# ECEN651 Lab Report 7

## JR and JAL

JR instructions can be considered as a sub case in R-type instructions only activated when OPCODE matches. JAL instructions can be considered as a new case only when JAL signal is activated.

The first modification is in Hazard unit. The new diagram is shown below.

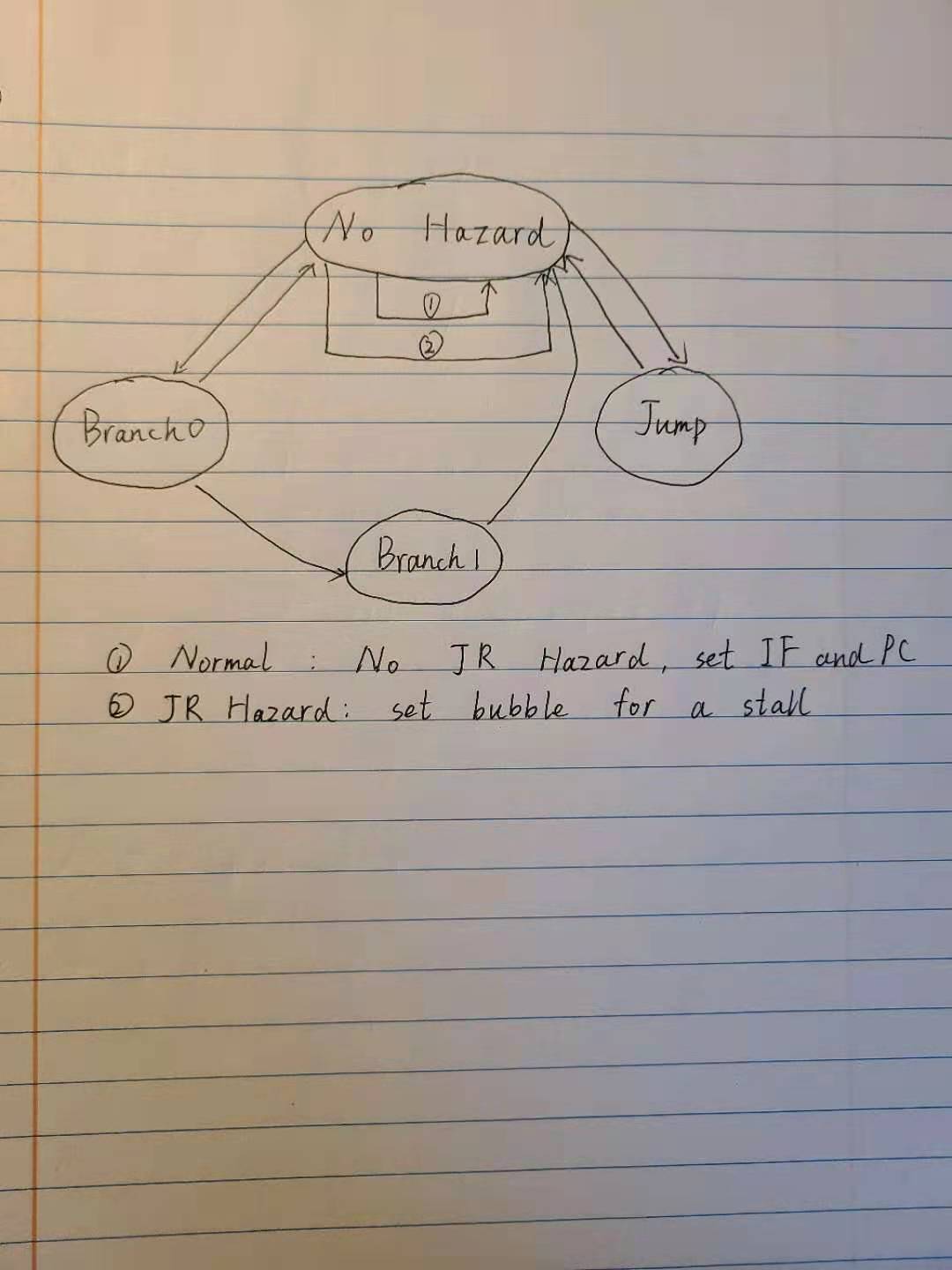


Fig 1.1 Hazard unit diagram

Hazard unit codes:

`timescale 1ns / 1ps

module HazardUnit(IF\_write, PC\_write, bubble, addrSel, Jump, Jr, Branch, ALUZero, memReadEX, ID\_Rs, ID\_Rt, EX\_Rw, MEM\_Rw, EX\_RegWrite, MEM\_RegWrite, UseShamt, UseImmed, Clk, Rst);

// Input and output ports

output reg IF\_write, PC\_write, bubble;

output reg [1:0] addrSel;

input Jump , Branch , ALUZero , memReadEX , Clk , Rst, Jr, EX\_RegWrite, MEM\_RegWrite ;

input [4:0] EX\_Rw, MEM\_Rw;

input UseShamt , UseImmed ;

input [4:0] ID\_Rs, ID\_Rt;

wire LdHazard, JrHazard;

reg [1:0] state, Next\_State;

// Load hazard, For R-type => if any register equal to load register

// For shift or I-type => if the register equal to the load inf

// Logic for detecting load hazard previous state

assign LdHazard = (((ID\_Rs == EX\_Rw && UseShamt != 1) || (ID\_Rt == EX\_Rw && UseImmed != 1)) && EX\_Rw != 0 && memReadEX == 1) ? 1 : 0;

// Jr hazard

assign JrHazard = ((Jr == 1) && ((EX\_Rw == ID\_Rs && EX\_RegWrite == 1)||(MEM\_Rw == ID\_Rs && MEM\_RegWrite == 1))) ? 1 : 0;

always @(\*) begin

// Check for hazard => Load, Branch, and Jump

case(state)

2'b00: begin

if ((LdHazard == 1) || (JrHazard == 1)) begin

Next\_State <= 2'b00;

{IF\_write, PC\_write, bubble} <= 3'b001;

addrSel <= 2'b00;

end

else if (Branch == 1) begin

Next\_State <= 2'b01;

{IF\_write, PC\_write, bubble} <= 3'b000;

addrSel <= 2'b00;

end

else if (Jump == 1) begin

Next\_State <= 2'b11;

{IF\_write, PC\_write, bubble} <= 3'b010;

addrSel <= 2'b01;

end

// NO Hazard

else begin

Next\_State <= 2'b00;

{IF\_write, PC\_write, bubble} <= 3'b110;

addrSel <= 2'b00;

end

end

// Branch 0

2'b01: begin

if (ALUZero == 1) begin

// If branch is taken

Next\_State <= 2'b10;

{IF\_write, PC\_write, bubble} <= 3'b011;

addrSel <= 2'b10;

end

else begin

// If branch NOT taken

Next\_State <= 2'b00;

{IF\_write, PC\_write, bubble} <= 3'b111;

addrSel <= 2'b00;

end

end

// Branch 1

2'b10: begin

Next\_State <= 2'b00;

{IF\_write, PC\_write, bubble} <= 3'b111;

addrSel <= 2'b00;

end

// Jump

2'b11: begin

Next\_State <= 2'b00;

{IF\_write, PC\_write, bubble} <= 3'b111;

addrSel <= 2'b00;

end

default: begin

Next\_State <= 2'b00;

{IF\_write, PC\_write, bubble} <= 3'b110;

addrSel <= 2'b00;

end

endcase

end

// State is changed in negedge of the clock

always @(negedge Clk) begin

// If reset

if(Rst == 0)

state <= 2'b00;

else

state <= Next\_State;

end

endmodule

Besides, control unit needs to activate JAL and JR signals. For JAL case, it acts like a jump instruction, but the combined address is replaced by the output of ALU this time. For JR case, a new mux is required, new PC selects from next PC or output of register A.

The control unit codes are shown below.

`timescale 1ns / 1ps

// Define Opcodes

`define RTYPEOPCODE 6'b000000

`define LWOPCODE 6'b100011

`define SWOPCODE 6'b101011

`define BEQOPCODE 6'b000100

`define JOPCODE 6'b000010

`define ORIOPCODE 6'b001101

`define ADDIOPCODE 6'b001000

`define ADDIUOPCODE 6'b001001

`define ANDIOPCODE 6'b001100

`define LUIOPCODE 6'b001111

`define SLTIOPCODE 6'b001010

`define SLTIUOPCODE 6'b001011

`define XORIOPCODE 6'b001110

`define JALOPCODE 6'b000011

// Define Function codes (used for R-type instructions)

