Ryerson University

Department of Electrical, Computer, & Biomedical Engineering
Faculty of Engineering & Architectural Science

Course Title:	COE
Course Number:	608
Semester/Year (e.g.F2016)	w2024

Instructor:	Dr. Hafeez

Assignment/Lab Number:	3
Assignment/Lab Title:	32-bit ALU Design

Submission Date:	02/05/2024
Due Date:	02/06/2024

Student LAST Name	Student FIRST Name	Student Number	Section	Signature*
Mohamed	Lama	501042394	042	L A M A

*By signing above you attest that you have contributed to this written lab report and confirm that all work you have contributed to this lab report is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a "0" on the work, an "F" in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct, which can be found online at: http://www.ryerson.ca/senate/current/pol60.pdf

COE608: Lab 3

1. Lab Objective

In this laboratory, we are implementing and testing a 32-bit Arithmetic Logic Unit (ALU) that can perform six operations. We will also be building a 1-bit, 4-bit, 16-bit and 32-bit adder in order to accomplish this.

The ALU that we are implementing and testing will have two 32-bit data input signals (**a** and **b**) and 3-bit control signals (**op**) that will specify what operation needs to be performed. The output of the ALU is a 32-bit result signal (**Result**), which will depend on the control signals (**op**), and two status flags (**Zero** and **CarryOut**).

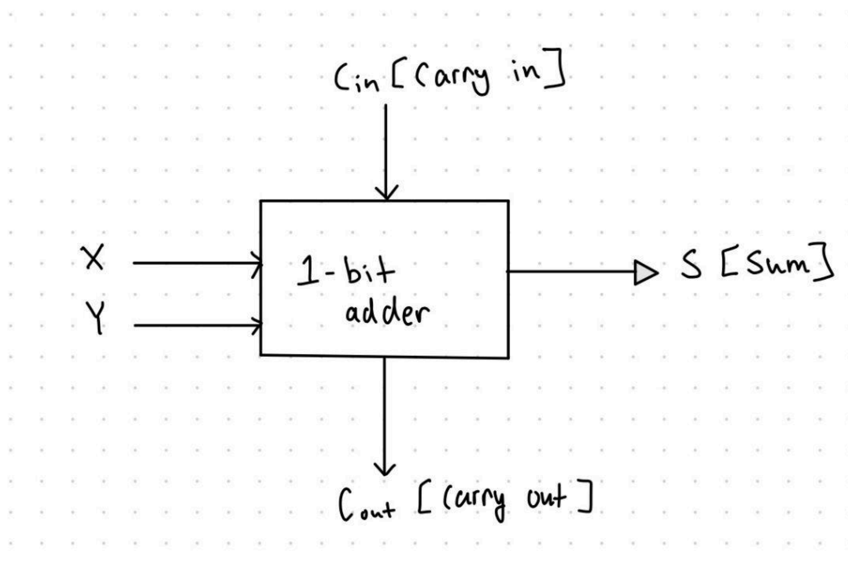


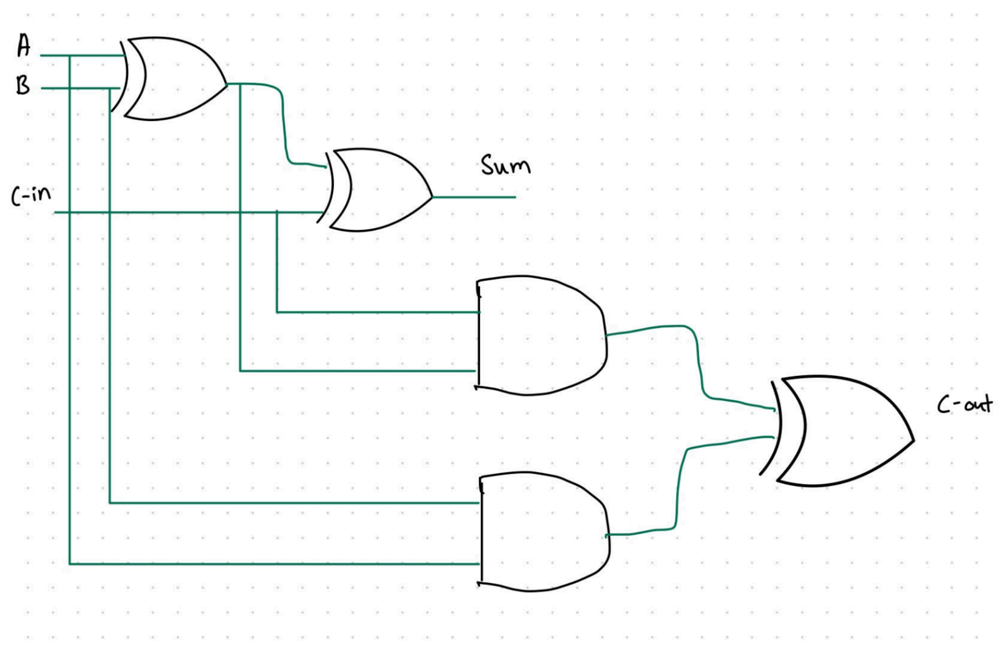
Figure 1: Block Diagram of 1-bit Adder

Input			**Output**	
X	**Y**	**Cin**	**Sum**	**Cout**
0	0	0	0	0

0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 1: Truth Table for full adder

