ENG 463-L1

Lab #3 ARM Pipeline Implementation with Forwarding and Hazard Detection

Brian Worts and Chris Jenson

04/06/21



**Table of Contents**

Introduction………………………………………………………………………………..……. 2

Problem Statement…...…………………………………………………………………………. 2

Procedure/Results…...……...…...…………………………………………………………….... 4

Discussion………………………………………………………………………………………..7

Conclusion ................................................................................................................................... 7

Appendix A (Arm Assembly)....................................................................................................... 8

Appendix B (Initial Memory)....................................................................................................... 8

Appendix C (Test Bench Code).................................................................................................... 9

Appendix D (Verilog Files)........................................................................................................... 11

**Introduction:**

For this lab, the pipelined ARM architecture processor from Lab 2 was expanded upon with the addition of forwarding and hazard detection using as high as possible level of abstraction design methodology. Pipelining is the process of fetching the next instruction while the current instruction is being executed. When instructions like LDUR and STUR occur, and they change registers required by future instructions, the potential for hazards arise. The concern is that, without detection, the registers will not contain the desired values as determined by earlier instructions. This problem is solved by adding detection and forwarding values in the event a hazard is detected. ARM processors are based on the RISC (reduced instruction set computer) architecture to perform a smaller set of instructions at high speeds. They provide high performance while requiring considerably less power than devices using CISC (complex instruction set computing). By having less instructions, they require fewer transistors allowing for small sizes of the integrated circuitry. The ARM processor’s smaller size, low complexity, and lower power usage make them ideal for small devices.

**Problem Statement:**

Implement the pipelined ARM architecture developed in Lab 2 augmented to include a Forwarding Unit and a Hazard Detection Unit using an as high as possible level of abstract design methodology. The architecture will be designed based on *Figure 1*. The memories, as well as the register file, are of the asynchronous read, synchronous write type. The control is based on *Figures 2 & 3*. The design must be verified using a testbench.

