BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CSE 306 (Computer Architecture Sessional)

4-bit MIPS Design Simulation and Implementation

Subsection: A2

Group 6

Gourab Biswas
2005034

Md. As-Aid Rahman
2005039

Md. Farhad Al Amin Dipto
2005042

Mahamudul Hasan Fahim
2005043

Sabbir Alam Saad
2005059

February 10, 2024

Contents

List of Figures

List of Tables

1 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 $x = a + b + c$]

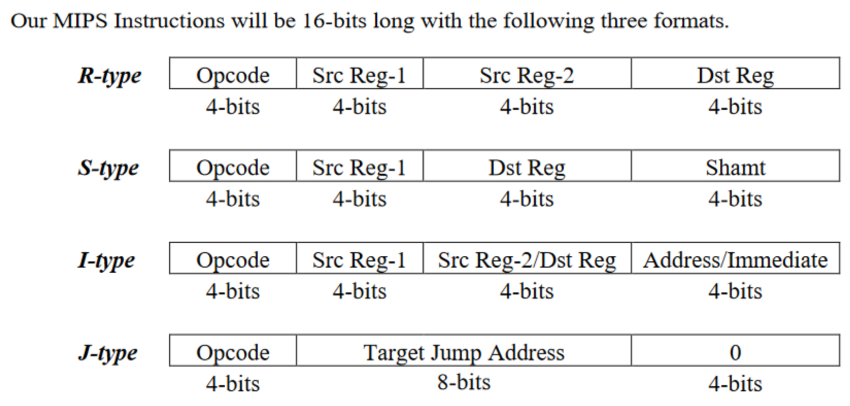
According to MIPS instruction rules, arithmetic operations can only take registers as arguments, size of a register is 32 bits and there are 32 registers in total. A datapath is built with registers, ALUs, MUXs, memories and controls elements that can process data and addresses in the CPU. MIPS instructions are fed through a datapath to perform various instructions like addition, load/store, branching or jump.

2 Problem Specification

2.1 Instruction Set

We have been assigned to implement 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.

MIPS INSTRUCTION FORMAT



Our MIPS Instructions will be 16-bits long with the following three formats.

R-type	Opcode	Src Reg-1	Src Reg-2	Dst Reg
	4-bits	4-bits	4-bits	4-bits
S-type	Opcode	Src Reg-1	Dst Reg	Shamt
	4-bits	4-bits	4-bits	4-bits
I-type	Opcode	Src Reg-1	Src Reg-2/Dst Reg	Address/Immediate
	4-bits	4-bits	4-bits	4-bits
J-type	Opcode	Target Jump Address		0
	4-bits	8-bits		4-bits

Instruction set for our MIPS is given below.

Table 1: INSTRUCTION SET DESCRIPTION

Instruction ID	Category	Type	Instruction
A	Arithmetic	R	add
B	Arithmetic	I	addi
C	Arithmetic	R	sub
D	Arithmetic	I	subi
E	Logic	R	and
F	Logic	I	andi
G	Logic	R	or
H	Logic	I	ori
I	Logic	S	sll
J	Logic	S	srl
K	Logic	R	nor
L	Memory	I	sw
M	Memory	I	lw
N	Control-conditional	I	beq
O	Control-conditional	I	bneq
P	Control-unconditional	J	j

Opcodes of the instructions are of 4 bits, so between 0 and 15. We're given the Instruction assignment **AHLOGBFNMJDPECKI**. So, our opcodes will point to corresponding operations as described below:

Opcode	Operation
0000	add
0001	ori
0010	sw
0011	bneq
0100	or
0101	addi
0110	andi
0111	beq
1000	lw
1001	srl
1010	subi
1011	j
1100	and
1101	sub
1110	nor
1111	sll

3 Complete Block diagram of a 4-bit MIPS processor

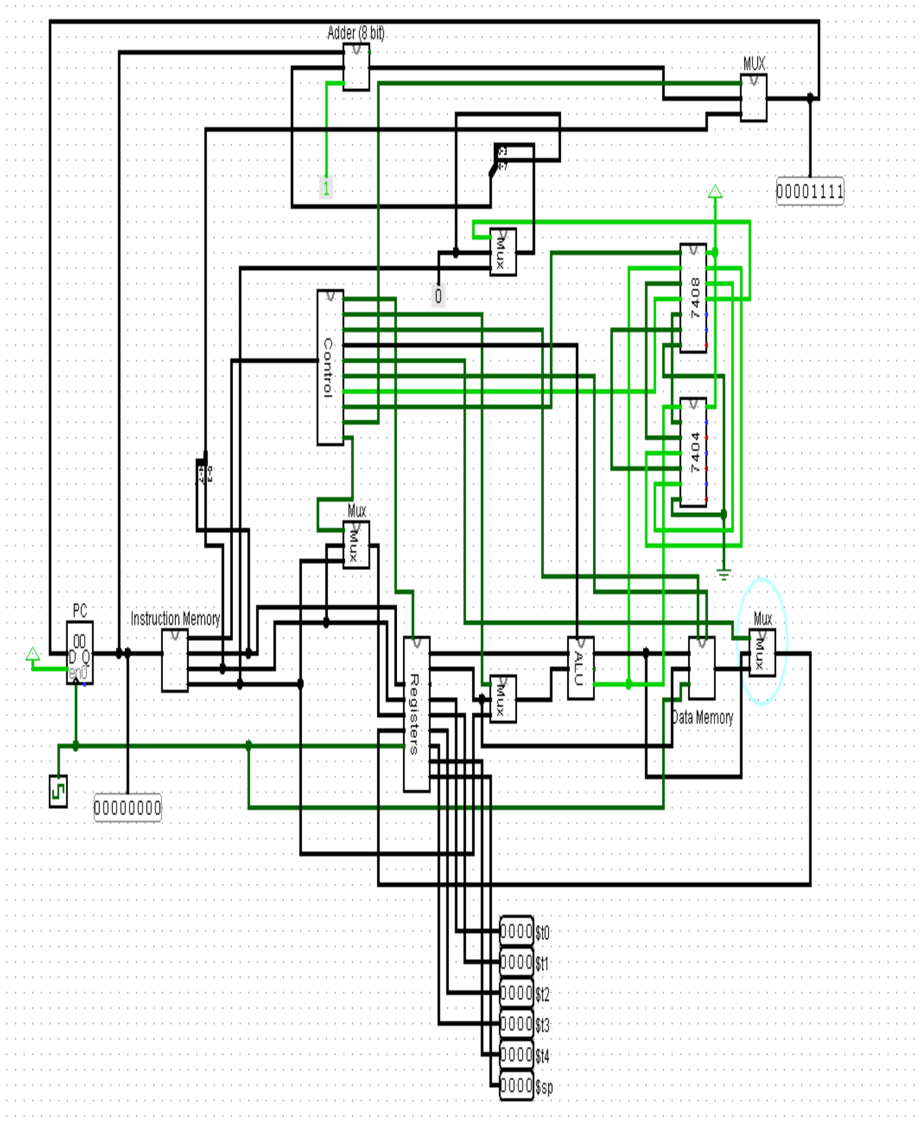


Figure 1: 4-bit MIPS Processor

4 Block diagrams of the main components

4.1 Instruction Memory with PC

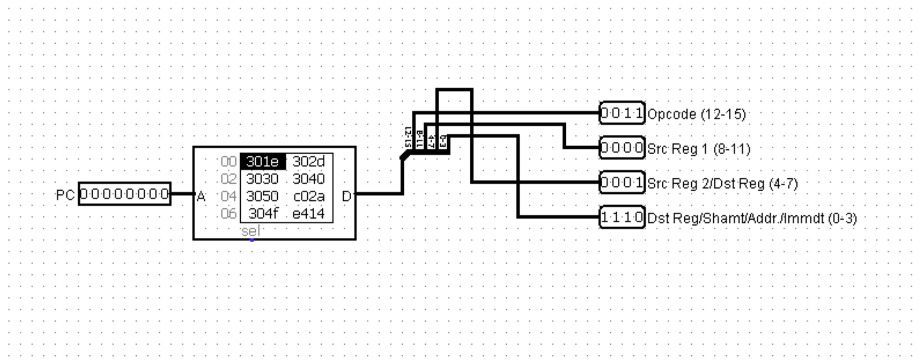


Figure 2: Instruction Memory with PC

4.2 Data Memory with the Stack

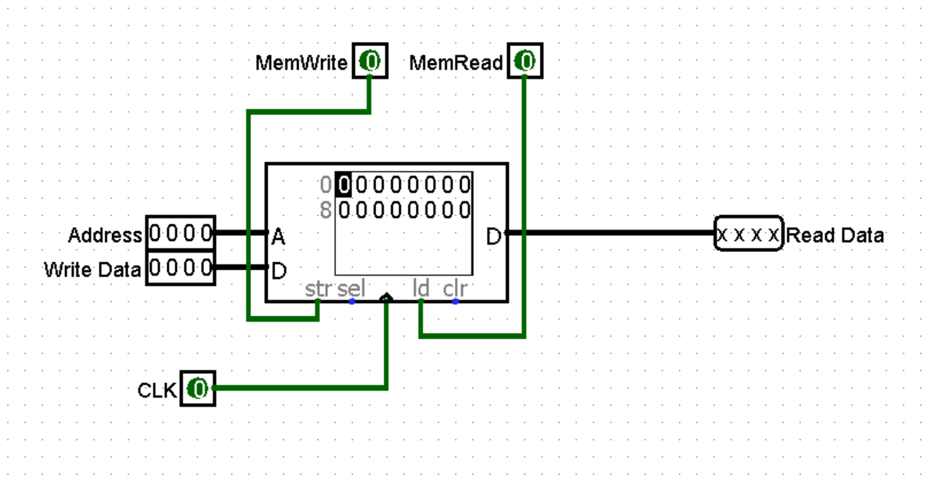


Figure 3: Data Memory with the Stack

