Lab 4: Arithmetic Logic Unit

Mert Örnek EEE 102-3 22302052 15.03.2024

Purpose

The purpose of this lab session is to design an ALU program in VHDL which can perform 8 different mathematical operations. We are expected to complete the ALU in a modular fashion and display our work by using the simulation and RTL schematic features of Vivado as well as by implementing the code on the Basys 3 FPGA.

Design Specifications

The first step of designing the ALU was to pick the 8 arithmetic operations I wanted to implement. These include the two obligatory functions of “Addition” and “Subtraction”, with the other 6 operations being “Multiplication”, “Bitwise AND Gate”, “Bitwise OR Gate”, “Bitwise XOR Gate”, “Rotate Right” and “Shift Logical Right”. For the first 6 of these functions, the logic circuit will have two inputs consisting of 4-bits and for the latter two it is to have a single 4-bit input. Each of these operations is to have a 4-bit output, as a 4-bit ALU is supposed to have inputs and outputs consisting of 4-bits by definition. Therefore, resulting numbers carried beyond 4-bits are not observable (in other words, overflow takes place).

The VHDL code for the ALU was constructed in modular fashion. The “top_module.vhd” is the top module of the program and receives input from 11 switches: switches 0 to 3 is input 1, switches 4 to 7 is input 2 and switches 8 to 10 control the type of operation being implemented. The top module sends output signals to 7 LEDs: LEDs 0 to 3 are the output and 4 to 6 light up to visualize the type of operation being implemented. The top module connects to 8 individual sub modules for each one of the 8 operations: “4_Bit_Adder.vhd”, “4_bit_subtraction.vhd”, “Multiplication.vhd”, “bitwise_and.vhd”, “bitwise_or.vhd”, “bitwise_xor.vhd”, “Rotate Right.vhd” and “Shift Logical Right.vhd”. The submodules responsible for addition, subtraction and multiplication also have their own submodules: 4-bit Adder uses the “Full Adder.vhd” submodule 4 times under the names “adder1” to “adder4”, 4-bit Subtraction module is made up of two “4_Bit_Adder.vhd” submodules (named twoscomplementer and resultfinder) and their related submodules, the 4-bit Multiplication submodules uses the “4_Bit_Adder.vhd” submodule three times (named “ad1” through "ad3”).

4-bit addition (input 1 + input 2) is implemented by using four full adders and connecting the “carry” of each bitwise addition to the next bit, with the fourth carry being open as we are trying to obtain a 4-bit output. Subtraction (input 1 – input 2) essentially converts the subtrahend to two’s complement by adding “1” to its inverse, and then adds the two values to obtain a solution. Multiplication (input 1 x input 2) uses the inputs to generate four 4-bit logic vectors with each progressively slipping one bit and adds the four vectors together by using the “4-bit addition” method three times. The 4-bit bitwise AND (input 1 AND input 2), OR(input 1 OR input 2) and XOR (input 1 XOR input 2) gates each take two 4-bit inputs and output a 4-bit value by performing the specified operation on the inputs’ corresponding bits (index 0-to-0, index 1-to-1 etc.). 4-bit rotate right (input 1 ROTATE RIGHT) and shift logical right functions(input 1 SHIFT LOGICAL RIGHT) were implemented by assigning the digits to the outputs through the methodology each type of function provides (Figure 1).(See Table 1)

Operation Type [s_type / i_SW(8 to 10)]	Inputs / Outputs	Example of Operation(in: in1 , in2)
1-) 4-bit addition / “000”	i_in1 & i_in2 / o_output	in: “0011” , ”0101” out: “1000”
2-) 4-bit subtraction / “001”	i_valone & i_valtwo / o_out	in: “1101” , ”0111” out: “0110”
3-) 4-bit multiplication / “010”	I_i1 & i_i2 / o_outp	in: “0011” , ”0100” out: “1100”
4-) 4-bit bitwise AND gate / “011”	i_in1 & i_in2 / o_out1	in: “0011” , ”0101” out: “0001”
5-) 4-bit bitwise OR gate / “100”	i_in1 & i_in2 / o_out1	in: “0011” , ”0101” out: “0111”
6-) 4-bit bitwise XOR gate / “101”	i_in1 & i_in2 / o_out1	in: “0011” , ”0101” out: “0110”
7-) 4-bit rotate right / “110”	i_in1 / o_out1	in: “1001” out: “1100”
8-) 4-bit shift logical right / “111”	i_in1 / o_out1	in: “1001” out: “0100”

Table 1: Implemented Operation Types

Methodology

The task is to code an ALU which can perform 8 different arithmetic operations. This ALU receives two 4-bit inputs and gives a 4-bit output depending on the chosen operation. From a methodological standpoint, this program was designed by first building the fundamental modules such as a “one bit adder”. By using these smaller building blocks, larger modules containing these arithmetic operations were achieved and all 8 operations were connected to a single “top module”. After the program was completed, its simulation results and RTL schematic were observed. The program was also implemented on a Basys 3 FPGA and observed via the relationship between the switch (SW) inputs and LED light(LED) outputs.

Results

The ALU design described in the “Design Specifications” was implemented in the same described modular fashion. Once the design was completed, the following RTL schematics were generated:

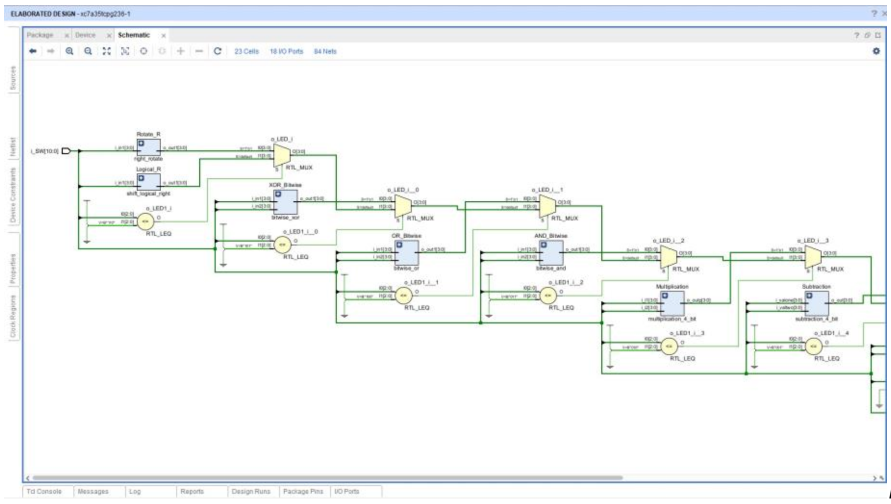

Figure 2.1: RTL Schematics (Left Side)

