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

| 1 | Abstract | 3 |
|---|--|----|
| 2 | Introduction/Goals | 3 |
| 3 | Requirements | 3 |
| 4 | Prelab | 5 |
| 5 | Report Write-up/Implementation | 5 |
| | 5.1 Design Code/Design Diagrams | 5 |
| | 5.2 Schematic(s) \ldots | 7 |
| | 5.3 Test-bench | 7 |
| | 5.4 Waveform/Results | 12 |
| 6 | Conclusion | 14 |

List of Figures

| 1 | Black-box diagram | 4 |
|----|---|----|
| 2 | 32-bit ALU operations table | 4 |
| 3 | Lab 8 generated schematic | 7 |
| 4 | Timing diagram demonstrating transfer & increment operations | 12 |
| 5 | Timing diagram demonstrating increment & decrement operations $\ldots \ldots \ldots \ldots \ldots \ldots$ | 12 |
| 6 | Timing diagram demonstrating decrement operation | 13 |
| 7 | Timing diagram demonstrating addition operation | 13 |
| 8 | Timing diagram demonstrating not & and operations | 13 |
| 9 | Timing diagram demonstrating and & or operations | 14 |
| 10 | Timing diagram demonstrating xor & shift right operations | 14 |
| 11 | Timing diagram demonstrating shift right & shift right operations | 14 |

Listings

| 1 | I/O stream | 4 |
|---|------------------------------|---|
| 2 | IC source/behavior file code | 5 |
| 3 | IC test-bench scr code | 7 |

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.
- 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:

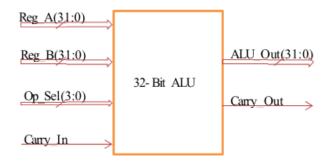


Figure 1: Black-box diagram

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;
```

| 0 | peration Se | lect (Op_S | el) | Carry_In | Operation | Function |
|-----------|-------------|------------|-----------|----------|--|-----------------------|
| Op_Sel(3) | Op_Sel(2) | Op_Sel(1) | Op_Sel(0) | | | |
| 0 | 0 0 0 0 | | | | ALU_Out = A | Transfer Reg_A |
| 0 | 0 | 0 | 1 | x | $ALU_Out = A + 1$ | Increment Reg_A |
| 0 | 0 0 1 0 | | | | 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 | х | 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 | х | 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
```

```
1library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4 use ieee.std_logic_unsigned.all;
5 --use ieee.std_logic_unsigned.all;
7-- Uncomment the following library declaration if instantiating
8-- any Xilinx leaf cells in this code.
9 -- library UNISIM;
10 --use UNISIM.VComponents.all;
12 entity final_lab_src is
     Port ( Reg_A : in std_logic_vector(31 downto 0); -- capacity is 32-bits
13
             Reg_B : in std_logic_vector(31 downto 0);--capacity is 32-bits
14
             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));
18
```

```
19 end final_lab_src;
20
21 architecture Behavioral of final_lab_src is
22
23 signal tmp_val: std_logic_vector(32 downto 0);
24
25 begin
26
27 process (Reg_A, Reg_B)
28 begin
29
30 -- TRANSFER
31 if (Op_Sel = "0000") then
   ALU_Out <= Reg_A;
32
33 -- INCREMENT
34 elsif (Op_Sel = "0001") then
35 if (unsigned(Reg_A) = x"ffffffff") then
        ALU_Out <= std_logic_vector(unsigned(Reg_A) + 1);
36
37
         Carry_Out <= '1';
    else
38
        ALU_Out <= std_logic_vector(unsigned(Reg_A) + 1);
39
        Carry_Out <= '0';
40
    end if;
41
42 -- DECREMENT
43 elsif (Op_Sel = "0010") then
   ALU_Out <= std_logic_vector(unsigned(Reg_A) - 1);
44
45 -- ADDITION
46 elsif (Op_Sel = "0011") then
   tmp_val <= ('0' & Reg_A) + ('0' & Reg_B) + Carry_In;</pre>
47
    ALU_Out <= tmp_val(31 downto 0);</pre>
48
49 Carry_Out <= tmp_val(32);</pre>
50 -- NOT
51 elsif (Op_Sel = "0100") then
52 ALU_Out <= NOT Reg_A;
53 -- AND
54 elsif (Op_Sel = "0101") then
55 ALU_Out <= Reg_A AND Reg_B;
56 -- OR
57 elsif (Op_Sel = "0110") then
58 ALU_Out <= Reg_A OR Reg_B;
59 -- XOR
60 elsif (Op_Sel = "0111") then
61 ALU_Out <= Reg_A XOR Reg_B;
62 -- ARITHMETIC SHIFT RIGHT
```

