Experiment 3 in Electrical engineering, ET061G

by Khaled Hamza

Mittuniversitetet

MID SWEDEN UNIVERSITY

Khaled Hamza ET061G
2024-02-19

Purpose

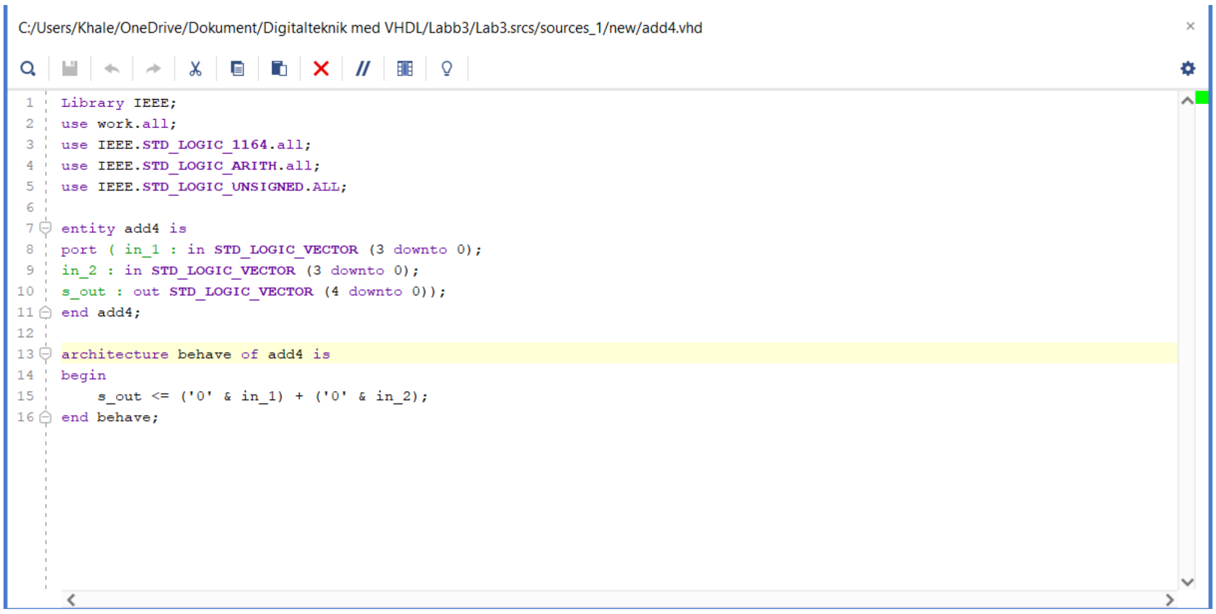
The aim of this experiment is to familiarize oneself with integrating various components in the Vivado software and to create a testbench for validating the functionality of these components by simulating each one within the testbench.

Theory and Implementation:

Section 1: 4-bit adder

This component is a simpler implementation of the 8-bit adder used in the previous experiment.

Code:


C:/Users/Khale/OneDrive/Dokument/Digitalteknik med VHDL/Labb3/Lab3.srcs/sources_1/new/add4.vhd

```
Library IEEE;
use work.all;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity add4 is
port ( in_1 : in STD_LOGIC_VECTOR (3 downto 0);
in_2 : in STD_LOGIC_VECTOR (3 downto 0);
s_out : out STD_LOGIC_VECTOR (4 downto 0));
end add4;

architecture behave of add4 is
begin
    s_out <= ('0' & in_1) + ('0' & in_2);
end behave;
```

Result:

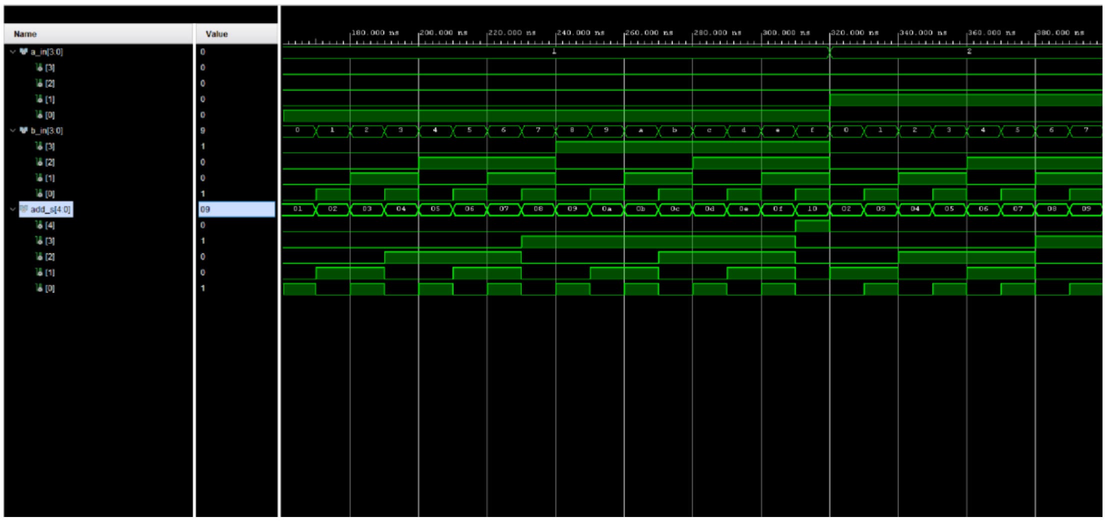
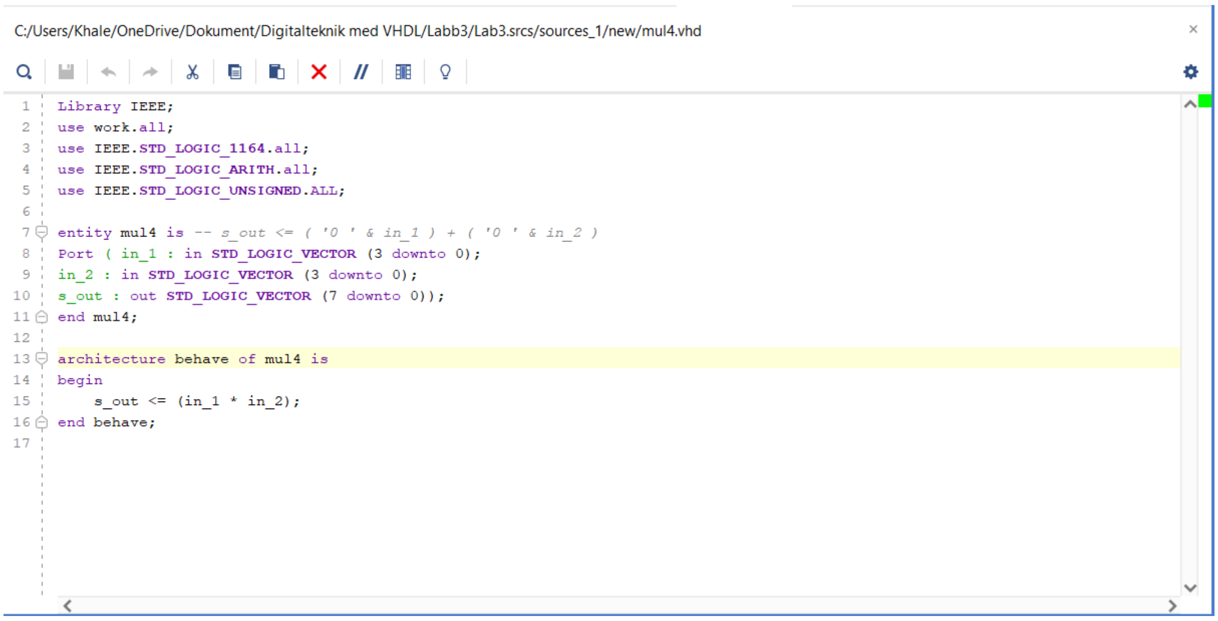


Figure 1: Simulation result of the 4-bit adder

Section 2: 4- bit multiplier

The 4 bit multiplier is used just by multiplying the inputs and storing the product in a vector that has the size of the two inputs size combined.