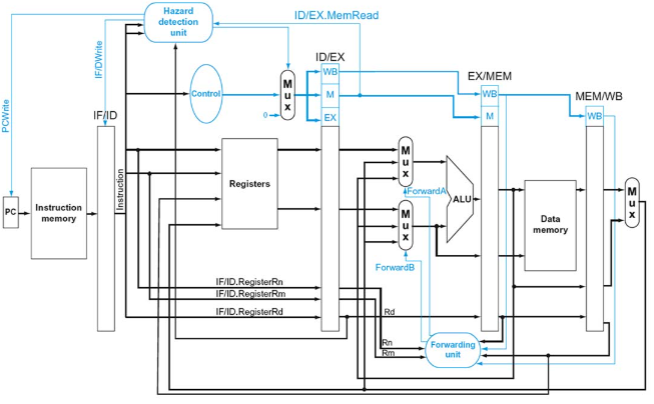


Figure 1: Overall Architecture

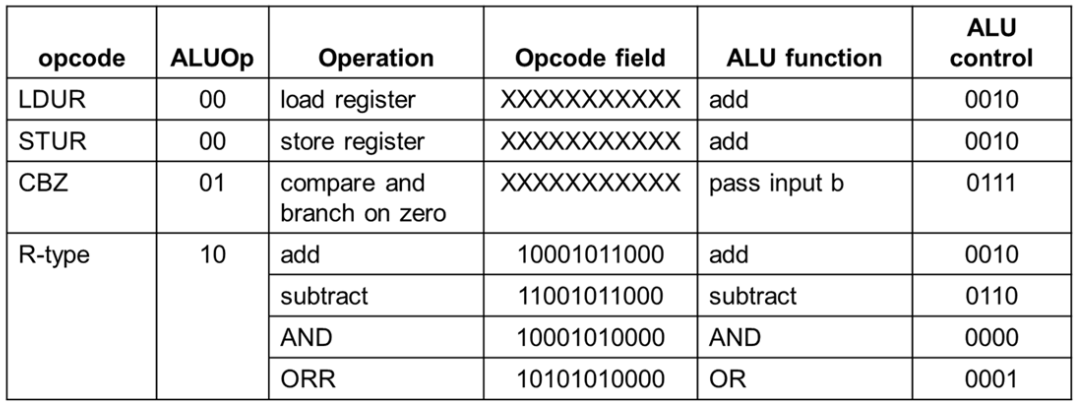


Figure 2: ALU Control Truth Table

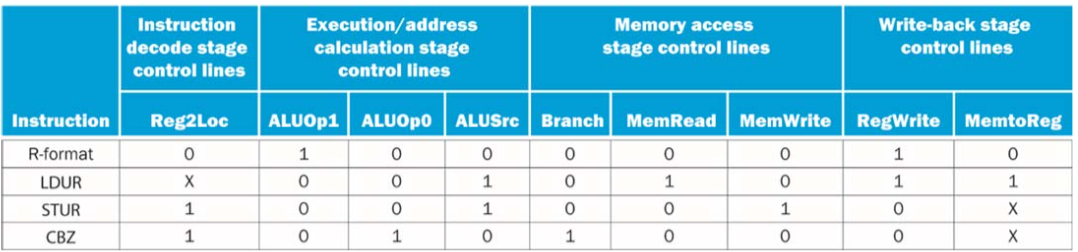


Figure 3: Main Control Truth Table

**Procedure/Results:**

The group started by examining Lab 2 where a pipelined version of the processor was made. Most of the blocks were the same except for the controls and the addition of hazard detection and forwarding, as well as some simplifications such as removal of the sign extend module. The instructions were chosen to create a comprehensive test of the system, specifically by adding hazards, and were originally written in Arm Assembly Language:

LDUR X2, [X1, #0]

AND X3, X5, X2

ORR X4, X4, X2

ADD X5, X4, X2

**Expected Results With Hazard Detection Unit and Forwarding Unit:**

X2 = 1

X3 = 1

X4 = 5

X5 = 6

With the inputs, outputs, instructions, and the various modules from the Lab 2 design already implemented, it was time to implement the new blocks. The first of these blocks to be implemented was Hazard Detection. This comes first because the processor has to know when there is a hazard to know when to create a “bubble”. Hazard Detection can be done by looking at the instruction and the registers it uses. When an instruction that occurs and there is an upcoming instruction that alters the same register, there is a potential hazard.

Now with hazard detection implemented, it was possible to implement the Forwarding Unit. The Forwarding Unit forwards the data ahead of the current cycle to an upcoming instruction so that the new instruction won’t look to an out of date value. This block changes the Mux selectors that send data to the ALU to ensure the correct data is inputted to the ALU. The simulation was run with a 100MHz clock signal. The results are shown below in Figures 4-7.

In Figures 4-7, the lavender colored dividers represent the different sections of the pipeline and the teal colored dividers represent modules inside of a section, when it was helpful to be able to differentiate values. Other values are highlighted as necessary.

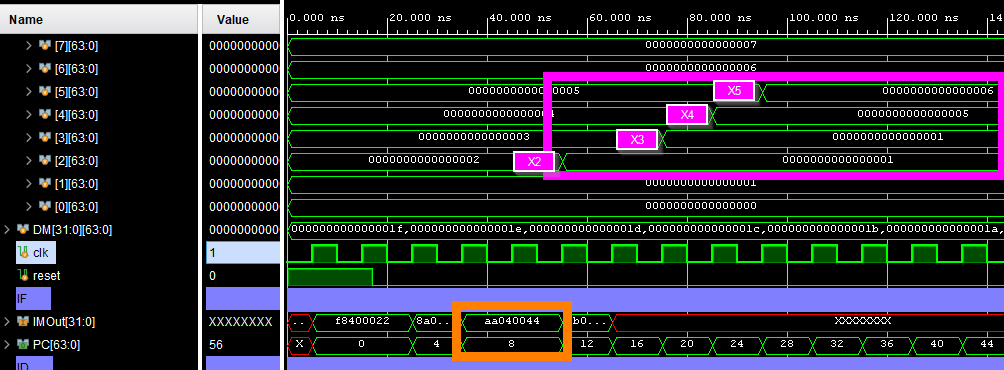


Figure 4: As shown, all registers get the expected values and the orange square shows the bubble properly working, as the third instruction sits in the IF section for 2 clock cycles. The figure shows Registers and IF Stage

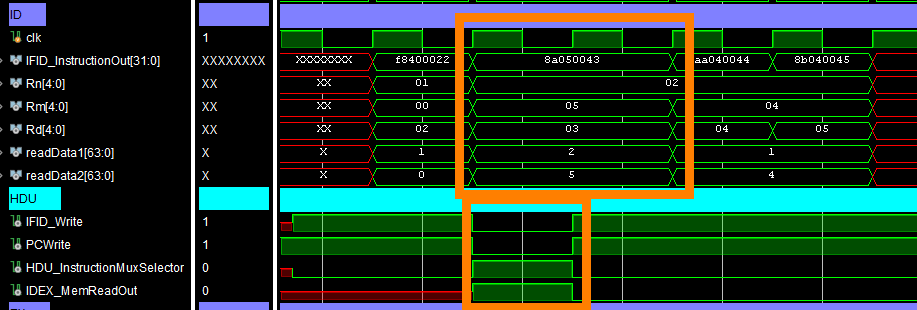
****

Figure 5: Shows the AND instruction (instruction 2) staying in the ID stage for 2 cycles as well as outputs of the HDU. Image is ID Stage

