به نام خدا

عنوان:

تکلیف شماره دوم درس طراحی سیستم‌های دیجیتالی

نام درس:

طراحی سیستم‌های دیجیتالی

استاد درس:

دکتر مروستی

اعضا:

محمدعلی مجتهدسلیمانی-992023039

تاریخ:

20/02/1403

**Q1.** For verify a 4-bit carry look-ahead adder we should first make a test bench to simulate all possible inputs and check the outputs against expected values which test bench going to do that. For the following code I assumed that you already have a Carry-Look-Ahead Adder module, for my code I already done it in first homework question 4.

* VHDL Test Bench Code:

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.STD\_LOGIC\_ARITH.ALL;

use IEEE.STD\_LOGIC\_UNSIGNED.ALL;

entity TestBench\_CLA is

end TestBench\_CLA;

architecture behavior of TestBench\_CLA is

component Carry\_Look\_Ahead\_Adder

Port (

A : in STD\_LOGIC\_VECTOR(3 downto 0);

B : in STD\_LOGIC\_VECTOR(3 downto 0);

Cin : in STD\_LOGIC;

Sum : out STD\_LOGIC\_VECTOR(3 downto 0);

Cout : out STD\_LOGIC

);

end component;

-- Test inputs

signal A, B: STD\_LOGIC\_VECTOR(3 downto 0);

signal Cin: STD\_LOGIC;

signal Sum: STD\_LOGIC\_VECTOR(3 downto 0);

signal Cout: STD\_LOGIC;

begin

-- Instantiate the Unit Under Test (UUT)

uut: Carry\_Look\_Ahead\_Adder port map (

A => A,

B => B,

Cin => Cin,

Sum => Sum,

Cout => Cout

);

-- Stimulus process to apply test vectors

stimulus: process

begin

-- Apply a set of test vectors

A <= "0000"; B <= "0000"; Cin <= '0'; wait for 10 ns;

A <= "0001"; B <= "0011"; Cin <= '0'; wait for 10 ns;

A <= "0101"; B <= "0101"; Cin <= '1'; wait for 10 ns;

A <= "1111"; B <= "1111"; Cin <= '1'; wait for 10 ns;

-- Complete the testing

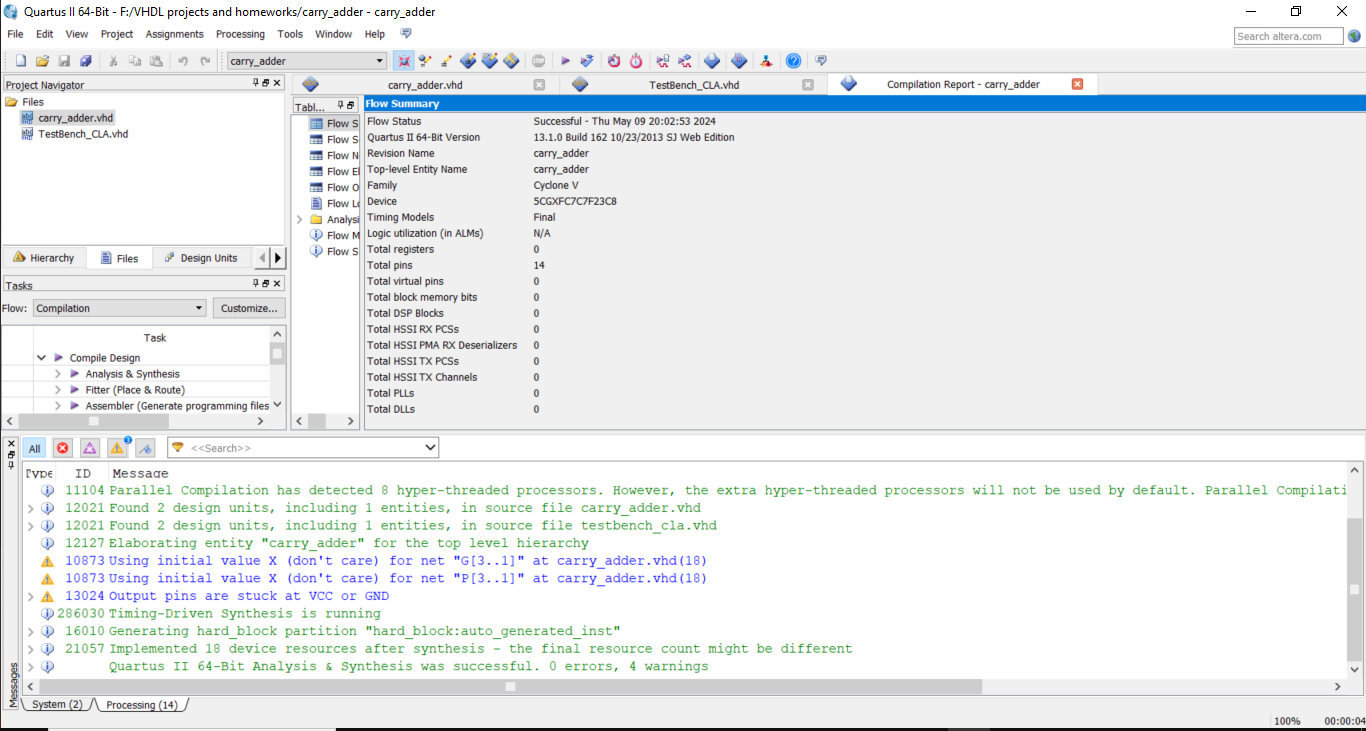
-- wait, then end the process

wait;

