University of Houston - Clear Lake [Summer 2023]

CENG 3151.01: Lab for Computer Architecture

Lab 10 (Final Project): 32-bit ALU Design

Submitted by: Brandon E Ramirez

Date: 7/20/2023

Due: Thursday, July 27th, 2023

STUDENT ID: 1952649

COMPUTER ENGINEERING
UNIVERSITY OF HOUSTON – CLEAR LAKE
HOUSTON, TEXAS 77058

Contents

List of Figures

Listings

1 Abstract

The Arithmetic Logic Unit (ALU) is a crucial component of a computer's central processing unit (CPU). It is responsible for performing arithmetic and logical operations necessary for processing data and executing instructions. The ALU's primary purpose is to carry out calculations and make decisions based on the data it receives from the computer's registers and memory. Some functions of the ALU are the following:

1. Arithmetic Operations: The ALU can perform basic arithmetic calculations such as addition, subtraction, multiplication, and division.
2. Logical Operations: The ALU can execute logical operations such as AND, OR, NOT, and XOR. These operations are vital for making decisions and determining the flow of a program based on Boolean conditions.
3. Data Comparison: The ALU can compare two pieces of data and set flags based on the result of the comparison. These flags are used to indicate whether data is equal, not equal, greater than, or less than, enabling conditional branching and control flow in programs.
4. Bit Shift Operations: The ALU can shift the bits of binary data left or right, this can be done via "arithmetic" or "logical" shifting.
5. Increment and Decrement: The ALU can increment or decrement the value of a data element by 'one' (usually done one bit at a time).

The ALU serves as the "computational engine" of a CPU, responsible for carrying out essential arithmetic, logical, and comparison operations necessary to execute programs and process data.

2 Introduction/Goals

The purpose of this lab is to design a 32-bit ALU capable of performing logical, arithmetic, shifting, loading, and storing operations with 2x 32-bit inputs. We need to be able to select the operation(s) using a control input and enable a "carry-in" input derived from preceding computations/operations. The ALU needs to output a 32-bit result and a carry-out. We are going to replicate this behavior with the hardware description language "VHDL" and Vivado to define, test, and generate time diagrams + schematics. The purpose of this lab is to design a 32-bit ALU with VHDL and Vivado hardware design software.

3 Requirements

Here are the requirements: Design a 32-bit ALU which can perform arithmetic and logic operations. The design must be able to perform the following:

1. The design has two 32-bit inputs, input A and input B, they are unsigned binary numbers
2. Addition, increment, decrement and transfer (arithmetic operations)
3. AND, OR, NOT, XOR (logical operations)

4. Right shift, left shift (shift operations)
5. The design must have 4-bit select line called Operation Select, which would direct the unit as to which operation to perform
6. The unit has a Carry-in and also a Carry-out.

The block diagram of this ALU is as shown below:

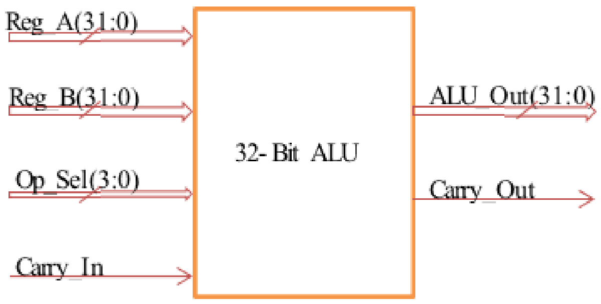
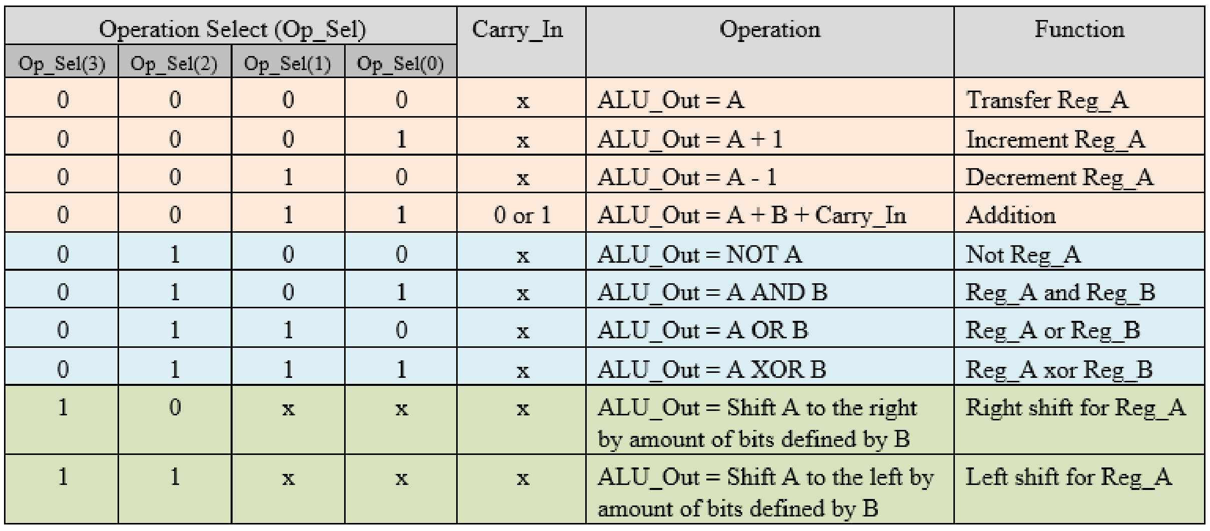


Figure 1: Black-box diagram

The interface of this design is as below:

Listing 1: I/O stream

```
        entity ALU_32Bits_Design is   port(
        Reg_A     :  in std_logic_vector(31 downto 0);
        Reg_B     :  in std_logic_vector(31 downto 0);
        Op_Sel    :  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 ALU_32Bits_Design;
```



Operation Select (Op_Sel)				Carry_In	Operation	Function
Op_Sel(3)	Op_Sel(2)	Op_Sel(1)	Op_Sel(0)			
0	0	0	0	x	ALU_Out = A	Transfer Reg_A
0	0	0	1	x	ALU_Out = A + 1	Increment Reg_A
0	0	1	0	x	ALU_Out = A - 1	Decrement Reg_A
0	0	1	1	0 or 1	ALU_Out = A + B + Carry_In	Addition
0	1	0	0	x	ALU_Out = NOT A	Not Reg_A
0	1	0	1	x	ALU_Out = A AND B	Reg_A and Reg_B
0	1	1	0	x	ALU_Out = A OR B	Reg_A or Reg_B
0	1	1	1	x	ALU_Out = A XOR B	Reg_A xor Reg_B
1	0	x	x	x	ALU_Out = Shift A to the right by amount of bits defined by B	Right shift for Reg_A
1	1	x	x	x	ALU_Out = Shift A to the left by amount of bits defined by B	Left shift for Reg_A

Figure 2: 32-bit ALU operations table

4 Prelab

No Pre-Lab required for this lab.

5 Report Write-up/Implementation

We will use one src/design file and one test-bench file. Our code will use Reg-A, Reg-B, Op-Sel, Carry-In, Carry-Out, and ALU-Out to satisfy our design I/O requirements. We will use a series of if/else-statements and a single process to achieve this logic within our source file. First, we will create a process which takes Reg-A & Reg-B as parameters, we do this to routinely check the contents of the registers. The first instruction we will implement is transfer, here we simply send input stream found in ”Reg-A” to ”ALU-Out”. We implement this logic by using an if-statement which checks the contents of ”Op-Sel” (Operation select); iff it equals ”0000”, then we know we will evaluate this particular operation. The same logic applies to the other operations. A series of if/elsif control structures will identify the appropriate operation to execute as shown in figure 2. Some novel challenges we encountered were converting register values to type ’unsigned’ (increment/decrement/shift operations) and manually setting Carry-Out bit depending on contents of register when evaluating computations (increment). We used the intermediary signal ”tmp-val” to assign Carry-Out value and ALU-Out in the addition operation. Shifting was done using a VHDL built-in function, the contents of Reg-A were shifted left-/right based on the value stored in Reg-B and sent to ALU-Out using ”std-logic-vector()”.

