Processor Datasheet

Logan Greif, Krutik Shah, Cristine Le Ny

Computer Architecture

5/10/21

Overview of Features

- 32-bit architecture
- 32 x 64-bit general purpose registers
- 255 x 64-bit data memory registers

Table of Contents

List of Figures

List of Tables

1 Architecture Overview

1.1 Simplified Block Diagram

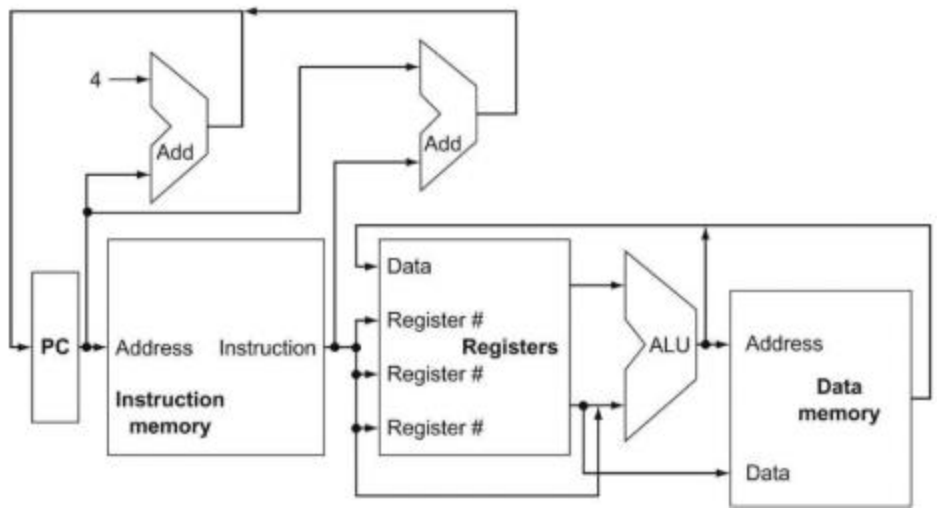


Figure 1.1. Simplified block diagram of processor

1.2 Description of Architecture Features

1.2.1 Program Counter

The PC is the program counter. The program counter counts by 4 for each instruction the processor executes. The program counter is how the processor knows what line of assembly to execute at any given time.

1.2.2 Instruction Memory

Instruction memory is where you write to/load the information into the register file. This may include changing the value of a register of moving the value of one register to another.

1.2.3 Registers

A register is temporary memory. For the register file, you can either write or read the data that it within each individual register. In total there is 32 x 64-bit registers.

1.2.4 ALU

The arithmetic logic unit (ALU) is the place where all the arithmetic and logic operations happen. The inputs of the ALU are the specific registers that were selected from the register file.

1.2.5 Data Memory

The data memory is another source to store memory. For our design, the data memory is our RAM file.

2 Register File Design

2.1 Figure of Register File Design

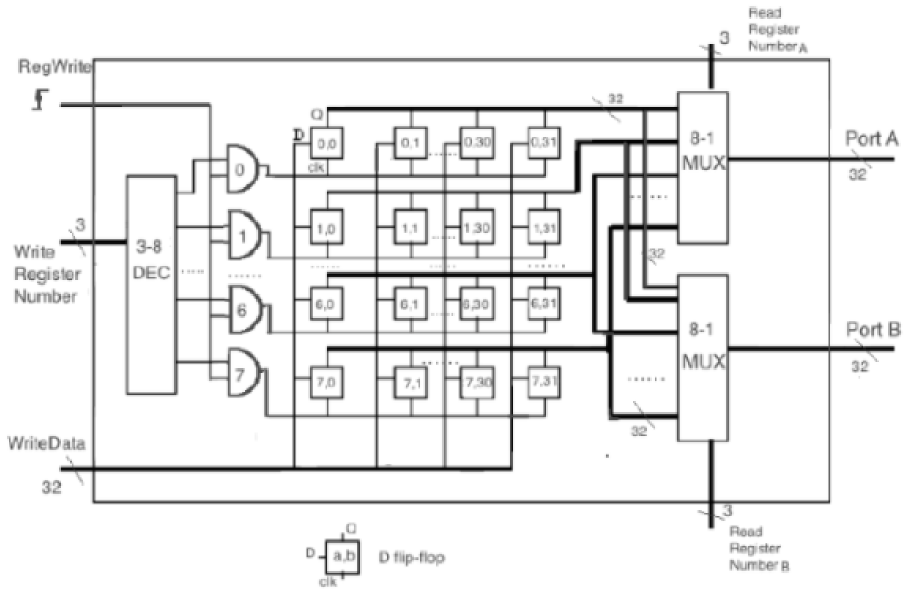


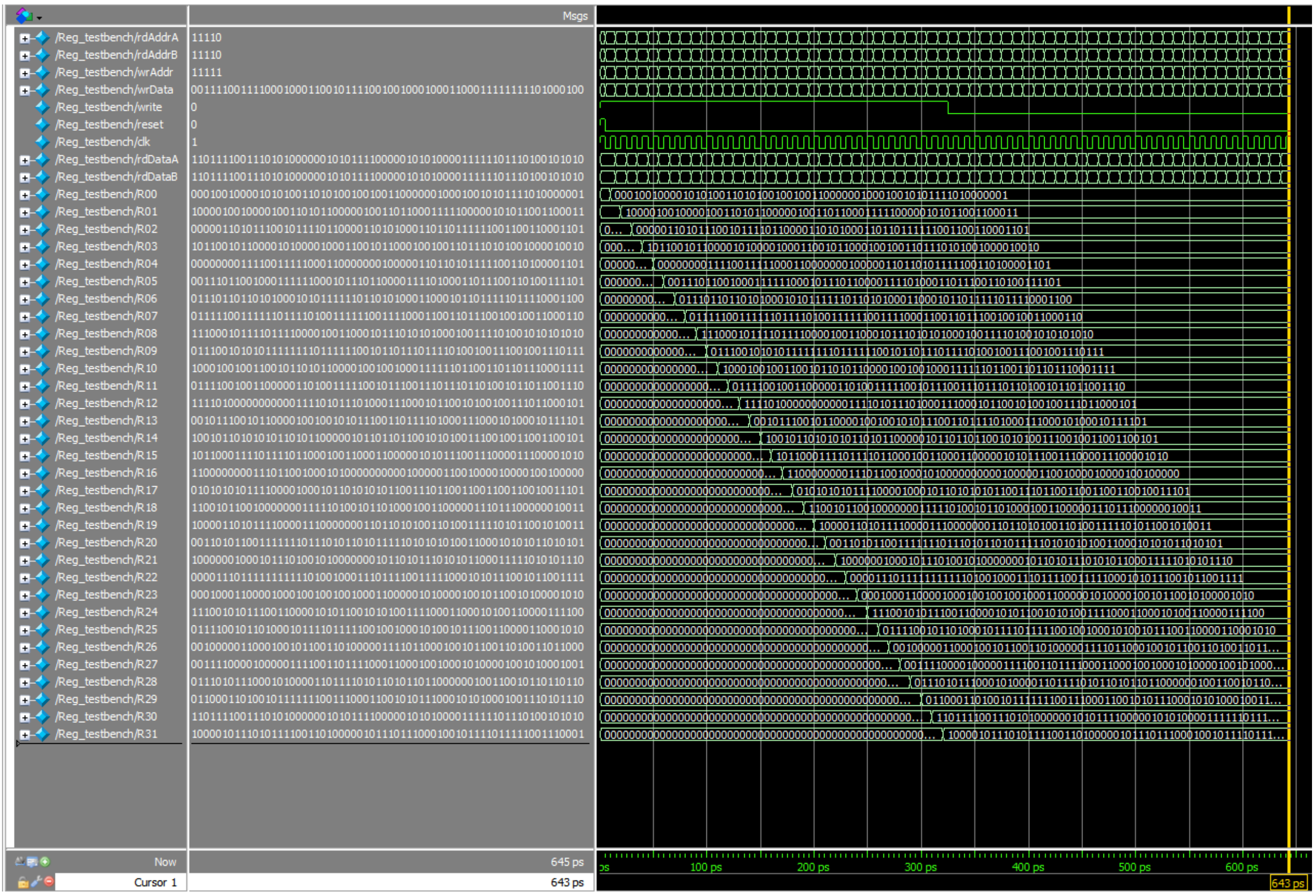
Figure 2.1. Register file block diagram.

2.2 Description of Design

This processor contains a register file housing 32 registers, each with 64 bits of memory. This processor's register file has two read busses, as well as one writes bus. To read data from the register file, the read address lines (rdAddrA and rdAddrB) need to be set to the 5-bit binary address of the register for which you wish to read from. Data from the registers will be made available to the ALU on the data output busses rdDataA and rdDataB. To write to the register file, the 5 bits write address (wrAddr) needs to be configured to the register number you wish to write to. The 64 bits of data to write will need to be present and stable on the write data bus (wrData), the write enable will need to be high, and on the next clock pulse, the data will be written to the selected register. All of the output and input busses to the register file are routed to the ALU, while the select addresses are routed to the control unit.

