Terasic DE2i-150 Board

Version 1.0

What you Learn by following this Documentation

By following this documentation you will learn how to use Quartus II tool to create designs and configure them on the FPGA. Next going a step forward you learn what is Qsys tool and how it is used in implementing the PCIe system on DE2i-150 board using the Terasic PCIe framework. Finally you build a system using Qsys tool with your own component(counter) and write a C application which runs on Intel Atom processor to read the output of the counter over PCIe.

Points to know before using this Documentation

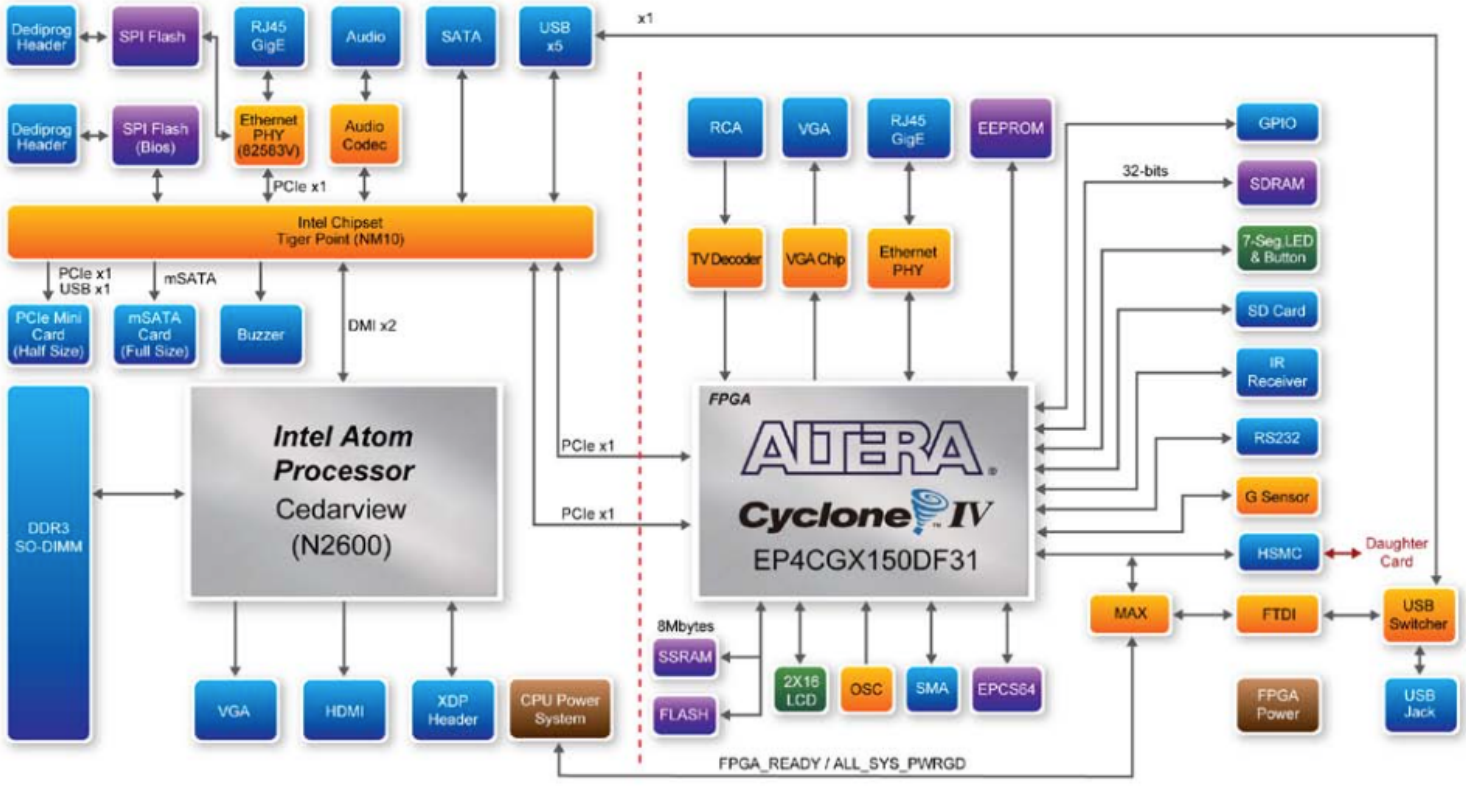
- We referred many documentations to learn about DE2i-150 and make this document. All referred documentations are mentioned in the index.
- We took some text directly from referred documentation because they are clearly understandable by person with minimal knowledge also. All the text in the **blue** color is taken from the referred documentations directly without any changes.
- Basic knowledge in Linux commands, verilog HDL and C programming language.

Chapter 1 : Introduction

1.1 About board

DE2i-150 development board is developed by **Terasic** which consists of **Altera Cyclone® IV 4GX150** FPGA device and **Intel Atom N2600** processor. The board has many other hardware peripherals which allow user to implement wide range of applications.

The block digram of DE2i-150: From red dotted line Right side CPU system and left side FPGA System



Source : DE2i - 150 Development Kit FPGA SYSTEM USERS MANUAL

FPGA system Hardware Components :

- Altera Cyclone® IV 4CX 150 FPGA device
- Altera serial configuration device – EPCS64
- USB Blaster (on board) for programming; both JTAG and Active Serial (AS) programming modes are supported
- Two 2MB SSRAM
- Two 64MB SDRAM
- 64MB Flash memory
- SD Card socket
- 4 Push-buttons
- 18 Slide switches
- 18 Red user LEDs
- 9 Green user LEDs
- 50MHz oscillator for clock sources
- VGA DAC (8-bit high-speed triple DACs) with VGA – out connector
- TV Decoder (NTSC/PAL/SECAM) and TV-in connector
- Gigabit Ethernet PHY with RJ45 connectors
- RS-232 transceiver and 9-pin connector
- IR Receiver
- 2 SMA connectors for external clock input/output
- One 40-pin Expansion Header with diode protection
- One High Speed Mezzanine Card (HSMC) connector
- 16x2 LCD module

CPU system Hardware Components:

- Atom N2600 (cedartrail) dual-core dual-thread 1.6GHz
- Atom SSE2, SSE3, SSSE3
- on-chip 400MHz Graphics Processing Unit
- 3.5W Thermal Design Power
- NM - 10 chipset
- USB, SATA, LAN, Audio, miniPCI, 2 PCIx1 to FPGA
- 2GB DDR3 (SO-DIMM) system memory
- 64GB mSATA SSD
- Ports: Wifi, Ethernet, 4xUSB, HDMI, VGA

The CPU System is installed with Yocto Embedded OS.

1.2 Brief Introduction to Quartus II

Altera Quartus II is a programmable logic device design software produced by Altera. Quartus II enables analysis and synthesis of HDL designs, which enables the developer to compile their designs, perform timing analysis, examine RTL diagrams, simulate a design's reaction to

different stimuli, and configure the target device with the programmer. Quartus includes an implementation of VHDL and Verilog for hardware description, visual editing of logic circuits, and vector waveform simulation.

1.3 Requirements to follow Documentation

A long with the DE2i-150 board following are required

- Workstation - Desktop or Laptop with Ubuntu 14.04.2 LTS
- Monitor - To operate Yocto embedded linux OS(host) on board
- Keyboard and mouse - To operate Yocto embedded linux OS(host) on board
- Ethernet LAN cable - To transfer files from Workstation to board

Chapter 2 : Installing Quartus II and setting up USB Blaster drivers

---

2.1 Downloading and Installation Quartus II 14.1.0 Web Edition

Downloading

Quartus II tools come with board. In the case you don't have the tools the other way of getting them is from the Altera website. To get Quartus II 14.1.0 form the altera Website go to the following link. The link opens the following webpage. Select the following options

- Release -14.1.
- Operating system - Linux.
- Download method of your own type. Akamai DLM3 Download manager is recommended because it resumes the download in the case of any interrupts.
- Under Combined Files tab, select download option to download.

Installation

The file **'Quartus-web-14.1.0.186-linux.tar'** will be downloaded. From the Ubuntu Terminal change directory to the file download location and extract the file using following command

```
navatejareddy@ubuntu:~$ tar -xvf Quartus-web-14.1.0.186-linux.tar
```

After executing the above command you can find the folder with same name in the directory. Change directory to that folder. You can find **'setup.sh'** shell script file. Run the shell script using the following command

```
navatejareddy@ubuntu:~$ sudo sh setup.sh
```

Running the shell script launches the standard setup window. The installing instructions are easily understandable, follow the step by step instructions to install the Quartus II..

Note: Quartus II may crash while running on Ubuntu. To overcome this problem, in altera/14.1/quartus/linux64 folder, rename libcurl.so.4 to libcurl.so.4.bak and restart the Quartus II

2.2 Setting up USB Blaster drivers

Before starting your design and configuring/programming the FPGA first **you need to setup the USB Blaster drivers on your workstation to communicate with the board through** integrated USB Blaster circuitry by which you can program the FPGA on board. The two types of programming supported by FPGA system are

JTAG programming: In this method of programming, named after the IEEE standards Joint Test Action Group, the configuration bit stream is downloaded directly into the Cyclone IV GX FPGA. The FPGA will retain this configuration as long as power is applied to the board; the configuration information will be lost when the power is turned off.

AS programming: In this method, called Active Serial programming[7], the configuration bit stream is downloaded into the Altera EPCS64 serial configuration device. It provides non-volatile storage of the bit stream, so that the information is retained even when the power supply to the DE2i-150 board is turned off. When the board's power is turned on, the configuration data in the EPCS64 device is automatically loaded in to the Cyclone IV GX FPGA.

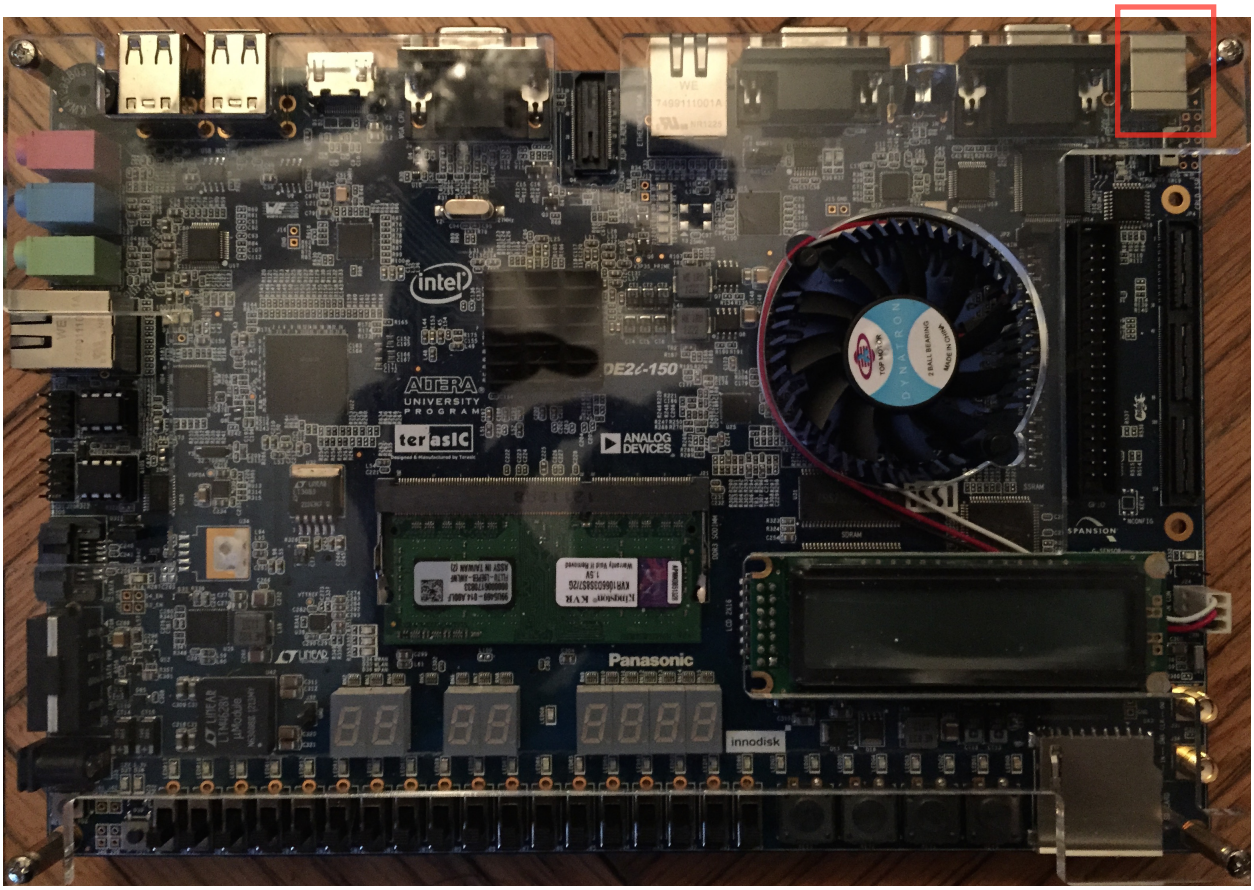
For the workstation to communicate with the FPGA through the USB blaster circuitry, Drivers are needed to be setup on the workstation. Follow the following steps to setup the drivers.

Before proceeding, to get command-line access to the commands `jtagd` and `jtagconfig` which is used in this setup add the following line to **.bashrc** file in your home folder.

```
export PATH=$PATH:/home/<username>/altera/14.1/quartus/bin
```

STEP 1 : Verify USB connection and check Product ID

At first, switch on the board and then connect the cable between the workstation and DE2i-150 board, don't forget to follow this order. The position of USB on board is shown in figure below(hi-lighted in red)



To make sure that board is recognized, type `lsusb` in your terminal, then you see the list of all USB devices connected to your workstation. From that list you need to notice line which looks similar as shown below(Highlighted in red)

```
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 012: ID 09fb:6001 Altera Blaster
Bus 002 Device 011: ID 0e0f:0008 VMware, Inc.
Bus 002 Device 003: ID 0e0f:0002 VMware, Inc. Virtual USB Hub
Bus 002 Device 002: ID 0e0f:0003 VMware, Inc. Virtual Mouse
Bus 002 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
```

Take note of the Product ID i.e **6001**

STEP 2 : Fixing USB driver permissions

The Quartus software will use the Linux built-in **usb_device** drivers. By default, only root has access to these so we must make sure the user is allowed to access them as well. **jtagd**, part of the Quartus tools, is a daemon that provides the interface between the Altera tool accessing the **JTAG chain** and the USB driver. If not already running, jtagd will be started automatically when the Quartus software or when **jtagconfig** command is executed. You'll usually run these as a user, which means **jtagd** will also run as a user. That is why edited permission for the usb_device is necessary.
Create a file **/etc/udev/rules.d/51-usbblaster.rules** , make sure it has read permissions for root, and fill it with this content:

```
# Altera USBBlaster permissions.
SUBSYSTEM=="usb",\
ENV{DEVTYPE}=="usb_device",\
ATTR{idVendor}=="09fb",\
ATTR{idProduct}=="6001",\
MODE="0666",\
NAME="bus/usb/$env{BUSNUM}/$env{DEVNUM}",\
RUN+="/bin/chmod 0666 %c"
```

For the changes to take effect, reboot the workstation.

STEP 3 : Copy devices data for `jtagd`

Make sure jtag has access to the list of devices by executing following command:

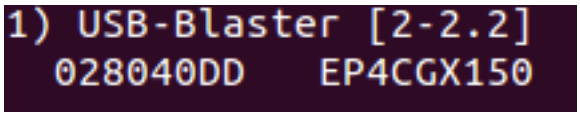
```
navatejareddy@ubuntu:~$ sudo cp /opt/altera/13.1/quartus/linux64/pgm_parts.txt /etc/jtagd/jtagd.pgm_parts
```

Also make sure this file has read access for the user.

This file allows Altera tools to translate Device IDs (left column of terminal listing below) to device names (right column) for found devices.

STEP 4 : Testing

To test the connection execute `jtagconfig` and you should see the output similar to shown below



If you see the lines similar to above you have configured USB Blaster drives successfully.

Note: To avoid Jtag errors when you execute jtagconfig command, every time you should start the board first and then connect the USB cable between board and workstation.

Chapter 3 : FPGA Programming : Running a simple counter

3.1 Creating Quartus project and designing the simple counter

In this chapter you learn how to create a design using the quartus II tool, how to program FPGA and finally test your design. Open quartus II tool by double clicking the Quartus II icon on the desktop. The starting window of quartus II tool is shown in fig 3.1.1

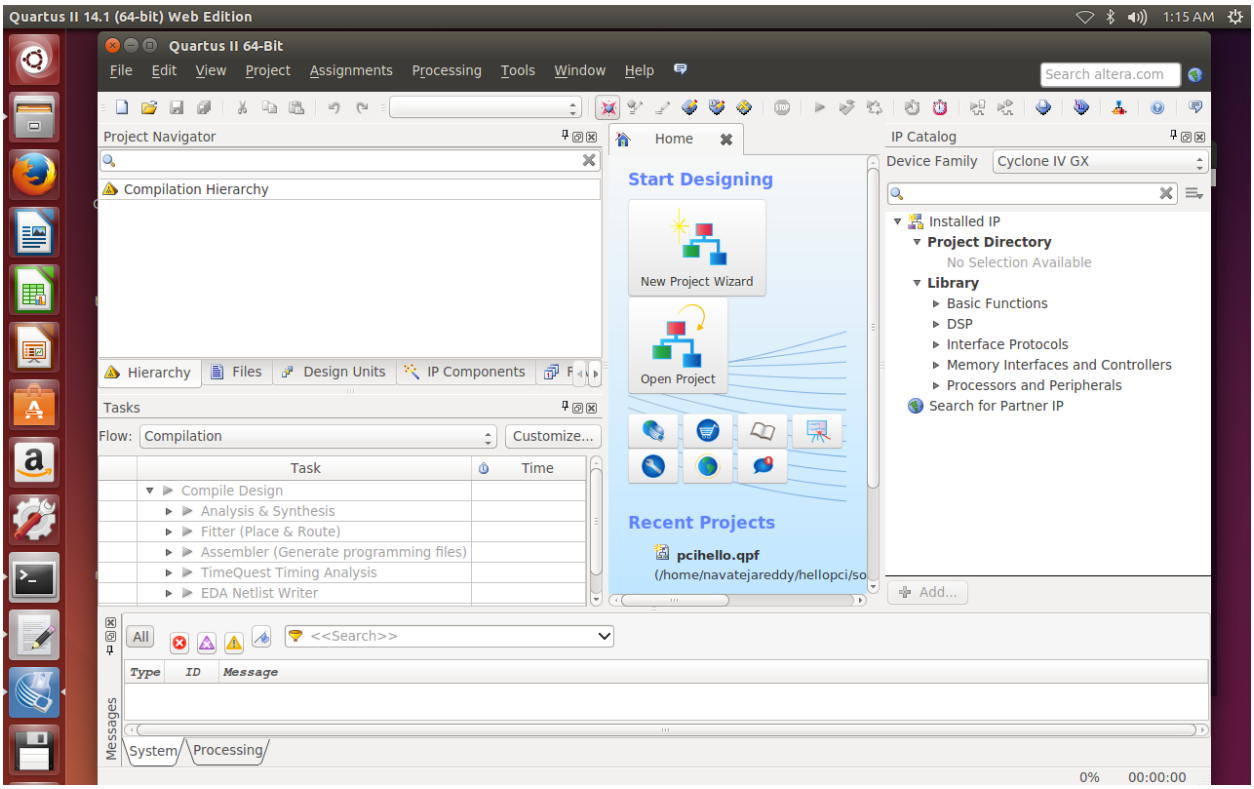


fig 3.1.1 : Quartus II

Creating the project

In the Quartus II software, select **File > New Project Wizard**. The introduction page opens. Click Next. From now follow step by step instructions as mentioned below

1. What is the working directory for this project? Enter a directory in which you will store your Quartus II project files for your design.
2. Give the name of the project. The top-level entity takes the same name as project. In case of this demo the name of project and top-level entity is **simple_counter**. See the fig 3.1.2

- note : File names, project names, and directories in the Quartus II software cannot contain spaces.

