Using PCI Express on DE4 Boards

For Quartus® Prime 18.1

1 Introduction

This tutorial describes how to use the PCI Express (Peripheral Component Interconnect) on Intel DE4 board. It first demonstrates how to build a system with the PCI Express IP Core using Platform Designer and then shows how to use the PCI driver on Linux* operating system. The discussion is based on the assumption that the reader has the basic knowledge of C language and Verilog hardware description language. Also the reader should be familiar with Quartus® Prime software and Linux operating system.

Contents:

- Background
- Building the PCI Express System
- Using the Driver
- Changing the Driver Codes

2 Background

PCI Express is a high-performance interconnect protocol for use in a variety of applications including network adapters, storage area networks, embedded controllers, graphic accelerator boards, and audio-video products.

An example of a system using PCI Express on DE4 board is shown in Figure 1. The PCI Express IP Core implements the PCI Express interface protocol to allows data transfer between a host computer and the on-chip memory on the board. The data can be transferred either directly or through the DMA controllers. Besides, a Nios® II processor is used to perform calculation on the data in the on-chip memory.

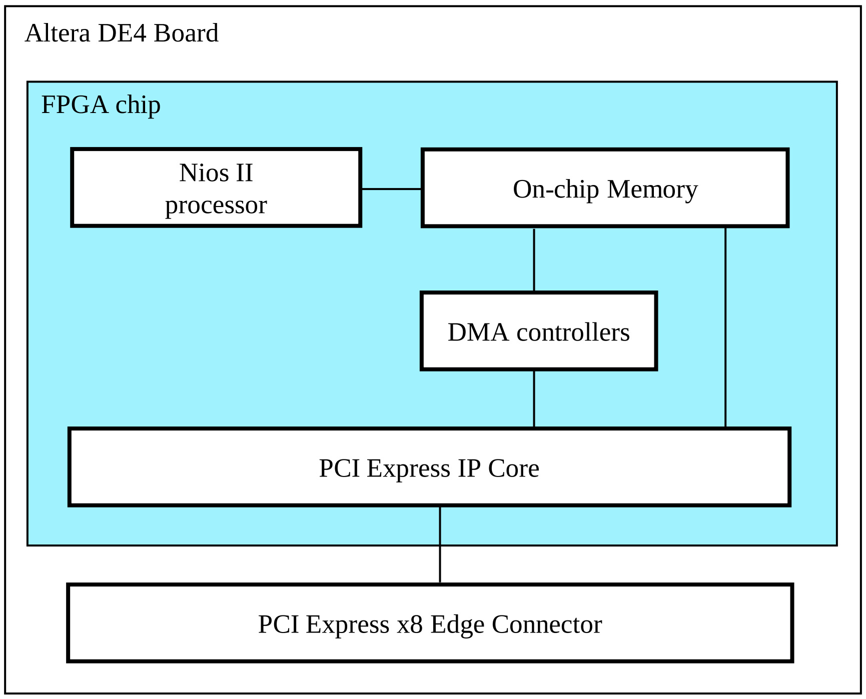


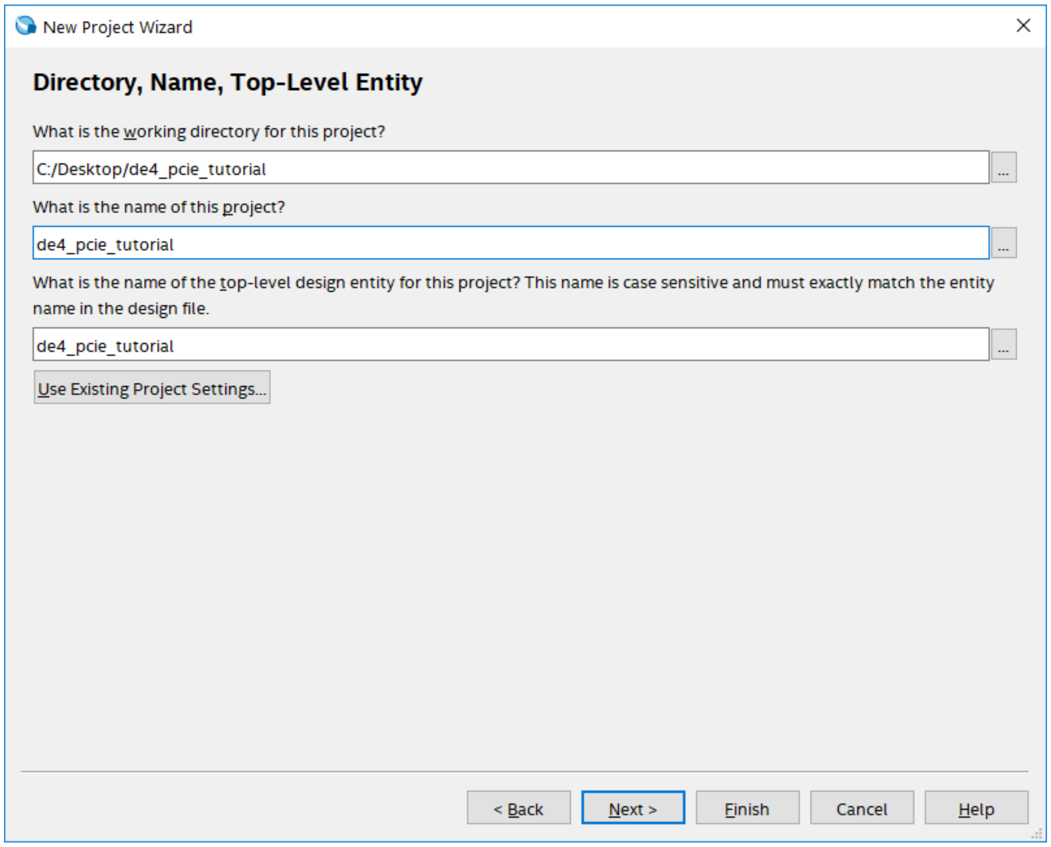
Figure 1. The block diagram for the system.

A simple application of this system is that the host computer can put data to the on-chip memory through a DMA controller and wait for the Nios II processor to perform calculation on the data. Once the data is calculated, the host computer can read the data back through the PCI Express IP Core.

Note that this tutorial shows the ways to use both DMA Controller and Scatter-Gather DMA Controller. In a real design, two different kinds of DMA controllers are not necessary.

3 Building the PCI Express System

To use the PCI Express on the DE4 board, you have to instantiate the PCI Express IP Core in your hardware system. To start with, create a new Quartus Prime project for your system. As shown in Figure 2, store your project in a directory called *de4_pcie_tutorial* and assign the same name to both the project and its top-level design entity. In your project, from the list of available devices, choose the appropriate device name for the FPGA used on the DE4 board. A list of devices names of DE4 boards can be found in Table 1.


New Project Wizard

Directory, Name, Top-Level Entity

What is the working directory for this project?

C:/Desktop/de4_pcie_tutorial

What is the name of this project?

de4_pcie_tutorial

What is the name of the top-level design entity for this project? This name is case sensitive and must exactly match the entity name in the design file.

de4_pcie_tutorial

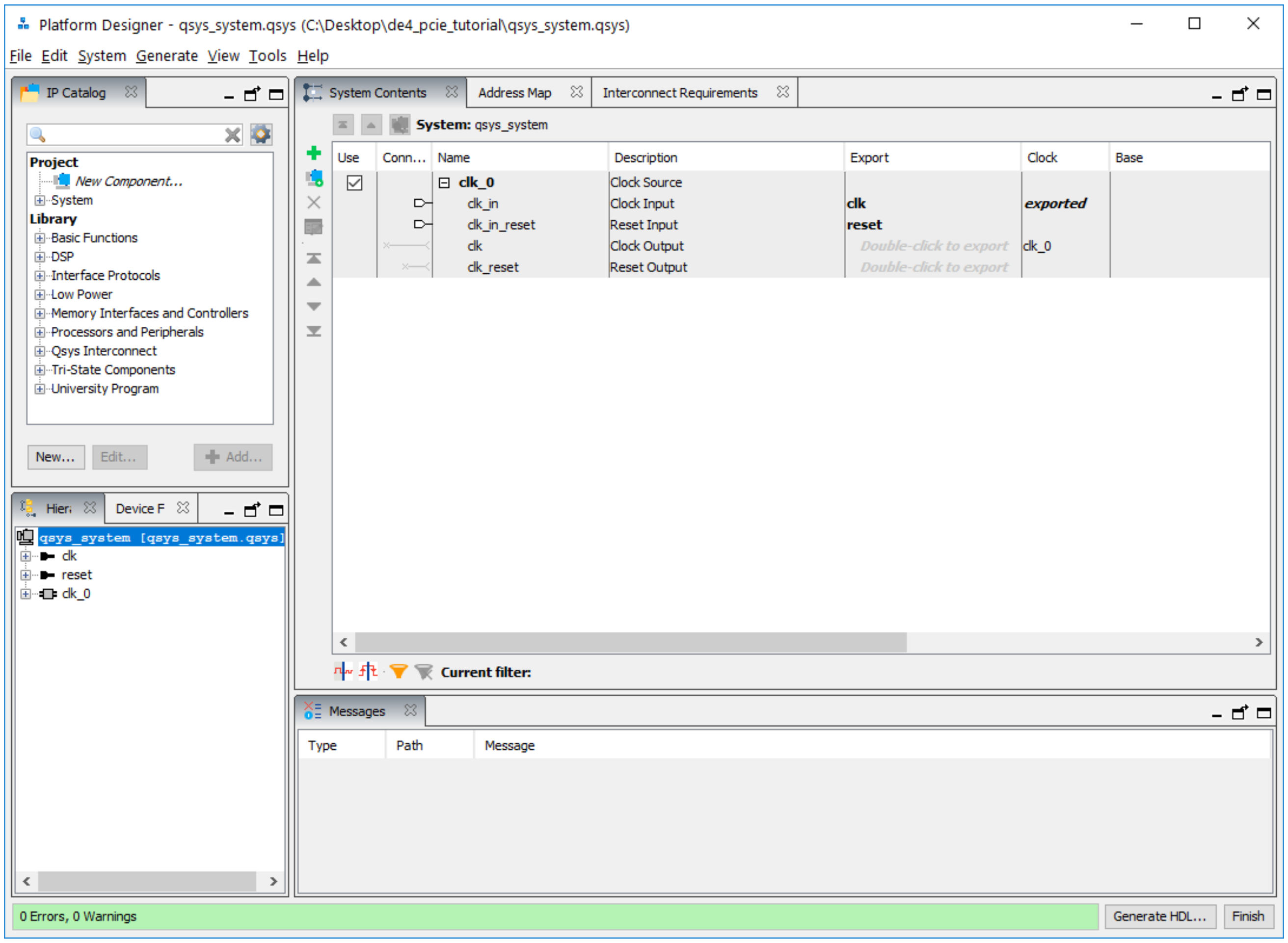
Figure 2. Create a new project.

Table 1. DE4 FPGA device names

Board	Device Name
DE4-230	Stratix® IV GX EP4SGX230KF40C2
DE4-530	Stratix® IV GX EP4SGX530KH40C2

3.1 Creating a Platform Designer System

In this section, we build a system with the PCI Express IP Core, DMA Controller, Scatter-Gather DMA Controller, on-chip memory, and the Nios II processor in Platform Designer. Select **Tools** > **Platform Designer** to open the Platform Designer as shown in Figure 3, and then save the file as *qsys_system.qsys*.



