**BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY**

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CSE 306

COMPUTER ARCHITECTURE SESSIONAL

Assignment 3

4-bit MIPS Design, Simulation and Implementation

Section B2

Group 6

Group Members:

19050104

19050105

19050107

19050108

19050109

Date of Submission: 01/03/2023

Introduction:

MIPS is an Instruction Set Architecture (ISA) that follows the RISC (Reduced Instruction Set Computer) design principle. The instruction set of MIPS is predetermined and remains constant, which guarantees simplicity, consistency and uniformity.

Although the instruction set remains constant, it has different instruction formats for different types of instructions.

R-Format: Deals with arithmetic operations with registers.

I-Format: Deals with immediate and data transfer operations.

J-Format: Supports long jump to a remote address.

Here is an example of add instruction:

|  |  |  |
| --- | --- | --- |
| Operation | Instruction | Action |
| ADDITION | add $s0, $s1, $s2 | $s0 = $s1 + $s2 |

Here, $s0, $s1, $s2 are registers that hold values.

To evaluate an expression f = (g + h) – (i+ j), we would do the following:

add $t0, $s1, $s2 # $t0 = g + h

add $t1, $s3, $s4 # $t1 = i + j

sub $s0, $t0, $t1 # f = (g + h) – (i+ j)

Datapath is a fundamental component of a CPU that comprises of registers, ALUs (Arithmetic Logic Units), MUXs (Multiplexers), memories, and control elements. It is designed to process data and addresses as per the requirements of a program. In the case of MIPS, the instructions are routed through the datapath to execute different operations such as addition, load/store, branching, and jumping.

Instruction Sets:

The objective of our task is to implement a customized version of the MIPS instruction set with certain modifications and limitations. Our version, named 4-bit PC, includes an 8-bit address bus, a 4-bit data bus, and a 4-bit ALU. Moreover, we need to implement various temporary registers, namely $zero, $t0, $t1, $t2, $t3, and $t4.

Instruction sets for MIPS:

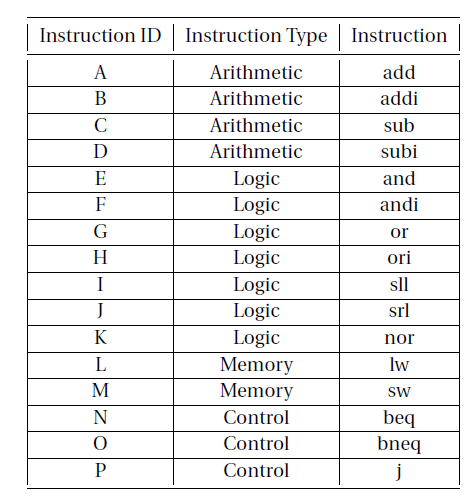


Figure 1: Instruction Set

Our MIPS instructions consist of 16 bits and are structured into four different formats:

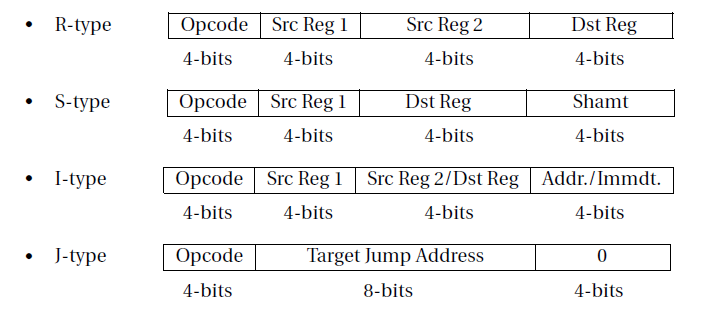


Figure 2: MIPS Instruction Format

The opcodes of our instructions are limited to 4 bits, which means they can range from 0 to 15. Our instruction set is defined by the assignments KGAMDCFBHNOJPELI. Based on these assignments, we can construct our instruction set accordingly

