CSE 306

Computer Architecture Sessional

Assignment 3

4-bit MIPS Design, Simulation, and Implementation

Section B1

Group 4

Group Members :

1905065

1905067

1905069

1905076

1905077

Date of Submission : 29/02/2023

**Introduction:**

MIPS is a RISC (Reduced Instruction Set Computer) ISA (Instruction Set Architecture). Instructions of MIPS are fixed, thus ensuring regularity.

Here is an example of add instruction.

|  |  |  |
| --- | --- | --- |
| Operation | Instruction | Action |
| Addition | Add $t2, $t1, $t3 | $t2 = $t1 + $t3 |

Here, $t1, $t2, $t3 are registers that hold values. To evaluate an expression x = a + b – c, we would do the following.

add $t0, $t1, $t2 #x = a + b

add $t0, $t0, $t3 # x = x + c or x = a + b + c

According to MIPS instruction rules, arithmetic operations can only take registers as arguments, size of a register is 32 bits and there are 32 registers in total.

A datapath is built with registers, ALUs, MUXs, memories, and controls elements that can process data and addresses in the CPU. MIPS instructions are fed through a datapath to perform various instructions like addition, load/store, branching, or jump.

**Instruction Sets:**

For this assignment, we have been tasked with implementing a modified and reduced version of the MIPS instruction set. Our implementation will feature an 8-bit address bus and a 4-bit data bus, as well as a 4-bit ALU, hence the name 4-bit MIPS.

As part of our design, we need to include several temporary registers, including $zero, $t0, $t1, $t2, $t3, and $t4.

Instruction set for our MIPS is given below.

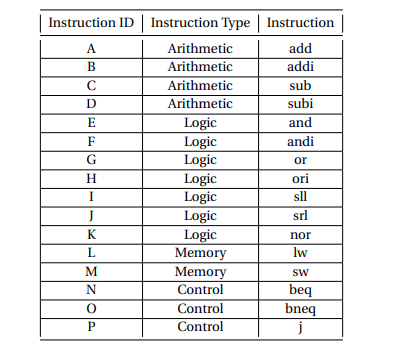
****

Figure 1: Instruction Set Description

Our MIPS instruction would be 16 bits long following these 4 formats.

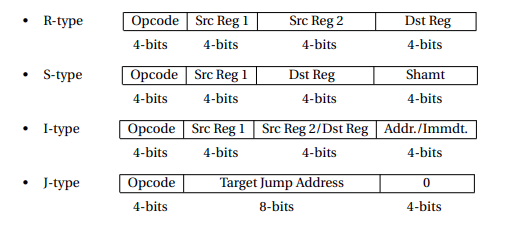


Figure 2: MIPS Instruction Format

Opcodes of the instructions are of 4 bits, so between 0 and 15. We’re given the Instruction assignment JMFBDLIGHPOENACK. So, our instruction set would be something like this.

|  |  |  |
| --- | --- | --- |
| Opcode | Instruction Type | Instruction |
| 0000 | Logic | srl |
| 0001 | Memory | sw |
| 0010 | Logic | andi |
| 0011 | Arithmatic | addi |
| 0100 | Arithmatic | subi |
| 0101 | Memory | lw |
| 0110 | Logic | sll |
| 0111 | Logic | or |
| 1000 | Logic | ori |
| 1001 | Control | j |
| 1010 | Control | bneq |
| 1011 | Logic | and |
| 1100 | Control | beq |
| 1101 | Arithmatic | add |
| 1110 | Arithmatic | sub |
| 1111 | Logic | nor |

