

# 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

<b>1. Introduction .....</b>	<b>3</b>
1.1. Introduction .....	3
1.2. Requirements .....	3
1.3. References.....	3
1.4. Implementation diagram.....	4
<b>2. Generating the hardware pipeline.....</b>	<b>6</b>
2.1. Setting up the Quartus project.....	6
2.2. Examining the project .....	6
2.3. Understanding the external connections .....	8
2.4. Build the video pipeline in Platform Designer .....	10
2.5. Integrating the VIP pipeline.....	16
2.6. Integrate the complete module at top level.....	19
2.7. Configuring the FPGA device .....	22
<b>3. Building the software application .....</b>	<b>23</b>
3.1. Setting up the Eclipse for Nios project .....	23
3.2. Importing the code.....	25
3.3. Adapting to C++ application .....	26
3.4. Getting familiar with VIP C++ API .....	26
3.5. Controlling VIP cores from the application.....	28
3.6. Building and launching program execution .....	29
<b>4. Summary .....</b>	<b>32</b>

# 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.

NOTE: You can follow all the steps in this guide, taking as a base project the one generated in the previous guide “**Session1\_1\_DP\_loopback\_design\_v1.pdf**”. Alternatively, you can find all the modified files in the archived project “**1\_2\_Adding\_VIP\_cores.tar.gz**” located in the repository.

**Hardware flow:** open `<project_dir>/quartus/c10_dp_demo.qpf` in Quartus Pro and Start Compilation to generate FPGA programming file.

**Software flow:** Follow the steps in the chapter 3 “Building the software applications”, but use the source files located in the `<project_dir>/software/source` directory instead of copying from the previous project.

## 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](https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/cyclone-10-gx-development-kit.html)
- Bitech DisplayPort daughter card rev.11  
<https://bitech-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](https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_vip.pdf)

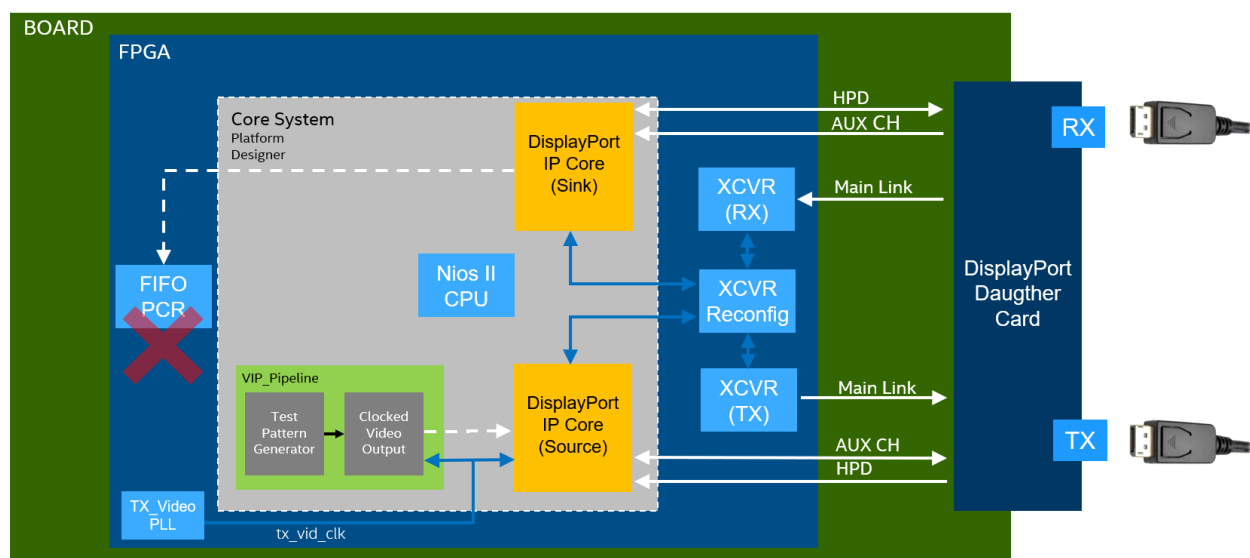
- AN745-Design Guidelines for DisplayPort Interface  
[https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/an/an\\_745.pdf](https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/an/an_745.pdf)
- Quartus Prime Pro Installation Guide  
[https://www.intel.com/content/dam/altera-www/global/en\\_US/pdfs/literature/manual/quartus\\_install.pdf](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](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](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).

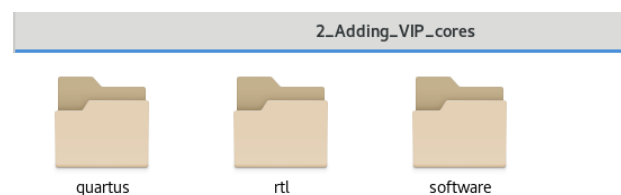
## 2. Generating the hardware pipeline

### 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

☐ Show subsystem modules ☐ Hide unconnectable interfaces Remove Dangling Exports

Use	Co...	Name	Description	Export	Clock	Base	End
<input checked="" type="checkbox"/>		cpu_reset_bridge	Reset Bridge Intel FPGA IP		clk_100...		
<input checked="" type="checkbox"/>		dp_rx_reset_bridge	Reset Bridge Intel FPGA IP				
<input checked="" type="checkbox"/>		dp_tx_reset_bridge	Reset Bridge Intel FPGA IP				
<input checked="" type="checkbox"/>		clk_100	Clock Bridge Intel FPGA IP		exported		
<input checked="" type="checkbox"/>		dp_rx_clk_16	Clock Bridge Intel FPGA IP		exported		
<input checked="" type="checkbox"/>		dp_tx_clk_16	Clock Bridge Intel FPGA IP		exported		
<input checked="" type="checkbox"/>		cpu	Nios II Processor		clk_100...	0x0010_2800	0x0010_2fff
<input checked="" type="checkbox"/>		onchip_mem	On-Chip Memory (RAM or ROM) In...		clk_100...	0x0008_0000	0x000c_5bfff
<input checked="" type="checkbox"/>		jtag_uart	JTAG UART Intel FPGA IP		clk_100...	0x0010_3048	0x0010_304f
<input checked="" type="checkbox"/>		sys_clock_timer	Interval Timer Intel FPGA IP		clk_100...	0x0010_3000	0x0010_301f
<input checked="" type="checkbox"/>		sysid	System ID Peripheral Intel FPGA IP		clk_100...	0x0010_3040	0x0010_3047
<input checked="" type="checkbox"/>		i2c_master	Avalon I2C (Master) Intel FPGA IP		clk_100...	0x0010_3080	0x0010_30bf
<input checked="" type="checkbox"/>		dp_rx	dp_rx		multiple	0x0010_1000	0x0010_1fff
<input checked="" type="checkbox"/>		dp_tx	dp_tx		multiple	0x0010_0000	0x0010_0fff

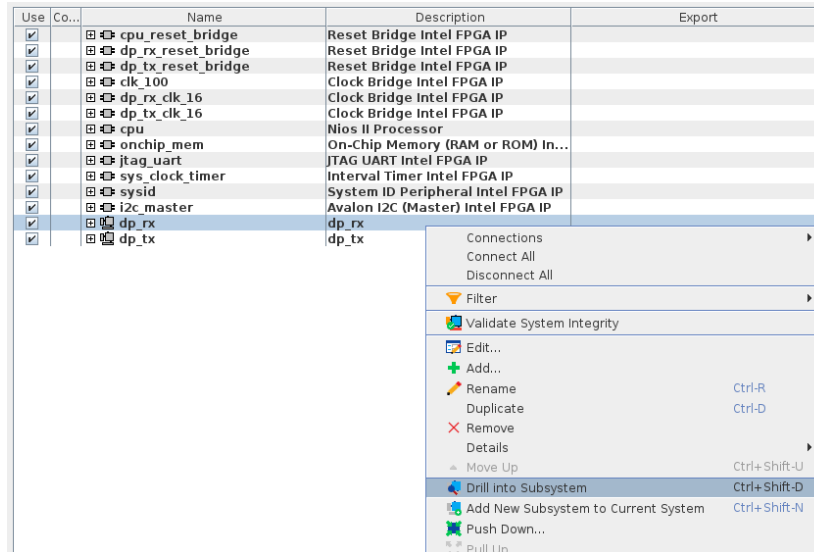
This Platform Designer system contains different IP variations:

- **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:



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.



Use	Connections	Name	Description	Export
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	reset_bridge	Reset Bridge Intel FPGA IP	reset_bridge_in_reset
		in_reset	Reset Input	Double-click to export
		out_reset	Reset Output	Double-click to export
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	clk_100	Clock Bridge Intel FPGA IP	clk_100_in_clk
		in_clk	Clock Input	Double-click to export
		out_clk	Clock Output	Double-click to export
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	clk_16	Clock Bridge Intel FPGA IP	clk_16_in_clk
		in_clk	Clock Input	Double-click to export
		out_clk	Clock Output	Double-click to export
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	dp_sink	DisplayPort Intel FPGA IP	dp_sink_rx_xcvr_interface
		rx_xcvr_interface	Conduit	dp_sink_rx_aux
		rx_aux	Conduit	dp_sink_rx_params
		rx_params	Conduit	dp_sink_rx_video_out
		rx_video_out	Conduit	dp_sink_rx_vid_clk
		rx_vid_clk	Clock Input	dp_sink_rx_stream
		rx_stream	Conduit	dp_sink_rx_msa_conduit
		rx_msa_conduit	Conduit	dp_sink_rx_reconfig
		rx_reconfig	Conduit	dp_sink_rx_analog_reconfig
		rx_analog_reconfig	Conduit	Double-click to export
		rx_mgmt	Avalon Memory Mapped Slave	dp_sink_rx_mgmt_interrupt
		rx_mgmt_interrupt	Interrupt Sender	Double-click to export
		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
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	dp_rx_mgmt_bridge	Avalon Memory Mapped Pipeline...	Double-click to export
		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
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	pio_0	PIO (Parallel I/O) Intel FPGA IP	Double-click to export
		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
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	pio_1	PIO (Parallel I/O) Intel FPGA IP	Double-click to export
		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**

