North South University

Department of Electrical & Computer Engineering

CSE332

Computer Organization and Architecture

Project Report

Submitted by:

Name: Ferdous Reza Niloy

ID: 2021281642

Submitted to: Tanjila Farah (TnF)

Introduction: In this project, I created a 13-bit CPU that can perform simple arithmetic, logical, branching, and data transfer operations. There are currently ten operations. These operations are carried out using 8 registers.

Components:

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

Circuits:

Data path: This is the complete Data path of the project. Here, for a program counter, I used a ROM and used the register as a buffer for program counter input. For storing data I used a ram and I used sub-circuits of the Register File, ALU and Control Unit to complete the operations

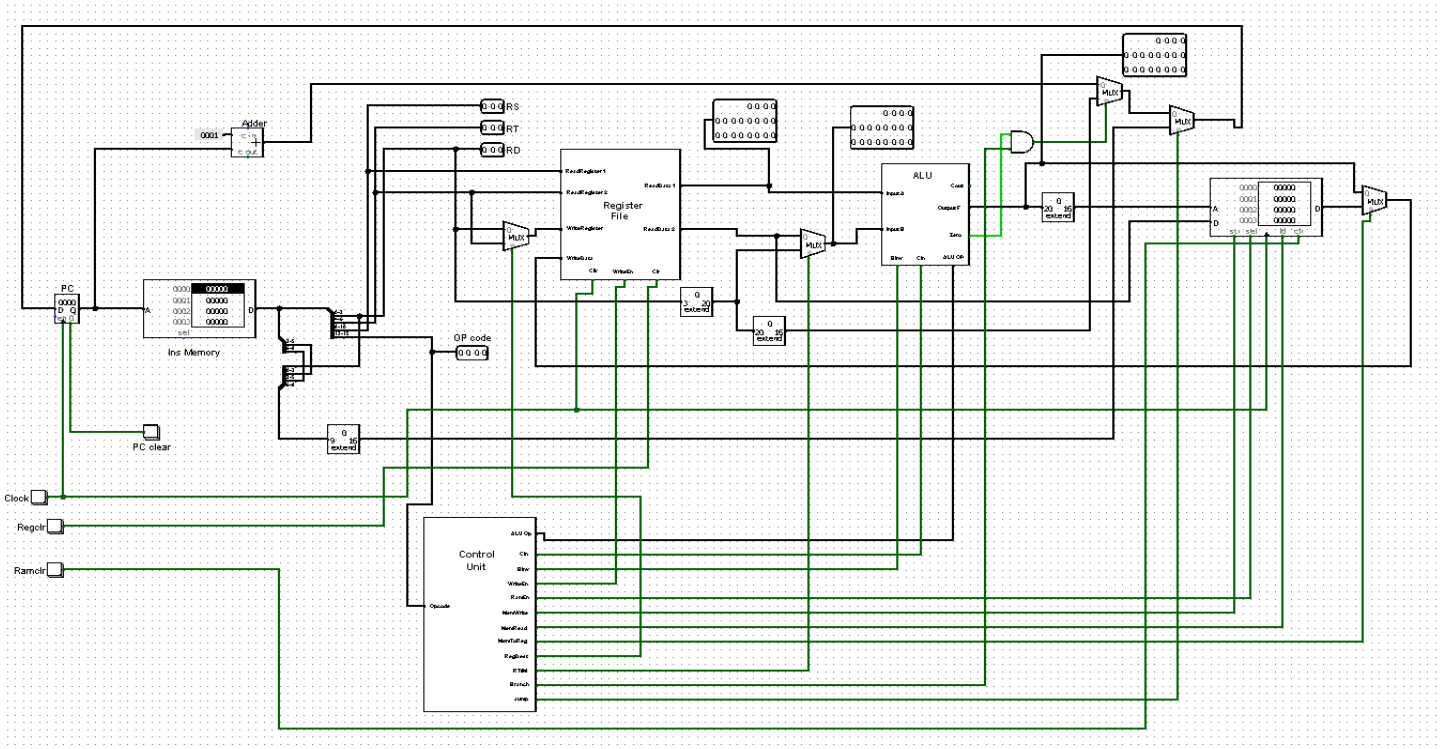

Figure: Complete CPU Data path

Register file: A register file is a small collection of high-speed storage cells located within the CPU. There are two read data pins, one write-data pin, and two register number pins on the register file. Here I used 16 registers. Here the “Write Data” input will be written to the register and the select pin named “Write Register” will determine the destination. I also added a write enable button and connected with and gates so that I can control when data can be written in the register. Then there are “Read Register 1” and “Read Register 2” for reading data of the registers to pass to ALU

