4-Bit Arithmetic Logic Unit (ALU) using Verilog

Tools Used

Language: Verilog HDL

Software/IDE: Xilinx Vivado

Kayilainathan J

Linkedin

Description

This project focuses on the design, implementation, and behavioral simulation of a **4-Bit Arithmetic Logic Unit (ALU)**. The ALU is a purely combinational digital circuit capable of performing eight distinct arithmetic and bitwise logical operations. The design was developed using **Verilog HDL** and validated through the **Xilinx Vivado** Design Suite to ensure logical accuracy across various input stimuli.

ALU

An **Arithmetic Logic Unit (ALU)** is the fundamental building block of a Central Processing Unit (CPU). It serves as the primary engine for mathematical calculation and logical decision-making within a computer system.

- **Role in CPU**: The ALU receives data from CPU registers, performs an operation based on an "opcode" (selection signal), and outputs the result back to a register.
- **Arithmetic Operations**: It handles fundamental math such as addition and subtraction.
- **Logical Operations**: It performs bitwise comparisons (AND, OR, XOR) and shifts, which are essential for data processing and conditional branching.

Functional Table

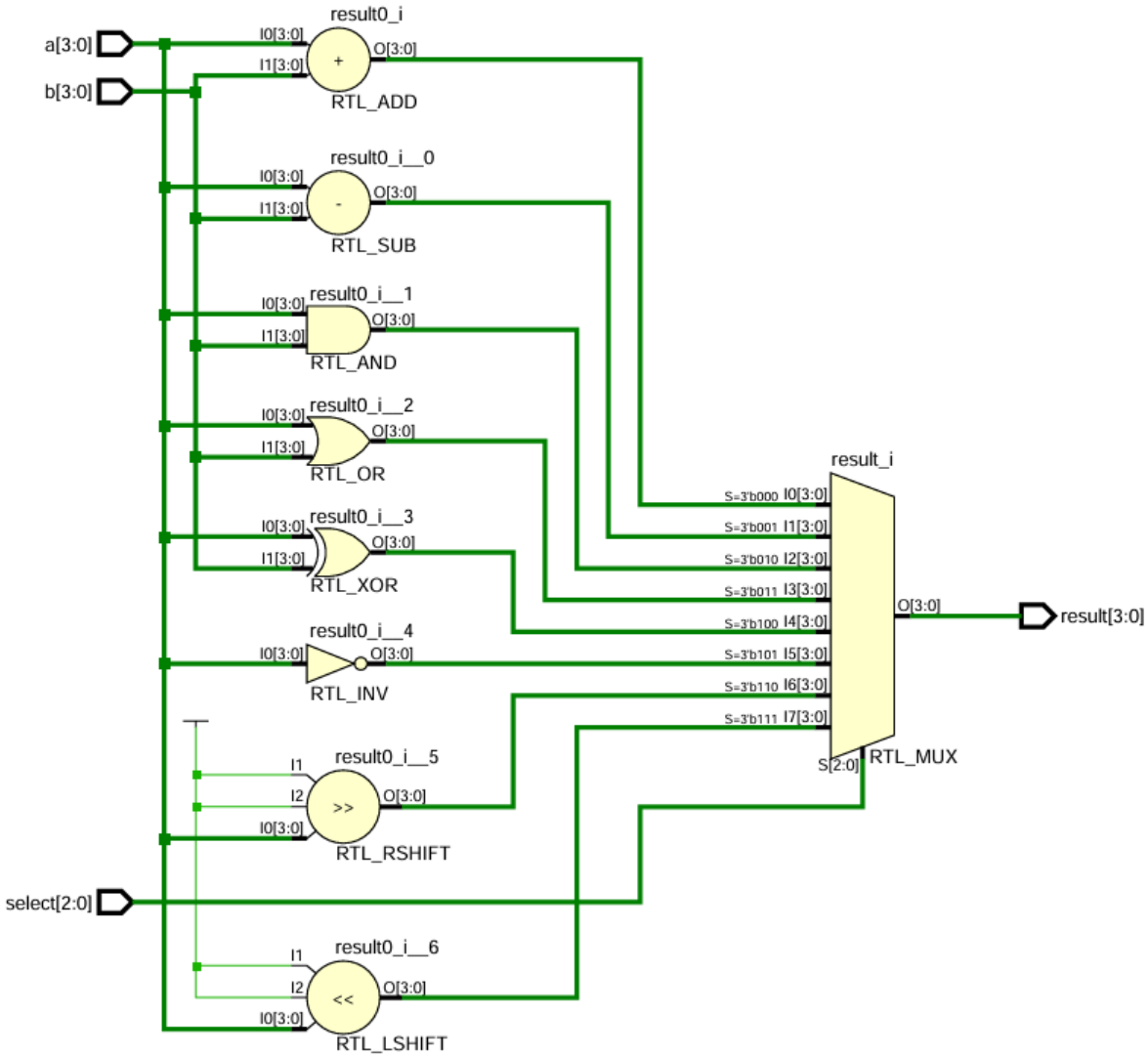
The ALU accepts two 4-bit inputs (a, b) and a 3-bit selector (select).

