National University of Singapore

Department of Electrical & Computer Engineering

CG3207/EE3207E
Computer Architecture

Getting Started – Vivado® Design Suite, Nexys™4 FPGA Development Board & VHDL/Verilog HDL

Shahzor Ahmad

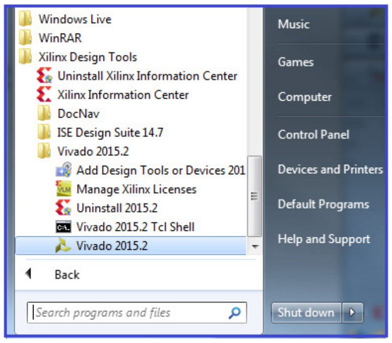
August 2016

Abstract

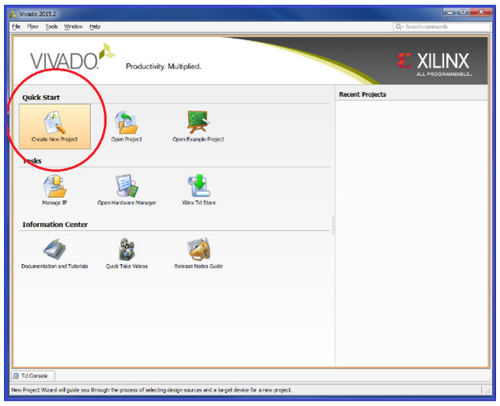
This manual provides an illustration of the tools you will use in CG3207/EE3207E through examples of a simple combinational and a simple sequential circuit. It will familiarize you with industry leader Xilinx®'s Vivado® Design Suite[1] – a comprehensive integrated development envrionment for FPGA design flow, Digilent Inc.'s Nexys™ 4 Development Board featuring an FPGA from Xilinx®'s state-of-the-art Artix®-7 family, and VHDL/Verilog HDL.

[1] This manual assumes Xilinx Vivado WebPACK 2015.2 installed on a Windows 7 PC, but the process should be similar on Windows 8, or later Vivado versions.

1. Creating a New Vivado Project

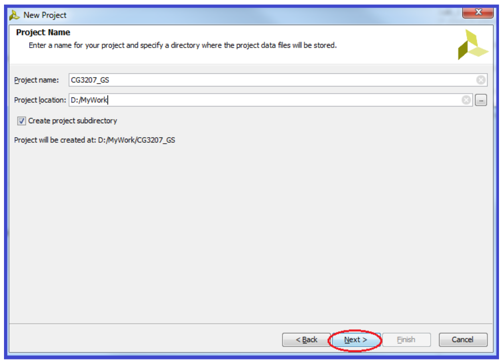


1.1. You can run Xilinx Vivado 2015.2 either by double clicking the icon on Desktop, or going to **Start → All Programs → Xilinx Design Tools → Vivado 2015.2 → Vivado 2015.2**.



New Project Wizard will guide you through the process of selecting design sources and a target device for a new project.

1.2. Having launched Vivado, click on **Create New Project**. The **Create a New Vivado Project** wizard will pop up. Click **Next**.



Enter a name for your project and specify a directory where the project data files will be stored.

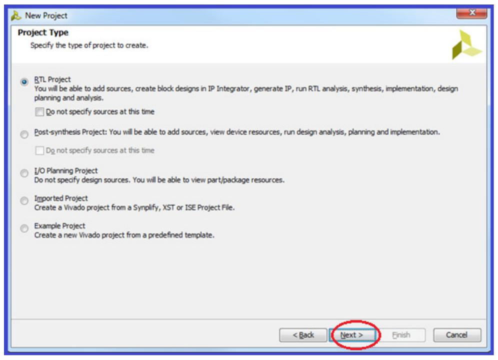
Project name: CG3207_GS

Project location: D:/MyWork

[✓] Create project subdirectory

Project will be created at: D:/MyWork/CG3207_GS

1.3. Enter a **Project name**, e.g., "CG3207_GS". Specify a **Project location**, e.g., "D:/MyWork". Click **Next**.



Specify the type of project to create.

● RTL Project
You will be able to add sources, create block designs in IP Integrator, generate IP, run RTL analysis, synthesis, implementation, design planning and analysis.
☐ Do not specify sources at this time

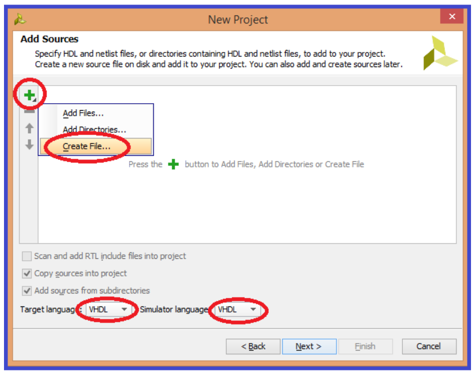
○ Post-synthesis Project: You will be able to add sources, view device resources, run design analysis, planning and implementation.
☐ Do not specify sources at this time

○ I/O Planning Project
Do not specify design sources. You will be able to view part/package resources.

○ Imported Project
Create a Vivado project from a Synplify, XST or ISE Project File.

○ Example Project
Create a new Vivado project from a predefined template.

1.4. Ensure **Project Type** is set to **RTL Project**. Click **Next**.



Add Sources

Specify HDL and netlist files, or directories containing HDL and netlist files, to add to your project.
Create a new source file on disk and add it to your project. You can also add and create sources later.

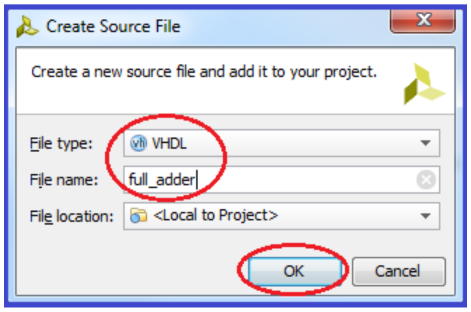
Press the + button to Add Files, Add Directories or Create File

☐ Scan and add RTL include files into project
☑ Copy sources into project
☑ Add sources from subdirectories

Target language: VHDL   Simulator language: VHDL

1.5. In the **Add Sources** window, ensure **Target** and **Simulator language** are set to **VHDL** or **Verilog** (depending on your preference).

Click on the green plus button, and then on **Create File…** from the resulting pop-up menu to create a new VHDL source file.



Create a new source file and add it to your project.

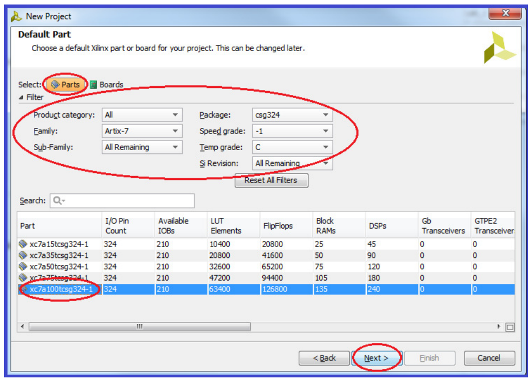
File type: VHDL
File name: full_adder
File location: <Local to Project>

1.6. A **Create Source File** window pops up. Ensure File type is set to your preferred HDL (either **VHDL** or **Verilog**). Enter "full_adder" as **File name**. Click **OK**, which adds a new file to our **Add Sources** window.

Click **Next**.

On the **Add Existing IP (optional)** window, click **Next**.

On the **Add Constraints (optional)** window, click **Next**. We shall add constraints later on.



Default Part

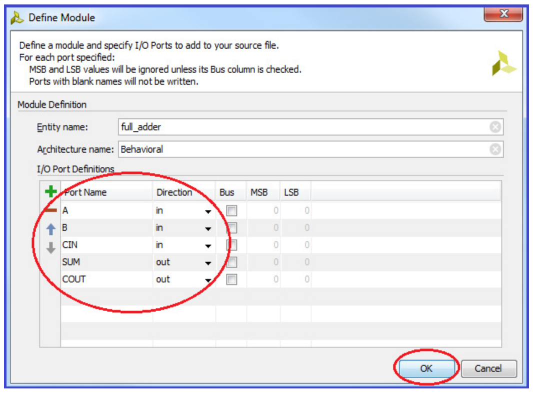
Choose a default Xilinx part or board for your project. This can be changed later.

Product category: All
Family: Artix-7
Sub-Family: All Remaining
Package: csg324
Speed grade: -1
Temp grade: C
Si Revision: All Remaining

Part	I/O Pin Count	Available IOBs	LUT Elements	FlipFlops	Block RAMs	DSPs	Gb Transceivers	GTPE2 Transceiver
xc7a15tcsg324-1	324	210	10400	20800	25	45	0	0
xc7a35tcsg324-1	324	210	20800	41600	50	90	0	0
xc7a50tcsg324-1	324	210	32600	65200	75	120	0	0
xc7a75tcsg324-1	324	210	47200	94400	105	180	0	0
xc7a100tcsg324-1	324	210	63400	126800	135	240	0	0

1.7. In the Default Part window, we must specify the FPGA chip in use. The Nexys-4 board contains the Xc7a100T chip from the Artix-7 family. We filter by **Parts**. Choose **Artix-7** for **Family**, **csg324** for the **Package**, **-1** for **Speed grade** and **C** for **Temp grade**[2]. This filtering considerably narrows down the options, from which **xc7a100tcsg324-1** should be selected.

Click **Next**. A summary is shown. Click **Finish**.



Define a module and specify I/O Ports to add to your source file.
For each port specified:
MSB and LSB values will be ignored unless its Bus column is checked.
Ports with blank names will not be written.