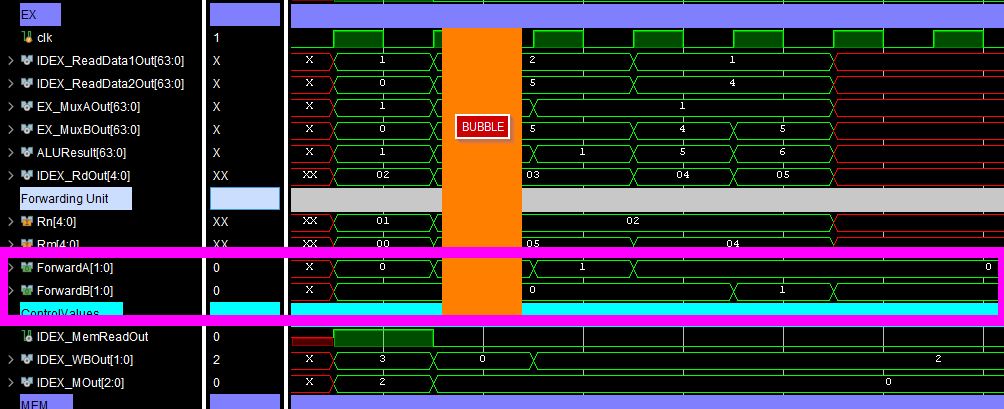
****

Figure 6: In the EX stage we can see the bubble where the data values are Don't Care since the control values on the bottom are all 0. The pink box shows the select lines for the Muxes going into the ALU. Image is EX Stage

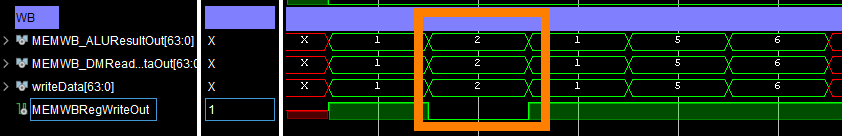
****

Figure 7: Though there are 4 instructions, it takes 5 cycles to complete due to the bubble produced when LDUR leaves the EX stage. Regwrite is low during the bubble so that the Don’t Care data does not get written to the registers. Image is WB Stage

**Discussion:**

One of the issues the group encountered was that there are an exorbitant amount of wires across the design. Naming conventions had to be strictly followed to avoid bugs and confusion throughout the entire process. Another issue that the group had to resolve was the bubble. When a hazard is detected, the processor has to “pause” for some amount of cycles to let the instruction finish. How to implement this bubble gave some issues but was eventually solved.

**Conclusion:**

The use of a testbench with simulation showed that the design in *Figure 1* was successfully implemented. Memory was able to be read in and written to. The waveforms in the Results section showed that the state of the CPU (PC and pertinent registers), and the pertinent memory contents after each instruction was executed matched what was expected. Data hazards were successfully detected and data was forwarded when needed. The given instructions were able to execute in 5 cycles, whereas the previous lab would have taken 4. However, this design is much less error prone due to the implementation of the Hazard Detection Unit and the Forwarding Unit. Implementing this design gave the group a deeper understanding of the ARM architecture as how it may be implemented in a real world application.

**Appendix A (ARM Assemble Test Program):**

LDUR X2, [X1, #0]

AND X3, X5, X2

ORR X4, X4, X2

ADD X5, X4, X2

**Appendix B (Initial Program Memory Load File):**

@0 F8400022

@4 8A050043

@8 AA040044

@c 8B040045

**Appendix C (Testbench Code):**

`timescale 1ns / 1ps

module testbench();

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

//CLK

reg clk;

//Generate Clock

initial

begin

clk = 0;

while(1)

begin

#5 clk = ~clk;

end

end

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

//Reset

reg reset; //Active Low

initial

//Initize reset

begin

reset = 1;

#17 reset = 0;

end

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

always @(posedge clk)

//FRAME

begin

if (reset)

begin

end

else

begin

end

end

wire [63:0] PC;

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

//IM

wire [31:0] IMOut;

IM u\_IM (

.PC(PC),

.IMOut(IMOut)

);

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

//DM

wire DMMemWrite;

wire DMMemRead;

wire [63:0] DMAddress;

wire [63:0] DMWriteData;

wire [63:0] DMReadData;

DM u\_DM(

.DMMemWrite(DMMemWrite),

.DMMemRead(DMMemRead),

.DMAddress(DMAddress),

.DMWriteData(DMWriteData),

.DMReadData(DMReadData)

);

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

Datapath u\_Datapath (

.clk(clk),

.reset(reset),

//PC

.PC(PC),

//IM

.IMOut(IMOut),

//DM

.DMMemWrite(DMMemWrite),

.DMMemRead(DMMemRead),

.DMAddress(DMAddress),

.DMWriteData(DMWriteData),

.DMReadData(DMReadData)

