CSC-270 Final Project Write-up

Name: Sam Rodberg

Processor Name: SR-13

Processor Description: 16-bit Single Cycle processor with support for pseudo instructions neg and mov. **Score Sheet**

Feature	Value	Passes Tests	Screenshots	Score
Basic: ALU	10	YES	YES	
Basic: R and I type	10	YES	YES	
Basic: LW and SW	10	YES	YES	
Basic: Assembler	20	YES	YES	
Basic: Writeup	20	YES	YES	
A la Carte				
16-bit	5	YES	YES	
Support for 2 Pseudo Instruction in Assemlber	5	YES	YES	
Total	80			(earned)

Introduction

The SR-13 is 16-bit microprocessor developed by Sam Rodberg is based on the architecture of the MIPS processor. The SR-13 includes several important features such as a 16-bit ALU with support for addition, subtraction, logical AND, and logical OR operations. It includes 8 register capable of storing 16-bit values. The SR-13 also includes support for load word and store functions. No processor would be complete without some bells and whistles which is why the SR-13 includes support for pseudo instructions neg, and mov. An easy to use command line assembler is also provided to allow for easy programming on the microprocessor.

Design Decisions Overview

The design decisions for this processor were mostly based around time constraints by the design team aka myself, as well which features logically worked together. The decisions were deciding on attempting to support BEQ operations and J-type instructions but this did not materialize unfortunately. I focused on implementing these because J-type instructions and BEQ are similar in nature and complement each other nicely. A 16-bit processor design was chosen because it allows for large computations to performed easily avoids the problems with logisim's 32-bit limits (Specifically RAM modules). Pseudo instructions were chosen to be supported by the assembler as they allow the programmer more flexibility and ease of use when programming the SR-13 microprocessor.

Machine Code Instruction Format

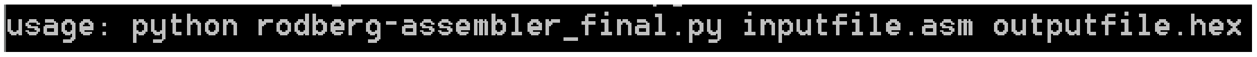
My instruction word for this "kitchen sink" type processor is 14 bits in length. For the 14 bits, the first 5 bits are the op code, the next 3 bits are the destination, the next 3 bits are src1 and the last 3 bits are sr3. This can be represented by the following diagram:

Op code (5)	Dest (3)	Src 1 (3)	Src (3)

Figure 1: Machine Code Instruction Format

Directions For Translating Assembly Code Into Hexadecimal Machine Code

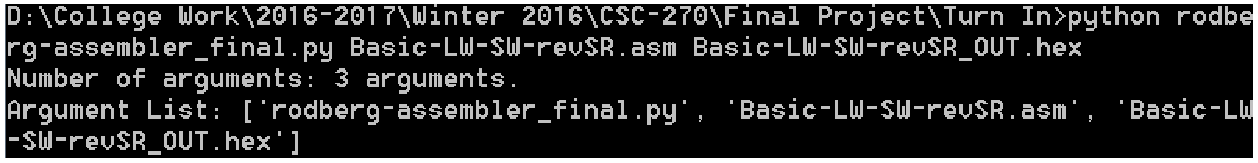
1. In the directory containing the unzipped contents of my project, open a command prompt on windows or terminal on Mac OS or Linux.
2. To translate the assembly code into machine code the following syntax needs to be followed:



```
usage: python rodberg-assembler_final.py inputfile.asm outputfile.hex
```

Figure 2: Assembler Syntax and Usage

3. An example usage case can be seen below in Figure 3:

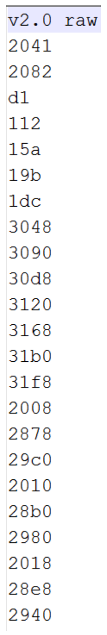


```
D:\College Work\2016-2017\Winter 2016\CSC-270\Final Project\Turn In>python rodbe
rg-assembler_final.py Basic-LW-SW-revSR.asm Basic-LW-SW-revSR_OUT.hex
Number of arguments: 3 arguments.
Argument List: ['rodberg-assembler_final.py', 'Basic-LW-SW-revSR.asm', 'Basic-LW
-SW-revSR_OUT.hex']
```

Figure 3: Example Assembler Usage

As seen in Figure 3, the first argument refers to the assembler itself that is being run, the second argument refers to the name of the input .asm file that will be assembled into hex. The third argument represents the name of the output .hex file that will be generated.

4. After the assembler has finished the resulting .hex file will appear in the same directory as the directory that contains python script for the assembler. The resulting hex file should appear similar to the following .hex file generated from the example in Figure 3:



```
v2.0 raw
2041
2082
d1
112
15a
19b
1dc
3048
3090
30d8
3120
3168
31b0
31f8
2008
2878
29c0
2010
28b0
2980
2018
28e8
2940
```

Figure 4: Resulting Hex File

These are all the steps that should be needed to successfully translate MIPS based assembly instructions into hex for the SR-13 processor. For an overview of supported instructions please the next section.

