Experiment 4: ALU

Karrthik Arya Roll Number 200020068
EE-214, WEL, IIT Bombay
22 September, 2021

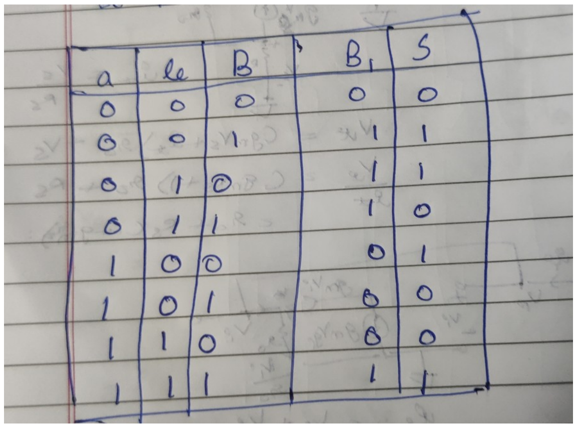
Overview of the experiment:

In this experiment we had to design an ALU which had to do 4 operations. It had 2 4 bit inputs A and B and 2 selection bits as input to choose which operation to do. The first opeartion was to rotate A's bits to the left by the number given by B[2:0]. Next operation was to subtract B from A. Then we had to do bitwise nor operation between A and B. Finally we had to find 4*A. The opeartion to do was descided by a 4x1 multiplexer.

Approach to the experiment:

We had to use behavioral modelling to for this experiment. The skeleton code was provided. We had to make 2 separate functions for subtracting and rotate left operations. The nor operation can be done easily by looping through the bits of each of A and B and taking nor between the corresponding bits. 4A corresponds to shifting the bits of A to left by 2. We can see this since multiplication by 2 shifts the bits by 1 to the left so multiplication with 4 will shift the bits by 2. So the function described for rotating left could be used.

For the subtractor function I separately calulated the diff and borrow bits by looping through the bits of A and B. For the LSB the borrow is 0. Then for the further bits we get the borrow from the calculation of the bit prior to it. We can get the functions for diff and borrow from the truth table. Diff would be equal to a xor b xor borrow. Here a and b are the corresponding bits of A and B. Borrow would be equal to $a^c c + b^c c + ba^c$. So from this we get A-B.



a	b	B	B_1	S
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

For the rotate left function I first calulated the decimal value corresponding to B[2:0]. That is the amount by which we have to rotate the bits of A. I then took 2 cases, first if this number was less than 5 when there is no need to wrap around but if the number is greater than 4 then wrap around would be needed. For the first cases I simply set the bits operand width + shift -1 to s as A and others as 0. Here operand width is the size of A i.e. 4 and shift is the number by which to shift. This would be the case because only normal left shift needs to be done.

For the second case I set the bit 2*operand width -1 to shift as the operand width – (shift -operand width) -1 to 0 bits of A. These would be the bits that don't get wrapped around. s- operand width to 0 would be equal to the rest of the bits of A. These would be the bits that got wrapped around. The other bits would be set to 0. This is how I approached the problem for the 4 operations. For the For the multiplexer we just needed if else statements to see which operation needs to be done and call the function or do the operation as described above.

Design document and VHDL code if relevant:

Here is the code for the subtractor function. The name of the function is sub(). It takes A and B 2, 4 bit vectors as input and gives diff an 8 bit vector as output.

```
function sub(A: in std_logic_vector(operand_width-1 downto 0); B: in
std_logic_vector(operand_width-1 downto 0))
        return std_logic_vector is
            -- declaring and initializing variables using aggregates
            variable diff : std_logic_vector(operand_width*2-1 downto 0):= (others=>'0');
            variable carry : std_logic:= '0';
             variable a1: std_logic_vector(operand_width*2-1 downto 0) := "0000"&A;
             variable b1: std_logic_vector(operand_width*2-1 downto 0) := "0000"&B;

        begin
            -- Hint: Use for loop to calculate value of "diff" and "carry" variable
            -- Use aggregates to assign values to multiple bits
       sub_loop : for i in   0 to operand_width*2 - 1 loop
               diff(i) := a1(i) xor b1(i) xor carry;
               carry := ((not a1(i))and carry) or (b1(i) and carry) or (b1(i) and (not a1(i)));
       end loop sub_loop;
            return diff;
    end sub;
```

The logic used is the one described above. I loop through the bits of A and B anc calulate diff and borrow(the variable is named as carry here). The boolean expressions are the one I got from the above truth table. Finally diff is given as the output.