Module Definition

Entity name: full_adder
Architecture name: Behavioral

I/O Port Definitions

Port Name	Direction	Bus	MSB	LSB
A	in	☐	0	0
B	in	☐	0	0
CIN	in	☐	0	0
SUM	out	☐	0	0
COUT	out	☐	0	0

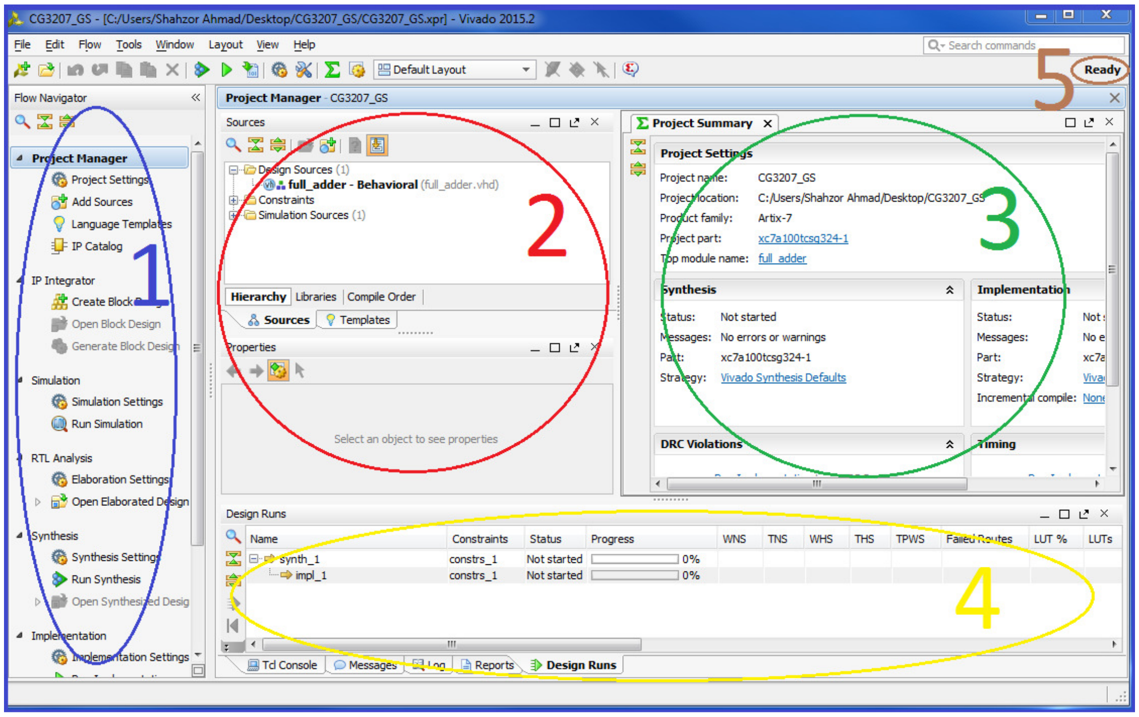
1.8. The wizard then asks us to define a **VHDL Entity** / **Verilog Module** to reside in the source file we just created. Leave **Entity / Module name** same as the source file (full_adder), and the Architecture name as **Behavioral**. Provide the **input** ports **A**, **B** and **CIN**[3] as shown, and the **output** ports **SUM** and **COUT**.

Click **Ok**.

---

[2] These part details for the Artix-7 chip can be seen from the Nexys 4 schematics/reference manual available from Digilent: (www.digilentinc.com/Products/Detail.cfm?Prod=NEXYS4)

[3] Recall the full adder as opposed to a half adder takes a carry-in as an input too.



Project name: CG3207_GS
Project location: C:/Users/Shahzor Ahmad/Desktop/CG3207_GS
Product family: Artix-7
Project part: xc7a100tcsg324-1
Top module name: full_adder

Synthesis
Status: Not started
Messages: No errors or warnings
Part: xc7a100tcsg324-1
Strategy: Vivado Synthesis Defaults

Name	Constraints	Status	Progress	WNS	TNS	WHS	THS	TPWS	Failed Routes	LUT %	LUTs
synth_1	constrs_1	Not started	0%								
impl_1	constrs_1	Not started	0%								

1.9. After creating a new project, one is presented with the Vivado IDE viewing environment as shown. The main parts of the GUI (graphical user interface) are highlighted[4].

1 – The Flow Navigator provides access to all the tools for a complete FPGA design flow – HDL text entry, simulation, synthesis, implementation, bit-stream creation and FPGA programming (configuration).

2 – The Data Windows Area displays design sources and data. Currently the **Sources Window** is seen.

3 – The Workspace displays the Text Editor, Schematic Window, Synthesis reports such as Resource Utilization, etc. Currently **Project Summary** is seen.

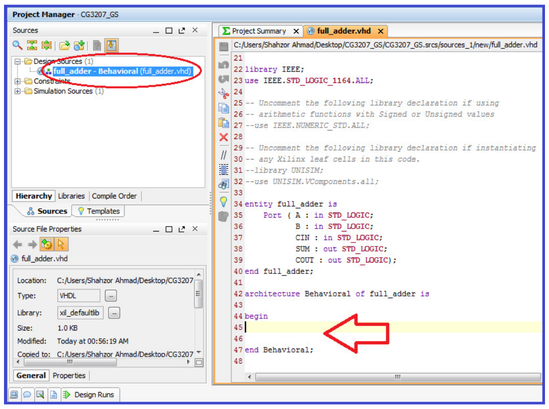
4 – The Results Windows display messages and log files as Vivado performs simulation, synthesis, and implementation, etc.

5 – The Project Status Bar displays current status of the active design.

[4] Xilinx Vivado Design Suite User Guide UG893 – Using the Vivado IDE (http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/ug893-vivado-ide.pdf)

2. Design Entry Using Vivado's Text Editor

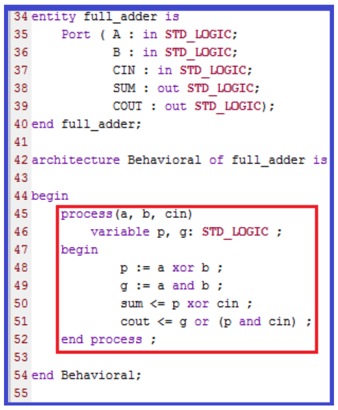
2.1. VHDL



Location: C:/Users/Shahzor Ahmad/Desktop/CG3207
Type: VHDL
Library: xil_defaultlib
Size: 1.0 KB
Modified: Today at 00:56:19 AM
Copied to: C:/Users/Shahzor Ahmad/Desktop/CG3207

C:/Users/Shahzor Ahmad/Desktop/CG3207_GS/CG3207_GS.srcs/sources_1/new/full_adder.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity full_adder is
    Port ( A : in STD_LOGIC;
           B : in STD_LOGIC;
           CIN : in STD_LOGIC;
           SUM : out STD_LOGIC;
           COUT : out STD_LOGIC);
end full_adder;

architecture Behavioral of full_adder is

begin


end Behavioral;

```

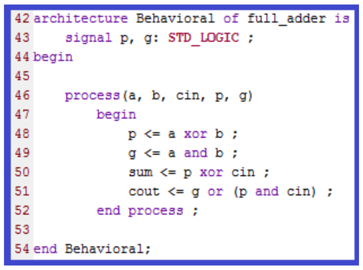


```
entity full_adder is
    Port ( A : in STD_LOGIC;
           B : in STD_LOGIC;
           CIN : in STD_LOGIC;
           SUM : out STD_LOGIC;
           COUT : out STD_LOGIC);
end full_adder;

architecture Behavioral of full_adder is

begin
    process(a, b, cin)
        variable p, g: STD_LOGIC ;
    begin
            p := a xor b ;
            g := a and b ;
            sum <= p xor cin ;
            cout <= g or (p and cin) ;
    end process ;

end Behavioral;

```

2.1.1. Double click on the design source **full_adder.vhd** to open and display the file in the Vivado Editor that will be docked in the Workspace.

A VHDL **entity** definition for a full adder circuit is seen. The **architecture** is currently empty. Enter the **combinational process** between the `begin` and `end` statements of the architecture as shown in the adjacent figure. VHDL is case in-sensitive, unlike Verilog.

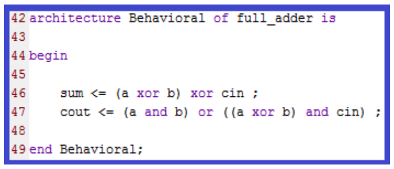
A **process** resumes whenever a signal in the **sensitivity list** changes and then suspends again till the next resume. It is often convenient to break a complex function into intermediate steps. We have thus employed internal variables `p` and `g` (which are invisible to the world outside the process), before computing the outputs `sum` and `cout`. Variable assignments are performed using **blocking assignments** (:=), which are instantaneous. That is, variables `p` and `g` immediately get their new values. Hence the following two signal assignments, `sum` and `cout` (performed using **non-blocking assignment**, <=), use the updated variable values. Unlike variables, signal assignments in a process take effect when the process suspends[5]. Also, variable assignments are evaluated sequentially, while signal assignments run in parallel to each other.



```
architecture Behavioral of full_adder is
    signal p, g: STD_LOGIC ;
begin

    process(a, b, cin, p, g)
        begin
            p <= a xor b ;
            g <= a and b ;
            sum <= p xor cin ;
            cout <= g or (p and cin) ;
        end process ;

end Behavioral;
```

An oft-employed alternate architecture definition for a full adder is shown in adjacent figure. Observe `p` and `g` in this scenario are declared as signals, and their new values will take effect (together with `sum` and `cout`) only when the process suspends. Hence, `sum` and `cout` essentially use the old values of `p` and `g` (from the previous run of the process). The process eventually attains the correct answer, but needs to be evaluated twice compared to the process in the previous figure. Both architectures synthesize to the same hardware, however. Note if one forgets to include `p` and `g` in the sensitivity list, simulation results for this architecture are incorrect (output signals remain undefined in the first cycle)! Vivado still synthesizes to the correct hardware, but that means a mismatch between simulation results and what the hardware actually does. Hence, it is recommended to use variable and not signal assignments for internal nets in combinational processes.

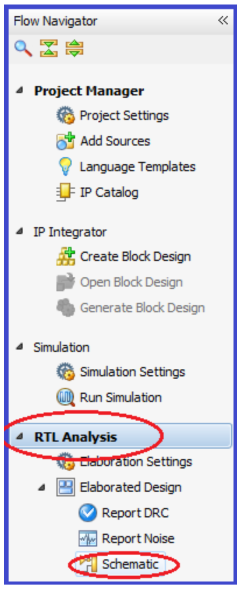
[5] Signal assignment (http://www.vhdl.renerta.com/mobile/source/vhd00063.htm)



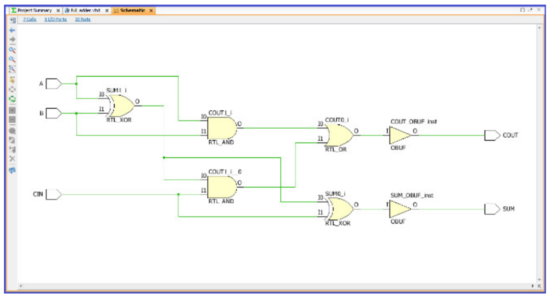
```
architecture Behavioral of full_adder is

begin

    sum <= (a xor b) xor cin ;
    cout <= (a and b) or ((a xor b) and cin) ;