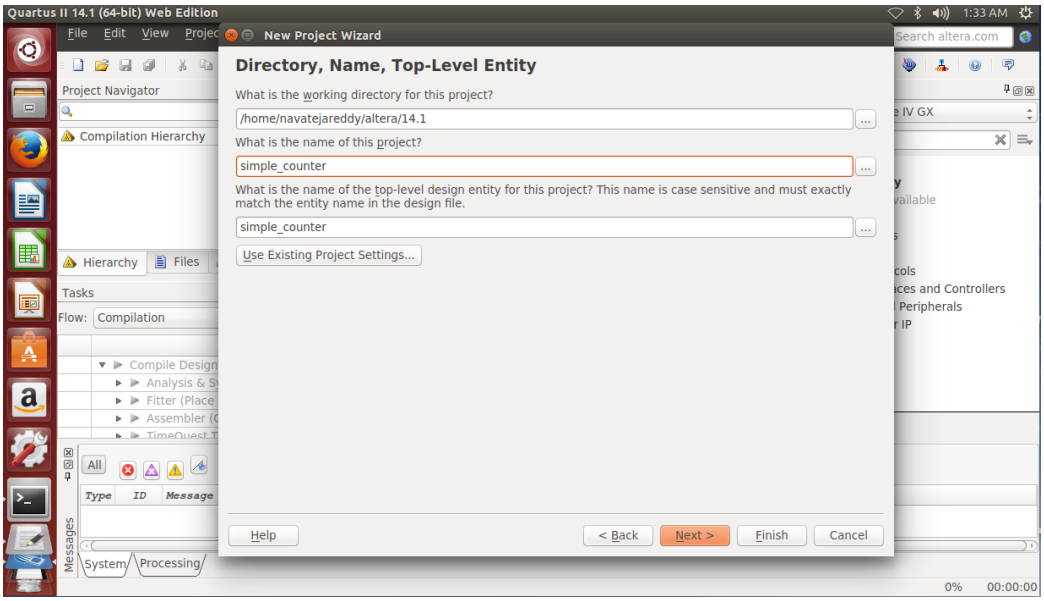


fig 3.1.2

3. Click Next.
4. Select the type of project as empty. See fig 3.1.3

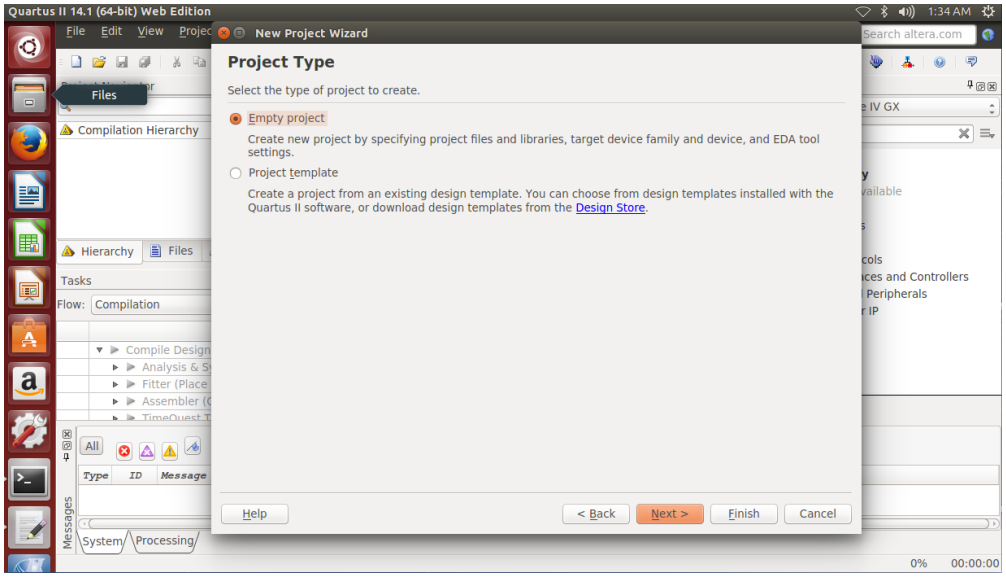


fig 3.1.3

5.Click Next

6.The next window is **Add File**. Here you can add any existing/previous project design files to your project. For this project we are not adding any design files so click next.

7. Now you should select the FPGA for which you are developing a designing. Under **Target device** select option **Specific device selected in Available devices list** by which the list **Available devices** get activated from that list you select cyclone IV **EP4CGX110DF31C7** FPGA device. See fig 3.1.4

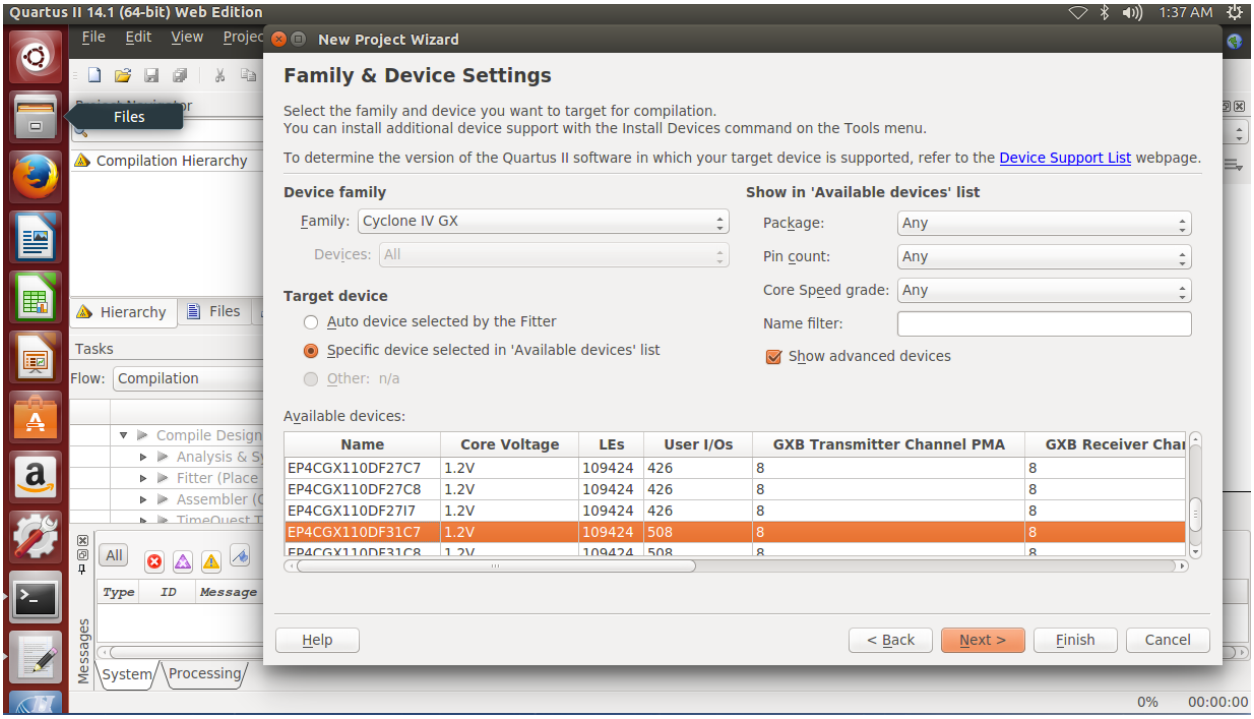


fig 3.1.4

8.Click Next

9.Here you can select any other EDA(Electronic Design Automation) tools you use along with Quartus II tool. For example you plan to use ModelSIM along with Quartus II tool to simulate the design then you need to mention it here in this EDA window. By default model sim is selected but any way we are not simulating our design before configuring the FPGA, so ignore this window and just click next.

10. Next you see the summary of all settings you gave to your project. After reviewing the summary you decide to make any changes just click back until you go the settings window you like to change. If everything is correct then click finish.

Designing the counter and using the altera PLL Megafunction

In the design entry step you create a schematic or **Block Design File (.bdf)** that is the top-level design. You will add altera **Megafunctions(PLL clock)** and use Verilog HDL code to add a **logic block(counter)**.

8.Choose **File > New > Block Diagram/Schematic File** (see fig 3.1.5 to create a new file, Block1.bdf, which will automatically save as the top-level design.

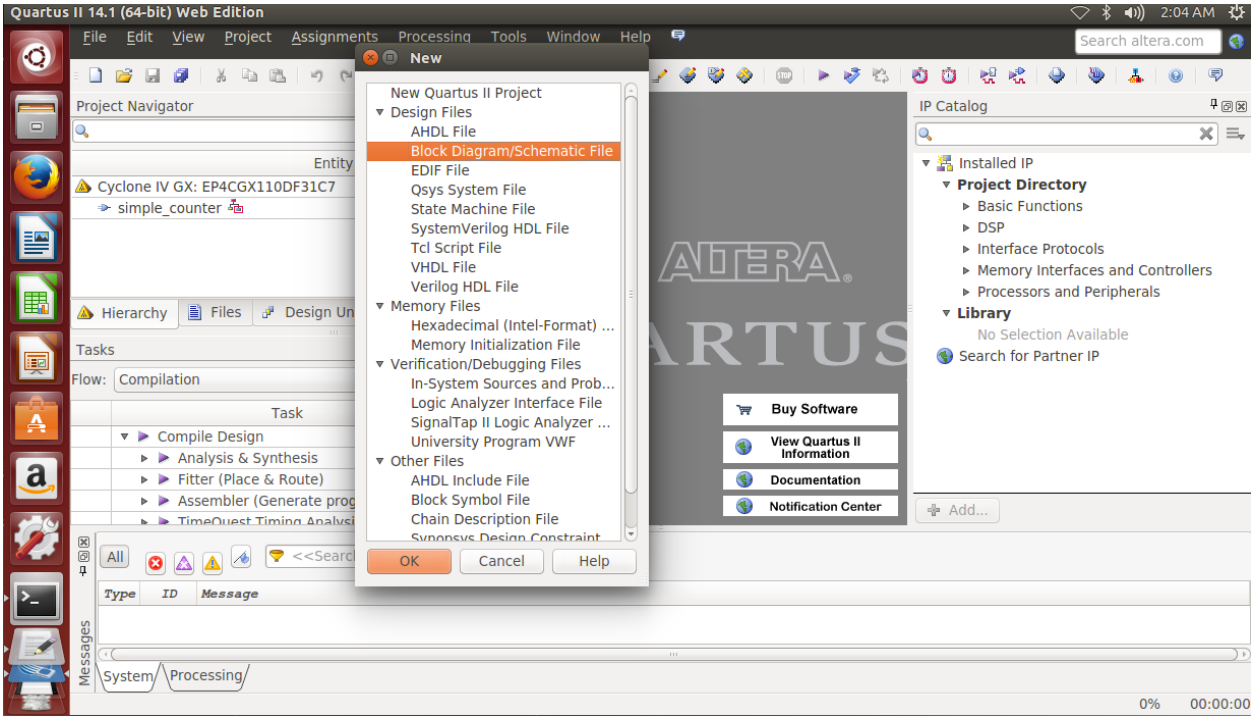


fig 3.1.5

9.Choose File > Save As and enter the name as '**simple_counter**'.

10.Click Save. The new design file appears in the Block Editor

11.Add HDL code to the blank block diagram by choosing **File > New > Verilog HDL File**.

12.Click OK to create a new file Verilog1.v, which you will save as **counter.v**

13.Type the following Verilog HDL code into the blank **counter.v** file

```
module simple_counter (
CLOCK_50,
counter_out
);
input CLOCK_50 ;
output [3:0] counter_out;
reg [3:0] counter_out;
always @ (posedge CLOCK_50)
begin
counter_out <= #1 counter_out + 1;// increment counter
end
endmodule
```

Note: The module name should be the same as top level module. In quarts II project the project name will be taken as top level module, which you can see in Hierarchy tab.

14. Save the file by choosing **File > Save**, pressing Ctrl + s, or by clicking the floppy disk icon.

15. **Choose File > Create/Update > Create Symbol Files for Current File** to convert the counter.v file to a **Symbol File(.sym)**.You use this Symbol File to add the HDL code to your BDF schematic.

The Quartus II software creates a Symbol File and displays a message showing “Create symbol file was successful”

16. To add the **counter.v** symbol to the top-level design, click the **simple_counter.bdf** tab.

17. Choose **Edit > Insert Symbol**.

18. Double-click the Project directory to expand it.

19. Select the newly created counter symbol by clicking it’s icon. See fig 3.1.6

You can also double-click in a blank area of the BDF to open the Symbol dialog box

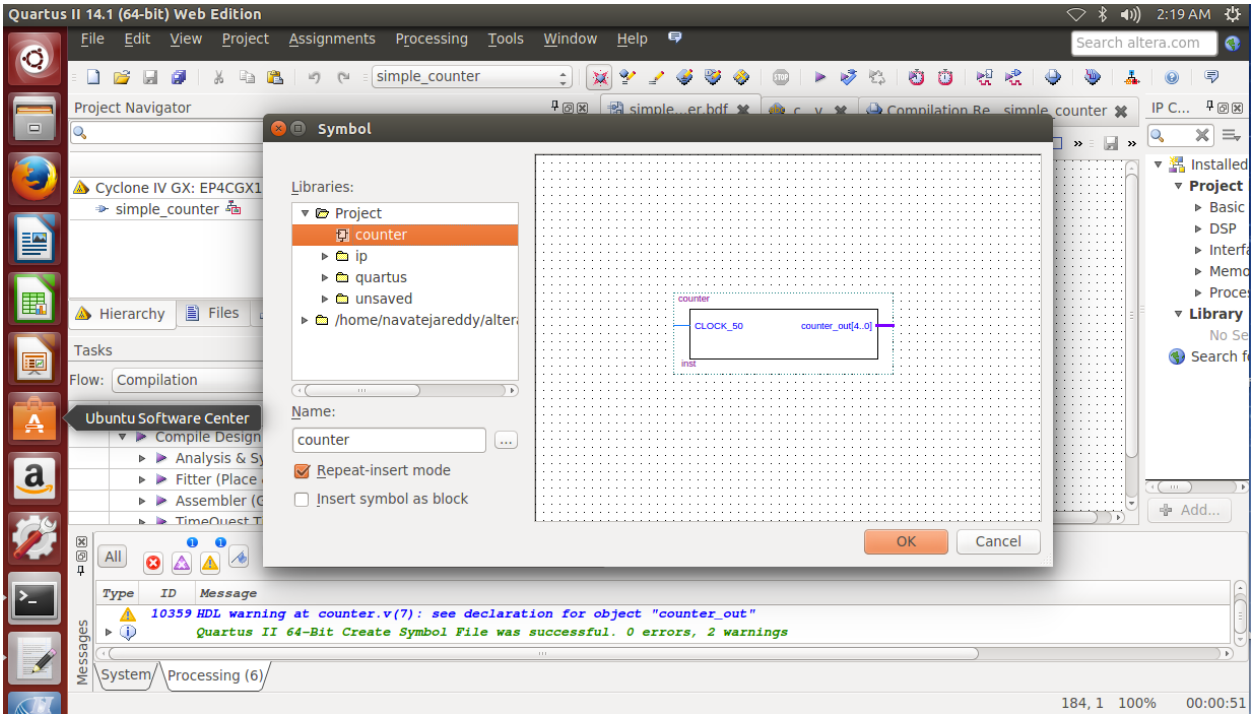


fig 3.1.6

20. Move the cursor to the BDF grid; the symbol image moves with the cursor. Click to place the counter symbol onto the BDF. You can move the block after placing it by simply clicking and dragging it to where you want it and releasing the mouse button to place it and press escape key to place further instance of this symbol.

21. Save your project regularly.

Adding a PLL Megafunction

Megafunctions are pre-designed modules that you can use in FPGA designs. These Altera-provided megafunctions are optimized for speed, area, and device family. You can increase

Efficiency by using a megafunction instead of writing the function yourself. Altera also provides more complex functions, called MegaCore functions, which you can evaluate for free but require a license file for use in production designs. This tutorial design uses a **PLL clock source** to drive a simple counter. A PLL uses the on-board oscillator (DE2i-150 Board is 50 MHz) to create a constant clock frequency as the input to the counter. To create the clock source, you will add a pre-built LPM megafunction named ALTPLL.