5.1 Design Code/Design Diagrams

Listing 2: IC source/behavior file code

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use ieee.std_logic_unsigned.all;
--use ieee.std_logic_unsigned.all;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity final_lab_src is
    Port ( Reg_A : in std_logic_vector(31 downto 0);--capacity is 32-bits
           Reg_B : in std_logic_vector(31 downto 0);--capacity is 32-bits
           Op_Sel : in std_logic_vector(3 downto 0);--4 bit value indicates the type of
    operation performed
           Carry_In : in STD_LOGIC;--carry from previous calculation
           Carry_Out : out STD_LOGIC;--output carry from current calculation.
           ALU_Out : out std_logic_vector(31 downto 0));
```

```vhdl
end final_lab_src;

architecture Behavioral of final_lab_src is

signal tmp_val: std_logic_vector(32 downto 0);

begin

process(Reg_A, Reg_B)
begin

-- TRANSFER
if (Op_Sel = "0000") then
    ALU_Out <= Reg_A;
-- INCREMENT
elsif (Op_Sel = "0001") then
    if (unsigned(Reg_A) = x"ffffffff") then
        ALU_Out <= std_logic_vector(unsigned(Reg_A) + 1);
        Carry_Out <= '1';
    else
        ALU_Out <= std_logic_vector(unsigned(Reg_A) + 1);
        Carry_Out <= '0';
    end if;
-- DECREMENT
elsif (Op_Sel = "0010") then
    ALU_Out <= std_logic_vector(unsigned(Reg_A) - 1);
-- ADDITION
elsif (Op_Sel = "0011") then
    tmp_val <= ('0' & Reg_A) + ('0' & Reg_B) + Carry_In;
    ALU_Out <= tmp_val(31 downto 0);
    Carry_Out <= tmp_val(32);
-- NOT
elsif (Op_Sel = "0100") then
    ALU_Out <= NOT Reg_A;
-- AND
elsif (Op_Sel = "0101") then
    ALU_Out <= Reg_A AND Reg_B;
-- OR
elsif (Op_Sel = "0110") then
    ALU_Out <= Reg_A OR Reg_B;
-- XOR
elsif (Op_Sel = "0111") then
    ALU_Out <= Reg_A XOR Reg_B;
-- ARITHMETIC SHIFT RIGHT
```

