

## Chapter-6

### How to use IP Cores

Intellectual property (IP) is a general name that covers unique creations of human intellect, and mostly encloses copyrights, patents, and trademarks. The term IP is also valid in FPGA implementations, i.e. called as IP cores. IP cores are configurable black boxes that can be added to main VHDL or Verilog based designs. Most of these cores were created by Xilinx. Some of them are allowed to be used free of charge while some of them requires charges. Moreover, third party IP cores also can be developed and presented to the other users with reasonable charges.

IP cores are classified with respect to their usage areas. Some of the classes are like below

- Basic Elements
  - Accumulators
  - Counters
  - Memory Elements
  - Registers, Shifters & Pipelining
- Communication & Networking
  - Error Correction
  - Ethernet
  - Modulation
  - Networking
  - Serial Interfaces
  - Telecommunications
  - Wireless
- Debug & Verification
- Digital Signal Processing
  - Building Blocks
  - Filters
  - Transforms
  - Trig Functions
  - Waveform Synthesis
- Math Functions
  - Adders & Subtractors
  - Conversions
  - CORDIC
  - Dividers
  - Square Root
  - Trig Functions.

There are many other IP cores that can be added to list. Aim of this chapter is to show how to use an IP core. A math function example, a clock manager example and a Read Only Memory (ROM) example is going to be demonstrated in detail throughout the chapter. It's important to

state that Some of the IP cores aren't valid for some FPGA chips. Artix XC7A100T should be selected at beginning of each example below.

**Example:** Design a 4-bit unsigned adder circuit by using Adder/Subtractor IP Core. Simulate the IP core based design on ISim with respect to below two sample calculations.

$$\begin{array}{r} A_3 A_2 A_1 A_0 \\ + B_3 B_2 B_1 B_0 \\ \hline C_{out} S_3 S_2 S_1 S_0 \end{array}$$

**Solution:** Step-1) As the first step to achieve example's task, create a new project and name it as IP\_CORE\_SAMPLE\_1. Additionally, name of the main design should be "Sum\_func". After that, **Adder/Subtractor** IP Core should be added to the main implementation files as a *component*. Right click to the FPGA chip and click **New Source**.

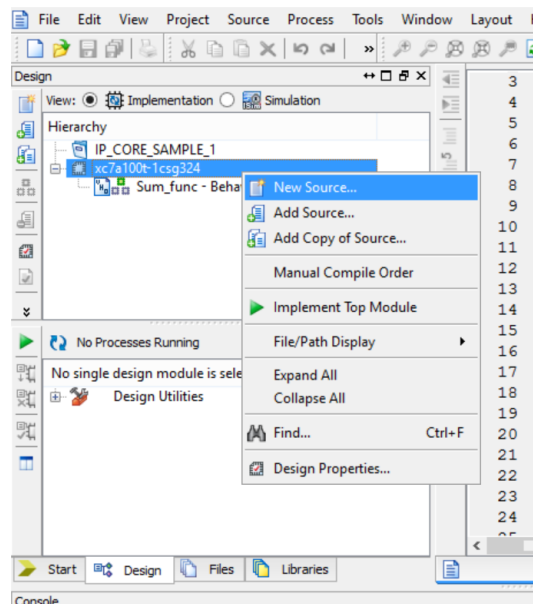
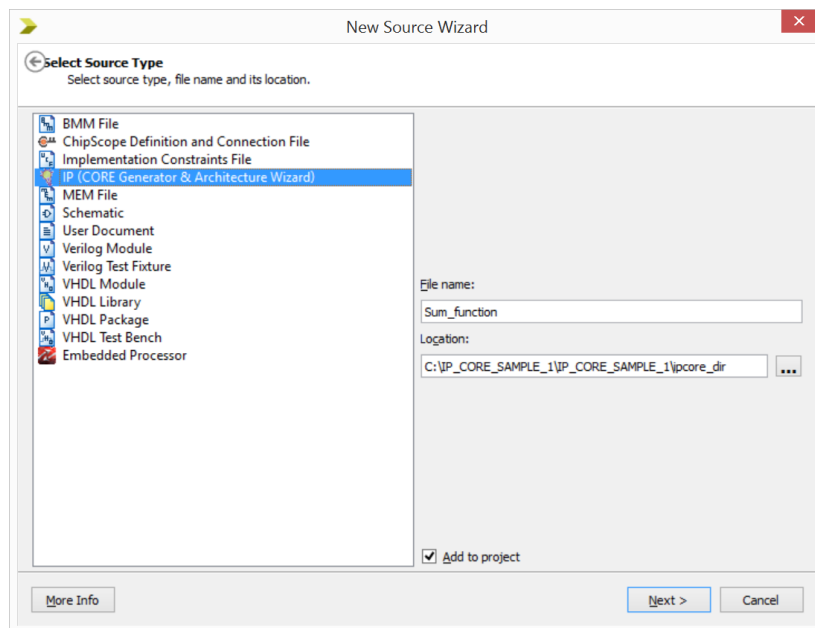


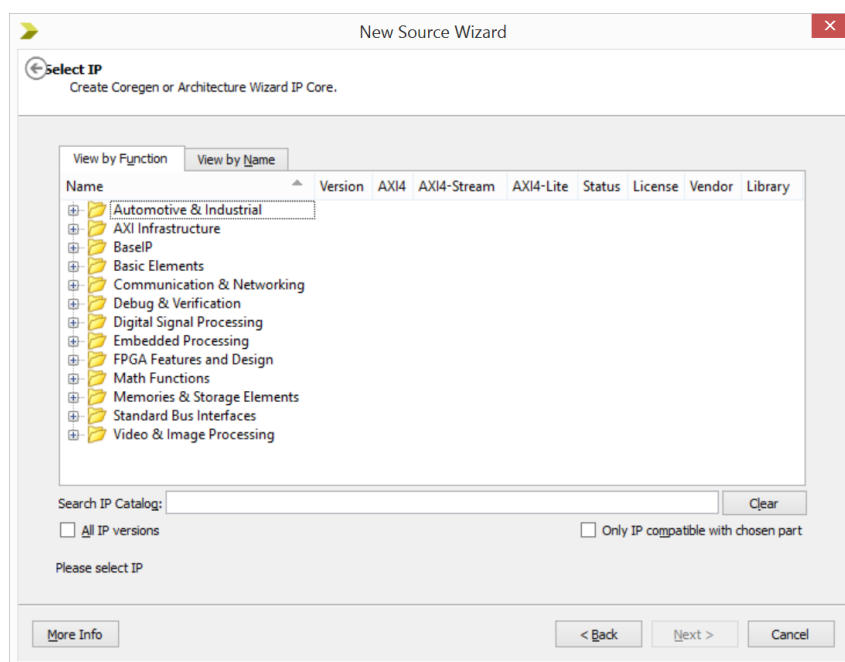
Figure 6-1

Choose **IP ( CORE Generator & Architecture Wizard)** and name it as "Sum\_function" . Click **Next**.



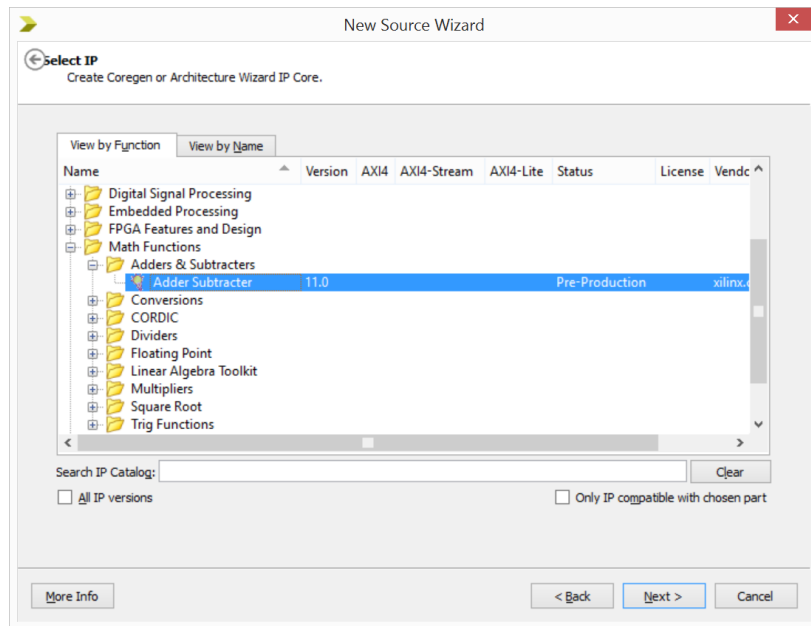
**Figure 6-2**

Step-2) As seen in Figure 6-3, there are many IP Core classes to be used.