Select Code	Operation	Verilog Operator	Description
000	**Addition**	a + b	Adds inputs.
001	**Subtraction**	a - b	Subtracts B from A.
010	**AND**	a & b	Bitwise AND.
011	**OR**	a | b	Bitwise OR.
100	**XOR**	a ^ b	Bitwise Exclusive OR.
101	**NOT**	~a	Bitwise Inversion of A.
110	**Shift Right**	a >> 1	Logical Right Shift (Divide by 2).
111	**Shift Left**	a << 1	Logical Left Shift (Multiply by 2).

(Note: If the selection is outside the specified range, the result defaults to 0000).

Hardware Architecture (RTL)

The synthesis results confirm a parallel architecture. As seen in the schematic below, the inputs feed into all operation blocks (RTL_ADD, RTL_SUB, RTL_AND, etc.) simultaneously.
The RTL_MUX at the end ensures stable output selection.



Code

```
module alu_4b(
input [3:0] a, //4 bit input
input [3:0] b, //4bit input
input [2:0] select, //3bit select
output reg [3:0] result //4bit result
    );
    always @(*) begin //@(*) - wave up and runs the block whenever the input changes
    // combinational logic no clock memory
    case(select)
    3'b000: result = a + b;  //Addition
    3'b001: result = a - b;  //Subtraction
    3'b010: result = a & b;  //Bitwise AND
    3'b011: result = a | b;  //Bitwise OR
    3'b100: result = a ^ b;  //Bitwise XOR
    3'b101: result = ~a;     //Bitwise NOT
    3'b110: result = a >> 1; //Right Shift - divides a by 2
    3'b111: result = a << 1; //Left Shift  - multiplies a by 2
    default: result = 4'b0000; //sets the result to 0000 if the input is not in select range
    endcase
  end
endmodule
```

Testbench

```
module tb_alu4b();
reg [3:0] a;
reg [3:0] b;
reg [2:0] select;

wire [3:0] result;

alu_4b ln (a,b,select,result);
initial begin
//to display the result in the tcl
$monitor("Time=%0t | Select=%b | a = %d | b = %d | Result=%b",$time, select, a, b, result);

a = 0; b = 0; select = 0; #10
a = 3; b = 5;
select = 3'b000; #10 //addition 3 + 5 = 8
a = 5; b = 2;
select = 3'b001; #10 //subtraction 5 - 2 = 3
select = 3'b010; #10 //AND 0101 & 0010 = 0000
select = 3'b011; #10 //OR 0101 & 0010 = 0111
select = 3'b100; #10 //XOR 0101 & 0010 = 0111
select = 3'b101; #10 //NOT 0101 = 1010
select = 3'b110; #10 //Right Shift 0101 >> 1 5/2 = 2 - 0010
select = 3'b111; #10 //Left Shift 0101 << 1 5*2 = 10 - 1010
$finish;
end

endmodule
```