Use	Conn...	Name	Description	Export	Clock	Base
☑		clk_0	Clock Source			
		clk_in	Clock Input	clk	exported	
		clk_in_reset	Reset Input	reset		
		clk	Clock Output	Double-click to export	clk_0	
		clk_reset	Reset Output	Double-click to export		

Type	Path	Message

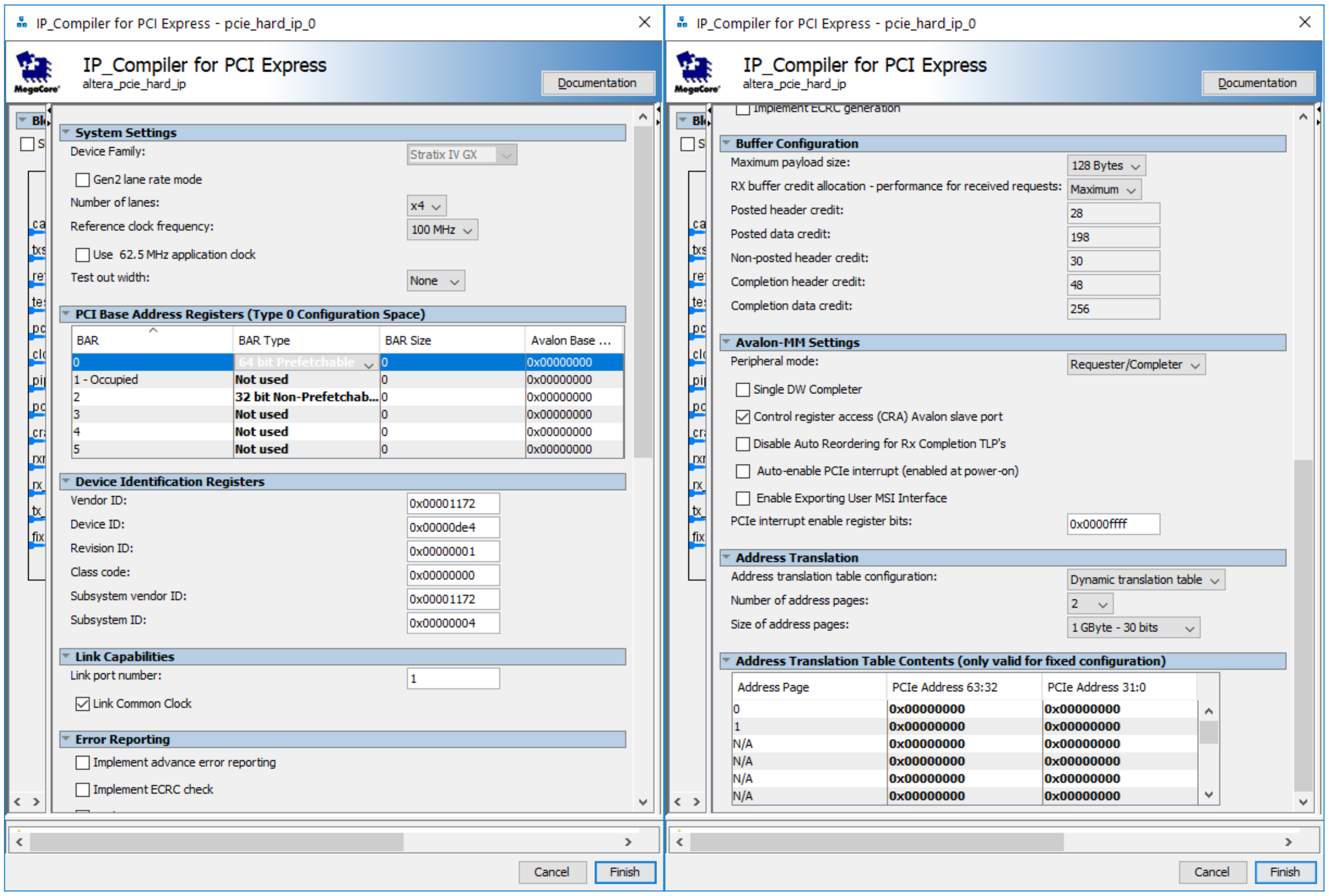
0 Errors, 0 Warnings

Figure 3. Open the Platform Designer tool.

To add the PCI Express IP Core, perform the following:

- Select **Interface Protocols** > **PCI Express** > **IP Compiler for PCI Express** and click **Add**. The Configuration Wizard window appears. You can use the scroll bar on the right to view parameters that are not initially visible.
- Under the **System Settings** heading, set the **Test out width** to *None* and leave others as default.
- Under the **PCI Base Address Registers** heading, set **BAR0** as *64 bit Prefetchable Memory* and **BAR2** as *32 bit Non-Prefetchable*.

- Under the **Device Identification Registers** heading, set the **Device ID** to *0x00000de4*.
- Under the **Address Translation** heading, set the **Address translation table configuration** to *Dynamic translation table*, **Number of address pages** to *2* and the **Size of address pages** to *1 GByte - 30 bits*.
- Click **Finish** to add the PCI Express IP Core *pcie_hard_ip_0* to the Platform Designer system. Figure 4 shows screenshots of the settings.


IP_Compiler for PCI Express - pcie_hard_ip_0
IP_Compiler for PCI Express
altera_pcie_hard_ip
Documentation
Cancel
Finish

System Settings

Device Family: Stratix IV GX
Gen2 lane rate mode ☐
Number of lanes: x4
Reference clock frequency: 100 MHz
☐ Use 62.5 MHz application clock
Test out width: None

PCI Base Address Registers (Type 0 Configuration Space)

BAR	BAR Type	BAR Size	Avalon Base ...
0	64 bit Prefetchable	0	0x00000000
1 - Occupied	Not used	0	0x00000000
2	32 bit Non-Prefetchab...	0	0x00000000
3	Not used	0	0x00000000
4	Not used	0	0x00000000
5	Not used	0	0x00000000

Device Identification Registers

Vendor ID: 0x00001172
Device ID: 0x00000de4
Revision ID: 0x00000001
Class code: 0x00000000
Subsystem vendor ID: 0x00001172
Subsystem ID: 0x00000004

Link Capabilities

Link port number: 1
☑ Link Common Clock

Error Reporting

☐ Implement advance error reporting
☐ Implement ECRC check
☐ Implement ECRC generation

Buffer Configuration

Maximum payload size: 128 Bytes
RX buffer credit allocation - performance for received requests: Maximum
Posted header credit: 28
Posted data credit: 198
Non-posted header credit: 30
Completion header credit: 48
Completion data credit: 256

Avalon-MM Settings

Peripheral mode: Requester/Completer
☐ Single DW Completer
☑ Control register access (CRA) Avalon slave port
☐ Disable Auto Reordering for Rx Completion TLP's
☐ Auto-enable PCIe interrupt (enabled at power-on)
☐ Enable Exporting User MSI Interface
PCIe interrupt enable register bits: 0x0000ffff

Address Translation

Address translation table configuration: Dynamic translation table
Number of address pages: 2
Size of address pages: 1 GByte - 30 bits

Address Translation Table Contents (only valid for fixed configuration)

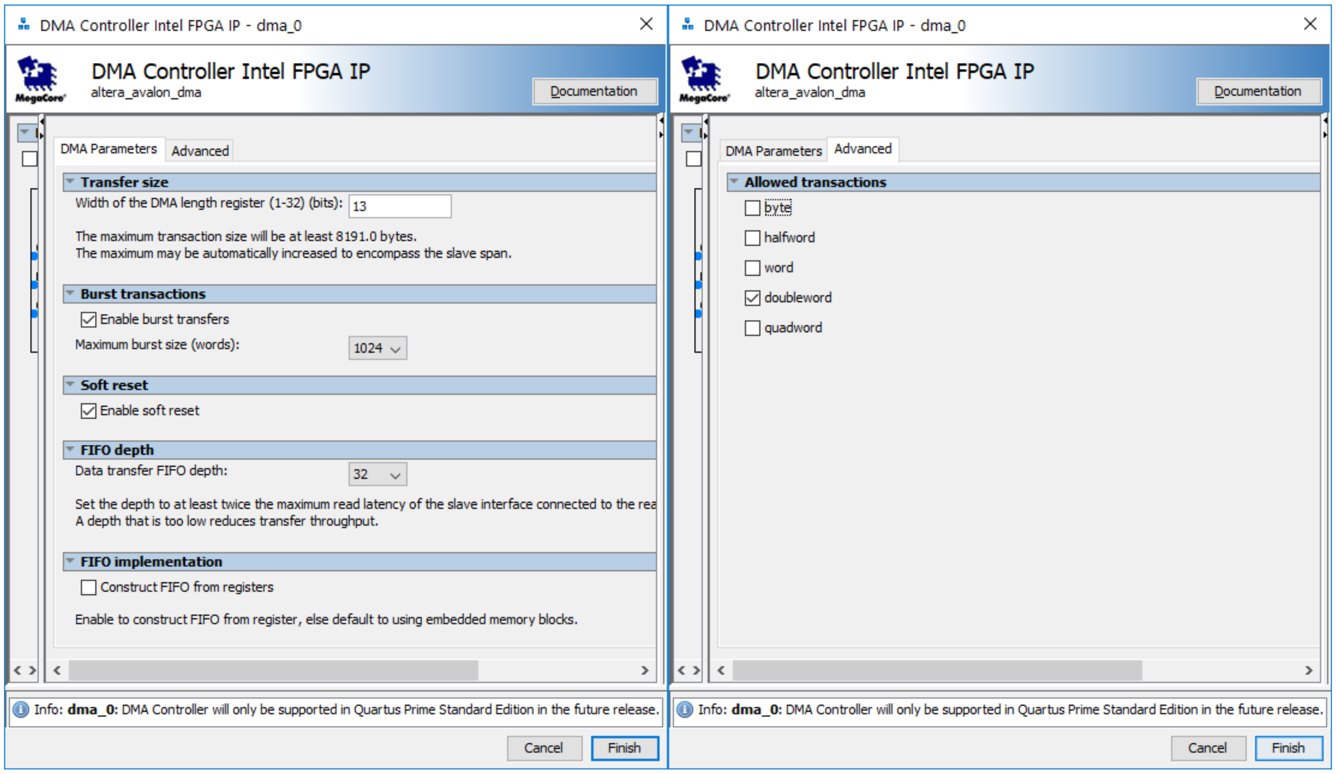
Address Page	PCIe Address 63:32	PCIe Address 31:0
0	0x00000000	0x00000000
1	0x00000000	0x00000000
N/A	0x00000000	0x00000000
N/A	0x00000000	0x00000000
N/A	0x00000000	0x00000000
N/A	0x00000000	0x00000000

Figure 4. Settings for PCI Express IP Core.

To add the DMA Controller, perform the following:

- Select **Basic Functions** > **DMA** > **DMA Controller Intel FPGA IP** and click **Add**.
- Under the **DMA Parameters** tab, check the checkbox **Enable burst transfers** and set the **Maximum burst size** to 1024 words.
- Select the **Advanced** tab, and uncheck all the checkboxes except **doubleword**. This makes the DMA Controller allow for doubleword (64 bits) transactions only.

- Click **Finish**, then the DMA Controller *dma_0* is added to the Platform Designer system. Figure 5 shows the screenshots of the settings.


DMA Controller Intel FPGA IP - dma_0
DMA Controller Intel FPGA IP
altera_avalon_dma
Documentation
DMA Parameters
Advanced
Cancel
Finish

Transfer size
Width of the DMA length register (1-32) (bits): 13
The maximum transaction size will be at least 8191.0 bytes.
The maximum may be automatically increased to encompass the slave span.

Burst transactions
☑ Enable burst transfers
Maximum burst size (words): 1024

Soft reset
☑ Enable soft reset

FIFO depth
Data transfer FIFO depth: 32
Set the depth to at least twice the maximum read latency of the slave interface connected to the rea
A depth that is too low reduces transfer throughput.

FIFO implementation
☐ Construct FIFO from registers
Enable to construct FIFO from register, else default to using embedded memory blocks.

Info: **dma_0:** DMA Controller will only be supported in Quartus Prime Standard Edition in the future release.

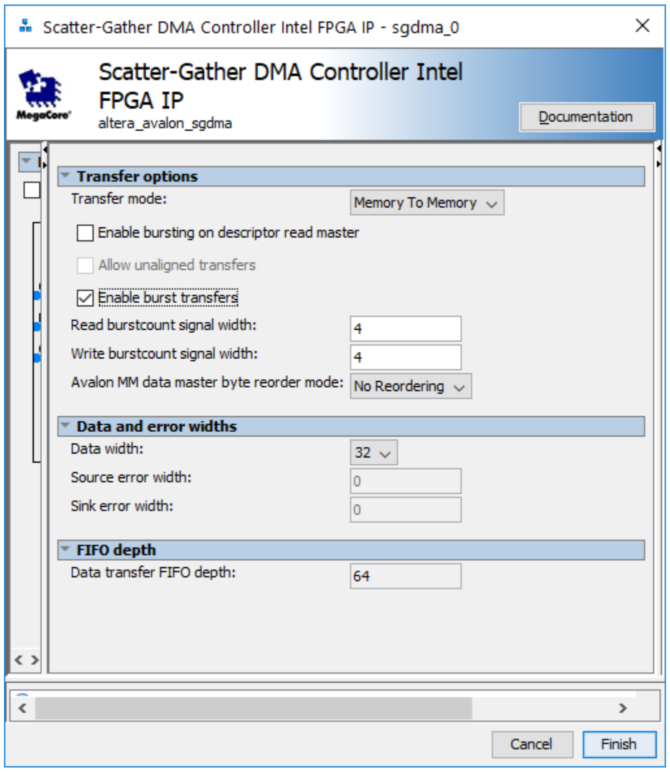
Allowed transactions
☐ byte
☐ halfword
☐ word
☑ doubleword
☐ quadword

Info: **dma_0:** DMA Controller will only be supported in Quartus Prime Standard Edition in the future release.

Figure 5. Settings for DMA Controller.

To add the Scatter-Gather DMA Controller, perform the following:

- Select **Basic Functions** > **DMA** > **Scatter-Gather DMA Controller Intel FPGA IP** and click **Add**.
- In the Configuration Wizard window, check the checkbox of **Enable burst transfers** and set the **Data width** to 32 as shown in Figure 6.
- Click **Finish**, then the Scatter-Gather DMA Controller *sgdma_0* is added to the Platform Designer system.