22. Choose Tools -> IP catalog. You can see Altera IP catalog menu.

23. Choose Basic Functions -> Clocks; PLLS and Resets -> PLL -> ALTPLL. Give name **clock_50.v** for the IP Variation file name. See fig 3.1.7

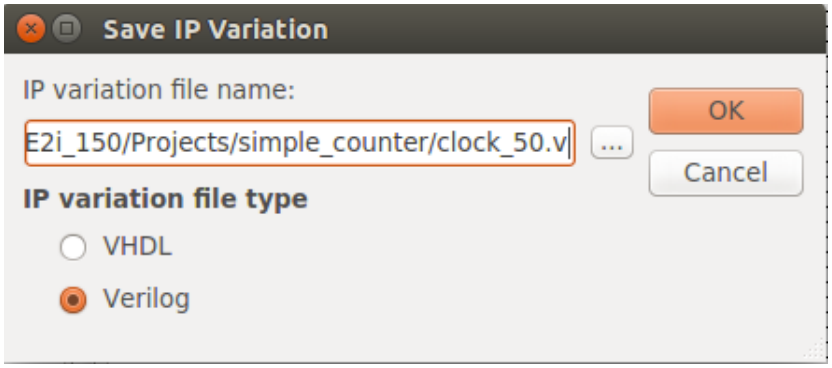


fig 3.1.7

24. The MegaWizard® Plug-In Manager opens

25. Under **Parameter settings -> General/Modes**, change the frequency of the **inclk0 input** to 50MHz and speed grade 6. See fig 3.1.8

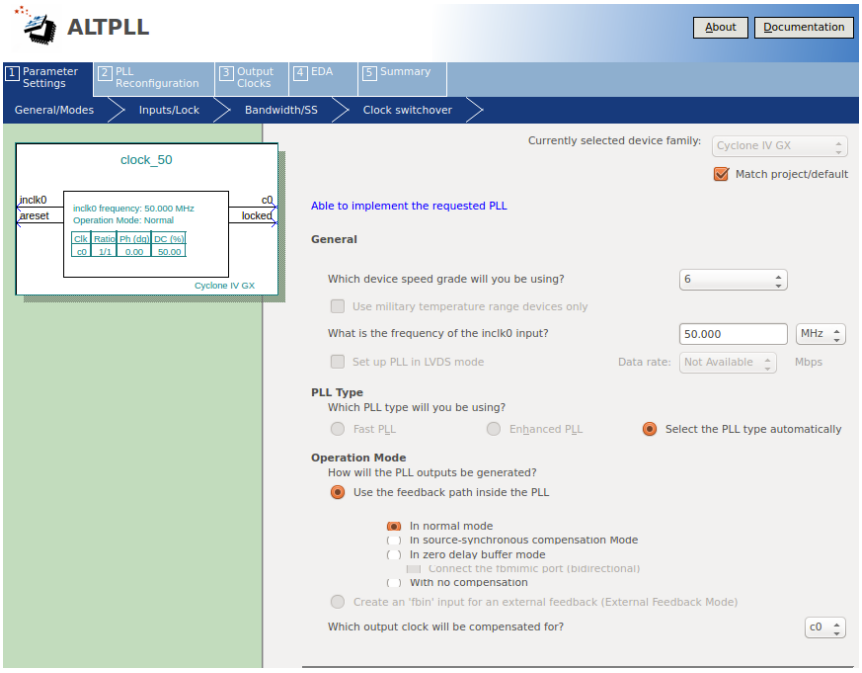


fig 3.1.8

26. Choose Parameter **settings -> Inputs/locks** and unselect all options. See fig 3.1.9

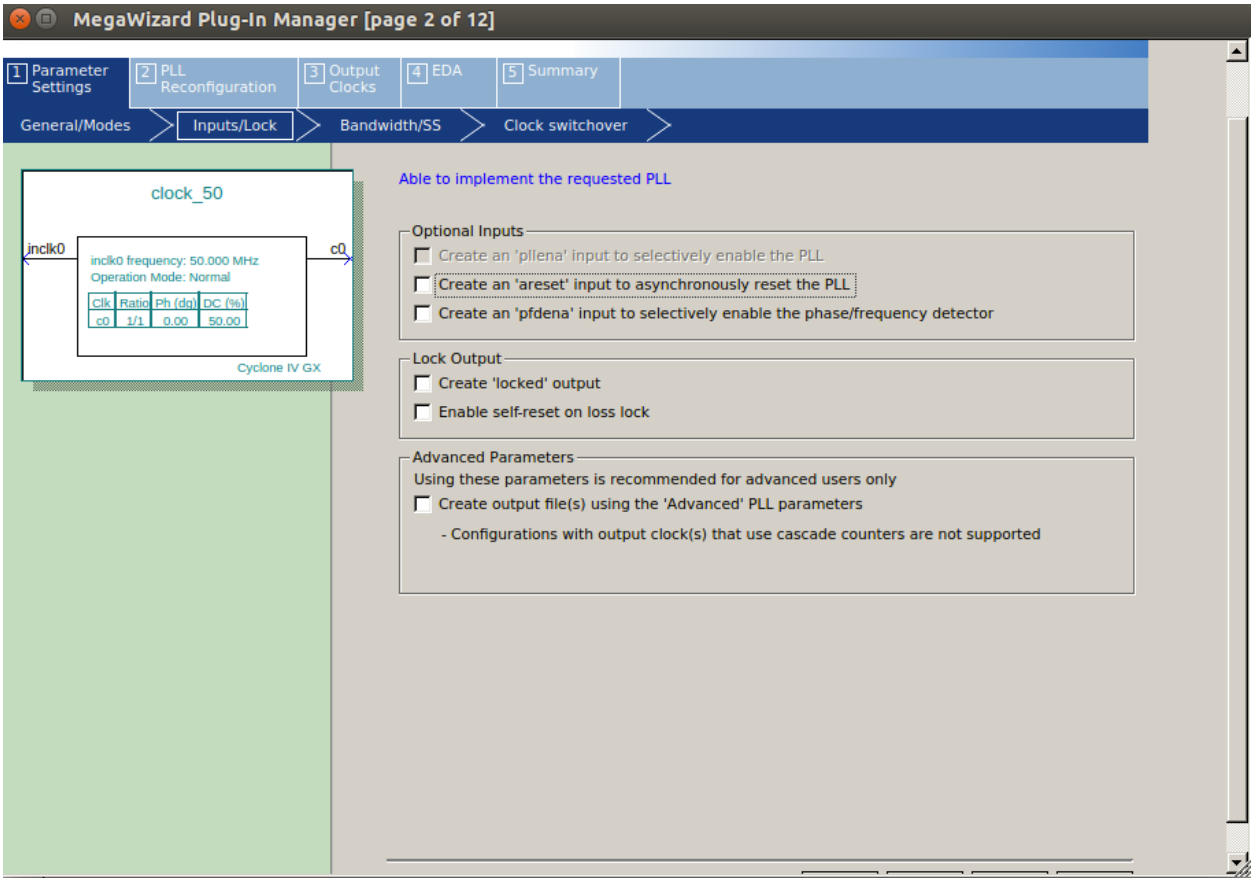


fig 3.1.9

27. Choose output **clocks -> clk c0** and change Clock division factor input to 10. See fig 3.1.10

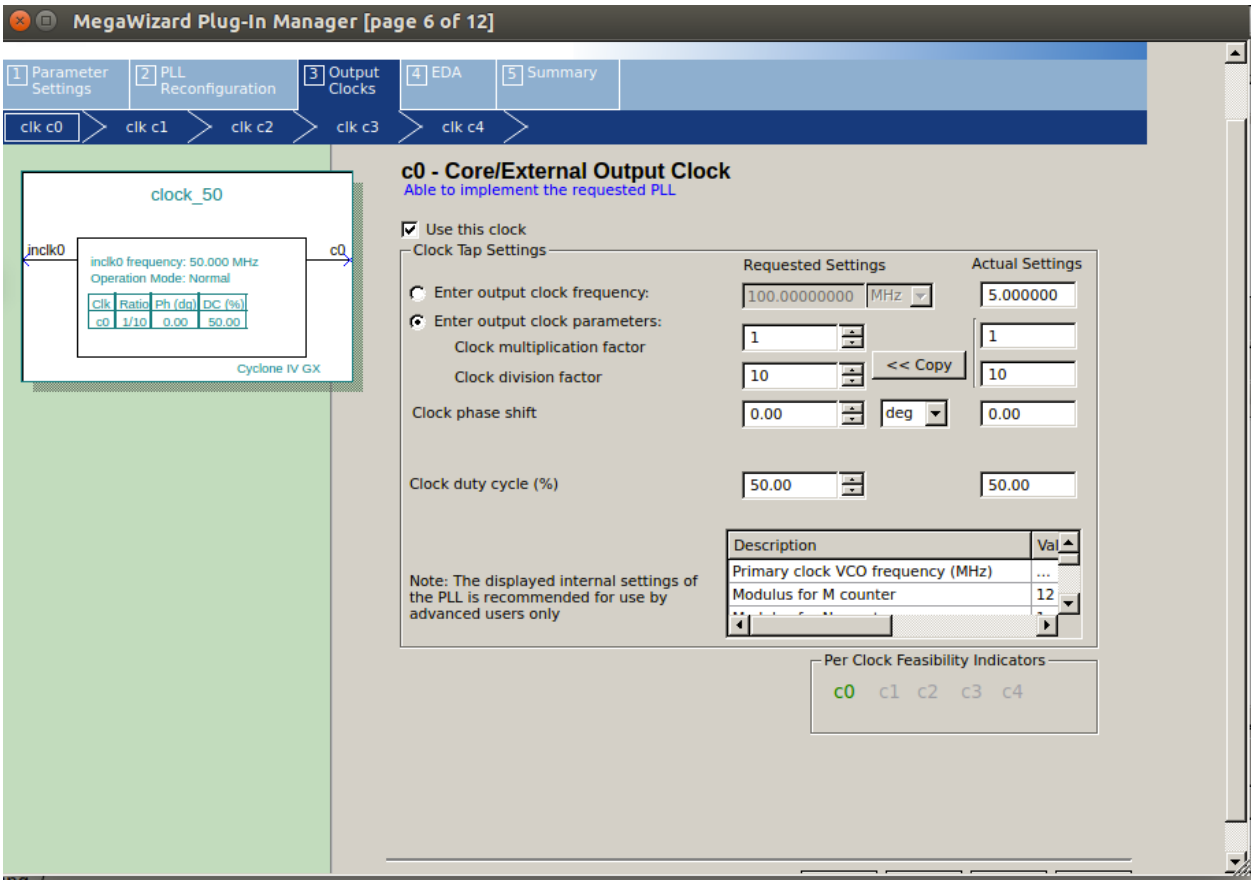


fig 3.1.10

28. Choose summary. Select file **clock_50.bdf** and click finish. See fig 3.1.11

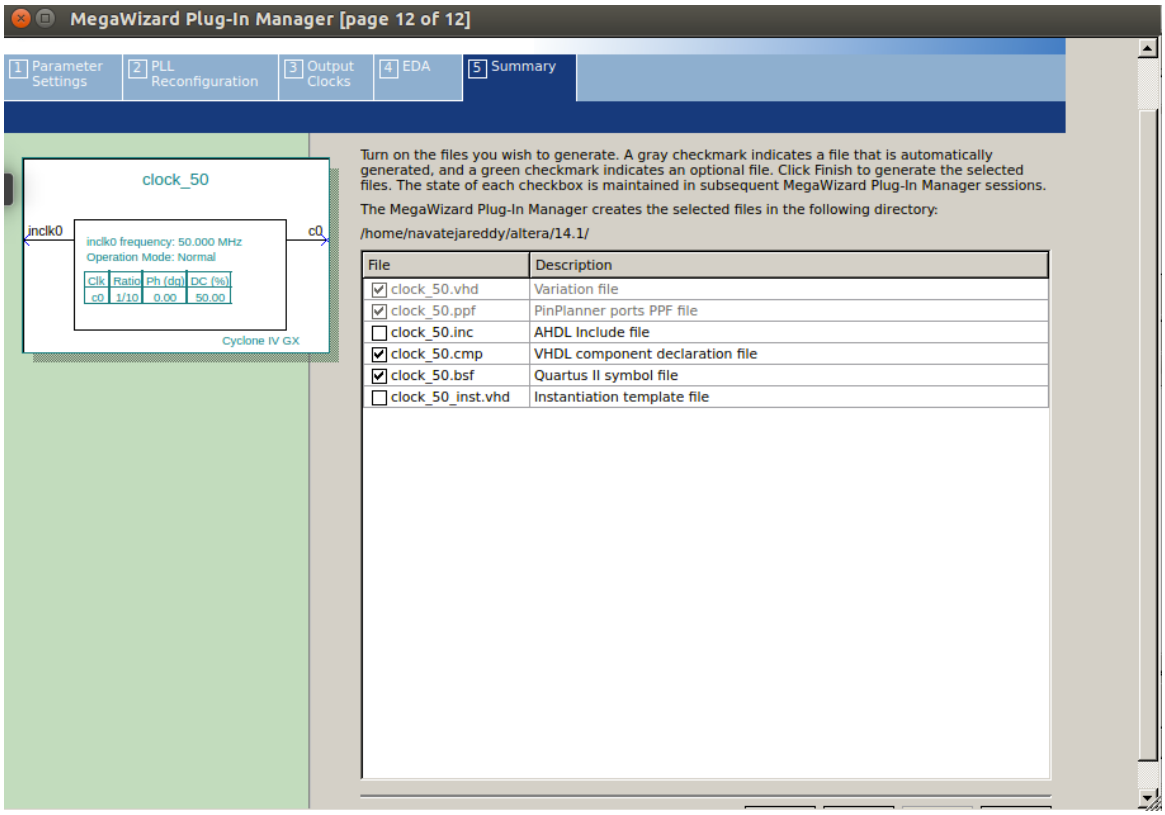


fig 3.1.11

29. Dialogue box appears saying whether to add or not Altera IP variation files, choose Yes. See fig 3.1.12

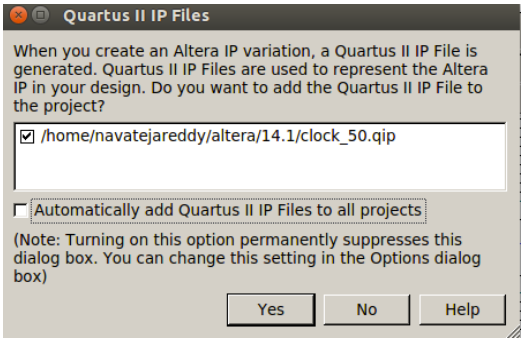


fig 3.1.12

30. Click OK and place the pll symbol onto the BDF to the left of the counter symbol. You can move the symbols around by holding down the left mouse button, helping you ensure that they line up properly.

31. Move the mouse so that the cursor (also called the selection tool) is over the pll symbol's c0 output pin. The orthogonal node tool (cross-hair) icon appears. Click and drag a bus line from the c0 output to the counter clock input. This action ties the pll output to the counter input. Next add bus line using orthogonal node tool to pll inclk0. See fig 3.1.13

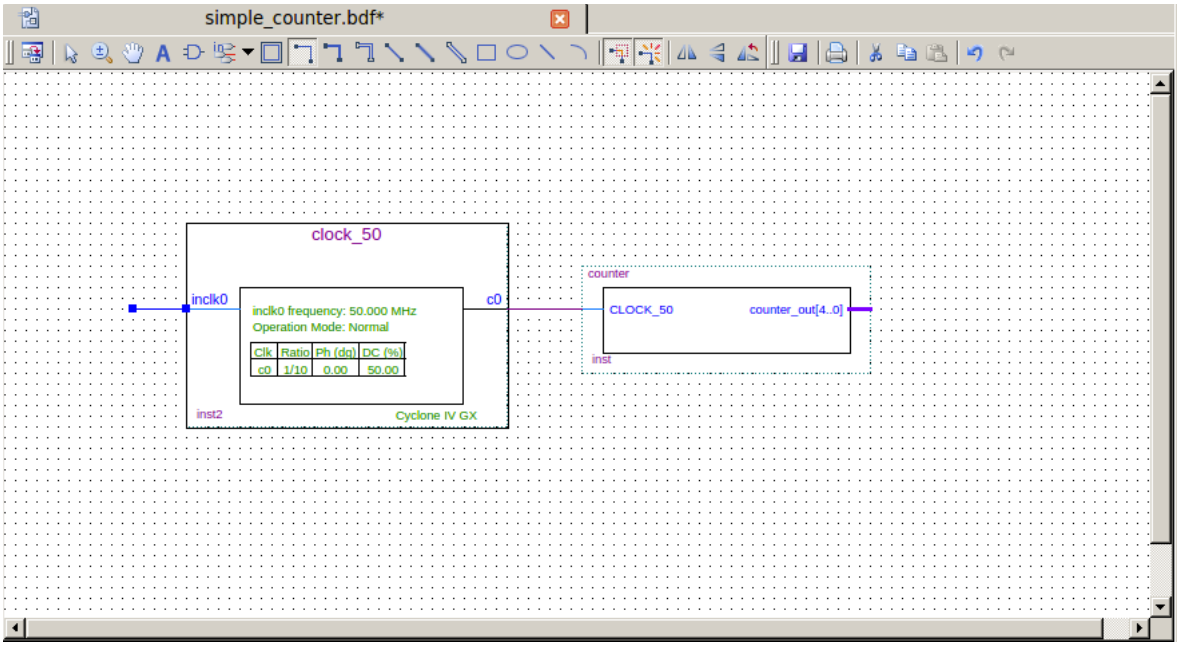


fig3.1.13

32. Add an input pin, output pin and an output bus with the following steps:

- Choose **Edit > Insert Symbol**.
- Under Libraries, select **quartus/libraries > primitives > pin >input**.
- Click **OK**
- Place the new pin onto the BDF so that it is touching the input to the pll symbol. See fig 3.1.14

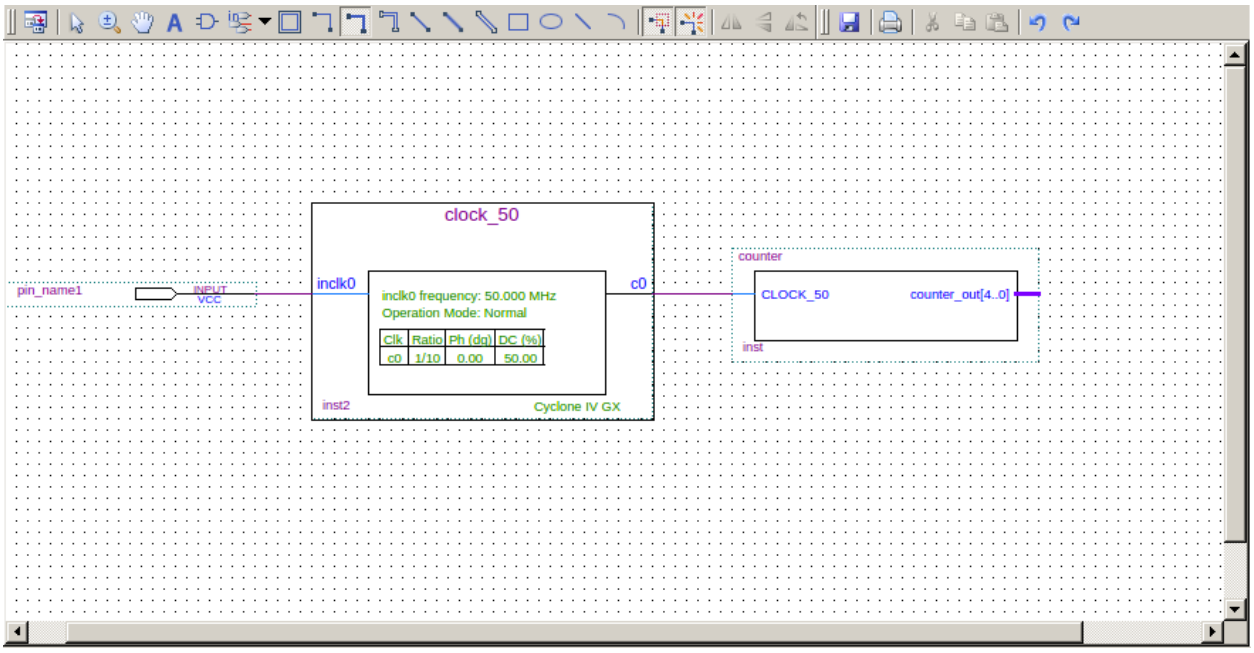


fig 3.1.14

- Using the Orthogonal Bus tool, draw a bus line connected on one side to the counter output port, and leave the other end unconnected.
- Under Libraries, select **quartus/libraries > primitives > pin >output**.
- Place the new pin onto the BDF so that it is touching the output of the counter.
- Rename the input and out pins as follow
  pin_name1 to clock_50 because it correlates to the oscillator clock that is connected to FPGA

pin_name2 to led[0..3]