);

endmodule

**Appendix D (Verilog Modules Code):**

`timescale 1ns / 1ps

module Datapath(

input clk,

input reset,

//PC

output [63:0] PC,

//IM

input [31:0] IMOut,

//DM

output DMMemWrite,

output DMMemRead,

output [63:0] DMAddress,

output [63:0] DMWriteData,

input [63:0] DMReadData

);

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

//IF

wire PCWrite;

//assign PCWrite = 1; //DEBUG REMOVE

PC u\_PC(

.clk(clk),

.reset(reset),

.PC(PC),

.PCWrite(PCWrite)

);

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

//IF/ID

wire [31:0] IFID\_InstructionOut;

wire IFID\_Write;

IF\_ID u\_IF\_ID(

.clk(clk),

.reset(reset),

.IFID\_InstructionIn(IMOut),

.IFID\_InstructionOut(IFID\_InstructionOut),

.IFID\_Write(IFID\_Write)

);

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

//ID

wire [4:0] Rn;

wire [4:0] Rm;

wire [4:0] Rd;

wire [63:0] readData1;

wire [63:0] readData2;

wire [63:0] writeData;

assign Rn = IFID\_InstructionOut[9:5]; //DEBUG

assign Rm = IFID\_InstructionOut[20:16];

assign Rd = IFID\_InstructionOut[4:0];

wire [4:0] MEMWB\_RdOut;

wire MEMWBRegWriteOut;

Registers u\_Registers(

.clk(clk),

.reset(reset),

.readRegister1(Rn),

.readRegister2(Rm),

.writeRegister(MEMWB\_RdOut),

.writeData(writeData),

.regWrite(MEMWBRegWriteOut),

.readData1(readData1),

.readData2(readData2)

);

wire HDU\_InstructionMuxSelector;

wire [4:0] IDEX\_RdOut;

wire [2:0] IDEX\_MOut;

wire IDEX\_MemReadOut;

assign IDEX\_MemReadOut = IDEX\_MOut[1];

wire Bubble;

HDU u\_HDU(

.reset(reset),

.Rm(Rm), //input

.Rn(Rn), //input

.IDEX\_RdOut(IDEX\_RdOut), //input

.IDEX\_MemReadOut(IDEX\_MemReadOut), //input

.PCWrite(PCWrite), //Output

.IFID\_Write(IFID\_Write), //Output

.HDU\_InstructionMuxSelector(HDU\_InstructionMuxSelector), //Output

.Bubble(Bubble)

);

wire [31:0] IFID\_MUX\_InstructionOut;

wire [1:0] WBControlOut;

wire [2:0] MControlOut;

wire [2:0] EXControlOut;

Control u\_Control(

.Instruction(IFID\_InstructionOut),

.WB(WBControlOut),

.M(MControlOut),

.EX(EXControlOut)

);

wire [1:0] MuxWBControlOut;

wire [2:0] MuxMControlOut;

wire [2:0] MuxEXControlOut;

ControlMux u\_ControlMux(

.inputWb(WBControlOut),

.inputM(MControlOut),

.inputEX(EXControlOut),

.selector(HDU\_InstructionMuxSelector),

.outputWb(MuxWBControlOut),

.outputM(MuxMControlOut),

.outputEX(MuxEXControlOut)

);

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

//ID/EX

wire [1:0] IDEX\_WBOut;

wire [1:0] IDEX\_ALUOpOut;

wire [63:0] IDEX\_ReadData1Out;

wire [63:0] IDEX\_ReadData2Out;

wire [4:0] IDEX\_RmOut;

wire [4:0] IDEX\_RnOut;

wire [4:0] IDEX\_RdOut;

wire [31:0] IDEX\_InstructionOut;

ID\_EX u\_ID\_EX(

.clk(clk),

.IDEX\_WBIn(MuxWBControlOut),

.IDEX\_WBOut(IDEX\_WBOut),

.IDEX\_MIn(MuxMControlOut),

.IDEX\_MOut(IDEX\_MOut),

.IDEX\_EXIn(MuxEXControlOut), //debug?

.IDEX\_ALUOpOut(IDEX\_ALUOpOut), //debug?

.IDEX\_ALUSrcOut(), //debug?

