CSE 306: Computer Architecture Sessional

Assignment-3: 4-bit MIPS Implementation

Section - B2

Group - 02

Members of the Group:

i. 2005092 Shahad Shariar Rahman

ii. 2005094 Golam Mostofa

iii. 2005099 Maisha Maksura

iv. 2005105 Mohammad Ishrak Adit

v. 2005108 Kazi Redwan Islam

Contents

1 Introduction

MIPS (Microprocessor without Interlocked Pipeline Stages) is a Reduced Instruction Set Computer (RISC) architecture developed in the early 1980s by researchers at Stanford University. It is characterized by its simplicity and efficiency, with a focus on maximizing performance by streamlining the instruction set and optimizing the pipeline structure.

One of the key principles of MIPS architecture is the RISC philosophy, which advocates for a smaller, simpler set of instructions that can be executed efficiently in hardware. This approach contrasts with Complex Instruction Set Computing (CISC) architectures, which feature a larger and more diverse instruction set.

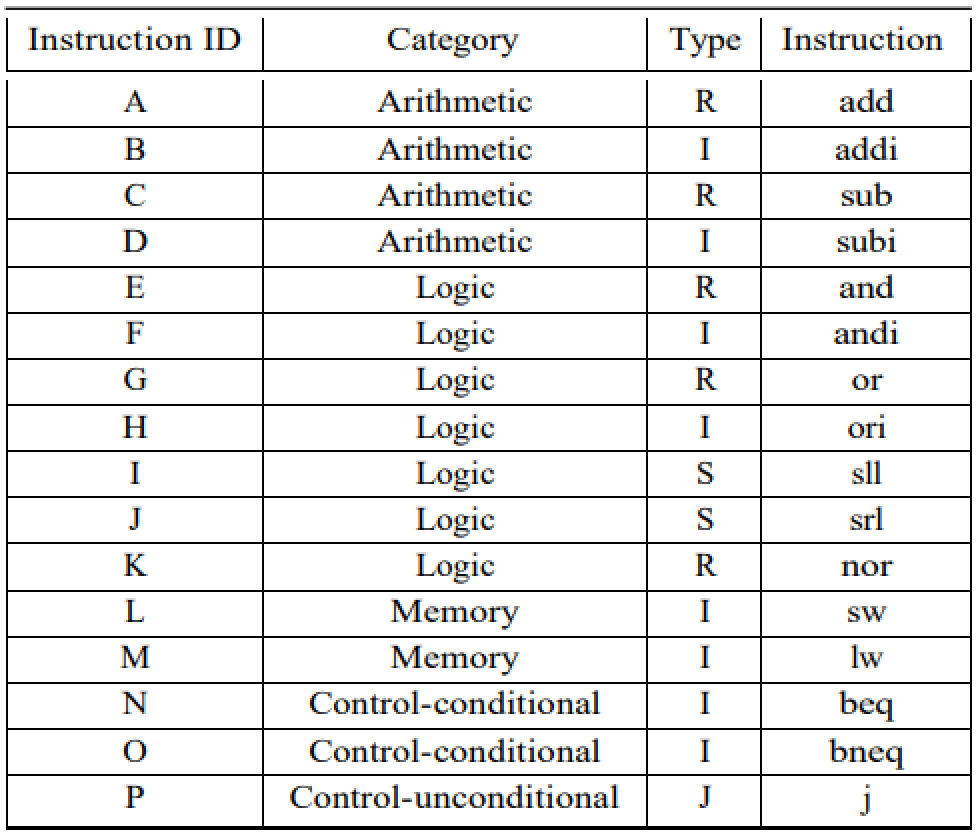
MIPS processors feature a fixed-length instruction format, typically 32 bits, which simplifies instruction decoding and execution. Instructions are divided into categories such as arithmetic and logic operations, memory access, and control flow, with each category represented by a small number of simple instructions.

The architecture's pipeline design enables multiple instructions to be processed simultaneously, with each stage of the pipeline handling a different aspect of instruction execution. This pipelining technique enhances performance by allowing the processor to overlap the execution of multiple instructions.

MIPS architecture has found widespread use in a variety of applications, including embedded systems, networking equipment, and consumer electronics. Its popularity stems from its balance of simplicity, performance, and power efficiency, making it a popular choice for a broad range of computing tasks. Additionally, MIPS instruction set has been influential in the development of other RISC architectures and has served as the basis for various academic and commercial processor designs.

