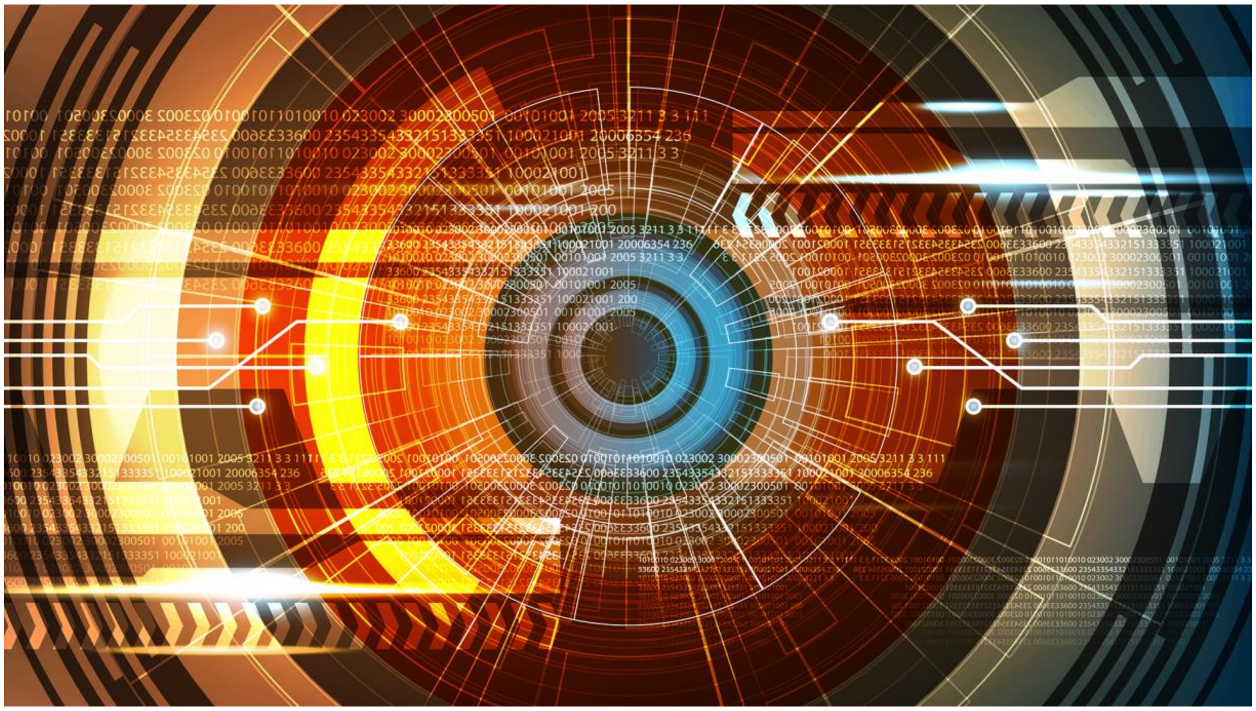


VIDEO PROCESSING WITH INTEL FPGAS WEBINAR SERIES – Q4'2020

Session 1.2 – Adding first VIP cores

Francisco Perez
Intel FPGA Field Applications Engineer
v.1 – October 2020

Contents

1. Introduction

1.1. Introduction

In this lab manual we are modifying the design generated in the previous session: "**Session 1.1 – DisplayPort Loopback Design**" by breaking the **FIFO/PCR** module and adding our own generated Video Processing Pipeline using VIP cores.

This session will cover the initial principles and we are just adding a **Test Pattern Generator** and a **Clocked Video Output** module to display some static patterns on the target monitor.

This will set the base for growing in complexity in upcoming sessions.

1.2. Requirements

On this specific implementation, we are using the following setup:

- Cyclone® 10 GX Development Kit
  https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/cyclone-10-gx-development-kit.html
- Bitec DisplayPort daughter card rev.11
  https://bitec-dsp.com/product/fmc-displayport-daughter-card-revision-11/
- Intel® Quartus Pro ACDS 20.3
  https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html
- CentOS 7.6 (but other Linux distros as well as Windows are supported)

1.3. References

The purpose of this document is to guide you through the process of creating the different building blocks and pull all together to assembly a working application. For more detailed information about all the potential combinations and settings, you can use the following resources:

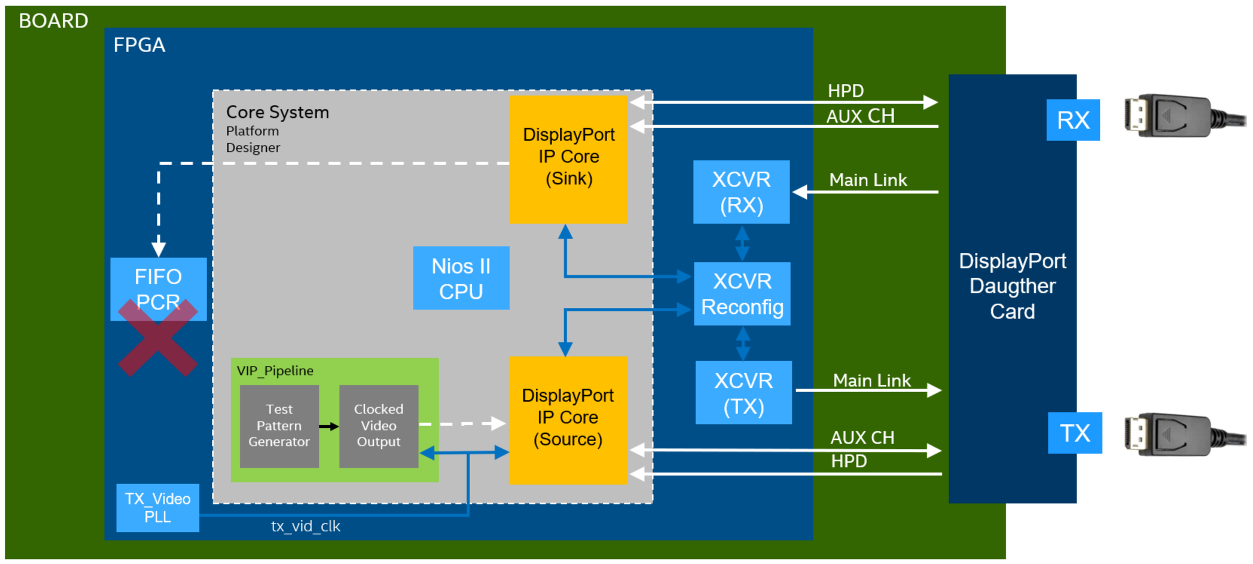
- Intel FPGA DisplayPort IP User Guide
  https://www.intel.com/content/www/us/en/programmable/products/intellectual-property/ip/interface-protocols/m-alt-displayport-megacore.html
- Cyclone 10 GX DisplayPort Design Example User Guide
  https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-dex-dp-c10gx.pdf
- VIP – Video and Image Processing User Guide
  https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_vip.pdf
- AN745-Design Guidelines for DisplayPort Interface
  https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/an/an_745.pdf
- Quartus Prime Pro Installation Guide
  https://www.intel.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/quartus_install.pdf
- NiosII EDS installation
  https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/nios2/n2sw_nii5v2gen2.pdf
- Embedded Design Handbook
  https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/nios2/edh_ed_handbook.pdf

1.4. Implementation diagram

Find in the below figure, a high-level block diagram with the hardware implementation. Inside the FPGA, we are configuring a set of high-speed transceivers to receive and transmit the DisplayPort video streams in serialized form, acting as the physical layer. Attached to them, we have the DisplayPort IP cores for Sink and Source implementation, these are our link layer blocks.

The video packets received by the Sink are just discarded, because our focus for this session is on getting a working output generator implementation.

We are adding an additional **PLL** to generate the `Output Pixel Clock frequency` and a new Platform Designer module (integrated into the Core System hierarchy) to generate the Output Video content. In this case, this content will be a collection of different static patterns to exercise with VIP IP cores.



In this design we are reusing most of the modules from the previous lab, we have included 2 new ones:

- **VIP_Pipeline subsystem** – We have created a new Platform Designer block that will be inserted into the Core System.
  - We are adding a Clocked Video Output module responsible to generate the parallel video data (RGB pixel information and synchronization signals: VSYNC/HSYNC/DE) to drive the DisplayPort IP Source core. We are configuring this block to generate a 1920x1080p60 resolution.
  - To feed the CVO module with video content, we add a Test Pattern Generator module configured to generate up to 3 different video patterns: Color Bars, Grayscale Bars and Uniform Background. The TPG is configured with width=1920 and height=1080, according to what CVO expects to receive.

- **TX_Video_PLL** – In the previous session we were recovering the pixel clock frequency, needed to drive the output, directly from the input using a PCR module. But this is not valid anymore, as we want to be able to drive our display, even in the case we don't have any video signal attached to our input. For this, we need to generate a free-run pixel clock frequency to always being able to drive the video output.
  For 1920x1080p60 resolution, the pixel clock frequency needed is 148.5MHz. We are using a Cyclone10GX IOPLL block with a 135MHz clock as reference input (we are using this as reference for the transceivers blocks).

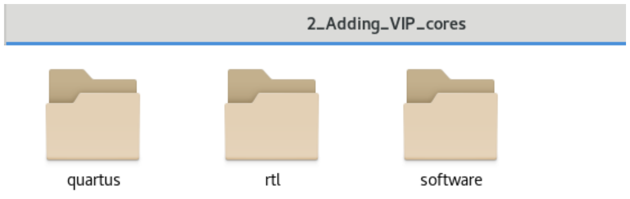
2. Generating the hardware pipeline

2.1. Setting up the Quartus project

We are using, as base project, the DisplayPort example design generated in the previous step. This project already contains the instantiations of DisplayPort sink and source, as well as other necessary blocks like PLLs and Nios II CPU.

Let's create a new folder and copy the following directories on it. I've used "**2_Adding_VIP_cores**" as folder name, but you can use your preferred one.

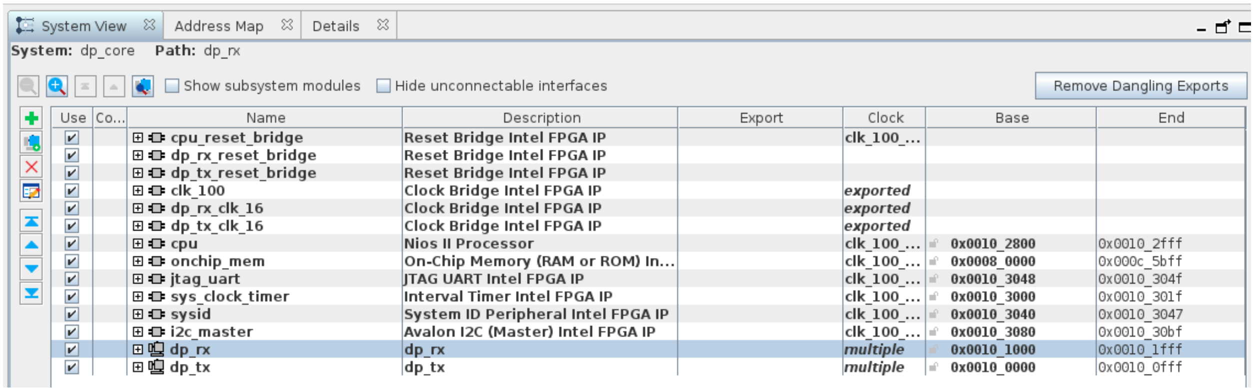
Copy the content from the previously generated "`dp_0_example_design`" folder. We just need the following ones: *`quartus`*, *`rtl`* and *`software`*.



Let's open the project in Quartus Pro and examine it

2.2. Examining the project

With the project already loaded in Quartus, open *`<project_dir>/rtl/core/dp_core.qsys`* in Platform Designer.



System: dp_core Path: dp_rx

Use	Co...	Name	Description	Export	Clock	Base	End
✓		cpu_reset_bridge	Reset Bridge Intel FPGA IP		clk_100_...		
✓		dp_rx_reset_bridge	Reset Bridge Intel FPGA IP				
✓		dp_tx_reset_bridge	Reset Bridge Intel FPGA IP				
✓		clk_100	Clock Bridge Intel FPGA IP		*exported*		
✓		dp_rx_clk_16	Clock Bridge Intel FPGA IP		*exported*		
✓		dp_tx_clk_16	Clock Bridge Intel FPGA IP		*exported*		
✓		cpu	Nios II Processor		clk_100_...	0x0010_2800	0x0010_2fff
✓		onchip_mem	On-Chip Memory (RAM or ROM) In...		clk_100_...	0x0008_0000	0x000c_5bff
✓		jtag_uart	JTAG UART Intel FPGA IP		clk_100_...	0x0010_3048	0x0010_304f
✓		sys_clock_timer	Interval Timer Intel FPGA IP		clk_100_...	0x0010_3000	0x0010_301f
✓		sysid	System ID Peripheral Intel FPGA IP		clk_100_...	0x0010_3040	0x0010_3047
✓		i2c_master	Avalon I2C (Master) Intel FPGA IP		clk_100_...	0x0010_3080	0x0010_30bf
✓		dp_rx	dp_rx		*multiple*	0x0010_1000	0x0010_1fff
✓		dp_tx	dp_tx		*multiple*	0x0010_0000	0x0010_0fff

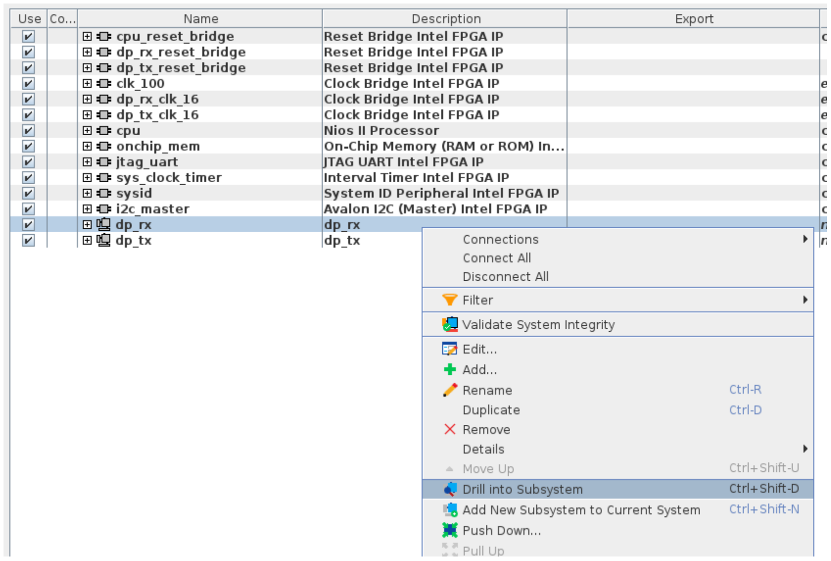
This Platform Designer system contains different IP variations:

- **Clock & Reset bridges**
- **Nios II Processor & peripherals** (onchip_mem, jtag uart, timer, i2c master, …)

It also contains another Platform Designer files as subsystems, to create hierarchical designs:

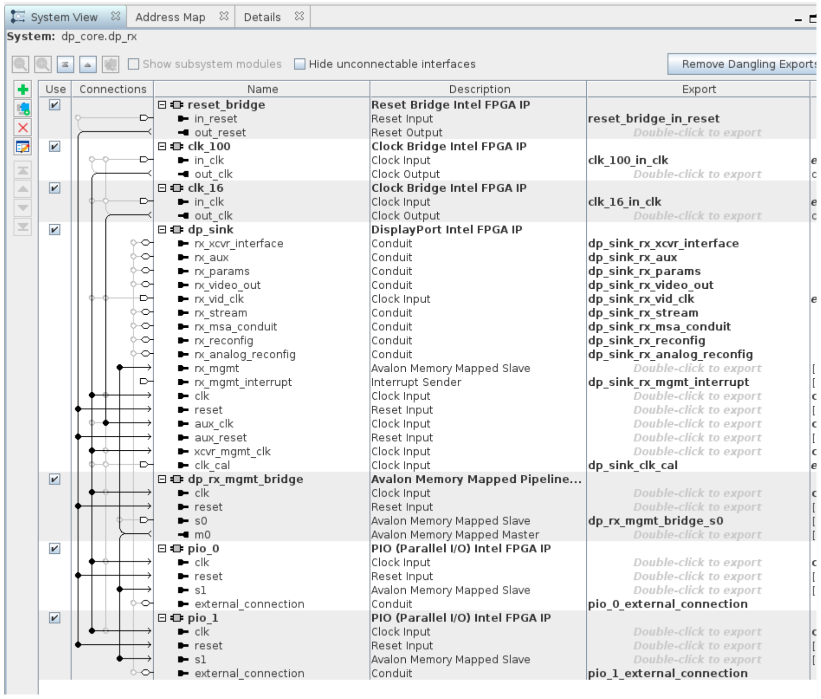
- **dp_rx** (This is a subsystem including the Displayport IP configured as Sink, with some additional clock bridges and AVMM_Pipelined_Bridge to connect the NiosII Processor)
- **dp_tx** (This is a subsystem including the Displayport IP configured as Sink, with some additional clock bridges and AVMM_Pipelined_Bridge to connect the NiosII Processor)

You can explore what is on every subsystem by just right-clicking on the block and select "*Drill into Subsystem*" option:



Use	Co...	Name	Description	Export
✓		cpu_reset_bridge	Reset Bridge Intel FPGA IP	
✓		dp_rx_reset_bridge	Reset Bridge Intel FPGA IP	
✓		dp_tx_reset_bridge	Reset Bridge Intel FPGA IP	
✓		clk_100	Clock Bridge Intel FPGA IP	
✓		dp_rx_clk_16	Clock Bridge Intel FPGA IP	
✓		dp_tx_clk_16	Clock Bridge Intel FPGA IP	
✓		cpu	Nios II Processor	
✓		onchip_mem	On-Chip Memory (RAM or ROM) In...	
✓		jtag_uart	JTAG UART Intel FPGA IP	
✓		sys_clock_timer	Interval Timer Intel FPGA IP	
✓		sysid	System ID Peripheral Intel FPGA IP	
✓		i2c_master	Avalon I2C (Master) Intel FPGA IP	
✓		dp_rx	dp_rx	
✓		dp_tx	dp_tx	

You can try to open either **dp_rx** or **dp_tx** to find that most of the high-speed interfaces (video pixel interfaces as well as connection with Native PHY Transceiver blocks) have been exported to allow the connection outside *Platform Designer*. We are doing those connections at top level entity as they are not compatible with the supported formats (Avalon-MM, Avalon-ST or AXI) and are considered as conduits. We use *Platform Designer* to build automatically all the Memory Mapped control architecture handled by the Nios II Processor.



