CSE 210

Computer Architecture Sessional

Project: MIPS Design and Simulation

Section - A2

Group - 01

Group Members:

1. 2105032 - Nawriz Ahmed Turjo
2. 2105047 - Himadri Gobinda Biswas
3. 2105048 - Shams Hossain Simanto

Date of Submission: 6 December 2024

1 Introduction

A processor or processing unit is a digital circuit which performs operations on some external data source, usually memory or some other data stream. The term is frequently used to refer to the Central Processing Unit (CPU) in a system. A central processing unit is the electronic circuitry that executes instructions comprising a computer program.The CPU performs basic arithmetic, logic, controlling, and input/output (I/O) operations specified by the instructions in the program.

Principal components of a CPU include the arithmetic logic unit (ALU) that performs arithmetic and logic operations, processor registers that supply operands to the ALU and store the results of ALU operations, and a control unit that orchestrates the fetching (from memory) and execution of instructions by directing the coordinated operations of the ALU, registers and other components.

There are many types of Processor Design Implementation. MIPS(Microprocessor without Interlocked Pipelined Stages) is a reduced instruction set computer(RISC) instruction set architecture(ISA). The Processor design implementing MIPS ISA is called MIPS processor.

In this assignment, we have designed an 8-bit processor that implements the MIPS ISA. Each instruction will take 1 clock cycle to be executed. We have designed instruction memory, data memory, register file, ALU, and a control unit of the Processor.

The Processor is composed of five components:

1. **Program Counter :** The program counter (PC) is a 8-bit Register which activates at the falling edge of the clock signal. After every clock it adds 1 to its previous address. The value it stores is used as the Instruction Memory Address.
2. **Register File :** Register File is a bank of seven Registers. They are denoted as \$zero, \$t0, \$t1, \$t2, \$t3, \$t4, \$t5, \$sp. \$sp register is the stack pointer which has initial value ffH. \$zero register stores 00H. Other Registers are used as General Purpose Registers.
3. **ALU :** The ALU does all the calculations. It is controlled by 3-bit ALUop code which sets the type of calculation it performs. It performs calculations with two 8-bit binary numbers.
4. **Control Unit :** The Control Unit decodes the instruction by giving the selection input to all the MUXs, Register File, Data Memory and ALU.
5. **Data Memory :** The Data Memory stores the Stack values and works as main memory. It has 256 bytes storage capacity. It stores Data as 8-bit value.

2 Instruction Set

Instruction ID	Op code	Category	Type	Instruction
A	0101	Arithmetic	R	add
B	1100	Arithmetic	I	addi
C	0001	Arithmetic	R	sub
D	0011	Arithmetic	I	subi
E	1011	Logic	R	and
F	1000	Logic	I	andi
G	1111	Logic	R	or
H	1101	Logic	I	ori
I	1010	Logic	R	sll
J	1110	Logic	R	srl
K	0000	Logic	R	nor
L	0110	Memory	I	sw
M	0111	Memory	I	lw
N	1001	Control-conditional	I	beq
O	0100	Control-conditional	I	bneq
P	0010	Control-unconditional	J	j