Scatter-Gather DMA Controller Intel FPGA IP - sgdma_0

Scatter-Gather DMA Controller Intel FPGA IP
altera_avalon_sgdma

Transfer options
Transfer mode: Memory To Memory
[ ] Enable bursting on descriptor read master
[ ] Allow unaligned transfers
[✓] Enable burst transfers
Read burstcount signal width: 4
Write burstcount signal width: 4
Avalon MM data master byte reorder mode: No Reordering

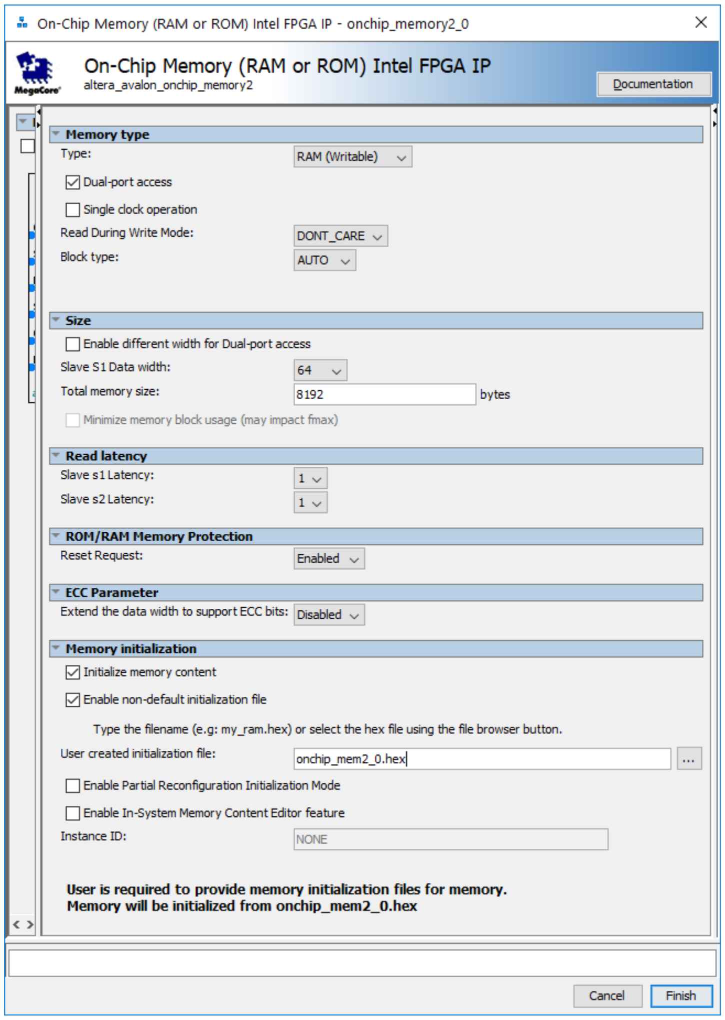
Data and error widths
Data width: 32
Source error width: 0
Sink error width: 0

FIFO depth
Data transfer FIFO depth: 64

Figure 6. Settings for Scatter-Gather DMA Controller.

To add the on-chip memory, perform the following:

- Select **Basic Functions** > **On Chip Memory** > **On-chip Memory(RAM or ROM) Intel FPGA IP** and click **Add**.
- In the Configuration Wizard window, check the checkbox of **Dual-port access**. This will enable the on-chip memory to support two different clock signals.
- Set the **Slafe S1 Data width** to *64 bits* and the **Total Memory Size** to *8 Kbytes (8192 bytes)*.
- Check the check box of **Enable non-default initialization file** and set the **User created initialization file** to *onchip_memory2_0.hex* as shown in Figure 7.
- Click **Finish**, then the on-chip memory *onchip_memory2_0* is added to the Platform Designer system.


On-Chip Memory (RAM or ROM) Intel FPGA IP - onchip_memory2_0
On-Chip Memory (RAM or ROM) Intel FPGA IP
altera_avalon_onchip_memory2
Documentation
Cancel
Finish

Memory type
Type: RAM (Writable)
[✓] Dual-port access
[ ] Single clock operation
Read During Write Mode: DONT_CARE
Block type: AUTO

Size
[ ] Enable different width for Dual-port access
Slave S1 Data width: 64
Total memory size: 8192 bytes
[ ] Minimize memory block usage (may impact fmax)

Read latency
Slave s1 Latency: 1
Slave s2 Latency: 1

ROM/RAM Memory Protection
Reset Request: Enabled

ECC Parameter
Extend the data width to support ECC bits: Disabled

Memory initialization
[✓] Initialize memory content
[✓] Enable non-default initialization file
Type the filename (e.g: my_ram.hex) or select the hex file using the file browser button.
User created initialization file: onchip_mem2_0.hex
[ ] Enable Partial Reconfiguration Initialization Mode
[ ] Enable In-System Memory Content Editor feature
Instance ID: NONE

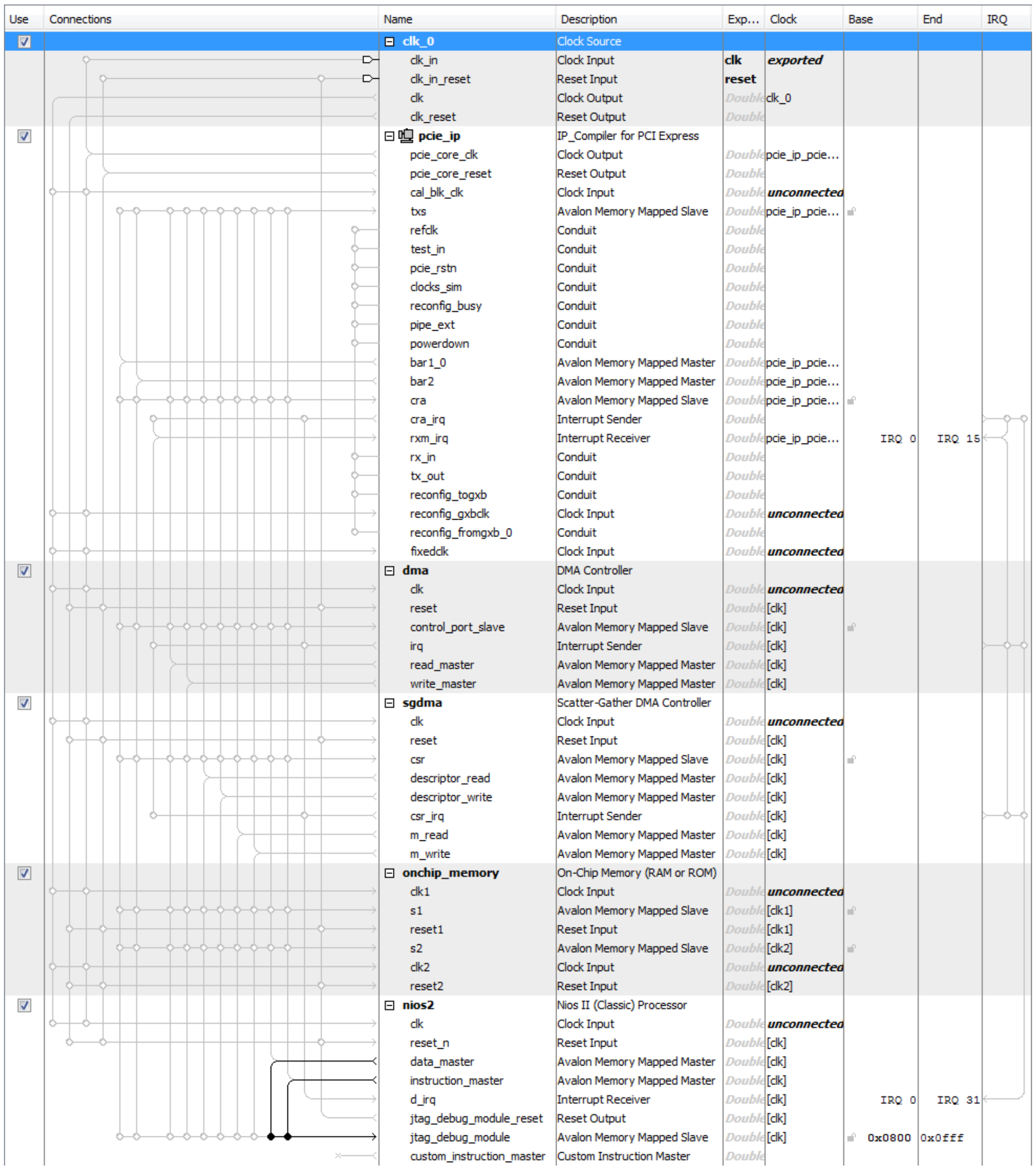
User is required to provide memory initialization files for memory.
Memory will be initialized from onchip_mem2_0.hex

Figure 7. Settings for On-chip Memory.

To add the Nios II processor, perform the following:

- Select **Processors and Peripherals > Embedded Processors > Nios II Processor** and click **Add**.
- Choose **Nios II/e**, which is the simplest version of the processor.
- Click **Finish** to add the Nios II Processor *nios2_qsys_0* to the Platform Designer system. There may be some error messages at the bottom of the screen, because some parameters have not been specified yet. Ignore these messages as we will provide the necessary data later.

Note that the Platform Designer automatically chooses names for the components. The name are not necessarily descriptive enough for the design, but they can be changed. To rename a component, right-click the component name and select **Rename**. Change the names of your components to the names shown in Figure 8.



Use	Connections	Name	Description	Exp...	Clock	Base	End	IRQ
✓		⊟ **clk_0**	Clock Source					
		clk_in	Clock Input	**clk**	***exported***			
		clk_in_reset	Reset Input	**reset**				
		clk	Clock Output	Double	clk_0			
		clk_reset	Reset Output	Double				
✓		⊟ **pcie_ip**	IP_Compiler for PCI Express					
		pcie_core_clk	Clock Output	Double	pcie_ip_pcie...			
		pcie_core_reset	Reset Output	Double				
		cal_blk_clk	Clock Input	Double	***unconnected***			
		txs	Avalon Memory Mapped Slave	Double	pcie_ip_pcie...			
		refclk	Conduit	Double				
		test_in	Conduit	Double				
		pcie_rstn	Conduit	Double				
		clocks_sim	Conduit	Double				
		reconfig_busy	Conduit	Double				
		pipe_ext	Conduit	Double				
		powerdown	Conduit	Double				
		bar1_0	Avalon Memory Mapped Master	Double	pcie_ip_pcie...			
		bar2	Avalon Memory Mapped Master	Double	pcie_ip_pcie...			
		cra	Avalon Memory Mapped Slave	Double	pcie_ip_pcie...			
		cra_irq	Interrupt Sender	Double				
		rxm_irq	Interrupt Receiver	Double	pcie_ip_pcie...	IRQ 0	IRQ 15	
		rx_in	Conduit	Double				
		tx_out	Conduit	Double				
		reconfig_togxb	Conduit	Double				
		reconfig_gxbclk	Clock Input	Double	***unconnected***			
		reconfig_fromgxb_0	Conduit	Double				
		fixedclk	Clock Input	Double	***unconnected***			
✓		⊟ **dma**	DMA Controller					
		clk	Clock Input	Double	***unconnected***			
		reset	Reset Input	Double	[clk]			
		control_port_slave	Avalon Memory Mapped Slave	Double	[clk]			
		irq	Interrupt Sender	Double	[clk]			
		read_master	Avalon Memory Mapped Master	Double	[clk]			
		write_master	Avalon Memory Mapped Master	Double	[clk]			
✓		⊟ **sgdma**	Scatter-Gather DMA Controller					
		clk	Clock Input	Double	***unconnected***			
		reset	Reset Input	Double	[clk]			
		csr	Avalon Memory Mapped Slave	Double	[clk]			
		descriptor_read	Avalon Memory Mapped Master	Double	[clk]			
		descriptor_write	Avalon Memory Mapped Master	Double	[clk]			
		csr_irq	Interrupt Sender	Double	[clk]			
		m_read	Avalon Memory Mapped Master	Double	[clk]			
		m_write	Avalon Memory Mapped Master	Double	[clk]			
✓		⊟ **onchip_memory**	On-Chip Memory (RAM or ROM)					
		clk1	Clock Input	Double	***unconnected***			
		s1	Avalon Memory Mapped Slave	Double	[clk1]			
		reset1	Reset Input	Double	[clk1]			
		s2	Avalon Memory Mapped Slave	Double	[clk2]			
		clk2	Clock Input	Double	***unconnected***			
		reset2	Reset Input	Double	[clk2]			
✓		⊟ **nios2**	Nios II (Classic) Processor					
		clk	Clock Input	Double	***unconnected***			
		reset_n	Reset Input	Double	[clk]			
		data_master	Avalon Memory Mapped Master	Double	[clk]			
		instruction_master	Avalon Memory Mapped Master	Double	[clk]			
		d_irq	Interrupt Receiver	Double	[clk]	IRQ 0	IRQ 31	
		jtag_debug_module_reset	Reset Output	Double	[clk]			
		jtag_debug_module	Avalon Memory Mapped Slave	Double	[clk]	0x0800	0x0fff	
		custom_instruction_master	Custom Instruction Master	Double				

Figure 8. The Platform Designer system with renamed components.