Console Log

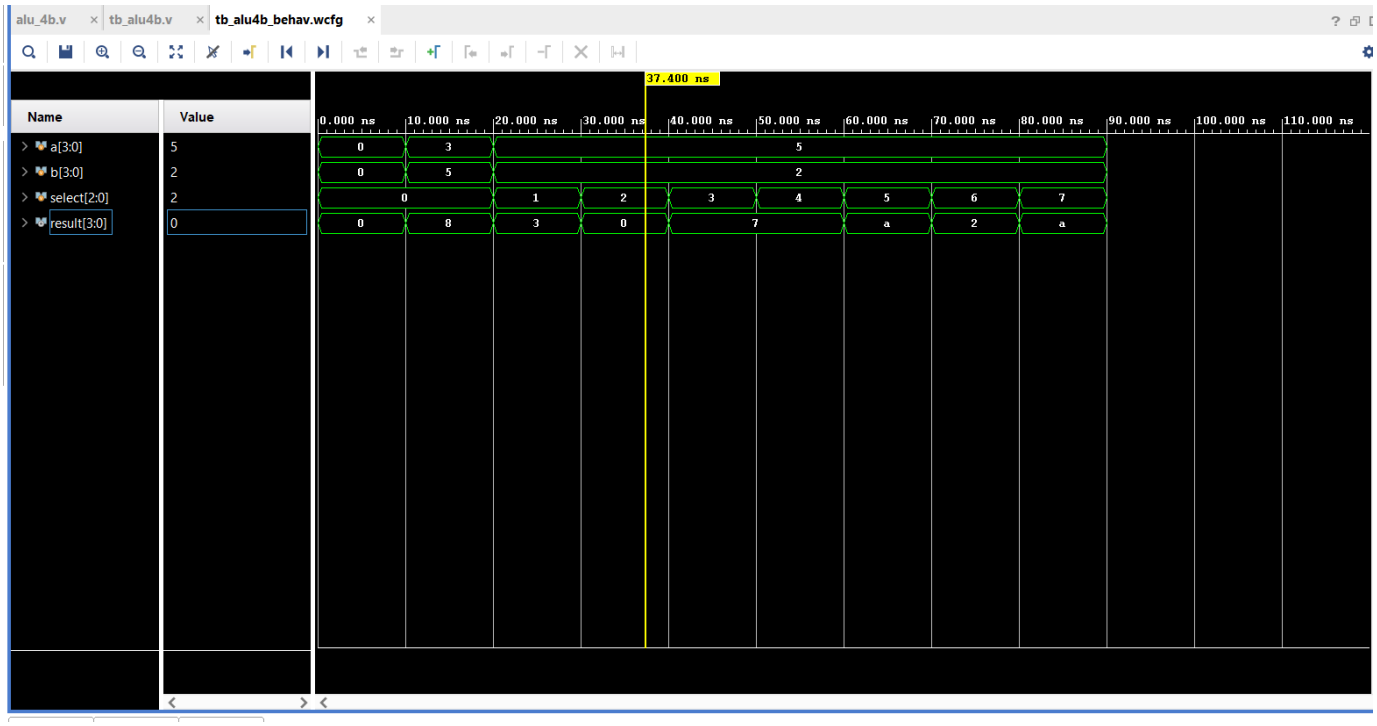
```
# run 1000ns
Time=0 | Select=000 | a =  0 | b =  0 | Result=0000
Time=10000 | Select=000 | a =  3 | b =  5 | Result=1000
Time=20000 | Select=001 | a =  5 | b =  2 | Result=0011
Time=30000 | Select=010 | a =  5 | b =  2 | Result=0000
Time=40000 | Select=011 | a =  5 | b =  2 | Result=0111
Time=50000 | Select=100 | a =  5 | b =  2 | Result=0111
Time=60000 | Select=101 | a =  5 | b =  2 | Result=1010
Time=70000 | Select=110 | a =  5 | b =  2 | Result=0010
Time=80000 | Select=111 | a =  5 | b =  2 | Result=1010
$finish called at time : 90 ns : File "C:/Users/kayil/Projects/Project_ALU/Project_ALU.srcs/sim_1/new/tb_alu4b.v" Line 46
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_alu4b_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:01 ; elapsed = 00:00:08 . Memory (MB): peak = 4783.660 ; gain = 0.000
```

