Best* Group

Brian Dalmar, Dylan Danowski, Olivia Elwell, Adam Kinsley

4 May 2022

*this is an accurate statement

TABLE OF CONTENTS

0. PREFACE

0a. List of Figures

0b. List of Tables

0c. Overview of Features

32x32 RegFile	**128x32 RAM**	**Control Unit**	**Program Counter**
ALU Supporting 13 Instructions	**32x32 ROM**	**Branch Comparator**	**Immediate Sign Extender**

1. REGISTER FILE

1a.Architecture Overview section with simplified block diagram

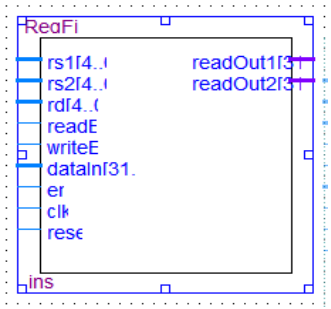

Figure 2. Block Diagram for RegFile

The inputs that are attached to the outputs are rs1, rs2, rd, and dataIn. Other inputs are readEn, writeEn, clk, reset, and en. The outputs are readOut1 and readOut2. Data either flows into the register or out of the register depending on what is enabled. If writeEn is on, dataIn is written into the register location rd. If readEn is on, data is read from rs1 and rs2 and written into readOut1 and readOut2.

1b. Design

Our register has a 32 bit data input, two 5-bit read address inputs, one 5-bit write address input, and a 1-bit input for the clock, enable, read enable, write enable, and reset signals. There is an array of 32 32-bit registers, as well as two 32-bit output signals. An integer variable is used to loop through and reset all register values to 32'b0. When write enable is on, and the clock signal hits its positive edge, the 32-bit data input is written to the register denoted by the 5-bit write address input. When read enable is on, the 32-bit output signals read out the register data denoted by the two 5-bit read address inputs. Enable must be on for any actions to occur.

```
module RegFile(rs1, rs2, readOut1, readOut2, rd, readEn, writeEn, dataIn, en, clk, reset);

   input [4:0] rs1, rs2; //destination of read reg
   input [4:0] rd; // address for where value will be written (i or o?)
   output reg [31:0] readOut1, readOut2; //bitstrings read from rs1 and rs2
   input en, clk, reset; // basic
   input readEn, writeEn; // read enable and write enable
   input [31:0] dataIn; // data written in reg
   reg [31:0] rf[31:0]; // 32 32 bit registers
   integer i; //loop variable

   always @(posedge clk) begin

      if (en) begin

         if (reset) begin // loop to set all reg to 0 if reset

            for (i=0; i<32; i = i+1) begin
            rf[i] <= 32'b00000000000000000000000000000000;
            end

         end

         else if (readEn && !writeEn) begin //read enable
            readOut1 <= rf[rs1];
            readOut2 <= rf[rs2];
         end

      end

      if (en) begin

         if (reset) begin // loop to set all reg to 0 if reset

            for (i=0; i<32; i = i+1) begin
            rf[i] <= 32'b00000000000000000000000000000000;
            end

         end

         else if (writeEn && !readEn) begin //wirte enable
            rf[rd] <= dataIn;
         end

      end

   end

endmodule
```