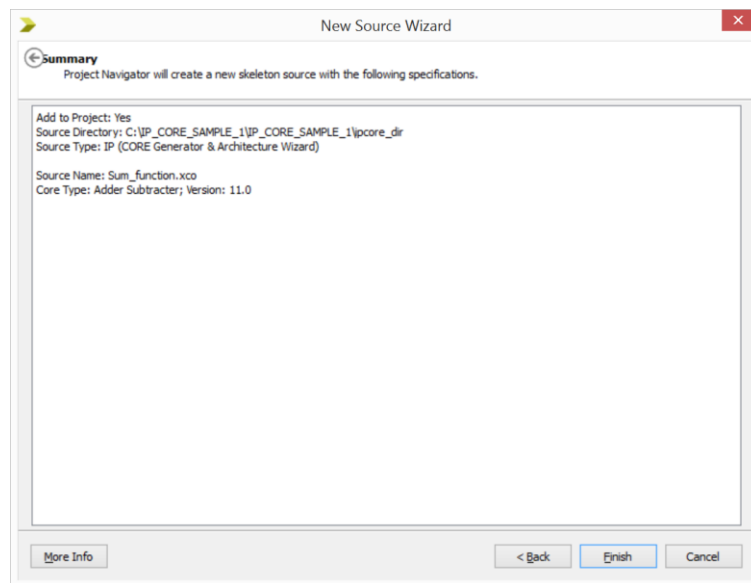
**Figure 6-3**

**Adder/Subtractor** core is under the class of *Math Functions*. Choose it and click **Next**.



**Figure 6-4**

Click **Finish** as seen in the Figure 6-5.



**Figure 6-5**

Step-3) Figure 6-6 is emerged and core configuration should be done to be able use the IP core as planned. This Figure shows the default arrangement of the core from this graphical user interface (GUI). On the left hand side of the figure, block diagram of the core can be seen. Inputs of the core appear on the left hand side of the diagram while outputs appear on the right hand side of the diagram.

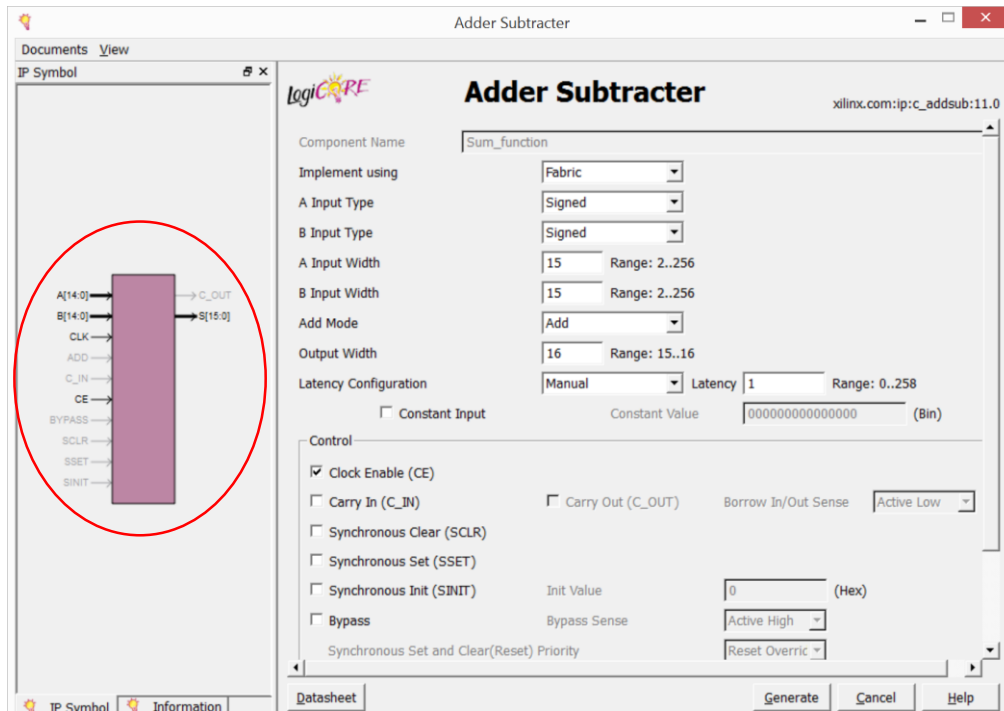


Figure 6-6

Aim of the example is to build 4-bit unsigned adder. Change Input type to unsigned and input width to 4. Check for the *Carry Out* option to be able to observe carry of the summation result. Moreover, set zero latency for simplicity during observation. As the configuration settings are changed inputs and outputs change. It's important to be aware of differences in between block diagrams of Figure 6-6 and Figure 6-7.

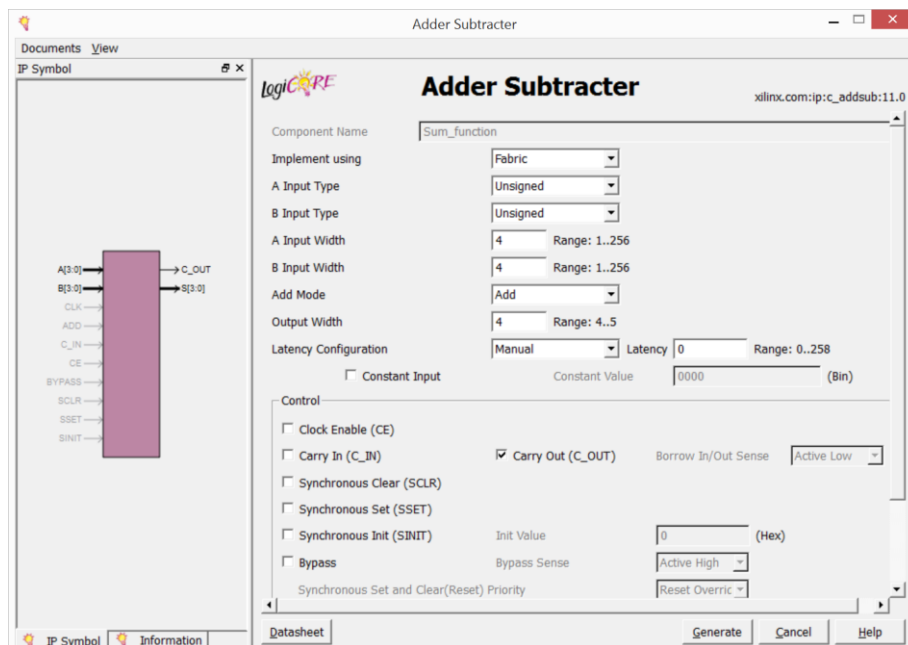


Figure 6-7

Each IP core has its own datasheet. When **Datasheet** button is clicked a pdf file will be opened as seen in Figure 6-8. These datasheets exist for all IP cores and should be read carefully in order to use them effectively.

