ECE 543

Simple Processor

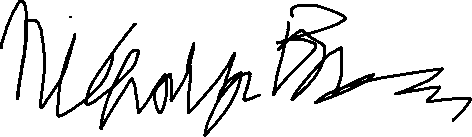
Final Project

Nicholas Barnes

01067990

Nicholas Barnes

Due: 12/10/2020



Honor Code: X\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Contents

[Table of Contents 2](#_Toc57813626)

[1. Introduction 4](#_Toc57813627)

[2. Arithmetic Logic Unit 6](#_Toc57813628)

[2.1 Description 6](#_Toc57813629)

[2.2 Half-Adder 6](#_Toc57813630)

[2.3 Full Adder 8](#_Toc57813631)

[2.4 Sixteen Bit Adder 10](#_Toc57813632)

[2.5 Sixteen Bit Subtractor 12](#_Toc57813633)

[2.6 16-Bit Multiplier 14](#_Toc57813634)

[2.7 16-Bit Mux 15](#_Toc57813635)

[2.8 16 bit Arithmetic Logic Unit 17](#_Toc57813636)

[2.9 Optimization/Improvements 20](#_Toc57813637)

[3. 8x16 Register File 21](#_Toc57813638)

[4. 256x16 Bit RAM 23](#_Toc57813639)

[5. Instruction Memory/Instruction Fetch 25](#_Toc57813640)

[6. Instruction Decoder/Executer // Processor 27](#_Toc57813641)

# Introduction

The objective for the Final Project for ECE 543 is to develop a simple processor, supporting 7 different instructions, two instruction formats, support 256 half words of RAM (256 addresses by 16 bits) and have 8 registers. Below is a high-level overview of the processor.

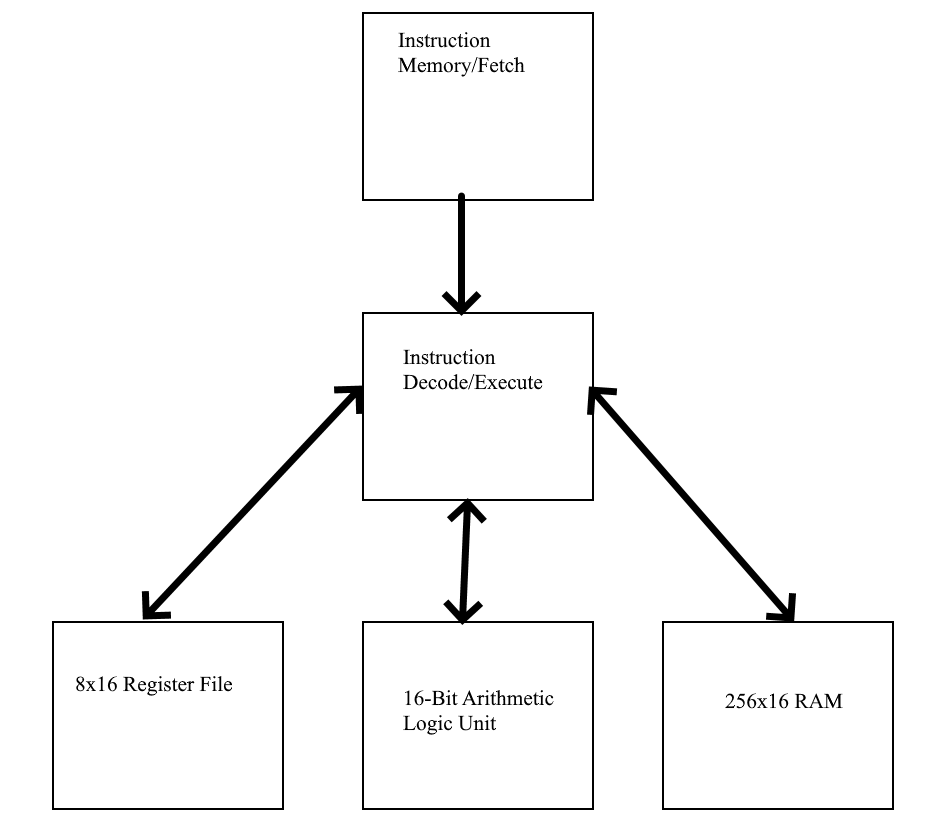


Figure X. High Level Overview of the Simple Processor

The 7 instructions the processor should support are Signed Addition, Signed Multiplication, Passthrough A, Passthrough B, Signed Subtraction, Load Immediate, Store Halfword, and Load Halfword. The opcode for the different instructions are given below.

