Project 2: Designing an 8-bit ALU

Adam Sumner - A20283081
Contribution - 100%

ECE 485

November 16th, 2015

1 Introduction

The purpose of this project is to implement an 8-bit ALU in the VHDL hardware-description language. The ALU should be able to perform the following functions:

- ADD
- SUBTRACT
- LESS-THAN
- AND
- NOT
- XOR
- Bit Shift Left
- Arithmetic Bit Shift Right

2 Design

2.1 Architecture

The architecture of the ALU is simple. First and foremost, a `clk` signal is included. This is necessary because during the design process, it was concluded that all operations will be performed and outputted on the positive edge of the `clk`. Also, as per the design specification, two inputs `a` and `b` must be included. These are both signed vectors of 8 bits because this ALU is designed for 8 bit operations. They are signed because operations like addition and subtraction are included in the ALU's capability. Because in the real world negative numbers are often part of computer programs, it was necessary to make sure that the functionality of the ALU would be able to handle them. Due to the fact that signed numbers are being used, two status bits are included. These are `ov` and `zero`. `ov` is the overflow bit that is set when an overflow/underflow occurs, and `zero` is the zero status bit that is set when the output is zero. `op` is the operation bit. This signal is used to select the operation of the ALU to be performed on inputs `a` and `b`. `s` is a selector bit. Because bit shift left, arithmetic bit shift right, and NOT are included in the functionality of the ALU and the ALU always takes two inputs, this signal is included to select which input to perform the shift or NOT operation on. Last an output must be included in the design of the ALU, and this is done by implementing an output `y` of 8 bits. A visual overview of the ALU is shown in Figure 1.

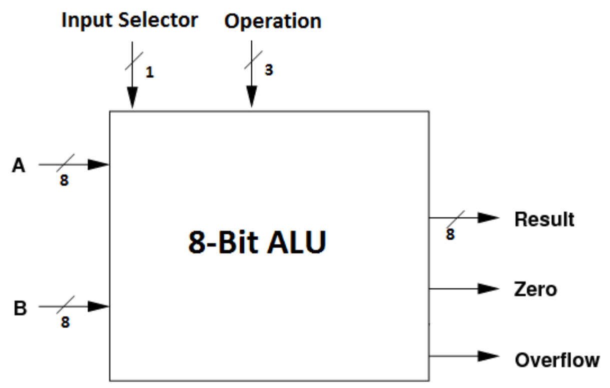


Figure 1: High Level Overview of Architecture

2.2 Behavior

The behavior section of the code in Section 2.3 is at its core a switch case on the operation to be performed. Two bit vectors are first declared, `temp9` and `temp`. `temp9` is used in the calculation for overflow and underflow and `temp` is a temporary storage location of the result so that before its value is outputted from the ALU, its value can be checked for zero to see if the zero status bit should be set or not. A process is defined for the `clk` signal. If the `clk` is at its rising edge, then an operation can be performed. The value of the operation to be performed is then assessed. The bit pattern-operation pairs are shown in Table 1. When an operation is performed, `temp` receives the value.

$\mathtt{op}_2\mathtt{op}_1\mathtt{op}_0$	`y` (output)
000	bit-wise AND of a and b
001	bit-wise NOT of a and b
010	ADD (a+b)
011	Less Than
100	Subtract (a-b)
101	bit-wise XOR of a and b
110	Logical left shift by one
111	Arithmetic right shift by one bit

Table 1: ALU Operation Description

AND and XOR are both straightforward operations that do not require any conditionals or overflow checking. NOT involves setting the selector bit to determine which input to perform the NOT operation on. This is done with a simple conditional where if `s` == 1 then `a` is complemented and if `s` == 0, then `b` is complemented. Addition and subtraction require a little more involvement. First the 8 bit signed vectors `a` and `b` are sign extended into 9 bit signed vectors and then added or subtracted and the result is stored in `temp9`. If the MSB or 9th bit of `temp9` differs from the 8th bit, then an overflow occurred in the case of addition, and an underflow occurred in the case of subtraction. The `ov` bit is then set accordingly. The result of `temp9` is then resized back to 8 bits into `temp` because the value must still be outputted by the ALU. For the Less Than operation, `a` is tested if it is less than `b`. If it is, then the value is set to 11111111_2, or -1 in decimal. In some old languages, such as Forth, -1 is denoted as the True value for a boolean. If `a` is not less than b, then temp is set to 00000000_2, which corresponds to the value of False in most languages. After the switch case is exited, the zero bit is set accordingly, where a 1 denotes False, and a 0 denotes True. The designer of the Assembly Language to correspond with this ALU design may use either the output values -1/0, or the zero status bit to determine the value of the conditional operation. The bit shift operations involve the use of the function SHIFT_LEFT and SHIFT_RIGHT. Because an arithmetic shift left and a logical shift left perform the same functionality, an unsigned or signed vector can be fed to the function without worry if the output will differ. However, an arithmetic right shift is used for signed numbers, while a logical right shift is used for unsigned numbers. Luckily, the function SHIFT_RIGHT has the ability to be overloaded, so depending on the type of vector that is passed in, it will perform the corresponding shift. Since in this case a signed vector is passed in, the function automatically performs an arithmetic shift right. Also, there is a conditional check on the bit selector signal in both shifts. If it is set to 1, then input a is shifted by one bit. If it is set to 0, then input b is shifted by one bit. Because outputs cannot be read if not using a VHDL compiler post 2008, the temp value is used in the conditional at the end of the switch case to determine the zero status bit value. Once established, the value of temp is then written to the output y. Figure 2 shows a more in depth block diagram of what the ALU looks like.