4.3 Register File

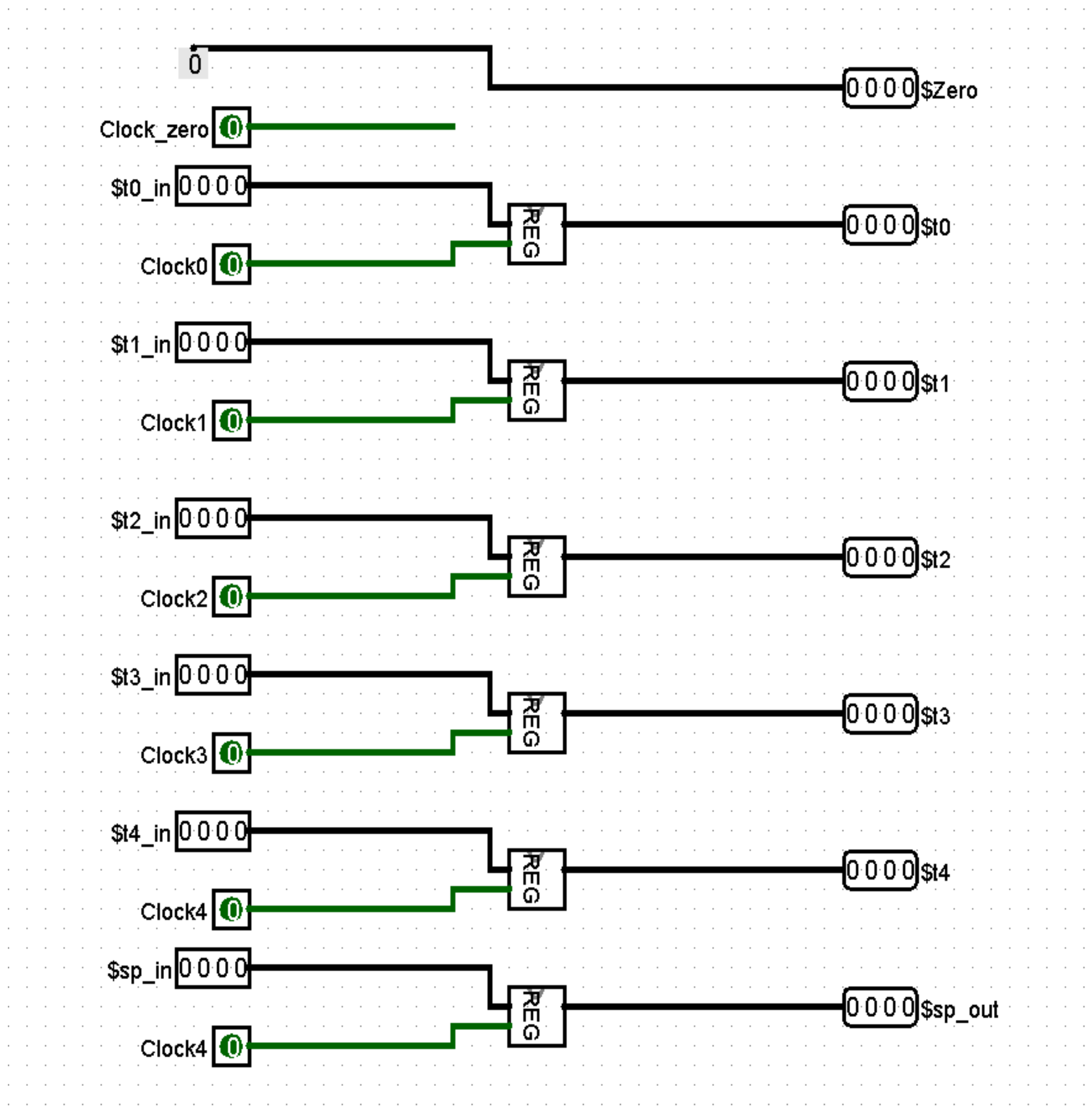


Figure 4: Register File

4.4 Control Unit

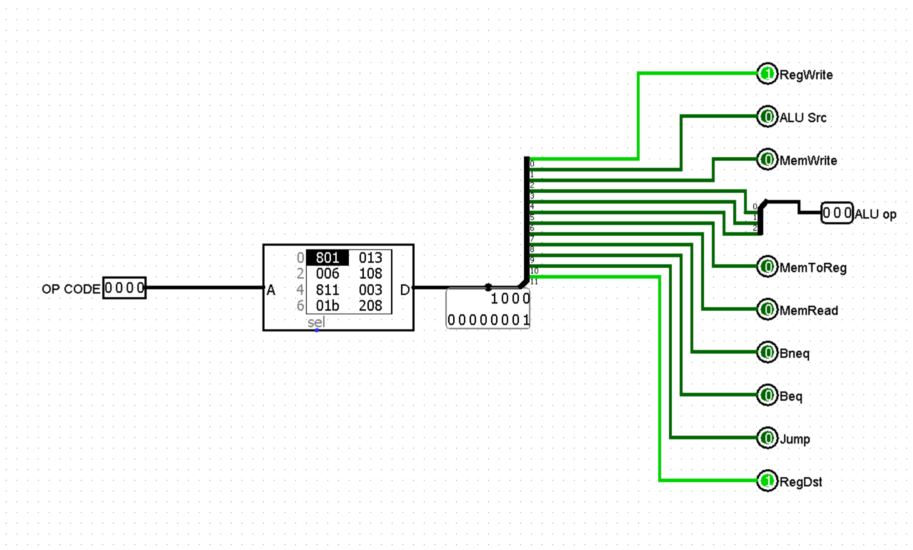


Figure 5: Control Unit

5 Approach to implement the push and pop instructions

We have implemented push and pop instructions using the assembler that we made using *FLEX* and *BISON*. We have made use of sw, lw, addi, subi operations here.

5.1 PUSH Operation

The **push_instruction** non-terminal in `parser.y` handles the implementation of PUSH. **Implementation Steps:**

Store Value: The value to be pushed is typically stored in a register.
Update Stack Pointer: After storing the value, the stack pointer needs to be decremented to point to the next available memory location.