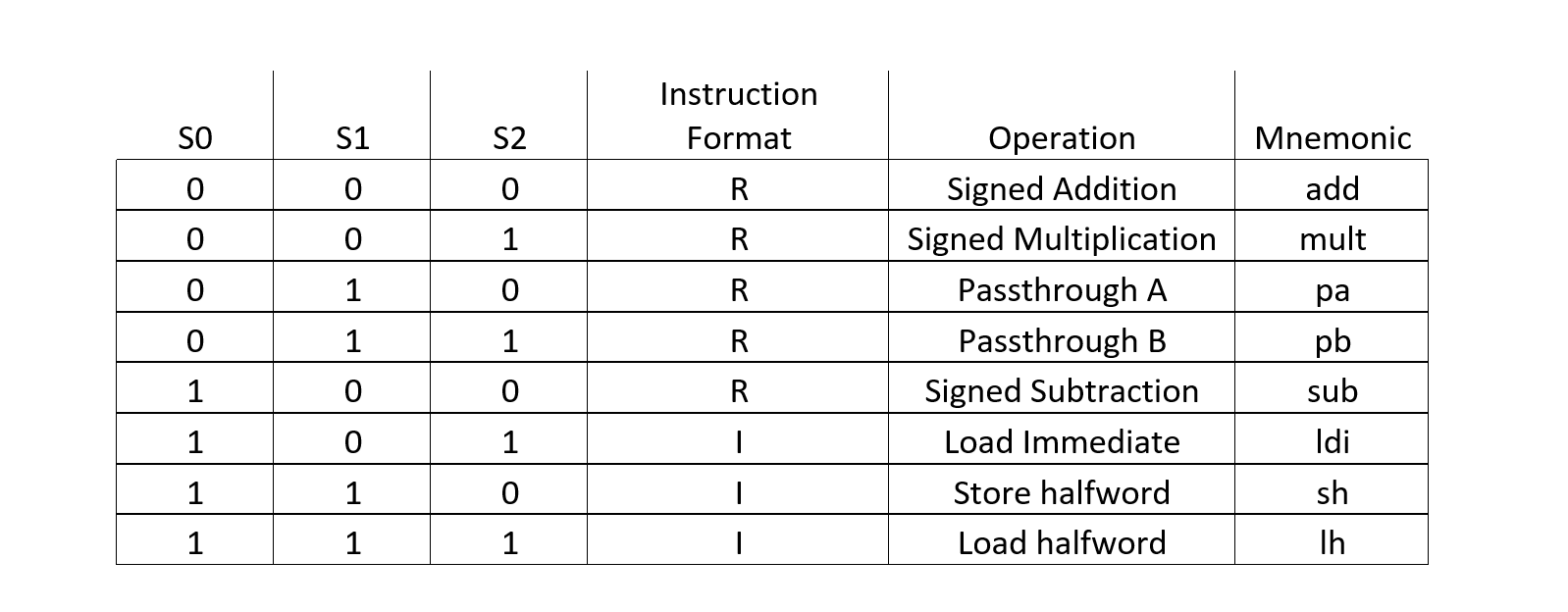


Figure X. Instructions Supported by the Processor

Finally, the processor should execute the following program:

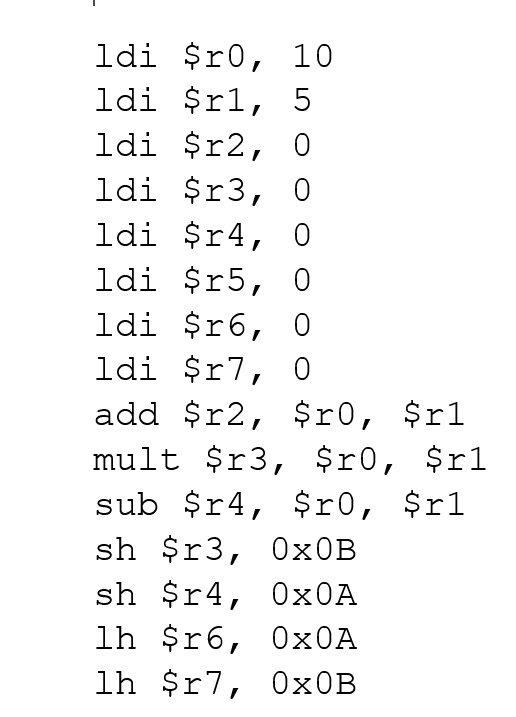


Figure X. Instruction Set to be Processed.

# Arithmetic Logic Unit

# Description

Utilizing the results of Project 1, the ALU needs to accept two signed 16-bit inputs and output a signed 16-bit solution. Three signals are also accepted to determine what operation the ALU will perform: signed addition, signed multiplication, passthrough A, passthrough B, or signed subtraction.

Each operation is to be implemented structurally, starting as small as the half-adder all the way to the 16-bit ALU. The student must test these structures and operations to demonstrate that the ALU is implemented correctly

# Half-Adder

* + 1. Description

The half-adder is a building block of a full adder. It accepts two 1-bit inputs and outputs a sum and a carry.



Figure 1a. Implementation of Half Adder using an XOR gate and an AND gate.

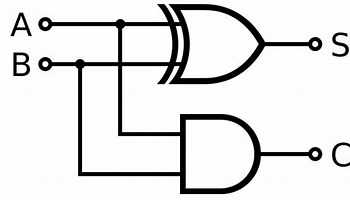


Figure 1b. Half Adder Structural Implementation

* + 1. Testing

Testing the half-adder involves making sure that the correct outputs for XOR and AND are met for the sum and carry. A truth table is given below for the possible outputs.

|  |  |  |  |
| --- | --- | --- | --- |
| A | B | O\_sum | O\_Carry |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

Figure 2. Truth Table for Half Adder

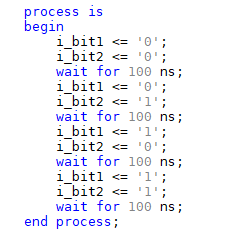


Figure 4. TestBench Signal Feed for Half Adder.

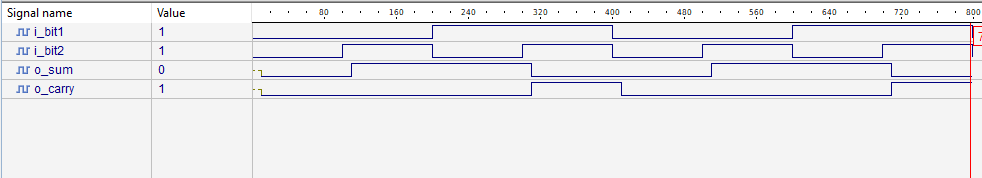


Figure 5. Waveform Diagram for Half Adder

It is important to consider a 10ns delay for each of the AND and XOR gates, so the sum and carry will be delayed appropriately.

The waveform diagram above follows the correct implementation for a half adder.

# Full Adder

* + 1. Description

The Full Adder is composed of two Half Adders. It accepts two 1-bit inputs to be added, and a 1-bit carry in input.