end Behavioral;
```

Since the full adder is really simple combinational logic, we may forego the `process` statement altogether, and use **concurrent signal assignments** as shown in the adjacent figure. A concurrent assignment is activated whenever there is a change in any signal on the R.H.S. It is independent of other statements in a given architecture, and is performed concurrently to other active statements. This architecture, similar to the first one, produces a simulation that is in agreement with the synthesised hardware.

2.1.2. **Save** your HDL entry by clicking **File → Save File**, or pressing Ctrl+S. If any syntax errors appear in your code (automatically checked by Vivado every time you perform a save), they are indicated in the **Messages** console in the **Results Windows** area at the bottom.



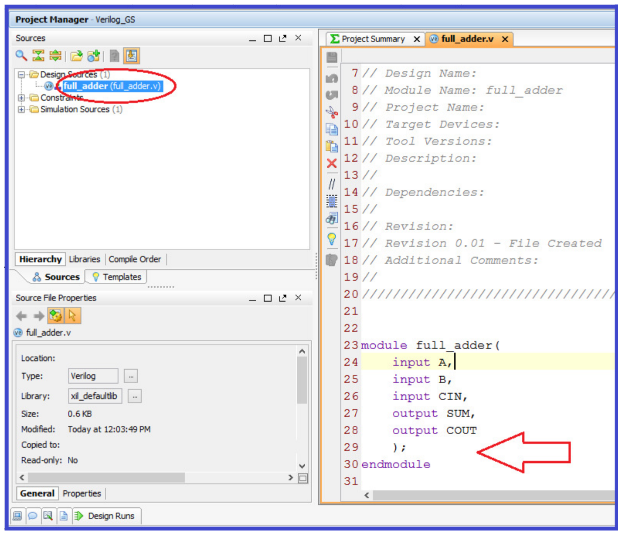
2.1.3. In the **Flow Navigator** pane, expand the **RTL Analysis**[6] tools by clicking on the arrow on the left. Similarly, expand **Open Elaborated Design**. Click on **Schematic**. If a message **Elaborate Design** pops up, click **OK**. Vivado takes a few seconds to perform RTL Analysis and generate the schematic which is a representation of the pre-optimized design in terms of generic components – adders, multipliers, counters, gates, etc. independent of the target device or technology library.



2.1.4. The schematic displayed in the Workspace is that of a full adder circuit. Observe we had conveyed our design to Vivado through text that describes our intended hardware. RTL analysis can help debug and optimize our design. Select a logic instance in the schematic, right-click to select **Go to Source** or **Go to Definition**, to open the source file containing the module definition or instance of selected object respectively

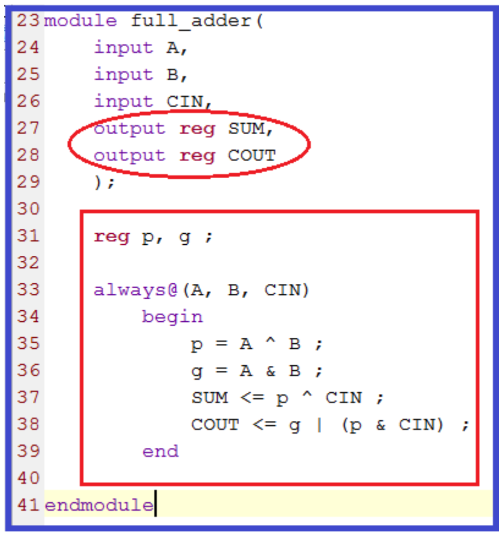
[6] HDL code for any reasonably complex design involving sequential circuits is written at a level of abstraction called Register Transfer Level (RTL) rather than actual gate-level representation.

2.2. Verilog



Location:
Type: Verilog
Library: xil_defaultlib
Size: 0.6 KB
Modified: Today at 12:03:49 PM
Copied to:
Read-only: No

```
// Design Name:
// Module Name: full_adder
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
/////////////////////////////////


module full_adder(
    input A,
    input B,
    input CIN,
    output SUM,
    output COUT
    );
endmodule
```

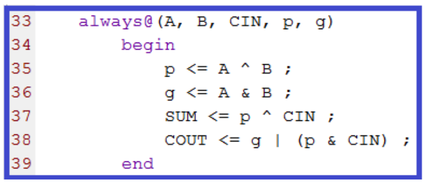


```
module full_adder(
    input A,
    input B,
    input CIN,
    output reg SUM,
    output reg COUT
    );

    reg p, g ;

    always@(A, B, CIN)
        begin
            p = A ^ B ;
            g = A & B ;
            SUM <= p ^ CIN ;
            COUT <= g | (p & CIN) ;
        end

endmodule
```

2.2.1. Double click on the design source **full_adder.v** to open and display the file in the Vivado Editor that will be docked in the Workspace. A Verilog module definition for a full adder circuit is seen. The body is currently empty. Modify the module output declarations and populate the body with combinational logic as seen in the adjacent figure. Note that Verilog is case-sensitive, unlike VHDL.

Verilog **`always`** blocks are evaluated every time the signals in the sensitivity list (or header) change, i.e., `A`, `B`, `CIN` in our example. In Verilog, all signals on the LHS of assignments in `always` blocks must be declared as **`reg`**[7]. This does not necessarily mean the signal is actually a register (registers are implied in hardware by assignments inside sequential `always` blocks). Since the outputs `SUM` and `COUT` are modified within our always block, they must be declared as `reg`.

It is often convenient to break a complex function into intermediate steps. We have thus employed internal signals `p` and `g` (which are invisible to the world outside the module), before computing the outputs `SUM` and `COUT`. Furthermore, we use **blocking assignments** (=) to instantaneously but sequentially update `p` and `g`. Hence the following two signal assignments, `SUM` and `COUT` use the updated values of `p` and `g`. Due to the use of **non-blocking assignment** (<=) the new values of `SUM` and `COUT` take effect concurrently when the `always` block suspends.

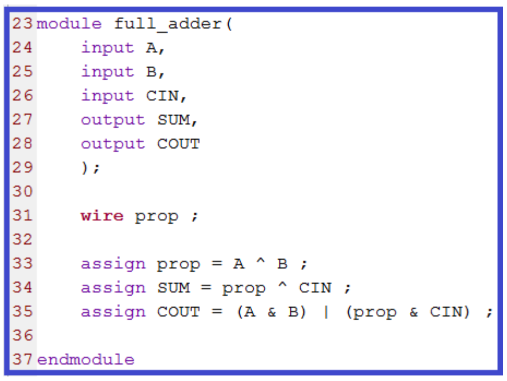


```
    always@(A, B, CIN, p, g)
        begin
            p <= A ^ B ;
            g <= A & B ;
            SUM <= p ^ CIN ;
            COUT <= g | (p & CIN) ;
        end
```

An alternate `always` block for a full adder design is shown in the adjacent figure. Observe the use of non-blocking assignments for `p` and `g` in this scenario. Their new values will take effect (together with `SUM` and `COUT`) concurrently when the `always` block suspends. Hence, `SUM` and `COUT` essentially use the old values of `p` and `g` (from the previous run of the process). This code eventually attains the correct answer, but needs to be evaluated twice compared to the block in the previous figure. Both modules synthesize to the same hardware, however. Note if one forgets to include `p` and `g` in the sensitivity list, simulation results for this module are incorrect (output signals remain undefined in the first cycle)! Vivado still synthesizes to the correct hardware, but that means a mismatch between simulation results and what the hardware actually does.

[7] "Structural Design with Verilog". David Harris. Available at (http://centaur.sch.bme.hu/~b4k1/labor1/honlaprol/verilog.pdf).

Since the full adder is really simple combinational logic, we may forego the `always` statement altogether, and use **continuous assignments** as shown in the adjacent figure. An **`assign`** statement updates the LHS any time the RHS changes, independent of other such continuous statements or `always` blocks in the module. Continuous assignments cannot appear inside an `always` block. Notice the use of `wire` to declare an internal signal `prop`. Technically, it is not necessary but good practice to declare single-bit wires. It is necessary to declare multi-bit busses. This module, similar to the first one, produces a simulation that is in agreement with the synthesised hardware.



```
module full_adder(
    input A,
    input B,
    input CIN,
    output SUM,
    output COUT
    );

    wire prop ;

    assign prop = A ^ B ;
    assign SUM = prop ^ CIN ;
    assign COUT = (A & B) | (prop & CIN) ;

endmodule
```

2.2.2. Now follow steps 2.1.2 – 2.1.4 to save your design, check for any syntax errors and perform RTL analysis.

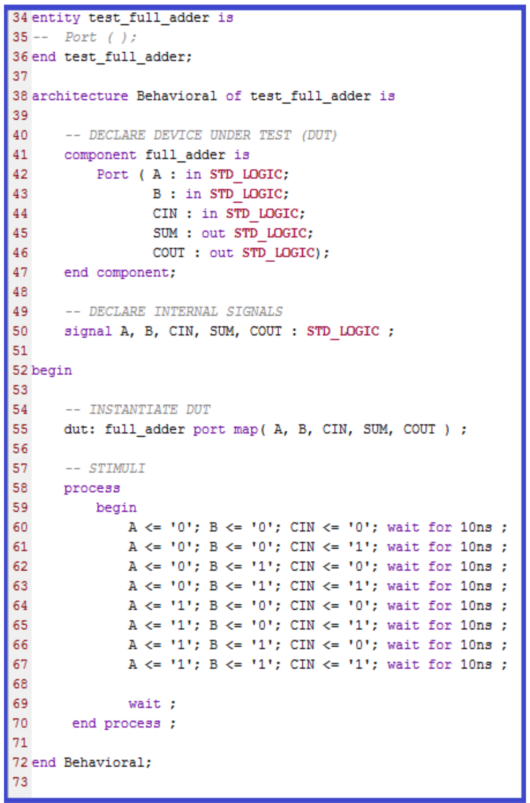
3. Testbench & Behavioral Simulation

Simulation is a major component in digital design flow. Stimuli are applied to the circuit, and the outputs are observed to verify the design behaves as intended. This ensures a system is tested before it is built.

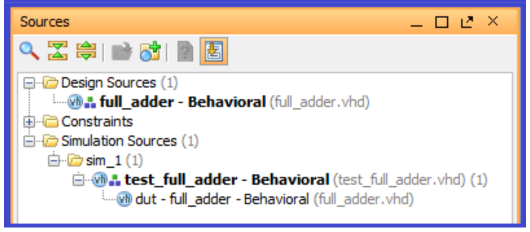
A *testbench* is HDL code to test your design which is called *device under test* (DUT) or *unit under test* (UUT). The testbench contains statements to apply inputs to the DUT, which in our case is our full adder design.

3.1. Expand the **Project Manager** in the Flow Navigator. Click **Add Sources**. In the Add Sources window that pops up, select the **Add or create simulation sources** radio button, and click **Next**. Similar to **Step 1.6** in Section 1, create a VHDL / Verilog file called "test_full_adder" by specifying the name and clicking **OK**. Click **Finish**. Unlike **Step 1.8**, however, do not specify any I/O ports for the module (a testbench is not intended to be synthesized into hardware and does not have I/O ports). Click **OK** on the **Define Module** window. Click **Yes** on the **Define Module** message that pops up.

3.1. VHDL