2.3 Testbench Screenshots and Description

2.3.1 Cristine

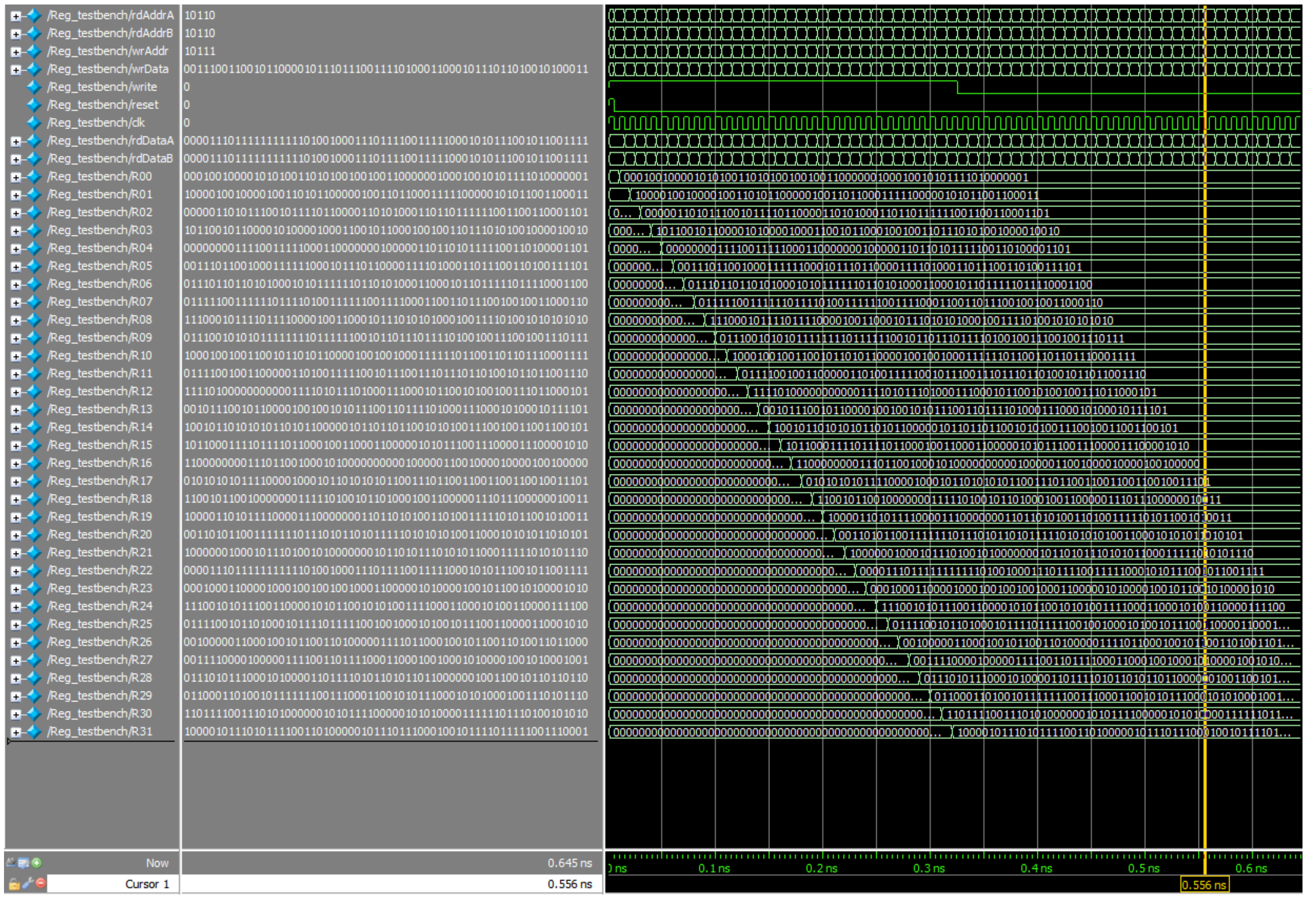


```
/Reg_testbench/rdAddrA   11110
/Reg_testbench/rdAddrB   11110
/Reg_testbench/wrAddr    11111
/Reg_testbench/wrData    001111001111000100011001011110010010001000110001111111101000100
/Reg_testbench/write     0
/Reg_testbench/reset     0
/Reg_testbench/clk       1
/Reg_testbench/rdDataA   1101111100111010100000010101111000001010100001111101110100101010
/Reg_testbench/rdDataB   1101111100111010100000010101111000001010100001111101110100101010
/Reg_testbench/R00       0001001000010101001101010010010011000000100010010101111010000001
/Reg_testbench/R01       1000010010000100110101100000100110110001111100000101011001100011
/Reg_testbench/R02       0000011010111001011110110000110101000110110111111001100110001101
/Reg_testbench/R03       1011001011000010100001000110010110001001001101110101001000010010
/Reg_testbench/R04       000000011110011111000110000000100000110110101111100110100001101
/Reg_testbench/R05       0011101100100011111100010111011000011110100011011100110100111101
/Reg_testbench/R06       0111011011010100010101111110110101000110001011011111011110001100
/Reg_testbench/R07       0111110011111101111010011111100111100011001101110010010011000110
/Reg_testbench/R08       1110001011110111100001001100010111010101000100111101001010101010
/Reg_testbench/R09       0111001010101111111101111110010110111011110100100111001001110111
/Reg_testbench/R10       1000100100110010110101100001001001000111111011001101101110001111
/Reg_testbench/R11       0111100100110000011010011111001011100111011101101001011011001110
/Reg_testbench/R12       11110100000000000011110101110100011100010110010100100111011000101
/Reg_testbench/R13       0010111001011000010010010101110011011110100011100010100010111101
/Reg_testbench/R14       1001011010101011010110000010110110110010101001110010011001100101
/Reg_testbench/R15       1011000111101111011000100110001100000101011100111000011100001010
/Reg_testbench/R16       1100000000111011001000101000000000010000011001000010000100100000
/Reg_testbench/R17       0101010101111000010001011010101011001110110011001100110010011101
/Reg_testbench/R18       1100101100100000001111101001011010001001100000111011100000010011
/Reg_testbench/R19       1000011010111100001110000000110110101001101001111101011001010011
/Reg_testbench/R20       0011010110011111110111010110101111101010101001100010101011010101
/Reg_testbench/R21       1000000100010111010010100000001011010111010101100011111010101110
/Reg_testbench/R22       0000111011111111111010010001110111100111110001010111001011001111
/Reg_testbench/R23       0001000110000100010010010010001100000101000010010110010100001010
/Reg_testbench/R24       11100101011100110000101011001010100111100011000010100110000111100
/Reg_testbench/R25       0111100101101000101111011111001001000101001011100110000110001010
/Reg_testbench/R26       0010000011000100101100110100000111101100010010110011010011011000
/Reg_testbench/R27       0011110000100000111100110111100011000100100010100001001010001001
/Reg_testbench/R28       0111010111000101000011011110101101011011000000100110010110110110
/Reg_testbench/R29       0110001101001011111110011100011001010111000101010001001110101110
/Reg_testbench/R30       1101111100111010100000010101111000001010100001111101110100101010
/Reg_testbench/R31       1000010111010111100110100000101110111000100101111011111001110001
```

Figure 2.2. Cristine's Register File Testbench Screenshot

Within the testbench, there was a random number generator, so we can make sure each register can be written to when the write input is high. It also makes sure that each register can be read from. The fifth line down shows when the write function is high and low. When it is high, you can see all the registers (lines 9 to the end) are all being written to with different numbers. Since the numbers are showing up, it also shows that the register files can be read from. Since none of the register files are being written to when the write function is low, we can conclude that the write input is working properly. With all the parts working individually, we can conclude that our register file works properly.