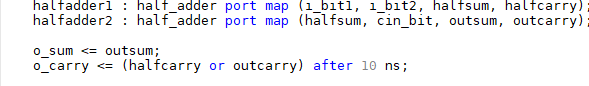


Figure 6a. Implementation of Full Adder using two half adders and an or gate

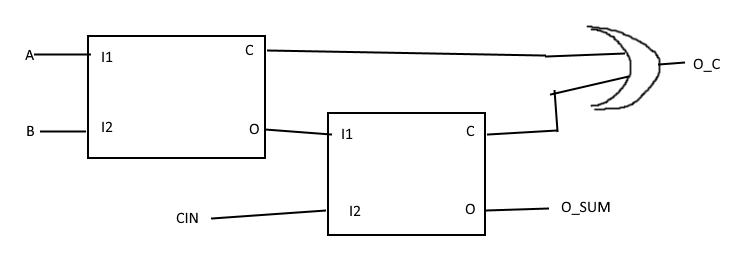


Figure 6b. Full Adder Structural Implementation

* + 1. Testing

Testing the Full Adder involves making sure that the correct outputs for the carry and sum follow the truth table. A truth table is given below for the possible outputs.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| I\_bit1 | I\_bit2 | Cin | O\_sum | O\_carry |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

Figure 7. Truth Table for Full Adder.

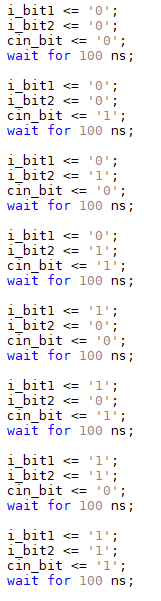


Figure 8. TestBench Signal Feed for Full Adder

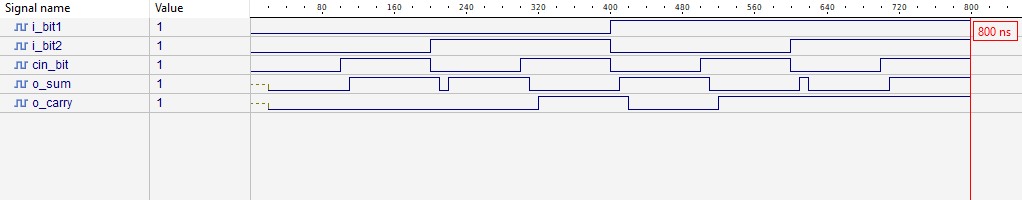


Figure 9. Waveform Diagram for Full Adder

There is a 10ns delay associated with the half adders, as well as an additional 10ns delay for the OR gate to calculate the carry. The waveform diagram above follows the correct implementation for a half adder.

# Sixteen Bit Adder

* + 1. Description

The Sixteen-Bit Adder is able to perform signed addition of two 16 bit numbers, and outputs a 16 bit sum and a carry. For simplicity, I implemented a ripple carry adder of with 16 full adders.

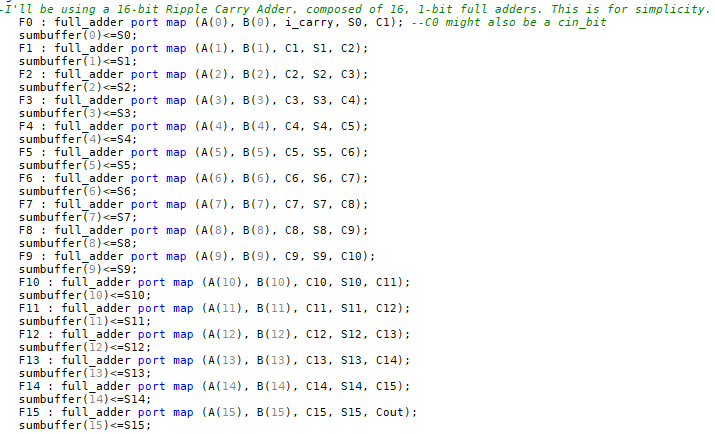


Figure 10a. Implementation of a Sixteen Bit Adder

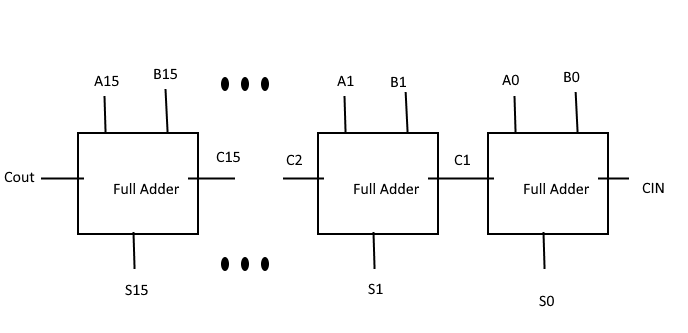


Figure 10b Sixteen-Bit Adder Structural Implementation

* + 1. Testing

Testing the Sixteen Bit Adder should involve cases for different types of signed addition. The cases for each will be outlined below, along with their corresponding waveform.

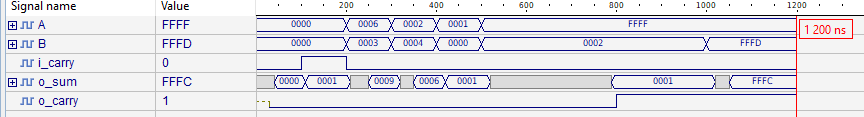


Figure 11. Waveform Diagram for a Sixteen Bit Adder.

* + - 1. A= B = A+B = 0

This case is run from 0-100ns and tests empty inputs. It does provide an empty output.

* + - 1. A= B = A+B = 0, Carry\_In = 1

This case is run from 100ns to 200ns and tests whether the carry in bit is added to the sum. It does.

* + - 1. A>B

This case is run from 200ns to 300 ns and tests whether the adder can handle addition where A>B. For the sum 6+3, the answer comes out to 9.

* + - 1. A<B

This case is run from 300ns to 400 ns and tests whether the adder can handle addition where A<B. For the sum 2+4, the answer comes out to 6.

* + - 1. N +0 = N

This case is run from 400 to 500 ns and tests whether the adder can handle addition where one input is 0. For the sum 1+0, the answer comes out to 0.

* + - 1. -A + B

This case is run from 500ns to 1000ns and tests whether the adder can handle signed addition where 1 input is negative and the other is positive. For the sum (-1)+2, the answer comes out to 1 after about 300ns.

* + - 1. -A + -B

This is the final case and tests whether the adder can handle signed addition where both inputs are negative. For the sum (-1) + (-3), the answer comes out to -4.