```
entity test_full_adder is
--  Port ( );
end test_full_adder;

architecture Behavioral of test_full_adder is

    -- DECLARE DEVICE UNDER TEST (DUT)
    component full_adder is
        Port ( A : in STD_LOGIC;
               B : in STD_LOGIC;
               CIN : in STD_LOGIC;
               SUM : out STD_LOGIC;
               COUT : out STD_LOGIC);
    end component;

    -- DECLARE INTERNAL SIGNALS
    signal A, B, CIN, SUM, COUT : STD_LOGIC ;

begin

    -- INSTANTIATE DUT
    dut: full_adder port map( A, B, CIN, SUM, COUT ) ;

    -- STIMULI
    process
        begin
            A <= '0'; B <= '0'; CIN <= '0'; wait for 10ns ;
            A <= '0'; B <= '0'; CIN <= '1'; wait for 10ns ;
            A <= '0'; B <= '1'; CIN <= '0'; wait for 10ns ;
            A <= '0'; B <= '1'; CIN <= '1'; wait for 10ns ;
            A <= '1'; B <= '0'; CIN <= '0'; wait for 10ns ;
            A <= '1'; B <= '0'; CIN <= '1'; wait for 10ns ;
            A <= '1'; B <= '1'; CIN <= '0'; wait for 10ns ;
            A <= '1'; B <= '1'; CIN <= '1'; wait for 10ns ;

            wait ;
     end process ;

end Behavioral;
```



```
Design Sources (1)
    full_adder - Behavioral (full_adder.vhd)
Constraints
Simulation Sources (1)
    sim_1 (1)
        test_full_adder - Behavioral (test_full_adder.vhd) (1)
            dut - full_adder - Behavioral (full_adder.vhd)
```

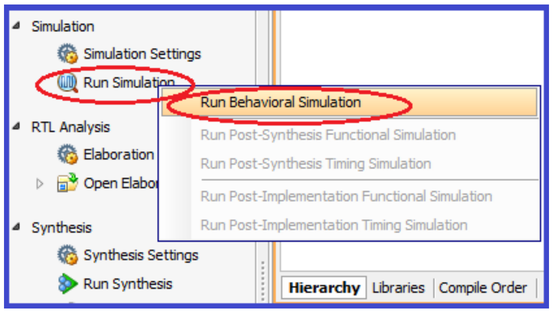
3.1.1. Similar to Step 2.1.1, open up the simulation source **test_full_adder.vhd** in the Vivado Editor. This is our **testbench**. The entity declaration is already present, with no port definitions. Code the architecture as shown in the adjacent figure.

The declarative part of our testbench's architecture contains a component declaration for the DUT defined earlier (the entity full_adder), as well as internal nets (used to handle I/O for the DUT).

The first statement after `begin` instantiates the DUT. Observe from **Simulation Sources** in the **Data Windows** area that this causes full_adder to become a leaf node in the hierarchy for test_full_adder.

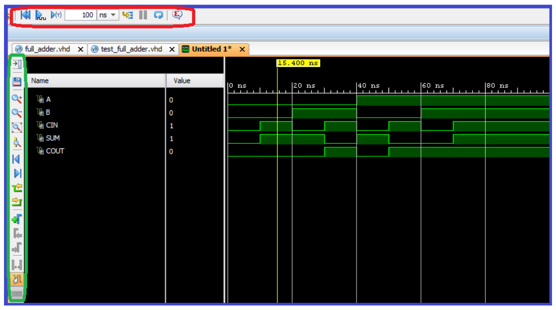
Then follows a process which provides input patterns to our DUT. The `wait` statements suspend the process for the specified interval between each stimulus[8]. The last wait suspends the process indefinitely, else the process would keep repeating the pattern.

[8] Wait statement (http://www.vhdl.renerta.com/mobile/source/vhd00081.htm)


Simulation
Simulation Settings
Run Simulation
Run Behavioral Simulation
Run Post-Synthesis Functional Simulation
Run Post-Synthesis Timing Simulation
Run Post-Implementation Functional Simulation
Run Post-Implementation Timing Simulation
RTL Analysis
Elaboration
Open Elabo
Synthesis
Synthesis Settings
Run Synthesis
Hierarchy
Libraries
Compile Order

3.1.2. Expand the **Simulation** tools in the **Flow Navigator**, click **Run Simulation → Run Behavioral Simulation**.

The Vivado simulator pops open a few seconds later, having run our testbench.



3.1.3. Play around with the tools in the red and green boxes, use the mouse wheel to zoom in/out, and move the yellow bar around (observing the Value column as you do).

Initially, you may not see the waveform as in the figure as the time resolution is very small, and the simulation by default has run for a larger period of time.

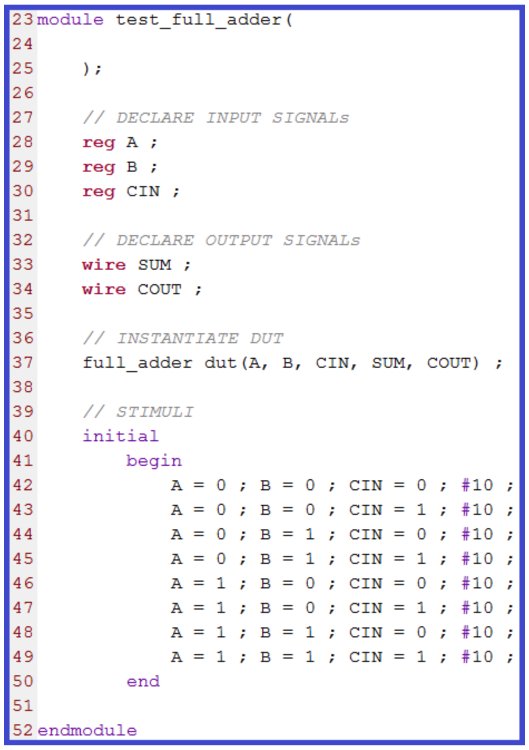
Select **View → Zoom Fit** or use the **Zoom Fit** icon in the green box.

Verify from this simulation that our full adder design behaves as intended.

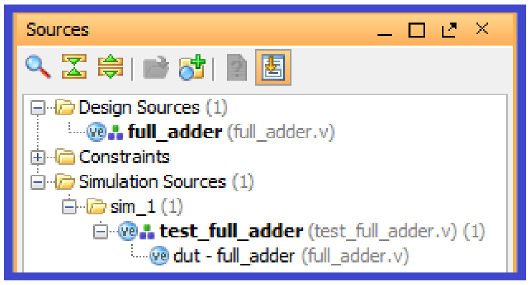
3.1.4. By default, the Vivado simulator adds signals from the testbench to the **Waveform** window. In more complex designs (such as CG3207/EE3207E Labs), it would be desirable to **add and monitor signals** from the instantiated DUT or even sub-entities/modules at various levels of the design hierarchy. To add and monitor signals from a desired entity/module, it may be selected from the **Scopes** window. The signals to be monitored may be selected from the **Objects** window, dragged and dropped into the **Waveform** window[9].

[9] For more information on adding and monitoring signals from instantiated DUT and sub-modules, and working with the Waveform window, see UG937 (http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/ug937-vivado-design-suite-simulation-tutorial.pdf).

3.2. Verilog



```
module test_full_adder(

    );

    // DECLARE INPUT SIGNALs
    reg A ;
    reg B ;
    reg CIN ;

    // DECLARE OUTPUT SIGNALs
    wire SUM ;
    wire COUT ;

    // INSTANTIATE DUT
    full_adder dut(A, B, CIN, SUM, COUT) ;

    // STIMULI
    initial
        begin
            A = 0 ; B = 0 ; CIN = 0 ; #10 ;
            A = 0 ; B = 0 ; CIN = 1 ; #10 ;
            A = 0 ; B = 1 ; CIN = 0 ; #10 ;
            A = 0 ; B = 1 ; CIN = 1 ; #10 ;
            A = 1 ; B = 0 ; CIN = 0 ; #10 ;
            A = 1 ; B = 0 ; CIN = 1 ; #10 ;
            A = 1 ; B = 1 ; CIN = 0 ; #10 ;
            A = 1 ; B = 1 ; CIN = 1 ; #10 ;
        end

endmodule
```



```
Sources
Design Sources (1)
    full_adder (full_adder.v)
Constraints
Simulation Sources (1)
    sim_1 (1)
        test_full_adder (test_full_adder.v) (1)
            dut - full_adder (full_adder.v)
```

3.2.1. Similar to Step 2.2.1, open up the simulation source **test_full_adder.v** in the Vivado Editor. This is our **testbench**. The module declaration is already present, with no port definitions. Code the body as shown in the adjacent figure.

The declarative part of our testbench module contains an instantiation of the DUT defined earlier (the module full_adder), as well as internal nets (used to handle I/O for the DUT).

Upon instantiating the DUT, observe from **Simulation Sources** in the **Data Windows** area that this causes full_adder to become a leaf node in the hierarchy for test_full_adder.

Then follows an `initial` block which provides input patterns to our DUT. Similar to the case of an `always` block, signals appearing on the LHS of assignments in an `initial` block must be declared as `reg`. Note `initial` statements should only be used in testbenches for simulation, not in modules intended to be synthesized into actual hardware.

The timescale directive at the top of our test_full_adder.v file:

```
        `timescale 1ns / 1ps
```

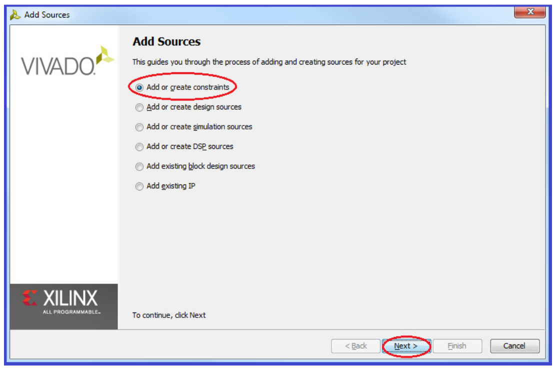
Specifies a time unit in this file to be 1ns, and the simulation to have a precision of 1ps. The `#` statements suspend the `initial` block for the specified interval between each stimulus.

3.2.2. Now follow steps 3.1.2 – 3.1.4 to simulate your design and analyse the resulting waveforms.

4. Design Constraints

Design constraints such as timing and physical I/O pin mapping must be defined before synthesizing a design for a target FPGA. We shall not specify any timing constraints for now as our circuit is combinational; no sequential elements exist in our design (though hardly the case in practice).

4.1. Constraints are specified via a .xdc file (Xilinx Design Constraints). Rather than writing constraints from scratch, let us use Nexys 4 Master XDC provided by Digilent here [https://reference.digilentinc.com/_media/reference/programmable-logic/nexys-4/nexys4_master_xdc.zip]. Download and unzip the file in some directory.



Add Sources

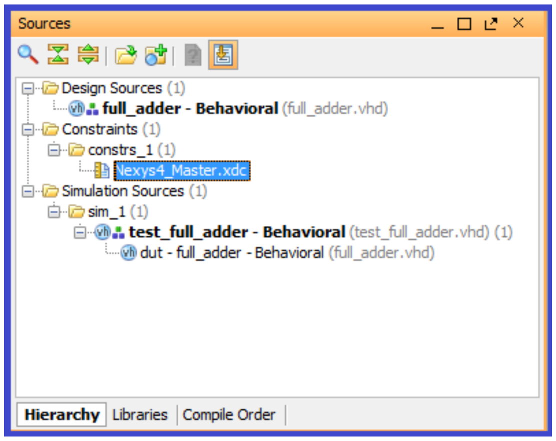
This guides you through the process of adding and creating sources for your project

- [✓ Add or create constraints]
- Add or create design sources
- Add or create simulation sources
- Add or create DSP sources
- Add existing block design sources
- Add existing IP

To continue, click Next

4.2. Click on **Project Manager** in the **Flow Navigator**. Click on **Add Sources**. Select the **Add or create constraints** radio button. Click **Next**.

On the **Add or Create Constraints** window, use the green plus button to **Add Files…**, and add the **Nexys4_Master.xdc** you just downloaded from the location you unzipped it into. Let the option **Copy constraints files into project** remain checked, so that a fresh copy is placed in your Vivado project folder.

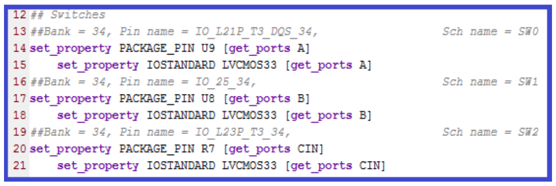


Sources

- Design Sources (1)
  - full_adder - Behavioral (full_adder.vhd)
- Constraints (1)
  - constrs_1 (1)
    - Nexys4_Master.xdc
- Simulation Sources (1)
  - sim_1 (1)
    - test_full_adder - Behavioral (test_full_adder.vhd) (1)
      - dut - full_adder - Behavioral (full_adder.vhd)

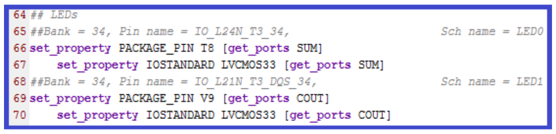
4.3. The file is added to the **Constrains** folder in the **Sources** window. Open it up for editing in the Workspace.

Currently all code is commented out in the file. Depending on our design, we simply need to un-comment the lines corresponding to the pins we use and rename the ports if necessary.



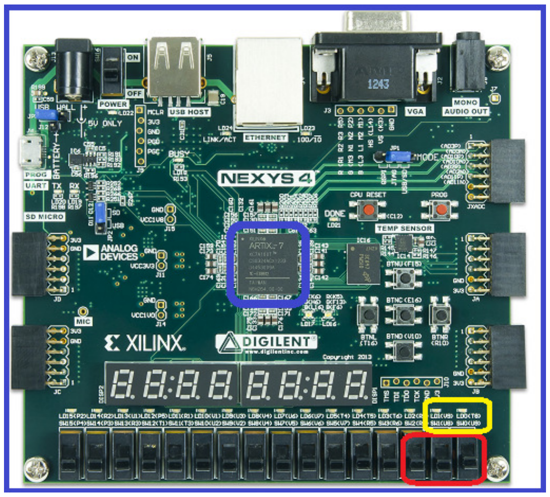
```
## Switches
##Bank = 34, Pin name = IO_L21P_T3_DQS_34,                   Sch name = SW0
set_property PACKAGE_PIN U9 [get_ports A]
    set_property IOSTANDARD LVCMOS33 [get_ports A]
##Bank = 34, Pin name = IO_25_34,                            Sch name = SW1
set_property PACKAGE_PIN U8 [get_ports B]
    set_property IOSTANDARD LVCMOS33 [get_ports B]
##Bank = 34, Pin name = IO_L23P_T3_34,                       Sch name = SW2
set_property PACKAGE_PIN R7 [get_ports CIN]
    set_property IOSTANDARD LVCMOS33 [get_ports CIN]
```

4.4. Un-comment the lines as in the adjacent figure. Rename the ports to `A`, `B` and `CIN` as shown. This essentially maps inputs `A`, `B` and `CIN` of our `full_adder` entity to SW0, SW1 and SW3 respectively on the Nexys 4 board.



```
## LEDs
##Bank = 34, Pin name = IO_L24N_T3_34,                       Sch name = LED0
set_property PACKAGE_PIN T8 [get_ports SUM]
    set_property IOSTANDARD LVCMOS33 [get_ports SUM]
##Bank = 34, Pin name = IO_L21N_T3_DQS_34,                   Sch name = LED1
set_property PACKAGE_PIN V9 [get_ports COUT]
    set_property IOSTANDARD LVCMOS33 [get_ports COUT]
```

4.5. Similarly, map the outputs `SUM` and `COUT` to LED0 and LED1 respectively as shown.

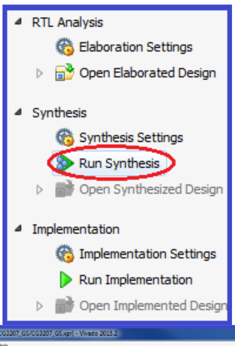


4.6. The said switches and LEDs are indicated by red and yellow boxes in the adjacent figure, respectively. The Artix-7 FPGA is also indicated in blue.

WARNING: the chips on the board are electro-static sensitive (and costly!). Avoid touching. Handle the board by the edges.

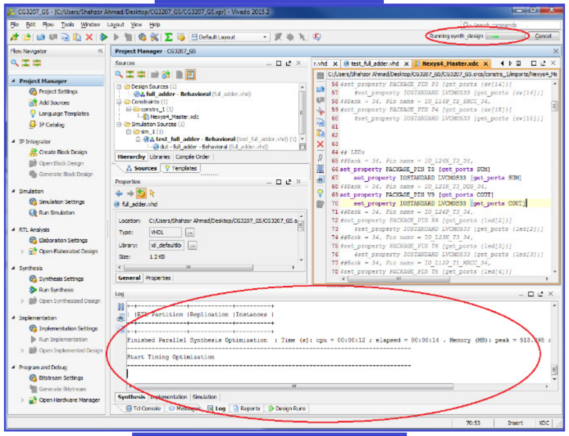
5. Synthesis

Logic synthesis transforms RTL HDL code into an optimized set of logic gates to reduce the amount of hardware and efficiently perform the intended function.



5.1. Expand **Synthesis** tools in the **Flow Navigator**, and click **Run Synthesis**.

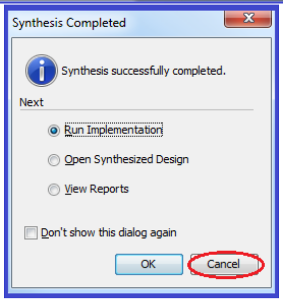
Synthesis is usually the most time-consuming part of the FPGA design flow. In our simple example, however, it should only take a few moments.



```
| |RTL Partition |Replication |Instances |
Finished Parallel Synthesis Optimization  : Time (s): cpu = 00:00:12 ; elapsed = 00:00:14 . Memory (MB): peak = 513.395 ;
---------------------------------------------------------------------------------
Start Timing Optimization
---------------------------------------------------------------------------------
```

5.2. While Vivado performs synthesis, the **Project Status Bar** at the top right of the IDE provides visual indication.

A log appears in the **Log** tab in the **Results Windows**, while any warnings or errors are displayed in the **Messages** tab.



Synthesis Completed

Synthesis successfully completed.

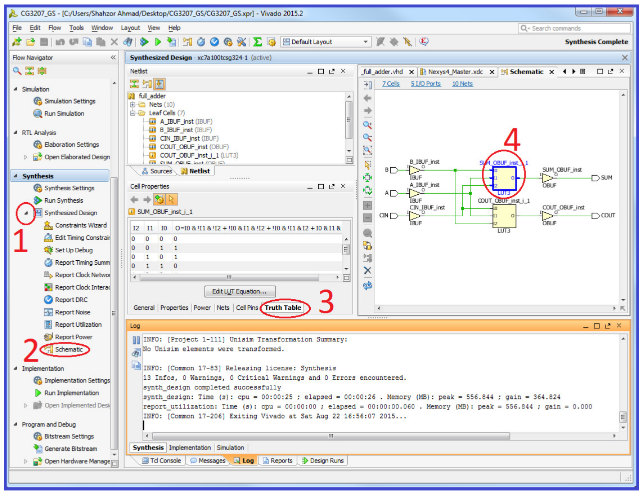
Next

[✓ Run Implementation]
Open Synthesized Design
View Reports

Don't show this dialog again

OK   Cancel

5.3. Upon completion, a **Synthesis Completed** messages pops up, asking to proceed to perform implementation. Click **Cancel**, as we'd like to first analyse the synthesized output.

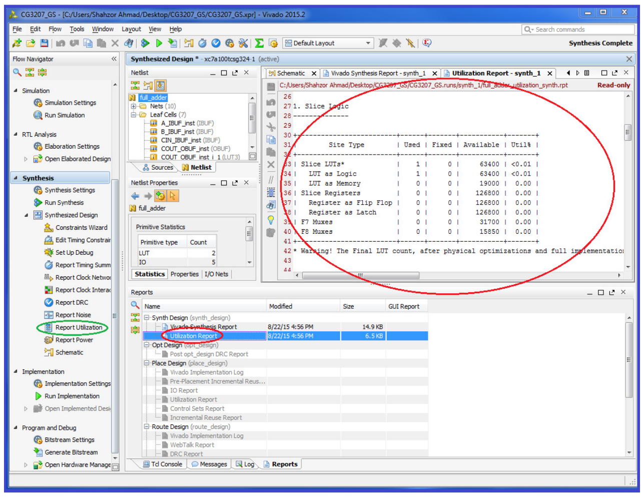


I2	I1	I0	O=I0 & !I1 & !I2 + !I0 & I1 & !I2 + !I0 & !I1 & I2 + I0 & I1 &
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0

```
INFO: [Project 1-111] Unisim Transformation Summary:
No Unisim elements were transformed.