Figure 3. RegFile Verilog

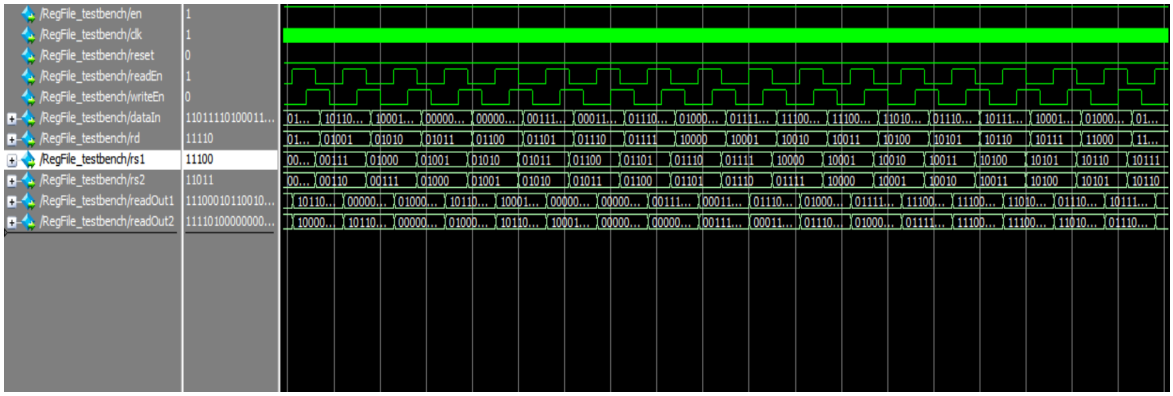

Figure 4. ModelSim Generated from RegFile Testbench

The testbench generates random values for dataIn which are written into the register using rd and writeEn. The values are read out of the register using rs1 and rs2 into readOut1 and readOut2 respectively. The testbench shows that the random generated values were able to be read out of the register using the read function when it was enabled. readOut1 and readOut2 outputting values also shows that the write functionality worked and was able to write the randomly generated values into the register.

2. ALU

2a.Block Diagram Design

The inputs to the ALU include the 32-bit contents of rs1, labeled A, the 32-bit contents of rs2, labeled B, and the 7-bit opcode to choose from the functions of the ALU. The outputs of the ALU include the 32-bit output from the operation performed on A and B and the 5-bit flags output.

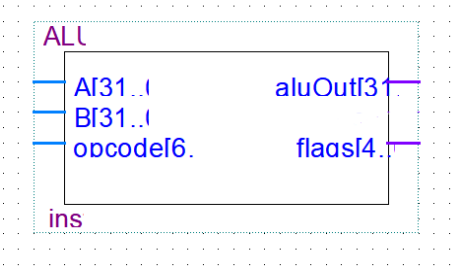

Figure 5. Block Diagram of ALU

2b. Overview of ALU Design

The ALU performs arithmetic and logical operations on inputs taken and outputs the result of that operation as well as flags. The ALU takes in two 32-bit inputs and a 7-bit opcode, a value that determines what operation the ALU will perform. A case statement was used to quickly select between the possible opcode values the ALU can take in. The ALU performs one of 13 operations, then records the 32-bit output. It then checks the output for each of the five flags, all of which are represented by 1 bit of a 5-bit flags output. Some random inputs and outputs taken from those random inputs can be viewed below in the ModelSim of our testbench.

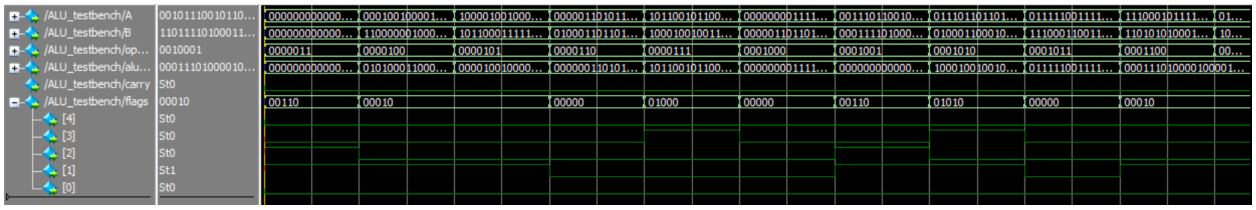

Figure 6. ALU ModelSim

2c. Description

The ALU is a component that performs logic and arithmetic operations. This ALU has functionality for or, and, xor, adder, subtractor, left shifter, right shifter, add 1, subtract 1, 0 word

out, not, reg output, and a 2's complement tool. The inputs are A and B and outputs are aluOut, the carry bit, as well as the flags. The five flags we have in place are the carryout, negative, zero, parity, overflow. Table 1 below shows the opcode values, their function name, and the purpose of the function they carry out. Table 2 shows the flag bits, the flag they are tied to, and the meaning of each flag.

Table 1. ALU Opcodes, Functions, and Functionalities

Opcode	Function	Functionality
0000000	OR	Performs bitwise or on A and B and outputs
0000001	AND	Performs bitwise and on A and B and outputs
0000010	XOR	Performs bitwise xor on A and B and outputs
0000011	ADD	Adds A and B and outputs
0000100	SUB	Subtracts B from A and outputs
0000101	LEFT SHIFT	Multiplies input A by 2 and outputs
0000110	RIGHT SHIFT	Divides input A by 2 and outputs
0000111	+1	Adds 1 to input A and outputs
0001000	-1	Subtracts 1 from input A and outputs
0001001	OUT 0	Outputs a word of 0s
0001010	NOT	Performs bitwise not on A and outputs
0001011	REG VALUE	Outputs the value of A
0001100	2'S COMP	Takes the 2's complement of A and outputs it

Table 2. ALU Flag Bits, Names, and Meanings

Flag Bit	Name of Flag	Meaning
0	Overflow	Flips to 1 if addition or subtraction results in a 33rd bit, both inputs are the same sign, and the signed bit is different that the signed bit of the inputs
1	Parity	Written as even parity, thi sbit flips to 1 if there are an even amount of 1s in the output
2	Zero	The output is a word of 0s
3	Negative	The output is negative.
4	cOut	Addition or subtraction has caused there to be a 33rd bit, if that is the case this flag flips to 1

2d. Testing and validation

A testbench was written in order to validate the functionality of the ALU. The testbench looped through each possible opcode value in order to test each function. Random A and B inputs were generated for each opcode value. The output and the flags were then checked to ensure that they performed their function correctly on the inputs.