System View | Address Map | Details

System: dp_core.dp_rx

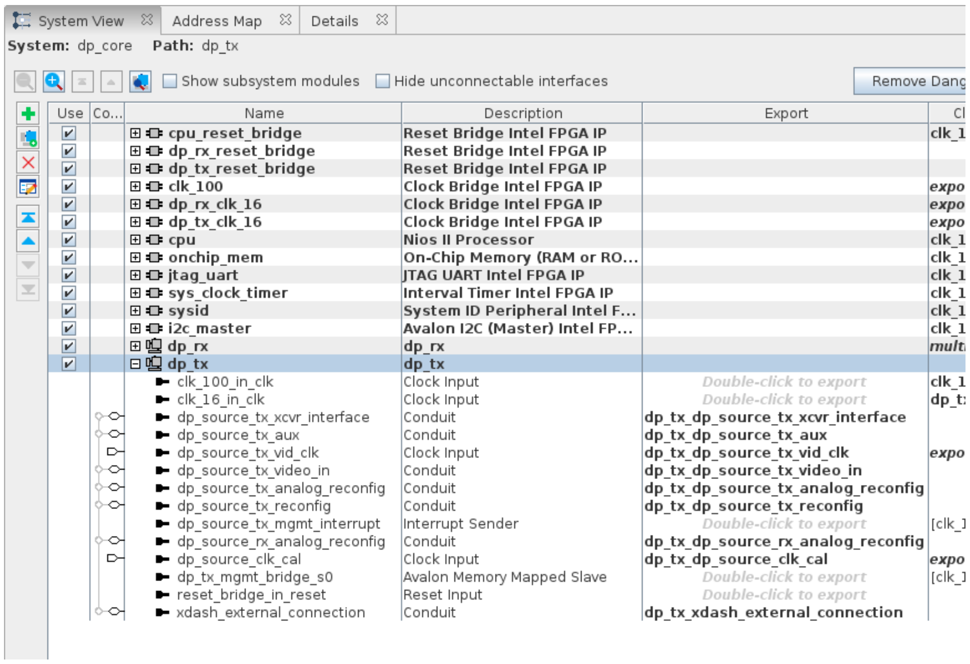
Show subsystem modules | Hide unconnectable interfaces | Remove Dangling Exports

Use	Name	Description	Export
✓	**reset_bridge**	**Reset Bridge Intel FPGA IP**	
	in_reset	Reset Input	**reset_bridge_in_reset**
	out_reset	Reset Output	*Double-click to export*
✓	**clk_100**	**Clock Bridge Intel FPGA IP**	
	in_clk	Clock Input	**clk_100_in_clk**
	out_clk	Clock Output	*Double-click to export*
✓	**clk_16**	**Clock Bridge Intel FPGA IP**	
	in_clk	Clock Input	**clk_16_in_clk**
	out_clk	Clock Output	*Double-click to export*
✓	**dp_sink**	**DisplayPort Intel FPGA IP**	
	rx_xcvr_interface	Conduit	**dp_sink_rx_xcvr_interface**
	rx_aux	Conduit	**dp_sink_rx_aux**
	rx_params	Conduit	**dp_sink_rx_params**
	rx_video_out	Conduit	**dp_sink_rx_video_out**
	rx_vid_clk	Clock Input	**dp_sink_rx_vid_clk**
	rx_stream	Conduit	**dp_sink_rx_stream**
	rx_msa_conduit	Conduit	**dp_sink_rx_msa_conduit**
	rx_reconfig	Conduit	**dp_sink_rx_reconfig**
	rx_analog_reconfig	Conduit	**dp_sink_rx_analog_reconfig**
	rx_mgmt	Avalon Memory Mapped Slave	*Double-click to export*
	rx_mgmt_interrupt	Interrupt Sender	**dp_sink_rx_mgmt_interrupt**
	clk	Clock Input	*Double-click to export*
	reset	Reset Input	*Double-click to export*
	aux_clk	Clock Input	*Double-click to export*
	aux_reset	Reset Input	*Double-click to export*
	xcvr_mgmt_clk	Clock Input	*Double-click to export*
	clk_cal	Clock Input	**dp_sink_clk_cal**
✓	**dp_rx_mgmt_bridge**	**Avalon Memory Mapped Pipeline...**	
	clk	Clock Input	*Double-click to export*
	reset	Reset Input	*Double-click to export*
	s0	Avalon Memory Mapped Slave	**dp_rx_mgmt_bridge_s0**
	m0	Avalon Memory Mapped Master	*Double-click to export*
✓	**pio_0**	**PIO (Parallel I/O) Intel FPGA IP**	
	clk	Clock Input	*Double-click to export*
	reset	Reset Input	*Double-click to export*
	s1	Avalon Memory Mapped Slave	*Double-click to export*
	external_connection	Conduit	**pio_0_external_connection**
✓	**pio_1**	**PIO (Parallel I/O) Intel FPGA IP**	
	clk	Clock Input	*Double-click to export*
	reset	Reset Input	*Double-click to export*
	s1	Avalon Memory Mapped Slave	*Double-click to export*
	external_connection	Conduit	**pio_1_external_connection**

2.3. Understanding the external connections

Let’s have a look at the different interfaces exported by the DisplayPort Source and Sink IPs to allow the connection of downstream modules.

- **DP_TX** -> Let’s expand the **dp_tx** instance in **dp_core.qsys**



System View | Address Map | Details

System: dp_core **Path:** dp_tx

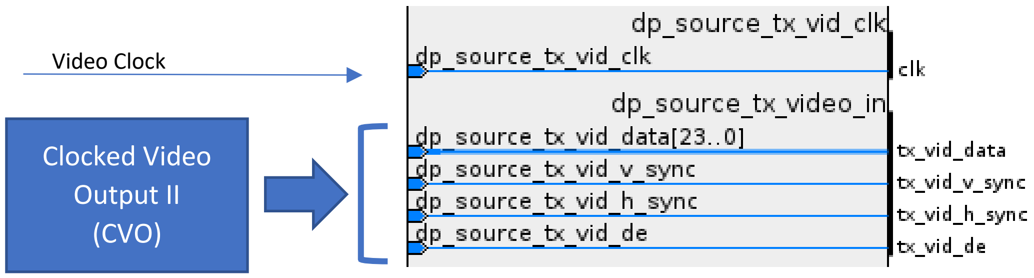
Show subsystem modules | Hide unconnectable interfaces | Remove Dang

Use	Name	Description	Export	Cl
✓	**cpu_reset_bridge**	**Reset Bridge Intel FPGA IP**		**clk_1**
✓	**dp_rx_reset_bridge**	**Reset Bridge Intel FPGA IP**		
✓	**dp_tx_reset_bridge**	**Reset Bridge Intel FPGA IP**		
✓	**clk_100**	**Clock Bridge Intel FPGA IP**		***expo***
✓	**dp_rx_clk_16**	**Clock Bridge Intel FPGA IP**		***expo***
✓	**dp_tx_clk_16**	**Clock Bridge Intel FPGA IP**		***expo***
✓	**cpu**	**Nios II Processor**		**clk_1**
✓	**onchip_mem**	**On-Chip Memory (RAM or RO...**		**clk_1**
✓	**jtag_uart**	**JTAG UART Intel FPGA IP**		**clk_1**
✓	**sys_clock_timer**	**Interval Timer Intel FPGA IP**		**clk_1**
✓	**sysid**	**System ID Peripheral Intel F...**		**clk_1**
✓	**i2c_master**	**Avalon I2C (Master) Intel FP...**		**clk_1**
✓	**dp_rx**	**dp_rx**		***multi***
✓	**dp_tx**	**dp_tx**		
	clk_100_in_clk	Clock Input	*Double-click to export*	**clk_1**
	clk_16_in_clk	Clock Input	*Double-click to export*	**dp_t:**
	dp_source_tx_xcvr_interface	Conduit	**dp_tx_dp_source_tx_xcvr_interface**	
	dp_source_tx_aux	Conduit	**dp_tx_dp_source_tx_aux**	
	dp_source_tx_vid_clk	Clock Input	**dp_tx_dp_source_tx_vid_clk**	***expo***
	dp_source_tx_video_in	Conduit	**dp_tx_dp_source_tx_video_in**	
	dp_source_tx_analog_reconfig	Conduit	**dp_tx_dp_source_tx_analog_reconfig**	
	dp_source_tx_reconfig	Conduit	**dp_tx_dp_source_tx_reconfig**	
	dp_source_tx_mgmt_interrupt	Interrupt Sender	*Double-click to export*	[clk_]
	dp_source_rx_analog_reconfig	Conduit	**dp_tx_dp_source_rx_analog_reconfig**	
	dp_source_clk_cal	Clock Input	**dp_tx_dp_source_clk_cal**	***expo***
	dp_tx_mgmt_bridge_s0	Avalon Memory Mapped Slave	*Double-click to export*	[clk_]
	reset_bridge_in_reset	Reset Input	*Double-click to export*	
	xdash_external_connection	Conduit	**dp_tx_xdash_external_connection**	

In the Export column, we can see which signals are exposed outside Platform Designed and must be connected externally. Please note that there are signals already connected internally, as part of the system level interconnect generation in Platform Designer.

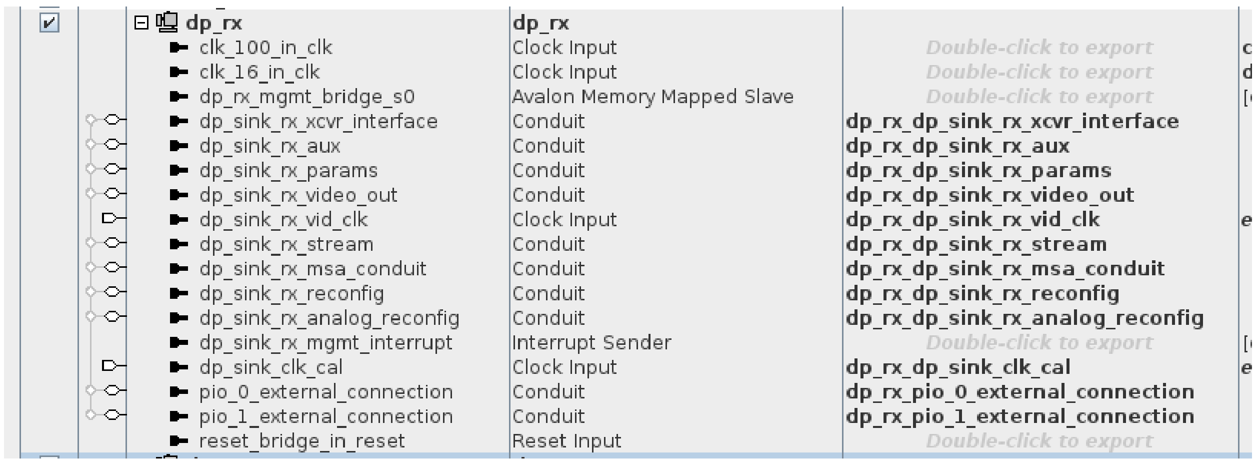
The exported signals are as follows:

- **tx_xcvr_interface** : to connect with external Native PHY instantiation. This is the video data already processed by the DisplayPort IP that will be serialized and sent to the DP output connector to drive our external display.
- **tx_aux** : this is the low speed (bidir) auxiliary channel available in the DP connector for EDID and link training management
- **tx_analog_reconfig, tx_reconfig** : used to manage the transceiver block reconfiguration
- **tx_vid_clk, tx_video_in** : this is where we will connect our VIP pipeline to drive video content through the DisplayPort IP. Let's explore the interface more detailed, having a look at the symbol generated in Platform Designer



The `dp_source_tx_video_in` interface allows glue-less connection with Clock Video Output component.

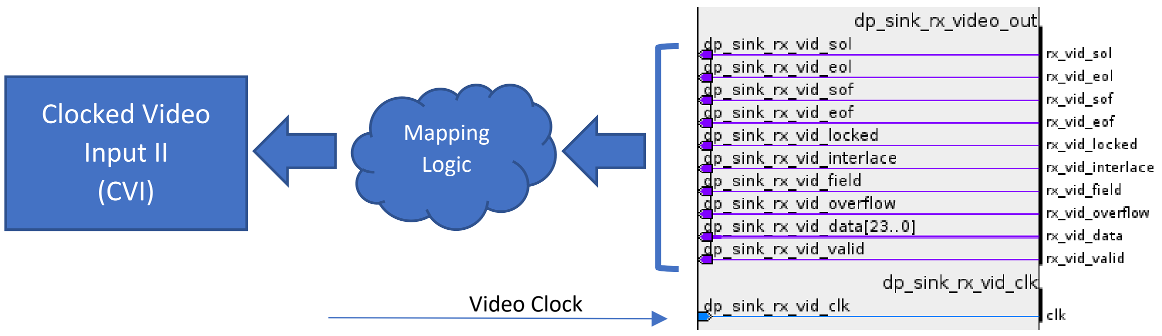
- **DP_RX**: For the Sink interface we can do a similar study and find interfaces to connect to the Native PHY RX transceiver block, as well as to manage reconfiguration and the auxiliary channel.



✓ dp_rx	dp_rx		
clk_100_in_clk	Clock Input	*Double-click to export*	c
clk_16_in_clk	Clock Input	*Double-click to export*	d
dp_rx_mgmt_bridge_s0	Avalon Memory Mapped Slave	*Double-click to export*	[
dp_sink_rx_xcvr_interface	Conduit	**dp_rx_dp_sink_rx_xcvr_interface**	
dp_sink_rx_aux	Conduit	**dp_rx_dp_sink_rx_aux**	
dp_sink_rx_params	Conduit	**dp_rx_dp_sink_rx_params**	
dp_sink_rx_video_out	Conduit	**dp_rx_dp_sink_rx_video_out**	
dp_sink_rx_vid_clk	Clock Input	**dp_rx_dp_sink_rx_vid_clk**	e
dp_sink_rx_stream	Conduit	**dp_rx_dp_sink_rx_stream**	
dp_sink_rx_msa_conduit	Conduit	**dp_rx_dp_sink_rx_msa_conduit**	
dp_sink_rx_reconfig	Conduit	**dp_rx_dp_sink_rx_reconfig**	
dp_sink_rx_analog_reconfig	Conduit	**dp_rx_dp_sink_rx_analog_reconfig**	
dp_sink_rx_mgmt_interrupt	Interrupt Sender	*Double-click to export*	[
dp_sink_clk_cal	Clock Input	**dp_rx_dp_sink_clk_cal**	e
pio_0_external_connection	Conduit	**dp_rx_pio_0_external_connection**	
pio_1_external_connection	Conduit	**dp_rx_pio_1_external_connection**	
reset_bridge_in_reset	Reset Input	*Double-click to export*	

We are interested in how to connect out Clocked Video Input (CVI) component to capture the video content received by the IP.

- **sink_rx_video_out & sink_rx_vid_clk** : are the conduits we will use for that purpose. Please note that, in this case, some simple adaption logic is required to map correctly the signals. We will go into more details in the next session where we are covering input video capturing

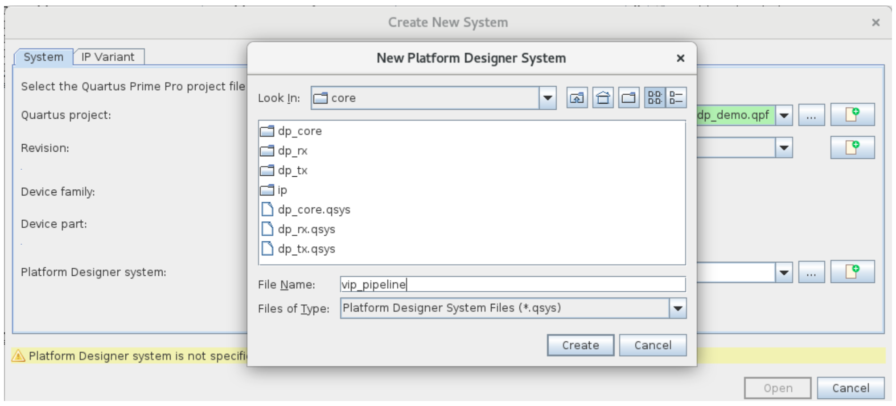


2.4. Build the video pipeline in Platform Designer

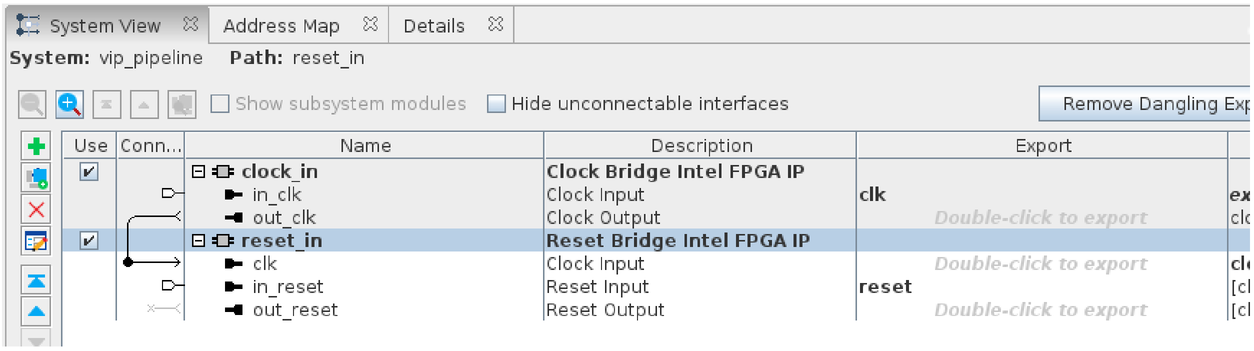
Now we know how to connect the CVI & CVO components together with the DisplayPort IP cores, let's build a simple design to test that we are generating correctly a video output signal to be displayed in an external monitor.

In this example we are using a simple pipeline composed by a **Test Pattern Generator** and a **Clock Video Output II** blocks.

1. Open Platform Designer
2. Go to **File->New System** in the main toolbar to open the dialog. Use **vip_pipeline.qsys** as File Name and save it in `<project_dir>/rtl/core`. Then click on create.


Create New System
System | IP Variant
Select the Quartus Prime Pro project file
Quartus project: dp_demo.qpf
Revision:
Device family:
Device part:
Platform Designer system:
Platform Designer system is not specifi
New Platform Designer System
Look In: core
dp_core
dp_rx
dp_tx
ip
dp_core.qsys
dp_rx.qsys
dp_tx.qsys
File Name: vip_pipeline
Files of Type: Platform Designer System Files (*.qsys)

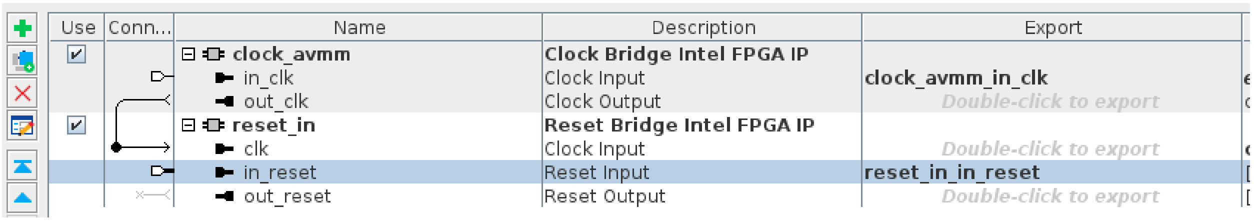
3. A new Platform Designer file will appear in the editor with 2 components already instantiated in the System View pane. A **Clock Bridge** and a **Reset Bridge**. Which are mandatory components in every system.