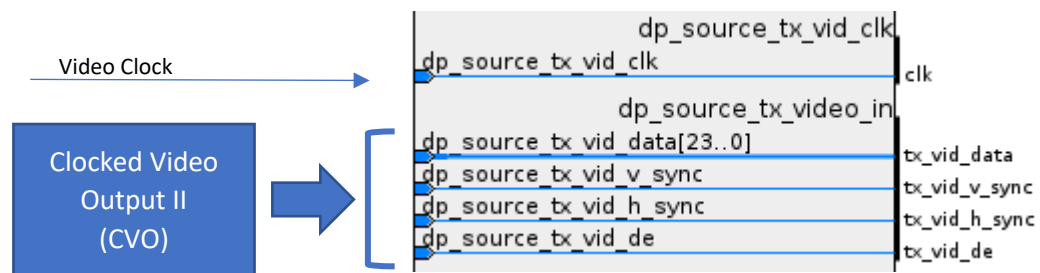
Use	Co...	Name	Description	Export	Cl
<input checked="" type="checkbox"/>		cpu_reset_bridge	Reset Bridge Intel FPGA IP		clk_1
<input checked="" type="checkbox"/>		dp_rx_reset_bridge	Reset Bridge Intel FPGA IP		
<input checked="" type="checkbox"/>		dp_tx_reset_bridge	Reset Bridge Intel FPGA IP		
<input checked="" type="checkbox"/>		clk_100	Clock Bridge Intel FPGA IP		expo
<input checked="" type="checkbox"/>		dp_rx_clk_16	Clock Bridge Intel FPGA IP		expo
<input checked="" type="checkbox"/>		dp_tx_clk_16	Clock Bridge Intel FPGA IP		expo
<input checked="" type="checkbox"/>		cpu	Nios II Processor		clk_1
<input checked="" type="checkbox"/>		onchip_mem	On-Chip Memory (RAM or RO...		clk_1
<input checked="" type="checkbox"/>		jtag_uart	JTAG UART Intel FPGA IP		clk_1
<input checked="" type="checkbox"/>		sys_clock_timer	Interval Timer Intel FPGA IP		clk_1
<input checked="" type="checkbox"/>		sysid	System ID Peripheral Intel F...		clk_1
<input checked="" type="checkbox"/>		i2c_master	Avalon I2C (Master) Intel FP...		clk_1
<input checked="" type="checkbox"/>		dp_rx	dp_rx		mult
<input checked="" type="checkbox"/>		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



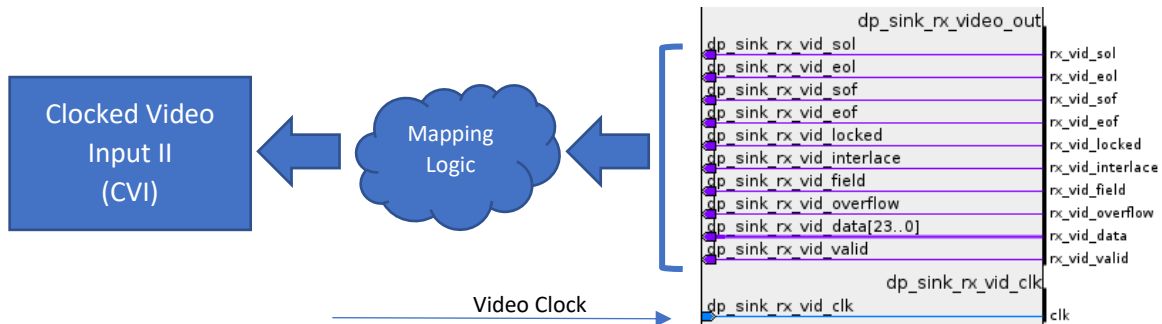
The `dp_source_tx_video_in` interface allows glue-less connection with Clock Video Output component.

- **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	
	clk_16_in_clk	Clock Input	
	dp_rx_mgmt_bridge_s0	Avalon Memory Mapped Slave	
	dp_sink_rx_xcvr_interface	Conduit	Double-click to export
	dp_sink_rx_aux	Conduit	Double-click to export
	dp_sink_rx_params	Conduit	Double-click to export
	dp_sink_rx_video_out	Conduit	
	dp_sink_rx_vid_clk	Clock Input	
	dp_sink_rx_stream	Conduit	
	dp_sink_rx_msa_conduit	Conduit	
	dp_sink_rx_reconfig	Conduit	
	dp_sink_rx_analog_reconfig	Conduit	
	dp_sink_rx_mgmt_interrupt	Interrupt Sender	
	dp_sink_clk_cal	Clock Input	
	pio_0_external_connection	Conduit	
	pio_1_external_connection	Conduit	
	reset_bridge_in_reset	Reset Input	

We are interested in how to connect our 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

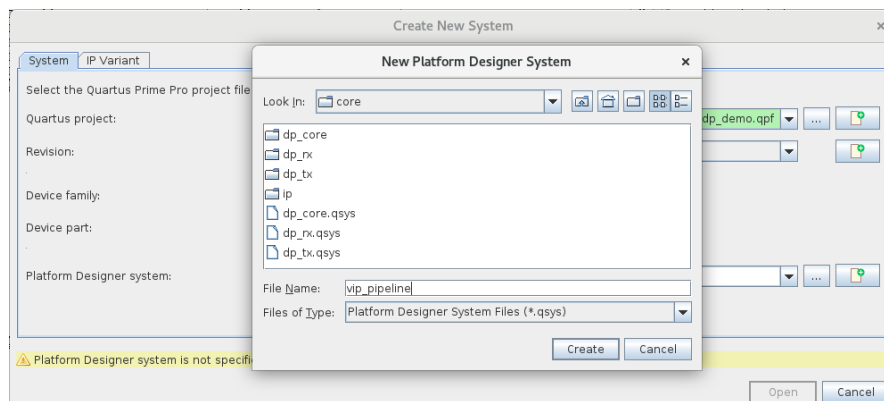


## 2.4. Build the video pipeline in Platform Designer

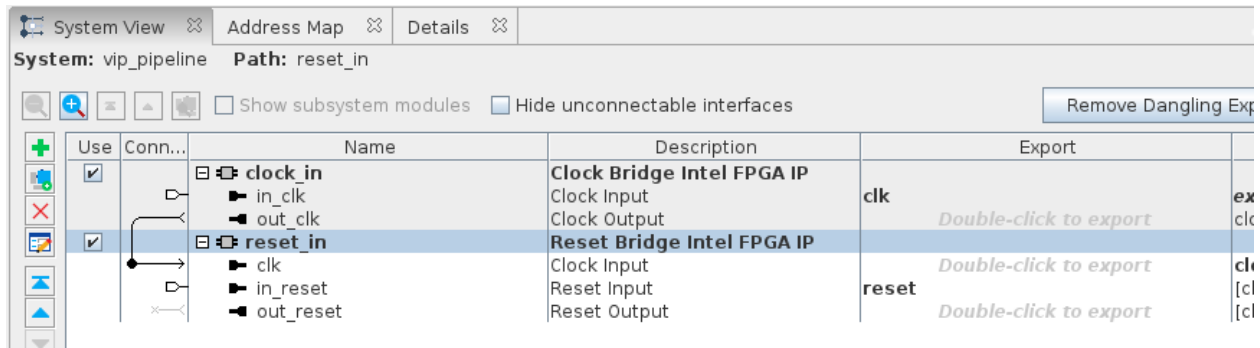
Now we know how to connect the CVI & CVO components together with the DisplayPort IP cores, let's build a simple design to test that we are generating correctly a video output signal to be displayed in an external monitor.

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

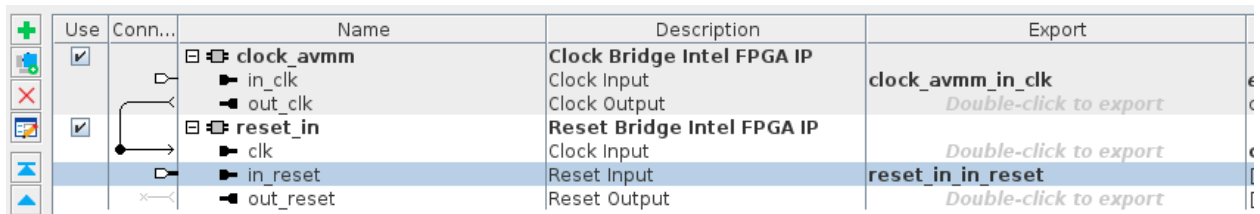
1. Open Platform Designer
2. Go to **File->New System** in the main toolbar to open the dialog. Use **vip\_pipeline.qsys** as File Name and save it in `<project_dir>/rtl/core`. Then click on create.



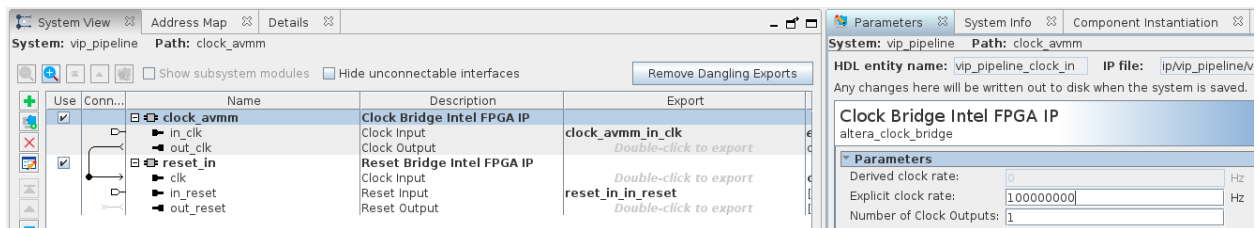
3. A new Platform Designer file will appear in the editor with 2 components already instantiated in the System View pane. A **Clock Bridge** and a **Reset Bridge**. Which are mandatory components in every system.



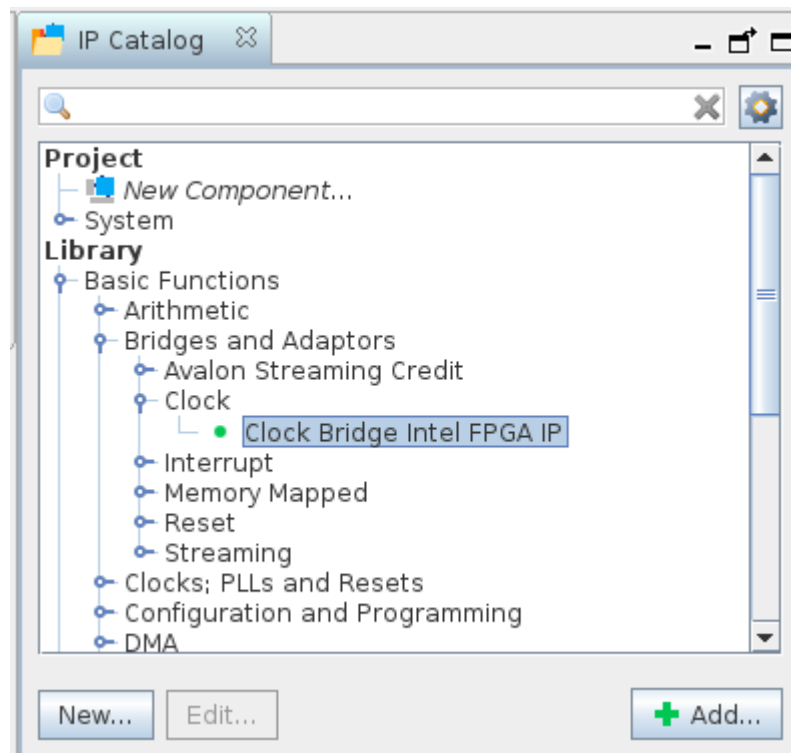
4. Let's rename the **clock\_in** component to **clock\_avmm** and delete the exported signals *clk* & *reset*. We will export them again after renaming, to account for the changes in naming on the exported signals.