After adding components, you have to connected them correctly. In Platform Designer, the **Connections** column displays the potential connection points between components. A filled dot shows that a connection is made, while an open dot shows a potential connection point. To complete the connections in the Platform Designer system, follow these steps:

1. Connect the **pcie_ip** *bar1_0* Avalon® MM master port to the **onchip_memory** *s1* Avalon MM slave port using the following procedures:

   (a) Right-click the **IP Compiler for PCI Express** component **pcie_ip**, hover in the **Connections** and then the **pcie_ip.bar1_0** to display possible connections.

   (b) Select the **onchip_memory.s1** on the list appeared.

2. Repeat step 1 to make the remaining connections listed in Table 2.

Table 2. Complete list of Platform Designer connections

Make Connection From:	**To:**
clk_50.clk	*pcie_ip.cal_blk_clk*
clk_50.clk	*onchip_memory.clk2*
clk_50.clk	*nios2.clk*
clk_50.clk_reset	*onchip_memory.reset2*
clk_50.clk_reset	*nios2.reset_n*
pcie_ip.pcie_core_clk	*onchip_memory.clk1*
pcie_ip.pcie_core_clk	*dma.clk*
pcie_ip.pcie_core_clk	*sgdma.clk*
pcie_ip.pcie_core_reset	*onchip_memory.reset1*
pcie_ip.pcie_core_reset	*dma .reset*
pcie_ip.pcie_core_reset	*sgdma.reset*
pcie_ip.bar1_0 (step 2)	*onchip_memory.s1*
pcie_ip.bar2	*dma.control_port_slave*
pcie_ip.bar2	*pcie_ip.cra*
pcie_ip.bar2	*sgdma.csr*
dma.read_master	*onchip_memory.s1*
dma.read_master	*pcie_ip.txs*
dma.write_master	*onchip_memory.s1*
dma.write_master	*pcie_ip.txs*
sgdma.descriptor_read	*pcie_ip.txs*
sgdma.descriptor_write	*pcie_ip.txs*
sgdma.m_read	*onchip_memory.s1*
sgdma.m_read	*pcie_ip.txs*
sgdma.m_write	*onchip_memory.s1*
sgdma.m_write	*pcie_ip.txs*
nios2.data_master	*onchip_memory.s2*
nios2.instruction_master	*onchip_memory.s2*

3. In the **IRQ** panel, click the connection from the Interrupt Sender of the *dma* component to the Interrupt Receiver *pcie_ip* component and type 1 into the box. Because the Platform Designer-generated IP Compiler for PCI Express implements an individual interrupt scheme, you must specify the bit to which interrupt connects. In this case, the DMA Controller's interrupt sender signal connects to bit 1 of the IP Compiler for PCI Express input interrupt bus.

4. In the **IRQ** panel, connect the Interrupt Sender of the *sgdma* component to the Interrupt Receiver *pcie_ip* component and type 2.

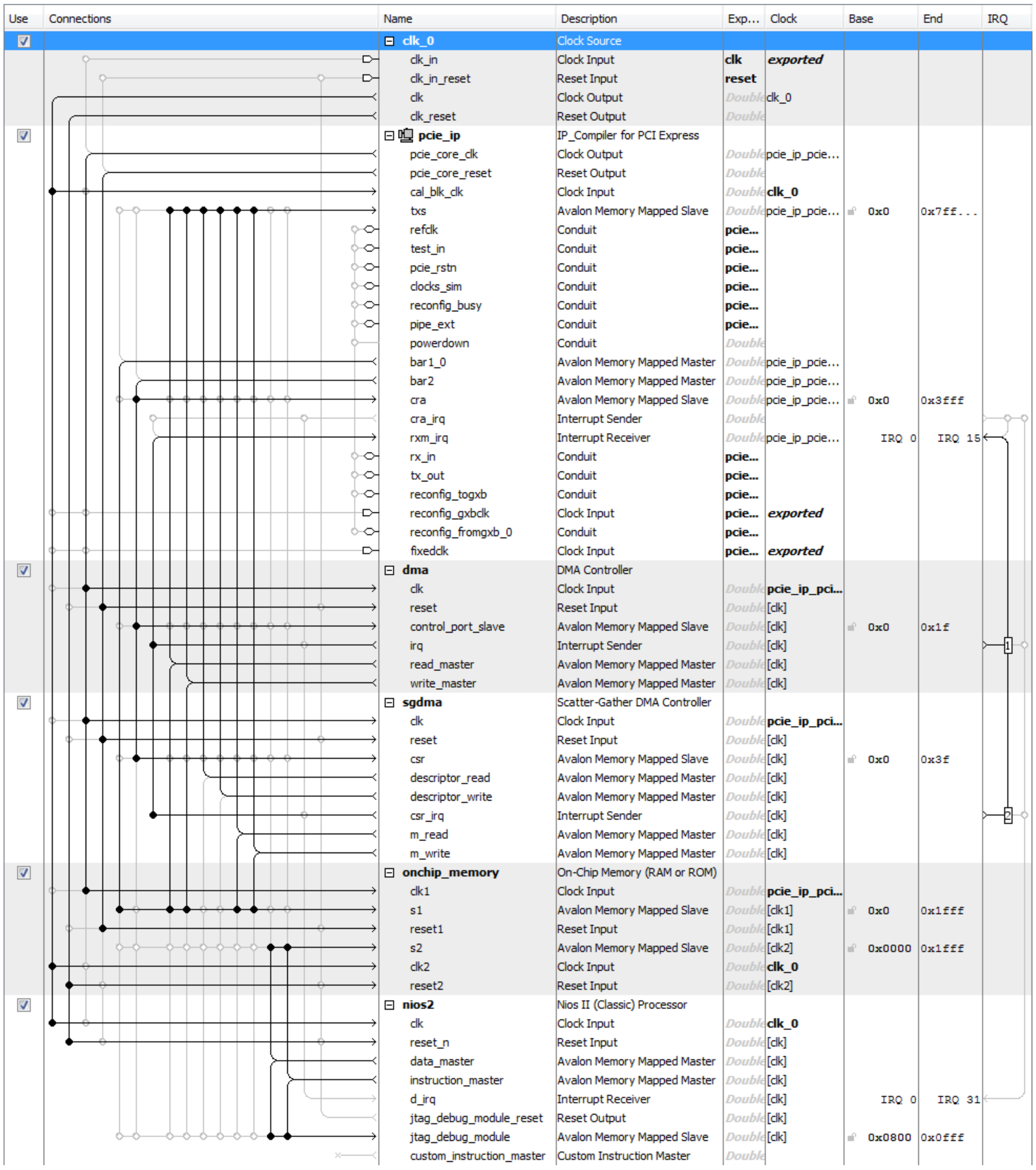
Besides connections inside the Platform Designer system, there are signals that have to be exported to make connections outside the system. To export signals, follow the steps:

1. In the row of the signal you want to export, click the **Export** column.
2. Accept the default name that appears in the **Export** column by clicking outside the cell without modifying the text.

Export the *pcie_ip* interfaces listed in Table 3. After the signals are connected or exported, your Platform Designer system should appear as indicated in Figure 9.

Table 3. List of the exported interfaces

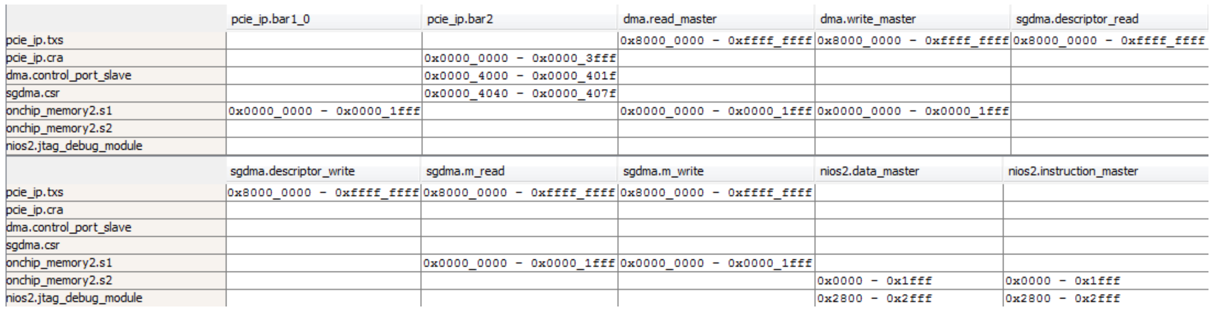
Interface Name	**Exported Name**
refclk	*pcie_ip_refclk*
test_in	*pcie_ip_test_in*
pcie_rstn	*pcie_ip_pcie_rstn*
clocks_sim	*pcie_ip_clocks_sim*
reconfig_busy	*pcie_ip_reconfig_busy*
pipe_ext	*pcie_ip_pipe_ext*
rx_in	*pcie_ip_rx_in*
tx_out	*pcie_ip_tx_out*
reconfig_togxb	*pcie_ip_reconfig_togxb*
reconfig_gxbclk	*pcie_ip_reconfig_gxbclk*
reconfig_fromgxb_0	*pcie_ip_reconfig_fromgxb_0*
fixedclk	*pcie_ip_fixedclk*



Use	Connections	Name	Description	Exp...	Clock	Base	End	IRQ
✓		**clk_0**	Clock Source					
		clk_in	Clock Input	**clk**	***exported***			
		clk_in_reset	Reset Input	**reset**				
		clk	Clock Output	Double	clk_0			
		clk_reset	Reset Output	Double				
✓		**pcie_ip**	IP_Compiler for PCI Express					
		pcie_core_clk	Clock Output	Double	pcie_ip_pcie...			
		pcie_core_reset	Reset Output	Double				
		cal_blk_clk	Clock Input	Double	**clk_0**			
		txs	Avalon Memory Mapped Slave	Double	pcie_ip_pcie...	0x0	0x7ff...	
		refclk	Conduit	**pcie...**				
		test_in	Conduit	**pcie...**				
		pcie_rstn	Conduit	**pcie...**				
		clocks_sim	Conduit	**pcie...**				
		reconfig_busy	Conduit	**pcie...**				
		pipe_ext	Conduit	**pcie...**				
		powerdown	Conduit	Double				
		bar1_0	Avalon Memory Mapped Master	Double	pcie_ip_pcie...			
		bar2	Avalon Memory Mapped Master	Double	pcie_ip_pcie...			
		cra	Avalon Memory Mapped Slave	Double	pcie_ip_pcie...	0x0	0x3fff	
		cra_irq	Interrupt Sender	Double				
		rxm_irq	Interrupt Receiver	Double	pcie_ip_pcie...	IRQ 0	IRQ 15	
		rx_in	Conduit	**pcie...**				
		tx_out	Conduit	**pcie...**				
		reconfig_togxb	Conduit	**pcie...**				
		reconfig_gxbclk	Clock Input	**pcie...**	***exported***			
		reconfig_fromgxb_0	Conduit	**pcie...**				
		fixedclk	Clock Input	**pcie...**	***exported***			
✓		**dma**	DMA Controller					
		clk	Clock Input	Double	**pcie_ip_pci...**			
		reset	Reset Input	Double	[clk]			
		control_port_slave	Avalon Memory Mapped Slave	Double	[clk]	0x0	0x1f	
		irq	Interrupt Sender	Double	[clk]			1
		read_master	Avalon Memory Mapped Master	Double	[clk]			
		write_master	Avalon Memory Mapped Master	Double	[clk]			
✓		**sgdma**	Scatter-Gather DMA Controller					
		clk	Clock Input	Double	**pcie_ip_pci...**			
		reset	Reset Input	Double	[clk]			
		csr	Avalon Memory Mapped Slave	Double	[clk]	0x0	0x3f	
		descriptor_read	Avalon Memory Mapped Master	Double	[clk]			
		descriptor_write	Avalon Memory Mapped Master	Double	[clk]			
		csr_irq	Interrupt Sender	Double	[clk]			2
		m_read	Avalon Memory Mapped Master	Double	[clk]			
		m_write	Avalon Memory Mapped Master	Double	[clk]			
✓		**onchip_memory**	On-Chip Memory (RAM or ROM)					
		clk1	Clock Input	Double	**pcie_ip_pci...**			
		s1	Avalon Memory Mapped Slave	Double	[clk1]	0x0	0x1fff	
		reset1	Reset Input	Double	[clk1]			
		s2	Avalon Memory Mapped Slave	Double	[clk2]	0x0000	0x1fff	
		clk2	Clock Input	Double	**clk_0**			
		reset2	Reset Input	Double	[clk2]			
✓		**nios2**	Nios II (Classic) Processor					
		clk	Clock Input	Double	**clk_0**			
		reset_n	Reset Input	Double	[clk]			
		data_master	Avalon Memory Mapped Master	Double	[clk]			
		instruction_master	Avalon Memory Mapped Master	Double	[clk]			
		d_irq	Interrupt Receiver	Double	[clk]	IRQ 0	IRQ 31	
		jtag_debug_module_reset	Reset Output	Double	[clk]			
		jtag_debug_module	Avalon Memory Mapped Slave	Double	[clk]	0x0800	0x0fff	
		custom_instruction_master	Custom Instruction Master	Double				

Figure 9. The Platform Designer system with connections.