Table 1: Instructions for each opcode

**Circuit Diagram:**

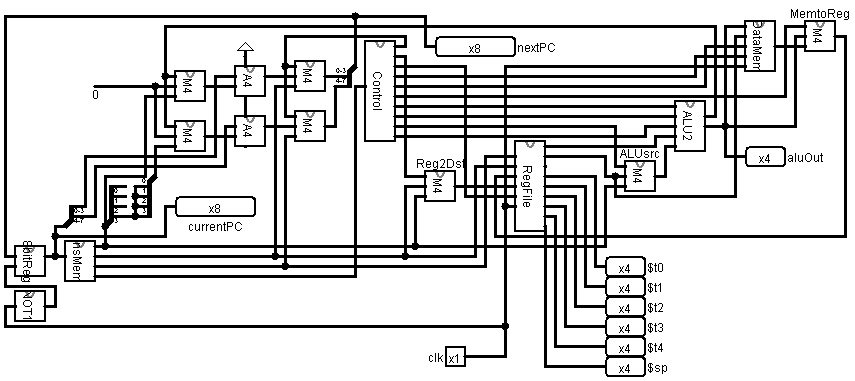


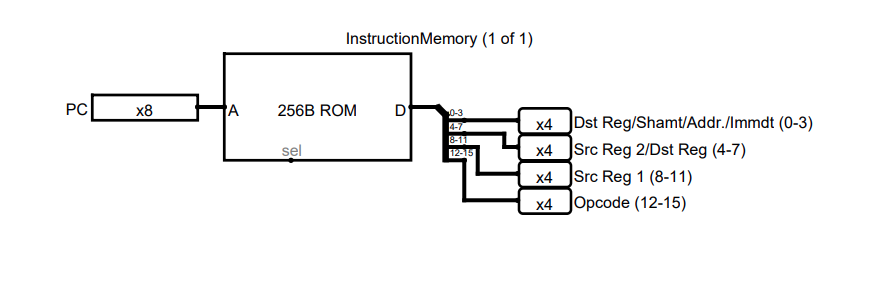
Figure 11: 4 bit PC

**Detailed Design Steps:**

In our MIPS architecture , the following components were used to build up the entire structure .

**1. Instruction Memory**:

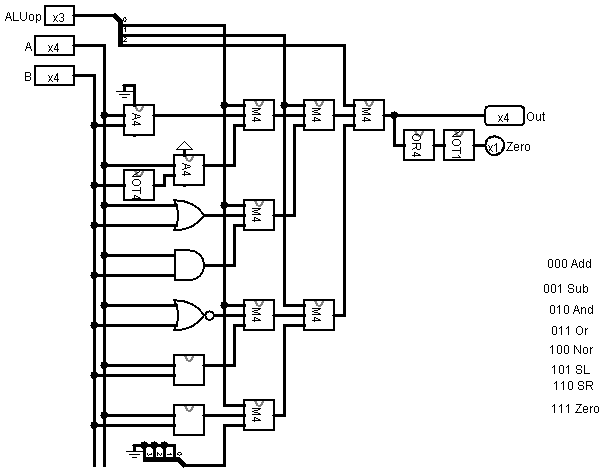
We have used an EPROM to store the instructions in our instruction memory. The instructions were provided to us as input and then converted to MIPS instruction code, resulting in 16-bit instructions. These instructions were divided into four 4-bit values: the first 4 bits represented the destination register or immediate value, the next 4 bits represented the second source register or destination register, the subsequent 4 bits represented the first source register, and the final 4 bits represented the Opcode that indicated the type of instruction. As the PC value incremented, the instruction memory outputted a new instruction.

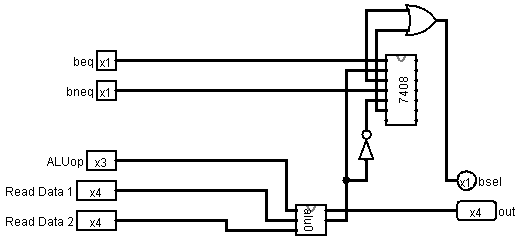


**2. ALU:**

Our next task is to design a 4-bit ALU to handle arithmetic and logical instructions. The ALU takes two input values and an ALUOP value, and produces a 4-bit output value, which is then used for further processing. Additionally, the ALU generates a zero flag that verifies whether the output is zero or not.

To use the ALU, our circuit takes the opcode along with two 4-bit input values. It also takes beq and bneq flags as inputs. The 4-bit ALU generates the final output value, while the zero flag, along with the beq and bneq flags (output from the control unit), produces a bsel flag that indicates whether we need to branch or not.