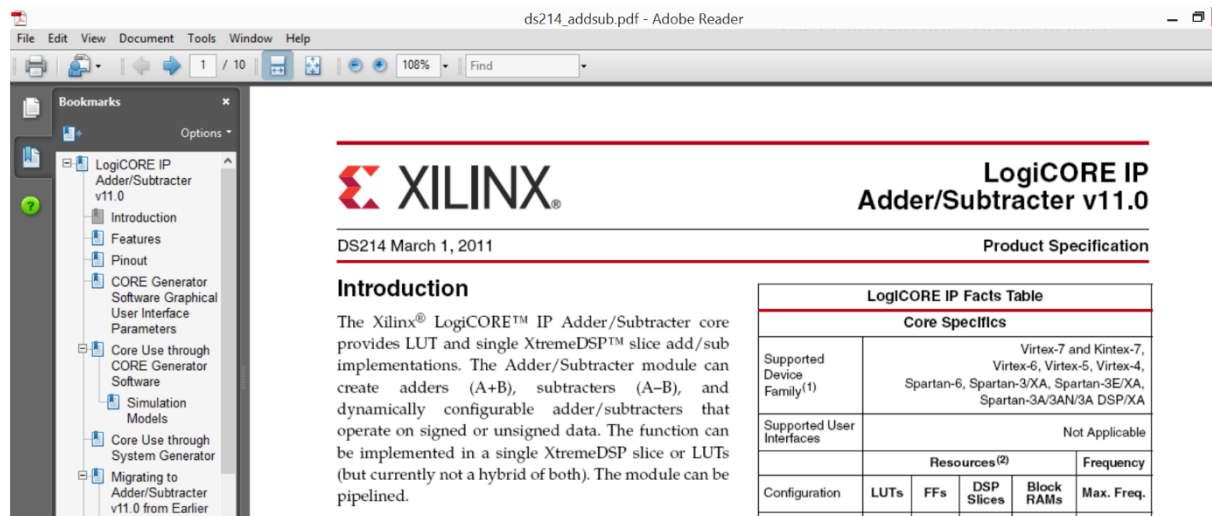


Figure 6-8

Click **Generate** button and IP core GUI will be closed. Configuration is done. If reconfiguration is needed double clicking to the core opens the GUI again. **Adder/Subtractor** core is added to design as seen in Figure 6-9.

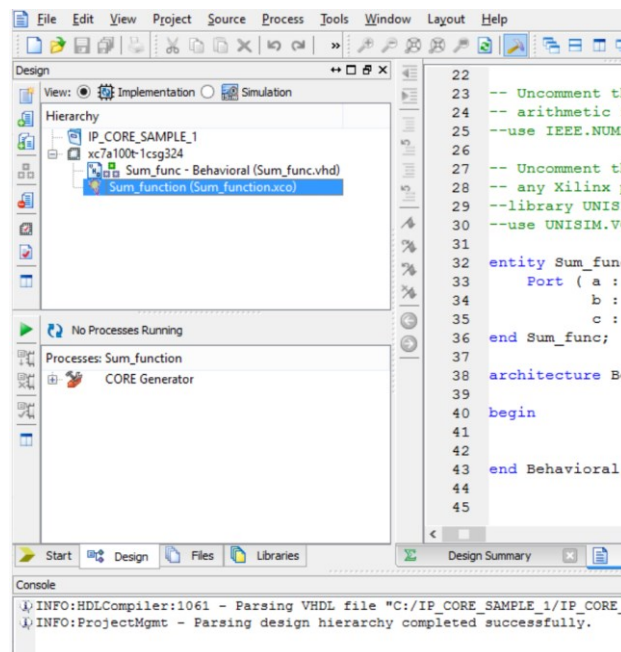
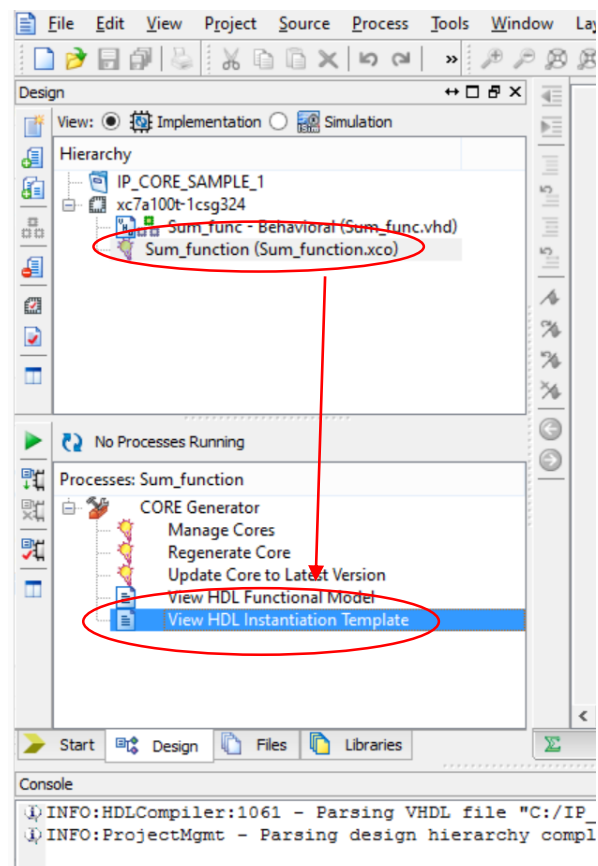


Figure 6-9

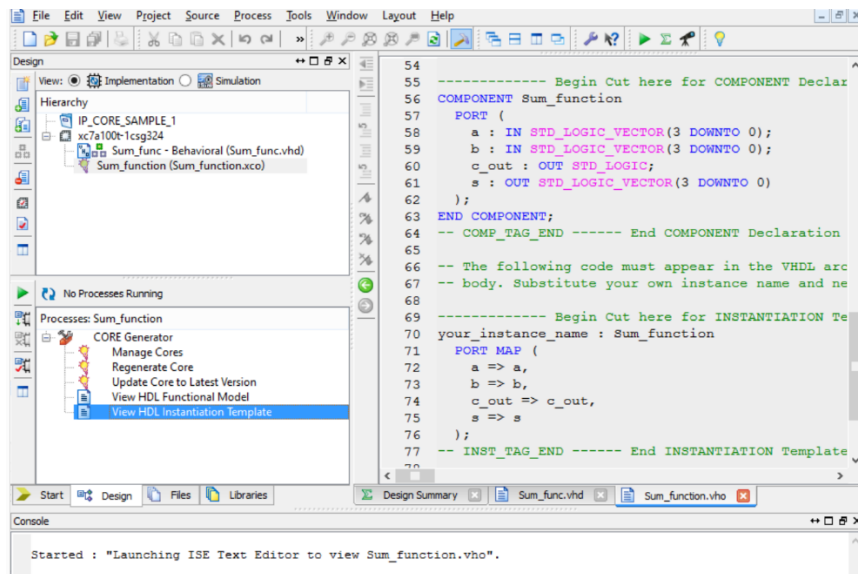
Step-4) IP core is added to design but it isn't tied to the main implementation of "Sum\_func".

Click **View HDL Instantiation Template** to be able to see port and component declarations of the generated IP core. This feature is presented by ISE in order to rapid component construction in main design.



**Figure 6-10**

Figure 6-11 shows the port and component declarations of the Adder/Subtractor IP core. As noticed, there are two 4-bit inputs and one 4 bit summation output and one bit carry output.



**Figure 6-11**

Arrange main implementation file as in PS 6-1.