Before you can generate the Platform Designer system, You notice that there are still some errors in the message box. Platform Designer requires that you resolve the base addresses of all Avalon MM slave interfaces in the Platform Designer system. You can either use the auto-assign feature, or specify the base addresses manually. To use the auto-assign feature, on the **System** menu, click **Assign Base Addresses**. Figure 10 shows the desired **Address Map** tab of the Platform Designer system after you have auto-assigned the base address. Make any changes needed to ensure that all addresses match those in Figure 10 before continuing.

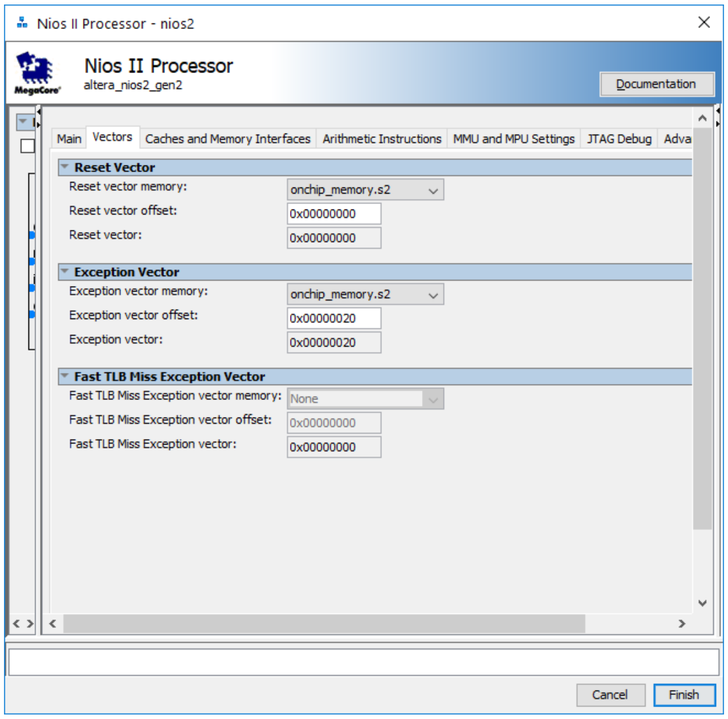


	pcie_ip.bar1_0	pcie_ip.bar2	dma.read_master	dma.write_master	sgdma.descriptor_read
pcie_ip.txs			0x8000_0000 - 0xffff_ffff	0x8000_0000 - 0xffff_ffff	0x8000_0000 - 0xffff_ffff
pcie_ip.cra		0x0000_0000 - 0x0000_3fff			
dma.control_port_slave		0x0000_4000 - 0x0000_401f			
sgdma.csr		0x0000_4040 - 0x0000_407f			
onchip_memory2.s1	0x0000_0000 - 0x0000_1fff		0x0000_0000 - 0x0000_1fff	0x0000_0000 - 0x0000_1fff	
onchip_memory2.s2					
nios2.jtag_debug_module					

	sgdma.descriptor_write	sgdma.m_read	sgdma.m_write	nios2.data_master	nios2.instruction_master
pcie_ip.txs	0x8000_0000 - 0xffff_ffff	0x8000_0000 - 0xffff_ffff	0x8000_0000 - 0xffff_ffff		
pcie_ip.cra					
dma.control_port_slave					
sgdma.csr					
onchip_memory2.s1		0x0000_0000 - 0x0000_1fff	0x0000_0000 - 0x0000_1fff		
onchip_memory2.s2				0x0000 - 0x1fff	0x0000 - 0x1fff
nios2.jtag_debug_module				0x2800 - 0x2fff	0x2800 - 0x2fff

Figure 10. The Platform Designer Address Map.

Also, you have to define the behaviour of the Nios II processor. Double-click on the *nios2* component and then select **onchip_memory.s2** to be the memory device for both reset vector and exception vector, as shown in Figure 11. The reset vector is the location in the memory device the processor fetches the next instruction when it is reset. Similarly, the exception vector is the memory address the processor goes to when an interrupt is raised.



Reset Vector
Reset vector memory: onchip_memory.s2
Reset vector offset: 0x00000000
Reset vector: 0x00000000

Exception Vector
Exception vector memory: onchip_memory.s2
Exception vector offset: 0x00000020
Exception vector: 0x00000020

Fast TLB Miss Exception Vector
Fast TLB Miss Exception vector memory: None
Fast TLB Miss Exception vector offset: 0x00000000
Fast TLB Miss Exception vector: 0x00000000

Figure 11. The Settings for Nios II processor.

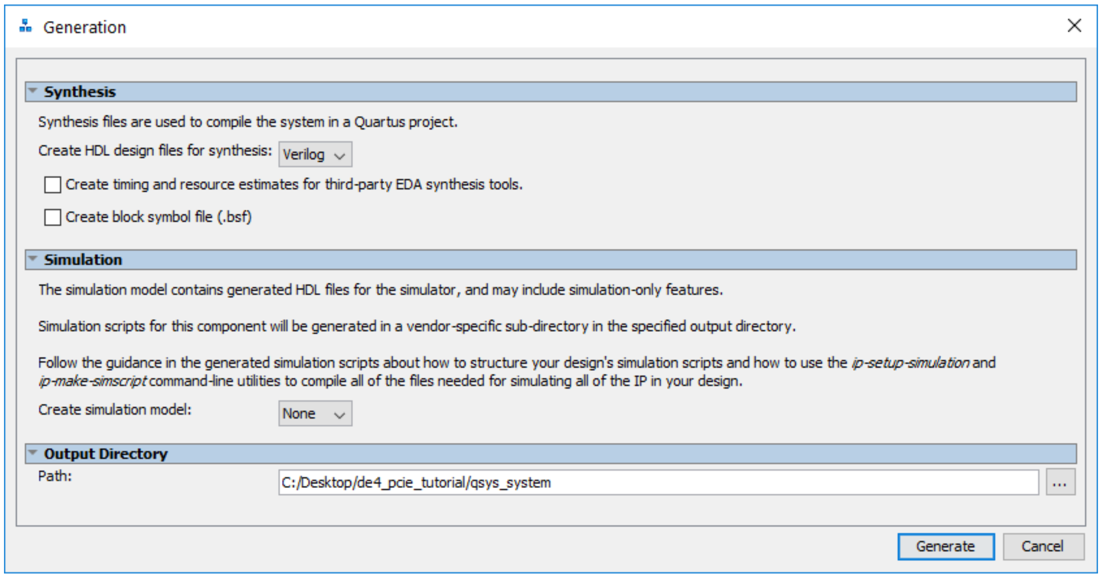
Now, the error messages should disappear and your Platform Designer system is finished. To generate the system, perform the following:

1. Select **Generate > Generate HDL**, uncheck the **Create block symbol file (.bsf)** in the **Synthesis** section as shown in Figure 12.

2. Click the **Generate** button at the bottom of the tab.

3. After Platform Designer reports **Generate Completed** in the **Generate** progress box title, click **Close**.

4. On the **File** menu, click **Save** and then close the Platform Designer application.



Synthesis files are used to compile the system in a Quartus project.

Create HDL design files for synthesis: Verilog

☐ Create timing and resource estimates for third-party EDA synthesis tools.

☐ Create block symbol file (.bsf)

The simulation model contains generated HDL files for the simulator, and may include simulation-only features.

Simulation scripts for this component will be generated in a vendor-specific sub-directory in the specified output directory.

Follow the guidance in the generated simulation scripts about how to structure your design's simulation scripts and how to use the *ip-setup-simulation* and *ip-make-simscript* command-line utilities to compile all of the files needed for simulating all of the IP in your design.

Create simulation model: None

Path: C:/Desktop/de4_pcie_tutorial/qsys_system

Figure 12. The Platform Designer Generation tab.

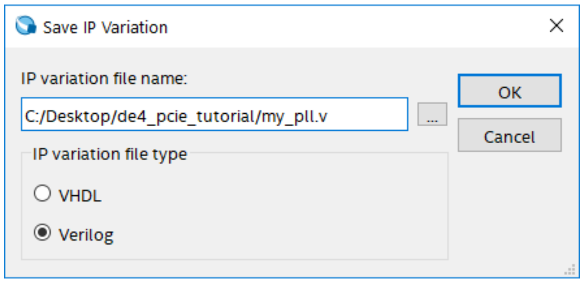
3.2 Adding PLL Using IP Catalog

Besides the Platform Designer system, you need to add a Phase-locked loop (PLL) block, which is not added as a Platform Designer component. The PLL block is used to provide different clock signals for the Platform Designer system. To add the PLL block, perform the following:

1. Select **Tools** > **IP Catalog**.

2. In the pop-up panel, select **Library > Basic Functions > Clocks; PLLs and Resets > PLL > ALTPLL** and click **Add...** button.

3. Choose the **Verilog** as the output file type for your design, and specify a variation name for output files. For this walkthrough, specify *my_pll.v* as the name of the IP core file: <working_dir>/my_pll.v, as shown in Figure 13, and click **OK**.

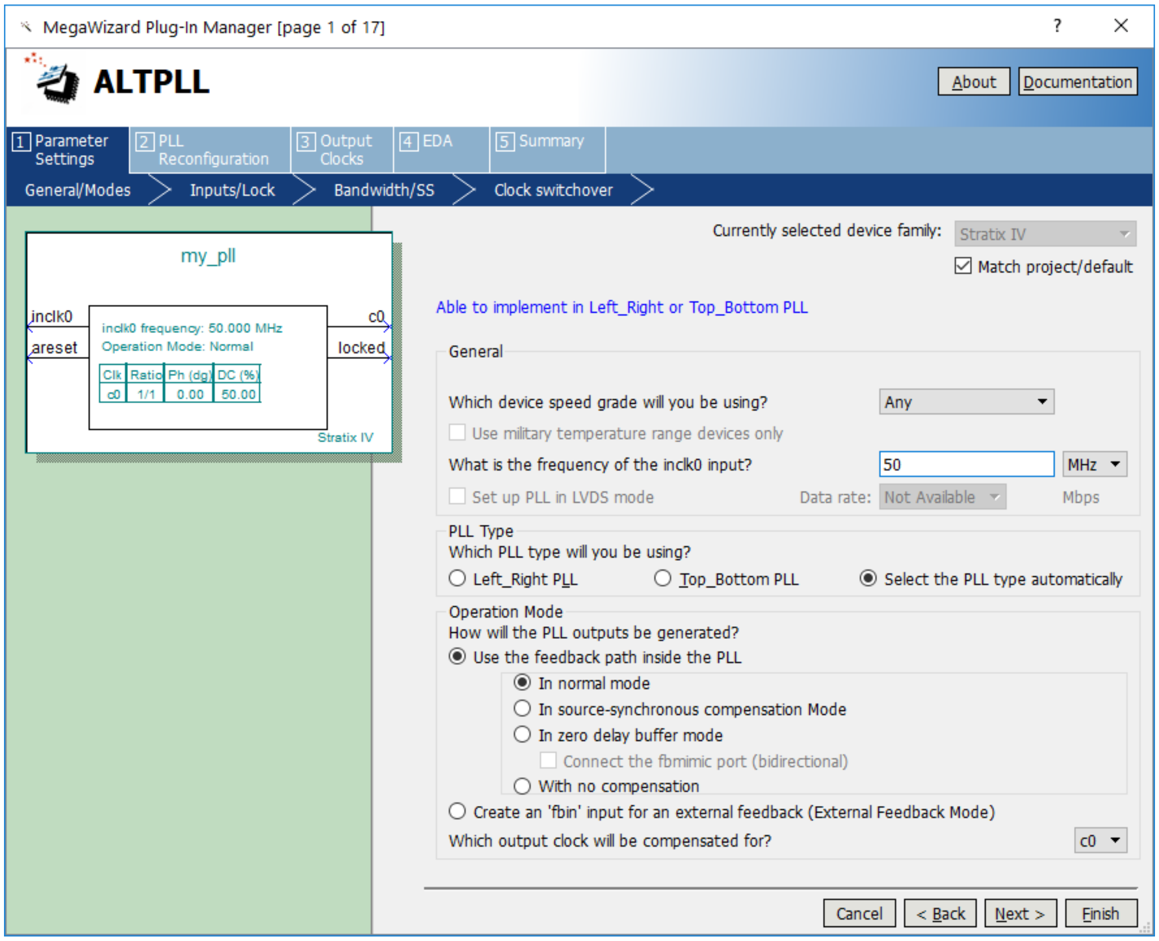

Save IP Variation

IP variation file name: C:/Desktop/de4_pcie_tutorial/my_pll.v

IP variation file type: VHDL [✓ Verilog]

Figure 13. The IP Core Plug In Manager.

4. Under the **Parameter Settings** tab, set **What is the frequency of the inclk0 input** to 50MHz as shown in Figure 14 and click **Next** to go to the **Inputs/Lock** section.


MegaWizard Plug-In Manager [page 1 of 17]

ALTPLL

Clk	Ratio	Ph (dg)	DC (%)
c0	1/1	0.00	50.00

Currently selected device family: Stratix IV

[✓] Match project/default

Able to implement in Left_Right or Top_Bottom PLL

General

Which device speed grade will you be using? Any

[ ] Use military temperature range devices only

What is the frequency of the inclk0 input? 50 MHz

[ ] Set up PLL in LVDS mode Data rate: Not Available Mbps

PLL Type

Which PLL type will you be using?