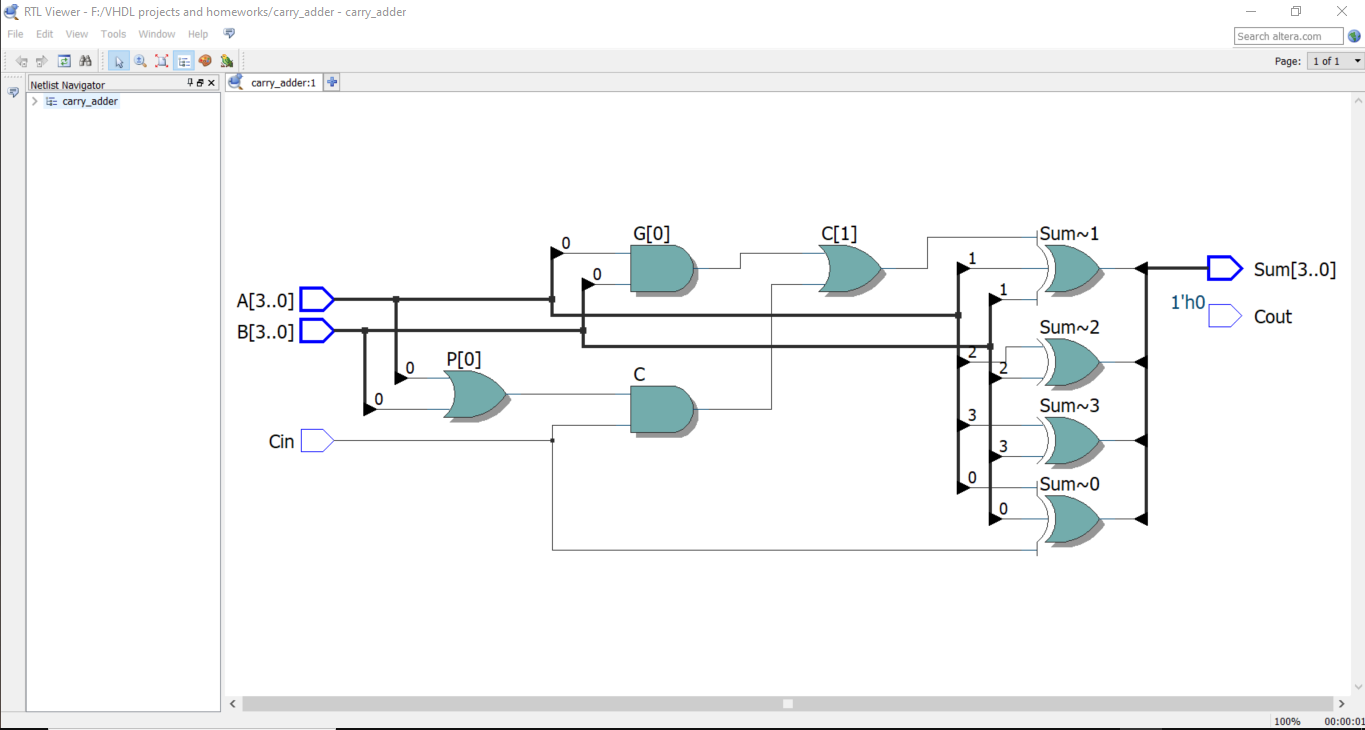
end process;

end behavior;

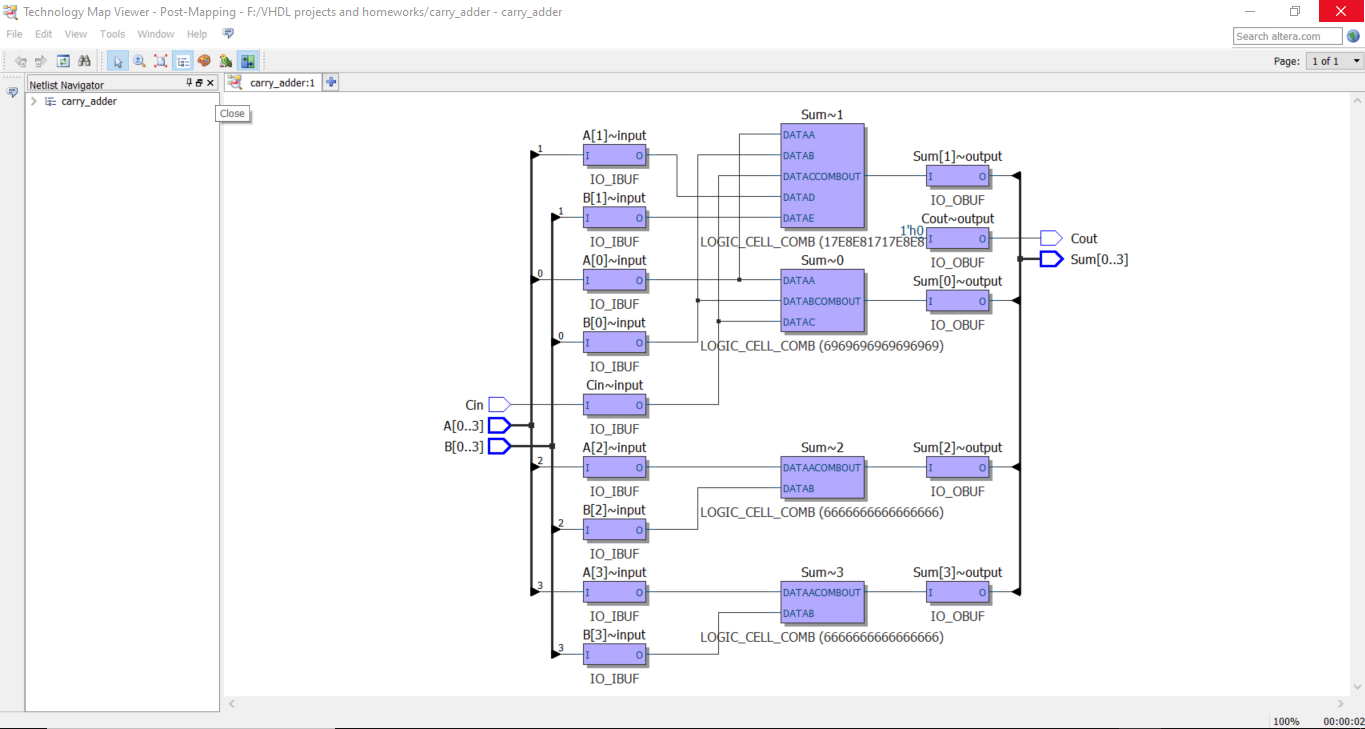
* Complication Report:



* RTL view:

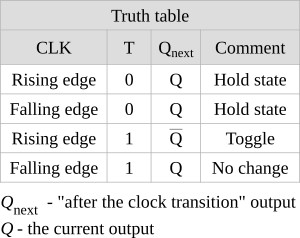


* Post-mapping view:



**Q2.** For a T flip flop, we the next state of flip flop driven by XOR T (input of circuit) and Q which is the current state of our circuit so based on this and based on t ff truth table I wrote a code for it. First here is a truth table of t ff:

* **T FF truth table:**



* **Code:**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity tff is

Port ( clk : in STD\_LOGIC;

async\_reset : in STD\_LOGIC;

Q : out STD\_LOGIC);

end entity tff;

architecture toggle of tff is

signal S\_Q : std\_LOGIC := '0';

begin

process(clk, async\_reset)

begin

if async\_reset = '1' then

S\_Q <= '0';

elsif rising\_edge(clk) then

S\_Q <= not S\_Q;

end if;

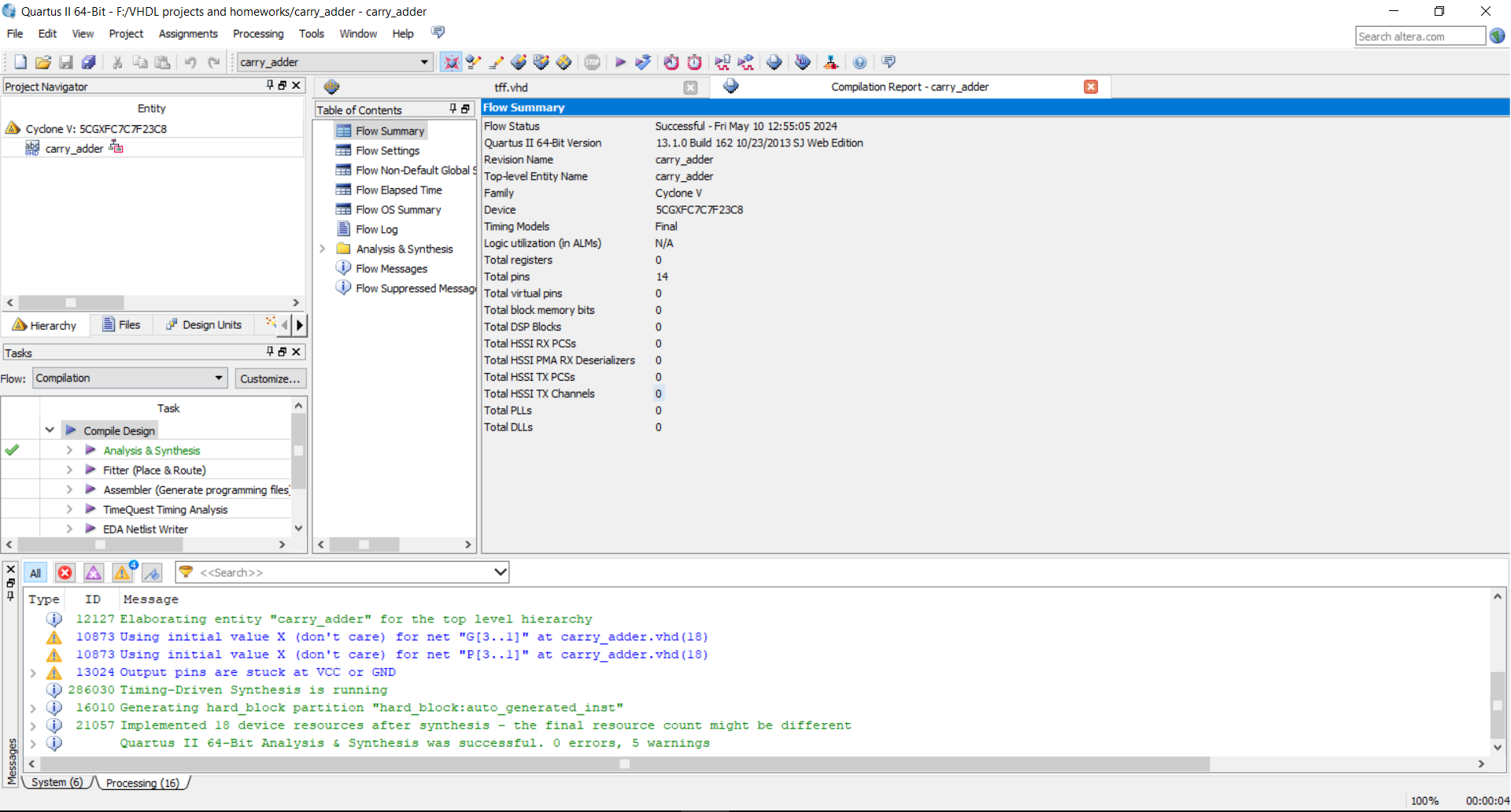
end process;

Q <= S\_Q;

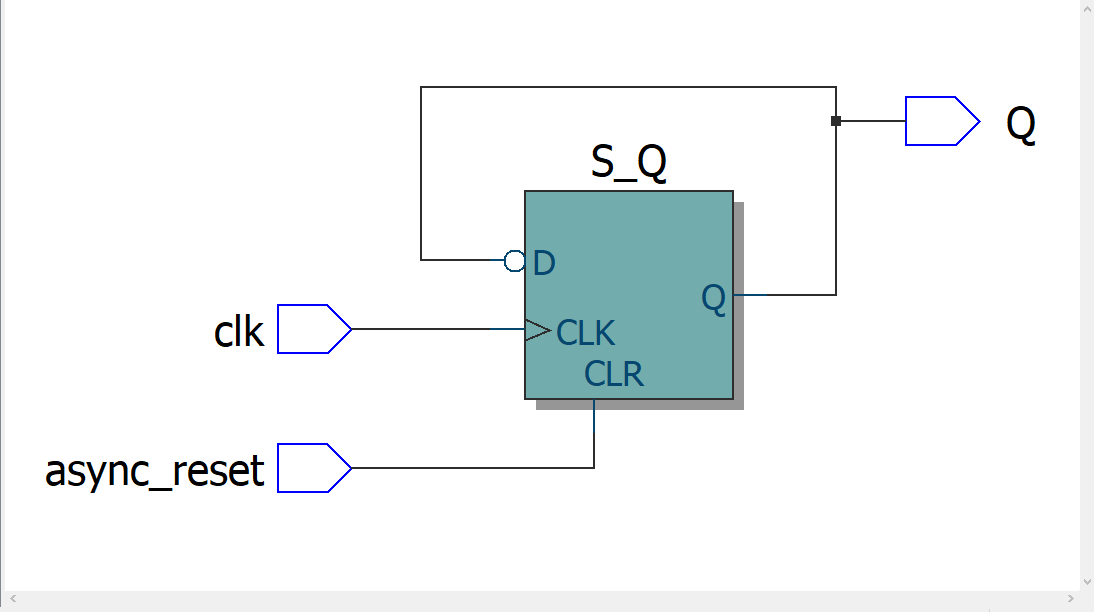
end architecture toggle;

**\***: in this code I declared 2 inputs: clk for clock pin of circuit and the async\_reset for reset pin of circuit. The output is Q which is next state of circuit. Reset pin Is asynchronous from clock pin and it is active high so it means when pin becomes 1 the output of circuit be 0. I used signals to work with outputs Q because we can not directly work with Q. signal is a wired in VHDL.

