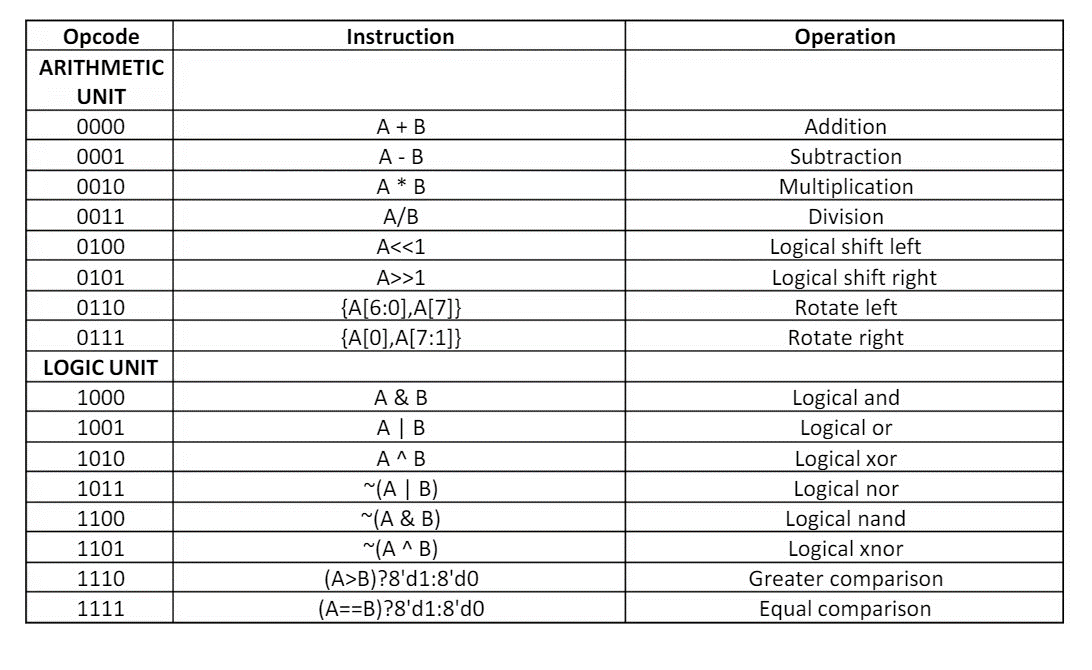
**ALU**

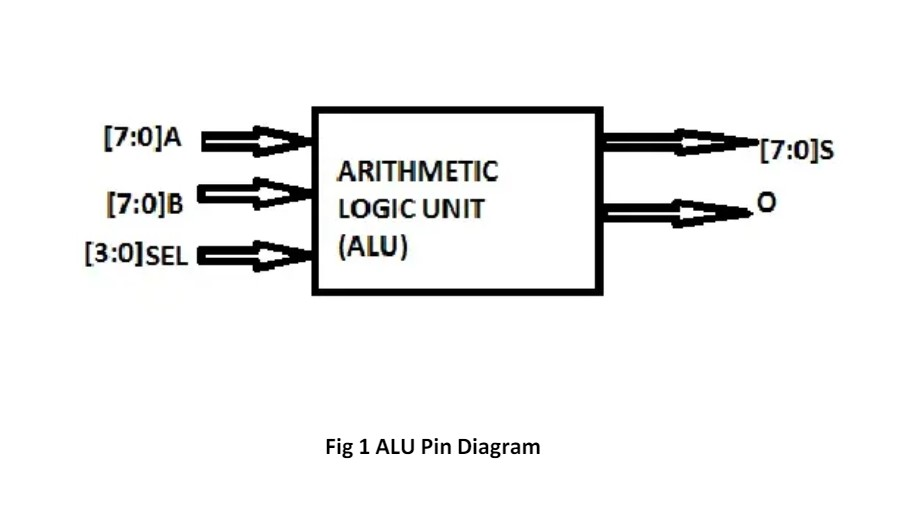
**Theory:**

The Arithmetic Logic Unit (ALU) is a fundamental component of a computer's central processing unit (CPU). It is responsible for performing arithmetic operations such as addition, subtraction, multiplication, and division, as well as logical operations like AND, OR, and NOT. The ALU operates on binary data, manipulating bits to execute these operations. It takes inputs from registers or memory, processes them according to the operation requested by the control unit, and produces outputs that are stored back in registers or memory. ALUs play a crucial role in executing the instructions and calculations necessary for a computer's operations.

**Functions:**



**Diagram:**



**Verilog Code**

**Design Code:**

`timescale 1ns / 1ps

module alu\_8bit(

output [7:0] result,

output carry\_out,

input [7:0] operand\_a,

input [7:0] operand\_b,

input [3:0] operation);

reg [7:0] ALU\_Result;

wire [8:0] extended\_result;

assign result = ALU\_Result;

assign extended\_result = {1'b0, operand\_a} + {1'b0, operand\_b};

assign carry\_out = extended\_result[8];

always @(\*)

begin

case (operation)