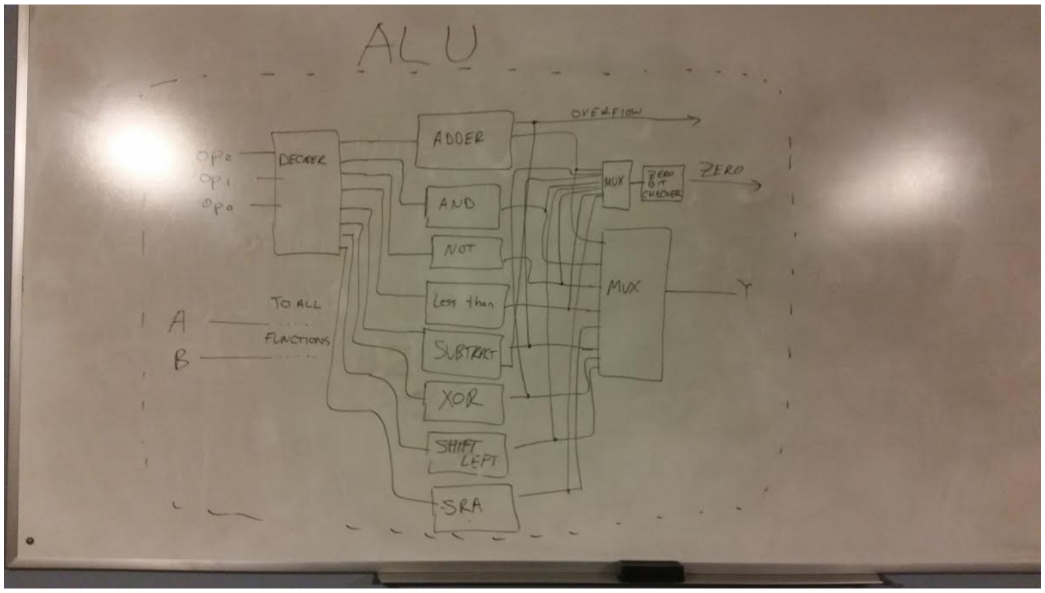


Figure 2: Block Diagram of ALU

2.3 Code

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

entity alu8 is
port(
    clk : in std_logic;
    s : in std_logic;
    ov, zero  : out std_logic;
    a,b : in signed(7 downto 0);
    op  : in unsigned(2 downto 0);
    y : out signed(7 downto 0)

    );
end alu8;

architecture Behavioral of alu8 is

```

```vhdl
signal temp9: signed(8 downto 0);
signal temp: signed(7 downto 0);

