Muhammed Enes İnanç 22001799

10.11.2022

LAB 4: Arithmetic Logic Unit

Purpose:

Aim of the lab4 is building ALU and understanding its hierarchy. Also, the purpose of lab4 is learning how to implement a source with in another source by using VHDL. In this way we will be able to run sources at the same time by using modularity of VHDL.

What is ALU:

ALU (Arithmetic Logic Unit) is used to construct addition, subtraction, bitwise and shifting operations via basic elements of logic operators such as NOR, OR, AND, NAND etc.

Design Specifications:

By entering one or two 4-bit inputs into basys3, I collected these inputs, subtracted them, shifted and sent them to logic gates and got outputs. These functions are explained in turn.

- **Half Adder and Full Adder:**

To activate addition and subtraction functions on basys3, it should be built four bit adder and four bit subtractor. And to build them first we need to build full adder by using half adder (figure 2). Half adder can be built by using one and gate AND one XOR gate (figure 1).

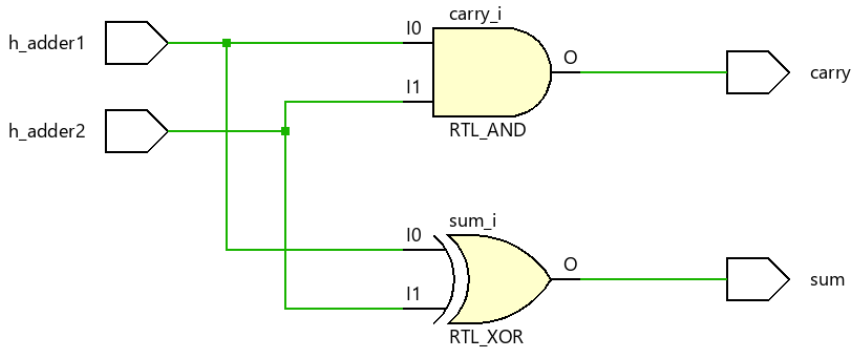

Figure 1 (Half Adder)

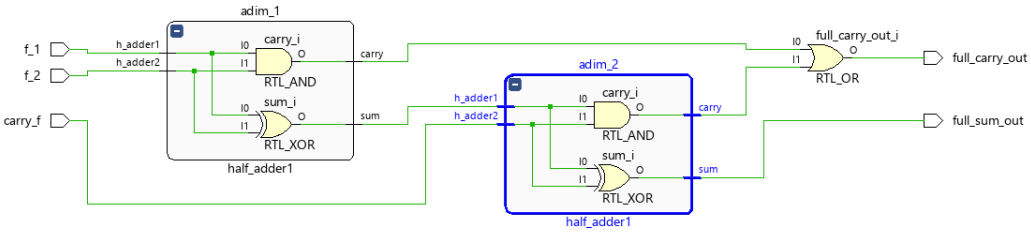

Figure 2 (Full Adder)

- **Four Bit Adder:**

To sum two four bit numbers with basys3 we should design a four bit adder. Four bit adder is comprised of four full adders. When selecting this function on basys3, we will able to sum two different four bit numbers and observe on basys3 via its leds.

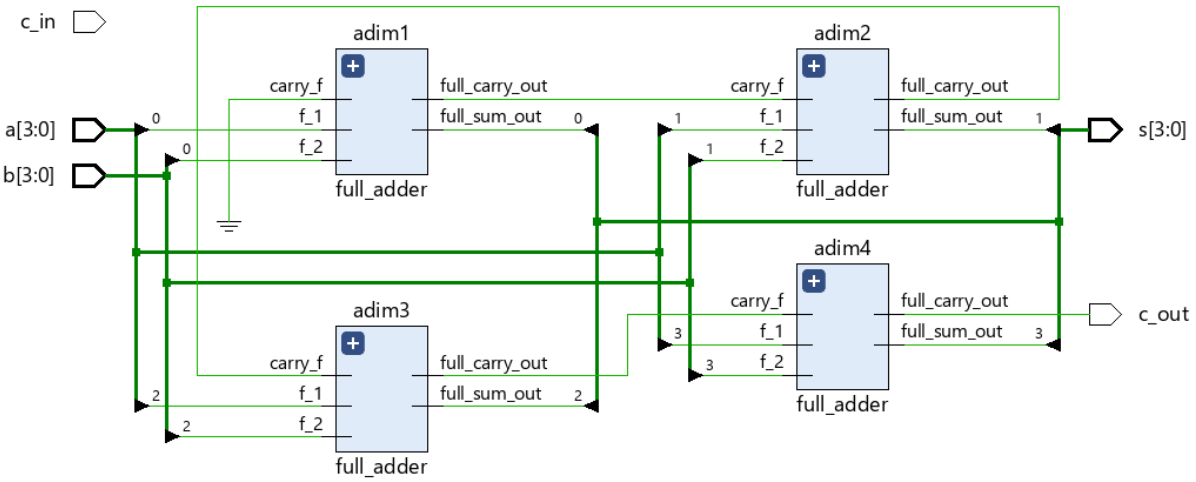

Figure 3 (Four bit Adder)

- **Four Bit Subtractor:**

Similarly, to subtract two four bit numbers with basys3 we should design a four bit subtractor. Four bit subtractor is comprised of four full adders. When selecting this function on basys3, we will able to subtract two different four bit numbers and observe on basys3 via its leds.

