

**本科实验报告**

|  |  |
| --- | --- |
| 课程名称： | 计算机组成 |
| 姓 名： | TANG ANNA YONGQI |
| 学 院： | 计算机科学与技术学院 |
| 专 业： | 计算机科学与技术（中加班）留学生 |
| 学 号： | 3180300155 |
| 生活照： |  |
| 指导教师： | 刘海风，洪奇军 |

2020年 6月06日

Lab 12 – Multicycle CPU Controller Extension

**Name:** Anna Yongqi Tang **ID:** 3180300155 **Major:** 计算机科学与技术（中加班）留学生

**Course:** Computer Organization

**Date:** 2020-06-06 **Instructor:** 洪奇军

1. **Method and Experimental Steps**

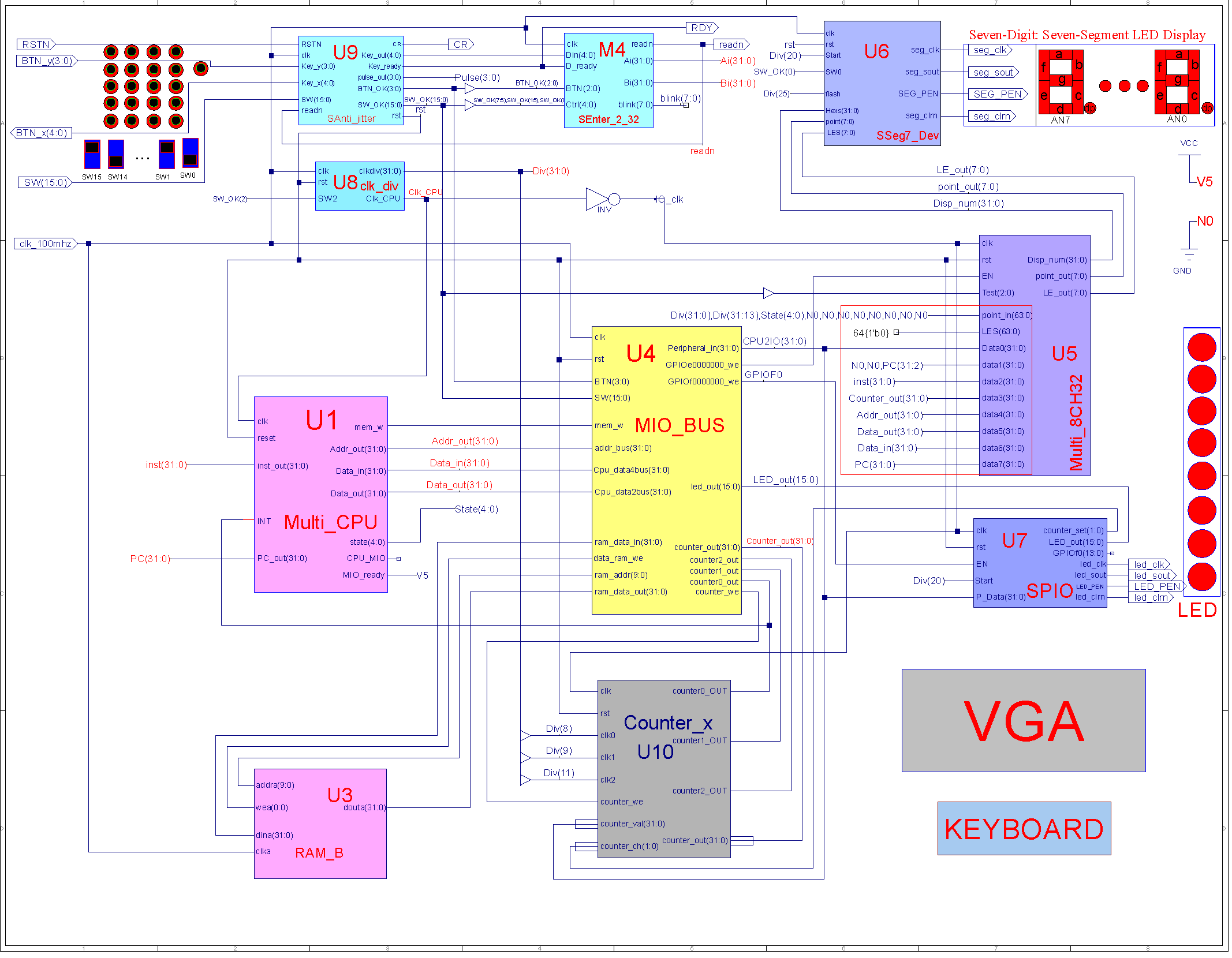


Figure 1 - topMod.sch

Lab 12 is an extension of lab 11, which more i-type and jump instructions are implemented. The .ucf for this program came from the provided courseware, and is linked to topMod.v. Synthesis had minimal warnings, and implementation was successful. A programmable file has been generated and is ready for testing on the SWORD board. The above diagram was retrieved from the lab PowerPoint. No schematic has been drawn for this experiment. The CPU currently supports add, sub, and, or, xor, nor, slt, srl, jr, jal, addi, andi, ori, xori, lui, lw, sw, beq, bne, slti, j and jal. For each state/circle in the FSM, we define different value controls. The opcodes are also set accordingly in this module. After the controller finishes processing each module, it goes back to the instruction fetch stage.

  
