Name: - Rishabh Sharma
Student Id: - 109874160

Aim: - **MIPS multi-cycled CPU Simulation Project**

Scope: - simulation of the MIPS pipeline is limited to 11 instructions only.

Instruction Type	**Instruction Name**
Arithmetic	add, sub
Data transfer	lw, sw
Logical	and, or, nor
Conditional branch	beq, bne, slt
Jump	j

Implementation of Simulator: -

Implementation of the simulation is divided into 2 phases,

- Phase 1: - Implementation of the Arithmetic Logic Unit
- Phase 2: - Implementation of the MIPS pipeline

Phase 1: - ALU implementation (task performed on EX cycle)

In this phase, 6 ALU operations were implemented: - add, sub, or, and, nor, and slt. ALU takes two 32-bit binary numbers as input along with the control signals (figure 2) ainv, binv, 2-bit operation-code (op) and returns the result of ALU operation (32-bit binary number). The control bit ainv and binv is used in 2*1 mux inside 1 – bit ALU for deciding which of a or ā and b or $\bar{b}$ respectively and op is used to decide which of “add”, “and”, and “or” operation is to be performed using 2-bit op. ALU operations on the 32-bit binary number is performed bit-by-bit, i.e., ALU operation is repeated 32 times performing operation on each bit forming 1-bit ALU. The control signals for 1-bit ALU includes carry_in (1 bit) apart from all the control signals used in 32 – bit ALU. ALU operation for the MSB (31st bit) is performed using a special 1-bit ALU component which checks the occurrence of the offset during ALU operations. Implementation of the Phase 1 is explained in figure 1.

Name: - Rishabh Sharma
Student Id: - 109874160

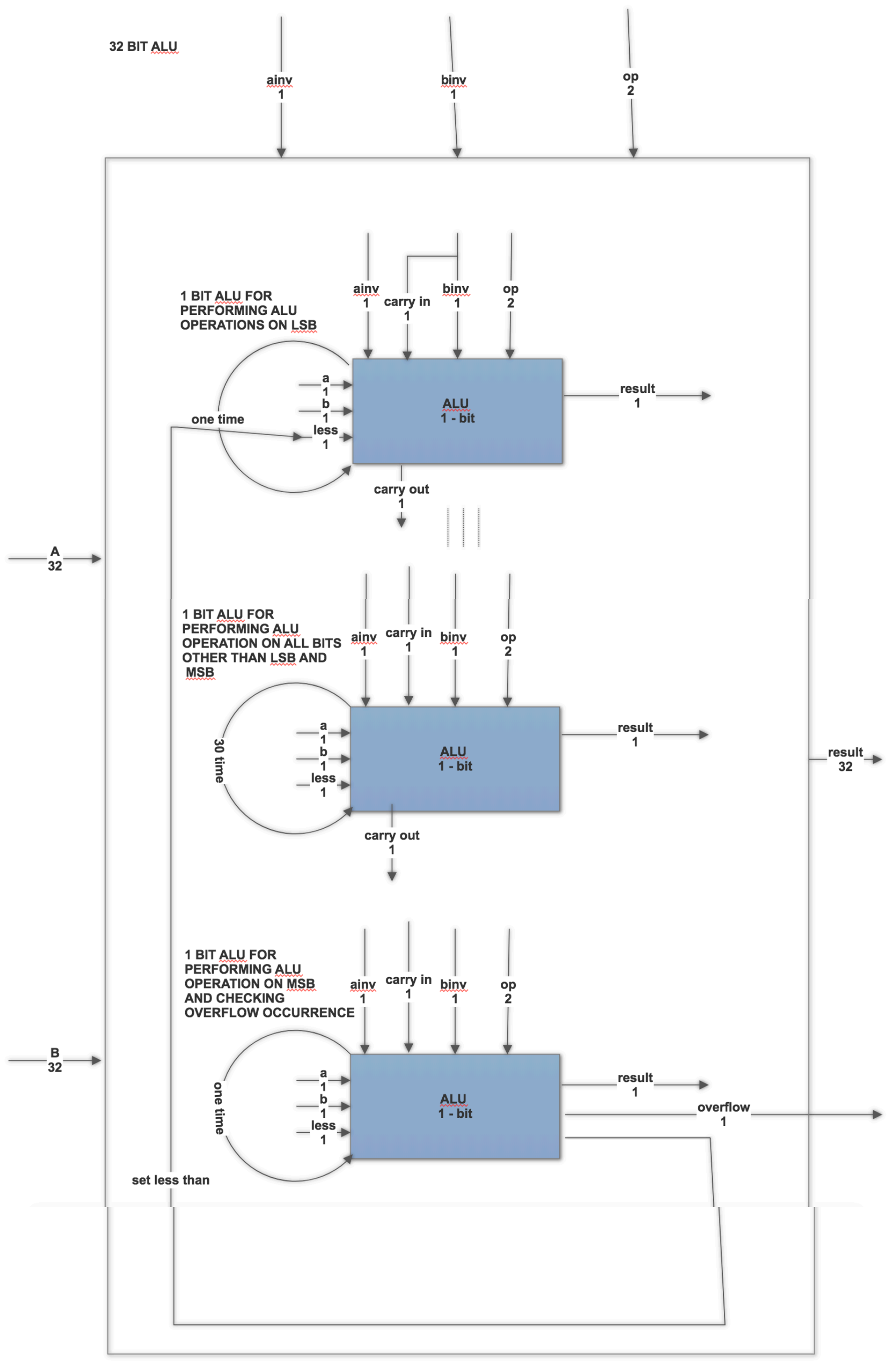
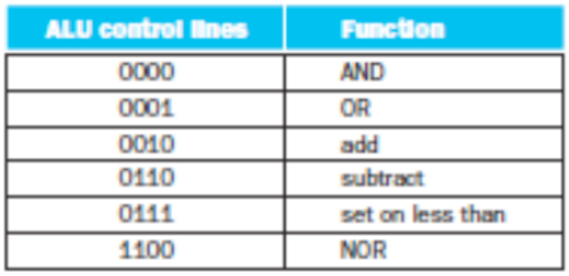