Table 1: Instruction Set

3 Complete Circuit Diagram (of all components)

3.1 Main circuit

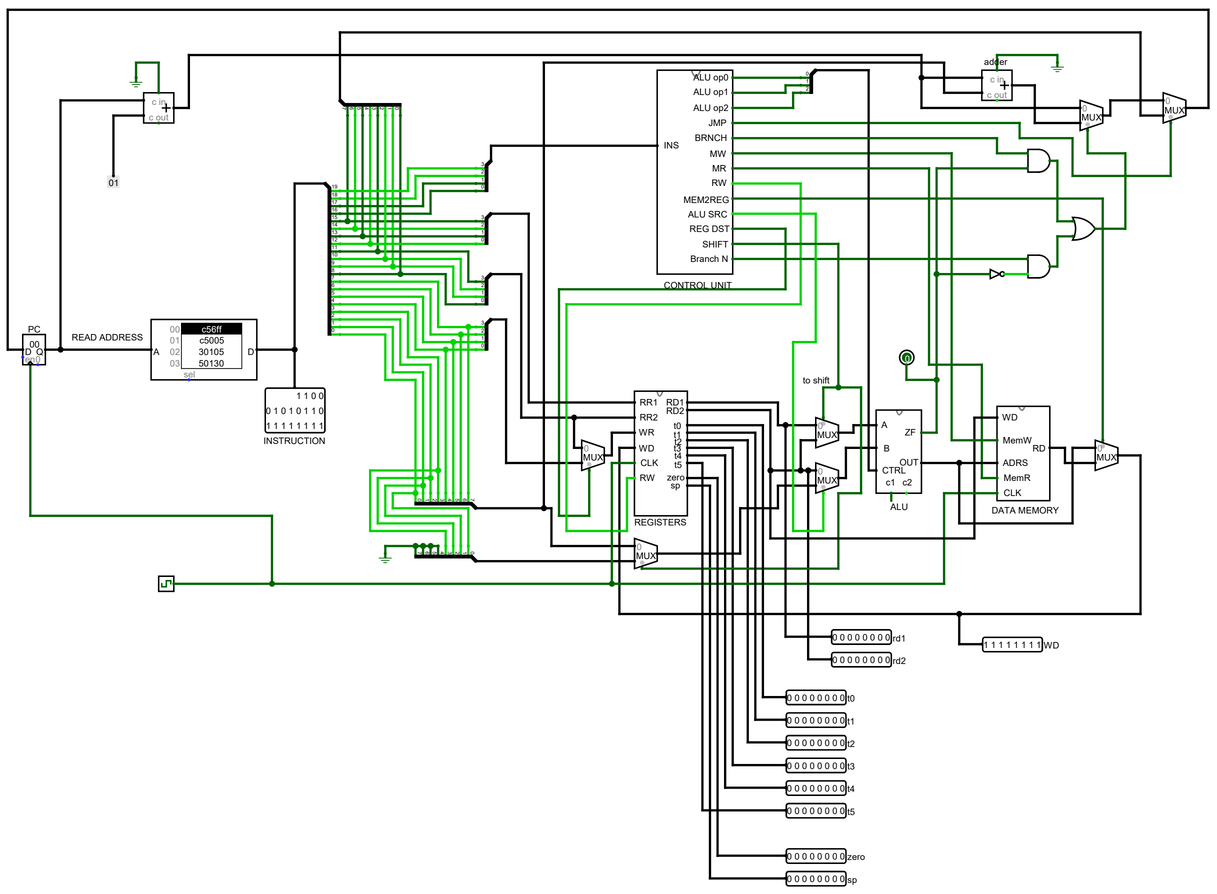

Figure 1: Complete Circuit Diagram

3.2 Data Memory

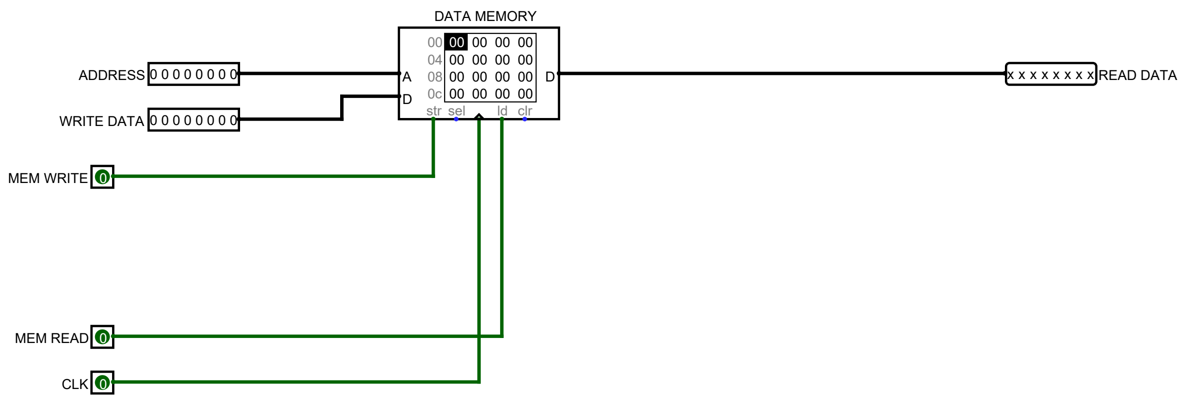

Figure 2: Data Memory

3.3 Instruction Memory

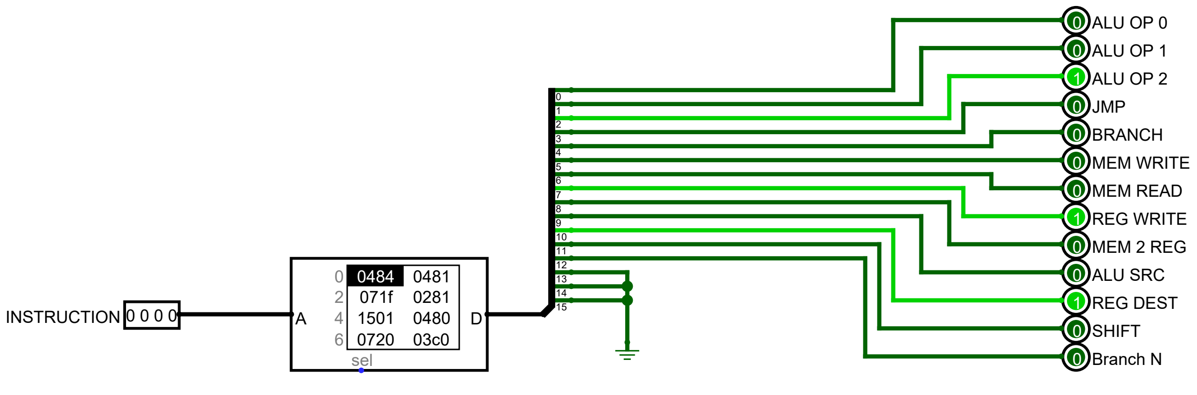

Figure 3: Instruction Memory

3.4 ALU

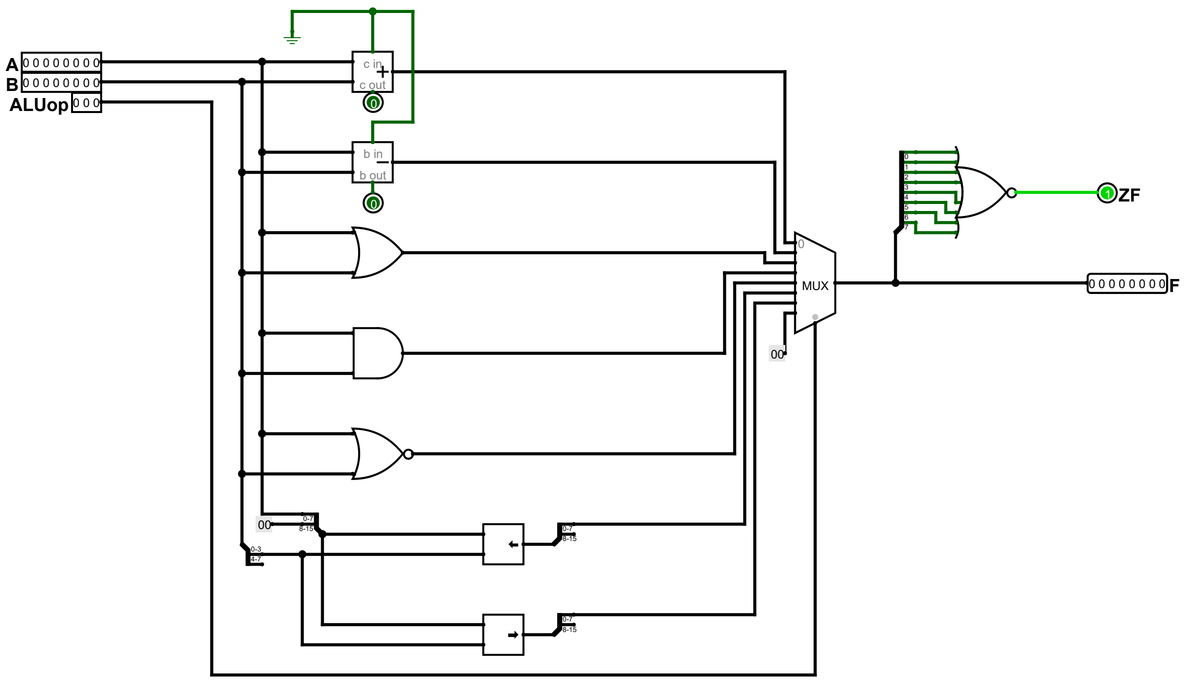

Figure 4: Custom ALU