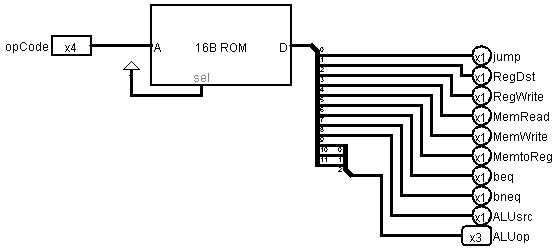




**3. Control Unit:**

In the control unit, we set the values for our MUX operations based on the provided instruction set for each group. To do this, we use a 16-bit ROM, where we store the hexadecimal values for the different operations.

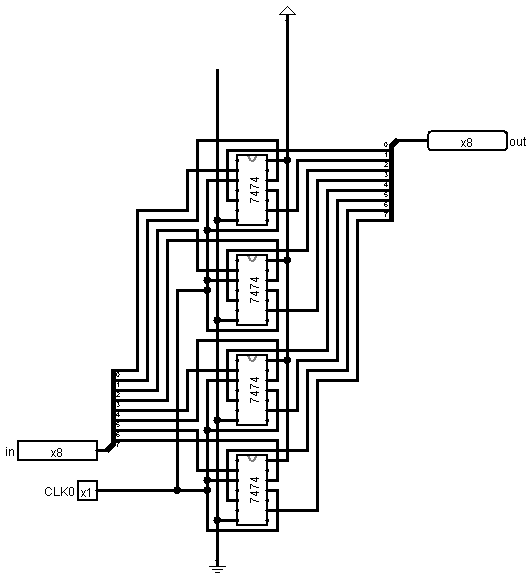
When we receive the 4-bit OpCode, we use it to select the corresponding operation from the ROM. The ROM outputs 9 selector bits for the operation, as well as a 3-bit ALUop that is used to control the operation of the 4-bit ALU.



**4. PC Register:**

To store the value of our PC at any given time, we use a PC register. This register consists of 4 D-flip flops, which are used to store the PC value.

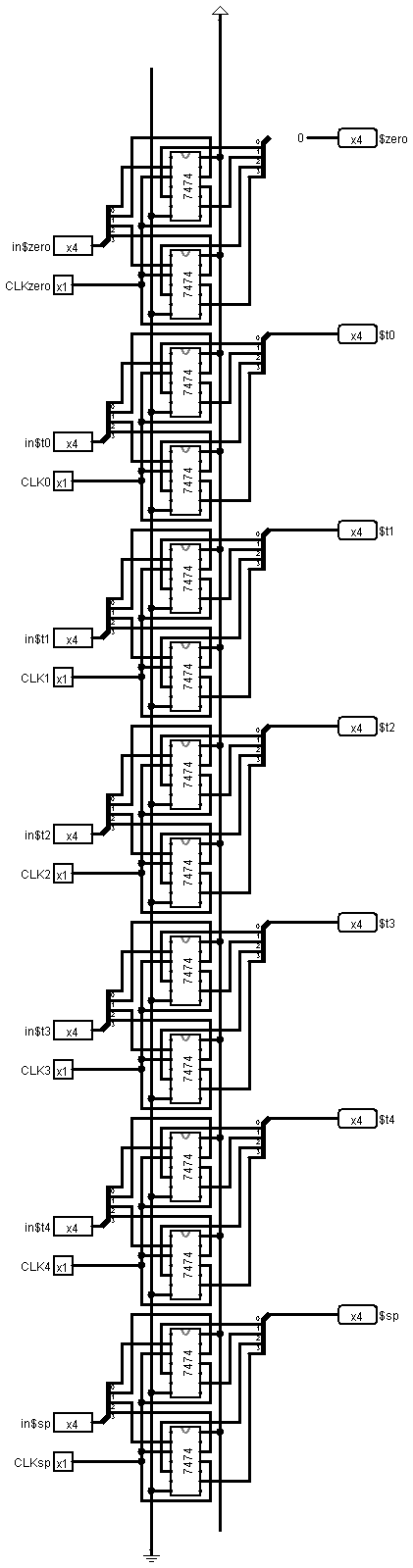
In every clock cycle, the PC register outputs the stored value of the PC, while the current PC instruction is being executed. At the same time, the PC instruction sends the next value for the PC as an input for the PC register. The next value can either be PC+1 or PC+jump amount, depending on the specific instruction being executed.