Figure 1 Implementation of the 32 – bit ALU

Name: - Rishabh Sharma
Student Id: - 109874160



ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

Figure 2: Control signals used for deciding the ALU operation to be performed

1 – bit ALU performs the ALU operations based in the control signals shown in Figure 2. These controls bits represent ainv, binv, op [0] and op [1] from left to right.

<u>Phase 2: - MIPS pipeline implementation</u>

Instructions/Data first loaded from the file to the global variable memory [200] [8] (0-99 represent instruction memory and 100-199 represent data memory) each memory address stores 8 bits of information. Information is stored in Big endian mode from the instruction-data file to the memory during loading operation. All the 32 registers for MIPS is stored and initialized in register [32] [32] represents 0 - 31 register each can contain data of 32 bits. Special Registers like PC - Program Counter, IR - Instruction register, ALU-Out - ALU output register, MDR - Memory Data Register, A and B register used as an input for ALU are also initialized to zero and used for executing the MIPS instructions. MIPS pipeline is implemented using 5 functions each representing 5 different clock cycles of MIPS namely Instruction Fetch, Instruction Decode, Execution, Instruction Memory Access, and Write Back. All the five clock cycles use 2*1 and 4*1 multiplexer with 13 different control signals for proper execution of the instruction. The flow chart of the implemented model of the MIPS is shown in figure 3. Implementation for each clock cycle is based on the reference figure 4 and figure 5