```vhdl
elsif (Op_Sel = "1000" OR Op_Sel = "1001" OR Op_Sel = "1010" OR Op_Sel = "1011") then
    ALU_Out <= std_logic_vector(shift_right(signed(Reg_A),to_integer(unsigned(Reg_B))));
-- ARITHMETIC SHIFT LEFT
elsif (Op_Sel = "1100" OR Op_Sel = "1101" OR Op_Sel = "1110" OR Op_Sel = "1111") then
    ALU_Out <= std_logic_vector(shift_left(signed(Reg_A),to_integer(unsigned(Reg_B))));
end if;
end process;
end Behavioral;
```

5.2 Schematic(s)

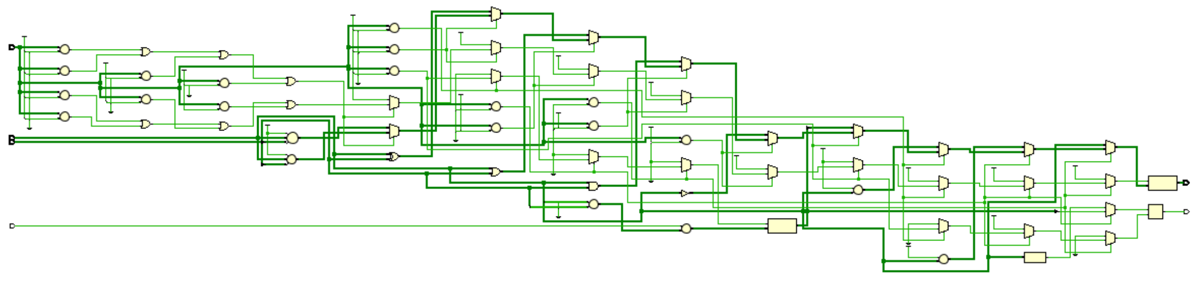


Figure 3: Lab 8 generated schematic

5.3 Test-bench

Listing 3: IC test-bench scr code

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity final_lab_tb is
--  Port ( );
end final_lab_tb;

architecture Behavioral of final_lab_tb is
```

```vhdl

component final_lab_src is
    Port ( Reg_A : in std_logic_vector(31 downto 0);
           Reg_B : in std_logic_vector(31 downto 0);
           Op_Sel : 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 Reg_B : std_logic_vector(31 downto 0);
signal Op_Sel : std_logic_vector(3 downto 0);
signal Carry_In : STD_LOGIC;
signal Carry_Out : STD_LOGIC;
signal ALU_Out : std_logic_vector(31 downto 0);

begin
uut: final_lab_src port map (Reg_A, Reg_B, Op_Sel, Carry_In, Carry_Out, ALU_Out);

process
begin
-- A
Reg_A <= x"00001111";
Reg_B <= x"11000101";
Op_Sel <= "0000";
Carry_In <= '0';
wait for 10ns;

--A + 1
Reg_A <= x"11111111";
Reg_B <= x"00001111";
Op_Sel <= "0001";
Carry_In <= '0';
wait for 10ns;

Reg_A <= x"00001111";
Reg_B <= x"10101010";
Op_Sel <= "0001";
Carry_In <= '0';
wait for 10ns;

Reg_A <= x"10101010";
Reg_B <= x"00000101";
```

```vhdl
Op_Sel <= "0001";
Carry_In <= '0';
wait for 10ns;

Reg_A <= x"11110000";
Reg_B <= x"00010101";
Op_Sel <= "0001";
Carry_In <= '0';
wait for 10ns;

--A - 1
Reg_A <= x"11111111";
Reg_B <= x"00000101";
Op_Sel <= "0010";
Carry_In <= '0';
wait for 10ns;

Reg_A <= x"00001111";
Reg_B <= x"00000101";
Op_Sel <= "0010";
Carry_In <= '0';
wait for 10ns;

Reg_A <= x"10101010";
Reg_B <= x"00000101";
Op_Sel <= "0010";
Carry_In <= '0';
wait for 10ns;

Reg_A <= x"00000000";
Reg_B <= x"00000101";
Op_Sel <= "0010";
Carry_In <= '0';
wait for 10ns;

--A + B + Carry_In
Reg_A <= x"11111111";
Reg_B <= x"11110101";
Op_Sel <= "0011";
Carry_In <= '1';
wait for 10ns;

Reg_A <= x"00001111";
Reg_B <= x"00001111";
```

```
Op_Sel <= "0011";
Carry_In <= '0';
wait for 10ns;

Reg_A <= x"10101010";
Reg_B <= x"11110101";
Op_Sel <= "0011";
Carry_In <= '0';
wait for 10ns;

Reg_A <= x"11110000";
Reg_B <= x"01010101";
Op_Sel <= "0011";
Carry_In <= '1';
wait for 10ns;

--NOT A
Reg_A <= x"10101111";
Reg_B <= x"11110101";
Op_Sel <= "0100";
Carry_In <= '1';
wait for 10ns;

Reg_A <= x"00001111";
Reg_B <= x"10101111";
Op_Sel <= "0100";
Carry_In <= '0';
wait for 10ns;


--A AND B
Reg_A <= x"10101010";
Reg_B <= x"11110101";
Op_Sel <= "0101";
Carry_In <= '0';
wait for 10ns;

Reg_A <= x"11110000";
Reg_B <= x"01010101";
Op_Sel <= "0101";
Carry_In <= '1';
wait for 10ns;