# Sixteen Bit Subtractor

* + 1. Description

The Sixteen-Bit Subtractor is able to perform signed subtraction of two 16 bit numbers, and outputs a 16 bit difference. It does this by inverting the second input and accepting a carry, mimicking the two’s complement conversion for negative numbers. Thus, the subtractor really is an adder that computes the sum of a number and an inverted number.

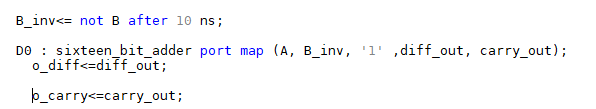


Figure 12a Implementation of a Sixteen Bit Subtractor

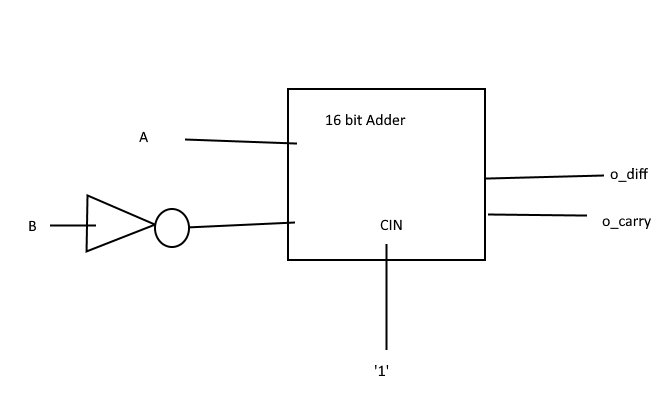


Figure 12b Sixteen Bit subtractor Structural Implementation.

* + 1. Testing

Testing the Sixteen Bit Subtractor should involve cases for different types of signed subtraction. The cases for each will be outlined below, along with their corresponding waveform.

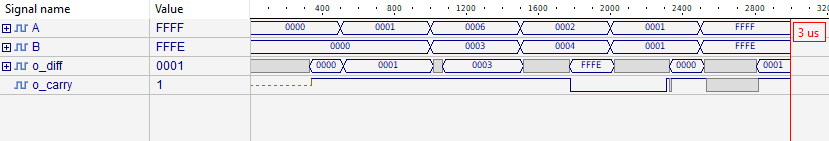


Figure 13 Waveform Diagram for Subtractor

* + - 1. A= B = A+B = 0

This case is run from 0-500ns and tests empty inputs. It does provide an empty output.

* + - 1. N +0 = N

This case is run from 500 to 1000 ns and tests whether the subtractor can handle subtraction where one input is 0. For the difference 1-0, the answer comes out to 0.

* + - 1. A>B

This case is run from 1000 to 1500 ns and tests whether the subtractor can handle subtraction where A>B. For the difference 6-3, the answer comes out to 3.

* + - 1. A<B

This case is run from 1500 ns to 2000 ns and tests whether the subtractor can handle subtraction where A<B. For the difference 2-4, the answer comes out to -2.

* + - 1. A=B /=0

This case is run from 2000 to 2500 ns and tests whether the subtractor can handle subtraction where both inputs are nonzero, equal inputs. For the difference 1-1, the answer comes out to 0.

* + - 1. -A>-B, (-A) - (-B)

This case is run from 2500ns to 3000 ns and tests whether the subtractor can handle signed subtraction where both inputs are negative, but the minuend is less than the subtrahend. For the difference (-1) – (-2), the answer comes out to 1.

# 16-Bit Multiplier

* + 1. Description

The Sixteen-Bit Multiplier is able to perform signed multiplication of two 16 bit numbers, and outputs a 16 bit product. It does this behaviorally, by multiplying inputs A\*B together.

* + 1. Testing

Testing the Sixteen Bit Multiplier should involve cases for different types of signed subtraction. The cases for each will be outlined below, along with their corresponding waveform.

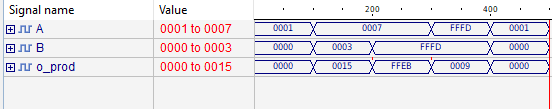


Figure 14 Waveform Diagram for Multiplier

* + - 1. A\*0 = 0

This case is run from 0-100ns and tests whether a number multiplied by zero is zero. The product of 1 and 0 is zero.

* + - 1. A and B are positive

This case is run from 100 to 200 ns and tests whether the multiplier can multiply two positive numbers. The product of 7 and 3 is 0015, or 21 base 10.

* + - 1. A\*-B

This case is run from 200ns to 300ns and tests whether the multiplier can multiply a positive and negative number. The product of 7 and -3 is FFEB, or -21 base 10.

* + - 1. -A\*-B

This case is run from 300ns to 400ns and tests whether the product of two negative numbers is positive. The product of -3 and -3 is 9.

# 16-Bit Mux

* + 1. Description

The 16 bit mux for this ALU takes in 3 signal bits which chooses from 1 of 8 inputs. It does this structurally, using a mix of AND and NOT gates. We’re assuming this multiplexer to be ideal, so there are no delays on any of the AND or NOT gates.

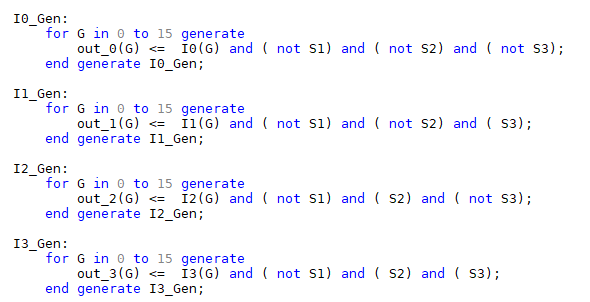


Figure 15a Structural Implementation of Multiplexer.