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

entity Sum_func is
    Port ( a : in std_logic_vector (3 downto 0);
          b : in std_logic_vector (3 downto 0);
          c : out std_logic_vector (4 downto 0));
end Sum_func;

architecture Behavioral of Sum_func is

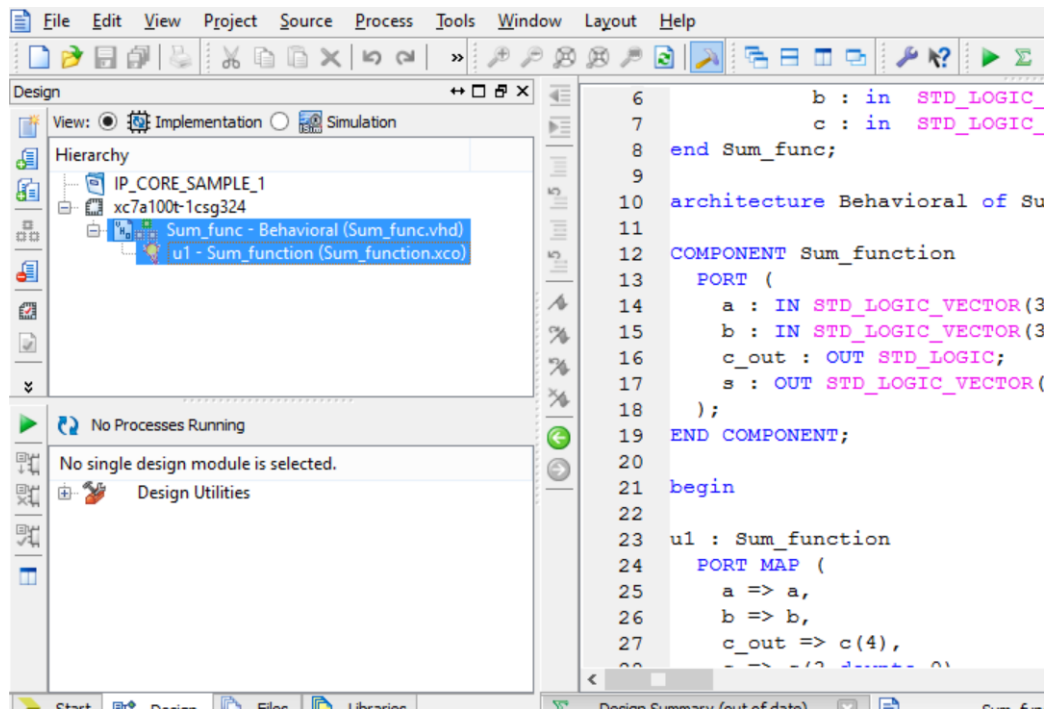
    component Sum_function
    port (
        a : in std_logic_vector(3 downto 0);
        b : in std_logic_vector (3 downto 0);
        c_out : out std_logic;
        s : out std_logic_vector (3 downto 0));
    end component;

begin
    u1 : Sum_function
    port map (
        a => a,
        b => b,
        c_out => c(4),
        s => c(3 downto 0));
end Behavioral;
```

**PS 6-1**

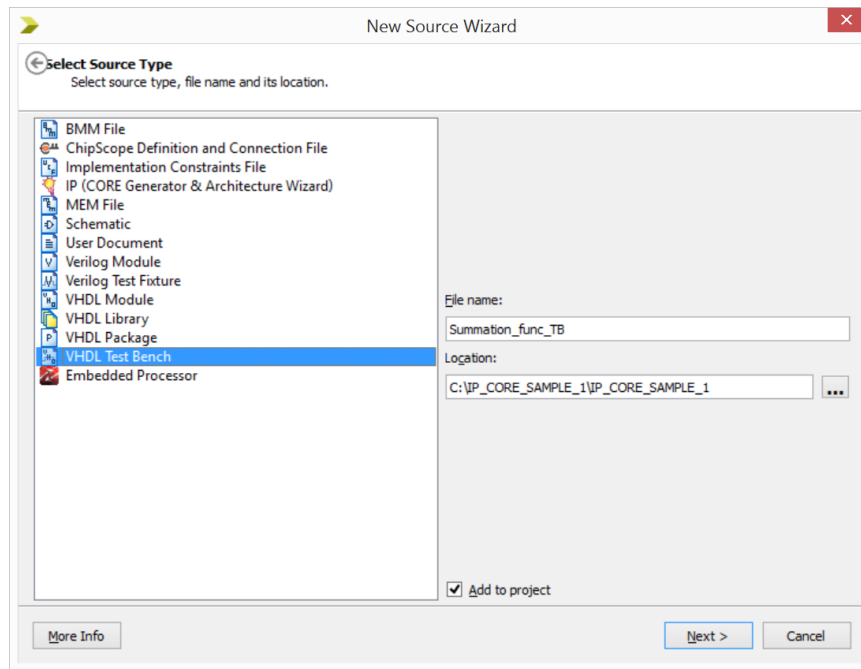


When PS 6-1 is completed IP Core – which is named as “u1-Sum\_function” becomes a sub unit of the main implementation design of “Sum\_func”.



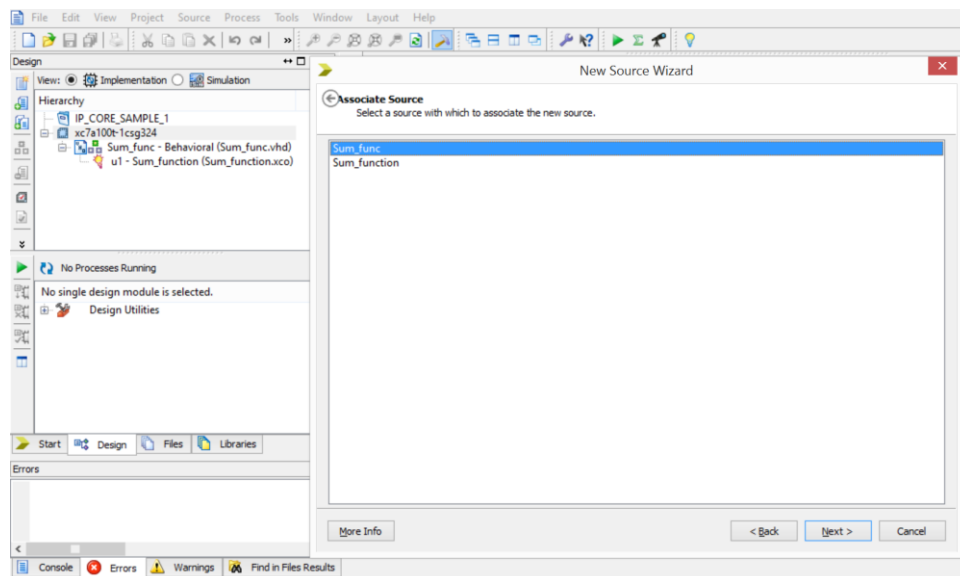
**Figure 6-12**

Step-5) Implementation is completed. Now, simulation scenario should be set. Open a VHDL Test Bench file and named it as “Summation\_func\_TB” and click **Next**.



**Figure 6-13**

In the next window of Figure 6-14, associate test bench with main implementation file of “Sum\_func” and click **Next**.



**Figure 6-14**

Arrange test bench as in PS 6-2.

```

library ieee;
use ieee.std_logic_1164.all;

entity Summation_func_TB is
end Summation_func_TB;

architecture behavior of Summation_func_TB is
    component Sum_func
    port(
        a : in std_logic_vector(3 downto 0);
        b : in std_logic_vector(3 downto 0);
        c : out std_logic_vector(4 downto 0)
    );
    end component;
    --Inputs
    signal a : std_logic_vector(3 downto 0) := (others => '0');
    signal b : std_logic_vector(3 downto 0) := (others => '0');
    --Outputs
    signal c : std_logic_vector(4 downto 0);