Code:



C:/Users/Khale/OneDrive/Dokument/Digitalteknik med VHDL/Labb3/Lab3.srcs/sources_1/new/mul4.vhd

```vhdl
Library IEEE;
use work.all;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity mul4 is -- s_out <= ( '0 ' & in_1 ) + ( '0 ' & in_2 )
Port ( in_1 : in STD_LOGIC_VECTOR (3 downto 0);
in_2 : in STD_LOGIC_VECTOR (3 downto 0);
s_out : out STD_LOGIC_VECTOR (7 downto 0));
end mul4;

architecture behave of mul4 is
begin
    s_out <= (in_1 * in_2);
end behave;

```

Result:

Khaled Hamza
2024-02-19

ET061G

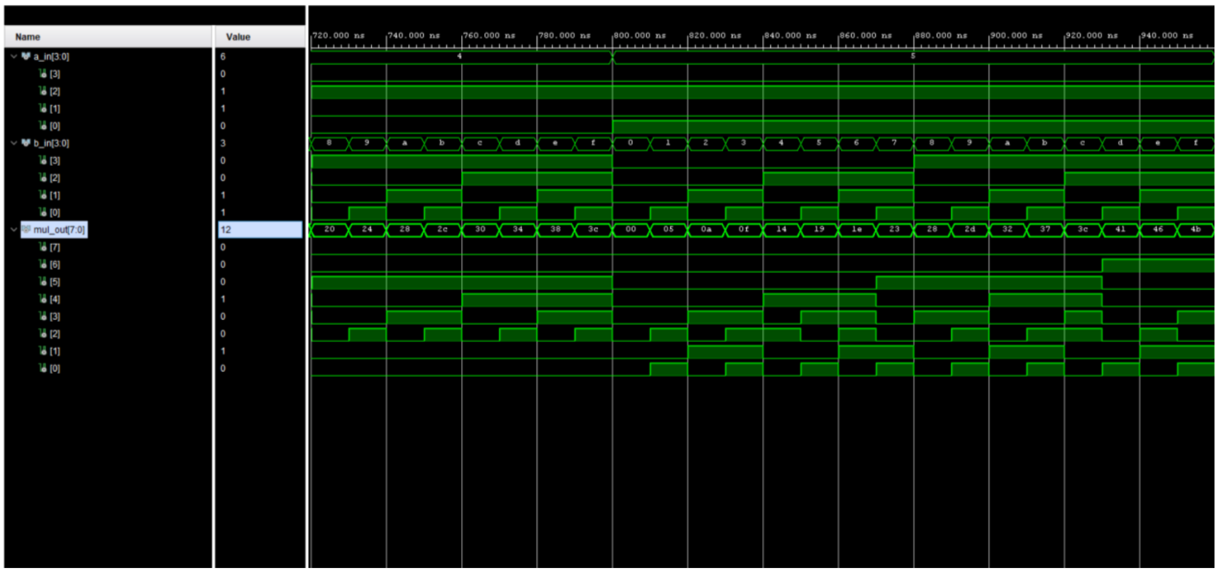
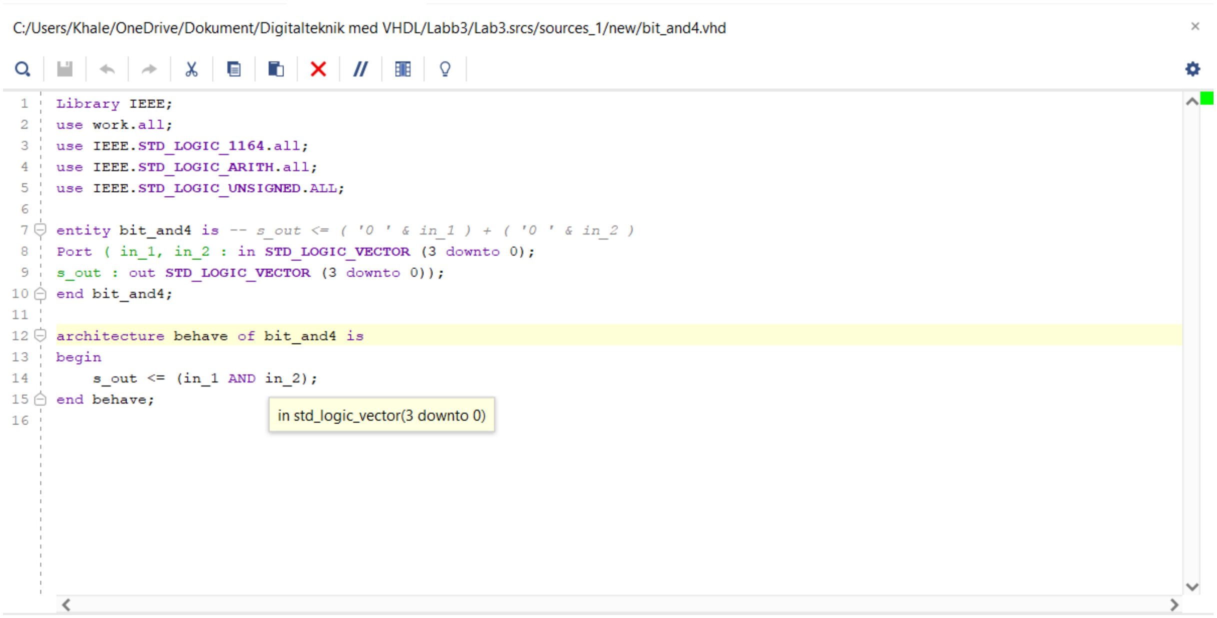


