**ECEN 323 – Winter 2020**

Lab 8: Pipelined RISC-V

Johnson, Ryan

Section 1

Preliminary

Example #1

sub x2,x1,x3

NOP

NOP

and x12,x2,x5

or x13,x6,x2

NOP

add x14,x12,x2

NOP

NOP

sw x14,100(x13)

Example #2

ld x2, 20(x1)

NOP

NOP

and x4,x2,x5

or x8,x2,x6

NOP

and x9,x4,x2

sub x1,x6,x7

Example #3

Note that if you insert NOP instructions you will need to update the branch offset in the instruction.

beq x1, x0, 16

and x12,x2,x5 => NOP

or x13, x6, x2 => NOP

add x14,x2,x2 => NOP

ld x4,100(x7)

Exercise #1

**Instruction Fetch (IF)**

**Instruction Decode (ID)**

**Execute (EX)**

**Memory Access (MEM)**

**Write-Back (WB)**

Exercise #2

Provide a copy of your working RISC-V pipelined processor

`timescale 1ns / 1ps

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* Module: riscv\_basic\_pipeline

\*

\* Author: Ryan Johnson

\* Class: ECEN 323, Section 01, Winter 2020

\* Date: 24 Feb 2020

\*

\* Description: pipelines the control and datapath instructions

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

module riscv\_basic\_pipeline #(parameter INITIAL\_PC = 32'h00400000) (

input wire logic clk,

input wire logic rst,

output logic [31:0] PC,

input wire logic [31:0] instruction,

output logic [31:0] ALUResult,

output logic [31:0] dAddress,

output logic [31:0] dWriteData,

input wire logic [31:0] dReadData,

output logic MemRead,

output logic MemWrite,

output logic [31:0] WriteBackData

);

logic [31:0] RF[31:0];

integer i; // i needs to be declared before it is used

initial

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

RF[i] = 0; // initializes each register to 0

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

// IF: Instruction Fetch

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

logic [31:0] if\_PC;

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

// ID: Instruction Decode

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

logic [31:0] id\_PC;

logic [3:0] id\_ALUCtrl;

logic id\_ALUSrc;

logic id\_MemWrite;

logic id\_MemRead;

logic id\_Branch;

logic id\_RegWrite;

logic id\_MemtoReg;

logic store;

logic load;

logic branch;

logic itype;

logic [6:0] opcode;

logic [2:0] funct3;

localparam AND = 4'b0000;

localparam OR = 4'b0001;

localparam ADD = 4'b0010;

localparam SUB = 4'b0110;

localparam SLT = 4'b0111;

localparam XOR = 4'b1100;

logic [31:0] id\_data1;

logic [31:0] id\_data2;

logic [31:0] id\_imm;

logic [31:0] immI;

logic [31:0] immS;

localparam SOpCode = 7'b0100011;

logic [4:0] id\_rs1;

logic [4:0] id\_rs2;

logic [4:0] id\_rd;

logic [31:0] id\_branch\_offset;

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

// EX: Execute

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

logic [31:0] ex\_PC;

logic ex\_Zero;

logic [3:0] ex\_ALUCtrl;

logic ex\_ALUSrc;

logic ex\_MemWrite;

logic ex\_MemRead;

logic ex\_Branch;

logic ex\_RegWrite;

logic ex\_MemtoReg;

logic [4:0] ex\_rd;

logic [31:0] ex\_data1;

logic [31:0] ex\_data2;

logic [31:0] ex\_imm;

logic [31:0] data2;

logic [31:0] ex\_aluResult;

logic [31:0] ex\_branch\_offset;

logic [31:0] ex\_branch\_target;

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

// MEM: Memory Access

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

logic [31:0] mem\_branch\_target;

logic mem\_PCSrc;

logic mem\_Zero;

logic mem\_MemWrite;

logic mem\_MemRead;

logic mem\_Branch;

logic mem\_RegWrite;

logic mem\_MemtoReg;

logic [4:0] mem\_rd;

logic [31:0] mem\_write\_data;

logic [31:0] mem\_aluResult;

logic [31:0] mem\_read\_data;

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

// WB: Write-Back

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

logic wb\_RegWrite;

logic wb\_MemtoReg;

logic [31:0] wb\_WriteData;

logic [31:0] wb\_read\_data;

logic [31:0] wb\_aluResult;

logic [4:0] wb\_rd;

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

// IF: Instruction Fetch

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

//PC signal update

always\_ff@ (posedge clk) begin

if (rst) if\_PC <= INITIAL\_PC;

else if (mem\_PCSrc) if\_PC <= mem\_branch\_target;

else if\_PC <= if\_PC + 4;

end

assign PC = if\_PC;