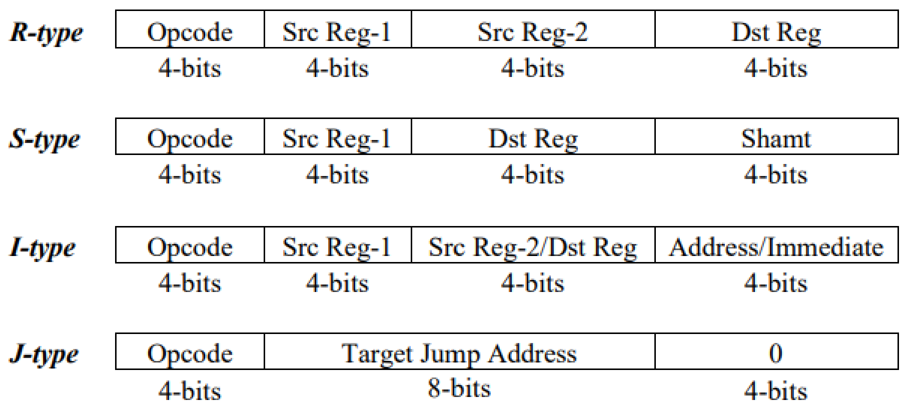
2 Instruction Set

2.1 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

2.2 MIPS Instruction Format



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

2.3 Instruction Set Assignment for Group B2-02

Serial No.	Instruction Id	OpCode
1	H	0000
2	C	0001
3	A	0010
4	G	0011
5	N	0100
6	M	0101
7	I	0110
8	J	0111
9	D	1000
10	B	1001
11	O	1010
12	F	1011
13	E	1100
14	P	1101
15	K	1110
16	L	1111