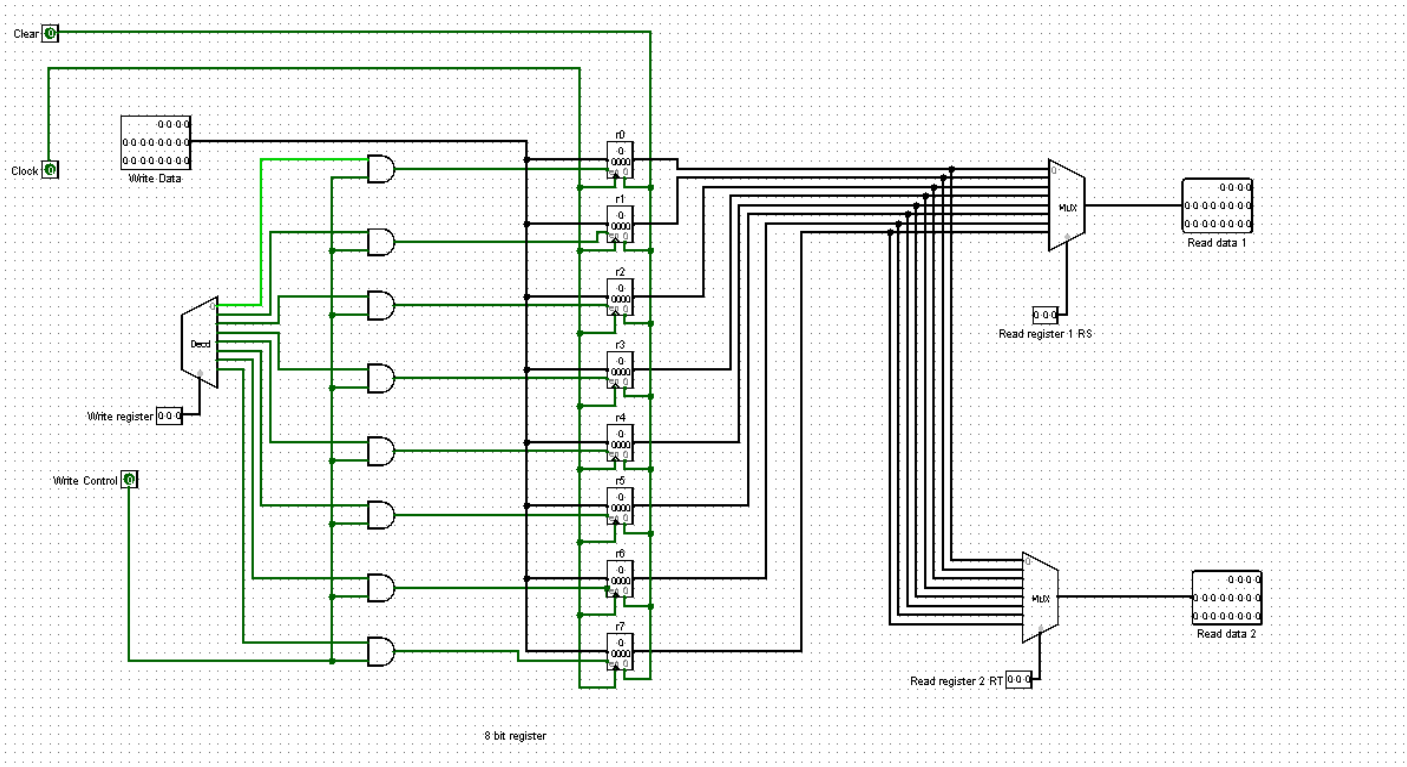

Figure: Register

ALU: Input A and B receives data from register and passes through the output F. As per the operations given to me, I needed to use Shifter for shifting leftward, AND gate for AND operation, Comparator to check equality or smaller for slt and branch on equal operation then and lastly a modified adder for addition, subtraction, lw , sw etc operations.