Figure 2: Simulation result of the 4-bit multiplier.

Section 3: 4- bit AND operator

The 4 bit AND operator is simulated using the built in AND operator. The operator multiplies the inputs together bit by bit so the output of each pair of bits would be either one or zero.

Code:


C:/Users/Khale/OneDrive/Dokument/Digitalteknik med VHDL/Labb3/Lab3.srcs/sources_1/new/bit_and4.vhd

```
Library IEEE;
use work.all;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity bit_and4 is -- s_out <= ( '0 ' & in_1 ) + ( '0 ' & in_2 )
Port ( in_1, in_2 : in STD_LOGIC_VECTOR (3 downto 0);
s_out : out STD_LOGIC_VECTOR (3 downto 0));
end bit_and4;

architecture behave of bit_and4 is
begin
    s_out <= (in_1 AND in_2);
end behave;
```

Result:

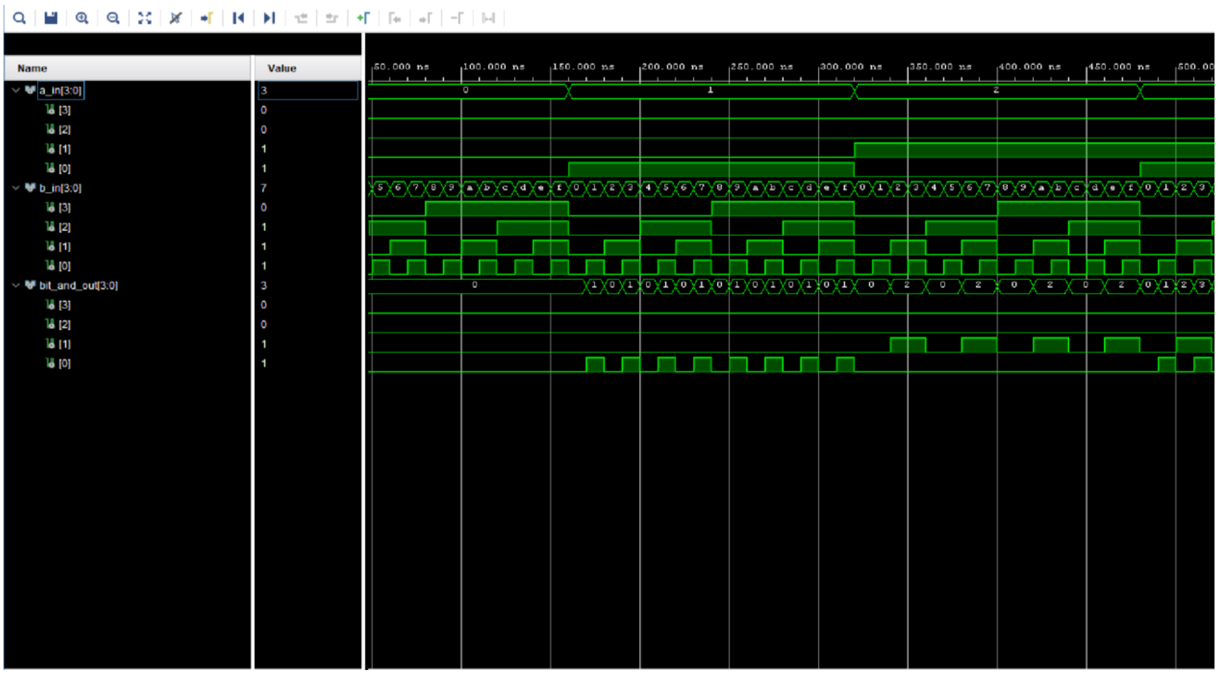
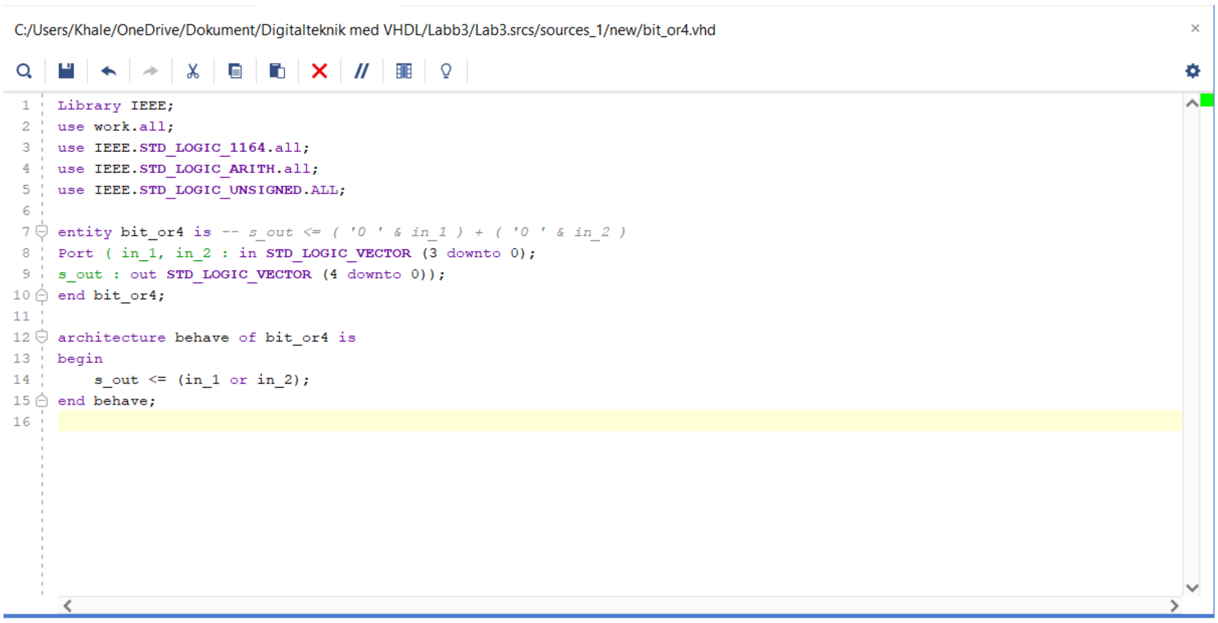