Note: Naming the multiple pins(pin_name2) in AHDL bus notation(example name[3..0]) or comma separated list of names is mandatory

3.2 Pin Planner

In this section, you will make pin assignments. Before making pin assignments, perform the following steps:

- Under task double click **Analysis and synthesis -> Analysis & Elaboration** or click **Processing -> Start -> Start Analysis & Elaboration** in preparation for assigning pin locations. See fig 3.2.1

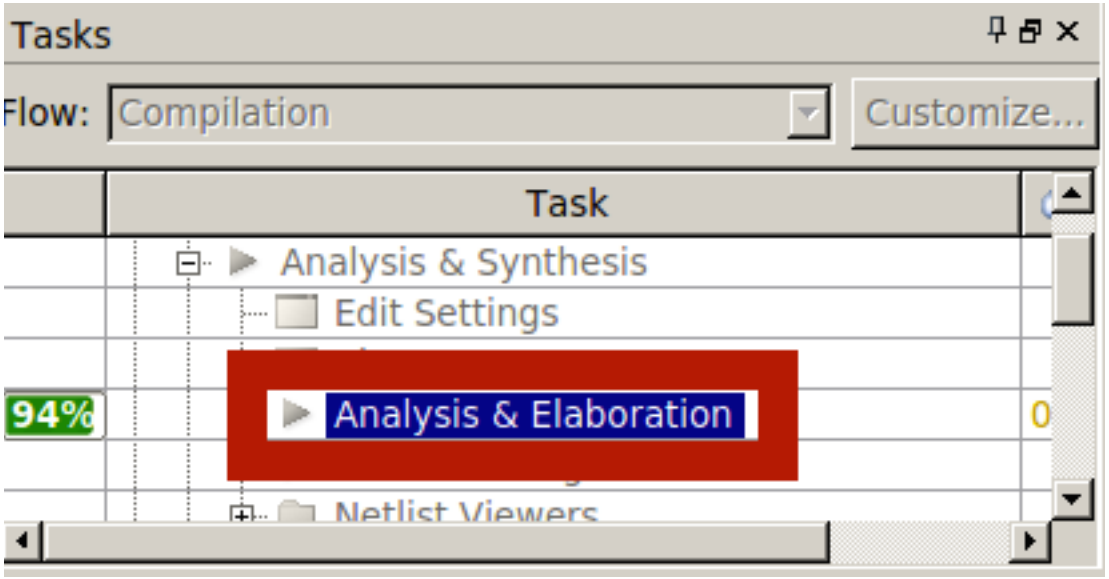


fig 3.2.1

- See fig 3.2.2 for the connections between the LED'S and Cyclone IV FPGA. We are using LEDG0 to LEDG3 and assigning them to the counter output. Choose **Assignments -> Pins**, which opens the Pin Planner, a spreadsheet-like table of specific pin assignments. The Pin Planner shows the design's 5 pins. Assign pins as shown in fig 3.2.3

fig 3.2.2

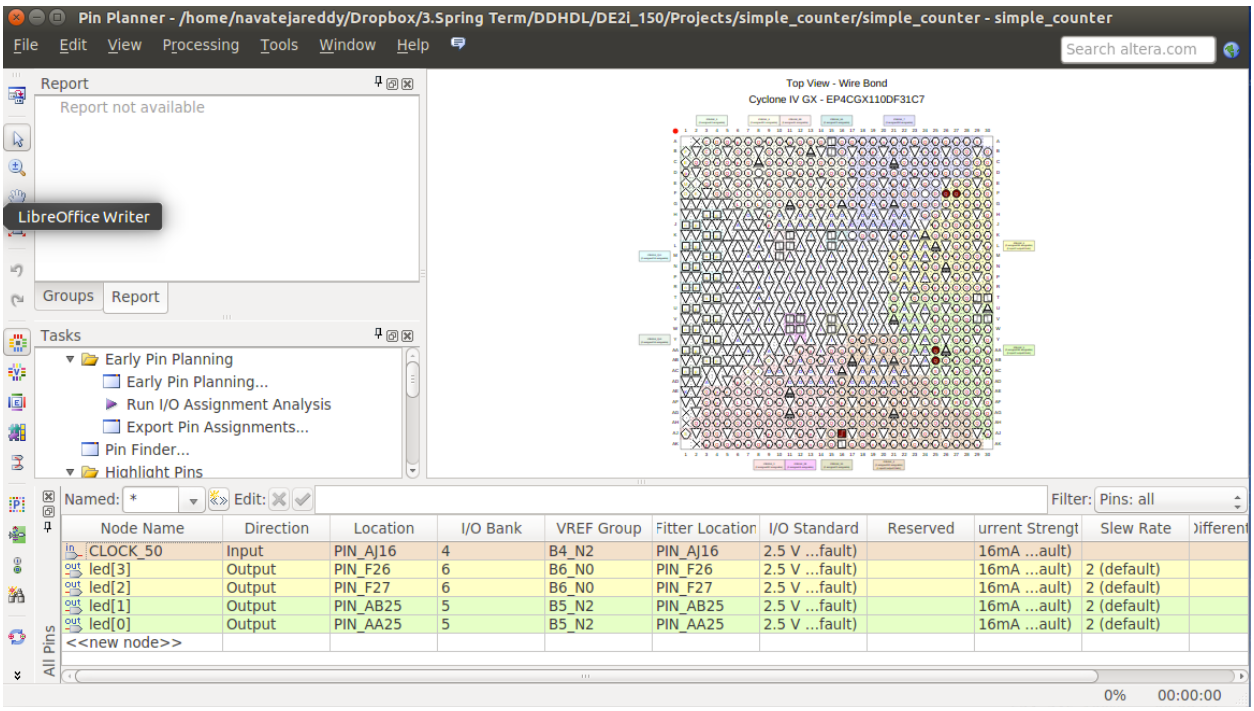


fig 3.2.3

- Double-click in the Location column for any of the six pins to open a drop-down list and type the location shown in the table alternatively, you can select the pin from a drop-down list. For

example, if you type F1 and press the Enter key, the Quartus II software fills in the full PIN_F1 location name for you. The software also keeps track of corresponding FPGA data such as the I/O bank and VREF Group. Each bank has a distinct color, which corresponds to the top-view wire bond drawing in the upper right window. You can also save the programmer state in the file called Chain Description File(.cdf). Go to **File->save** for creating a .cdf file or while closing the programmer, by default a pop up displays either to create .cdf file or not.

Compiling your design

Before compiling the design i want to mention about Synopsys TimeQuest timing analysis which is the part of quartus tool. The Synopsys TimeQuest do the timing analysis of the design and as per quartus II documentation it is mandatory to create TimeQuest SDC file with the same name as top-level module file because if time quest file is not found during the compilation of the deign then the compiler raises critical warning saying

'Critical Warning (332012): Synopsys Design Constraints File file not found: 'simple_counter.sdc'. A Synopsys Design Constraints File is required by the TimeQuest Timing Analyzer to get proper timing constraints. Without it, the Compiler will not properly optimize the design'

As our design is simple counter we can ignore critical warning but for the complex designs TimeQuest analysis is very important.

Now that you have created a complete Quartus II project and entered all assignments, you can compile the design.

In the Processing menu, choose Start Compilation or click the Play button on the toolbar.
If you are asked to save changes to your BDF, click Yes.

While compiling your design, the Quartus II software provides useful information about the compilation

When compilation is complete, the Quartus II software displays a message. Click OK to close the message box.

The Quartus II Messages window displays many messages during compilation. It may display a few warnings that indicate that the device timing information is preliminary or that some parameters on the I/O pins used for the LEDs were not set. The software provides the compilation results in the Compilation Report tab

3.3 Programming the FPGA and Testing

After compiling and verifying your design you are ready to program the FPGA on the development board. You download the SOF you just created into the FPGA using the USB-Blaster circuitry on the board. Set up your hardware for programming using the following steps:

a) Connect the power supply cable to your board and to a power outlet.

b) As instructed in chapter two connect the USB cable between workstation and board after turning on the board.

Program the FPGA using the following steps

- Choose Tools > Programmer. The Programmer window opens
- Click Hardware Setup
- If it is not already turned on, turn on the USB-Blaster option under currently selected hardware
- Click Close
- If the file name in the Programmer does not show simple_counter.sof, click Add File. Select the simple_counter.sof file from the output_files folder in project directory

Verifying the Hardware

Observe that the four development board LEDs appear to be advancing in a binary count pattern, which is driven by the counter bits.

Chapter 4 : PCIE Express on board

4.1 Introduction to Terasic PCIe Framework

The DE2i-150 comes with 2 PCIEx1 lanes by which user can communicate with FPGA from Yocto linux running on the intel processor.

The FPGA PCI Express system is built based on the Altera Qsys builder. In the framework, all hardware controllers are connected with each other through the Avalon Memory-Mapped (Avalon-MM) interface, so master controllers (e.g. DMA) can easily access slave controller (e.g. memory) in a memory-mapped manner. Scatter-Gather DMA (SG-DMA) is included in the frame to provide high-speed data transmission.

Terasic provides the PCIe framework by which users can easily perform basic direct I/O functions for control or data transmission, or high-speed data transmission via DMA. The block digram of PCIe frame work is shown in fig 4.1.1

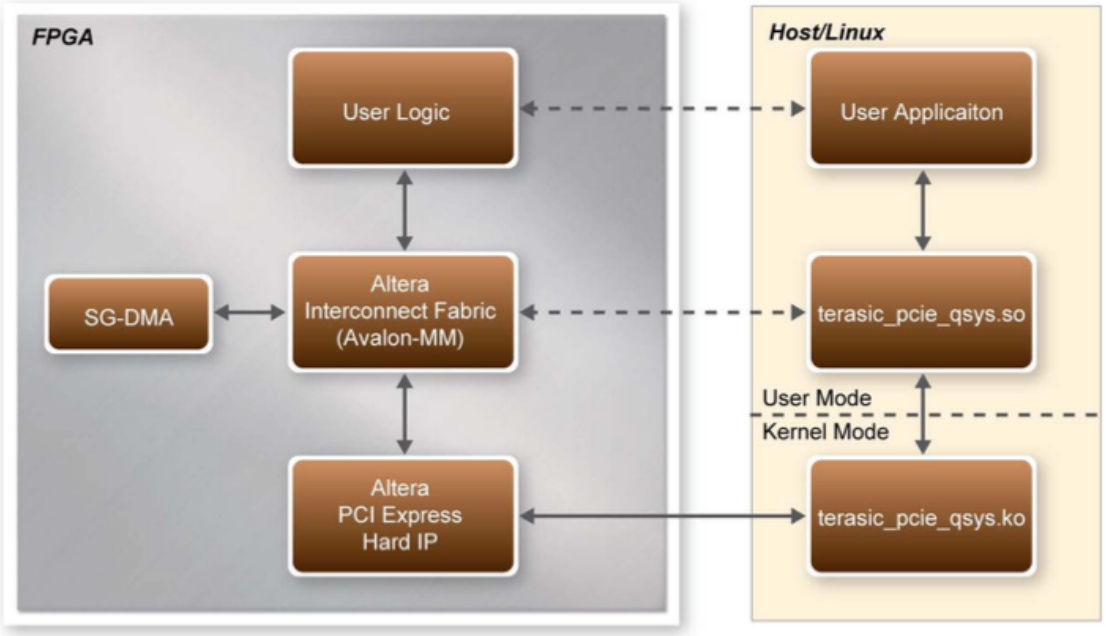


fig 4.1.1 : Terasic PCIe framework

The PCI Express framework consists of two primary parts, the PCI express system in the FPGA and the PCI express software development kit (SDK) in the host computer. The FPGA system is developed based on the Altera Qsys Builder. In a host computer system, the PCI express SDK is provided so the developer can easily design their applications to communicate with the FPGA.

The Terasic PCIe frame work consists of following files:

PCIe Driver files:
- terasic_qsys_pcie.ko: pcie kernel driver
- load_terasic_qsys_pcie_driver.sh: shell script to load kernel driver
- de2i_150_config_file: config file for kernel driver:
- unload_terasic_qsys_pcie_driver.sh: shell script to unload kernel driver

Files for developing the application which runs on Yocto OS:
- TERASIC_PCIE.h
- terasic_pcie_qsys.so
- PCIE.c
- PCIE.h

You can find the above files in the CD provided along with the board under Demonstrations or can be downloaded from the Terasic Website.

The **TERASIC_PCIE.h** defines the required data structure for using **terasic_pcie_qsys.so**. **PCIE.c** and **PCIE.h** implement the function to dynamically load the shared object file **terasic_pcie_qsys.so**

The implementation of PCIe framework is explained in 6th chapters.

4.2 Introduction to Quartus Qsys

In this section i introduce you to Qsys tool basics like how launch and add alter IP's by which you get basic understanding on Qsys tool.

The FPGA PCI Express system is built based on the Altera Qsys builder. With Altera FPGA user can develop complex hardware modules(user logic) and also can use vast Altera hardware IP's and can easily interconnect with user logic using Qsys tool. After opening Quartus II tool Qsys can be launched from **Tools->Qsys**, fig 4.2.1 shows the launch window of Qsys tool.

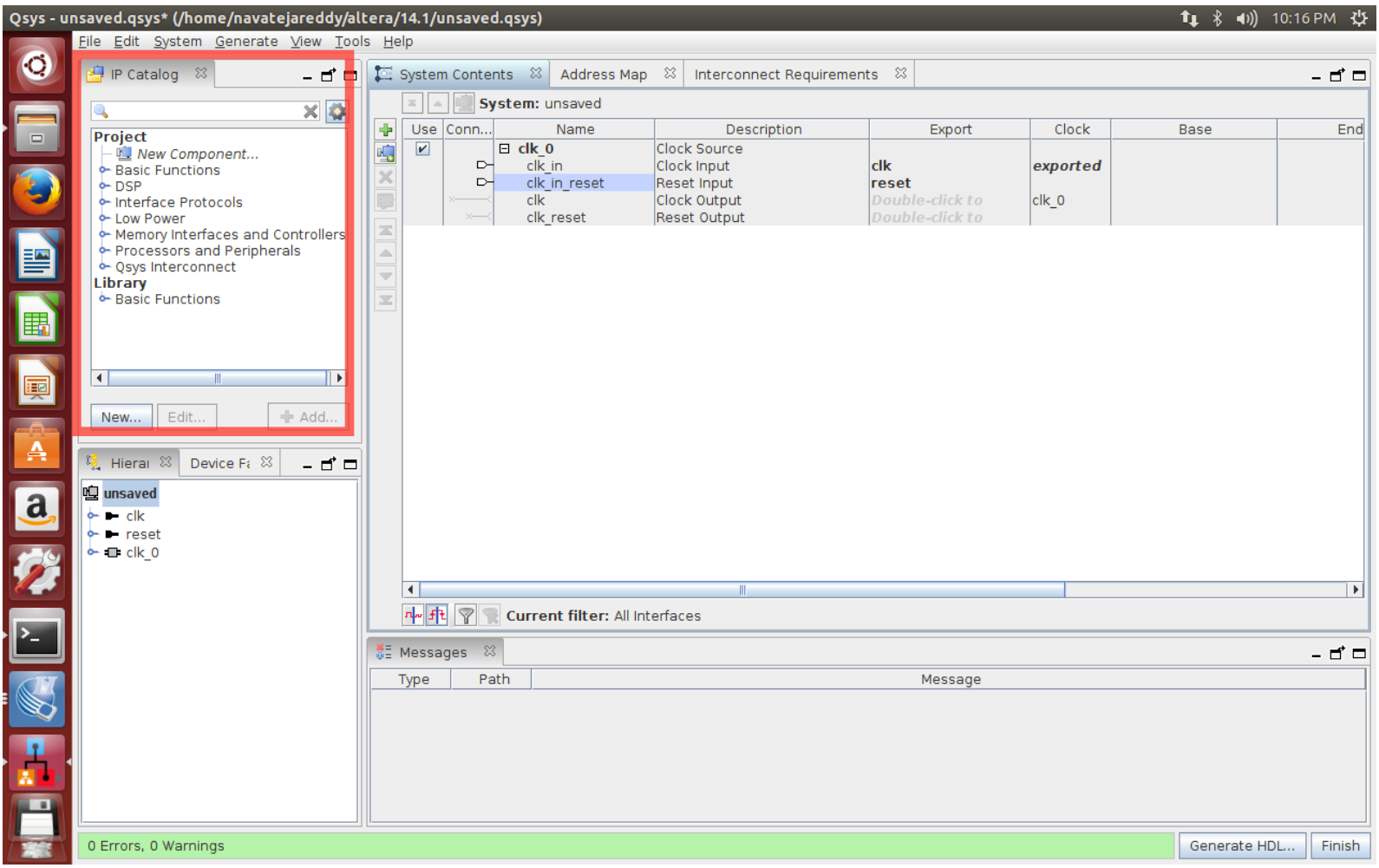


fig 4.2.1 Qsys Interface

Along with user logic and vast Altera IP's you can create a system with Qsys. Altera IP can be added form the IP catalog menu which is located at top left corner as shown in fig 4.2.1. Later after adding the components you can interconnect the components using Avalon interface protocols with easy to use GUI.

For example let us add the 'IP_Compiler for PCI Express' from the altera IP catalog. To add double click or select ADD option after selecting 'IP_Compiler for PCI Express' which you can

found at **Interfaces Protocols->PCI Express** in the IP catalog. It will pop up the menu of 'IP_Compiler for PCI Express' where you can customize, as this is just introduction to the Qsys tool don't bother about customizing the PCIe compiler IP, just click Finish. The Qsys interface looks like as shown in fig 4.2.2 after adding the 'PCI Express Compiler'.

