Project 3: Bitwise Operations

Computer Organization CSC 34300 - EF

By: Sebastian Grygorczuk

Tabel of Content:

Objective:

The objective of this lab is to create eight bitwise operations, AND, OR, XOR, NOT, SLL, SRL, ROL, and ROR, and display their functionality on the board using six output LEDs, two six bit inputs, one three bit operations input, and a start input.

Bitwise Operations:

We need to create eight bitwise operations, meaning that the operation will look at the vector and perform normal operations such as and, or and ects, on bit by bit bases.

Bitwise AND

In this case we want to have two inputs do a AND operations in such a fashion that only when both inputs have a 1 then the output has a 1.

Input 1: 1010
Input 2: 1000
Output: 1000

Below is the code that implemented the AND functionality and the test bench code used to produce Figure 1, which proves it functionality.

```
library ieee;
use ieee.std_logic_1164.all;


entity BW_AND is
  generic(
    n: integer := 6
  );
        port(
                grygorczuk_and_1, grygorczuk_and_2: in std_logic_vector(n-1 DOWNTO 0);
                grygorczuk_output_and: out std_logic_vector(n-1 DOWNTO 0)
        );
end BW_AND;

architecture BW_AND_LOGIC of BW_AND is

begin

AND_LOOP: for i in 0 to n-1 generate
  grygorczuk_output_and(i) <= grygorczuk_and_1(i) and grygorczuk_and_2(i);
```

```
end generate;

end BW_AND_LOGIC;
```

Code 1. BW_AND

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;


entity BW_AND_TESTBENCH is
end BW_AND_TESTBENCH;

architecture BW_AND_TESTBENCH_LOGIC of BW_AND_TESTBENCH is

constant n: integer := 6;
constant m : integer := (2**n)-1;

component BW_AND is
  generic(
    n: integer := 6
  );
        port(
                grygorczuk_and_1, grygorczuk_and_2: in std_logic_vector(n-1 DOWNTO 0);
                grygorczuk_output_and: out std_logic_vector(n-1 DOWNTO 0)
        );
end component;

signal grygorczuk_X, grygorczuk_Y, grygorczuk_Z: std_logic_vector(n-1 DOWNTO 0) := (0 => '0', others => '0');
signal grygorczuk_add_one: std_logic_vector(n-1 DOWNTO 0):= (0 => '1', others => '0');

begin
  ------Instantiate the Unit Under Test (UUT)
utt: BW_AND port map(
  grygorczuk_and_1 => grygorczuk_X,
  grygorczuk_and_2 => grygorczuk_Y,
  grygorczuk_output_and => grygorczuk_Z
);

--Test Bench
Test_Bench : process
begin
  for i in 0 to m loop
    for j in 0 to m loop
    wait for 100 ps;
    grygorczuk_Y <= grygorczuk_Y + grygorczuk_add_one;
    end loop;
  grygorczuk_X <= grygorczuk_X + grygorczuk_add_one;
  end loop;
  report "End of Test";
end process;

end BW_AND_TESTBENCH_LOGIC;
```

CODE 2. BW_AND_TESTBENCH

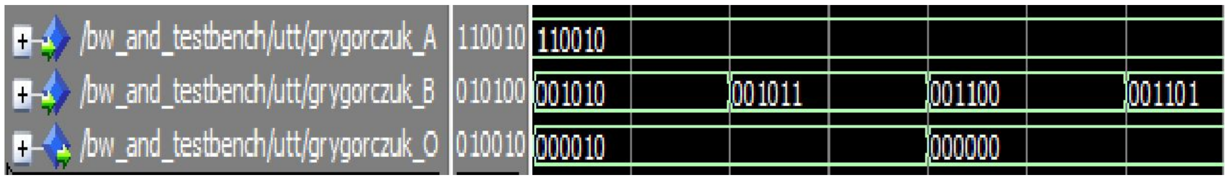

/bw_and_testbench/utt/grygorczuk_A
110010
110010
/bw_and_testbench/utt/grygorczuk_B
010100
001010
001011
001100
001101
/bw_and_testbench/utt/grygorczuk_O
010010
000010
000000

FIGURE 1. BW_AND Output

Bitwise OR

In this case we want to have two inputs do a OR operations in such a fashion that only when one of the inputs has a 1 then the output has a 1.

Input 1: 1010
Input 2: 1000
Output: 1010

Below is the code that implemented the OR functionality and the test bench code used to produce Figure 2, which proves it functionality.

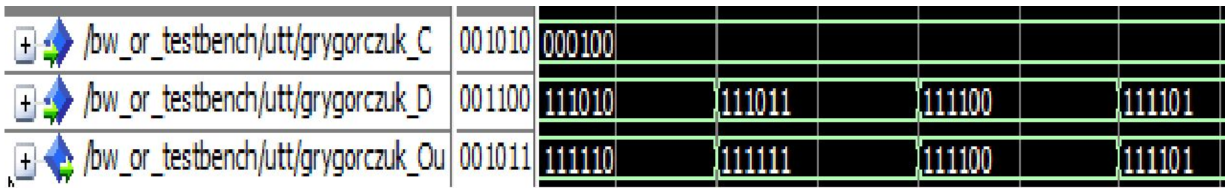
```
library ieee;
use ieee.std_logic_1164.all;


entity BW_OR is
  generic(
    n: integer := 6
  );
        port(
                grygorczuk_or_1, grygorczuk_or_2: in std_logic_vector(n-1 DOWNTO 0);
                grygorczuk_output_or: out std_logic_vector(n-1 DOWNTO 0)
        );
end BW_OR;

architecture BW_OR_LOGIC of BW_OR is

begin

OR_LOOP: for i in 0 to n-1 generate
  grygorczuk_output_or(i) <= grygorczuk_or_1(i) or grygorczuk_or_2(i);
end generate;