`define SLLFunc 6'b000000

`define SRLFunc 6'b000010

`define SRAFunc 6'b000011

`define ADDFunc 6'b100000

`define ADDUFunc 6'b100001

`define SUBFunc 6'b100010

`define SUBUFunc 6'b100011

`define ANDFunc 6'b100100

`define ORFunc 6'b100101

`define XORFunc 6'b100110

`define NORFunc 6'b100111

`define SLTFunc 6'b101010

`define SLTUFunc 6'b101011

`define JRFunc 6'b001000

// Define ALU operation codes

`define AND 4'b0000

`define OR 4'b0001

`define ADD 4'b0010

`define SLL 4'b0011

`define SRL 4'b0100

`define SUB 4'b0110

`define SLT 4'b0111

`define ADDU 4'b1000

`define SUBU 4'b1001

`define XOR 4'b1010

`define SLTU 4'b1011

`define NOR 4'b1100

`define SRA 4'b1101

`define LUI 4'b1110

module PipelinedControl(Jump, Branch, RegDst, UseImmed, UseShamt, Jal,RegSrc,RegWrite, MemRead, MemWrite, SignExtend, ALUOp, JumpSel, OpCode, FuncCode);

// Define inputs and outputs

input [5:0] OpCode;

input [5:0] FuncCode;

output reg UseShamt, UseImmed;

output reg [3:0] ALUOp;

output reg RegDst, RegSrc, RegWrite, MemRead, MemWrite, Branch, Jump, JumpSel, Jal, SignExtend;

always @(\*) begin

case(OpCode)

`RTYPEOPCODE: begin

// If R-Type instruction

ALUOp = 4'b1111;

case(FuncCode)

// Shift Instruction

`SLLFunc: begin

{RegDst, UseShamt,UseImmed, RegSrc, RegWrite, MemRead, MemWrite, Branch, Jump, SignExtend} = 10'b110010000x;

{JumpSel, Jal} = 2'b00;

end

// Shift Instruction

`SRLFunc: begin

{RegDst, UseShamt,UseImmed, RegSrc, RegWrite, MemRead, MemWrite, Branch, Jump, SignExtend} = 10'b110010000x;

{JumpSel, Jal} = 2'b00;

end

// Shift Instruction

`SRAFunc: begin

{RegDst, UseShamt,UseImmed, RegSrc, RegWrite, MemRead, MemWrite, Branch, Jump, SignExtend} = 10'b110010000x;

{JumpSel, Jal} = 2'b00;

end

`JRFunc: begin

{RegDst, UseShamt,UseImmed, RegSrc, RegWrite, MemRead, MemWrite, Branch, Jump, SignExtend} = 10'b0001000010;

{JumpSel, Jal} = 2'b10;

end

//R type instruction that are not shift

default: begin

{RegDst, UseShamt,UseImmed, RegSrc, RegWrite, MemRead, MemWrite, Branch, Jump, SignExtend} = 10'b100010000x;

{JumpSel, Jal} = 2'b00;

end

endcase

end

// Load opcode

`LWOPCODE: begin

ALUOp = `ADD;

{RegDst, UseShamt,UseImmed, RegSrc, RegWrite, MemRead, MemWrite, Branch, Jump, SignExtend} = 10'b0011110001;

{JumpSel, Jal} = 2'b00;

end

// Store opcode

`SWOPCODE: begin

{RegDst, UseShamt,UseImmed, RegSrc, RegWrite, MemRead, MemWrite, Branch, Jump, SignExtend} = 10'bx01x001001;

ALUOp = `ADD;

{JumpSel, Jal} = 2'b00;

end

// Branch if equal opcode

`BEQOPCODE: begin

{RegDst, UseShamt,UseImmed, RegSrc, RegWrite, MemRead, MemWrite, Branch, Jump, SignExtend} = 10'bx00x000101;

ALUOp = `SUB;

{JumpSel, Jal} = 2'b00;

end

// Jump opcode

`JOPCODE: begin

{RegDst, UseShamt,UseImmed, RegSrc, RegWrite, MemRead, MemWrite, Branch, Jump, SignExtend} = 10'bx00x00001x;

ALUOp = 4'bxxxx;

{JumpSel, Jal} = 2'b00;

end

// Or immediate opcode

`ORIOPCODE: begin

{RegDst, UseShamt,UseImmed, RegSrc, RegWrite, MemRead, MemWrite, Branch, Jump, SignExtend} = 10'b0010100000;