Figure 3 – 3180300155\_TANGANNAYONGQI\_12

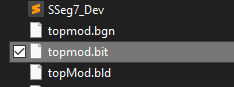
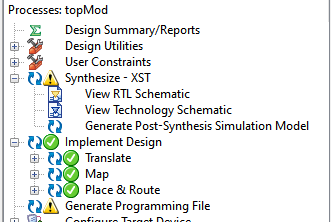


Figure 4 - .bit file generation Figure 5 - .bit file generated in directory

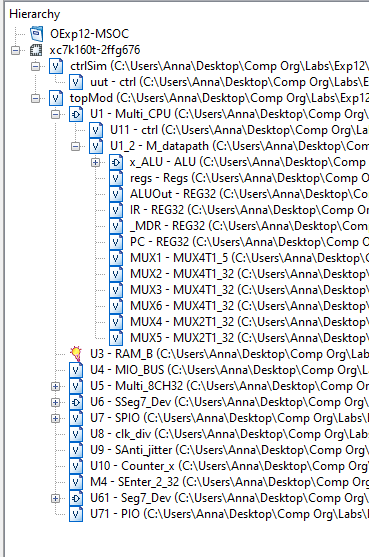
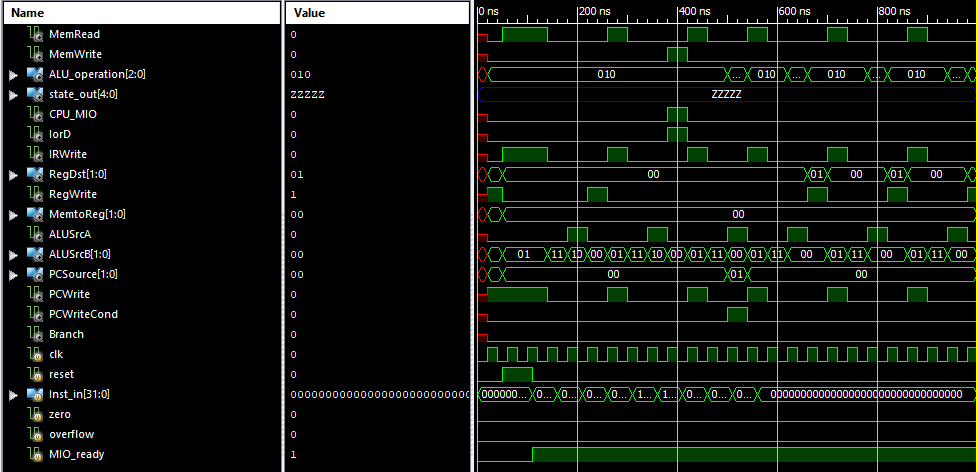


Figure 6 - file hierarchy

1. **Simulations and Observations**

This lab requires a MIPS program (mem.coe) to be tested on the multicyle CPU. The demo will be performed later in a future exercise. The controller module was simulated in this lab. I have simulated the add, addi, beq, j, lw, sw, jal, bne and lui instructions, and the results seemed to be consistent.

*ctrl Simulation*



*ctrlSim.v*

module ctrlSim;

// Inputs

reg clk;

reg reset;

reg [31:0] Inst\_in;

reg zero;

reg overflow;

reg MIO\_ready;

// Outputs

wire MemRead;

wire MemWrite;

wire [2:0] ALU\_operation;

wire [4:0] state\_out;

wire CPU\_MIO;

wire IorD;

wire IRWrite;

wire [1:0] RegDst;

wire RegWrite;

wire [1:0] MemtoReg;

wire ALUSrcA;

wire [1:0] ALUSrcB;

wire [1:0] PCSource;

wire PCWrite;

wire PCWriteCond;

wire Branch;

