Bitwise, Logical Shift, Arithmetic Shift, and Rotation Operations

By: VITO KLAUDIO with team member Ferley Taras
CSC343 / AB – FALL 2015 – LAB 2 – PROF. IZIDOR GERTNER
9/19/15
# Table of contents

1. Objective............................................................................................................................. pg. 2
2. Functionality and Simulation ............................................................................................... pg. 3
3. Quartus................................................................................................................................. pg. 12
4. Analysis................................................................................................................................. pg. 18
5. Conclusion............................................................................................................................. pg. 19
6. Appendix............................................................................................................................... pg. 20
1. Objective

The objective of this laboratory is to extend knowledge and understanding of the VHD Language and try to simulate and implement more complicated circuits. We will be focusing on bitwise operations. These operations are performed on a bit by bit base on the inputs. We will start with very simple VHDL code to perform an AND operation and then we will do the same the same thing for the LEFT SHIFT operation. The final result of this lab will be the design and implement a circuit that performs eight different operations based on an operation input code which will decide which operation to compute. The rest of the performed operations are the OR, XOR, NOT, RIGHT SHIFT, LEFT ROTATION, RIGHT ROTATION. What these operations do to the input bits will be explained also. We will see how the operation code relates to each operation and why we need a processing signal to perform all operations in one circuit. The operations LEFT SHIFT and RIGHT SHIFT are very interesting because they are also the multiplication and the division, respectively, of the number by two. This design will be simulated on modelSim and it will be implemented on the DE2-70 board by using the pin assignment and Quartus. We will make some analysis on our circuit to understand how it is related with high level languages. At the end we will state a conclusion to demonstrate what we learned in this lab experiment.
2. Functionality and Simulation

This part of the lab explains the design of the circuits, how they function and the simulation of their functionality. We will start with the circuit that computes the AND operations. We will be using modelSim to code and simulate our design. We open a new project and add a new file to it, the name of the file will be “bitwise.vhd”. The AND operation on is explained by the following truth table:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>AND</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The table shows that the output of an AND operation will be “1” if both inputs are “1”, otherwise it will be “0”. Since VHDL is a very commonly used language in circuit design, its libraries have included all the bitwise operations that we need to use in this lab. The VHDL code to perform AND is:

```
Library ieee;
use ieee.std_logic_1164.all;

entity bitwise is
port(
a,b: in std_logic_vector(2 downto 0);
result: out std_logic_vector(2 downto 0)
);
end bitwise;

architecture arch of bitwise is
```

BY: VITO KLAUDIO WITH TEAM MEMBER FERLEY TARAS
begin
    result <= a and b;
end arch;

In this code we declare two inputs “a” and “b” to be logic vectors from 2 to 0 which means that the inputs are 3-bit numbers. The result will also be a 3-bit number. We compile and run a simulation on this code. The result of the simulation is shown in the screenshot below:

We can see from the picture some inputs and the result of the AND operation. For example when both inputs are “000” the result is “000”. But when the inputs are “010” and “111” we can see that the result is “010”. This simulation meets our expectation of what the AND operation should do.
The next operation that we explain is the LEFT SHIFT. This operation shifts all the bits of the number one position to the left and fills the rightmost position with a zero, which is the least significant bit. The following picture explains the procedure of LEFT SHIFT:

```
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

In this case we declare only one input and one output since the operation is performed only on a single number. We choose to have a 5-bit number, which means a vector from 4 to 0. The result in this code is achieved through the “sll” command. We have to change the logic vector
input to bit vector because the “sll” keyword works only with bit vectors. We compile and simulate this code and the results are shown below:

This simulation is performed on four random numbers to check the performance of the LEFT SHIFT operation. We can see that it is correct by inspection. The first input to be tested is “10111” and the result of the operation is “01110”. The third tested input is “00110” and the LEFT SHIFT is “01100”. These tests verify that the LEFT SHIFT operation works correctly on the simulation.

Our big circuit will be implementing more than these. We will have eight operations to compute, namely the AND, OR, XOR, NOT, LEFT SHIFT, RIGHT SHIFT, LEFT ROTATION, RIGHT ROTATION. The AND and LEFT SHIFT operations were already explained, hence we will explain the rest of these operations.
The OR operation is described by the following truth table:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>OR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

This shows that the output of the OR operation will be “0” if both inputs are “0” and it will be “1” otherwise.

The XOR operation, where X stands for exclusive, results in a logical “1” if the inputs are the same and it is “0” when the inputs differ. The following truth table summarizes XOR:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The rest of the operations are performed on a single input. The NOT operation will reverse every bit of the input. This means that all “0s” will become “1s” and vice-versa. The following table explains the possible cases:

<table>
<thead>
<tr>
<th>A</th>
<th>NOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
The LEFT SHIFT operation was described earlier in this section. We have to understand that the LEFT SHIFT operation is the same as multiplying the input by 2. The RIGHT SHIFT operation takes the input and shifts every bit one position to the right, and fills the leftmost bit with a zero, in this case the most significant bit. The following picture describes this operation:

![LEFT SHIFT Operation Diagram](image)

The operations that are left discussing are the LEFT and RIGHT ROTATION. A LEFT ROTATION means that the most significant bit will be sent to the least significant bit position and then all the other bits will be shifted one position to the left. The following picture gives a visual representation of this operation:

![LEFT ROTATION Operation Diagram](image)
The opposite happens when we perform a RIGHT ROTATION, where the least significant bit becomes the most significant bit and all the bits are shifted to the right. The picture below explains:

![Diagram of right rotation](image)

All these operations will be implemented into a single circuit which will perform a specific operation based on an operation input which selects the operation. Since we have eight operations we will use an operation code of 3-bits, because \( \log_2(8) = 3 \). The full VHDL code for this circuit is as follow is shown in the appendix. We will take portions of the code while explaining how it works. In the entity method of this code we declare our inputs and outputs. We will have two 6-bit inputs “a” and “b” represented by vectors that go from 5 to 0. And the same type of vector will represent the result. The operation code which is called “op” is declared to be an input logic vector of 3 bits. The last thing declared the “start” input which is a 1-bit logical input. This input will serve as the processing signal to start the operations.

The architecture section of the code starts by processing the “start” signal. We use an if-statement to check whether the start signal is “1” or “0”. When start is “1” we proceed, otherwise we stop. The start input processing signal is used in order to have a separation between operations. If we would not use a processing signal then the result would not change after the
first operation since the inputs do not change. Therefore the start signal does the trick by going between 0 and 1 to reset the result accordingly.

After the condition of the if-statement is verified we assign our operations to the operation code by using a case statement. The following VHDL code explains how the operations are related to the input of the op code:

```vhdl
case op is
  when "000" => result <= a and b;
  when "001" => result <= a or b;
  when "010" => result <= a xor b;
  when "011" => result <= not a;
  when "100" => result <= to_stdlogicvector(to_bitvector(a) sll 1);
  when "101" => result <= to_stdlogicvector(to_bitvector(a) srl 1);
  when "110" => result <= to_stdlogicvector(to_bitvector(a) rol 1);
  when "111" => result <= to_stdlogicvector(to_bitvector(a) ror 1);
  when others => NULL;
end case;
```

We must not forget to mention that writing the code in Quartus rather than modelSim was much more helpful because of the better warning messages that we receive in this program. After we compile the code we go ahead and simulate it. The following screenshot show the results:
As we can see the code performs all the operations. We chose our first input to be “001011” and the second input to be “001101”. The result are:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>001001</td>
</tr>
<tr>
<td>OR</td>
<td>001111</td>
</tr>
<tr>
<td>XOR</td>
<td>000110</td>
</tr>
<tr>
<td>NOT</td>
<td>110100</td>
</tr>
<tr>
<td>SLL</td>
<td>010110</td>
</tr>
<tr>
<td>SRL</td>
<td>000101</td>
</tr>
<tr>
<td>ROL</td>
<td>010110</td>
</tr>
<tr>
<td>ROR</td>
<td>100101</td>
</tr>
</tbody>
</table>

Our expectations are fulfilled. The results are compatible with the theory and we can say that our circuit implementation was successful.
3. Quartus

The next step is to make sure that this circuit works on the DE2-70 board. We will do an implementation on the board. Quartus can provide the schematics of the circuit that we are implementing and the schematics for our circuit is shown below:
In this picture we can see how complicated our circuit looks in real life implementation with the logic gates, multiplexers, etc. This is shown to understand the importance of simulation. If we would not have a simulation program it would be a huge loss of resources to get this circuit running because we would have to try so many different possibilities.

For our implementation into the board we need to assign the pins that we will use. The pin assignment is as follows:

- Input 1 will be assigned the switches from SW[0] to SW[5]
- Input 2 will be assigned the switches from SW[6] to SW[11]
- Result will be displayed using LEDs from zero to five. If the LED is on the bit is “1” and “0” otherwise
- The operation code will be assigned to switches SW[15] to SW[17].
- The start processing signal will be assigned to the KEY[0] pin.

The pin assignment file is shown in the appendix section. The result of each operation is shown in the following pictures of the board taken in the lab:
This is the first operation performed on the board. Inside the orange square is where the operation code switches are located. They are switches number 15, 16 and 17. As we can see from the picture the operation code is “000” since all the switches are down. By our own convention, this code represents the logical AND operation. The first input is represented by the switches inside the green square. The switches are from switch 0 to 5 and according to their position the first input is “001011”. The second input was assigned switches from 6 to 11 which are shown inside the red square in the picture. The second input is “001101”. Since the objective of this lab is to understand how the operation code and the processing signal work, we will not change the inputs; every case will have the same inputs “a” and “b”. The KEY[0] pin is where the “start” processing signal is assigned. When the key is released it has the value of “1”. When the key is pressed it gains a value of “0” and remains in this value until the key is released. The result is represented by the green LEDs from 0 to 5. If the LED is on it means that the value of that bit is “1” and it is “0” otherwise. For the AND operation that we are performing in this first demonstration, the result is “001001”. The result is the same from the simulation and the theory. This means that the circuit is performing its task as expected.

The next operation is shown in the figure below:

![Image of circuit board with LEDs and switches labeled with operation codes and inputs]

OPCODE => "001"
INPUT 1 => "001011"
INPUT 2 => "001101"
RESULT => "001111"
The position of the switches for each signal are not going to change between operations. They are going to be the same as the first operation figure, so we don’t need squares to represent them anymore. In this picture we can see that the operation code is “001” and by our convention this is an OR operation. The inputs are the same, we just need to press and release the start key to see the new result. The result from the OR operation is “001111”. Again, the result is correct.

We can continue with the third operation, which is shown below:

This operation has code “010” which we can see from the picture. This code corresponds to the XOR operation. The inputs are the same and the result is “000110” which is correct once more.

The next operations will be shown in sequence without any explanation since the operation code, the inputs, and the results are written inside the picture and their meaning has been explained in detail in the previous figures in their corresponding colors.
OPCODE => "011"
INPUT 1 => "001011"
INPUT 2 => "001101"