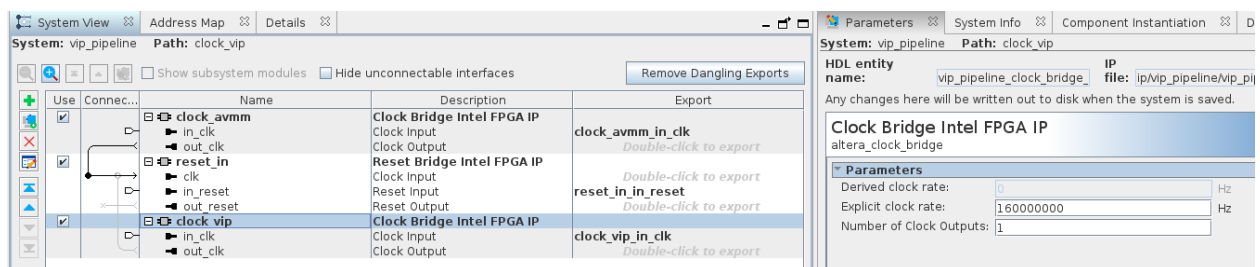
5. We will use **clock\_avmm** clock signal to connect our management clock also used to clock the Nios Processor. This clock has a freq=100MHz, so let's get this into the clock parameters in the *Explicit clock rate* field.



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**

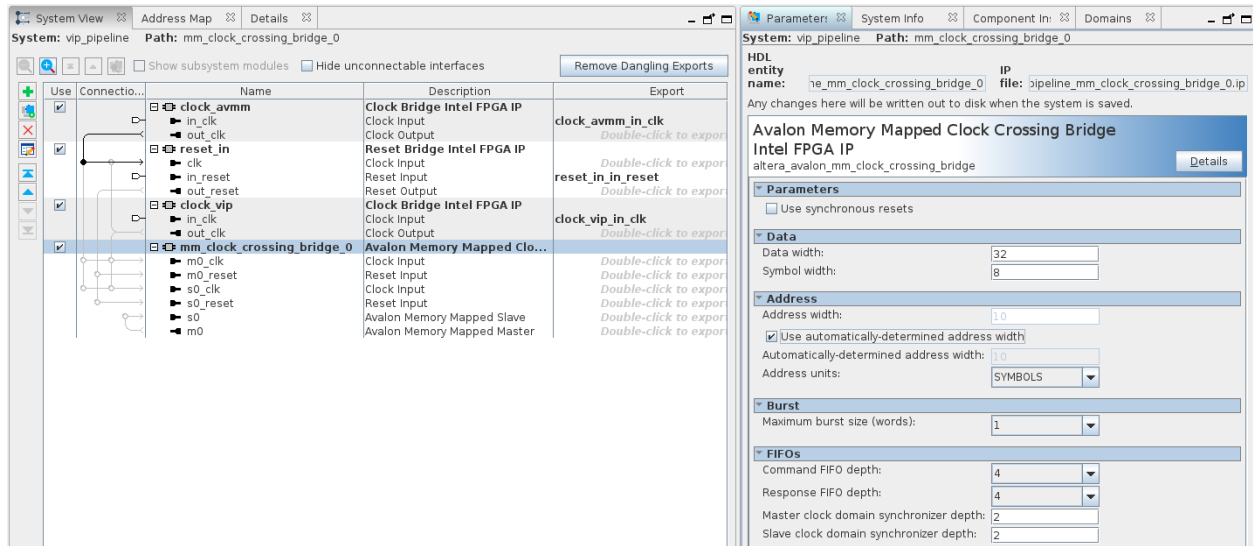


Rename it as `clock_vip` and in *Explicit clock rate* set 160000000. Make double click on the `in_clk` row in the **Export** column to get the input signal exposed to allow external clock connection into the system.



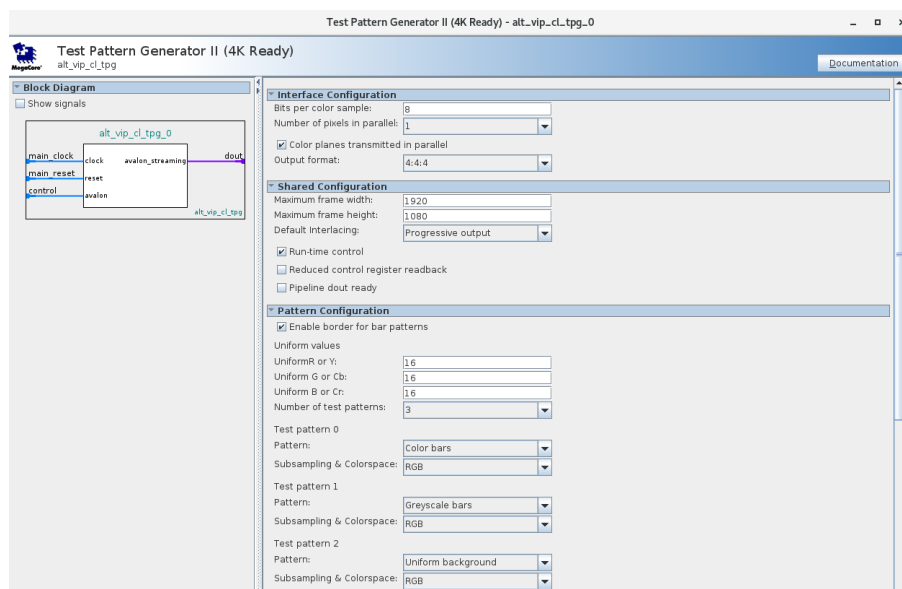
**NOTE:** In our system, we will work with 1080p60 resolution, which equals a clock rate of 148.5MHz. In order to make the VIP blocks function properly we need a signal equal or higher frequency of the pixel clock used. As we already have in our system a 160MHz, used for `dp_vid_rx`, let's use this one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

- 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



**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.

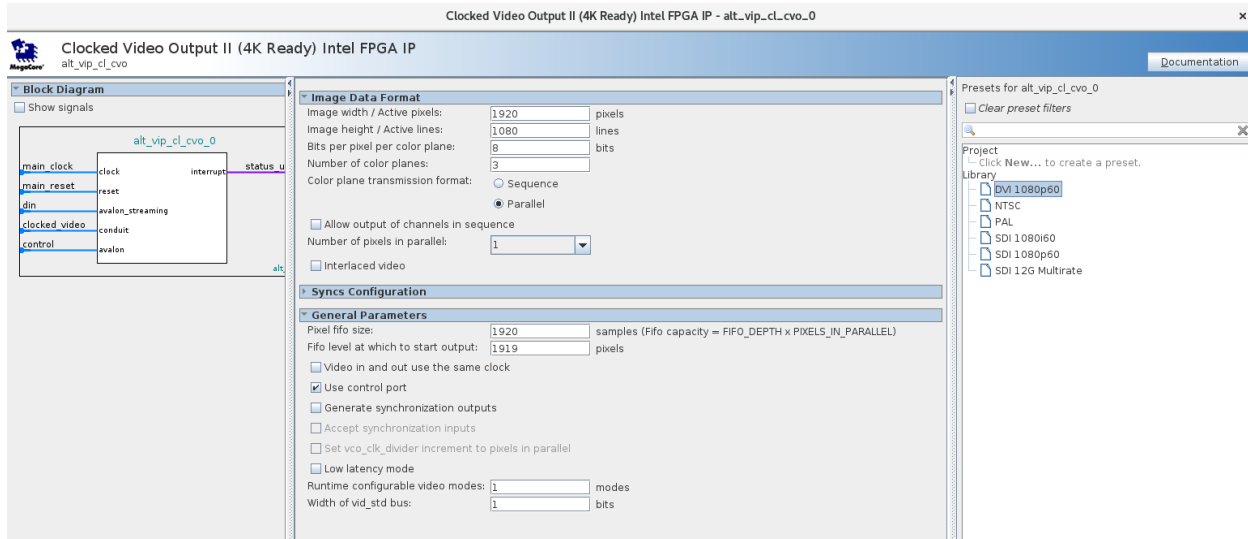
8. Now, let's add our VIP blocks. Go to **IP Catalog->Library->DSP->Video and Image Processing->Test Pattern Generator II**, select it and click **Add**. The parameterization GUI appears, select:
  - Enable Run-time Control
  - Number of test patterns: 3
    - Pattern 0: Color bars
    - Pattern 1: Grayscale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



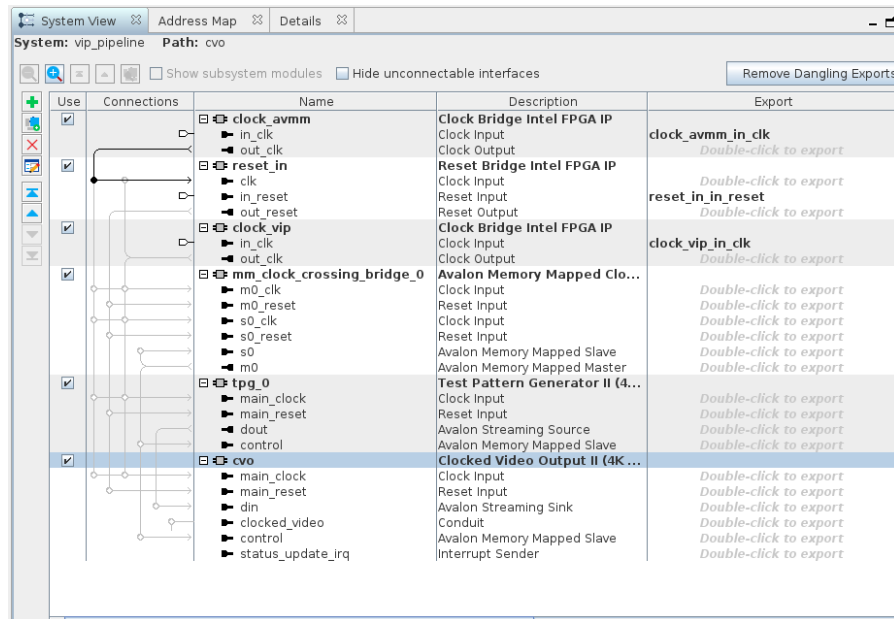
9. Rename it to **tpg\_0**

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:

- Under Presets (on the rightmost side), select **DVI 1080p60** and click **Apply**
- Then enable General Parameters->Use control port
- Leave all the other settings as default



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.