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

5.2 Schematic(s)

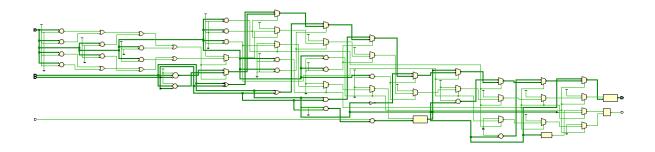


Figure 3: Lab 8 generated schematic

5.3 Test-bench

Listing 3: IC test-bench scr code

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 -- Uncomment the following library declaration if using
_5 -- arithmetic functions with Signed or Unsigned values
6 -- use IEEE.NUMERIC_STD.ALL;
8 -- Uncomment the following library declaration if instantiating
9-- any Xilinx leaf cells in this code.
10 --library UNISIM;
11 --use UNISIM.VComponents.all;
12
13 entity final_lab_tb is
14 -- Port ( );
15 end final_lab_tb;
16
17 architecture Behavioral of final_lab_tb is
```

```
18
19 component final_lab_src is
      Port ( Reg_A : in std_logic_vector(31 downto 0);
20
              Reg_B : in std_logic_vector(31 downto 0);
21
              Op_Sel : in std_logic_vector(3 downto 0);
22
              Carry_In : in STD_LOGIC;
23
              Carry_Out : out STD_LOGIC;
24
              ALU_Out : out std_logic_vector(31 downto 0));
25
26 end component;
27
28 signal Reg_A : std_logic_vector(31 downto 0);
29 signal Reg_B : std_logic_vector(31 downto 0);
30 signal Op_Sel : std_logic_vector(3 downto 0);
31 signal Carry_In : STD_LOGIC;
32 signal Carry_Out : STD_LOGIC;
33 signal ALU_Out : std_logic_vector(31 downto 0);
34
35 begin
36 uut: final_lab_src port map (Reg_A, Reg_B, Op_Sel, Carry_In, Carry_Out, ALU_Out);
37
38 process
39 begin
40 -- A
41 Reg_A <= x"00001111";
42 \text{ Reg}_B \leq x "11000101";
43 Op_Sel <= "0000";
44 Carry_In <= '0';
45 wait for 10ns;
46
47 --A + 1
48 Reg_A <= x"111111111;
49 Reg_B <= x"00001111";
50 Op_Sel <= "0001";
51 Carry_In <= '0';
52 wait for 10ns;
53
54 \text{ Reg}_A \leq x"00001111";
55 \text{ Reg}_B \leq x "10101010";
56 Op_Sel <= "0001";
57 Carry_In <= '0';
58 wait for 10ns;
59
60 Reg_A <= x "10101010";
_{61} \text{Reg}_B \leq x"00000101";}
```

```
62 Op_Sel <= "0001";
63 Carry_In <= '0';
64 wait for 10ns;
65
66 \operatorname{Reg}_A \leq x"11110000";
67 Reg_B <= x"00010101";
68 Op_Sel <= "0001";
69 Carry_In <= '0';
70 wait for 10ns;
71
72 --A - 1
73 Reg_A <= x"11111111";
74 \text{ Reg}_B \leq x"00000101";
75 Op_Sel <= "0010";
76 Carry_In <= '0';
77 wait for 10ns;
78
79 Reg_A <= x"00001111";
80 Reg_B <= x"00000101";
81 Op_Sel <= "0010";</pre>
82 Carry_In <= '0';</pre>
83 wait for 10ns;
84
85 Reg_A <= x"10101010";
86 Reg_B <= x"00000101";
87 Op_Sel <= "0010";
88 Carry_In <= '0';</pre>
89 wait for 10ns;
90
91 Reg_A <= x"00000000";
92 \text{ Reg}_B \le x"00000101";
93 Op_Sel <= "0010";</pre>
94 Carry_In <= '0';
95 wait for 10ns;
96
97 --A + B + Carry_In
98 Reg_A <= x"11111111";
99 \text{Reg}_B \leq x"11110101";
100 Op_Sel <= "0011";
101 Carry_In <= '1';</pre>
102 wait for 10ns;
103
104 \text{ Reg}_A <= x"00001111";
105 \text{ Reg}_B \le x"00001111";
```