INFO: [Common 17-83] Releasing license: Synthesis
13 Infos, 0 Warnings, 0 Critical Warnings and 0 Errors encountered.
synth_design completed successfully
synth_design: Time (s): cpu = 00:00:25 ; elapsed = 00:00:26 . Memory (MB): peak = 556.844 ; gain = 364.824
report_utilization: Time (s): cpu = 00:00:00 ; elapsed = 00:00:00.060 . Memory (MB): peak = 556.844 ; gain = 0.000
INFO: [Common 17-206] Exiting Vivado at Sat Aug 22 16:56:07 2015...
```

5.4. Expand **Open Synthesized Design** under **Synthesis** in the **Flow Navigator**. Click **Schematic**. This opens up the synthesized netlist in the Sources window describing the hardware. A schematic appears in the Workspace.

Select the LUT3[10] labeled **SUM_OBUF_inst_i_1** in the schematic and then click on **Truth Table** in the Data Windows area. You will observe from the Truth Table that it indeed encodes the behaviour for SUM output. Similarly the CARRY output is encoded as another Truth Table **COUT_OBUF_inst_i_1**.



```
C:/Users/Shahzor Ahmad/Desktop/CG3207_GS/CG3207_GS.runs/synth_1/full_adder_utilization_synth.rpt