```

```vhdl


--A OR B
Reg_A <= x"11111111";
Reg_B <= x"11110101";
Op_Sel <= "0110";
Carry_In <= '1';
wait for 10ns;

Reg_A <= x"00001111";
Reg_B <= x"00001111";
Op_Sel <= "0110";
Carry_In <= '0';
wait for 10ns;


--A XOR B
Reg_A <= x"10101010";
Reg_B <= x"11110101";
Op_Sel <= "0111";
Carry_In <= '0';
wait for 10ns;

Reg_A <= x"11110000";
Reg_B <= x"01010101";
Op_Sel <= "0111";
Carry_In <= '1';
wait for 10ns;


--ALU_Out = Shift A to the right by 'B' amount
Reg_A <= x"F0101010";
Reg_B <= x"00000001";
Op_Sel <= "1000";
Carry_In <= '0';
wait for 10ns;

Reg_A <= x"11110000";
Reg_B <= x"0000010A";
Op_Sel <= "1010";
Carry_In <= '1';
wait for 10ns;

--ALU_Out = Shift A to the left by 'B' amount
```

```vhdl
Reg_A  <= x"F0101010";
Reg_B  <= x"00000005";
Op_Sel  <= "1100";
Carry_In  <= '0';
wait for 10ns;

Reg_A  <= x"11110000";
Reg_B  <= x"00000A0A";
Op_Sel  <= "1101";
Carry_In  <= '1';
wait for 10ns;
wait;

end process;
end Behavioral;
```

5.4 Waveform/Results

The vertical yellow bar hovering over the time diagram sets the moment in which a specific combination of inputs/outputs occur. For the following time diagrams I will describe the circuits behavior by discussing the I/O streams involved, the logic and the expected output. The expected behavior holds for all test-cases.

At 20ns we see that the contents of Reg-A are sent directly to ALU-out, as expected.

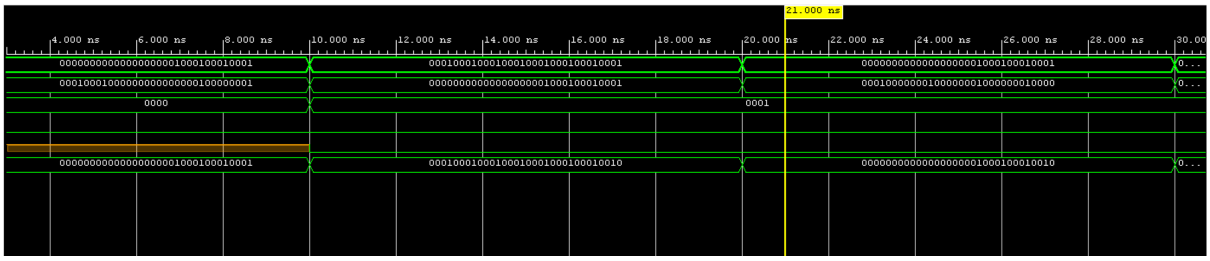


Figure 4: Timing diagram demonstrating transfer & increment operations

We see that the binary string in Reg-A is incremented by '1' as expected.

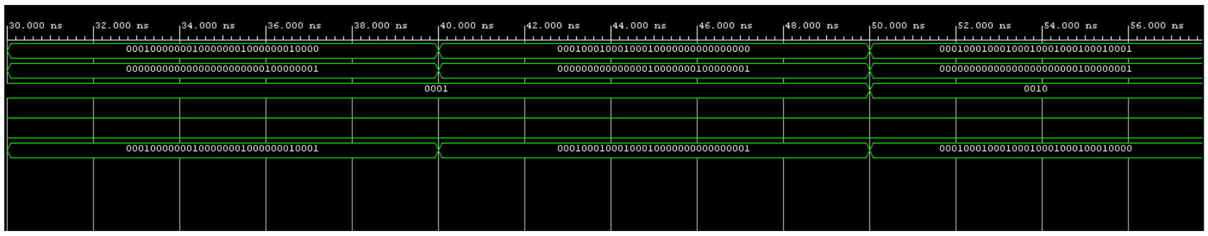


Figure 5: Timing diagram demonstrating increment & decrement operations

We see that the binary string in Reg-A is decremented by '1' as expected.