Left_Right PLL Top_Bottom PLL [✓ Select the PLL type automatically]

Operation Mode

How will the PLL outputs be generated?

[✓ Use the feedback path inside the PLL]

[✓ In normal mode]

In source-synchronous compensation Mode

In zero delay buffer mode

[ ] Connect the fbmimic port (bidirectional)

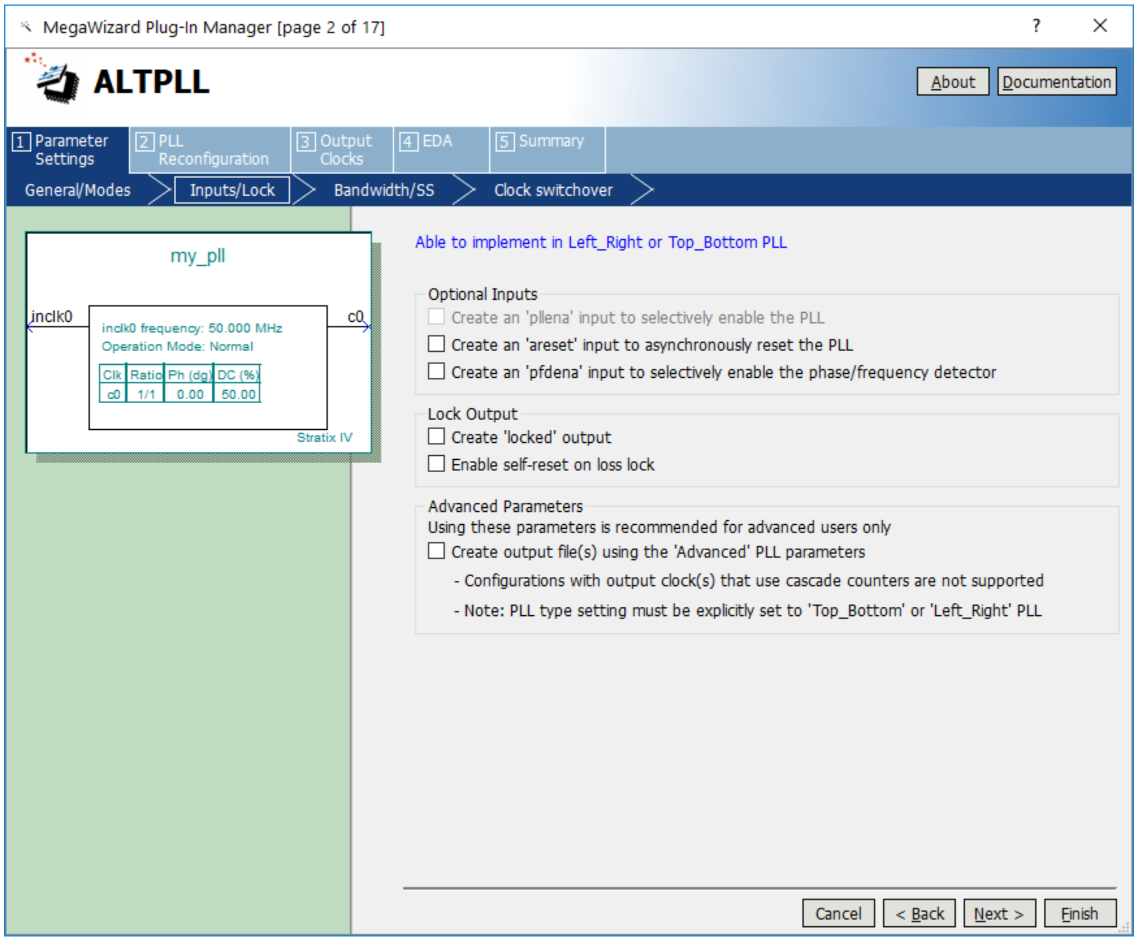
With no compensation

Create an 'fbin' input for an external feedback (External Feedback Mode)

Which output clock will be compensated for? c0

Figure 14. Settings for General/Mode section.

5. Uncheck the checkboxes of **Create an ‘areset’ input to asynchronously reset the PLL** and **Create ‘locked’ output** as shown in Figure 15.



inclk0 frequency: 50.000 MHz
Operation Mode: Normal

Clk	Ratio	Ph (dg)	DC (%)
c0	1/1	0.00	50.00

Able to implement in Left_Right or Top_Bottom PLL

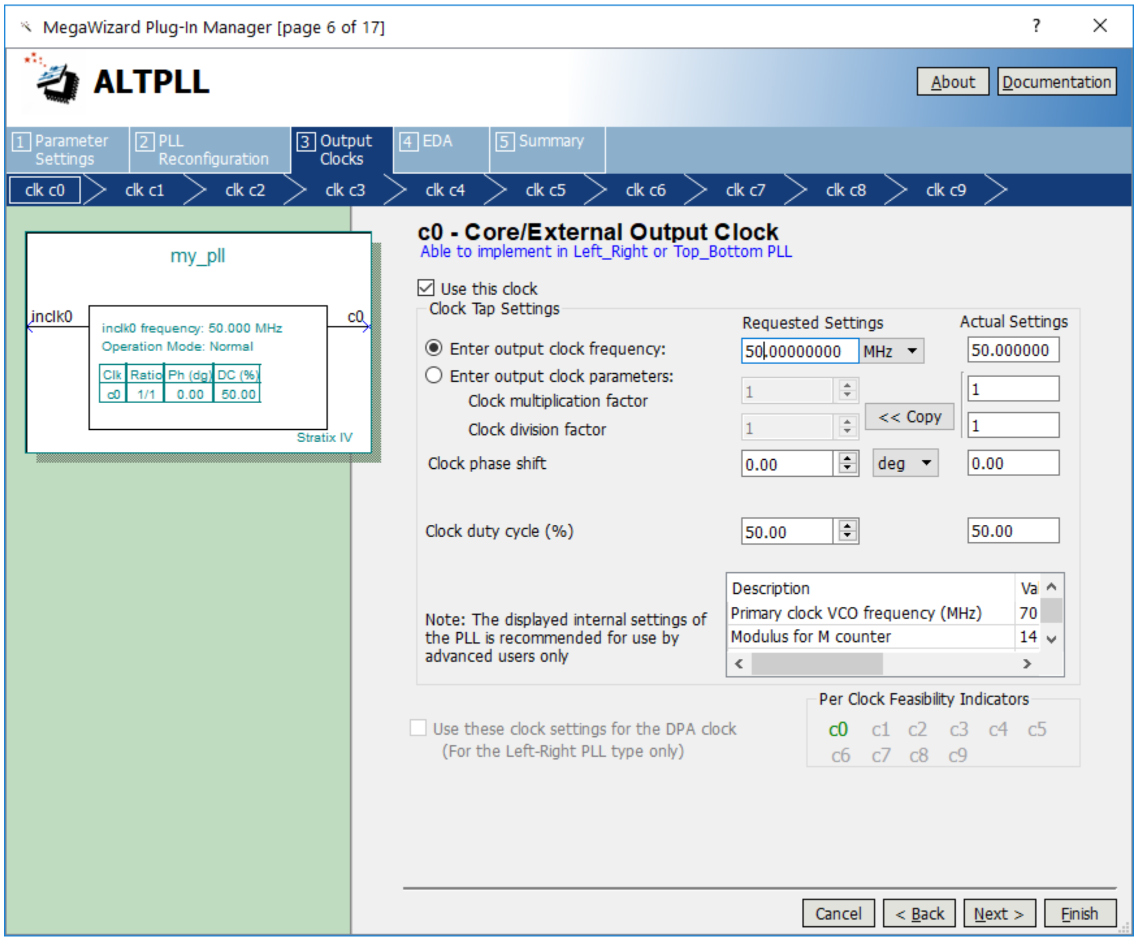
Optional Inputs
- [ ] Create an 'pllena' input to selectively enable the PLL
- [ ] Create an 'areset' input to asynchronously reset the PLL
- [ ] Create an 'pfdena' input to selectively enable the phase/frequency detector

Lock Output
- [ ] Create 'locked' output
- [ ] Enable self-reset on loss lock

Advanced Parameters
Using these parameters is recommended for advanced users only
- [ ] Create output file(s) using the 'Advanced' PLL parameters
  - Configurations with output clock(s) that use cascade counters are not supported
  - Note: PLL type setting must be explicitly set to 'Top_Bottom' or 'Left_Right' PLL

Figure 15. Settings for Input/Lock section.

6. Under the **Output Clocks** tab, enter 50MHz after selecting **Enter output clock frequency** as shown in Figure 16, and click **Next** to go to the **clk_c1** section.



inclk0 frequency: 50.000 MHz
Operation Mode: Normal

Clk	Ratio	Ph (dg)	DC (%)
c0	1/1	0.00	50.00

c0 - Core/External Output Clock
Able to implement in Left_Right or Top_Bottom PLL

☑ Use this clock

Clock Tap Settings

	Requested Settings		Actual Settings
● Enter output clock frequency:	50.00000000	MHz	50.000000
○ Enter output clock parameters:			
Clock multiplication factor	1	<< Copy	1
Clock division factor	1		1
Clock phase shift	0.00	deg	0.00
Clock duty cycle (%)	50.00		50.00

Note: The displayed internal settings of the PLL is recommended for use by advanced users only

Description	Va
Primary clock VCO frequency (MHz)	70
Modulus for M counter	14

☐ Use these clock settings for the DPA clock (For the Left-Right PLL type only)

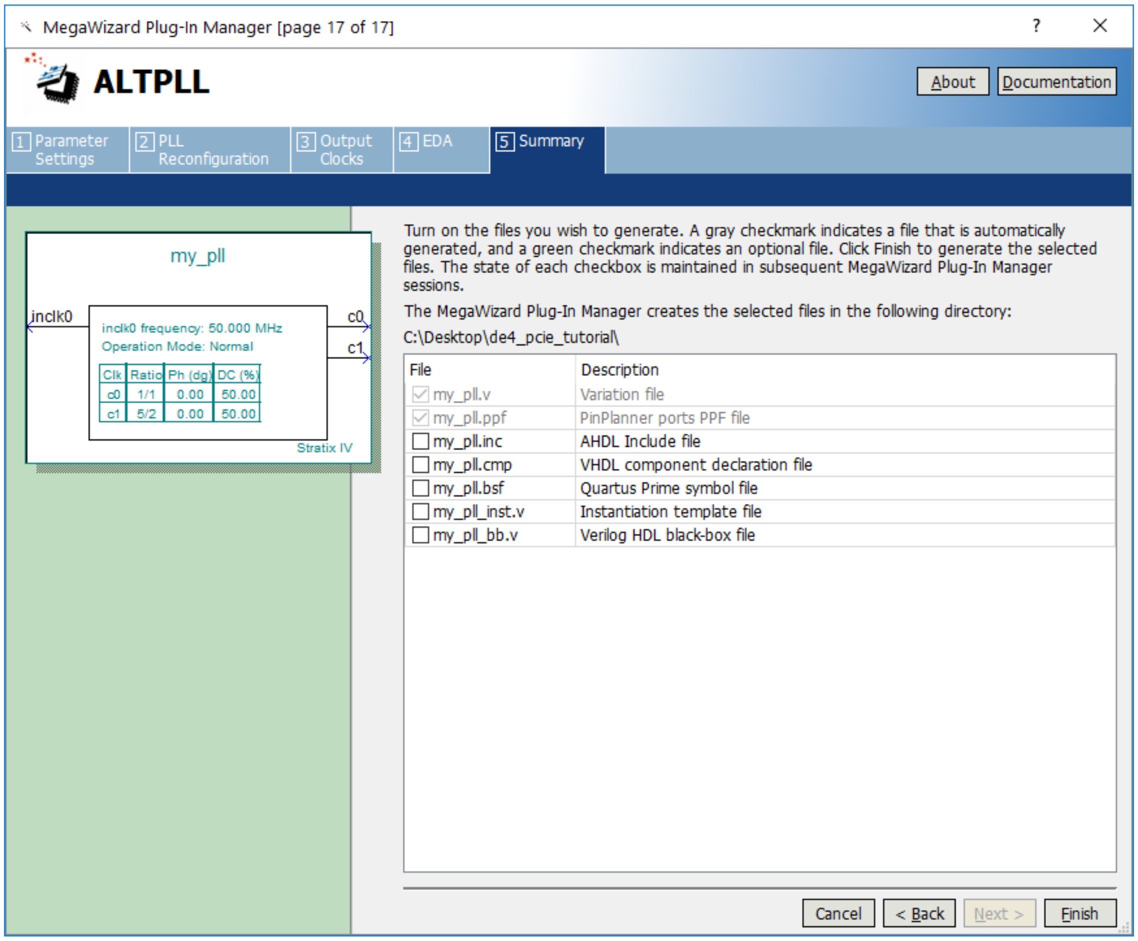
Per Clock Feasibility Indicators
c0 c1 c2 c3 c4 c5
c6 c7 c8 c9

Figure 16. Settings for output c0.

7. Select the checkbox of **Use this clock** and then set the output clock frequency to 125MHz with the same procedures of the last step, and click **Next** to go to the **clk_c2** section.