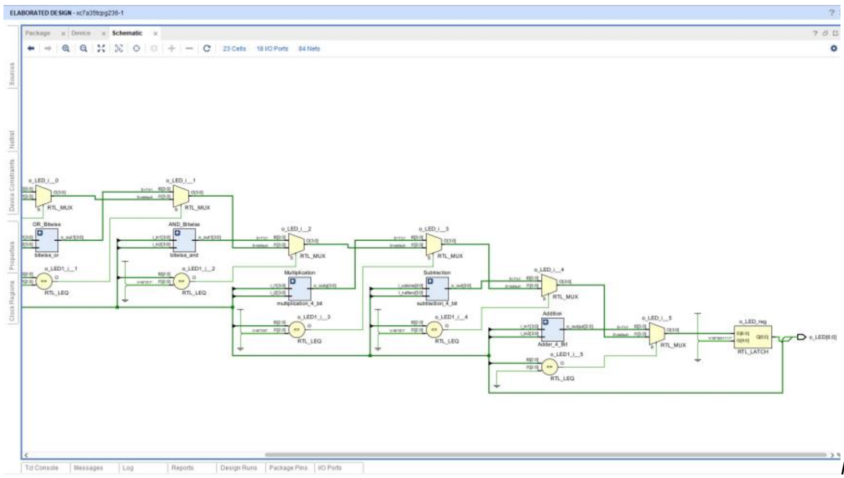
Figure 2.2: RTL Schematics (Right Side)

The schematics display the modular structure of the design and how each module connects to other modules and inputs/outputs. These include both the connections on the top module as well as the ones found in the modules that also themselves have modules such as addition, subtraction and multiplication (Figures 2.4, 2.5 and 2.6). These modules are good examples of "layered architecture" as -depending on their functionality- they utilize other modules such as full adders and -in multiplications case- the 4-bit addition method. This form of program structure makes the system easier to implement and understand by dividing the program into smaller sections that maintain similar functionalities (such as the half adder used in both addition and subtraction). The RTL schematics mark modules of lower levels by using a blue rectangle with black outlines, whose functions can be exposed further by using the "+" icon on the top left of the rectangle icon.

The program was simulated by using 24 different combinations of switch (i_SW/s_SW) inputs. The results of this simulation can be observed:

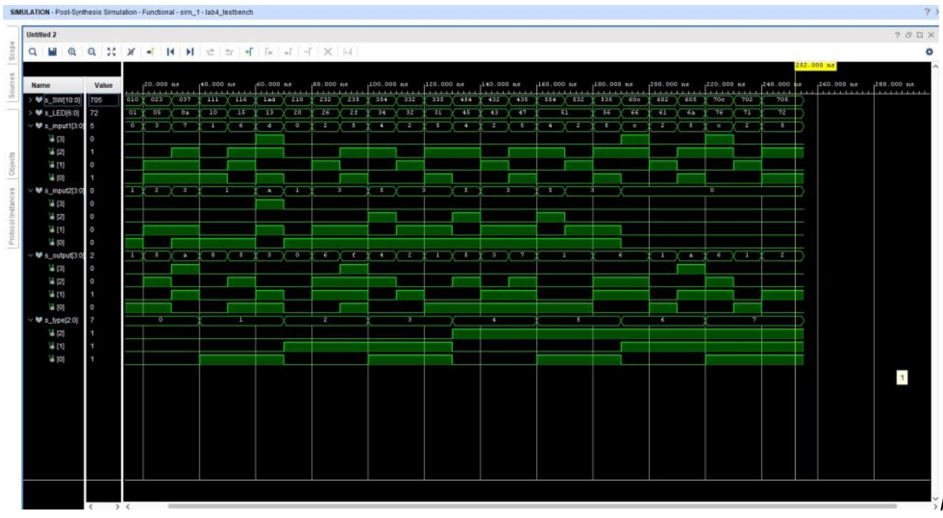
Figure 3: Simulation results

The simulation results (Figure 3) represent 3 different input combinations (s_input1 and s_input2) for each of the 8 different functionalities (s_type) of the ALU program and their respective outputs (s_output). The type of implemented simulation was "Post-Synthesis Functional Simulation".

For the implementation stage of the program, a BASYS 3 FPGA was utilized with the following assigned pins:

i_SW (from index 0 to 10) : V17, V16, W16, W17, W15, V15, W14, W13, U1, T1, R2

o_LED (from index 0 to 6) : U16, E19, U19, V19, N3, P1, L1

i_SW (index 0 to 3) : input 1 *i_SW (index 4 to 7) : input 2* *i_SW (index 8 to 10) : operation type*

o_LED (index 0 to 3) : output *o_LED (index 4 to 6) : operation type indicator*

Images for the following implementations can be found in the appendices section (Figures 4.1 to 4.24). Each type of operation was trialed 3 times. (3x8=24 combinations in total)

Operation Type	Input 1	Input 2	Output (Implemented)	Output (Simulated)
“000” Addition	0001	0000	0001	0001
“000” Addition	0011	0010	0101	0101
“000” Addition	0111	0011	1010	1010
“001” Subtraction	0001	0001	0000	0000
“001” Subtraction	0110	0001	0101	0101
“001” Subtraction	1101	1010	0011	0011
“010” Multiplication	0000	0001	0000	0000
“010” Multiplication	0010	0011	0110	0110
“010” Multiplication	0101	0011	1111	1111
“011” AND	0100	0101	0100	0100
“011” AND	0010	0011	0010	0010
“011” AND	0101	0011	0001	0001
“100” OR	0100	0101	0101	0101
“100” OR	0010	0011	0011	0011
“100” OR	0101	0011	0111	0111
“101” XOR	0100	0101	0001	0001
“101” XOR	0010	0011	0001	0001
“101” XOR	0101	0011	0110	0110
“110” Rotate R.	1100	x	0110	0110
“110” Rotate R.	0010	x	0001	0001
“110” Rotate R.	0101	x	1010	1010
“111” Shift Logical R.	1100	x	0110	0110
“111” Shift Logical R.	0010	x	0001	0001
“111” Shift Logical R.	0101	x	0010	0010

Every single one of the simulated implementations’ outputs matches the simulation waveform we generated. Therefore, we can conclude that the ALU is functioning as intended based on our observations.

Conclusion

In this lab, we designed and implemented an ALU and observed its functioning through tools such as RTL schematics, simulating waveforms and Basys 3 FPGA implementation via bitstream. The design utilized either two or a single 4-bit input and outputs a 4-bit value as well due to the definition of a 4-bit ALU implementation necessitating that both the input and output have 4-bits. This creates the issue of overflow for certain input combinations. An output of 5 or more bits could have helped overcome this issue, however that sort of implementation would not be a 4-bit ALU design by definition. An ALU design of 5 or more bits on the other hand could compute these larger values that a 4-bit ALU cannot while still encountering the issue of overflow for certain larger input combinations for that sort of circuit. There were not any major problems during the experiment process outside of the necessity for a specific type of simulation implementation. The default sort of simulation assigned by Vivado is “Behavioral Simulation” which, when run, gave all outputs as “U”s, which means that the outputs of the program were uncertain possibly due to a variety of reasons such as the timing of certain modules not being parallel or one of the logic functions having an unspecified input/output. The former of these reasons is the more likely one as the testbench ran without issue under the “Post-Synthesis Functional Simulation” mode, which means the code written had no logical errors. This possibility and its solution could be added as a “note” to Lab 4 instructions. Outside of that, the lab session went with minimal issues. One thing I learned was that VHDL does not allow for modules to be defined and implemented during processes, which necessitated the use of signal assignments for the output of each submodule included in the top module. I also gained valuable experience when it comes to implementing logic designs in an architecturally layered manner in VHDL, which was one of the objectives of the experiment. On these grounds, the experiment can be considered a success as it has reached its initial goals.