begin
process(clk)
begin
  if(rising_edge(clk)) then
    case op is
      when "000" =>
        temp <= a AND b;
      when "001" =>
        if s = '1' then
          temp <= NOT a;
        else
          temp <= NOT b;
        end if;
      when "010" =>
        temp9 <= (a(7)& a) + (b(7) & b);
        if temp9(8) /= temp9(7) then
          ov <= '1'; --overflow!
        else
          ov <= '0';
        end if;
        --unsigned on temp so resize doesn't save the sign
        temp <= signed(resize(unsigned(temp9), temp'length));
      when "011" =>
        if(a < b) then
          temp <= "11111111";
        else
          temp <= "00000000";
        end if;
      when "100" =>
        temp9 <= (a(7)& a) - (b(7) & b);
        if temp9(8) /= temp9(7) then
          ov <= '1'; --underflow!
        else ov <= '0';
        end if;
        --unsigned on temp so resize doesn't save the sign
        temp <= signed(resize(unsigned(temp9), y'length));
      when "101" =>
```

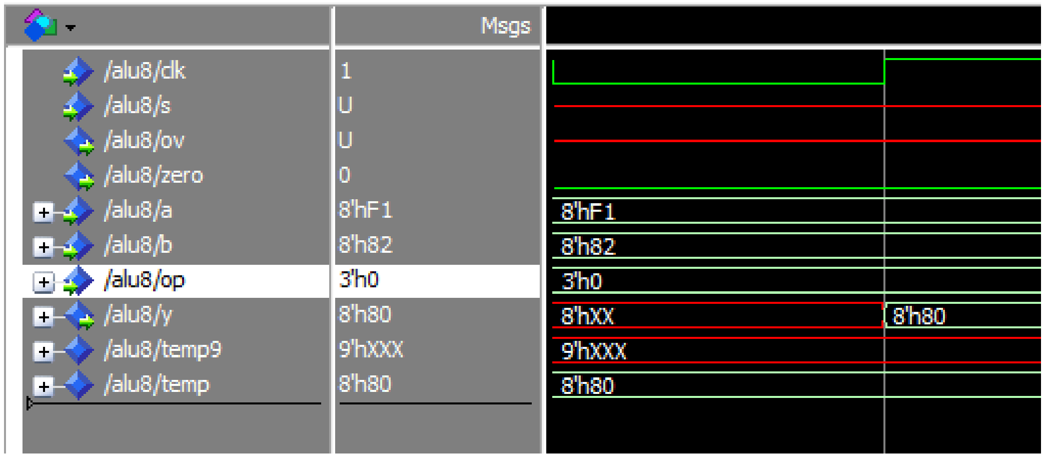
```vhdl
            temp <= a XOR b;
          when "110" =>
            if s = '1' then
              temp <= SHIFT_LEFT(a,1);
            else
              temp <= SHIFT_LEFT(b,1);
            end if;
          when "111" =>
            if s = '1' then
              temp <= SHIFT_RIGHT(a,1);
            else
              temp <= SHIFT_RIGHT(b,1);
            end if;
          when others =>
            NULL; --default case
        end case;
        if temp = "00000000" then
          zero <= '1';
        else
          zero <= '0';
        end if;
        y <= temp;
      end if;
    end process;

    end Behavioral;
```

3 Analysis

While the specification of the ALU to be implemented was not specific, the goal of the design was to give the ALU the basic defined functionality with added features. For example, the design simply said to include operations such as addition and subtraction, while the implemented design included signed addition and subtraction with the ability to detect overflow/underflow or if the value is zero using status bits. While this increased the complexity of the logic of the design, it enables the ALU to be a unit that can actually be utilized rather than a unit that “just gets the bare minimum job done”. A downfall to this decision however is the range of numbers that can be calculated. 1 bit is lost due to the sacrifice for a sign bit, but this sacrifice is necessary if the ALU is to work in conjunction with any modern program. To improve upon the design however, a carry bit should be implemented. This would mean that the ALU would support unsigned addition and subtraction as well as signed. An ALU in practice does not know whether it is performing signed or unsigned operations, it just sets the status bits accordingly. The carry bit would be an easy addition to the current design by just checking if the MSB of `temp9` is set to 1 . Furthermore, more operations could be included in the design. OR, NAND, NOR, etc. could be included in the list of supported operations, as well as multiplication and division. The logic gates would be easy to implement, while multiplication and division would change the size of the output vector to be 16 bits. The current state of the design allows these additional functions to be easily implemented in the future. As more functionality is added, more hardware will be needed. Thanks to Moore's law, hardware is quite cheap to implement, so adding more hardware to the 8 bit ALU would not cost much to the manufacturer.

4 Simulation Results

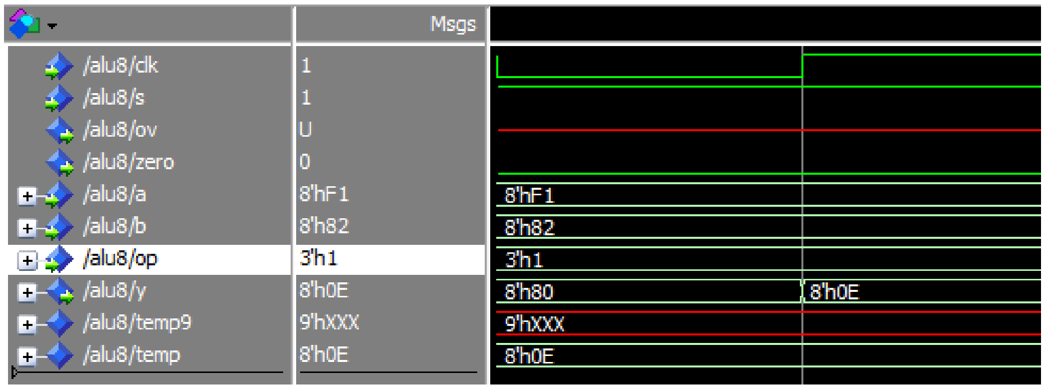
This section will walk through each operation that the ALU is capable of performing. A Clock cycle of 10ns was used for the simulation. Figure 3 shows the results of anding $F1_{16}$ with 82_{16}. The proper output 80_{16} is displayed as well as the proper value of the zero status bit.



```
                  Msgs
/alu8/clk         1
/alu8/s           U
/alu8/ov          U
/alu8/zero        0
/alu8/a           8'hF1      8'hF1
/alu8/b           8'h82      8'h82
/alu8/op          3'h0       3'h0
/alu8/y           8'h80      8'hXX                8'h80
/alu8/temp9       9'hXXX     9'hXXX
/alu8/temp        8'h80      8'h80
```

Figure 3: Result of AND

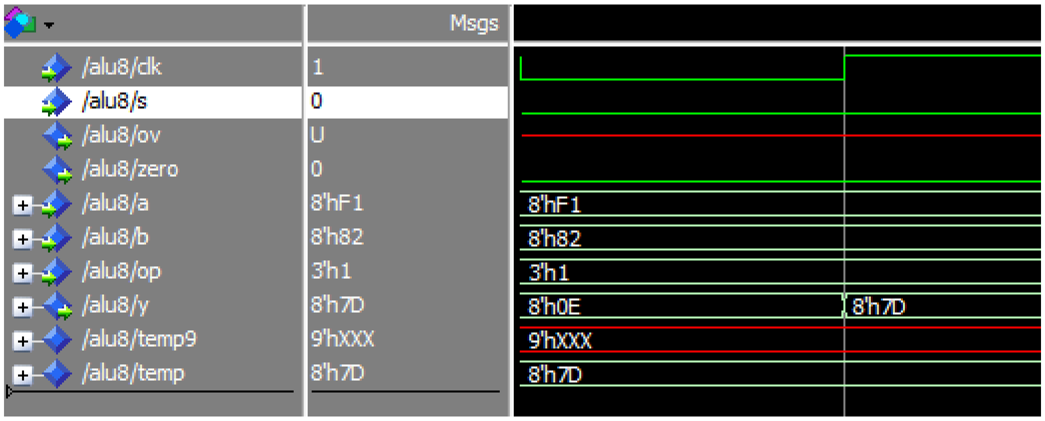
Figure 4 shows the result of selecting the input a to be negated. Input a has the value of $F1_{16}$ and the correct output of $0E_{16}$ is calculated. The zero bit is also correctly set.



```
                 Msgs
/alu8/clk        1
/alu8/s          1
/alu8/ov         U
/alu8/zero       0
/alu8/a          8'hF1      8'hF1
/alu8/b          8'h82      8'h82
/alu8/op         3'h1       3'h1
/alu8/y          8'h0E      8'h80     8'h0E
/alu8/temp9      9'hXXX     9'hXXX
/alu8/temp       8'h0E      8'h0E
```

Figure 4: Result of NOT with input a

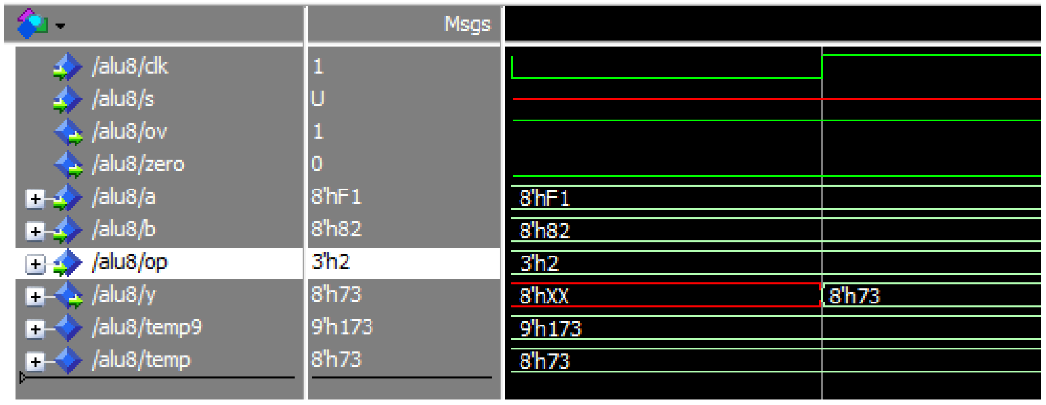
Figure 5 shows the result of selecting the input a to be negated. Input a has the value of 82_{16} and the correct output of $7D_{16}$ is calculated. The zero bit is also correctly set.