* **Output after compile**:

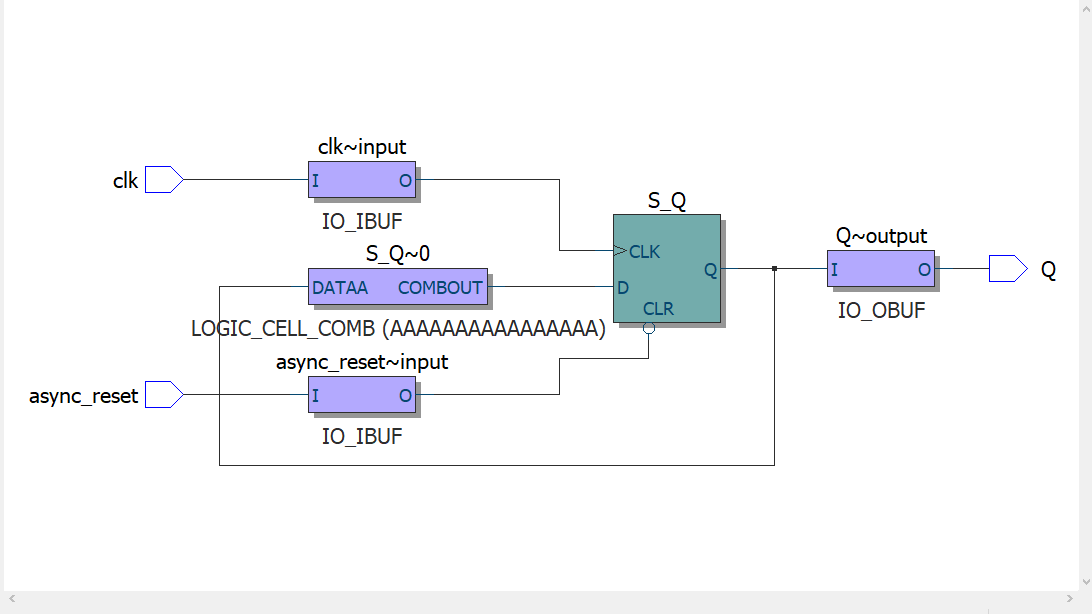


* **RTL view**:

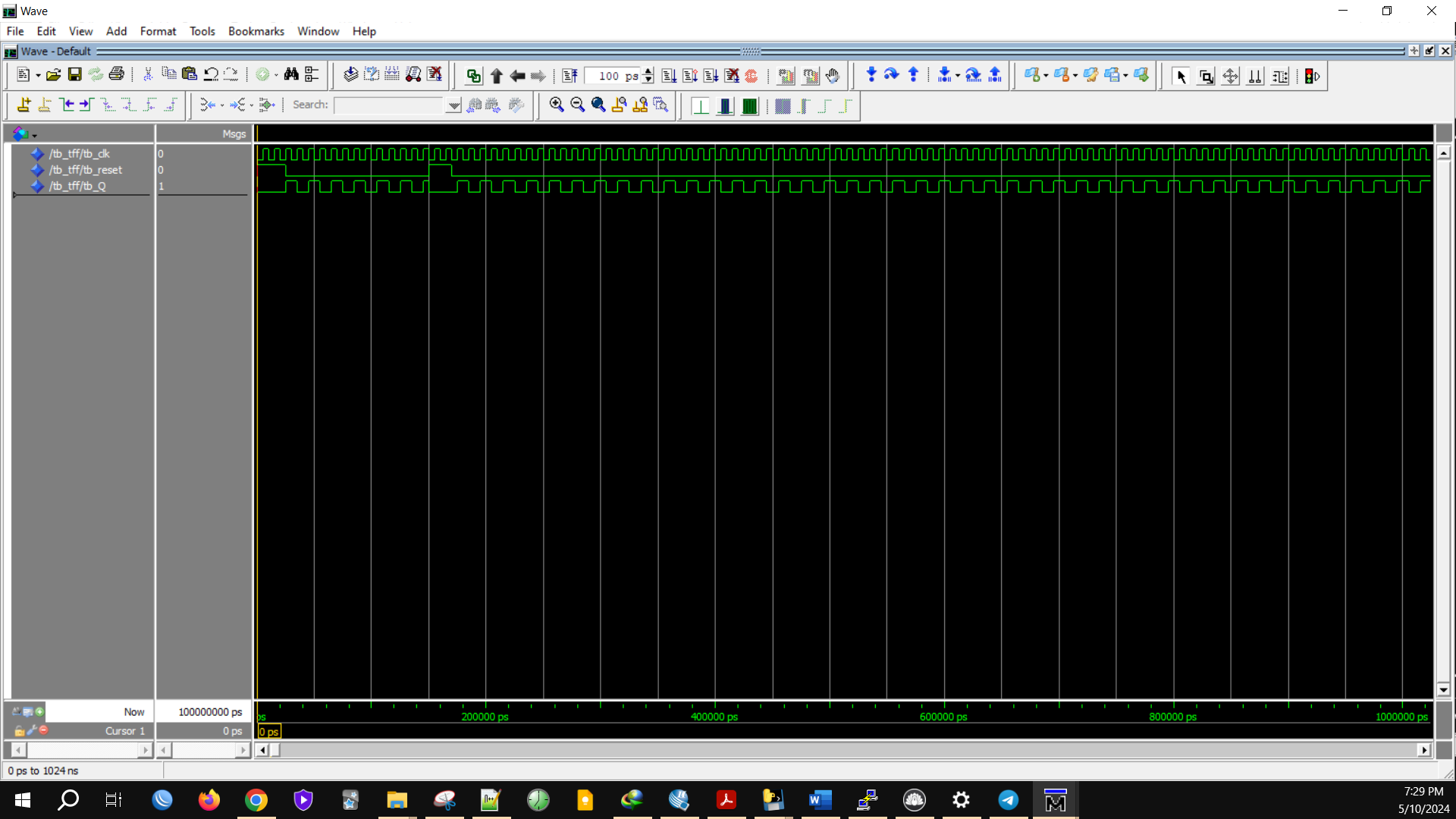


**\***: notice that in this code we built a T flip flop using D flip flop. This is achieved by feeding the output of the D Flip flop to its input through an XOR gate.