Gate-level Structure of a Full Adder Module:



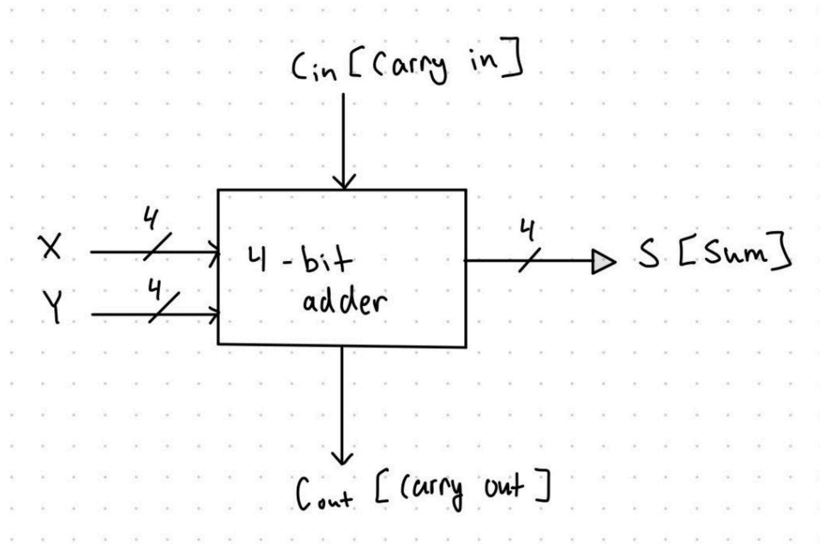


Figure 2: Block diagram for 4-bit Adder

Connecting full adders to form a 4-bit adder:

- To create a 4-bit adder using 4 full adders, we need to connect them in a cascading fashion.
- Each full adder takes two input bits (A and B), a carry input (Cin), and produces a sum output (S) and a carry output (Cout).
- We need to connect the A and B inputs of the least significant full adder to the two 4-bit binary numbers we want to add.
- Next, we connect the Cin of the least significant full adder to the ground since there is no carry-in initially
- After that, we connect the Sum (S) output of the least significant full adder to the least significant bit of the result. We then repeat these steps for the remaining full adders
- This is illustrated in the diagram below:

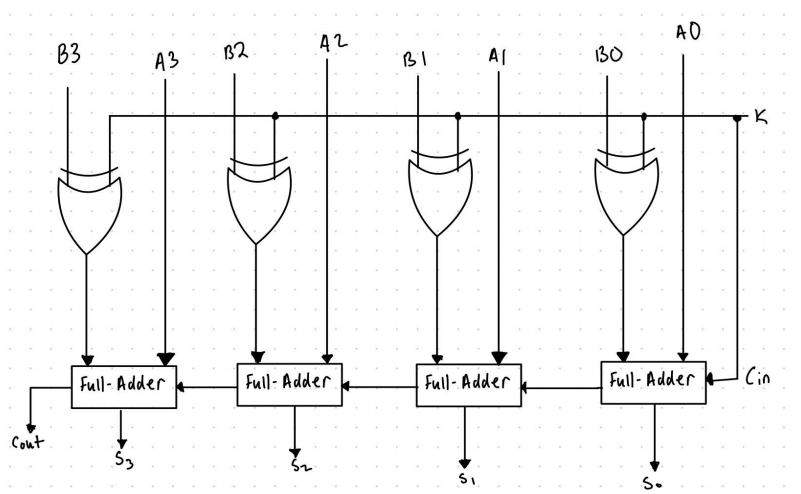


Figure 3: Cascading system of 4-bit adder from 4 full adders

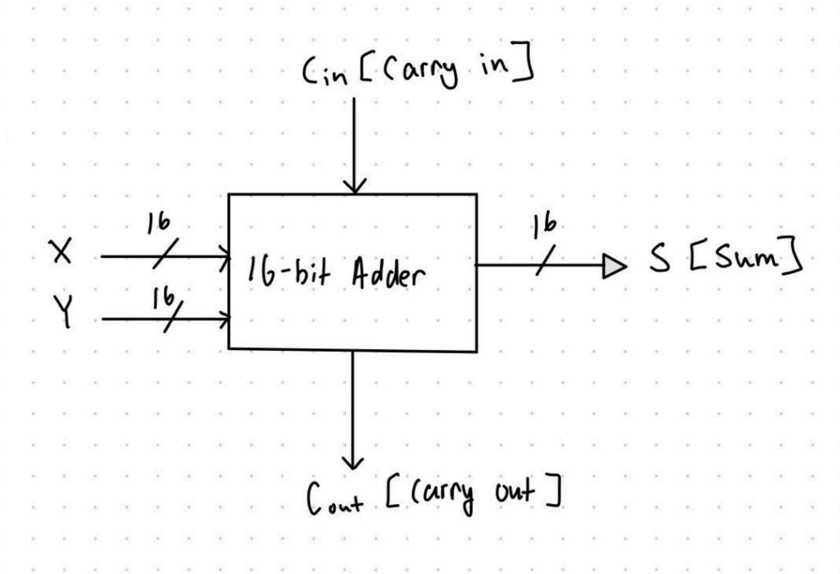


Figure 4: Block diagram for 16-bit Adder

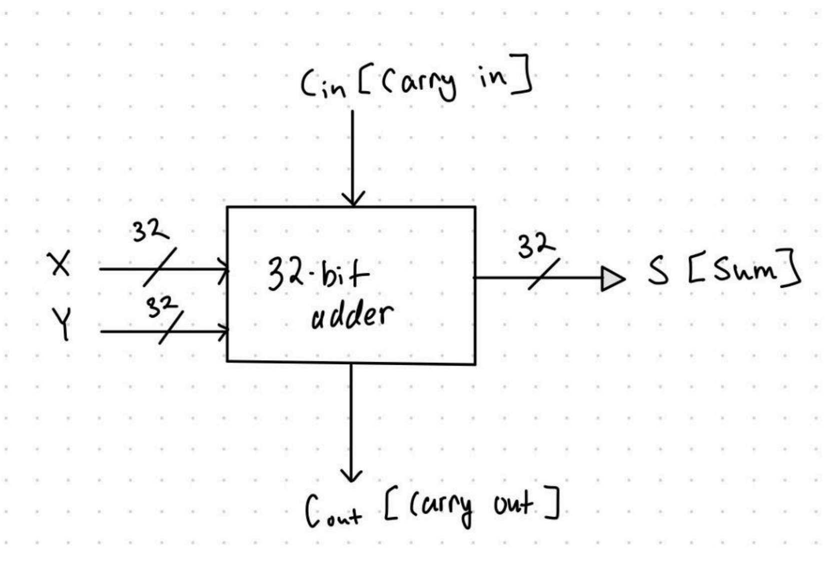


Figure 5: 32-bit Adder

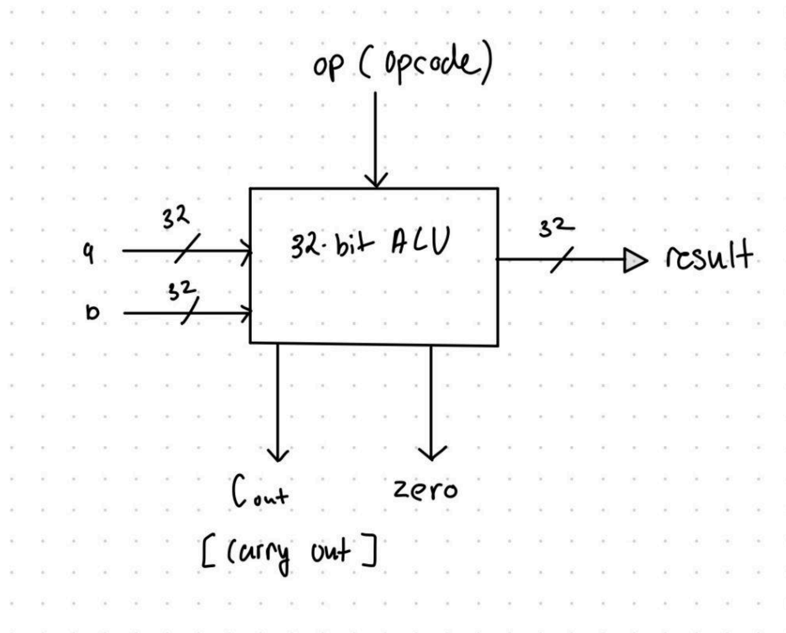


Figure 6: 32-bit ALU

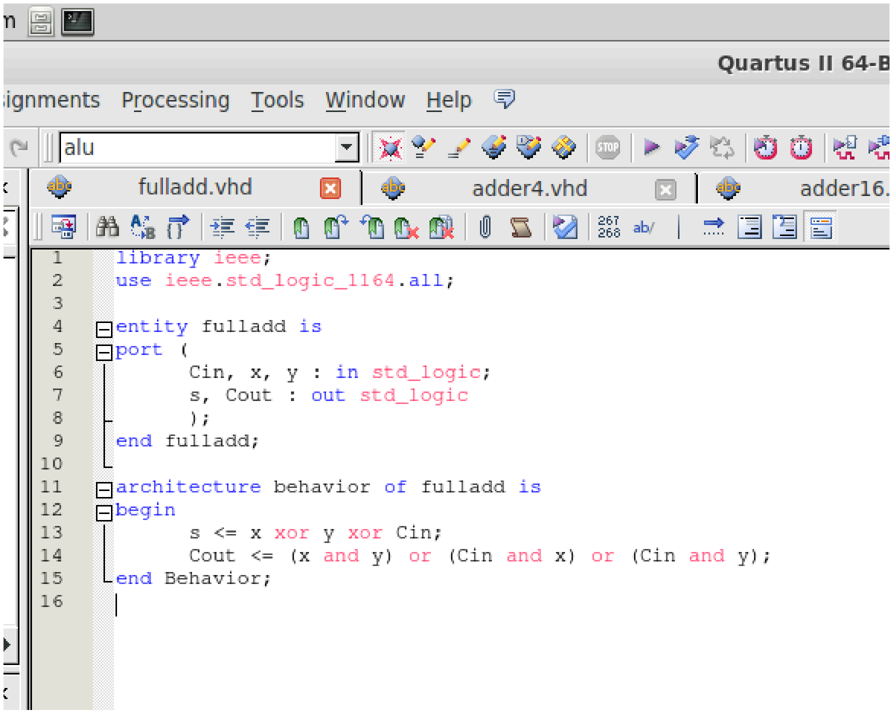