Appendices

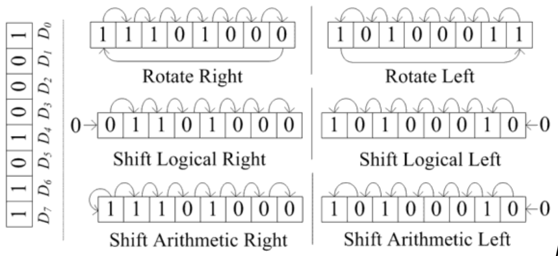

Figure 1: Types of shift operations [1]

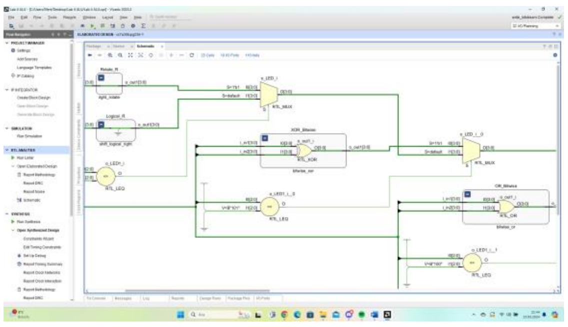

Figure 2.3: Exposed submodules 1

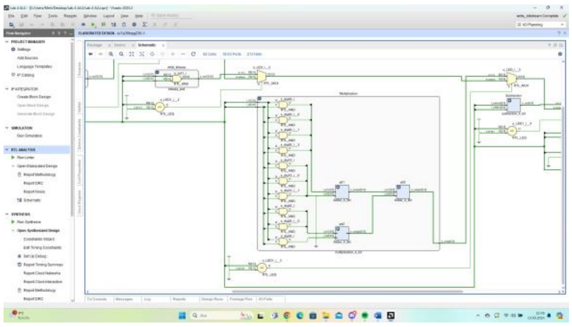

Figure 2.4: Exposed submodules 2

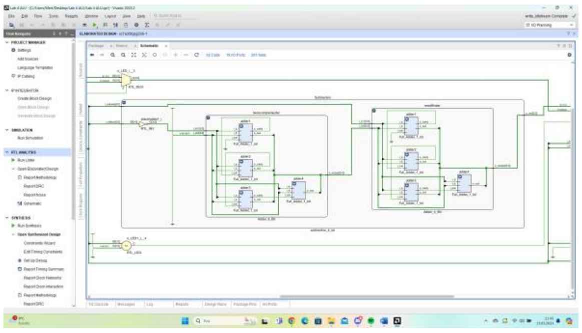

Figure 2.5 : Exposed submodules 3

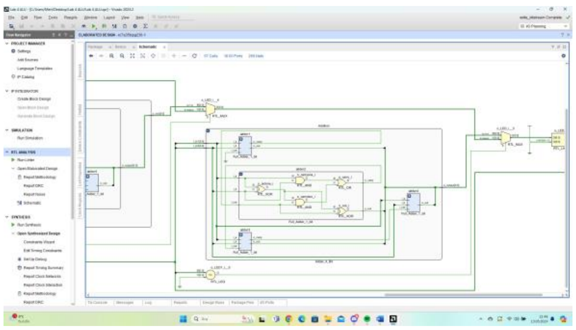

Figure 2.6: Exposed submodules 4

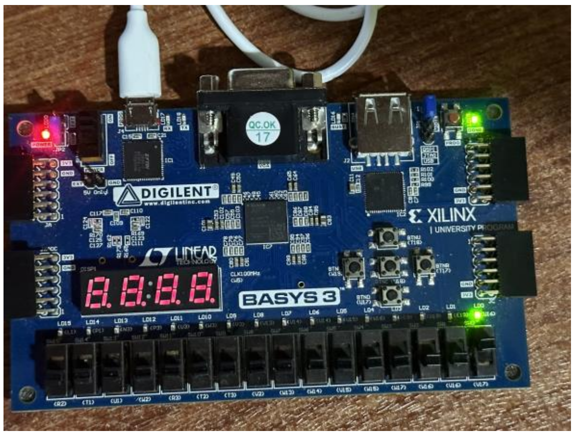

Figure 4.1: Addition implementation 1

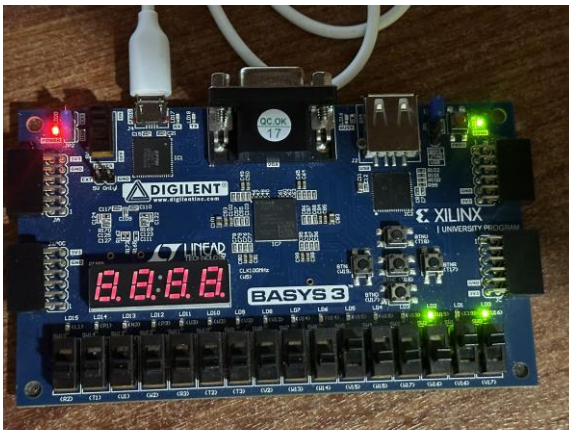

Figure 4.2: Addition implementation 2

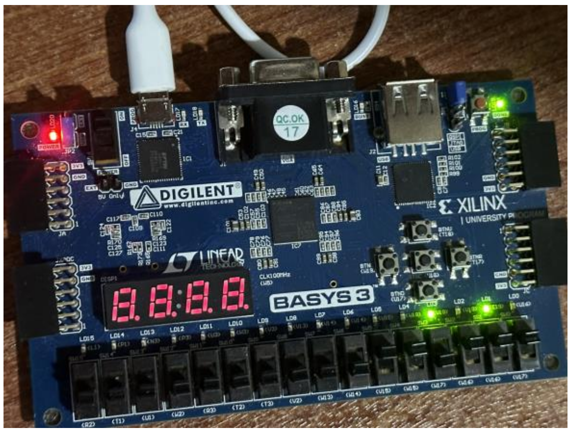

Figure 4.3: Addition implementation 3

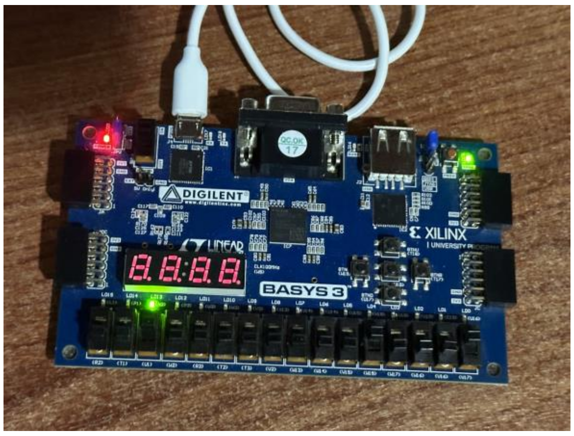