System: vip_pipeline **Path:** reset_in

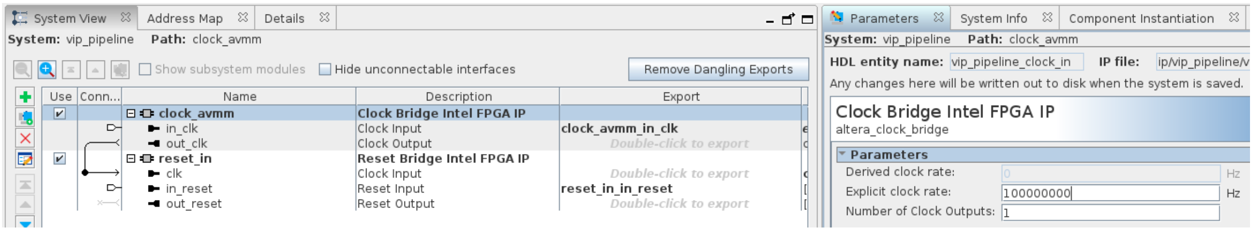
Use	Conn...	Name	Description	Export
✓		clock_in	Clock Bridge Intel FPGA IP	
		in_clk	Clock Input	clk
		out_clk	Clock Output	Double-click to export
✓		reset_in	Reset Bridge Intel FPGA IP	
		clk	Clock Input	Double-click to export
		in_reset	Reset Input	reset
		out_reset	Reset Output	Double-click to export

4. Let's rename the **clock_in** component to **clock_avmm** and delete the exported signals `clk` & `reset`. We will export them again after renaming, to account for the changes in naming on the exported signals.



Use	Conn...	Name	Description	Export
✓		clock_avmm	Clock Bridge Intel FPGA IP	
		in_clk	Clock Input	clock_avmm_in_clk
		out_clk	Clock Output	Double-click to export
✓		reset_in	Reset Bridge Intel FPGA IP	
		clk	Clock Input	Double-click to export
		in_reset	Reset Input	reset_in_in_reset
		out_reset	Reset Output	Double-click to export

5. We will use **clock_avmm** clock signal to connect our management clock also used to clock the Nios Processor. This clock has a freq=100MHz, so let's get this into the clock parameters in the `Explicit clock rate` field.



System: vip_pipeline **Path:** clock_avmm

Use	Conn...	Name	Description	Export
✓		clock_avmm	Clock Bridge Intel FPGA IP	
		in_clk	Clock Input	clock_avmm_in_clk
		out_clk	Clock Output	Double-click to export
✓		reset_in	Reset Bridge Intel FPGA IP	
		clk	Clock Input	Double-click to export
		in_reset	Reset Input	reset_in_in_reset
		out_reset	Reset Output	Double-click to export

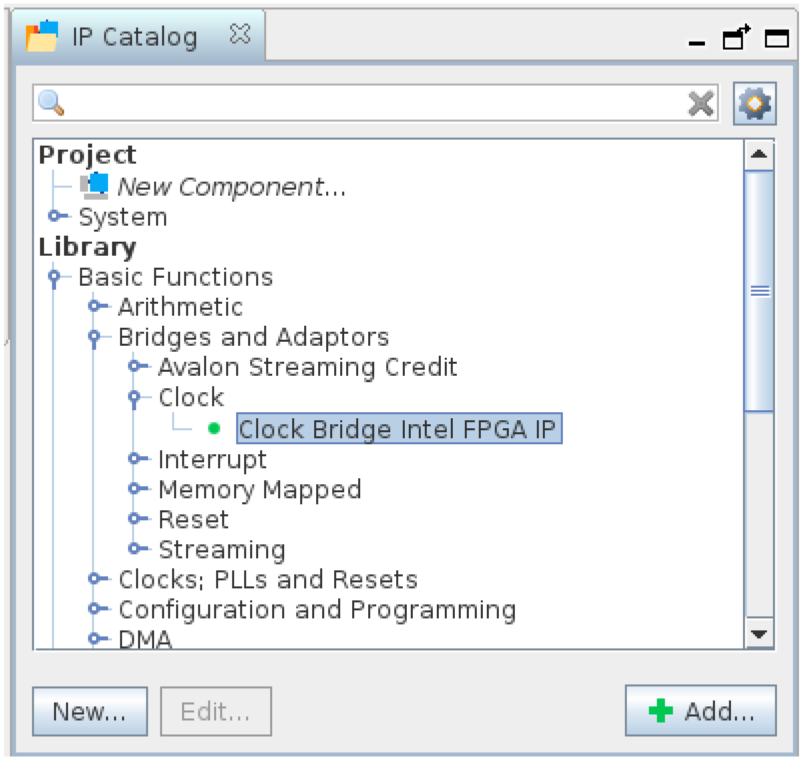
System: vip_pipeline **Path:** clock_avmm
HDL entity name: vip_pipeline_clock_in **IP file:** ip/vip_pipeline/
Any changes here will be written out to disk when the system is saved.

Clock Bridge Intel FPGA IP
altera_clock_bridge

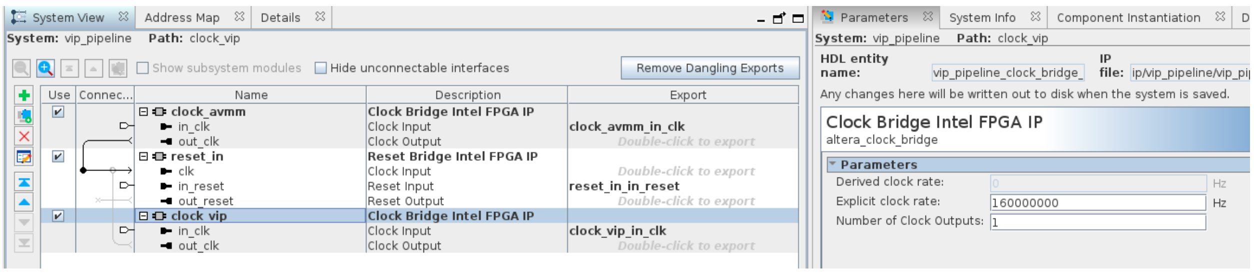
Parameters

Derived clock rate:	0	Hz
Explicit clock rate:	100000000	Hz
Number of Clock Outputs:	1	

6. Now, let's add the rest of the components to the system. We start for adding an additional clock we will use for driving our VIP pipeline. We are naming this as `clock_vip`, with a freq=160MHz. In the **IP Catalog->Library->Basic Functions->Bridges and Adaptors->Clock->Clock Bridge Intel FPGA IP** and click on **Add**



Rename it as `clock_vip` and in *`Explicit clock rate`* set 160000000. Make double click on the **in_clk** row in the **Export column** to get the input signal exposed to allow external clock connection into the system.



System: vip_pipeline **Path:** clock_vip

Use	Connec...	Name	Description	Export
✓		clock_avmm	Clock Bridge Intel FPGA IP	
		in_clk	Clock Input	clock_avmm_in_clk
		out_clk	Clock Output	Double-click to export
✓		reset_in	Reset Bridge Intel FPGA IP	
		clk	Clock Input	Double-click to export
		in_reset	Reset Input	reset_in_in_reset
		out_reset	Reset Output	Double-click to export
✓		clock vip	Clock Bridge Intel FPGA IP	
		in_clk	Clock Input	clock_vip_in_clk
		out_clk	Clock Output	Double-click to export

System: vip_pipeline **Path:** clock_vip

HDL entity name: vip_pipeline_clock_bridge_ **IP file:** ip/vip_pipeline/vip_pi

Any changes here will be written out to disk when the system is saved.

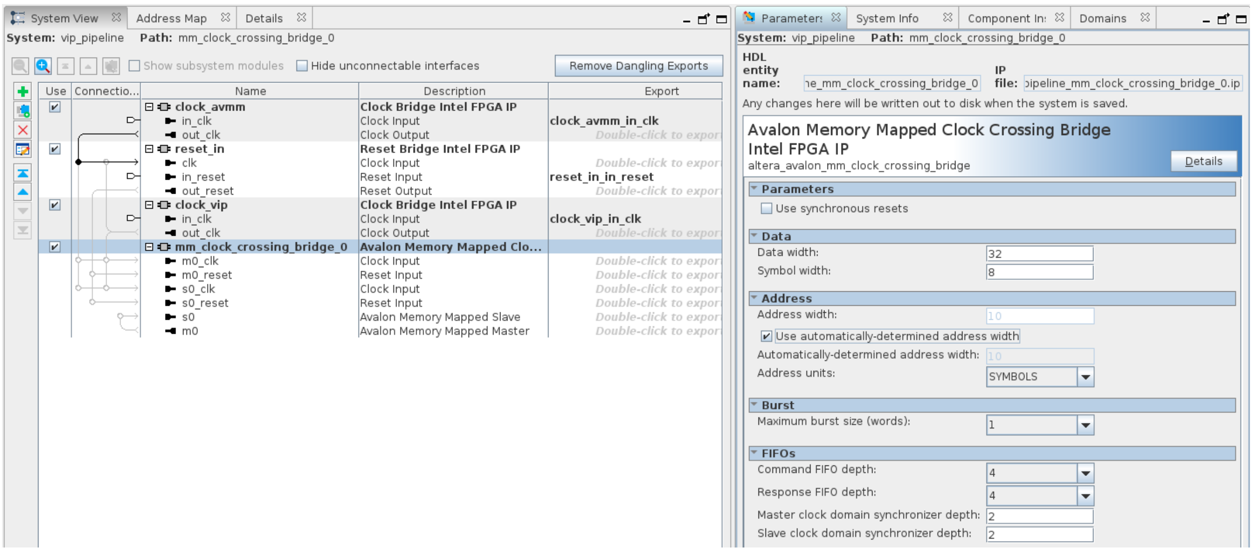
Clock Bridge Intel FPGA IP
altera_clock_bridge

Parameters

Derived clock rate:	0	Hz
Explicit clock rate:	160000000	Hz
Number of Clock Outputs:	1	

NOTE: In our system, we will work with 1080p60 resolution, which equals a clock rate of 148.5MHz. In order to make the VIP blocks function properly we need a signal equal or higher frequency of the pixel clock used. As we already have in our system a 160MHz, used for `dp_vid_rx`, let's use this one to drive our VIP pipeline.

7. Go to **IP Catalog->Library->Basic Functions->Bridges and Adaptors->Memory Mapped-> Avalon Memory Mapped Clock Crossing Bridge** and Add one to the system.
   In the *Parameters* tab tick on **Address->Use automatically-determined address width**, leaving all the rest at default


System View | Address Map | Details

System: vip_pipeline **Path:** mm_clock_crossing_bridge_0

☐ Show subsystem modules ☐ Hide unconnectable interfaces [Remove Dangling Exports]

Use	Name	Description	Export
✓	**clock_avmm**	**Clock Bridge Intel FPGA IP**	
	in_clk	Clock Input	**clock_avmm_in_clk**
	out_clk	Clock Output	*Double-click to export*
✓	**reset_in**	**Reset Bridge Intel FPGA IP**	
	clk	Clock Input	*Double-click to export*
	in_reset	Reset Input	**reset_in_in_reset**
	out_reset	Reset Output	*Double-click to export*
✓	**clock_vip**	**Clock Bridge Intel FPGA IP**	
	in_clk	Clock Input	**clock_vip_in_clk**
	out_clk	Clock Output	*Double-click to export*
✓	**mm_clock_crossing_bridge_0**	**Avalon Memory Mapped Clo...**	
	m0_clk	Clock Input	*Double-click to export*
	m0_reset	Reset Input	*Double-click to export*
	s0_clk	Clock Input	*Double-click to export*
	s0_reset	Reset Input	*Double-click to export*
	s0	Avalon Memory Mapped Slave	*Double-click to export*
	m0	Avalon Memory Mapped Master	*Double-click to export*

Parameters | System Info | Component In: | Domains

System: vip_pipeline **Path:** mm_clock_crossing_bridge_0

HDL entity name: ne_mm_clock_crossing_bridge_0 **IP file:** pipeline_mm_clock_crossing_bridge_0.ip

Any changes here will be written out to disk when the system is saved.

Avalon Memory Mapped Clock Crossing Bridge Intel FPGA IP
altera_avalon_mm_clock_crossing_bridge

Parameters
☐ Use synchronous resets

Data
Data width: 32
Symbol width: 8

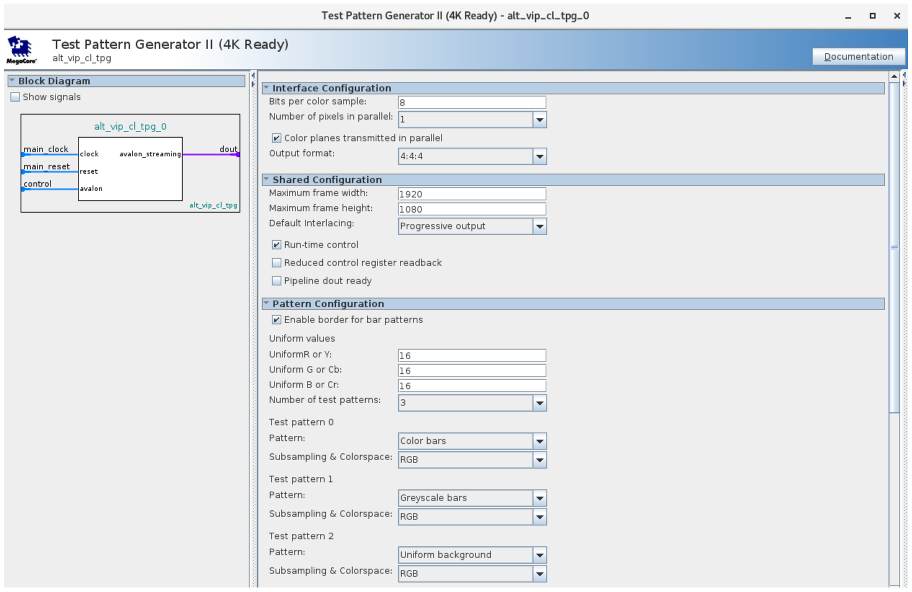
Address
Address width: 10
☑ Use automatically-determined address width
Automatically-determined address width: 10
Address units: SYMBOLS

Burst
Maximum burst size (words): 1

FIFOs
Command FIFO depth: 4
Response FIFO depth: 4
Master clock domain synchronizer depth: 2
Slave clock domain synchronizer depth: 2

NOTE: It's a good policy to isolate different branches of the complete Avalon-MM implementation with Pipeline Bridges to ease timing closure in congested systems. In this case, it's even more necessary as the VIP blocks run at different clock frequencies than the NiosII Processor. Having a Clock Crossing Bridge handles these crossing boundaries efficiently by adding synchronizers.

8. Now, let's add our VIP blocks. Go to **IP Catalog->Library->DSP->Video and Image Processing->Test Pattern Generator II**, select it and click **Add**. The parameterization GUI appears, select:
    - Enable Run-time Control
    - Number of test patterns: 3
        - Pattern 0: Color bars
        - Pattern 1: Grayscale bars
        - Pattern 2: Uniform Background



Test Pattern Generator II (4K Ready) - alt_vip_cl_tpg_0

Test Pattern Generator II (4K Ready)
alt_vip_cl_tpg

Block Diagram
☐ Show signals

Interface Configuration
Bits per color sample: 8
Number of pixels in parallel: 1
☑ Color planes transmitted in parallel
Output format: 4:4:4

Shared Configuration
Maximum frame width: 1920
Maximum frame height: 1080
Default Interlacing: Progressive output
☑ Run-time control
☐ Reduced control register readback
☐ Pipeline dout ready

Pattern Configuration
☑ Enable border for bar patterns
Uniform values
UniformR or Y: 16
Uniform G or Cb: 16
Uniform B or Cr: 16
Number of test patterns: 3

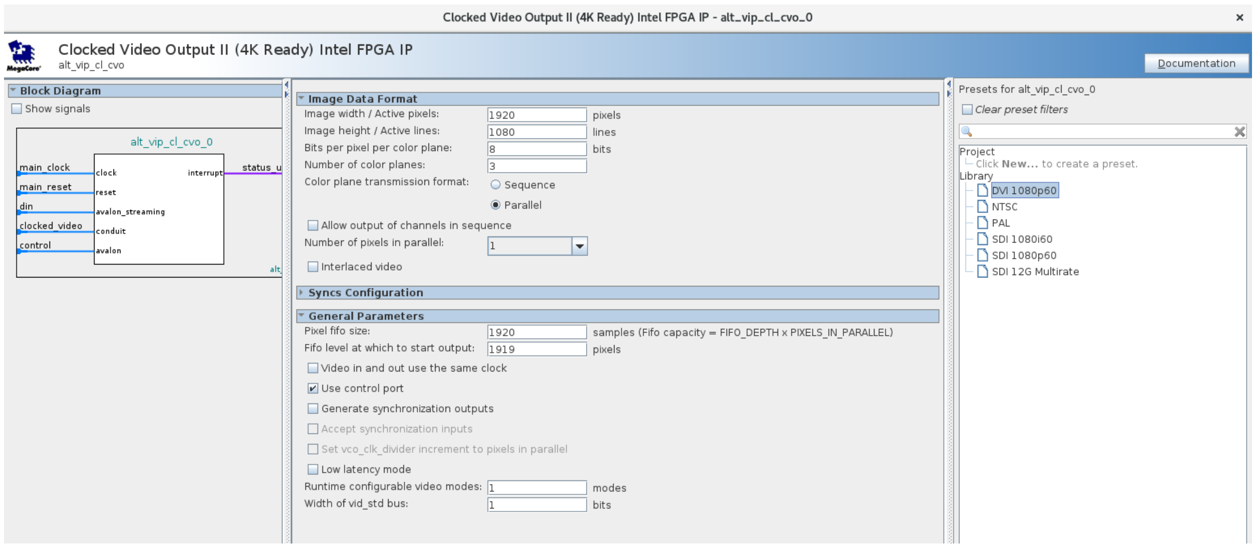
Test pattern 0
Pattern: Color bars
Subsampling & Colorspace: RGB

Test pattern 1
Pattern: Greyscale bars
Subsampling & Colorspace: RGB

Test pattern 2
Pattern: Uniform background
Subsampling & Colorspace: RGB

9. Rename it to **tpg_0**

10. Go to **IP Catalog->Library->DSP->Video and Image Processing->Clocked Video Output II**, select it and click **Add**. The parameterization GUI appears, select:
    - o Under Presets (on the rightmost side), select **DVI 1080p60** and click **Apply**
    - o Then enable General Parameters->Use control port
    - o Leave all the other settings as default



Clocked Video Output II (4K Ready) Intel FPGA IP - alt_vip_cl_cvo_0

Clocked Video Output II (4K Ready) Intel FPGA IP
alt_vip_cl_cvo

Documentation

Block Diagram
Show signals

alt_vip_cl_cvo_0: main_clock → clock; main_reset → reset; din → avalon_streaming; clocked_video → conduit; control → avalon; interrupt → status_u

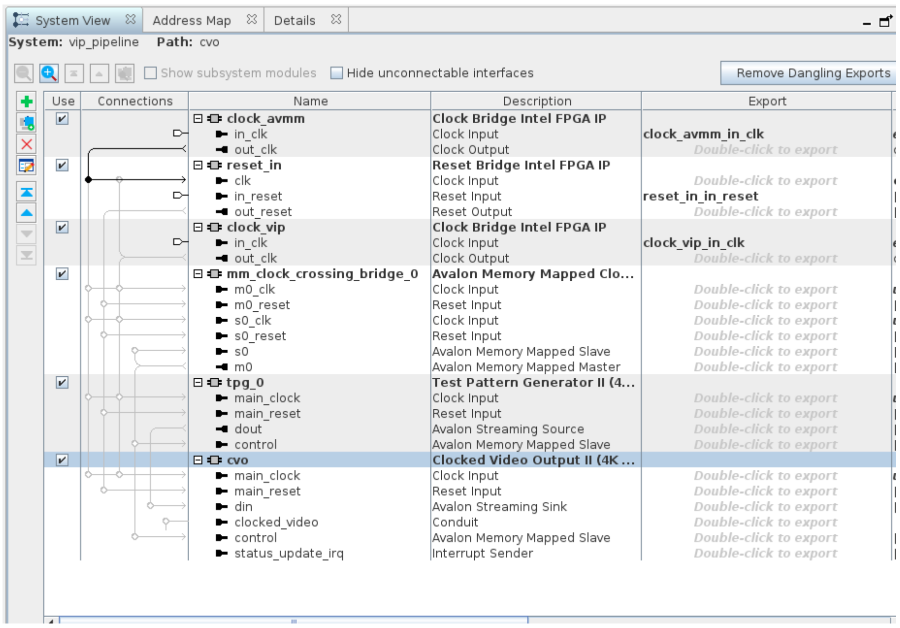
Image Data Format
Image width / Active pixels: 1920 pixels
Image height / Active lines: 1080 lines
Bits per pixel per color plane: 8 bits
Number of color planes: 3
Color plane transmission format: ( ) Sequence [✓ Parallel]
[ ] Allow output of channels in sequence
Number of pixels in parallel: 1
[ ] Interlaced video

Syncs Configuration

General Parameters
Pixel fifo size: 1920 samples (Fifo capacity = FIFO_DEPTH x PIXELS_IN_PARALLEL)
Fifo level at which to start output: 1919 pixels
[ ] Video in and out use the same clock
[✓] Use control port
[ ] Generate synchronization outputs
[ ] Accept synchronization inputs
[ ] Set vco_clk_divider increment to pixels in parallel
[ ] Low latency mode
Runtime configurable video modes: 1 modes
Width of vid_std bus: 1 bits

Presets for alt_vip_cl_cvo_0
[ ] Clear preset filters
Project
Click New... to create a preset.
Library
DVI 1080p60
NTSC
PAL
SDI 1080i60
SDI 1080p60
SDI 12G Multirate

11. Rename it to **cvo**. Let's have a look at the components added. We need now to start making the connections and exporting the relevant interfaces.