.IDEX\_ReadData1In(readData1),

.IDEX\_ReadData1Out(IDEX\_ReadData1Out),

.IDEX\_ReadData2In(readData2),

.IDEX\_ReadData2Out(IDEX\_ReadData2Out),

.IDEX\_RnIn(Rn),

.IDEX\_RnOut(IDEX\_RnOut),

.IDEX\_RmIn(Rm),

.IDEX\_RmOut(IDEX\_RmOut),

.IDEX\_RdIn(Rd),

.IDEX\_RdOut(IDEX\_RdOut),

.IDEX\_InstructionIn(IFID\_InstructionOut),

.IDEX\_InstructionOut(IDEX\_InstructionOut),

.Bubble(Bubble)

);

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

//EX

wire [63:0] EX\_MuxAOut;

wire [63:0] EX\_MuxBOut;

wire [63:0] EXMEM\_ALUResultOut;

wire [1:0] ForwardA;

wire [1:0] ForwardB;

Mux u\_ALUInputAMux(

.inputA(IDEX\_ReadData1Out),

.inputB(writeData),

.inputC(EXMEM\_ALUResultOut),

.selector(ForwardA),

.outputData(EX\_MuxAOut)

);

Mux u\_ALUInputBMux(

.inputA(IDEX\_ReadData2Out),

.inputB(EXMEM\_ALUResultOut),

.inputC(writeData),

.selector(ForwardB),

.outputData(EX\_MuxBOut)

);

wire [63:0] ALUResult;

wire [3:0] ALUControlOut;

ALUControl u\_ALUControl(

.OpcodeIn(IDEX\_InstructionOut[31:21]),

.ALUOp(IDEX\_ALUOpOut),

.ALUControlOut(ALUControlOut)

);

ALU u\_ALU(

//Input Data

.inputA(EX\_MuxAOut),

.inputB(EX\_MuxBOut),

//ALUOp

.ALUOp(IDEX\_ALUOpOut),

//Control

.ALUControl(ALUControlOut),

//Output Data

.ALUOutput(ALUResult)

//.Zero(ALUZeroOut)

);

wire [4:0] EXMEM\_RdOut;

wire [1:0] MEMWB\_WBOut;

wire [1:0] EXMEM\_WBOut;

ForwardingUnit u\_ForwardingUnit(

.Rn(IDEX\_RnOut),

.Rm(IDEX\_RmOut),

.MEM\_Rd(EXMEM\_RdOut),

.WB\_Rd(MEMWB\_RdOut),

.MEM\_WB(EXMEM\_WBOut),

.WB\_WB(MEMWB\_WBOut),

.ForwardA(ForwardA),

.ForwardB(ForwardB)

);

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

//EX/MEM

wire EXMEM\_MemWriteOut;

wire EXMEM\_MemReadOut;

EX\_MEM u\_EX\_MEM(

.clk(clk),

.EXMEM\_WBIn(IDEX\_WBOut),

.EXMEM\_WBOut(EXMEM\_WBOut),

.EXMEM\_MIn(IDEX\_MOut),

//.EXMEM\_BranchOut(),

.EXMEM\_MemWriteOut(EXMEM\_MemWriteOut),

.EXMEM\_MemReadOut(EXMEM\_MemReadOut),

//.EXMEM\_PC2AddIn(),

//.EXMEM\_PC2AddOut(),

//.EXMEM\_ALUZeroIn(),

//.EXMEM\_ALUZeroOut(),

.EXMEM\_ALUResultIn(ALUResult),

.EXMEM\_ALUResultOut(EXMEM\_ALUResultOut),

.EXMEM\_MuxBIn(EX\_MuxBOut),

.EXMEM\_MuxBOut(DMWriteData),

.EXMEM\_RdIn(IDEX\_RdOut),

.EXMEM\_RdOut(EXMEM\_RdOut)

);

assign DMAddress = EXMEM\_ALUResultOut;

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

//MEM

assign DMMemWrite = EXMEM\_MemWriteOut;

assign DMMemRead = EXMEM\_MemReadOut;

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

//MEM/WB

wire [63:0] MEMWB\_DMReadDataOut;

wire [63:0] MEMWB\_ALUResultOut;

wire MemToRegOut;

MEM\_WB u\_MEM\_WB(

.clk(clk),

.MEMWB\_WBIn(EXMEM\_WBOut),

.MEMWB\_WBOut(MEMWB\_WBOut),

.MEMWB\_MemToRegOut(MemToRegOut),

.MEMWB\_RegWriteOut(MEMWBRegWriteOut),

.MEMWB\_DMReadDataIn(DMReadData),

.MEMWB\_DMReadDataOut(MEMWB\_DMReadDataOut),

.MEMWB\_ALUResultIn(EXMEM\_ALUResultOut),

.MEMWB\_ALUResultOut(MEMWB\_ALUResultOut),

.MEMWB\_RdIn(EXMEM\_RdOut),

.MEMWB\_RdOut(MEMWB\_RdOut)

);

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

//WB

Mux u\_WBMux(

.inputA(MEMWB\_ALUResultOut),

.inputB(MEMWB\_DMReadDataOut),

.selector(MemToRegOut),

.outputData(writeData)

);