The operations that the ALU should be able to perform are as follows on this truth table:

Operation Name	ALU-Op		Operation Performed
	Neg/TSel	ALU-Select	
AND (Logical)	0	00	Result ⇐ a AND b
OR (Logical)	0	01	Result <= a OR b
ADD	0	10	Result <= a + b
SUB	1	10	Result <= a - b
ROL	1	00	Result <= a << 1
ROR	1	01	Result <= a >> 1

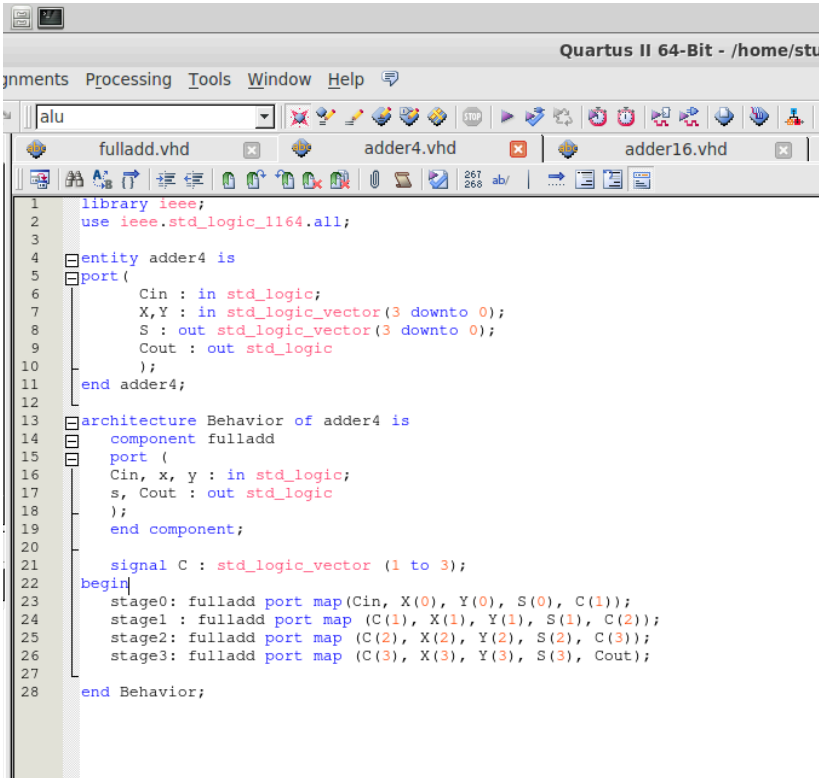
Table 2: Truth Table of 32-bit ALU

2. Experiment Details

1-bit Adder:

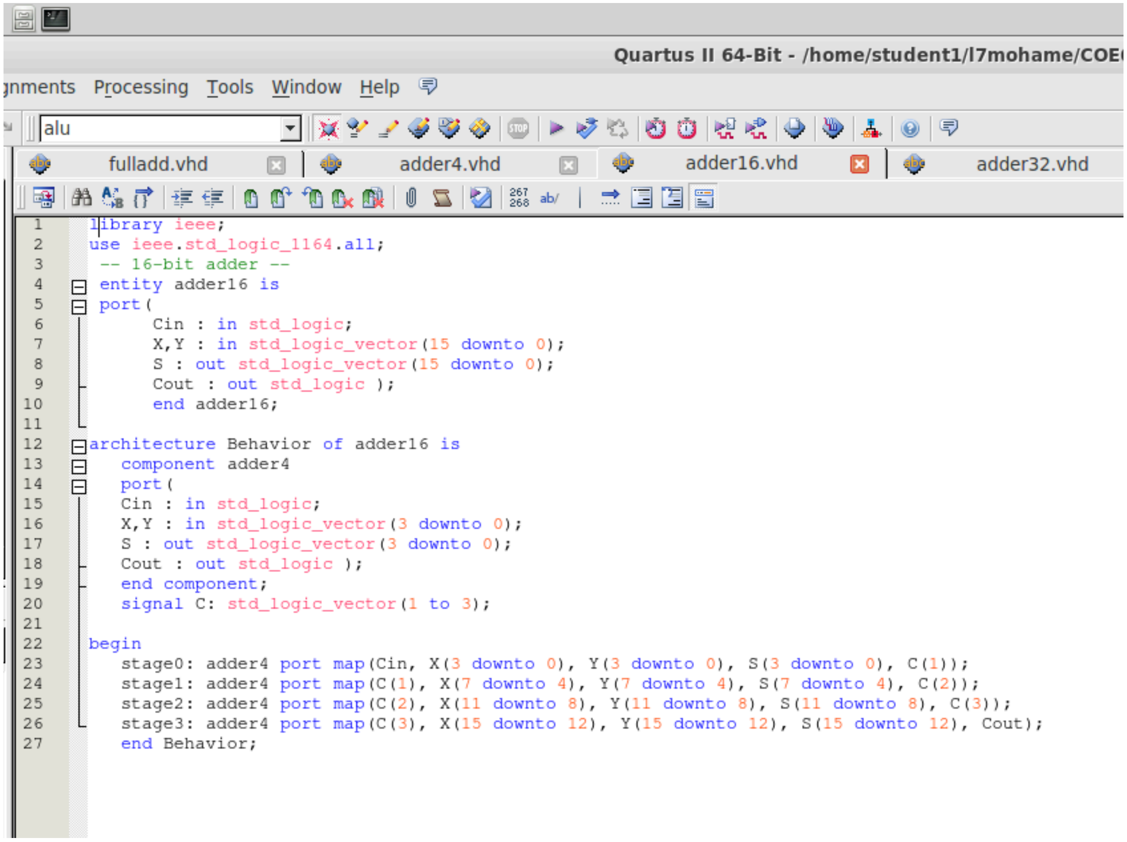


4-bit Adder:




```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity adder4 is
port(
        Cin : in std_logic;
        X,Y : in std_logic_vector(3 downto 0);
        S : out std_logic_vector(3 downto 0);
        Cout : out std_logic
        );
end adder4;

architecture Behavior of adder4 is
   component fulladd
   port (
   Cin, x, y : in std_logic;
   s, Cout : out std_logic
   );
   end component;

   signal C : std_logic_vector (1 to 3);
begin
   stage0: fulladd port map(Cin, X(0), Y(0), S(0), C(1));
   stage1 : fulladd port map (C(1), X(1), Y(1), S(1), C(2));
   stage2: fulladd port map (C(2), X(2), Y(2), S(2), C(3));
   stage3: fulladd port map (C(3), X(3), Y(3), S(3), Cout);

end Behavior;
```

16-bit Adder:




```vhdl
library ieee;
use ieee.std_logic_1164.all;
 -- 16-bit adder --
 entity adder16 is
 port(
        Cin : in std_logic;
        X,Y : in std_logic_vector(15 downto 0);
        S : out std_logic_vector(15 downto 0);
        Cout : out std_logic );
        end adder16;

architecture Behavior of adder16 is
   component adder4
   port(
   Cin : in std_logic;
   X,Y : in std_logic_vector(3 downto 0);
   S : out std_logic_vector(3 downto 0);
   Cout : out std_logic );
   end component;
   signal C: std_logic_vector(1 to 3);

begin
   stage0: adder4 port map(Cin, X(3 downto 0), Y(3 downto 0), S(3 downto 0), C(1));
   stage1: adder4 port map(C(1), X(7 downto 4), Y(7 downto 4), S(7 downto 4), C(2));
   stage2: adder4 port map(C(2), X(11 downto 8), Y(11 downto 8), S(11 downto 8), C(3));
   stage3: adder4 port map(C(3), X(15 downto 12), Y(15 downto 12), S(15 downto 12), Cout);
   end Behavior;
```