1. Slice Logic
--------------

+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs*             |    1 |     0 |     63400 | <0.01 |
|   LUT as Logic          |    1 |     0 |     63400 | <0.01 |
|   LUT as Memory         |    0 |     0 |     19000 |  0.00 |
| Slice Registers         |    0 |     0 |    126800 |  0.00 |
|   Register as Flip Flop |    0 |     0 |    126800 |  0.00 |
|   Register as Latch     |    0 |     0 |    126800 |  0.00 |
| F7 Muxes                |    0 |     0 |     31700 |  0.00 |
| F8 Muxes                |    0 |     0 |     15850 |  0.00 |
+-------------------------+------+-------+-----------+-------+
* Warning! The Final LUT count, after physical optimizations and full implementatio
```

5.5. Resource utilization may be opened in the Workspace for analysis by clicking **Utilization Report** (red) in the **Reports** tab at the bottom of the IDE.

We observe only one slice[11] LUT is used as logic. No flip-flops are used as this is a purely combinational circuit.

Utilization may also be observed by clicking on **Report Utilization** in the **Flow Navigator**. Similarly, Report Timing Summary, Report Noise and Report Power may be used.

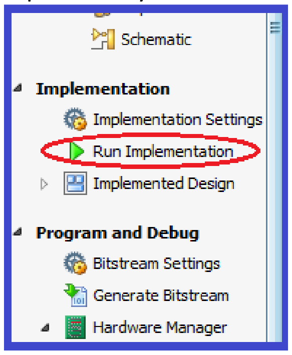
Post-implementation reports may be similarly analysed, and are more accurate as they are generated after placement and routing.

[10] Field-Programmable Gate Arrays (FPGAs) don't really contain gates! Instead they contain lots of CLBs (configurable logic blocks), consisting of multi-input LUTs (look-up tables) to implement combinational logic, and flip-flops to implement registers. It is easy to appreciate from this example how a LUT has been configured to behave as a full adder.

[11] XC7A100T contains 15850 slices. Each slice contains 04 LUTs, 8 flip-flops. Two slices form a CLB. Other resources on the FPGA include dedicated DSP slices, Block RAM, Clock Management & PLL for frequency synthesis, ADCs and I/O.

6. Implementation, Bitstream Generation & FPGA Configuration

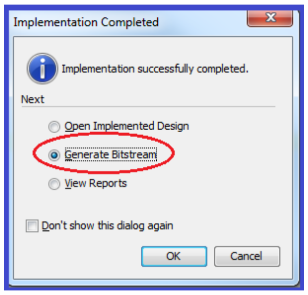
Implementation involves optimal placement and routing (PAR), where the netlist elements are mapped to the FPGA's physical resources and interconnected together. The I/O and timing constraints specified by the user are also respected.



6.1. Click on **Run Implementation** under **Implementation** in the **Flow Navigator**. If a Synthesis is Out-of-date message pops up, prompting to re-run synthesis, click **Yes**.

While Vivado performs synthesis and/or implementation, the **Project Status Bar** at the top right of the IDE provides visual indication.

A log appears in the **Log** tab in the **Results Windows**, while any warnings or errors are displayed in the **Messages** tab.


Implementation Completed

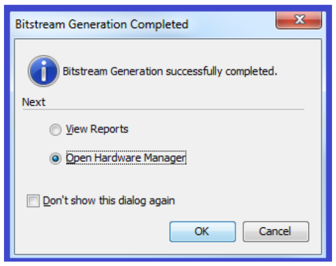
Implementation successfully completed.

Next

- Open Implemented Design
- Generate Bitstream [✓]
- View Reports

Don't show this dialog again

6.2. Once implementation is finished, Vivado asks whether to proceed to generate a bitstream (which would then be used to configure the FPGA with our design). Select **Generate Bitstream** radio button, and click **OK**.


Bitstream Generation Completed

Bitstream Generation successfully completed.

Next

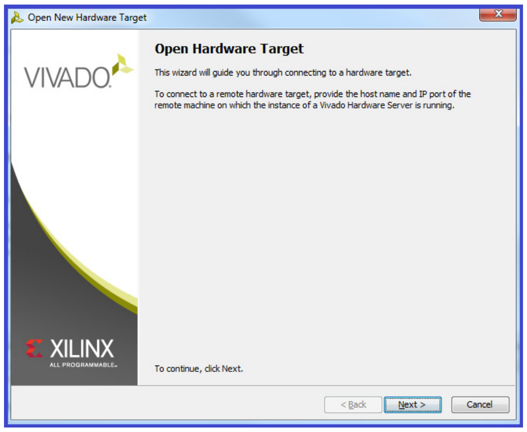
- View Reports
- Open Hardware Manager [✓]

Don't show this dialog again

6.3. Once done, select **Open Hardware Manager** and click **OK**.

In the Flow Navigator, under **Program and Debug → Hardware Manager → click Open Target → Open New Target.**

Since Hardware Manager was opened just now, same may also be done from the pane just above the Sources Window / Text Editor.


Open New Hardware Target

Open Hardware Target

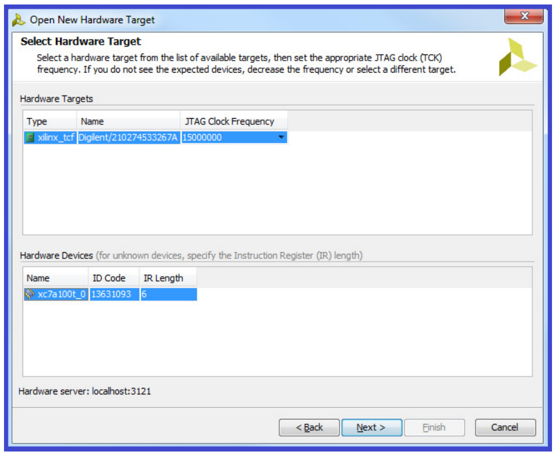
This wizard will guide you through connecting to a hardware target.

To connect to a remote hardware target, provide the host name and IP port of the remote machine on which the instance of a Vivado Hardware Server is running.

To continue, click Next.

6.4. Click **Next**. Ensure **Local server** is selected. At this point plug in your FPGA board into the USB port of your PC, and turn on the power switch (SW16)[12]. Click **Next**.

[12] Every time you power it up, the FPGA is auto configured to behave as a demo circuit by an on-board flash device. Some of the cool stuff: pressing BTNU triggers a recording for 5s. For the next 5s, you can hear it if you plug your earphones into the audio jack! Also try hooking up the VGA output to a screen, and connecting a mouse into the USB host port.



Select Hardware Target

Select a hardware target from the list of available targets, then set the appropriate JTAG clock (TCK) frequency. If you do not see the expected devices, decrease the frequency or select a different target.

Hardware Targets

Type	Name	JTAG Clock Frequency
xilinx_tcf	Digilent/210274533267A	15000000

Hardware Devices (for unknown devices, specify the Instruction Register (IR) length)

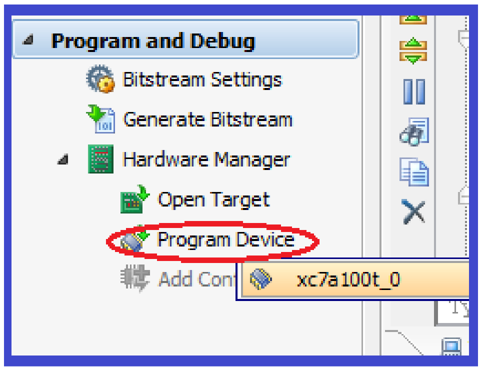
Name	ID Code	IR Length
xc7a100t_0	13631093	6

Hardware server: localhost:3121

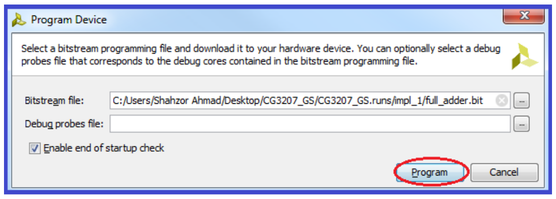
6.5. If the board cannot be detected, power cycle the board, wait for Windows USB drivers to load up and try again.

If successfully detected, a hardware target (JTAG interface on the Nexys-4 board), and a hardware device (our Artix-7 chip) is shown.

Select the **xc7a100t_0** device. Click **Next** and **Finish**.



6.6. Click **Program Device**. Then click the **xc7a100t_0** that pops up.



Select a bitstream programming file and download it to your hardware device. You can optionally select a debug probes file that corresponds to the debug cores contained in the bitstream programming file.

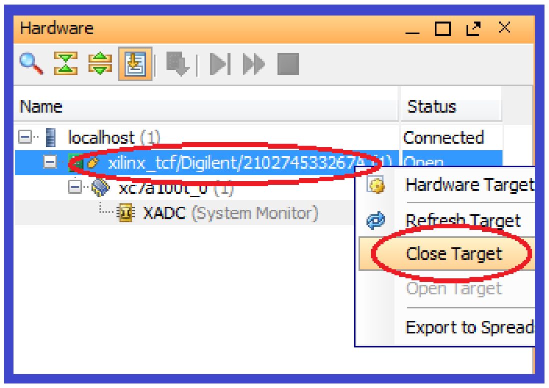
Bitstream file: C:/Users/Shahzor Ahmad/Desktop/CG3207_GS/CG3207_GS.runs/impl_1/full_adder.bit

Debug probes file:

☑ Enable end of startup check

6.7. Click **Program** on the **Program Device** window that appears.

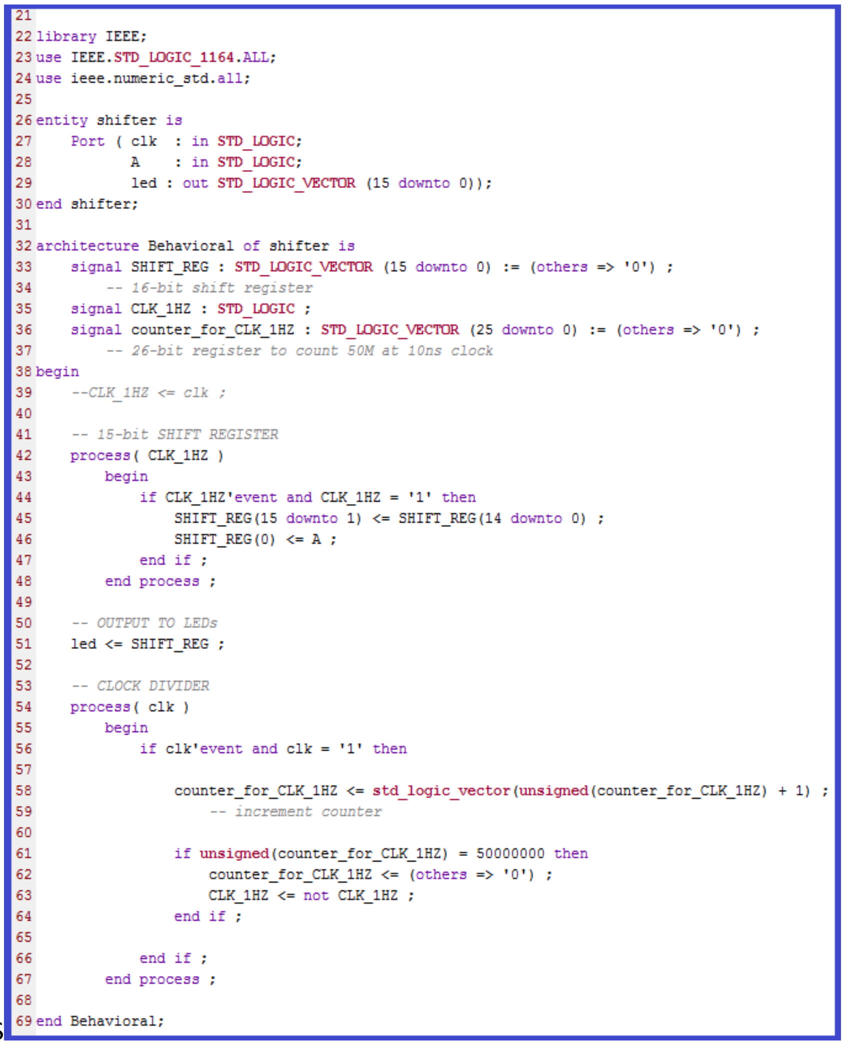
6.8. Once the FPGA is programmed with our design, verify the functionality by providing inputs with switches SW0, SW1 and SW2, and observing LEDs LD0 and LD1.



6.9. Before switching off your Nexys 4, disconnect it from Vivado by right-clicking on the the taget shown in **Hardware** window → **Close Target**.

Congratulations!
You just successfully completed your first Artix-7 FPGA design flow ☺

7. Design Flow for a Sample Sequential Circuit – VHDL



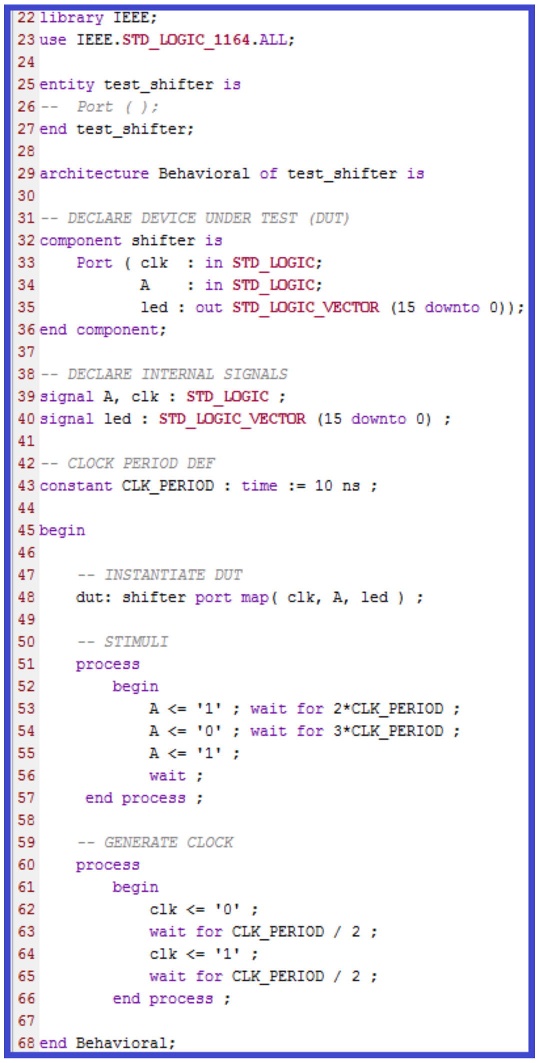
```
21
22 library IEEE;
23 use IEEE.STD_LOGIC_1164.ALL;
24 use ieee.numeric_std.all;
25
26 entity shifter is
27     Port ( clk  : in STD_LOGIC;
28            A    : in STD_LOGIC;
29            led : out STD_LOGIC_VECTOR (15 downto 0));
30 end shifter;
31
32 architecture Behavioral of shifter is
33     signal SHIFT_REG : STD_LOGIC_VECTOR (15 downto 0) := (others => '0') ;
34         -- 16-bit shift register
35     signal CLK_1HZ : STD_LOGIC ;
36     signal counter_for_CLK_1HZ : STD_LOGIC_VECTOR (25 downto 0) := (others => '0') ;
37         -- 26-bit register to count 50M at 10ns clock
38 begin
39     --CLK_1HZ <= clk ;
40
41     -- 15-bit SHIFT REGISTER
42     process( CLK_1HZ )
43         begin
44             if CLK_1HZ'event and CLK_1HZ = '1' then
45                 SHIFT_REG(15 downto 1) <= SHIFT_REG(14 downto 0) ;
46                 SHIFT_REG(0) <= A ;
47             end if ;
48         end process ;
49
50     -- OUTPUT TO LEDs
51     led <= SHIFT_REG ;
52
53     -- CLOCK DIVIDER
54     process( clk )
55         begin
56             if clk'event and clk = '1' then
57
58                 counter_for_CLK_1HZ <= std_logic_vector(unsigned(counter_for_CLK_1HZ) + 1) ;
59                     -- increment counter
60
61                 if unsigned(counter_for_CLK_1HZ) = 50000000 then
62                     counter_for_CLK_1HZ <= (others => '0') ;
63                     CLK_1HZ <= not CLK_1HZ ;
64                 end if ;
65
66             end if ;
67         end process ;
68
69 end Behavioral;
```

S

The HDL above implements a **16-bit shift register** via a clocked process. A crystal oscillator provides a 100MHz clock to the Artix-7 (connected to input port clk). The code above also implements a 26-bit counter to divide the clock and generate a 1Hz clock. This 1Hz signal is used to clock the shift register. Input port A – connected to SW0 – is used to provide new input to bit 0 of the shift register every clock period. The status of the shift register is displayed on LEDs.

7.1. Create a project and design source as in Sec. 1. As in Sec. 2, edit the source file to look like the figure above.

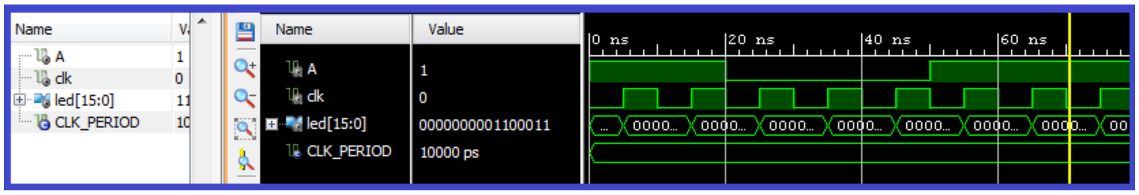
7.2. Create a simulation source (see Sec. 3) to enter the testbench as in figure below. Observe a process using wait statements essentially generates the `clk`. Sample stimuli have been added, which can be used to verify the behaviour of the design. NOTE: for simulation, un-comment line#39 and comment out lines # 54 – 69 in the design source, while the opposite should be done for implementation (Why?). You should be able to obtain the simulation output as given in the figure and analyse it.



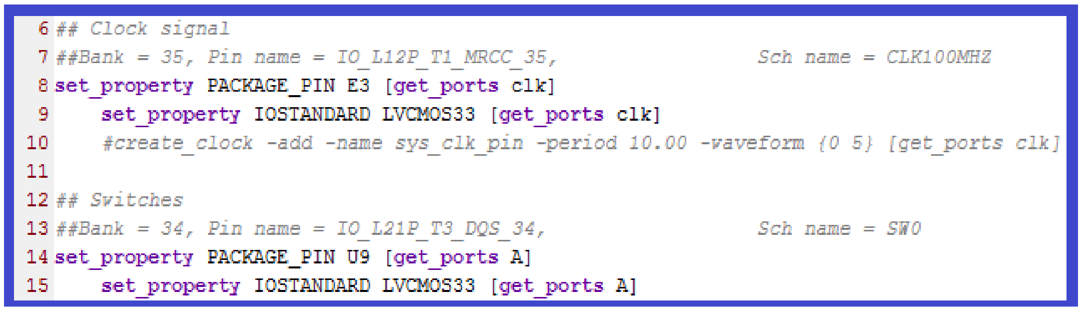
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity test_shifter is
--  Port ( );
end test_shifter;

architecture Behavioral of test_shifter is

-- DECLARE DEVICE UNDER TEST (DUT)
component shifter is
    Port ( clk  : in STD_LOGIC;
           A    : in STD_LOGIC;
           led : out STD_LOGIC_VECTOR (15 downto 0));
end component;

-- DECLARE INTERNAL SIGNALS
signal A, clk : STD_LOGIC ;
signal led : STD_LOGIC_VECTOR (15 downto 0) ;

-- CLOCK PERIOD DEF
constant CLK_PERIOD : time := 10 ns ;

begin

    -- INSTANTIATE DUT
    dut: shifter port map( clk, A, led ) ;

    -- STIMULI
    process
        begin
            A <= '1' ; wait for 2*CLK_PERIOD ;
            A <= '0' ; wait for 3*CLK_PERIOD ;
            A <= '1' ;
            wait ;
     end process ;

    -- GENERATE CLOCK
    process
        begin
            clk <= '0' ;
            wait for CLK_PERIOD / 2 ;
            clk <= '1' ;
            wait for CLK_PERIOD / 2 ;
        end process ;

end Behavioral;
```