ALUOp = `OR;

{JumpSel, Jal} = 2'b00;

end

// Add immediate opcode

`ADDIOPCODE: begin

{RegDst, UseShamt,UseImmed, RegSrc, RegWrite, MemRead, MemWrite, Branch, Jump, SignExtend} = 10'b0010100001;

ALUOp = `ADD;

{JumpSel, Jal} = 2'b00;

end

// Add immediate unsigned opcode

`ADDIUOPCODE: begin

{RegDst, UseShamt,UseImmed, RegSrc, RegWrite, MemRead, MemWrite, Branch, Jump, SignExtend} = 10'b0010100000;

ALUOp = `ADDU;

{JumpSel, Jal} = 2'b00;

end

// AND bitwise with immediate opcode

`ANDIOPCODE: begin

{RegDst, UseShamt,UseImmed, RegSrc, RegWrite, MemRead, MemWrite, Branch, Jump, SignExtend} = 10'b0010100000;

ALUOp = `AND;

{JumpSel, Jal} = 2'b00;

end

// Load Upper Immediate opcode

`LUIOPCODE: begin

{RegDst, UseShamt,UseImmed, RegSrc, RegWrite, MemRead, MemWrite, Branch, Jump, SignExtend} = 10'b0010100000;

ALUOp = `LUI;

{JumpSel, Jal} = 2'b00;

end

// Set less than Immediate opcode

`SLTIOPCODE: begin

{RegDst, UseShamt,UseImmed, RegSrc, RegWrite, MemRead, MemWrite, Branch, Jump, SignExtend} = 10'b0010100001;

ALUOp = `SLT;

{JumpSel, Jal} = 2'b00;

end

// Set less than Immediate unsigned opcode

`SLTIUOPCODE: begin

{RegDst, UseShamt,UseImmed, RegSrc, RegWrite, MemRead, MemWrite, Branch, Jump, SignExtend} = 10'b0010100000;

ALUOp = `SLTU;

{JumpSel, Jal} = 2'b00;

end

// XOR bitwise Immediate opcode

`XORIOPCODE: begin

{RegDst, UseShamt,UseImmed, RegSrc, RegWrite, MemRead, MemWrite, Branch, Jump, SignExtend} = 10'b0010100000;

ALUOp = `XOR;

{JumpSel, Jal} = 2'b00;

end

// Jump and Link

`JALOPCODE:begin

{RegDst, UseShamt,UseImmed, RegSrc, RegWrite, MemRead, MemWrite, Branch, Jump, SignExtend} = 10'b0000100010;

ALUOp = 4'bxxxx;

{JumpSel, Jal} = 2'b01;

end

default: begin

{RegDst, UseShamt,UseImmed, RegSrc, RegWrite, MemRead, MemWrite, Branch, Jump, SignExtend} = 10'b0000000000;

ALUOp = 4'b0000;

{JumpSel, Jal} = 2'b00;

end

endcase

end

endmodule

## Static Branch Prediction

In this section, we are going to implement a prediction about branch taken or not taken. By using this the biggest change is in Hazard unit. We can use ALUZero to detect if this branch is taken or not. Thus the new diagram is shown below.