```
106 Op_Sel <= "0011";
107 Carry_In <= '0';
108 wait for 10ns;
109
110 \operatorname{Reg}_A \leq x "10101010";
111 Reg_B <= x"11110101";</pre>
112 Op_Sel <= "0011";</pre>
113 Carry_In <= '0';
114 wait for 10ns;
115
116 \operatorname{Reg}_A \leq x"11110000";
117 Reg_B <= x " 01010101";
118 Op_Sel <= "0011";</pre>
119 Carry_In <= '1';
120 wait for 10ns;
121
122 --NOT A
123 \operatorname{Reg}_A \leq x"10101111";
124 \text{ Reg}_B \leq x"11110101";
125 Op_Sel <= "0100";
126 Carry_In <= '1';
127 wait for 10ns;
128
129 \operatorname{Reg}_A \leq x"00001111";
130 \text{ Reg}_B \leq x "101011111";
131 Op_Sel <= "0100";</pre>
132 Carry_In <= '0';</pre>
133 wait for 10ns;
134
135
136 --A AND B
137 \text{ Reg}_A \leq x"10101010";
138 Reg_B <= x"11110101";
139 Op_Sel <= "0101";
140 Carry_In <= '0';
141 wait for 10ns;
142
143 \text{Reg}_A \leq x"11110000";
144 Reg_B <= x"01010101";
145 Op_Sel <= "0101";
146 Carry_In <= '1';
147 wait for 10ns;
148
149
```

150

```
152 --A OR B
153 Reg_A <= x"111111111;
154 \text{ Reg}_B \leq x"11110101";
155 Op_Sel <= "0110";</pre>
156 Carry_In <= '1';
157 wait for 10ns;
158
159 \operatorname{Reg}_A \leq x"00001111";
160 \text{ Reg}_B \leq x"00001111";
161 Op_Sel <= "0110";</pre>
162 Carry_In <= '0';</pre>
163 wait for 10ns;
164
165
166 --A XOR B
167 \operatorname{Reg}_A \leq x"10101010";
168 \text{ Reg}_B \le x"11110101";
169 Op_Sel <= "0111";</pre>
170 Carry_In <= '0';
171 wait for 10ns;
172
173 \text{ Reg}_A \leq x"11110000";
174 \text{ Reg}_B \leq x"01010101";
175 Op_Sel <= "0111";
176 Carry_In <= '1';
177 wait for 10ns;
178
179
180 --ALU_Out = Shift A to the right by 'B' amount
181 \operatorname{Reg}A \leq x"F0101010";
182 \operatorname{Reg}_B \leq x"0000001";
183 Op_Sel <= "1000";
184 Carry_In <= '0';</pre>
185 wait for 10ns;
186
187 \text{Reg}_A \leq x"11110000";
188 \text{ Reg}_B \le x"0000010 \text{ A}";
189 Op_Sel <= "1010";</pre>
190 Carry_In <= '1';
191 wait for 10ns;
192
193 -- ALU_Out = Shift A to the left by 'B' amount
```

```
194 Reg_A <= x"F0101010";
195 Reg_B <= x"00000005";
196 Op_Sel <= "1100";
197 Carry_In <= '0';
198 wait for 10ns;
199
200 Reg_A <= x"11110000";
201 Reg_B <= x"00000A0A";
202 Op_Sel <= "1101";
203 Carry_In <= '1';
204 wait for 10ns;
205 wait;
206
207 end process;
208 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.

| | | 21 | 1.000 ns | |
|---|--|------|---|-----------------------------|
| 4.000 ns 6.000 ns 8.000 ns 1 | 10.000 ns 12.000 ns 14.000 ns 16.00 | | | 26.000 ns 28.000 ns 30.00 |
| 000000000000000000000000000000000000000 | 000100010001000100010001000100010 | | 000000000000000000000000000000000000000 | |
| 000100010000000000000000000000000000000 | 000000000000000000000000000000000000000 | 2001 | 0001000000100000010000 | 00010000 0 |
| 00000 | | 0001 | | |
| | | | | |
| | | | | |
| 000000000000000000000000000000000000000 | 00010000 | 0010 | 000000000000000000000000000000000000000 | 00010010 |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

Figure 4: Timing diagram demonstrating transfer & increment operations

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

| 30.000 ns | | | | | | | | | | | | | 56.000 ns | |
|-----------|-----------|---|-----------|----|----------|-----------|----------------|-----------|---|-----|--|----------------|-----------|--|
| | | 00100000001000 | | | * | | 00100010000000 | |) | | | | | |
| | 00000000 | 000000000000000000000000000000000000000 | 100000001 | | k | 000000000 | 00000010000000 | 100000001 | | 000 | n# \$2.000 ns \$4.000 ns \$4.000 ns \$5.000 ns 00010010010010010001000100010001 00000000 | | | |
| | 1 | | | 00 | 01 | | | | | | 001 | 0 | | |
| | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | |
| k | 000100000 | 00100000001000 | 000010001 | | <u>k</u> | 000100010 | 00100010000000 | 000000001 | | 000 | 10001000100010 | 00100010001000 |) | |
| | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | |

Figure 5: Timing diagram demonstrating increment & decrement operations

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

| 60.000 ns | 62.000 ns | | | | | | | | | | | | 86.000 ns | |
|-----------|-----------|---|-----------|---|-----|---|----------------|-----------|---|-----|----------------|-------------------|-----------|--|
| (| | 000000000000000000000000000000000000000 | | , | (| | 00100000001000 | | , | | | | | |
| 1 | | | | | 000 | 000000000000000000000000000000000000000 | 00000010000000 | | | | | | | |
| | | | | | | 001 | 0 | | | | | | | |
| | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | |
| | 00000000 | 000000000000000000000000000000000000000 | 100010000 | | | 000100000 | 00100000001000 | 000001111 | | 111 | 11111111111111 | 11111111111111111 | 1 | |
| | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | |

Figure 6: Timing diagram demonstrating decrement operation

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

| | | | 100.000 ns | | | | 110.000 ns | | | 116.000 ns |
|---|-----------|------|------------|---|-----------|--|------------|-------|------|------------|
| | | | | | | | | | | |
| | 286331153 | | K | | 4369 | | X | 26948 | 8144 | |
| | 286327041 | | k | | 4369 | | X | 28632 | 7041 | |
| | | | | 3 | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| < | U | · | k | | 572658195 | | X | 873 | 8 | |
| | | | | | | | | | | |
| | | | | | | | | | | |

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 $\tilde{1}44\mathrm{ns}$

| ,130.000 ns | ,132.000 ns | 134.000 ns | 136.000 ns | ,138.000 ns | 140.000 ns | 142.000 ns | 144.000 ns | 146.000 ns | 148.000 ns | 150.000 ns | 152.000 ns | 154.000 ns | 156.000 ns | |
|-------------|---|------------------|------------|-------------|------------|------------|---|------------|------------|------------|--|---|------------|--|
| | | | | | | | | | | | | | | |
| | 00010000 | 0001000000001000 | 100010001 | | K | 00000000 | 000000000000000000000000000000000000000 | 100010001 | | 001 | 010000000100000 | 00100000001000 | 0 | |
| | 000100010001000000000000000000000000000 | | | | | 000100000 | 00100000001000 | 100010001 | | 001 | 10000000100000000100000000000000000000 | | | |
| | | | | . 01 | .00 | | | | | * | 010 |)1 | | |
| | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | |
| | 11101111 | 111011111110111 | 011101110 | | k | 11111111 | 1111111111110111 | 011101110 | | 001 | 01000000010000 | 000000000000000000000000000000000000000 | 0 | |
| | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | |

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

| | | 180.000 ns 182.000 ns 184.000 ns 186.000 ns |
|---|---|---|
| | | |
| 000100010001000000000000000000000000000 | 0001000100010001000100010001 | 000000000000000000000000000000000000000 |
| 0000000100000010000000100000001 | 000100010001000000000000000000000000000 | 000000000000000000000000000000000000000 |
| 0101 | 0110 | |
| | | |
| | | |
| 000000010000001000000000000000 | 00010001000100010001000100010001 | 000000000000000000000000000000000000000 |
| | | |
| | | |
| | | |
| | | |

Figure 9: Timing diagram demonstrating and & or operations

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

| | | 204.000 ns 206.000 ns | 208.000 ns | 210.000 ns | 212.000 ns | 214.000 ns | 216.000 ns |
|---|------------|---|------------|------------|----------------|-----------------|------------|
| | | | | | | | |
| 000100000010000001000000000000000000000 | 0001000100 | 0010001000000000000000 | | 111 | 10000000100000 | 001000000010000 | 0 |
| 000100010001000000000000000000000000000 | 000000100 | 000000010000000000000000000000000000000 | | | | | |
| 01 | 11 | | 1000 | | | | |
| | | | | | | | |
| | | | | | | | |
| 0000000100000001000100010001 | 0001000000 | 00100000000000100000001 | | 111 | 11000000010000 | 000100000000100 |) |
| | | | | | | | |

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.

| | 224.000 ns | | | 232.000 ns | 234.000 ns | 236.000 ns | 240.000 ns | | 244.000 ns | 246.000 ns |
|-----------|----------------|-----------|---|------------|---|------------|------------|-----------------|---|------------|
| 000100010 | 00100010000000 | 000000000 | (| 111100000 | 00100000001000 | 000010000 | 000 | 010001000100010 | 0000000000000000 | io - |
| 00000000 | | 100001010 | | 00000000 | 000000000000000000000000000000000000000 | 000000101 | 000 | | 0001010000010 | 10 |
| | 1010 | | | | 1100 | | * | 110 | 1 | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| 00000000 | | 000000000 | | 000000100 | 00000100000001 | 000000000 | 000 | | 000000000000000000000000000000000000000 | 0 |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |

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.