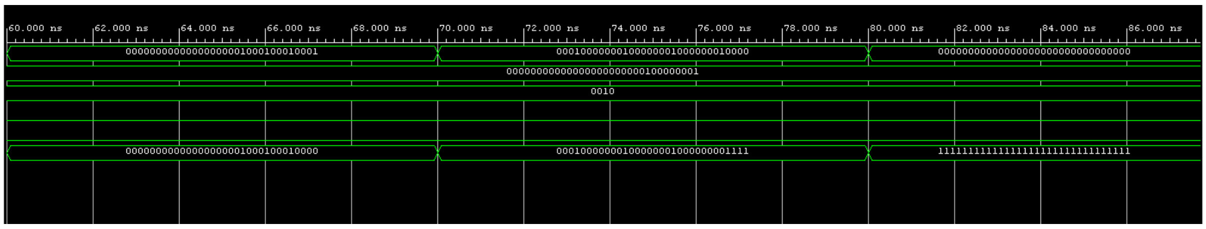


Figure 6: Timing diagram demonstrating decrement operation

Here we add the contents of Reg-A (”4369”, $100ns$), Reg-B(”4369”, $110ns$), and Carry-In(’0’). Gives us the result ”8738” which is valid.

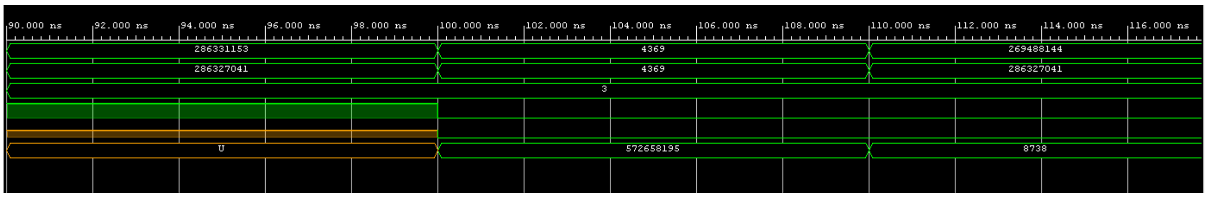


Figure 7: Timing diagram demonstrating addition operation

Here we see that the op-select codes ’0100’(NOT) gives us the expected ALU-Out value at around ˜144ns

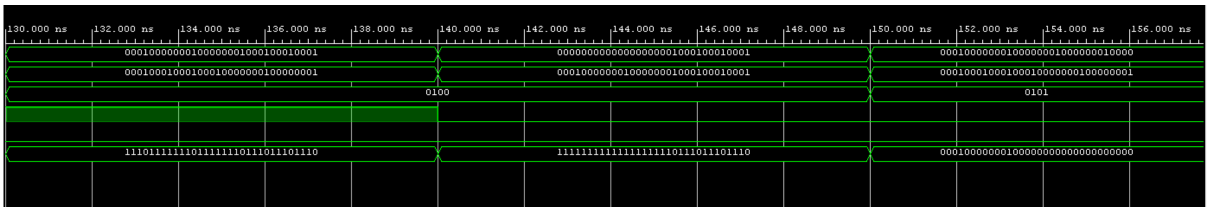


Figure 8: Timing diagram demonstrating not & and operations

Here we see that the op-select codes ’0101’(AND) & ’0110’(OR) give us the expected ALU-Out values at around 164ns and 174ns respectively