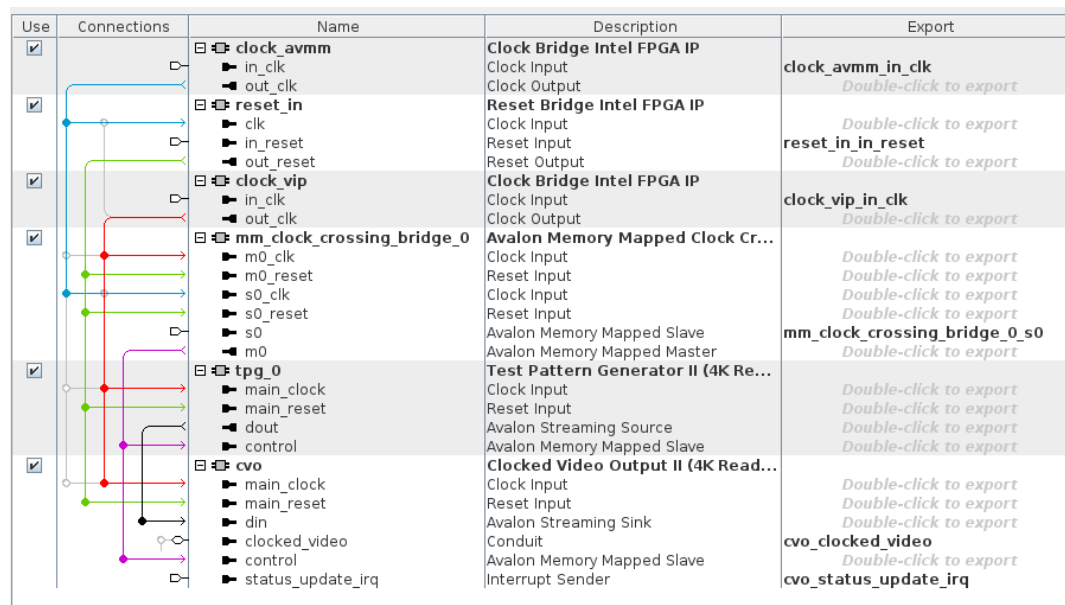


12. Exporting interfaces

Let's export the following conduits and signals

- mm\_clock\_crossing\_bridge\_0: s0
- cvo: clocked\_video
- 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

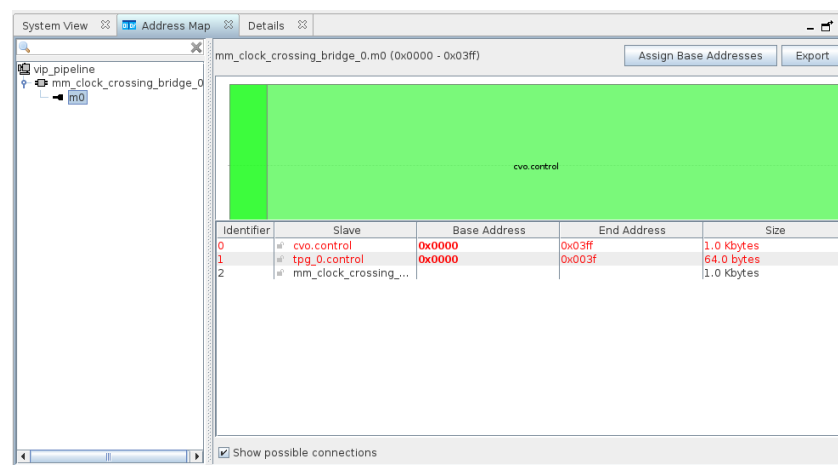


## 14. Solving Memory Mapped address conflicts

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

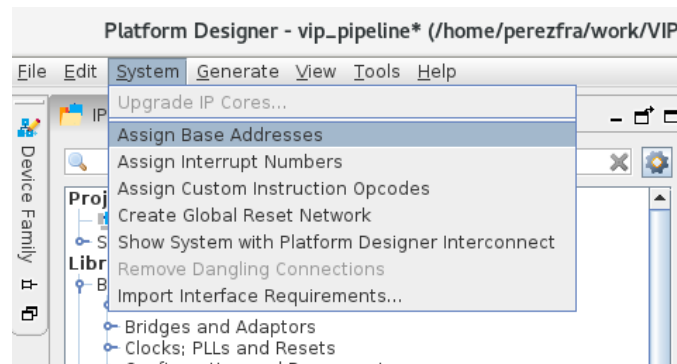
Type	Path	Message
1	System Connectivity Error	
	Platform Designer Tip	Please Sync All System Infos before attempting to resolve the error
	vip_pipeline.mm_clock_crossing_bridge_0.m0	cvo.control (0x0..0x3ff) overlaps tpg_0.control (0x0..0x3f)

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.

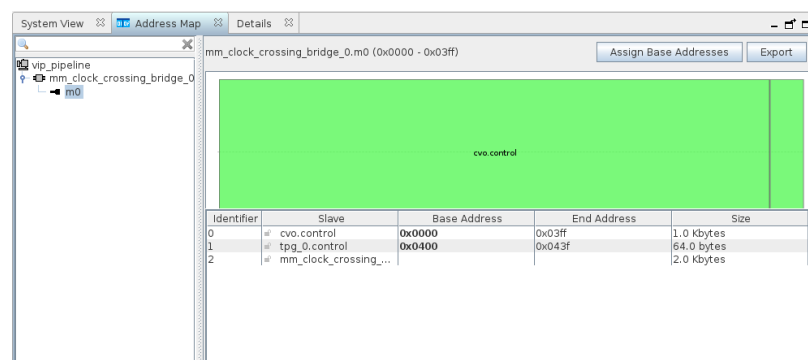




We can solve this issue by selecting System->Assign Base Addresses in the main toolbar



After execution, each slave will have its exclusive address space

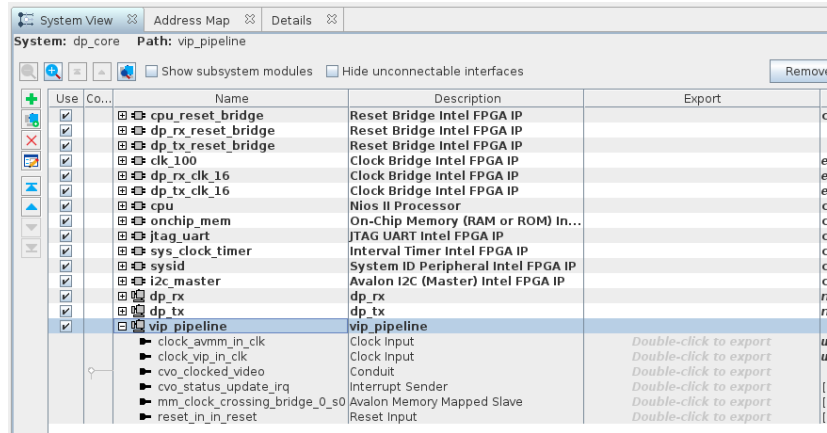


15. With this, we have finished the elaboration of our VIP pipeline. Let's now integrate it with the rest of the system

## 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`

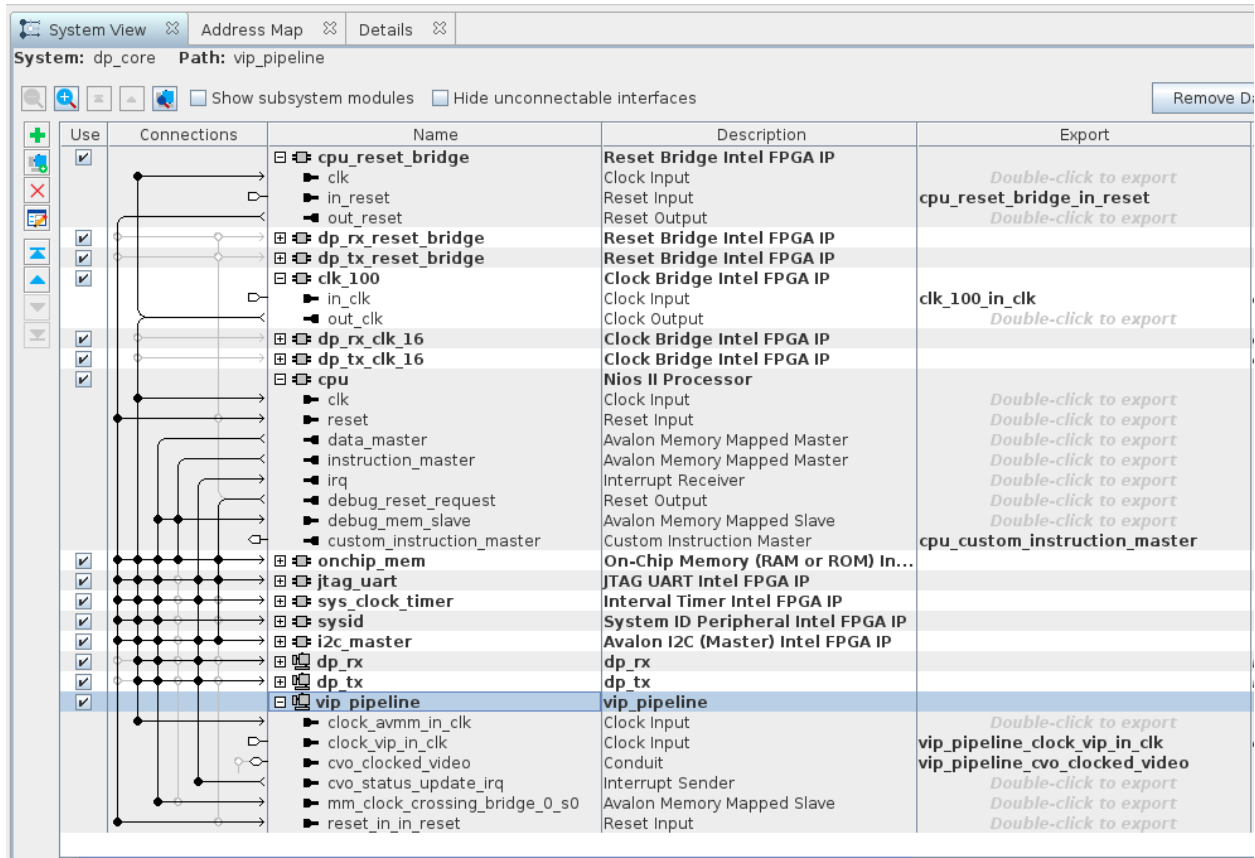
2. Under **IP Catalog->Project**, expand **System** group and you will find the different subsystems that are in the search path and available to use (some of them already used indeed). Double-click on `vip_pipeline` to add it. Rename to **vip\_pipeline**