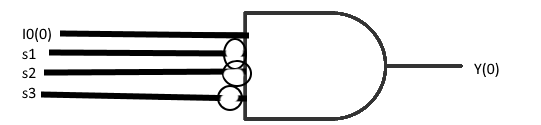
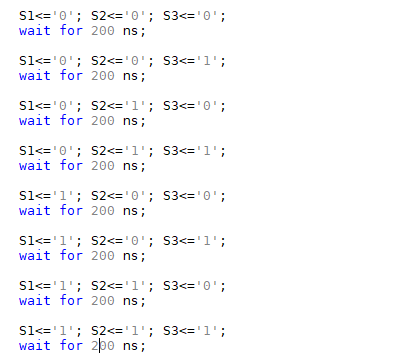


Figure 15b Structural Implementation of Multiplexer, given bit 0 of I0 and outputting bit 0 of output Y

* + 1. Testing

Testing the multiplexer should consider all possible combinations of inputs, and it should output a value according to the truth table below.

|  |  |  |  |
| --- | --- | --- | --- |
| S1 | S2 | S3 | Y |
| 0 | 0 | 0 | I0 |
| 0 | 0 | 1 | I1 |
| 0 | 1 | 0 | I2 |
| 0 | 1 | 1 | I3 |
| 1 | 0 | 0 | I4 |
| 1 | 0 | 1 | I5 |
| 1 | 1 | 0 | I6 |
| 1 | 1 | 1 | I7 |

Figure 16 Truth Table for a 8 input multiplexer.

Figure 17 Testbench input for an 8 input multiplexer

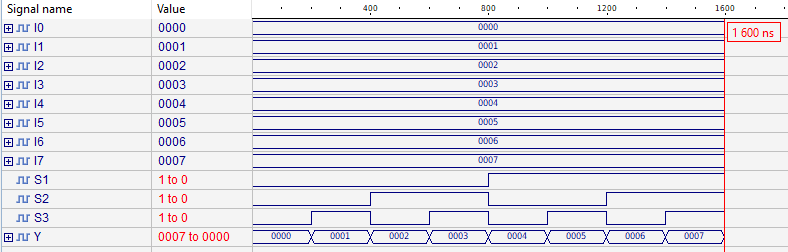


Figure 18 Waveform Diagram for 8 input multiplexer

I0-7 output their respective numbers 0-7, and each possible input is done sequentially. The waveform does follow the truth table.

# 16 bit Arithmetic Logic Unit

* + 1. Description

Putting together the 16 bit Adder, Subtractor, Multiplier, and Multiplexer forms a 16 bit ALU. The ALU takes two 16 bit inputs and 3 signal inputs to determine the operation, and outputs a 16 bit output and a status vector.

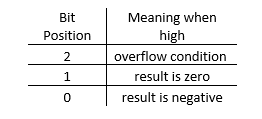
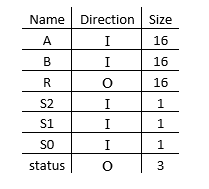


Figure 19. I/O Table for ALU Figure 20. Status Output Description

* + 1. Testing (R1)

Testing the ALU involves making sure that the correct signals output the correct operation. The opcodes for each signal are shown in the Truth Table below.

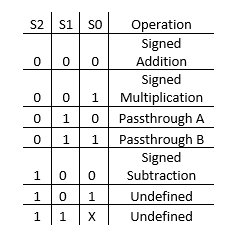


Figure 21. Opcode.

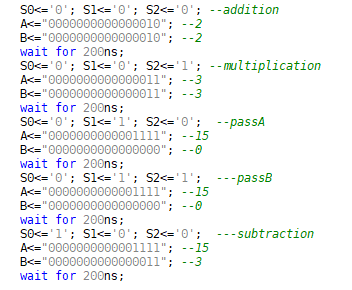


Figure 22. First round of testing, ensures all operations work

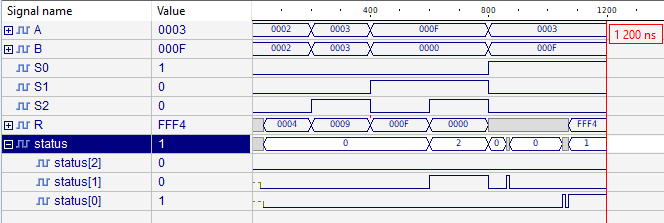


Figure 23. Waveform Diagram for ALU operations

* + - 1. Addition

This case is run from 0-100ns and tests addition between 2 and 2. The sum is 4.

* + - 1. Multiplication

This case is run from 100 to 200 ns and tests multiplication between 3 and 3. The product is 9.

* + - 1. PassA and Pass B

This case is run from 200ns to 300ns and 300 to 400ns respectively and tests whether the ALU outputs either input A or B. A=15 and B = 0; For PassB, the status bit is in position 1.

* + - 1. Subtraction

This case is run from 400ns to 500ns and tests subtraction between 3 and 12. The difference is -12, and the status bit is in position 0.

* + 1. Testing (R2)

This section contains the data to answer the questions: the sum of 1000 and 2000 and the product of 128 and 128.

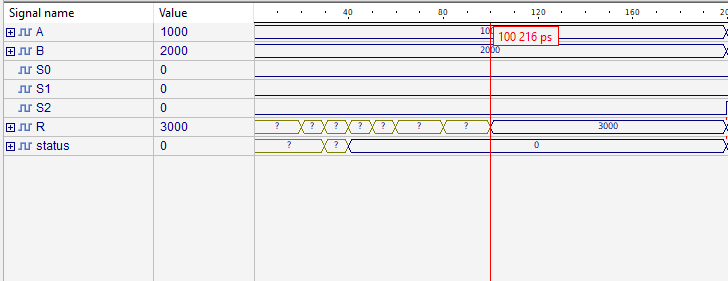


Figure 24. Waveform Diagram for 1000+2000

It takes 100 ns to calculate the sum of 1000 and 2000.

At 1000ns, the operation becomes the multiplication of 128 and 128.

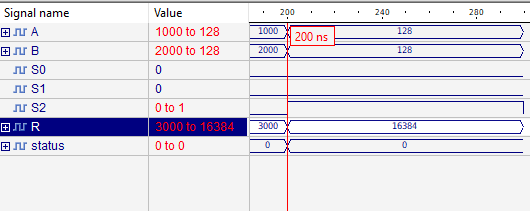
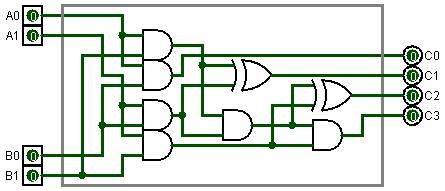


Figure 25. Waveform Diagram for 128 \* 128

Because the multiplier is behavioral, the multiplication is instant. However, it realistically should be slower, since structurally a 2 by 2 multiplier would take about 30 ns of calculation.