3. Memory Organization

3a. Memory Addressing Figure

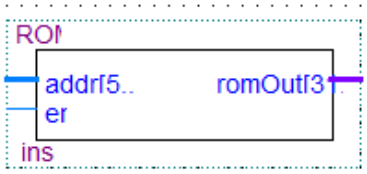

Figure 7. Block Diagram of ROM

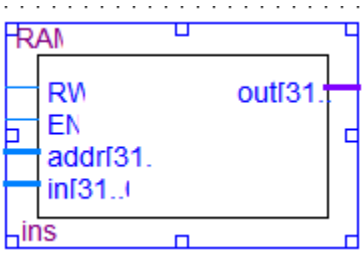

Figure 8. Block Diagram of RAM

3b. Memory Design and Implementation

ROM

The ROM will function as the instruction memory (IMEM) in the datapath. The ROM was constructed as a 32x32 bit register that cannot be written into. It has an enable input that allows values to be read out when it is 1 and disallows when it is 0. It takes in an address and outputs the information held at that address.

RAM

The RAM will function as the data memory (DMEM) in the datapath. The ROM was constructed as a 256x32 bit register that can be written into or read out of. It has an enable that allows it to function when 1 and disallows it to function when it is 0. It has a RW input from the control unit that allows the RAM to be written into when 1 and read out of when 0. An address is inputted into the RAM to decide which register of the RAM will be accessed. If the RAM is

written into, data taken into the RAM will be written to the address inputted. If the RAM is being read out of, data will be taken from the input address to be stored in the regfile.