2.3.2 Krutik



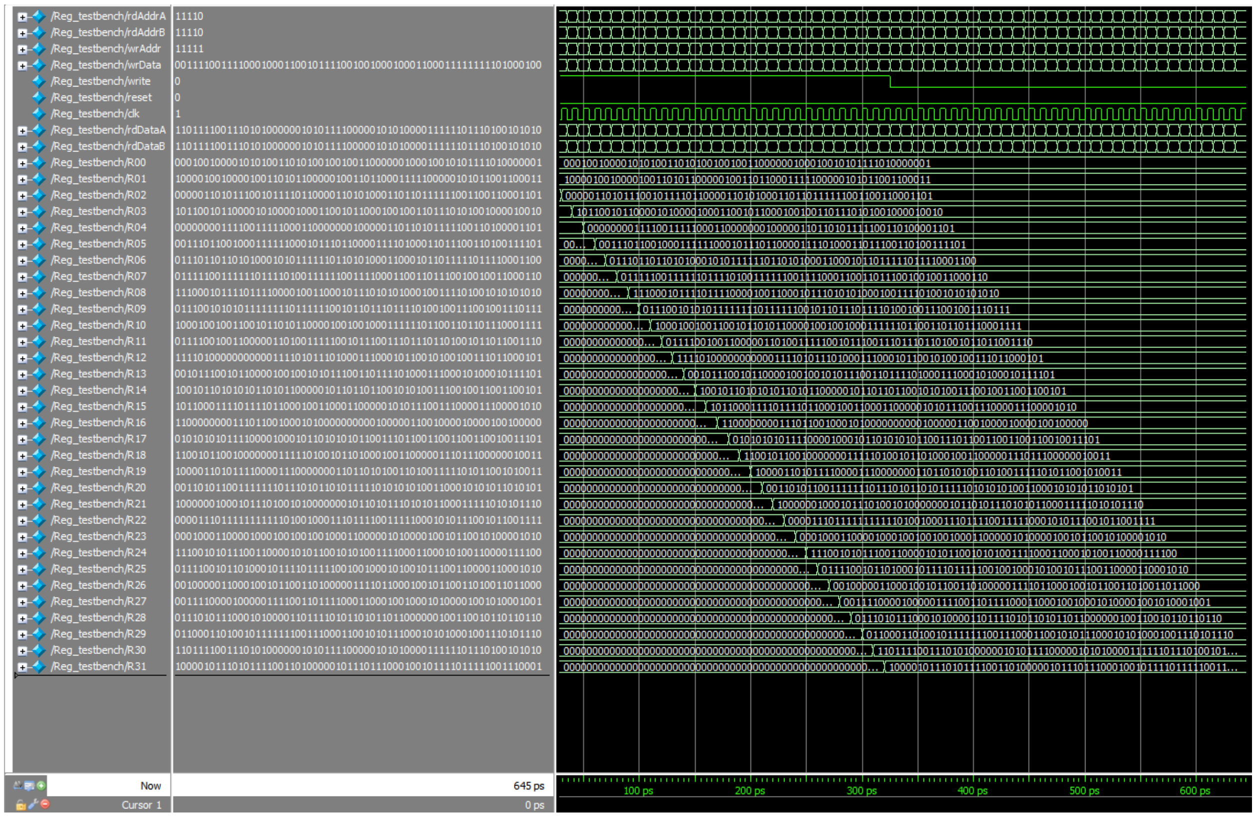
```
/Reg_testbench/rdAddrA   10110
/Reg_testbench/rdAddrB   10110
/Reg_testbench/wrAddr    10111
/Reg_testbench/wrData    0011100110010110000101110111001111010001100010111011010010100011
/Reg_testbench/write     0
/Reg_testbench/reset     0
/Reg_testbench/clk       0
/Reg_testbench/rdDataA   0000111011111111111010010001110111100111110001010111001011001111
/Reg_testbench/rdDataB   0000111011111111111010010001110111100111110001010111001011001111
/Reg_testbench/R00       0001001000010101001101010010010011000000100010010101111010000001
/Reg_testbench/R01       1000010010000100110101100000100110110001111100000101011001100011
/Reg_testbench/R02       0000011010111001011110110000110101000110110111111001100110001101
/Reg_testbench/R03       1011001011000010100001000110010110001001001101110101001000010010
/Reg_testbench/R04       0000000111100111110001100000000100000110110101111100110100001101
/Reg_testbench/R05       0011101100100011111100010111011000011110100011011100110100111101
/Reg_testbench/R06       0111011011010100010101111110110101000110001011011111011110001100
/Reg_testbench/R07       0111110011111101111010011111100111100011001101110010010011000110
/Reg_testbench/R08       1110001011110111100001001100010111010101000100111101001010101010
/Reg_testbench/R09       0111001010101111111110111111001011011101111010010011100100111011
/Reg_testbench/R10       1000100100110010110101100001001001000111111011001101101110001111
/Reg_testbench/R11       0111100100110000011010011111001011100111011101101001011011001110
/Reg_testbench/R12       1111010000000000001111010111010001110001011001010010011101100010
/Reg_testbench/R13       0010111001011000010010010101110011011110100011100010100010111101
/Reg_testbench/R14       1001011010101011010110000010110110110010101001110010011001100101
/Reg_testbench/R15       1011000111101111011000100110001100000101011100111000011100001010
/Reg_testbench/R16       1100000000111011001000101000000000001000001100100001000010100000
/Reg_testbench/R17       0101010101111000010001011010101011001110110011001100110010011101
/Reg_testbench/R18       1100101100100000001111101001011010001001100000111011100000010011
/Reg_testbench/R19       1000011010111100001110000000110110101001101001111101011001010011
/Reg_testbench/R20       0011010110011111110111010110101111010101010011000101010110101101
/Reg_testbench/R21       1000000100010111010010100000001011010111010101100011111010101110
/Reg_testbench/R22       0000111011111111111010010001110111100111110001010111001011001111
/Reg_testbench/R23       0001000110000100010010010010001100000101000010010110010100001010
/Reg_testbench/R24       1110010101110011000010101100101010011110001100010100110000111100
/Reg_testbench/R25       0111100101101000101111011111001001000101001011100110000110001010
/Reg_testbench/R26       0010000011000100101100110100000111101100010010110011010011011000
/Reg_testbench/R27       0011110000100000111100110111100011000100100010100001001010001001
/Reg_testbench/R28       0111010111000101000011011110101101011011000000100110010110110110
/Reg_testbench/R29       0110001101001011111110011100011001010111000101010001001110101110
/Reg_testbench/R30       1101111001110101000000101011110000010101000011111101110100101010
/Reg_testbench/R31       1000010111010111100110100000101110111000100101111011111001110001

Now          0.645 ns
Cursor 1     0.556 ns
```

Figure 2.3. Krutik's Register File Testbench Screenshot

Similarly, to Cristine's testbench simulation, the random number generator is used to write to the registers using randomly generated values. The registers are being written to when the write input is high, and the registers will stop taking data when the write input is low. Since the numbers are visible for all registers, that means the read variables are working as well. The figure above is just the output values being read from the register. When the registers are being read a constant line of zeros, that just means the register has yet to be written to at that point in time. Every time the clock goes high the next register will get data, as long as the write enable is high as well. This testbench is merely just writing to and reading from the 64-bit registers.

2.3.3 Logan



```
/Reg_testbench/rdAddrA   11110
/Reg_testbench/rdAddrB   11110
/Reg_testbench/wrAddr    11111
/Reg_testbench/wrData    0011110011110001000110010111100100100010001100011111111101000100
/Reg_testbench/write     0
/Reg_testbench/reset     0
/Reg_testbench/clk       1
/Reg_testbench/rdDataA   1101111001110101000000101011110000010101000011111101110100101010
/Reg_testbench/rdDataB   1101111001110101000000101011110000010101000011111101110100101010
/Reg_testbench/R00       0001001000010101001101010010010011000000100010010101111010000001
/Reg_testbench/R01       1000010010000100110101100000100110110001111100000101011001100011
/Reg_testbench/R02       0000011010111001011110110000110101000110110111111001100110001101
/Reg_testbench/R03       1011001011000010100001000110010110001001001101110101001000010010
/Reg_testbench/R04       0000000111100111110001100000001000001101101011111001101000011101
/Reg_testbench/R05       0011101100100011111100010111011000011110100011011100110100111101
/Reg_testbench/R06       0111011011010100010101111110110101000110001011011111011110001100
/Reg_testbench/R07       0111110011111101111010011111100111100011001101110010010011000110
/Reg_testbench/R08       1110001011110111100001001100010111010101000100111101001010101010
/Reg_testbench/R09       0111001010101111111101111110010110111011110100100111001001110111
/Reg_testbench/R10       1000100100110010110101100001001001000111111011001101101110001111
/Reg_testbench/R11       0111100100110000011010011111001011100111011101101001011011001110
/Reg_testbench/R12       1111010000000000001111010111010001110001011001010010011101100010 1
/Reg_testbench/R13       0010111001011000010010010101110011011110100011100010100010111101
/Reg_testbench/R14       1001011010101011010110000010110110110010101001110010011001100101
/Reg_testbench/R15       1011000111101111011000100110001100000101011100111000011100001010
/Reg_testbench/R16       1100000000111011001000101000000000001000001100100001000010010000 0
/Reg_testbench/R17       0101010101111000010001011010101011001110110011001100110010011101
/Reg_testbench/R18       1100101100100000001111101001011010001001100000111011100000010011
/Reg_testbench/R19       1000011010111100001110000000110110101001101001111101011001010011
/Reg_testbench/R20       0011010110011111111011101011010111110101010100110001010101101010 1
/Reg_testbench/R21       1000000100010111010010100000001011010111010101100011111010101110
/Reg_testbench/R22       0000111011111111111101001000111011110011111000101011100101100111 1
/Reg_testbench/R23       0001000110000100010010010010001100000101000010010110010100001010
/Reg_testbench/R24       1110010101110011000010101100101010011110001100010100110000111100
/Reg_testbench/R25       0111100101101000101111011111001001000101001011100110000110001010
/Reg_testbench/R26       0010000011000100101100110100000111101100010010110011010011011000
/Reg_testbench/R27       0011110000100000111100110111100011000100100010100001001010001001
/Reg_testbench/R28       0111010111000101000011011110101101011011000000100110010110110110
/Reg_testbench/R29       0110001101001011111111001110001100101011100010101000100111010111 0
/Reg_testbench/R30       1101111001110101000000101011110000010101000011111101110100101010
/Reg_testbench/R31       1000010111010111100110100000101110111000100101111011111001110001

Now        645 ps
Cursor 1   0 ps
```

