Selected Topics in Electronics

Team Project

8bits Sequential Multiplier

Done by: Shady Medhat Salah **2017/13034**

Ahmed Hisham **2017/01234**

Ahmed Mohamed Abdel Hamid **2017/09051**

supervised by: Dr. Ashraf Abdel Haq

Eng.Mariam Elhussein Ibrahim Mahmoud

Table of content

Table of Figures

1. Introduction.

The digital circuits are one of the most essential parts in our everyday life and affect every aspect of it. VHDL, also known as Very High-Speed Integrated Circuit (VHSIC) Hardware Description Language, is a language used in order to ease the designing process of the digital circuit as it deals with the digital circuit as if it is a programming language like c/c++ or python.

Adder and Multipliers are one of the essential parts in any ALU circuit in any device as any instruction being carried out in any system is consists of simple math instructions of basic arithmetic operations.

The objective of this is to implement an 8 by 8 sequential multiplier to get an output of 16 bit using a simple shift register circuits and Full Adder circuit.

2. Design procedure for your project.

The design for this project is based on the, firstly, behavioral structure as first this project starts by building the basic blocks of the circuit of the sequential multiplier.

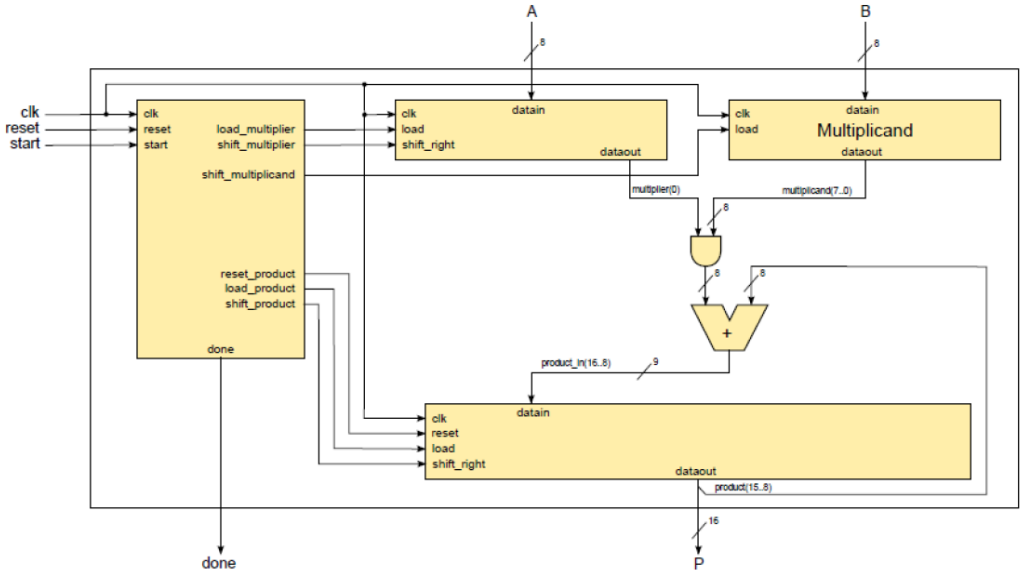


Figure 1 8-bits sequential multiplier

First the project starts with implementing each block as shown in the previous figure. Then determining the signals that is going to connect the blocks with each other's.

lastly, map the connections and implement external processes like the “ANDing” process before the Full Adder block.

The last part of the design is the system controller. The system controller is designed using Finite State Machine which is given as shown in the next figure

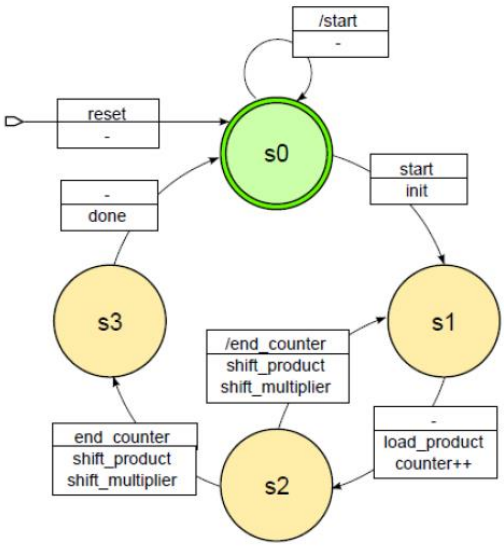


Figure 2 system controller FSM

3. Design code and simulation code for each block.

1- Full Adder

a. VHDL code

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity Full_Add_8bits is
generic (size : integer := 8 );
port(
x,y : in std_logic_vector ((size-1) downto 0);
s       : out std_logic_vector (size downto 0)
);
end entity;
architecture behave of Full_Add_8bits is
begin
process(x,y)
variable c : std_logic_vector (size downto 0);