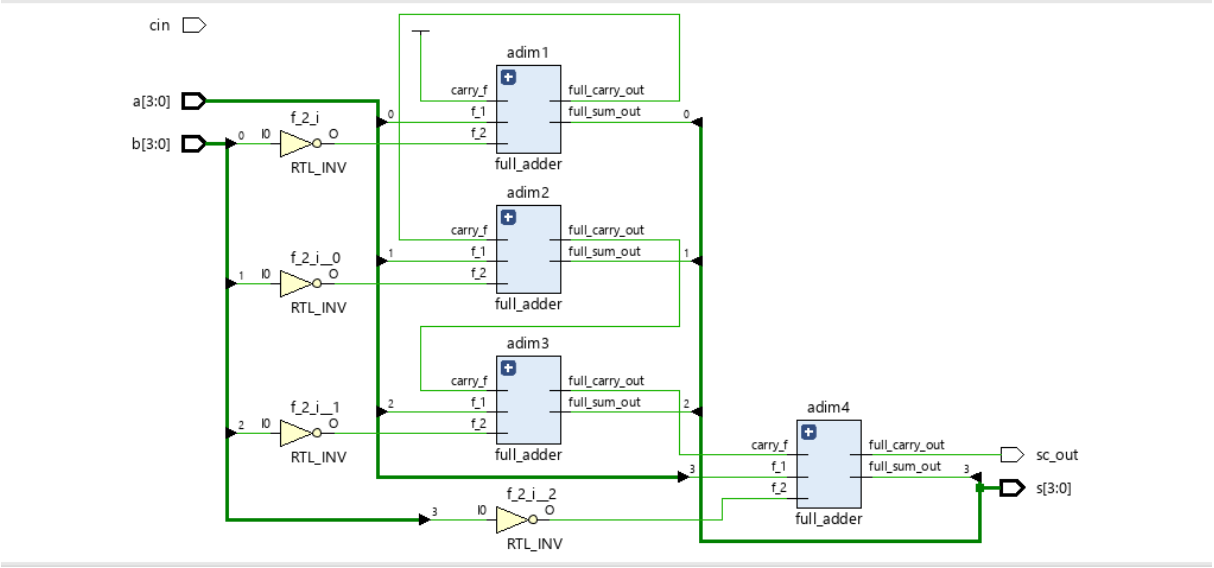

Figure 4 (Four Bit Subtractor)

- **Logical Right Shifting:**

Thanks to this function, it can be done that logical right shifting.

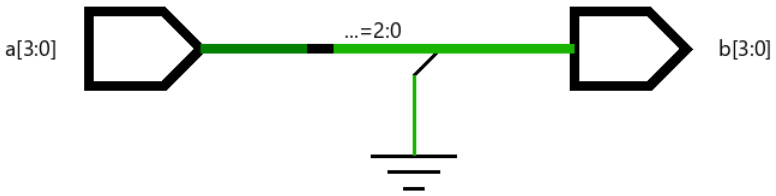

Figure 5 (Right Shifting)

- **Arithmetic Left Shifting:**

Thanks to this function, it can be done that arithmetic left shifting.

Figure 6 (Left Shifting)

- **And Gate:**

To get four bit output from and gate by using two four bit numbers.

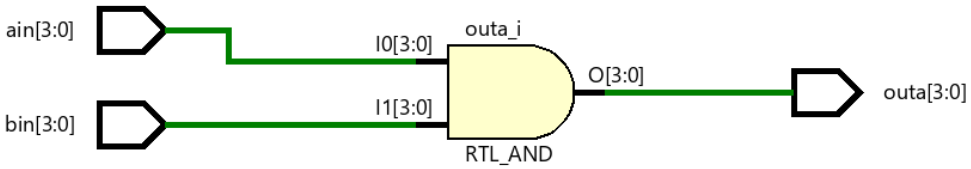

Figure 7 (And Gate)

- **Nand Gate:**

To get four bit output from nand gate by using two four bit numbers.

Figure 8 (Nand Gate)

- **OR Gate:**

To get four bit output from or gate by using two four bit numbers.

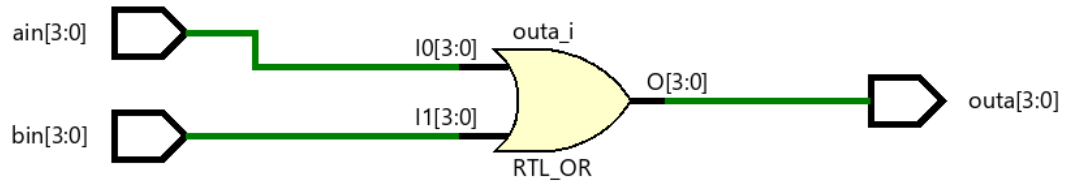

Figure 9 (Or Gate)

- **XOR Gate:**

To get four bit output from xor gate by using two four bit numbers.

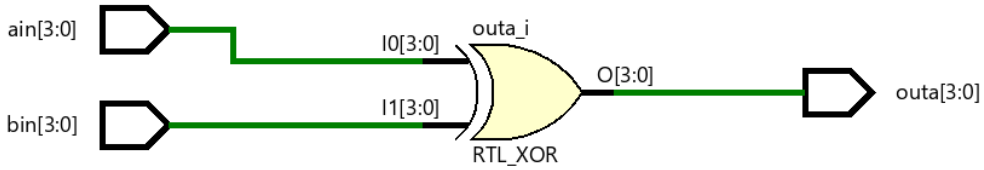

Figure 10 (Xor Gate)

Methodology:

Some functions must be selected to create an ALU. I chose the addition and subtraction of two 4-bit numbers, logical right shifting, arithmetic left shifting, and gate, nand gate, or gate, and nor gate. Thanks to the select part of the multiplexer, one of the eight functions is selected and the outputs become the outputs of the desired operation. A BASYS3 is required to select inputs and view outputs. Switches are input, LEDs are output.

Results:

Testbench:

Thanks to the features of vivado it is possible to simulate a circuit before programming it into basys3. The simulation of this code can be seen below.

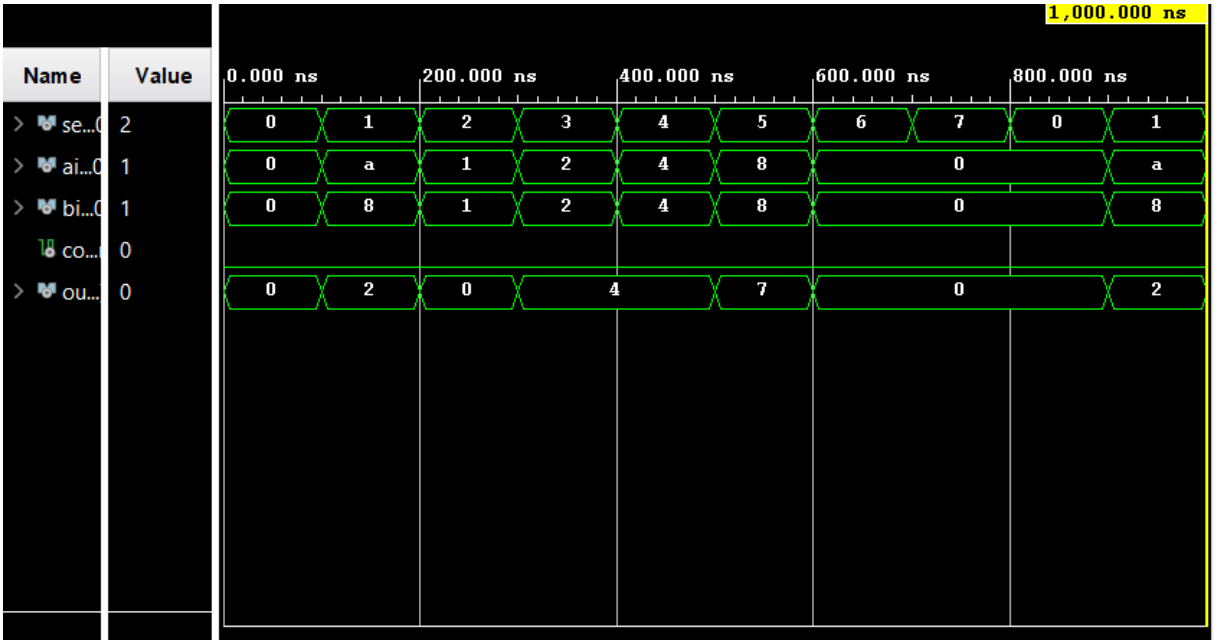

Figure 11 (Testbench)

Basys3 :

Different functions can be activated with the V2, T3 and T2 switches. The first input can be given with the V17, V16, W16 and W17 switches. Second input can be given with W15, V15, W14 and W13 switches. Their results are included below.

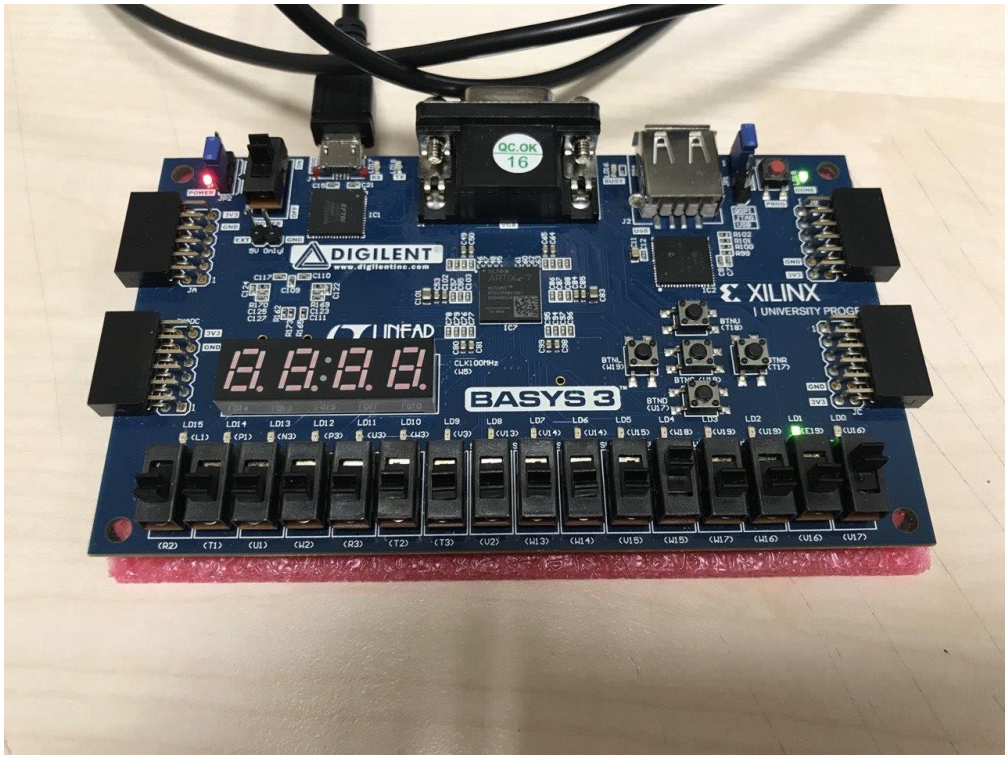

Figure 12 Adder

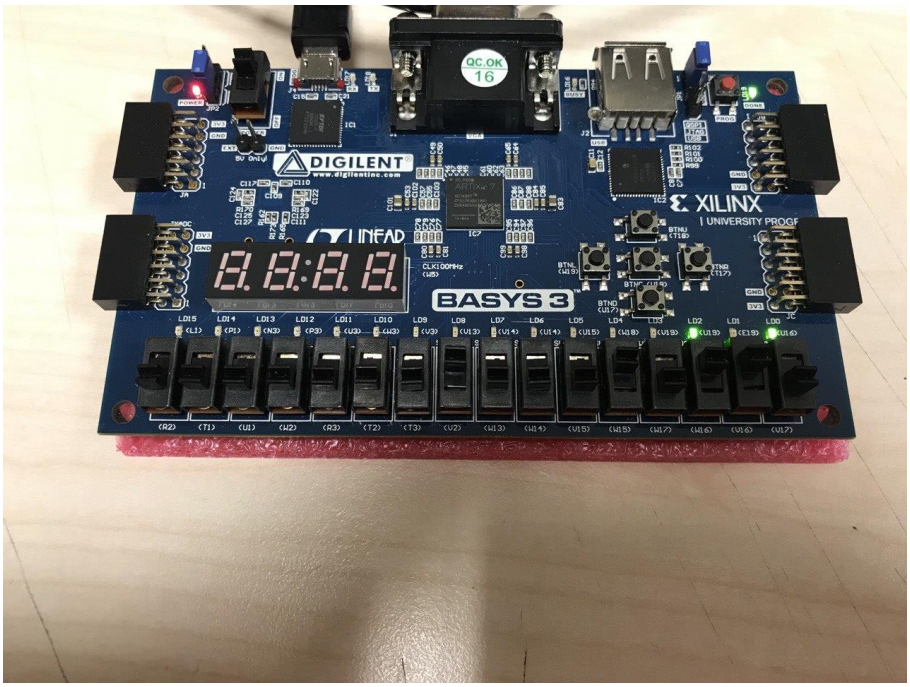

Figure 13 Subtractor

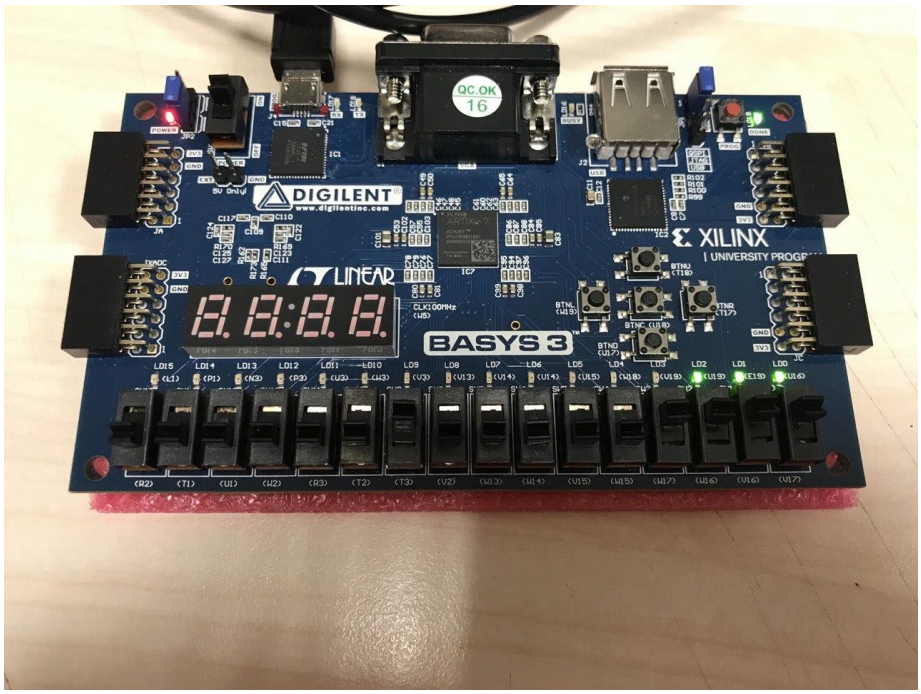

Figure 14 Right Shifting

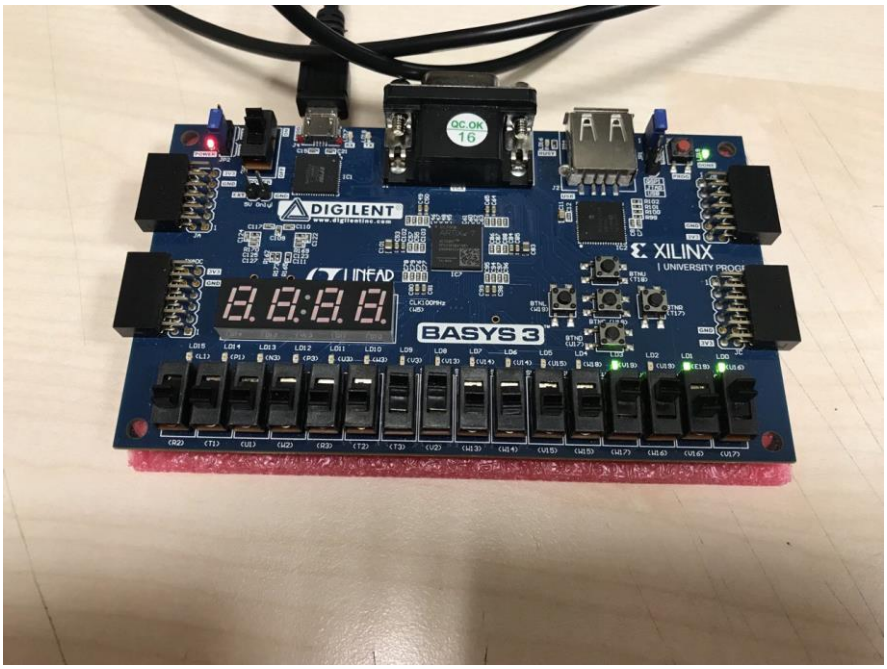

Figure 15 Left Shifting

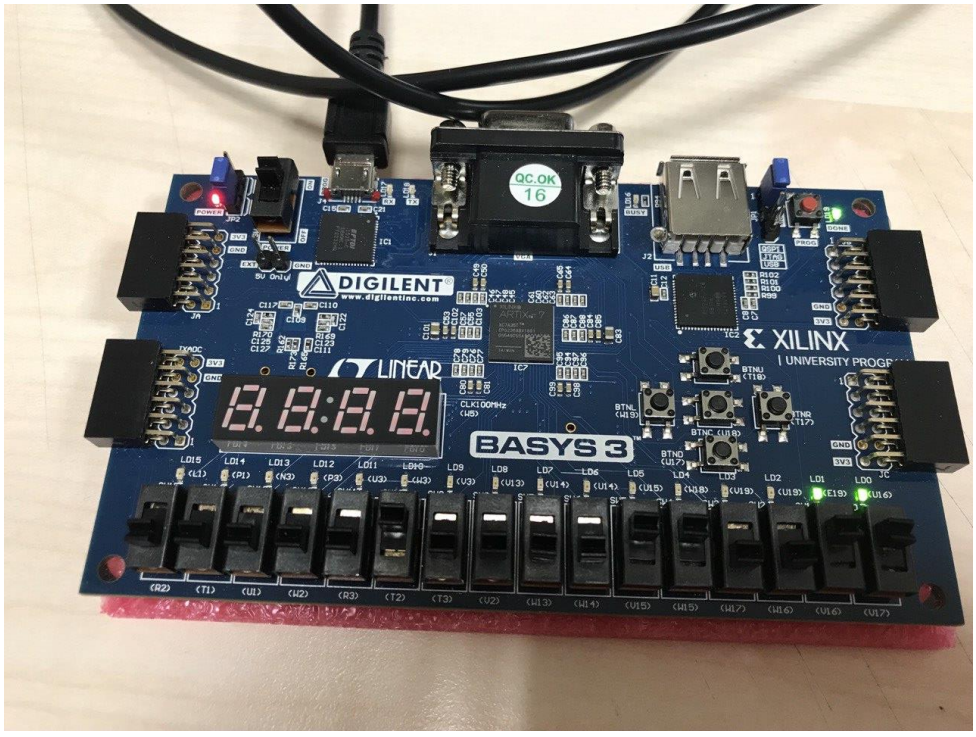

Figure 16 AND Gate

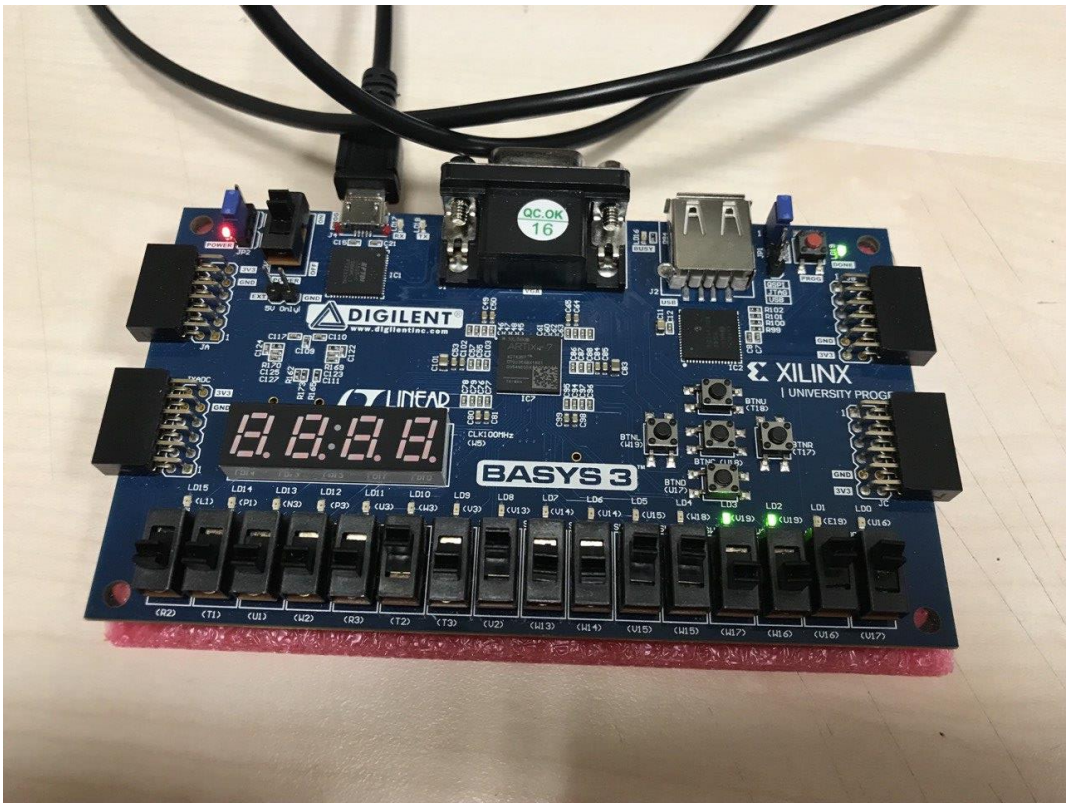

Figure 17 NAND Gate

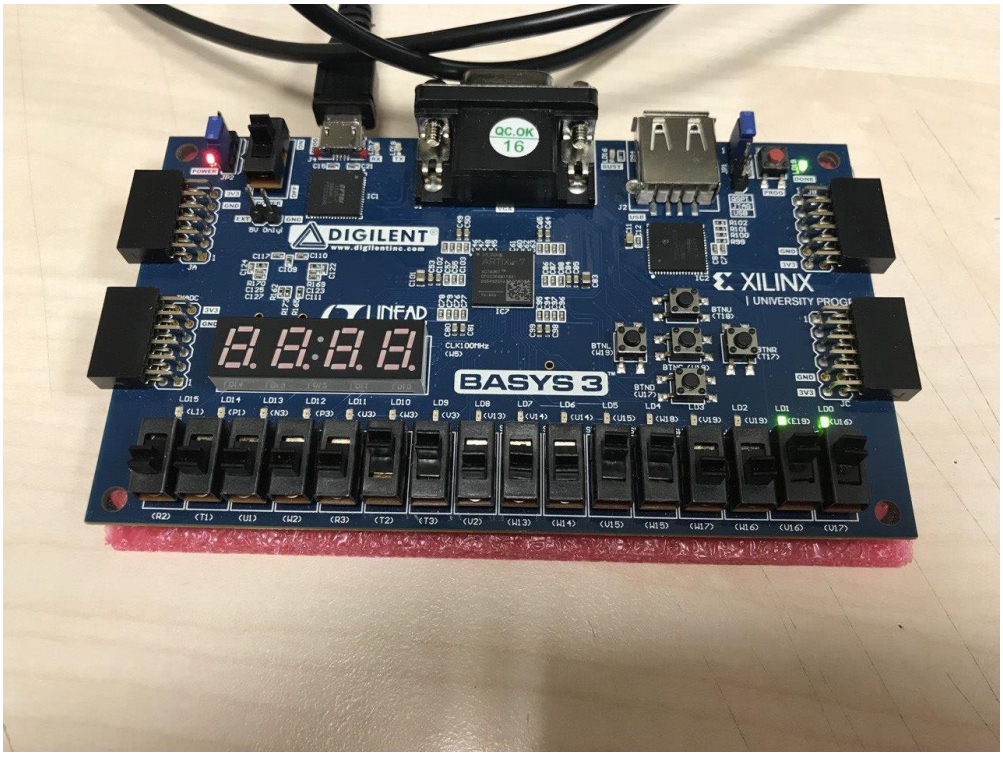

Figure 18 OR Gate

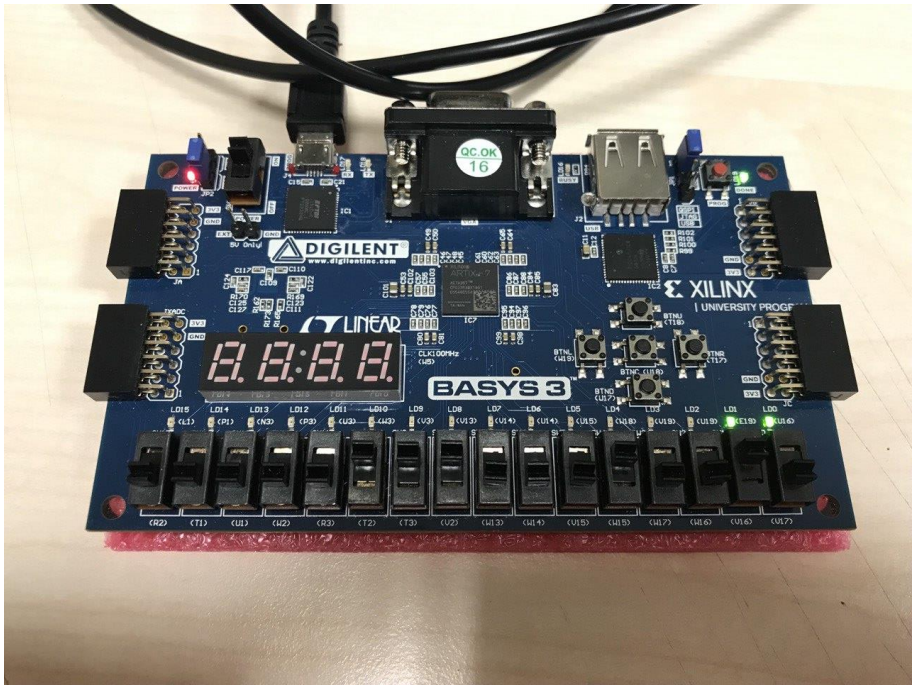

Figure 19 XOR Gate

Conclusion:

An ALU has been implemented in Basys3. In this lab, a full adder was created starting from the smallest units, namely half adders. Then, 4-bit-subtractor and 4-bit-adder were created using full adders. Operations on 4-bit numbers were observed in this lab, for example shifting and outputs of logic gates. Then, after obtaining an ALU, some conditions were written to the testbench and a simulation was obtained on vivado. This simulation is very important because it gives us advance notice of any errors. Finally, the switches to be mapped on basys3 are specified for inputs and selects. For the outputs, the leds to be mapped are specified.

Appendices:

- **Codes for half adder:**

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;



entity half_adder1 is
```

```
    Port ( h_adder1 : in STD_LOGIC;
           h_adder2 : in STD_LOGIC;
           carry : out STD_LOGIC;
           sum : out STD_LOGIC);
end half_adder1;

architecture Behavioral of half_adder1 is

begin
sum<= (h_adder1 xor h_adder2);
carry<= (h_adder1 and h_adder2);



end Behavioral;
```

- **Codes for fulladder:**

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



entity full_adder is
    Port ( f_1 : in STD_LOGIC;
           f_2 : in STD_LOGIC;
           carry_f : in STD_LOGIC;
           full_carry_out : out STD_LOGIC;
           full_sum_out : out STD_LOGIC);
```

```
end full_adder;


architecture Behavioral of full_adder is
signal sinyal1,sinyal2,sinyal3 : std_logic;
component half_adder1 port ( h_adder1 : in STD_LOGIC;
        h_adder2 : in STD_LOGIC;
        carry : out STD_LOGIC;
        sum : out STD_LOGIC);
end component;



begin
adim_1: half_adder1 port map ( f_1,f_2, carry => sinyal1 ,sum => sinyal2);
adim_2: half_adder1 port map ( h_adder2=> carry_f,h_adder1=> sinyal2, sum=>
full_sum_out, carry=>sinyal3);
full_carry_out<= sinyal1 or sinyal3;




end Behavioral;
```

- **Codes for four bit adder:**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity four_bit_adder is
   Port ( c_in : in std_logic ;
        a : in STD_LOGIC_VECTOR (3 downto 0);
```