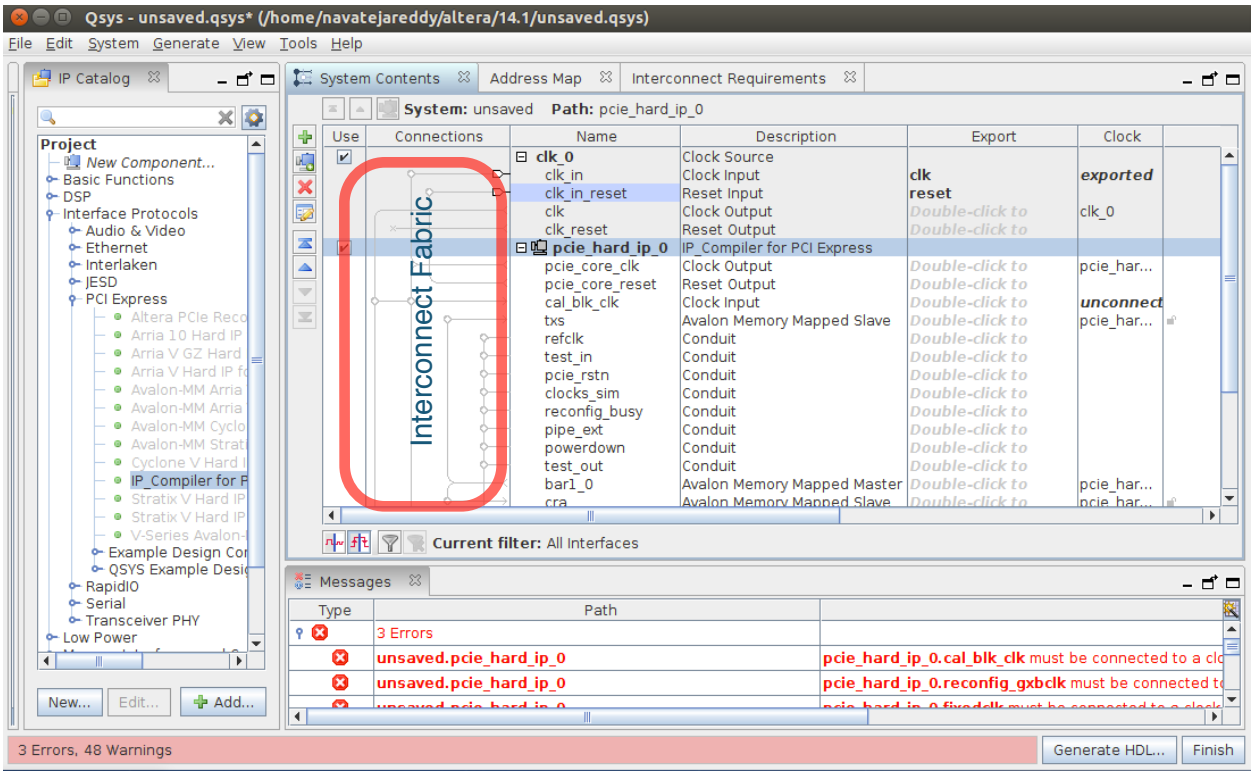


fig 4.2.2

You see errors because the component you added is not supplied with the signals it required. In case of **PCIe_hard_ip_0** it is not provided with the clock signals it required. You can provide the clock from the same component(if clock source is available) or from the other components like **clk_0.** The components can be interconnected using the interconnect fabric. To connect the components just click on the appropriate bubbles under connections tab(left to Name tab).

The errors you notice after adding the IP_Compiler for PCI Express are

unsaved.clk_0	**clk_0.clk_in** must be connected to a clock output
unsaved.pcie_hard_ip_0	**pcie_hard_ip_0.cal_blk_clk** must be connected to a clock output
unsaved.pcie_hard_ip_0	**pcie_hard_ip_0.reconfig_gxbclk** must be connected to a clock output
unsaved.pcie_hard_ip_0	**pcie_hard_ip_0.fixedclk** must be connected to a clock output

You can observe that the errors are due to unconnected clock signals. To remove errors following interconnections should be done

- **clk_0.clk_in** interconnected with **pcie_hard_ip_0.pcie_core_clk**
- **pcie_hard_ip_0.cal_blk_clk** interconnected with **pcie_hard_ip_0.pcie_core_clk** or **clk_0.clk**
- **pcie_hard_ip_0.reconfig_gxbclk** interconnected with **pcie_hard_ip_0.pcie_core_clk** or **clk_0.clk**
- **pcie_hard_ip_0.fixedclk** interconnected with **pcie_hard_ip_0.pcie_core_clk** or **clk_0.clk**

Fig 4.2.3 shows the Qsys tool after fabric interconnections.

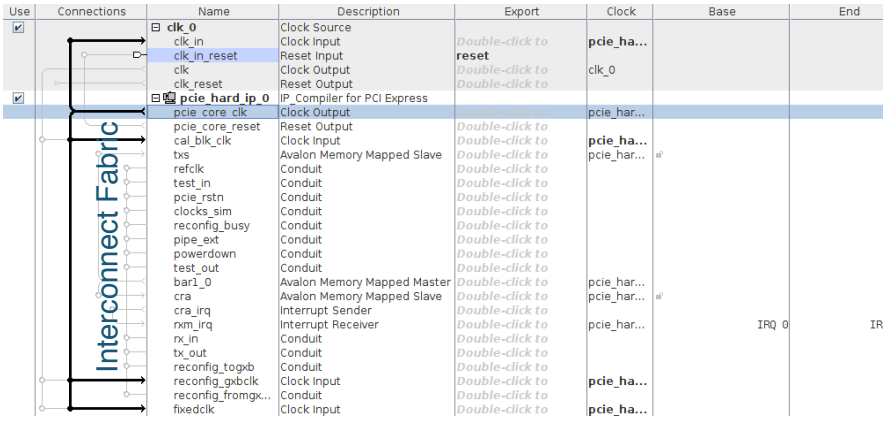


fig 4.2.3

To save the system before closing the tool, from menu click **File->Save** and it saves as a file with extension .qsys

This section is just introductory part to the Qsys tool. To build a perfect system using the Qsys, each component should be configured according to the requirements and all the signals should be interconnected correctly.

4.3 Avalon Interface

Avalon® interfaces simplify system design by allowing you to easily connect components in an Altera® FPGA. The Avalon interface family defines interfaces appropriate for streaming high-speed data, reading and writing registers and memory, and controlling off-chip devices. These standard interfaces are designed into the components available in Qsys. You can also use these standardized interfaces in your custom components. By using these standard interfaces, you enhance the interoperability of your designs.

This specification defines all of the Avalon interfaces. After reading it, you should understand which interfaces are appropriate for your components and which signal roles to use for particular behaviors. This specification defines the following seven interfaces:

- Avalon Streaming Interface (Avalon-ST)—an interface that supports the unidirectional flow of data, including multiplexed streams, packets, and DSP data.
- Avalon Memory Mapped Interface (Avalon-MM)—an address-based read/write interface typical of master–slave connections.
- Avalon Conduit Interface— an interface type that accommodates individual signals or groups of signals that do not fit into any of the other Avalon types. You can connect conduit interfaces inside a Qsys system. Or, you can export them to make connections to other modules in the design or to FPGA pins.
- Avalon Tri-State Conduit Interface (Avalon-TC) —an interface to support connections to off-chip peripherals. Multiple peripherals can share pins through signal multiplexing, reducing the pin count of the FPGA and the number of traces on the PCB.
- Avalon Interrupt Interface—an interface that allows components to signal events to other components.
- Avalon Clock Interface—an interface that drives or receives clocks.
- Avalon Reset Interface—an interface that provides reset connectivity.

Learn more about Avalon Interface specifications here.

Chapter 5 : Simple application : Demonstration of communication between CPU and FPGA using Terasic PCIE Fundamentals design

---

5.1 Setting up TFTP server on workstation for file transfer

Before proceeding forward let us configure a TFTP daemon on the workstation. This enables you to quickly transfer files from the development environment to the DE2i-150.

Note : There are many other alternatives to transfer files between workstation and board.

Install TFTP on the workstation using following command in Linux terminal(Workstation) :

```
navatejareddy@ubuntu:~$ sudo apt-get install xinetd tftp tftpd
```

We'll use the directory `/tftpboot` on the workstation as the transfer point. Add this configuration in the TFTP configuration file `/etc/xinet.d/tftp`

```
service tftp
{
protocol = udp
port =69
socket_type = dgram
wait = yes
user = nobody
server = /usr/sbin/in.tftpd server_args = /tftpboot
disable =no
}
```

The tftp directory in /tftpboot needs to belong to user nobody, and have 777 permissions.

Change ownership to nobody using following command in workstation terminal

```
navatejareddy@ubuntu:/$ sudo chown nobody tftpboot
```

Change permission to 777 using following command in workstation terminal

```
navatejareddy@ubuntu:/$ sudo chmod 777 tftpboot
```

Now everything is configured on the workstation, now connect ethernet cable between the workstation and the board. Location of Ethernet port on the board is shown in fig 5.1.1

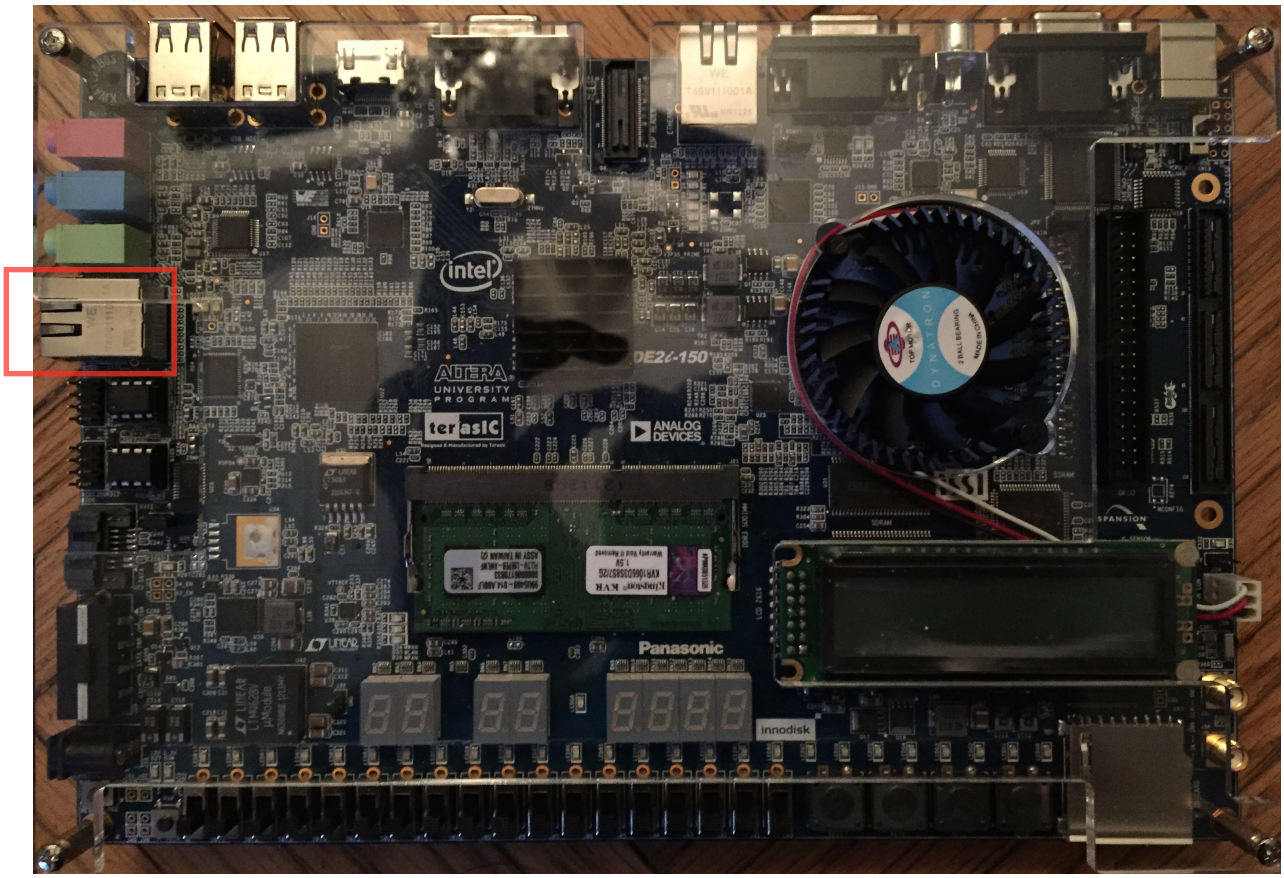


fig 5.1.1

After connecting the ethernet cable, the green led light(first LED behind the Ethernet port) should glow steadily, if not the TFTP is not working properly and following may be the reasons and solutions

- Network settings on the board may not have been load properly, reset network settings using following command `sudo /etc/init.d/networking restart` in yocto terminal or reboot the board using `reboot` command in yocto terminal

- **eth0** setting may not have been configured on the host, add **eth0** setting in `/etc/network/interfaces` and restart the network using following command

```
sudo /etc/init.d/networking restart
```

- If you are running Ubuntu on the virtual machine (VMWare Fusion), first change the Network adapter setting in VMWare Fusion to Ethernet under Bridged Networking and add **eth0** setting in `/etc/network/interfaces` on the workstation and restart the virtual machine.

Learn more about how networking works on Virtual machine here
After everything setup,

To transfer a file from the workstation to DE2i-150, copy the file in /tftpboot and use the following command on DE2i- 150:

```
tftp -g -r <File Name> <ipaddress of workstation>
```

5.2 Glance at PCIE Fundamentals and Programming the FPGA

PCIE Fundamentals is the Quartus II project provided by the Terasic to understand the basics of communication between CPU system and FPGA system using PCIE channel, SG-DMA and Avalon Interface. You can find the project files in the CD provided along with the board under `Demonstrations/FPGA`. The design shows how to implement fundamental control and data transfer. In the design, direct I/O is used to access the BUTTON and LED on the FPGA board. High-speed data transfer is performed by SG-DMA on both On-Chip Memory and On-Chip FIFO.

From CD copy the folder `Demonstrations/FPGA/PCIE_Fundamental` to your desired location on to the workstation and open the project from Quartus II. After opening the project, by selecting Hierarchy Tab under project navigator you notice that `de2i_150_qsys_pcie` as top level module in the design as shown in fig 5.2.1

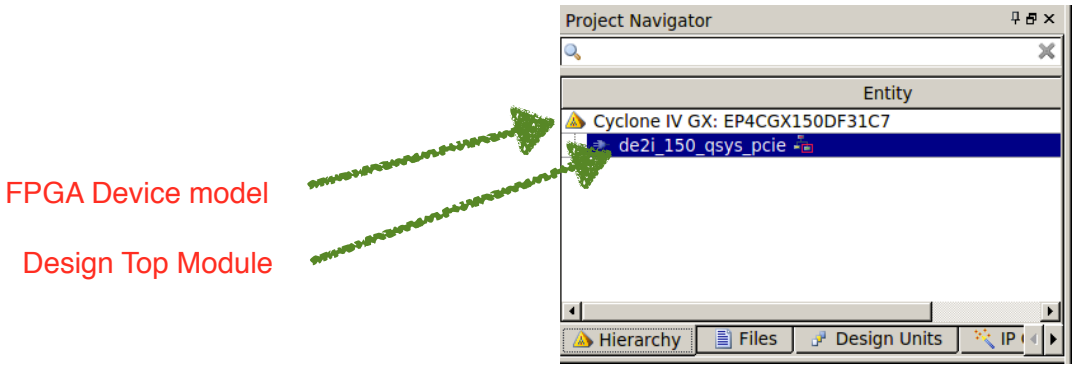
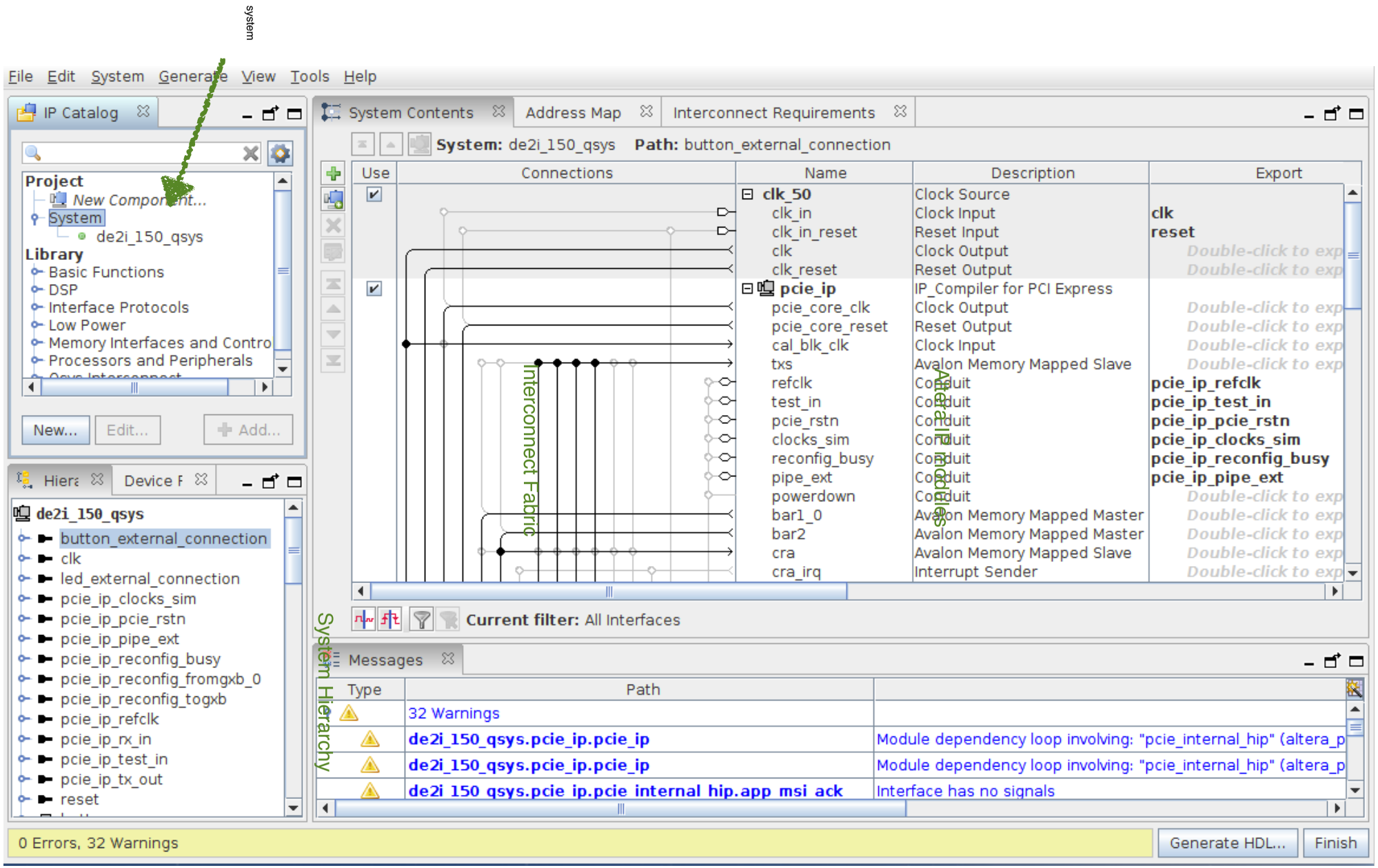


fig 5.2.1

Next open Qsys by going to **tools->Qsys**, it prompts to select the file with .qsys extension(Qsys system files) in which the subsystem designed using Qsys for this project is saved. After opening the Qsys file the developed system opens as shown in below figure.


system
File Edit System Generate View Tools Help
IP Catalog
Project
New Component...
System
de2i_150_qsys
Library
Basic Functions
DSP
Interface Protocols
Low Power
Memory Interfaces and Contro
Processors and Peripherals
New...
Edit...
Add...
Hiera
Device F
de2i_150_qsys
button_external_connection
clk
led_external_connection
pcie_ip_clocks_sim
pcie_ip_pcie_rstn
pcie_ip_pipe_ext
pcie_ip_reconfig_busy
pcie_ip_reconfig_fromgxb_0
pcie_ip_reconfig_togxb
pcie_ip_refclk
pcie_ip_rx_in
pcie_ip_test_in
pcie_ip_tx_out
reset
System Contents
Address Map
Interconnect Requirements
System: de2i_150_qsys Path: button_external_connection
Use
Connections
Name
Description
Export
clk_50
Clock Source
clk_in
Clock Input
clk
clk_in_reset
Reset Input
reset
clk
Clock Output
clk_reset
Reset Output
pcie_ip
IP_Compiler for PCI Express
pcie_core_clk
Clock Output
pcie_core_reset
Reset Output
cal_blk_clk
Clock Input
txs
Avalon Memory Mapped Slave
refclk
Conduit
pcie_ip_refclk
test_in
Conduit
pcie_ip_test_in
pcie_rstn
Conduit
pcie_ip_pcie_rstn
clocks_sim
Conduit
pcie_ip_clocks_sim
reconfig_busy
Conduit
pcie_ip_reconfig_busy
pipe_ext
Conduit
pcie_ip_pipe_ext
powerdown
Conduit
bar1_0
Avalon Memory Mapped Master
bar2
Avalon Memory Mapped Master
cra
Avalon Memory Mapped Slave
cra_irq
Interrupt Sender
Double-click to exp
Interconnect Fabric
Avalon Modules
Current filter: All Interfaces
System Hierarchy
Messages
Type
Path
32 Warnings
de2i_150_qsys.pcie_ip.pcie_ip
Module dependency loop involving: "pcie_internal_hip" (altera_p
de2i_150_qsys.pcie_ip.pcie_ip
Module dependency loop involving: "pcie_internal_hip" (altera_p
de2i_150_qsys.pcie_ip.pcie_internal_hip.app_msi_ack
Interface has no signals
0 Errors, 32 Warnings
Generate HDL...
Finish

The **de2i_150_qsys** system is build using altera IP modules. The Altera IP modules used are

- **Clock Source** - named as clk_50 and used as clock source
- **IP_Compiler for PCI Express** - named as pcie_ip, compiler for altera's PCI hard IP
- **Scatter-Gather DMA Controller** - named as sg-dma and used as DMA controller
- **Nios II(classic)processor** - named as nois2, Altera's softcore processor
- **PIO(Parallel I/O)** - named as led and buttons, connected to FPGA
- **Avalon FIFO Memory** - named as fifo_memory, On-chip FIFO memory
- **On-Chip Memory(RAM or ROM)** - named as onchip_memory, On-chip RAM

You can click on the names to see its properties.