Programmers Guide And Assembly Code Instructions to Machine Code Mapping Guide

The following section describes the available instructions, their format and the corresponding mapping to machine code. Note, that in the mapping fields such as rd, rs, and rt are written as ‘000’ this just indicates a 3 bit-value should be present for that field.

ADD – Add

Description: Adds two registers and stores the result in a register
Operation: [rd] = [rs] + [rt]
Syntax: add rd, rs, rt
Mapping:

00000 (op)	000 (rd)	000 (rs)	000 (rt)

ADDI – Add Immediate

NOTE: Immediate is sign extended

Description: Adds a register and an immediate value and stores the result in a register
Operation: [rt] = [rs] + immediate
Syntax: addi rt, rs, immediate
Mapping:

10000 (op)	000 (rt)	000 (rs)	Immediate

AND – Logical AND Operator

Description: Logical ands two registers and stores the result in a register
Operation: [rd] = [rs] & [rt]
Syntax: and rd, rs, rt
Mapping:

00001 (op)	000 (rd)	000 (rs)	000 (rt)

OR – Logical OR Operator

Description: Logical ors two registers and stores the result in a register
Operation: [rd] = [rs] | [rt]
Syntax: or rd, rs, rt
Mapping:

00010 (op)	000 (rd)	000 (rs)	000 (rt)

SUB – Subtract

Description: Subtracts two registers and stores the result in a register
Operation: [rd] = [rs] – [rt]
Syntax: sub rd, rs, rt
Mapping:

00011 (op)	000 (rd)	000 (rs)	000 (rt)

LW – Load Word

Description: A word is loaded into a register from the specified memory address
Operation: [rt] = [address]
Syntax: lw rt, immediate(rs)
Mapping:

10100 (op)	000 (rt)	Immediate	000 (rs)

SW – Store Word

Description: The contents of a register is stored at the specified memory address
Operation: [Address] = [rt]
Syntax: sw rt, immediate(rs)
Mapping:

11000 (op)	000 (rt)	Immediate	000 (rs)

MOV – Move

Description: Copy the contents of a register into another register
Operation: [rt] = [rs]
Syntax: mov rt, rs
Mapping:

00001 (op)	000 (rt)	000 (rs)	000 (padding)

NEG – Negate

Description: Negate the contents of a register into another register
Operation: [rt] = -[rs]
Syntax: neg rt, rs
Mapping:

00011 (op)	000 (rt)	000 (rs)	000 (padding)

<u>Evidence of Working Features – Basic Processor</u>

In this section I will be providing evidence that all required features of the basic processor work.

Basic: ALU and Basic: R Type and I Type

The first piece of evidence will be the provided test code file Basic-R-Type-rev2.asm:

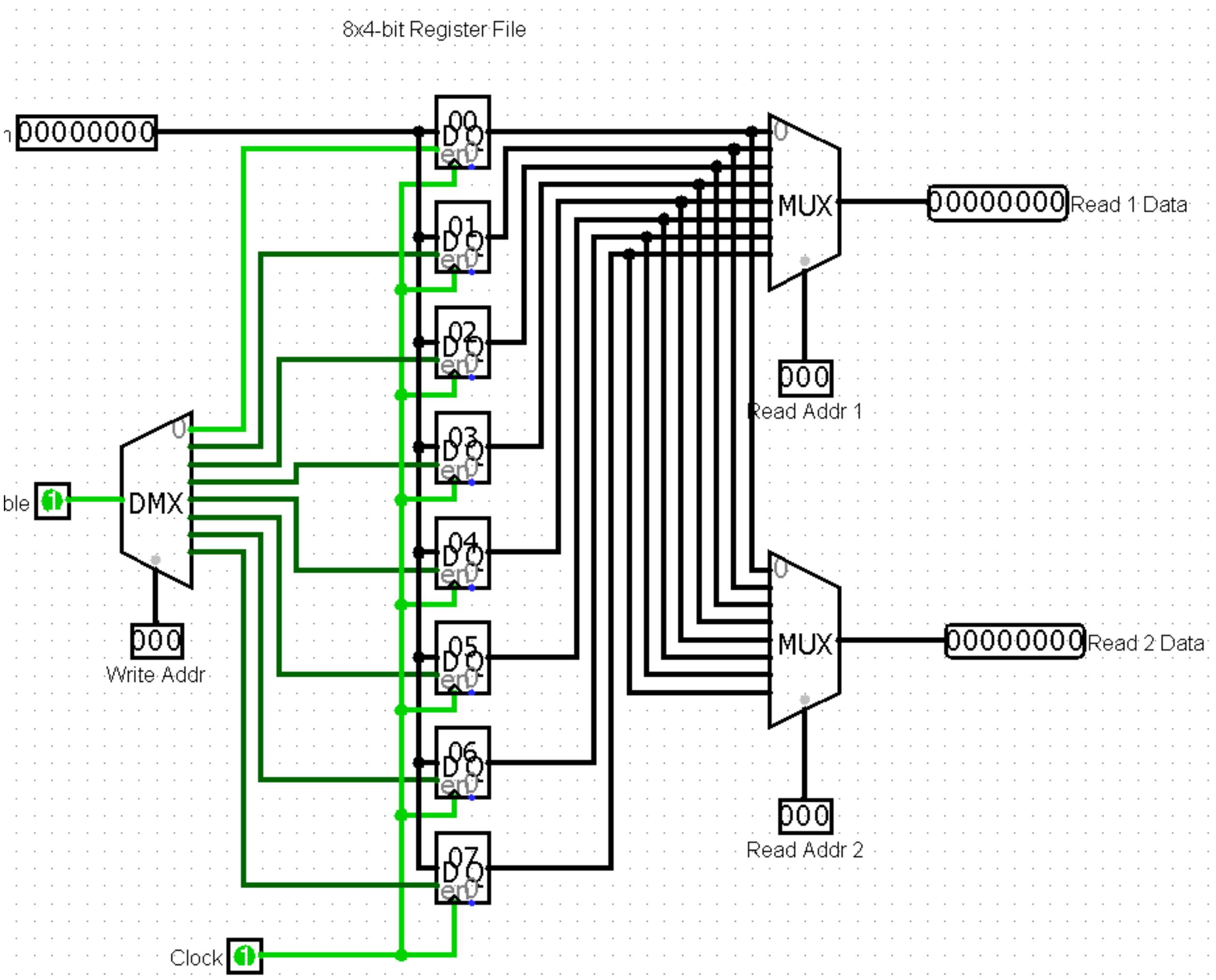


Figure 5: Basic ALU John Test Code

The goal of this code was to put the values 1 through 7 into register 1 through 7 which proved to be successful at demonstrating both add and addi operations.

The next piece of evidence that will provided comes from my own test code file and-or.asm:

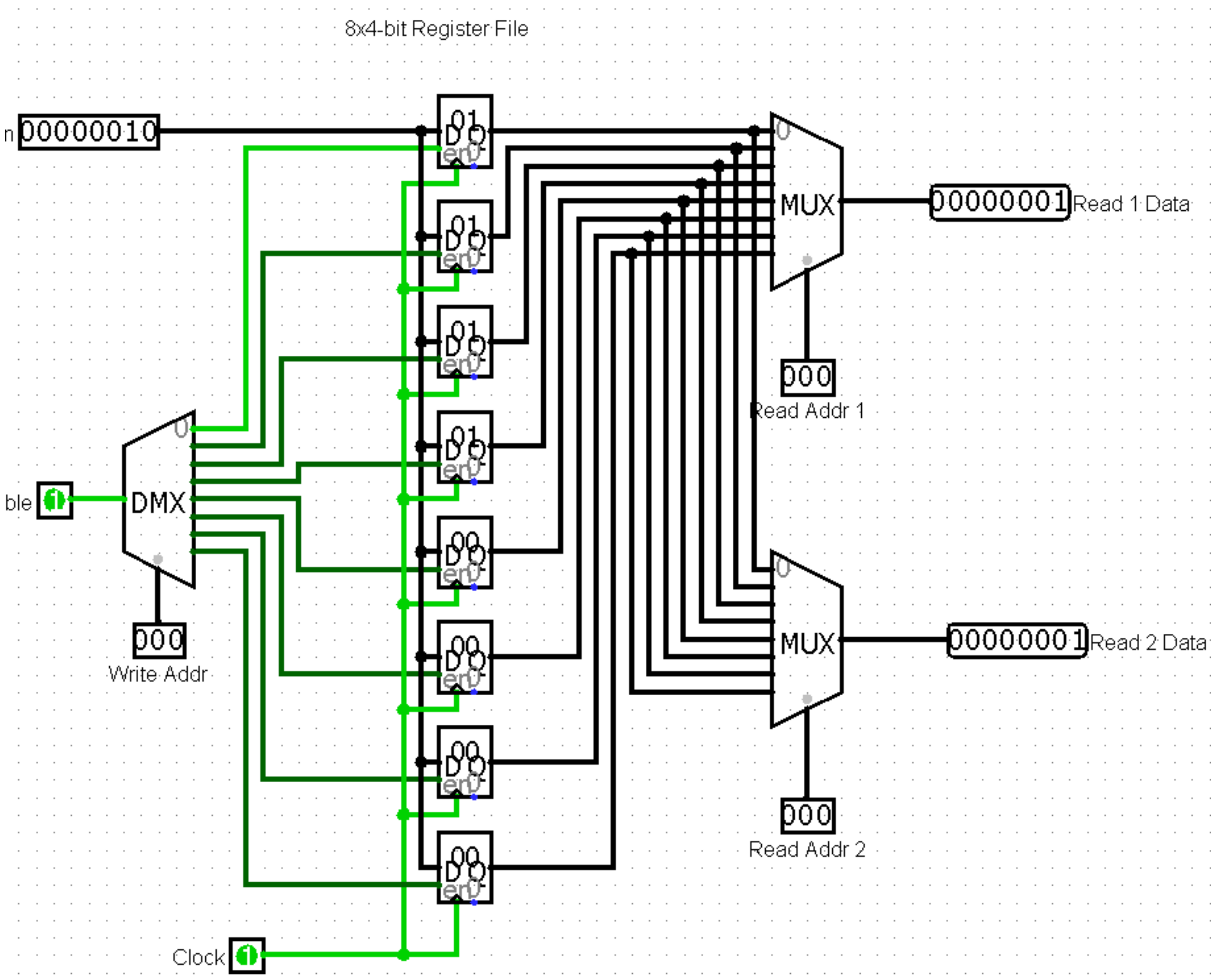


Figure 6: Basic ALU Sam Test Code

The goal of the code is it first does an addi of 1 into register 0, it does an addi of 1 into register 1 as well. Next it performs logical and on registers 1 and register 0 storing the result in register 2 which is as expected 1 as (1 and 1 is 1). Finally it performs logical or on registers 1 and register 0 and stores the result in register 3 which is as expected 1 (1 or 0 is 1). The next piece of test code demonstrates the sub and add operation of the ALU in the test file add-sub.asm as see in Figure 7 on the next page:

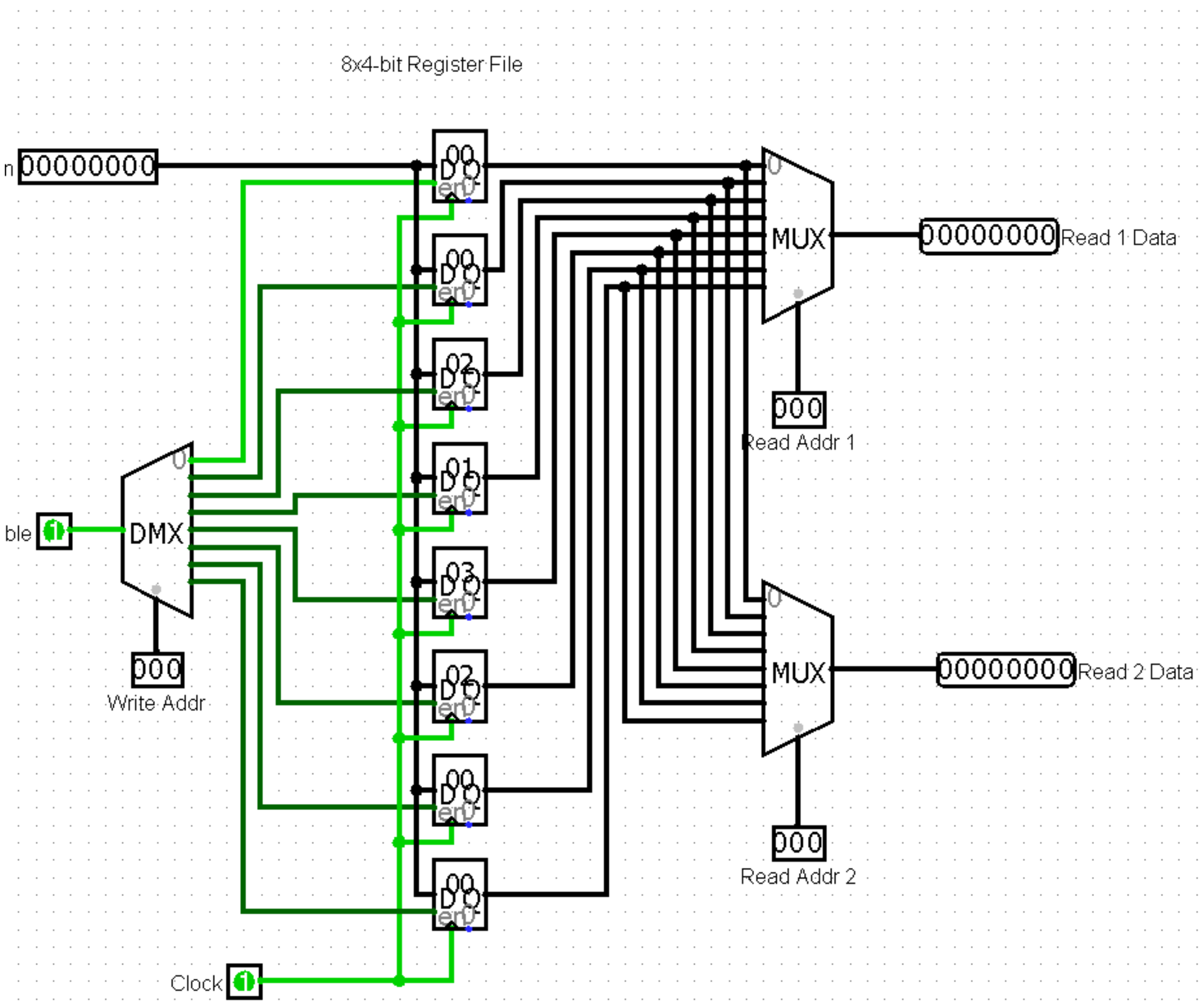


Figure 7: More Basic ALU Test Code

The first line of the test code does an addi of 1 into register 3 and an addi of 2 into register 2. The contents of registers 3 and 2 are added and stored into register 4 which produce the value 3, as 1 + 2 = 3. The last line of the test code uses the subtraction operation to subtract the contents of register 4 from the contents of register 3 which the result is stored in register 5 and produces the value of 2 since, 3 – 1 = 2. While I have already shown that R-type and I-type instructions work I will provide screenshots from the main FULL CPU wires to show they work as see in Figure 8 and Figure 9 below on the next page corresponding to test code rtype-itype.asm:

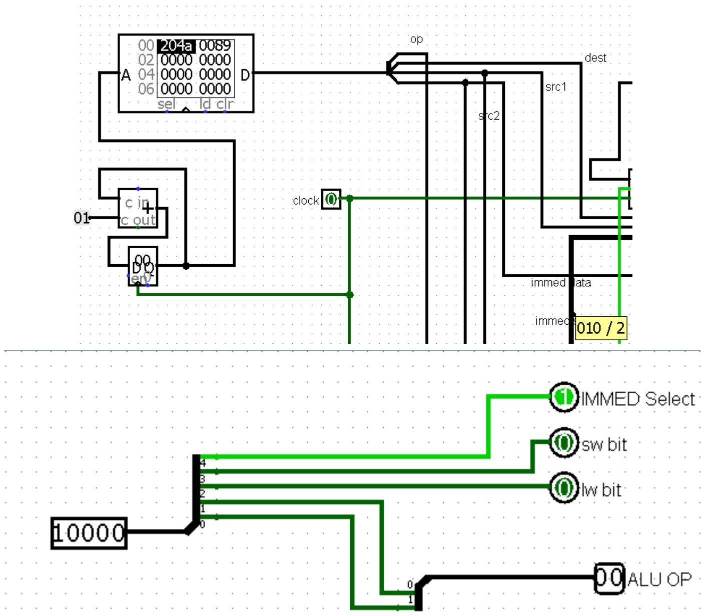


Figure 8: I-Type Instruction

We can tell I-type instructions are working because, the immediate select wire has 1 meaning an I-type instruction is successfully being performed (Also opcode is 10000 indicating addi). Likewise in Figure 9 below we can when performing a regular add instruction aka an R-type instruction the immediate select wire value is 0 meaning an R-type instruction is being performed (Also opcode is 00000 indicating add):

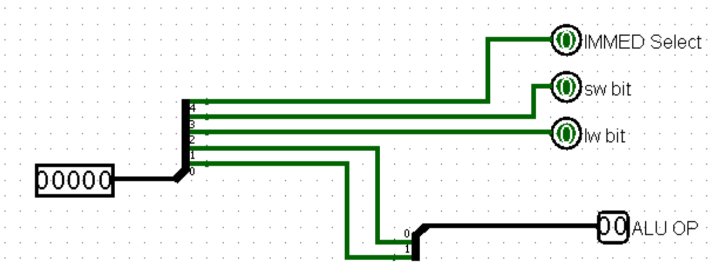


Figure 9: R-Type Instruction

Basic ALU: LW and SW

The first piece of evidence that will be provided to showcase that lw and sw work is the provided test code file that I have modified to handle my 3-bit addi limit:

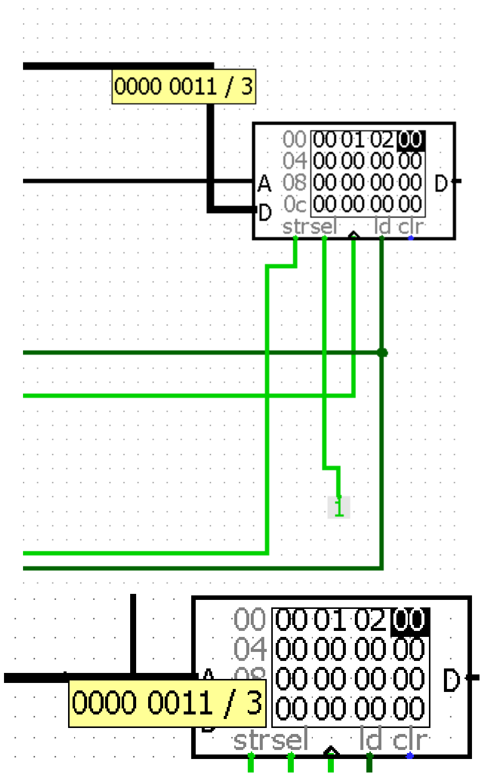


Figure 10: SW Demo

We can see that the data about to be stored into address 3 in memory is a 3, store word has been a successful operation as well as 1 and 2 have also been stored into memory at addresses 1 and 2 respectfully. In Figure 11 below we can see that load word is working properly:

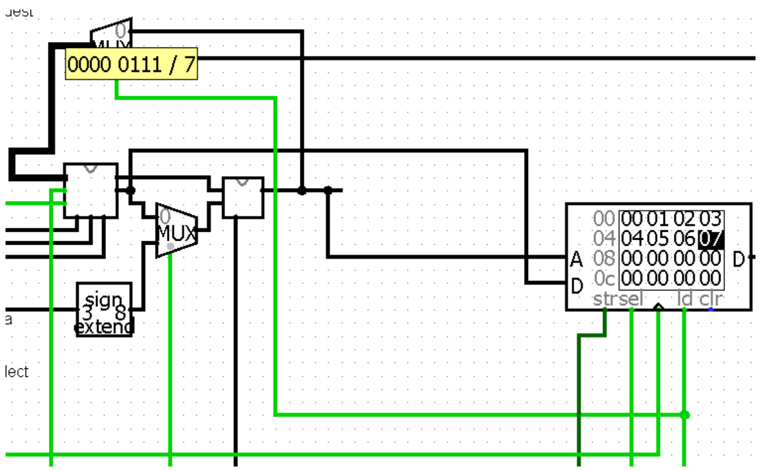


Figure 11: LW Demo

We know that load word is working because the load word wire going into the ld port of the RAM module is 1 and is active. Most importantly we see from the MUX the value coming out and being feed into the register (Data in) is 7, indicating it will be stored in the register file. When the code successfully completes the results will be that the values 7 through 1 will be in registers 1 through 7 as seen in Figure 12:

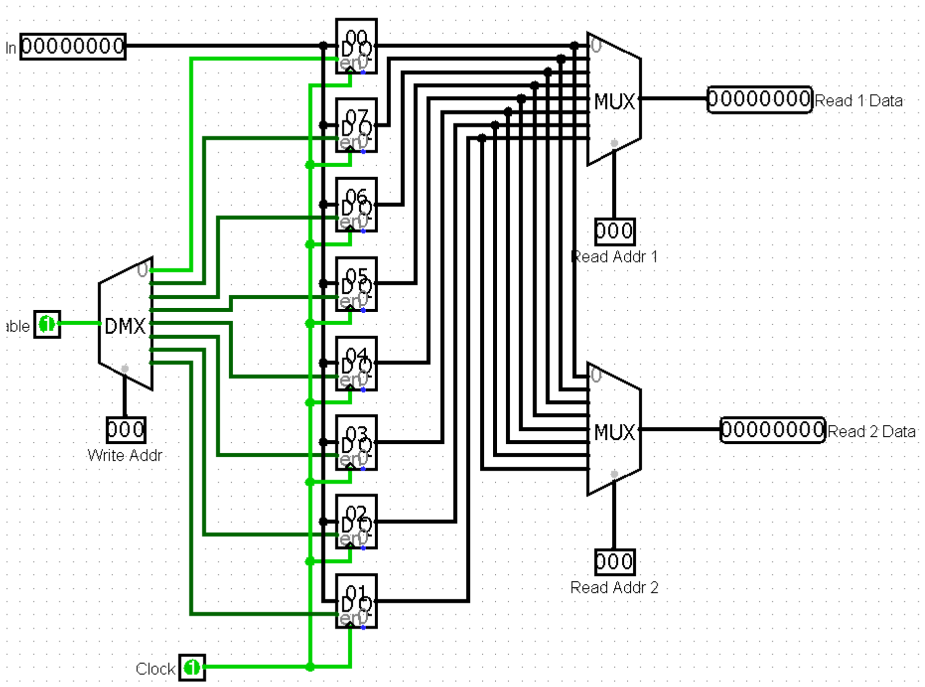
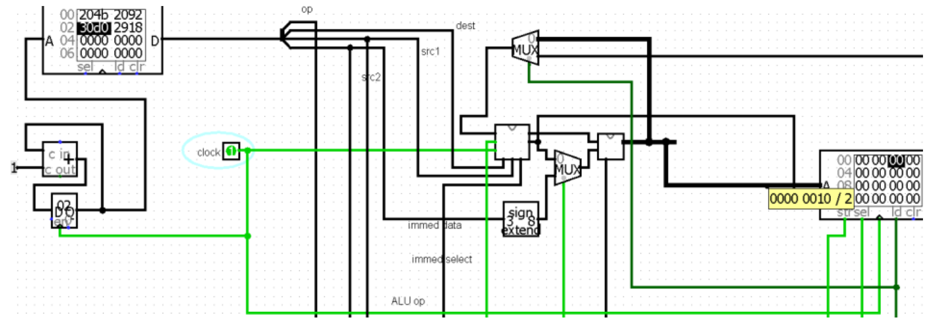
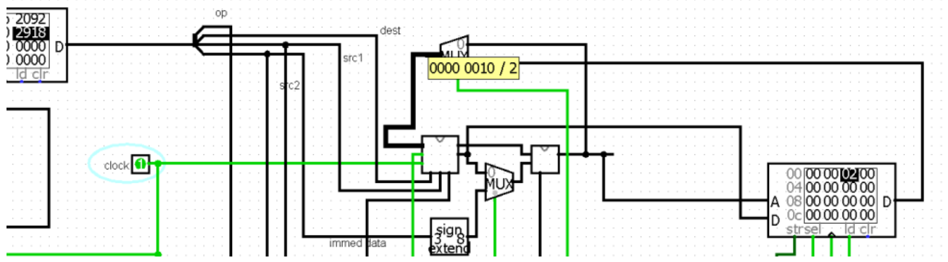


Figure 12: Final LW and SW Result

The final piece of evidence I will be providing is my own test code called lw-sw.asm which evidence of load word and store word working can be seen in Figure 13 below (Multiple Images):