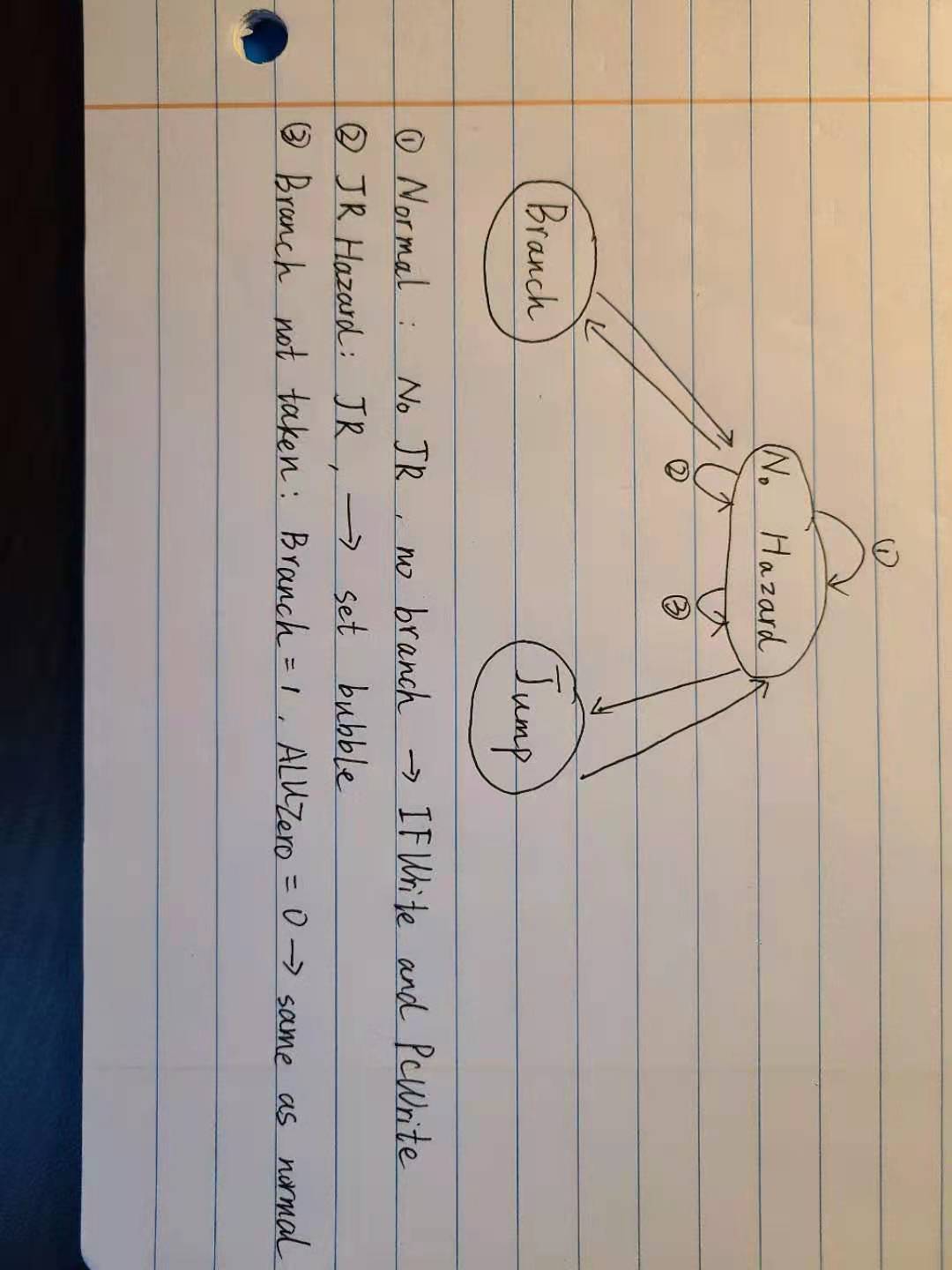


Fig 2.1 Hazard unit diagram

Hazard unit codes are shown below.

// 00 - Normal State, 01 - Branch, 11 - Jump

`timescale 1ns / 1ps

module HazardUnit(IF\_write, PC\_write, bubble, addrSel, Jump, Branch, ALUZero, memReadEX, currRs, currRt, prevRt, UseShamt, UseImmed , Clk , Rst, Jr, EX\_RegWrite, MEM\_RegWrite, EX\_Rw, MEM\_Rw);

// Input and output ports

output reg IF\_write, PC\_write, bubble;

output reg [1:0] addrSel;

input Jump , Branch , ALUZero , memReadEX , Clk , Rst, Jr, EX\_RegWrite, MEM\_RegWrite ;

input [4:0] EX\_Rw, MEM\_Rw;

input UseShamt , UseImmed ;

input [4:0] currRs, currRt, prevRt;

wire LdHazard, JrHazard;

reg [1:0] state, Next\_State;

// Logic for detecting load hazard

assign LdHazard = (memReadEX == 1) & (prevRt != 0)

& (((currRs == prevRt || currRt == prevRt) & UseShamt == 0 & UseImmed == 0) || (UseShamt == 1 & currRs == prevRt) ||(UseImmed == 1 & currRs == prevRt ))?1:0 ;

// Check Jr\_hazard

assign JrHazard = ((Jr == 1) && ((EX\_Rw == currRs && EX\_RegWrite == 1)||(MEM\_Rw == currRs && MEM\_RegWrite == 1))) ? 1:0;

always @(\*) begin

// Check for hazard

case(state)