4. Datapath

4a. Instruction Register Figure

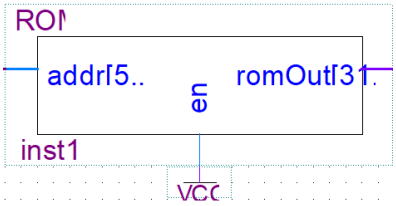

Figure 9. Instruction Register Block Schematic

4b. Program Counter Figure

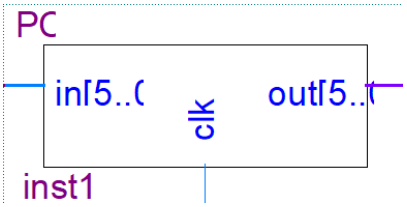

Figure 10. Program Counter Block Schematic

4c. Datapath Figure

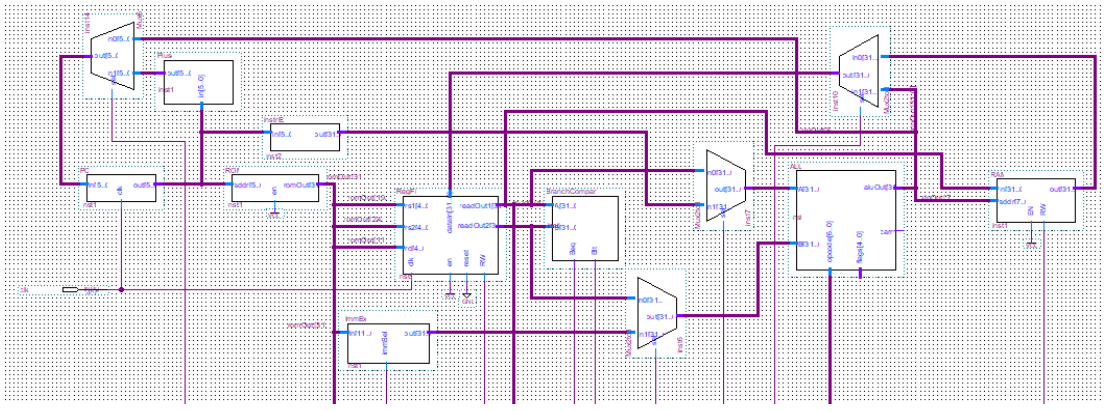

Figure 11. Datapath Without Control Unit

4d. Signals and Function Table

Table 3. Control Unit Signals for Each Function

Instruction	Signal Outputs
R-Type	PCsel => 0, REGwen =>1->0, IMMsel => 0, Bsel => 0, BRun => 0, Beq => dc, Blt => dc, Asel => 0, ALUop => appropriate opcode for instruction, WBsel => 1, MEMrw => 1
LW	PCsel => 0, Asel => 0, REGwen => 1 -> 0, IMMsel => 1, Bsel => 1, Beq => dc, Blt => dc, Asel => 0, ALUop => appropriate opcode for instruction, WBsel => 0, MEMrw => 1
SW	PCsel => 0, Asel => 0, REGwen => 1 -> dc, IMMsel => 1, Bsel => 1, Beq => dc, Blt => dc, Asel => 0, ALUop => appropriate opcode for instruction, WBsel => dc, MEMrw => 1
Beq	PCsel => 0, Asel => 0 if false, 1 if true, REGwen => dc, IMMsel => 1, Bsel => 1, Beq => depends on inputs, Blt => depends on inputs, Asel => 1, ALUop => appropriate opcode for instruction, WBsel => dc, MEMrw => 1
Bne	PCsel => 0, Asel => 0 if false, 1 if true, REGwen => dc, IMMsel => 1, Bsel => 1, Beq => depends on inputs, Blt => depends on inputs, Asel => 1, ALUop => appropriate opcode for instruction, WBsel => dc, MEMrw => 1
Bge	PCsel => 0, Asel => 0 if false, 1 if true, REGwen => dc, IMMsel => 1, Bsel => 1, Beq => depends on inputs, Blt => depends on inputs, Asel => 1, ALUop => appropriate opcode for instruction, WBsel => dc, MEMrw => 1
Blt	PCsel => 0, Asel => 0 if false, 1 if true, REGwen => dc, IMMsel => 1, Bsel => 1, Beq => depends on inputs, Blt => depends on inputs, Asel => 1, ALUop => appropriate opcode for instruction, WBsel => dc, MEMrw => 1

4e. Control Signals Description Table

Table 4. Function of Each Control Unit Signal

Variable Name	Function
PCsel	Selects whether the counter continues with +4 or if a branch is taken
REGwen	Enables reading/writing from the register
IMMsel	Controls mux that chooses between

	immediate and reg values
BRun	Always comparing signed numbers, no real functionality
Beq	Output based on if the comparators are equal
Blt	Checks if the first value is less than the second value
Bsel	If enabled, sign extension is used.
Asel	Controls if reg value or current PC counter value enters ALU
ALUop	7 LSBs of instruction word, controls which operation is carried out
WBsel	If enabled, allows for writing back into the register. Used for R and I type instructions.
MEMrw	If enabled, allows for reading and writing in memory. Used for I type instructions

4f. Design Choices and Capabilities

An extra input was added to the DMEM on the same bus so that an input and an address could be taken in. The BRun signal is always set to 0 as this processor does not need to compare unsigned numbers. An extra sign extender was added so that the branched address fit the number of bits for the ALU. For simplicity in creating the CU, all R-Type instructions have a lower opcode value than I-Type instructions. The instruction word was formatted as immediate value as 31:22, rs2 as 21:17, rs1 as 16:12, rd as 11:7, and opcode as 6:0.

5. Control Unit

5a. Control Unit Figure

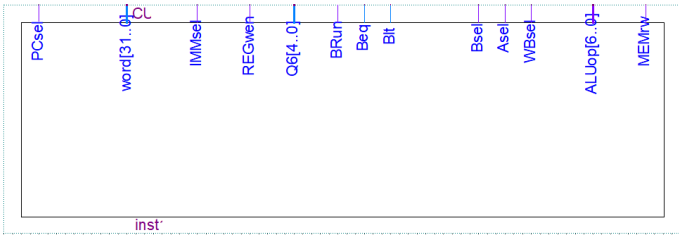

Figure 12. Control Unit Block Schematic

5b. Design Approach and Developed Design

The control unit is written by determining what each control signal would be for each opcode. Each scenario was written as an if statement. Each of the R-Type instructions have opcodes that are less than a certain value, the load and store word instructions have the following two opcodes, and the branch instructions have the opcodes after that. This made it easy to group the instructions and use less than or greater than in the if statement definition for control signals where the control signal is the same for each instruction type. For example, Bsel selects the register value for all R-Type instructions, so having the R-Type opcodes all next to each other in vale allows for a simple less than argument in the if statement.