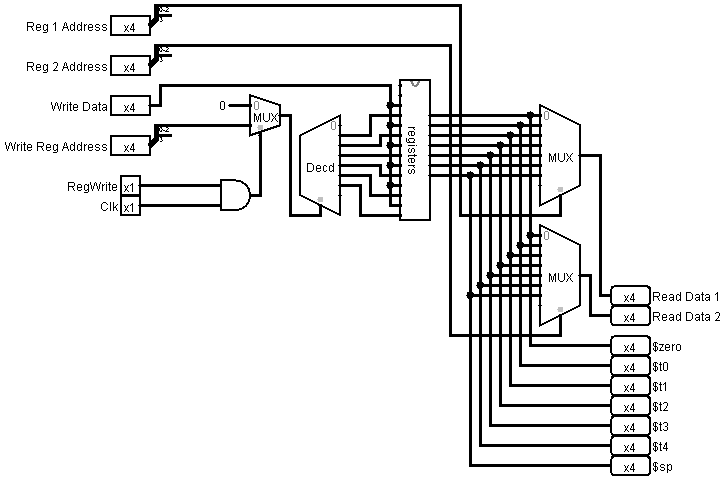


**5. Register File:**

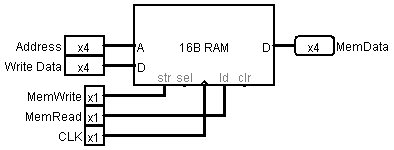
To implement our register file, we first created a file consisting of 7 registers. Among these registers, we implemented 5 temporary registers, namely $t0, $t1, $t2, $t3, and $t4. Additionally, we used the $sp register to handle stack push and pop instructions, and implemented the $zero register to handle arithmetic operations with zero value.

To read from or write to a register, the register file takes the register address as an input. We also need to provide a write data value and the corresponding write register address as input when writing to a register. However, if we are executing a store word instruction, we do not need to write in the register file. Therefore, we take a selection bit, RegWrite, as an input to determine whether we need to write the value or not.





**6. Data Memory :**



Data Memory is our storage for memory . It takes the memory address to write and the data to write in the 16B RAM. Now there can be read from memory in lw instruction and write in memory in store word instruction. So based on two selector bits , we select our operation either to read or write . For memory read (lw) , the output is our 4 bit memory data.

**Tools and Apparatus:**

* **Integrated Circuits:**

|  |  |  |
| --- | --- | --- |
| ATMEGA - 32 |  | 6 |
| SN74HC157N | QUAD 2 TO 1 MUX | **7** |
| SN74HC83N | 4 bit Binary Full Adder | 2 |

* **Simulator :**

Software: Logisim

Version: Logisim-win-2.7.1

**Discussion:**

1. For implementing the circuit in software level , we used 7400-library integrated circuits. In hardware level, necessary coding has been done at ATMEGA level.
2. In hardware level, we showed the value of each register by setting ShowReg in the register file On. From the ALU, we demonstrated output in each step of operation. The value of PC was also shown for each clock cycle. The value of each data memory was also visible at software level.
3. All of our ATmega were tested before integrating it to our final circuit.
4. Wires with different sizes were used to make the whole circuit organized and easily understandable for the user.
5. The outputs were checked multiple times for a large number of input test cases to ensure that our circuit satisfies the expected output values from the given input file.
6. In order to optimize the number of IC , we performed PC+1 using the Cin value of the adder instead of using a separate adder. Similarly instead of generating a circuit to create a selector bit bsel ( whether to branch or not ) , we calculated the value in ATmega designated for ALU operation .