North South University

Department of Electrical & Computer Engineering

CSE332

Computer Organization and Architecture

Project Report

Completed by: Emon Sarker (1931461642)

Submitted to

Ms. Tanjila Farah (TNF)

Submission Date: 12/09/2021

Introduction

This project aims to create a simulated data path and an assembler. This is a 16 bit data path which can carry out multiple operations. The assembler is capable of converting assembly code to binary (in a hexadecimal format) that can be inputted in the data path so that the data path functions properly.

Components

Register File

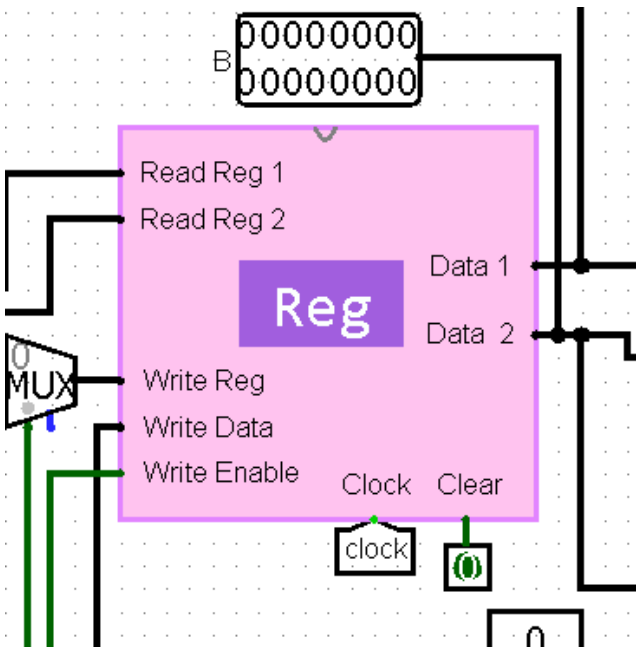

Fig: Screenshot of Register File from data path

The Register File is an integrated circuited built manually that hosts multiple registers inside of it. The primary purpose of it is to store, write and read data in each of the individual registers available in the file. For my project I used **16 registers** that hold **16 bits** of data each.

It has multiple input and output ports that were created to add functionality. The **Clock** port takes in a clock pulse that is needed when updating the state of a register. The **Clear** port takes in a pulse that will reset all the registers to have no value (zero). **Read Reg 1 & 2** are two input ports that take in a 4 bit input that determines which of the 16 registers will be read. When the register is selected, **Data 1 & 2** are the two output ports that displays the 16 bit data stored in the selected registers. **Write Data** takes in a 16 bit input that will be written to a register that has been chosen by the **Write Reg**, which takes in 4 bit to determine to which register the file will be written to. The **Write Enable** takes in a 1 bit signal that enables/disables writing to register files.