Figure 4.4: Subtraction implementation 1

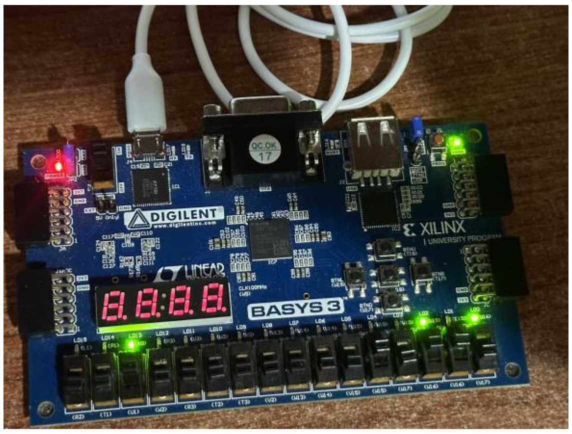

Figure 4.5: Subtraction implementation 2

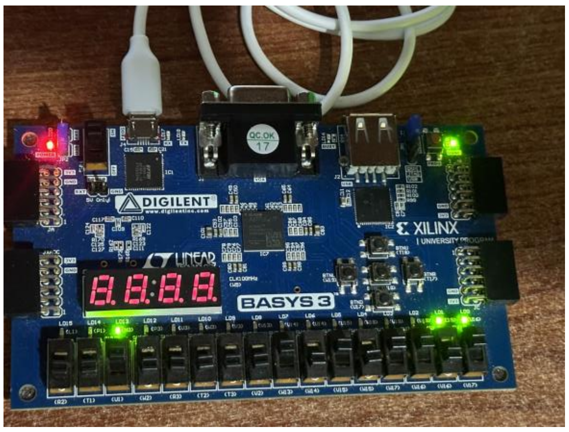

Figure 4.6: Subtraction implementation 3

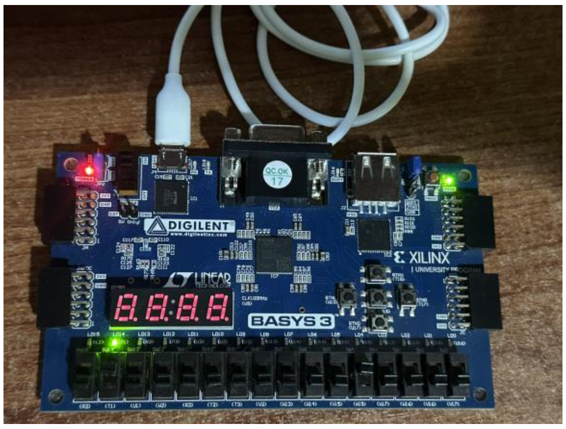

Figure 4.7: Multiplication implementation 1

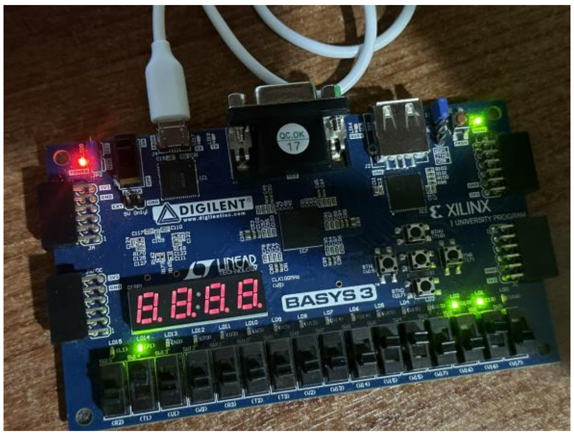

Figure 4.8: Multiplication implementation 2

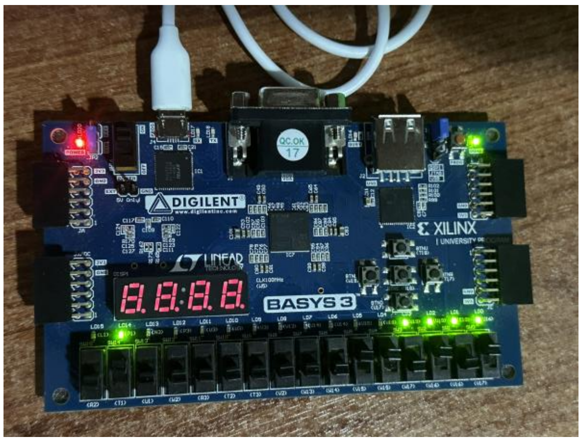
Figure 4.9: Multiplication implementation 3

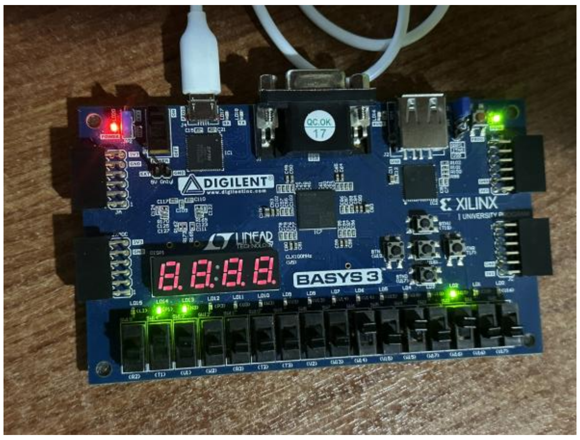
Figure 4.10: Bitwise AND implementation 1

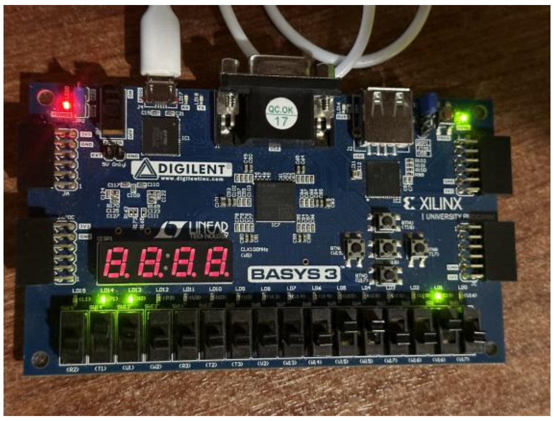
Figure 4.11: Bitwise AND implementation 2

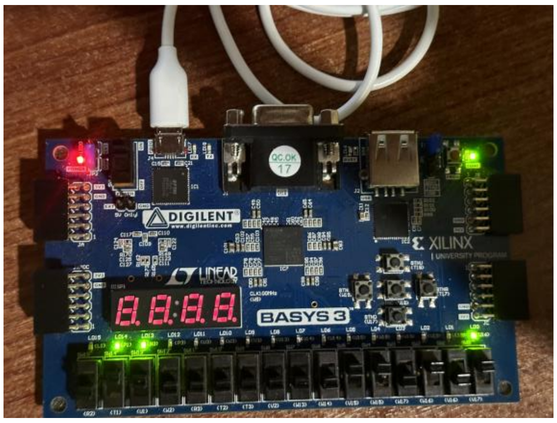
Figure 4.12: Bitwise AND implementation 3