3.5 Register

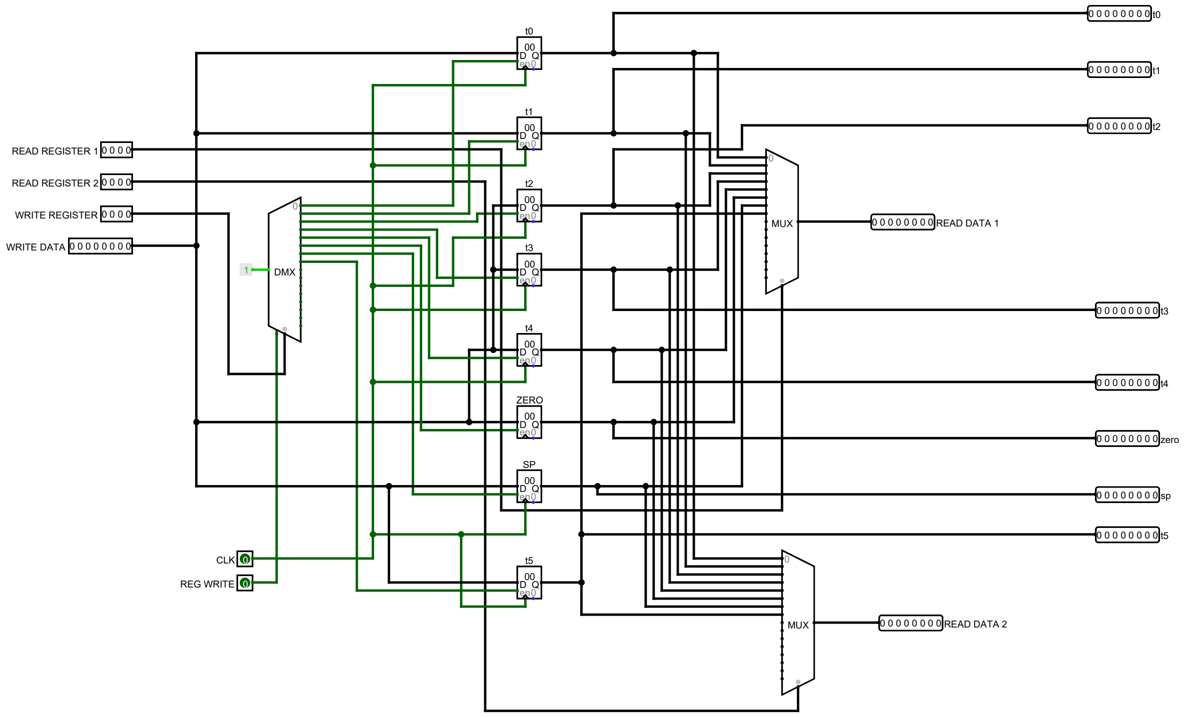

Figure 5: Register

4 How to Write and Execute a Program

To write and execute a program, follow these steps:

1. Prepare the MIPS instructions and save them in a text file named `MIPS_INPUT.txt`.
2. Use the following command to load the instructions into the instruction memory:

```
python RomEditor_HEX_auto.py
```

3. The `RomEditor_HEX_auto.py` script first loads the `INSTRUCTION_CONVERTOR.exe` file, which is generated from the `INSTRUCTION_CONVERTOR.cpp` code. Ensure that these files are in the same folder as the circuit.
4. The script saves the hex information obtained from the converter into the `hexdata.hex` file.
5. Finally, the Python script places this hex information into the circuit's instruction part, ensuring the program is ready for execution.

