**32-bit ALU Design**

**CENG 3151**

Submitted by

Michael Lanford

1816337

Computer Engineering

University of Houston-Clear Lake

Houston, Texas 77058

April 12, 2023

# Table of Contents

**1** [**Abstract 3**](#_Toc126060766)

[**2** **Introduction** 3](#_Toc126060767)

**3 Requirements……………………………………………………………………….……......3**

**4 Prelab………………………………………………………………………………..……….3**

**5 Implementation……………………………………………………….…………………......4**

5.1 Design Code / Design Diagrams**……………………………………………………………4**

5.2 Schematics**……………………………………………………………………………….....6**

5.3 Testbench**…………………………………………………………………………………....6**

5.4 Waveform / Results**………………………………………………………………………...9**

[**6** **Conclusion** 10](#_Toc126060768)

**Abstract**

An ALU is a circuit that can perform arithmetic and logic operations such as: incrementing a number, decrementing a number, and shifting bits left or right. It can also perform addition, subtraction, multiplication, and division along with the logical operations NOT, AND, OR, and XOR.

# Introduction

For this project, we will be using Xilinx Vivado to build and test a 32-bit ALU that will accept some input and produce some output.

1. **Requirements**

Design a 32-bit ALU that can perform arithmetic and logic operations with 4 inputs: Reg\_A, Reg\_B, Op\_Sel, and Carry\_In. This circuit will output 2 values: ALU\_Out and Carry\_Out. The figure of this circuit can be seen below:

Diagram

Description automatically generated

**Figure 1:** Diagram for the 32-bit ALU to be designed.

1. **Prelab**

For this prelab, we were required to study the concept of an ALU.

An ALU is a circuit that can perform arithmetic and logic operations such as: incrementing a number, decrementing a number, and shifting bits left or right. It can also perform addition, subtraction, multiplication, and division along with the logical operations NOT, AND, OR, and XOR.

1. **Implementation**

The first step of implementation was to create a new project in Xilinx Vivado. After setting up the project with the correct options, we added a design source file for the 32-bit ALU and added the necessary inputs and outputs to it. We then coded the temporary signals and the process that would do all the ALU operations. After that, we created the simulation file and added in the component instantiation, interface signal declarations, instance port map, and the test cases to it then tested the waveform.

**4.1 Design Code / Design Diagrams**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.std\_logic\_unsigned.ALL;

use IEEE.NUMERIC\_STD.ALL;

--Input and Output Declarations

entity Lab10Design is

Port ( Reg\_A : in STD\_LOGIC\_VECTOR (31 downto 0);

Reg\_B : in STD\_LOGIC\_VECTOR (31 downto 0);

OpSel : in STD\_LOGIC\_VECTOR (3 downto 0);

Carry\_In : in STD\_LOGIC;

Carry\_Out : out STD\_LOGIC;

ALU\_Out : out STD\_LOGIC\_VECTOR (31 downto 0));

end Lab10Design;

architecture Behavioral of Lab10Design is

signal temp: std\_logic\_vector (32 downto 0); --Signal declarations

signal temp\_a: std\_logic\_vector (32 downto 0);

signal temp\_b: std\_logic\_vector (32 downto 0);

begin

temp\_a <= '0' & Reg\_A; --Add 0 as MSB to Reg\_A and Reg\_B extended by 1 bit

temp\_b <= '0' & Reg\_B;

process(Reg\_A, Reg\_B, OpSel, temp, Carry\_In) is begin --Process for ALU

case OpSel is

when "0000" => temp <= temp\_A; --Output is Reg\_A

when "0001" => temp <= std\_logic\_vector((unsigned(temp\_a)) + "1"); --Increment A

when "0010" => temp <= std\_logic\_vector((unsigned(temp\_a)) - "1"); --Decrement A

when "0011" => temp <= (temp\_a + temp\_b + Carry\_in); --Add A and B with carry

when "0100" => temp <= (NOT temp\_a); --Logical NOT of A

when "0101" => temp <= (temp\_a AND temp\_b); --Logical AND for A and B

when "0110" => temp <= (temp\_a OR temp\_b); --Logical OR for A and B

when "0111" => temp <= (temp\_a XOR temp\_b); --Logical XOR for A and B

when "10--" => temp <= std\_logic\_vector(shift\_left(unsigned(temp\_a), to\_integer(unsigned(temp\_b)))); --Logical left shift on A by bits B

when "11--" => temp <= std\_logic\_vector(shift\_right(unsigned(temp\_a), to\_integer(unsigned(temp\_b)))); --Logical right shift on A by bits B

when others => temp <= temp\_a; --Any other operation signals, just output A

end case;

end process;

ALU\_Out <= temp(31 downto 0);--Put the output from temp into ALU\_Output

Carry\_Out <= temp(32);--Indicates overflow

end Behavioral;

**4.2 Schematics**

**Diagram, schematic

Description automatically generated**

**Figure 2:** 32-bit ALU.

**4.3 Testbench**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Lab10Sim is

-- Port ( );

end Lab10Sim;

architecture Behavioral of Lab10Sim is

component Lab10Design --Component Instantiation

Port ( Reg\_A : in STD\_LOGIC\_VECTOR (31 downto 0);

Reg\_B : in STD\_LOGIC\_VECTOR (31 downto 0);

OpSel : in STD\_LOGIC\_VECTOR (3 downto 0);

Carry\_In : in STD\_LOGIC;