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

endmodule

`timescale 1ns / 1ps

module MEM\_WB(

input clk,

output reg MEMWB\_RegWriteOut,

output reg MEMWB\_MemToRegOut,

input [63:0] MEMWB\_DMReadDataIn,

output reg [63:0] MEMWB\_DMReadDataOut,

input [63:0] MEMWB\_ALUResultIn,

output reg [63:0] MEMWB\_ALUResultOut,

input [4:0] MEMWB\_RdIn,

output reg [4:0] MEMWB\_RdOut,

input [1:0] MEMWB\_WBIn,

output reg [1:0] MEMWB\_WBOut

);

always @ (posedge clk)

begin

MEMWB\_RegWriteOut <= MEMWB\_WBIn[1];

MEMWB\_MemToRegOut <= MEMWB\_WBIn[0];

MEMWB\_WBOut <= MEMWB\_WBIn;

MEMWB\_DMReadDataOut <= MEMWB\_DMReadDataIn;

MEMWB\_ALUResultOut <= MEMWB\_ALUResultIn;

MEMWB\_RdOut <= MEMWB\_RdIn;

end

endmodule

`timescale 1ns / 1ps

module ID\_EX(

input clk,

input [1:0] IDEX\_WBIn,

output reg [1:0] IDEX\_WBOut,

input [2:0] IDEX\_MIn,

output reg [2:0] IDEX\_MOut,

input [2:0] IDEX\_EXIn,

output reg [1:0] IDEX\_ALUOpOut,

output reg IDEX\_ALUSrcOut,

//input [63:0] IDEXPCIn,

//output reg [63:0] IDEXPCOut,

input [63:0] IDEX\_ReadData1In,

output reg [63:0] IDEX\_ReadData1Out,

input [63:0] IDEX\_ReadData2In,

output reg [63:0] IDEX\_ReadData2Out,

input [4:0] IDEX\_RnIn,

output reg [4:0] IDEX\_RnOut,

input [4:0] IDEX\_RmIn,

output reg [4:0] IDEX\_RmOut,

/\*input [63:0] IDEXSignExtendOutIn,

output reg [63:0] IDEXSignExtendOutOut,

input [10:0] IDEXOpcodeIn,

output reg [10:0] IDEXOpcodeOut,

\*/

input [4:0]IDEX\_RdIn,

output reg [4:0] IDEX\_RdOut,

input [31:0] IDEX\_InstructionIn,

output reg [31:0] IDEX\_InstructionOut,

input Bubble

);

always @(posedge clk)

begin

if (Bubble)

begin

IDEX\_WBOut <= 0;

IDEX\_MOut <= 0;

IDEX\_ALUOpOut <= 0;

IDEX\_ALUSrcOut <= 0;

// IDEXPCOut <= IDEXPCIn;

IDEX\_RmOut <= IDEX\_RmIn;

IDEX\_RnOut <= IDEX\_RnIn;

IDEX\_RdOut <= IDEX\_RdIn;

IDEX\_InstructionOut <= IDEX\_InstructionIn;

IDEX\_ReadData1Out <= IDEX\_ReadData1In;

IDEX\_ReadData2Out <= IDEX\_ReadData2In;

// IDEXSignExtendOutOut <= IDEXSignExtendOutIn;

// IDEXOpcodeOut <= IDEXOpcodeIn;

// IDEXWriteRegisterOut <= IDEXWriteRegisterIn;

end

else

begin

IDEX\_WBOut <= IDEX\_WBIn;

IDEX\_MOut <= IDEX\_MIn;

IDEX\_ALUOpOut <= IDEX\_EXIn[2:1];

IDEX\_ALUSrcOut <= IDEX\_EXIn[0];

// IDEXPCOut <= IDEXPCIn;

IDEX\_RmOut <= IDEX\_RmIn;

IDEX\_RnOut <= IDEX\_RnIn;

IDEX\_RdOut <= IDEX\_RdIn;

IDEX\_InstructionOut <= IDEX\_InstructionIn;

IDEX\_ReadData1Out <= IDEX\_ReadData1In;

IDEX\_ReadData2Out <= IDEX\_ReadData2In;

// IDEXSignExtendOutOut <= IDEXSignExtendOutIn;

// IDEXOpcodeOut <= IDEXOpcodeIn;

// IDEXWriteRegisterOut <= IDEXWriteRegisterIn;

end

end

endmodule

`timescale 1ns / 1ps

module PC(

input clk,

input reset,

input PCWrite,

output reg [63:0] PC

);

always @(posedge clk)

begin

if (reset)

begin

PC <= 0;

end

else

begin

if (PCWrite == 1)

begin

PC <= PC + 4;

end

end

end

endmodule

`timescale 1ns / 1ps

module IM(

input [63:0] PC,

output [31:0] IMOut

);

reg [31:0] IM[31:0];

initial

begin

//Initialize IM

$readmemh("D:/CELab2/Lab3/ARM-Pipeline-Processor-With-Hazard-Avoidance/IM.dat", IM);

end

assign IMOut = IM[PC];