32-bit Adder:

```vhdl
library ieee;
use ieee.std_logic_1164.all;
 -- 32-bit adder --
entity adder32 is
port (
   Cin  : in std_logic;
   X,Y  : in std_logic_vector(31 downto 0);
   S  : out std_logic_vector(31 downto 0);
   Cout : out std_logic
   );
end adder32;

architecture Behavior of adder32 is
   component adder16
   port(
      Cin : in std_logic;
      X,Y : in std_logic_vector(15 downto 0);
      S : out std_logic_vector(15 downto 0);
      Cout : out std_logic );
   end component;

   signal C : std_logic;
begin
   stage0: adder16 port map(Cin, X(15 downto 0), Y(15 downto 0), S(15 downto 0), C);
   stage1: adder16 port map(C, X(31 downto 16), Y(31 downto 16), S(31 downto 16), Cout);
end Behavior;
```

ALU:

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity alu is
port (
    a : in std_logic_vector(31 downto 0);
    b : in std_logic_vector(31 downto 0);
    op : in std_logic_vector(2 downto 0);
    result : out std_logic_vector(31 downto 0);
    zero : out std_logic;
    cout : out std_logic );
end alu;

architecture Behavior of alu is
    component adder32
        port (
            Cin : in std_logic;
            X,Y : in std_logic_vector(31 downto 0);
            S : out std_logic_vector(31 downto 0);
            Cout : out std_logic
            );
        end component;
        signal result_s: std_logic_vector(31 downto 0):=(others=>'0');
        signal result_add: std_logic_vector(31 downto 0):=(others=>'0');
```

```vhdl
        signal result_sub: std_logic_vector(31 downto 0):=(others=>'0');
        signal cout_s: std_logic := '0';
        signal cout_add: std_logic := '0';
        signal cout_sub: std_logic := '0';
        signal zero_s: std_logic := '0';

begin
        add0 : adder32 port map(op(2), a, b, result_add, cout_add);
        sub0 : adder32 port map(op(2), a, not b, result_sub, cout_sub);

    process (a,b,op)
    begin
        case(op) is
            when "000" => -- "000" a and b
                result_s <= a and b;
                cout_s <= '0';
            when "001" => -- "001" a or b
                result_s <= a or b;
                cout_s <= '0';
            when "010" => -- "010" a + b
                result_s <= result_add;
                cout_s <= cout_add;
            when "110" => -- "110" a - b
                result_s <= result_sub;
                cout_s <= cout_sub;
            when "100" => -- "100" a << 1 "ROL" "sll
                result_s <= a(30 downto 0) & '0';
                cout_s <= a(31);
            when "101" => -- "101" a >> 1 "ROR"
                result_s <= '0' & a(31 downto 1);
                cout_s <= '0';
            when others => -- any other value of opcode defaults to passi
                result_s <= a;
                cout_s <= '0';
        end case;

        case (result_s) is --case statement is used to determine the value
            when(others => '0') =>
                zero_s <= '1';
            when others =>
                zero_s <= '0';
        end case;
    end process;