// Instantiate the Unit Under Test (UUT)

ctrl uut (

.clk(clk),

.reset(reset),

.Inst\_in(Inst\_in),

.zero(zero),

.overflow(overflow),

.MIO\_ready(MIO\_ready),

.MemRead(MemRead),

.MemWrite(MemWrite),

.ALU\_operation(ALU\_operation),

.state\_out(state\_out),

.CPU\_MIO(CPU\_MIO),

.IorD(IorD),

.IRWrite(IRWrite),

.RegDst(RegDst),

.RegWrite(RegWrite),

.MemtoReg(MemtoReg),

.ALUSrcA(ALUSrcA),

.ALUSrcB(ALUSrcB),

.PCSource(PCSource),

.PCWrite(PCWrite),

.PCWriteCond(PCWriteCond),

.Branch(Branch)

);

initial begin

// Initialize Inputs

clk = 0;

reset = 0;

Inst\_in = 0;

zero = 0;

overflow = 0;

MIO\_ready = 0;

// Wait 100 ns for global reset to finish

#50;

reset=1;

#60;

reset=0;

MIO\_ready=1;

Inst\_in = 32'h014B4820; //add t1, t2, t3

#50;

Inst\_in = 32'h2014003f; //addi s4, zero, 3f

#50;

Inst\_in = 32'h11600005; //beq t3, zero, 5

#50;

Inst\_in = 32'h0800000c; //j 12

#50;

Inst\_in = 32'h8D69FFFF; //lw t1, 0xffff(t3)

#50;

Inst\_in = 32'hAD71FFFF; //sw s1, 0xffff(t3)

#50;

Inst\_in = 32'h0C00BFAF; //jal bfaf

#50;

Inst\_in = 32'h15700005; //bne s0 5

#50;

Inst\_in = 32'h3C0B0001; //lui t3 1

#50;

Inst\_in = 32'h00000000;

#50;

end

always begin

clk=0;

#20;

clk=1;

#20;

end

endmodule

**3. Conclusion**

This week’s lab was a continuation of last week’s lab and concludes the lab portion of the Computer Organization course. Both the single cycle and multicycle CPUs have been completed, and I look forward to comparing their performances in a later analysis. It is a shame that I will not be able to implement these onto the SWORD board, but I greatly enjoyed doing these labs. It was a good supplement to what I have learned in the theoretical portion of this course.

1. **Source Code**

All modules and components were either retrieved from previous labs or directly taken from the given files.

*Controller – multi\_ctrl\_IO.v*

module ctrl(input clk,

input reset,

input [31:0] Inst\_in,

input zero,

input overflow,

input MIO\_ready,

output reg MemRead,

output reg MemWrite,

output reg[2:0]ALU\_operation,

output [4:0]state\_out,

output reg CPU\_MIO,

output reg IorD,

output reg IRWrite,

output reg [1:0]RegDst,

output reg RegWrite,

output reg [1:0]MemtoReg,

output reg ALUSrcA,

output reg [1:0]ALUSrcB,

output reg [1:0]PCSource,

output reg PCWrite,

output reg PCWriteCond,

output reg Branch

);

wire Rtype, LS, IBeq, Jump, Load, Store;

wire[5:0] OP = Inst\_in[31:26];

reg[3:0] state;

reg[1:0] ALUop;

parameter IF = 4'b0000, ID = 4'b0001, Mem\_Ex = 4'b0010, Mem\_RD = 4'b0011,

LW\_WB = 4'b0100, Mem\_W = 4'b0101, R\_Exc = 4'b0110, R\_WB = 4'b0111,

Beq\_Exc = 4'b1000, J = 4'b1001, I\_Exc = 5'b01010, I\_WB = 5'b01011,

Lui\_Exc = 5'b01100, Bne\_Exc = 5'b01101, Jr = 5'b01110, Jal = 5'b01111,

Jalr = 5'b10000, Error = 4'b1111;

`define Datapath\_signals {PCWrite, PCWriteCond,IorD, MemRead, MemWrite,IRWrite, MemtoReg, PCSource, ALUSrcA, ALUSrcB, RegWrite, RegDst, Branch, ALUop, CPU\_MIO}

parameter value0 = 20'b10010100000010000000, value1 = 20'b00000000000110000000, value2 = 20'b00000000001100000000, value3 = 20'b00110000000000000001, value4 = 20'b00000001000001000000, value5 = 20'b00101000000000000001, value6 = 20'b00000000001000000100, value7 = 20'b00000000000001010000,

value8 = 20'b01000000011000001010, value9 = 20'b10000000100000000000,

value10 = 20'b00000000001100000110, value11 = 20'b00000000000001000000,

value12 = 20'b00000010001111000000, value13 = 20'b01000000011000000010,

value14 = 20'b10000000110000000000, value15 = 20'b10000011100001100000,

value16 = 20'b10000011110001000000, value17 = 20'b10000011100001100000;

parameter AND=3'b000, OR=3'b001, ADD=3'b010, SUB=3'b110, NOR=3'b100, SLT=3'b111, XOR=3'b011, SRL=3'b101;