// Arithmetic operations

4'b0000: ALU\_Result = operand\_a + operand\_b; // Addition

4'b0001: ALU\_Result = operand\_a - operand\_b; // Subtraction

4'b0010: ALU\_Result = operand\_a \* operand\_b; // Multiplication

4'b0011: ALU\_Result = operand\_a / operand\_b; // Division

// Bitwise operations

4'b0100: ALU\_Result = operand\_a << 1; // Logical shift left

4'b0101: ALU\_Result = operand\_a >> 1; // Logical shift right

4'b0110: ALU\_Result = {operand\_a[6:0], operand\_a[7]}; // Rotate left

4'b0111: ALU\_Result = {operand\_a[0], operand\_a[7:1]}; // Rotate right

// Logical operations

4'b1000: ALU\_Result = operand\_a & operand\_b; // Logical AND

4'b1001: ALU\_Result = operand\_a | operand\_b; // Logical OR

4'b1010: ALU\_Result = operand\_a ^ operand\_b; // Logical XOR

4'b1011: ALU\_Result = ~(operand\_a | operand\_b); // Logical NOR

4'b1100: ALU\_Result = ~(operand\_a & operand\_b); // Logical NAND

4'b1101: ALU\_Result = ~(operand\_a ^ operand\_b); // Logical XNOR

// Comparison operations

4'b1110: ALU\_Result = (operand\_a > operand\_b) ? 8'd1 : 8'd0; // Greater comparison

4'b1111: ALU\_Result = (operand\_a == operand\_b) ? 8'd1 : 8'd0; // Equal comparison

default: ALU\_Result = operand\_a + operand\_b; // Addition (default case)

endcase

end

endmodule

**Testbench code:**

`timescale 1ns / 1ps

module tb\_alu\_8bit;

wire [7:0] result;

wire carry\_out;

reg [7:0] operand\_a;

reg [7:0] operand\_b;

reg [3:0] operation;

alu\_8bit uut(result, carry\_out, operand\_a, operand\_b, operation);

initial begin

operand\_a = 8'b00110011;

operand\_b = 8'b11001100;

operation = 4'b0000;

#1000 $finish;

$monitor("operand\_a=%b operand\_b=%b operation=%b result=%b carry\_out=%b", operand\_a, operand\_b, operation, result, carry\_out);

end

always begin

operand\_a =$random;

operand\_b =$random;

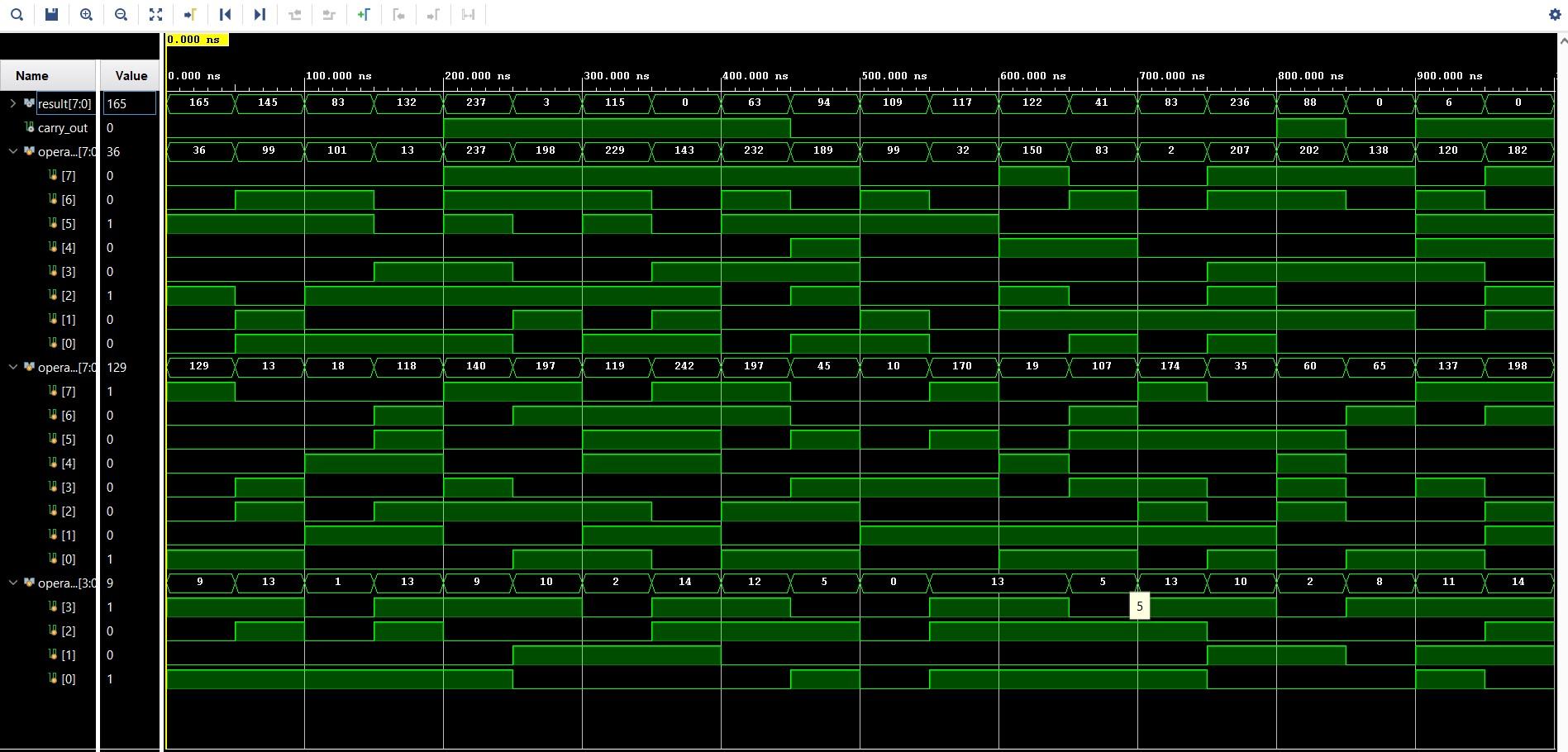
operation =$random;

#50;

end

endmodule

**Simulation:**



**Schematic:**