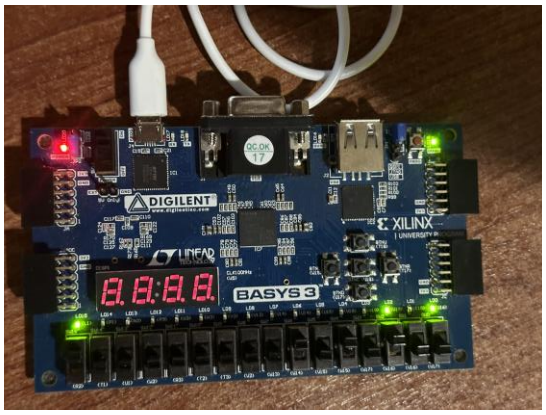
Figure 4.13: Bitwise OR implementation 1

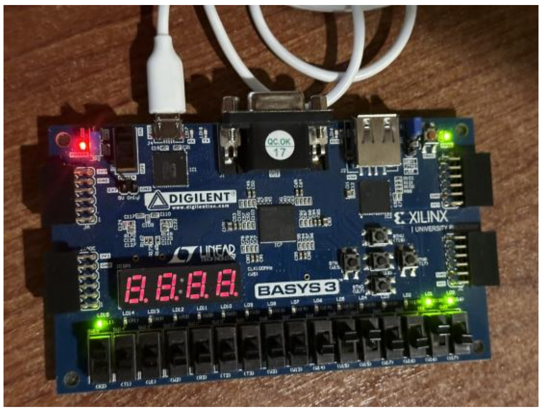
Figure 4.14: Bitwise OR implementation 2

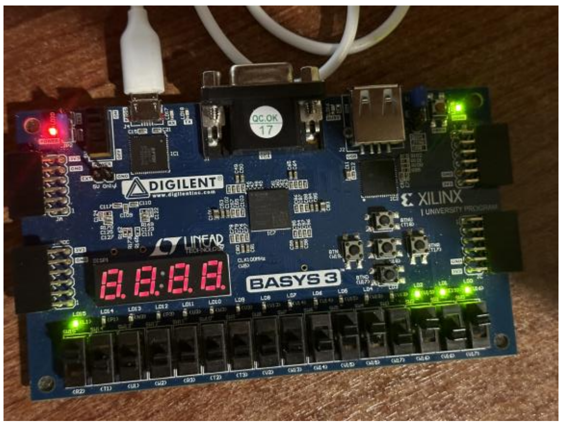
Figure 4.15: Bitwise OR implementation 3

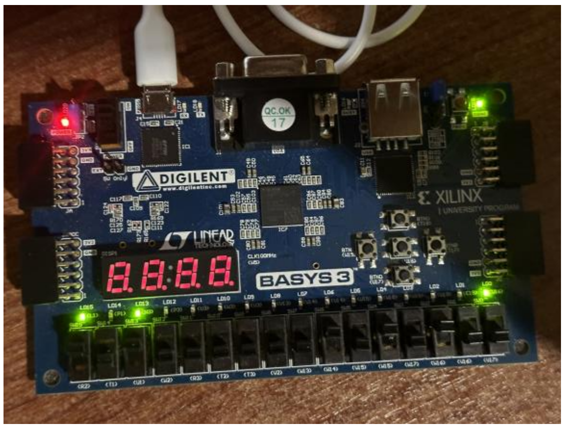
Figure 4.16: Bitwise XOR implementation 1

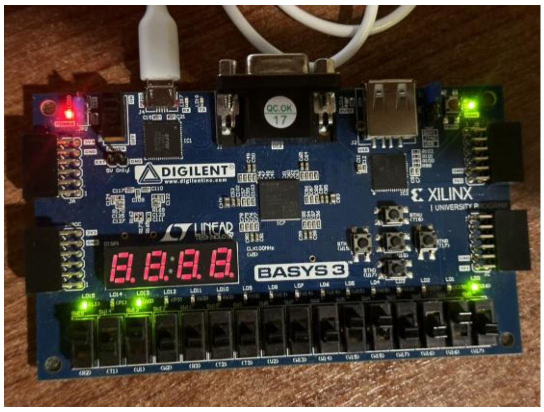
Figure 4.17: Bitwise XOR implementation 2

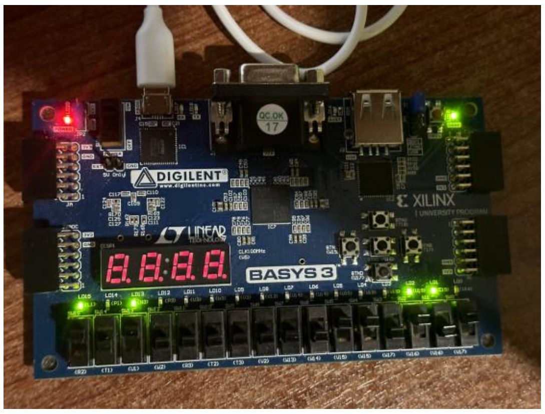
Figure 4.18: Bitwise XOR implementation 3

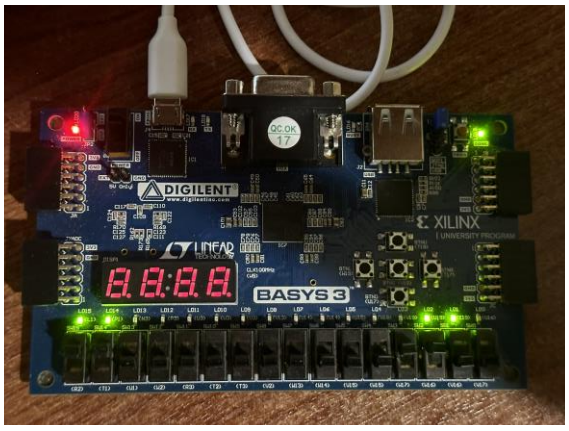
Figure 4.19: Rotate right implementation 1

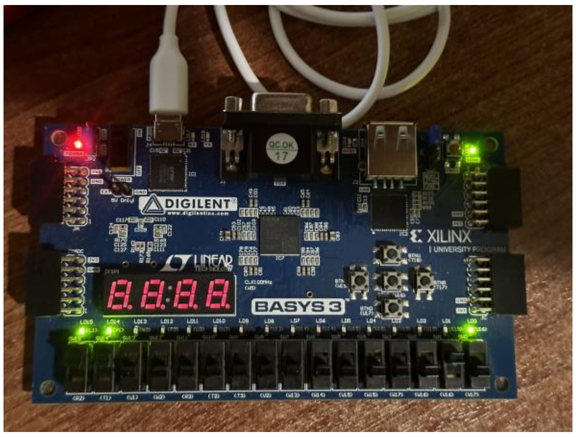
Figure 4.20: Rotate right implementation 2