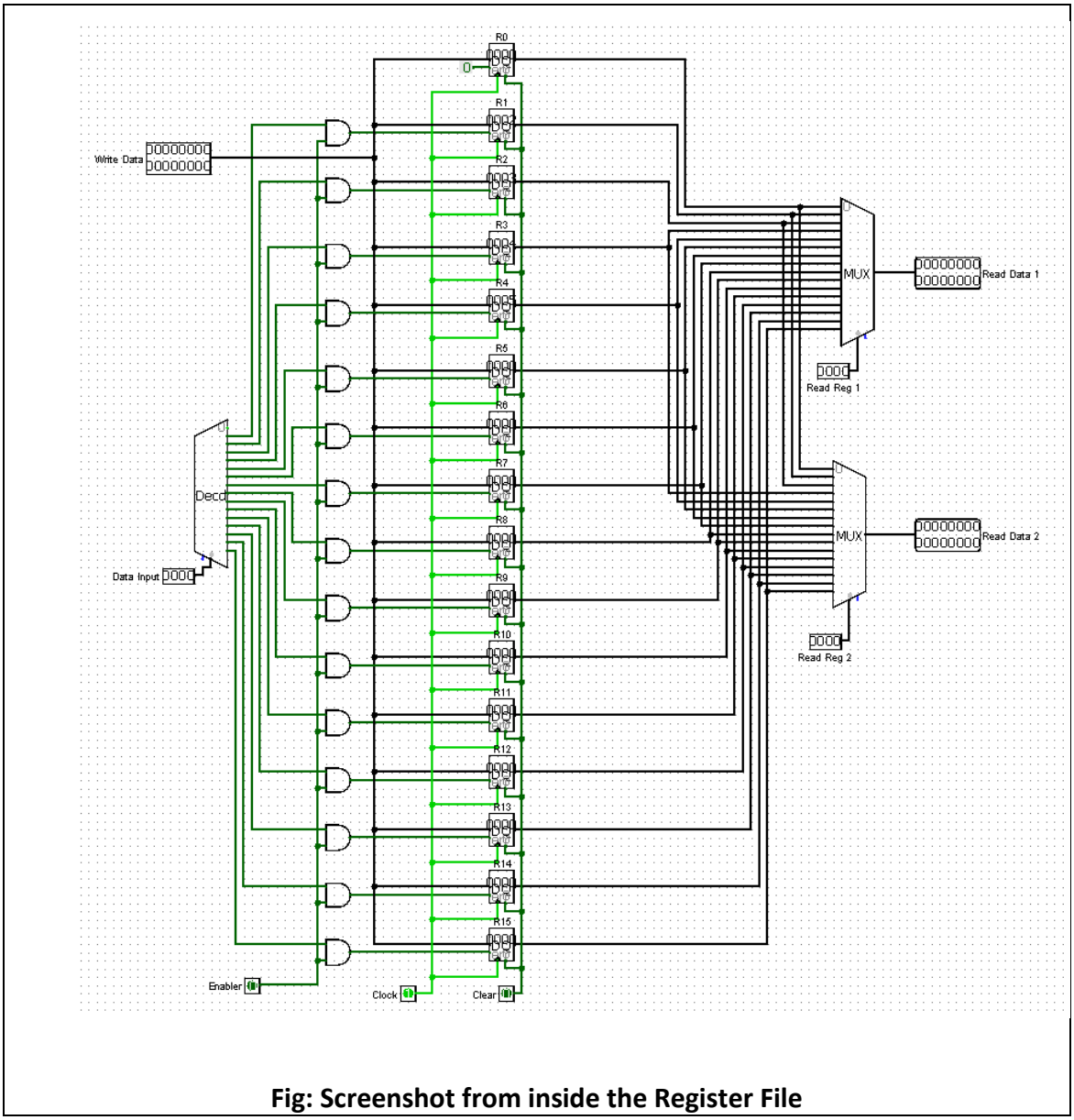

Fig: Screenshot from inside the Register File

The interior of the Register file lets us see the working of it. One notable thing I have done is set the first register (R0) as permanently zero- this is further explained in the documentation. The 16 bit input of the **Write Data** port and the 16 bit outputs of **Read Data 1 & 2** are self-explanatory. We used a Decoder and two Multiplexers to handle selection of registers and output lines respectively. **AND gates** were used to ensure that to write to a register, the register would have to be selected and as well as the write enable signal has to be true.

Arithmetic and Logic Unit

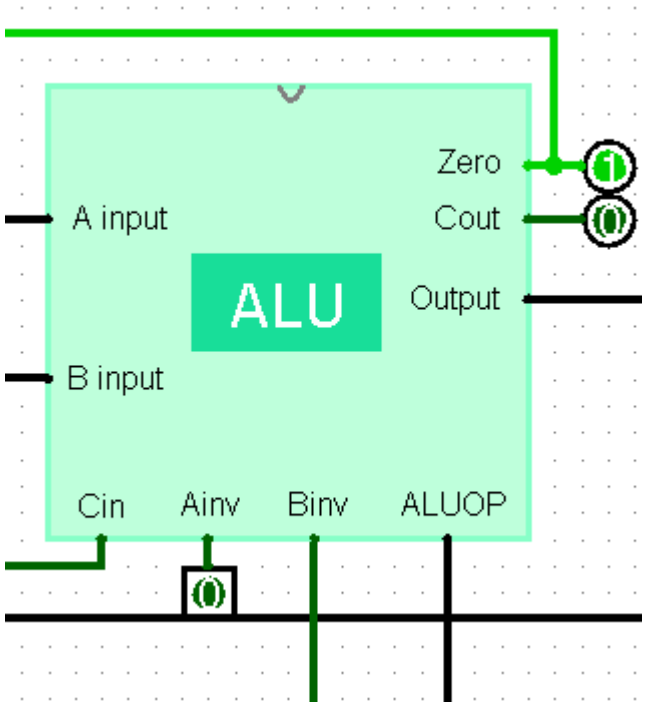

Fig: Screenshot of ALU from data path

The Arithmetic and Logic Unit handles all the operations that requires data manipulation. For example, it is used to add, subtract, shift left, etc. In my project I have built a **16 bit ALU.** The ALU has two input ports: **A input, B input**. Both inputs take in 16 bits of data that is going to be processed. After processing it comes out of the ALU through the **Output**.

There are other parts of the ALU such as **Cin,** which stands for carry in. This is handled by the control unit and is turned on when required is accordance to the OP Code. **Ainv,** which stands of A invert is set to a constant of 0 as it has no purpose in our data path. **B inv,** which stands for B invert, is used therefore it is connected to the control unit. **Zero** shows 1 if the output line is zero, it acts both as an indicator and an enabler for beq operations. **Cout** stands of carry out, this is 1 if there is an overflow of data from addition. Lastly, we have **ALUOP,** which stands for ALU op code, which is used to determine which ALU function will be executed.

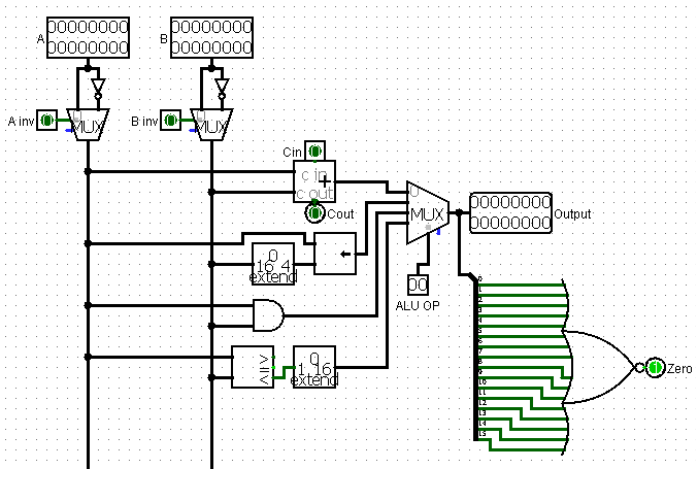

Fig: Screenshot from inside the ALU

Inside the ALU we see that we have two 2:1 multiplexers and a single 4:1 multiplexer. The 2:1 mux are used to determine whether the A or B's inverted input needs to be taken.

We then see the 4 lines of operation that can be taken. The **16 bit adder,** takes in values of A/B/Binv/cin and puts out an output and cout. It is used to handle operations: **sub, add, addi, sq, lw, beq.** The second line of operation we do **sll** operations. Over here the A is the value that is shifted by B. Since we have a 16 bit data path, the maximum shift can be of 16

places therefore we convert the 16 bit input of B to a 4 bit input before it reaches the **shifter** by using a **bit extender**. The third line is operation is a basic **AND** operation done with an **AND gate**. The last line of operation does **slt**, which compares A to B. If A < B, then an output of 1 is found or else an output of 1 is taken. This is done by a **comparator.** The 16 bit **NOR gate** is used to see if the output is zero or not. If it is zero, it shows a 1.

Random Access Memory

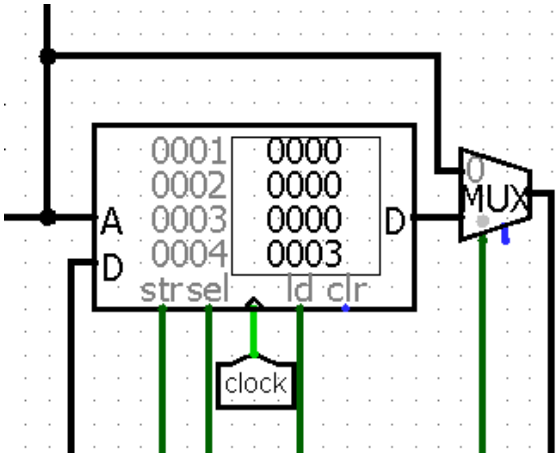

Fig: Screenshot of RAM from data path

RAM, stands for Random Access Memory, is memory component in our datapath. For our 16 bit data path, we set the **Address Bit Width** and **Data Bit Width** to 16 bits.

The input **A** stands for address, it tells us which address to point to based on the value inputted. The input **D** stands for Data, which is where the data that is to be stored in the RAM is taken. The output **D** also stands for data but this one outputs the data based on the value of **A**. **Str** stands for store, when enabled it allows data to be stored in the RAM. **Sel** stands for chip select, when 0 it disables the RAM. **Ld** stands for load, when 1 it allows us to load data from RAM to the output. The inputs **str, sel, ld** is handled by the Control Unit.

Read Only Memory

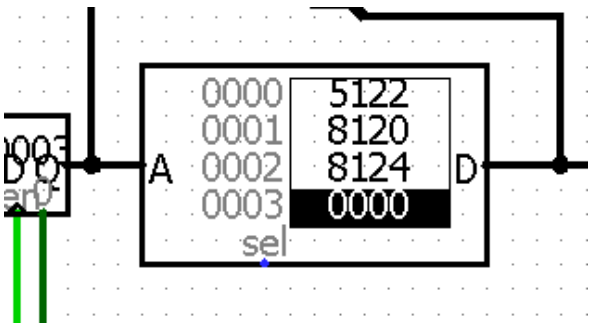

Fig: Screenshot of ROM from data path

ROM, stands for Read Only Memory, it is used in this data path to store information of the instructions. For our 16 bit data path we set the **Address Bit Width** and **Data Bit Width** to 16 bits.

The instruction structure is written in the ISA. Since all instructions ar 16 bits, so is the data bit width. They get executed each time the ROM increments. **A** stands for Address, it is used to access the data stored in the ROM. **D** stands for data and is an output that takes the data from the memory outside.

Program counter

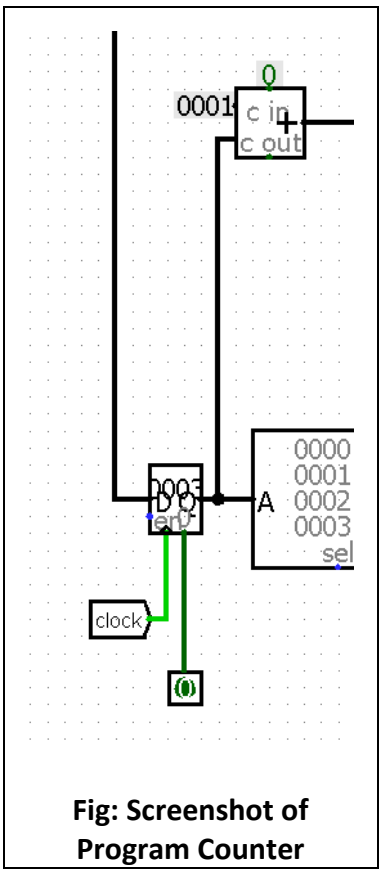

Fig: Screenshot of Program Counter

The program counter is used to keep track, cycle through and jump to different instructions stored in the ROM, in the data path.

The program counter constructed in this data path consist of a **register** and an **adder**. The register keeps count of the current ROM address and therefore stores a 16 bit value. The adder is responsible to increment the data stored in the register therefore it adds 1 to the register. The value in the register is updated when a clock pulse is received.

There also exists a clear that will reset the register when it gets an input of 1.

This program counter is also capable of jumping to different lines if a **beq** or **jmp** operation is performed on it. The way that is possible is explained in the overview.

Overview of the Data Path

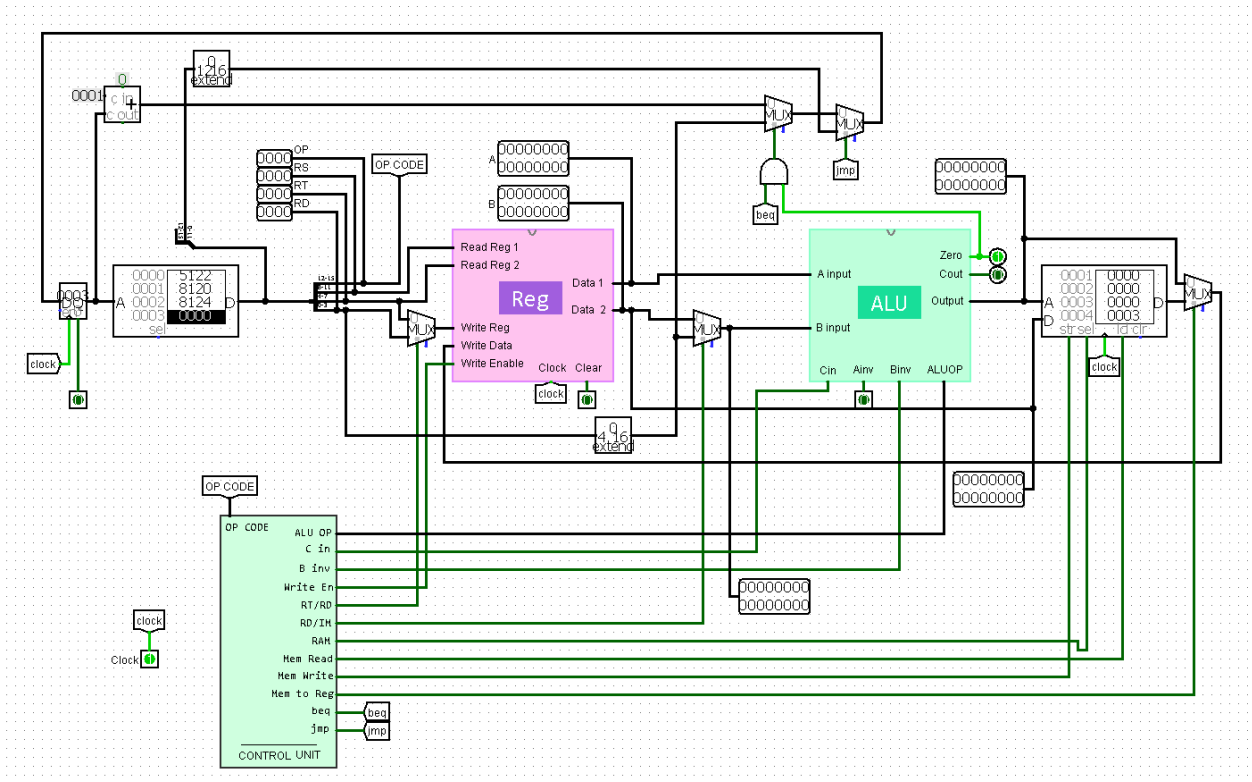

Fig: Data path overview

The data path as shown above is comprised of an array of Registers, Arithmetic and Logic Unit, ROM, RAM, Program Counter, Control Unit and multiple bit extenders and multiplexers spread throughout.

The data path I constructed requires us to write us the instructions and register values as hex codes and then give a clock pulse for It to start working. The control unit is attached to various multiplexers and input nodes in the Reg, ALU, ROM, RAM which lets it control the data path to carry out multiple functions.

The general way the data path functions is once the ROM has instructions loaded onto it, it executed with a clock pulse. The output is split. For **R type** and **I type** instructions it interacts with the Registers (as both source registers and destination registers) and executed logic with the ALU. Afterwards depending on the operation it either stores a value in the RAM or Register; if its is a beq instruction then it jumps to the new line in the program counter if conditions are matched. If it is a jmp instruction then it jumps to the instruction line without passing through either register nor ALU.

Control Unit

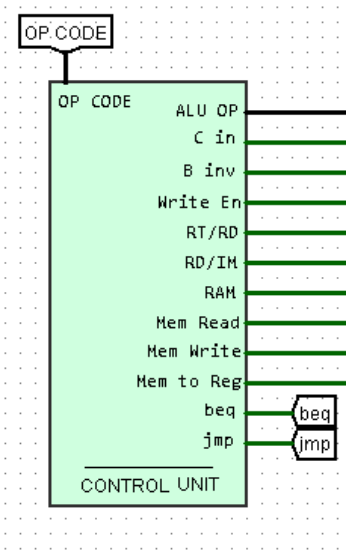

Fig: Overview of Control Unit

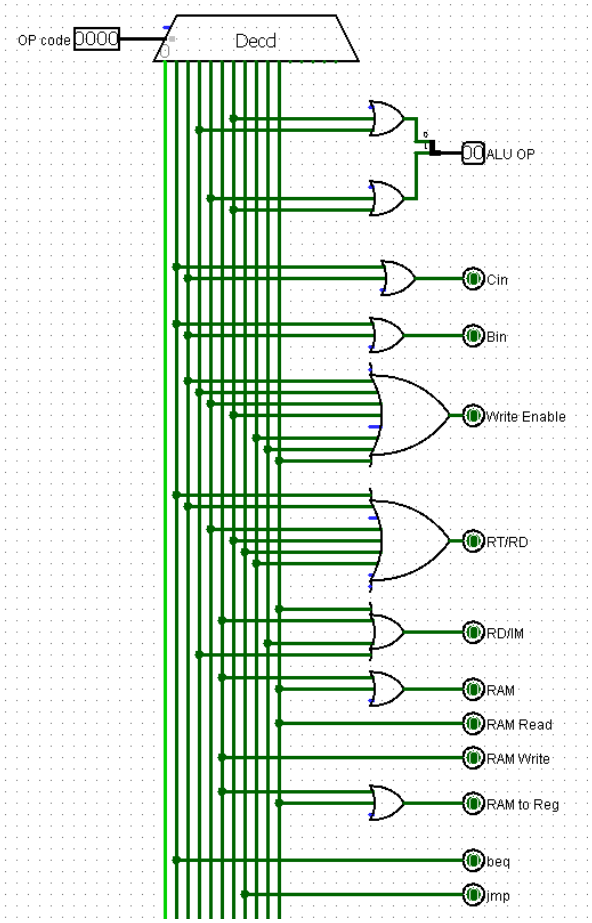

Fig: Inside the Control Unit

Overview

The control unit controls the entire data path. It takes in a 4 bit **OP CODE** and then outputs signals in accordance to the design of the data path (as shown in the ISA) to make the data path functional.

For this design, the controls are listed and explained below:

List of controls

1. ALU OP outputs a signal that controls the Arithmetic Logic Unit of the data path. Depending on the OP CODE, it changes the line of logic that the ALU completes
2. C in means carry in, this the carry in for the adder in the ALU. It is 1 for sub and beq operations
3. B inv means B inverse, this is 1 for beq and sub as well. It inverses the input B in the ALU
4. Write En enables the ability to write into the registers in the register file.
5. RT/RD toggles a multiplexer that lets the data path to switch from R type to I type instruction. It is 1 for R type and 0 for I type
6. RD/IM toggles a multiplexer that lets the data path to switch from R type to I type instruction. It is 0 for R type and 1 for I type
7. RAM En enables the RAM to be modified
8. Mem Read gives a value to the ld of the RAM, if 1 it lets us load value from RAM to store in a register
9. Mem Write gives a value to str of the RAM to help write a value to the RAM
10. RAM to Reg toggles a multiplexer that switches path of write back, if 1 then value is taken from RAM, if 0 then it is taken from 0
11. beq is only 1 when the operation is beq
12. jmp is only 1 when the operation is jmp

Assembler Documentation

Files:

- Assembler_EmonSarker_1931461642.py
- Input.txt
- Output.txt

Requirements:

In order to use the assembler the following is required:

- Python 2.7+
- A python IDE or any console capable of executing a python script

How to use:

- Write the assembly code in the input.txt file in the correct format
  note: there must be only a space between each operand
  - op rs rt rd – for R-type
  - op rs rd im – for I-type
  - op target – for J-type
- Run the .py file
- Read the output.txt file will contain the output in a hexadecimal format