System View | Address Map | Details

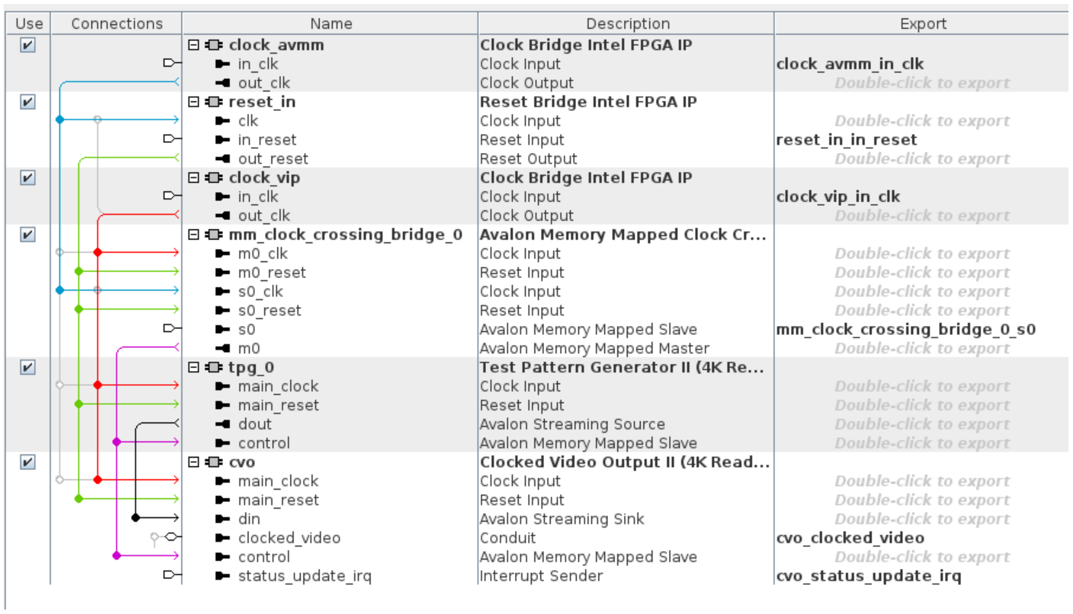
System: vip_pipeline **Path:** cvo

[ ] Show subsystem modules [ ] Hide unconnectable interfaces Remove Dangling Exports

Use	Connections	Name	Description	Export
✓		**clock_avmm**	**Clock Bridge Intel FPGA IP**	
		in_clk	Clock Input	**clock_avmm_in_clk**
		out_clk	Clock Output	*Double-click to export*
✓		**reset_in**	**Reset Bridge Intel FPGA IP**	
		clk	Clock Input	*Double-click to export*
		in_reset	Reset Input	**reset_in_in_reset**
		out_reset	Reset Output	*Double-click to export*
✓		**clock_vip**	**Clock Bridge Intel FPGA IP**	
		in_clk	Clock Input	**clock_vip_in_clk**
		out_clk	Clock Output	*Double-click to export*
✓		**mm_clock_crossing_bridge_0**	**Avalon Memory Mapped Clo...**	
		m0_clk	Clock Input	*Double-click to export*
		m0_reset	Reset Input	*Double-click to export*
		s0_clk	Clock Input	*Double-click to export*
		s0_reset	Reset Input	*Double-click to export*
		s0	Avalon Memory Mapped Slave	*Double-click to export*
		m0	Avalon Memory Mapped Master	*Double-click to export*
✓		**tpg_0**	**Test Pattern Generator II (4...**	
		main_clock	Clock Input	*Double-click to export*
		main_reset	Reset Input	*Double-click to export*
		dout	Avalon Streaming Source	*Double-click to export*
		control	Avalon Memory Mapped Slave	*Double-click to export*
✓		**cvo**	**Clocked Video Output II (4K ...**	
		main_clock	Clock Input	*Double-click to export*
		main_reset	Reset Input	*Double-click to export*
		din	Avalon Streaming Sink	*Double-click to export*
		clocked_video	Conduit	*Double-click to export*
		control	Avalon Memory Mapped Slave	*Double-click to export*
		status_update_irq	Interrupt Sender	*Double-click to export*

12. Exporting interfaces
    Let's export the following conduits and signals
    - o mm_clock_crossing_bridge_0: s0
    - o cvo: clocked_video
    - o cvo: status_update_irq

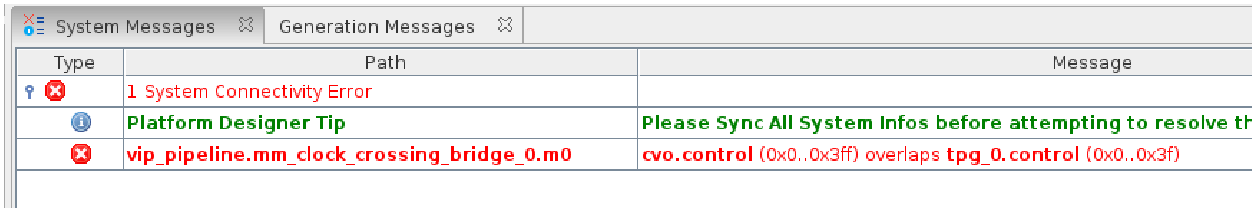
13. Making the connections. Follow the figure below to make the right connections. You can apply different colors per signal for easier identification



Use	Connections	Name	Description	Export
✓		clock_avmm	Clock Bridge Intel FPGA IP	
		in_clk	Clock Input	clock_avmm_in_clk
		out_clk	Clock Output	Double-click to export
✓		reset_in	Reset Bridge Intel FPGA IP	
		clk	Clock Input	Double-click to export
		in_reset	Reset Input	reset_in_in_reset
		out_reset	Reset Output	Double-click to export
✓		clock_vip	Clock Bridge Intel FPGA IP	
		in_clk	Clock Input	clock_vip_in_clk
		out_clk	Clock Output	Double-click to export
✓		mm_clock_crossing_bridge_0	Avalon Memory Mapped Clock Cr...	
		m0_clk	Clock Input	Double-click to export
		m0_reset	Reset Input	Double-click to export
		s0_clk	Clock Input	Double-click to export
		s0_reset	Reset Input	Double-click to export
		s0	Avalon Memory Mapped Slave	mm_clock_crossing_bridge_0_s0
		m0	Avalon Memory Mapped Master	Double-click to export
✓		tpg_0	Test Pattern Generator II (4K Re...	
		main_clock	Clock Input	Double-click to export
		main_reset	Reset Input	Double-click to export
		dout	Avalon Streaming Source	Double-click to export
		control	Avalon Memory Mapped Slave	Double-click to export
✓		cvo	Clocked Video Output II (4K Read...	
		main_clock	Clock Input	Double-click to export
		main_reset	Reset Input	Double-click to export
		din	Avalon Streaming Sink	Double-click to export
		clocked_video	Conduit	cvo_clocked_video
		control	Avalon Memory Mapped Slave	Double-click to export
		status_update_irq	Interrupt Sender	cvo_status_update_irq

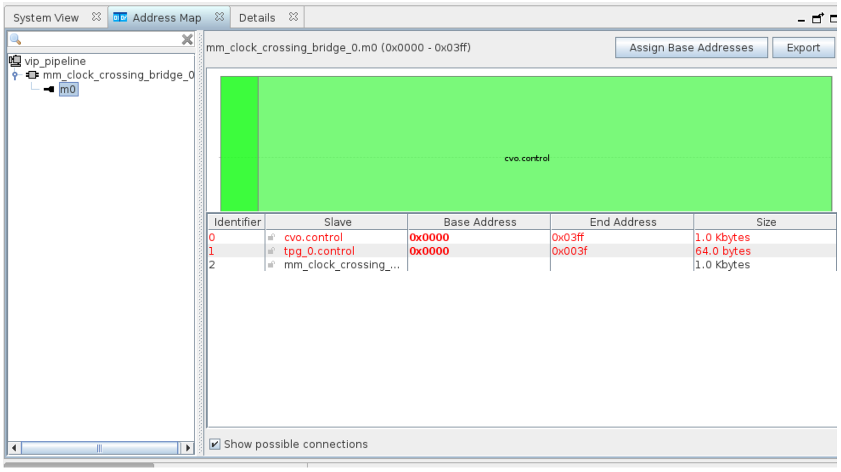
14. Solving Memory Mapped address conflicts

You might have noticed an error message in the System Messages tab highlighting a Connectivity Error.



Type	Path	Message
	1 System Connectivity Error	
	Platform Designer Tip	Please Sync All System Infos before attempting to resolve th
	vip_pipeline.mm_clock_crossing_bridge_0.m0	cvo.control (0x0..0x3ff) overlaps tpg_0.control (0x0..0x3f)

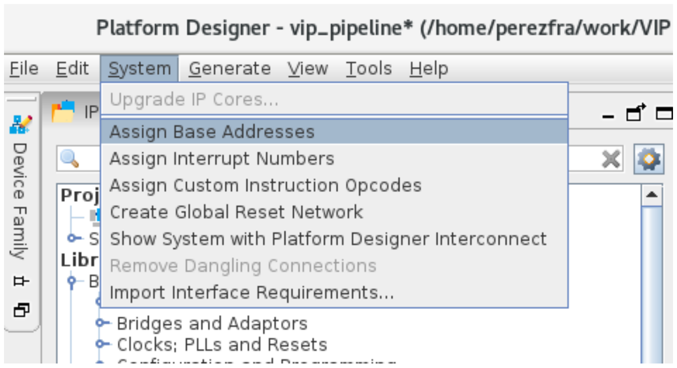
We have connected 2 slaves (CVO, TPG) to the same master `mm_clock_crossing_bridge:m0`, and they both have the same Base address. We can identify this by selecting m0 master in the Address Map tab.



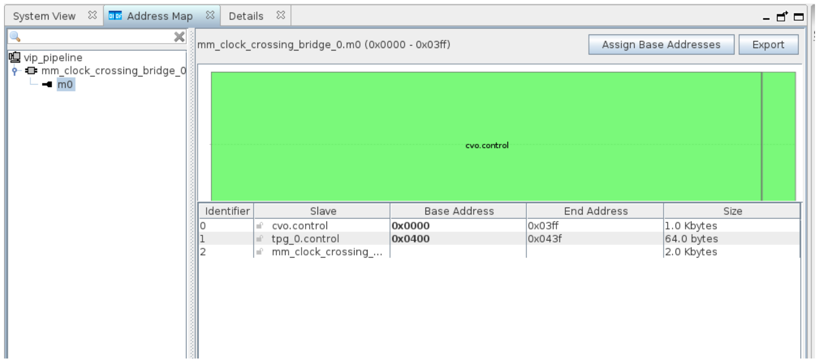
mm_clock_crossing_bridge_0.m0 (0x0000 - 0x03ff)

Identifier	Slave	Base Address	End Address	Size
0	cvo.control	0x0000	0x03ff	1.0 Kbytes
1	tpg_0.control	0x0000	0x003f	64.0 bytes
2	mm_clock_crossing_...			1.0 Kbytes

We can solve this issue by selecting System->Assign Base Addresses in the main toolbar


Platform Designer - vip_pipeline* (/home/perezfra/work/VIP
File Edit System Generate View Tools Help
Upgrade IP Cores...
Assign Base Addresses
Assign Interrupt Numbers
Assign Custom Instruction Opcodes
Create Global Reset Network
Show System with Platform Designer Interconnect
Remove Dangling Connections
Import Interface Requirements...
Device Family
Bridges and Adaptors
Clocks; PLLs and Resets

After execution, each slave will have its exclusive address space



mm_clock_crossing_bridge_0.m0 (0x0000 - 0x03ff)

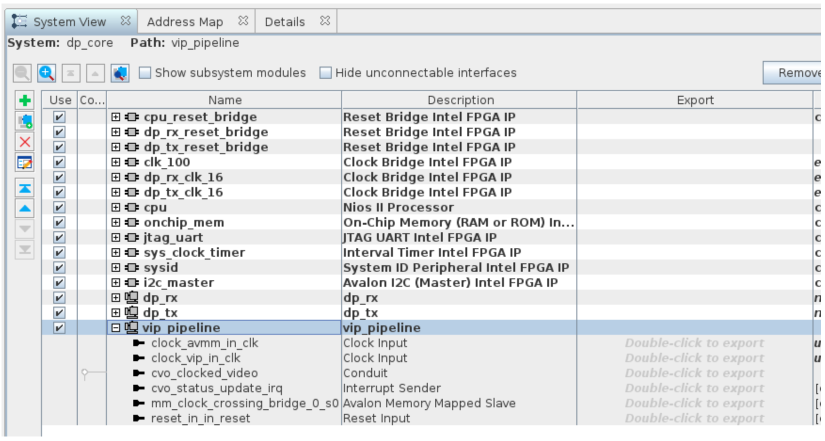
Identifier	Slave	Base Address	End Address	Size
0	cvo.control	**0x0000**	0x03ff	1.0 Kbytes
1	tpg_0.control	**0x0400**	0x043f	64.0 bytes
2	mm_clock_crossing_...			2.0 Kbytes

15. With this, we have finished the elaboration of our VIP pipeline. Let's now integrate it with the rest of the system

2.5. Integrating the VIP pipeline

1. In Platform Designer, open the *<project_dir>/rtl/core/dp_core.qsys*. This will be our higher hierarchical level where we will instantiate and connect our newly generated `vip_pipeline.qsys`

2. Under **IP Catalog->Project**, expand **System** group and you will find the different subsystems that are in the search path and available to use (some of them already used indeed). Double-click on *vip_pipeline* to add it. Rename to **vip_pipeline**



System View | Address Map | Details

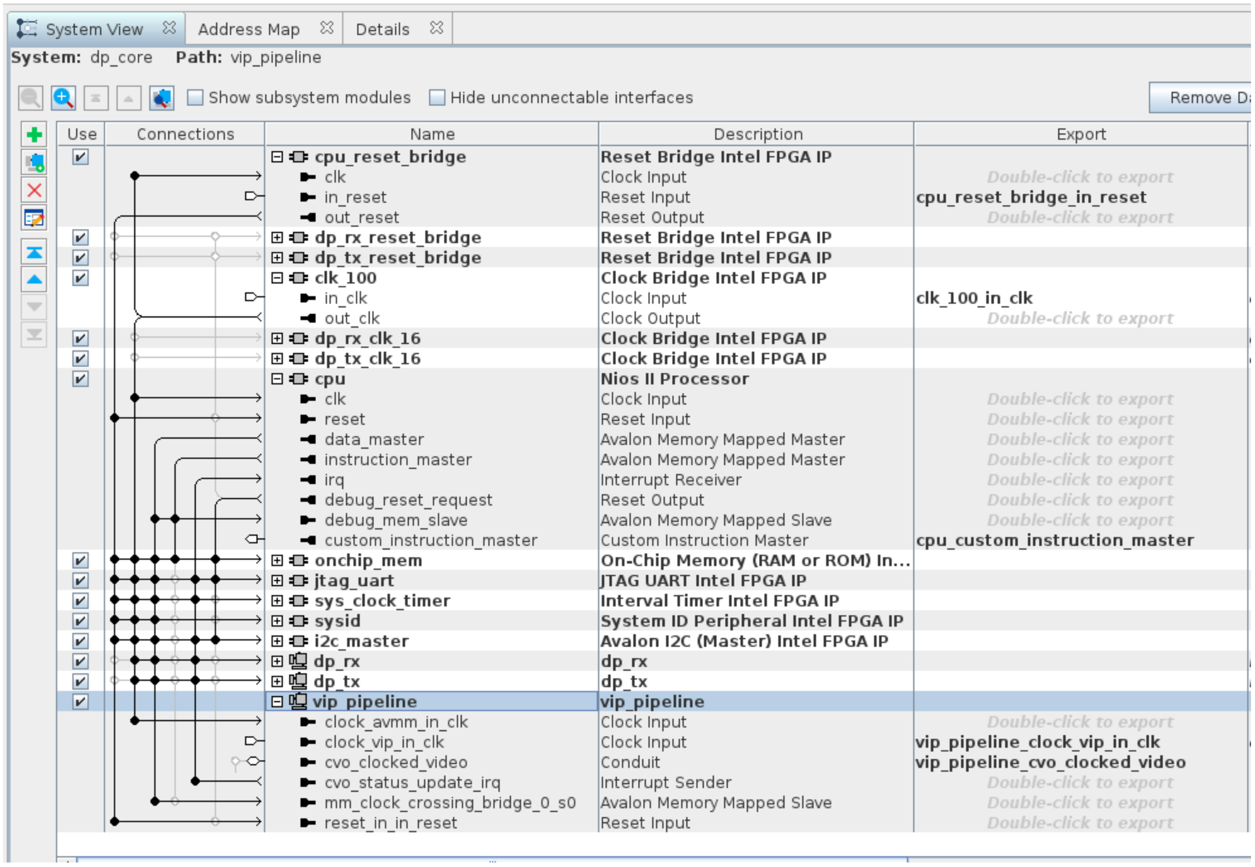
System: dp_core **Path:** vip_pipeline

Show subsystem modules | Hide unconnectable interfaces | Remove

Use	Co...	Name	Description	Export
✓		⊞ cpu_reset_bridge	Reset Bridge Intel FPGA IP	
✓		⊞ dp_rx_reset_bridge	Reset Bridge Intel FPGA IP	
✓		⊞ dp_tx_reset_bridge	Reset Bridge Intel FPGA IP	
✓		⊞ clk_100	Clock Bridge Intel FPGA IP	
✓		⊞ dp_rx_clk_16	Clock Bridge Intel FPGA IP	
✓		⊞ dp_tx_clk_16	Clock Bridge Intel FPGA IP	
✓		⊞ cpu	Nios II Processor	
✓		⊞ onchip_mem	On-Chip Memory (RAM or ROM) In...	
✓		⊞ jtag_uart	JTAG UART Intel FPGA IP	
✓		⊞ sys_clock_timer	Interval Timer Intel FPGA IP	
✓		⊞ sysid	System ID Peripheral Intel FPGA IP	
✓		⊞ i2c_master	Avalon I2C (Master) Intel FPGA IP	
✓		⊞ dp_rx	dp_rx	
✓		⊞ dp_tx	dp_tx	
✓		⊟ vip_pipeline	vip_pipeline	
		clock_avmm_in_clk	Clock Input	Double-click to export
		clock_vip_in_clk	Clock Input	Double-click to export
		cvo_clocked_video	Conduit	Double-click to export
		cvo_status_update_irq	Interrupt Sender	Double-click to export
		mm_clock_crossing_bridge_0_s0	Avalon Memory Mapped Slave	Double-click to export
		reset_in_in_reset	Reset Input	Double-click to export