Source: https://en.wikipedia.org/wiki/Binary\_multiplier

Thus, this ALU can perform 1s/100ns = 10,000,000 additions per second.

The multiplier however can perform infinitely many multiplications per second as units of time get smaller. Therefore, it would be more realistic to use the time it takes for the 2 by 2 multiplier mentioned earlier, even if it would be inaccurate: 1s/30ns = 33,333,333 multiplications per second.

Reasonably, this ALU would perform 1s/T = T\*10^9 multiplications per second, where T is the total delay for the multiplier’s calculation in nanoseconds.

# Optimization/Improvements

The 16-bit adder implemented in this design is a ripple carry adder. Processing time can be reduced if a Carry Lookahead adder was implemented instead. A carry lookahead provides the carry bit for the next bit without waiting for other additions in the series.

Since the Subtractor also uses the 16-bit adder, the carry-lookahead would save time there, too.

Because the multiplier would take many AND and NOR gates to implement, a dedicated multiplier chip could be used in the hardware to simplify the design.

I could not implement the overflow properly, but I attempted to implement it by taking the carry bit from the adder/subtractor and setting it to a “isoverflow” std\_logic bit. This, however, did not work since the carry bit could be different for both the adder and the subtractor.

# 8x16 Register File

* 1. Description

The 8x16 Register File should be able to store and retrieve eight 16-bit values. It receives a value for rs, rt, rd, and an additional rd\_alu value which is used for loading from the ALU or data memory. It also takes in the values to be loaded into a registers, whether that be from an ldi instruction, an ALU result, or a halfword of data.

Alu\_write loads the data from the ALU into the Rd of the previous instruction. Mem2reg loads the data from memory into the Rd of the previous instruction. The Ldi loads an immediate value into Rd of the current instruction.

Register values 0-7 are held in an array of 8 16 bit std\_logic\_vectors.

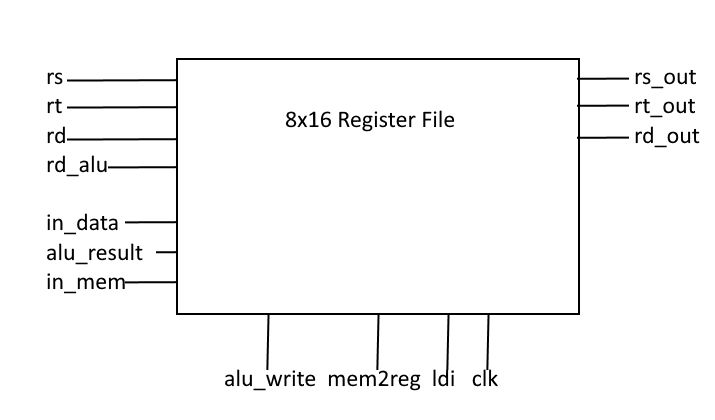
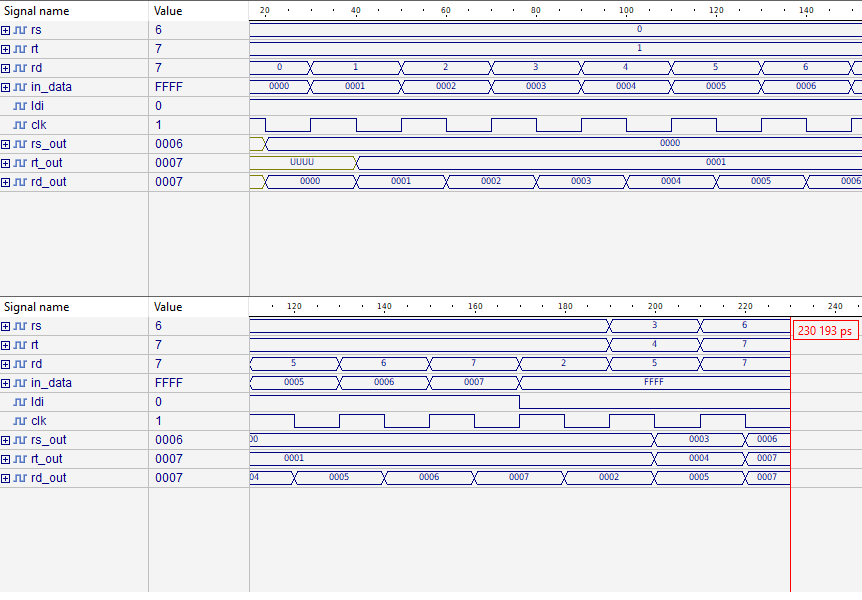


Figure X. 8x16 Register File Design

* 1. Testing



From 0 to 180ns, values 0-7 are loaded into registers 0-7 respectively. Then the corresponding values are displayed.

From 180ns to 200ns, the values of registers 0-2 are displayed in rs\_out, rt\_out, and rd\_out respectively.

From 200ns to 220ns, the values of registers 3-5 are displayed in rs\_out, rt\_out, and rd\_out respectively.

From 220ns on, the values of registers 6-7 are displayed in rs\_out and rt\_out respectively.

# 256x16 Bit RAM

* 1. Description

The 256x16 bit RAM should be able to store and retrieve sixteen 16-bit values. The RAM receives a 4-bit address and data to be stored. It also receives 2 flags, memwrite and mem2reg, which writes and reads data from memory respectively. It outputs the data to be written to the register on the next clock cycle.