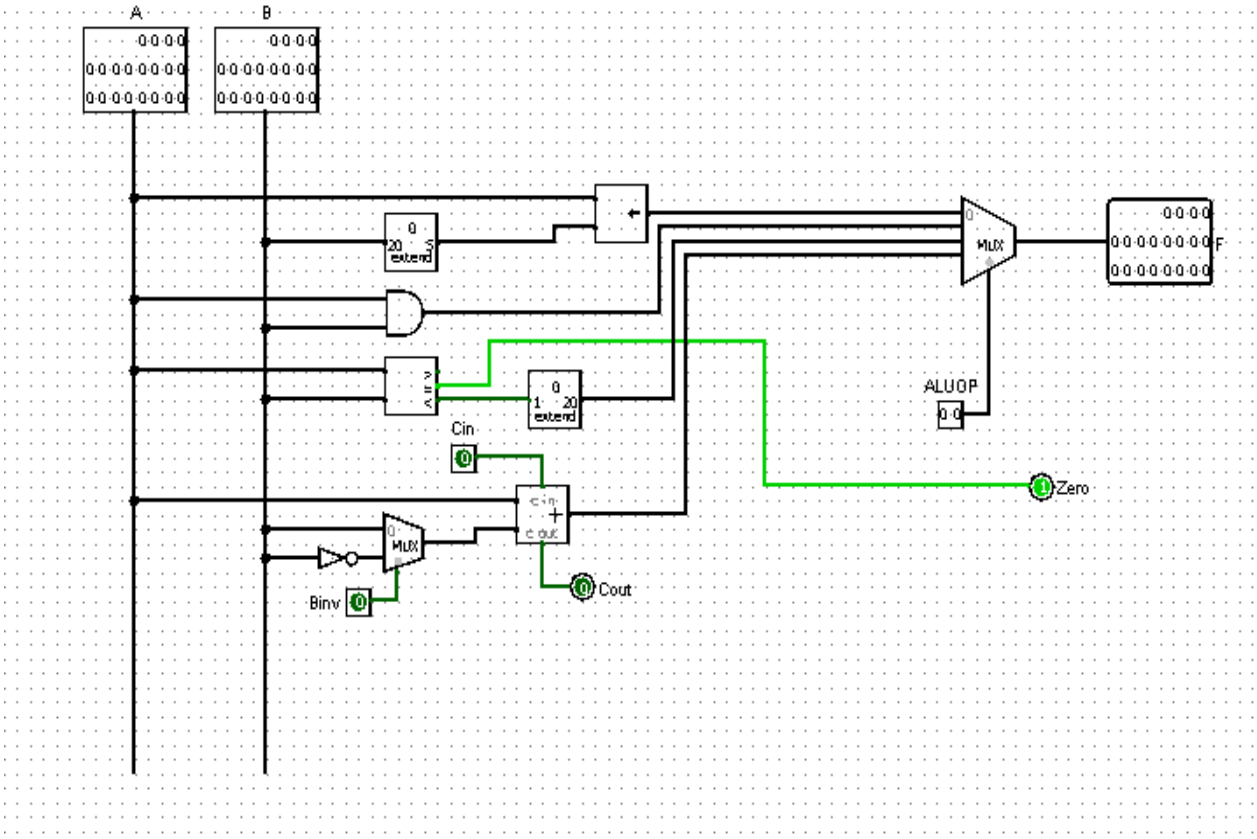

Figure: ALU

Control Unit: I need a control unit to send some specific signal for a specific operation.

ALUop: ALU op signal control the ALU operation. According to signal ALU execute the operation and give the output value.

- **Cin:** Control unit gives Cin signal only sub operation.
- **Binv:** Control unit gives this signal when the sub operation executed.
- **WriteEn:** This signal works for write into the register.
- **RAMEn:** This signal works when lw and sw instruction is executed.
- **MemWrite:** This signal use when store value into the memory.
- **MemRead:** Control unit provides this signal when need to read from memory.
- **MemToReg:** This signal provides when need to load a value.
- **RegDest:** RegDest select the register.
- **RT/IM:** This works for ALU source register.
- **Branch:** This signal provides control unit only for beq.
- **Jump:** This signal provides control unit only for jmp.