3. In order to connect our CVO with the DisplayPort TX interface, we would need to export the video output conduit and the corresponding clock. Then we do the remaining connection with the rest of components.
    - vip_pipeline: clock_avmm_in_clk **->** clk_100: out_clk
    - vip_pipeline: cvo_status_update_irq**->** cpu: irq
    - vip_pipeline: mm_clock_crossing_bridge_0_s0 **->** cpu: data_master
    - vip_pipeline: reset_in_in_reset **->** cpu_reset_bridge: out_reset

We should get connections like in the picture below



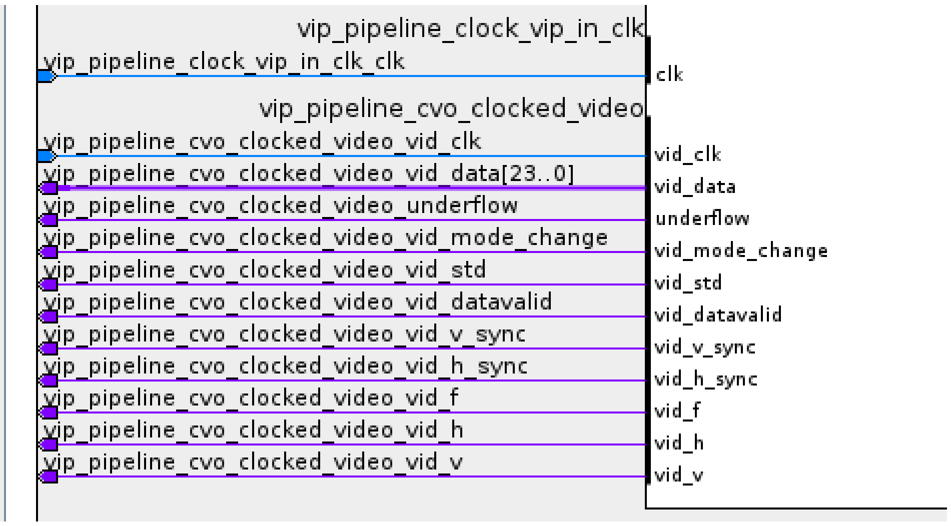
System View | Address Map | Details

System: dp_core **Path:** vip_pipeline

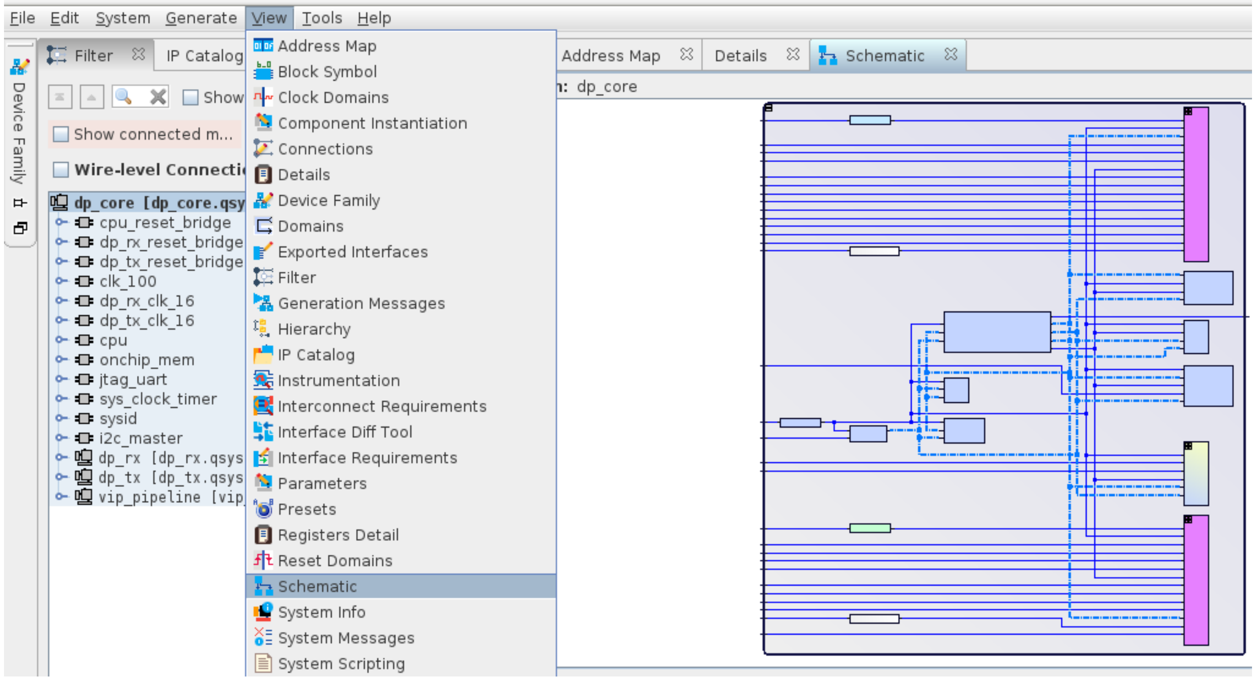
Show subsystem modules | Hide unconnectable interfaces | Remove D

Use	Connections	Name	Description	Export
✓		⊟ cpu_reset_bridge	Reset Bridge Intel FPGA IP	
		clk	Clock Input	Double-click to export
		in_reset	Reset Input	cpu_reset_bridge_in_reset
		out_reset	Reset Output	Double-click to export
✓		⊞ dp_rx_reset_bridge	Reset Bridge Intel FPGA IP	
✓		⊞ dp_tx_reset_bridge	Reset Bridge Intel FPGA IP	
✓		⊟ clk_100	Clock Bridge Intel FPGA IP	
		in_clk	Clock Input	clk_100_in_clk
		out_clk	Clock Output	Double-click to export
✓		⊞ dp_rx_clk_16	Clock Bridge Intel FPGA IP	
✓		⊞ dp_tx_clk_16	Clock Bridge Intel FPGA IP	
✓		⊟ cpu	Nios II Processor	
		clk	Clock Input	Double-click to export
		reset	Reset Input	Double-click to export
		data_master	Avalon Memory Mapped Master	Double-click to export
		instruction_master	Avalon Memory Mapped Master	Double-click to export
		irq	Interrupt Receiver	Double-click to export
		debug_reset_request	Reset Output	Double-click to export
		debug_mem_slave	Avalon Memory Mapped Slave	Double-click to export
		custom_instruction_master	Custom Instruction Master	cpu_custom_instruction_master
✓		⊞ onchip_mem	On-Chip Memory (RAM or ROM) In...	
✓		⊞ jtag_uart	JTAG UART Intel FPGA IP	
✓		⊞ sys_clock_timer	Interval Timer Intel FPGA IP	
✓		⊞ sysid	System ID Peripheral Intel FPGA IP	
✓		⊞ i2c_master	Avalon I2C (Master) Intel FPGA IP	
✓		⊞ dp_rx	dp_rx	
✓		⊞ dp_tx	dp_tx	
✓		⊟ vip_pipeline	vip_pipeline	
		clock_avmm_in_clk	Clock Input	Double-click to export
		clock_vip_in_clk	Clock Input	vip_pipeline_clock_vip_in_clk
		cvo_clocked_video	Conduit	vip_pipeline_cvo_clocked_video
		cvo_status_update_irq	Interrupt Sender	Double-click to export
		mm_clock_crossing_bridge_0_s0	Avalon Memory Mapped Slave	Double-click to export
		reset_in_in_reset	Reset Input	Double-click to export

4. Then, save the system. We can go to **View->Block Symbol** in the main tool bar to enable the *Symbol View* tab. By selecting the top level *dp_core* module, we can analyze the exported ports to connect externally. We will find how the *clocked_video_output* interface and related clock have been exported.


vip_pipeline_clock_vip_in_clk
vip_pipeline_clock_vip_in_clk_clk
clk
vip_pipeline_cvo_clocked_video
vip_pipeline_cvo_clocked_video_vid_clk
vip_pipeline_cvo_clocked_video_vid_data[23..0]
vip_pipeline_cvo_clocked_video_underflow
vip_pipeline_cvo_clocked_video_vid_mode_change
vip_pipeline_cvo_clocked_video_vid_std
vip_pipeline_cvo_clocked_video_vid_datavalid
vip_pipeline_cvo_clocked_video_vid_v_sync
vip_pipeline_cvo_clocked_video_vid_h_sync
vip_pipeline_cvo_clocked_video_vid_f
vip_pipeline_cvo_clocked_video_vid_h
vip_pipeline_cvo_clocked_video_vid_v
vid_clk
vid_data
underflow
vid_mode_change
vid_std
vid_datavalid
vid_v_sync
vid_h_sync
vid_f
vid_h
vid_v

You can also enable the **View->Schematic** to see how all the modules have been connected and the interfaces exposed. You can "pan & zoom" and go into/out the modules to navigate across all the hierarchy.

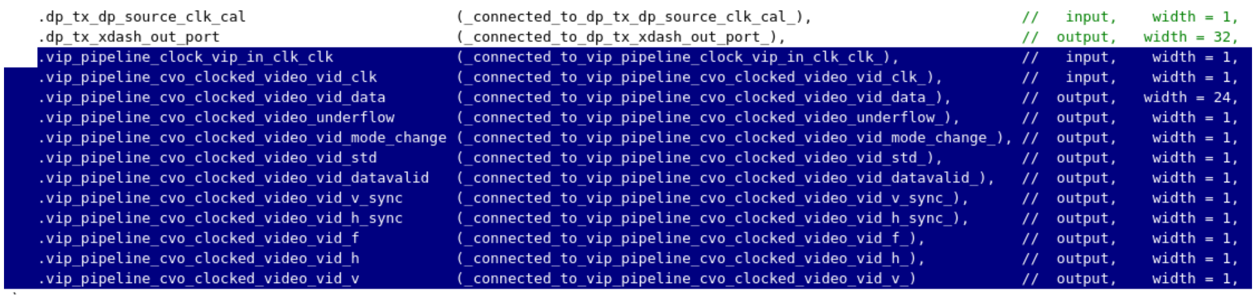


5. Generate the system by selecting **Generate->Generate HDL**

2.6.Integrate the complete module at top level

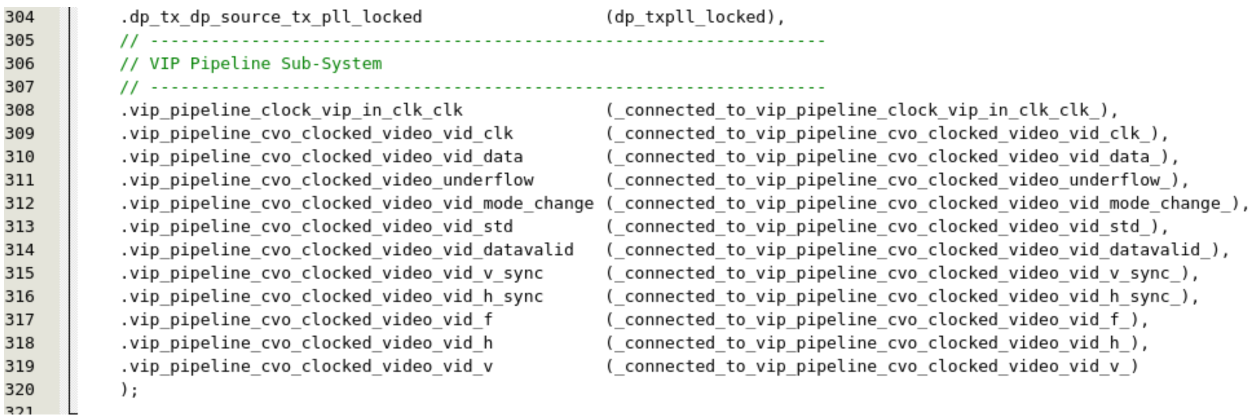
After generation, let's modify the top level entity **c10_dp_demo.v** in Quartus to incorporate the changes.

1. Open the newly generated platform designed system instantiation file: *<project_dir>/rtl/core/dp_core/dp_core_inst.v*, scroll till the end of the file until find the new ports added for **cvo**. Copy all the complete lines:



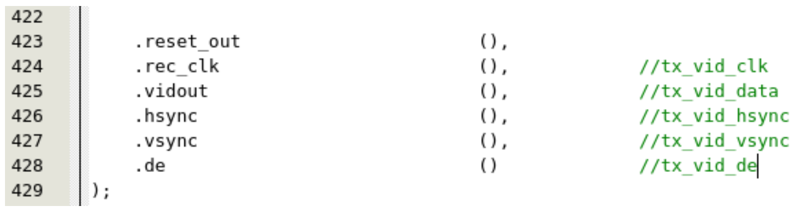
```
.dp_tx_dp_source_clk_cal                          (_connected_to_dp_tx_dp_source_clk_cal_),                         //   input,    width = 1,
.dp_tx_xdash_out_port                             (_connected_to_dp_tx_xdash_out_port_),                            //  output,   width = 32,
.vip_pipeline_clock_vip_in_clk_clk                (_connected_to_vip_pipeline_clock_vip_in_clk_clk_),               //   input,    width = 1,
.vip_pipeline_cvo_clocked_video_vid_clk           (_connected_to_vip_pipeline_cvo_clocked_video_vid_clk_),          //   input,    width = 1,
.vip_pipeline_cvo_clocked_video_vid_data          (_connected_to_vip_pipeline_cvo_clocked_video_vid_data_),         //  output,   width = 24,
.vip_pipeline_cvo_clocked_video_underflow         (_connected_to_vip_pipeline_cvo_clocked_video_underflow_),        //  output,    width = 1,
.vip_pipeline_cvo_clocked_video_vid_mode_change (_connected_to_vip_pipeline_cvo_clocked_video_vid_mode_change_), //  output,    width = 1,
.vip_pipeline_cvo_clocked_video_vid_std           (_connected_to_vip_pipeline_cvo_clocked_video_vid_std_),          //  output,    width = 1,
.vip_pipeline_cvo_clocked_video_vid_datavalid     (_connected_to_vip_pipeline_cvo_clocked_video_vid_datavalid_),    //  output,    width = 1,
.vip_pipeline_cvo_clocked_video_vid_v_sync        (_connected_to_vip_pipeline_cvo_clocked_video_vid_v_sync_),       //  output,    width = 1,
.vip_pipeline_cvo_clocked_video_vid_h_sync        (_connected_to_vip_pipeline_cvo_clocked_video_vid_h_sync_),       //  output,    width = 1,
.vip_pipeline_cvo_clocked_video_vid_f             (_connected_to_vip_pipeline_cvo_clocked_video_vid_f_),            //  output,    width = 1,
.vip_pipeline_cvo_clocked_video_vid_h             (_connected_to_vip_pipeline_cvo_clocked_video_vid_h_),            //  output,    width = 1,
.vip_pipeline_cvo_clocked_video_vid_v             (_connected_to_vip_pipeline_cvo_clocked_video_vid_v_)             //  output,    width = 1,
```

2. Open `c10_dp_demo.v` and look for where the **dp_core_i** is instantiated (around line 205). Scroll right after the last port declared and paste the new ones. You can add some comments to the code, if you wish.



```
    .dp_tx_dp_source_tx_pll_locked                  (dp_txpll_locked),
    // ------------------------------------------------------------------
    // VIP Pipeline Sub-System
    // ------------------------------------------------------------------
    .vip_pipeline_clock_vip_in_clk_clk              (_connected_to_vip_pipeline_clock_vip_in_clk_clk_),
    .vip_pipeline_cvo_clocked_video_vid_clk         (_connected_to_vip_pipeline_cvo_clocked_video_vid_clk_),
    .vip_pipeline_cvo_clocked_video_vid_data        (_connected_to_vip_pipeline_cvo_clocked_video_vid_data_),
    .vip_pipeline_cvo_clocked_video_underflow       (_connected_to_vip_pipeline_cvo_clocked_video_underflow_),
    .vip_pipeline_cvo_clocked_video_vid_mode_change (_connected_to_vip_pipeline_cvo_clocked_video_vid_mode_change_),
    .vip_pipeline_cvo_clocked_video_vid_std         (_connected_to_vip_pipeline_cvo_clocked_video_vid_std_),
    .vip_pipeline_cvo_clocked_video_vid_datavalid   (_connected_to_vip_pipeline_cvo_clocked_video_vid_datavalid_),
    .vip_pipeline_cvo_clocked_video_vid_v_sync      (_connected_to_vip_pipeline_cvo_clocked_video_vid_v_sync_),
    .vip_pipeline_cvo_clocked_video_vid_h_sync      (_connected_to_vip_pipeline_cvo_clocked_video_vid_h_sync_),
    .vip_pipeline_cvo_clocked_video_vid_f           (_connected_to_vip_pipeline_cvo_clocked_video_vid_f_),
    .vip_pipeline_cvo_clocked_video_vid_h           (_connected_to_vip_pipeline_cvo_clocked_video_vid_h_),
    .vip_pipeline_cvo_clocked_video_vid_v           (_connected_to_vip_pipeline_cvo_clocked_video_vid_v_)
    );
```

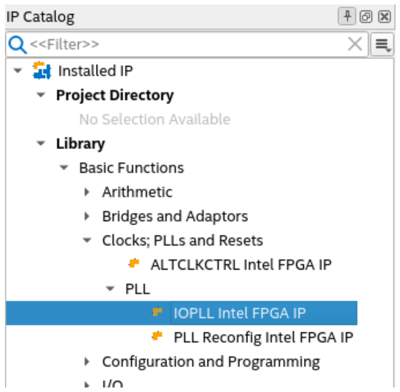
3. Now, let's make the connections. Scroll to where **bitec_clkrec_i** is declared and "disconnect" the signals driving the DisplayPort output interface. We will connect out VIP created system instead.



