CSE 306

Computer Architecture Sessional

Assignment 3: 4-bit MIPS Design, Simulation, and Implementation

Submitted By:

Group No: 02
Sub-Section: A2
Level: 3 **Term:** 1

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology (BUET)

Group Members:

1. Sumaiya Saeha (1905033)
2. Fariha Zaman Aurin (1905049)
3. Debjany Ghosh Aronno(1905053)
4. Kh Md Ibtihal (1905055)
5. Nur Uddin Ibne Huda(1905056)

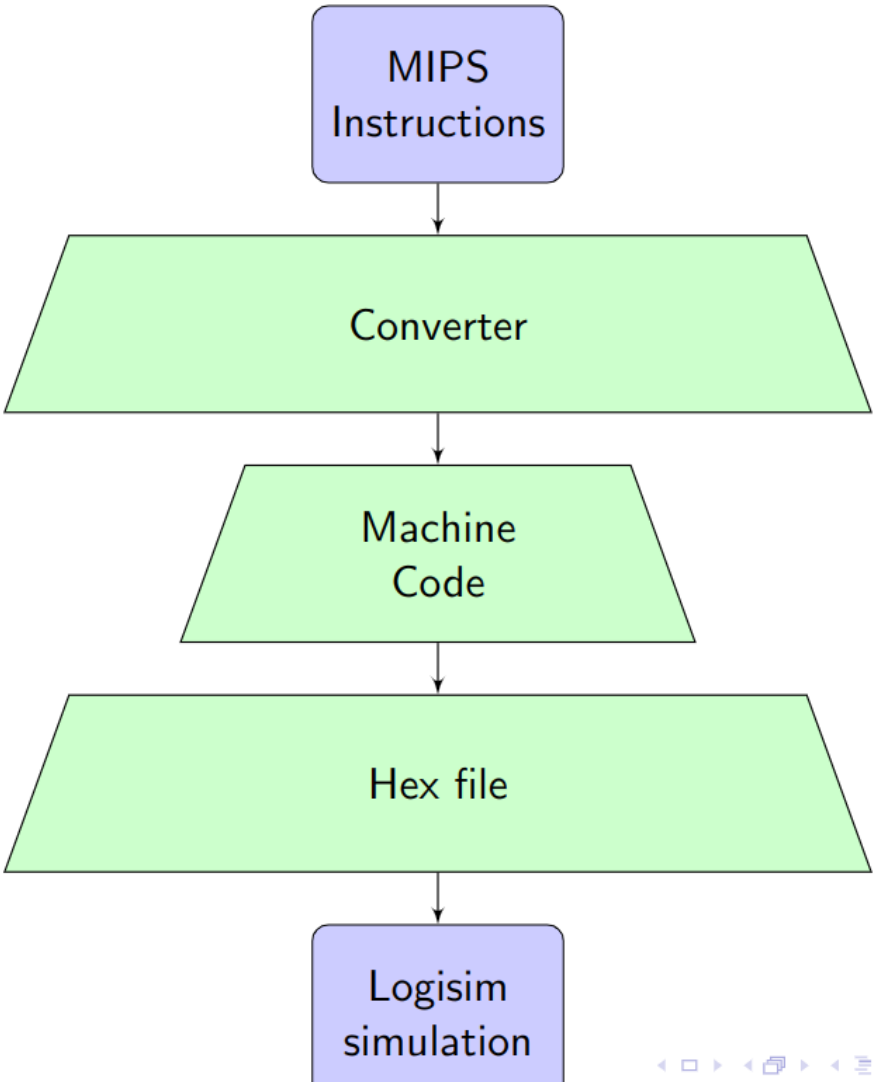
Date of submission: 27 February 2023

Problem Specification:

In this assignment, we had to design, simulate (in S/W), and implement (in H/W) a modified and reduced version of the MIPS instruction set

Introduction:

A processor (CPU) is the logic circuit that responds to and processes the basic instructions that drive a computer.
The processor that uses MIPS (Microprocessor without Interlocked PipelinedStages) is a family of reduced instruction set computer(RISC) instruction set architectures (ISA) is known as a MIPS processor.Each instruction takes 1 clock cycle to be executed. So, clock cycle is the time to execute the longest instruction in the MIPS instruction set.
The main components of the processor are as follows:



1. 4-bit ALU:
Responsible for all the arithmetic and logical operations along with all the addition and subtraction operations required for finding load or store addresses, unconditional jump or branching instructions.

2. Program Counter:
Program Counter (PC) is an 8-bit register that keeps track of the current instruction. After each instruction, PC is incremented by 1 (the ROM we have used is word (16-bit) addressable, adding 1 takes us to the next instruction written in the ROM). The stored value indicates the Instruction memory address.

3. Instruction memory:
Stores the actual hex code of the instruction that is converted to binary to run the processor.

4. Data memory:
Stores 4-bit data and works as the main memory.

5. Register file:
We have used 6 registers namely $zero, $t0, $t1, $t2, $t3, $t4. $zero register stores value 0H. The others are general purpose registers.

6. Control unit:
Decodes the instruction by giving the selection input to all the MUXs, Register File, Data Memory, and ALU.

We will construct the data path and control unit of MIPS Instruction set:
■ The memory-reference instructions load word (lw) and store word (sw).
■ The arithmetic-logical instructions add, sub, addi, subi, AND, OR, NOR, ANDI, ORI, sll and srl.
■ The instructions branch equal (beq), branch not equal (bneq) and jump (j)

Instruction Set:
Our MIPS instruction is 16 bits long and we have 4 formats.