begin
    uut: Sum_func port map (
        a => a,
        b => b,
        c => c
    );
    stim_proc: process
    begin
        a <= "0001"; b <= "0010";
        wait for 100 ns;
        a <= "1001"; b <= "0111";
        wait for 100 ns;
    end process;
end;

```

PS 6-2

Figures 6-15 and 6-16 shows two different summation operations. As a result, Adder Subtractor block is used successfully.

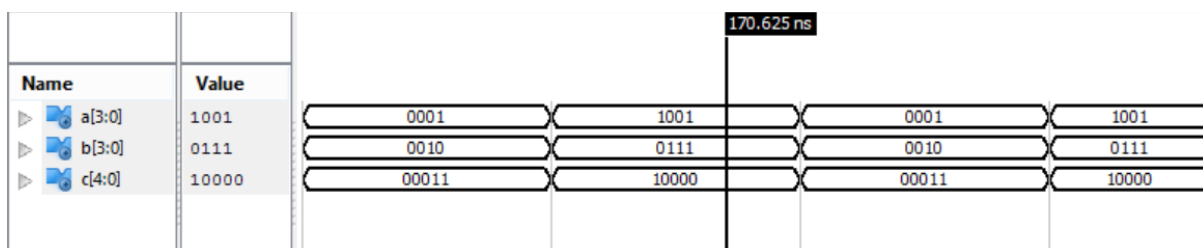


Figure 6-15

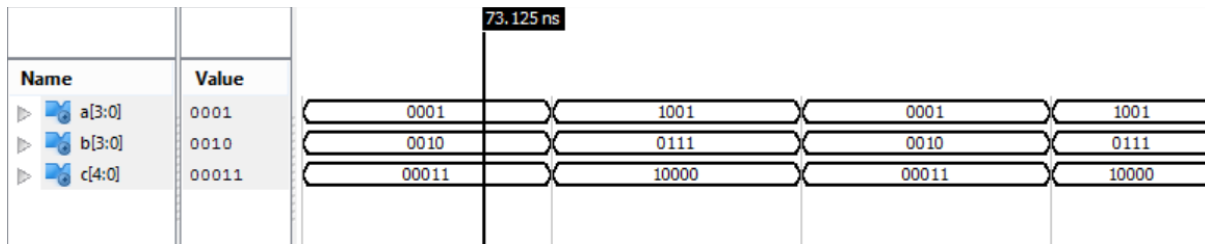


Figure 6-16

**Example:** Increase the clock rate of the FPGA from 100 MHz to 300 MHz. Observe the result in ISim.

**Solution:** Step-1) Open a new project and create a main VHDL design, name it as “Clock\_Manager”. This design should have an input port for 100 MHz clock and a output port for 300 MHz clock signal. Clocking Wizard IP core is suitable for such an aim. Add a new source as represented in Figure 6-2 and name it as “clock\_manager\_core”. After typing core name click **Next**. Click **View By Name** option in the window as seen in Figure 6-17 and choose **Clocking Wizard** IP core. Click **Next**.

**Clocking Wizard** inherits choices of Phase Locked Loop (PLL) and Mixed Mode Clock Manager (MMCM). The PLL is an analog clock management cell that can generate different phases of clock, does clock division and de-skew a clock. Moreover, it can generate different frequencies at the same time and has better jitter performance with respect to digital clock manager (DCM). MMCM cell is a simply PLL cell that is modified with DCM features. Since DCM has more precise phase shifting ability, analog and digital managers are used together.

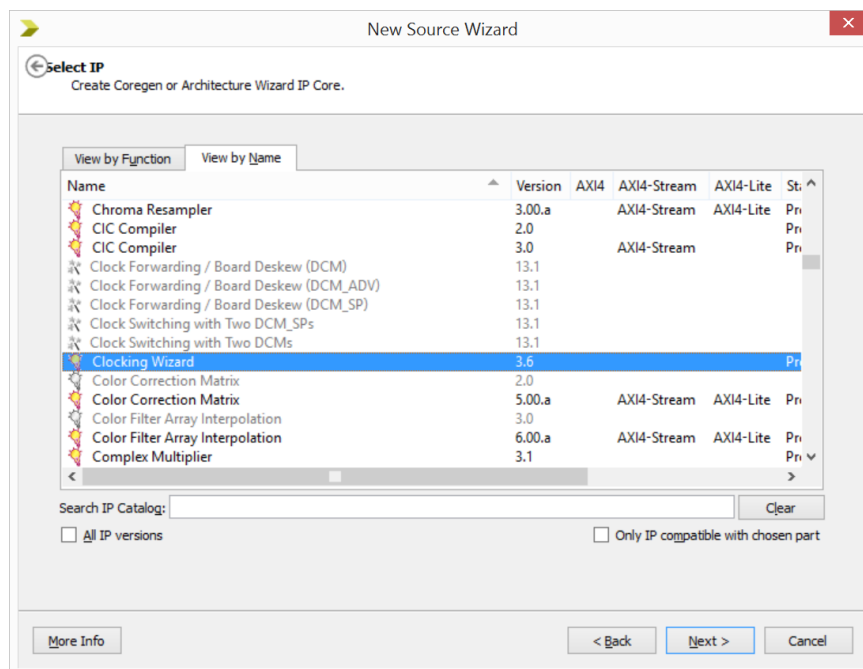


Figure 6-17

**Clocking Wizard** core includes six pages for configuration. GUI from Figure 6-18 to 6-21 shows required settings for the example's aim. Since Nexys 4 DDR board has 100 MHz oscillator, check the primary input clock value as seen in below Figure 6-18. Click Back and Next buttons to pass from page to page.

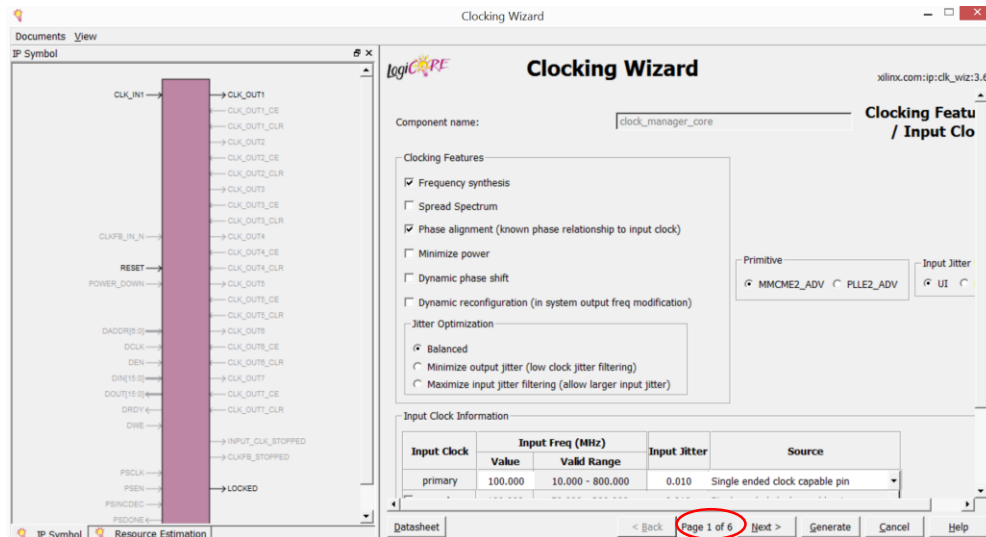


Figure 6-18

Arrange requested output frequency to 300 MHz liken in Figure 6-19.

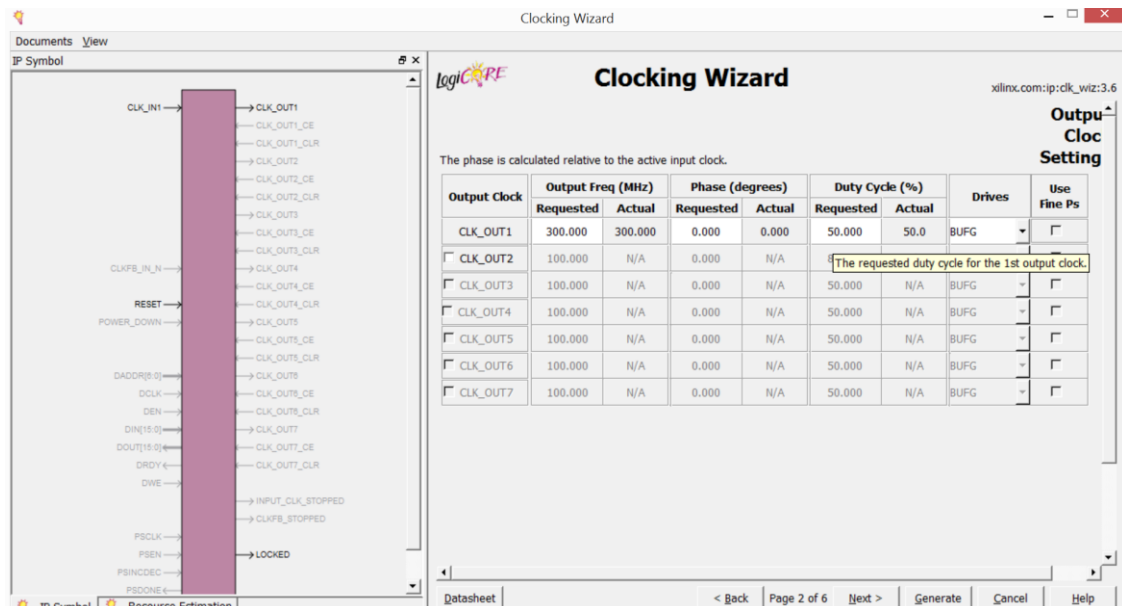


Figure 6-19

*Reset* and *Locked* features can be added to the design. Click on the check boxes to set them. *Reset* port is the input of the core while *Locked* is shown as output of the core in Figure 6-20. *Locked* output will be logic-1 when input and output clock signals are phase aligned.

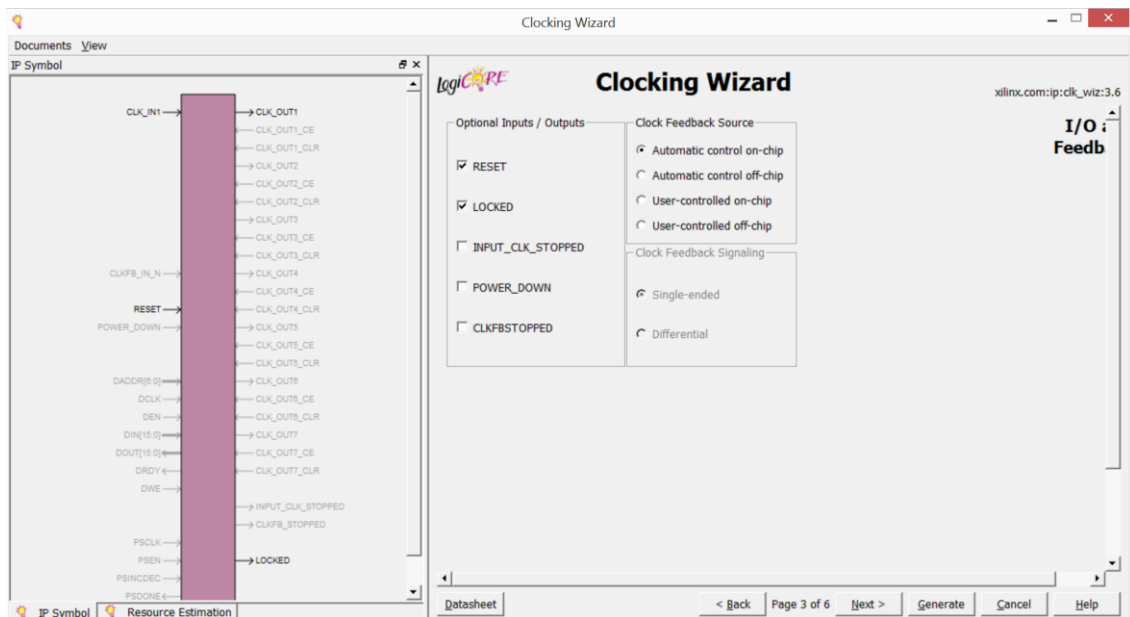


Figure 6-20

Don't change anything in page 4 and 5. Figure 6-21 shows the last page. This page shows the multiplier and divider settings to reach 300 MHz clock from 100 MHz. Multiplier coefficient is set as 10.125 while divider is set to 3.375 where

$$100\text{MHz} * \frac{10.125}{3.375} = 300\text{ MHz}.$$

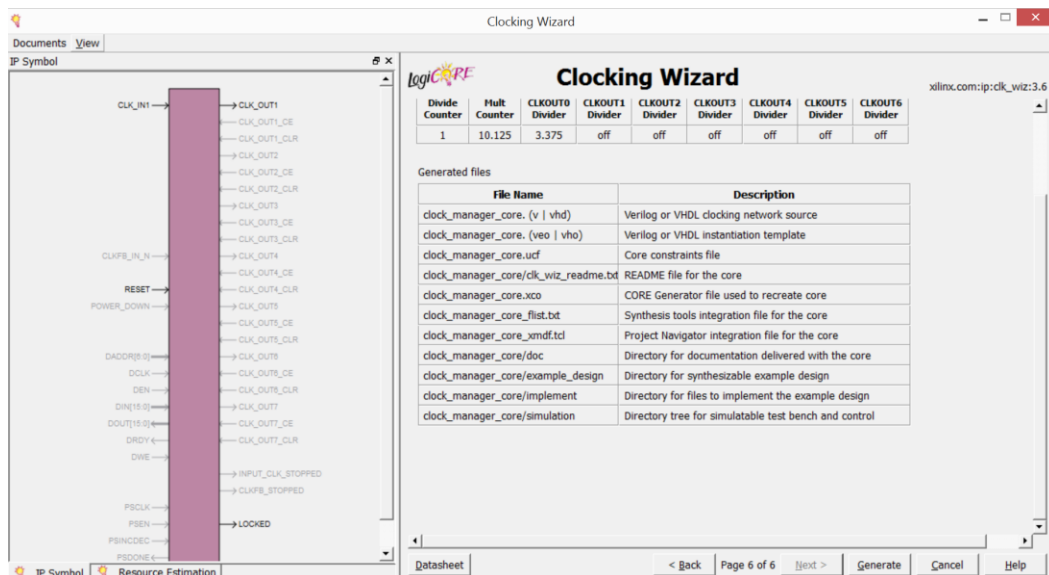


Figure 6-21

Configuration is done. Click **Generate** button.

Step-2) Arrange the main code as given in PS 6-3.

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

entity Clock_Manager is
  Port ( clk_100MHz,reset : in std_logic;
        clk_300MHz,clock_locked :out std_logic
        );
end Clock_Manager;

architecture Behavioral of Clock_Manager is
  component clock_manager_core
  port
  (
    CLK_IN1  : in  std_logic;
    CLK_OUT1 : out std_logic;
    RESET    : in  std_logic;
    LOCKED   : out std_logic
  );
  end component;

begin

u1: clock_manager_core
  port map
    (CLK_IN1 => clk_100MHz,
     CLK_OUT1 => clk_300MHz,
     RESET => reset,
     LOCKED => clock_locked);
end Behavioral;
```