Simulation & Analysis

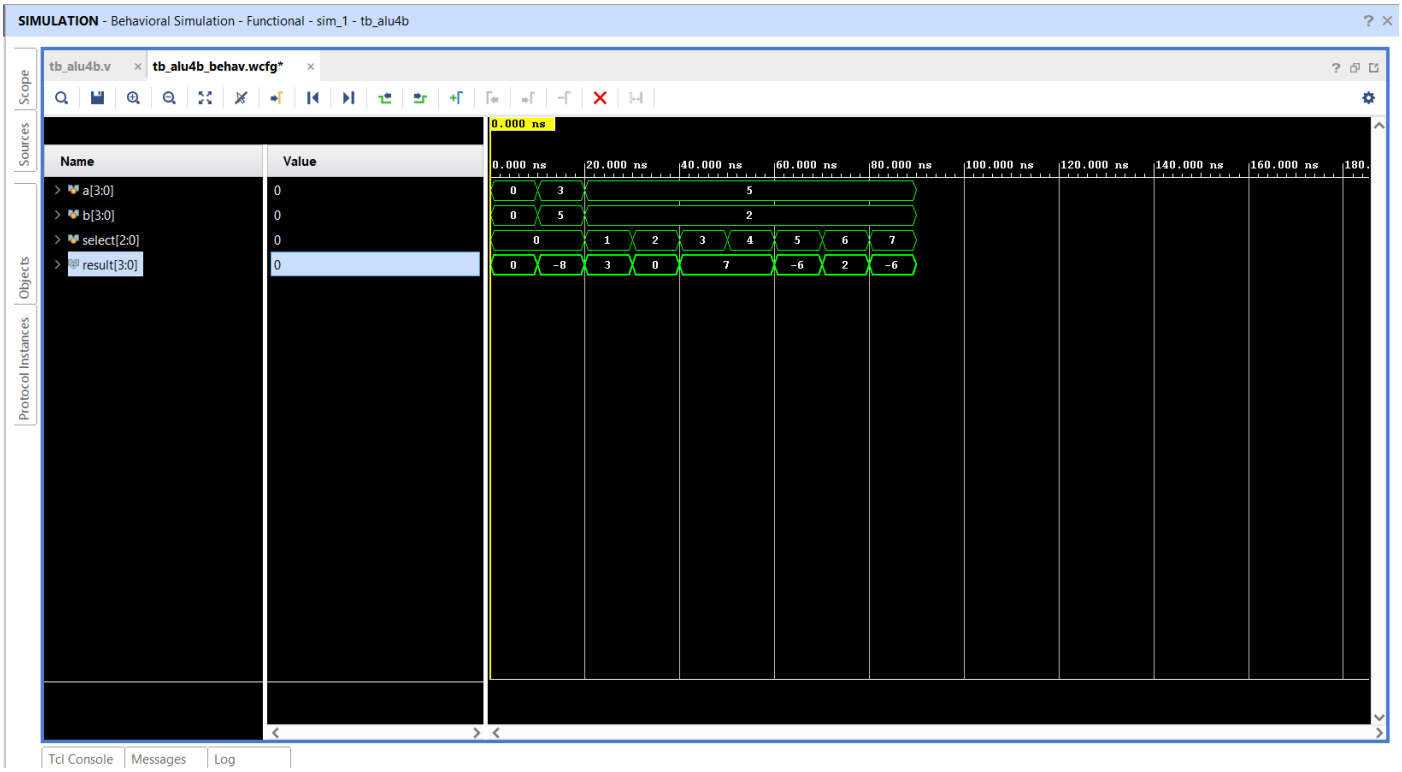
The design was verified using behavioral simulation in Vivado.

Key Observation: Unsigned vs Signed Output In the waveform below, the addition of 3 + 5 results in 8 (Binary 1000).

- **Unsigned View:** 1000 is correctly displayed as **8**.



- **Signed Context:** If interpreted as a 4-bit signed number, 1000 represents **-8** (Overflow), as the magnitude exceeds the 4-bit signed limit (+7).



Constraints

- **Arithmetic Overflow**: Since the output is limited to 4 bits, adding a=3 and b=5 results in 1000. In a 4-bit signed system, the range is **-8 to +7**, meaning the positive result "8" overflows into the negative range.

Room for Improvement

5th Bit Necessity (Signed & Unsigned)

- **The Problem:** In the current 4-bit implementation, adding 3 + 5 results in the binary value 1000.
- **Unsigned Issue:** While the mathematical result is **8**, a 4-bit output can only represent up to 15. If a carry occurs at the MSB (e.g., 8+9), the data is lost because there is no 5th bit to store the carry-out.
- **Signed Issue:** In 2's complement, 1000 is interpreted as **-8**. Because the result exceeded the 4-bit signed maximum of +7, it "overflowed" into the negative range, leading to an incorrect result for positive addition.
- **The Solution:** Expanding the result bus to **5 bits** allows the ALU to capture the carry bit for unsigned math (representing up to 31) and provides the necessary headroom to represent +8 in signed math without flipping the sign bit.