Control Unit Table: According to the operations given above, I prepared this control unit table and created a circuit following the table. As per given instructions, I needed to use a 4:16 decoder. According to the Opcodes given, it will generate a specific signal.

hex	*OP*	*Instruction*	*ALU OP*		*Cin*	*Binv*	*Write En*	*Ram En*	*Mem Write*	*Mem Read*	*Mem to Reg*	*Reg Dest*	*RT/IM*	*Branch*	*Jump*
0	**0000**	**nop**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	**0001**	**lw**	**1**	**1**	**0**	**0**	**1**	**1**	**0**	**1**	**1**	**1**	**1**	**0**	**0**
2	**0010**	**addi**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**0**
3	**0011**	**sw**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**0**	**0**
4	**0100**	**beq**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**
5	**0101**	**slt**	**1**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
6	**0110**	**and**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
7	**0111**	**sub**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
8	**1000**	**jmp**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**
9	**1001**	**sll**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
A	**1010**	**add**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Figure : Control Unit Table

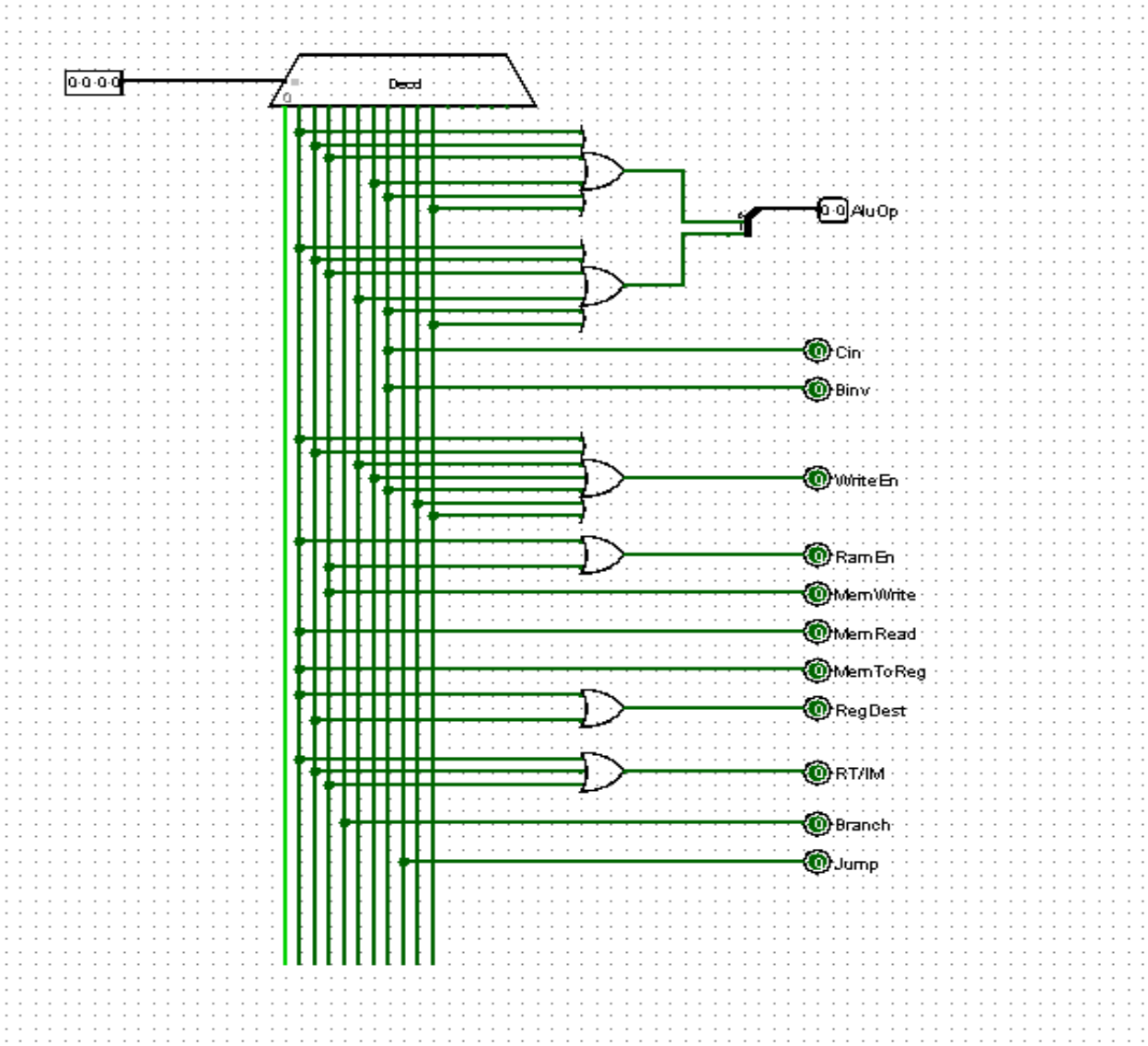

Figure: Control Unit

Instruction Set Architecture Design(ISA)

Objectives: My objective was to design a 13 Bit ISA which can solve a particular problems like simple arithmetic & logic operations, branching and loops.