### PS 6-3

Step-3) VHDL implementation is completed. Now simulation can be done. Use PS 6-4 for simulation. Open a VHDL Test Bench file and named it as “Clock\_Manager\_TB”.

```

library ieee;
use ieee.std_logic_1164.all;

entity Clock_Manager_TB is
end Clock_Manager_TB;

architecture behavior of Clock_Manager_TB is

    component Clock_Manager
    port(
        clk_100MHz : in std_logic;
        RESET : in std_logic;
        clock_locked : out std_logic;
        clk_300MHz : out std_logic
    );
    end component;
    --Inputs
    signal clk_100MHz : std_logic := '0';
    signal RESET : std_logic := '0';
    --Outputs
    signal clock_locked : std_logic;
    signal clk_300MHz : std_logic;
    -- Clock period definitions
    constant clk_100MHz_period : time := 10 ns;
begin

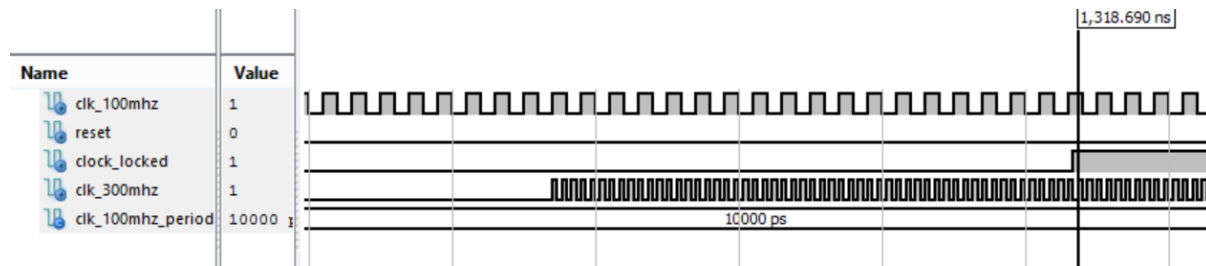
    uut: Clock_Manager port map (
        clk_100MHz => clk_100MHz,
        RESET => RESET,
        clock_locked => clock_locked,
        clk_300MHz => clk_300MHz);
    clk_100MHz_process : process
    begin
        clk_100MHz <= '0';
        wait for clk_100MHz_period/2;
        clk_100MHz <= '1';
        wait for clk_100MHz_period/2;
    end process;
    stim_proc: process
    begin
        reset <= '1';
        wait for 100 ns;
        reset <= '0';
        wait;
    end process;
end;