Table 1: Team Specific Instruction Set

3 Complete Block diagram of a 4-bit MIPS processor

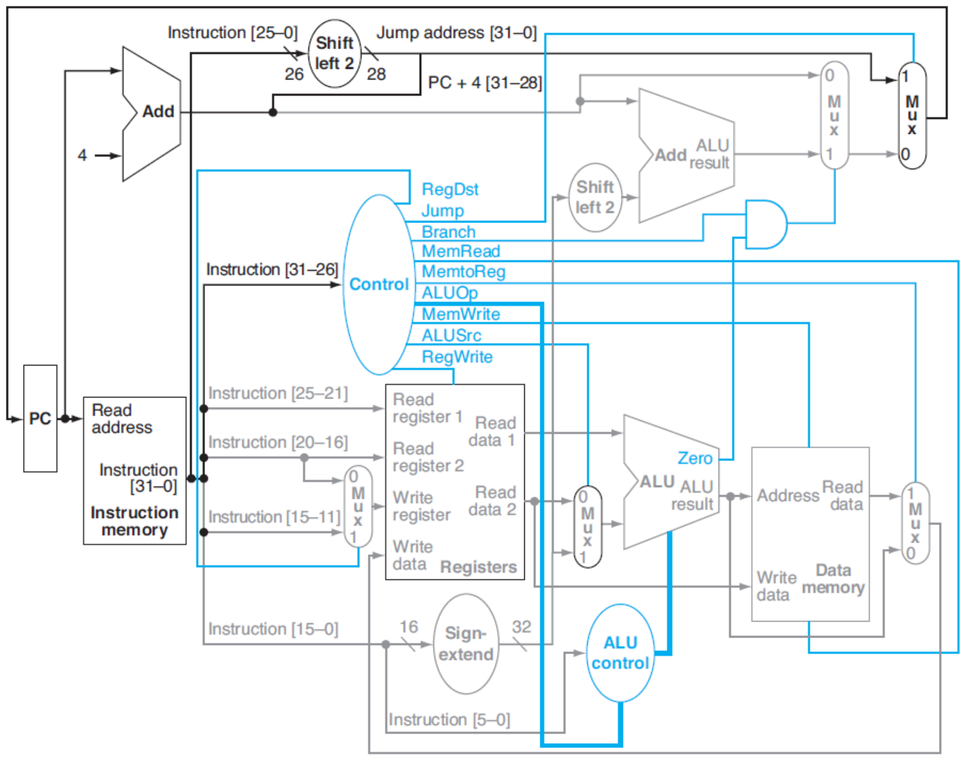


Figure 1 : MIPS Complete Datapath

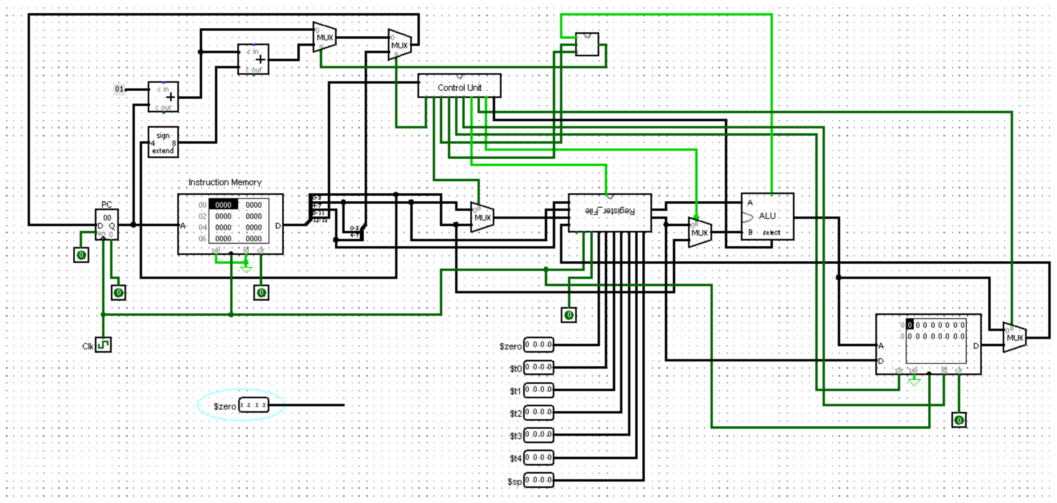


Figure 2 : Implemented Complete Circuit On Logisim

4 Block diagrams of the main components

4.1 Instruction Memory with PC

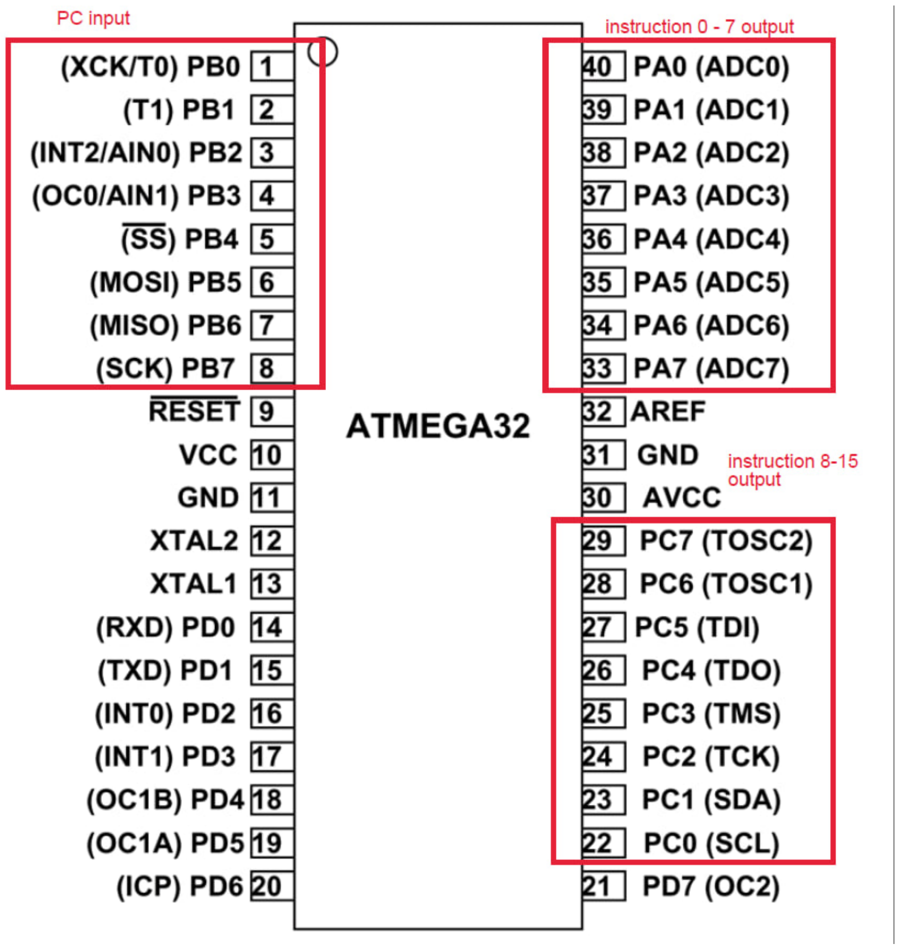


Figure 3 : Instruction memory with ATmega32

4.2 Register File

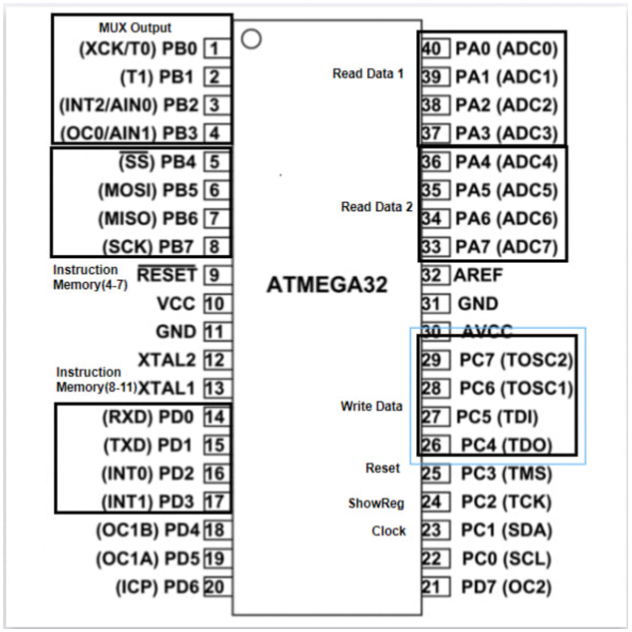


Figure 4 : Register memory with ATmega32

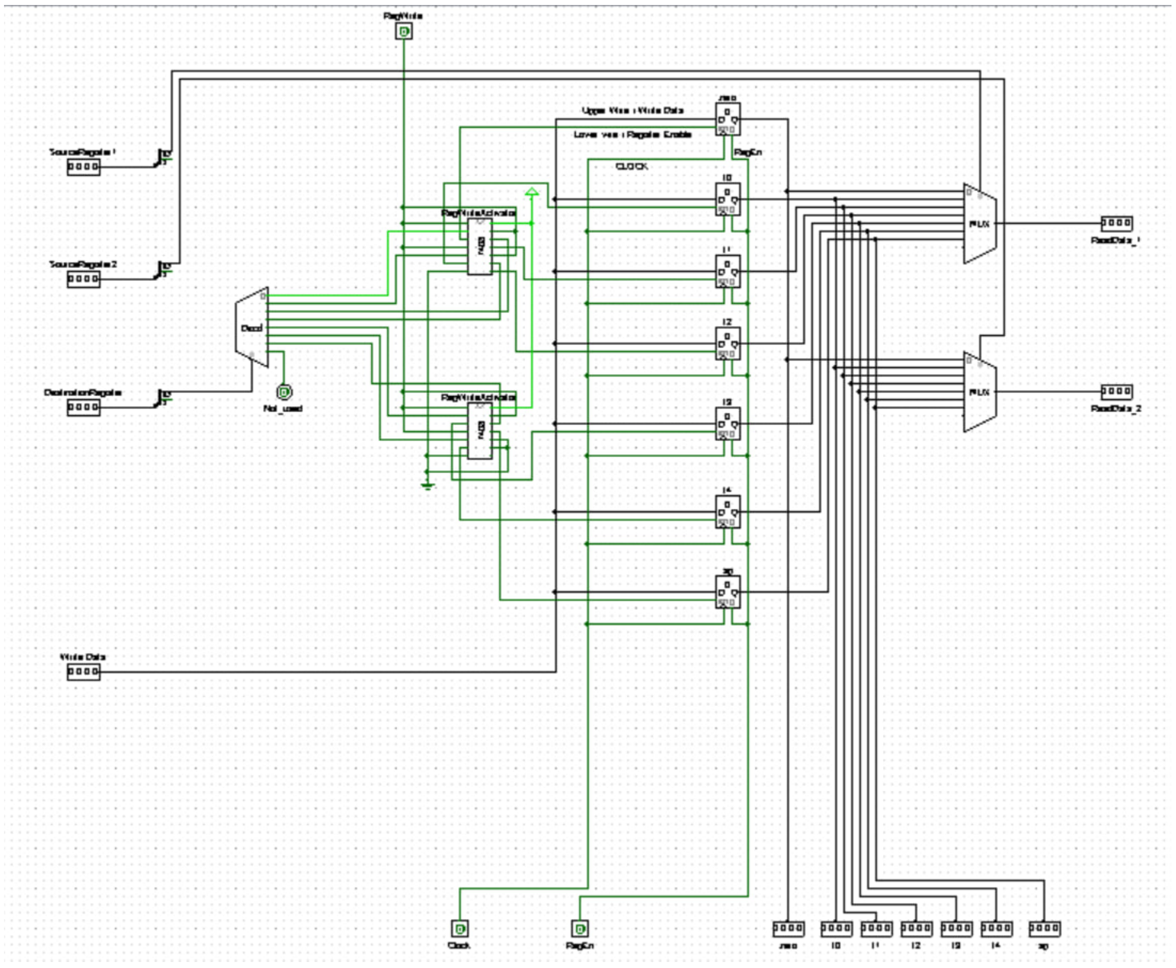


Figure 5 : Register memory in Logisim

4.3 Control Unit