The code for function to rotate left is the following. It also takes A and B 2, 4 bit vectors as input and gives an 8 bit vector as output.

```
function rolf(A: in std_logic_vector(operand_width-1 downto 0); B: in
std_logic_vector(operand_width-1 downto 0))
        return std_logic_vector is
            variable shift : std_logic_vector((operand_width*2)-1 downto 0):= (others=>'0');
            variable tmp : std_logic := '0';
            variable s: integer := 0;
        begin
            shift(operand_width-1 downto 0):= A;
            shifter_loop: for  i in   0 to operand_width-2 loop
               if (B(i) = '1') then
                      s := s+2**i;
               end if;
               end loop shifter_loop;
            if (s < 5) and  (s > 0) then
              shift(operand_width + s -1  downto s):= shift(operand_width-1 downto 0);
              shift( s -1  downto 0) := (others=>'0');
            elsif (s > 4) then
              shift(2*operand_width-1 downto s) := shift(2*operand_width - 1 - s downto 0);
               shift(s-operand_width-1 downto 0) := shift(operand_width-1 downto 
2*operand_width-s);
               shift(operand_width-1 downto s -operand_width) := (others =>'0');
            end if;
             return shift;
      end rolf;
```

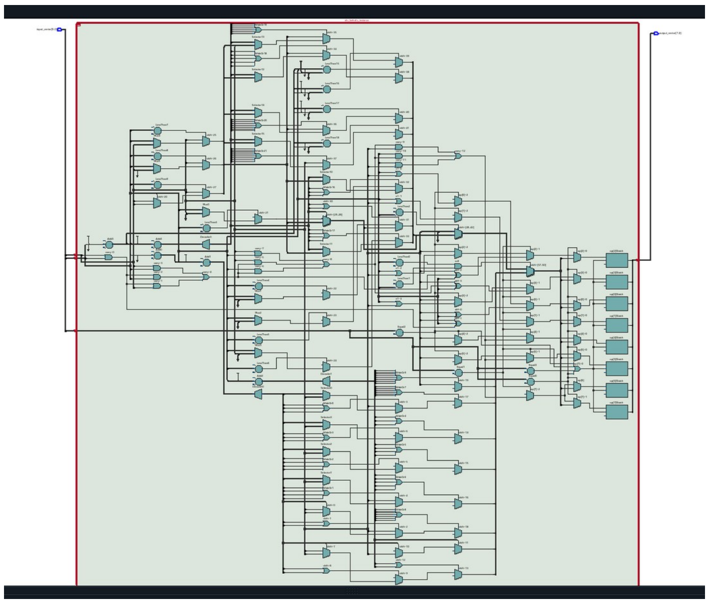
Here “s” is the decimal value of B[2:0]. This is the value by which we have to rotate the bits of A. So we first calulate the value of s using the for loop. Then take 2 cases using if else as was described earlier. Then in each case just the appropriate bits are given the values.

These are the 2 separate function that had to be made.
The nor and 4 times operation don’t need to have a separate function altogether.
Thus I coded for the multiplexer as follows:

```
alu : process( A, B, sel )
variable n1: std_logic_vector(operand_width-1 downto 0) := (others => '0');
begin
      if (sel = "00") then
            op <= rolf(A,B);
      elsif (sel = "01") then
            op <= sub(A,B);
      elsif (sel = "10") then
            loop_nor: for i in 0 to operand_width-1 loop
                  n1(i) := A(i) nor B(i);
            end loop loop_nor;
            op <= "0000"&n1;
      elsif (sel = "11") then
            op<= rolf(A, "0010");
      end if;
```

Here for sel “00” and “01” we simply need to call the 2 functions we made above. For “10” nor operation need to be done in this we loop and accordingly take the bitwise nor opration.
For option “11” we need to multiply by 4 i.e. shift the bits to left by 2. Since 2 < operand width rotate left and shift left would be the same operation thus we use the same function.

RTL View:

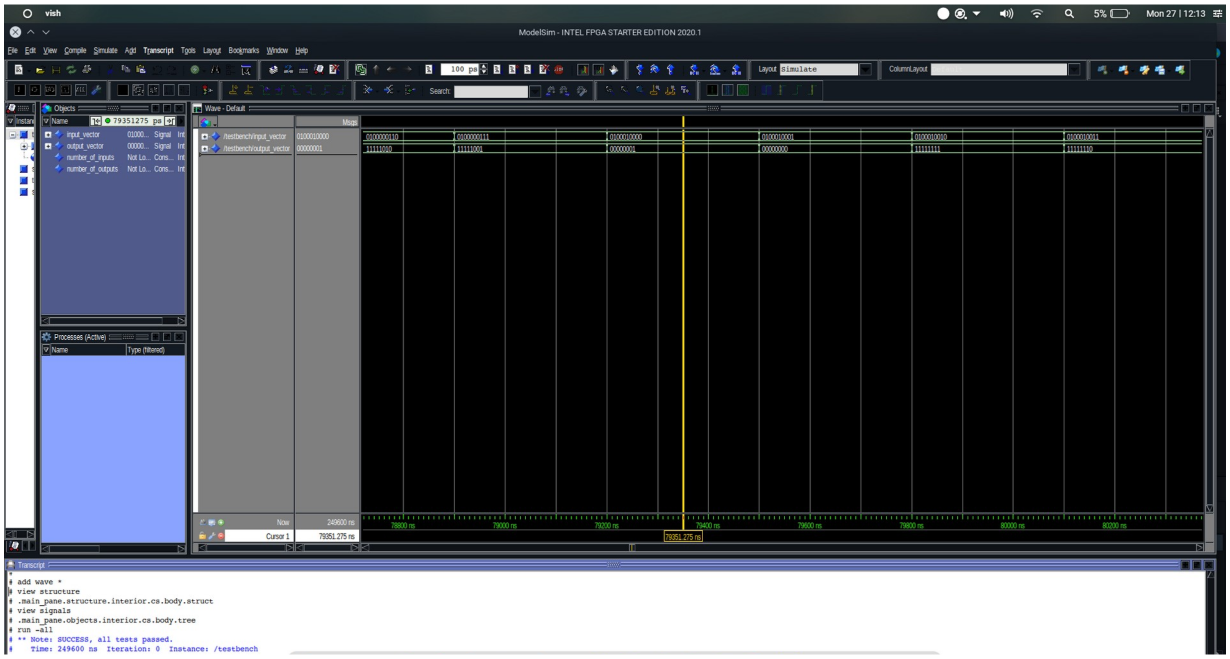


DUT Input/Output Format:

The DUT format looks like this:
S1S0A3A2A1A0B3B2B1B0 Y7Y6Y5Y4Y3Y2Y1Y0 11111111
The first 2 bits (starting from MSB) are the selections bits to decide which operation to do. The next 4 bits represent A and the last 4 represent B, the two 4 bit input vectors. Finally the output is Y a 8 bit output vector. 111...1 are the mask bits that decide which cases to test and which to reject. Here are a few of the test cases:

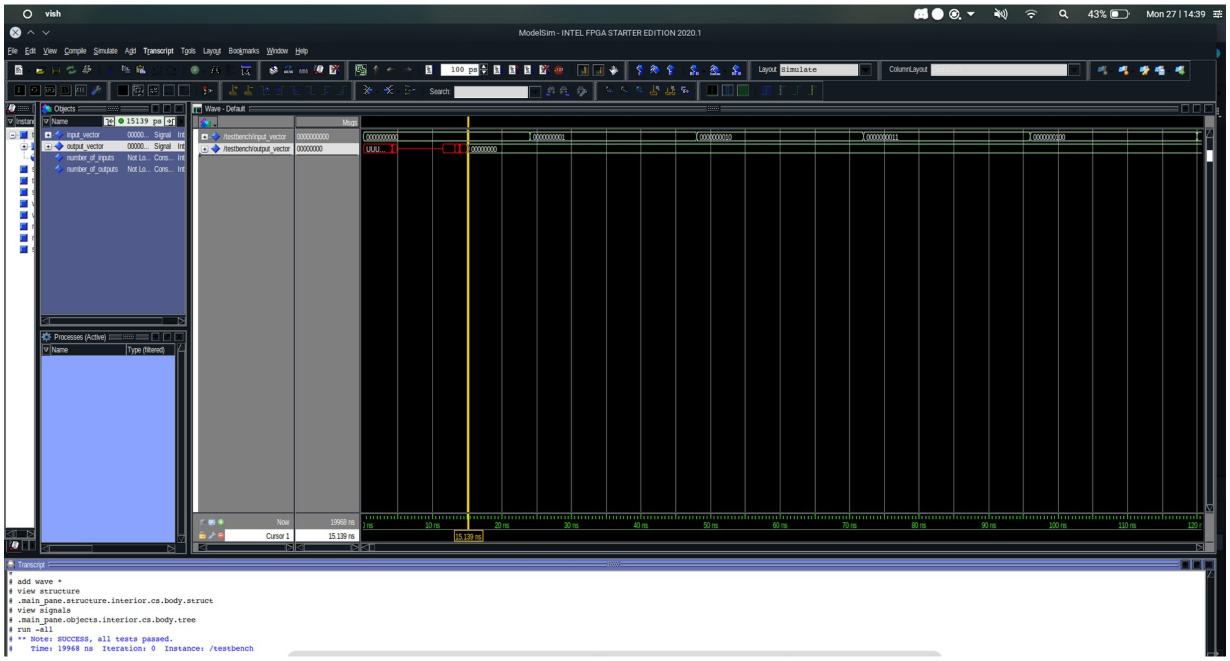
```
0000000010 00000000 11111111
0000000011 00000000 11111111
0000000100 00000000 11111111
1010101111 00000000 11111111
1010110000 00000100 11111111
1010110001 00000100 11111111
1010110010 00000100 11111111
1010110011 00000100 11111111
0100100111 11111011 11111111
0100110000 00000011 11111111
0100110001 00000010 11111111
```

RTL Simulation:



```
#
# add wave *
# view structure
# .main_pane.structure.interior.cs.body.struct
# view signals
# .main_pane.objects.interior.cs.body.tree
# run -all
# ** Note: SUCCESS, all tests passed.
#    Time: 249600 ns  Iteration: 0  Instance: /testbench
```

Gate-level Simulation:



```
#
# add wave *
# view structure
# .main_pane.structure.interior.cs.body.struct
# view signals
# .main_pane.objects.interior.cs.body.tree
# run -all
# ** Note: SUCCESS, all tests passed.
#    Time: 19968 ns  Iteration: 0  Instance: /testbench
```

Krypton board*:

I used scanchain to test the design on the Krypton board. I added the approrpriate files, set TopLevel.vhdl as the top level entity, compiled and created the svf file. I then uploaded the svf file on the Krypton board and ran the scanchain script.

Observations*:

All the testcases passed through scan chain. Here are a few of the test cases.

```
0000001100 00000000 Success
0000001101 00000000 Success
0000001110 00000000 Success
0000001111 00000000 Success
1010001101 00000010 Success
1010001110 00000001 Success
1010001111 00000000 Success
1010010000 00000110 Success
```

References:

You may include the references if any.

* To be submitted after the tutorial on ”Using Krypton.