```

PS 6-4

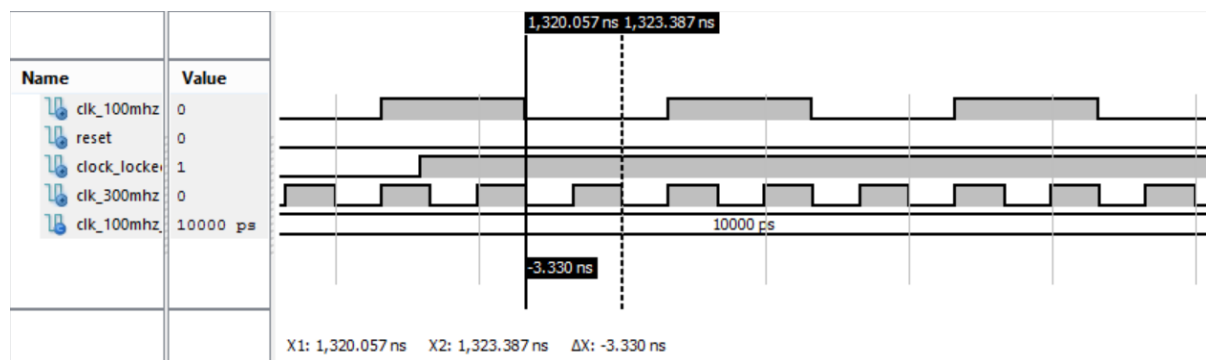


Figure 6-22 shows the result of the simulation. Phases of 100 MHz and 300 MHz clocks become same when *clock\_locked* becomes '1' at approximately at 1,3  $\mu$ s.



**Figure 6-22**

Figure 6-23 demonstrates a zoomed version of the resulting waves. In this figure, period of the output is measured by using cursor feature of the ISim. Period of 300 MHz clock output is approximately 3.3 ns. This result shows that **Clocking Wizard IP** core is used correctly.



**Figure 6-23**

**Example:** Design a ROM that hold the values from 0 to 15, in 4-bit format. Read and show the values of ROM via simulation on ISim.

**Solution:** Step-1) Open a new project and create a main VHDL design, name it as “ROM\_Usage”. Add a new source as represented in Figure 6-2 and name it as “ROM\_core”. After typing core name click **Next**. Choose **Distributed Memory Generator** IP core as seen in Figure 6-24. Click **Next**.

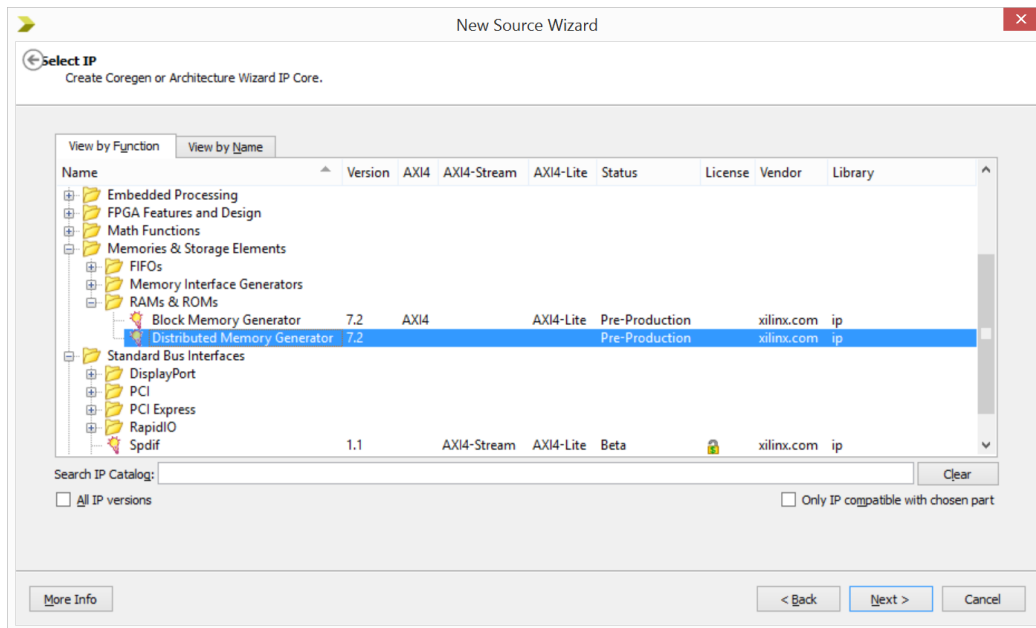


Figure 6-24

**Distributed Memory Generator** IP core settings consists of three pages. In the first page, *Depth* and *Data Width* should be set as 16 and 4, respectively. As stated in the question, memory type should be set as ROM.

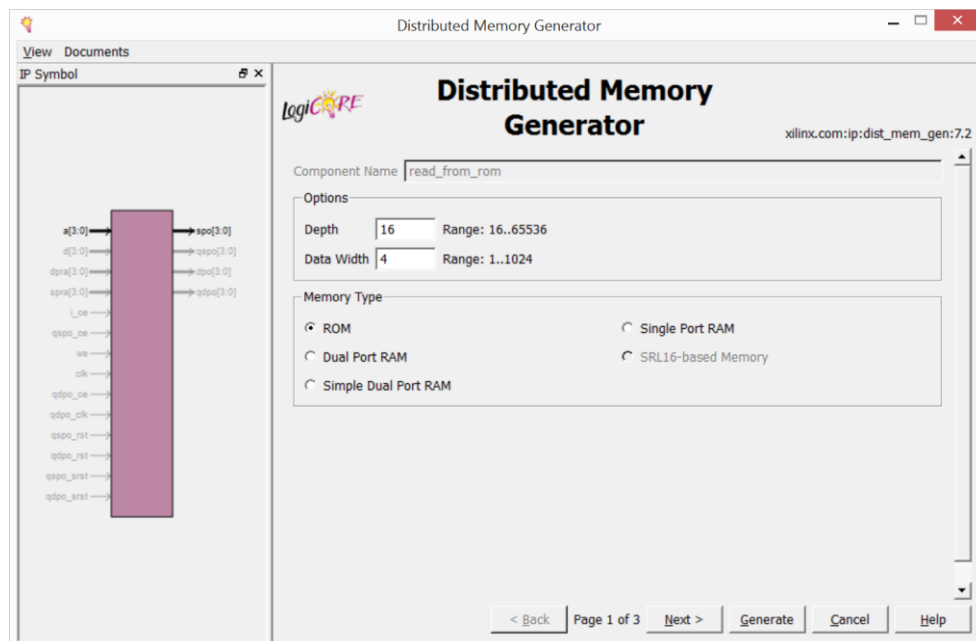
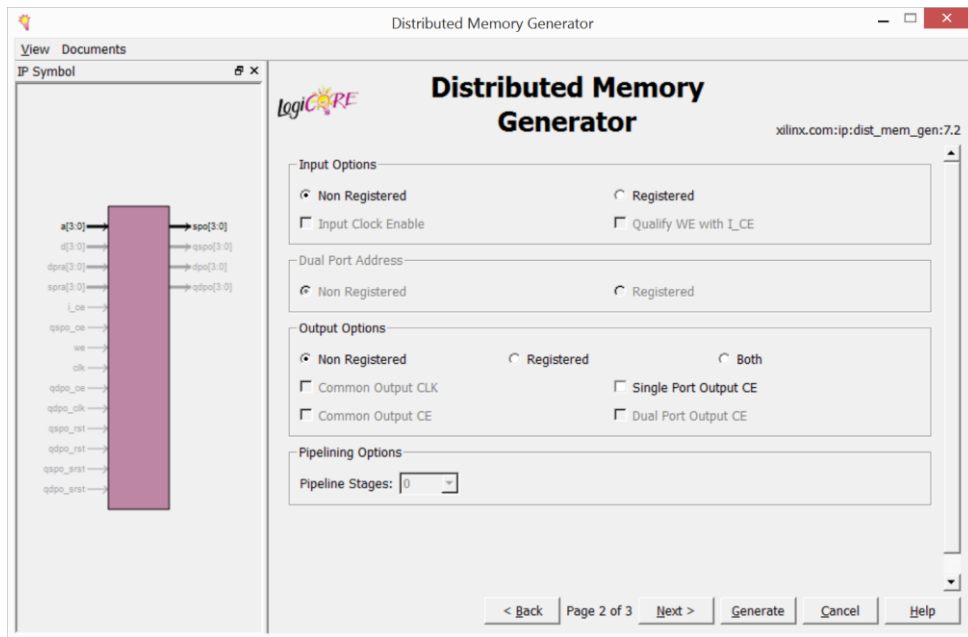