Carry\_Out : out STD\_LOGIC;

ALU\_Out : out STD\_LOGIC\_VECTOR (31 downto 0));

end Component;

signal Reg\_A: std\_logic\_vector(31 downto 0); --Signal Declarations

signal Reg\_B: std\_logic\_vector(31 downto 0);

signal OpSel: std\_logic\_vector(3 downto 0);

signal Carry\_In: std\_logic := '0';

signal Carry\_Out: std\_logic := '0';

signal ALU\_Out: std\_logic\_vector(31 downto 0);

begin

uut: Lab10Design port map(Reg\_A => Reg\_A, Reg\_B => Reg\_B, OpSel => OpSel, Carry\_In => Carry\_In, Carry\_Out => Carry\_Out, ALU\_Out => ALU\_Out); --Port map

stim\_proc: process --Test case process

begin

OpSel <= "0000"; --Test Case 1: Output Reg\_A Output should just be all f

Reg\_A <= x"ffffffff";

wait for 20 ns;

OpSel <= "0001"; --Test Case 2: Increment Reg\_A by 1 Output should be just 1

Reg\_A <= x"00000000";

wait for 20 ns;

OpSel <= "0010"; --Test case 3: Decrement Reg\_A by 1 Output should be 1

Reg\_A <= x"00000002";

wait for 20 ns;

OpSel <= "0011"; --Test case 4: Add Reg\_A, Reg\_B, and Carry = 1 Output should be 5

Reg\_A <= x"0000\_0002";

Reg\_B <= x"0000\_0002";

Carry\_in <= '1';

wait for 20 ns;

OpSel <= "0011"; --Test Case 5: Add Reg\_A, Reg\_B, and Carry = 0 with overflow ALU\_Out should be 0, but Carry\_Out should show 1

Reg\_A <= x"ffff\_ffff";

Reg\_B <= x"0000\_0001";

Carry\_In <= '0';

wait for 20 ns;

OpSel <= "0100"; --Test Case 6: NOT Reg\_a Output should be all f

Reg\_A <= x"0000\_0000";

wait for 20 ns;

OpSel <= "0101"; --Test Case 7: Reg\_A AND Reg\_B Output should be 1's in lsb

Reg\_A <= x"0000\_1111";

Reg\_B <= x"1111\_1111";

wait for 20ns;

OpSel <= "0110"; --Test Case 8: Reg\_A OR Reg\_B Output should be all 1's

Reg\_A <= x"1010\_1010";

Reg\_B <= x"0101\_0101";

wait for 20 ns;

OpSel <= "0111"; --Test Case 9: Reg\_A XOR Reg\_B Output should be leading 1's

Reg\_A <= x"0000\_0000";

Reg\_B <= x"1111\_0000";

OpSel <= "1011"; --Test Case 10: Shift Reg\_A left Output should be 2

Reg\_A <= x"0000\_0001";

Reg\_B <= x"0000\_0001";

wait for 20ns;

OpSel <= "1100"; --Test Case 11: Shift Reg\_A right Output should be 8

Reg\_A <= x"0000\_0010";

Reg\_B <= x"0000\_0001";

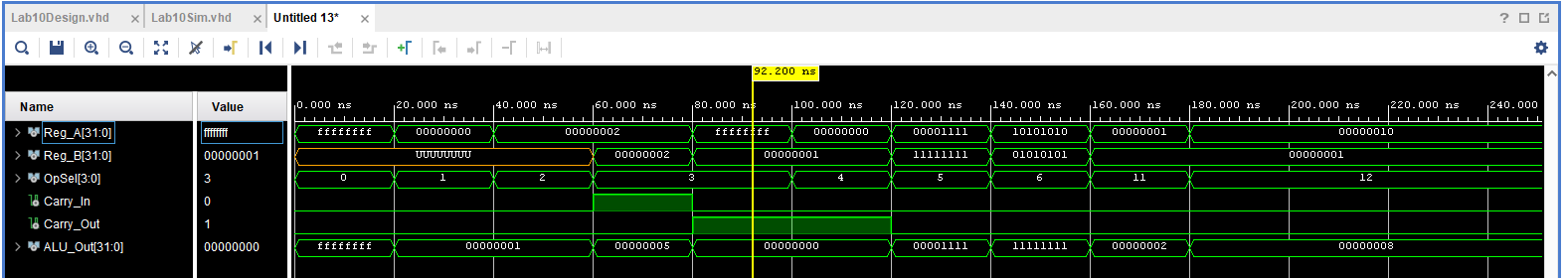
wait;

end process;

end Behavioral;

**4.4 Waveform / Results**

The waveform below shows that the two programs we made above in the Testbench and Design Code / Design Diagrams sections were able to produce correct results for each of our inputs that we created. For example, at 40ns on Figure 3 when Op\_Sel was 2, Reg\_A was decremented by 1, which is the correct output we should have gotten. Also, at 120ns when Op\_Sel was 5, the AND logical operation was performed on Reg\_A and Reg\_B and produced the correct output of 00001111.



**Figure 3:** 32-bit ALU.

# Conclusion

In this lab, we were able to successfully code a 32-bit ALU with 4 inputs and 2 outputs by using the little code snippets given to us in our prelab as a base for our code along with the previous labs we completed throughout the semester. These programs were made to be able to simulate how an ALU works inside a CPU, which can be seen in the waveform due to the correct output being produced for what we input.