Figure 3: Simulation result of the 4-bit AND operator.

Section 4: 4-bit OR operator

The 4 bit OR operator takes in two 4-bit inputs, adds each bit of the first input to the other input and returns 1 if the for each bit if the sum is more or equal to 1.

Code:



C:/Users/Khale/OneDrive/Dokument/Digitalteknik med VHDL/Labb3/Lab3.srcs/sources_1/new/bit_or4.vhd

```vhdl
Library IEEE;
use work.all;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity bit_or4 is -- s_out <= ( '0 ' & in_1 ) + ( '0 ' & in_2 )
Port ( in_1, in_2 : in STD_LOGIC_VECTOR (3 downto 0);
s_out : out STD_LOGIC_VECTOR (4 downto 0));
end bit_or4;

architecture behave of bit_or4 is
begin
    s_out <= (in_1 or in_2);
end behave;
```

Result:

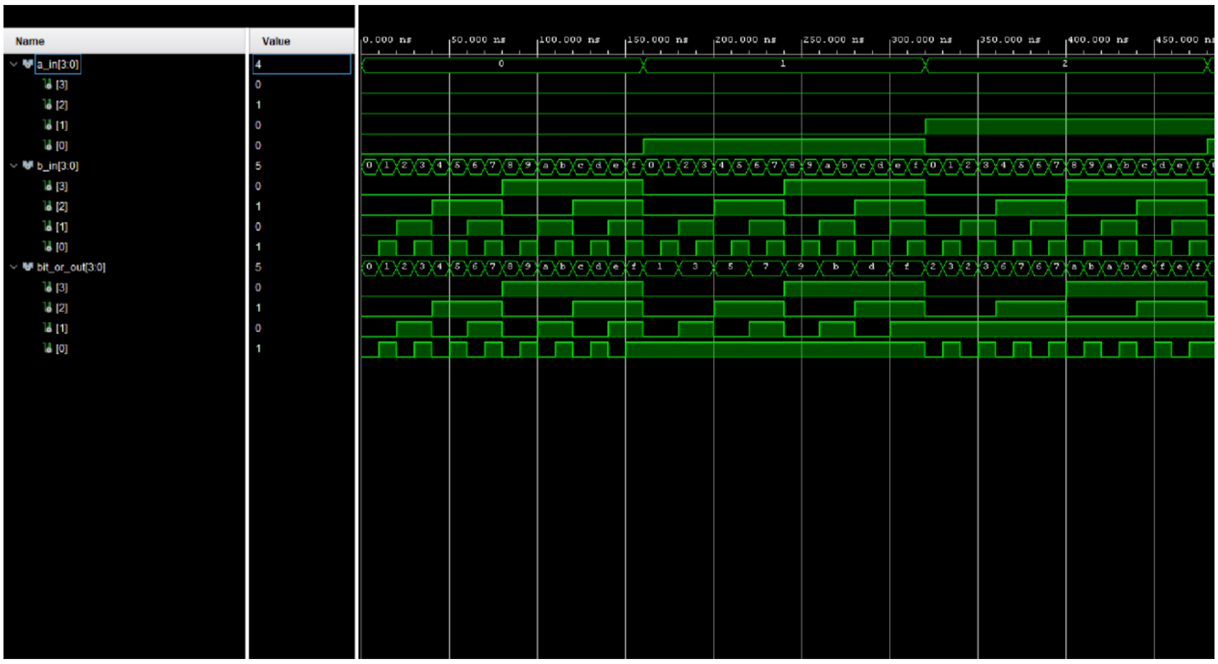
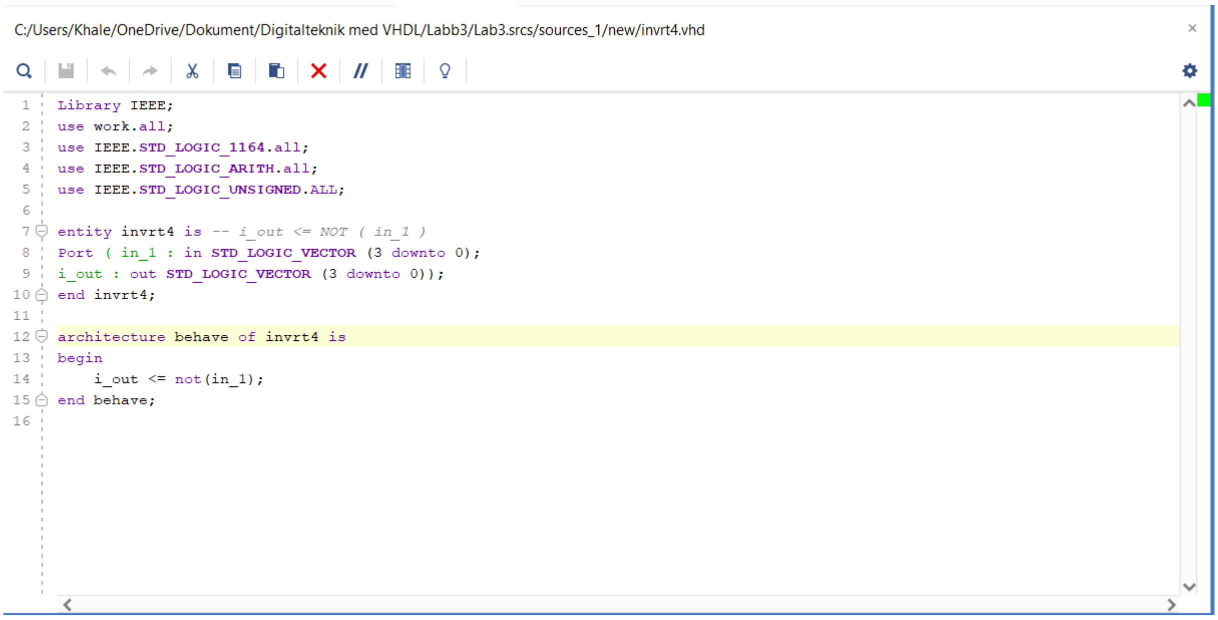


Figure 4: Simulation result of the 4-bit OR operator.

