ENG 363

Lab #5 ARM Processor Implementation

Brian Worts and Chris Jenson

12/5/19



**Table of Contents**

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

Procedure……….…...……………………………………………………………………….. 2

Deliverables……...…...…………………………………………………………………...….. 4

Discussion/Conclusion ……………………………………………………………………….11

Appendix A (*Processor*).......................................................................................................... 12

Appendix B (*Datapath*)............................................................................................................ 15

Appendix C (*Controller*)...........................................................................................................17

Appendix D (*ALU*)................................................................................................................... 21

Appendix E (*Storage*)............................................................................................................... 23

Appendix F (*Register*).............................................................................................................. 24

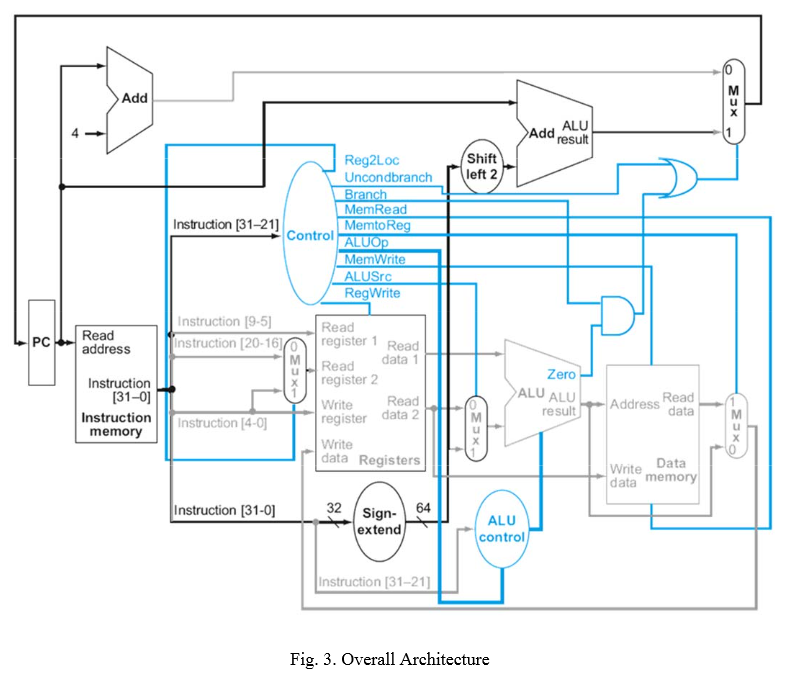
Appendix G (*Test Bench*).......................................................................................................... 25

**Intro:**

Computers are playing an ever increasing role in our society, and the processor is at the core of this innovation. They are becoming increasingly complex and faster while also shrinking in size. In this lab, we designed our second processor, a single cycle 64 bit ARM V7 processor at as high an abstraction level as possible.

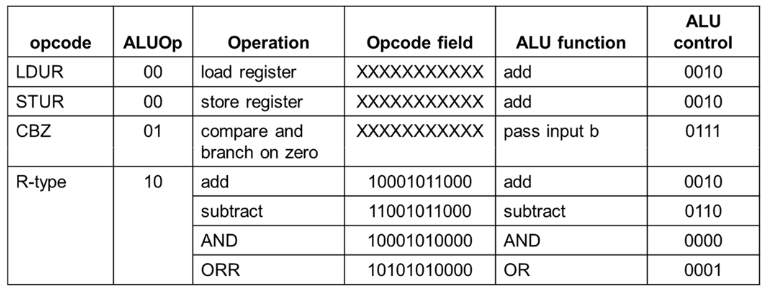
**Procedure:**

The first step of this lab was to understand the supplied architecture of the processor, shown in *Figure 1*. This architecture helped give an understanding of the basic components that make up the processor as well as how the components are connected. *Table 1* shows the necessary operations that were implemented in the processor and *Table 2* shows the control values for each operation.