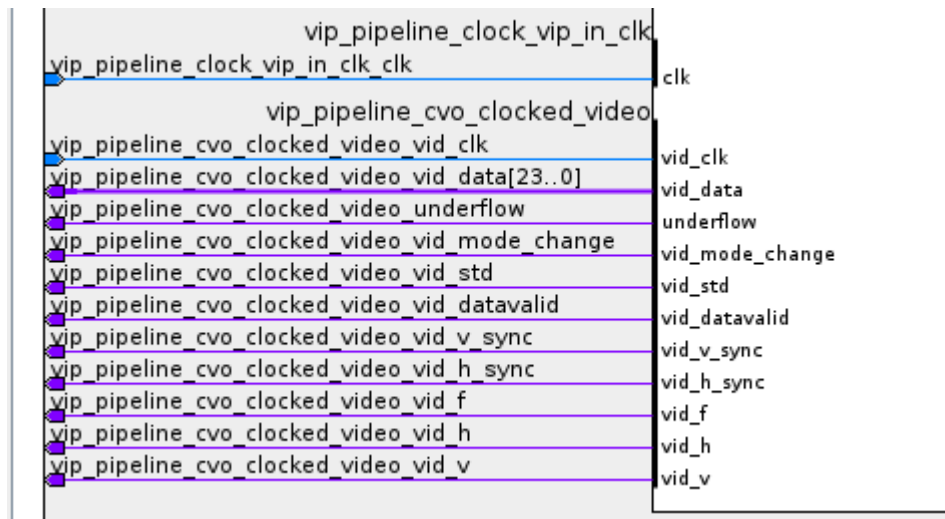
3. In order to connect our CVO with the DisplayPort TX interface, we would need to export the video output conduit and the corresponding clock. Then we do the remaining connection with the rest of components.

- o vip\_pipeline: clock\_avmm\_in\_clk -> clk\_100: out\_clk
- o vip\_pipeline: cvo\_status\_update\_irq -> cpu: irq
- o vip\_pipeline: mm\_clock\_crossing\_bridge\_0\_s0 -> cpu: data\_master
- o vip\_pipeline: reset\_in\_in\_reset -> cpu\_reset\_bridge: out\_reset

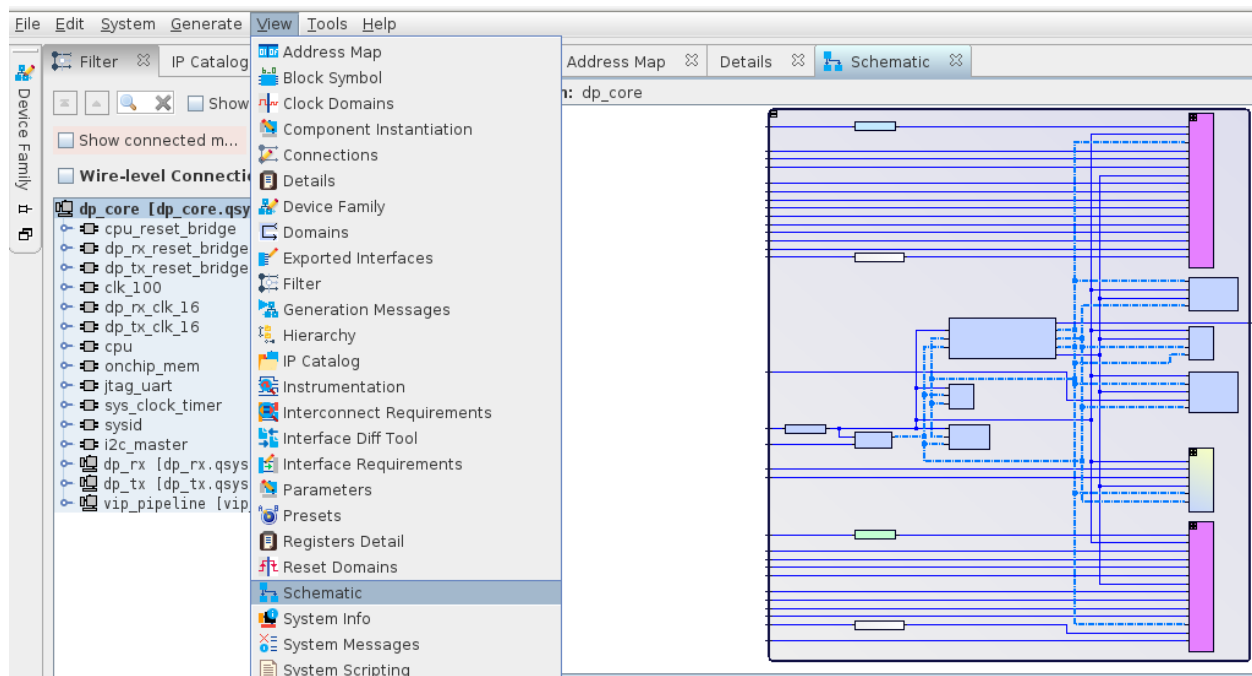
We should get connections like in the picture below



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.



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.

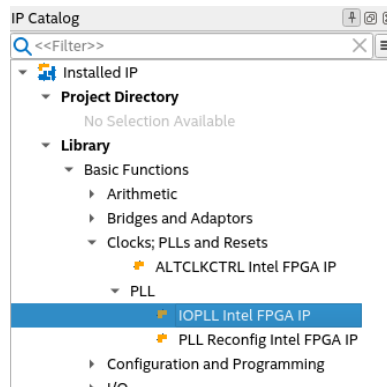
```
304 .dp_tx_dp_source_tx_pll_locked      (dp_txpll_locked),
305 // -----
306 // VIP Pipeline Sub-System
307 // -----
308 .vip_pipeline_clock_vip_in_clk_clk (connected_to_vip_pipeline_clock_vip_in_clk_clk_),
309 .vip_pipeline_cvo_clocked_video_vid_clk (connected_to_vip_pipeline_cvo_clocked_video_vid_clk_),
310 .vip_pipeline_cvo_clocked_video_vid_data (connected_to_vip_pipeline_cvo_clocked_video_vid_data_),
311 .vip_pipeline_cvo_clocked_video_underflow (connected_to_vip_pipeline_cvo_clocked_video_underflow_),
312 .vip_pipeline_cvo_clocked_video_vid_mode_change (connected_to_vip_pipeline_cvo_clocked_video_vid_mode_change_),
313 .vip_pipeline_cvo_clocked_video_vid_std (connected_to_vip_pipeline_cvo_clocked_video_vid_std_),
314 .vip_pipeline_cvo_clocked_video_vid_datavalid (connected_to_vip_pipeline_cvo_clocked_video_vid_datavalid_),
315 .vip_pipeline_cvo_clocked_video_vid_v_sync (connected_to_vip_pipeline_cvo_clocked_video_vid_v_sync_),
316 .vip_pipeline_cvo_clocked_video_vid_h_sync (connected_to_vip_pipeline_cvo_clocked_video_vid_h_sync_),
317 .vip_pipeline_cvo_clocked_video_vid_f (connected_to_vip_pipeline_cvo_clocked_video_vid_f_),
318 .vip_pipeline_cvo_clocked_video_vid_h (connected_to_vip_pipeline_cvo_clocked_video_vid_h_),
319 .vip_pipeline_cvo_clocked_video_vid_v (connected_to_vip_pipeline_cvo_clocked_video_vid_v_)
320 );
321
```

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.

```
422
423 .reset_out      (),
424 .rec_clk        (), //tx_vid_clk
425 .vidout         (), //tx_vid_data
426 .hsync          (), //tx_vid_hsync
427 .vsync          (), //tx_vid_vsync
428 .de             ()  //tx_vid_de
429 );
```

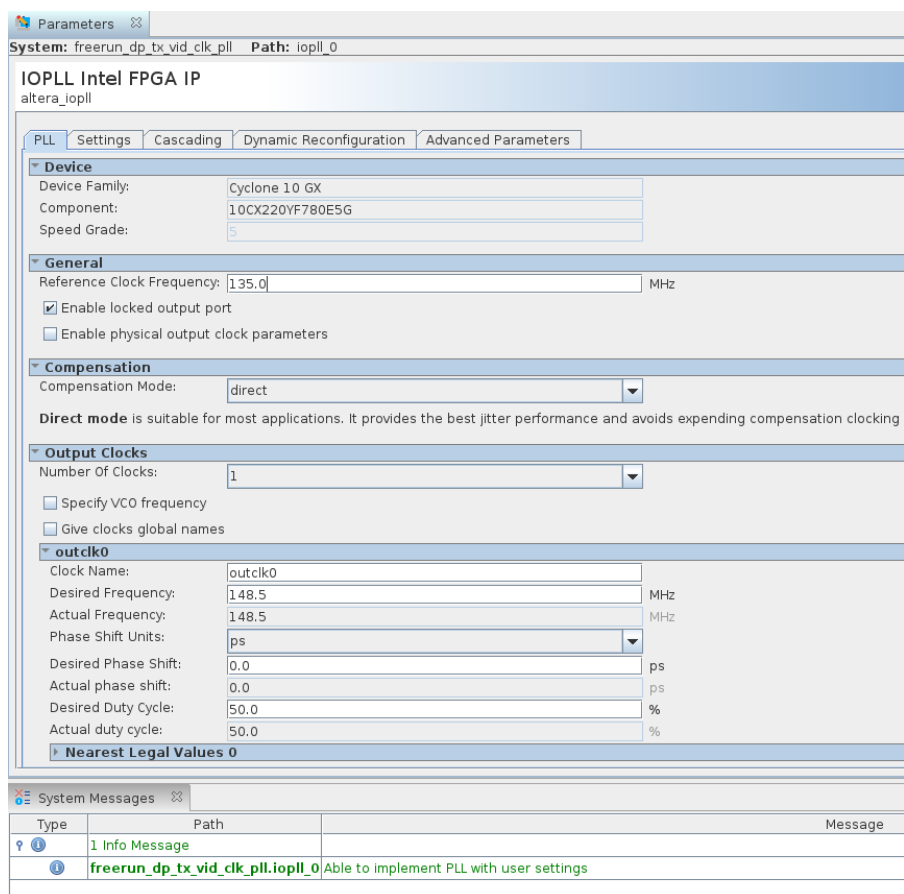
4. We need to generate our **pixel\_clock=148.5MHz** signal. For that, we will use an **IOPLL** with an already available 135MHz clock as input reference.