Click on **Address Map** to see the address mappings of the modules. To access the module from the host, these address are used i.e CPU system and FPGA system share the common address space over PCIe. For example from host if you write data to address 0x0000_0020, the data get writes to the module which is having that address.

Now let us test the design. Before closing the tool, you need to generate synthesis file for the developed Qsys system, from menu go to **Generate -> Generate HDL**, it will open a window as shown in figure 5.2.2

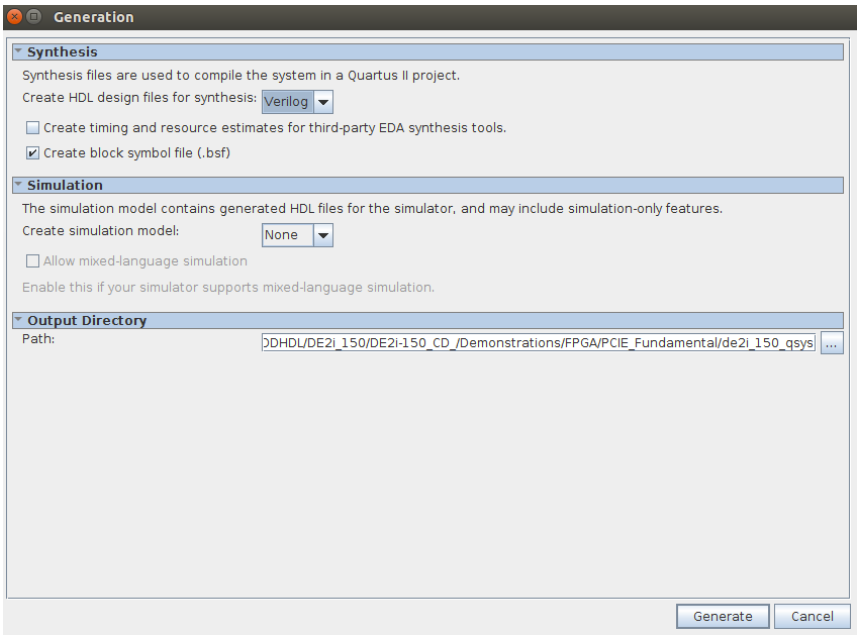


fig 5.2.2

You can select **Verilog** or **VHDL**, let us select Verilog and click **Generate**. After code is generated close and compile the design as explained in section 3.3

After compiling next step is programming the FPGA, program the FPGA as explained in section 3.3 .

5.3 Copying app and PCIE Drivers and loading the PCIE drivers

We have already configured the TFTP server, so now copy the following files from work station to host

PCIe Driver files: which can be find on CD at `Demonstrations/PCIe_SW_KIT/linux/PCIe_DriverInstall`

- terasic_qsys_pcie.ko: pcie kernel driver
- load_terasic_qsys_pcie_driver.sh: shell script to load kernel driver
- de2i_150_config_file: config file for kernel driver:
- unload_terasic_qsys_pcie_driver.sh: shell script to unload kernel driver

Application files which runs on Yocto OS: which can be find on CD at `Demonstrations/PCIE_Fundamental/linux_app`

- terasic_pcie_qsys.so
- app.c

To copy files to host, run the following command in the Yocto OS terminal

```
tftp -g -r <File Name> <ipaddress of workstation>
```

Note : All files should be copied to /tftpboot on workstation before executing the above command

When you run the above command the files get copied to root directory of the host.

Now before loading the PCIe drives on host, restart the host by executing `reboot` command in yocto terminal so that atom processor sees the right device on the FPGA.

After rebooting the host load the drivers by running the load_terasic_qsys_pcie_driver.sh shell script file. To run the shell script file, simply execute the following command in yocto terminal

```
sh load_terasic_qsys_pcie_driver.sh
```

To verify drivers are loaded are not run the following command in Yocto terminal

```
lsmod
```

You should see output not same but similar to this

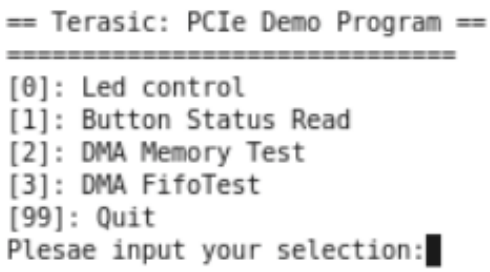
```
Module                  Size                Used by
terasic_driver          2158                    0
```

5.4 Running the application and results

After loading the drivers the last step is running the application. Application files are already copied to host so to run the app simply execute the following command in yocto terminal

```
./app
```

You see the following output



Select the desired function to test.

Led control is writing data to LEDG0 to LEDG3 connected to FPGA

Button Status Read is reading data from Push switches(key0-key3) connected to FPGA. Switches are active high i.e binary 1 by default so when you select this option without pushing any switch you see 1111 as output and to make the switch active low i.e binary 0 push the switch/switches.

DMA memory Test is Reading data from on chip RAM

DMA FifoTest is Reading data from on chip FIFO

Chapter 6 : Counter design and interconnecting with PCI Hard IP using Qsys

---

6.1 Designing the user logic

We have seen how to program FPGA with custom circuitry and Introduction of Qsys interconnect tool to use and implement the functionalities of the PCIe system on the board. Now in this chapter we will design a 8 bit counter and interconnect it with altera's PCIe hard IP using Qsys tool. Later we will develop app using Terasic PCIe framework which runs on Yocto OS to read the counters 8 bit output.

Open the Quartus II tool by double clicking the Quartus II icon on the desktop. Create a project with name **qsys** as explained in section 3.1-creating a project. Follow the below steps to create counter design

Step 1: After creating project with your own name, click **File -> New**, a window will open, then select a new verilog file under Design files and click OK. This will create a new verilog file and save it as **counter.v** .

Step 2 : Add the following verilog code to the file

```
module qsys(
out_put,
clk,
rst
);
output [7:0] out_put;
input clk;
input rst;
reg [7:0] out_put;
always@(posedge clk or posedge rst)
begin
 if(rst)
  out_put <= 8'b00000000  ;
 else
  out_put <= #1 out_put+1'b1;
end
endmodule
```

save the file by clicking **File->Save**.

Step 3: Before moving forward let us verify the counter.v for any errors, to do so click **Processing -> Start -> Start Analysis & Elaboration**. If there are any errors you can see details in message Tab under Processing.

6.2 Developing system using Qsys

We completed designing the counter the next step is interconnecting the counter to PCIe hard IP using Qsys tool.

Launch Qsys tool by clicking **Tools -> Qsys**. After Qsys is launched you see **clk_0** component by default in splash window, remove that component by selecting the component and click **X** mark as shown in fig 6.2.1

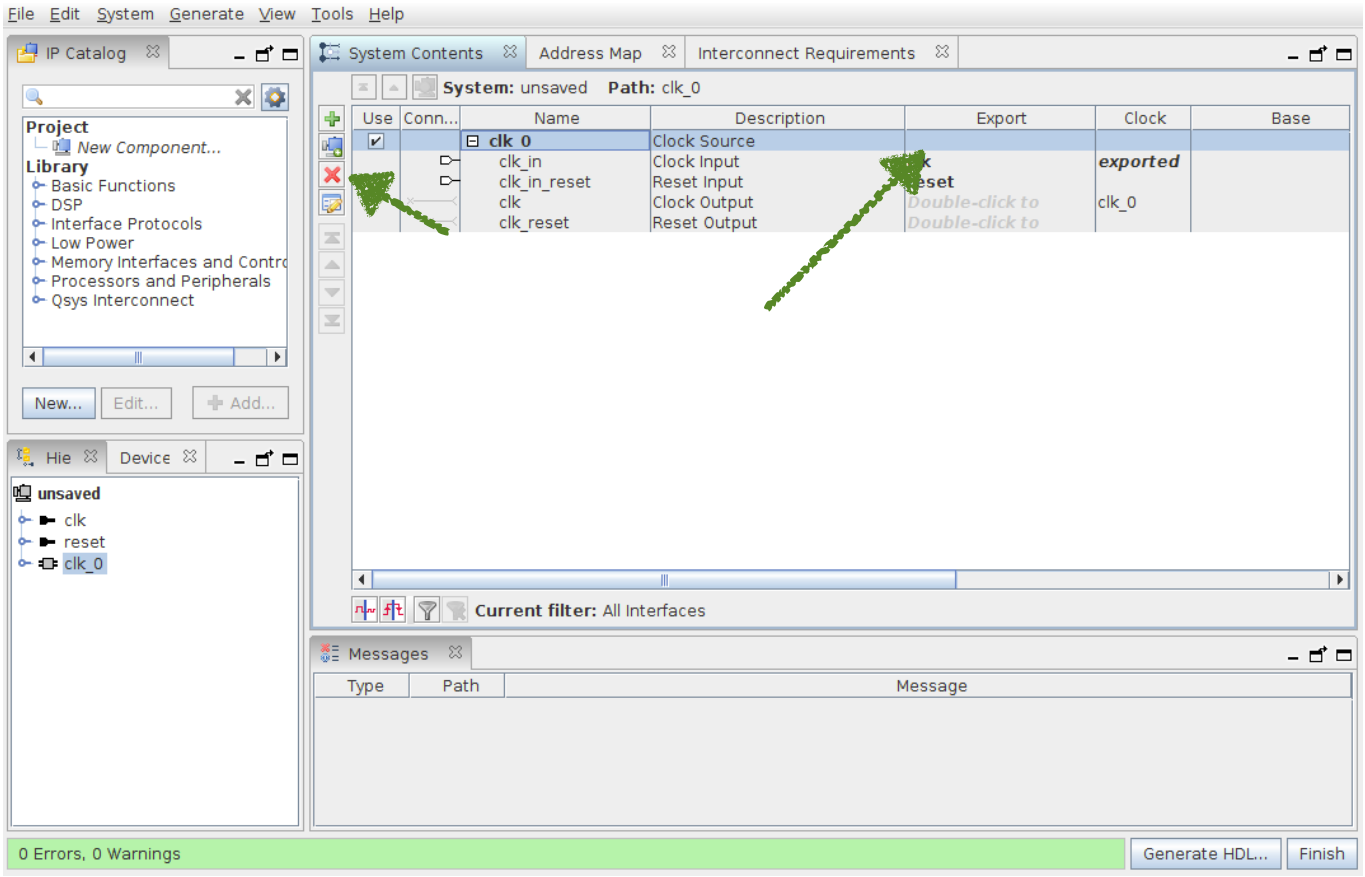


fig 6.2.1

Now in IP Catalog under **Interface Protocols -> PCI Express** select **IP_Compiler for PCI Express** and click **+ Add** for adding it to system as shown in fig 6.2.2

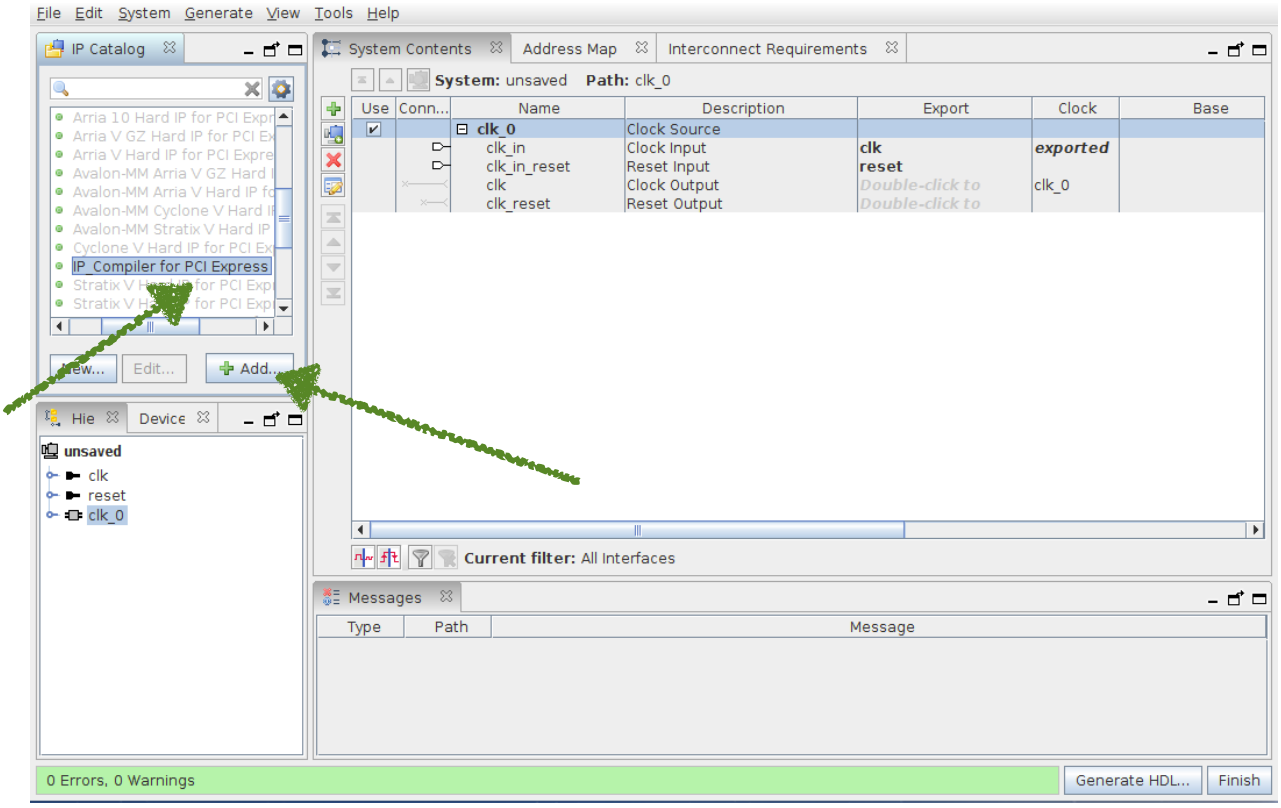


fig 6.2.2

After clicking **+ADD**, Window with **IP_Compiler for PCI Express** properties will be opened. Under **PCI Base Address Registers** Change **BAR Type** to **32-bit Non- prefetchable** and click **Finish**, the Qsys window looks like as shown in fig 6.2.3

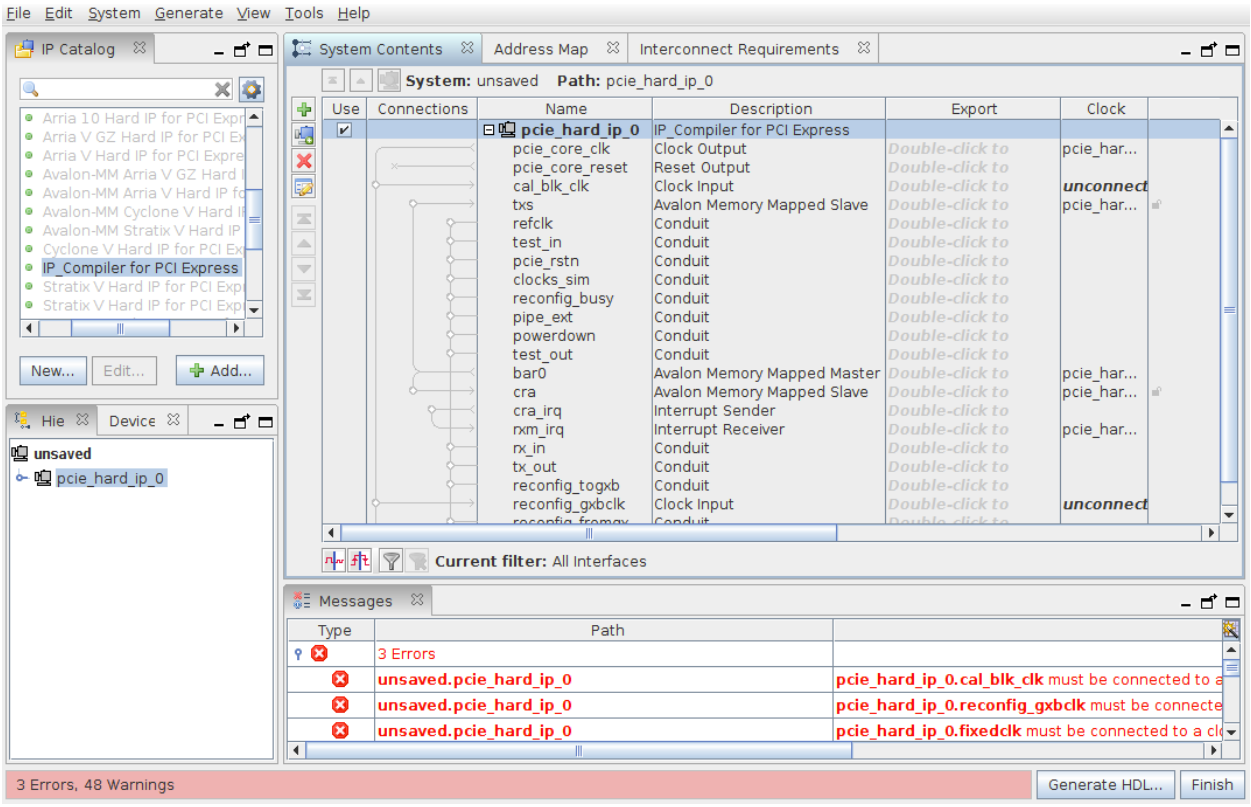


fig 6.2.3

You see errors because of un-supplied signals to the **pcie_hard_ip_0** component. We will remove those errors in coming steps.