Figure 2.4. Logan's Register File Testbench Screenshot

As you can see in Figure 2.4 above, the testbench is storing and extracting data from the register file. To test the register file with many different scenarios, a random number generator was created in the testbench file. The random number generator was configured to generate 64-bit binary numbers and write them into the 32 registers of the register file (numbered 0 through 31). Looking at the results on the right side of the testbench, we can see the triangle of zeros on the left side of the output window. Since all registers started with 64 zeros in them, this makes sense to see. Since only one register is written at a time, we can see the triangle effect that advances down one step each time the clock goes high. We can see that at once all the registers have been written, the write line is pulled to zero, and all of the registers hold their data as they are supposed to.

3 ALU Design

3.1 Figure of ALU

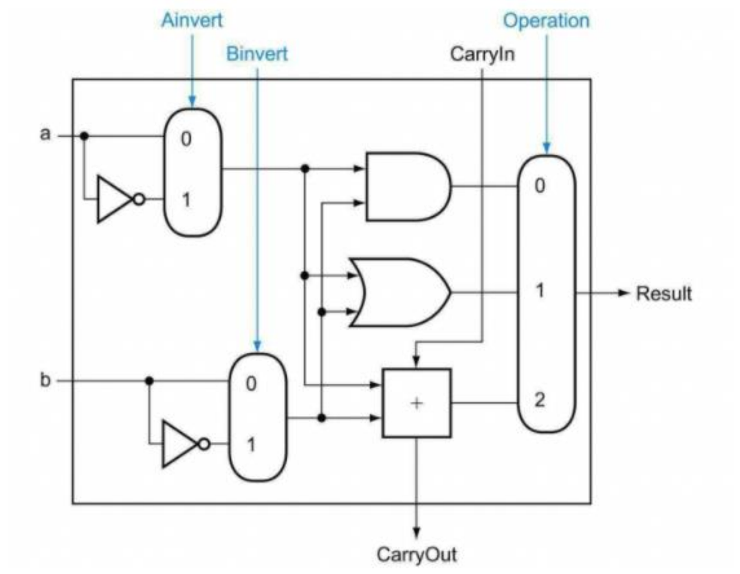


Figure 3.1. General ALU Layout

3.2 Function Select Codes and RTL Operations

To understand what function to preform, the ALU needs to have its function select bits set by the control unit. The 5-bit options below in Table 3.1 show the function select bits required for the desired RTL operations in the right column to take place.

Table 3.1. Function selects codes and RTL Operations.

Function Select	RTL Operation
01000	A+B
01001	A-B
11000	0
00000	A&B
00100	A|B
01100	A^B
11100	16’b1111111111111111
10000	A>>1
10100	A<<1

3.3 Description of Design and Optimization Steps

The processor’s arithmetic logic unit (ALU) is the heart of the processor. Its job is to perform the requested mathematical operation provided by the programmer in assembly form. The ALU has two 64-bit busses as inputs that are passed to it from the register file. To control the operation to perform with the inputs, there is a 5-bit function select input. When processing the function select input, the processor uses the first three bits (bits 4:2) to determine which operation to perform. The last two bits of the input (bits 1:0) are used denote whether the ALU needs to invert the A or B bus inputs.

The ALU has two outputs, the 64-bit output data bus and a 4-bit status output. The 64-bit output data bus is routed to the write data bus of the register file, while the status signal is routed to the control unit. As you can see in Table 3.2 below, the four status lines each have their own purpose, and are all fed into the control unit.

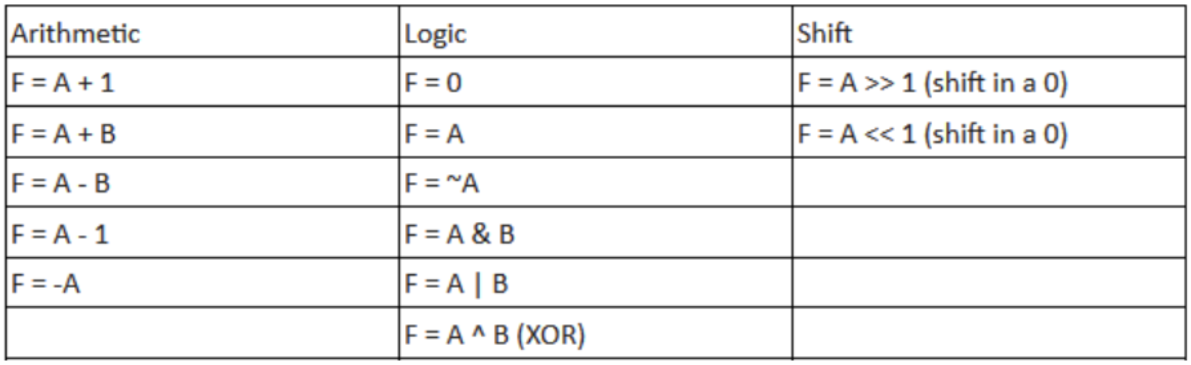
Table 3.2. Status bit functions

Stat bit	Signal	Description
0	Z	(zero) the result from operation is zero.
1	N	(negative) indicates if the result was negative.
2	C	(Carry out) indicates overflow of unsigned arithmetic.
3	V	Indicated overflow of signed arithmetic.

3.4 Testbench Design

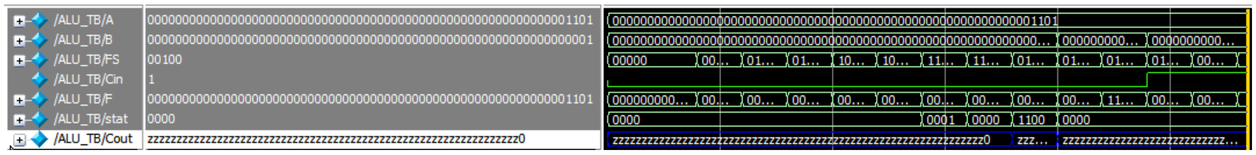
The testbenches below showcase different functions of the ALU. Note that not all the functions that the ALU can compute are shown below, however we have tried to provide some of the core functions in the testbenches shown. Below is Table 3.3 which shows all of the logic operations that the ALU can compute.

Table 3.3. List of necessary ALU operations



Arithmetic	Logic	Shift
F = A + 1	F = 0	F = A >> 1 (shift in a 0)
F = A + B	F = A	F = A << 1 (shift in a 0)
F = A - B	F = ~A	
F = A - 1	F = A & B	
F = -A	F = A | B	
	F = A ^ B (XOR)	

3.4.1 Cristine

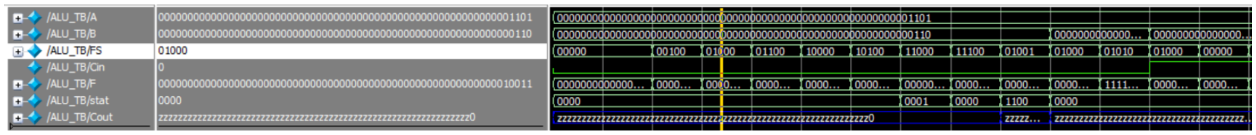


```
/ALU_TB/A     0000000000000000000000000000000000000000000000000000000000001101
/ALU_TB/B     0000000000000000000000000000000000000000000000000000000000000001
/ALU_TB/FS    00100
/ALU_TB/Cin   1
/ALU_TB/F     0000000000000000000000000000000000000000000000000000000000001101
/ALU_TB/stat  0000
/ALU_TB/Cout  zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz0
```

Figure 3.2. Cristine's ALU Testbench

The lines in order are variables A, B, FS (function select), Cin, F(output), stat, Cout. To determine if the value is correct, you have to test all of the function select bits and test different values of A and B. For the example that is selected on, A = 1101, B=1, FS=00100, Cin=1, F=1101, stat =0000 and Cout=0. The function select bits are selected on the or function. This means is either A or B has a 1 value the output will have the same output value. This is correct for our function since all of the positions with 1's in it has 1. The stat being 0000 means that it does not meet any of the signal description; it is our default function. It means that the result is not zero or negative and there is no overflow. This is true for this case.