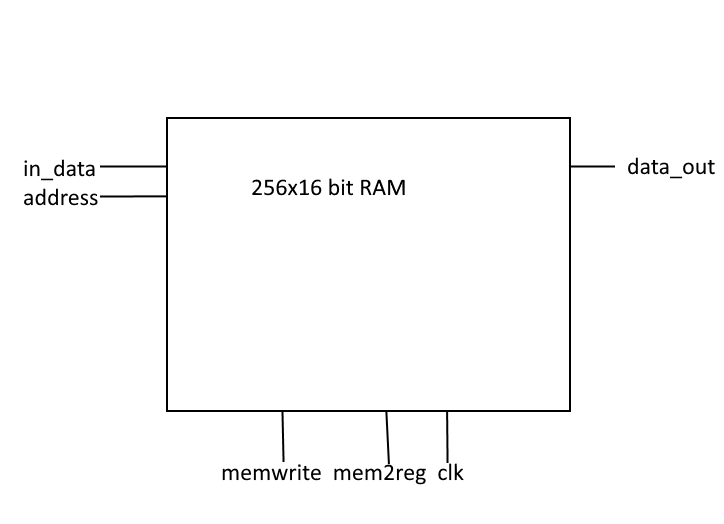
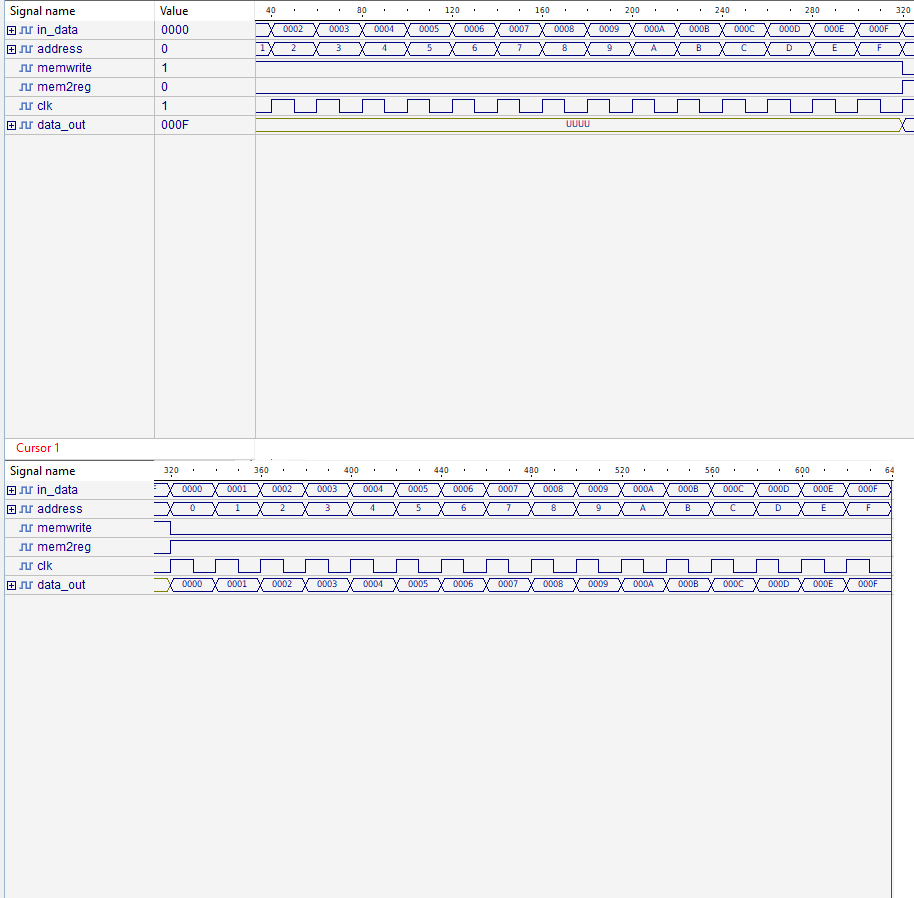


Figure X. 256 x 16 bit RAM Design.

* 1. Testing



From 0-320ns, values are loaded into addresses 0000-000F. From 320ns on, values are outputted from each address in memory.

# Instruction Memory/Instruction Fetch

* 1. Description

For simplicity, the instruction memory and instruction fetch are combined. Every clock cycle, the imem should output an instruction. The instruction on the next clock cycle should be the next instruction. The current instruction is tracked using an internal counter.

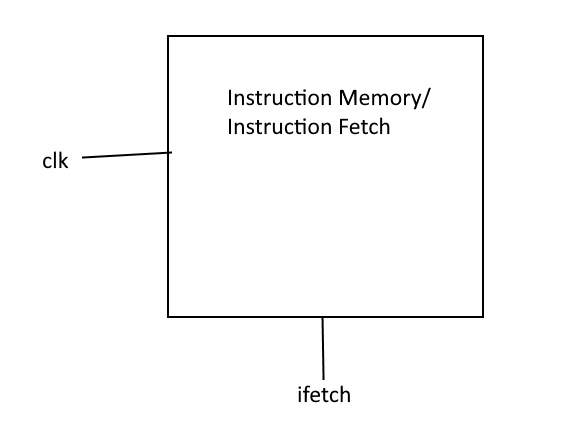


Figure X. Instruction Memory/Instruction Fetch Design.

The list of instructions and their compiled hexadecimal code are as follows:

ldi $r0, 10 --500A

ldi $r1, 5 --5105

ldi $r2, 0 --5200

ldi $r3, 0 --5300

ldi $r4, 0 --5400

ldi $r5, 0 --5500

ldi $r6, 0 --5600

ldi $r7, 0 --5700

add $r2, $r0, $r1 --0201

mult $r3, $r0, $r1 --1301

sub $r4, $r0, $r1 --4401

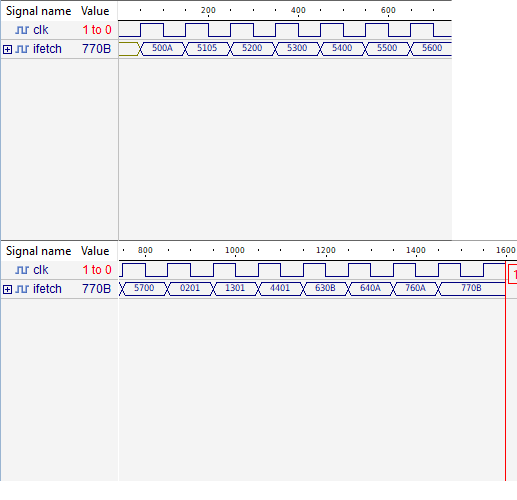
sh $r3, 0x0B --630B

sh $r4, 0x0A --640A

lh $r6, 0x0A --760A

lh $r7, 0x0B --770B

* 1. Testing



Each clock cycle, an instruction will be outputted. The code outputted each clock cycle accurately follows the instruction list given in 5.1.