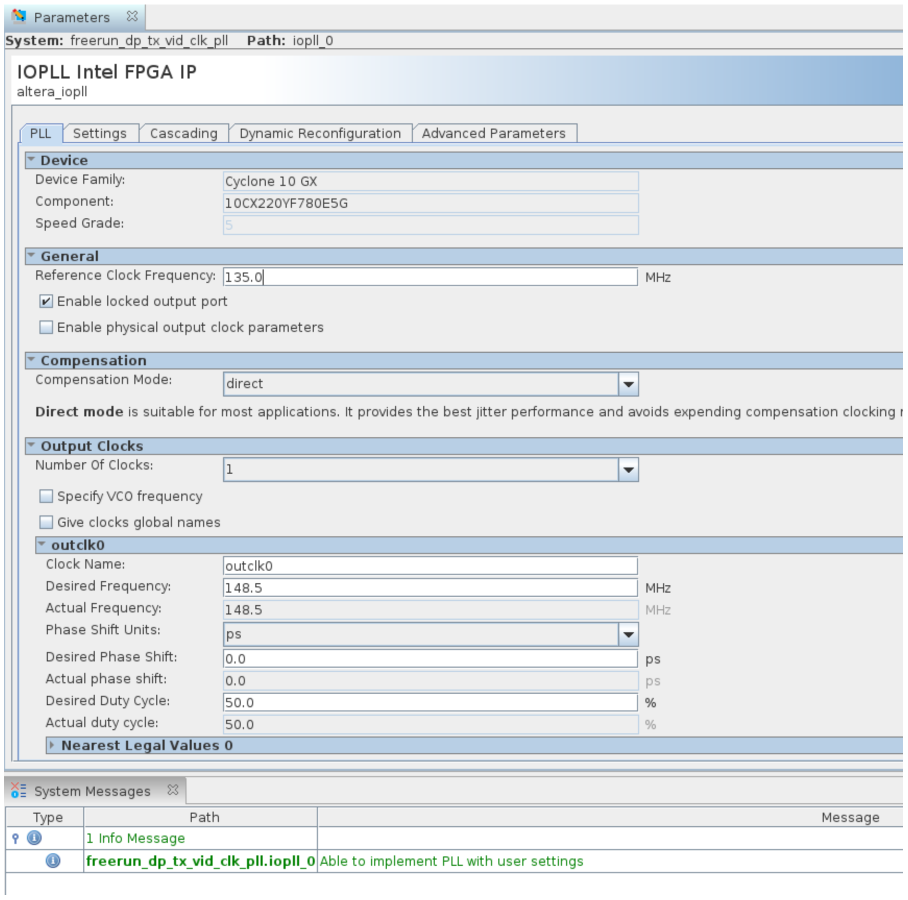
```

    .reset_out              (),
    .rec_clk                (),         //tx_vid_clk
    .vidout                 (),         //tx_vid_data
    .hsync                  (),         //tx_vid_hsync
    .vsync                  (),         //tx_vid_vsync
    .de                     ()          //tx_vid_de
);
```

4. We need to generate our **pixel_clock=148.5MHz** signal. For that, we will use an **IOPLL** with an already available 135MHz clock as input reference.

Double click on the **IP Catalog->Library->Basic Functions->Clocks; PLLs and Resets->IOPLL Intel FPGA**to open the parameters GUI



Select a name for the new variant: `<project_dir>/rtl/`**`freerun_dp_tx_vid_clk_pll`** and parameterize. We will be touching only the PLL tab to select a Reference Frequency=135MHz and an Output Clock Frequency=148.5MHz. The System Messages tab says that is able to implement the PLL, so we are ready to go



System: freerun_dp_tx_vid_clk_pll **Path:** iopll_0

IOPLL Intel FPGA IP
altera_iopll

Device

Device Family:	Cyclone 10 GX	
Component:	10CX220YF780E5G	
Speed Grade:	5	

General

Reference Clock Frequency:	135.0	MHz

- [✓] Enable locked output port
- [ ] Enable physical output clock parameters

Compensation

Compensation Mode: direct

Direct mode is suitable for most applications. It provides the best jitter performance and avoids expending compensation clocking

Output Clocks

Number Of Clocks: 1

- [ ] Specify VCO frequency
- [ ] Give clocks global names

outclk0

Clock Name:	outclk0	
Desired Frequency:	148.5	MHz
Actual Frequency:	148.5	MHz
Phase Shift Units:	ps	
Desired Phase Shift:	0.0	ps
Actual phase shift:	0.0	ps
Desired Duty Cycle:	50.0	%
Actual duty cycle:	50.0	%

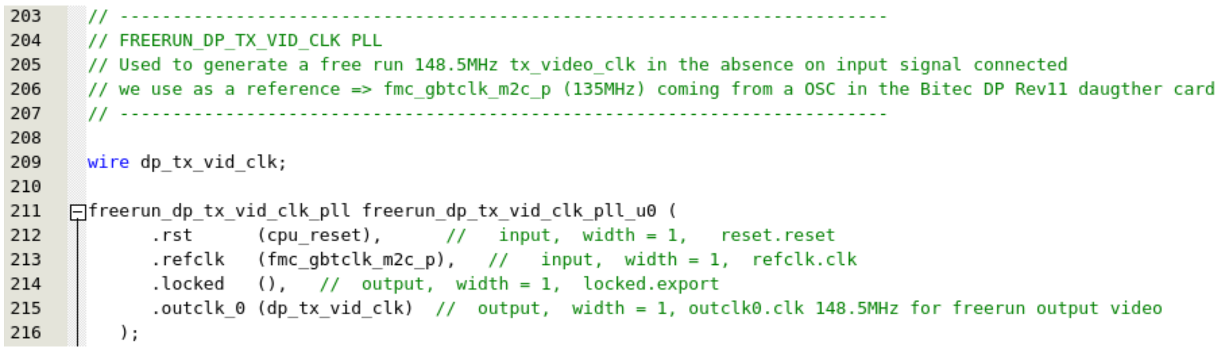
Nearest Legal Values 0

Type	Path	Message
	1 Info Message	
	freerun_dp_tx_vid_clk_pll.iopll_0	Able to implement PLL with user settings

Close the IP Parameter editor, choose **No** when prompted to generate, we will do it later.

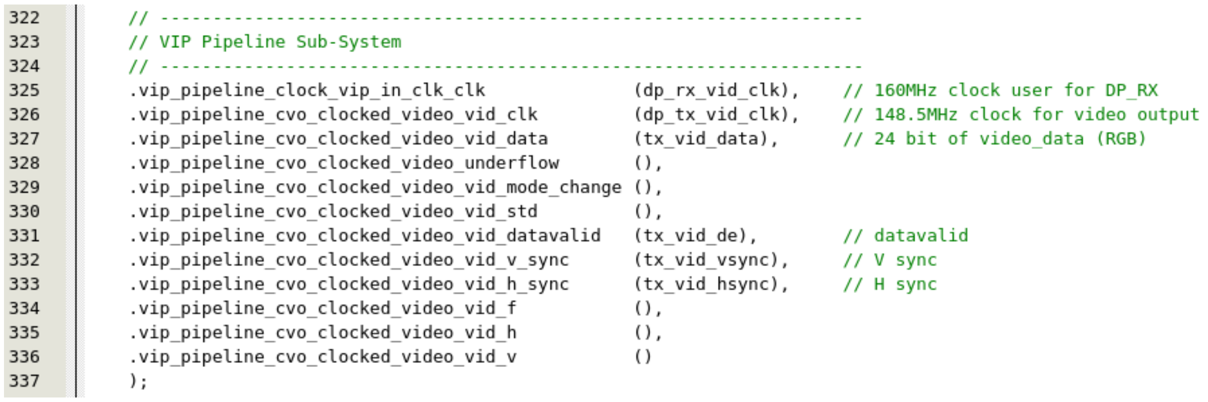
5. Open the file `<project_dir>/rtl/freerun_dp_tx_vid_clk_pll/freerun_dp_tx_vid_clk_pll_inst.v` and copy the content.

6. Find your desired place in top level `c10_dp_demo.v` file and paste the instantiation template for the PLL. Declare a new wire signal for the ***outclk_0*** port (**dp_tx_vid_clk**), make the remaining connections and add some comments to your code.



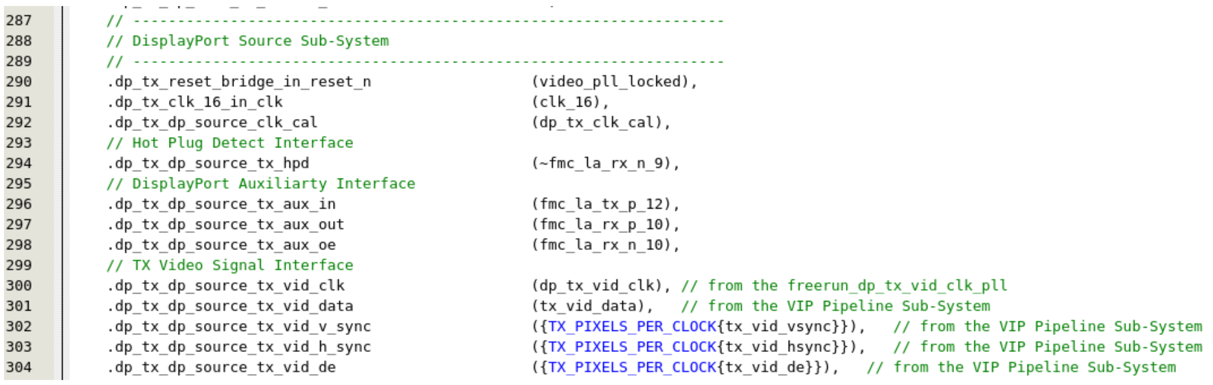
```
// ------------------------------------------------------------------------
// FREERUN_DP_TX_VID_CLK PLL
// Used to generate a free run 148.5MHz tx_video_clk in the absence on input signal connected
// we use as a reference => fmc_gbtclk_m2c_p (135MHz) coming from a OSC in the Bitec DP Rev11 daugther card
// ------------------------------------------------------------------------

wire dp_tx_vid_clk;

freerun_dp_tx_vid_clk_pll freerun_dp_tx_vid_clk_pll_u0 (
      .rst       (cpu_reset),      //   input,  width = 1,   reset.reset
      .refclk    (fmc_gbtclk_m2c_p),   //   input,  width = 1,  refclk.clk
      .locked    (),   //  output,  width = 1,  locked.export
      .outclk_0 (dp_tx_vid_clk)  //  output,  width = 1, outclk0.clk 148.5MHz for freerun output video
   );
```

7. Now, let's make the connection to out VIP created interface



```
    // -----------------------------------------------------------------
    // VIP Pipeline Sub-System
    // -----------------------------------------------------------------
    .vip_pipeline_clock_vip_in_clk_clk             (dp_rx_vid_clk),    // 160MHz clock user for DP_RX
    .vip_pipeline_cvo_clocked_video_vid_clk        (dp_tx_vid_clk),    // 148.5MHz clock for video output
    .vip_pipeline_cvo_clocked_video_vid_data       (tx_vid_data),      // 24 bit of video_data (RGB)
    .vip_pipeline_cvo_clocked_video_underflow      (),
    .vip_pipeline_cvo_clocked_video_vid_mode_change (),
    .vip_pipeline_cvo_clocked_video_vid_std        (),
    .vip_pipeline_cvo_clocked_video_vid_datavalid  (tx_vid_de),        // datavalid
    .vip_pipeline_cvo_clocked_video_vid_v_sync     (tx_vid_vsync),     // V sync
    .vip_pipeline_cvo_clocked_video_vid_h_sync     (tx_vid_hsync),     // H sync
    .vip_pipeline_cvo_clocked_video_vid_f          (),
    .vip_pipeline_cvo_clocked_video_vid_h          (),
    .vip_pipeline_cvo_clocked_video_vid_v          ()
    );
```

8. Modify the `dp_source_tx_vid_clk` port and connect the clock generated with our added PLL. The rest of connections are coming from the VIP subsystem (CVO)



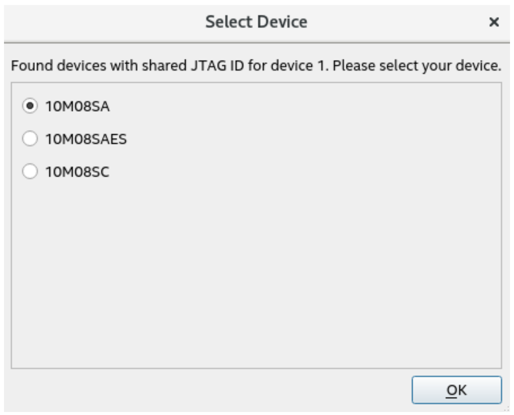
```
// -----------------------------------------------------------------
// DisplayPort Source Sub-System
// -----------------------------------------------------------------
.dp_tx_reset_bridge_in_reset_n                  (video_pll_locked),
.dp_tx_clk_16_in_clk                            (clk_16),
.dp_tx_dp_source_clk_cal                        (dp_tx_clk_cal),
// Hot Plug Detect Interface
.dp_tx_dp_source_tx_hpd                         (~fmc_la_rx_n_9),
// DisplayPort Auxiliarty Interface
.dp_tx_dp_source_tx_aux_in                      (fmc_la_tx_p_12),
.dp_tx_dp_source_tx_aux_out                     (fmc_la_rx_p_10),
.dp_tx_dp_source_tx_aux_oe                      (fmc_la_rx_n_10),
// TX Video Signal Interface
.dp_tx_dp_source_tx_vid_clk                     (dp_tx_vid_clk), // from the freerun_dp_tx_vid_clk_pll
.dp_tx_dp_source_tx_vid_data                    (tx_vid_data),   // from the VIP Pipeline Sub-System
.dp_tx_dp_source_tx_vid_v_sync                  ({TX_PIXELS_PER_CLOCK{tx_vid_vsync}}),   // from the VIP Pipeline Sub-System
.dp_tx_dp_source_tx_vid_h_sync                  ({TX_PIXELS_PER_CLOCK{tx_vid_hsync}}),   // from the VIP Pipeline Sub-System
.dp_tx_dp_source_tx_vid_de                      ({TX_PIXELS_PER_CLOCK{tx_vid_de}}),   // from the VIP Pipeline Sub-System
```

9. These are all the changes needed. We can now compile our design and generate the FPGA bitstream. Go to **Processing-> Start_Compilation** to trigger the process. It will take ~8 minutes, depending on machine configuration.

2.7. Configuring the FPGA device

Open the **Quartus Programmer**, select your USB-Blaster cable in the Hardware Setup and click on **Auto Detect** to retrieve the JTAG chain on the Cyclone10 GX Devkit.

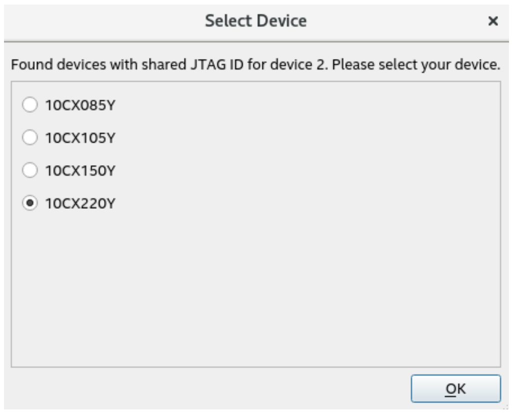
When prompted, select 10M08SA & 10CX220Y as target devices



Select Device

Found devices with shared JTAG ID for device 1. Please select your device.

[✓ 10M08SA] 10M08SAES 10M08SC



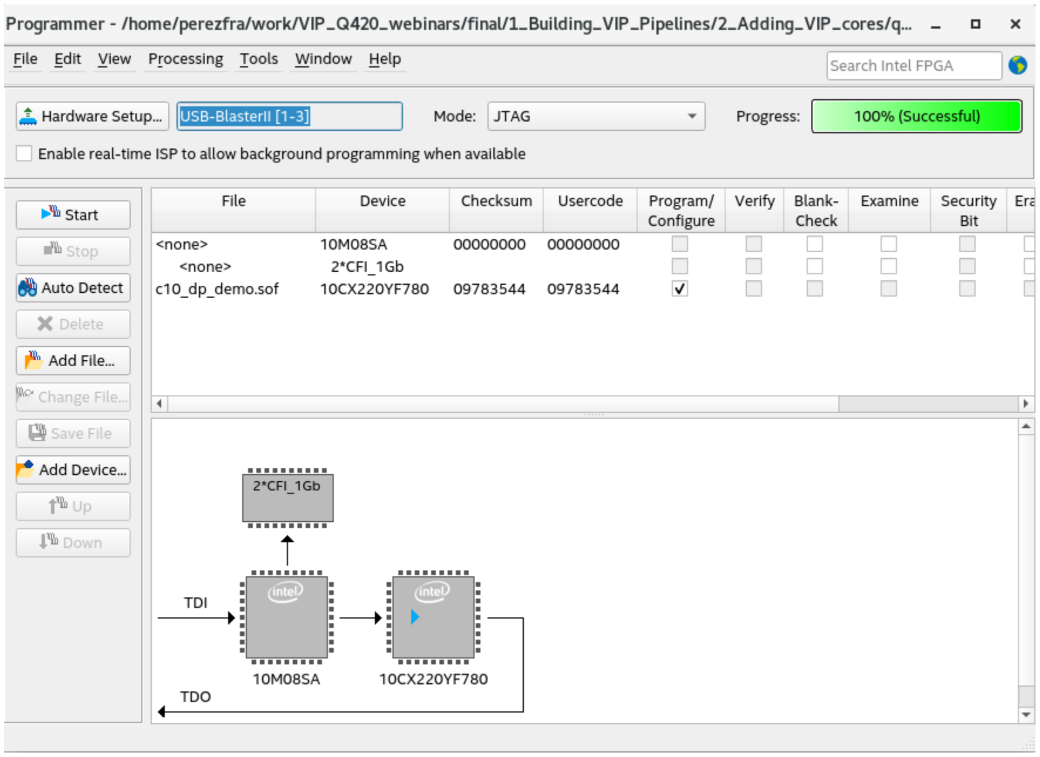
Select Device

Found devices with shared JTAG ID for device 2. Please select your device.

10CX085Y 10CX105Y 10CX150Y [✓ 10CX220Y]

Then, select the **10CX220YF780** device and click on **Change File** option, use *<project_dir>/quartus/c10_dp_demo.sof* as configuration file.

Enable **Program/Configure** option and click on **Start** button. You should see a 100% successful result in the **Progress** Bar.



Programmer - /home/perezfra/work/VIP_Q420_webinars/final/1_Building_VIP_Pipelines/2_Adding_VIP_cores/q...