Control unit takes 4 bit OpCode (15-12 bit of instruction) as input and generates a 12 bit output that indicates all the flags respectively the columns of the following table. the hex representation of the 12 bit output is stored in the control unit memory. We have implemented it with ATmega32, also merged it with the branch segment that says if the branching will be active or not.

OpCode	Ins ID	Ins	Jump	RegDst	Beq	Bneq	MemWrite	MemRead	RegWrite	MemToReg	ALUsrc	ALUop	hex
0000	H	ORi	0	0	0	0	0	0	1	0	1	001	029
0001	C	SUB	0	1	0	0	0	0	1	0	0	011	423
0010	A	ADD	0	1	0	0	0	0	1	0	0	010	422
0011	G	OR	0	1	0	0	0	0	1	0	0	001	421
0100	N	Beq	0	0	1	0	0	0	0	0	0	011	203
0101	M	LW	0	0	0	0	0	1	1	1	1	010	07a
0110	I	SLL	0	0	0	0	0	0	1	0	1	110	02e
0111	J	SRL	0	0	0	0	0	0	1	0	1	111	02f
1000	D	SUBi	0	0	0	0	0	0	1	0	1	011	02b
1001	B	ADDi	0	0	0	0	0	0	1	0	1	010	02a
1010	O	Bneq	0	0	0	1	0	0	0	0	0	011	103
1011	F	ANDi	0	0	0	0	0	0	1	0	1	000	028
1100	E	AND	0	1	0	0	0	0	1	0	0	000	420
1101	P	Jump	1	0	0	0	0	0	0	0	0	000	800
1110	K	NOR	0	1	0	0	0	0	1	0	0	100	424
1111	L	SW	0	0	0	0	1	0	0	0	1	010	08a

Table 2: Table of Control Flags

Bneq	Beq	Zero flag	Branch decision
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Table 3: Truth Table of Branch Decision