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

// ID: Instruction Decode

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

//IF to ID PC signal

always\_ff@ (posedge clk) begin

if (rst) id\_PC <= 0;

else id\_PC <= if\_PC;

end

assign opcode = instruction[6:0];

assign funct3 = instruction[14:12];

assign store = (opcode == 7'b0100011) ? 1'b1 : 1'b0;

assign load = (opcode == 7'b0000011) ? 1'b1 : 1'b0;

assign branch = (opcode == 7'b1100011) ? 1'b1 : 1'b0;

assign itype = (opcode == 7'b0010011) ? 1'b1 : 1'b0;

// Sets control signals based on instruction values

always\_comb begin

id\_ALUCtrl = (funct3 == 3'b111) ? AND :

(funct3 == 3'b110) ? OR :

(funct3 == 3'b100) ? XOR :

(store || load) ? ADD :

(funct3 == 3'b010) ? SLT :

(branch) ? SUB :

(itype) ? ADD :

(instruction[30] == 1'b1) ? SUB : ADD;

id\_ALUSrc = itype | load | store;

id\_MemtoReg = load;

id\_MemRead = load;

id\_Branch = branch;

id\_MemWrite = store;

id\_RegWrite = (!branch & !store);

end

assign immI = {{20{instruction[31]}},instruction[31:20]};

assign immS = {{20{instruction[31]}},instruction[31:25],instruction[11:7]};

assign id\_imm = (instruction[6:0] == SOpCode) ? immS : immI;

assign id\_rd = instruction[11:7];

assign id\_rs1 = instruction[19:15];

assign id\_rs2 = instruction[24:20];

// Sets data1 and data2 to the values in the specified registers

always\_comb begin

id\_data1 <= RF[id\_rs1];

id\_data2 <= RF[id\_rs2];

if (wb\_RegWrite) begin

if (id\_rs1 == wb\_rd) begin

if (wb\_rd == 0) id\_data1 <= 0;

else id\_data1 <= wb\_WriteData;

end

if (id\_rs2 == wb\_rd) begin

if (wb\_rd == 0) id\_data2 <= 0;

else id\_data2 <= wb\_WriteData;

end

end

end

// synchronous write and read with "write first" mode

always\_ff@ (posedge clk) begin

if (wb\_RegWrite && (wb\_rd != 0)) RF[wb\_rd] <= wb\_WriteData;

end

assign id\_branch\_offset = {{20{instruction[31]}},instruction[7],

instruction[30:25],instruction[11:8],1'b0};

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

// EX: Execute

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

//ID to EX PC signal

always\_ff@ (posedge clk) begin

if (rst) ex\_PC <= 0;

else ex\_PC <= id\_PC;

end

//ID to EX control signals

always\_ff@ (posedge clk) begin

if (rst) begin

ex\_ALUCtrl <= 0;

ex\_ALUSrc <= 0;

ex\_MemWrite <= 0;

ex\_MemRead <= 0;

ex\_Branch <= 0;

ex\_RegWrite <= 0;

ex\_MemtoReg <= 0;

end

else if (mem\_PCSrc) begin

ex\_MemWrite <= 0;

ex\_MemRead <= 0;

ex\_Branch <= 0;

ex\_RegWrite <= 0;

ex\_MemtoReg <= 0;

end

else begin

ex\_ALUCtrl <= id\_ALUCtrl;

ex\_ALUSrc <= id\_ALUSrc;

ex\_MemWrite <= id\_MemWrite;

ex\_MemRead <= id\_MemRead;

ex\_Branch <= id\_Branch;

ex\_RegWrite <= id\_RegWrite;

ex\_MemtoReg <= id\_MemtoReg;

end

end

//ID to EX rd signal

always\_ff@ (posedge clk) begin

if (rst) ex\_rd <= 0;

else ex\_rd <= id\_rd;

end

//ID to EX branch\_offset signal

always\_ff@ (posedge clk) begin

if (rst) ex\_branch\_offset <= 0;

else ex\_branch\_offset <= id\_branch\_offset;

end

//ID to EX data1 data2 imm signals

always\_ff@ (posedge clk) begin

if (rst) begin

ex\_data1 <= 0;

ex\_data2 <= 0;

ex\_imm <= 0;

end

else begin

ex\_data1 <= id\_data1;

ex\_data2 <= id\_data2;

ex\_imm <= id\_imm;

end

end

assign data2 = (ex\_ALUSrc) ? ex\_imm : ex\_data2;

assign ex\_aluResult = (ex\_ALUCtrl == AND) ? ex\_data1 & data2 :