PUSH REGISTER
Example: PUSH $t0
Explanation:

- The machine code instructions generated for this PUSH operation involve storing the value of register $t0 onto the stack and updating the stack pointer.
- Suppose the current stack pointer $sp points to address 0x1000.
- The value of register $t0 needs to be pushed onto the stack.
- Let's assume that $t0 contains the value 0xABCD.
- The generated machine code instructions might look something like this:
    1. `sw $t0, 0($sp)` - Store the value of $t0 (0xABCD) at memory address 0x1000.
    2. `subi $sp, $sp, 1` - Update the stack pointer to point to the next available memory location 0x0FFF, decrementing it by 1.

PUSH INT LPAREN REGISTER RPAREN
Example: PUSH 8($t1)
Explanation:

- This form of PUSH operation involves loading a value from memory into a register and then pushing that value onto the stack.
- Let's assume that $t1 contains the value 0x2000 and we want to push the value stored at memory address 0x2008 onto the stack.
- The generated machine code instructions might look like this:
    1. `lw $t1, 8($t1)` - Load the value stored at memory address 0x2008 into register $t1.
    2. `sw $t1, 0($sp)` - Store the value of $t1 (value at 0x2008) onto the stack at the current stack pointer address.
    3. `subi $sp, $sp, 1` - Update the stack pointer to point to the next available memory location, decrementing it by 1.

5.2 POP Operation

The **pop_instruction** non-terminal handles the implementation of POP operation.

POP REGISTER

Example: ‘POP $t2‘

Explanation:

- The machine code instructions generated for this POP operation involve loading a value from the stack into register $t2 and updating the stack pointer.
- Suppose the current stack pointer $sp points to address 0x0FFF.
- The value to be popped from the stack is stored at memory address 0x1000.
- The generated machine code instructions might look like this:
  1. `lw $t2, 0($sp)` - Load the value stored at memory address 0x1000 into register $t2.
  2. `addi $sp, $sp, 1` - Update the stack pointer to point to the next available memory location 0x1000, incrementing it by 1.

In summary, PUSH and POP instructions manipulate the stack by storing and loading values using sw and lw operations respectively, and they update the stack pointer accordingly to reflect changes in the stack structure. These operations ensure proper stack management during program execution.

6 ICs used with their count

Name	IC Number	Count
40 pin microprocessor	ATmega32	6
4bit Binary Full Adder	7483	2
Quad 2 to 1 MUX	74157	6

Table 2: ICs with counts

7 Simulator Info

Logisim - Java Platform (Version 2.7.1)

8 Discussion

- We designed individual components in Logisim and recreated the whole circuit in Proteus to ease the hardware implementation.
- We learnt about probable incompatibility issues between HC and LS IC series. We avoided them during the hardware implementation.
- For the designing of ALU according to the given functions, we formed truth tables, used it to construct the required k-maps for X_i, Y_i and Z_i, derived equations for each of them.
- We used 6 ATmega32 to separate the stages like a proper MIPS processor.
- For implementing the circuit in software level , we used 7400-library integrated circuits. In hardware level, necessary coding has been done at ATmega32 level.

9 Contribution

- 2005034 : Provided necessary codes and wrote the report taking help from everyone else. Lended a hand in hardware implementation. Helped in simulation in Proteus.
- 2005039 : Managed all the hardware equipments, played a great role in hardware implementation. Simulated MIPS in Proteus.
- 2005042 : Proposed the overall implementation idea and guided the team while designing the hardware. Designed the circuits in logisim and also contributed in Proteus simulation. Implemented MIPS processor in hardware.
- 2005043 : Provided necessary help in report and codes. Helped in designing in Proteus.
- 2005059 : Designed control unit, program counter, adder and MUX in logisim. Assisted in hardware implementation.

All of us collectively completed the 4-bits MIPS processor, going through multiple sessions of designing and debugging, lending each other shoulders. After 3-4 days of uncertainty, we finally saw our hardwork