- **R Type**

Opcode	Src Reg1	Src Reg2	Dst Reg
4 bits	4 bits	4 bits	4 bits

- **S Type**

Opcode	Src Reg1	Dest Reg	Shamt
4 bits	4 bits	4 bits	4 bits

- **I Type**

Opcode	Src Reg1	Src Reg2/Dest Reg	Address / Immediate
4 bits	4 bits	4 bits	4 bits

- **J Type**

Opcode	Target Jump Address	0
4 bits	8 bits	4 bits

ALU operation	OpCode
add	000
sub	001
and	010
or	011
nor	100
sll	101
slr	110

ID Sequence	Instruction name	Type	Format	Op Code	Control Signal	Control in Hex
J	srl	Logic	S	0000	0101 0000 0110	0x506
A	add	Arithmetic	R	0001	1001 0000 0000	0x900
L	lw	Memory	I	0010	0111 1000 0000	0x780
M	sw	Memory	I	0011	0100 0100 0000	0x440
K	nor	Logic	R	0100	1001 0000 0100	0x904
D	subi	Arithmetic	I	0101	0101 0000 0001	0x501
F	andi	Logic	I	0110	0101 0000 0010	0x502
G	or	Logic	R	0111	1001 0000 0011	0x903
P	j	Memory	J	1000	0000 0000 1010	0x00A
I	sll	Logic	S	1001	0101 0000 0101	0x505
E	and	Logic	R	1010	1001 0000 0010	0x902
C	sub	Arithmetic	R	1011	1001 0000 0001	0x901
N	beq	Memory	I	1100	0000 0010 0001	0x021
B	addi	Arithmetic	I	1101	1010 000 0000	0x500
O	bneq	Memory	I	1110	0000 0001 0001	0x011
H	ori	Logic	I	1111	0101 0000 0011	0x503