This process ensures that the MIPS instructions are loaded correctly and ready for execution in the instruction memory.

5 ICs used with their count

IC category	**IC Number**	**Count**
ROM		2
RAM		1
MUX	74150	2
DeMUX	74154	1
Registers		9
MUX	74157	7
NOT	7404	1
AND	7408	2
OR	7432	1
Adder	7483	2
Total		28

Table 2: IC Count

6 Discussion

Prior to designing the circuitry, it was a great challenge for the team to accurately determine the opcodes of each instructions assigned to us and coding for it. We agreed to first write the `INSTRUCTION_CONVERTOR.cpp` file at least to an extent so that it can convert given MIPS instructions to hexadecimal codes that could be loaded in the instruction memory(ROM). This decision turned out to be fruitful as we did not have to generate the hexadecimal codes manually again and again while testing the circuit logic of various instructions. Rather it gave us a sense of motivation while working for each instruction and control bits' circuitry that our implemented code is helping us to accurately determine and test the codes for control unit ROM.

Then we started to design different modules parallelly and later integrated them in the complete circuit. On the way, we faced a number of hurdles. First time using RAM, ROM and registers in Logisim created a lot of confusions but after repeated thinking and testing, we were able to use them efficiently. One great impediment came when the immediate arithmetic operations were not reflecting the results in the destination registers. After careful observation, we caught the bug that it was a blunder in formatting of those instructions i.e. which register should be written prior and which later. Another tough hurdle was actually seeing the branching and jumping happen in the instruction memory. After meticulous debugging in different modules and parts of the circuit, we could successfully see it occur before our eyes.

For the circuit design in Logisim, we whole-heartedly followed the MIPS datapath design of the book "Computer Organization and Design" by David A. Patterson and John L. Hennessy (5th edition). We positioned and labeled the wires just as in the design of the book, so that we could maintain a good correlation with our theoretical knowledge. We crafted our own 8-bit ALU implementation to have a better grasp on the opcode handling. We have used an extra control bit "BRANCH_N" in the CONTROL UNIT circuit. This handles the instruction on BNEQ. 1 is added to go to the next instruction whereas in the book, 4 is added. We have implemented SLL and SRL which increased the number of MUX in our circuit by 2. To implement conditional BNEQ we used extra AND gate, OR gate and NOT gate.

The assignment significantly helped us bridge through the chasms of our knowledge of Computer Architecture enabling us to visualize the entirety of the whole workflow from writing a piece of code in a fancy editor to carrying out bit-level instructions in real hardware. Overall, understanding, designing, and implementing the 8 bit MIPS processor was a thrilling task for all of us.

7 Contribution of Each Member

Design

- ✐ Instruction Memory - 2105032
- ✐ Data Memory - 2105048
- ✐ Control Unit - 2105047
- ✐ ALU - 2105032, 2105048
- ✐ Registers - 2105047, 2105048
- ✐ Optimiztion - 2105032
- ✐ Whole Circuit Integration - 2105032, 2105047, 2105048
- ✐ Circuit Debugging - 2105032, 2105047
- ✐ Circuit Testing - 2105047, 2105048
- ✐ cpp code for assigning instuctions and registers - 2105047
- ✐ cpp codes for accurately generating instructions- 2105048
- ✐ python code for loading instructions in IM- 2105032
- ✐ Code debugging- 2105032
- ✐ Code testing with custom test cases - 2105032, 2105047, 2105048

Report

- ➥ Section: 1, 2, 3 - 2105047
- ➥ Section: 3, 4, 5 - 2105032
- ➥ Section: 5, 6, 7 - 2105048