Types of Operands: I need register operands to implement arithmetic instructions, and memory operands to implement data transfer instructions from memory to register. As a result, I'll need two sorts of operands.

- Register based.
- Memory based.

Operations: I will allocate 4 bits of opcode, so the executable instructions number will be 2^4 or 16.

Types of operations: In my design, there will be five different types of operation. The operations are:

- Arithmetic
- Logical
- Data Transfer
- Conditional Branch
- Unconditional Jump

Formats:

I use three types of formats for my ISA. They are:

- Register Type – R type
- Immediate Type – I type
- Jump Type - J Type

Category	Operation	Name	Type	OpCode	Syntax	Comments
	No operation	nop		0000	nop	
Data transfer	Load word	lw	I	0001	lw r0 r1 2	r1 = Mem[r0+2]
Arithmetic	Add number with an immediate	addi	I	0010	addi r1 r2 5	r2 = r1+5
Data transfer	Store word	sw	I	0011	sw r0 r1 2	Mem[r0+2] = r1
Conditional	Check equality	beq	I	0100	beq r1 r2 4	If(r1==r2) then go to line 4
Conditional	Compare less than	slt	R	0101	slt r1 r2 r3	If(r1<r2) then r3 = 1 else r3 = 0
Logical	Bit-by-bit and	and	R	0110	and r1 r2 r3	R3 = r1 & r2
Arithmetic	Subtraction	sub	R	0111	sub r1 r2 r3	r3 = r1-r2
Unconditional	Jump	jmp	J	1000	jmp 6	Go to line 6
Logical	Shift left	sll	R	1001	sll r1 r2 r3	r3 = r1<<r2
Arithmetic	Add two numbers	add	R	1010	add r1 r2 r3	r3 = r1 + r2

R Type ISA Format

OpCode	rs	rt	rd
4 bits	3 bits	3 bits	3 bits

I Type ISA Format

OpCode	rs	rt	Immediate
4 bits	3 bits	3 bits	3 bits

J Type ISA Format

OpCode	Target Address
4 bits	9 bits

List of Register:

As we have allocated three bits register, so the number of registers will be $2^3 = 8$.

Register Number	Conventional Name	Usage	Binary Value
0	R0	General Purpose	000
1	R1	General Purpose	001
2	R2	General Purpose	010
3	R3	General Purpose	011
4	R4	General Purpose	100
5	R5	General Purpose	101
6	R6	General Purpose	110
7	R7	General Purpose	111

Assembler Documentation:

Introduction: Our task was to design an assembler which will convert the assembly code to machine language.

Objective: Our main goal was to generate a machine code from a file containing assembly language. The assembler reads a program written in an assembly language, then translate it into binary code and generates an output file containing machine code.

How to use: In the input file, the user has to give some instructions to convert into machine codes. The system will convert valid MIPS instructions into machine language and generate those codes into an output file.

Input File: The input file named “inputs”. User will write down the MIPS code in this file.

List of Tables

Register List

We have selected registers from r0-r7 for general purpose. We assigned 3 bits for each of the register as we know in the instruction field in our ISA containing the register rs, rt and rd contains 3 bits each.

Conventional Name	Register Number	Binary Value
r0	0	000
r1	1	001
r2	2	010
r3	3	011
r4	4	100
r5	5	101
r6	6	110
r7	7	111

Op-Code List: We have selected following op codes and assigned op-code binary values (4 bits) for each of the op codes.

Name	Type	OpCode
nop		0000
lw	I	0001
addi	I	0010
sw	I	0011
beq	I	0100
slt	R	0101
and	R	0110
sub	R	0111
jmp	J	1000
sll	R	1001
add	R	1010

Instruction Description

nop: No operation.

sll: It shifts bits to the left and fill the empty bits with zeros. The shift amount is depended on the register value.

- **Operation:** r3 = r1 << r2
- **Syntax:** sll r1 r2 r3 r1 = value , r2 = no of shift , r3 = destination register

and: It AND's two register values and stores the result in destination register. Basically, it sets some bits to 0.

- **Operation:** r3 = r1 & r2
- **Syntax:** and r1 r2 r3

lw: It loads required value from the memory and write it back into the register.

- **Operation:** r1 = Mem[r0+immediate]
- **Syntax:** lw r0 r1 immediate

slt: If r1 is less than r2, r3 is set to one. It gets zero otherwise.

