clock,

input MemRead, //new for forward

input RegWrite,

input [1:0] MemtoReg,

input [31:0] imm,

input [31:0] nextPC,

input [31:0] ReadData,

input [31:0] ALUResult,

input [4:0] Reg\_rd,

output reg MemRead\_out,

output reg RegWrite\_out,

output reg [1:0] MemtoReg\_out,

output reg [31:0] imm\_out,

output reg [31:0] nextPC\_out,

output reg [31:0] ReadData\_out,

output reg [31:0] ALUResult\_out,

output reg [4:0] Reg\_rd\_out

);

initial begin

MemRead\_out = 0;

RegWrite\_out = 0;

MemtoReg\_out = 0;

nextPC\_out = 0;

ReadData\_out = 0;

ALUResult\_out = 0;

Reg\_rd\_out = 0;

imm\_out = 0;

end

always @ (posedge clock) begin

MemRead\_out = MemRead;

RegWrite\_out = RegWrite;

MemtoReg\_out = MemtoReg;

imm\_out = imm;

nextPC\_out = nextPC;

ReadData\_out = ReadData;

ALUResult\_out = ALUResult;

Reg\_rd\_out = Reg\_rd;

end

endmodule