```
                 Msgs
/alu8/clk        1
/alu8/s          0
/alu8/ov         U
/alu8/zero       0
/alu8/a          8'hF1      8'hF1
/alu8/b          8'h82      8'h82
/alu8/op         3'h1       3'h1
/alu8/y          8'h7D      8'h0E     8'h7D
/alu8/temp9      9'hXXX     9'hXXX
/alu8/temp       8'h7D      8'h7D
```

Figure 5: Result of NOT with input b

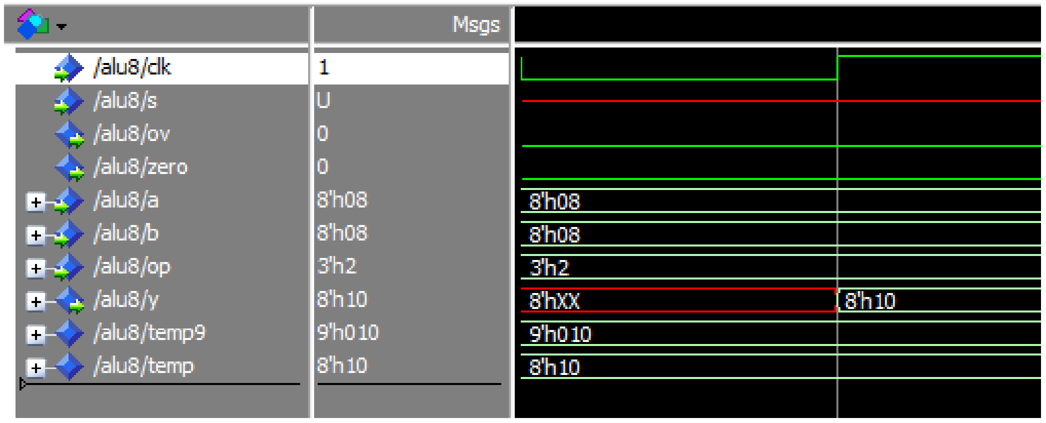
Figure 6 shows the result of adding $F1_{16}$ and 82_{16}. The correct output of 73_{16} is shown and the overflow bit is in fact set to one. This is because two negative numbers produced a positive number! The zero bit is also set correctly.



	Msgs
/alu8/clk	1
/alu8/s	U
/alu8/ov	1
/alu8/zero	0
/alu8/a	8'hF1
/alu8/b	8'h82
/alu8/op	3'h2
/alu8/y	8'h73
/alu8/temp9	9'h173
/alu8/temp	8'h73

Figure 6: Result of Addition

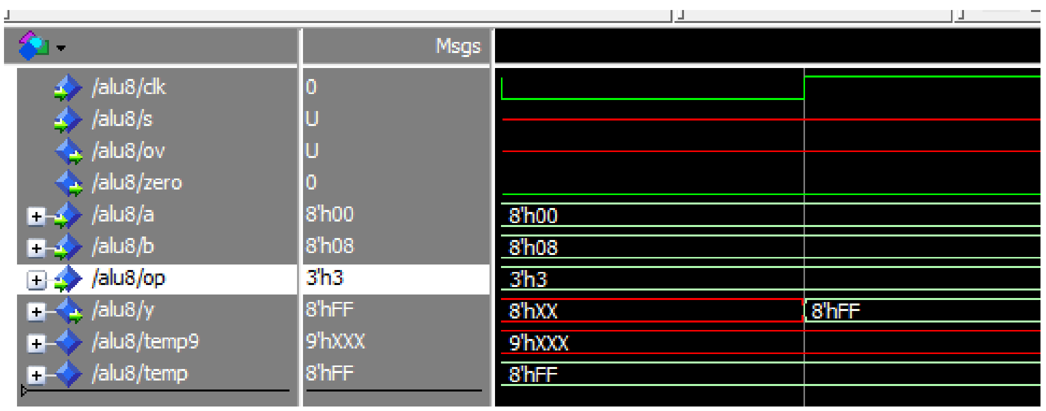
Figure 7 shows the result of adding 08_{16} and 08_{16}. The correct output of 10_{16} is shown. Both status bits are set correctly.



	Msgs
/alu8/clk	1
/alu8/s	U
/alu8/ov	0
/alu8/zero	0
/alu8/a	8'h08
/alu8/b	8'h08
/alu8/op	3'h2
/alu8/y	8'h10
/alu8/temp9	9'h010
/alu8/temp	8'h10

Figure 7: Addition Without Overflow

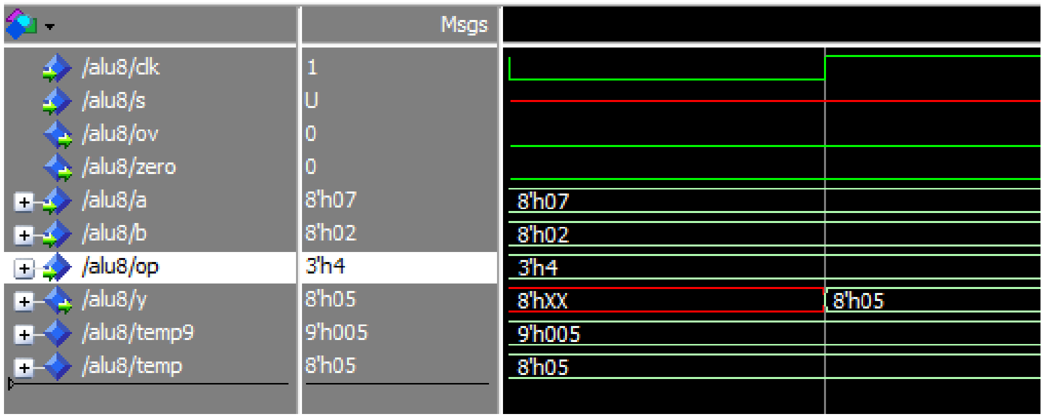
Figure 8 shows the result of checking if 00_{16} is less than 08_{16}. The output is FF_{16} and the zero bit is set to 0 which is the expected result.