Instruction List

Op code	Type	Binary	Hexadecimal
nop	X	0000	0
beq	I	0001	1
sub	R	0010	2
sll	I	0011	3
and	R	0100	4
sw	I	0101	5
slt	R	0110	6
jmp	J	0111	7
add	R	1000	8
addi	I	1001	9
lw	I	1010	A

Instruction Description

Name	Description
nop	Do nothing, lets the data path remain idle
beq	Check equality between two registers and then jumps to a line if they are equal
sub	Subtraction between two values in the register and stores it to another register
sll	Shifts the values in a register to the left based on the immediate input
and	Applied AND logic bit by bit to two inputs and stores output to a register
Sw	Stores value into the RAM
Slt	Compares two inputs from the registers and stores 1 if A < B in a register
Jmp	Jumps to an instruction line unconditionally
Add	Adds two numbers in registers and stores it to a another register
addi	Adds a number from a register with immediate and then stores it to a register
Lw	Loads value from the RAM

Register List

Note: $zero is a register with a permanent value of 0.

Notation	Register No.	Binary	Hexadecimal
$zero	0	0000	0
$s1	1	0001	1
$s2	2	0010	2
$s3	3	0011	3
$s4	4	0100	4
$s5	5	0101	5
$s6	6	0110	6
$s7	7	0111	7
$s8	8	1000	8
$s9	9	1001	9
$s10	10	1010	A
$s11	11	1011	B
$s12	12	1100	C
$s13	13	1101	D
$s14	14	1110	E
$s15	15	1111	F

Example of reach operation:

Name	Syntax	Comment
nop	X	X
beq	beq $s2 $s3 7	If($s3==$s2) then 7
sub	sub $s1 $s2 $s3	$s3 = $s1 - $s2
sll	sll $s1 $s2 3	$s2=$s1<<3
and	and $s1 $s2 $s3	$s3 = $s1 AND $s2
Sw	sw $s1 $s2 2	Mem[$s1+2]=$s2
Slt	slt $s1 $s2 $s3	If($s1<$s2)then $s3=1 else $s3=0
Jmp	jmp 13	Go to line 13
Add	add $s1 $s2 $s3	$s3 = $s1 + $s2
addi	addi $s1 $s2 3	$s2 = $s1 + 3
Lw	Lw $s1 $s2 $s3	$s2=Mem[$s1+2]

Discussion

In this project I have learned how to build an assembler and data path and get them to work in harmony with each other. We were guided on what to do with the project by our lab instructor from the very beginning and based on what he has taught us, we built our own projects.

For the assembler, we were first shown a demonstration of how it works. The assembler at its most basic level converts high level language to machine language. For example, it turns Assembly language to Binary. However, since our project was in Logisim, it was more convenient for us to write an assembler that outputted machine language in a hexadecimal format as we can easily load that to our ROM.

The assembler code was initially given to us and we were asked to adopt it to our need and even improve it. I changed the assembler to fit the instructions and instruction format for my project based on. Furthermore I improved the assembler by adding a error handler that will handle logical input errors done by the user. It will also tell us the line where the error was caused and halt the entire script.

For the datapath, we had to build on what we were taught in lab class and in theory class. We first had to build a register file with 16 registers that could hold 16 bits of data. We had to make sure we could select, write, store and output data from all of these registers.

Afterwards we had to create an arithmetic logic unit. This was based on the instructions we were given and was up to us to arrange it to any format we deemed necessary. The primary purpose of it was to carry out arithmetic and logical functions and give an output.

We were then introduced to the ROM and the RAM. The ROM is where we store all of our instructions in hexadecimal format and RAM is the memory in which we can write them to.

The entire data path is controlled by a component called a Control Unit. The construction of the control unit is based on a truth table that we also have to design on our own. The control unit controls the entire data path and therefore carries out different operations. The truth table is attached in the different file.

The entire project comes to life in the following way. We write our assembly language in an input.txt file and run the assembler code. We get our binary values in an output.txt file in a hexadecimal format. We input the hex values in the ROM and apply clock pulse to it and the data path carries it all out.