2'b00: begin

// Load Hazard

if ((LdHazard == 1) || (JrHazard == 1)) begin

Next\_State <= 2'b00;

{IF\_write, PC\_write, bubble} <= 3'b001;

addrSel <= 2'b00;

end

// Branch

else if (Branch == 1) begin

if (ALUZero == 1) begin

Next\_State <= 2'b01;

{IF\_write, PC\_write, bubble} <= 3'b011;

addrSel <= 2'b10;

end

else begin

// Branch NOT taken

Next\_State <= 2'b00;

{IF\_write, PC\_write, bubble} <= 3'b110;

addrSel <= 2'b00;

end

end

// Jump Hazard

else if (Jump == 1) begin

Next\_State <= 2'b11;

{IF\_write, PC\_write, bubble} <= 3'b010;

addrSel <= 2'b01;

end

// NO Hazard

else begin

Next\_State <= 2'b00;

{IF\_write, PC\_write, bubble} <= 3'b110;

addrSel <= 2'b00;

end

end

// Branch

2'b01: begin

Next\_State <= 2'b00;

{IF\_write, PC\_write, bubble} <= 3'b110;

addrSel <= 2'b00;

end

// Jump

2'b11: begin

Next\_State <= 2'b00;

{IF\_write, PC\_write, bubble} <= 3'b111;

addrSel <= 2'b00;

end

default: begin

Next\_State <= 2'b00;

{IF\_write, PC\_write, bubble} <= 3'b110;

addrSel <= 2'b00;

end

endcase

end

// State is changed in negedge of the clock

always @(negedge Clk) begin

// If reset = 0, then keep state at 0

if(Rst == 0)

state <= 2'b00;

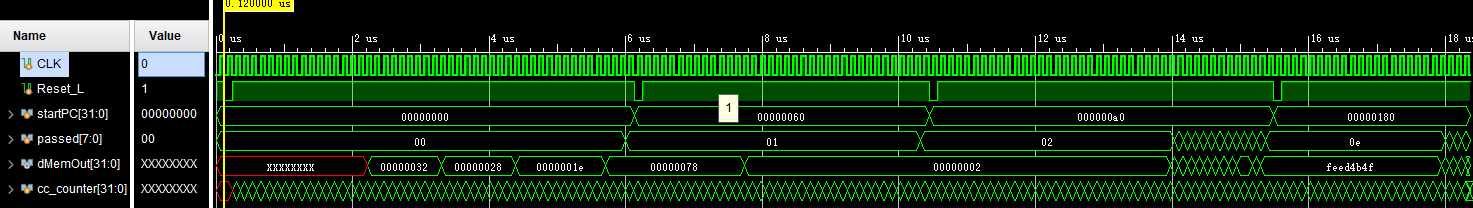
else

state <= Next\_State;

end

endmodule

## Test Results



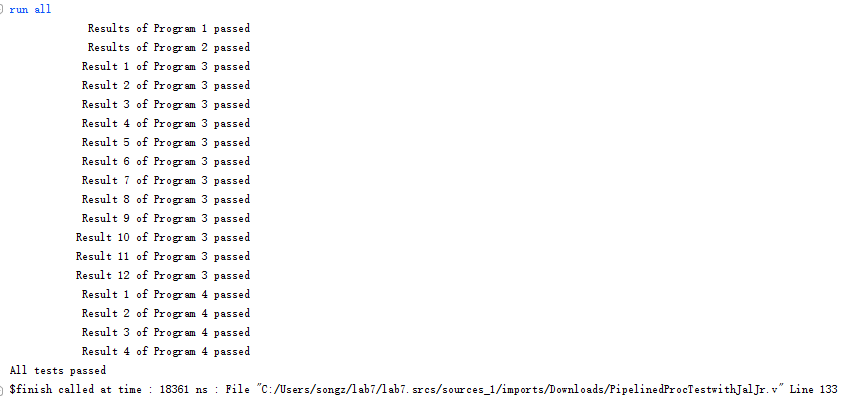
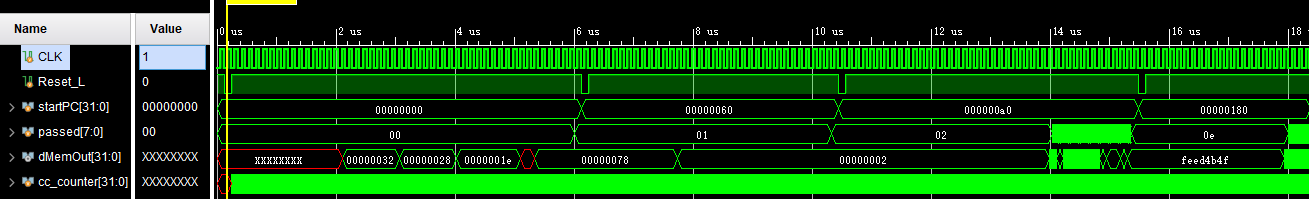