Section 5: 4-bit NOT operator

The 4 bit inverter uses the NOT operator to invert each bit of a 4-bit input. For example if the input is 5 it returns 10 because $5_{10} = 0101_2$, after performing the NOT operator it becomes 1010_2 which is 10_{10}.

Code:



C:/Users/Khale/OneDrive/Dokument/Digitalteknik med VHDL/Labb3/Lab3.srcs/sources_1/new/invrt4.vhd

```
Library IEEE;
use work.all;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity invrt4 is -- i_out <= NOT ( in_1 )
Port ( in_1 : in STD_LOGIC_VECTOR (3 downto 0);
i_out : out STD_LOGIC_VECTOR (3 downto 0));
end invrt4;

architecture behave of invrt4 is
begin
    i_out <= not(in_1);
end behave;
```

Result:

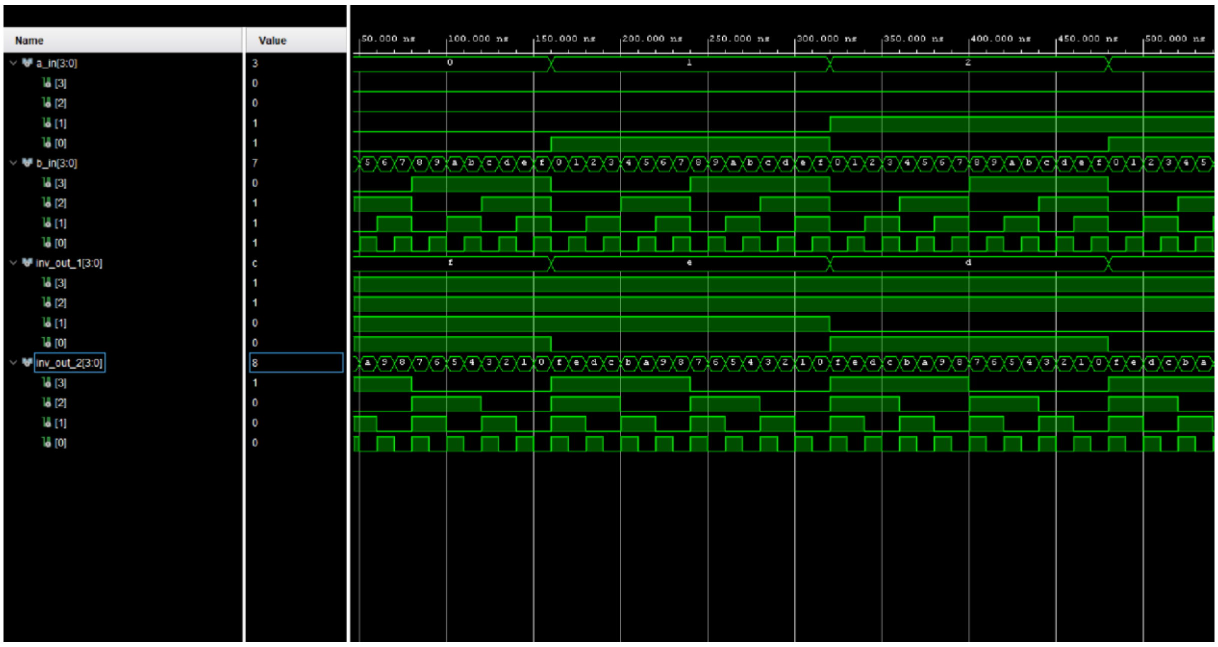


Figure 5: Simulation result of the 4-bit NOT operator on two different inputs.

Results and Discussions:

Section 6: Test bench

The testbench is an expanded version of the one utilized in the second experiment. It includes five components (representing the operations to be performed), two input signals, and six output signals. Each operation has one output signal, except for the inverter, which has two outputs to accommodate its functionality on both input signals, as it only processes one input at a time.