# Instruction Decoder/Executer // Processor

* 1. Description

The Instruction Decoder/Executer is the main body of the processor. It utilizes all the preceding components to execute instructions given in the instruction memory.

The decoder itself separates the instruction values into their respective slots.

|  |  |
| --- | --- |
| **Signal** | **Instruction Section** |
| OP\_S1 | Instruction(14) |
| OP\_S2 | Instruction(13) |
| OP\_S3 | Instruction(12) |
| Rd | Instruction(10 downto 8) |
| Rs | Instruction(6 downto 4) |
| Rt | Instruction(2 downto 0) |
| Imm | Instruction(3 downto 0) |

Figure X. Signal List for the Decoder

A signal ldi\_flag is used to execute the ldi instruction.



Figure X. Ldi\_flag condition.

A signal alu\_write is used to execute ALU instructions.

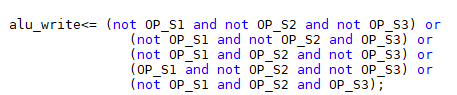


Figure X. alu\_write condition.

A signal mem2reg is used to execute the load halfword (lh) instruction.



Figure X. mem2reg condition

A signal memwrite is used to execute the store halfword (sh) instruction.



Figure X. memwrite condition.

The value of rd is stored into a rd\_alu signal so the value retrieved from memory or the ALU can be stored in the register file on the next clock cycle. This does not cause a hazard if an ldi instruction is called immediately after, since rd\_alu is not updated until after the register file is written to/loaded from for the current instruction.

* 1. Decoder/Executer/Processor Design

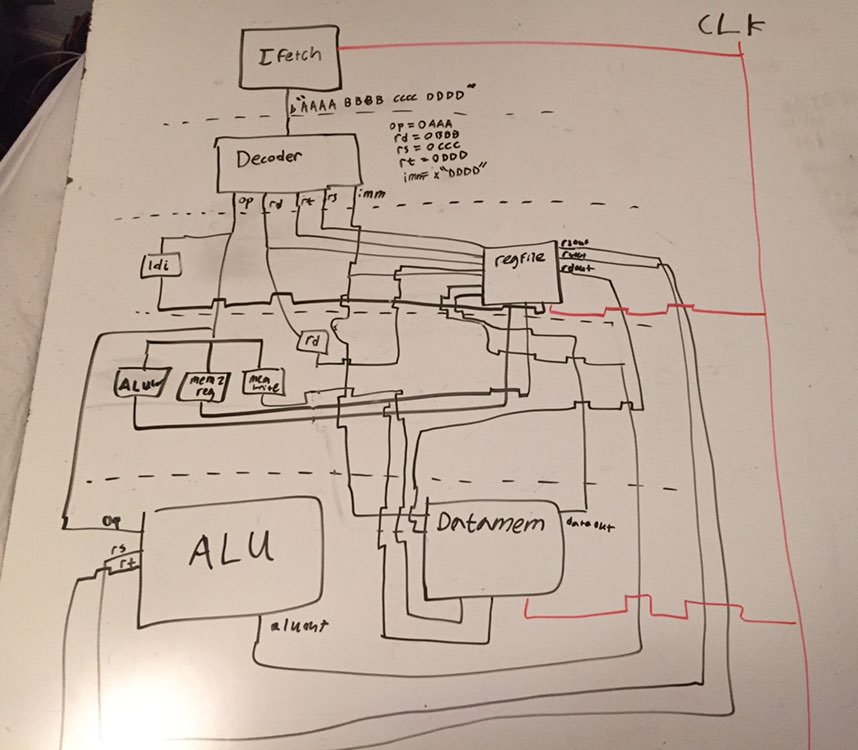
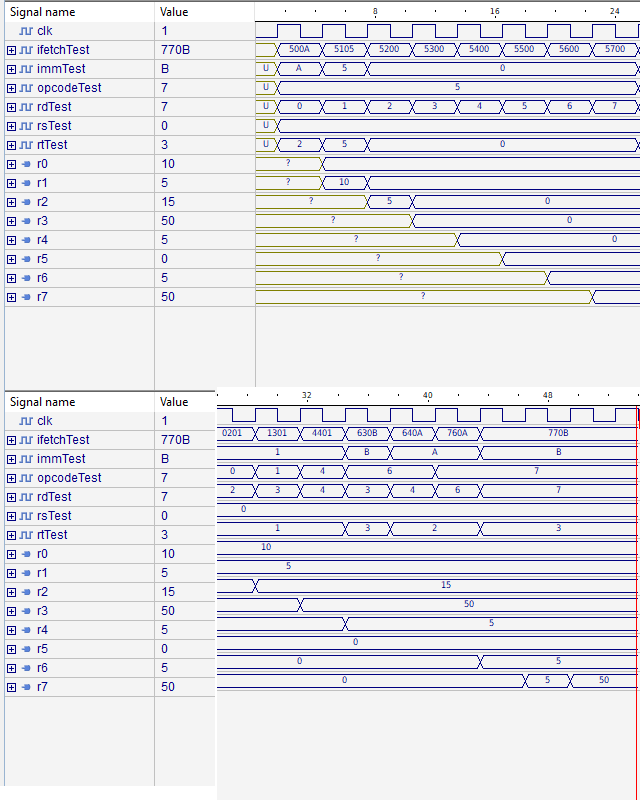


Figure X. Decoder/Executer/Processor Design

* 1. Processor Operation



* 1. Processor Results

The program may end at 47us, but the execution of the lh instruction 770B does not execute until a full clock cycle later. The final values of r6 and r7 are 5 and 50, respectively.