Fig 3.1 Part1 result



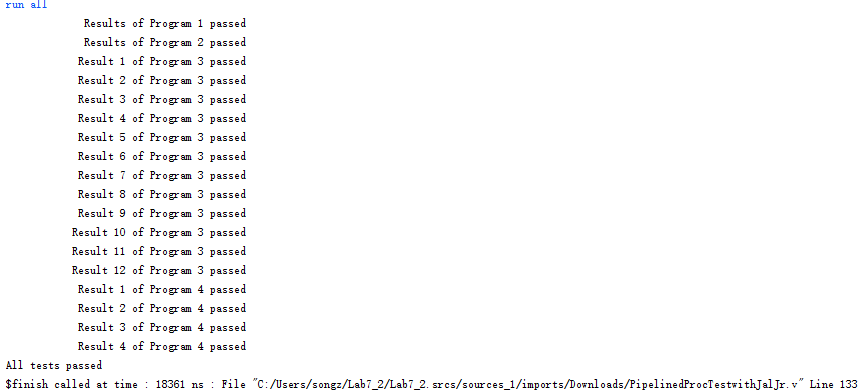


Fig 3.2 Test result part 2

## Questions

1. **Compare the synthesis report from last lab with the one generated during Part 1 of this lab. Did the clock rate of your design change at all? Why or why not? Please explain the results! Can you think of any improvements that could be made to the current design?**

Clock rates of the last lab and this lab are both 120ns. I assume the reason is that the clock rate is a simulated clock rate by Vivado. It merely represents the compared running rate of current design. The influence of improvement in this lab does not change the synthesized clock rate.

An improvement can be applied is that Jump instructions are still handled in EX stage, if they can be resolved in IF stage some stalls can be removed.

1. **Recompute the CPI of your processor executing Program 1. Did it change? Why or why not?**

For the same clock rate, the running time for last lab was 5761 ns, this lab is 6001ns. I assume the worse performance comes from new-added flushing procedure.

**(c) Recompute the average CPI of your processor executing all programs including the new program for testing jal and jr. Compare the new CPI with that of last lab. Explain the changes.**

Program1: 33

Program2: 11

Program3: 37

Program4: 18

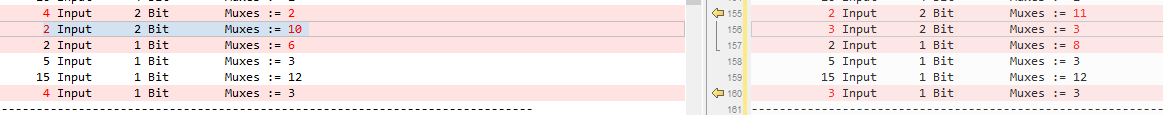
Total: 99

Running time 18361/120 = 153 cycles

CPI = 153/99 = 1.54

In the last lab CPI was 1.58. The difference came from the last program with a lot of jump instructions in it. The average CPI was drawn down.

**(d)Compare the synthesis report from the last part of lab to that of the first part of lab. Was there a cost in terms of resource utilization for the improved design? If so, what was it? Explain your results.**



The number of MUXes are changed, in the left(first part) 4 input muxes are mainly used. In the last part, 2 input muxes are instead. This way utilization is improved.

**(e) Examine the loop in Program 1. What characteristic does it have that makes static branch prediction worth while? Describe what would happen to the CPI of our processor executing Program 1 if the number of loop iterations were to increase.**

In the loop, with branch static prediction, only the last branch will be taken. The former branches can be predicted as not taken. Thus, if the loop times increase, the CPI will drop, which means closer to 1.