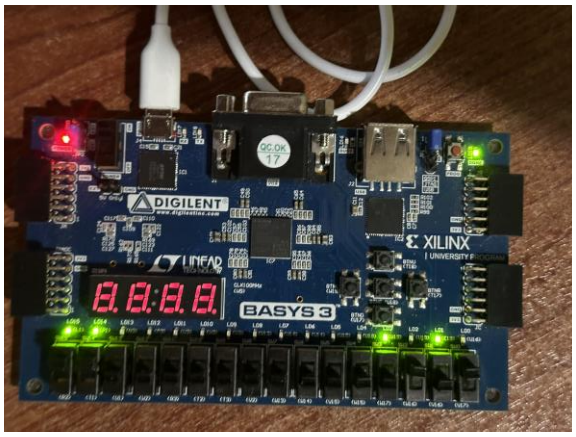
Figure 4.21: Rotate right implementation 3

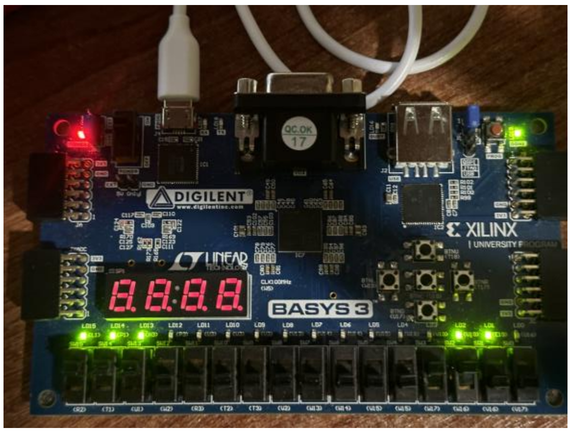
Figure 4.22: Shift logical right implementation 1

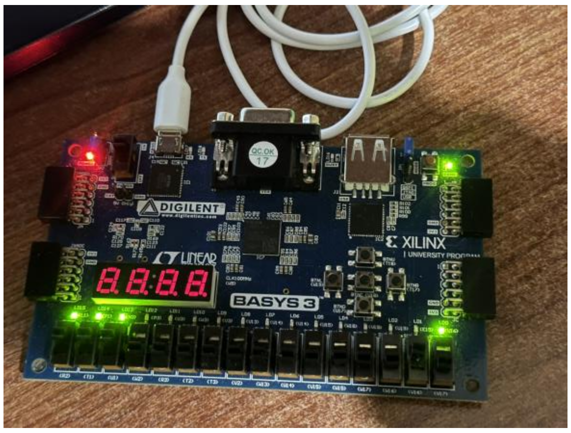
Figure 4.23: Shift logical right implementation 2

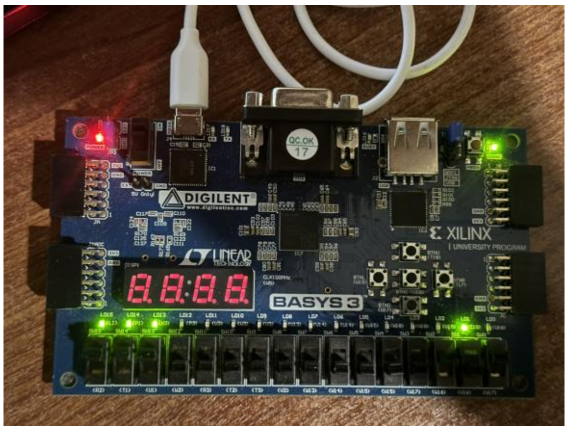
Figure 4.24: Shift logical right implementation 3

VHDL Code:

top_module.vhd

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;



entity top_module is

  Port (i_SW : in std_logic_vector(10 downto 0);

  o_LED : out std_logic_vector(6 downto 0)) ;
```

```
end top_module;

architecture Behavioral of top_module is

  component Adder_4_Bit is
    Port ( i_in1 : in std_logic_vector(3 downto 0);
      i_in2 : in std_logic_vector(3 downto 0);
      o_output : out std_logic_vector(3 downto 0));
    end component;

  component subtraction_4_bit is
    Port (i_valone: in std_logic_vector(3 downto 0);
      i_valtwo: in std_logic_vector(3 downto 0);
      o_out: out std_logic_vector(3 downto 0));
    end component;

  component multiplication_4_bit is
    Port ( i_i1 : in std_logic_vector(3 downto 0);
       i_i2 : in std_logic_vector(3 downto 0);
       o_outp : out std_logic_vector(3 downto 0));
    end component;

  component bitwise_and is
    Port ( i_in1 : in std_logic_vector(3 downto 0);
      i_in2 : in std_logic_vector(3 downto 0);
      o_out1 : out std_logic_vector(3 downto 0));
    end component;
```

```
 component bitwise_or is

   Port ( i_in1 : in std_logic_vector(3 downto 0);

      i_in2 : in std_logic_vector(3 downto 0);

      o_out1 : out std_logic_vector(3 downto 0));

   end component;


 component bitwise_xor is

   Port ( i_in1 : in std_logic_vector(3 downto 0);

      i_in2 : in std_logic_vector(3 downto 0);

      o_out1 : out std_logic_vector(3 downto 0));

   end component;


  component right_rotate is

   Port ( i_in1 : in std_logic_vector(3 downto 0);

      o_out1 : out std_logic_vector(3 downto 0));

   end component;


  component shift_logical_right is

     Port ( i_in1 : in std_logic_vector(3 downto 0);

      o_out1 : out std_logic_vector(3 downto 0));

     end component;



signal s_input1 : std_logic_vector(3 downto 0) := (others => '0');

signal s_input2 : std_logic_vector(3 downto 0):= (others => '0');

signal s_type : std_logic_vector(2 downto 0):= (others => '0');
```

```
   signal s_output0 : std_logic_vector(3 downto 0):= (others => '0');

   signal s_output1 : std_logic_vector(3 downto 0):= (others => '0');

   signal s_output2 : std_logic_vector(3 downto 0):= (others => '0');

   signal s_output3 : std_logic_vector(3 downto 0):= (others => '0');

   signal s_output4 : std_logic_vector(3 downto 0):= (others => '0');

   signal s_output5 : std_logic_vector(3 downto 0):= (others => '0');

   signal s_output6 : std_logic_vector(3 downto 0):= (others => '0');

   signal s_output7 : std_logic_vector(3 downto 0):= (others => '0');



begin

s_input1 <= i_SW(3 downto 0);

s_input2 <= i_SW(7 downto 4);

s_type <= i_SW(10 downto 8);


o_LED(6 downto 4)<=s_type;


Addition: Adder_4_Bit port map (

         i_in1    => s_input1,

         i_in2    => s_input2,

         o_output => s_output0);