endmodule

`timescale 1ns / 1ps

module IF\_ID(

input clk,

input reset,

input [31:0] IFID\_InstructionIn,

output reg [31:0] IFID\_InstructionOut,

input IFID\_Write

);

always @(posedge clk)

begin

if (reset)

begin

end

else

begin

if (IFID\_Write == 0)

begin

end

else

begin

IFID\_InstructionOut <= IFID\_InstructionIn;

end

end

end

endmodule

`timescale 1ns / 1ps

module Registers(

input clk,

input reset,

//Control value

input regWrite,

//Reg locations (64)

input [4:0] readRegister1,

input [4:0] readRegister2,

input [4:0] writeRegister,

//Data (64)

input [63:0] writeData,

//Data (64)

output [63:0] readData1,

output [63:0] readData2

);

reg [63:0] RM[31:0];

initial

begin

$readmemh("D:/CELab2/Lab3/ARM-Pipeline-Processor-With-Hazard-Avoidance/RM.dat", RM);

end

assign readData1 = RM[readRegister1];

assign readData2 = RM[readRegister2];

always @(\*)

begin

//Drive RM

//Output

//RM

if (reset)

begin

$readmemh("D:/CELab2/ARM-Pipeline-Processor/RM.dat", RM);

end

else

begin

if (regWrite)

begin

RM[writeRegister] = writeData;

end

end

end

endmodule

`timescale 1ns / 1ps

module Control(

input [31:0] Instruction,

output reg [1:0] WB,

output reg [2:0] M,

output reg [2:0] EX

);

always @(\*)

begin

if ((Instruction[31:21] == 11'b10001011000) || (Instruction[31:21] == 11'b10001010000) || (Instruction[31:21] == 11'b10101010000) || (Instruction[31:21] == 11'b10001010000))

//add //sub //or //and

begin

EX = 3'b100;

M = 3'b000;

WB = 2'b10;

end

if ((Instruction[31:21] == 11'b11111000010)) //LDUR

begin

EX = 3'b001;

M = 3'b010;

WB = 2'b11;

end

if ((Instruction[31:21] == 11'b11111000000)) //STUR

begin

EX = 3'b001;

M = 3'b001;

WB = 2'b00;

end

if ((Instruction[31:24] == 11'b10110100)) //CBZ

begin

EX = 3'b010;

M = 3'b100;

WB = 2'b00;

end

end

endmodule

`timescale 1ns / 1ps

module Mux(

input clk,

input reset,

input [63:0] inputA,

input [63:0] inputB,

input [63:0] inputC,

input [1:0] selector,

output reg [63:0] outputData

);

always @(\*)

//Select data in Mux

//Output

//outputData

begin

if(reset)

begin

end

else

begin

if (selector == 1)

begin

outputData = inputB;

end

else if (selector == 2)

begin

outputData = outputData;

end

else

begin

outputData = inputA;

end

end

end

endmodule

`timescale 1ns / 1ps

module HDU(

input reset,

input [4:0] Rm,

input [4:0] Rn,

input [4:0] IDEX\_RdOut,

input IDEX\_MemReadOut,

output reg HDU\_InstructionMuxSelector,

output reg PCWrite,

output reg IFID\_Write,

output reg Bubble

);

always @(\*)

begin

if (reset)

begin

PCWrite <= 1;

end

else

begin

if (((IDEX\_RdOut == Rm) || (IDEX\_RdOut == Rn)) && IDEX\_MemReadOut)

begin

IFID\_Write = 0;

PCWrite = 0;

HDU\_InstructionMuxSelector = 1;

Bubble = 1;

end

else

begin

IFID\_Write = 1;

PCWrite = 1;

HDU\_InstructionMuxSelector = 0;

Bubble = 0;

end

end

end

endmodule

`timescale 1ns / 1ps

module ALU(

//Input Data

input [63:0] inputA,

input [63:0] inputB,

//ALUOP(debug)

input [1:0] ALUOp,

//Control

input [3:0] ALUControl,

//Output Data

output reg [63:0] ALUOutput,

output reg Zero

);

reg [63:0] addInputB;

wire [9:0] DebugInputB;

assign DebugInputB = addInputB;

always @(\*)

begin

case(ALUOp)

2'b00: begin

addInputB = 0;//inputB[20:12];

end

default: begin

addInputB = inputB;

end

endcase

end

always@(\*) begin //aluc

//switch statement

case(ALUControl)

//case 0000: inputA AND(&) inputB

4'b0000: begin

ALUOutput = inputA & inputB;

end

//case 0001: inputA OR(|) inputB

4'b0001: begin

ALUOutput = inputA | inputB;

end

//case 0010: inputA add(+) inputB

4'b0010: begin

ALUOutput = inputA + addInputB;

end

//case 0110: inputA subtract(-) inputB

4'b0110: begin

ALUOutput = inputA - inputB;

end

//case 0111: Pass inputB, result = inputB

4'b0111: begin

ALUOutput = inputB; //For CBZ

end

endcase

end

always@(\*) begin //posedge CKLK

if(ALUOutput == 0)