    result <= result_s;
    cout <= cout_s;
    zero <= zero_s;
    end Behavior;
```

- This is the VHDL code that I used to implement the ALU.
- As can be seen, the ALU can perform addition, subtraction, AND, OR, left shift and right shift operations on 32-bit vectors (‘a’ and ‘b’).
- The ALU has inputs (‘a’, ‘b’, ‘op’) and outputs(‘result’, zero, cout)
    - Zero is a flag indication if the result is zero
    - Cout is the carry-out flag

Component declaration:

- The ALU uses a 32-bit adder as a component for both addition and subtraction operations.
- It carriers a carry input (‘Cin’) and two 32-bit operands (‘X’ and ‘Y’) and produces a sum ‘S’ and a carry output (‘Cout’)

Signal declarations:

- ‘Result_s’, ‘result_add’ and ‘result_sub’ hold the intermediate results for the ALU operations
- ‘Cout_s’, ‘cout_add’ and ‘cout_sub’ hold the carry-out flags for the ALU operations.
- ‘Zero_s’ holds the result of the zero flag

Instantiation of 32-bit Adder Components:

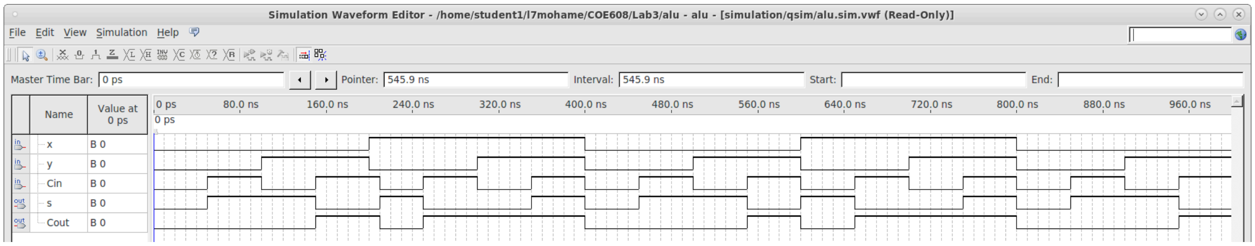
- Two instances of the 32-bit adder component are instantiated:
    - ‘add0’ for addition and ‘sub0’ for subtraction
    - For subtraction, we use ‘not b’ since we are performing two’s complement.

Main Process:

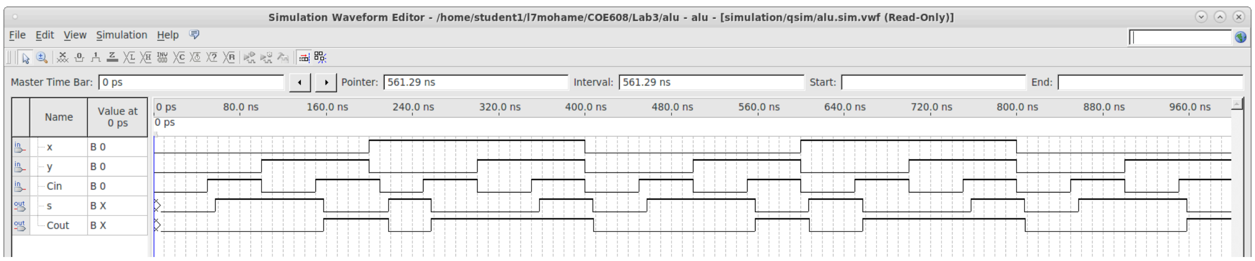
- The main process is where we select the appropriate operation based on the ‘op’ input.
- Depending on the opcode, it performs the appropriate operation.
- The result of each operation is held in ‘result_s’ and the corresponding carry out ‘cout_s’ is set.
- Another case statement is then used to determine the value of the zero flag ‘zero_s’ If all bits in the ‘result_s’ are ‘0’, then the zero flag is set to ‘1’. Otherwise it’ll be ‘0’.

3. Results:

1-bit Adder:

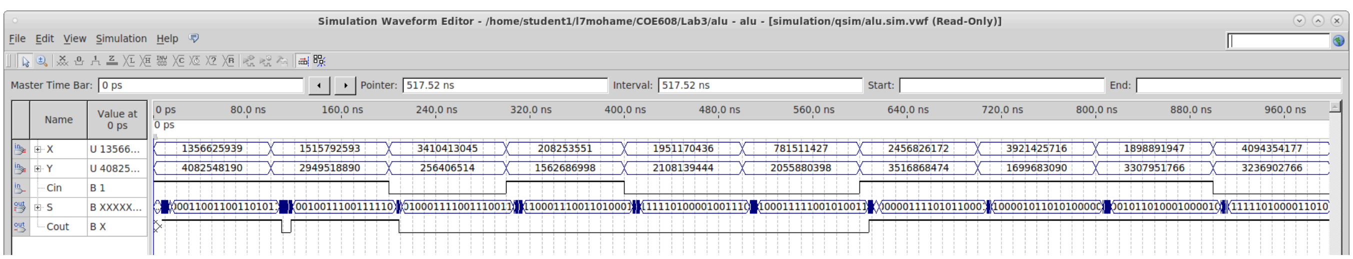


Waveform 1: Full-adder Functional waveform



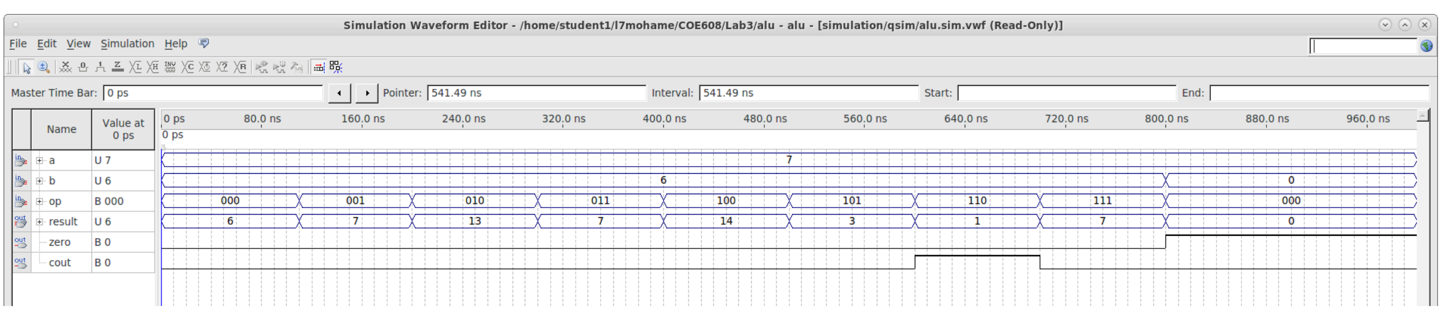
Waveform 2: Full-adder Timing waveform

32-bit Adder:



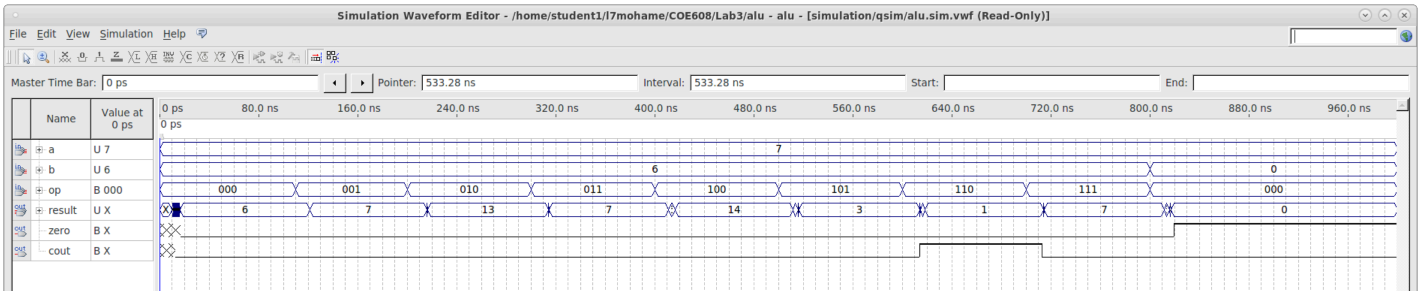
Waveform 3: 32-bit Adder Timing Waveform

Functional Simulation of ALU:



Waveform 4: Functional Simulation of ALU

Timing Simulation of ALU:



Waveform 5: Timing simulation of ALU

4. Discussion:

- As can be seen, the ALU performed as expected.