/\*

assign Rtype = ~|OP; //if OP=000000 then Rtype = 1assign LS = (OP == 6'b10x011) ? 1 : 0; //if OP=10x011 then LS = 1assign IBeq = (OP == 6'b000100) ? 1 : 0; //if OP=000100 then Ibeq = 1assign Jump = (OP == 6'b000010) ? 1 : 0; //if OP=000010 then Jump = 1

assign Load = (OP == 6'b100011) ? 1 : 0; //if OP=100011 then Load = 1

assign Store = (OP == 6'b101011) ? 1 : 0; //if OP=101011 then Store = 1

\*/

always @ (posedge clk or posedge reset)

if (reset==1) state <= IF;

else

case(state)

IF: if(MIO\_ready) state <= ID;

else state <= IF;

ID: case (Inst\_in[31:26])

6'b000000:

begin

case(Inst\_in[5:0])

6'b001000: state <= Jr; //Jr

6'b001001: state <= Jalr; //Jalr

default: state <= R\_Exc; //R-type OP

endcase

end

6'b100011: state <= Mem\_Ex; //Lw

6'b101011: state <= Mem\_Ex; //Sw

6'b001000: state <= I\_Exc; //Addi

6'b001100: state <= I\_Exc; //Andi

6'b001101: state <= I\_Exc; //Ori

6'b001110: state <= I\_Exc; //Xori

6'b001010: state <= I\_Exc; //Slti

6'b001111: state <= Lui\_Exc; //Lui

6'b000100: state <= Beq\_Exc; //Beq

6'b000101: state <= Bne\_Exc; //Bne

6'b000010: state <= J; //Jump

6'b000011: state <= Jal; //Jal

default: state <= Error;

endcase

Mem\_Ex: if(Inst\_in[29]) state <= Mem\_W;

else state <= Mem\_RD;

Mem\_RD: state <= LW\_WB;

LW\_WB: state <= IF;

Mem\_W: state <= IF;

R\_Exc: state <= R\_WB;

R\_WB: state <= IF;

I\_Exc: state <= I\_WB;

I\_WB: state <= IF;

Lui\_Exc: state <= IF;

Beq\_Exc: state <= IF;

Bne\_Exc: state <= IF;

Jal: state <= IF;

Jr: state <= IF;

J: state <= IF;

Error: state <= Error;

default: state <= Error;

endcase

always @ \* begin

case(state) //state

IF: `Datapath\_signals = value0;

ID: `Datapath\_signals = value1;

Mem\_Ex: `Datapath\_signals = value2;

Mem\_RD: `Datapath\_signals = value3;

LW\_WB: `Datapath\_signals = value4;

Mem\_W: `Datapath\_signals = value5;

R\_Exc: `Datapath\_signals = value6;

R\_WB: `Datapath\_signals = value7;

Beq\_Exc: `Datapath\_signals = value8;

J: `Datapath\_signals = value9;

I\_Exc: `Datapath\_signals = value10;

I\_WB: `Datapath\_signals = value11;

Lui\_Exc: `Datapath\_signals = value12;

Bne\_Exc: `Datapath\_signals = value13;

Jr: `Datapath\_signals = value14;

Jal: `Datapath\_signals = value15;

Jalr: `Datapath\_signals = value16;

default: `Datapath\_signals = value0;

endcase

end

always @ \* begin

case(ALUop)

2'b00: ALU\_operation = 3'b010; //add????

2'b01: ALU\_operation = 3'b110; //sub????

2'b10:

case (Inst\_in[5:0])

6'b100000: ALU\_operation = ADD;

6'b100010: ALU\_operation = SUB;

6'b100100: ALU\_operation = AND;

6'b100101: ALU\_operation = OR;

6'b100111: ALU\_operation = NOR;

6'b101010: ALU\_operation = SLT;

6'b000010: ALU\_operation = SRL; //shfit 1bit right

6'b000000: ALU\_operation = XOR;

default: ALU\_operation = ADD;

endcase

2'b11:

case (Inst\_in[31:26])

6'b001010: ALU\_operation = SLT; //slti

6'b001000: ALU\_operation = ADD; //addi

6'b001100: ALU\_operation = AND; //andi

6'b001101: ALU\_operation = OR; //ori

6'b001110: ALU\_operation = XOR; //xori

default: ALU\_operation = ADD;

endcase

endcase

end

endmodule