RESULT => "110100"

OP CODE => "100"
INPUT 1 => "001011"
INPUT 2 => "001101"
RESULT => "010110"

OPCODE => "101"
INPUT 1 => "001011"
INPUT 2 => "001101"
RESULT => "000101"
In this experiment we proved that our circuit is correct by trying all eight combinations of the operation code and checking the result that was displayed with the green lights. Every operation was performed correctly and in accordance with the simulation and the theory.
4. Analysis

The operations that were designed and implemented in this laboratory are also available in all
the high level languages that are used to code programs. The corresponding keyword in high
level languages such as C++, Java, etc. are shown below:

AND => &    OR => |
XOR => ^    NOT => ~
SLL => <<   SRL => >>

The operations SHIFT LEFT and SHIFT RIGHT can be used to multiply or divide a number
by two. When we perform the LEFT SHIFT operation on the number “10111” we shift all the
bits one position to the left and fill the least significant bit with a zero to get the result “01110”.
If we were to multiply the same number by two, which is “10” in binary, we still get the same
result. Also we can do the same for RIGHT SHIFT which corresponds to a division by two.

If we need to multiply a binary number by a multiple of two we just have to keep shifting.
For example if we SHIFT LEFT twice the number “10111” we get the result “11100”, and we
still get the same result if we multiply it by 4, which in binary is “0100”. We can keep shifting as
much as we want to keep multiplying by multiples of two.
5. Conclusion

In this lab we achieved a better understanding of the VHD Language and how to use its libraries to perform bitwise operations. We performed eight bitwise operations in this lab. Three of which had to deal with the bits of two different inputs. The other five were dealing with only one input. The AND, OR, and XOR operations compare bit by bit two inputs. The NOT, SLL, SRL, ROL, and ROR operations change and move around bits of the same input. We realized that the SLL and SRL operations can be used as multipliers and dividers by two of an input. The more shifts we perform the more division by two we perform. We implemented the code for these operations and the processing signal and the trigger signal to modelSim and performed a simulation. The simulation results were according to the theory. Therefore the last step was to implement the circuit into the DE2-70 board using Quartus where we used pin assignment to control operations, inputs and outputs. The results from the board were consistent with the simulations and the theory. The laboratory was successful and the bitwise operations were mastered.
6. Appendix

**Bitwise.vhd**

Library ieee;

use ieee.std_logic_1164.all;

entity bitwise is

port(

a,b: in std_logic_vector(2 downto 0);

result: out std_logic_vector(2 downto 0)

);

end bitwise;

architecture arch of bitwise is

begin

result <= a and b;

end arch;

**bitshift.vhd**

library ieee;

use ieee.std_logic_1164.all;

entity bitshift is
port(

    a: in std_logic_vector(4 downto 0);

result: out std_logic_vector(4 downto 0)
);

end bitshift;

architecture arch of bitshift is
begin
result <= to_stdlogicvector(to_bitvector(a) sll 1);
end arch;

library ieee;
use ieee.std_logic_1164.all;

entity bitshift_bitvector is
port(
    a: in bit_vector(5 downto 0);
result: out bit_vector(5 downto 0)
);
end bitshift_bitvector;
architecture arch of bitshift_bitvector is
begin
result <= a sll 1;
end arch;

bitwise_operations.vhd

Library ieee;
use ieee.std_logic_1164.all;

entity lab is
port (a,b : in std_logic_vector(5 downto 0);
     op : in std_logic_vector(2 downto 0);
     start : in std_logic;
     result : out std_logic_vector (5 downto 0)
     );
end lab;

architecture arch of lab is
begin
process(start)
begin

if(start = '1') then

    case op is

    when "000" => result <= a and b;
    when "001" => result <= a or b;
    when "010" => result <= a xor b;
    when "011" => result <= not a;
    when "100" => result <= to_stdlogicvector(to_bitvector(a) sll 1);
    when "101" => result <= to_stdlogicvector(to_bitvector(a) srl 1);
    when "110" => result <= to_stdlogicvector(to_bitvector(a) rol 1);
    when "111" => result <= to_stdlogicvector(to_bitvector(a) ror 1);
    when others => NULL;

    end case;

end if;

end process;

dend arch;
pin_assignment.txt

To, Location

start, PIN_G26 – start at switch KEY0

a[0], PIN_N25 – first input bit 1 at SW0

a[1], PIN_N26 – first input bit 2 at SW1

a[2], PIN_P25 – first input bit 3 at SW2

a[3], PIN_AE14 – first input bit 4 at SW3

a[4], PIN_AF14 – first input bit 5 at SW4

a[5], PIN_AD13 – first input bit 6 at SW5

b[0], PIN_AC13 – second input bit 1 at SW6

b[1], PIN_C13 – second input bit 2 at SW7

b[2], PIN_B13 – second input bit 3 at SW8

b[3], PIN_A13 – second input bit 4 at SW9

b[4], PIN_N1 – second input bit 5 at SW10

b[5], PIN_P1 – second input bit 6 at SW11
op[0], PIN_U4 – OpCode bit 1 at SW15

op[1], PIN_V1 – OpCode bit 2 at SW16

op[2], PIN_V2 – OpCode bit 3 at SW17

result[0], PIN_AE22 – result bit 1 at LED0

result[1], PIN_AF22 – result bit 2 at LED1

result[2], PIN_W19 – result bit 3 at LED2

result[3], PIN_V18 – result bit 4 at LED3

result[4], PIN_U18 – result bit 5 at LED4

result[5], PIN_U17 – result bit 6 at LED5