3.4.2 Krutik

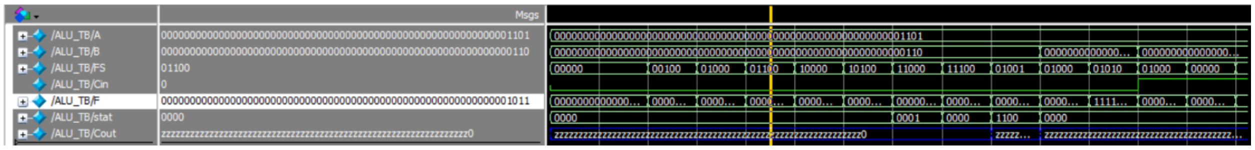


```
/ALU_TB/A     0000000000000000000000000000000000000000000000000000000000001101
/ALU_TB/B     0000000000000000000000000000000000000000000000000000000000000110
/ALU_TB/FS    01000
/ALU_TB/Cin   0
/ALU_TB/F     0000000000000000000000000000000000000000000000000000000000010011
/ALU_TB/stat  0000
/ALU_TB/Cout  zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz0
```

Figure 3.3. Krutik's ALU Testbench

When FS is selected at 01000, this operation is A+B. A = 1101, and B = 0110. Theoretically, since this is a 64-bit number, there is no carry bit and the last 5 bits of the operation would result in 10011 when you add A and B, and this is evident in the testbench simulation above. Since the result is not zero, negative, and no overflow, the status signal is 0000. This testbench simulation shows that it is doing the required functions.

3.4.3 Logan



```
                                                                              Msgs
/ALU_TB/A     0000000000000000000000000000000000000000000000000000000000001101
/ALU_TB/B     0000000000000000000000000000000000000000000000000000000000000110
/ALU_TB/FS    01100
/ALU_TB/Cin   0
/ALU_TB/F     0000000000000000000000000000000000000000000000000000000000001011
/ALU_TB/stat  0000
/ALU_TB/Cout  zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz0
```

Figure 3.4. Logan's ALU Testbench

In Figure 3.4 above, you can see that the ALU is preforming an XOR operation using data from a simulated register file. Input A is set to 1101 and input B is set to 0110. Computing the XOR of these two 4-bit numbers gives the result of 1011, which we can see the ALU reports on the "F" line above in the grey region. The ALU also output the status as 0000, which is correct for this combination of inputs and outputs. This shows that the ALU's XOR function is working correctly. Other team members have tested different functions of the ALU in their testbenches.

4 Memory Organization

4.1 Memory Address Spaces

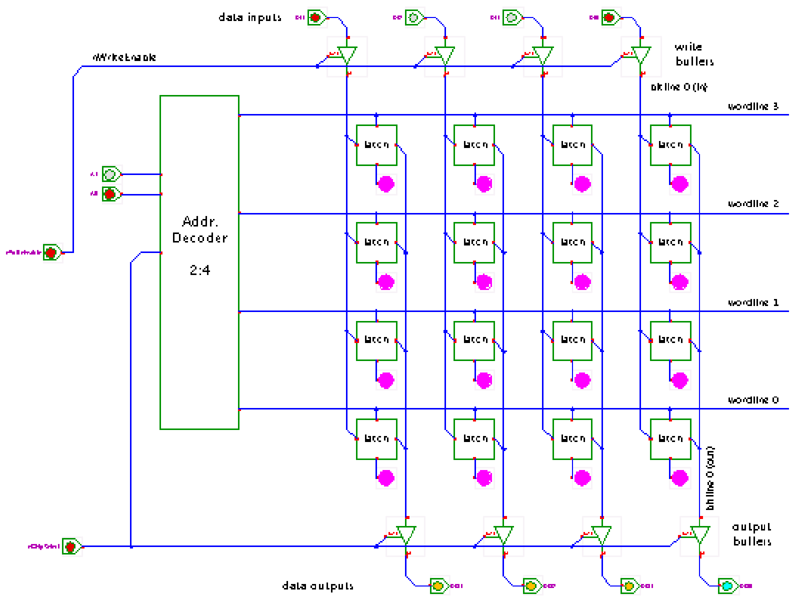
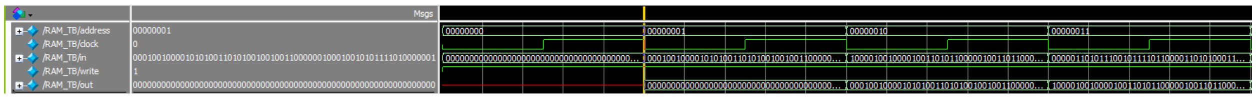


Figure 4.1. Schematic of RAM

4.2 Design and Implementation

When RAM is placed into the top-level entity, it acts as a short-term data storage that you can write to. The RAM stores the information that your computer is currently using. The bigger the RAM, the more programs the computer can run since it can allocate more memory to it. For our specific RAM, the memory is 255 bits with 64 different registers. It has a write function to prevent the function from being written to and read from simultaneously.

4.3 Testbench



```
                   Msgs
/RAM_TB/address    00000001
/RAM_TB/clock      0
/RAM_TB/in         0001001000010101001101010010010011000000100010010101111010000001
/RAM_TB/write      1
/RAM_TB/out        0000000000000000000000000000000000000000000000000000000000000000

/RAM_TB/address    00000000 | 00000001 | 00000010 | 00000011
/RAM_TB/in         00000000000000000000000000000000000... | 0001001000010101001101010010010011000000... | 1000010010000100110101100000100110110000... | 000001101011100101111011000011010100011...
/RAM_TB/out        | 000000000000000000000000000000000000000... | 0001001000010101001101010010010011000000... | 1000010010000100110101100000100110110000...
```

Figure 4.2. RAM Testbench

To test the processor's memory, we created a testbench that wrote and read random data to the RAM to make sure that it was fully functional. One set of results can be seen above in Figure 4.2. The RAM address to write to is incremented each clock cycle as you can see on the first line of the testbench. The RAM data input line is connected to a 64-bit random number generator. The write line on the RAM is pulled high, so each time we change the address and data, the data is written to the specified address. We can verify that this is working by looking at the RAM output line. For the first clock cycle, we see a red line – this is normal since there is nothing in the RAM at this point. The RAM is configured to output values on the negative clock edge, while it writes values on the positive clock edge as it cannot do both simultaneously, it will cause errors when reading and writing to the same memory address. After the first negative clock edge, we can see that the output of the memory is what the input was before (in the first case – all zeros). During the next clock cycle, the random number is changed, and once the clock edge falls, you can see that the RAM reports the same random number on the output line.

5 Datapath

5.1 Figure of Instruction Register

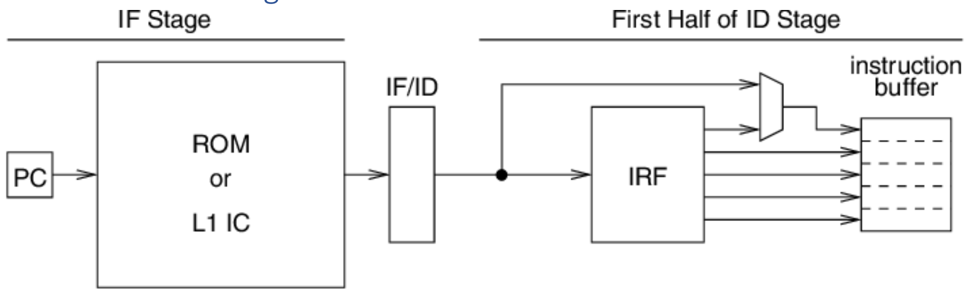
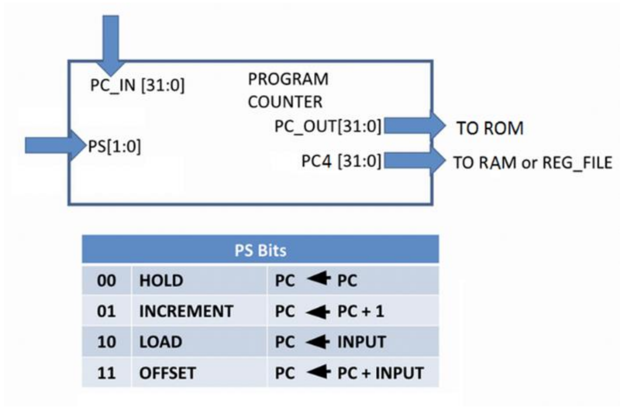


Figure 5.1. Process of Instructor Register

5.2 Figure of Program Counter Design and Description



PS Bits		
00	HOLD	PC ← PC
01	INCREMENT	PC ← PC + 1
10	LOAD	PC ← INPUT
11	OFFSET	PC ← PC + INPUT

Figure 5.2. Diagram of Program Counter (Note all 32-bit inputs should be 64-bit)

The program counter is the portion of the processor that keeps track of what line of assembly in the program is being executed. The program counter increments by 4 for each instruction it runs.