```
        b : in STD_LOGIC_VECTOR (3 downto 0);
        s : out STD_LOGIC_VECTOR (3 downto 0);
        c_out : out STD_LOGIC);
end four_bit_adder;

architecture Behavioral of four_bit_adder is
signal sinyal_fb1, sinyal_fb2,sinyal_fb3 : std_logic;
component full_adder port ( f_1 : in STD_LOGIC;
        f_2 : in STD_LOGIC;
        carry_f : in STD_LOGIC;
        full_carry_out : out STD_LOGIC;
        full_sum_out : out STD_LOGIC);

 end component;

begin
adim1: full_adder port map( a(0),b(0),'0' ,sinyal_fb1,s(0));
adim2: full_adder port map( a(1),b(1),sinyal_fb1,sinyal_fb2,s(1));
adim3: full_adder port map( a(2),b(2),sinyal_fb2,sinyal_fb3,s(2));
adim4: full_adder port map( f_1 =>a(3) ,full_carry_out=>c_out, f_2=> b(3),carry_f
=>sinyal_fb3,full_sum_out=>s(3));

end Behavioral;
```

- **Codes for four bit subtractor:**

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


entity four_bit_sub is
    Port ( cin : in STD_LOGIC;
           a : in STD_LOGIC_VECTOR (3 downto 0);
           b : in STD_LOGIC_VECTOR (3 downto 0);
           s : out STD_LOGIC_VECTOR (3 downto 0);
           sc_out : out STD_LOGIC);
end four_bit_sub;


architecture Behavioral of four_bit_sub is
signal sinyal1, sinyal2, sinyal3 : std_logic;
--signal osinyal1,osinyal2, osinyal3, osinyal4: std_logic;
component full_adder port ( f_1 : in STD_LOGIC;
           f_2 : in STD_LOGIC;
           carry_f : in STD_LOGIC;
           full_carry_out : out STD_LOGIC;
           full_sum_out : out STD_LOGIC);
end component;



begin
--sc_out <= '1';
--osinyal1 <= ();
```

```
--osinyal2 <= ();

--osinyal3 <= ();

--osinyal4 <= ();


adim1: full_adder port map( a(0),not b(0),'1',sinyal1,s(0));

adim2: full_adder port map( a(1),not b(1),sinyal1,sinyal2,s(1));

adim3: full_adder port map( a(2),not b(2),sinyal2,sinyal3,s(2));

adim4: full_adder port map( f_1 => a(3),full_carry_out=>sc_out, f_2=>not b(3),carry_f
=>sinyal3,full_sum_out=>s(3));



end Behavioral;
```

- Codes for logical right shifting:

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;


entity right_shifting_1 is

   Port ( a : in STD_LOGIC_VECTOR (3 downto 0);

          b : out STD_LOGIC_VECTOR (3 downto 0));

end right_shifting_1;


architecture Behavioral of right_shifting_1 is


begin

b(0)<= a(1);
```

```
b(1)<=a(2);
b(2)<=a(3);
b(3)<= '0';



end Behavioral;
```

- **Codes for arithmetic left shifting:**

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

entity rotatelefts is
    Port ( a : in STD_LOGIC_VECTOR (3 downto 0);
           b : out STD_LOGIC_VECTOR (3 downto 0));
end rotatelefts;

architecture Behavioral of rotatelefts is

begin
b(0)<= a(3);
b(1)<=a(0);
b(2)<=a(1);
b(3)<= a(2);

end Behavioral;
```

- **Codes for and gate:**

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



entity andkap is
    Port ( ain : in STD_LOGIC_VECTOR (3 downto 0);
           bin : in STD_LOGIC_VECTOR (3 downto 0);
           outa : out STD_LOGIC_VECTOR (3 downto 0));
end andkap;

architecture Behavioral of andkap is

begin
outa <= ain and bin;



end Behavioral;
```

- Codes for nand gate:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity nandkap is
    Port ( ain : in STD_LOGIC_VECTOR (3 downto 0);
           bin : in STD_LOGIC_VECTOR (3 downto 0);
```

```
        outa : out STD_LOGIC_VECTOR (3 downto 0));
end nandkap;

architecture Behavioral of nandkap is

begin
outa <= ain nand bin;



end Behavioral;
```

- **Codes for or gate:**

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

entity orkap is
    Port ( ain : in STD_LOGIC_VECTOR (3 downto 0);
           bin : in STD_LOGIC_VECTOR (3 downto 0);
           outa : out STD_LOGIC_VECTOR (3 downto 0));
end orkap;

architecture Behavioral of orkap is

begin
outa <= ain or bin;
```

```
end Behavioral;
```

- Codes for nor gate:

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

entity xorkap is
    Port ( ain : in STD_LOGIC_VECTOR (3 downto 0);
           bin : in STD_LOGIC_VECTOR (3 downto 0);
           outa : out STD_LOGIC_VECTOR (3 downto 0));
end xorkap;

architecture Behavioral of xorkap is

begin
outa <= ain xor bin;



end Behavioral;
```

- **Codes for final step:**

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

entity multip is
    Port ( selects : in STD_LOGIC_VECTOR (2 downto 0);
           ain : in STD_LOGIC_VECTOR (3 downto 0);
```

```
        bin : in STD_LOGIC_VECTOR (3 downto 0);
        coutm : out std_logic;
        outa : out STD_LOGIC_VECTOR (3 downto 0));
end multip;

architecture Behavioral of multip is
signal sinyal1, sinyal2,sinyal3,sinyal4,sinyal5,sinyal6,sinyal7,sinyal8: std_logic_vector(3
downto 0);
signal osinyal1,osinyal2 : std_logic;
component four_bit_adder  Port ( c_in : in std_logic ;
        a : in STD_LOGIC_VECTOR (3 downto 0);
        b : in STD_LOGIC_VECTOR (3 downto 0);
        s : out STD_LOGIC_VECTOR (3 downto 0);
        c_out : out STD_LOGIC);

end component;

component four_bit_sub Port( cin : in STD_LOGIC;
        a : in STD_LOGIC_VECTOR (3 downto 0);
        b : in STD_LOGIC_VECTOR (3 downto 0);
        s : out STD_LOGIC_VECTOR (3 downto 0);
        sc_out : out STD_LOGIC);
end component;
component rotatelefts Port ( a : in STD_LOGIC_VECTOR (3 downto 0);
        b : out STD_LOGIC_VECTOR (3 downto 0));
end component;
```

```
component right_shifting_1 Port( a : in STD_LOGIC_VECTOR (3 downto 0);
        b : out STD_LOGIC_VECTOR (3 downto 0));
end component;

component andkap Port( ain : in STD_LOGIC_VECTOR (3 downto 0);
        bin : in STD_LOGIC_VECTOR (3 downto 0);
        outa : out STD_LOGIC_VECTOR (3 downto 0));
end component;

component nandkap Port( ain : in STD_LOGIC_VECTOR (3 downto 0);
        bin : in STD_LOGIC_VECTOR (3 downto 0);
        outa : out STD_LOGIC_VECTOR (3 downto 0));
end component;

component orkap Port( ain : in STD_LOGIC_VECTOR (3 downto 0);
        bin : in STD_LOGIC_VECTOR (3 downto 0);
        outa : out STD_LOGIC_VECTOR (3 downto 0));
end component;

component xorkap Port( ain : in STD_LOGIC_VECTOR (3 downto 0);
        bin : in STD_LOGIC_VECTOR (3 downto 0);
        outa : out STD_LOGIC_VECTOR (3 downto 0));
end component;

component notkap Port( ain : in STD_LOGIC_VECTOR (3 downto 0);
```

```
        outa : out STD_LOGIC_VECTOR (3 downto 0));
end component;

begin
case1: four_bit_adder port map ( c_in=>'0', a(0)=>ain(0) , a(1)=>ain(1), a(2)=>ain(2),
a(3)=>ain(3), b(0)=>bin(0), b(1)=>bin(1), b(2)=>bin(2), b(3)=>bin(3), s(0)=>sinyal1(0),
s(1)=>sinyal1(1),s(2)=>sinyal1(2),s(3)=>sinyal1(3),c_out=>osinyal1);
case2: four_bit_sub port map (cin=>
'1',a(0)=>ain(0),a(1)=>ain(1),a(2)=>ain(2),a(3)=>ain(3),b(0)=>bin(0),b(1)=>bin(1),b(2)=>bin(2),
b(3)=>bin(3),s(0)=>sinyal2(0),
s(1)=>sinyal2(1),s(2)=>sinyal2(2),s(3)=>sinyal2(3),sc_out=>osinyal2);
case3: right_shifting_1 port map(a =>ain, b => sinyal3 );
case4: rotatelefts port map(a =>ain, b => sinyal4 );
case5: andkap port map(ain=>ain, bin=>bin,outa=>sinyal5 );
case6: nandkap port map(ain=>ain, bin=>bin,outa=>sinyal6 );
case7: orkap port map(ain=>ain, bin=>bin,outa=>sinyal7 );
case8: xorkap port map(ain=>ain, bin=>bin,outa=>sinyal8 );

with selects select
outa <= sinyal1 when "000",
      sinyal2 when "001",
      sinyal3 when "010",
      sinyal4 when "011",
      sinyal5 when "100",
      sinyal6 when "101",
      sinyal7 when "110",
      sinyal8 when others;
with selects select
```

```
coutm <=osinyal1 when "000",
      '0' when others;



end Behavioral;
```

- **Codes for testbench:**

```
entity sims is
--  Port ( );
end sims;

architecture Behavioral of sims is
component multip port
( selects : in STD_LOGIC_VECTOR (2 downto 0);
        ain : in STD_LOGIC_VECTOR (3 downto 0);
        bin : in STD_LOGIC_VECTOR (3 downto 0);
        coutm : out std_logic;
        outa : out STD_LOGIC_VECTOR (3 downto 0));
end component ;
signal selects: std_logic_vector(2 downto 0);
signal ain: std_logic_vector(3 downto 0);
signal bin: std_logic_vector(3 downto 0);
signal coutm: std_logic;
signal outa: std_logic_vector(3 downto 0);
```

```
begin
uut: multip port map ( ain=> ain,
bin=>bin,
selects=>selects,
coutm=>coutm,
outa=>outa);
sim: process
begin
ain<= "0000";bin<= "0000";selects<= "000";
wait for  100ns;
ain<= "1010";bin<= "1000";selects<= "001";
wait for  100ns;
ain<= "0001";bin<= "0001";selects<= "010";
wait for  100ns;
ain<= "0010";bin<= "0010";selects<= "011";
wait for  100ns;
ain<= "0100";bin<= "0100";selects<= "100";
wait for  100ns;
ain<= "1000";bin<= "1000";selects<= "101";
wait for  100ns;
ain<= "0000";bin<= "0000";selects<= "110";
wait for  100ns;
ain<= "0000";bin<= "0000";selects<= "111";
wait for  100ns;
end process;
```

end Behavioral;

- **Codes for constraint:**

```
# Switches
set_property PACKAGE_PIN V17 [get_ports { ain[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ain[0]}]
set_property PACKAGE_PIN V16 [get_ports {ain[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ain[1]}]
set_property PACKAGE_PIN W16 [get_ports {ain[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ain[2]}]
set_property PACKAGE_PIN W17 [get_ports {ain[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ain[3]}]
set_property PACKAGE_PIN W15 [get_ports {bin[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {bin[0]}]
set_property PACKAGE_PIN V15 [get_ports {bin[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {bin[1]}]
set_property PACKAGE_PIN W14 [get_ports {bin[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {bin[2]}]
set_property PACKAGE_PIN W13 [get_ports {bin[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {bin[3]}]
set_property PACKAGE_PIN V2 [get_ports {selects[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {selects[0]}]
set_property PACKAGE_PIN T3 [get_ports {selects[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {selects[1]}]
```

```
set_property PACKAGE_PIN T2 [get_ports {selects[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {selects[2]}]
# LEDs
set_property PACKAGE_PIN U16 [get_ports {outa[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outa[0]}]
set_property PACKAGE_PIN E19 [get_ports {outa[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outa[1]}]
set_property PACKAGE_PIN U19 [get_ports {outa[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outa[2]}]
set_property PACKAGE_PIN V19 [get_ports {outa[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outa[3]}]
set_property PACKAGE_PIN W18 [get_ports {coutm}]
set_property IOSTANDARD LVCMOS33 [get_ports {coutm}]
```