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



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

- Open the file <project\_dir>/rtl/freerun\_dp\_tx\_vid\_clk\_pll/freerun\_dp\_tx\_vid\_clk\_pll\_inst.v and copy the content.
- 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.

```

203 // -----
204 // FREERUN_DP_TX_VID_CLK_PLL
205 // Used to generate a free run 148.5MHz tx_video_clk in the absence on input signal connected
206 // we use as a reference => fmc_gbtclk_m2c_p (135MHz) coming from a OSC in the Bitec DP Rev11 daughter card
207 // -----
208
209 wire dp_tx_vid_clk;
210
211 freerun_dp_tx_vid_clk_pll freerun_dp_tx_vid_clk_pll_u0 (
212     .rst      (cpu_reset),      // input, width = 1, reset.reset
213     .refclk   (fmc_gbtclk_m2c_p), // input, width = 1, refclk.clk
214     .locked   (),              // output, width = 1, locked.export
215     .outclk_0 (dp_tx_vid_clk) // output, width = 1, outclk0.clk 148.5MHz for freerun output video
216 );

```

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

```

322 // -----
323 // VIP Pipeline Sub-System
324 // -----
325 .vip_pipeline_clock_vip_in_clk_clk      (dp_rx_vid_clk), // 160MHz clock user for DP_RX
326 .vip_pipeline_cvo_clocked_video_vid_clk (dp_tx_vid_clk), // 148.5MHz clock for video output
327 .vip_pipeline_cvo_clocked_video_vid_data (tx_vid_data), // 24 bit of video_data (RGB)
328 .vip_pipeline_cvo_clocked_video_underflow (),
329 .vip_pipeline_cvo_clocked_video_vid_mode_change (),
330 .vip_pipeline_cvo_clocked_video_vid_std (),
331 .vip_pipeline_cvo_clocked_video_vid_datavalid (tx_vid_de), // datavalid
332 .vip_pipeline_cvo_clocked_video_vid_v_sync (tx_vid_vsync), // V sync
333 .vip_pipeline_cvo_clocked_video_vid_h_sync (tx_vid_hsync), // H sync
334 .vip_pipeline_cvo_clocked_video_vid_f (),
335 .vip_pipeline_cvo_clocked_video_vid_h (),
336 .vip_pipeline_cvo_clocked_video_vid_v (),
337 );

```

- 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)

```

287 // -----
288 // DisplayPort Source Sub-System
289 // -----
290 .dp_tx_reset_bridge_in_reset_n      (video_pll_locked),
291 .dp_tx_clk_16_in_clk                 (clk_16),
292 .dp_tx_dp_source_clk_cal             (dp_tx_clk_cal),
293 // Hot Plug Detect Interface
294 .dp_tx_dp_source_tx_hpd              (~fmc_la_rx_n_9),
295 // DisplayPort Auxiliary Interface
296 .dp_tx_dp_source_tx_aux_in           (fmc_la_tx_p_12),
297 .dp_tx_dp_source_tx_aux_out          (fmc_la_rx_p_10),
298 .dp_tx_dp_source_tx_aux_oe           (fmc_la_rx_n_10),
299 // TX Video Signal Interface
300 .dp_tx_dp_source_tx_vid_clk          (dp_tx_vid_clk), // from the freerun_dp_tx_vid_clk_pll
301 .dp_tx_dp_source_tx_vid_data         (tx_vid_data), // from the VIP Pipeline Sub-System
302 .dp_tx_dp_source_tx_vid_v_sync       ({TX_PIXELS_PER_CLOCK{tx_vid_vsync}}), // from the VIP Pipeline Sub-System
303 .dp_tx_dp_source_tx_vid_h_sync       ({TX_PIXELS_PER_CLOCK{tx_vid_hsync}}), // from the VIP Pipeline Sub-System
304 .dp_tx_dp_source_tx_vid_de           ({TX_PIXELS_PER_CLOCK{tx_vid_de}}), // from the VIP Pipeline Sub-System

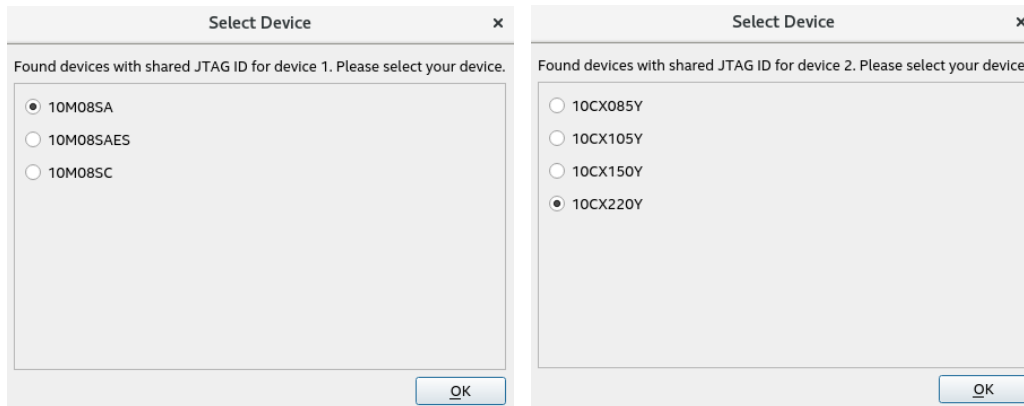
```

- 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

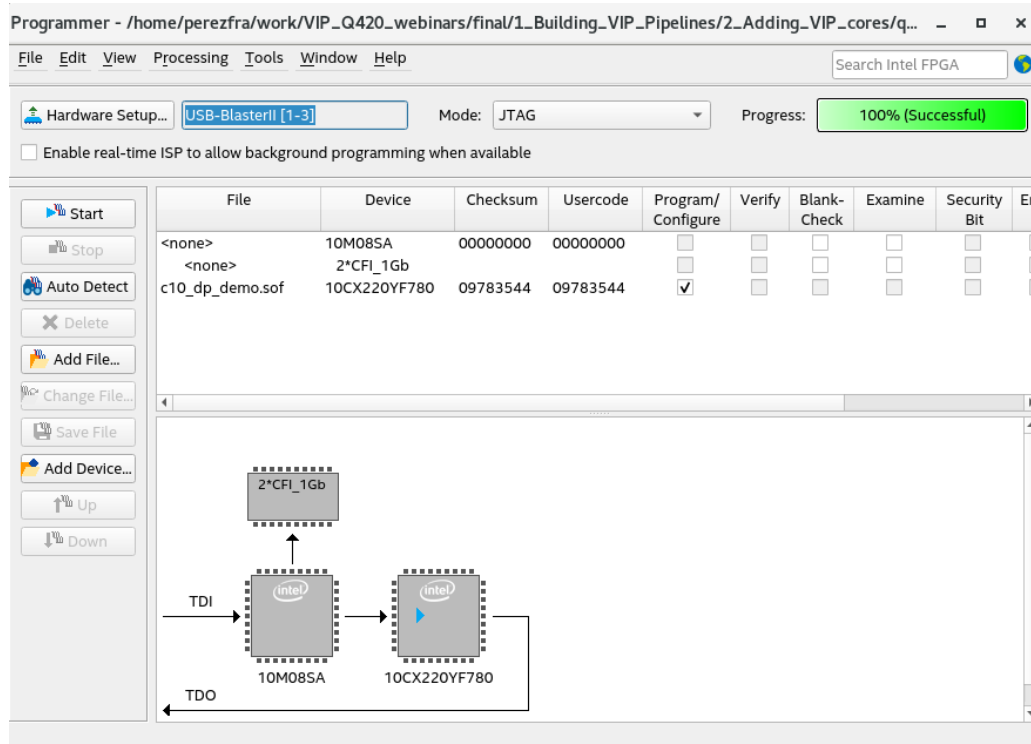
Open the **Quartus Programmer**, select your USB-Blaster cable in the Hardware Setup and click on **Auto Detect** to retrieve the JTAG chain on the Cyclone10 GX Devkit.

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



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**.





## 3. Building the software application

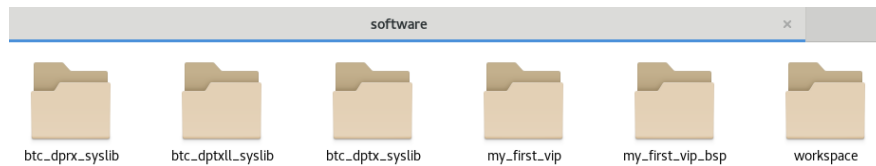
### 3.1. Setting up the Eclipse for Nios project

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:

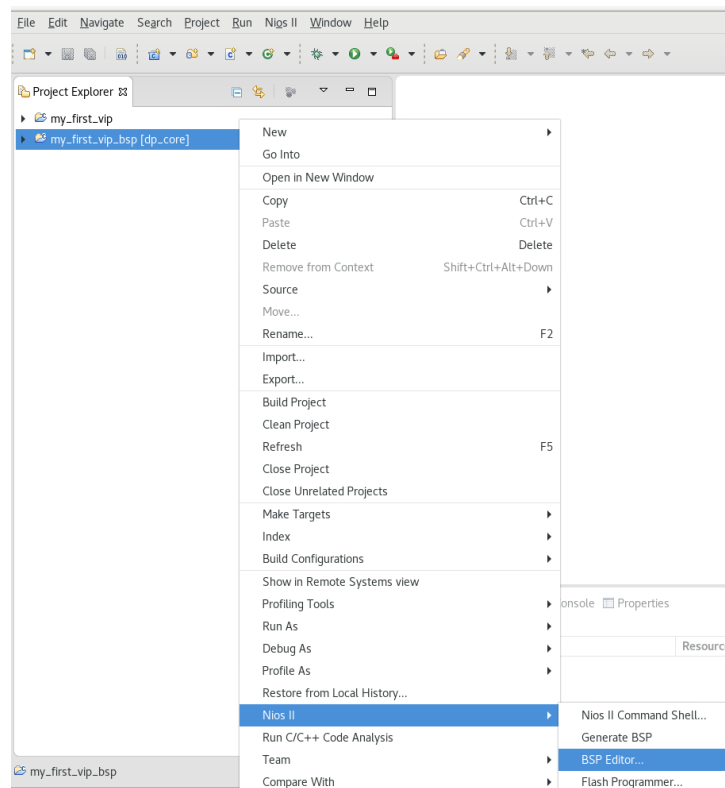
- workspace: `<project_dir>/software/workspace`
- application and bsp: `<project_dir>/software/my_first_vip`