5.3 Figure of Datapath

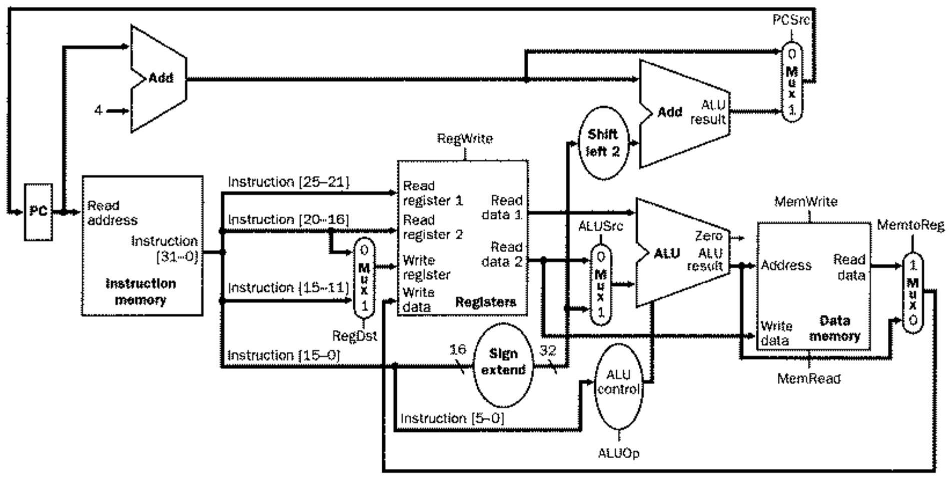


Figure 5.3. ARM Processor Datapath

5.4 Control Word

assign {PS, DA, SA, SB, FS, regW, ramW, EN_MEM, EN_ALU, EN_B, EN_PC, selB, PCsel, SL} = controlWord;

Table 5.1. Control Signals, Their Function and What Each Value Does

Control Word Portion	Name	Description	Values

[30:29]	PS	Program Counter Control	`00: PC <- PC`
			`01: PC <- PC + 4`
			`10: PC <- in`
			`11: PC <- PC + 4 + in * 4`

[28:24]	DA	Register Data Address (Write Register #)	`5-bit select for 32 registers`
[23:19]	SA	Register A Address (Read Register # A)	`5-bit select for 32 registers`
[18:14]	SB	Register B Address (Read Register # B)	`5-bit select for 32 registers`

[13:9]	FS	Function Select	`FS[0]: B Invert`
			`FS[1]: A Invert`
			`000XX: AND`
			`001XX: OR`
			`010XX: ADD`
			`011XX: XOR`
			`100XX: SHIFT LEFT`
			`101XX: SHIFT RIGHT`

[8]	regW	Write to Register (Write)	`0: Don’t write`
			`1: Write`

[7]	ramW	Write to RAM (RAMWrite)	`0: Don’t write`
			`1: Write`

[6]	EN_MEM	Enable RAM on Data Bus	`0: Don’t use`
			`1: Use`

[5]	EN_ALU	Enable ALU on Data Bus	`0: Don’t use`
			`1: Use`

[4]	EN_B	Enable Register B output on Data Bus	`0: Don’t use`
			`1: Use`

[3]	EN_PC	Enable PC+4 output on Data Bus	`0: Don’t use`
			`1: Use`

[2]	selB	Select Register B / Literal K for ALU	`0: Output B`
			`1: Output K`

[1]	PCsel	Select Register A / Literal K for Program Counter	`0: Output A`
			`1: Output K`

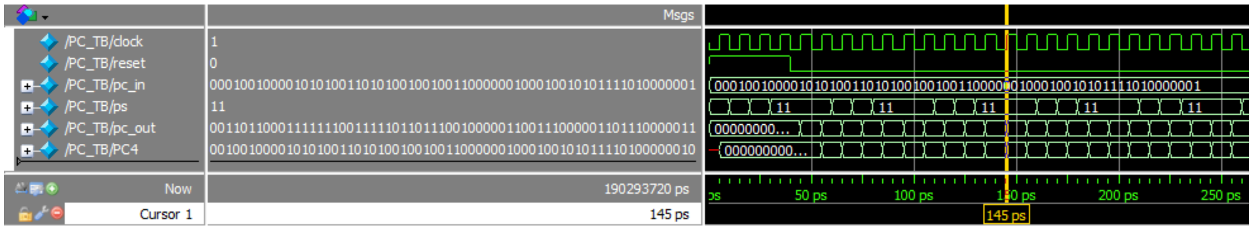
[0]	SL	Status Lines from ALU	`0: Don't use`
			`1: Use`

5.5 Description of Design Choices and Capabilities of Datapath

The Datapath is determined by the control word. Based on the control word, different values will be set to all of the variables. It does this by accessing different modules within the program. This will allow to processor to do operations like addition, branching and moving.

5.6 Testbench

We did not finish the control unit so we could not build a complete Datapath. But we did finish the program counter, so here is a testbench for that.



```
                                                                                      Msgs
/PC_TB/clock     1
/PC_TB/reset     0
/PC_TB/pc_in     0001001000010101001101010010010011000000100010010101111010000001
/PC_TB/ps        11
/PC_TB/pc_out    0011011000111111100111110110111001000001100111000001101110000011
/PC_TB/PC4       0010010000101010011010100100100110000000100010010101111010000010

Now                                                                          190293720 ps
Cursor 1                                                                           145 ps
```

Figure 5.4. Program counter testbench.

As you can see in Figure 5.4 above, the program counter is counting by 4 each time a clock cycle passes. When the processor is reset, it clears out the program counter, so the code execution returns to the first line.

6 Control Unit

6.1 Figure(s) of Control unit design

Control Unit of a Basic Computer:

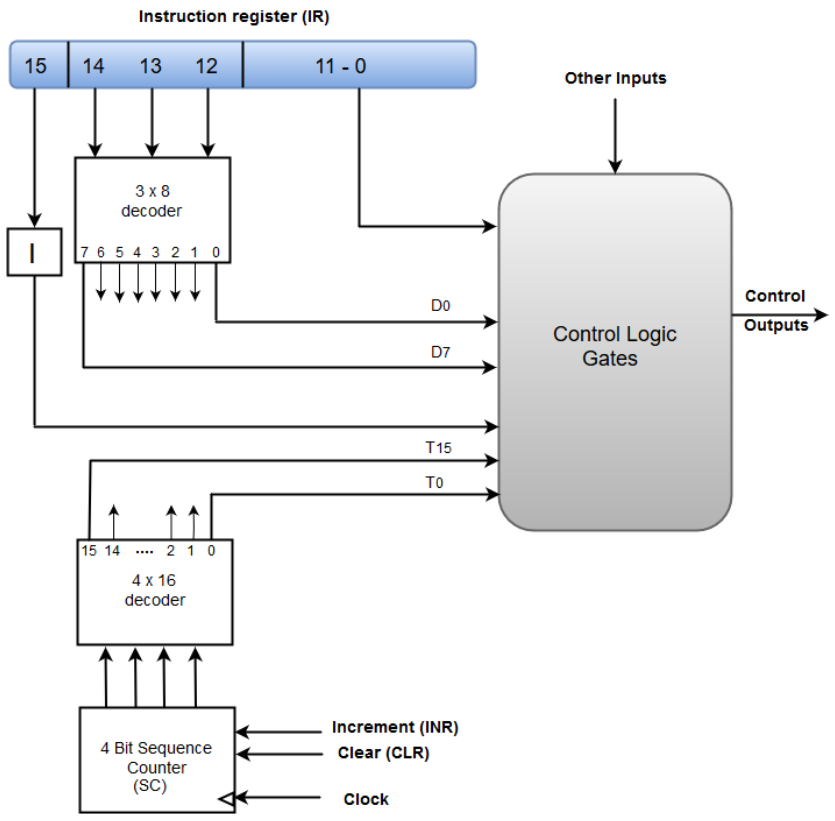


Figure 6.1. Control unit depiction

6.2 State diagram of control unit

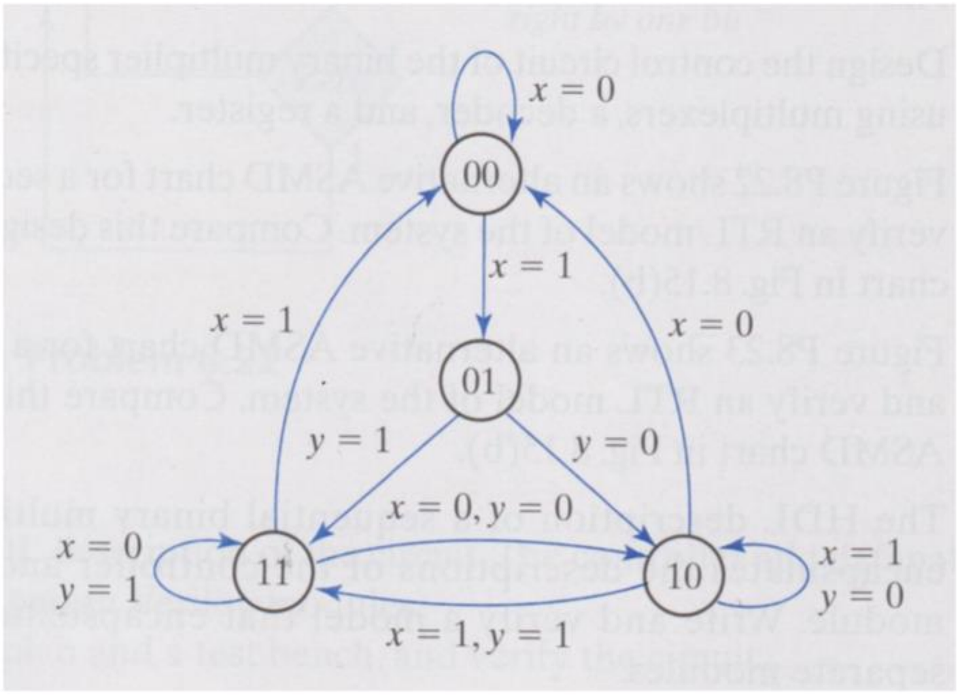


Figure 6.2. Control unit state diagram

6.3 Description of design approach

For each type data type, we created a table. Within each data type, there are different operations that needed to be accomplished. For each one of those there was a column made. From there, each column was filled with the values needed to achieve those.

After all of the tables were filled, it was transferred to code. Each operation got their own module, which would set the values needed to the respective variable.

6.4 Description of CU registers

Table 6.1. Control Signal Descriptions

Abbreviation	Meaning
PS	Selecting which program counter, we are using.
DA	Data within register a
SA	Address of register a
SB	Address of register b
FS	Function select (operation of the ALU)
regW	If you are writing to the registers
ramW	If you are writing to the Ram
EN_MEM	Enabling memory
EN_ALU	Enabling the use of the ALU
EN_B	Enabling the use of b
EN_PC	Enabling the program counter
selB	If you are selecting register b o r the immediate generator

PCsel	Program counter select
SL	Status of ALU
nextS	The next state of the control unit

6.5 Description of testing done and testbench design.

For the test bench, we tested each control word individually. Mathematically, we can determine what the output values would be from the input values. Our hand calculations would then be compared to the outcome. If they were the same, then we knew it was correct.

7 CPU

7.1 Figure of Datapath, control unit, and memories.

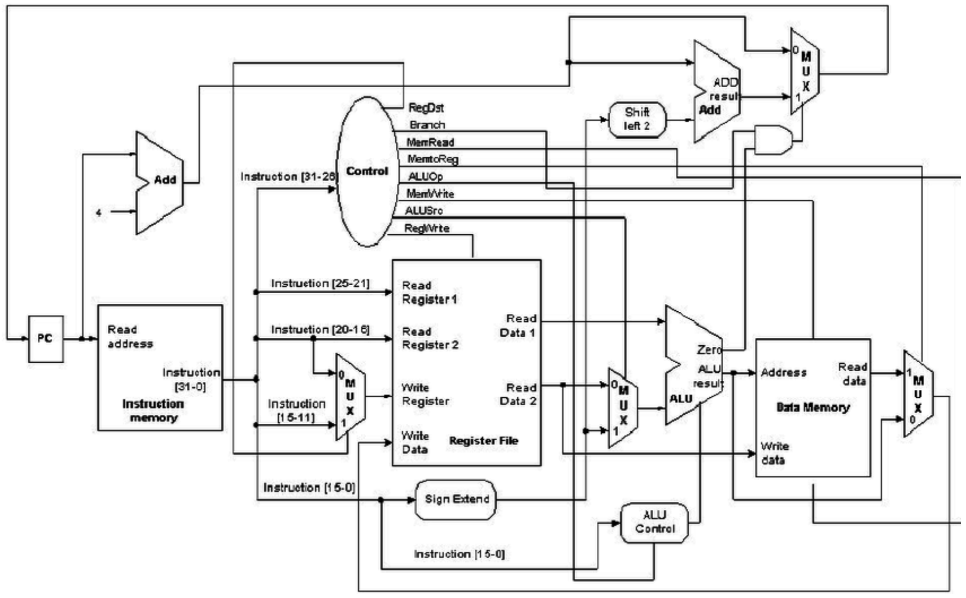


Figure 7.1. CPU Block Diagram

7.2 Description of testing done to validate the CPU and the testbench design.

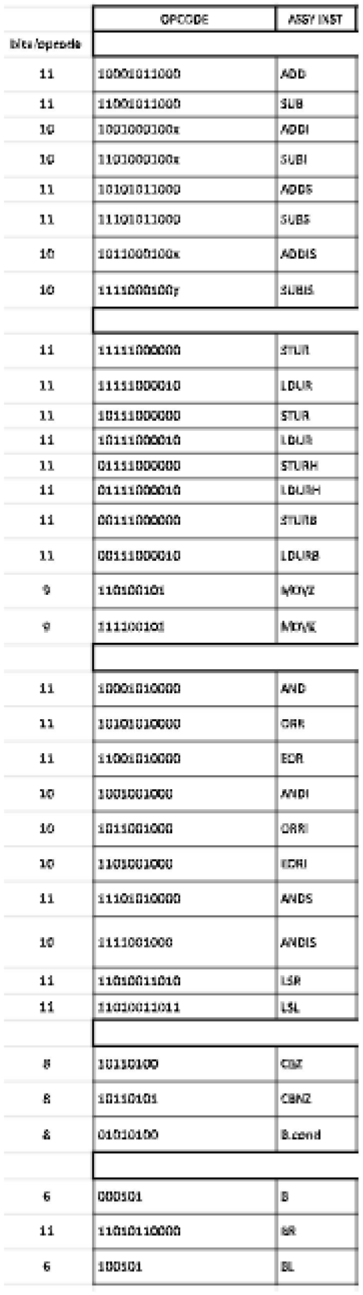
The best way for us to test the CPU is for us to write a meaningful program in assembly and run it on the CPU. Since we were unable to complete the datapath, we were unable to run code on our CPU.

8 Instruction Set

8.1 Section for common Instructions:

8.1.1 Table of opcode field descriptions / terminology

Table 8.1. Assembly Instructions to Opcode



	OPCODE	ASSY INST
bits/opcode		
11	10001011000	ADD
11	11001011000	SUB
10	1001000100x	ADDI
10	1101000100x	SUBI
11	10101011000	ADDS
11	11101011000	SUBS
10	1011000100x	ADDIS
10	1111000100y	SUBIS
11	11111000000	STUR
11	11111000010	LDUR
11	10111000000	STUR
11	10111000010	LDUR
11	01111000000	STURH
11	01111000010	LDURH
11	00111000000	STURB
11	00111000010	LDURB
9	110100101	MOVZ
9	111100101	MOVK
11	10001010000	AND
11	10101010000	ORR
11	11001010000	EOR
10	1001001000	ANDI
10	1011001000	ORRI
10	1101001000	EORI
11	11101010000	ANDS
10	1111001000	ANDIS
11	11010011010	LSR
11	11010011011	LSL
8	10110100	CBZ
8	10110101	CBNZ
8	01010100	B.cond
6	000101	B
11	11010110000	BR
6	100101	BL