Now let us add the counter we designed to the system, to do so click **New** in IP Catalog menu. Component editor will be open as show in fig 6.2.4

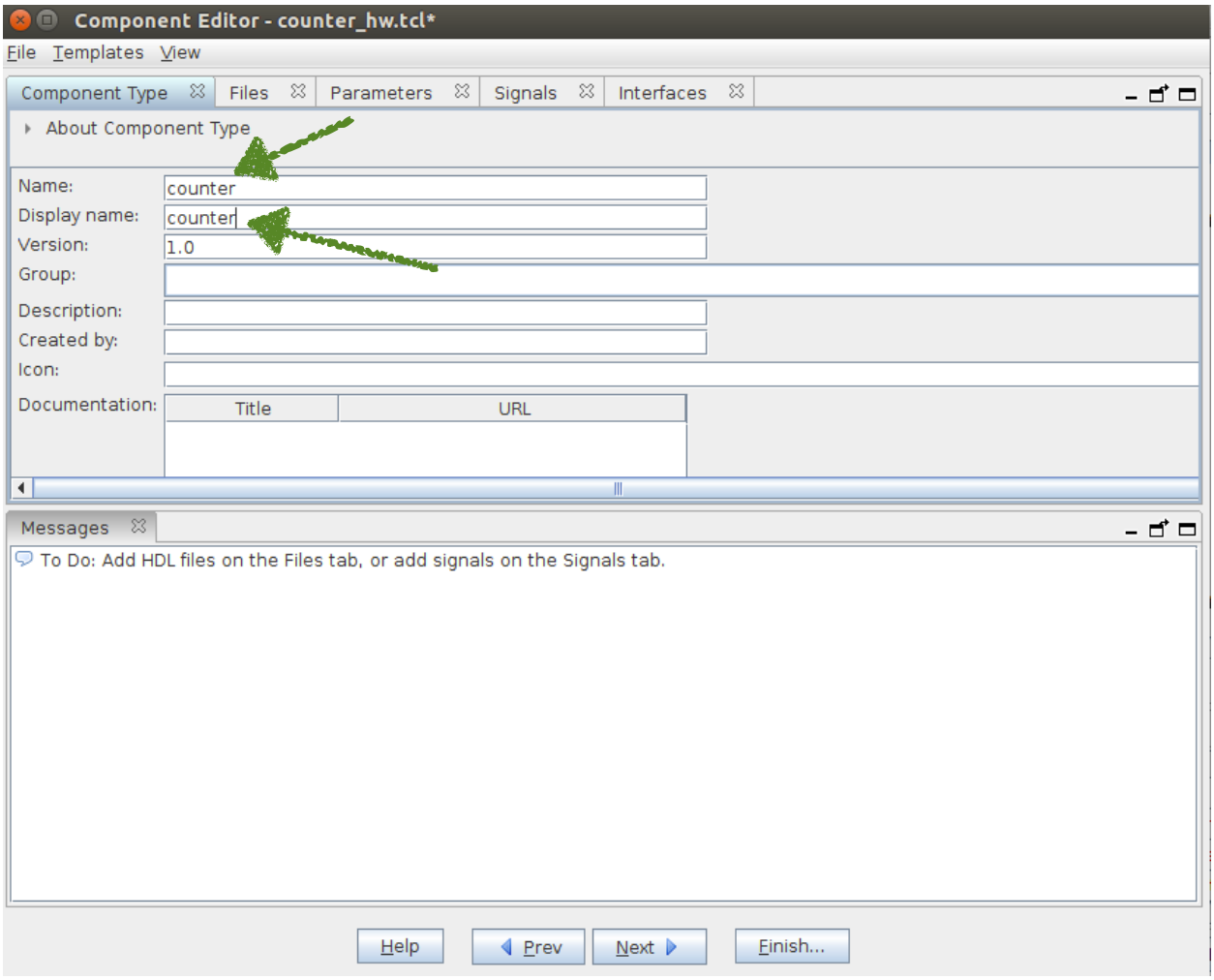


fig 6.2.4

Change **Name** and **Display name** to **counter** as shown in fig 6.2.4. Next select Files tab and add counter.v under Synthesis Files as shown in fig 6.2.5. After adding counter.v click Analyze Synthesis Files as shown in fig 6.2.5

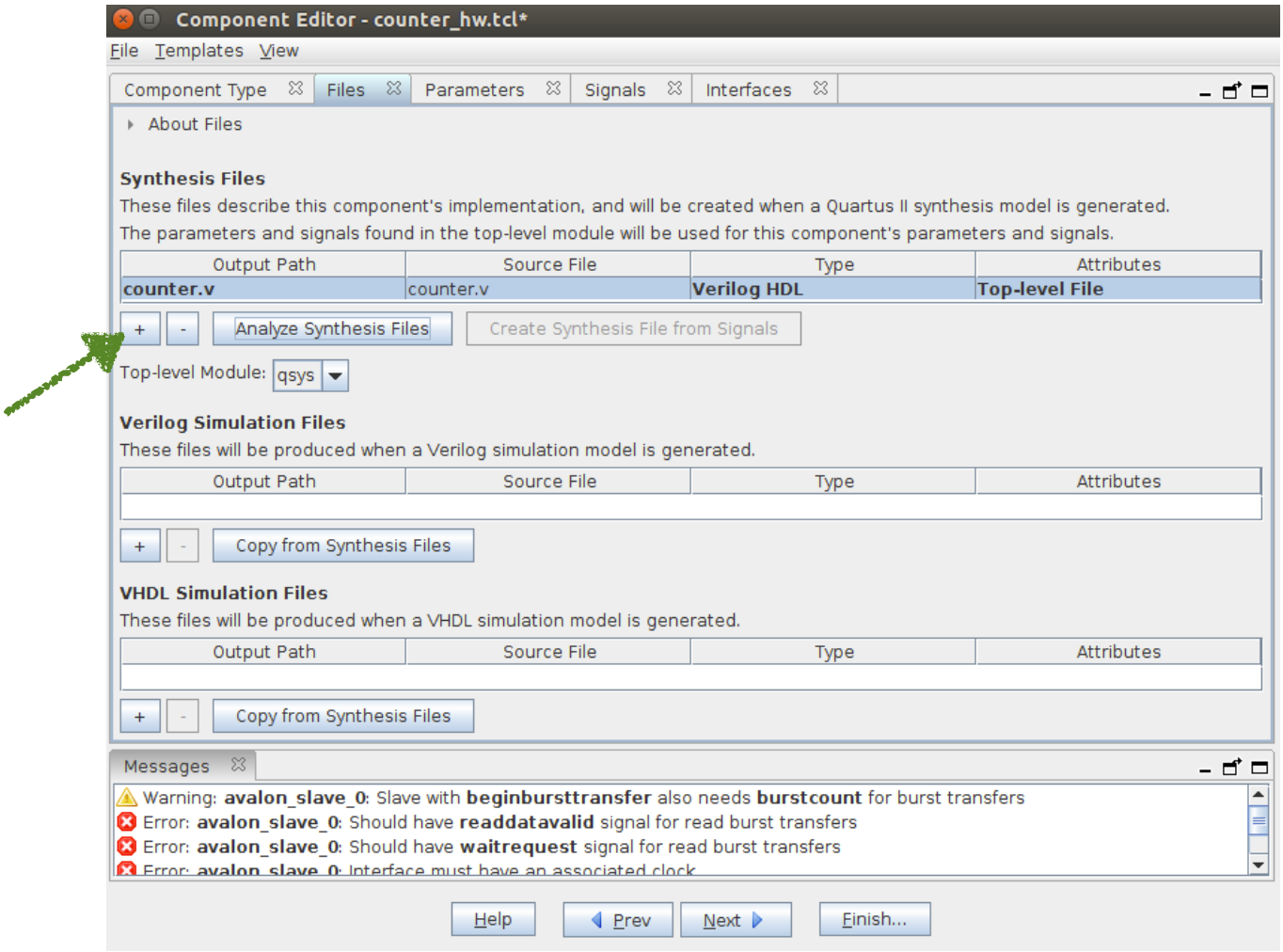


fig 6.2.5

After analysis you see errors due to wrong signal parameters and interfaces . To remove errors due to signal parameters click Signals tab as shown in fig 6.2.6 and change signal interface of **rst** to **new reset input** from I**nterface** dropdown menu.

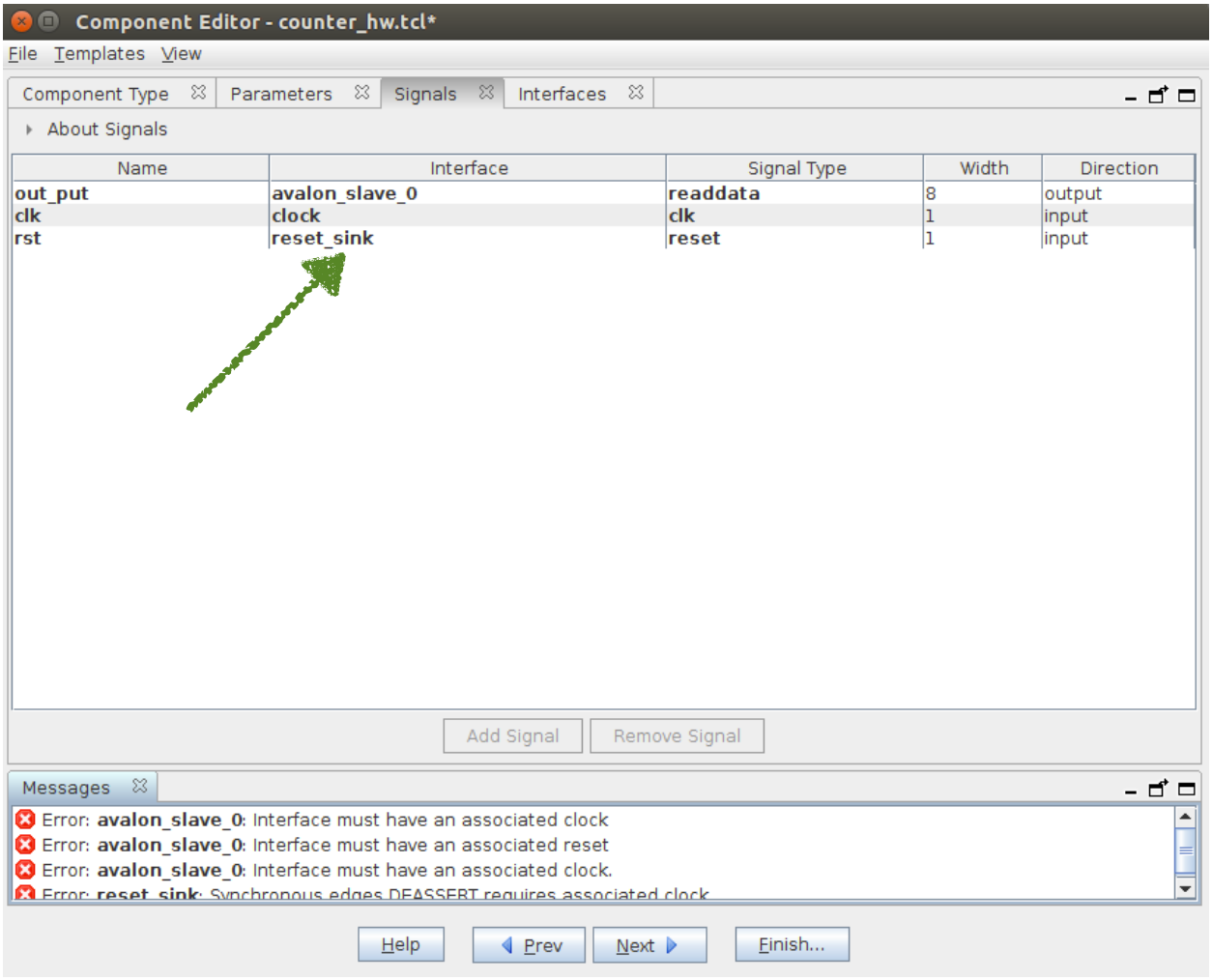
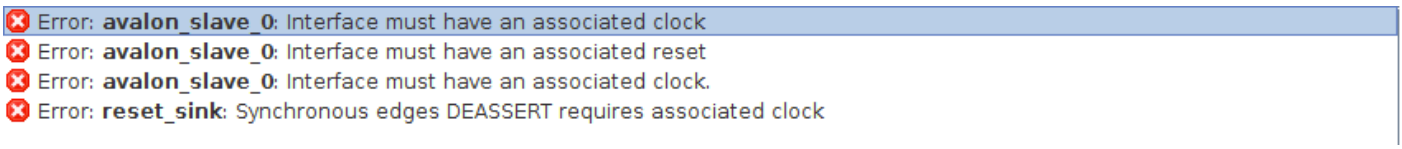


fig 6.2.6

After changing parameters now you see the following errors



The first three errors can be removed by assigning the component with signals it required and the fourth error can be removed by changing reset_sink to Asynchronous which can be done in Interfaces tab.

After selecting Interfaces Tab change **Associated Clock** to **clock** and **Associated Reset** to **reset_sink** of **avalon_slave_0** interface as shown in fig 6.2.7

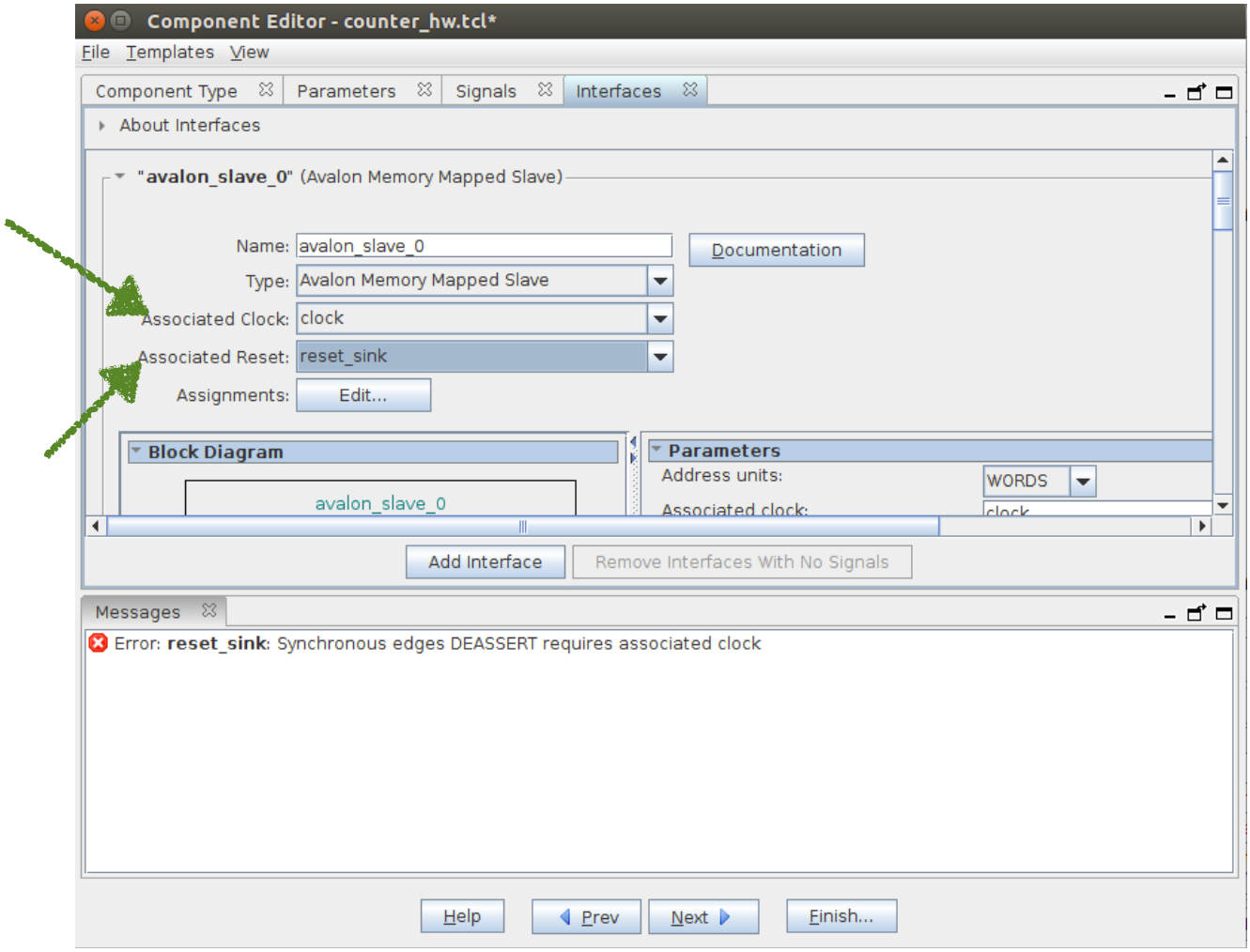


fig 6.2.7

After changing the **avalon_slave_0** interface signals you see only one error of reset_sink interface. To change reset sink to Asynchronous, change the Synchronous edges to **None** as shown in fig 6.2.8.