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Order | Instruction | Type | Format | Op code |
| K | Nor | Logic | R | 0 |
| G | Or | Logic | R | 1 |
| A | Add | Arithmetic | R | 2 |
| M | Sw | Memory | I | 3 |
| D | subi | Arithmetic | I | 4 |
| C | sub | Arithmetic | R | 5 |
| F | andi | Logic | I | 6 |
| B | Addi | Arithmetic | I | 7 |
| H | Ori | Logic | I | 8 |
| N | beq | Control | I | 9 |
| O | bneq | Control | I | 10 |
| J | srl | Logic | S | 11 |
| P | J | Control | J | 12 |
| E | And | Logic | R | 13 |
| L | Lw | Memory | I | 14 |
| I | Sll | Logic | S | 15 |

Table 1: Instructions for each opcode

Circuit Diagram:

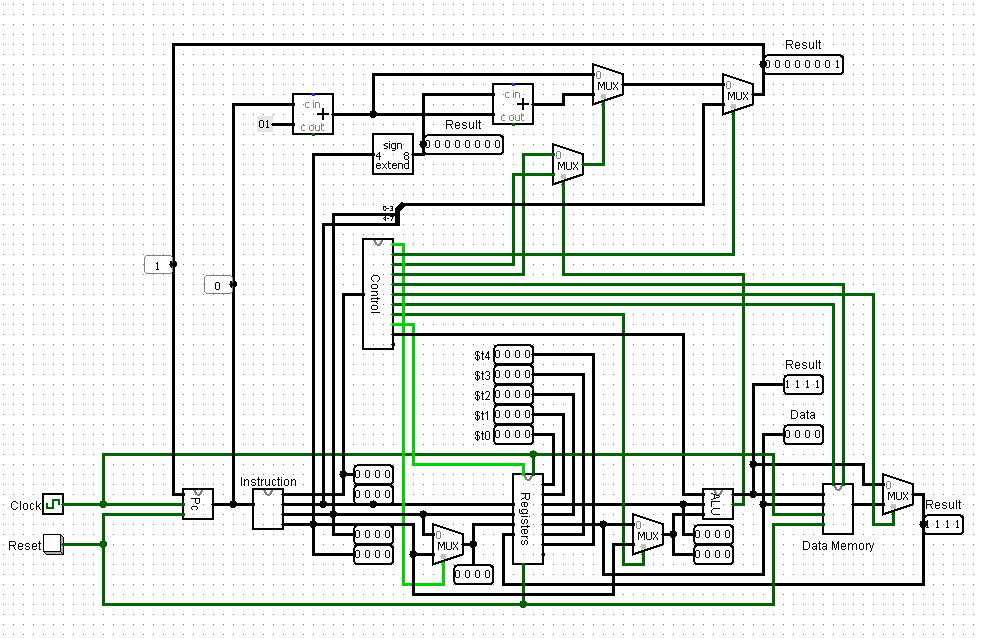


Figure 3: 4-bit PC

Detailed Design Steps:

The MIPS architecture is a complex system consisting of multiple essential components that work together to provide efficient and reliable processing. Each component plays a critical role in the functioning of the MIPS architecture and contributes to its overall performance. These component include:

1. **PC:**

This component was used for the address of instruction memory. We have used a D-Flipflop to implement this component.

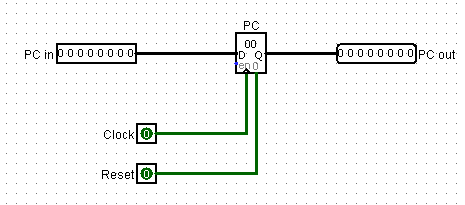
****

Figure 4: PC (Program Counter)

1. **Instruction Memory:**

To store the instructions in our 4-bit PC, we used an EPROM as instruction memory. The input instructions were converted to 16 bit MIPS instruction code. These instructions were then split into four 4-bit values, where the most significant 4 bits represented the Opcode that indicated the instruction type, the 2nd most significant 4 bit represented the address of the first source register, the next 4 bits represented the second source register or destination register and the least significant 4 bits represented the destination register or immediate value. The instruction memory outputted an instruction depending on the PC (Program Counter) value.