begin
        c(0):= '0';
        for i in 0 to (size-1) loop
                s(i)<= x(i) xor y(i) xor c(i);
                c(i+1):= (x(i)and y(i))or(y(i)and c(i))or(x(i) and c(i));
        end loop;
        s(size) <= c(size);
        end process;
end behave;
```

b. RTL viewer

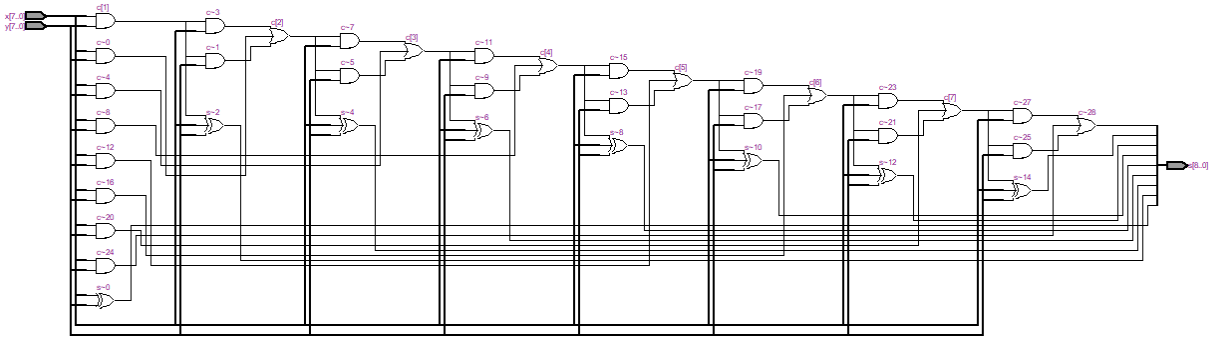


Figure 3 Full Adder RTL circuit viewer

c. Block simulation

The Full Adder is responsible of the summation part in the sequential multiplier circuit.

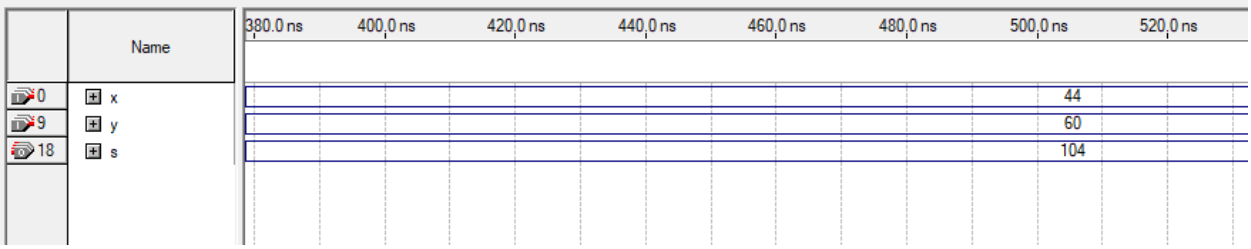


Figure 4 Full Adder simulation test

2- 8-bit Register

a. VHDL code

```
library IEEE;
use IEEE.std_logic_1164.all;
entity REG_8bits is
generic (size : integer := 8);
port(
data_in : in std_logic_vector (size-1 downto 0);
in_load , in_clk : in std_logic;
data_out        : out std_logic_vector (size-1 downto 0)
);
end REG_8bits;
architecture behave of REG_8bits is
signal reg_signal: std_logic_vector (size-1 downto 0);
begin
process (in_clk , in_load)
begin
        if rising_edge(in_clk) then
                if in_load='1' then
                reg_signal<= data_in;
                end if;
        end if;
end process;
data_out <= reg_signal;
end architecture;
```

b. RTL viewer

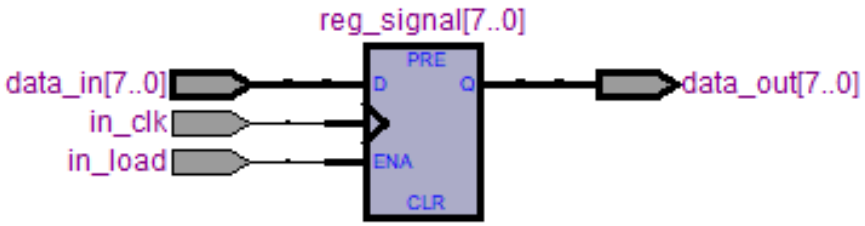


Figure 5 Register RTL circuit viewer

c. Block simulation

The function of the register is to load and store the data in it if the in_load is high (in_load = '1').

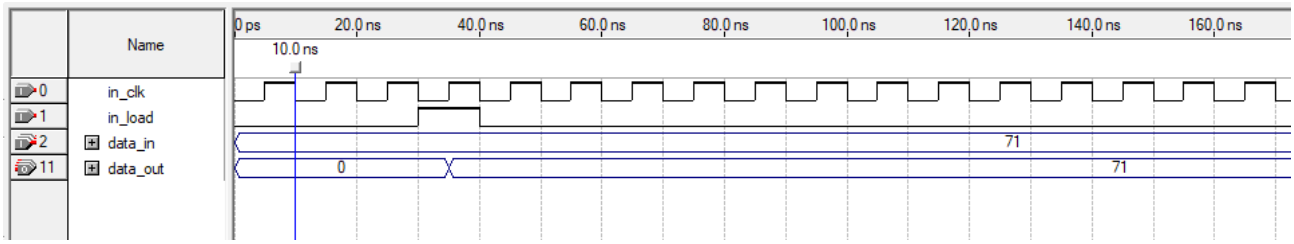


Figure 6 Register simulation test

3- 8-bit Shift Right Register

a. VHDL code