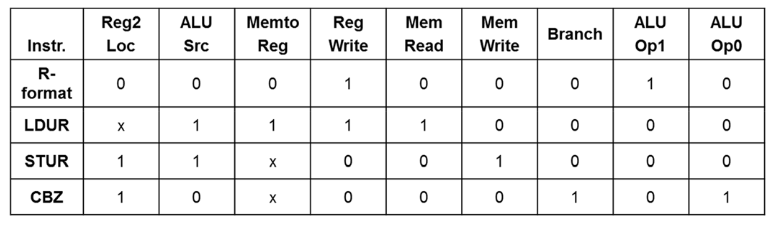
**

***Figure 1:*** The processor architecture

***Table 1:*** The required operations of the processor



***Table 2:*** The control values for each operation



Next, using *Figure 1*, and *Tables 1-2* the processor was designed and tested in Verilog HDL. In order to implement the processor, the code was split up into 6 modules: The processor (*Appendix A,*) the datapath (*Appendix B,*) the ALU (adapted from the ALU in Lab 4, *Appendix C,*) the controller *(Appendix D,)* the register module (*Appendix E,*) and the storage module (*Appendix F*.)Furthermore, a testbench was written to allow for the design to be simulated and tested (*Appendix G.*)

**Deliverables:**

1. **All of the verilog code is shown in the appendices**
2. **Test Program in Arm Assembly Language**

Add X2, X5, X4

And X2, X3, X1

Sub X5, X7, X2

Orr X8, X5, X9

Ldur X5, [X2, #0]

Stur X0, [X3, #0]

Add X4, X5, X2

Cbz X0, #-8

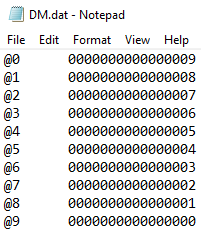
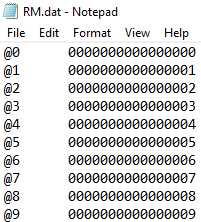
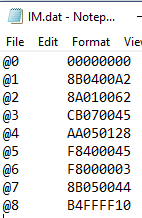
1. **Initial program memory load file (assembled test program in machine language)**

***Table 3:***This table shows the values stored in the instruction memory file (IM.dat)

|  |  |
| --- | --- |
| Hex | Binary |
| 8B0400A2 | 10001011000001000000000010100010 |
| 8A010062 | 10001010000000010000000001100010 |
| CB070045 | 11001011000001110000000001000101 |
| AA050128 | 10101010000001010000000100101000 |
| F8400045 | 11111000010000000000000001000101 |
| F8000003 | 11111000000000000000000000000011 |
| 8B050044 | 10001011000001010000000001000100 |
| B4FFFF10 | 10110100111111111111111110001000 |

***Table 4:*** This table shows the deconstruction of the instructions

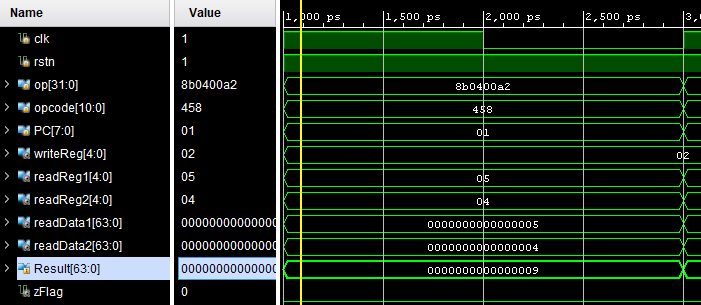
|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Assembly | Instruct | Opcode | Rm | Shamt | Add | Op | Rn | Rd/Rt | Hex | Binary |
| Add X2, X5, X4 | ADD | 10001011000 | 00101 | 000000 |  |  | 00100 | 00010 | 8B0400A2 | 1000101100000100  0000000010100010 |
| And X2, X3, X1 | AND | 10001010000 | 00011 | 000000 |  |  | 00001 | 00010 | 8A010062 | 1000101000000001  0000000001100010 |
| Sub X5, X7, X2 | SUB | 11001011000 | 00111 | 000000 |  |  | 00010 | 00101 | CB070045 | 1100101100000111  0000000001000101 |
| Orr X8, X5, X9 | Or | 10101010000 | 00101 | 000000 |  |  | 01001 | 01000 | AA050128 | 1010101000000101  0000000100101000 |
| Ldur X5, [X2 #0] | LDUR | 11111000010 |  |  | 000000000 | 00 | 00010 | 00101 | F8400045 | 1111100001000000  0000000001000101 |
| Stur X0, [X3 #0] | STUR | 11111000000 |  |  | 000000000 | 00 | 00000 | 00011 | F8000003 | 1111100000000000  0000000000000011 |
| add X4, X5, X2 | ADD | 10001011000 | 00101 | 000000 |  |  | 00010 | 00100 | 8B050044 | 1000101100000101  0000000001000100 |
| Cbz X0 #-8 | CBX | 10110100 |  |  | 1111111111  111111000 |  |  | 00000 | B4FFFF10 | 1011010011111111  1111111100010000 |