5c. Testbench Figure and Description of Instruction Decoded

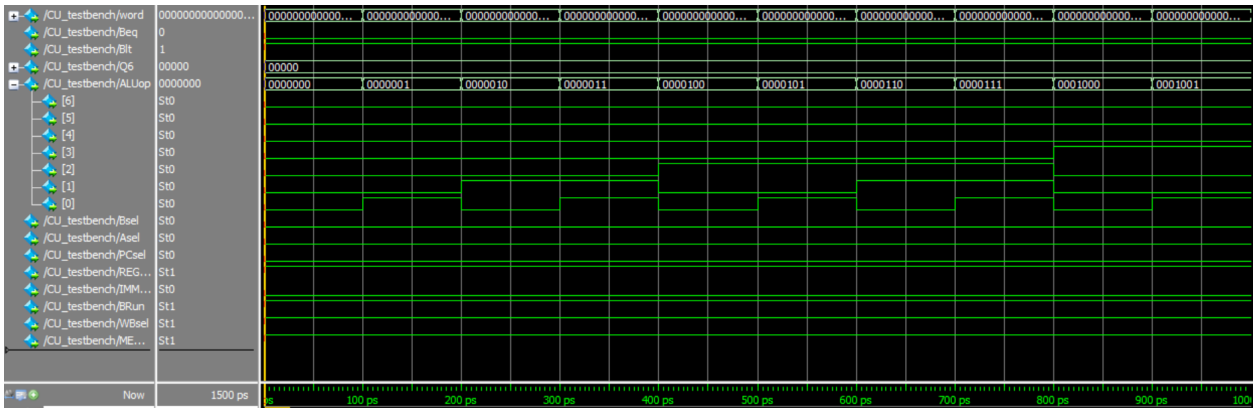

Figure 13. Testbench Execution for Control Unit

The control unit was tested with opcodes that were incremented by one every 100ps. In each case, the signals that were desired were produced.

6. CPU

6a. Datapath/Control Unit/ Memory Figure

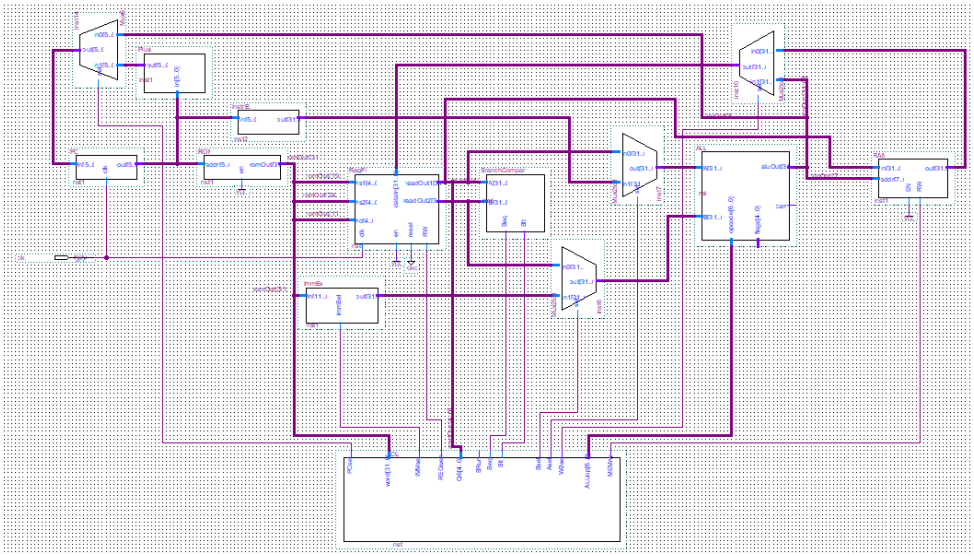

Figure 14. Block Schematic of Completed Datapath

6b. CPU Testbench Validation Figure

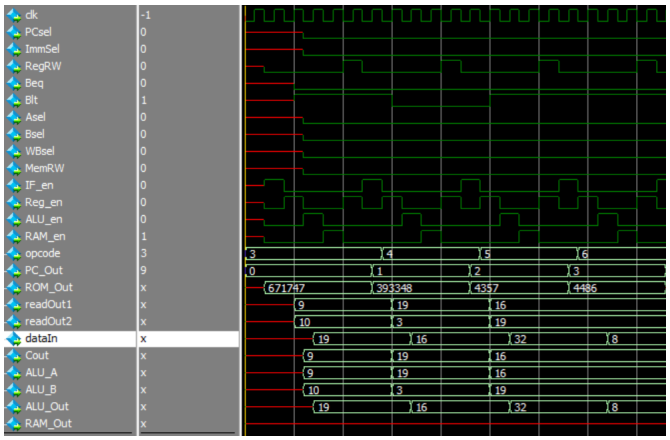

Figure 15. Testbench of R-Type Instructions

7. Instruction Set

Common Instructions

7a. Opcode Table

The table below shows the opcodes of each of the supported instructions for the CPU.