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

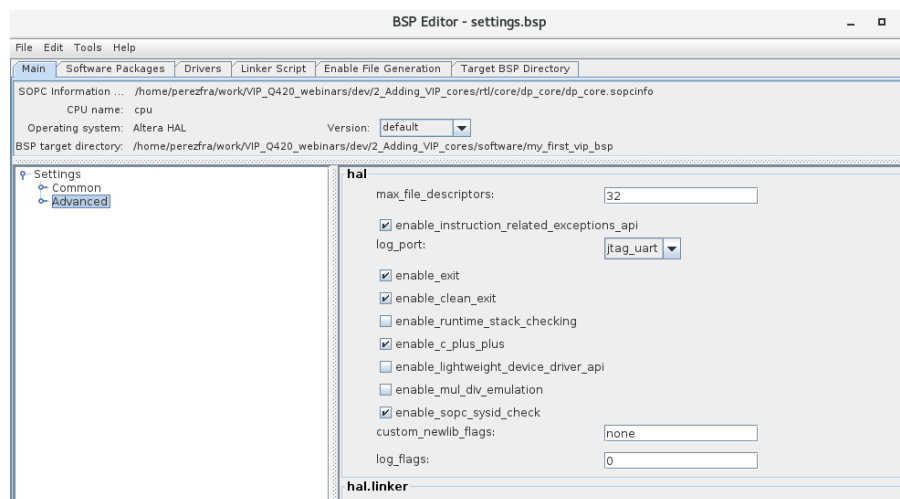
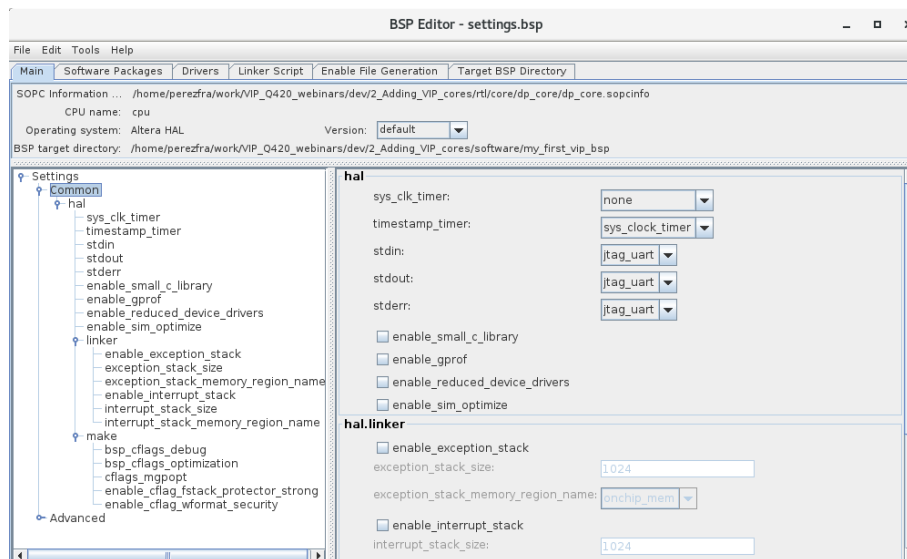
Then, you need to copy all the DisplayPort libraries generated in the first lab to `<project_dir>/software`. You will end up with a setup like the following



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



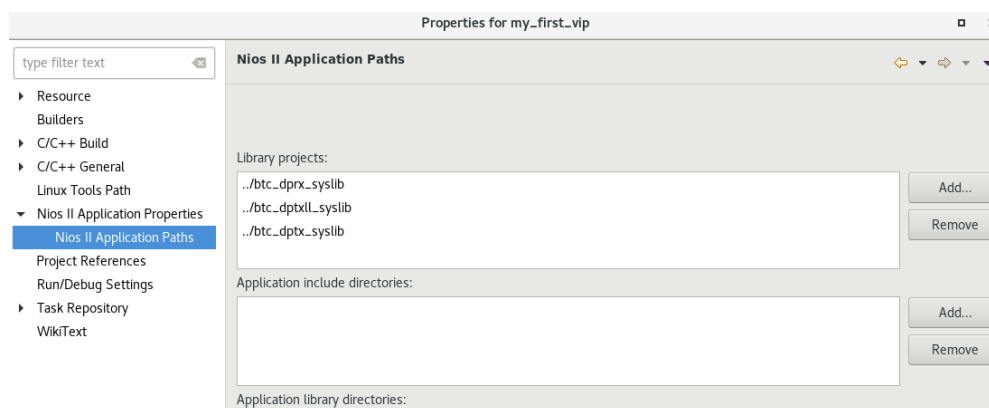
Please note that in the **Drivers** tab, the bsp generation has already included supporting code for the VIP components we are using: CVO and TGP.

BSP Editor - settings.bsp					
File Edit Tools Help					
Main	Software Packages	Drivers	Linker Script	Enable File Generation	Target BSP Directory
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	<input checked="" type="checkbox"/>
dp_rx_dp_sink	altera_dp	19.4.0	none	none	<input checked="" type="checkbox"/>
dp_rx_pio_0	altera_avalon_pio	19.1.0	altera_avalon_pio_driver	default	<input checked="" type="checkbox"/>
dp_rx_pio_1	altera_avalon_pio	19.1.0	altera_avalon_pio_driver	default	<input checked="" type="checkbox"/>
dp_tx_dp_source	altera_dp	19.4.0	none	none	<input checked="" type="checkbox"/>
dp_tx_xdash	altera_avalon_pio	19.1.0	altera_avalon_pio_driver	default	<input checked="" type="checkbox"/>
i2c_master	altera_avalon_i2c	19.2.0	altera_avalon_i2c_driver	default	<input checked="" type="checkbox"/>
jtag_uart	altera_avalon_jtag_uart	19.1.0	altera_avalon_jtag_uart_driver	default	<input checked="" type="checkbox"/>
onchip_mem	altera_avalon_onchip_memory2	19.2.0	none	none	<input checked="" type="checkbox"/>
sys_clock_timer	altera_avalon_timer	19.1.0	altera_avalon_timer_driver	default	<input checked="" type="checkbox"/>
sysid	altera_avalon_sysid_qsys	19.1.0	altera_avalon_sysid_qsys_driver	default	<input checked="" type="checkbox"/>
vip_pipeline_cvo	alt_vip_cl_cvo	20.3.0	alt_vip_cl_cvo_driver	default	<input checked="" type="checkbox"/>
vip_pipeline_tpq_0	alt_vip_cl_tpq	20.3.0	alt_vip_cl_tpq_driver	default	<input checked="" type="checkbox"/>

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

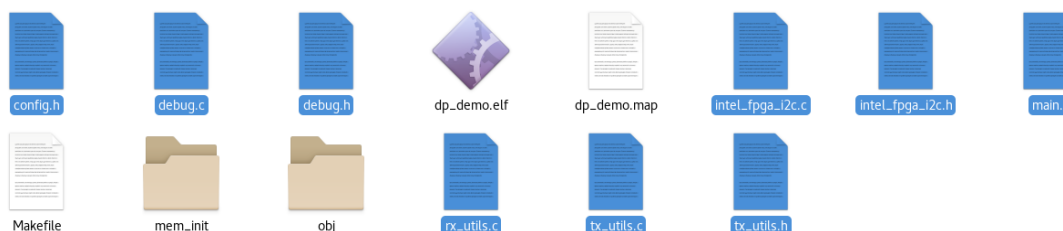


Then click **Apply** and **OK**

### 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

.....

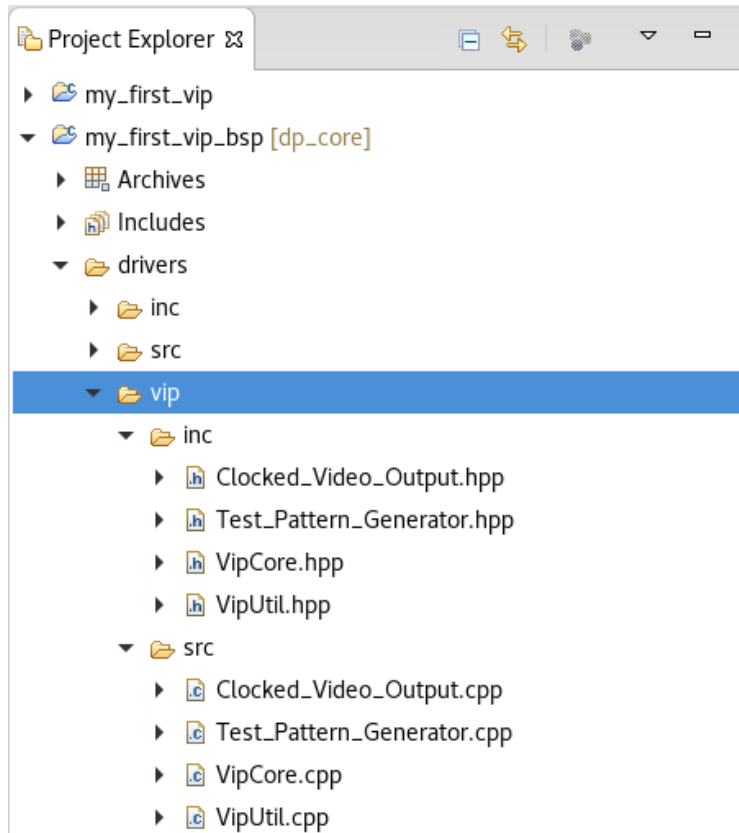
```
#ifdef __cplusplus
}
#endif /* __cplusplus */
|
#endif /* TX_UTILS_H */
```

**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
  - o 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*
  - o 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*)



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.

#### o **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
// -----

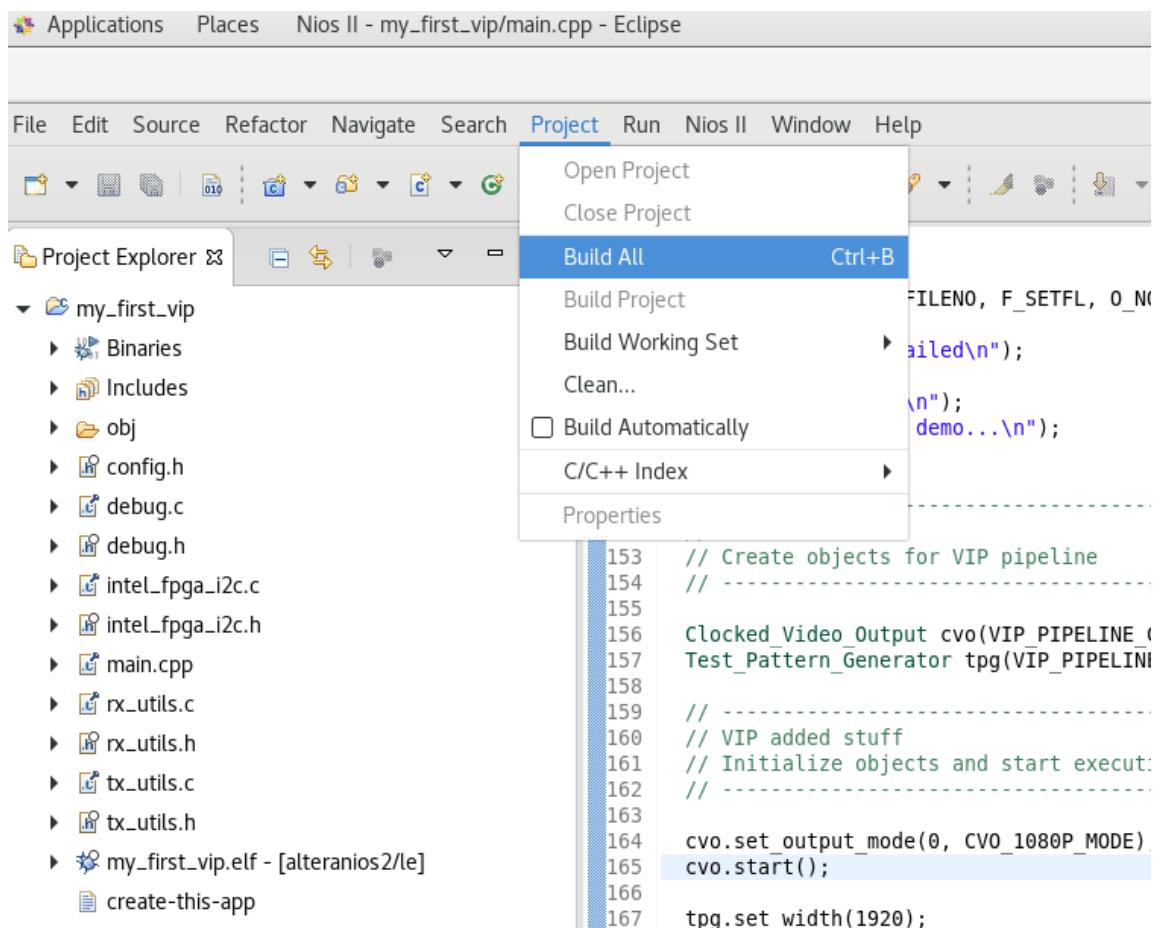
cvo.set_output_mode(0, CVO_1080P_MODE);
cvo.start();