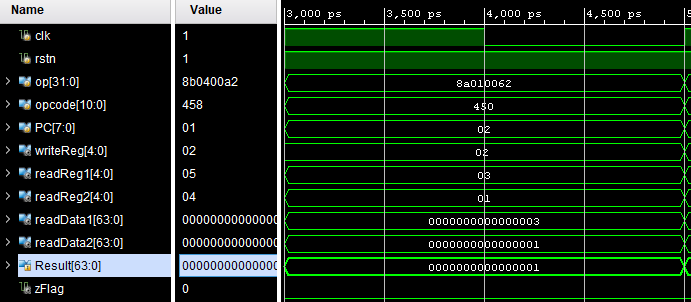
***Figure 2:*** These 3 images show the content of the instruction memory, register memory, and data memory data files. These files are loaded in as the initial contents of IM, RM, and DM.

**D. The waveforms showing the state of the CPU and memory content.**

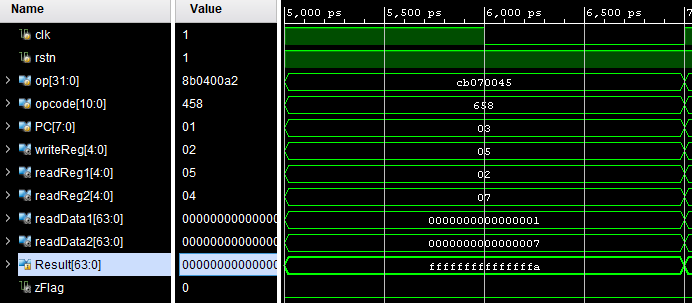
For the waveforms, op has the instructions given in hex and opcode is the instruction opcode in hex. PC is the program counter. ReadReg holds the register number and readData hold the value at that register. Result is the output of the ALU. IM is the instruction memory, DM is the data memory, and RM is the register memory. WriteReg is the register being written to. *Figures 3.1-3.9* show the functioning of the operations.



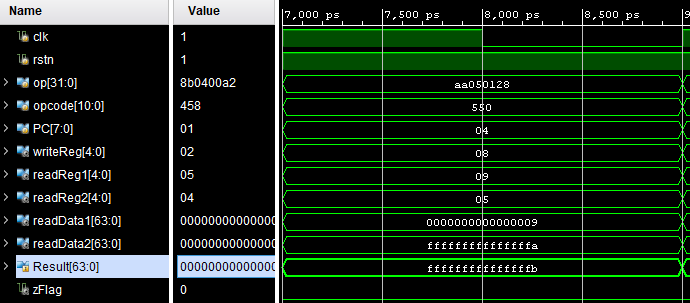
***Figure 3.1:*** The resulting waveform of the first add instruction. Showing that it is adding X5 and X4 into X2 resulting in 4+5 = 9.