Table 5. Opcodes

Opcode	Function	Functionality
0000000	OR	Performs bitwise or on A and B and outputs
0000001	AND	Performs bitwise and on A and B and outputs
0000010	XOR	Performs bitwise xor on A and B and outputs
0000011	ADD	Adds A and B and outputs
0000100	SUB	Subtracts B from A and outputs
0000101	SL	Multiplies input A by 2 and outputs
0000110	SR	Divides input A by 2 and outputs
0000111	PLS	Adds 1 to input A and outputs
0001000	MIN	Subtracts 1 from input A and outputs
0001001	ZER	Outputs a word of 0s

0001010	NOT	Performs bitwise not on A and outputs
0001011	DSP	Outputs the value of A
0001100	CMP	Takes the 2's complement of A and outputs it
0001101	LW	Takes a value from the DMEM and loads it into a regfile spot
0001110	SW	Takes a value from the regfile and loads it into the DMEM
0001111	BEQ	Takes a branch if the two checked values are equal, continues as normal if not equal
0010000	BNE	Takes a branch if the two checked values are not equal, continues as normal if equal
0010001	BGE	Take the branch if one of the checked values is greater or equal to the second value.
0010010	BLT	Take the branch if one of the checked values is less than the second value.

7b. Instruction Format Figure

The figure below shows the format of the instruction word the CPU must take in.

[31:22] imm	[21:17] rs2	[16:12] rs1	[11:7] rd	[6:0] opcode

Figure 16. Bit Format of Instruction Word

7c. Instructions Supported

The figure below shows the type of instructions supported by the CPU and each instruction in each type.

Table 6. Instructions and Instruction Types

R-Type	OR
	AND
	XOR
	ADD
	SUB
	SL
	SR
	PLS
	MIN
	ZER
	NOT
	DSP
	CMP
I-Type	LW
	SW
B-Type	BEQ
	BNE
	BGE
	BLT

8. Example Programs

A. Program 1 - Add, Subtract, Multiply, Divide

```
initial begin
    ROM[0] <= 32'b00000000000010100100000000000011; //ADD x0, x4, x5
    ROM[1] <= 32'b00000000000001100000000010000100; //SUB x1, x0, x3
    ROM[2] <= 32'b00000000000000000001000100000101; //SL x2, x1
    ROM[3] <= 32'b00000000000000000001000110000110; //SR x3, x1
end
```

Figure 17. Instructions For Program 1

Table 7. Initial Values and Result of Program 1

Instructions	Register File	Memory

ADD x0, x4, x5
SUB x1, x0, x3
SL x2, x1
SR x3, x1
	Reg[0] = 0
	Reg[1] = 1
	Reg[2] = 2
	Reg[3] = 3
	Reg[4] = 9
	Reg[5] = 10
		N/A

RESULT	Reg[0] = 19 Reg[1] = 16 Reg[2] = 32 Reg[3] = 8	N/A

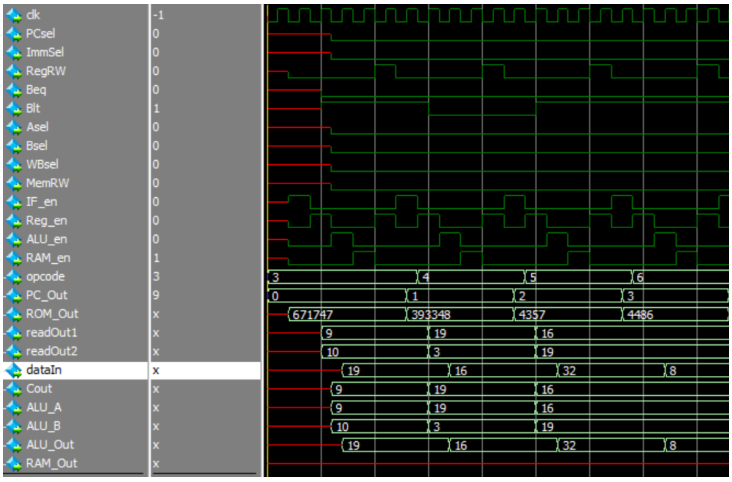

Figure 18. Waveform for Program 1. This program adds, subtracts, multiplies by 2 (shift left), and divides by 2 (shift right)