- **Operation:**

```
if (r1 < r2)
        r3 = 1
else
        r3 = 0
```

- **Syntax:** slt r1 r2 r3

add: It adds two registers and stores the result in the destination register.

- **Operation:** r3 = r1 + r2
- **Syntax:** add r1 r2 r3

addi: It adds a value from a register with an integer value and stores the result in a destination register.

- **Operation:** r1 = r2 + immediate
- **Syntax:** addi r2 r1 immediate

sub: It subtracts two registers and stores the result in the destination register.

- **Operation:** r3 = r1 – r2
- **Syntax:** sub r1 r2 r3

sw: It stores specific value from register to memory.

- **Operation:** Mem[r0+immediate] = r1
- **Syntax:** sw r0 r1 immediate

jmp: Jumps to the calculated address.

- **Operation:** jum to target address
- **Syntax:** jmp target

beq: It checks whether the values of two registers are the same or not. If it's the same, it performs the operation located in the address at offset value.

- **Operation:** if (r1==r2)

  jump to immediate

  else

  goto next line
- **Syntax:** beq r1 r2 immediate

Limitation:

The user has to give spaces between instruction words and nothing else like "," or "-" in between them in the "inputs" file. If user don't follow this format, the system will show a valid code as invalid. This CPU uses direct address and also the immediate has only is 3 bits for the I type instruction for this reason when using immediate , it will work with constants between 0-7 range.

User Manual:

To run the program, one needs to run the python file called "**main.py**" which is provided in the Assembler folder. If one wants to see the code then open the "**main.py**" file, it is absolutely necessary that the folder which is containing the program, has a file named "**inputs**" with no extension. This is the file from where the assembler reads the assembly codes. The program reads the code from **"inputs"** file and writes the corresponding binary code in a file called **"outputs".** We have already provided an input file with corresponding output file in the project folder. If one wants to try his/her own assembly code then, he/she needs to write the codes to the application through the input file. Then he/she has to load the **"outputs"** in the rom. One important thing to notice is that, each line of the input file can only contain one instruction and words must be separated by spaces.

Conclusion:

The major purpose of this project was to develop a 13-bit CPU that could perform basic arithmetic, logical, branching, and data transfer functions. First and foremost, I developed an ISA that defines the operations, op-codes, syntax, instruction formats, and register list. The main Datapath was developed in compliance with the ISA. Instructions are classified into three categories. The first is R-type, while the others are J and I-type. Because the CPU has 13 bits, I assigned each register 3 bits. This CPU has 8 registers in total. Then, in accordance with the ISA, I constructed the ALU with the required arithmetic operations. I used 20-bit inputs and outputs, logic gates, adders, and MUXs to properly design the 20-bit ALU. For 'zero detection,' I isolated the output's 20 bits and attached them to a NOR gate. ALU employs Op-Codes. As a result, depending on the Op-Codes, ALU will conduct various operations. After that, I made a 20-bit register file. This has eight registers, each of which is associated with a decoder. Depending on the decoder's 3-bit selection pin, different registers will be triggered. Using two distinct MUXs, two registers can be sent to the output.

Furthermore, we can write data to a specified register. Then I started building the CPU's main Datapath. To do this, I needed ROM, a 20-bit register, a 20-bit ALU, RAM, a bit extender, MUXs, an adder, and logic gates. The ROM is used to order instructions serially. A register and an adder are attached to the ROM in order to correctly finish all of the instructions. The 13-bit instructions are then extracted from the ROM and divided into one 4-bit binary and three 3-bit binaries, which are subsequently given to the 20-bit register file. The register outputs are then connected to the ALU, and values are stored in RAM. The RAM is linked to the register file as well, allowing it to store values in registers. Our instructions now contain four sections: Op-Code, RS, RT, and RD/IM.More MUXs and bit extenders have been included to guarantee that all processes operate well. However, we must currently toggle each of them ON/OFF manually based on the Op-Code, which is wasteful.

Finally, I created the control unit that will toggle the relevant signals in the Datapath on and off based on the Op-Code. I generated a table that details all of the operations and input signals required in the Datapath in order to build the control unit. Then, for each control signal, I connected all of the actions that will use that signal with an OR gate. As a result, anytime an Op-Code is input, it is delivered to the control unit, and all of the relevant control signals for that operation are activated. By using this control unit, we avoid the requirement to manually toggle distinct input signals ON/OFF based on the Op-Code. Finally, I linked all of the circuit's input signals to the corresponding control unit signal. I put the Datapath through its paces using multiple instructions dependent on the ISA.