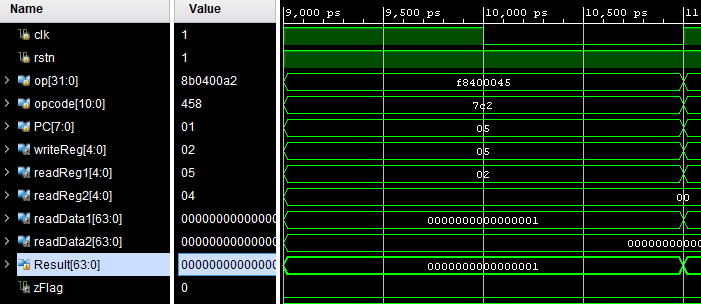
***Figure 3.2:*** The resulting waveform and instruction. Showing that it is and-ing X3 and X1 into X2 and that the result is 3&1 = 1.



***Figure 3.3:*** The resulting waveform of the sub instruction. There is an overflow when subtracting 7 from one so the value goes negative. Showing that it is subtracting X2 and X7 into X5 and that the result is 1-7 = fff..ffa which is correct

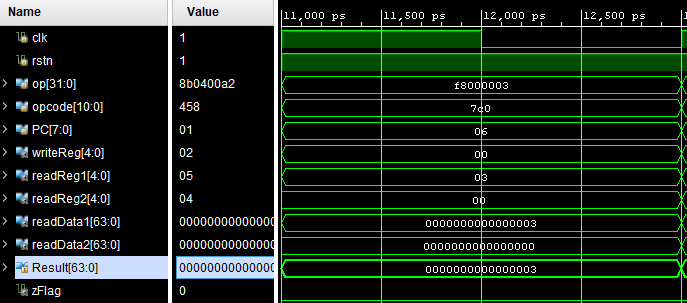


***Figure 3.4*:** The resulting waveform of the or instruction. Showing the or-ing of X9 and X5 into X8 and the result is 9 | ffff...fffa = ffff..fffb.

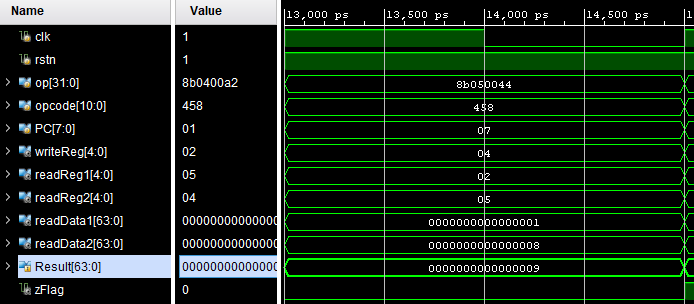


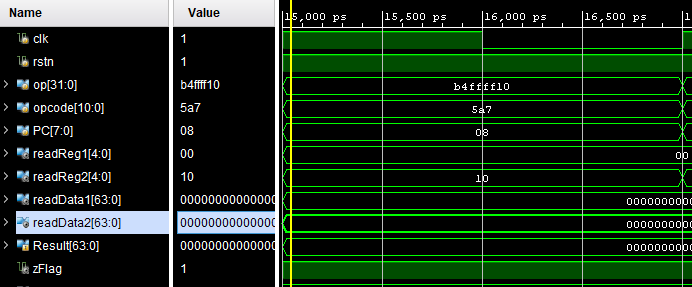


***Figure 3.5*:** The resulting waveform of the ldur instruction. Showing that the register value in X2 was stored in the register value of X5.



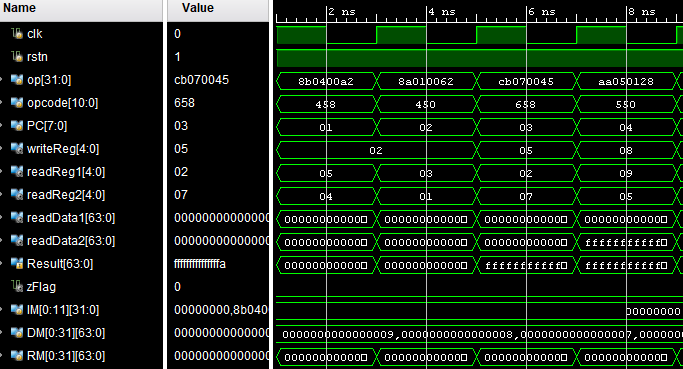
***Figure 3.6:*** The resulting waveform of the stur instruction. Showing that the register value in X0 was stored in the data memory value for X3.