B. Program 2 - Load Word

```
initial begin
    ROM[0] <= 32'b00000000100000000010000010001101; //LW x1, x2(2)
    ROM[1] <= 32'b00000000110000000010000110001101; //LW x3, x2(3)
    ROM[2] <= 32'b00000000000000000001000000001011; //DSP x1
    ROM[3] <= 32'b00000000000000000001100000001011; //DSP x3
end
```

Figure 19. Instructions For Program 2

Table 8. Initial Values and Result of Program 2

Instructions	Register File	Memory

LW x1, x2(2)
LW x3, x2(3)
DSP x1
DSP x3
	Reg[1] = 0
	Reg[2] = 10
	Reg[3] = 0
		RAM[12] = 69
		RAM[13] = 420
		
		Wow so silly :P

RESULT	Reg[1] = 69 Reg[2] = 10 Reg[3] = 420	Unchanged

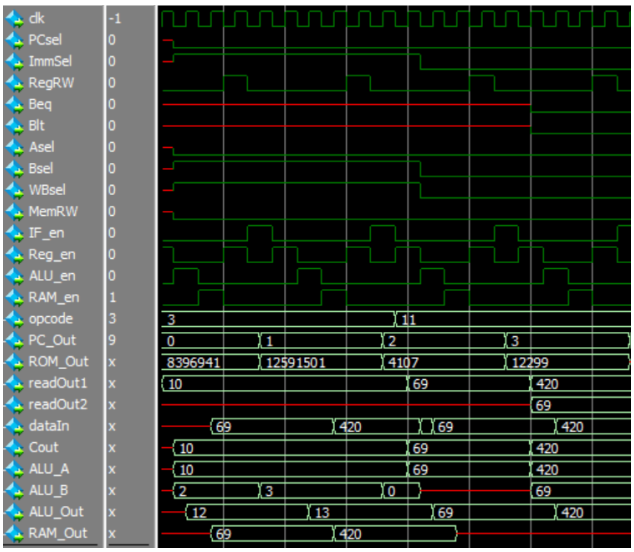

Figure 20. Waveform for Program 2. This program loads two words from the RAM and displays them

C. Program 3 - Store Word

```
initial begin
    ROM[0] <= 32'b00000000101011010101000000001110; //SW  x21, x22(2)
    ROM[1] <= 32'b00000000000101101010110101000010; //XOR x21, x21, x22
    ROM[2] <= 32'b00000001001011010101000000001110; //SW  x21, x22(4)
    ROM[3] <= 32'b00000000100000010110101110001101; //LW  x23, x22(2)
    ROM[4] <= 32'b00000001000000010110110000001101; //LW  x24, x22(4)
end
```

Figure 21. Instructions For Program 3

Table 9. Initial Values and Result of Program 3

Instructions	Register File	Memory

SW x21, x22(2)
XOR x21, x21, x22
SW x21, x22(4)
LW x23, x22(2)
LW x24, x22(4)
	Reg[21] = 99
	Reg[22] = 58
	Reg[23] = 3
	Reg[24] = 4
		RAM[60] = 8
		RAM[62] = 3

RESULT	Reg[21] = 89 Reg[22] = 58 Reg[23] = 99 Reg[24] = 89	RAM[60] = 99 RAM[62] = 89

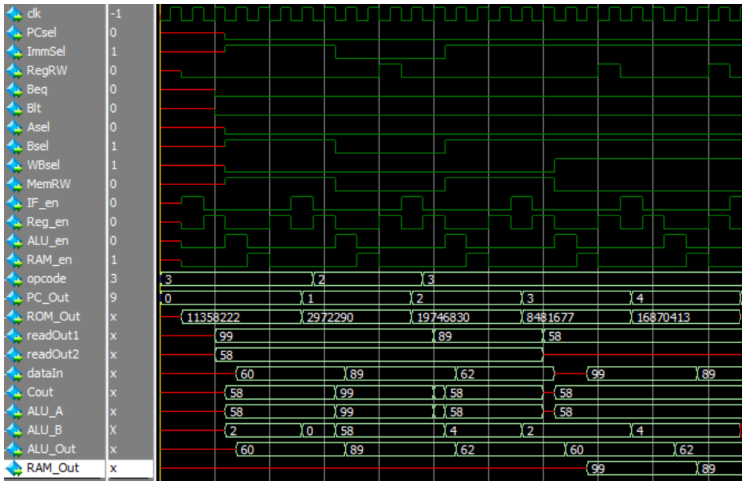

Figure 22. Waveform for Program 3. This program creates a history of register 1. It stores register 1 in RAM, modifies its value, stores it again at a different index, and then loads both values sequentially