Code:

```
Library IEEE;
use work.all;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_ARITH.all;
entity sim is
end sim;
architecture behave of sim is
component add4 port (in_1, in_2 : in std_logic_vector(3 downto 0);
s_out : out std_logic_vector(4 downto 0));
end component;
component bit_and4 port (in_1, in_2 : in std_logic_vector(3 downto 0);
s_out : out std_logic_vector(3 downto 0));
end component;
component bit_or4 port (in_1, in_2 : in std_logic_vector(3 downto 0);
s_out : out std_logic_vector(3 downto 0));
end component;
component mul4 port (in_1, in_2 : in std_logic_vector(3 downto 0);
s_out : out std_logic_vector(7 downto 0));
end component;
component invrt4 port(in_1 : in std_logic_vector(3 downto 0);
i_out: out std_logic_vector(3 downto 0));
end component;
signal a_in, b_in: std_logic_vector(3 downto 0);
signal mul_out: std_logic_vector(7 downto 0);
signal add_s: std_logic_vector(4 downto 0);
signal bit_and_out: std_logic_vector(3 downto 0);

signal bit_or_out: std_logic_vector(3 downto 0);

signal inv_out_1: std_logic_vector(3 downto 0);

signal inv_out_2: std_logic_vector(3 downto 0);

begin

add4_0 : add4 port map (in_1=> a_in,in_2=> b_in,s_out=> add_s);

bit_and4_0 : bit_and4 port map (in_1=> a_in,in_2=> b_in,s_out=> bit_and_out);

bit_or4_0 : bit_or4 port map (in_1=> a_in,in_2=> b_in,s_out=> bit_or_out);

mul4_0 : mul4 port map (in_1=> a_in,in_2=> b_in,s_out=> mul_out);

invrt4_0 : invrt4 port map (in_1=> a_in,i_out=> inv_out_1);

invrt4_1 : invrt4 port map (in_1=> b_in,i_out=> inv_out_2);

process

variable a_vector, b_vector : std_logic_vector(3 downto 0):=(others=>'0');

begin

for a_vector in 0 to 15 loop

for b_vector in 0 to 15 loop

a_in <= conv_std_logic_vector(a_vector,4);

b_in <= conv_std_logic_vector(b_vector,4);

wait for 10 ns;

end loop;

end loop;

end process;

end behave;
```

Result:

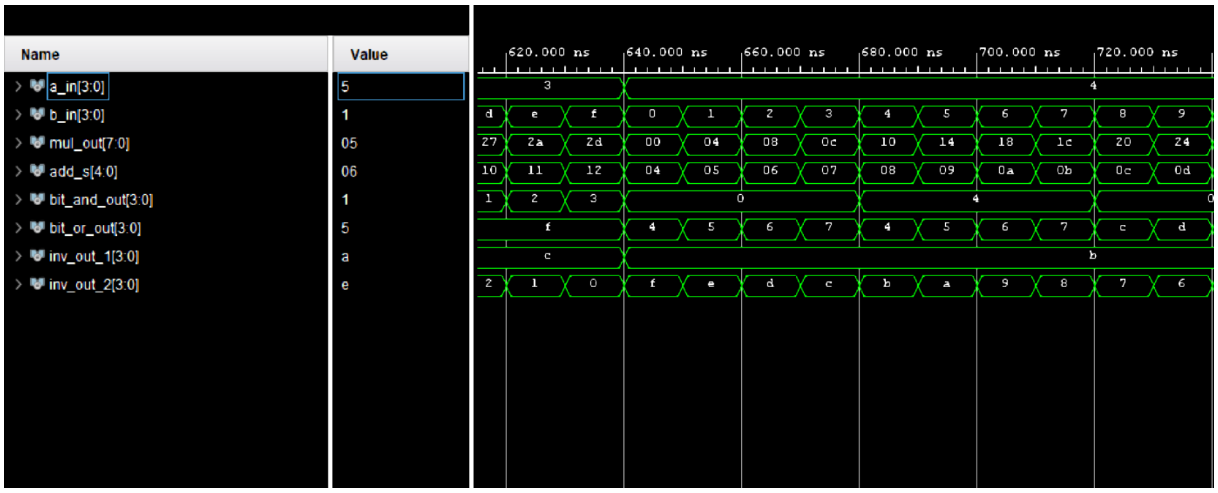


Figure 6: Simulation result of the test bench showing results of all operations.

As mentioned earlier, some of the output signals have a larger bit width compared to the inputs. For instance, the mul_out vector is 8 bits wide, even though each input is only 4 bits. This is because the maximum possible product of two 4-bit inputs requires 8 bits to be represented accurately. In such a scenario, if both inputs are at their maximum 4-bit values, the product will still fit correctly within the 8-bit output without any loss of information.

$15_{10} = 1111_2$ then the output would be $1110\ 0001_2 = 225_{10}$.

References:

https://www.xilinx.com/support/download.html

https://docs.xilinx.com/v/u/2019.2-English/ug901-vivado-synthesis