8.1.2 Figure of instruction formats

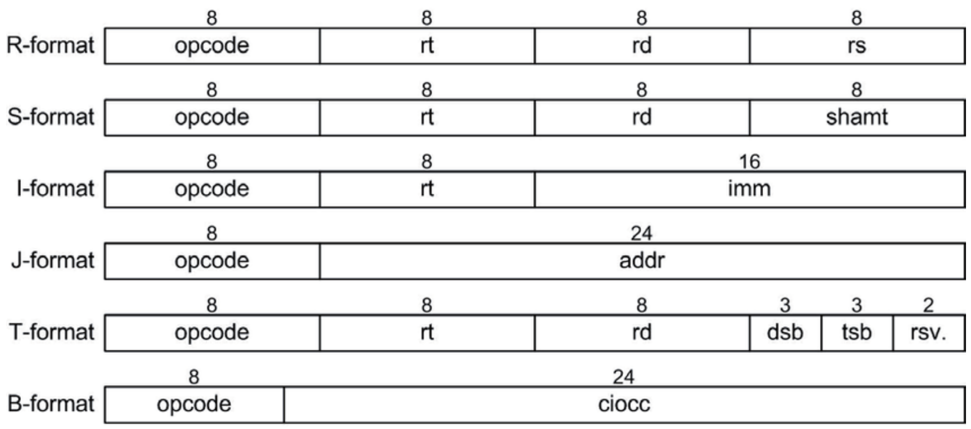
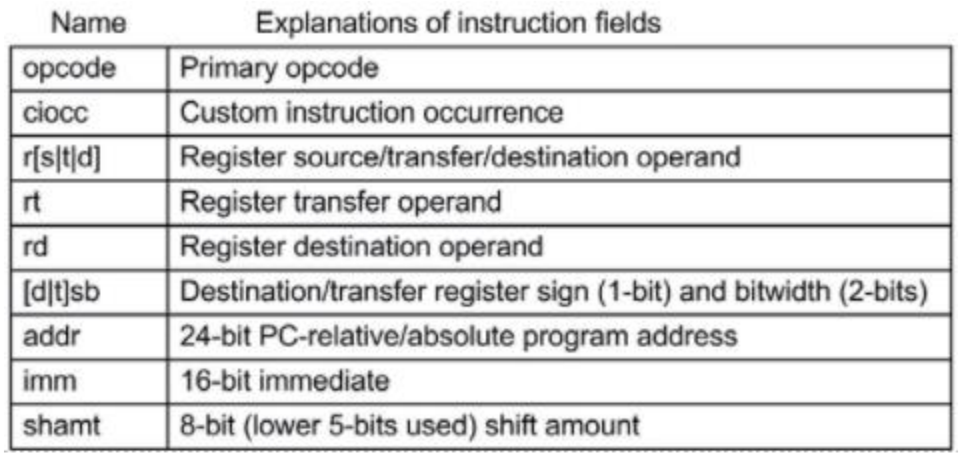


Figure 8.1. Bitfield addresses for different instruction formats

8.1.3 Table of instruction set summary

Table 8.2. Instruction Set Summary



Name	Explanations of instruction fields
opcode	Primary opcode
ciocc	Custom instruction occurrence
r[s|t|d]	Register source/transfer/destination operand
rt	Register transfer operand
rd	Register destination operand
[d|t]sb	Destination/transfer register sign (1-bit) and bitwidth (2-bits)
addr	24-bit PC-relative/absolute program address
imm	16-bit immediate
shamt	8-bit (lower 5-bits used) shift amount

8.1.4 Table of instructions and control words

Table 8.3. Branch Control Word

	B	BL	B.cond	CBZ	CBNZ	BR
PS	11	11				
DA	11111	11111	xxxxx	xxxxx	xxxxx	xxxxx
SA	11111	11111	xxxxx	xxxxx	xxxxx	xxxxx
SB	11111	11111	xxxxx	xxxxx	xxxxx	xxxxx
FS	00000	00000	00000	00000	00000	00000
regW	0	1	0	1	1	0
ramW	0	0	1	1	1	1
EN_MEM	0	0	0	0	0	0

EN_ALU	0	1	1	1	1	1
EN_B	0	0	0	1	1	1
EN_PC	0	1	1	0	0	0
selB	0	0	0	1	1	0
PCsel	1	1	0	0	0	0
SL	0	0	1	1	0	0
nextS	00	00	00	00	00	00

Table 8.4. Arithmetic Control Word

	Add	Sub	Adds	Subs	And	ORR	EOR	ANDS	LSR	LSL
PS	01	01	01	01	01					
DA	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
SA	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
SB	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx	xxxxx
FS	01000	01001	01000	01001	00000	00100	00100	00000	01011	01011
regW	1	1	1	1	1	1	1	1	1	1
ramW	0	0	0	0	0	0	0	0	0	0
EN_MEM	0	0	0	0	0	0	0	0	0	0
EN_ALU	1	1	1	1	1	1	1	1	1	1
EN_B	1	1	0	0	1	1	1	0	1	1
EN_PC	1	1	1	1	1	1	1	1	1	1
selB	0	0	0	0	0	0	0	0	0	0
PCsel										
SL	1	1	1	1	1	1	1	1	1	1
nextS	00	00	01	01	00	00	00	00	00	00

Table 8.5. Arithmetic Immediate Control Word

	Addi	Subi	Addis	Subis
PS	01	01	01	01
DA	xxxxx	xxxxx	xxxxx	xxxxx
SA	xxxxx	xxxxx	xxxxx	xxxxx
SB	xxxxx	xxxxx	xxxxx	xxxxx
FS	01000	01001	01000	01001
regW	1	1	1	1
ramW	0	0	0	0
EN_MEM	0	0	0	0
EN_ALU	1	1	1	1
EN_B	1	1	1	1
EN_PC	1	1	1	1
selB	0	0	0	0
PCsel	0	0	0	0

SL	10	10	10	10
nextS	01	00	01	00

Table 8.6. Logical Immediate Control Word

	ANDI	ORI	EORI	ANDIS
PS	01			
DA	xxxxx			
SA	xxxxx			
SB	xxxxx			
FS	00000			
regW	1			
ramW	0			
EN_MEM	0			
EN_ALU	1			
EN_B	0			
EN_PC	1			
selB	0			
PCsel				
SL	1	1	1	1
nextS	00	00	01	11

Table 8.7. Wide Immediate Control Word

	MOVZ	MOVK
PS	01	01
DA	xxxxx	xxxxx
SA	xxxxx	xxxxx
SB	xxxxx	xxxxx
FS	0100	0100
regW	1	1
ramW	0	0
EN_MEM	0	0
EN_ALU	1	1
EN_B	1	1
EN_PC	1	1
selB	1	1
PCsel	0	1
SL	0	0
nextS	01	01

Table 8.8. Data Transfer Control Word

	STUR	LDUR
PS	11	11

DA	Xxxxx	Xxxxx
SA	Xxxxx	xxxxx
SB	xxxxx	Xxxxx
FS	xxxxx	xxxxx
regW	1	1
ramW	1	1
EN_MEM	0	0
EN_ALU	0	0
EN_B	0	0
EN_PC	0	0
selB	0	0
PCsel	0	1
SL	0	0
nextS	01	01

8.1.5 Detailed instruction set list and descriptions

Table 8.9. Instruction Categories and Operations

Category	Operations
Branch	B - Branch
Branch with Link	BL - Branch with Link
Branch Conditional	B.cond - Branch Conditionally
Compare & Branch	CBZ - Compare & Branch on = 0 CBNZ - Compare & Branch on not = 0
Branch to Register	BR - Branch to Register
Arithmetic	ADD - Add SUB - Subtract ADDS - Add & Set Flags SUBS - Subtract & Set Flags AND - Logic AND ORR - Logic OR EOR - Logic XOR ANDS - Logic AND & Set Flags LSR - Logical Shift Right LSL - Logical Shift Left
Immediate Arithmetic	ADDI - Add Immediate SUBI - Subtract Immediate ADDIS - Add Immediate & Set Flags SUBIS - Subtract Immediate & Set Flags
Immediate Logic	ANDI - Logic AND Immediate ORRI - Logic OR Immediate EORI - Logic XOR Immediate ANDIS - Logic AND Immediate & Set Flags

Wide Immediate	MOVZ - Move Wide with Zero MOVK - Move Wide with Keep
Data Transfer	STUR - Store Register LDUR - Load Register

MOVZ moves an immediate value (16-bit value) to a register, and all the other bits outside the immediate value are set to Zero.

MOVK moves an immediate value (16-bit value) to a register, and all the other bits outside the immediate value remain the same.

9 Programming Examples

9.1 Section for each team member's program

Each member worked on the code together to make one collective script.

9.1.1 Table of assembly instructions and binary instruction words

Refer to the tables within section 8.1.4.

9.1.2 Description of what your program does

The program takes in a control word which dictates the Datapath. This will determine what signals are off and on. From there values are added into the registers and the operation is done within the ALU.

10 Performance

Performance can be improved by allowing the length of the control word to be changed. If it changes, then the time it takes for the program to run would be shortened. The same goes for the ALU and Register file. If the number within the files are small enough to be represented by a smaller number of bits, then it should be. Adding a pipelined datapath would also increase the overall speed of the processor due to not waiting for each instruction to make it all of the way thorough the CPU. We could also add some peripherals for more versatility in real world applications.

11 Errata

11.1 Listing of features that do not work as expected.

1. Control unit
2. Datapath

11.2 Description with as much detail as is known about why these problems exist.

The control unit and Datapath were a part of the same lab. The group was having trouble determining the values for the different variables such as PS and DA. We could not figure out all the values for each one. This caused the code to not function properly. In the end, we did not have enough time to work out all the different values, so that the control unit and Datapath could properly work.