end BW_OR_LOGIC;
```

CODE 3. BW_OR

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;
```

```
entity BW_OR_TESTBENCH is
end BW_OR_TESTBENCH;

architecture BW_OR_TESTBENCH_LOGIC of BW_OR_TESTBENCH is

constant n: integer := 6;
constant m : integer := (2**n)-1;

component BW_OR is
  generic(
    n: integer := 6
  );
        port(
                grygorczuk_or_1, grygorczuk_or_2: in std_logic_vector(n-1 DOWNTO 0);
                grygorczuk_output_or: out std_logic_vector(n-1 DOWNTO 0)
        );
end component;

signal grygorczuk_X, grygorczuk_Y, grygorczuk_Z: std_logic_vector(n-1 DOWNTO 0) := (0 => '0', others => '0');
signal grygorczuk_add_one: std_logic_vector(n-1 DOWNTO 0):= (0 => '1', others => '0');

begin
  ------Instantiate the Unit Under Test (UUT)
utt: BW_OR port map(
  grygorczuk_or_1 => grygorczuk_X,
  grygorczuk_or_2 => grygorczuk_Y,
  grygorczuk_output_or => grygorczuk_Z
);

--Test Bench
Test_Bench : process
begin

  for i in 0 to m loop
    for j in 0 to m loop
    wait for 100 ps;
    grygorczuk_Y <= grygorczuk_Y + grygorczuk_add_one;
    end loop;
  grygorczuk_X <= grygorczuk_X + grygorczuk_add_one;
  end loop;
  report "End of Test";
end process;

end BW_OR_TESTBENCH_LOGIC;
```

CODE 4. BW_OR_TESTBENCH



```
/bw_or_testbench/utt/grygorczuk_C   001010  000100
/bw_or_testbench/utt/grygorczuk_D   001100  111010  111011  111100  111101
/bw_or_testbench/utt/grygorczuk_Ou  001011  111110  111111  111100  111101
```

FIGURE 2. BW_OR Output

Bitwise XOR

In this case we want to have two inputs do a XOR operations in such a fashion that only when the two inputs have different values then the output has a 1.

Input 1: 1010
Input 2: 1000
Output: 0010

Below is the code that implemented the XOR functionality and the test bench code used to produce Figure 3, which proves it functionality.

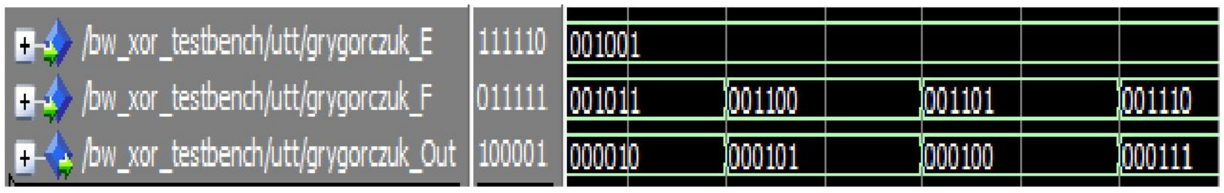
```
library ieee;
use ieee.std_logic_1164.all;


entity BW_XOR is
  generic(
    n: integer := 6
  );
        port(
                grygorczuk_xor_1, grygorczuk_xor_2: in std_logic_vector(n-1 DOWNTO 0);
                grygorczuk_output_xor: out std_logic_vector(n-1 DOWNTO 0)
        );
end BW_XOR;

architecture BW_XOR_LOGIC of BW_XOR is

begin

XOR_LOOP: for i in 0 to n-1 generate
  grygorczuk_output_xor(i) <= grygorczuk_xor_1(i) xor grygorczuk_xor_2(i);
end generate;

end BW_XOR_LOGIC;
```

CODE 5. BW_XOR

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;


entity BW_XOR_TESTBENCH is
end BW_XOR_TESTBENCH;

architecture BW_XOR_TESTBENCH_LOGIC of BW_XOR_TESTBENCH is

constant n: integer := 6;
constant m : integer := (2**n)-1;

component BW_XOR is
```

```
  generic(
    n: integer := 6
  );
              port(
                        grygorczuk_xor_1, grygorczuk_xor_2: in std_logic_vector(n-1 DOWNTO 0);
                        grygorczuk_output_xor: out std_logic_vector(n-1 DOWNTO 0)
              );
end component;

signal grygorczuk_X, grygorczuk_Y, grygorczuk_Z: std_logic_vector(n-1 DOWNTO 0) := (0 => '0', others => '0');
signal grygorczuk_add_one: std_logic_vector(n-1 DOWNTO 0):= (0 => '1', others => '0');

begin
  ------Instantiate the Unit Under Test (UUT)
utt: BW_XOR port map(
  grygorczuk_xor_1 => grygorczuk_X,
  grygorczuk_xor_2 => grygorczuk_Y,
  grygorczuk_output_xor => grygorczuk_Z
);

--Test Bench
Test_Bench : process
begin

  for i in 0 to m loop
    for j in 0 to m loop
    wait for 100 ps;
    grygorczuk_Y <= grygorczuk_Y + grygorczuk_add_one;
    end loop;
  grygorczuk_X <= grygorczuk_X + grygorczuk_add_one;
  end loop;
  report "End of Test";
end process;

end BW_XOR_TESTBENCH_LOGIC;
```

Code 6. BW_XOR_TESTBENCH



```
/bw_xor_testbench/utt/grygorczuk_E    111110  001001
/bw_xor_testbench/utt/grygorczuk_F    011111  001011  001100  001101  001110
/bw_xor_testbench/utt/grygorczuk_Out  100001  000010  000101  000100  000111
```

Bitwise NOT

In this case we want to have one inputs do a NOT operations in such a fashion that the output is the inverse of the input.

Input 1: 1010
Output: 0101

Below is the code that implemented the NOT functionality and the test bench code used to produce Figure 3, which proves it functionality.

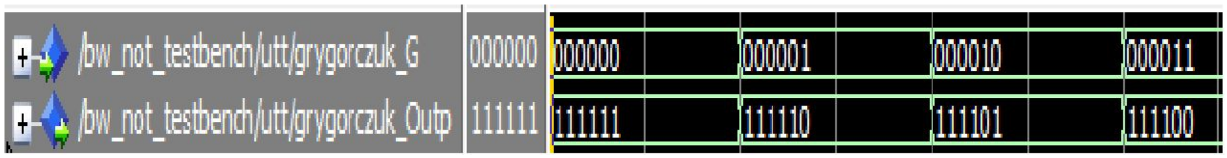
```
library ieee;
use ieee.std_logic_1164.all;