Hardware Setup... USB-BlasterII [1-3]   Mode: JTAG   Progress: 100% (Successful)

Enable real-time ISP to allow background programming when available

File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank-Check	Examine	Security Bit
<none>	10M08SA	00000000	00000000					
<none>	2*CFI_1Gb							
c10_dp_demo.sof	10CX220YF780	09783544	09783544	✓				

3. Building the software application

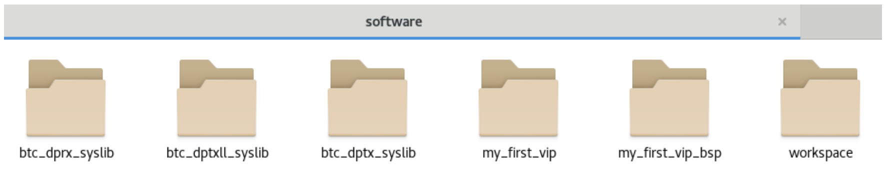
3.1.Setting up the Eclipse for Nios project

Follow the steps in the previous guide "*Session 1.1 – DisplayPort Loopback Design*" to create a **workspace**, an **application** and a **bsp**. You can set:

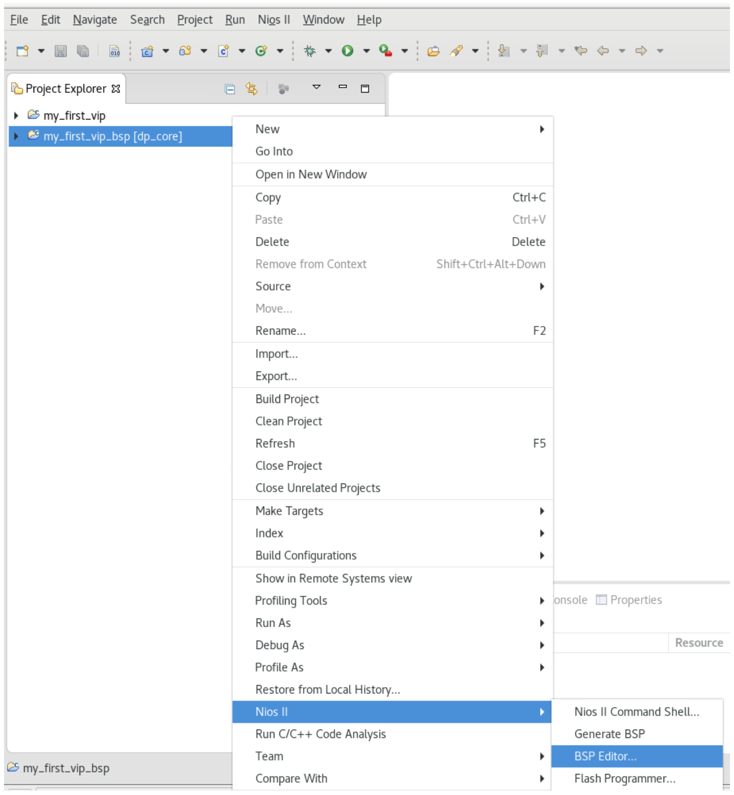
- workspace: `<project_dir>/software/workspace`
- application and bsp: `<project_dir>/software/my_first_vip`

Note: Use `<project_dir>/rtl/core/dp_core/dp_core.sopcinfo` as Target Hardware for generation

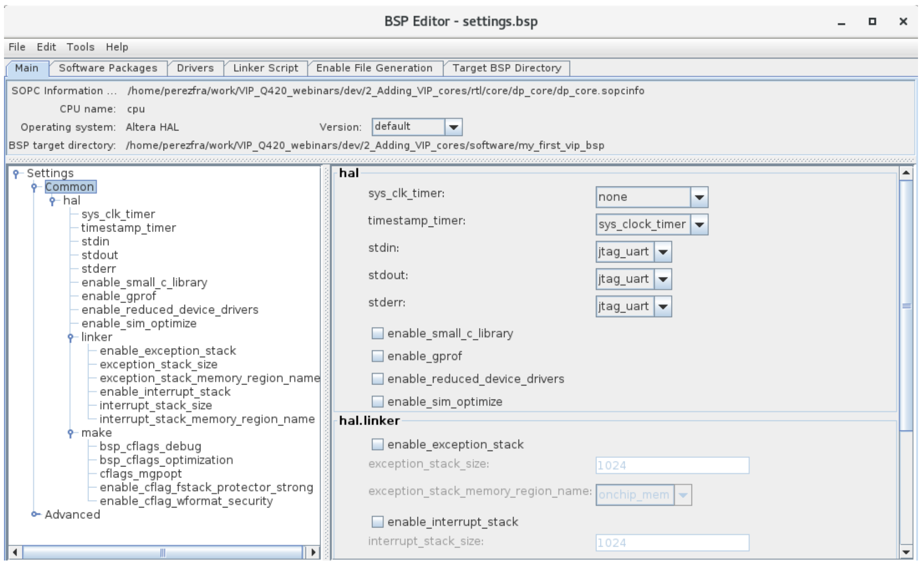
Then, you need to copy all the DisplayPort libraries generated in the first lab to `<project_dir>/software`. You will end up with a setup like the following



1. Configure the **bsp**. In eclipse, right-click on **Project Explorer->my_first_vip_bsp** and select **Nios II->BSP Editor**



- The **BSP Editor** opens, make the following changes:
    - **Settings->Common->hal->sys_clk_timer**: none
    - **Settings->Common->hal->timestamp_timer**: sys_clock_timer
    - **Settings->Advanced->hal->log_port**: jtag_uart

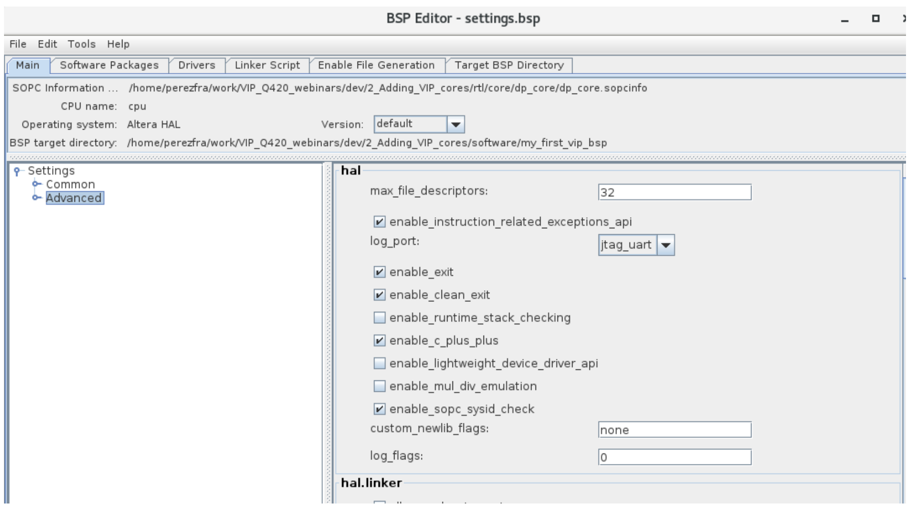

BSP Editor - settings.bsp
File Edit Tools Help
Main | Software Packages | Drivers | Linker Script | Enable File Generation | Target BSP Directory

SOPC Information ... /home/perezfra/work/VIP_Q420_webinars/dev/2_Adding_VIP_cores/rtl/core/dp_core/dp_core.sopcinfo
CPU name: cpu
Operating system: Altera HAL Version: default
BSP target directory: /home/perezfra/work/VIP_Q420_webinars/dev/2_Adding_VIP_cores/software/my_first_vip_bsp

```
Settings
  Common
    hal
      sys_clk_timer
      timestamp_timer
      stdin
      stdout
      stderr
      enable_small_c_library
      enable_gprof
      enable_reduced_device_drivers
      enable_sim_optimize
      linker
        enable_exception_stack
        exception_stack_size
        exception_stack_memory_region_name
        enable_interrupt_stack
        interrupt_stack_size
        interrupt_stack_memory_region_name
      make
        bsp_cflags_debug
        bsp_cflags_optimization
        cflags_mgpopt
        enable_cflag_fstack_protector_strong
        enable_cflag_wformat_security
  Advanced
```

hal
sys_clk_timer: none
timestamp_timer: sys_clock_timer
stdin: jtag_uart
stdout: jtag_uart
stderr: jtag_uart
☐ enable_small_c_library
☐ enable_gprof
☐ enable_reduced_device_drivers
☐ enable_sim_optimize

hal.linker
☐ enable_exception_stack
exception_stack_size: 1024
exception_stack_memory_region_name: onchip_mem
☐ enable_interrupt_stack
interrupt_stack_size: 1024


BSP Editor - settings.bsp
File Edit Tools Help
Main | Software Packages | Drivers | Linker Script | Enable File Generation | Target BSP Directory

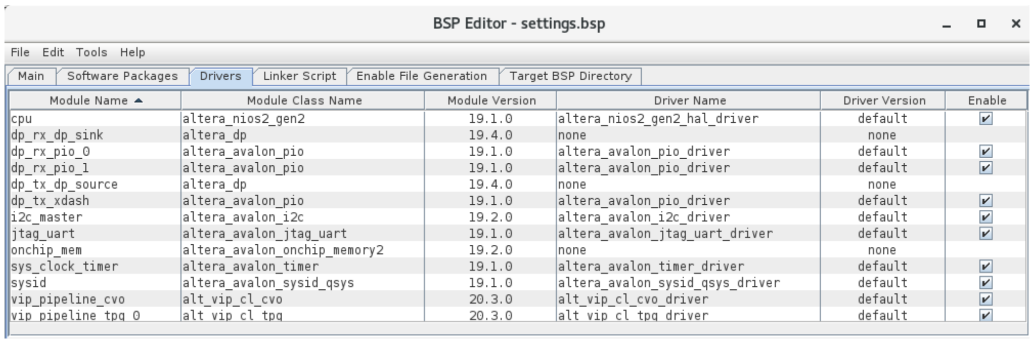
SOPC Information ... /home/perezfra/work/VIP_Q420_webinars/dev/2_Adding_VIP_cores/rtl/core/dp_core/dp_core.sopcinfo
CPU name: cpu
Operating system: Altera HAL Version: default
BSP target directory: /home/perezfra/work/VIP_Q420_webinars/dev/2_Adding_VIP_cores/software/my_first_vip_bsp

```
Settings
  Common
  Advanced
```

hal
max_file_descriptors: 32
☑ enable_instruction_related_exceptions_api
log_port: jtag_uart
☑ enable_exit
☑ enable_clean_exit
☐ enable_runtime_stack_checking
☑ enable_c_plus_plus
☐ enable_lightweight_device_driver_api
☐ enable_mul_div_emulation
☑ enable_sopc_sysid_check
custom_newlib_flags: none
log_flags: 0

hal.linker

Please note that in the **Drivers** tab, the bsp generation has already included supporting code for the VIP components we are using: CVO and TGP.

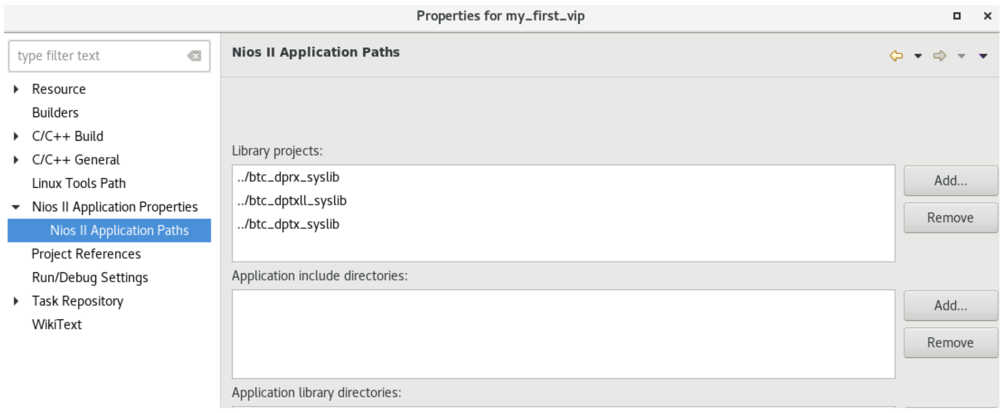


Module Name ▲	Module Class Name	Module Version	Driver Name	Driver Version	Enable
cpu	altera_nios2_gen2	19.1.0	altera_nios2_gen2_hal_driver	default	✓
dp_rx_dp_sink	altera_dp	19.4.0	none	none	
dp_rx_pio_0	altera_avalon_pio	19.1.0	altera_avalon_pio_driver	default	✓
dp_rx_pio_1	altera_avalon_pio	19.1.0	altera_avalon_pio_driver	default	✓
dp_tx_dp_source	altera_dp	19.4.0	none	none	
dp_tx_xdash	altera_avalon_pio	19.1.0	altera_avalon_pio_driver	default	✓
i2c_master	altera_avalon_i2c	19.2.0	altera_avalon_i2c_driver	default	✓
jtag_uart	altera_avalon_jtag_uart	19.1.0	altera_avalon_jtag_uart_driver	default	✓
onchip_mem	altera_avalon_onchip_memory2	19.2.0	none	none	
sys_clock_timer	altera_avalon_timer	19.1.0	altera_avalon_timer_driver	default	✓
sysid	altera_avalon_sysid_qsys	19.1.0	altera_avalon_sysid_qsys_driver	default	✓
vip_pipeline_cvo	alt_vip_cl_cvo	20.3.0	alt_vip_cl_cvo_driver	default	✓
vip_pipeline_tpg_0	alt_vip_cl_tpg	20.3.0	alt_vip_cl_tpg_driver	default	✓

Then click on **Generate** and **Exit**.

2. Adding libraries to the application: In **Project Explorer**, right-click on *my_first_vip* application and select **Properties**.

In the dialog box, select **Nios II Application Properties->Nios II Application Paths** and add the Library projects



Library projects:
../btc_dprx_syslib
../btc_dptxll_syslib
../btc_dptx_syslib

Application include directories:

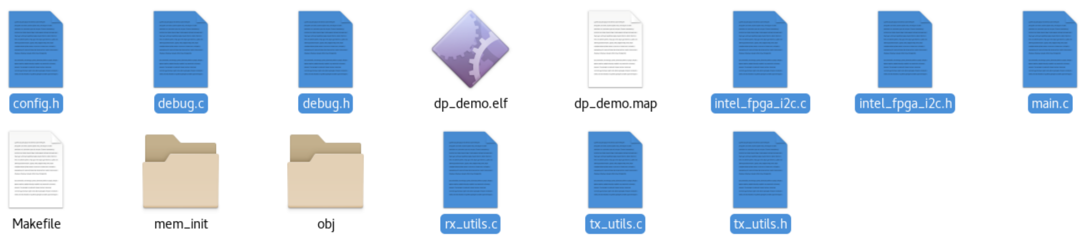
Application library directories:

Then click **Apply** and **OK**

3.2.Importing the code

As we did in the previous example, we are using the generated code with the example design as base application to manage all related to control the DisplayPort source and sink IP cores.

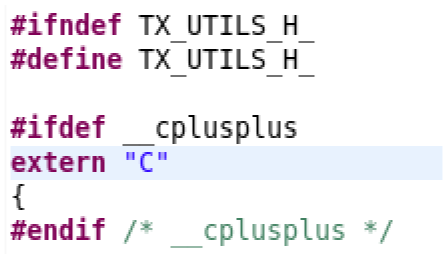
Copy all the source files in *dp_0_example_design_folder/software* into the newly created *<project_dir>/software* folder where our application resides.



3.3.Adapting to C++ application

The auto generated application with the example design is a C application, however the VIP cores are better managed with the provided C++ API. So, if we want to exploit the benefits, we need to make some changes.

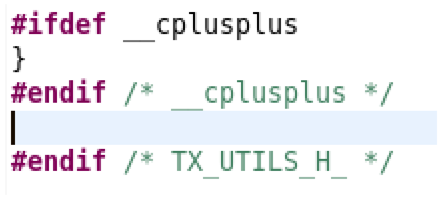
- Rename **main.c** >> **main.cpp**
- We need to add the following excerpts of code to all the header files (.h) in the application



```
#ifndef TX_UTILS_H_
#define TX_UTILS_H_

#ifdef __cplusplus
extern "C"
{
#endif /* __cplusplus */
```

………C

………



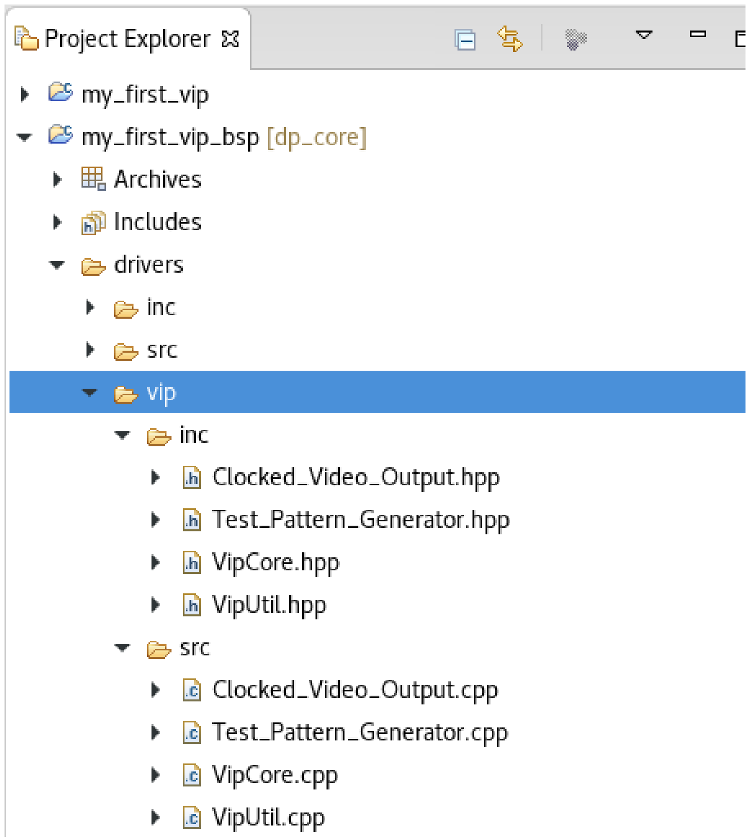
```
#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif /* TX_UTILS_H_ */
```

NOTE: For each file, the **#ifndef** and **#define** should be modified according to the file name.

- We need to do a modification in the `btc_dprx_syslib/btc_dprx_syslib.h` file
  - In the line 47, replace "*BYTE **new** : 1*" by "*BYTE **new_byte** : 1*"
- We also need to expose the functions in `rx_utils.c` by creating a new header file `rx_utils.h`
  - We have already included this file in the software application folder

3.4.Getting familiar with VIP C++ API

If we expand the generated `my_first_vip_bsp` in the **Project Explorer**, we can notice that a new set of drivers have been added to control the VIP cores. In this case, as we have used only CVO and TPG, these are the ones included along with some utility classes and methods generic to all VIP cores (*VipCore.hpp*)



```
▸ my_first_vip
▾ my_first_vip_bsp [dp_core]
    ▸ Archives
    ▸ Includes
    ▾ drivers
        ▸ inc
        ▸ src
        ▾ vip
            ▾ inc
                ▸ Clocked_Video_Output.hpp
                ▸ Test_Pattern_Generator.hpp
                ▸ VipCore.hpp
                ▸ VipUtil.hpp
            ▾ src
                ▸ Clocked_Video_Output.cpp
                ▸ Test_Pattern_Generator.cpp
                ▸ VipCore.cpp
                ▸ VipUtil.cpp
```

All the Video and Image Processing IP cores have an optional simple run-time control interface that comprises a set of control and status registers, accessible through an Avalon Memory-Mapped (Avalon-MM) slave port. A runtime control configuration has a mandatory set of three registers for every IP core, followed by any function-specific registers.