```
                Msgs
/alu8/clk       0
/alu8/s         U
/alu8/ov        U
/alu8/zero      0
/alu8/a         8'h00      8'h00
/alu8/b         8'h08      8'h08
/alu8/op        3'h3       3'h3
/alu8/y         8'hFF      8'hXX    8'hFF
/alu8/temp9     9'hXXX     9'hXXX
/alu8/temp      8'hFF      8'hFF
```

Figure 8: Result of Less Than

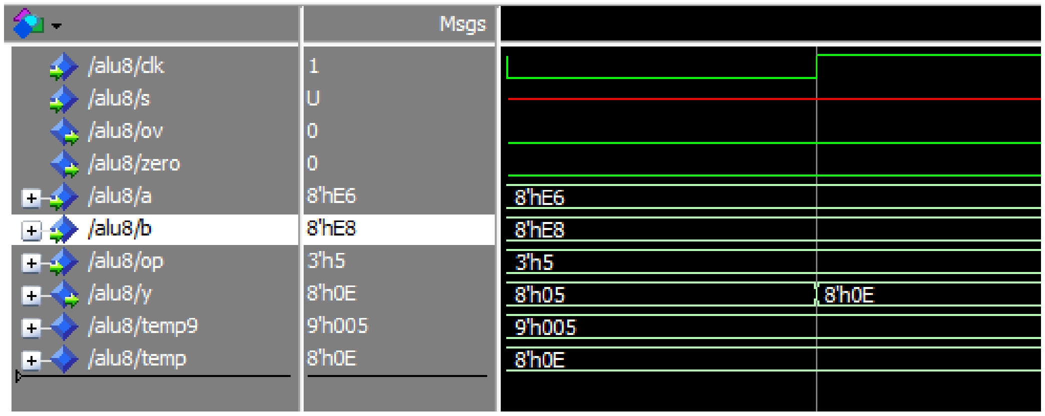
Figure 9 shows the result of $07_{16} - 02_{16}$. The expected result 05_{16} is outputted and the overflow and zero status bits are both set correctly to 0.



```
                Msgs
/alu8/clk       1
/alu8/s         U
/alu8/ov        0
/alu8/zero      0
/alu8/a         8'h07      8'h07
/alu8/b         8'h02      8'h02
/alu8/op        3'h4       3'h4
/alu8/y         8'h05      8'hXX    8'h05
/alu8/temp9     9'h005     9'h005
/alu8/temp      8'h05      8'h05
```

Figure 9: Result of Subtract

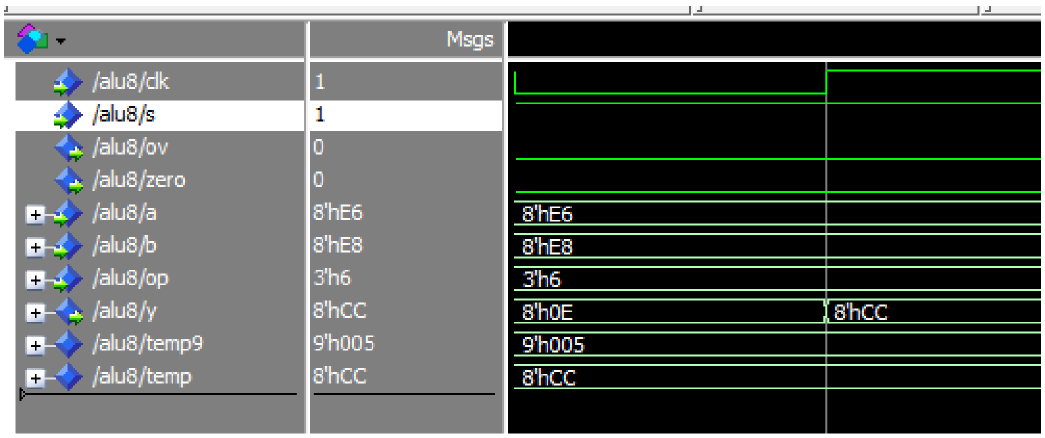
Figure 10 shows the result of $E6_{16} \oplus E8_{16}$. The correct result of $0E_{16}$ is outputted.



```
                     Msgs
/alu8/clk            1
/alu8/s              U
/alu8/ov             0
/alu8/zero           0
/alu8/a              8'hE6       8'hE6
/alu8/b              8'hE8       8'hE8
/alu8/op             3'h5        3'h5
/alu8/y              8'h0E       8'h05            8'h0E
/alu8/temp9          9'h005      9'h005
/alu8/temp           8'h0E       8'h0E
```

Figure 10: Result of XOR

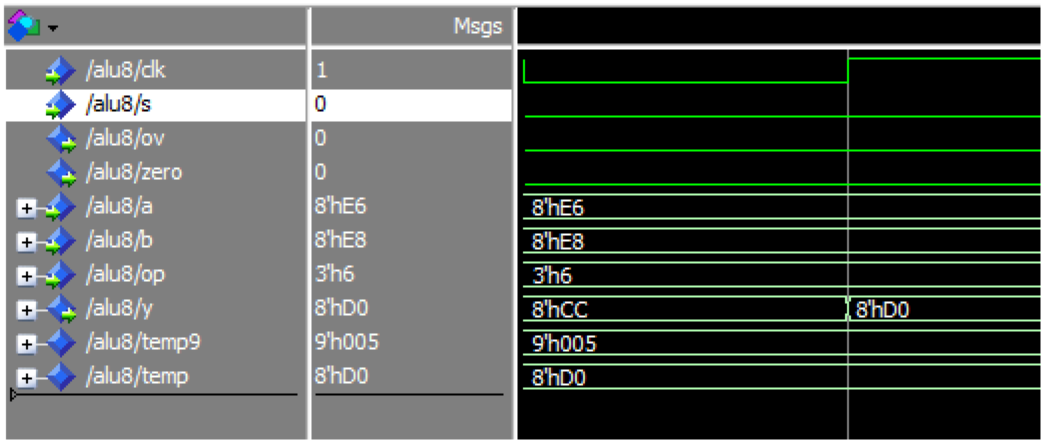
Figure 11 shows the result of shifting input `a` with a value of $E6_{16}$ left by 1 bit. The correct output is displayed as CC_{16}.



```
                     Msgs
/alu8/clk            1
/alu8/s              1
/alu8/ov             0
/alu8/zero           0
/alu8/a              8'hE6       8'hE6
/alu8/b              8'hE8       8'hE8
/alu8/op             3'h6        3'h6
/alu8/y              8'hCC       8'h0E            8'hCC
/alu8/temp9          9'h005      9'h005
/alu8/temp           8'hCC       8'hCC
```

Figure 11: Shift Left on Input `a`

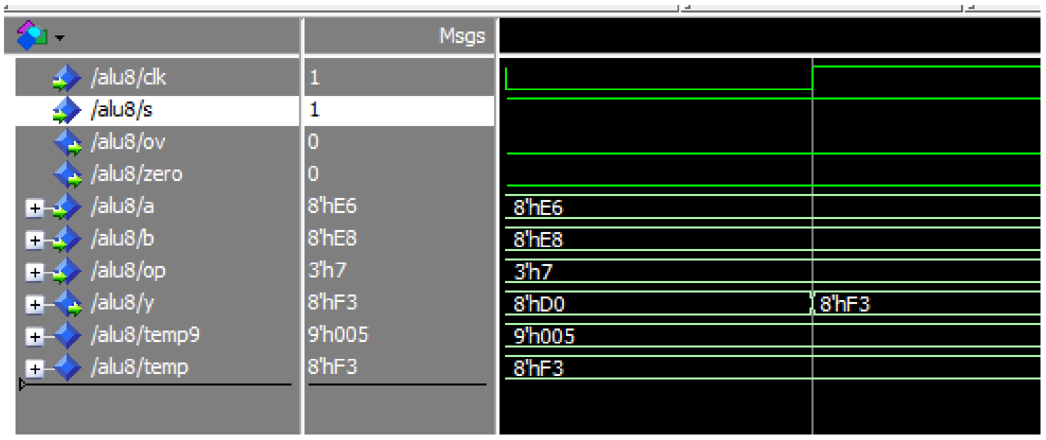
Figure 12 shows the result of shifting input `b` with a value of $E8_{16}$ left by 1 bit. The correct output is displayed as $D0_{16}$.



```
                  Msgs
/alu8/clk         1
/alu8/s           0
/alu8/ov          0
/alu8/zero        0
/alu8/a           8'hE6      8'hE6
/alu8/b           8'hE8      8'hE8
/alu8/op          3'h6       3'h6
/alu8/y           8'hD0      8'hCC    8'hD0
/alu8/temp9       9'h005     9'h005
/alu8/temp        8'hD0      8'hD0
```

Figure 12: Shift Left on Input `b`

Figure 13 shows the result of arithmetically shifting input `a` with a value of $E6_{16}$ right by 1 bit. The correct result of $F3_{16}$ is outputted.



```
                  Msgs
/alu8/clk         1
/alu8/s           1
/alu8/ov          0
/alu8/zero        0
/alu8/a           8'hE6      8'hE6
/alu8/b           8'hE8      8'hE8
/alu8/op          3'h7       3'h7
/alu8/y           8'hF3      8'hD0    8'hF3
/alu8/temp9       9'h005     9'h005
/alu8/temp        8'hF3      8'hF3
```

Figure 13: Arithmetic Shift Right on Input `a`

Figure 14 shows the result of arithmetically shifting input `b` with a value of $E8_{16}$ right by 1 bit. The correct result of $F4_{16}$ is outputted.

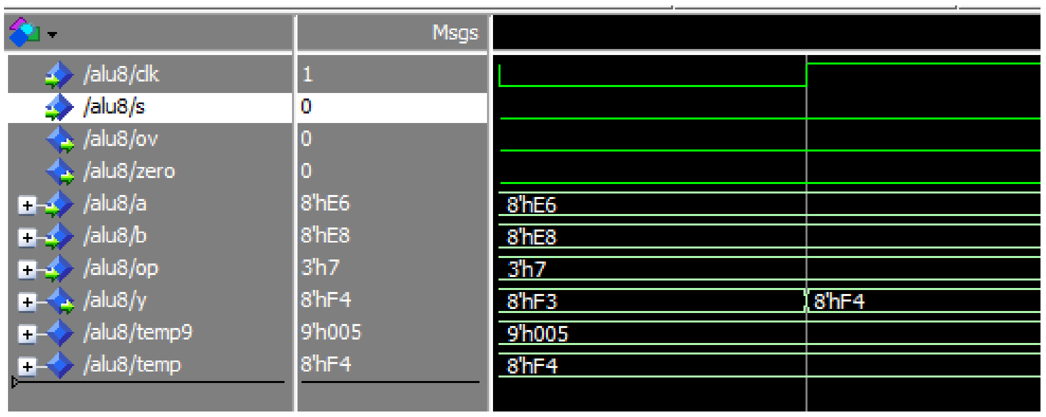


Figure 14: Arithmetic Shift Right on Input `b`

On average, each operation took around 2-3 clock cycles to display output which shows the efficiency of the designed ALU.

5 Conclusion

The design and implementation of an 8-bit ALU was a success. An architecture was designed and a behavior was successfully put into practice. All requested functionality was achieved with added on features. This 8-bit ALU can now be used in further projects.