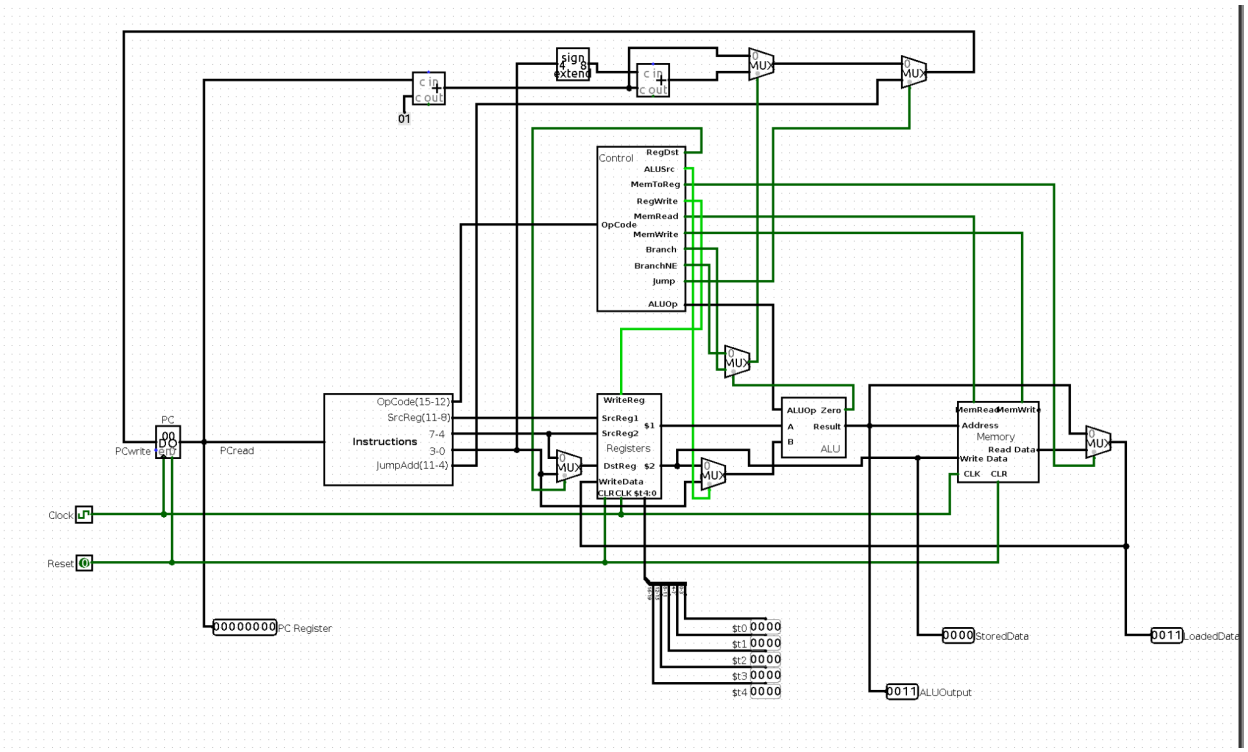


Figure 1: Circuit Diagram

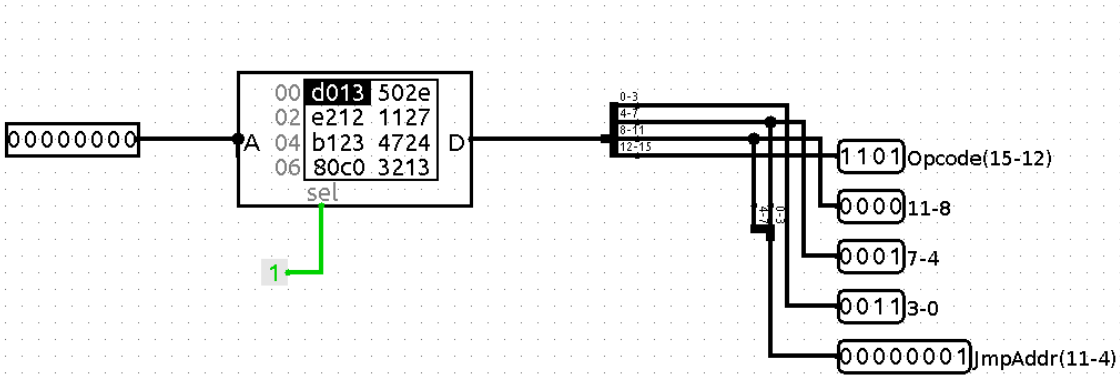


Figure 2: Instruction Memory

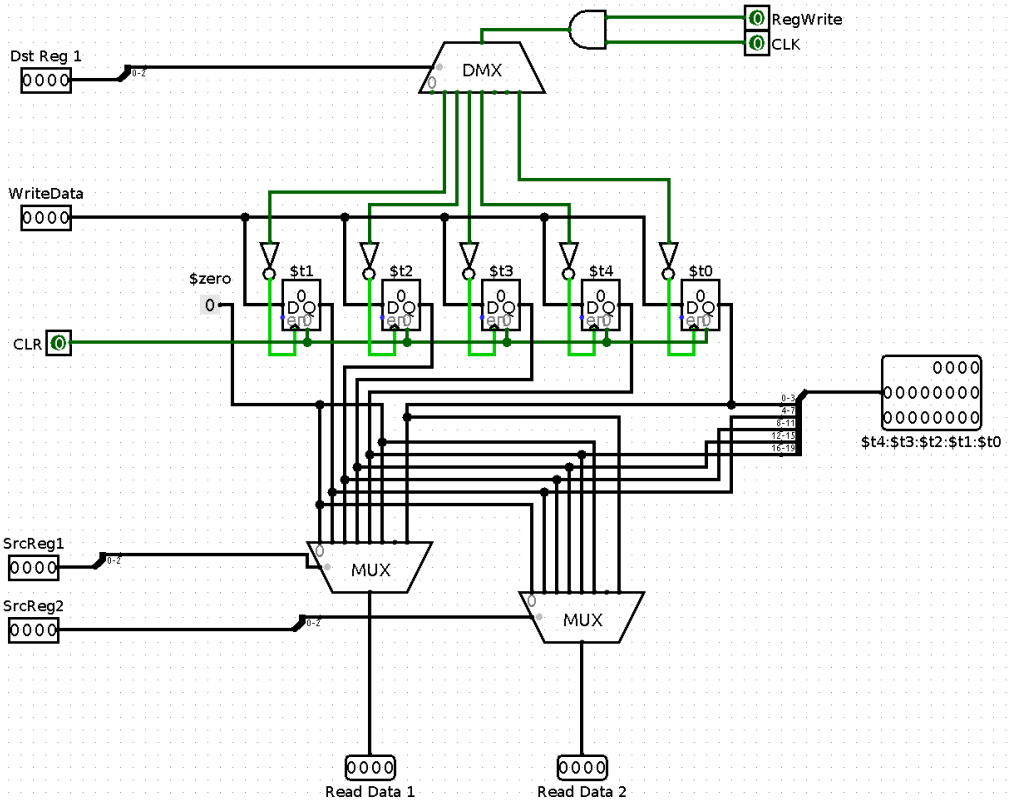


Figure 3: Register File

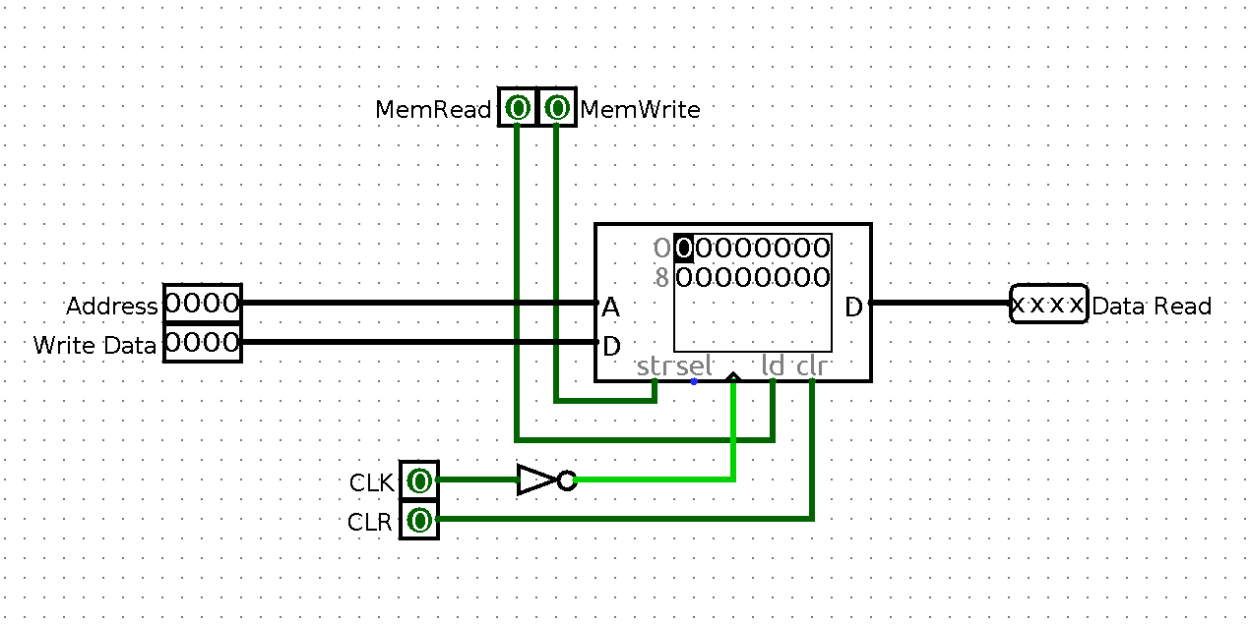


Figure 4: Data Memory

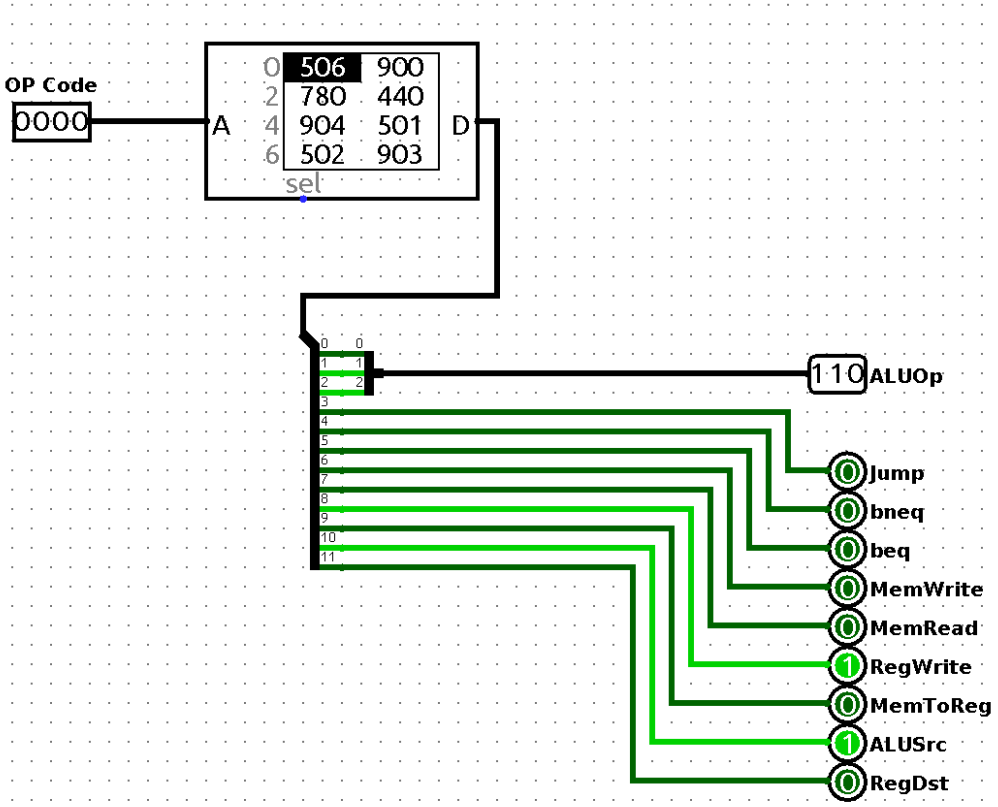


Figure 5: Control Unit

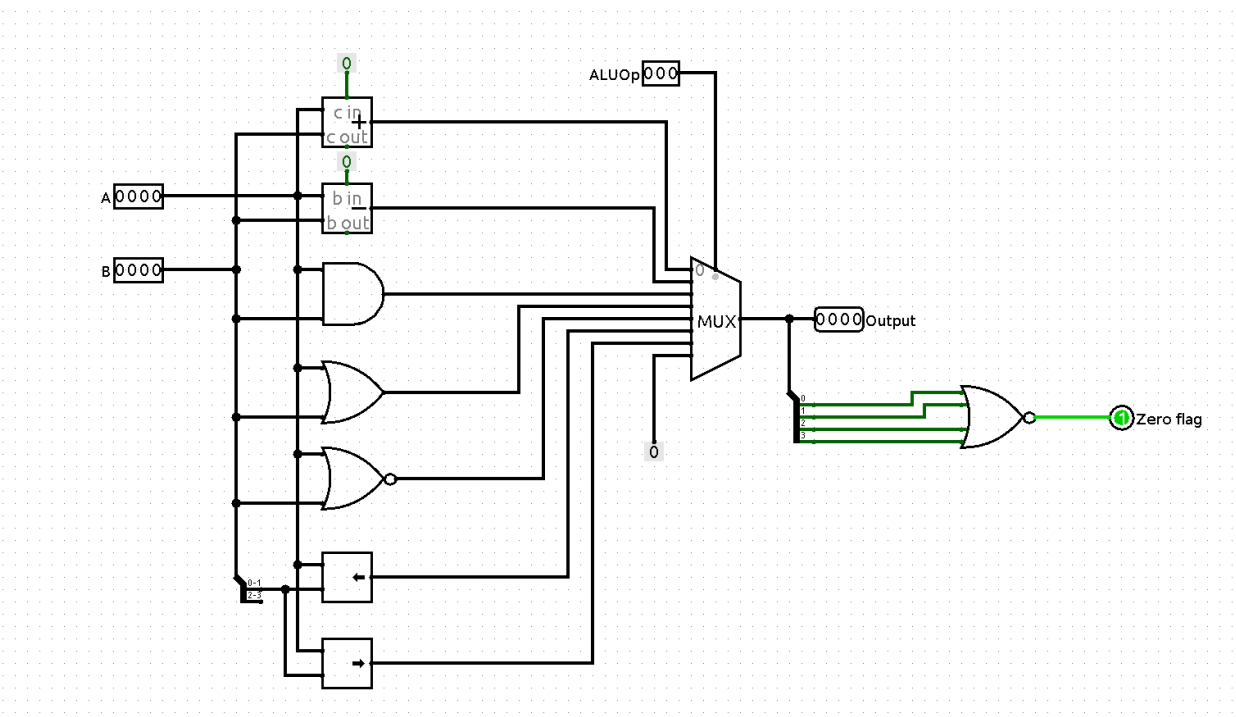
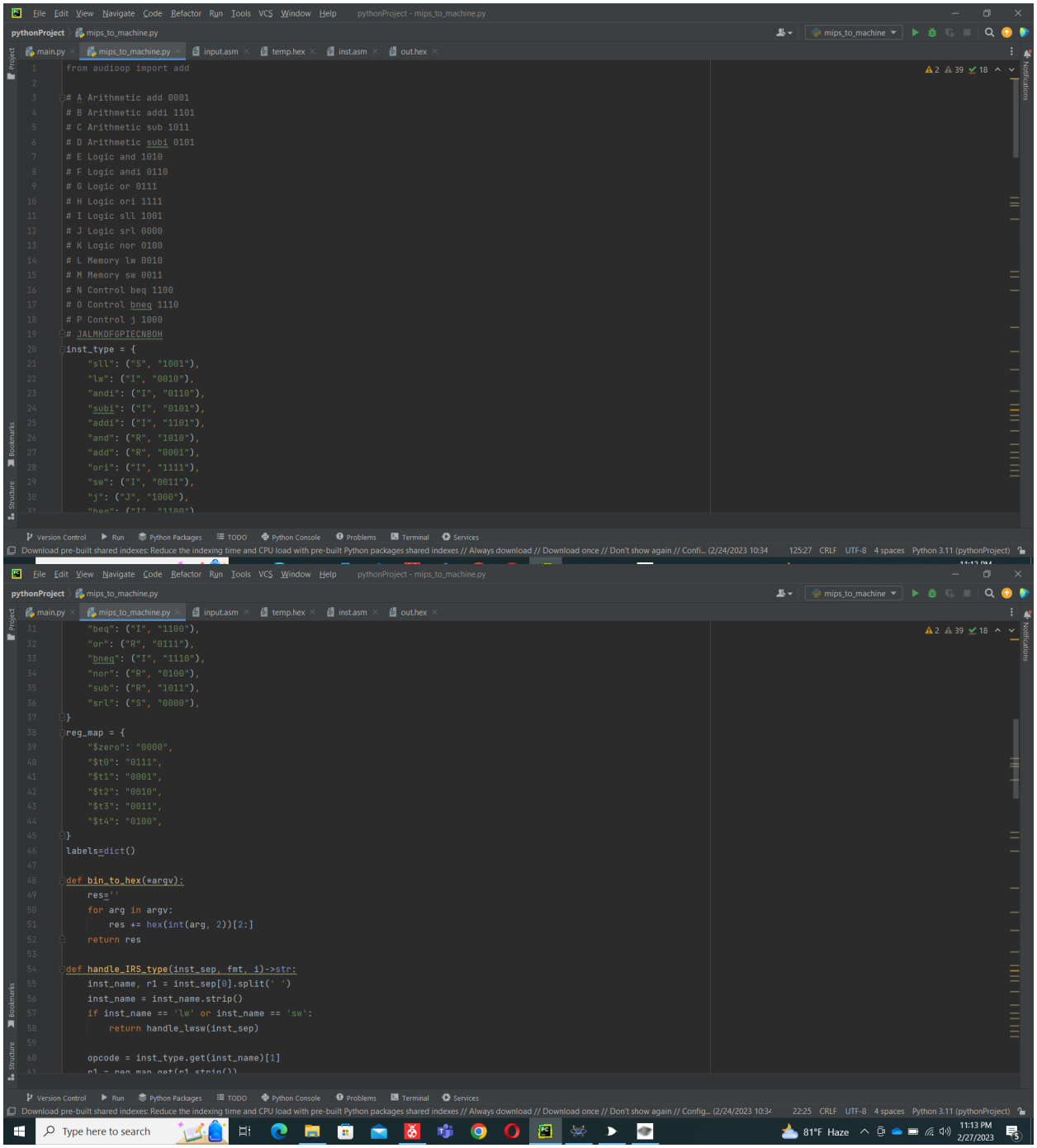


Figure 6: ALU

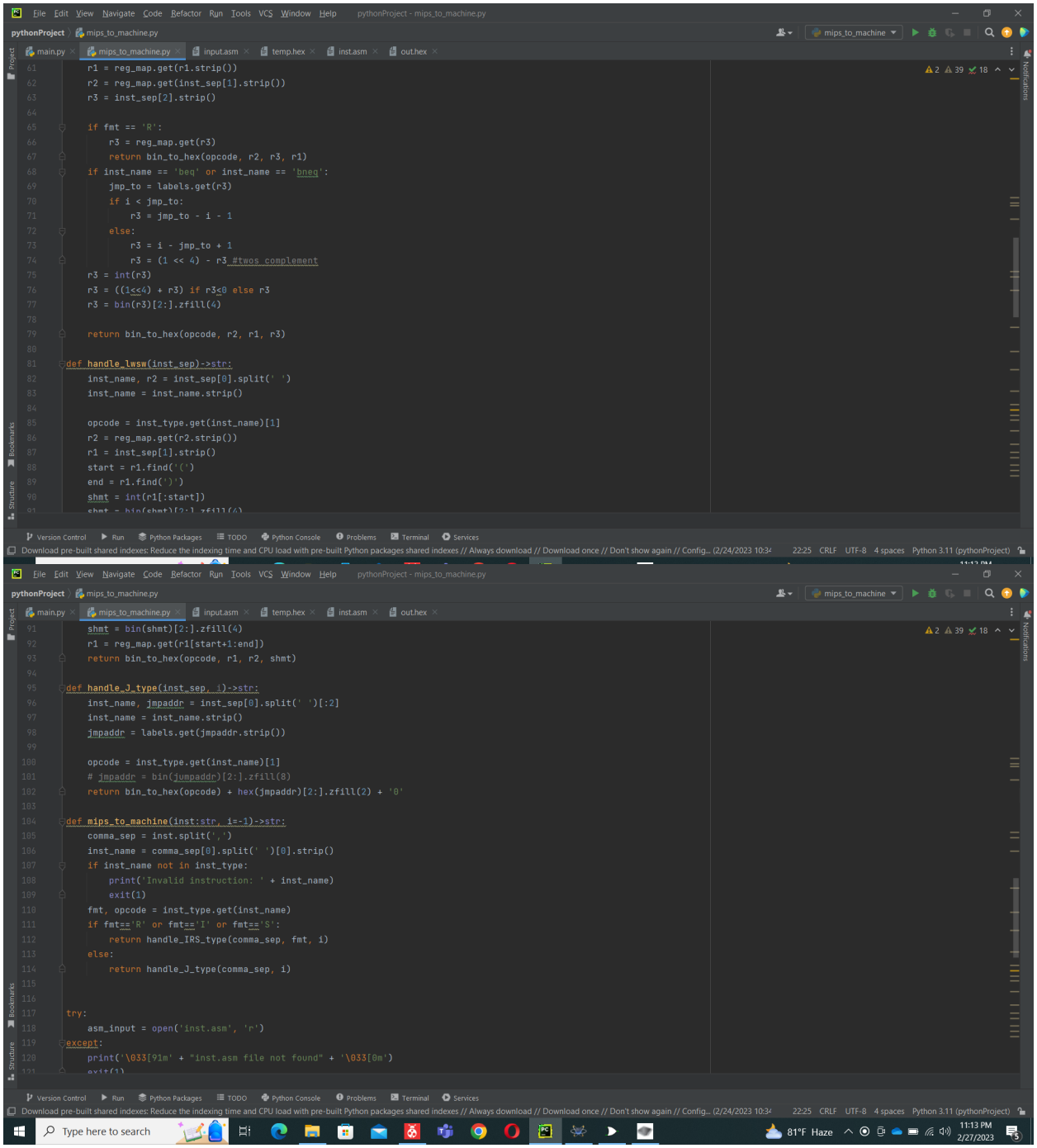
How to write and execute a program in this machine:

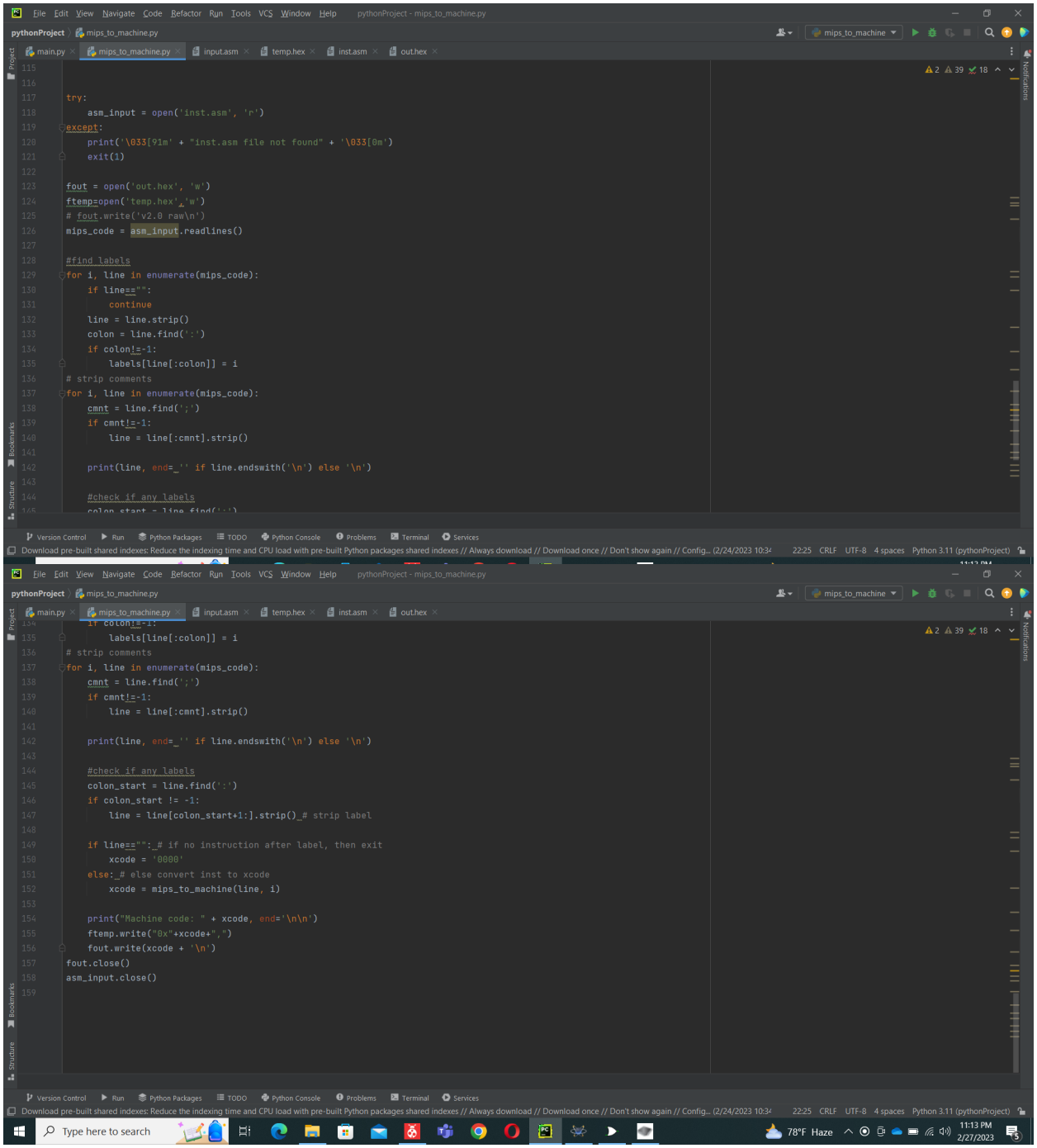
We write a program which takes the MIPS code and convert it to a file which contains the hexadecimal value of the instruction machine code. Then we need to upload it to the instruction ROM. For simulation at Logisim, we will upload the file at ROM. The program will start as soon as we give clock to the processor.

The file looks like:


File Edit View Navigate Code Refactor Run Tools VCS Window Help
pythonProject - mips_to_machine.py
pythonProject
mips_to_machine.py
mips_to_machine
main.py
mips_to_machine.py
input.asm
temp.hex
inst.asm
out.hex
from audioop import add
# A Arithmetic add 0001
# B Arithmetic addi 1101
# C Arithmetic sub 1011
# D Arithmetic subi 0101
# E Logic and 1010
# F Logic andi 0110
# G Logic or 0111
# H Logic ori 1111
# I Logic sll 1001
# J Logic srl 0000
# K Logic nor 0100
# L Memory lw 0010
# M Memory sw 0011
# N Control beq 1100
# O Control bneg 1110
# P Control j 1000
# JALMKDFGPIECNBOH
inst_type = {
"sll": ("S", "1001"),
"lw": ("I", "0010"),
"andi": ("I", "0110"),
"subi": ("I", "0101"),
"addi": ("I", "1101"),
"and": ("R", "1010"),
"add": ("R", "0001"),
"ori": ("I", "1111"),
"sw": ("I", "0011"),
"j": ("J", "1000"),
Version Control
Run
Python Packages
TODO
Python Console
Problems
Terminal
Services
Download pre-built shared indexes: Reduce the indexing time and CPU load with pre-built Python packages shared indexes // Always download // Download once // Don't show again // Confi... (2/24/2023 10:34
125:27
CRLF
UTF-8
4 spaces
Python 3.11 (pythonProject)

File Edit View Navigate Code Refactor Run Tools VCS Window Help
pythonProject - mips_to_machine.py
pythonProject
mips_to_machine.py
mips_to_machine
main.py
mips_to_machine.py
input.asm
temp.hex
inst.asm
out.hex
"beq": ("I", "1100"),
"or": ("R", "0111"),
"bneg": ("I", "1110"),
"nor": ("R", "0100"),
"sub": ("R", "1011"),
"srl": ("S", "0000"),
}
reg_map = {
"$zero": "0000",
"$t0": "0111",
"$t1": "0001",
"$t2": "0010",
"$t3": "0011",
"$t4": "0100",
}
labels=dict()
def bin_to_hex(*argv):
res=''
for arg in argv:
res += hex(int(arg, 2))[2:]
return res
def handle_IRS_type(inst_sep, fmt, i)->str:
inst_name, r1 = inst_sep[0].split(' ')
inst_name = inst_name.strip()
if inst_name == 'lw' or inst_name == 'sw':
return handle_lwsw(inst_sep)
opcode = inst_type.get(inst_name)[1]
Version Control
Run
Python Packages
TODO
Python Console
Problems
Terminal
Services
Download pre-built shared indexes: Reduce the indexing time and CPU load with pre-built Python packages shared indexes // Always download // Download once // Don't show again // Config... (2/24/2023 10:34
22:25
CRLF
UTF-8
4 spaces
Python 3.11 (pythonProject)
Type here to search
81°F Haze
11:13 PM
2/27/2023


File Edit View Navigate Code Refactor Run Tools VCS Window Help
pythonProject - mips_to_machine.py
pythonProject
mips_to_machine.py
mips_to_machine
main.py
mips_to_machine.py
input.asm
temp.hex
inst.asm
out.hex
r1 = reg_map.get(r1.strip())
r2 = reg_map.get(inst_sep[1].strip())
r3 = inst_sep[2].strip()
if fmt == 'R':
r3 = reg_map.get(r3)
return bin_to_hex(opcode, r2, r3, r1)
if inst_name == 'beq' or inst_name == 'bneq':
jmp_to = labels.get(r3)
if i < jmp_to:
r3 = jmp_to - i - 1
else:
r3 = i - jmp_to + 1
r3 = (1 << 4) - r3 #twos complement
r3 = int(r3)
r3 = ((1<<4) + r3) if r3<0 else r3
r3 = bin(r3)[2:].zfill(4)
return bin_to_hex(opcode, r2, r1, r3)
def handle_lwsw(inst_sep)->str:
inst_name, r2 = inst_sep[0].split(' ')
inst_name = inst_name.strip()
opcode = inst_type.get(inst_name)[1]
r2 = reg_map.get(r2.strip())
r1 = inst_sep[1].strip()
start = r1.find('(')
end = r1.find(')')
shmt = int(r1[:start])
2 39 18
Version Control
Run
Python Packages
TODO
Python Console
Problems
Terminal
Services
Download pre-built shared indexes: Reduce the indexing time and CPU load with pre-built Python packages shared indexes // Always download // Download once // Don't show again // Config... (2/24/2023 10:34
22:25 CRLF UTF-8 4 spaces Python 3.11 (pythonProject)
File Edit View Navigate Code Refactor Run Tools VCS Window Help
pythonProject - mips_to_machine.py
pythonProject
mips_to_machine.py
mips_to_machine
main.py
mips_to_machine.py
input.asm
temp.hex
inst.asm
out.hex
shmt = bin(shmt)[2:].zfill(4)
r1 = reg_map.get(r1[start+1:end])
return bin_to_hex(opcode, r1, r2, shmt)
def handle_J_type(inst_sep, i)->str:
inst_name, jmpaddr = inst_sep[0].split(' ')[:2]
inst_name = inst_name.strip()
jmpaddr = labels.get(jmpaddr.strip())
opcode = inst_type.get(inst_name)[1]
# jmpaddr = bin(jumpaddr)[2:].zfill(8)
return bin_to_hex(opcode) + hex(jmpaddr)[2:].zfill(2) + '0'
def mips_to_machine(inst:str, i=-1)->str:
comma_sep = inst.split(',')
inst_name = comma_sep[0].split(' ')[0].strip()
if inst_name not in inst_type:
print('Invalid instruction: ' + inst_name)
exit(1)
fmt, opcode = inst_type.get(inst_name)
if fmt=='R' or fmt=='I' or fmt=='S':
return handle_IRS_type(comma_sep, fmt, i)
else:
return handle_J_type(comma_sep, i)
try:
asm_input = open('inst.asm', 'r')
except:
print('\033[91m' + "inst.asm file not found" + '\033[0m')
2 39 18
Version Control
Run
Python Packages
TODO
Python Console
Problems
Terminal
Services
Download pre-built shared indexes: Reduce the indexing time and CPU load with pre-built Python packages shared indexes // Always download // Download once // Don't show again // Config... (2/24/2023 10:34
22:25 CRLF UTF-8 4 spaces Python 3.11 (pythonProject)
Type here to search
81°F Haze
11:13 PM
2/27/2023



sub, $t3, $t1, $t2 → 1011 0001 0010 0011 → B123

For hardware implementation, the hex file has to be loaded to an AVRAtmega32, which we used as instruction memory. In both hardware and software, one clock pulse will execute one instruction. In software simulation, the changes made by an instruction to both registers and memory are clearly visible. In hardware implementation, only the changes made to registers are visible by LEDs.

How we mapped registers:

$zero : 0000
$t0 : 0111
$t1 : 0001
$t2 : 0010
$t3 : 0011
$t4 : 0100

ICs

Components	Details	Number
IC 74273	Not Gate	1
IC 7483	4-bit Adder	4
IC 74157	Quad 2-to-1 MUX	9
IC 74273	8-bit D Flip-Flop	1
ATMEGA32	Microcontroller	5

Discussion:

In this assignment, we designed an 4-bit processor that implements the MIPS instruction set. For this purpose, we used basic gates (AND, OR, NOT, NOR), universal gates (XOR, XNOR), some other necessary gates (MUX, Shifter, Bit Finder, Adder, Subtractor, Bit Extender), RAM, ROM and register.
The Processor takes a 16-bit binary number (Instruction) and the circuit reads/stores the register values and memory values. Generally Program Counter is incremented by 4. But in our assignment we incremented the PC by 1 since the entire instruction resides in a single address. This is due to the software where we can only read content of a single address at a time.
Usually Control unit is implemented using combinational circuits. However, we

implemented the Control unit using ROM which significantly simplified the entire circuit.We used the known technique of designing the processor for MIPS instruction set. Here all types of instructions are executed in a single clock pulse. Moreover, Control ROM made our mapping quite easier, otherwise it would have been tedious by Introducing combinational logic or elsehow.
We also found it necessary to handle branch instruction whose offset is greater than 7 or less than -8. To handle such situation, the branch instruction has to be changed and extra jump and label have to be added. This in turn may change the offset of some labels appearing later. We think a multi-pass system is relatively simple way to handle this case.
We designed a converter in python and loaded the produced code in another ROM (Instruction Memory).We could not use the single memory (i.e., using Data Memory as Instruction Memory) because we cannot include 2 different addresses (or 2 different data read at a time) in a single memory unit while working in a single clock cycle method.We followed the provided circuit diagram and implemented the circuit.(attached below). While designing, we put emphasis on simplifying the circuit. We asserted our design bytesting various inputs and matching the corresponding outputs. However, testing the outputs, we successfully finished our simulation.

Datapath With Jumps Added