Address	Data		Description
0	Bits 31:1 = X	Bit 0 = `Go`	Control register
1	Bits 31:1 = X	Bit 1 = `Status`	Status register
2	Core specific		Interrupt register

When you enable run-time control in hardware configuration, the *Go* bit gets de-asserted by default. If you do not enable run-time control, the `Go` is asserted by default. Every IP core retains address 2 in its address space to be used as an interrupt register. However, this address is often unused because only some of the IP cores require interrupts.

- **VipCore.hpp**

`VipCore` class contains a series of utility methods generic to all VIP cores. All the VIP core classes are inherited from `VipCore`, so all of them have methods available to:

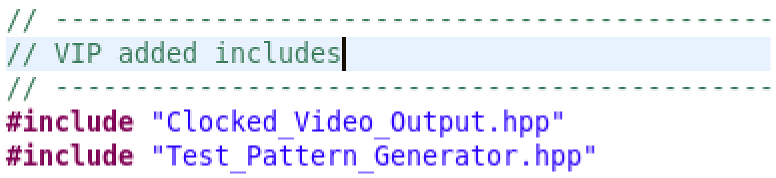
- start() & stop() the execution
- Manage interrupts: enable(), disable(), clear(), …

- Read & write registers: do_read(), do_write()

3.5. Controlling VIP cores from the application

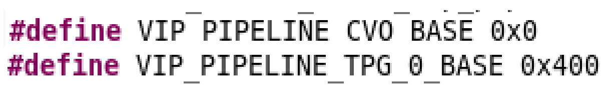
We need to do the following modifications in *main.cpp*

- **Include the header files for each VIP core**, please note that `VipCore.hpp` is added automatically as it's included in every vip_core `*.hpp` file

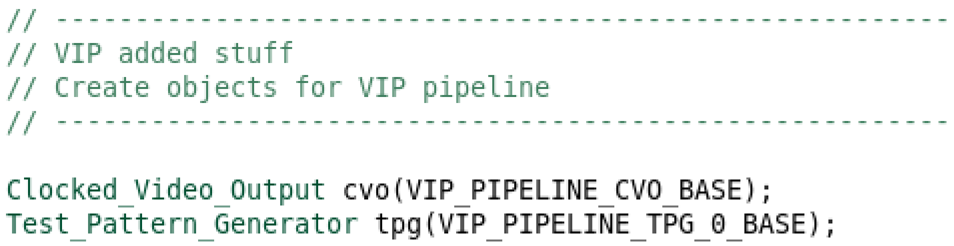


```
// ------------------------------------------
// VIP added includes
// ------------------------------------------
#include "Clocked_Video_Output.hpp"
#include "Test_Pattern_Generator.hpp"
```

- **Create objects to build the pipeline**. We create as many objects of the same class as we have included in our Platform Designer system. Eventually, you can have more than one CVI, CVO, TPG…
  As minimum, you need to provide to the constructor the base address of the component. This information can be extracted from the file *system.h*, located in the bsp folder.



```
#define VIP_PIPELINE_CVO_BASE 0x0
#define VIP_PIPELINE_TPG_0_BASE 0x400
```



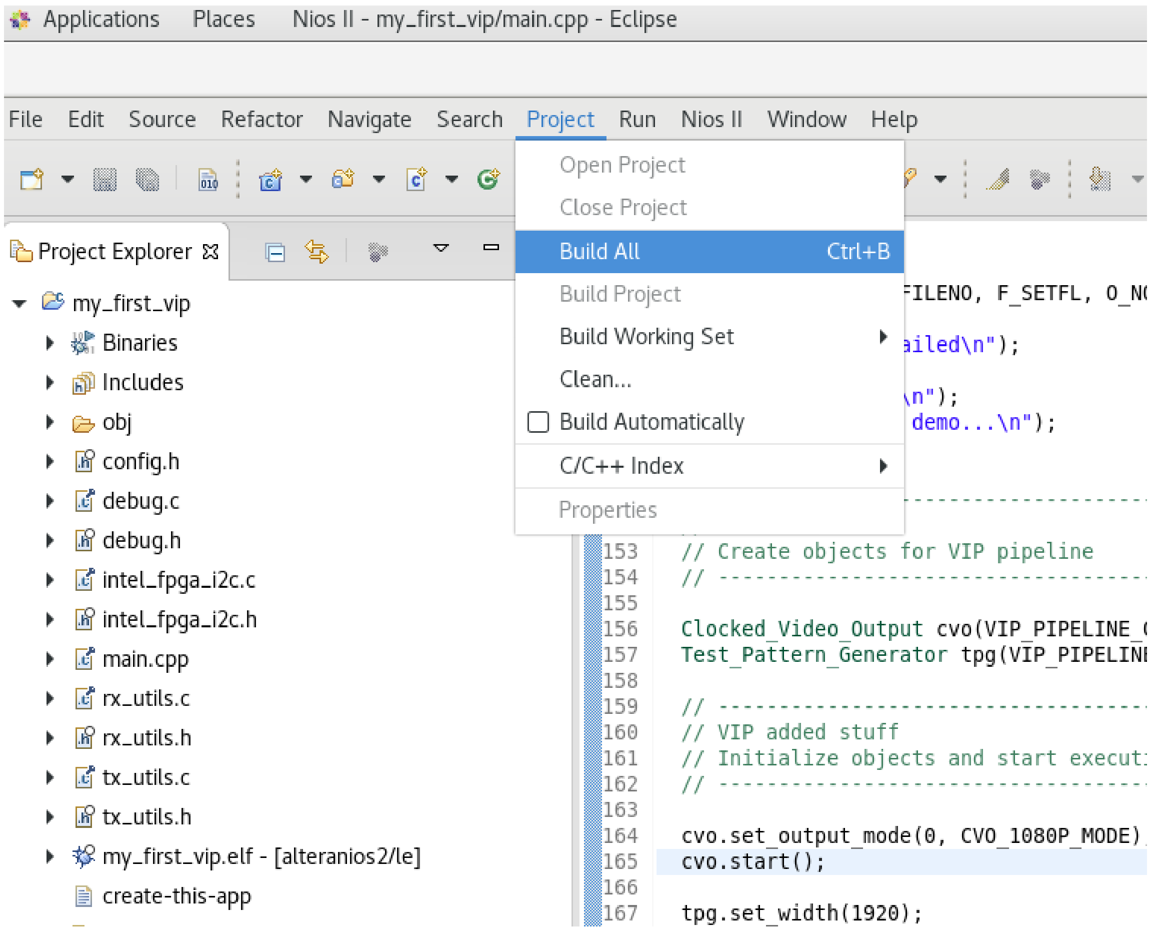
```
// ---------------------------------------------------------
// VIP added stuff
// Create objects for VIP pipeline
// ---------------------------------------------------------

Clocked_Video_Output cvo(VIP_PIPELINE_CVO_BASE);
Test_Pattern_Generator tpg(VIP_PIPELINE_TPG_0_BASE);
```

- **Initialize cores and start execution**. Then we can configure the run-time parameters of our cores (if different from default values assigned in hardware generation) and enable the `Go` bit (using the `start()` method) to begin process video.

```
// -----------------------------------------------------------
// VIP added stuff
// Initialize objects and start execution
// -----------------------------------------------------------

cvo.set_output_mode(0, CVO_1080P_MODE);
cvo.start();

tpg.set_width(1920);
tpg.set_height(1080);
tpg.set_uniform_color(255, 0, 255);
tpg.select_pattern(0);    // according to setup in HW
                          // pattern 0 - Color Bars
                          // pattern 1 - Gray Bars
                          // pattern 2 - Uniform Background
tpg.start();
```

3.6. Building and launching program execution

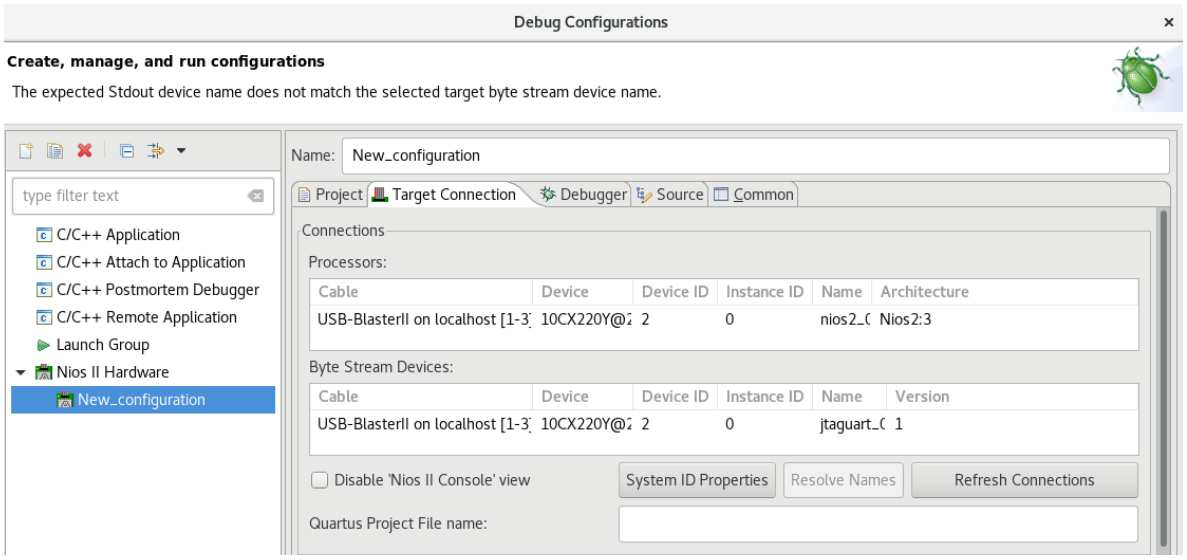
After doing that modifications, we can generate our executable file. In Eclipse IDE, go to **Project->Build All** to compile the *bsp* and generate `my_first_vip.elf` file.



```
FILENO, F_SETFL, O_N
ailed\n");
\n");
demo...\n");

-------------------
// Create objects for VIP pipeline
// -----------------------------------

Clocked_Video_Output cvo(VIP_PIPELINE_
Test_Pattern_Generator tpg(VIP_PIPELIN

// -----------------------------------
// VIP added stuff
// Initialize objects and start execut
// -----------------------------------

cvo.set_output_mode(0, CVO_1080P_MODE)
cvo.start();

tpg.set_width(1920);
```

To launch our application, we can go to **Run->Debug Configurations**... in the main toolbar to open *Debug Configuration* dialog.

We double-click on **Nios II Hardware** option to create a *New_configuration*. In **Target Connection** tab, click on **Refresh Connections** to select the right *USB-BlasterII* adapter


Debug Configurations

Create, manage, and run configurations

The expected Stdout device name does not match the selected target byte stream device name.

Name: New_configuration

Connections

Processors:

Cable	Device	Device ID	Instance ID	Name	Architecture
USB-BlasterII on localhost [1-3]	10CX220Y@2	2	0	nios2_(	Nios2:3

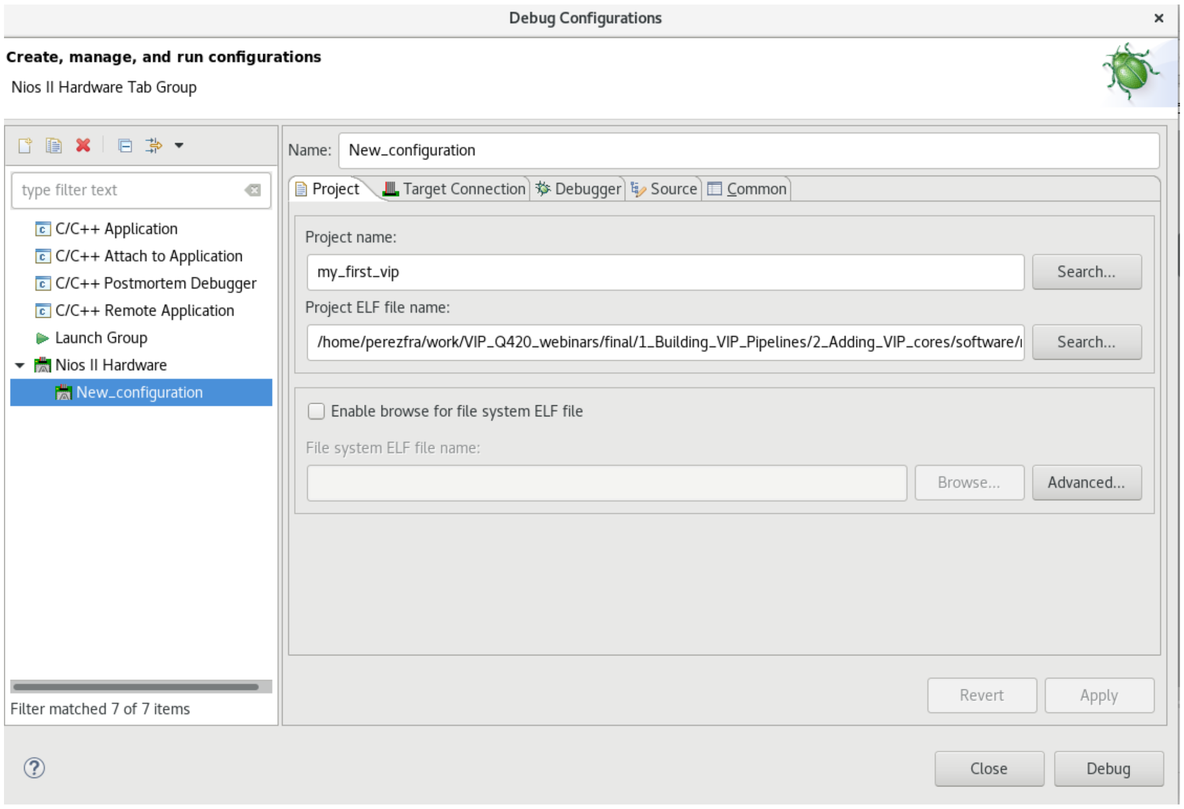
Byte Stream Devices:

Cable	Device	Device ID	Instance ID	Name	Version
USB-BlasterII on localhost [1-3]	10CX220Y@2	2	0	jtaguart_(	1

☐ Disable 'Nios II Console' view

Quartus Project File name:

Back to the **Project** tab, make sure the right *Project Name* and *ELF File Name* are selected and just click on **Apply** and **Debug** to launch the session


Debug Configurations

Create, manage, and run configurations

Nios II Hardware Tab Group

Name: New_configuration

Project name: my_first_vip

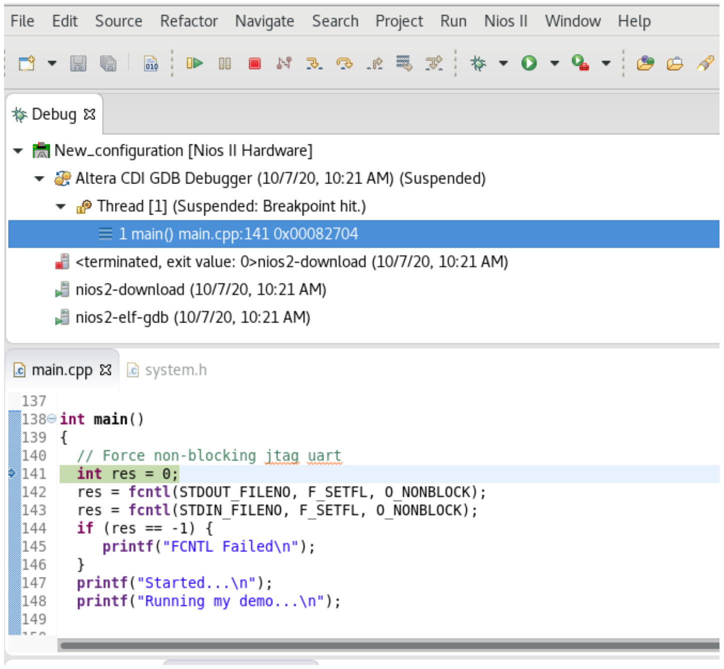
Project ELF file name: /home/perezfra/work/VIP_Q420_webinars/final/1_Building_VIP_Pipelines/2_Adding_VIP_cores/software/

☐ Enable browse for file system ELF file

File system ELF file name:

Filter matched 7 of 7 items

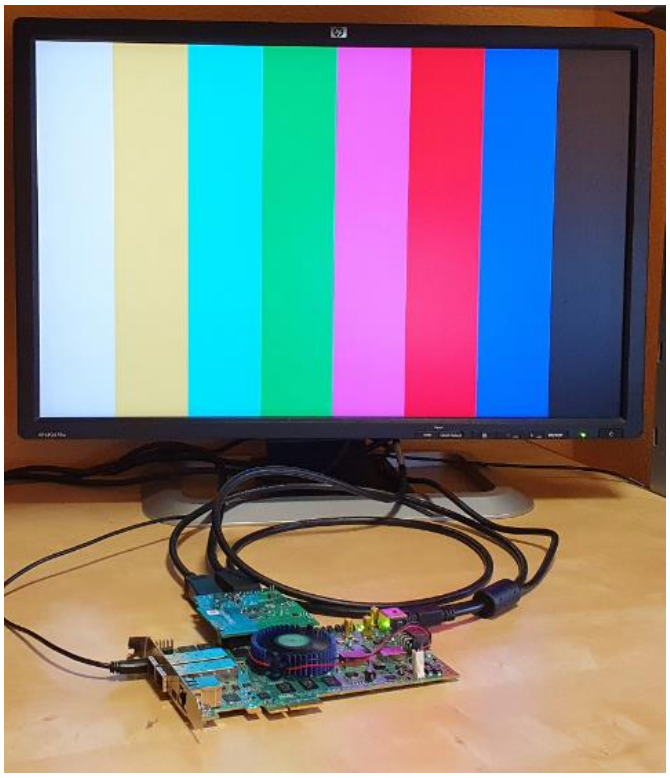
The executable file is then downloaded to the NiosII program memory and the execution is stopped right after main function is called. Just press the **Resume** button in the debugger to launch the complete execution.


File Edit Source Refactor Navigate Search Project Run Nios II Window Help
Debug
main.cpp
system.h

```
New_configuration [Nios II Hardware]
  Altera CDI GDB Debugger (10/7/20, 10:21 AM) (Suspended)
    Thread [1] (Suspended: Breakpoint hit.)
      1 main() main.cpp:141 0x00082704
  <terminated, exit value: 0>nios2-download (10/7/20, 10:21 AM)
  nios2-download (10/7/20, 10:21 AM)
  nios2-elf-gdb (10/7/20, 10:21 AM)
```

```
int main()
{
  // Force non-blocking jtag uart
  int res = 0;
  res = fcntl(STDOUT_FILENO, F_SETFL, O_NONBLOCK);
  res = fcntl(STDIN_FILENO, F_SETFL, O_NONBLOCK);
  if (res == -1) {
     printf("FCNTL Failed\n");
  }
  printf("Started...\n");
  printf("Running my demo...\n");
```

You should see now a nice Color Bars pattern displayed on your attached monitor.



4. Summary

In this lab, we have exercised with our first video pipeline implementation using VIP cores.

- We have developed a Platform Designer subsystem with the video pipeline and we have integrated it with the rest of the control modules.
- We have connected the interfaces with the DisplayPort IP cores
- We have learnt how to build a software application to control the VIP cores from a Nios II Processor