(ex\_ALUCtrl == OR) ? ex\_data1 | data2 :

(ex\_ALUCtrl == ADD) ? ex\_data1 + data2 :

(ex\_ALUCtrl == SUB) ? ex\_data1 - data2 :

(ex\_ALUCtrl == SLT) ? ($signed(ex\_data1) < $signed(data2)) :

(ex\_ALUCtrl == XOR) ? ex\_data1 ^ data2 : 0;

assign ex\_Zero = (ex\_aluResult == 0) ? 1'b1 : 1'b0;

assign ex\_branch\_target = ex\_PC + ex\_branch\_offset;

assign ALUResult = ex\_aluResult;

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

// MEM: Memory Access

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

//EX to MEM control signals

always\_ff@ (posedge clk) begin

if (rst | mem\_PCSrc) begin

mem\_MemWrite <= 0;

mem\_MemRead <= 0;

mem\_Branch <= 0;

mem\_RegWrite <= 0;

mem\_MemtoReg <= 0;

end

else begin

mem\_MemWrite <= ex\_MemWrite;

mem\_MemRead <= ex\_MemRead;

mem\_Branch <= ex\_Branch;

mem\_RegWrite <= ex\_RegWrite;

mem\_MemtoReg <= ex\_MemtoReg;

end

end

//EX to MEM rd signal

always\_ff@ (posedge clk) begin

if (rst) mem\_rd <= 0;

else mem\_rd <= ex\_rd;

end

//EX to MEM Zero and branch\_target signals

always\_ff@ (posedge clk) begin

if (rst) begin

mem\_Zero <= 0;

mem\_branch\_target <= 0;

end

else begin

mem\_Zero <= ex\_Zero;

mem\_branch\_target <= ex\_branch\_target;

end

end

//EX to MEM write\_data signal

always\_ff@ (posedge clk) begin

if (rst) mem\_write\_data <= 0;

else mem\_write\_data <= ex\_data2;

end

//EX to MEM aluResult signal

always\_ff@ (posedge clk) begin

if (rst) mem\_aluResult <= 0;

else mem\_aluResult <= ex\_aluResult;

end

assign mem\_PCSrc = mem\_Zero & mem\_Branch;

assign MemWrite = mem\_MemWrite;

assign MemRead = mem\_MemRead;

assign dAddress = mem\_aluResult;

assign dWriteData = mem\_write\_data;

assign mem\_read\_data = dReadData;

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

// WB: Write-Back

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

//MEM to WB control signals

always\_ff@ (posedge clk) begin

if (rst) begin

wb\_RegWrite <= 0;

wb\_MemtoReg <= 0;

end

else begin

wb\_RegWrite <= mem\_RegWrite;

wb\_MemtoReg <= mem\_MemtoReg;

end

end

//MEM to WB aluResult

always\_ff@ (posedge clk) begin

if (rst) wb\_aluResult <= 0;

else wb\_aluResult <= mem\_aluResult;

end

assign wb\_read\_data = dReadData;

assign wb\_WriteData = (wb\_MemtoReg) ? wb\_read\_data : wb\_aluResult;

//MEM to WB wb\_rd signal

always\_ff@ (posedge clk) begin

if (rst) wb\_rd <= 0;

else wb\_rd <= mem\_rd;

end

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

// Top-Level Ports //

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

assign WriteBackData = wb\_WriteData;

endmodule

Provide the last two lines of your simulation testbench console to demonstrate that you have passed this lab

You Passed!

$finish called at time : 795 ns : File "J:/ECEN 323 Labs/project\_8/riscv\_pipeline\_tb.v" Line 156

Exercise #3

Summarize and justify any warnings you had during synthesis. If you did not have any warnings, say such in your laboratory report.

The only warning was an unused sequential element ex\_MemtoReg register for when reset puts it back to zero. It think it is not important whether this is set to zero, because it only matters when RegWrite is 1, which in the case of a reset, it is not.

Summarize the estimated resources for your synthesized logic in the table below.

| **Resource** | **Estimation** |
| --- | --- |
| LUT | 475 |
| FF | 327 |
| IO | 228 |
| BUFG | 1 |

Contrast the size of your pipelined processor (in terms of FF and LUTRAM) against that of your single-cycle processor from the previous lab. If they are noticeably different describe why you think they are different.

There are 475 LUTs and 327 FFs while there were only 317 LUTs and 32 FFs in my single cycle design. The reason for the large differences is that there are many stages and needs to have flipflops to hold values for future stages, multiplying that number by more than 10, and lookup tables also go into that implementation so there are more of them.

How many hours did you work on the lab?

7 hours

Please provide any suggestions for improving this lab in the future:

None. Great stuff.