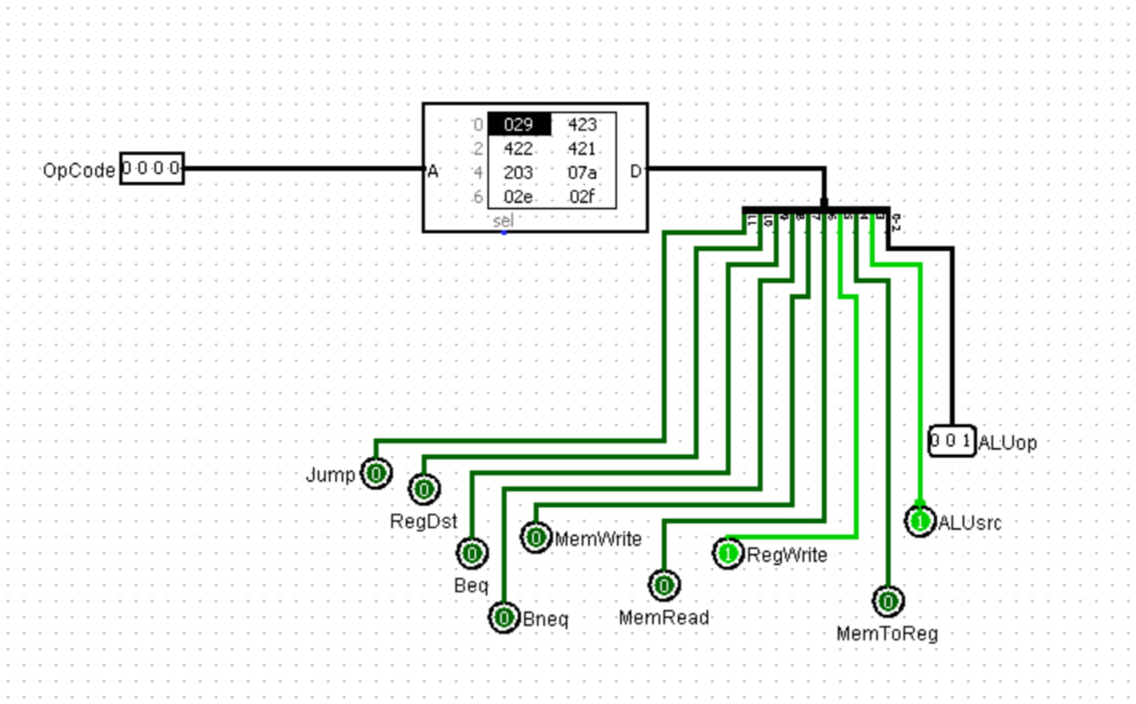


Figure 6 : Control Unit

5 Approach to implement the push and pop instructions

In the specification it was said to implement push and pop instructions as extra feature.
At first the stack pointer was moved at top of the stack.According to the specification we were told to implement two types of push instruction.

Instruction	**Description**
push $t0	mem[$sp] = $t0
push 3($t0)	mem[$sp] = mem[$t0+3]

Table 4: Instruction Descriptions

For the first type push $t0 we need need to first store the value of the given register at the address pointed by stack pointer. Then we need to decrement the offset of the stack pointer by 1.
For the later type we'd load the value of the specific memory into a register. Then we need to store the value of the register at the address pointed by stack pointer.Then we need to decrement the offset of the stack pointer by 1.

There is only a single type of pop.

Instruction	**Description**
pop $t0	$t0 = mem[$sp] (keep)

Table 5: Instruction Descriptions

At first we increased the offset of the stack pointer by 1.Then we loaded the value pointed by the stack pointer into the specific register.

6 ICs used with their count

Name of IC	IC Count
AtMega32A	5
Total	5

Table 6: IC name and Count

7 Discussion

Implementing a MIPS architecture design from scratch, beginning with a simulation in Logisim and progressing to hardware implementation, presented several challenges and opportunities for learning. This discussion delves into the experiences encountered, the errors faced, and the lessons gained throughout the process.

One of the primary challenges encountered during the hardware implementation of the MIPS architecture was translating the simulated design from Logisim to physical hardware. While Logisim provides a convenient platform for designing and simulating digital circuits, the transition to hardware introduces real-world constraints such as signal propagation delays, power consumption considerations, and managing the interface with physical components.

Several errors were encountered during the hardware implementation phase, ranging from timing violations to issues with signal integrity. Timing violations, caused by failing to meet setup and hold time requirements for flip-flops and other sequential elements, manual errors while combining Atmel studio codes with circuit design and building circuit according to proper pin diagram were particularly prevalent. These errors often manifested as unexpected behavior in the circuit or, in severe cases, complete system failure. Addressing timing violations required careful analysis of signal paths, and adjustment of clock frequencies. Optimization of circuits and voltage checking through the paths helped to resolve unexpected errors.

In addition to technical challenges, the hardware implementation process provided valuable insights into the practical considerations involved in designing and deploying digital systems. These considerations include component selection, board layout, power management. Balancing performance, cost, and power efficiency constraints requires careful consideration at each stage of the design process.

8 Contribution

Student ID	Contribution
2005092	Register File
2005094	Instruction Memory
2005099	Control Unit
2005105	Data Memory
2005108	Arithmatic Logic Unit

Table 7: Contribution