8. Select the checkbox of **Use this clock** and then set the output clock frequency to 100MHz with the same procedures of the last step.

9. Under the **Summary** tab, uncheck **my_pll_bb.v** as shown in Figure 17, and then click **Finish** to generate the file.



Clk	Ratio	Ph (dg)	DC (%)
c0	1/1	0.00	50.00
c1	5/2	0.00	50.00

Turn on the files you wish to generate. A gray checkmark indicates a file that is automatically generated, and a green checkmark indicates an optional file. Click Finish to generate the selected files. The state of each checkbox is maintained in subsequent MegaWizard Plug-In Manager sessions.

The MegaWizard Plug-In Manager creates the selected files in the following directory:

C:\Desktop\de4_pcie_tutorial\

File	Description
✓ my_pll.v	Variation file
✓ my_pll.ppf	PinPlanner ports PPF file
☐ my_pll.inc	AHDL Include file
☐ my_pll.cmp	VHDL component declaration file
☐ my_pll.bsf	Quartus Prime symbol file
☐ my_pll_inst.v	Instantiation template file
☐ my_pll_bb.v	Verilog HDL black-box file

Figure 17. Settings for Summary tab.

3.3 Integrating modules into the Quartus® Prime Project

To complete your hardware design, perform the following:

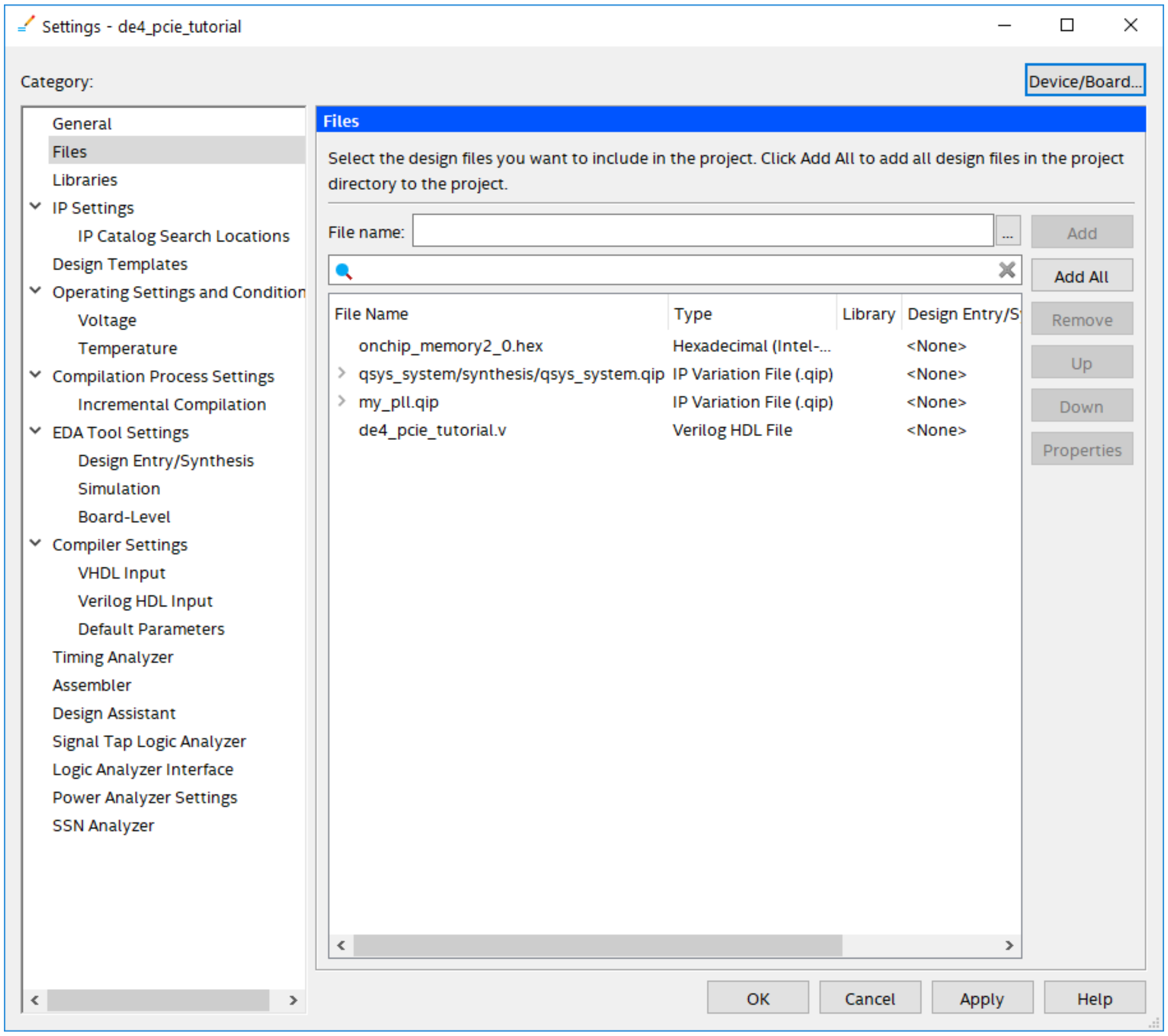
1. Type the code in Figure 18 into a file called *de4_pcie_tutorial.v*. This code is a top-level Verilog module that instantiates the Platform Designer system and the PLL block. The module is named *de4_pcie_tutorial*, because this is the name we specified in Figure 2 for the top-level design entity in the Quartus Prime project.

```
module de4_pcie_tutorial(
    input OSC_50_BANK2,
    input PCIE_PREST_n,
    input PCIE_REFCLK_p,
    input [3:0] PCIE_RX_p,
    output [3:0] PCIE_TX_p,
    output [7:0] LED
);

assign LED[7] = 1'b1;
assign LED[3] = 1'b1;
wire clk50, clk125, clk100;
my_pll pll_inst(
    .inclk0 (OSC_50_BANK2),
    .c0 (clk50),
    .c1 (clk125),
    .c2 (clk100)
);

qsys_system system_inst(
    .clk_clk (clk50),
    .pcie_ip_refclk_export (PCIE_REFCLK_p),
    .pcie_ip_fixedclk_clk (clk125),
    .reset_reset_n (1'b1),
    .pcie_ip_pcie_rstn_export (PCIE_PREST_n),
    .pcie_ip_rx_in_rx_datain_0 (PCIE_RX_p[0]),
    .pcie_ip_rx_in_rx_datain_1 (PCIE_RX_p[1]),
    .pcie_ip_rx_in_rx_datain_2 (PCIE_RX_p[2]),
    .pcie_ip_rx_in_rx_datain_3 (PCIE_RX_p[3]),
    .pcie_ip_tx_out_tx_dataout_0 (PCIE_TX_p[0]),
    .pcie_ip_tx_out_tx_dataout_1 (PCIE_TX_p[1]),
    .pcie_ip_tx_out_tx_dataout_2 (PCIE_TX_p[2]),
    .pcie_ip_tx_out_tx_dataout_3 (PCIE_TX_p[3]),
);
endmodule
```

Figure 18. Verilog code for the top-level module

2. Add the *de4_pcie_tutorial.v* file, the *qsys_system.qip* file, the *my_pll.qip* file, and the *onchip_memory2_0.hex* file to your Quartus Prime project as shown in Figure 19. The *onchip_memory2_0.hex* file is used to initialize the on-chip memory and can be found in the directory *design_files*, which can be downloaded along with this tutorial from the Intel FPGA University Program website. It contains the executable file of the demo application for the Nios II processor, which will be used in the next section.

3. Add the necessary pin assignments on the DE4 board to your project. Note that an easy way of making the pin assignments when you use the same pin names as in the DE4 User Manual is to import the assignments from file. The pin assignments can be found in the *DE4_230_pin_assignments.qsf* or the *DE4_530_pin_assignments.qsf* file (depending on your board), in the directory *design_files*.


Settings - de4_pcie_tutorial

Category:

Files

Select the design files you want to include in the project. Click Add All to add all design files in the project directory to the project.

File name:

File Name	Type	Library	Design Entry/S
onchip_memory2_0.hex	Hexadecimal (Intel-...		<None>
qsys_system/synthesis/qsys_system.qip	IP Variation File (.qip)		<None>
my_pll.qip	IP Variation File (.qip)		<None>
de4_pcie_tutorial.v	Verilog HDL File		<None>

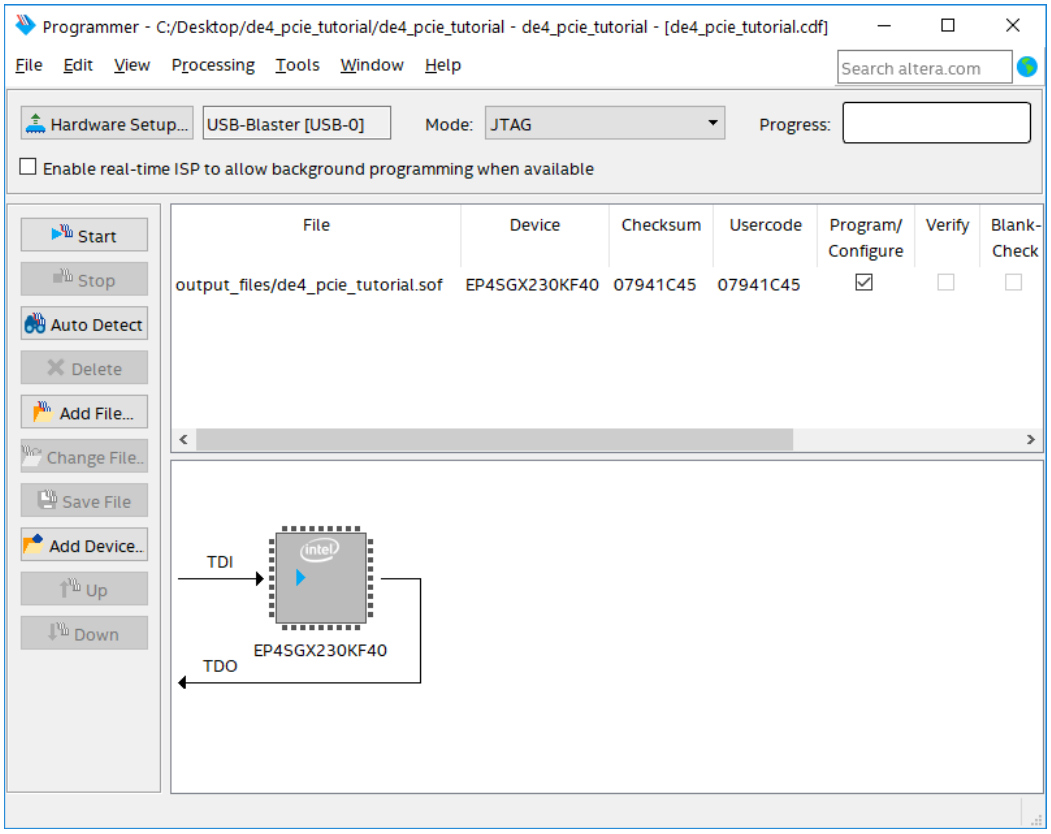
Figure 19. Files settings for the Quartus Project

4. Compile the Quartus Prime project.

3.4 Programming and Configuration

Before you can configure the DE4 board, you should plug the DE4 board into the PCI Express port of your Linux computer and connect the board to the computer by means of a USB cable plugged into the USB-Blaster port. Then turn on the power of the board and program the FPGA in the JTAG* programming mode as follows:

1. Select **Tools > Programmer** to reach the window in Figure 20.
2. If the USB-Blaster is not chosen by default, press the **Hardware Setup...** button and select the USB Blaster in the window that pops up.


Programmer - C:/Desktop/de4_pcie_tutorial/de4_pcie_tutorial - de4_pcie_tutorial - [de4_pcie_tutorial.cdf]
File Edit View Processing Tools Window Help
Search altera.com
Hardware Setup...
Start
Stop
Auto Detect
Delete
Add File...
Change File...
Save File
Add Device...
Up
Down
TDI
TDO
EP4SGX230KF40

Hardware Setup: USB-Blaster [USB-0] Mode: JTAG Progress:

☐ Enable real-time ISP to allow background programming when available

File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank-Check
output_files/de4_pcie_tutorial.sof	EP4SGX230KF40	07941C45	07941C45	☑	☐	☐

Figure 20. The Programmer window.

3. The configuration file *de4_pcie_tutorial.sof* should be listed in the window. If the file is not already listed, then click **Add File** and select it from the *output_files* subdirectory of the project.

4. At this point the window settings should appear as indicated in Figure 20. Press **Start** to configure the FPGA.

After you have successfully configured the hardware in the FPGA device, you should reboot your Linux computer to let the operating system to detect the PCI Express device you built. Ensure that the DE4 board is powered.

After the reboot, type command *lspci* in the terminal to ask the operating system to list all the PCI devices on the computer. You should be able to see an Intel *0xde4* device shown on the terminal.