Subtraction : subtraction_4_bit port map (

         i_valone    => s_input1,

         i_valtwo    => s_input2,

         o_out => s_output1);
```

```
 Multiplication : multiplication_4_bit port map(
          i_i1    => s_input1,
          i_i2    => s_input2,
          o_ouTP => s_output2);

AND_Bitwise : bitwise_and port map(
       i_in1 => s_input1,
       i_in2 => s_input2,
       o_out1 => s_output3
   );

OR_Bitwise : bitwise_or port map(
       i_in1 => s_input1,
       i_in2 => s_input2,
       o_out1 => s_output4
       );

XOR_Bitwise : bitwise_xor port map(
       i_in1 => s_input1,
       i_in2 => s_input2,
       o_out1 => s_output5
       );

Rotate_R : right_rotate port map(
       i_in1 => s_input1,
       o_out1 => s_output6
       );
```

```
Logical_R : shift_logical_right port map(

        i_in1 => s_input1,

        o_out1 => s_output7

        );


process

begin



if s_type <= "000" then --addition

    o_LED(3 downto 0) <= s_output0;


 elsif s_type <= "001" then --subtraction

    o_LED(3 downto 0) <= s_output1;


 elsif s_type <= "010" then --multiplication

    o_LED(3 downto 0) <= s_output2;


 elsif s_type <= "011" then --bitwise and

    o_LED(3 downto 0) <= s_output3;


 elsif s_type <= "100" then  --bitwise or

    o_LED(3 downto 0) <= s_output4;


 elsif s_type <= "101" then --bitwise xor

    o_LED(3 downto 0) <= s_output5;
 elsif s_type <= "110" then --rotate right

   o_LED(3 downto 0) <= s_output6;


 elsif s_type <= "111" then --shift logical right

   o_LED(3 downto 0) <= s_output7;


 else

    o_LED(3 downto 0) <= (others => '0');


end if;

wait;

end process;


end Behavioral;
```

4_Bit_Adder.vhd

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;


entity Adder_4_Bit is

   Port ( i_in1 : in std_logic_vector(3 downto 0);

          i_in2 : in std_logic_vector(3 downto 0);

          o_output : out std_logic_vector(3 downto 0));

end Adder_4_Bit;
```

```
architecture Behavioral of Adder_4_Bit is

   component Full_Adder_1_bit is

      port (i_a: in std_logic;

      i_b: in std_logic;

      i_car: in std_logic;

      o_out: out std_logic;

      o_carry: out std_logic);

    end component;


   signal s_carryone: std_logic;

   signal s_carrytwo: std_logic;

   signal s_carrythree: std_logic;


begin

   adder1: Full_Adder_1_bit

      port map( i_a => i_in1(0),

      i_b => i_in2(0),

      i_car => '0',

      o_out => o_output(0),

      o_carry => s_carryone);


   adder2: Full_Adder_1_bit

      port map( i_a => i_in1(1),

      i_b => i_in2(1),

      i_car => s_carryone,

      o_out => o_output(1),

      o_carry => s_carrytwo);
    adder3: Full_Adder_1_bit

      port map( i_a => i_in1(2),

      i_b => i_in2(2),

      i_car => s_carrytwo,

      o_out => o_output(2),

      o_carry => s_carrythree);


    adder4: Full_Adder_1_bit

      port map( i_a => i_in1(3),

      i_b => i_in2(3),

      i_car => s_carrythree,

      o_out => o_output(3),

      o_carry => open);


end Behavioral;
```

Full Adder.vhd

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;


entity Full_Adder_1_bit is

Port (i_a: in std_logic;

i_b: in std_logic;

i_car: in std_logic;

o_out: out std_logic;
o_carry: out std_logic);

end Full_Adder_1_bit;


architecture Behavioral of Full_Adder_1_bit is

   signal s_outone : std_logic;

   signal s_carryone : std_logic;

   signal s_carrytwo : std_logic;

begin

   s_outone <= i_a xor i_b;

   s_carryone <= i_a and i_b;


   o_out <= s_outone xor i_car;

   s_carrytwo <= s_outone and i_car;

   o_carry <= s_carryone or s_carrytwo;

end Behavioral;
```

4_bit_subtraction.vhd

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;


entity subtraction_4_bit is

Port (i_valone: in std_logic_vector(3 downto 0);

i_valtwo: in std_logic_vector(3 downto 0);

o_out: out std_logic_vector(3 downto 0));

end subtraction_4_bit;
```

```
architecture Behavioral of subtraction_4_bit is


component Adder_4_Bit is
   Port ( i_in1 : in std_logic_vector(3 downto 0);
          i_in2 : in std_logic_vector(3 downto 0);
          o_output : out std_logic_vector(3 downto 0));
end component;
signal placeholder1 : std_logic_vector (3 downto 0);
signal s_valtwo_complement : std_logic_vector(3 downto 0);
--this module uses two's compolement to perform subtraction
begin
placeholder1 <= not i_valtwo;
twoscomplementer : Adder_4_Bit
   port map( i_in1=>placeholder1 , i_in2=>"0001" , o_output => s_valtwo_complement );


resultfinder : Adder_4_Bit
   port map ( i_in1 => i_valone , i_in2 => s_valtwo_complement , o_output => o_out);


end Behavioral;
```

Multiplication.vhd

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


entity multiplication_4_bit is
   Port ( i_i1 : in std_logic_vector(3 downto 0);
          i_i2 : in std_logic_vector(3 downto 0);

          o_outp : out std_logic_vector(3 downto 0));

end multiplication_4_bit;


architecture Behavioral of multiplication_4_bit is


component Adder_4_Bit is

   Port ( i_in1 : in std_logic_vector(3 downto 0);

          i_in2 : in std_logic_vector(3 downto 0);

          o_output : out std_logic_vector(3 downto 0));

end component;


          signal s_dig0 : std_logic_vector(3 downto 0) := (others => '0');

          signal s_dig1 : std_logic_vector(3 downto 0) := (others => '0');

          signal s_dig2 : std_logic_vector(3 downto 0) := (others => '0');

          signal s_dig3 : std_logic_vector(3 downto 0) := (others => '0');


          signal s_hold1 : std_logic_vector(3 downto 0) := (others => '0');

          signal s_hold2 : std_logic_vector(3 downto 0) := (others => '0');


begin

s_dig0(3 downto 0) <= ((i_i1(3) and i_i2(0)) , (i_i1(2) and i_i2(0)) , (i_i1(1) and i_i2(0)) , (i_i1(0) and i_i2(0)));

s_dig1(3 downto 0) <= ((i_i1(2) and i_i2(1)) , (i_i1(1) and i_i2(1)) , (i_i1(0) and i_i2(1)) , '0');

s_dig2(3 downto 0) <= ((i_i1(1) and i_i2(2)) , (i_i1(0) and i_i2(2)) , '0' , '0' );

s_dig3(3 downto 0) <= ((i_i1(0) and i_i2(3)) , '0' , '0' , '0');
```

```
ad1: Adder_4_Bit
   port map (i_in1 => s_dig0,
         i_in2 => s_dig1,
         o_output => s_hold1);