Zero = 1;

else

Zero = 0;

end

endmodule

`timescale 1ns / 1ps

module ForwardingUnit(

input [4:0] Rn,

input [4:0] Rm,

input [4:0] MEM\_Rd,

input [4:0] WB\_Rd,

input [1:0] MEM\_WB, //DEBUG?

input [1:0] WB\_WB, //DEBUG?

output reg [1:0] ForwardA,

output reg [1:0] ForwardB

);

always @(\*)

begin

if (Rn == WB\_Rd)

begin

ForwardA = 1;

end

else if (Rn == MEM\_Rd)

begin

ForwardA = 2;

end

else

begin

ForwardA = 0;

end

if (Rm == WB\_Rd)

begin

ForwardB = 2;

end

else if (Rm == MEM\_Rd)

begin

ForwardB = 1;

end

else

begin

ForwardB = 0;

end

end

endmodule

`timescale 1ns / 1ps

module EX\_MEM(

input clk,

input [1:0] EXMEM\_WBIn,

output reg [1:0] EXMEM\_WBOut,

input [2:0] EXMEM\_MIn,

//output [2:0] EXMEM\_WBOut,

//output reg EXMEMBranchOut,

output reg EXMEM\_MemWriteOut,

output reg EXMEM\_MemReadOut,

//input [63:0] EXMEMPC2AddIn,

//output reg [63:0] EXMEMPC2AddOut,

//input EXMEMALUZeroIn,

//output reg EXMEMALUZeroOut,

input [63:0] EXMEM\_ALUResultIn,

output reg [63:0] EXMEM\_ALUResultOut,

input [63:0] EXMEM\_MuxBIn,

output reg [63:0] EXMEM\_MuxBOut,

input [4:0] EXMEM\_RdIn,

output reg [4:0] EXMEM\_RdOut

);

always @(posedge clk)

begin

EXMEM\_WBOut <= EXMEM\_WBIn;

//EXMEM\_BranchOut <= EXMEM\_MIn[2];

EXMEM\_MemWriteOut <= EXMEM\_MIn[0];

EXMEM\_MemReadOut <= EXMEM\_MIn[1];

//EXMEM\_PC2AddOut <= EXMEM\_PC2AddIn;

//EXMEM\_ALUZeroOut <= EXMEM\_ALUZeroIn;

EXMEM\_ALUResultOut <= EXMEM\_ALUResultIn;

EXMEM\_MuxBOut <= EXMEM\_MuxBIn;

EXMEM\_RdOut <= EXMEM\_RdIn;

end

endmodule

`timescale 1ns / 1ps

module DM(

input DMMemWrite,

input DMMemRead,

input [63:0] DMAddress,

input [63:0] DMWriteData,

output [63:0] DMReadData

);

reg [63:0] DM[31:0];

initial begin

//Initialize IM

$readmemh("D:/CELab2/Lab3/ARM-Pipeline-Processor-With-Hazard-Avoidance/DM.dat", DM);

end

assign DMReadData = DM[DMAddress];

always @(\*)

begin

if (DMMemWrite)

begin

DM[DMAddress] = DMWriteData;

end

end

endmodule

`timescale 1ns / 1ps

module ControlMux(

input [1:0] inputWb,

input [2:0] inputM,

input [2:0] inputEX,

input selector,

output reg [1:0] outputWb,

output reg [2:0] outputM,

output reg [2:0] outputEX

);

always @(\*)

//Select data in Mux

//Output

//outputData

begin

if (selector == 1)

begin

outputWb = 0;

outputM = 0;

outputEX = 0;

end

else

begin

outputWb = inputWb;

outputM = inputM;

outputEX = inputEX;

end

end

endmodule

`timescale 1ns / 1ps

module ALUControl(

input [10:0] OpcodeIn,

input [1:0] ALUOp,

output reg [3:0] ALUControlOut

);

always @(\*)

begin

if (ALUOp == 2'b00)

begin

ALUControlOut = 4'b0010;

end

else if (ALUOp == 2'b01)

begin

ALUControlOut = 4'b0111;

end

else if (ALUOp == 2'b10)

begin

case(OpcodeIn)

//case 0000: inputA AND(&) inputB

11'b10001010000: begin

ALUControlOut = 4'b0000;

end

//case 0001: inputA OR(|) inputB

11'b10101010000: begin

ALUControlOut = 4'b0001;

end

//case 0010: inputA add(+) inputB

11'b10001011000: begin

ALUControlOut = 4'b0010;

end

//case 0110: inputA subtract(-) inputB

11'b11001011000: begin

ALUControlOut = 4'b0110;

end

endcase

end

end

endmodue