* **Post-mapping view**:



* **Wave Form:**



* **Test bench code:**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity tb\_tff is

-- Testbench does not have any ports

end tb\_tff;

architecture behavior of tb\_tff is

-- Component Declaration for the T Flip-Flop

component tff

Port ( clk : in STD\_LOGIC;

async\_reset : in STD\_LOGIC;

Q : out STD\_LOGIC);

end component;

-- Signals for interfacing with the component

signal tb\_clk : STD\_LOGIC := '0';

signal tb\_reset : STD\_LOGIC := '0';

signal tb\_Q : STD\_LOGIC;

-- Clock period definition

constant clk\_period : time := 10 ns;

begin

-- Instantiate the T Flip-Flop

uut: tff port map (

clk => tb\_clk,

async\_reset => tb\_reset,

Q => tb\_Q

);

-- Clock process definitions

clk\_process : process

begin

while true loop

tb\_clk <= '0';

wait for clk\_period/2;

tb\_clk <= '1';

wait for clk\_period/2;

end loop;

end process;

-- Stimulus process

stim\_proc: process

begin

-- Test 1: Reset the flip-flop

tb\_reset <= '1';

wait for 25 ns;

tb\_reset <= '0';

wait for 25 ns;

-- Test 2: Allow some clock pulses to toggle the flip-flop

tb\_reset <= '0';

wait for 100 ns;

-- Test 3: Assert reset while clock is running

tb\_reset <= '1';

wait for 20 ns;

tb\_reset <= '0';

wait for 100 ns;

-- End of testing

wait;

end process;

end behavior;

Q3. Designing a 4-bit BCD (Binary-Coded Decimal) up/down counter in VHDL involves creating a circuit that counts from 0 to 9 and back again, with the count direction controlled by an input signal **Dir**.

* **Code:**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.STD\_LOGIC\_UNSIGNED.ALL;

entity BCD\_Counter is

Port (

clk : in STD\_LOGIC;

reset : in STD\_LOGIC;

Dir : in STD\_LOGIC; -- '0' for down, '1' for up

count : out STD\_LOGIC\_VECTOR(3 downto 0)

);

end BCD\_Counter;

architecture Behavioral of BCD\_Counter is

signal r\_count : STD\_LOGIC\_VECTOR(3 downto 0) := "0000";

begin

process(clk, reset)

begin

if reset = '1' then

r\_count <= "0000"; -- Reset the counter

elsif rising\_edge(clk) then

if Dir = '1' then -- Count up

if r\_count = "1001" then -- Max value for BCD

r\_count <= "0000"; -- Wrap around

else

r\_count <= r\_count + 1;

end if;

else -- Count down

if r\_count = "0000" then

r\_count <= "1001"; -- Wrap around from 0 to 9

else

r\_count <= r\_count - 1;

end if;

end if;

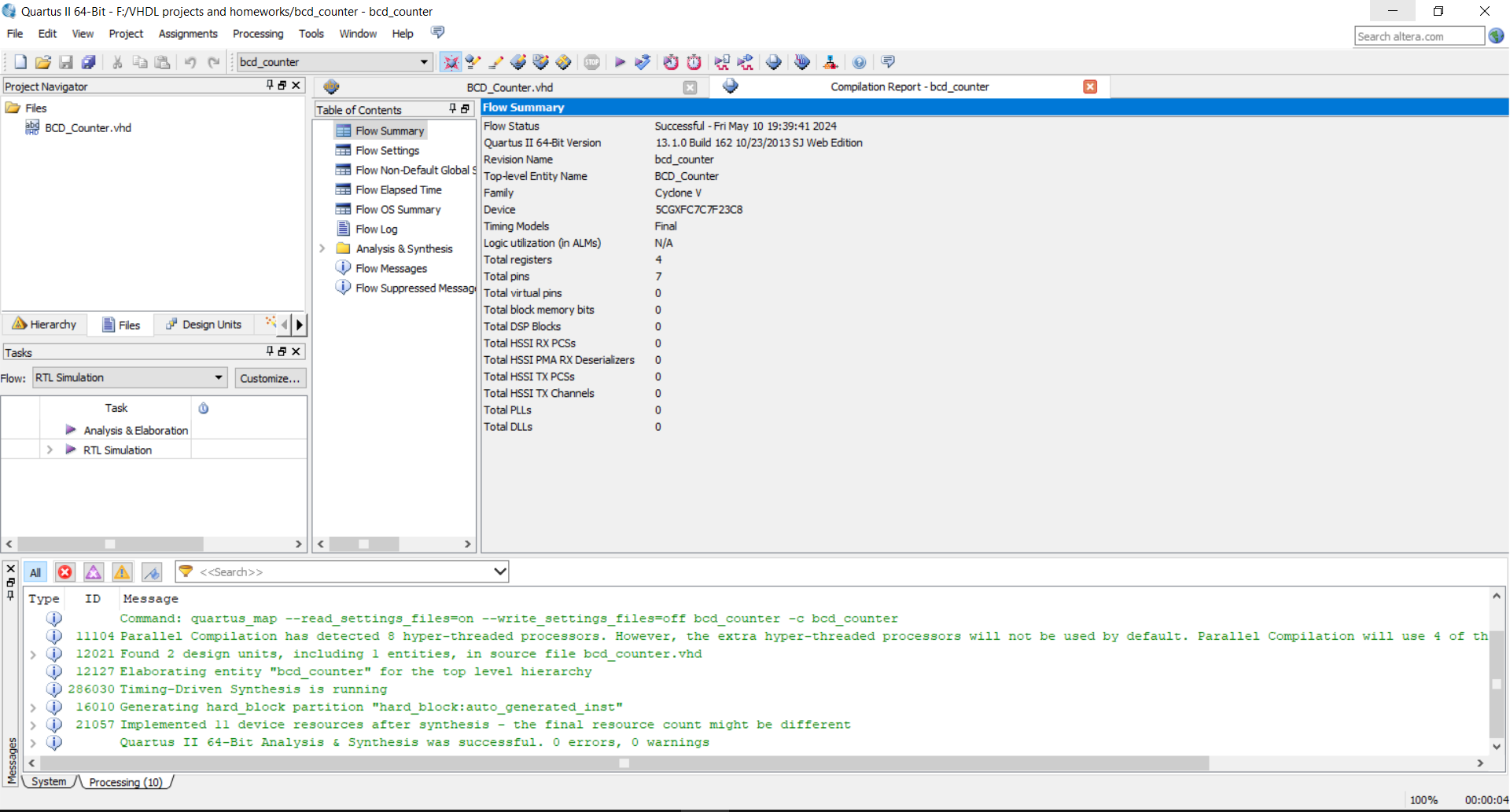
end if;