7.3. Now revert back to the original source. Add the Nexys 4 master constraints file as in Sec. 4. Un-comment the lines for clock signal and input SW0 (rename it to A) as in figure below. You will also need to un-comment constraint lines for all LEDs (no renaming required).

7.4. Synthesize and implement this design as in Secs. 5 – 6. Verify the functionality of the circuit in hardware.

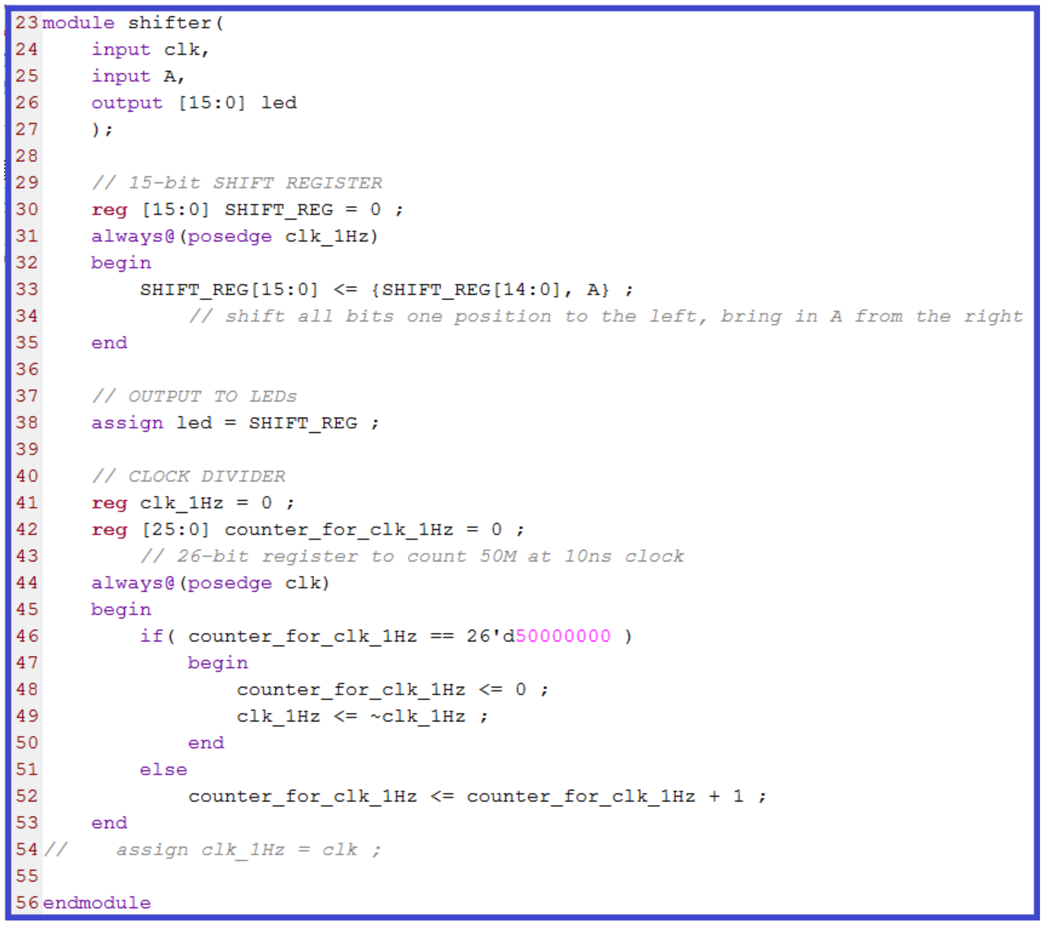


```
Name          V
  A           1
  clk         0
  led[15:0]   11
  CLK_PERIOD  10

Name          Value
  A           1
  clk         0
  led[15:0]   0000000001100011
  CLK_PERIOD  10000 ps
```



```
## Clock signal
##Bank = 35, Pin name = IO_L12P_T1_MRCC_35,                    Sch name = CLK100MHZ
set_property PACKAGE_PIN E3 [get_ports clk]
    set_property IOSTANDARD LVCMOS33 [get_ports clk]
    #create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports clk]

## Switches
##Bank = 34, Pin name = IO_L21P_T3_DQS_34,                     Sch name = SW0
set_property PACKAGE_PIN U9 [get_ports A]
    set_property IOSTANDARD LVCMOS33 [get_ports A]
```

8. Design Flow for a Sample Sequential Circuit – Verilog

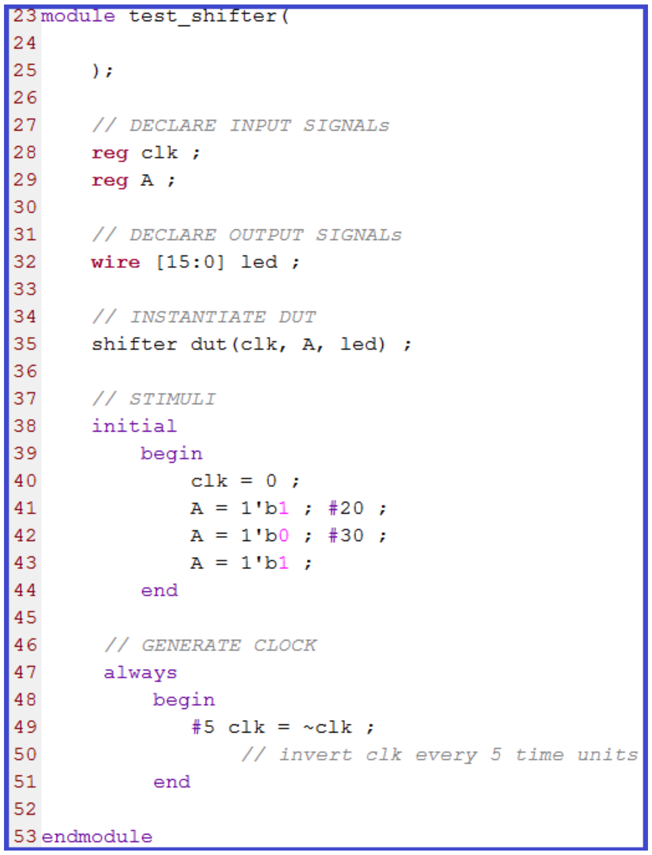


```
23 module shifter(
24     input clk,
25     input A,
26     output [15:0] led
27     );
28
29     // 15-bit SHIFT REGISTER
30     reg [15:0] SHIFT_REG = 0 ;
31     always@(posedge clk_1Hz)
32     begin
33         SHIFT_REG[15:0] <= {SHIFT_REG[14:0], A} ;
34             // shift all bits one position to the left, bring in A from the right
35     end
36
37     // OUTPUT TO LEDs
38     assign led = SHIFT_REG ;
39
40     // CLOCK DIVIDER
41     reg clk_1Hz = 0 ;
42     reg [25:0] counter_for_clk_1Hz = 0 ;
43         // 26-bit register to count 50M at 10ns clock
44     always@(posedge clk)
45     begin
46         if( counter_for_clk_1Hz == 26'd50000000 )
47             begin
48                 counter_for_clk_1Hz <= 0 ;
49                 clk_1Hz <= ~clk_1Hz ;
50             end
51         else
52             counter_for_clk_1Hz <= counter_for_clk_1Hz + 1 ;
53     end
54 //   assign clk_1Hz = clk ;
55
56 endmodule
```

The HDL above implements a **16-bit shift register** using an `always` block sensitized to the positive clock edge. A crystal oscillator provides a 100MHz clock to the Artix-7 (connected to input port clk). The code above also implements a 26-bit counter to divide the clock and generate a 1Hz clock. This 1Hz signal is used to clock the shift register. Input port A – connected to SW0 – is used to provide new input to bit 0 of the shift register every clock period. The status of the shift register is displayed on LEDs.

8.1. Create a project and design source as in Sec. 1. As in Sec. 2, edit the source file to look like the figure above.

8.2. Create a simulation source (see Sec. 3) to enter the testbench as in figure below. Sample stimuli have been added, which can be used to verify the behaviour of the design. NOTE: for simulation, un-comment line#54 and comment out lines # 41 – 53 in the design source, while the opposite should be done for implementation (Why?). You should be able to obtain the simulation output as given in the figure on page 18 and analyse it.



```
module test_shifter(

    );

    // DECLARE INPUT SIGNALs
    reg clk ;
    reg A ;

    // DECLARE OUTPUT SIGNALs
    wire [15:0] led ;

    // INSTANTIATE DUT
    shifter dut(clk, A, led) ;

    // STIMULI
    initial
        begin
            clk = 0 ;
            A = 1'b1 ; #20 ;
            A = 1'b0 ; #30 ;
            A = 1'b1 ;
        end

     // GENERATE CLOCK
     always
         begin
            #5 clk = ~clk ;
                // invert clk every 5 time units
         end

endmodule
```

8.3. Now revert back to the original source. Add the Nexys 4 master constraints file as in Sec. 4. Un-comment the lines for clock signal and input SW0 (rename it to A) as in figure on page 18. You will also need to un-comment constraint lines for all LEDs (no renaming required).

8.4. Synthesize and implement this design as in Secs. 5 – 6. Verify the functionality of the circuit in hardware.