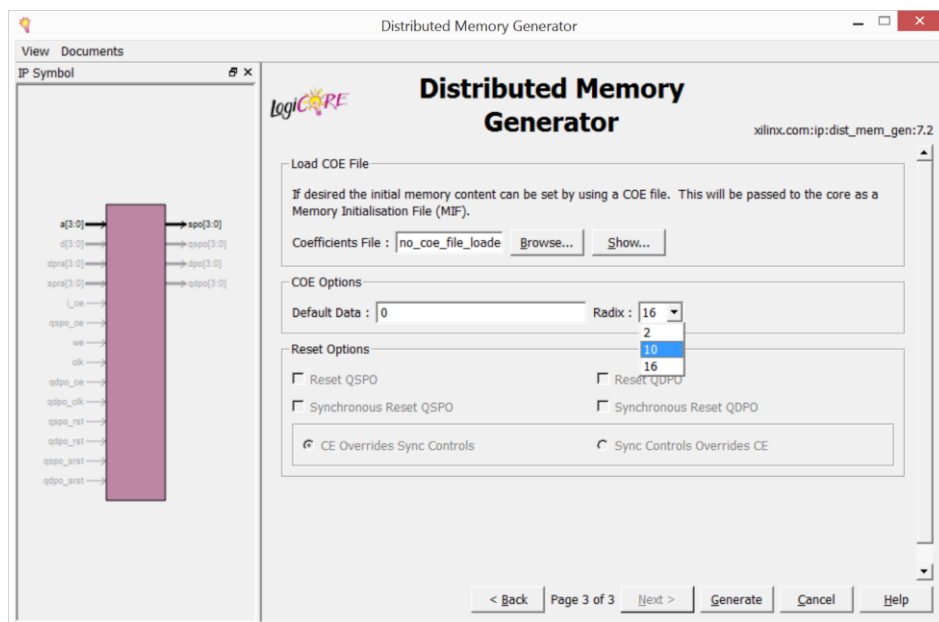
Figure 6-25

In the second page change nothing and click **Next**.



**Figure 6-26**

Figure 6-27 shows the last page of the GUI. In this page, values of the ROM should be stated. ROM values can be inserted by a Coefficients File.



**Figure 6-27**

Coefficients file construction can be done in MATLAB. Below MATLAB script create values from 0 to 15 in base 10.

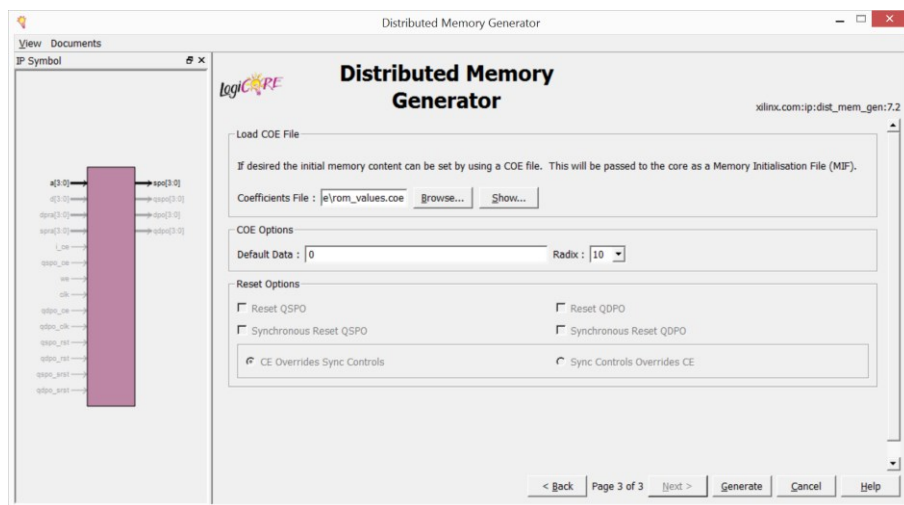
```

clc;clear all;close all;
fid = fopen('rom_values.coe','wt');
fprintf(fid, sprintf('memory_initialization_radix=10;\n\n'));
fprintf(fid, sprintf('memory_initialization_vector=\n\n'));
for i = 1:16
    fprintf(fid, sprintf('%d,\n',i-1));
end
fclose(fid);

```

## PS 6-5

Click **Browse** button to upload generated \*.coe file into ROM.



**Figure 6-28**

“rom\_values.coe” file is loaded into ROM. Uploaded values can be observed by clicking **Show** button. Figure 6-29 shows the values of each index.

Index	Value
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10
11	11
12	12
13	13
14	14
15	15

**Figure 6-29**

Step-2) Arrange the main code as given in PS 6-6.

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

entity ROM_Usage is
  Port ( clk : in std_logic;
        data : out std_logic_vector (3 downto 0));
end ROM_Usage;

architecture Behavioral of ROM_Usage is
  component ROM_core
    PORT (
      a : in std_logic_vector(3 downto 0);
      spo : out std_logic_vector(3 downto 0)
    );
  end component;

  signal address : std_logic_vector(3 downto 0):="0000";
begin

  u1 : ROM_core
  port map (
    a => address,
    spo => data);

  process(clk)
  begin
    if (rising_edge(clk)) then
      address <= address +1;
    end if;
  end process;
end Behavioral;
```

#### PS 6-6

Step-3) VHDL implementation is completed. Now, simulation can be done. Use PS 6-7 for simulation. Open a VHDL Test Bench file and named it as “ROM\_Usage\_TB”.

```

library ieee;
use ieee.std_logic_1164.all;

entity ROM_Usage_TB is
end ROM_Usage_TB;

architecture behavior of ROM_Usage_TB is

    component ROM_Usage
    port (
        clk : in std_logic;
        data : out std_logic_vector(3 downto 0)
    );
    end component;

    signal clk : std_logic := '0';
    signal data : std_logic_vector(3 downto 0);
    constant clk_period : time := 10 ns;

begin
    uut: ROM_Usage port map (
        clk => clk,
        data => data
    );
    clk_process : process
    begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
    end process;
end;

```

PS 6-7

Simulation scenario is constructed such that addresses of the ROM are read at each clock cycle. Figure 6-30 shows the result of reading operation on the output port of *data*.



Figure 6-30