
# NORTH SOUTH UNIVERSITY

Department of Electrical & Computer Engineering

**Computer Organization and Architecture**

CSE332 Project

Submitted To

**Ms. Tanjila Farah (TnF)**

Submitted by

**Shafayet Rajit**

**1921325042**

**Section 4**

Submission Date

**11th September, 2021**

**Introduction:** In this project, I have built a 16-bit CPU that is capable of performing simple arithmetic, logical, branching, and data transfer operations. Currently there are 11 operations. There are 16 registers to carry out these operations.

## Components:

- ROM
- 16-bit Register
- 16-bit ALU
- Control Unit
- RAM
- Bit Extender
- MUXs
- Adder
- Logic Gates

## Circuits:

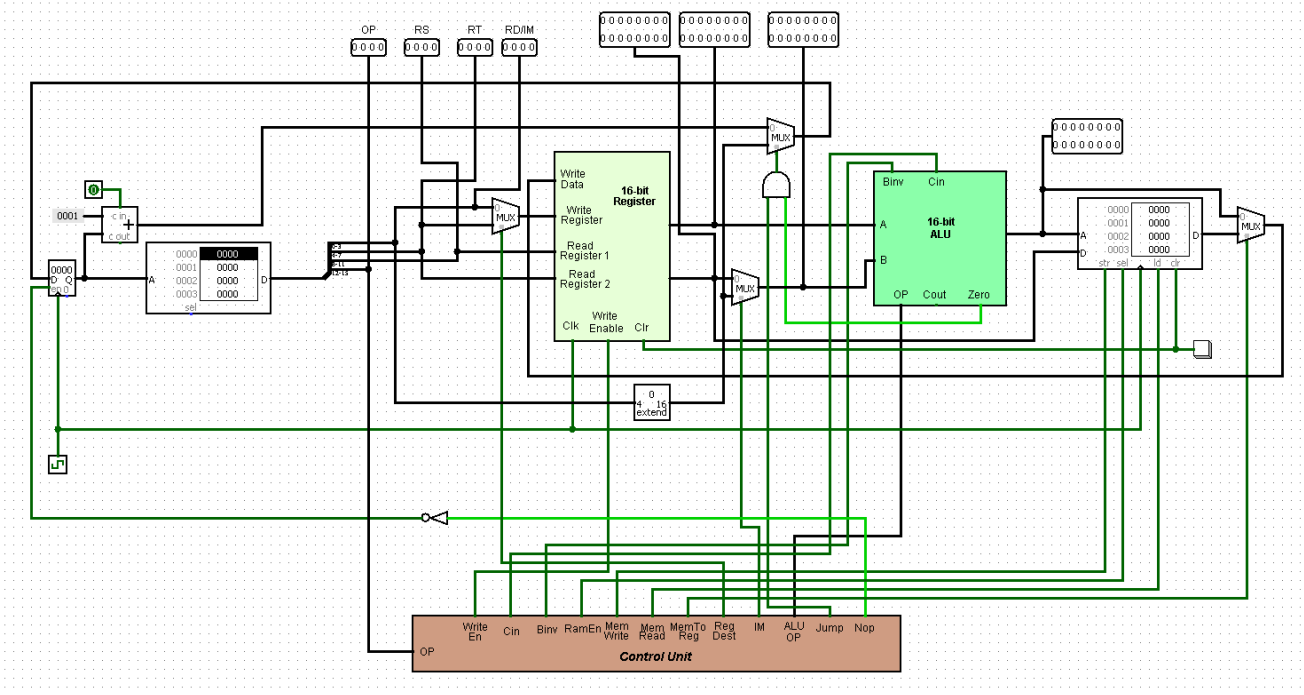


*Fig: Complete CPU Datapath*

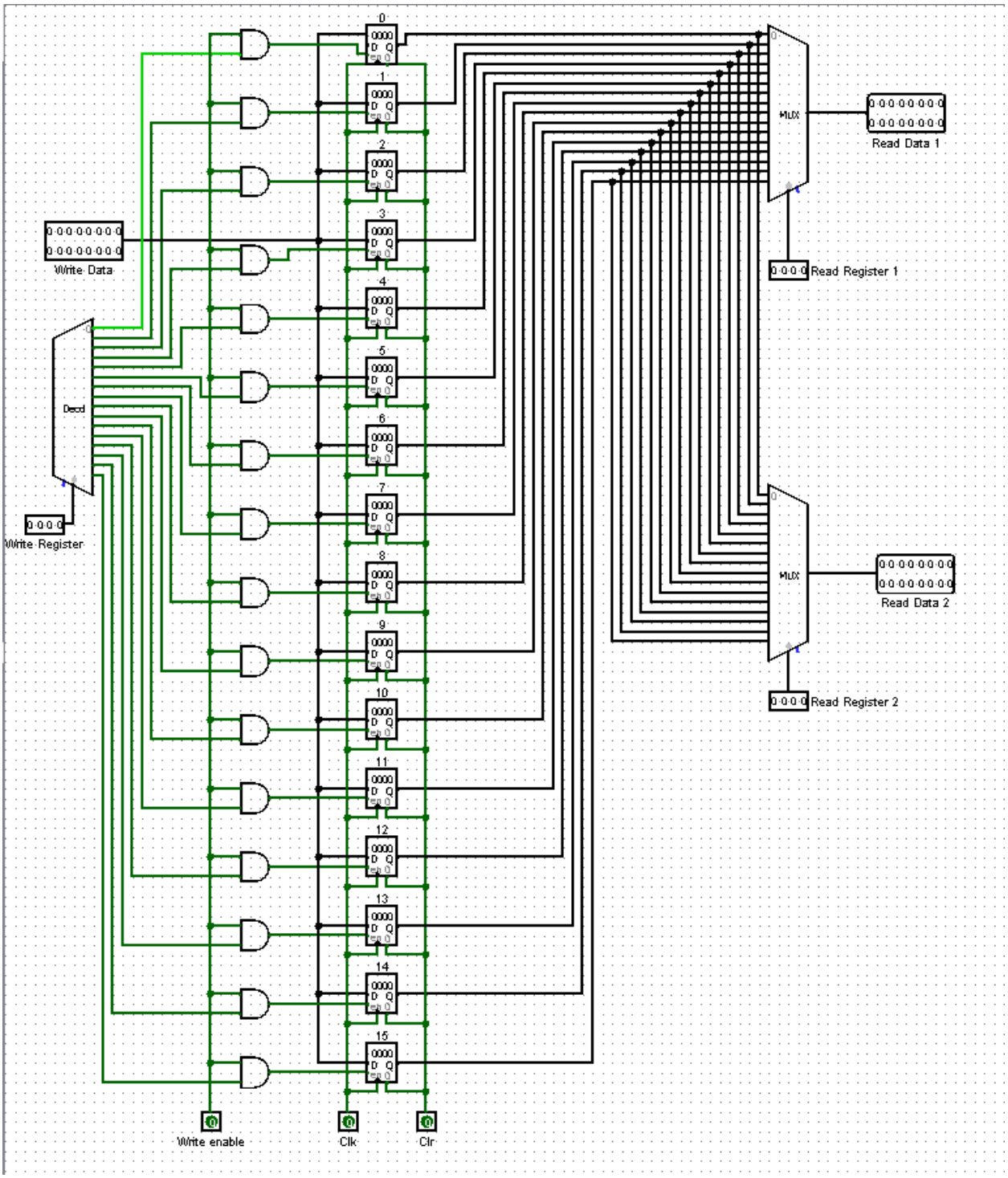


*Fig: 16-bit Register*

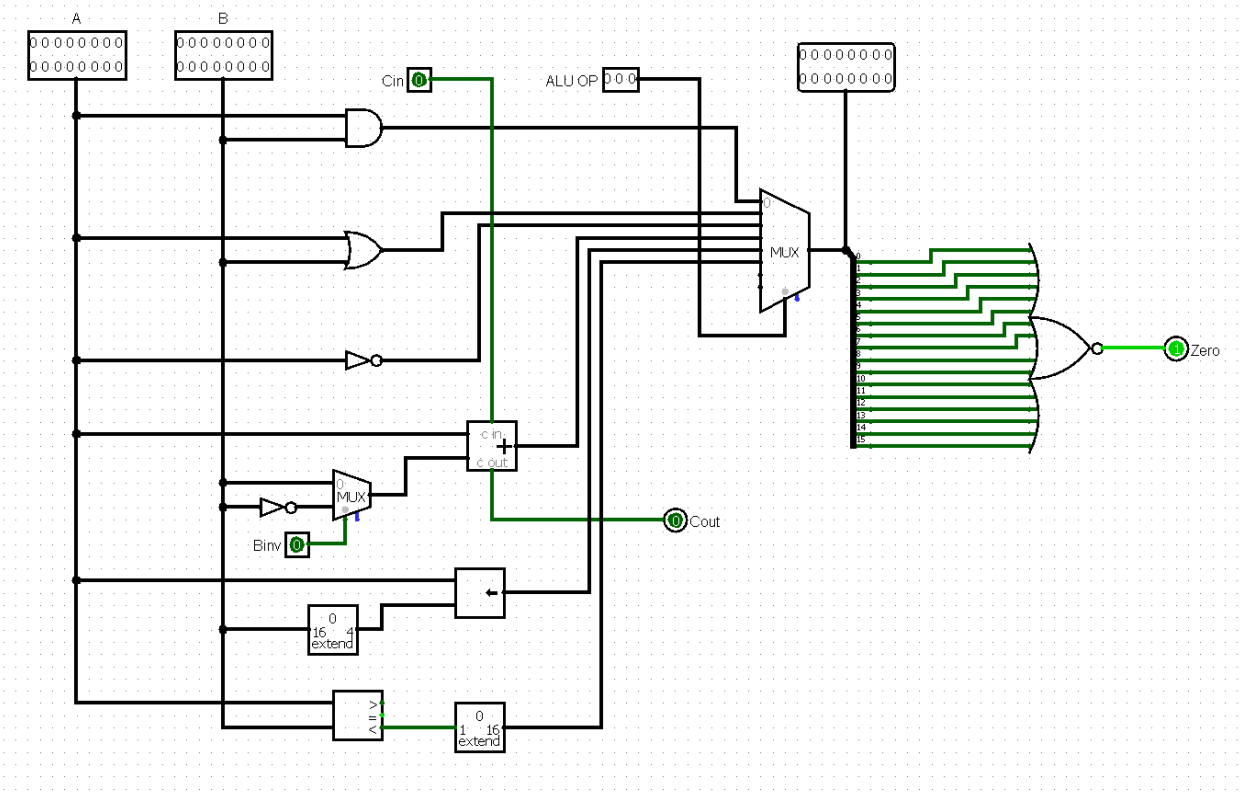


*Fig: 16-bit ALU*

## Control Unit:

The control unit was built to automate the whole process. Without the control unit, I would have to manually turn on numerous inputs in the circuit. As control unit is used, all these inputs signals are automatically generated based on the Op-Code of the running instruction.

In order to automate this process, at first, I have created a table listing all the operations and input signals needed in the Datapath. Then, for each control signal, I have connected all the operations that will use that particular signal using an OR gate. Hence, whenever an Op-Code is entered, the Op-Code will be passed to the control unit and all the necessary control signal for that operation will be ON. By using this control unit, we are eliminating the need for manually turning different input signals ON/OFF based on the Op-Code every time.

The mentioned table is given below:

| Name | Op | ALU OP | | | Cin | Binv | Write En | Ram En | Mem Write | Mem Read | MemTo Reg | RT /IM | Reg Dest | Jump | Nop |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| nop | 0000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| sll | 0001 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| sub | 0010 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| sw | 0011 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| and | 0100 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| beq | 0101 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| jmp | 0110 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| slt | 0111 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| add | 1000 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| addi | 1001 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| lw | 1010 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |

*Table: Control Unit*

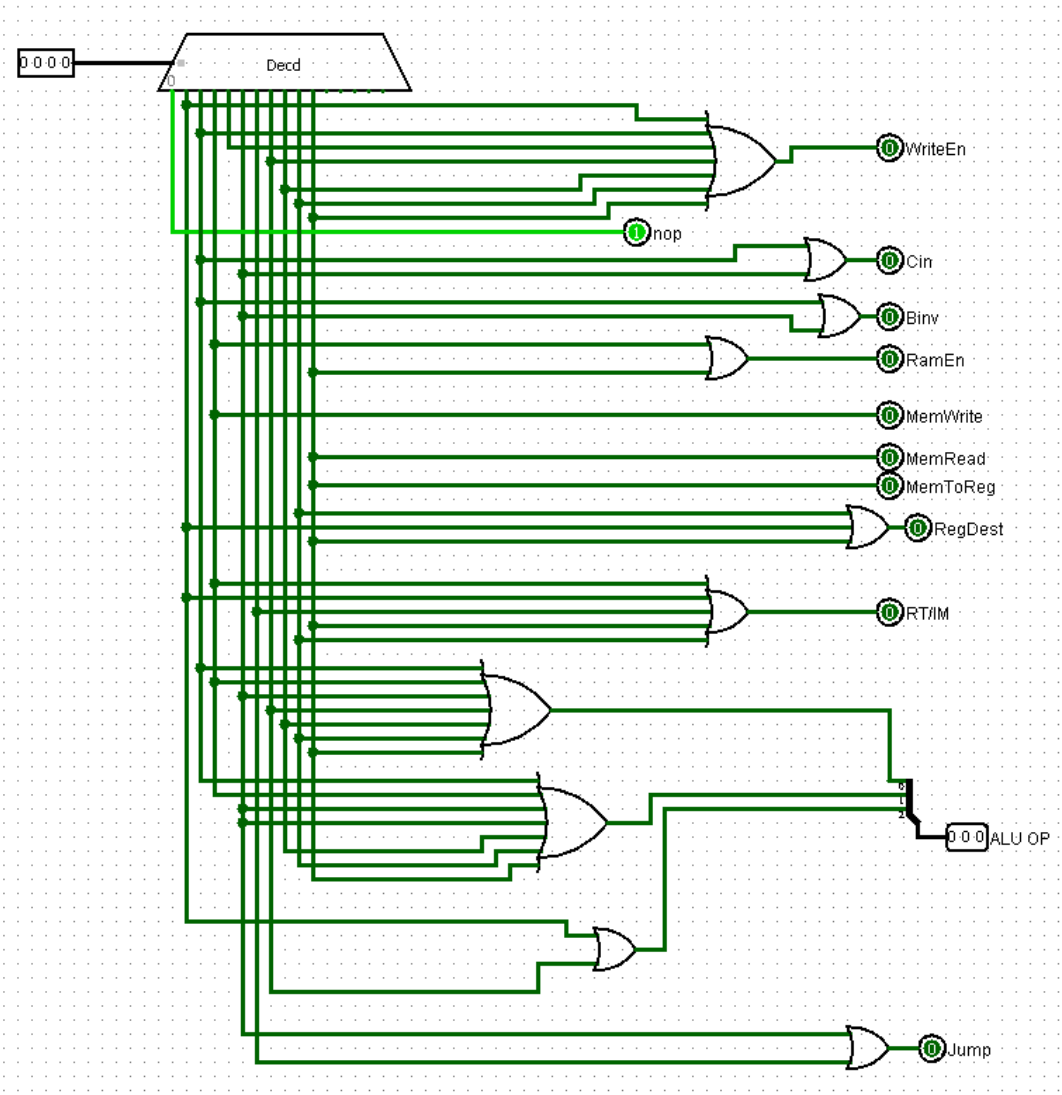


*Fig: Control Unit*

## Assembler:

The main purpose of the assembler is to convert the assembly language to machine code that will be used in the main Datapath.

The user will write instructions in the "input" file using the correct format. The assembler will read the instructions written in assembly language and translate it to proper machine language. After that, the generated machine language will be written in an "output" file which I can use to directly load instructions in the Datapath.

The assembly language instructions will be written in a file named "inputs" and the converted machine language will be written in a file named "outputs". Both of these files must stay in the same directory as the assembler.

As I am building a 16-bit CPU, I have used 16 registers in designing this. Each of the register contains 4 bits. The complete list of registers is given below:

| Register Number | Conventional Name | Usage | Binary Value |
|---|---|---|---|
| 0 | $zero | General purpose | 0000 |
| 1 | $s1 | General purpose | 0001 |
| 2 | $s2 | General purpose | 0010 |
| 3 | $s3 | General purpose | 0011 |
| 4 | $s4 | General purpose | 0100 |
| 5 | $s5 | General purpose | 0101 |
| 6 | $s6 | General purpose | 0110 |
| 7 | $s7 | General purpose | 0111 |
| 8 | $d1 | General purpose | 1000 |
| 9 | $d2 | General purpose | 1001 |
| 10 | $d3 | General purpose | 1010 |
| 11 | $d4 | General purpose | 1011 |
| 12 | $d5 | General purpose | 1100 |
| 13 | $d6 | General purpose | 1101 |
| 14 | $d7 | General purpose | 1110 |
| 15 | $d8 | General purpose | 1111 |

*Table: Registers.*

The Op-Code will be used to decide what operation will run. The Op-Code consists of 4 bits.

| Instructions | Type | Op-Code |
|---|---|---|
| nop | I-Type | 0000 |
| sll | I-Type | 0001 |
| sub | R-Type | 0010 |
| sw | I-Type | 0011 |
| and | R-Type | 0100 |
| beq | I-Type | 0101 |
| jmp | I-Type | 0110 |
| slt | R-Type | 0111 |
| add | R-Type | 1000 |
| addi | I-Type | 1001 |
| lw | I-Type | 1010 |