***Figure 3.7*:** The resulting waveform of the second add instruction. Showing the addition of X5 and X2 into X4 resulting in 1 + 8 = 9

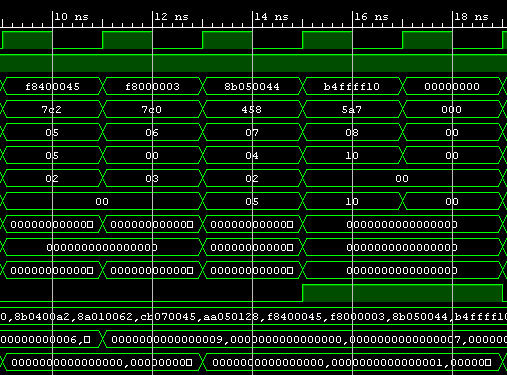
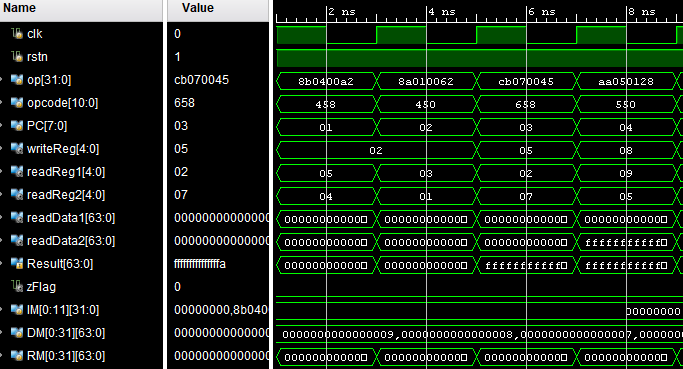




***Figure 3.8:*** The resulting waveform of the cbz instruction. Showing that registers X2 and X7 holding 1 and 8 is 9 and that the program counter went to the branch of 0.



***Figure 3.9A*:** This figure shows that the register memory (RM) is changing after each R-type instruction (Add, And, Sub, Orr). Though the contents of RM are too large to show here, the values in readReg1 and readReg2 are written to register writeReg.



***Figure 3.9B:*** This figure shows the continuation of Figure 3.9A. The values of DM are too large to show, but one can see that the value changes after performing an Ldur where register Writereg gets the value from DM at result. Next, Stur is performed and one can see a change in RM, though the values are too large to show here, where DM at result gets the value at readReg2. Next an add is performed for aesthetic reasons, and then CBZ is performed. CBZ goes back 8 instructions by subtracting 8 from the PC, which brings the PC back to 0. Thus, all operations work successfully.

**Conclusion and Discussion:**

The first challenge to overcome in this lab was understanding the processor specifications. These specifications differed from the first in that it was supposed to be a single cycle processor as opposed to the previous processor we designed which was implemented as a state machine using a minimum CPI Von Newman approach. However, this challenge was overcome and we were successful in this regard.

Another challenge we encountered was the use of data files to store the initial values of the registers, the data memory, and the instruction memory. Doing this allowed the code to be significantly shorter since we did not have to code the values into the modules. While we did not implement the functionality to write to these files, we could still change the values within the program as shown in *Figure 3.9A&B.* Thus, we accomplished the goals of the lab and were successful here as well.

One specification we were unsuccessful in implementing was incrementing the program counter by 4. We developed the project using a program counter that incremented by 1, so the effort required to switch to incrementing by 4 was more than we had time to do. When attempting to increment by 4, after adjusting all files to match this, we were able to read the instruction memory, but received an array with null values in the indexes that were not multiples of 4. Given more time we believe that we could remedy this issue, but given the constraints, we did not.

**Appendix A: Processor**

`timescale 1ns / 1ps

module Processor(

input clk,

input rstn,

input [31:0]operation,

input [7:0]PC,

output [7:0] PCout,

output reg [31:0] RD, //This is the Read data registed, not destination register

output reg [31:0] A, //This is the output register

output [383:0]One\_D\_Array\_IM,

output [63:0]readData1,

output [63:0]readData2,

output [4:0]readReg1,

output [4:0]readReg2,

output zFlag,

output [63:0] Result,

output [3:0] ALUc,

output [63:0] ALUB

);

reg [10:0]opcode; //DEBUG check bit length... 10?

reg [4:0] Rn;

reg [4:0] Rd;

reg [4:0] Rm;

reg [5:0] Shamt;

reg [8:0] DT\_Address;

reg [1:0] op;

reg [25:0] BR\_Address;

reg [18:0] COND\_BR\_Address;

wire Reg2Loc, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, ALUOp1, ALUOp0, BranchMux;

wire [4:0] writeReg;

reg [63:0] Extended;

wire [63:0] ReadDataStorage;

wire [63:0] WriteDataReg;

always @(\*) begin //operation

opcode = operation[31:21];

COND\_BR\_Address = 0;

//BR\_Address = 0; //dont need to have branch

Rm = 0;

Rn = 0;

Rd = 0;

op = 0;

Shamt = 0;

case(opcode[10:3])

//CBZ

8'b10110100: begin

COND\_BR\_Address = operation[23:5];

Rd = operation[4:0];

Extended <= $signed(COND\_BR\_Address);

end

endcase

// case(opcode[10:5])

// //B

// 6'b000101: begin

// BR\_Address = operation[25:0];

// Extended <= $signed(BR\_Address);

// end

// endcase

case (opcode)

//LDUR

11'b11111000010: begin

DT\_Address = operation[20:12];

op = operation[11:10];

Rn = operation[9:5];

Rd = operation[4:0];

Extended <= $signed(DT\_Address);

end

//STUR

11'b11111000000: begin

DT\_Address = operation[20:12];

op = operation[11:10];

Rd = operation[9:5];

Rn = operation[4:0];

Extended <= $signed(DT\_Address);

end

//ADD

11'b10001011000: begin

Rm = operation[20:16];

Shamt = operation[15:10];

Rn = operation[9:5];

Rd = operation[4:0];

end

//AND

11'b10001010000: begin

Rm = operation[20:16];

Shamt = operation[15:10];

Rn = operation[9:5];

Rd = operation[4:0];

end

//SUB

11'b11001011000: begin

Rm = operation[20:16];

Shamt = operation[15:10];

Rn = operation[9:5];

Rd = operation[4:0];

end

//ORR

11'b10101010000: begin

Rm = operation[20:16];

Shamt = operation[15:10];

Rn = operation[9:5];

Rd = operation[4:0];

end

endcase

end

Controller Controller(clk, opcode, Reg2Loc, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, ALUOp1, ALUOp0, ALUc, rstn);

Datapath Datapath(clk, rstn, Reg2Loc, ALUSrc, MemtoReg, Branch, ALUc, readReg1, readReg2, writeReg, readData1, readData2, Rn,Rd,Rm, PC, PCout, Result, zFlag, Extended, ReadDataStorage, WriteDataReg, ALUB);

Register Register(clk,readReg1, readReg2, readData1,readData2,RegWrite,writeReg,WriteDataReg); //One\_D\_Array\_DM\_Output

Storage Storage(clk, Result, MemWrite, MemRead, readData2, ReadDataStorage);

Endmodule

**Appendix B: Datapath**

`timescale 1ns / 1ps

module Datapath(

input clk, rstn, Reg2Loc, ALUSrc, MemtoReg, RegWrite, Branch,

input [3:0] ALUc,

output reg [4:0] readReg1, readReg2, writeReg,

input [63:0] readData1, readData2,

input [4:0] Rn,Rd,Rm,

input [7:0] PC,

output reg[7:0] PCout,

output [63:0] result,

output zFlag,

input [63:0]Extended,

input [63:0]ReadDataStorage,

output reg[63:0] WriteDataReg,

output reg [63:0] ALUB

);

reg BranchMux;

ALU ALU(clk, readData1, ALUB, ALUc, result, zFlag);

always @(\*) begin //posedgeclk

readReg1 = Rn;

writeReg = Rd;

case(Reg2Loc)

0: begin

readReg2 = Rm;

end

1: begin

readReg2 = Rd;

end

endcase

case(ALUSrc)

0: begin

ALUB = readData2;

end

1: begin

ALUB = Extended;

end

endcase

case(MemtoReg)

0: begin

WriteDataReg = result;

end

1: begin

WriteDataReg = ReadDataStorage;

end

endcase

if (BranchMux == 1) begin //BranchMux == 1

PCout = (Extended) + PC; //DEBUG: go back to where the branch was called and Extended \* 4

end

if ((Branch == 1) && (zFlag == 1)) begin

BranchMux = 1;

end

else begin

BranchMux = 0;

end

end

endmodule

**Appendix C: Controller**

`timescale 1ns / 1ps

//DEBUG chekc if passing in is correct

module Controller(

input clk,

input [10:0]opcode,

output reg Reg2Loc, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, ALUOp1, ALUOp0,

output reg[3:0] ALUc,

input rstn

);

always @(\*) begin //As soon as it gets input, runs operation

if (rstn == 0) begin

Reg2Loc = 0;

ALUSrc = 0;

MemtoReg = 0;

RegWrite = 0;

MemRead = 0;

MemWrite = 0;

Branch = 0;

ALUOp1 = 0;

ALUOp0 = 0;

//ALUc = 4'b0000;

end

case(opcode[10:3])

//CBZ

8'b10110100: begin

Reg2Loc = 1;

ALUSrc = 0;

RegWrite = 0;

MemRead = 0;

MemWrite = 0;

Branch = 1;

ALUOp1 = 0;

ALUOp0 = 1;

ALUc = 4'b0111;

end

endcase

// case(opcode[11:6]) //dont need to have branch

// //B

// 6'b000101: begin

// ALUSrc = 0;

// MemtoReg = 0;

// RegWrite = 0;

// MemRead = 0;

// MemWrite = 0;

// Branch = 0;

// ALUOp1 = 0;

// ALUOp0 = 0;

// ALUc = 4'b0000;

// UncondBranch = 1;

// end

// endcase

case (opcode)

//LDUR

11'b11111000010: begin

ALUSrc = 1;

MemtoReg = 1;

RegWrite = 1;

MemRead = 1;

MemWrite = 0;

Branch = 0;

ALUOp1 = 0;

ALUOp0 = 0;

ALUc = 4'b0010;

end

//STUR

11'b11111000000: begin

Reg2Loc = 1;

ALUSrc = 1;

RegWrite = 0;

MemRead = 0;

MemWrite = 1;

Branch = 0;

ALUOp1 = 0;

ALUOp0 = 0;

ALUc = 4'b0010;

end

//ADD

11'b10001011000: begin

Reg2Loc = 0;

ALUSrc = 0;

MemtoReg = 0;

RegWrite = 1;

MemRead = 0;

MemWrite = 0;

Branch = 0;

ALUOp1 = 1;

ALUOp0 = 0;

ALUc = 4'b0010;

end

//AND

11'b10001010000: begin

Reg2Loc = 0;

ALUSrc = 0;

MemtoReg = 0;

RegWrite = 1;

MemRead = 0;

MemWrite = 0;

Branch = 0;

ALUOp1 = 1;

ALUOp0 = 0;

ALUc = 4'b0000;

end

//SUB

11'b11001011000: begin

Reg2Loc = 0;

ALUSrc = 0;

MemtoReg = 0;

RegWrite = 1;

MemRead = 0;

MemWrite = 0;

Branch = 0;

ALUOp1 = 1;

ALUOp0 = 0;

ALUc = 4'b0110;

end

//ORR

11'b10101010000: begin

Reg2Loc = 0;

ALUSrc = 0;

MemtoReg = 0;

RegWrite = 1;

MemRead = 0;

MemWrite = 0;

Branch = 0;

ALUOp1 = 1;

ALUOp0 = 0;