- When a = 7 and b = 6, all the operation results are accurate.
- we can also see that during the subtraction operation, ‘110’, Cout is 1, which is the correct result as a burrow was required. We can also see that it remained 0 for the rest of the operations, which is also the correct result. We can also see that the zero output was only 1 when the value of the result was 0, which is also the correct output.

I will now be outlining the worst-case timing characteristics of a 32-bit Arithmetic Logic Unit (ALU) Based on functional and timing simulations. The ALU performs basic operations such as addition, subtraction, logical operations and shift operations.

In terms of Addition and Subtraction, there are two types of delays that can occur; propagation delay and carry propagation delay. Propagation delay refers to the time that it takes for a signal to travel from the input of a circuit to its output. It is the time delay experienced by a signal as it passes through various gates within a circuit. It is mostly influenced by the physical characteristics of the circuit components. Carry propagation delay specifically relates to arithmetic operations, particularly addition in multi-bit adders, like the one we are using in this lab. When adding two binary numbers, each bit addition generates a sum bit and a carry-out bit, which propagates to the next higher-order bit. Carry propagation delay is the time it takes for the carry-out from a lower-bit addition to affect the sum in the next higher-order bit addition. In a multi-bit adder, this delay accumulates as the carry signal ripples through each stage. The propagation delay is expected to be just a few nanoseconds, and the carry propagation delay is between 5 and 15 nanoseconds in our ALU. Overall, we can see approximately In terms of logical operations, such as AND and OR, there is also a propagation delay that is just a few nanoseconds, and also a gate delay of 2-8 nanoseconds. Gate delays refer to the time it takes for the output of a logic gate to respond to a change in its inputs.

In terms of shift operations, which are left and right, there is a shift operation delay of 2-6 nanoseconds. This delay refers to the time that it takes to move the bits of a binary number to the left or right.