*Table: Op-Codes.*

## Instruction Description:

**nop:** Stops the program from running.

- Syntax: nop

**sll:** Shifts a register value to the left.

- Syntax: sll $s1 $d2 2
- Operation: $d2 = $s1<<2

**sub:** Subtraction between two registers value and stores the value in a register.

- Syntax: sub $s1 $s2 $d3
- Operation: $d3 = $s1 - $s2

**sw:** Stores the value of a register in the memory.

- Syntax: sw $d1 $s2 3
- Operation: Mem[$d1+3] = $s2

**and:** Bit-by-bit AND between two registers value and stores the value in a register.

- Syntax: and $s1 $s2 $d3
- Operation: $d3 = $s1 && $s2

**beq:** Equal check between two registers. If they are equal, program counter jumps to a certain instruction.

- Syntax: beq $s1 $s2 11
- Operation: if ($s1==$s2), go to $11^{th}$ instruction.

**jmp:** Jump to a certain instruction.

- Syntax: jmp 11
- Operation: Go to $11^{th}$ instruction.

**slt:** Compare less than between two register values. If true, value of the specified register changes.

- Syntax: slt $s1 $s2 $d3
- Operation: If $s1 < $s2, $d3=1. Else, $d3 = 0.

**add:** Addition between two registers value and stores the value in a register.

- Syntax: add $s1 $s2 $d3
- Operation: $d3 = $s1 + $s2

**addi:** Addition between immediate value and a register value. Stores the value in a register.

- Syntax: add $s1 $d2 5
- Operation: $d2 = $s1 + 5

**lw:** Loads a value to the register file from memory.

- Syntax: lw $s1 $d2 3
- Operation: $d2 = Mem[$s1+3]

## Discussion:

The main goal of this project was to build a 16-bit CPU capable of performing simple arithmetic, logical, branching, and data transfer operations.

Firstly, I have created an ISA specifying the operations, op-codes, their syntaxes, instruction formats, and list of registers. I have followed the ISA to build the main Datapath. There are instructions of two formats. One of them is R-type and the other one is I-type. As the CPU consists of 16 bits, I have allocated 4 bits to each register. There is total 16 registers in this CPU.

Secondly, following the ISA, I have built the ALU with necessary arithmetic operations. I have used 16-bit inputs and outputs, logic gates, adder, and MUXs to properly build the 16-bit ALU. I have split the 16 bits of the output and connected them to a NOR gate for ‘zero detection’. ALU have Op-Codes. So, based on different Op-Codes, ALU will perform different operations.

Thirdly, I have built a 16-bit register file. There are 16 registers in this and they are all connected to a decoder. Based on the 4-bit selection pin of the decoder, different register will be activated.

Using two different MUXs, two registers can be passed to the output. We can also write data in a specific register.

Fourthly, I have started building the main Datapath of the CPU. In order to do that, I needed ROM, 16-bit Register, 16-bit ALU, RAM, Bit Extender, MUXs, Adder, and Logic Gates. The ROM is used to organize instructions serially. A register, and an adder are connected to the ROM to successfully go through all the instructions. Then, from the ROM, the 16-bit instructions get split in four 4-bit binaries which get passed to the 16-bit register file. From there, the register outputs are connected to the ALU and in the end, there is a RAM to store values. The RAM is also connected to the register file to store values in a register. Now, in our instructions, we have four parts, Op-Code, RS, RT, RD/IM. We have added additional MUXs, and bit extenders to carry out all the operations perfectly. However, at this moment, we have to manually turn each of them ON/OFF based on the Op-Code which is not an efficient design.

So, fifthly, I have constructed the control unit which will turn necessary signals in the Datapath ON/OFF based on the Op-Code. In order to build the control unit, I have created a table listing all the operations and input signals needed in the Datapath. Then, for each control signal, I have connected all the operations that will use that particular signal using an OR gate. Hence, whenever an Op-Code is entered, the Op-Code will be passed to the control unit and all the necessary control signal for that operation will be ON. By using this control unit, we are eliminating the need for manually turning different input signals ON/OFF based on the Op-Code every time.

Finally, I have connected all the input signals in the circuit to the appropriate signal of the control unit. Based on the ISA, I have tested the Datapath using multiple instructions.