end process;

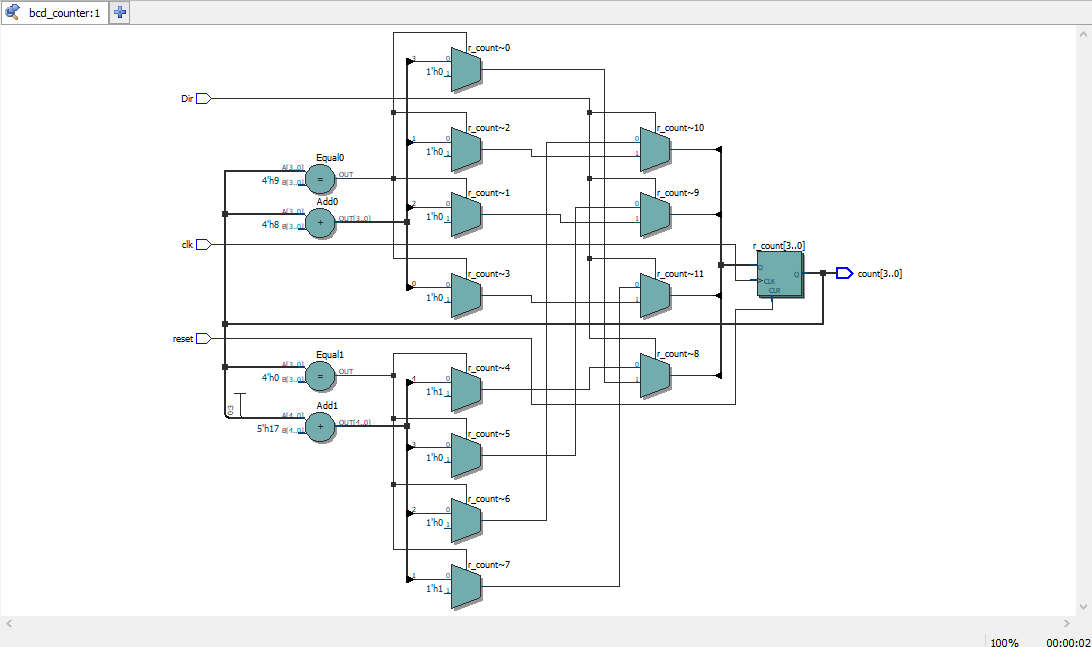
count <= r\_count; -- Output the current count

end Behavioral;