ad2: Adder_4_Bit
   port map (i_in1 => s_dig2,
         i_in2 => s_dig3,
         o_output => s_hold2);

ad3: Adder_4_Bit
   port map (i_in1 => s_hold1,
         i_in2 => s_hold2,
         o_output => o_outp);

end Behavioral;
```

bitwise_and.vhd

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


entity bitwise_and is
   Port ( i_in1 : in std_logic_vector(3 downto 0);
         i_in2 : in std_logic_vector(3 downto 0);
         o_out1 : out std_logic_vector(3 downto 0));
```

```
end bitwise_and;


architecture Behavioral of bitwise_and is


begin

o_out1 <= i_in1 and i_in2 ;


end Behavioral;
```

bitwise_or.vhd

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;



entity bitwise_or is

   Port ( i_in1 : in std_logic_vector(3 downto 0);

          i_in2 : in std_logic_vector(3 downto 0);

          o_out1 : out std_logic_vector(3 downto 0));

end bitwise_or;


architecture Behavioral of bitwise_or is


begin
```

```
o_out1 <= i_in1 or i_in2 ;


end Behavioral;
```

bitwise_xor.vhd

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



entity bitwise_xor is
   Port ( i_in1 : in std_logic_vector(3 downto 0);
          i_in2 : in std_logic_vector(3 downto 0);
          o_out1 : out std_logic_vector(3 downto 0));
end bitwise_xor;


architecture Behavioral of bitwise_xor is


begin
o_out1 <= i_in1 xor i_in2 ;


end Behavioral;
```

Rotate Right.vhd

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

```
entity right_rotate is
   Port ( i_in1 : in std_logic_vector(3 downto 0);
          o_out1 : out std_logic_vector(3 downto 0));
end right_rotate;

architecture Behavioral of right_rotate is


begin

o_out1(0)<=i_in1(1);
o_out1(1)<=i_in1(2);
o_out1(2)<=i_in1(3);
o_out1(3)<=i_in1(0);

end Behavioral;
```

Shift Logical Right.vhd

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

entity shift_logical_right is
   Port ( i_in1 : in std_logic_vector(3 downto 0);
          o_out1 : out std_logic_vector(3 downto 0));
end shift_logical_right;
```

```
architecture Behavioral of shift_logical_right is

begin

o_out1(0)<=i_in1(1);
o_out1(1)<=i_in1(2);
o_out1(2)<=i_in1(3);
o_out1(3)<=('0' AND i_in1(0));  --to prevent i_in1(0) from staying unused

end Behavioral;
```

TestBench4.vhd

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

entity lab4_testbench is
end lab4_testbench;

architecture Behavioral of lab4_testbench is

   component top_module is
     Port (
       i_SW : in std_logic_vector(10 downto 0);
```

```
        o_LED : out std_logic_vector(6 downto 0)

      );

    end component;


  signal s_SW : std_logic_vector(10 downto 0) := (others => '0');

  signal s_LED : std_logic_vector(6 downto 0) := (others => '0');

  signal s_input1 : std_logic_vector(3 downto 0) := (others => '0');

  signal s_input2 : std_logic_vector(3 downto 0) := (others => '0');

  signal s_output : std_logic_vector(3 downto 0) := (others => '0');

  signal s_type : std_logic_vector(2 downto 0) := (others => '0');


begin


s_SW(3 downto 0) <= s_input1;

s_SW(7 downto 4) <= s_input2;

s_output <= s_LED(3 downto 0);

s_SW(10 downto 8) <= s_type;


  UUT: top_module Port Map (

    i_SW => s_SW,

    o_LED => s_LED

  );


  stim_proc: process

  begin

    wait for 10ns;

  --type 000: Addition
```

```
        s_type <= "000";

        s_input1 <= "0000";

        s_input2 <= "0001";


      wait for 10ns;

        s_type <= "000";

        s_input1 <= "0011";

        s_input2 <= "0010";


       wait for 10ns;

        s_type <= "000";

        s_input1 <= "0111";

        s_input2 <= "0011";


--type 001: Subtraction

      wait for 10ns;

        s_type <= "001";

        s_input1 <= "0001";

        s_input2 <= "0001";


      wait for 10ns;

        s_type <= "001";

        s_input1 <= "0110";

        s_input2 <= "0001";


       wait for 10ns;
```

```
        s_type <= "001";

        s_input1 <= "1101";

        s_input2 <= "1010";


--type 010: Multiplication

    wait for 10ns;

        s_type <= "010";

        s_input1 <= "0000";

        s_input2 <= "0001";


    wait for 10ns;

        s_type <= "010";

        s_input1 <= "0010";

        s_input2 <= "0011";


     wait for 10ns;

        s_type <= "010";

        s_input1 <= "0101";

        s_input2 <= "0011";


--type 011: Bitwise AND

    wait for 10ns;

        s_type <= "011";

        s_input1 <= "0100";

        s_input2 <= "0101";


    wait for 10ns;
```

```
        s_type <= "011";

        s_input1 <= "0010";

        s_input2 <= "0011";


      wait for 10ns;

        s_type <= "011";

        s_input1 <= "0101";

        s_input2 <= "0011";


--type 100: Bitwise OR

     wait for 10ns;

        s_type <= "100";

        s_input1 <= "0100";

        s_input2 <= "0101";


     wait for 10ns;

        s_type <= "100";

        s_input1 <= "0010";

        s_input2 <= "0011";


      wait for 10ns;

        s_type <= "100";

        s_input1 <= "0101";

        s_input2 <= "0011";


--type 101: Bitwise XOR

     wait for 10ns;
```

```
    s_type <= "101";

    s_input1 <= "0100";

    s_input2 <= "0101";


  wait for 10ns;

    s_type <= "101";

    s_input1 <= "0010";

    s_input2 <= "0011";


   wait for 10ns;

    s_type <= "101";

    s_input1 <= "0101";

    s_input2 <= "0011";


--type 110: Rotate Right

  wait for 10ns;

    s_type <= "110";

    s_input1 <= "1100";

    s_input2 <= "0000";


  wait for 10ns;

    s_type <= "110";

    s_input1 <= "0010";

    s_input2 <= "0000";


   wait for 10ns;

    s_type <= "110";
```

```
          s_input1 <= "0101";

          s_input2 <= "0000";


  --type 111: Shift Logical Right

       wait for 10ns;

          s_type <= "111";

          s_input1 <= "1100";

          s_input2 <= "0000";


       wait for 10ns;

          s_type <= "111";

          s_input1 <= "0010";

          s_input2 <= "0000";


        wait for 10ns;

          s_type <= "111";

          s_input1 <= "0101";

          s_input2 <= "0000";


        wait for 10ns;

  end process stim_proc;


end Behavioral;
```

Sources

1) Single bit left/right rotating, logical shift and arithmetic shift... (n.d.). https://www.researchgate.net/figure/Single-bit-left-right-rotating-logical-shift-and-arithmetic-shift-operations-on-8-bit_fig6_282306909