4 Using the Driver

In the previous section, you have built a hardware system that supports the PCI Express; however, you need a driver to use the PCI Express to communicate with the DE4 board. Intel FPGA University Program provides an open source PCI driver for the usage of DE4 board in Linux operating system. The driver is designed to meet the general needs of using the PCI Express. It allows a user to:

1. create a configuration file to define the hardware specific information for the driver.
2. access any Base Address Registers (BAR).
3. use DMA Controller and Scatter-Gather DMA Controller.
4. use either polling or interrupt for DMA transfers.

The driver is in the directory *design_files/driver*. The driver is tested on Linux 2.6.32, but it should work with newer versions of the Linux kernel.

4.1 Creating the Configuration File

Before you can use the driver, you need to create a file to configure the driver for your hardware system. An example configuration file, *config_file_example*, can be found in the *driver* folder. To make the file for the system you built, open *config_file_example* and perform the following:

1. Set the Vendor ID and Device ID. They should be the same with the IDs shown in Figure 4.

   vendor_id = 0x1172

   device_id = 0x0de4

2. Set the parameters related to the *PCI Express IP Core*.

   pci_dma_bit_range = 31 This defines the bit mask for DMA buffers. By setting it to 31, you force Linux to allocate physical address between 0x00000000 − 0x7FFFFFFF for DMA buffers. This parameter is decided by the settings of **Address Translation** in the IP Compiler for PCI Express.

   tx_base_addr = 0x80000000 The base address of the *txs* Avalon MM slave.

   pcie_cra_bar_no = 2 This defines the number of the BAR which is connected to the *cra* Avalon MM slave.

   pcie_cra_base_addr = 0x00000000 The base address of *cra* Avalon MM slave.

3. Set the parameters for DMA controllers. By default, the driver supports at most four DMA controllers, so there are four columns in the *config_file_example*. You only need to fill in the first two columns for the *DMA Controller* and the *Scatter-Gather DMA Controller* in your system, and leave other columns as zeros.

   dma_type = 1, 2 This defines the type of the DMA controller. The *DMA Controller* is represented by 1, while the *Scatter-Gather DMA Controller* is represented by 2.

   dma_irq_no = 1, 2 This defines the interrupt number of DMA controller as shown Figure 9.

   dma_ctrl_bar_no = 2, 2 This defines the number of the BAR, which is connected to the control port of the DMA controller.

   dma_ctrl_base_addr = 0x00004000, 0x00004040 This defines the base address of the control port.

4. Set the data width of the allowed transactions for *DMA Controller*. This is necessary if you are using the *DMA Controller*.

sdma_data_width = 8, 0 Set the first number to 8 (words) to match the settings in Figure 5 and ignore it for the *Scatter-Gather DMA Controller*.

5. Save the file as tutorial_config_file.

4.2 Load the Driver Module into the Kernel

To install the driver, you need to load the *.ko* module file into the Linux kernel. If it's your first time to use the driver, you have to compile the driver codes into the *.ko* module file. Run the makefile in the folder *driver* to compile the driver. Note that you may have to change the file format from the Windows* Format to the UNIX* Format first, so go to the demo and driver folders before compiling any programs, type:

```
dos2unix *
```

After you compiled the driver, you will get the kernel module file *alt_up_pci.ko*. The name of the module file is determined by the makefile and you are recommended not to change the name. Once you have the *.ko* module file, you don't have to recompile the driver for your future usage of the driver.

Each time you use the driver, you will have to dynamically load the driver module with the correct configuration file. This is done by a shell script *load_alt_up_pci_driver.sh* provided in the *driver* folder. Load the driver module with the configuration file you created by typing:

```
sudo ./load_alt_up_pci_driver.sh tutorial_config_file
```

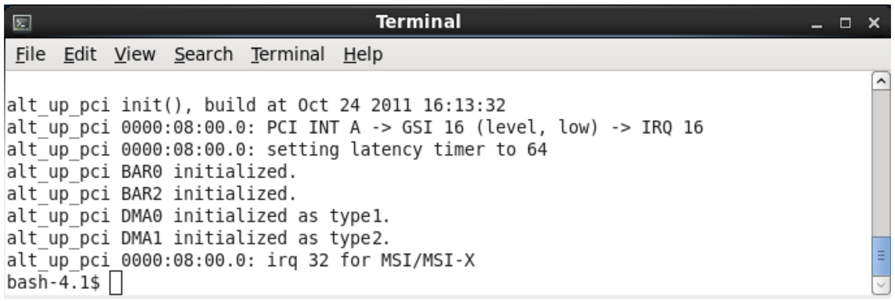
Note that you need to pass the file name of the configuration file as the first argument to the shell script, otherwise the script will use *config_file* as default. If you have changed the makefile when compiling the driver codes, you have to modify the variable *DRV_MODULE* in the shell script to match the name of your *.ko* module file.

This script requires that your account has the administrator right. After typing in the password, you will see *Matching Device Found* on success, or *Matching Device Not Found* when failed. If the module is successfully loaded, the script will automatically create a character special file *alt_up_pci0* in directory **/dev/**. This file is important because it represents the DE4 device and you will need to access it when you want to use the driver.

Since the driver is loaded into the kernel, the information printed out by the driver will not show on the terminal window directly. Instead, they will be printed as driver messages. To see the driver messages, type command:

```
dmesg
```

The driver messages are important messages to check the status of the driver especially when you encounter problems. You should be able to see similar driver messages shown in Figure 21 after you have loaded the driver module successfully.



```
alt_up_pci init(), build at Oct 24 2011 16:13:32
alt_up_pci 0000:08:00.0: PCI INT A -> GSI 16 (level, low) -> IRQ 16
alt_up_pci 0000:08:00.0: setting latency timer to 64
alt_up_pci BAR0 initialized.
alt_up_pci BAR2 initialized.
alt_up_pci DMA0 initialized as type1.
alt_up_pci DMA1 initialized as type2.
alt_up_pci 0000:08:00.0: irq 32 for MSI/MSI-X
bash-4.1$
```

Figure 21. Messages shown on the terminal

If you want to remove the module from the kernel, you can use the shell script below:

```
sudo ./unload_alt_up_pci_driver.sh
```

4.3 How to Use the Driver

To help you understand how to use the driver, a demo application is provided. The application will ask you to input a sentence and the sentence will be sent to the on-chip memory of the DE4 board. Then the Nios II processor will change the sentence, modifying letters from lower-case to upper-case and vice versa. After the processing, the application will read the sentence back and print it on the screen. To ensure the data is shared correctly between the PC host and the DE4 board, one byte (0x00000FFF) in the on-chip memory is used to as a flag to indicate who owns the control of the data. ‘H’ means the data is owned by the host while ‘B’ means it’s owned by the board. When the data is owned by one side, the other side should do nothing other than polling the control byte.

Figure 22 shows the simplified code of the demo application, which does not contain error checking. You can find the complete code *demo.c* in directory *design_files/demo*.

```c
#include <stdio.h>
#include <string.h>
#include "alt_up_pci_lib.h"

#define ONCHIP_CONTROL 0x00000FFF
#define ONCHIP_DATA 0x00001000
#define MAX_DATA_SIZE 4096
#define CTRLLER_ID 0

int main() {
    int fd, length_str;
    char buff[MAX_DATA_SIZE], control;

    alt_up_pci_open( &fd, "/dev/alt_up_pci0" );

    control = 'H'; // controlled by the host PC
    alt_up_pci_write( fd, BAR0, ONCHIP_CONTROL, &control, sizeof(control) );

    while(1) {
        fgets( buff, MAX_DATA_SIZE, stdin );
        length_buff = strlen(buff) + ( 8 0 - (strlen(buff)%8) );

        alt_up_pci_dma_add(fd, CTRLLER_ID, ONCHIP_DATA, buff, length_str,
           TO_DEVICE);
        alt_up_pci_dma_go (fd, CTRLLER_ID, INTERRUPT);

        control = 'B'; // pass the control right to the DE4 board
        alt_up_pci_write (fd, BAR0, ONCHIP_CONTROL, &control, sizeof(control) );

        while( control != 'H') // polling the control byte
            alt_up_pci_read( fd, BAR0, ONCHIP_CONTROL, &control, sizeof(control) );

        alt_up_pci_dma_add(fd, CTRLLER_ID, ONCHIP_DATA, buff, length_str,
           FROM_DEVICE);
        alt_up_pci_dma_go(fd, CTRLLER_ID, INTERRUPT) ;

        printf("Received : \n%s\n", buff);
    }

    alt_up_pci_close(fd);

    return 0;
}
```

Figure 22. Simplified code for demo application

As you can see in Figure 22, there are six functions prefixed with *alt_up_pci_*. These functions are provided in the header file *alt_up_pci_lib.h*. The descriptions of the functions are shown below. Note that all these functions will return 0 on success, and return −1 when failed.

alt_up_pci_open() – This function is used to open the DE4 device file.

You will have to input the character special file of the device, which in this case is *alt_up_pci0*, then you will get a file descriptor. After calling this function successfully, you can start using other functions to perform operations on the DE4 board.

alt_up_pci_close() – This function used to close the DE4 device file.

You will have to pass the file descriptor got from *alt_up_pci_open()* to the function. This function should be called at the end of the application.

alt_up_pci_read() – This function is used to do the read operation.

You will have to select which BAR to read from and what is the starting address of the read operation. Also, you need to pass a pointer to the buffer and the size of the buffer to the function.

alt_up_pci_write() – This function is used to do the write operation.

This function is similar to *alt_up_pci_read()* and does the write operation instead.

alt_up_pci_dma_add() – This function is used to add a DMA transfer into the queue of the DMA controller.

Inside the driver, there is a queue for each DMA controller. You are allowed to push up to ten DMA transfers into the queue before the DMA controller really performs the transfers. You will need to pick a DMA controller to perform the transfer, and give it the pointer to the buffer, the address in the DE4 board, how many bytes you want to transfer and the direction of the transfer. Note that the driver will not check the overflow of the address, so the application is responsible to ensure that the DMA controller is not reading from or writing to an illegal address.

alt_up_pci_dma_go() – This function is used to start the DMA transfers in the queue.

By calling this function, the selected DMA controller will start performing all the DMA transfers in the queue, and it will use either polling or interrupt to check whether a transfer is finished.

The header file *alt_up_pci_lib.h* also contains three enum types for the BARs, the direction of the DMA, and the method of checking to increase the readability of the codes. We recommend you to include this file when you are writing your own application.

Pay attention to the transfer length when using the *DMA Controller*, because the data length have to match the settings you made for the *DMA Controller*. In this tutorial, an error will occur, if the length is not a multiple of 8 or the address is not aligned with 8 bytes.

4.4 Running the Demo Application

To try the demo application, compile and run the demo codes in the *demo* folder. By typing sentence into the terminal and typing **Enter** to send the sentence, you can see how the sentence is changed by the Nios II processor. You can also change the value of *CTRLLER_ID* in the *demo.c* to select a different DMA controller.

The code shown in Figure 23 is the code run by the Nios II processor. It is downloaded to the on-chip memory by adding the *onchip_memory2_0.hex* during the compile time.

```c
#include <stdio.h>

volatile char *onchip_control = (char *) 0x00000FFF;
volatile char *onchip_data = (char *) 0x00001000;

int main() {
    int i;
    char ch;

    while( 1 ) {
        while( *onchip_control != 'B' ) // polling the control byte
            ;

        i = 0;
        while( (ch = *(onchip_data + i) ) != '\n' ) {
            if ( ch >= 'a' && ch <= 'z' )
                *(onchip_data + i) = ch - 'a' + 'A';
            else if ( ch >= 'A' && ch <= 'Z' )
                *(onchip_data + i) = ch - 'A' + 'a';
            else
                *(onchip_data + i) = ch;
            i++;
        }

        *onchip_control = 'H' ; // pass the control right to the host PC
    }
    return 0;
}
```

Figure 23. Program run by the Nios II Processor

4.5 Reprogramming the FPGA without Rebooting the Host Computer

After you successfully configure the FPGA and install the driver on your host computer, you may still want to modify your PCI Express settings on the FPGA and reprogram it. However, reprogramming the FPGA will disable the connection between the FPGA and the host computer. It requires you to reboot the host computer and reload the driver every time you download a new configuration onto the FPGA, which is inconvenient and time-consuming. Here we provide a way to reprogram the FPGA without rebooting the host computer if a valid PCI Express connection has already been established:

1. To save the PCI Express registers, go to the demo folder, run:

   ./reprogram save

2. Program a new image(.sof) on FPGA following the instructions in section 3.4. Note that the memory space, the Vendor ID and the Device ID of the PCI Express shouldn't be changed in Platform Designer, otherwise restoring the register information will give the wrong information to the host computer.

3. Then we need to restore the registers information, go to the demo folder, run:

```
./reprogram restore
```

4. Go to the driver folder to unload and reload the driver using:

```
./unload_alt_up_pci_driver.sh
./load_alt_up_pci_driver.sh tutorial_config_file
```

The device connection should be re-established.

5 Changing the Driver

Now, you can start building your own system and applications. However, if you are not satisfied with the functionality provided by the driver and want to design your custom driver, you can read the *index.html* in directory *design_files/doc/html*. This file is generated by the Doxygen and it will give you a general idea of the driver, so that you can change it.