```
library IEEE;
use IEEE.std_logic_1164.all;
entity Shift_REG_R_8bits is
generic (size : integer := 8);
port(
data_in : in std_logic_vector (size-1 downto 0);
in_load , in_clk , in_shift : in std_logic;
data_out        : out std_logic_vector (size-1 downto 0)
);
end Shift_REG_R_8bits;
architecture behave of Shift_REG_R_8bits is
signal reg_signal: std_logic_vector (size-1 downto 0);
begin
process (in_clk , in_load , in_shift)
        begin
        if rising_edge(in_clk) then
                if in_load='1' then
                reg_signal<= data_in;
                elsif in_shift = '1' then
                reg_signal <= '0'& reg_signal(size-1 downto 1);
                end if;
        end if;
end process;

data_out <= reg_signal;
end architecture;
```

b. RTL viewer

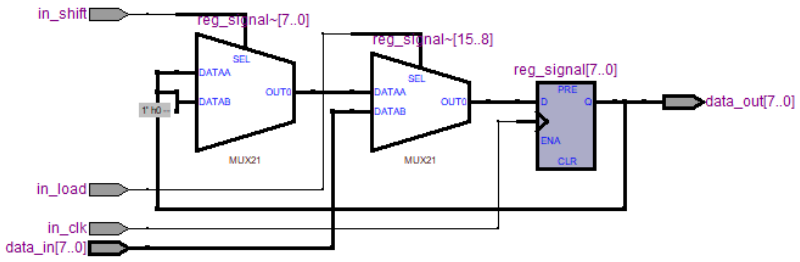


Figure 7 8-bits Shift Register RTL circuit viewer

c. Block simulation

The 8-bit Shift register has 2 main functions. First is to load the input data. secondly is to shift right ,in other words divide by 2, to the stored data.

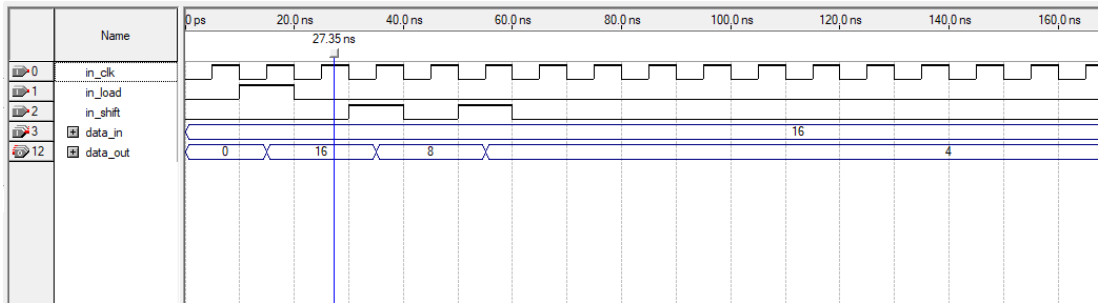


Figure 8 8-bits Shift Register simulation test

4- 16-bit Shift Right Register

a. VHDL code

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Shift_REG_R_16bits is
generic (size : integer := 8);
port(
data_in : in std_logic_vector (size downto 0);
in_load , in_clk , in_shift , in_rst : in std_logic;
data_out        : out std_logic_vector ((2*size-1) downto 0)
);
end Shift_REG_R_16bits;
architecture behave of Shift_REG_R_16bits is
signal reg_signal: std_logic_vector ((2*size) downto 0);
begin
process (in_clk , in_load , in_shift , in_rst)
        begin
        if in_rst = '1' then
                reg_signal <= (others => '0');
        elsif rising_edge(in_clk) then
                if in_load='1' then
                reg_signal((2*size) downto size)<= data_in;
                elsif in_shift = '1' then
                reg_signal <= '0'& reg_signal(2*size downto 1);
                end if;
        end if;
end process;

data_out <= reg_signal((2*size-1) downto 0);

end architecture;
```

b. RTL viewer

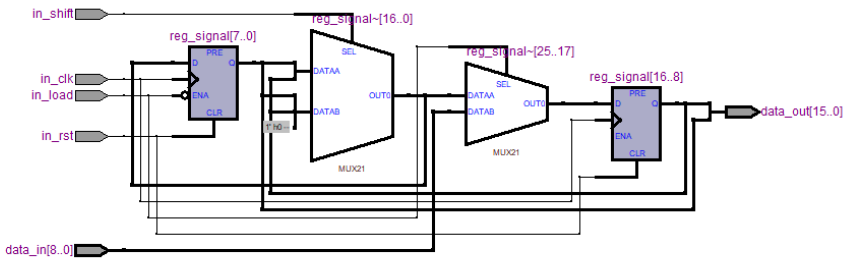


Figure 9 16-bits Shift Register RTL circuit viewer

c. Block simulation

The 16 bit shift register is designed insert the input 9 bits in to the most significant bits of the register , this shift register is a Right shift register.

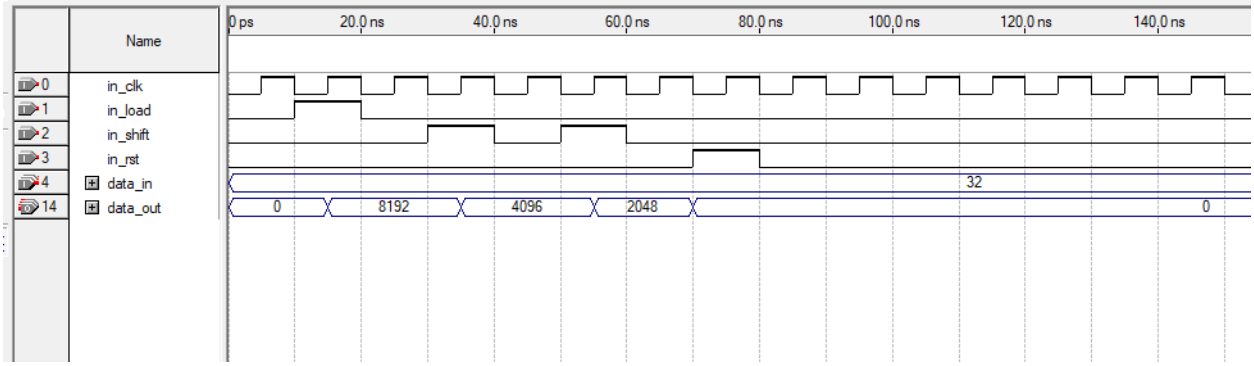


Figure 10 16-bits Shift Register simulation test

5- System Controller

a. VHDL code

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

entity system_controller is
port (

in_clk : in std_logic ;
in_rst, in_start : in std_logic := '0' ;
out_load_multiplier, out_shift_multiplier : out std_logic;
out_shift_multiplicand : out std_logic;
out_reset_product ,out_load_product, out_shift_product : out std_logic;
out_done : out std_logic := '0'
);
end entity;

architecture behave of system_controller is
-- initializing the states
TYPE type_fstate IS (S0,S1,S2,S3);
SIGNAL reg_fstate : type_fstate := S0;
begin
-- process to load the state
 PROCESS (in_clk,reg_fstate , in_rst , in_start)
   variable counter : std_logic_vector(2 downto 0) := (others => '0');
   BEGIN
   if in_rst = '1' then
         out_reset_product <= '1';
         counter := (others => '0');
         reg_fstate <= S0 ;
         elsIF rising_edge(in_clk) THEN
         ------------------------------------------------------------
               case reg_fstate is
               when S0 =>
                              out_done <= '0';
                              if in_start = '1' then
                              reg_fstate <= S1;
                        ------------------------------------------
                              out_load_product <= '0';
                              out_shift_product <='0';
                              out_shift_multiplier <= '0';
```

```
        ------------------------------------------------
                out_load_multiplier <= '1';
                out_shift_multiplicand <= '1';
                out_reset_product <= '1';
                counter := (others => '0');
        -- Inserting 'else' block to prevent latch inference
         else
        reg_fstate <= S0;
                end if;
-------------------------------------------------
when S1 =>
        reg_fstate <= S2 ;

        out_load_multiplier <= '0';
        out_shift_multiplicand <= '0';
        out_done <= '0';
-----------------------------------------------------------
        out_shift_product <='0';
        out_shift_multiplier <= '0';
        out_reset_product <= '0';
        out_load_product <= '1';
        -- first time for S2 , counter == 001 , first summtion
        counter := unsigned(counter)+1;
-----------------------------------------------------------
when S2 =>

        out_load_multiplier <= '0';
        out_shift_multiplicand <= '0';
        out_reset_product <= '0';
        out_done <= '0';
-----------------------------------------------------------
        out_load_product <= '0';
        out_shift_product <='1';
        out_shift_multiplier <= '1';
        if counter = "000" then
        reg_fstate <= S3;
        else
        reg_fstate <= S1;
        end if;
-------------------------------------------------
```

```
            When S3 =>
                out_load_multiplier <= '0';
                out_shift_multiplicand <= '0';
                out_reset_product <= '0';
                out_load_product <= '0';
                out_shift_product <='0';
                out_shift_multiplier <= '0';
                ---------------------------------------------
                out_done <= '1';
                reg_fstate <= S0;
            ------------------------------------------------
            end case;
        end if;
        end process;
end behave;
```

b. RTL viewer

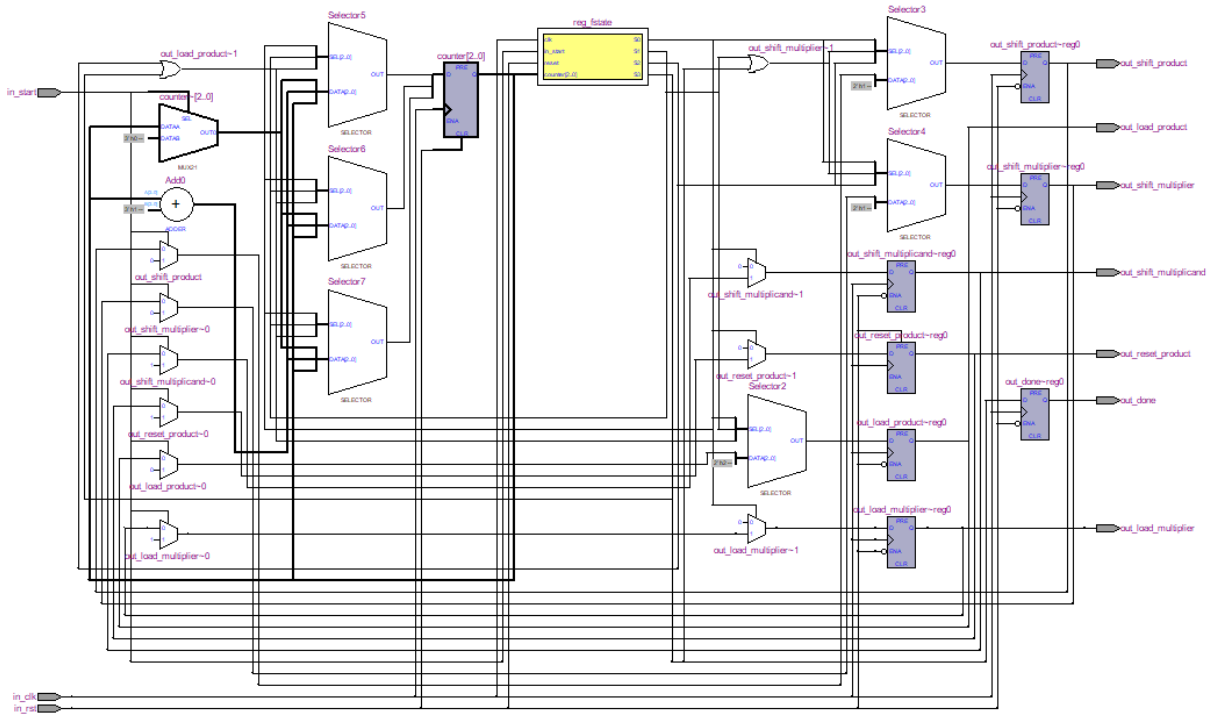


Figure 11 system controller RTL circuit viewer

c. Block simulation

In this block simulation, the system controller has 4 states where it has different effect on the output signals such as the out_load_multiplier and out_shift_multiplicand. The behavior of the states can be shown in the following table.

state	S0	S1	S2	S3	
outputs	start = '1'	counter +=1		counter over flows	reset = '1'
out_load_multiplier	1	0	0	0	X
out_shift_multiplicand	1	0	0	0	X
out_reset_product	1	0	0	0	1
out_load_product	0	1	0	0	X
out_shift_product	0	0	1	0	X
out_shift_multiplier	0	0	1	0	X
out_done	0	0	0	1	0

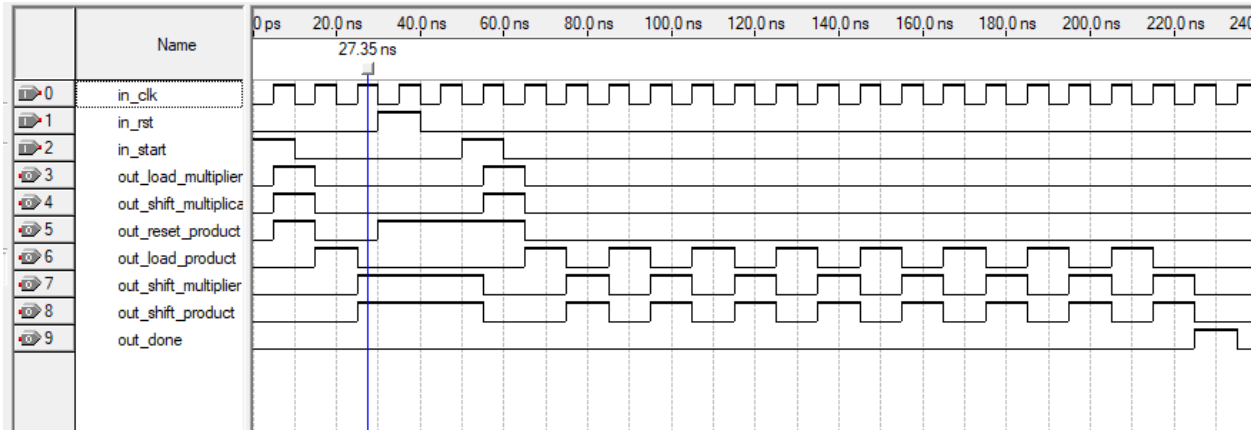


Figure 12 system controller simulation test

6- Complete circuit

a. VHDL code

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity Sequential_Multiplier_8bits is
generic (n : integer := 8);
port(
clk , rst , start : in std_logic;
A , B : in std_logic_vector ((n-1) downto 0);
done : out std_logic;
P : out std_logic_vector ((2*n-1) downto 0)
);
end entity;
Architecture behave of Sequential_Multiplier_8bits is
-- signals
-------------
--signals for the multiplicand
-- load input for the multiplicand REG
signal shift_multiplicand : std_logic;
-- in this case it will have the value of B
signal Multiplicand : std_logic_vector ((n-1) downto 0);
--signals for the multipier
-- multiplier(0) is the indicator it the mutilpilicand is the same or '0's
signal multiplier : std_logic_vector ((n-1)downto 0);
--load_multiplier , shift_multiplier
signal load_multiplier , shift_multiplier : std_logic;
-- signals for the full adder
--inputs of the adder
signal adder_x, adder_y : std_logic_vector ((n-1)downto 0);
signal adder_out : std_logic_vector (n downto 0);
--signals for the output of the adder and the final stage
signal product : std_logic_vector ((2*n-1)downto 0);
signal reset_product , load_product , shift_product : std_logic;
-----------------------------------------------------------------------------------------
-- components
---------------
-- register (used for loading B)
Component REG_8bits is
generic (size : integer := 8);
port(
data_in : in std_logic_vector (size-1 downto 0);
```

```
in_load , in_clk : in std_logic;
data_out        : out std_logic_vector (size-1 downto 0)
);
end component;
-- shift right register 8 bits (used for loading A)
component Shift_REG_R_8bits is
generic (size : integer := 8);
port(
data_in : in std_logic_vector (size-1 downto 0);
in_load , in_clk , in_shift : in std_logic;
data_out        : out std_logic_vector (size-1 downto 0)
);
end component;
-- Full adder 8 bits
component Full_Add_8bits is
generic (size : integer := 8 );
port(
x,y : in std_logic_vector ((size-1) downto 0);
s         : out std_logic_vector (size downto 0)
);
end component;
-- shift right register 16 bits (used for loading A)
component Shift_REG_R_16bits is
generic (size : integer := 8);
port(
data_in : in std_logic_vector (size downto 0);
in_load , in_clk , in_shift , in_rst : in std_logic;
data_out        : out std_logic_vector ((2*size-1) downto 0)
);
end component;
--system controller
component system_controller is
port (
in_clk : in std_logic ;
in_rst, in_start : in std_logic := '0' ;
out_load_multiplier, out_shift_multiplier : out std_logic;
out_shift_multiplicand : out std_logic;
out_reset_product ,out_load_product, out_shift_product : out std_logic;
out_done : out std_logic
);
end component;
```

```
-- start behave
begin
-- port mapping phase
From_B_to_multiplicand : REG_8bits
port map (data_in => B , in_load => shift_multiplicand , in_clk => clk , data_out =>
Multiplicand);
-----------------------------------------------------------
From_A_to_multiplier : Shift_REG_R_8bits
port map (data_in => A , in_load => load_multiplier , in_clk => clk , in_shift => shift_multiplier ,
data_out => multiplier);
-----------------------------------------------------------
From_OutputOfAND_to_FullAdder : Full_Add_8bits
port map (x=> adder_x ,y=> adder_y ,s=> adder_out);
--from the Adder to the product
From_FullAdder_to_product : Shift_REG_R_16bits
port map ( data_in => adder_out , in_load => load_product , in_clk => clk , in_shift =>
shift_product , in_rst => reset_product , data_out => product);
-- system controller
sys_con : system_controller
port map (in_clk => clk , in_rst => rst , in_start => start ,
                    out_load_multiplier =>load_multiplier , out_shift_multiplier=> shift_multiplier,
                    out_shift_multiplicand => shift_multiplicand,
                    out_reset_product => reset_product ,out_load_product => load_product,
out_shift_product => shift_product ,
                    out_done => done );
-----------------------------------------------------------------
adder_y <= product(15 downto 8);
-----------------------------------------------------------
Anding:process(load_multiplier , shift_multiplicand)
        begin
                for i in 0 to 7 loop
                        adder_x(i) <= multiplicand(i) and multiplier(0);
                end loop;
        end process;
-----------------------------------------------------------
p <= product;
end behave;
```

b. RTL viewer

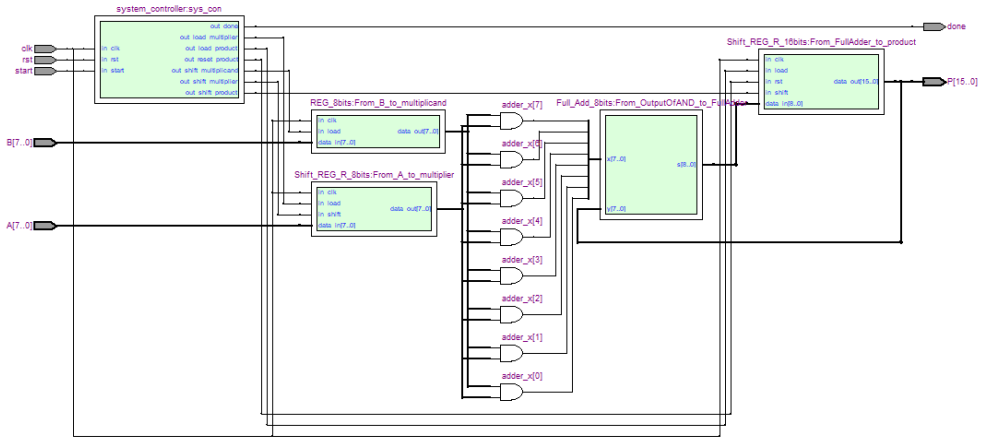


Figure 13 Sequential Multiplier RTL circuit viewer

4. Test bench for the complete design.

1- Test Bench code format

```
library ieee;
use ieee.std_logic_1164.all;
entity Sequential_multiplier_8bits_tb is
end entity;
architecture test of Sequential_multiplier_8bits_tb is
------------------------------------------------------------
signal A , B : std_logic_vector(7 downto 0);
signal clk , rst , start : std_logic ;
signal done : std_logic;
signal P : std_logic_vector (15 downto 0);
------------------------------------------------------------
component Sequential_Multiplier_8bits is
generic (n : integer := 8);
port(
clk , rst , start : in std_logic;
A , B : in std_logic_vector ((n-1) downto 0);
done : out std_logic;
P : out std_logic_vector ((2*n-1)downto 0)
);
end component;
begin
T1 :  Sequential_Multiplier_8bits
port map (clk,rst,start,A , B ,done,P);
```

```
clk <= '0', '1' after 10 ns, '0' after 20 ns, '1' after 30 ns,'0' after 40 ns,'1' after 50 ns,'0' after 60 ns;
start <= '1' , '0' after 20ns;
--rst <= '0', '1' after 40 ns, '0' after 60 ns;
a <= "00000101";
b <= "00001111";
end test;
```

Since this version of Quartus is not compatible with this format of Test Bench files, so this project will use a simple Vector Waveform to analysis the different cases

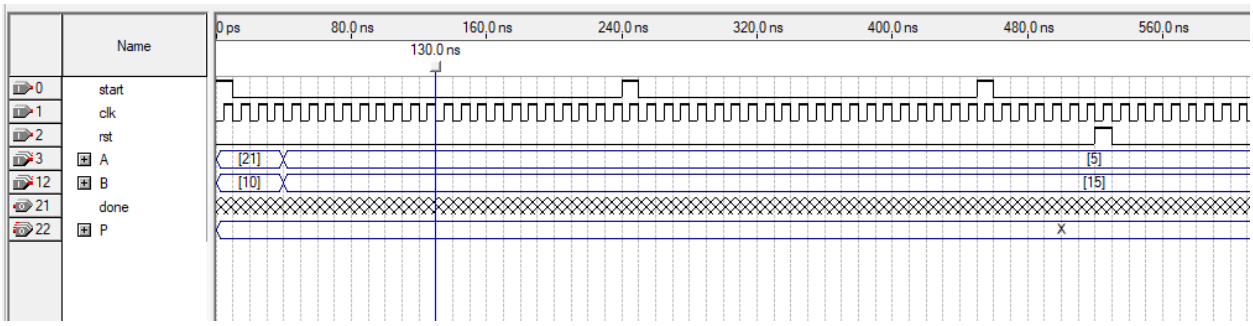


Figure 14 test bench vector waveform setup

2- Case 1 (constant input)

In the first test case the inputs are kept constant in order to make sure that there is no probability of error from the inputs during this test.

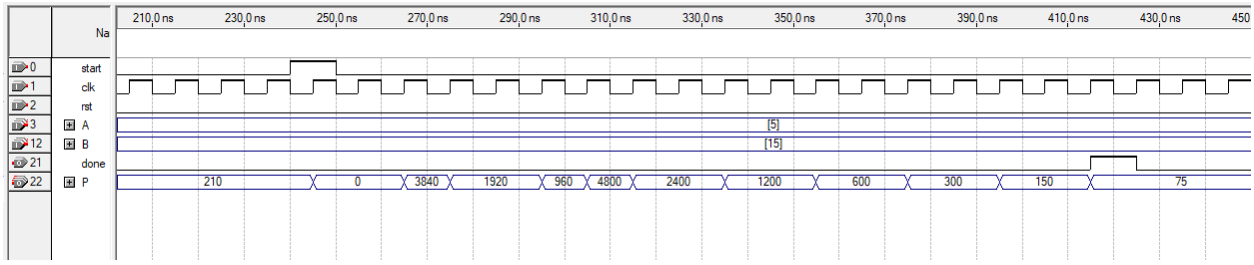
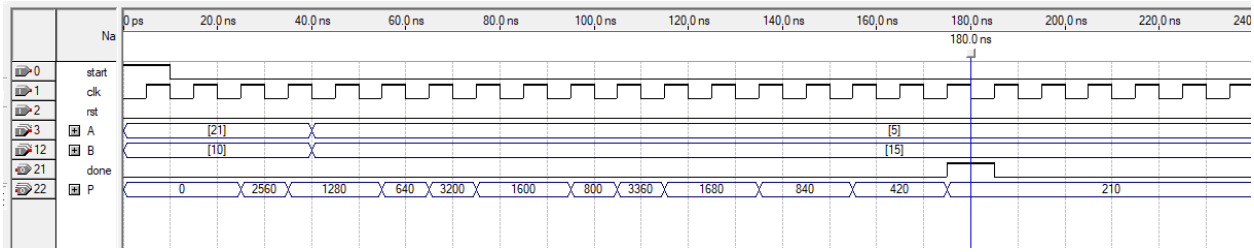


Figure 15 Case 1 (constant input)

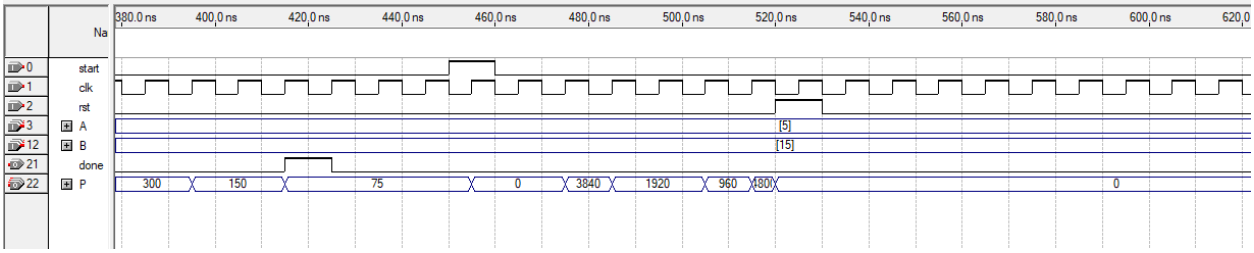
3- Case 2 (instantaneous input)

In this case the inputs are instantaneous, which means that the inputs are only available for a short amount of time (during the loading phase to the registers)



4- Case 3 (reset)

In this case, the last test is the reset of the circuit. The reset acts as a force stop where the output becomes zeros ‘0’ and it also stops the process of multiplication and starts the process from the beginning and waits for the starts signal.



5. Synthesis for your design.

After finishing implementing this project and testing all the cases for the wanted circuit, the code is uploaded on an Altair development kit to be tested in real life. In order to burn (upload) the tested code to be tested in real life, an FPGA kit is used. The upload is being processed according to the datasheet of the used type of kits.

6. Conclusion.

In this project, the output of the product P is out after 2*n clocks, which means 16 in this case, as there are 2 states that are responsible for the processing of the multiplication which are S1 and S2.

In case of separating the state loading and the process of the state itself, the variable counter that is introduced in the state machine in order to make a loop like effect in the states, the counter behaves independently outside the wanted scope. And for that the counter is needed to be controlled by the clock of the system. This is one of the reasons why the output P is produced after 2*n clocks.