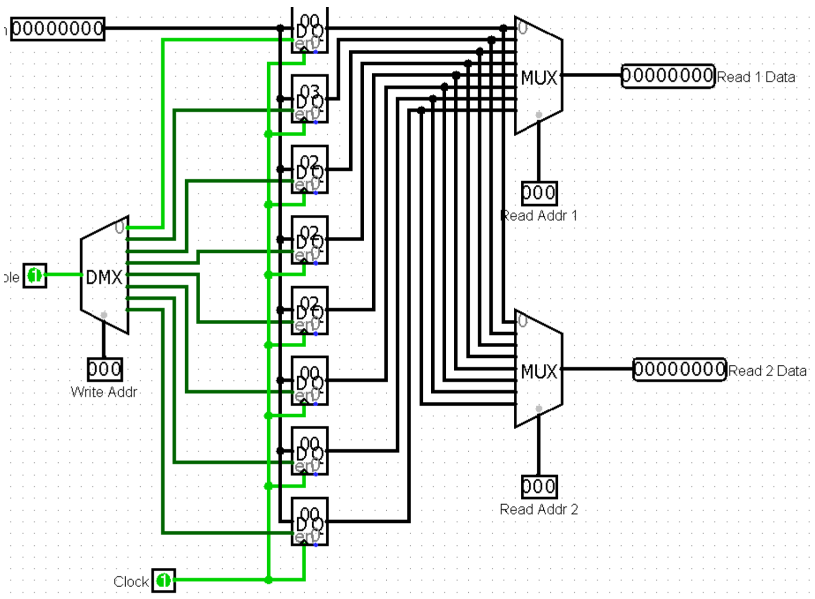


Figure 13: LW SW Sam Demo

We can see from the top screenshot that there is a value (It will be 2) that is being stored at memory address 2 and that he store word bit is 1 on the store word port wire on RAM. In the second screenshot we see the load word bit is 1 on the load word port wire on RAM and that 2 is coming out the must meaning a value of 2 is being stored into the register file. And the last screenshot is the result of the test code's load word and store word operations all completed successfully.

<u>Evidence Of Working Features: Kitchen Sink</u>

In this section I will detail the two features of the kitchen sink processor, both of which are worth 5 points: 2 pseudo instructions in the assembler and 16-bit circuit.

Kitchen Sink: 2 Pseudo Instructions and 16-bit support

I choose to include the 2 pseudo instructions mov, and neg the descriptions, syntax and other information can be found the section on programmers reference and machine code mappings. Figure 14 below show mov in action using test code mov.asm:

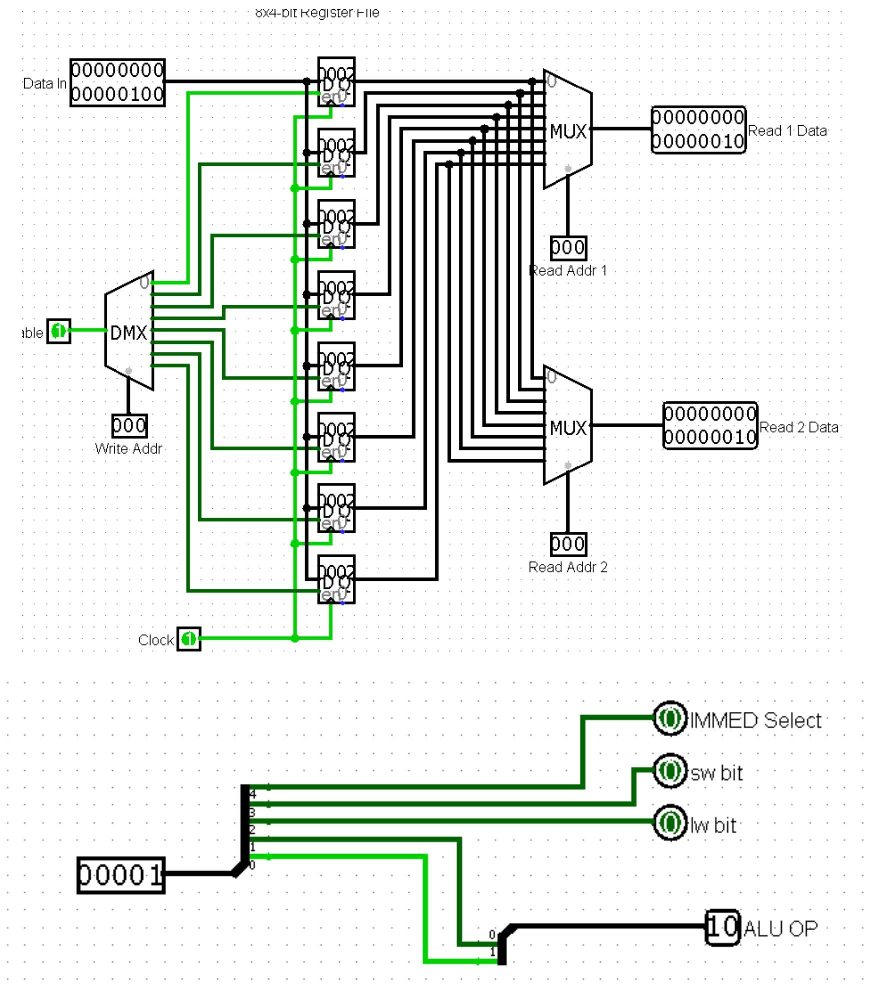


Figure 15: MOV Demo

The test code successfully shows that mov works as first line of code does an addi of 2 into register 0. And then it uses mov to move to copy the value of register 0 into the 7 other remaining registers. Further evidence is that it works is that the control logic is what we expect because mov is just a fancy way of essentially doing add, so the op code and ALU op are the same as add! The second pseudo instruction that was implemented is neg the test code of which is simply neg.asm:

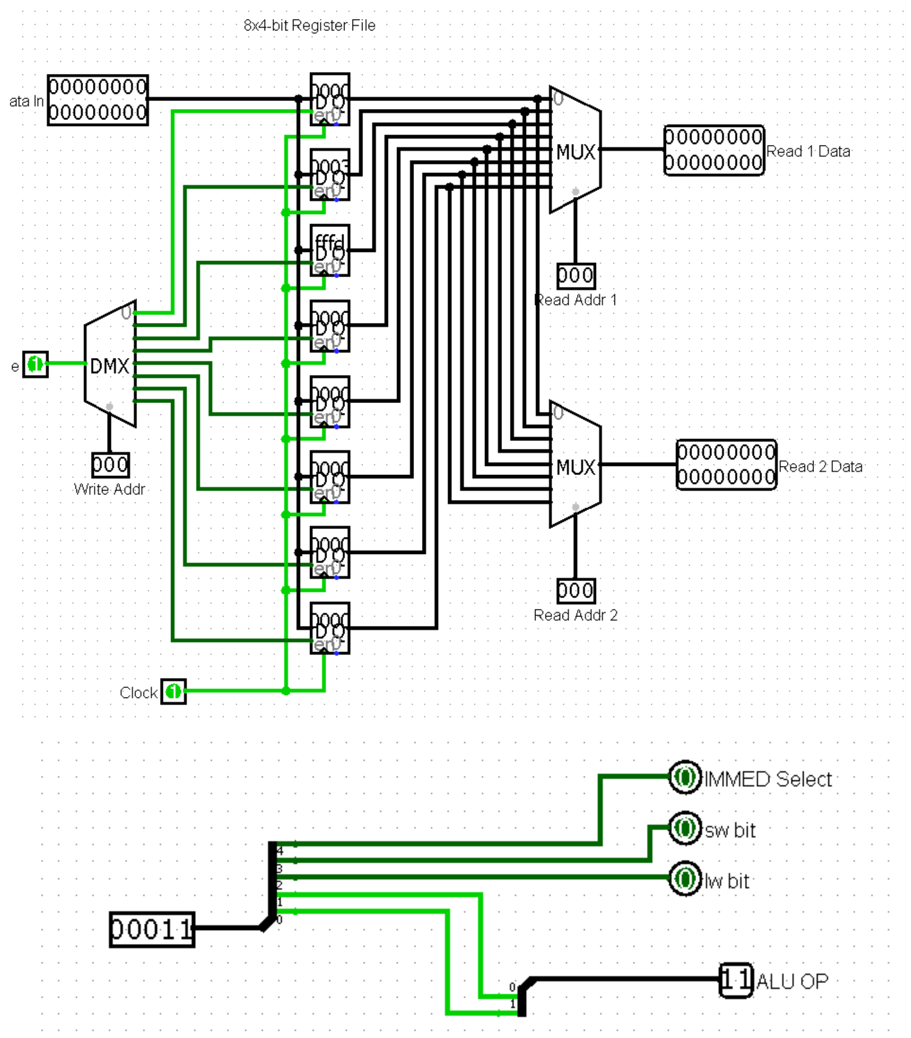


Figure 16: Neg Demo

We can see that neg works as the code first does an addi and puts the value 3 into register 1. The negated value goes into register 2 which is fffd as expected which is -3. It's also fffd because the processor is 16-bits a piece of evidence for 16-bit support. Also the op code is the same as sub because neg is just a glorified subtraction operation! The final piece of evidence we can see is that the processor is 16-bits:

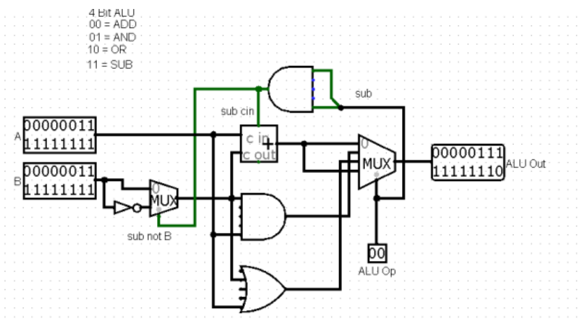


Figure 17: 16-bit Demo ALU

The ALU performs a simple addition (Opcode 00) of the 16-bit numbers 1111111111 + 1111111111 = 11111111110 or 1023 + 1023 = 2026 as expected!

Conclusions

Overall this project was enjoyable my only regret is that I didn’t have more time to work on it. For fun I will likely try and add more to my processor over Spring break and show you when we get back on the 28th! Thanks for teaching a great class John!