entity BW_NOT is
  generic(
    n: integer := 6
  );
        port(
                grygorczuk_not: in std_logic_vector(n-1 DOWNTO 0);
                grygorczuk_output_not: out std_logic_vector(n-1 DOWNTO 0)
        );
end BW_NOT;

architecture BW_NOT_LOGIC of BW_NOT is

begin

NOT_LOOP: for i in 0 to n-1 generate
  grygorczuk_output_not(i) <= not grygorczuk_not(i);
end generate;

end BW_NOT_LOGIC;
```

CODE 7. BW_NOT

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;


entity BW_NOT_TESTBENCH is
end BW_NOT_TESTBENCH;

architecture BW_NOT_TESTBENCH_LOGIC of BW_NOT_TESTBENCH is

constant n: integer := 6;
constant m : integer := (2**n)-1;

component BW_NOT is
  generic(
    n: integer := 6
  );
        port(
                grygorczuk_not: in std_logic_vector(n-1 DOWNTO 0);
                grygorczuk_output_not: out std_logic_vector(n-1 DOWNTO 0)
        );
end component;

signal grygorczuk_X, grygorczuk_Z: std_logic_vector(n-1 DOWNTO 0) := (0 => '0', others => '0');
signal grygorczuk_add_one: std_logic_vector(n-1 DOWNTO 0):= (0 => '1', others => '0');

begin
  ------Instantiate the Unit Under Test (UUT)
utt: BW_NOT port map(
  grygorczuk_not => grygorczuk_X,
  grygorczuk_output_not => grygorczuk_Z
);
```

```
--Test Bench
Test_Bench : process
begin

  for i in 0 to m loop
    wait for 100 ps;
    grygorczuk_X <= grygorczuk_X + grygorczuk_add_one;
    end loop;
  report "End of Test";
end process;

end BW_NOT_TESTBENCH_LOGIC;
```

Code 8. BW_NOT_TESTBENCH



```
/bw_not_testbench/utt/grygorczuk_G     000000  000000  000001  000010  000011
/bw_not_testbench/utt/grygorczuk_Outp  111111  111111  111110  111101  111100
```

Bitwise SLL

In this case we want to have one inputs do a SLL operations in such a fashion that the output is that of input shifted to the left by 1, the nth bit being disregarded and 0th bit becoming 0. We do this for the effect of having multiplication of 2^n

Input 1: 1010
Output: 0100

Below is the code that implemented the SLL functionality and the test bench code used to produce Figure 3, which proves it functionality.

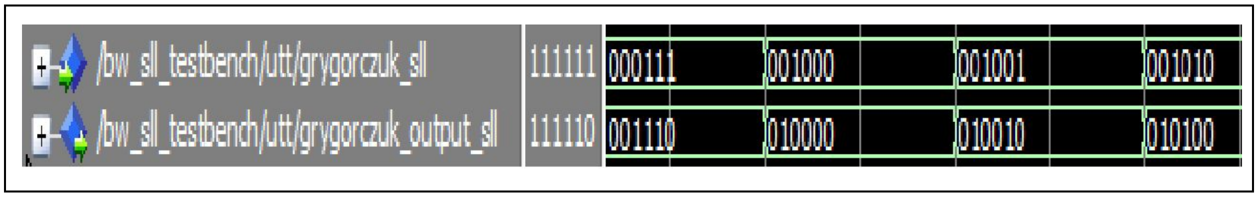
```
library ieee;
use ieee.std_logic_1164.all;

entity BW_SLL is
  generic(
    n: integer := 6
  );
        port(
                grygorczuk_sll: in std_logic_vector(n-1 DOWNTO 0);
                grygorczuk_output_sll: out std_logic_vector(n-1 DOWNTO 0)
        );
end BW_SLL;

architecture BW_SLL_LOGIC of BW_SLL is

begin
```

```
SLL_LOOP: for i in 0 to n-2 generate
  grygorczuk_output_sll(i+1) <= grygorczuk_sll(i);
end generate;
grygorczuk_output_sll(0) <= '0';

end BW_SLL_LOGIC;
```

CODE 9. BW_SLL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;


entity BW_SLL_TESTBENCH is
end BW_SLL_TESTBENCH;

architecture BW_SLL_TESTBENCH_LOGIC of BW_SLL_TESTBENCH is

constant n: integer := 6;
constant m : integer := (2**n)-1;

component BW_SLL is
  generic(
    n: integer := 6
  );
        port(
                grygorczuk_sll: in std_logic_vector(n-1 DOWNTO 0);
                grygorczuk_output_sll: out std_logic_vector(n-1 DOWNTO 0)
        );
end component;

signal grygorczuk_X, grygorczuk_Z: std_logic_vector(n-1 DOWNTO 0) := (0 => '0', others => '0');
signal grygorczuk_add_one: std_logic_vector(n-1 DOWNTO 0):= (0 => '1', others => '0');

begin
  ------Instantiate the Unit Under Test (UUT)
utt: BW_SLL port map(
  grygorczuk_sll => grygorczuk_X,
  grygorczuk_output_sll => grygorczuk_Z
);