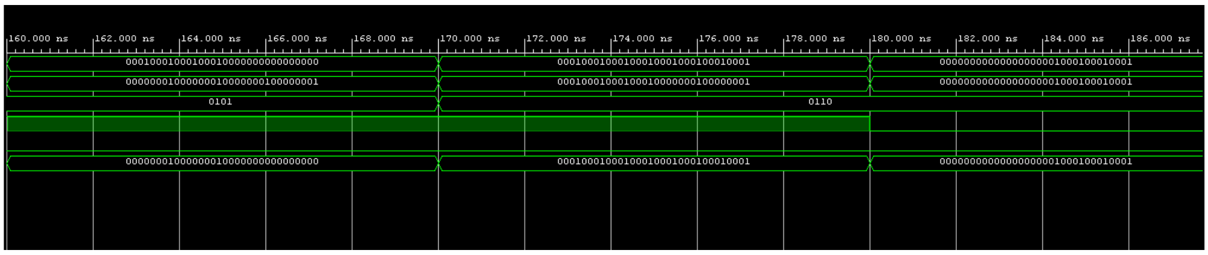


Figure 9: Timing diagram demonstrating and & or operations

At around 194ns we see that XOR operation behaves as expected, particularly at the 4th MSB.

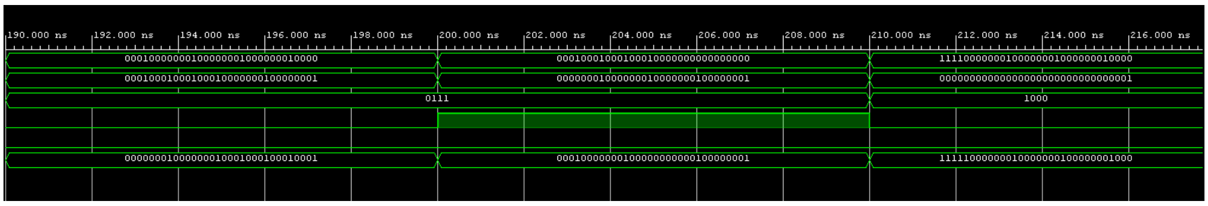


Figure 10: Timing diagram demonstrating xor & shift right operations

With Op-Select value '1010'(shift right) the contents of Reg-A are shifted left by the amount declared in Reg-B, At 224ns we see ALU-Out contains all zeros because the shift amount was too great. At 232ns we see that '1100'(shift left) shifts Reg-A's contents by '101' (Reg-B) to the left 5-bits which is what we observe in ALU-out.

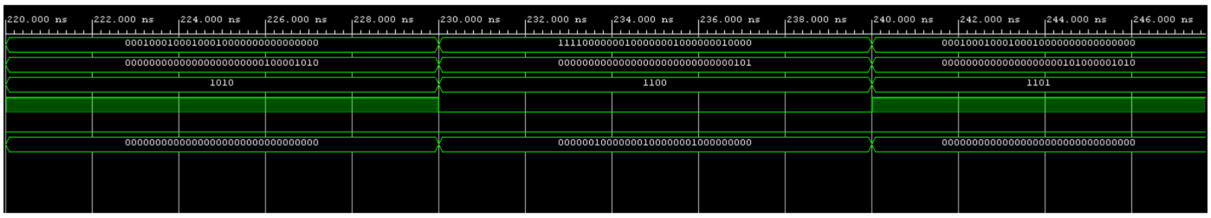


Figure 11: Timing diagram demonstrating shift right & shift right operations

6 Conclusion

In conclusion, the Arithmetic Logic Unit (ALU) is a fundamental component of a computer's central processing unit (CPU) responsible for executing arithmetic, logical, and comparison operations on data. Acting as the computational heart of the CPU, the ALU performs tasks like addition, subtraction, multiplication, and division, as well as logical operations such as AND, OR, NOT, and XOR. It also handles data comparison, bit shift operations, and increment/decrement functions. Working in conjunction with the control unit, the ALU retrieves data from registers, processes it through micro-operations, and stores the results back into the registers. Its key role in implementing essential computer operations makes it the heart of the computer; executing programs and manipulating data in all modern computing systems.