Name: - Rishabh Sharma
Student Id: - 109874160

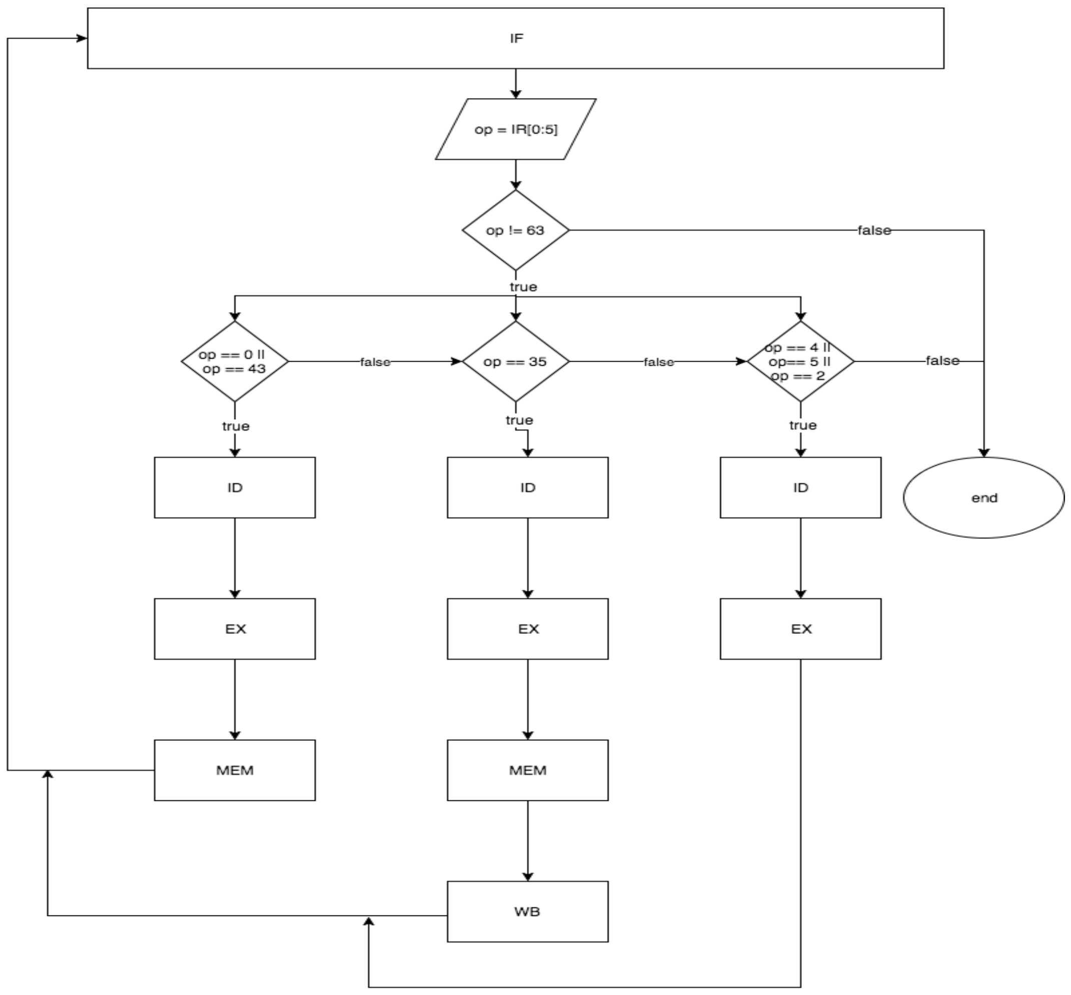
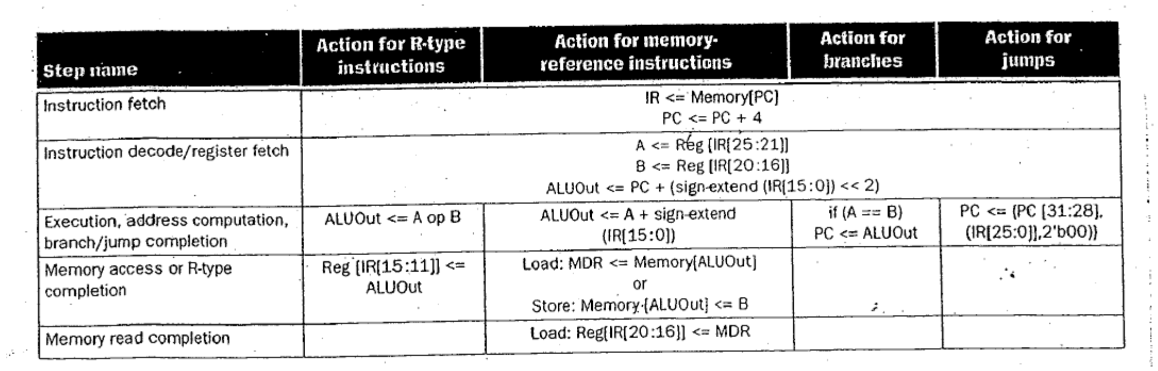


Figure 3: - Flow Chart of the model used for implementation of MIPS architecture (op = opcode of the fetched instruction)



Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	if (A == B) PC <= ALUOut	PC <= {PC [31:28], (IR[25:0]],2'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