D. Program 4 - Branch Loop

```
initial begin
    ROM[0] <= 32'b00000000000000000001000010000111; //PLS x1, x1
    ROM[1] <= 32'b11111111110001100001000000010000; //BNE x1, x3, -1
    ROM[2] <= 32'b00000000000001100011000110000001; //XOR x3, x3, x3
end
```

Figure 23. Instructions For Program 4

Table 10. Initial Values and Result of Program 3

Instructions	Register File	Memory
PLS x1, x1 BNE x1, x3, -1 CMP x3, x3	Reg[1] = 0 Reg[3] = 3	N/A
RESULT	Reg[1] = 3 Reg[3] = -3	N/A

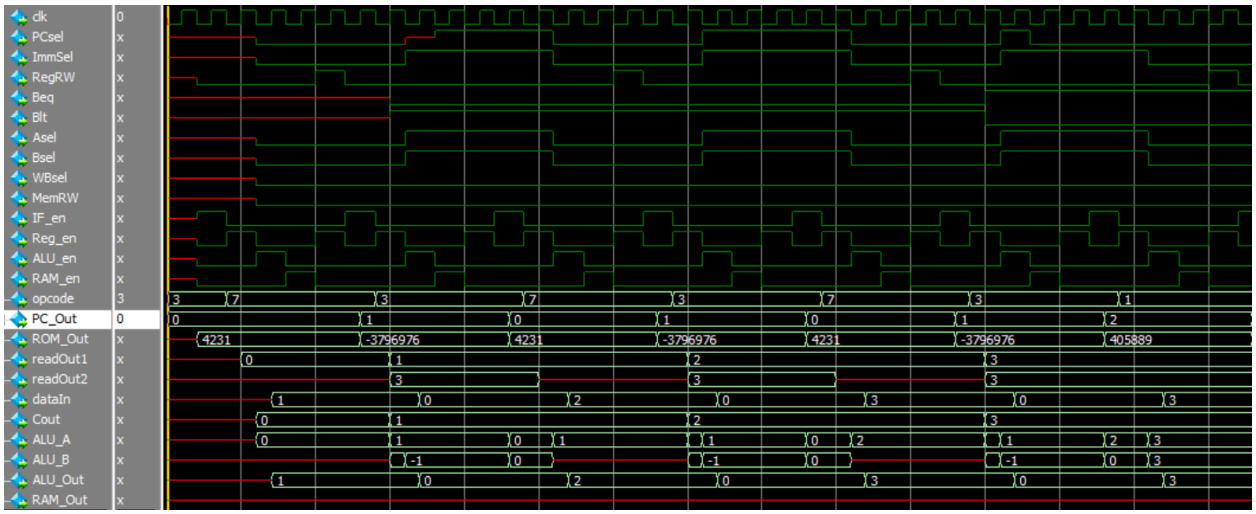

Figure 24. Waveform for Program 4. This program iterates through instructions 0 and 1 three times before meeting the condition of instruction 1, and then clears register 3 with a XOR operation

9. Performance

9a. Performance Retrospective

All instructions the program is meant to run work as expected. Each of the R-Type instructions produce the correct value and store that value in the determined register as seen in Figure 18. Load word and store word are able to take a word from the RAM and store it in the RegFile or take a word from the RegFile and store it in the RAM respectively as seen in Figures 20 and 22. Finally, branch instructions work as expected and can create a loop as seen in figure 24. Each instruction takes 5 clock cycles which is expected, 1 clock cycle each for IF, ID, EX, MEM, and WB.

10. Errata

10a. Malfunctioning Features

There are no features of the CPU that are expected to work that do not work. All instructions produce the desired result in an expected amount of time. The instructions could run faster with a pipelined datapath, but that was not a requirement. Overall, some shortcuts were taken that may hurt the functionality of the CPU on a large scale.

10b. Why These Problems Exist

The datapath is not pipelined because there was no intent to pipeline the datapath. One of the shortcuts taken that may hurt the CPU on a large scale is that the operation words were all formatted to be identical. Only so many more instructions can be added without a f3 field in the opcode. Support for instructions that take in two user inputs would require that the format of the instruction word be changed.

11. Appendix

11a. Dwaipayan Chakraborty

May your registers be vast and your datapaths forever pipelined