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

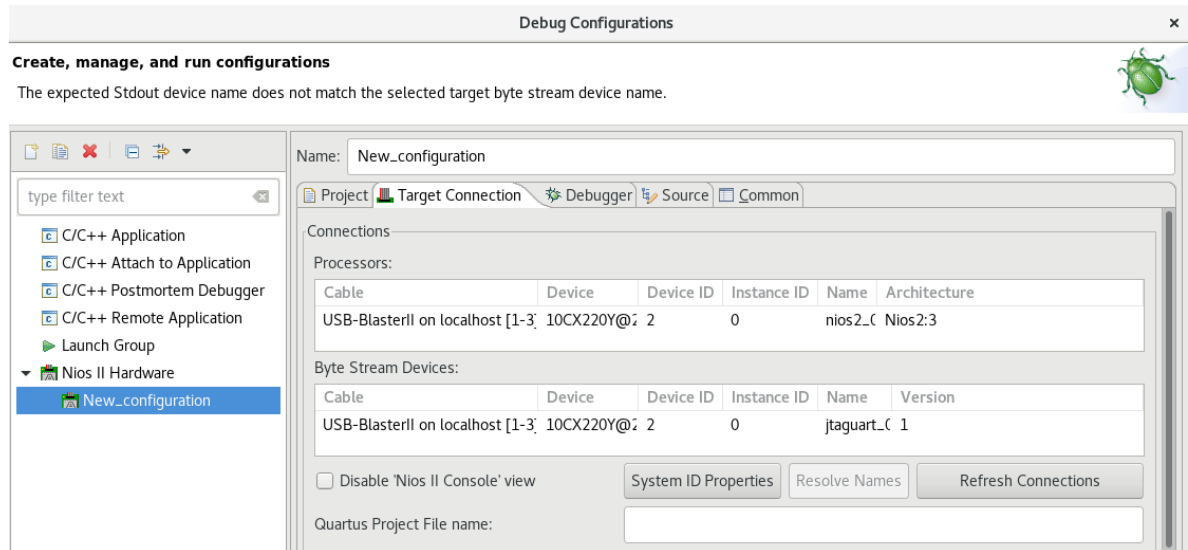
After doing that modifications, we can generate our executable file. In Eclipse IDE, go to **Project->Build All** to compile the *bsp* and generate *my\_first\_vip.elf* file.



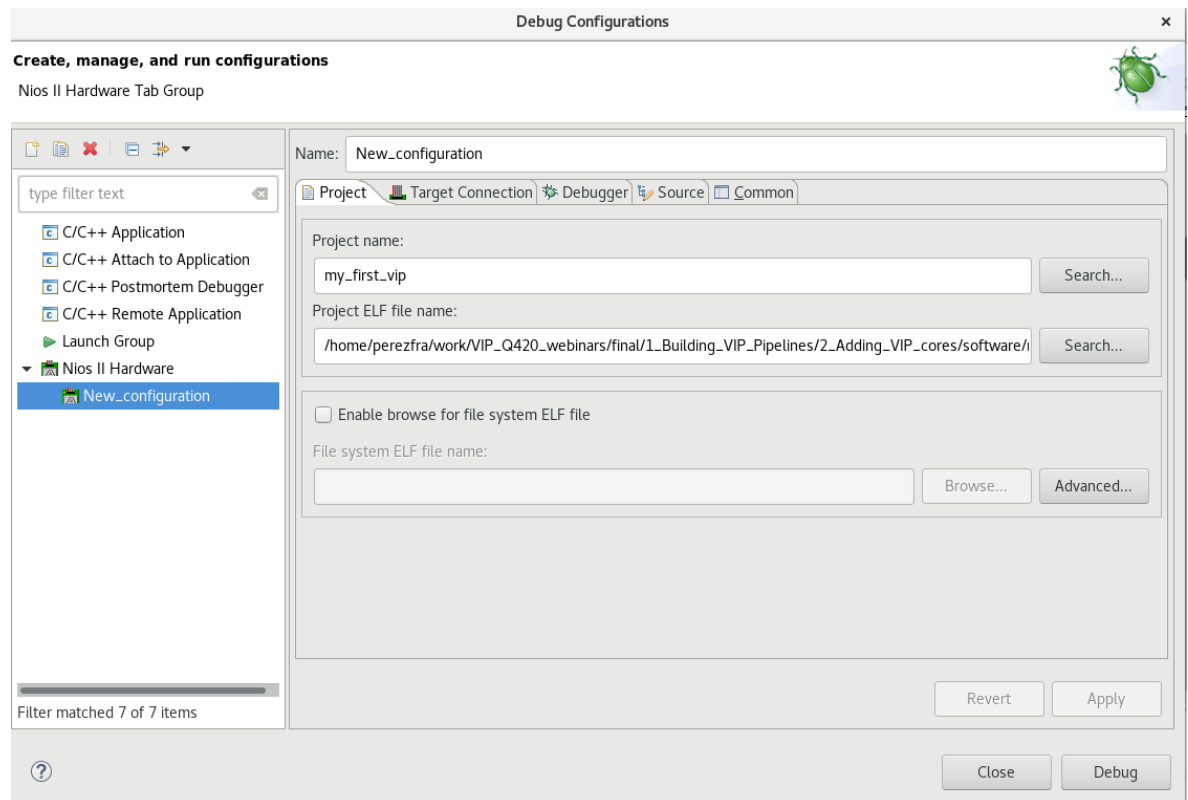


To launch our application, we can go to **Run->Debug Configurations...** in the main toolbar to open *Debug Configuration* dialog.

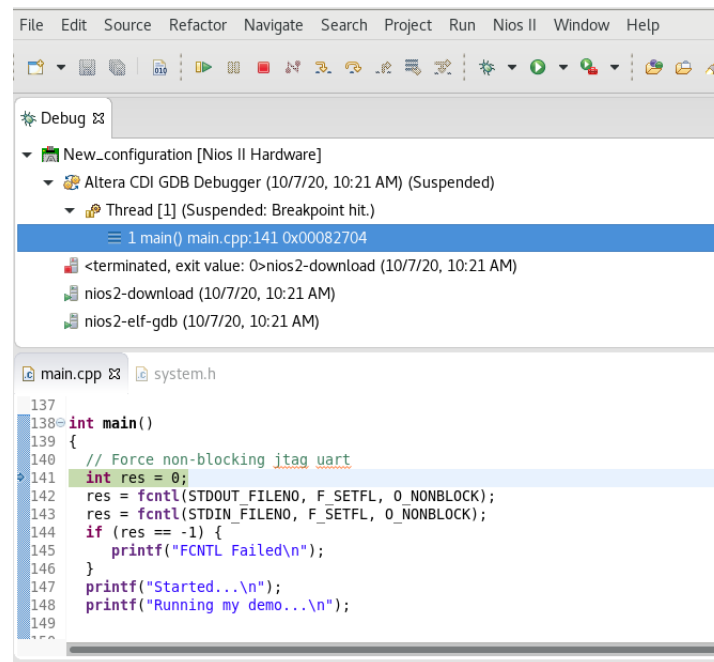
We double-click on **Nios II Hardware** option to create a *New\_configuration*. In **Target Connection** tab, click on **Refresh Connections** to select the right *USB-BlasterII* adapter



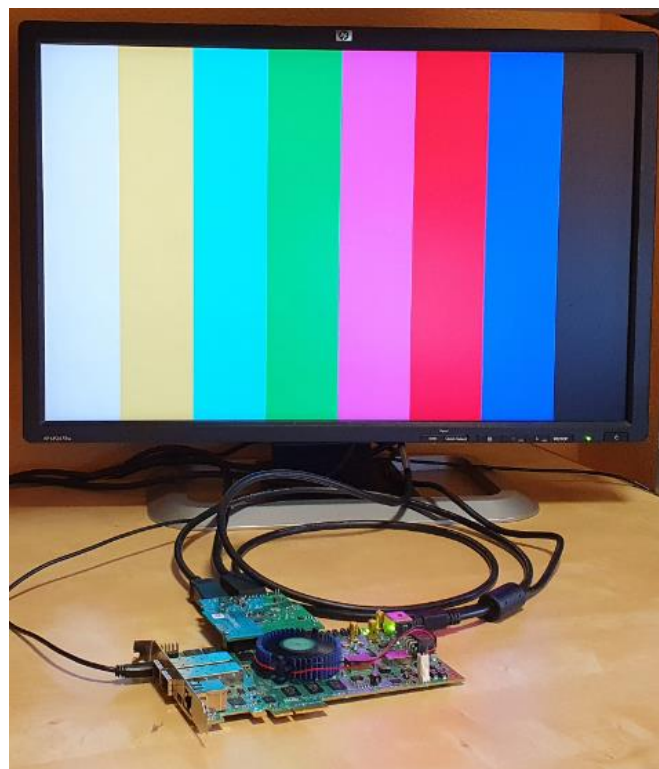
Back to the **Project** tab, make sure the right *Project Name* and *ELF File Name* are selected and just click on **Apply** and **Debug** to launch the session



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.



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