ALUc = 4'b0001;

end

endcase

end

endmodule

**Appendix D: ALU**

`timescale 1ns / 1ps

module ALU(CLK, inputA, inputB, ALUc, result, zFlag

);

input CLK;

input [63:0] inputA, inputB;

input [3:0] ALUc;

output reg [63:0] result;

output reg zFlag;

reg [64:0] extra; //used to calculate carry out

always@(\*) begin //aluc

//switch statement

case(ALUc)

//case 0000: inputA AND(&) inputB

4'b0000: begin

assign result = inputA & inputB;

end

//case 0001: inputA OR(|) inputB

4'b0001: begin

assign result = inputA | inputB;

end

//case 0010: inputA add(+) inputB

4'b0010: begin

assign result = inputA + inputB;

end

//case 0110: inputA subtract(-) inputB

4'b0110: begin

assign result = inputA - inputB;

end

//case 0111: Pass inputB, result = inputB

4'b0111: begin

assign result = inputB;

end

//case 1100: inputA NOR(~|) inputB

// 4'b1100: begin

// assign result = ~(inputA | inputB);

// end

endcase

end

always@(\*) begin //posedge CKLK

if(result == 0)

assign zFlag = 1;

else

assign zFlag = 0;

end

endmodule

**Appendix E: Storage**

`timescale 1ns / 1ps

module Storage(

input clk,

input [63:0]Result,

input MemWrite,MemRead,

input [63:0] WriteDataStorage,

output reg [63:0] ReadDataStorage

);

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

initial begin

$readmemh("H:/363/project\_5/DM.dat", DM);

end

always @(\*) begin

if (MemRead == 1) begin

ReadDataStorage = DM[Result];

end

if (MemWrite == 1) begin

DM[Result] = WriteDataStorage;

end

end

endmodule

**Appendix F: Register**

`timescale 1ns / 1ps

//In this file we will do the reading writing to registers

module Register(

input clk,

input [4:0] readReg1,

input [4:0] readReg2,

input RegWrite,

input [4:0] WriteReg,

input [63:0] WriteDataReg,

output reg[63:0] readData1,

output reg[63:0] readData2

);

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

initial begin

$readmemh("H:/363/project\_5/RM.dat", RM);

end

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

genvar i;

always @(\*) begin

readData1 = RM[readReg1];

readData2 = RM[readReg2];

if (RegWrite == 1) begin

RM[WriteReg] = WriteDataReg;

end

end

endmodule

**Appendix G: Testbench**

`timescale 1ns / 1ps

module testbench(

);

reg clk;

reg rstn;

reg [31:0] op;

reg [31:0] IM[0:11];

reg [7:0] PC;

wire [7:0] PCout;

wire [383:0]One\_D\_Array\_IM;

wire [63:0]readData1;

wire [63:0]readData2;

wire [4:0]readReg1;

wire [4:0]readReg2;

wire [3:0] ALUc;

wire zFlag;

wire [63:0] Result;

wire [63:0] ALUB;

initial begin

clk = 0;

op = 0;

PC = 0;

rstn = 0;

$readmemh("H:/363/project\_5/IM.dat", IM);

#1

rstn = 1; //Doing a high active reset

end

genvar i;

for (i = 0; i < 12; i = i + 1) begin

assign One\_D\_Array\_IM[32\*i+31:i\*32] = IM[i];

end

always @(clk) begin

#1

clk <= ~clk;

end

always @(PC & (rstn==1)) begin

assign op = IM[PC];

end

always @(posedge clk & rstn == 1) begin //at the end of processing, increment... or @aluoutput

//PC = PC + 1;

if (op[31:24] != 8'b10110100) begin

PC = PC + 1;

end

else begin

PC = PCout;

end

end

always @(posedge clk) begin

if (rstn == 0) begin

PC = 0;

end

end

Processor uut (

.clk(clk),

.operation(op),

.rstn(rstn),

.One\_D\_Array\_IM(One\_D\_Array\_IM),

.readData1(readData1),

.readData2(readData2),

.readReg1(readReg1),

.readReg2(readReg2),

.zFlag(zFlag),

.Result(Result),

.ALUc(ALUc),

.ALUB(ALUB),

.PC(PC),

.PCout(PCout)

);

endmodule