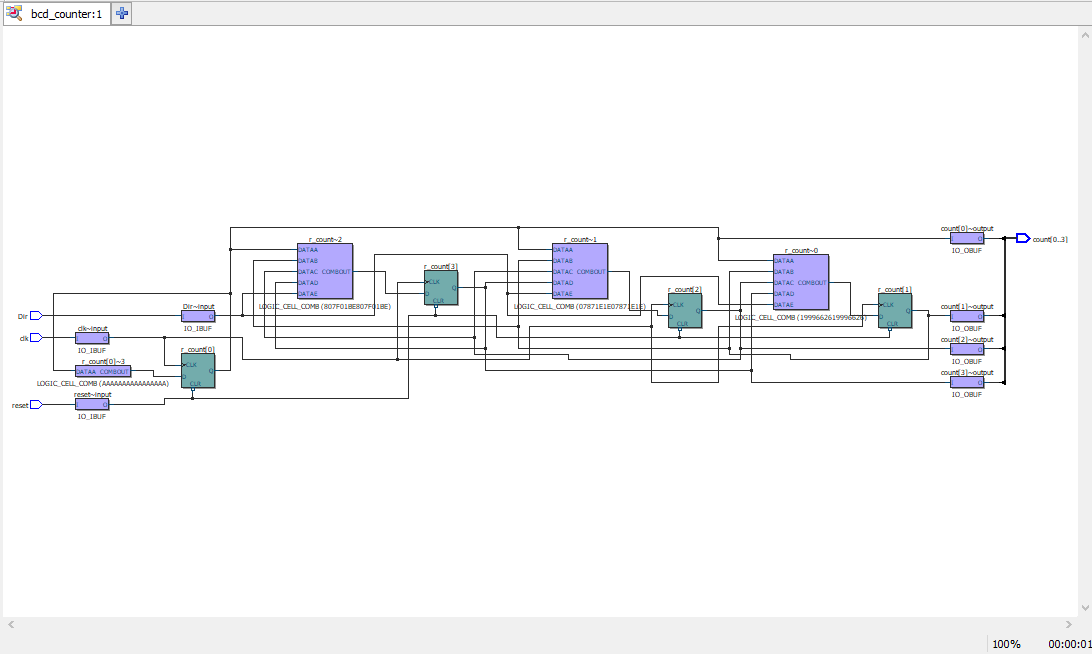
* **compilation report:**



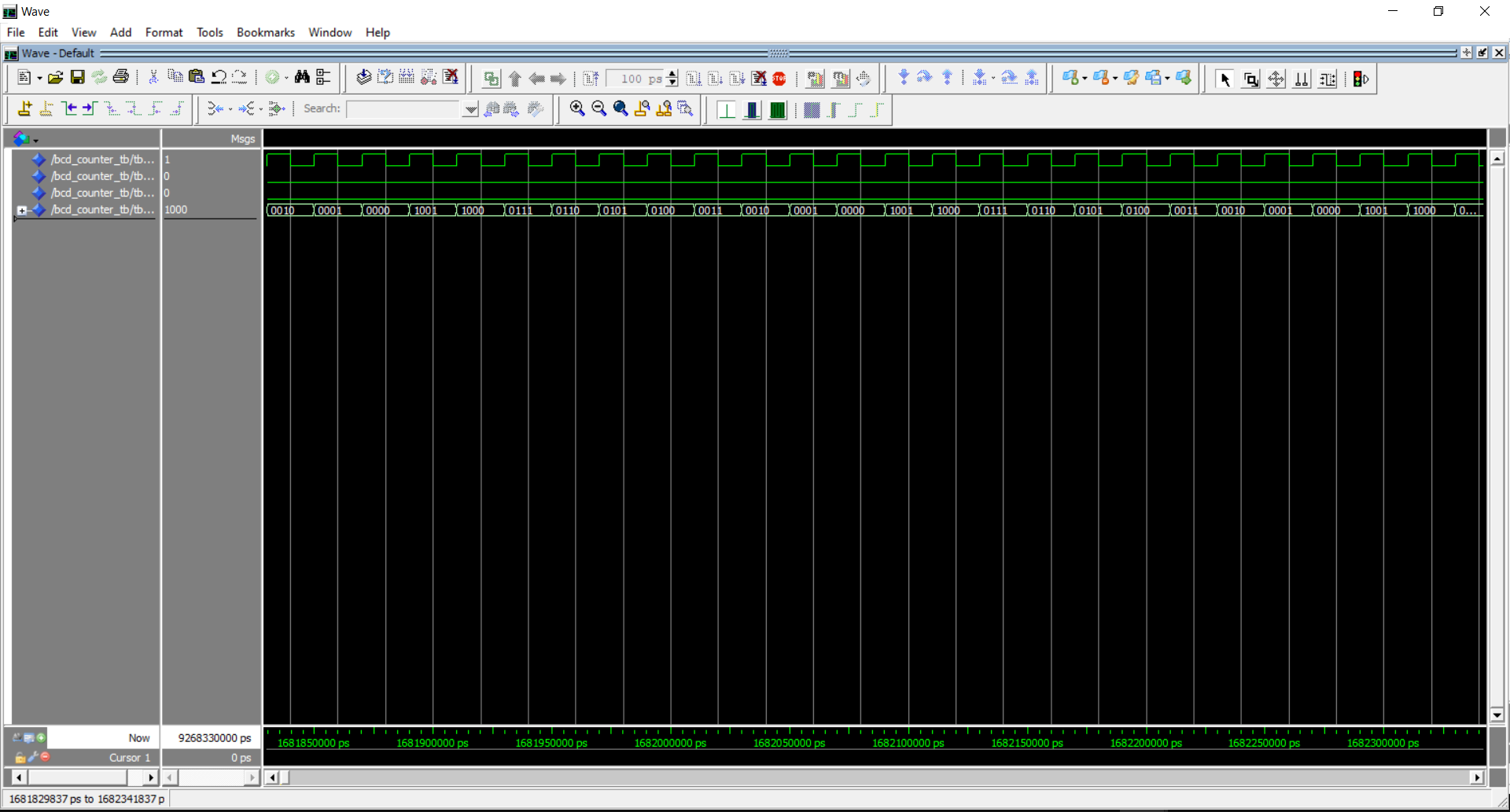
* **RTL view:**



* **Post-mapping view:**



* **Wave form:**



* **Test bench code:**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.STD\_LOGIC\_UNSIGNED.ALL;

entity BCD\_Counter\_TB is

-- Testbench has no ports

end BCD\_Counter\_TB;

architecture behavior of BCD\_Counter\_TB is

-- Component Declaration

component BCD\_Counter

Port (

clk : in STD\_LOGIC;

reset : in STD\_LOGIC;

Dir : in STD\_LOGIC;

count : out STD\_LOGIC\_VECTOR(3 downto 0));

end component;

-- Test Signals

signal tb\_clk : STD\_LOGIC := '0';

signal tb\_reset : STD\_LOGIC := '0';

signal tb\_Dir : STD\_LOGIC := '0';

signal tb\_count : STD\_LOGIC\_VECTOR(3 downto 0);

begin

-- Clock process

clk\_process : process

begin

tb\_clk <= '0';

wait for 10 ns; -- Clock period of 20 ns

tb\_clk <= '1';

wait for 10 ns;

end process;

-- Instantiate the Unit Under Test (UUT)

uut: BCD\_Counter port map (

clk => tb\_clk,

reset => tb\_reset,

Dir => tb\_Dir,

count => tb\_count);

-- Test Stimulus Process

stim\_proc: process

begin

-- Initialize Inputs

tb\_reset <= '1'; -- Assert reset initially

wait for 40 ns;

tb\_reset <= '0'; -- De-assert reset

wait for 20 ns;

-- Test Count Up

tb\_Dir <= '1'; -- Set direction to up

wait for 220 ns; -- Allow enough time to observe the count

-- Test Count Down

tb\_Dir <= '0'; -- Change direction to down

wait for 220 ns; -- Allow enough time to observe the count

-- Finish simulation

wait;

end process;

end behavior;

Q4. to write a VHDL code that implement pattern transformation for the given table I’ll create a state machine where each state corresponds to one of the input patterns and transition to the corresponding next pattern.