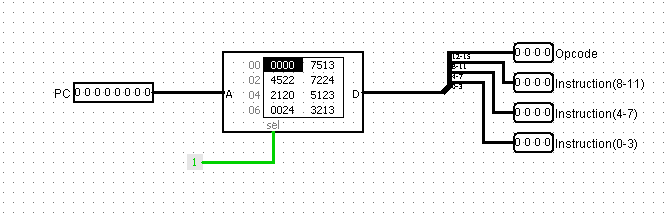


Figure 5: Instruction Memory

1. **Register:**

Then we designed a register bank consisting of seven registers, which includes five temporary registers and a zero register for handling arithmetic operations with zero values. For reading from a register, the register address was taken as input. For writing into a register, corresponding write address and data value was taken as inputs. To handle this, a selection bit called RegWrite was also taken as input.

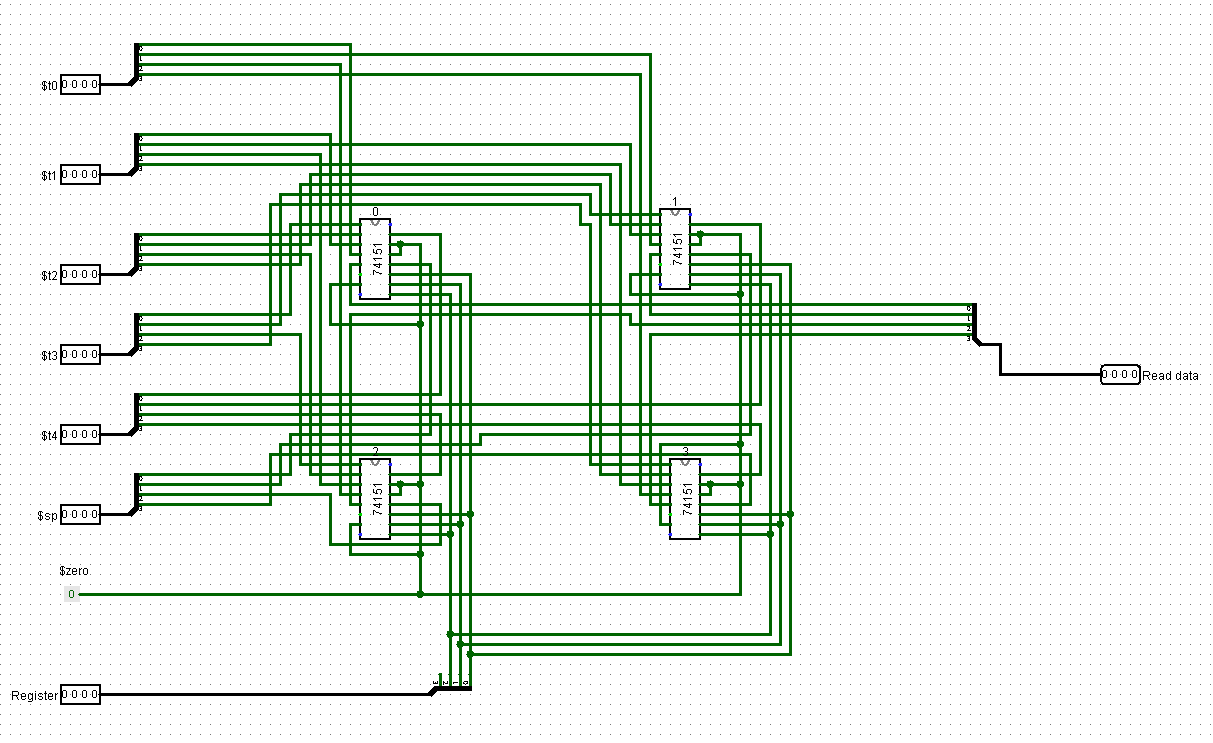


Figure 6: 4 input 8x1 Mux

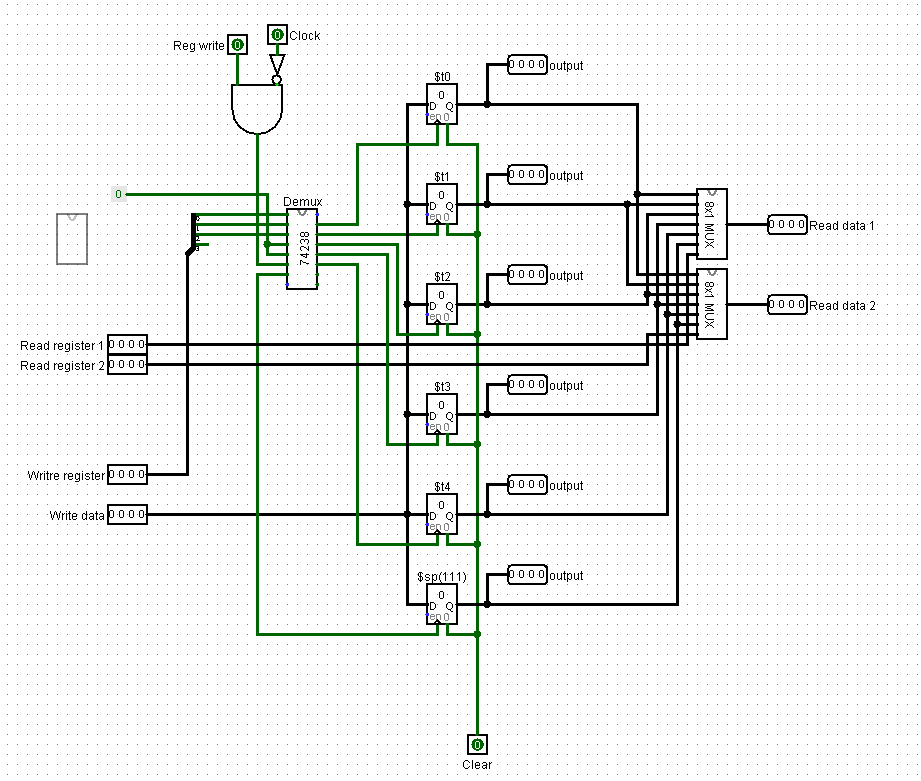
****

Figure 7: Register

1. **ALU:**

Then we designed 4-bit ALU for handling arithmetic and logical operations. The ALU takes two input values and a control bit as ALUOP and generates a 4-bit output value. It also generates a zero flag to indicate whether the output value is zero or not.

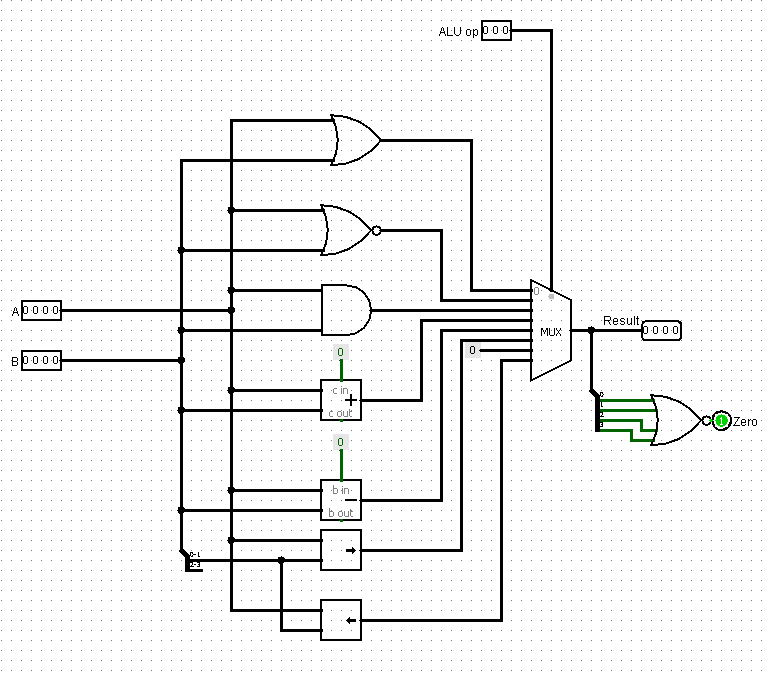


Figure 8: ALU

1. **Data Memory:**

We used a RAM to implement the Data Memory. For load operation, it takes memory address and outputs data and for store operation it takes memory address and data as input to store data in the memory. To select the read or write operation, two selector bits and a clock was used.