Figure 4: - summary of steps taken to execute any instruction class

Name: - Rishabh Sharma
Student Id: - 109874160

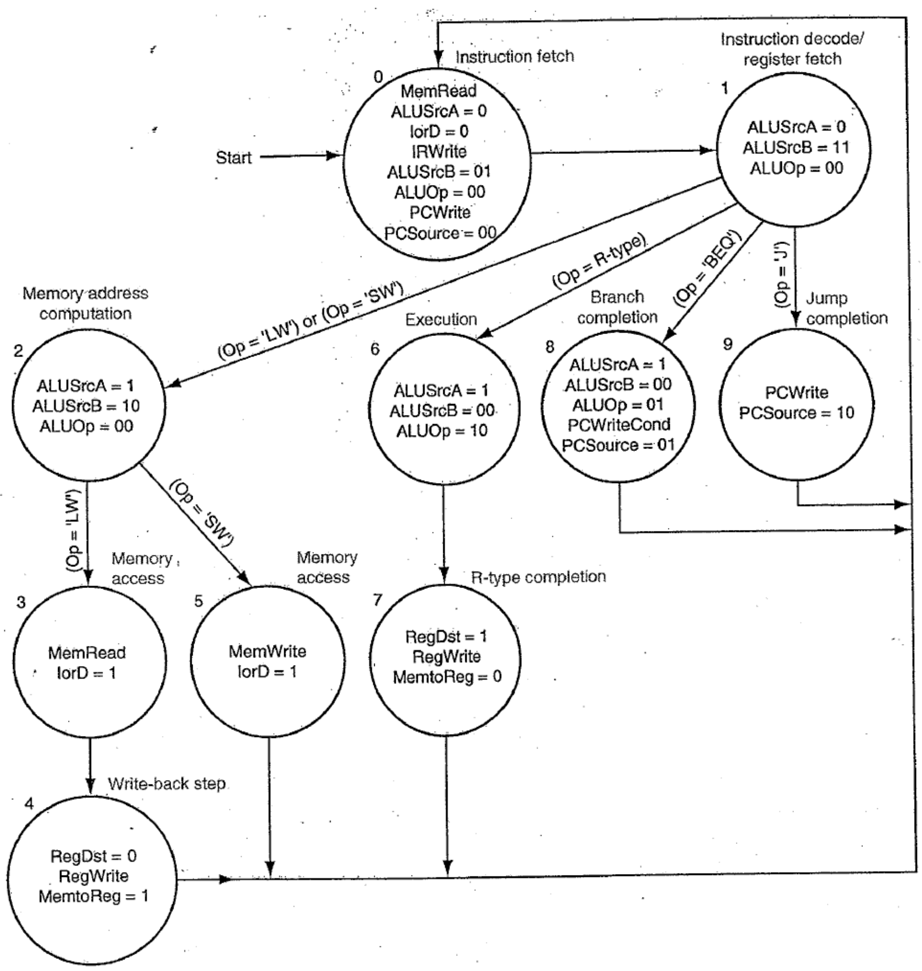


Figure 5: - Finite Sate diagram for multicycle control

The identification of the instruction based on the op-code is done using the table 1.

Instruction Name	**Op Code in Decimal**	**Functional Code in Decimal**
add	0	32
sub	0	34
lw	35	
sw	43	
and	0	36
or	0	37
nor	0	39

beq	4	
bne	5	
slt	0	42
j	2	

Table 1: - Op code and function code values for each instruction

Constrains in the code: -

1. Only 6 ALU instruction and 5 other instructions has been implemented in the code.
2. For the successful termination of the MIPS pipeline a special instruction with op = $(63)_d$ must be added (after the instructions ends and before the data line begins) to the object code file before loading it to the memory.

Steps to Run the Simulation: -

1. For compiling the main.cpp (simulation code file), any C++ compiler with C++11 standard is required.
2. The object code file must be present in the same directory level as of the main.cpp file.

Coding Environment Used for Developing Simulator: -

IDE Used: - Xcode - Version 9.2

Compiler: - Clang C++ compiler, with LLVM as backend which is conform to the C++11 standart and uses libc++ as the standart library.

File Structure: - main.cpp and input.txt (file containing object code in binary) must be in the same directory.