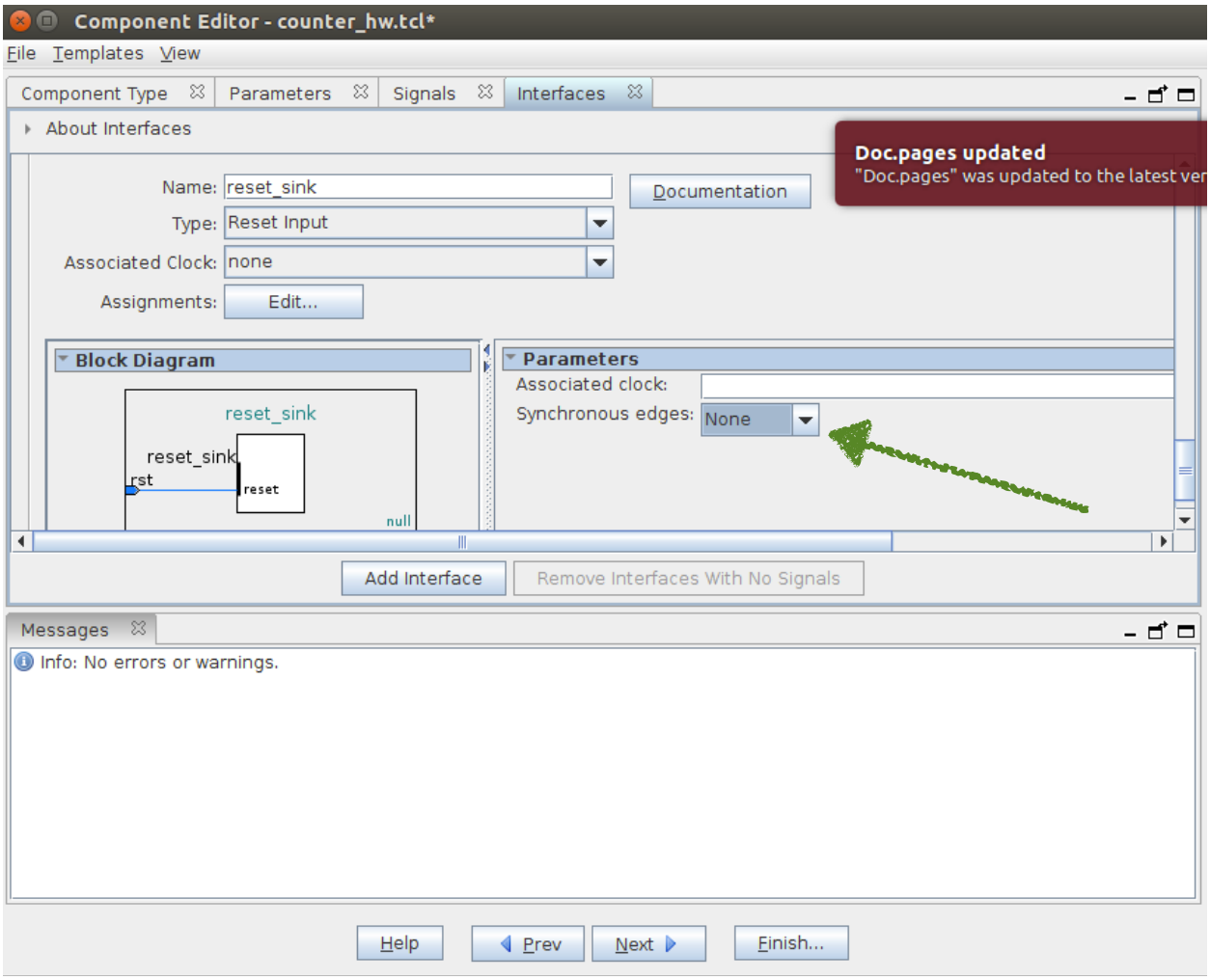


fig 6.2.8

Now you are done with configuring counter, click finish to create the component. You can find the created component in IP Catalog under Project. Add the counter to the system in the same way you added the **pcie_hard_ip_0**.

You see the following errors due to un-supplied signals to the components. This errors can be removed by interconnecting the components with signals they require using interconnect fabric. Interconnect the components as explained in 4.2. Along with clock and reset signals interconnect PCI component **cra** and **txs** to **bar0 . cra** and **txs** are PCIe control signals.

⊗	**unsaved.pcie_hard_ip_0**	**pcie_hard_ip_0.cal_blk_clk** must be connected to a clock output
⊗	**unsaved.pcie_hard_ip_0**	**pcie_hard_ip_0.reconfig_gxbclk** must be connected to a clock output
⊗	**unsaved.pcie_hard_ip_0**	**pcie_hard_ip_0.fixedclk** must be connected to a clock output
⊗	**unsaved.counter_0**	**counter_0.clock** must be connected to a clock output
⊗	**unsaved.counter_0**	**counter_0.reset_sink** must be connected to a reset source

After interconnecting the signals you see errors due to address over lapping. We can change the addresses in **Address Map** tab. To terasic PCIe framework to work the base address of **cra** and **txs** are fixed to some specific value, so before moving to Address Map tab change the base address of **cra** and **txs** to **0x0000_0000** and **0x8000_0000** respectively under B**ase** in S**ystem Contents** tab as shown in fig 6.2.9

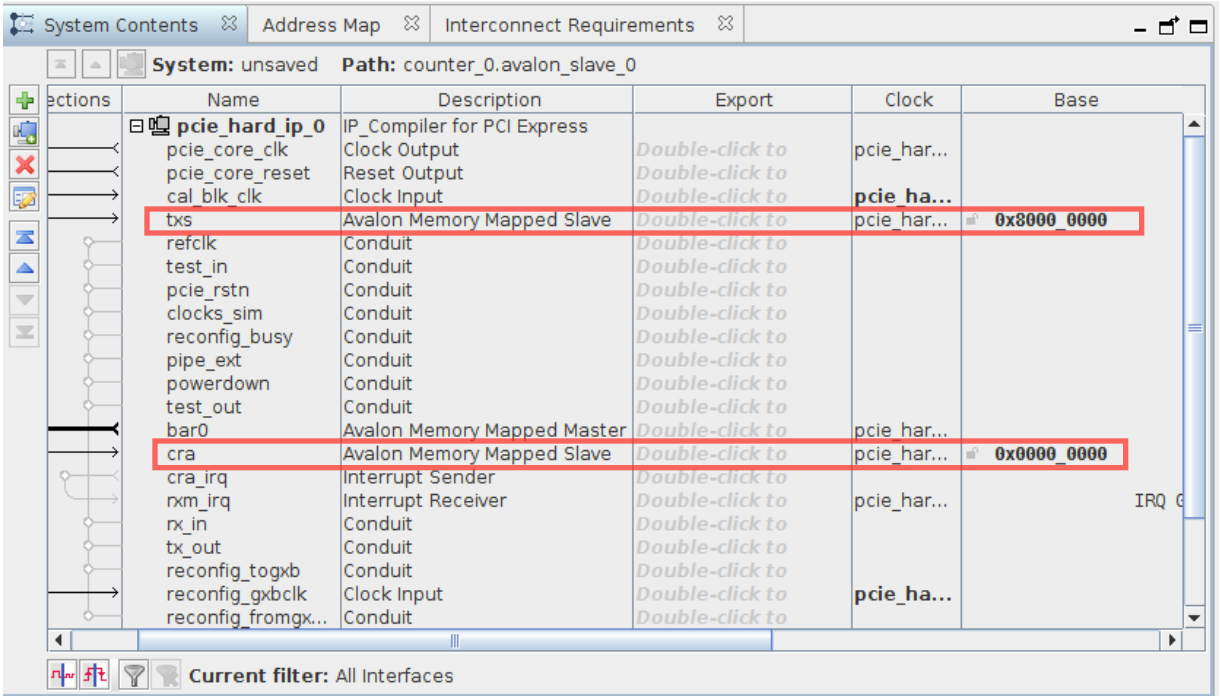


fig 6.2.9

Now move to **Address Map** tab. Change **counter_0.avalon_slave_0** address to **0x0000_4444 - 0x0000_4444** as shown in figure 6.2.10.

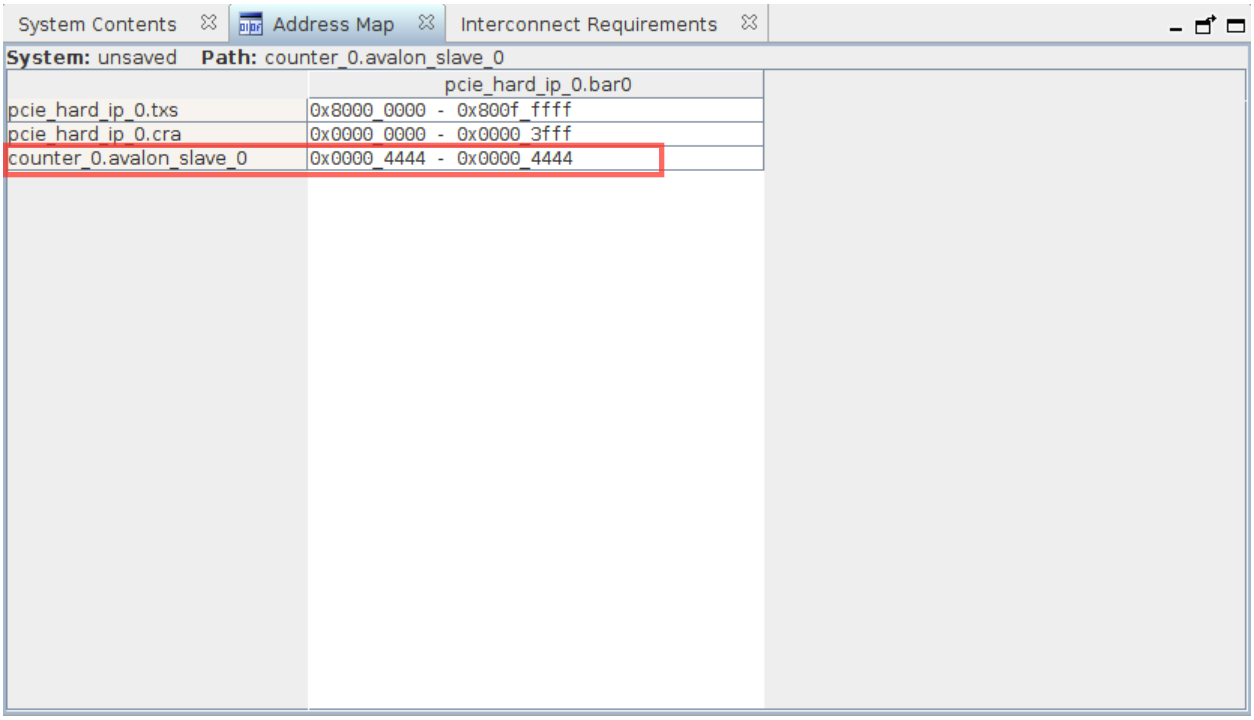


fig 6.2.10

By changing the address **of counter_0.avalon_slave_0** you should see no errors. You are done with system design. Now save the system by clicking **File -> Save**. Give the name **counter_pci.qsys** and click save.

After saving the system generate synthesis file for the system as explained in 5.2 and close the Qsys tool.

After closing the Qsys tool compile the design as explained in 3.3. Now FPGA is ready for configuring with the design. Before configuring the FPGA let us write the c application to read the counter output from Yocto OS running on Atom processor.

6.3 Writing and building c application

We use Terasic frame work to build the c application on the work station and later move that application to Yocto OS. More about Terasic framework is explained in 4.1. Before building the app, let us go throw the terasic API.

PCIE_Open

Function:
Open a specified PCIe card with vendor ID, device ID, and matched card index.

Prototype:
PCIE_HANDLE PCIE_Open(WORD wVendorID, WORD wDeviceID, WORD wCardIndex);

Parameters:
wVendorID:
Specify the desired vendor ID. A zero value means to ignore the vendor ID.
wDeviceID:
Specify the desired device ID. A zero value means to ignore the device ID.
wCardIndex:
Note, these three parameters are ignored in Linux edition..

Return Value:
Return a handle to presents specified PCIe card. A positive value is return if the PCIe card is opened successfully. A value zero means failed to connect the target PCIe card.
This handle value is used as a parameter for other functions, e.g. PCIE_Read32.
Users need to call PCIE_Close to release handle once the handle is no more used.

PCIE_Close

Function:
Close a handle associated to the PCIe card.

Prototype:
void PCIE_Close(PCIE_HANDLE hPCIE);

Parameters:
hPCIE:
A PCIe handle return by PCIE_Open function.

Return Value:
None.

PCIE_Read32

Function:
Read a 32-bits data from the FPGA board.

Prototype:
bool PCIE_Read32(PCIE_HANDLE hPCIE, PCIE_BAR PcieBar, PCIE_ADDRESS PcieAddress, DWORD * pdwData);

Parameters:
hPCIE:
A PCIe handle return by PCIE_Open function.
PcieBar:
Specify the target BAR.
PcieAddress:
Specify the target address in FPGA.
pdwData:
A buffer to retrieve the 32-bits data.

Return Value:
Return TRUE if read data is successful; otherwise FALSE is returned.

PCIE_Write32

Function:
Write a 32-bits data to the FPGA Board.

Prototype:
bool PCIE_Write32(PCIE_HANDLE hPCIE, PCIE_BAR PcieBar,
PCIE_ADDRESS PcieAddress, DWORD dwData);

Parameters:
hPCIE:
A PCIe handle return by PCIE_Open function.
PcieBar:
Specify the target BAR.
PcieAddress:
Specify the target address in FPGA.
dwData:
Specify a 32-bits data which will be written to FPGA board.

Return Value:
Return TRUE if write data is successful; otherwise FALSE is returned.

PCIE_DmaRead

Function:
Read data from the memory-mapped memory of FPGA board in DMA function.

Prototype:
bool PCIE_DmaRead(
PCIE_HANDLE hPCIE, PCIE_LOCAL_ADDRESS LocalAddress, void *pBuffer,
DWORD dwBufSize
);

Parameters:
hPCIE:
A PCIe handle return by PCIE_Open function.
LocalAddress:
Specify the target memory-mapped address in FPGA.
pBuffer:
A pointer to a memory buffer to retrieved the data from FPGA. The size of buffer should be equal or larger the dwBufSize.
dwBufSize:
Specify the byte number of data retrieved from FPGA.

Return Value:
Return TRUE if read data is successful; otherwise FALSE is returned.

PCIE_DmaWrite

Function:
Write data to the memory-mapped memory of FPGA board in DMA function.

Prototype:
bool PCIE_DmaWrite(
PCIE_HANDLE hPCIE, PCIE_LOCAL_ADDRESS LocalAddress, void *pData,
DWORD dwDataSize
);

Parameters:
hPCIE:
A PCIe handle return by PCIE_Open function.
LocalAddress:
Specify the target memory mapped address in FPGA.
pData:
A pointer to a memory buffer to store the data which will be written to FPGA.
dwDataSize:
Specify the byte number of data which will be written to FPGA.

Return Value:
Return TRUE if write data is successful; otherwise FALSE is returned.

PCIE_DmaFIFORead

Function:
Read data from the memory FIFO of FPGA board in DMA function.

Prototype:
bool PCIE_DmaFIFORead(PCIE_HANDLE hPCIE, PCIE_LOCAL_FIFO_ID LocalFIFOId,
void *pBuffer,
DWORD dwBufSize);

Parameters:
hPCIE:
A PCIe handle return by PCIE_Open function.
LocalFIFOId:
Specify the target memory FIFO ID in FPGA.
pBuffer:
A pointer to a memory buffer to retrieved the data from FPGA. The size of buffer should be equal or larger
the dwBufSize. dwBufSize:
Specify the byte number of data retrieved from FPGA.

Return Value:
Return TRUE if read data is successful; otherwise FALSE is returned.

PCIE_DmaFIFOWrite

Function: Write data to the memory FIFO of FPGA board in DMA function.
Prototype: bool PCIE_DmaFIFOWrite(PCIE_HANDLE hPCIE, PCIE_LOCAL_FIFO_ID LocalFIFOId, void *pData, DWORD dwDataSize);
Parameters: hPCIE: A PCIe handle return by PCIE_Open function. LocalFIFOId: Specify the target memory FIFO ID in FPGA. pData: A pointer to a memory buffer to store the data which will be written to FPGA. dwDataSize: Specify the byte number of data which will be written to FPGA.
Return Value: Return TRUE if write data is successful; otherwise FALSE is returned.

We went through Terasic frame work, now using those functions we write c application to read the counter output. As app should read the output from the counter we use only **PCIE_Read32, PCIE_Open** and **PCIE_Close** functions to write our application.

The C code for the application is

```
#include <stdio.h>
#include <stdlib.h>
#include "PCIE.h"    //including PCIE.h to implement terasic API

#define DEMO_PCIE_USER_BAR       PCIE_BAR0
#define DEMO_PCIE_COUNTER_ADDR   0x4444 //base address of counter_0.avalon_slave_0

int main(void)
{
    void *lib_handle;
    PCIE_HANDLE hPCIE;
    BOOL bQuit = FALSE;
    int select0 = 0;
    int temp_sel;

    printf("== Terasic: PCIe counter Demo Program ==\r\n");

    lib_handle = PCIE_Load();
    if (!lib_handle)
    {
        printf("PCIE_Load failed!\r\n");
        return 0;
    }
    hPCIE = PCIE_Open(0,0,0);
    if (!hPCIE)
    {
        printf("PCIE_Open failed\r\n");
    }
    else
    {
        while(!bQuit)
        {

            printf("select your option: 0-counter read,
AnyOtherDecimalNumber-exit \r\n");
            scanf("%d",temp_sel);
            if(temp_sel==select0)
            {
               PCIE_Read_counter(hPCIE);
            }
            else
            {
               bQuit = TRUE;
               printf("Bye!\r\n");
            }



        }
        PCIE_Close(hPCIE);
```

```
    }

    PCIE_Unload(lib_handle);
    return 0;

}


BOOL PCIE_Read_counter(PCIE_HANDLE hPCIe)
{
    BOOL bpass = TRUE;
    DWORD Status;
    bpass =
PCIE_Read32(hPCIe,DEMO_PCIE_USER_BAR,DEMO_PCIE_COUNTER_ADDR,&Status);
    if (bpass)
        printf("Counter Value:=%xh\r\n", Status);
    else
        printf("Failed to read counter value\r\n");

    return bpass;
}
```

Save the file as app.c.

Now its time to build the application. Before building the application make sure following files are along with app.c in the same directory.

- TERASIC_PCIE.h
- terasic_pcie_qsys.so
- PCIE.c
- PCIE.h

In the same directory create a file with name **Makefile**(without any extension) and add the following lines

```
#
TARGET = app

#
CFLAGS = -g -Wall
LDFLAGS =  -g -Wall
#LDFLAGS =  -g -Wall  -Iteraisc_pcie_qsys.so -ldl
#-ldl must be placed after the file calling lpxxxx funciton


build: $(TARGET)

app: app.o PCIE.o
```

```
	$(CC) $(LDFLAGS) $^ -o $@ -ldl

%.o : %.c
	$(CC) $(CFLAGS) -c $< -o $@

.PHONY: clean
clean:
	rm -f $(TARGET) *.a *.o *~
```

Note : We have created application with name app.c, if you change to any other name make sure you change it in the Makefile at places highlighted in red.

Now change the working directory of workstation terminal to directory where all these files are located. Then run the following command to build app.exe file

```
makefile
```

After executing the above command you can find executable file(**app.exe**) in the same folder. Now along with app.exe move **terasic_pcie_qsys.so** to the yocto OS running on the board. About how to move files to Yocto OS is explained in 5.1 .

6.4 Running and Results

Running the app

Before running the app on yocto OS follow the below steps in order
step 1: Program the FPGA(refer section 3.3)
step 2: Restart the Yocto OS(execute `reboot` command in yocto terminal)
step 3: Load the terasic PCIe drivers(refer section 5.3)

Result

Now run the application by executing below command in yocto terminal

```
./app
```

After executing the above command you will see a message **PCIE_Open failed,** which is not the expected result. We are unable to figure out the exact problem to this error i.e due to FPGA configuration or app code. If this error is figured out we get a clear path on how the communication is happening between Atom processor and FPGA. We are working on it and update this documentation with expected result as soon as possible and other more implementations in future.