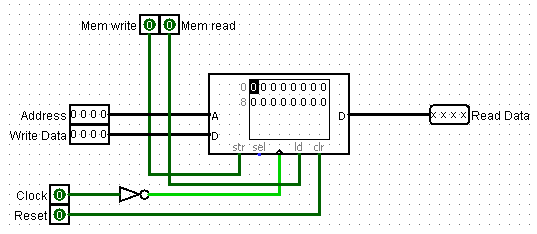


Figure 9: Data Memory

1. **Control:**

The control unit determines which operation will take place and which component will be used based on the provided instruction set. A ROM was used to implement it. It stores the hexadecimal values for the different operations. It takes 4-bit OpCode and selects corresponding operation(s) and component(s).

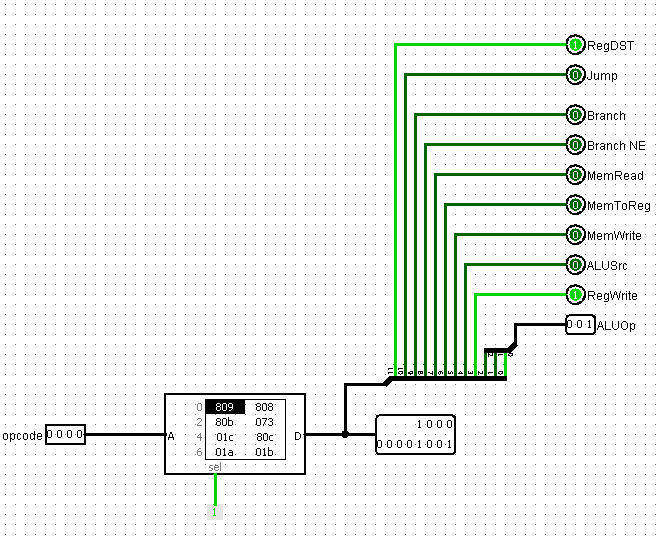


Figure 10: Control Unit

1. **Other:**

4 Adder were used to increment pc (for pc+1 and for pc+1+branch address) and some mux were used to select the correct data or address.

**How to write and execute a program:**

We implemented our circuit at both software and hardware levels. At the software level, we have to give hex values generated from the Assembly program by the converter to simulate the circuit. At the hardware level, we had written code for ATMEGA32 for each part. Then we had burnt the ATMEGA32 with the code. For executing a program we have to give hex values of instructions in the instruction memory. Then we have to give power and by using clock we have to increase the value of the PC. Thus, we can execute a program.

**Integrated Circuits:**

|  |  |  |
| --- | --- | --- |
| **Component** | **Details** | **Amount** |
| **ATMEGA32** | **Microcontroller** | **5** |
| **IC 74157** | **Quad 2-to-1 MUX** | **8** |
| **IC 7483** | **4-bit ADDER** | **4** |
| **IC 74273** | **8-bit D Flip-Flop** | **1** |

**Simulator:**

Software: Logisim

Version: Logisim-win-2.7.1

**Discussion:**

1. We designed and simulated a reduced version of the MIPS instruction set and tried to implement the hardware for the 4-bit PC.
2. The functionality of each component was ensured by testing individual ATmega components and checking output values against expected values for different input test cases.
3. Then the integrated circuit was tested incrementally.
4. At last PC was implemented using D-Flipflop and adder. It was expected that the whole circuit will execute. However, the final circuit was not functional. As other components were tested before, it might be the problem of the PC.