--Test Bench
Test_Bench : process
begin

  for i in 0 to m loop
    wait for 100 ps;
    grygorczuk_X <= grygorczuk_X + grygorczuk_add_one;
    end loop;
  report "End of Test";
end process;

end BW_SLL_TESTBENCH_LOGIC;
```

Code 10. BW_SLL_TESTBENCH



```
/bw_sll_testbench/utt/grygorczuk_sll         111111  000111  001000  001001  001010
/bw_sll_testbench/utt/grygorczuk_output_sll  111110  001110  010000  010010  010100
```

Bitwise SRL

In this case we want to have one inputs do a SRL operations in such a fashion that the output is that of input shifted to the right by 1, the 0th bit being disregarded and nth bit becoming 0. We do this for the effect of having division of 2^n

Input 1: 1010
Output: 0101

Below is the code that implemented the SRL functionality and the test bench code used to produce Figure 3, which proves it functionality.

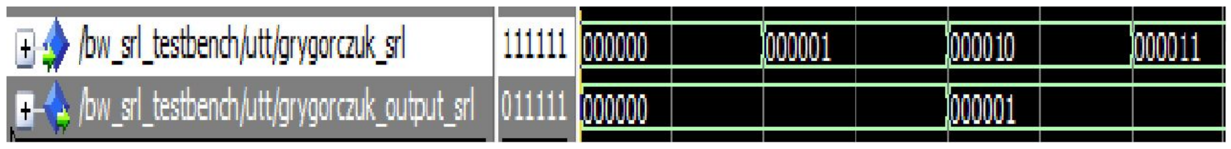
```
library ieee;
use ieee.std_logic_1164.all;

entity BW_SRL is
  generic(
    n: integer := 6
  );
        port(
                grygorczuk_srl: in std_logic_vector(n-1 DOWNTO 0);
                grygorczuk_output_srl: out std_logic_vector(n-1 DOWNTO 0)
        );
end BW_SRL;

architecture BW_SRL_LOGIC of BW_SRL is

begin

SRL_LOOP: for i in 0 to n-2 generate
  grygorczuk_output_srl(i) <= grygorczuk_srl(i+1);
end generate;
grygorczuk_output_srl(n-1) <= '0';

end BW_SRL_LOGIC;
```

CODE 11. BW_SRL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;
```

```
entity BW_SRL_TESTBENCH is
end BW_SRL_TESTBENCH;

architecture BW_SRL_TESTBENCH_LOGIC of BW_SRL_TESTBENCH is

constant n: integer := 6;
constant m : integer := (2**n)-1;

component BW_SRL is
  generic(
    n: integer := 6
  );
        port(
                grygorczuk_srl: in std_logic_vector(n-1 DOWNTO 0);
                grygorczuk_output_srl: out std_logic_vector(n-1 DOWNTO 0)
        );
end component;

signal grygorczuk_X, grygorczuk_Z: std_logic_vector(n-1 DOWNTO 0) := (0 => '0', others => '0');
signal grygorczuk_add_one: std_logic_vector(n-1 DOWNTO 0):= (0 => '1', others => '0');

begin
  ------Instantiate the Unit Under Test (UUT)
utt: BW_SRL port map(
  grygorczuk_srl => grygorczuk_X,
  grygorczuk_output_srl => grygorczuk_Z
);

--Test Bench
Test_Bench : process
begin

  for i in 0 to m loop
    wait for 100 ps;
    grygorczuk_X <= grygorczuk_X + grygorczuk_add_one;
    end loop;
  report "End of Test";
end process;

end BW_SRL_TESTBENCH_LOGIC;
```

Code 12. BW_SRL_TESTBENCH



```
/bw_srl_testbench/utt/grygorczuk_srl          111111  000000  000001  000010  000011
/bw_srl_testbench/utt/grygorczuk_output_srl   011111  000000          000001
```

Bitwise ROL

In this case we want to have one inputs do a ROL operations in such a fashion that the output is that of input shifted to the right by 1, the 0th bit being placed in the nth bit. We do this so we can test instructions non-destructively.

Input 1: 1010
Output: 0101

Below is the code that implemented the ROL functionality and the test bench code used to produce Figure 3, which proves it functionality.

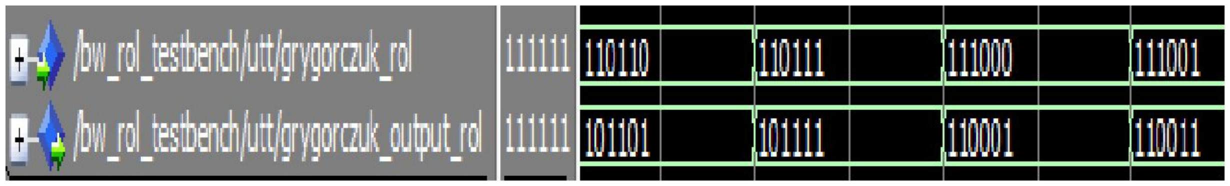
```
library ieee;
use ieee.std_logic_1164.all;

entity BW_ROL is
  generic(
    n: integer := 6
  );
        port(
                grygorczuk_rol: in std_logic_vector(n-1 DOWNTO 0);
                grygorczuk_output_rol: out std_logic_vector(n-1 DOWNTO 0)
        );
end BW_ROL;

architecture BW_ROL_LOGIC of BW_ROL is

begin

ROL_LOOP: for i in 0 to n-2 generate
  grygorczuk_output_rol(i+1) <= grygorczuk_rol(i);
end generate;
grygorczuk_output_rol(0) <= grygorczuk_rol(n-1);

end BW_ROL_LOGIC;
```

CODE 13. BW_ROL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;


entity BW_ROL_TESTBENCH is
end BW_ROL_TESTBENCH;

architecture BW_ROL_TESTBENCH_LOGIC of BW_ROL_TESTBENCH is

constant n: integer := 6;
constant m : integer := (2**n)-1;

component BW_ROL is
  generic(
    n: integer := 6
  );
        port(
                grygorczuk_rol: in std_logic_vector(n-1 DOWNTO 0);
                grygorczuk_output_rol: out std_logic_vector(n-1 DOWNTO 0)
        );
end component;
```

```
signal grygorczuk_X, grygorczuk_Z: std_logic_vector(n-1 DOWNTO 0) := (0 => '0', others => '0');
signal grygorczuk_add_one: std_logic_vector(n-1 DOWNTO 0):= (0 => '1', others => '0');

begin
  ------Instantiate the Unit Under Test (UUT)
utt: BW_ROL port map(
  grygorczuk_rol => grygorczuk_X,
  grygorczuk_output_rol => grygorczuk_Z
);

--Test Bench
Test_Bench : process
begin

  for i in 0 to m loop
    wait for 100 ps;
    grygorczuk_X <= grygorczuk_X + grygorczuk_add_one;
    end loop;
  report "End of Test";
end process;

end BW_ROL_TESTBENCH_LOGIC;
```

Code 14. BW_ROL_TESTBENCH



```
/bw_rol_testbench/utt/grygorczuk_rol         111111  110110  110111  111000  111001
/bw_rol_testbench/utt/grygorczuk_output_rol  111111  101101  101111  110001  110011
```

Bitwise ROR

In this case we want to have one inputs do a ROR operations in such a fashion that the output is that of input shifted to the right by 1, the 0th bit being placed in the nth bit. We do this so we can test instructions non-destructively.

Input 1: 1010
Output: 0101

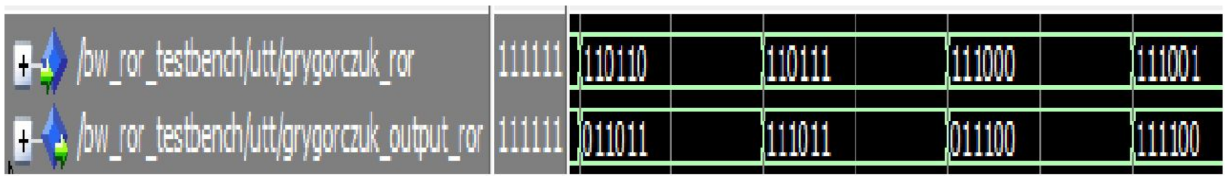
Below is the code that implemented the ROR functionality and the test bench code used to produce Figure 3, which proves it functionality.

```
library ieee;
use ieee.std_logic_1164.all;

entity BW_ROR is
  generic(
    n: integer := 6
  );
        port(
                grygorczuk_ror: in std_logic_vector(n-1 DOWNTO 0);
                grygorczuk_output_ror: out std_logic_vector(n-1 DOWNTO 0)
        );
end BW_ROR;

architecture BW_ROR_LOGIC of BW_ROR is

begin

SRL_LOOP: for i in 0 to n-2 generate
  grygorczuk_output_ror(i) <= grygorczuk_ror(i+1);
end generate;
grygorczuk_output_ror(n-1) <= grygorczuk_ror(0);

end BW_ROR_LOGIC;
```

CODE 15. BW_ROR

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;


entity BW_ROR_TESTBENCH is
end BW_ROR_TESTBENCH;

architecture BW_ROR_TESTBENCH_LOGIC of BW_ROR_TESTBENCH is

constant n: integer := 6;
constant m : integer := (2**n)-1;

component BW_ROR is
  generic(
    n: integer := 6
  );
        port(
                grygorczuk_ror: in std_logic_vector(n-1 DOWNTO 0);
                grygorczuk_output_ror: out std_logic_vector(n-1 DOWNTO 0)
        );
end component;

signal grygorczuk_X, grygorczuk_Z: std_logic_vector(n-1 DOWNTO 0) := (0 => '0', others => '0');
signal grygorczuk_add_one: std_logic_vector(n-1 DOWNTO 0):= (0 => '1', others => '0');

begin
  ------Instantiate the Unit Under Test (UUT)
utt: BW_ROR port map(
  grygorczuk_ror => grygorczuk_X,
  grygorczuk_output_ror => grygorczuk_Z
);

--Test Bench
Test_Bench : process
begin

  for i in 0 to m loop
    wait for 100 ps;
    grygorczuk_X <= grygorczuk_X + grygorczuk_add_one;
    end loop;
  report "End of Test";
end process;

end BW_ROR_TESTBENCH_LOGIC;
```

Code 16. BW_ROL_TESTBENCH


```
/bw_ror_testbench/utt/grygorczuk_ror         111111 110110 110111 111000 111001
/bw_ror_testbench/utt/grygorczuk_output_ror  111111 011011 111011 011100 111100
```

Complete Circuit:

Code:

As the objective stated we are required to have two six bit inputs, a six bit output, three bit operation code and a start code with all of this we will be able to perform all of these bitwise operations on a single board. In this project we learned a new aspect which shows the use of a if statement and cases used to determine which output should be displayed.

```
library ieee;
use ieee.std_logic_1164.all;


entity BW_CIRCUIT is
        port(
          grygorczuk_start: in std_logic;
                grygorczuk_op0, grygorczuk_op1, grygorczuk_op2: in std_logic;
                grygorczuk_x0, grygorczuk_x1, grygorczuk_x2, grygorczuk_x3, grygorczuk_x4, grygorczuk_x5: in std_logic;
                grygorczuk_y0, grygorczuk_y1, grygorczuk_y2, grygorczuk_y3, grygorczuk_y4, grygorczuk_y5: in std_logic;
                grygorczuk_o0, grygorczuk_o1, grygorczuk_o2, grygorczuk_o3, grygorczuk_o4, grygorczuk_o5: out
std_logic
        );
end BW_CIRCUIT;

architecture BW_CIRCUIT_LOGIC of BW_CIRCUIT is

constant n: integer := 6;
signal grygorczuk_input_1, grygorczuk_input_2, grygorczuk_output: std_logic_vector(n-1 DOWNTO 0) := (0 => '0', others => '0');
signal grygorczuk_and_o, grygorczuk_or_o, grygorczuk_xor_o, grygorczuk_not_o, grygorczuk_sll_o, grygorczuk_srl_o,
grygorczuk_rol_o, grygorczuk_ror_o: std_logic_vector(n-1 DOWNTO 0) := (0 => '0', others => '0');
signal grygorczuk_op_input : std_logic_vector(2 DOWNTO 0) := (0 => '0', others => '0');

component BW_AND is
  generic(
    n: integer := 6
    );
        port(
                grygorczuk_and_1, grygorczuk_and_2: in std_logic_vector(n-1 DOWNTO 0);
                grygorczuk_output_and: out std_logic_vector(n-1 DOWNTO 0)
        );
end component;

component BW_OR is
  generic(
    n: integer := 6
  );
        port(
                grygorczuk_or_1, grygorczuk_or_2: in std_logic_vector(n-1 DOWNTO 0);
                grygorczuk_output_or: out std_logic_vector(n-1 DOWNTO 0)
        );
end component;

component BW_XOR is
  generic(
    n: integer := 6
  );
        port(
                grygorczuk_xor_1, grygorczuk_xor_2: in std_logic_vector(n-1 DOWNTO 0);
                grygorczuk_output_xor: out std_logic_vector(n-1 DOWNTO 0)
        );
end component;

component BW_NOT is
  generic(
    n: integer := 6
  );
        port(
                grygorczuk_not: in std_logic_vector(n-1 DOWNTO 0);
                grygorczuk_output_not: out std_logic_vector(n-1 DOWNTO 0)
        );
end component;

component BW_SLL is
  generic(
    n: integer := 6
  );
        port(
                grygorczuk_sll: in std_logic_vector(n-1 DOWNTO 0);
                grygorczuk_output_sll: out std_logic_vector(n-1 DOWNTO 0)
        );
end component;

component BW_SRL is
  generic(
    n: integer := 6
  );
        port(
                grygorczuk_srl: in std_logic_vector(n-1 DOWNTO 0);
                grygorczuk_output_srl: out std_logic_vector(n-1 DOWNTO 0)
        );
end component;

component BW_ROL is
  generic(
    n: integer := 6
  );
        port(
                grygorczuk_rol: in std_logic_vector(n-1 DOWNTO 0);
                grygorczuk_output_rol: out std_logic_vector(n-1 DOWNTO 0)
        );
end component;
```

```
component BW_ROR is
  generic(
    n: integer := 6
  );
        port(
                grygorczuk_ror: in std_logic_vector(n-1 DOWNTO 0);
                grygorczuk_output_ror: out std_logic_vector(n-1 DOWNTO 0)
        );
end component;

begin
  grygorczuk_input_1(0) <= grygorczuk_x0;
  grygorczuk_input_1(1) <= grygorczuk_x1;
  grygorczuk_input_1(2) <= grygorczuk_x2;
  grygorczuk_input_1(3) <= grygorczuk_x3;
  grygorczuk_input_1(4) <= grygorczuk_x4;
  grygorczuk_input_1(5) <= grygorczuk_x5;

  grygorczuk_input_2(0) <= grygorczuk_y0;
  grygorczuk_input_2(1) <= grygorczuk_y1;
  grygorczuk_input_2(2) <= grygorczuk_y2;
  grygorczuk_input_2(3) <= grygorczuk_y3;
  grygorczuk_input_2(4) <= grygorczuk_y4;
  grygorczuk_input_2(5) <= grygorczuk_y5;

  grygorczuk_op_input(0) <= grygorczuk_op0;
  grygorczuk_op_input(1) <= grygorczuk_op1;
  grygorczuk_op_input(2) <= grygorczuk_op2;

  BW_AND_PART: BW_AND port map(grygorczuk_input_1, grygorczuk_input_2, grygorczuk_and_o);
  BW_OR_PART: BW_OR port map(grygorczuk_input_1, grygorczuk_input_2, grygorczuk_or_o);
  BW_XOR_PART: BW_XOR port map(grygorczuk_input_1, grygorczuk_input_2, grygorczuk_xor_o);
  BW_NOT_PART: BW_NOT port map(grygorczuk_input_1, grygorczuk_not_o);
  BW_SLL_PART: BW_SLL port map(grygorczuk_input_1, grygorczuk_sll_o);
  BW_SRL_PART: BW_SRL port map(grygorczuk_input_1, grygorczuk_srl_o);
  BW_ROL_PART: BW_ROL port map(grygorczuk_input_1, grygorczuk_rol_o);
  BW_ROR_PART: BW_ROR port map(grygorczuk_input_1, grygorczuk_ror_o);

  process (grygorczuk_start, grygorczuk_op_input)
    begin
                if (not grygorczuk_start = '1') then
      case grygorczuk_op_input is
                        when "000" =>
                                grygorczuk_output <= grygorczuk_and_o;
                        when "001" =>
                                grygorczuk_output <= grygorczuk_or_o;
                        when "010" =>
                                grygorczuk_output <= grygorczuk_xor_o;
                        when "011" =>
                                grygorczuk_output <= grygorczuk_not_o;
                        when "100" =>
                                grygorczuk_output <= grygorczuk_sll_o;
                        when "101" =>
                                grygorczuk_output <= grygorczuk_srl_o;
                        when "110" =>
                                grygorczuk_output <= grygorczuk_rol_o;
                        when "111" =>
                                grygorczuk_output <= grygorczuk_ror_o;
                        when others =>
                                NULL;
                end case;
         end if;
        end process;

  grygorczuk_o0 <= grygorczuk_output(0);
  grygorczuk_o1 <= grygorczuk_output(1);
  grygorczuk_o2 <= grygorczuk_output(2);
  grygorczuk_o3 <= grygorczuk_output(3);
  grygorczuk_o4 <= grygorczuk_output(4);
  grygorczuk_o5 <= grygorczuk_output(5);


end BW_CIRCUIT_LOGIC;
```

Code 17. Complete Circuit

The major part of the code is the if statement and cases, the if statement checks if the button is pressed, 1 is its ideal state so we have to check for zero, and the cases allow us to decide which one of the calculated values should be displayed based on the operations code that's put in. Here is the truth table that I used to set up the operations.

OP2	OP1	OP0	AND	OR	XOR	NOT	SLL	SLR	ROL	ROR
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Tabel 1. Operations Code Truth Table

Block Diagram Symbol and Waveform:

Below we see the symbol that was created using the VHDL code as well as the pin assignments used for all of the outputs and all the extra outputs used to clear the boards other LEDs and Seven Segment Displays.

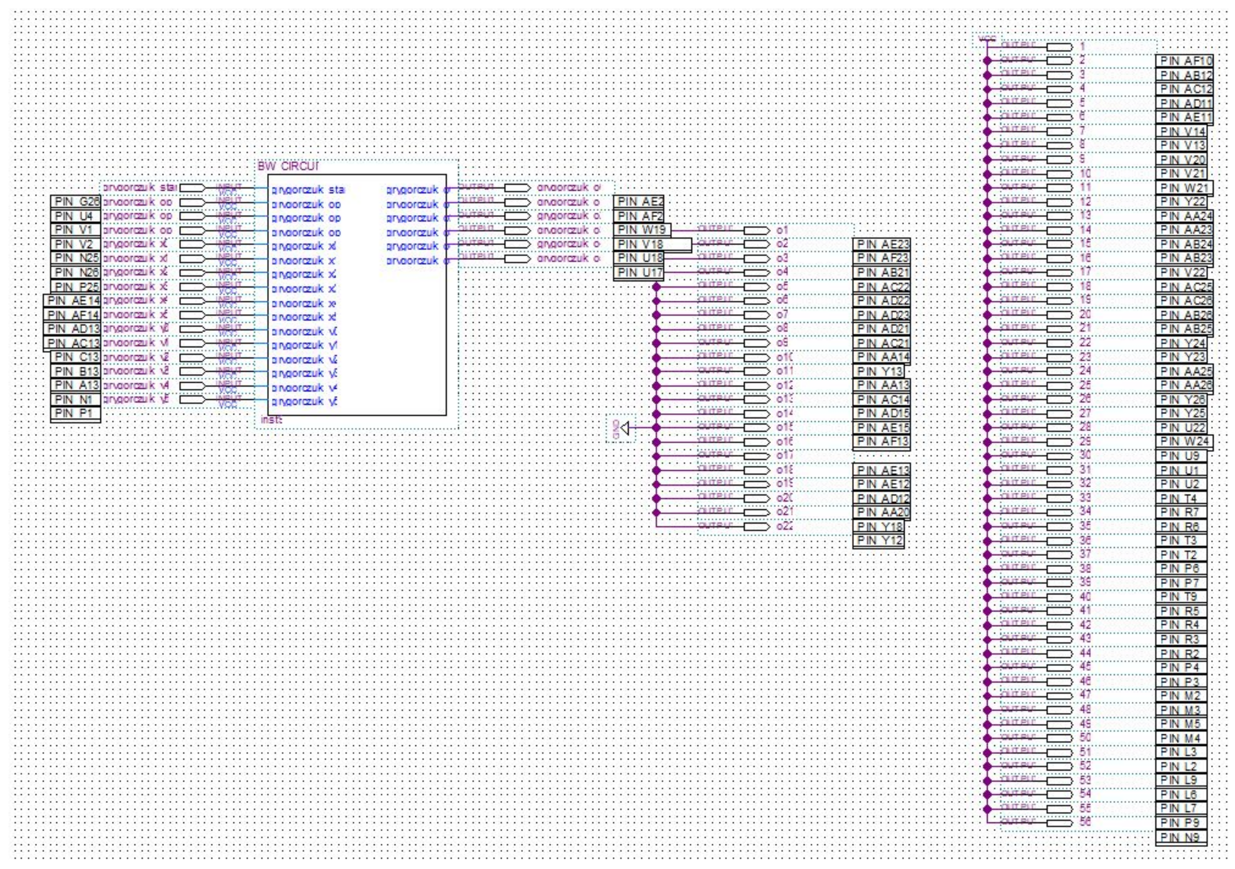


Figure 1. Bitwise Symbol

Below is the code used to set up the assignments to those pins.

```
To, Location
grygorczuk_start, PIN_G26
grygorczuk_op0, PIN_u4
grygorczuk_op1, PIN_V1
grygorczuk_op2, PIN_V2
grygorczuk_x0, PIN_N25
grygorczuk_x1, PIN_N26
grygorczuk_x2, PIN_P25
grygorczuk_x3, PIN_AE14
grygorczuk_x4, PIN_AF14
grygorczuk_x5, PIN_AD13
grygorczuk_y0, PIN_AC13
grygorczuk_y1, PIN_C13
grygorczuk_y2, PIN_B13
grygorczuk_y3, PIN_A13
grygorczuk_y4, PIN_N1
grygorczuk_y5, PIN_P1
grygorczuk_o0, PIN_AE22
grygorczuk_o1, PIN_AF22
grygorczuk_o2, PIN_W19
grygorczuk_o3, PIN_V18
grygorczuk_o4, PIN_U18
grygorczuk_o5, PIN_U17
o1, PIN_AE23
```

```
o2,  PIN_AF23
o3, PIN_AB21
o4, PIN_AC22
o5, PIN_AD22
o6, PIN_AD23
o7, PIN_AD21
o8, PIN_AC21
o9, PIN_AA14
o10, PIN_Y13
o11, PIN_AA13
o12, PIN_AC14
o13, PIN_AD15
o14, PIN_AE15
o15, PIN_AF13
o17, PIN_AE13
o18, PIN_AE12
o19, PIN_AD12
o20, PIN_AA20
o21, PIN_Y18
o22, PIN_Y12
1, PIN_AF10
2, PIN_AB12
3, PIN_AC12
4, PIN_AD11
5, PIN_AE11
6, PIN_V14
7, PIN_V13
8, PIN_V20
9, PIN_V21
10, PIN_W21
11, PIN_Y22
12, PIN_AA24
13, PIN_AA23
14, PIN_AB24
15, PIN_AB23
16, PIN_V22
17, PIN_AC25
18, PIN_AC26
19, PIN_AB26
20, PIN_AB25
21, PIN_Y24
22, PIN_Y23
23, PIN_AA25
24, PIN_AA26
25, PIN_Y26
26, PIN_Y25
27, PIN_U22
28, PIN_W24
29, PIN_U9
30, PIN_U1
31, PIN_U2
32, PIN_T4
33, PIN_R7
34, PIN_R6
35, PIN_T3
36, PIN_T2
37, PIN_P6
38, PIN_P7
39, PIN_T9
40, PIN_R5
41, PIN_R4
42, PIN_R3
43, PIN_R2
44, PIN_P4
45, PIN_P3
```

```
46, PIN_M2
47, PIN_M3
48, PIN_M5
49, PIN_M4
50, PIN_L3
51, PIN_L2
52, PIN_L9
53, PIN_L6
54, PIN_L7
55, PIN_P9
56, PIN_N9
```

Code 18. Pin Assignment

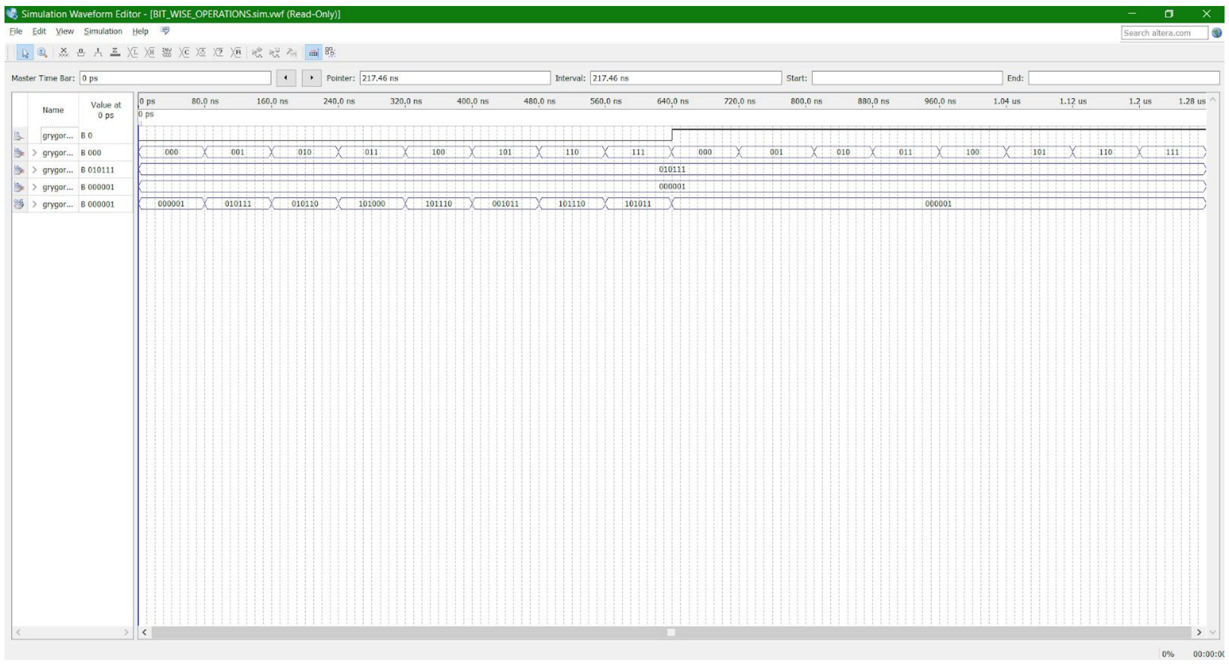


Figure Waveform

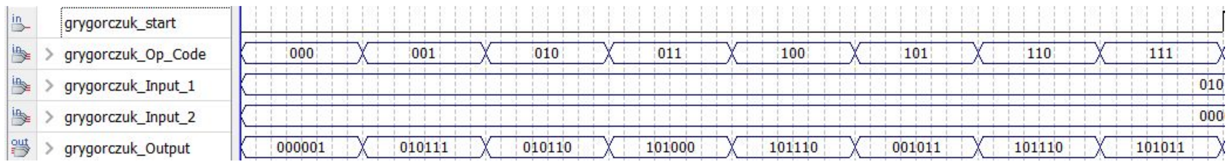


Figure Waveform Button Clicked

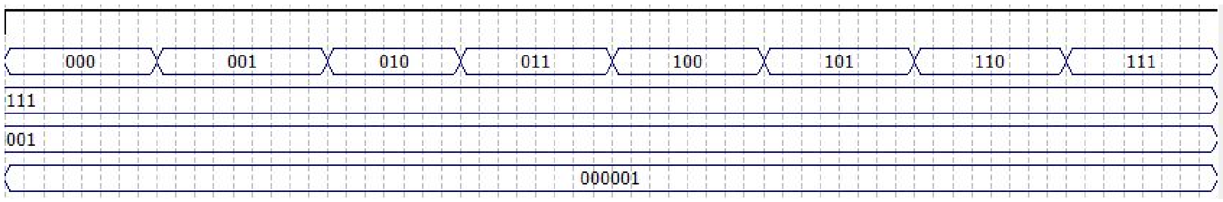


Figure Button Not Clicked

As shown in the waveforms above the signals are operational when the button is clicked and give the correct answers to their corresponding operations code, 000 gives us AND, 001 gives OR, 010 gives XOR, 011 gives NOT, 100 gives SLL, 101 gives SRL, 110 gives ROL and 111 gives ROR. Meanwhile if the button is not clicked the board is left in an idle state which leave the previously calculated number on the screen.