Code:

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

-- Define the entity for the pattern transformation

entity Pattern\_Finder is

Port (

clk : in STD\_LOGIC;

reset : in STD\_LOGIC;

input\_pattern : in STD\_LOGIC\_VECTOR(2 downto 0);

next\_pattern : out STD\_LOGIC\_VECTOR(2 downto 0)

);

end Pattern\_Finder;

-- Architecture declaration using a behavioral model

architecture Behavioral of Pattern\_Finder is

signal current\_state, next\_state : STD\_LOGIC\_VECTOR(2 downto 0);

begin

-- State register process

process(clk, reset)

begin

if reset = '1' then

current\_state <= "000"; -- Default state

elsif rising\_edge(clk) then

current\_state <= next\_state;

end if;

end process;

-- Next state logic based on current state

process(current\_state)

begin

case current\_state is

when "000" => next\_state <= "011";

when "011" => next\_state <= "110";

when "110" => next\_state <= "101";

when "101" => next\_state <= "111";

when "111" => next\_state <= "000";

when others => next\_state <= "000"; -- Safety net

end case;

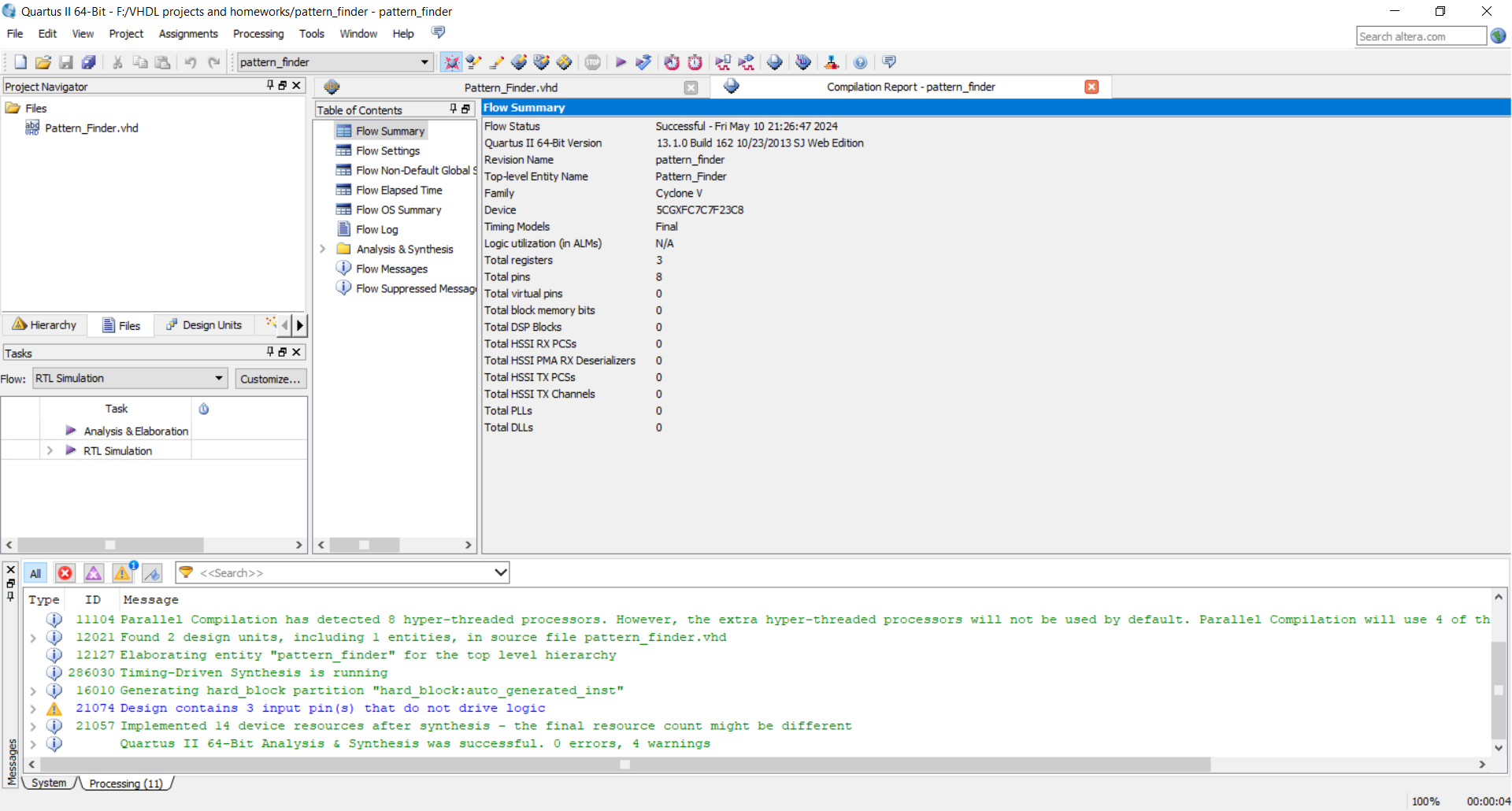
end process;

-- Output logic

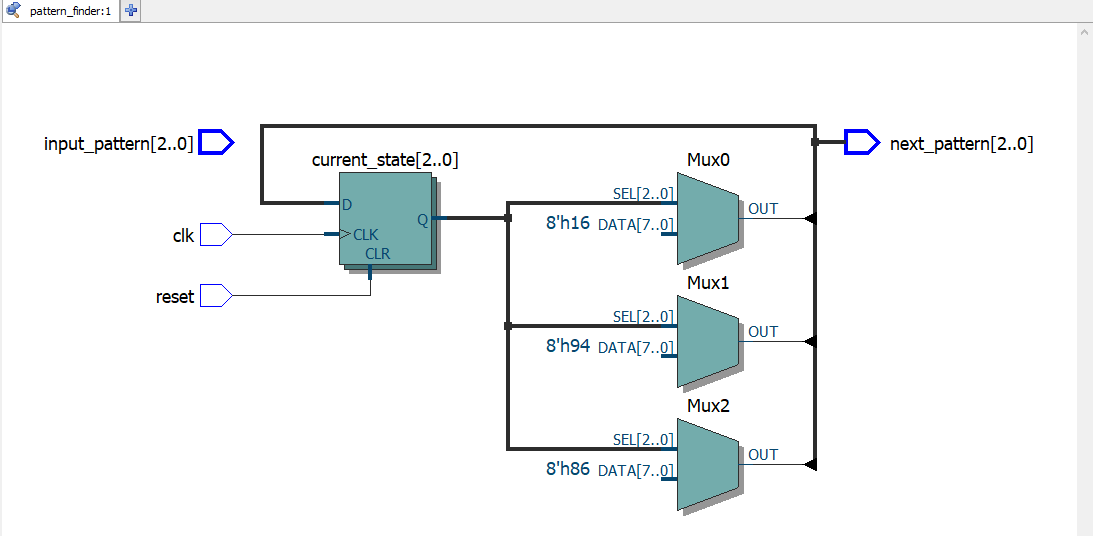
next\_pattern <= next\_state; -- Directly map next state to output

end Behavioral;

output after compilation:



RTL view:



Post-mapping view:

