

VIDEO PROCESSING WITH INTEL FPGAS

WEBINAR SERIES – Q4'2020

Session 1.3 – Complete VIP pipeline

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

Contents

| | |
|--|-----------|
| 1. Introduction | 3 |
| 1.1. Introduction | 3 |
| 1.2. Requirements | 3 |
| 1.3. References..... | 3 |
| 1.4. Implementation diagram..... | 4 |
| 2. Generating the hardware pipeline..... | 6 |
| 2.1. Setting up the Quartus project..... | 6 |
| 2.2. Examining the project | 6 |
| 2.3. Understanding the external connections | 7 |
| 2.4. Build the video pipeline in Platform Designer | 9 |
| 2.5. Integrating the VIP pipeline..... | 23 |
| 2.6. Integrate the complete module at top level..... | 29 |
| 2.7. Generate the programming file..... | 37 |
| 2.8. Configuring the FPGA device | 37 |
| 3. Building the software application | 39 |
| 3.1. Setting up the Eclipse for Nios project | 39 |
| 3.2. Importing the code..... | 43 |
| 3.3. Getting familiar with VIP C++ API | 44 |
| 3.4. Controlling VIP cores from the application..... | 49 |
| 3.5. Building and launching program execution | 52 |
| 4. Summary | 56 |

1. Introduction

1.1. Introduction

In this lab manual we are expanding the design generated in the previous session: “**Session 1.2 – Adding VIP cores**” by adding additional modules to our pipeline. This will enable us to capture the incoming video (through the DisplayPort input) and do some processing like cropping and/or scaling. The processed live video will be then mixed with a test pattern background to compose the final output result.

This session will cover how to build the hardware implementation of the video pipeline and how to develop a software application for an embedded Nios II Processor to control the modules in run time.

You can follow all the steps in this guide, taking as a base project the one generated in the previous guide “**Session1_2_Addign_VIP_cores_v1.pdf**”.

NOTE: Alternatively, you can find all the final files in the archived project “**1_3_Complete_VIP_pipeline.tar.gz**” located in the github repository.

https://github.com/perezfra/VIP_webinars_Intel_FPGA

- **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
- 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:

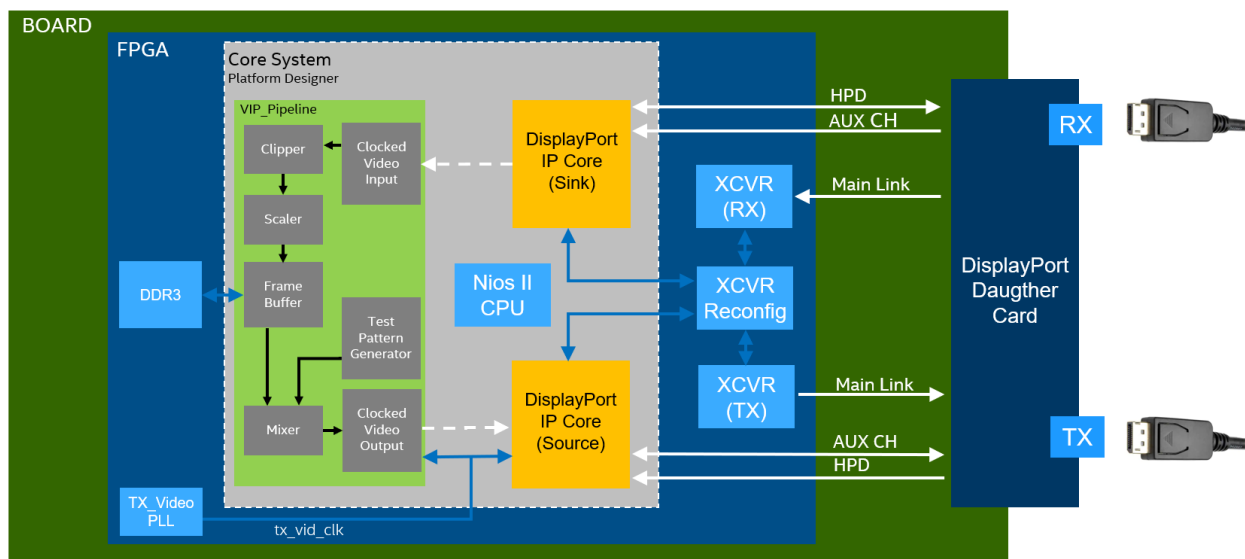
- Intel FPGA DisplayPort IP User Guide
<https://www.intel.com/content/www/us/en/programmable/products/intellectual-property/ip/interface-protocols/m-alt-displayport-megacore.html>

- 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
- Nios II 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 connected to a **Clocked_Video_Input** module in the **VIP_Pipeline** subsystem. This video flow is then connected to downstream modules, to process it (cropping/scaling), until get it finally mixed with a test pattern background before sent to the **Clocked_Video_Output** component connected to the DisplayPort TX interface.



In this design we are modifying the `vip_pipeline.qsys` subsystem by adding additional modules.

- **VIP_Pipeline subsystem** – We are adding the following VIP cores
 - **Clocked Video Input:** module used to capture pixel video data from the DisplayPort Sink core and convert it to a compatible Avalon ST-Video stream, to be consumed by downstream modules.
 - **Clipper:** Module we can use to crop and select a rectangular portion of the incoming video frames. Can be controlled in run time.
 - **Scaler:** Allows to change the resolution of the image. This is a high-quality polyphase scaler. It has the capabilities to load the optimal coefficient sets in runtime, according to input/output resolutions configured.
 - **Frame Buffer:** Configured as Triple Buffer mode, allows to compensate the jitter generated by the Scaler and absorbs any drift between input/output video clocks. Can be used to perform a controlled frame rate conversion.
 - **Test Pattern Generator:** we use this module to generate static video patterns we can use to drive the output video interface in the absence of live input video. We use the generated frames as background layer.
 - **Mixer:** configured as 2 input layers, it mixes the content from the TPG and the live video. Allows the freely placement of the video layer, on top of the background, as long as the boundaries are respected.
 - **Clocked Video Output:** it converts back the Avalon ST-Video streams processed by VIP cores to parallel pixel 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.
- **DDR3 EMIF** – In order to buffer the video data, we need to have access to some memory banks used as frame buffers. In the Cyclone10 GX devkit we have available a DDR3 interface with 32bit data width running at 933MHz. This provides us enough bandwidth to our purposes. We will implement a suitable DDR3 EMIF controller that will connect to our VIP Frame Buffer core within the VIP_pipeline.
- **TX_Video_PLL** – We are reusing the PLL added in the previous session 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 input clock (already used as reference for the transceivers blocks).

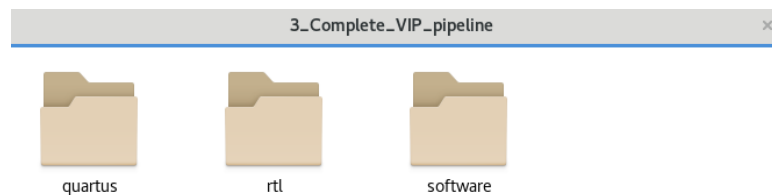
2. Generating the hardware pipeline

2.1. Setting up the Quartus project

We are using, as base project, the design generated in the previous step (“2_Adding_VIP_cores”). This project already contains the instantiations of DisplayPort sink and source, other necessary blocks like PLLs and Nios II CPU, as well as an initial VIP pipeline to generate an output video pattern.

Let’s create a new empty folder “3_Complete_VIP_pipeline” and copy the following directories from the mentioned previous project.

Copy the *quartus*, *rtl* and *software* folders from `<session1.2_adding_vip_cores_project_dir>/` to the new created folder, from now we will refer to it as `<project_dir>/`



Let’s open the project in Quartus Pro and examine it. Open `<project_dir>/quartus/c10_dp_demo.qpf`

NOTE: If you decide to use the files provided in **1_3_Complete_VIP_pipeline.tar.gz** package, just extract the files in the archived file, open the Quartus project located at `<extracted_folder>/quartus/c10_dp_demo.qpf` and you can jump directly to **Generating the Programming file**, as all the modifications have been done already.

2.2. Examining the project

With the project already loaded in Quartus, open `<project_dir>/rtl/core/dp_core.qsys` in Platform Designer. Right-click on the system view to Collapse all.

| Use | Co... | Name | Description | Export | Clock | Base | End | IRQ |
|-------------------------------------|-------|--------------------|------------------------------------|----------|-------------|-------------|--------------|-----|
| <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 | clk_100_... | 0x0010_1000 | 0x0010_1fff | |
| <input checked="" type="checkbox"/> | | dp_tx | dp_tx | multiple | clk_100_... | 0x0010_0000 | 0x0010_0fff | |
| <input checked="" type="checkbox"/> | | vip_pipeline | vip_pipeline | multiple | clk_100_... | 0x0000_0000 | 0x0000_07fff | |

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 build a hierarchical design:

- **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
- **vip_pipeline**: This is the subsystem we created in the last session and we will be editing it to include additional VIP cores to complete our video processing application.

2.3. Understanding the external connections

We already discussed, in the previous design, which are the interfaces exported out of Platform Designer. We saw how to connect them externally, to close the video pipeline with the DisplayPort IP cores, to receive and transmit video from/to external devices.

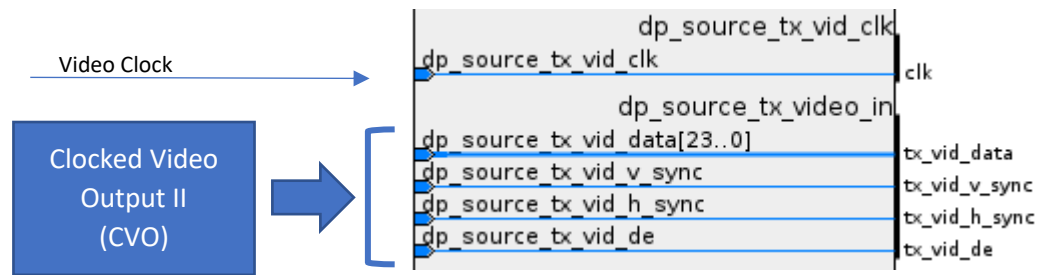
As a recap, let's discuss again here those interfaces related to Video Pixel data and sync signals, and how to connect them to the CVI and CVO modules in the VIP_pipeline

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

| 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 | | mult |
| 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 | [clk_1] |
| dp_source_tx_mgmt_interrupt | Interrupt Sender | Double-click to export | |
| 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_1] |
| reset_bridge_in_reset | Reset Input | Double-click to export | |
| xdash_external_connection | Conduit | dp_tx_xdash_external_connection | |

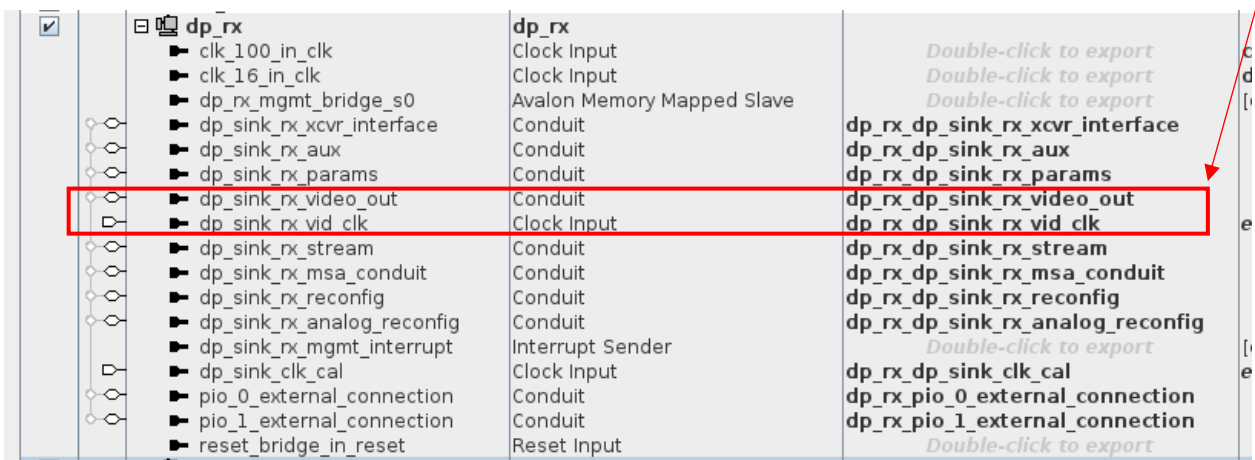
The target exported signals to connect our CVO are:

- **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



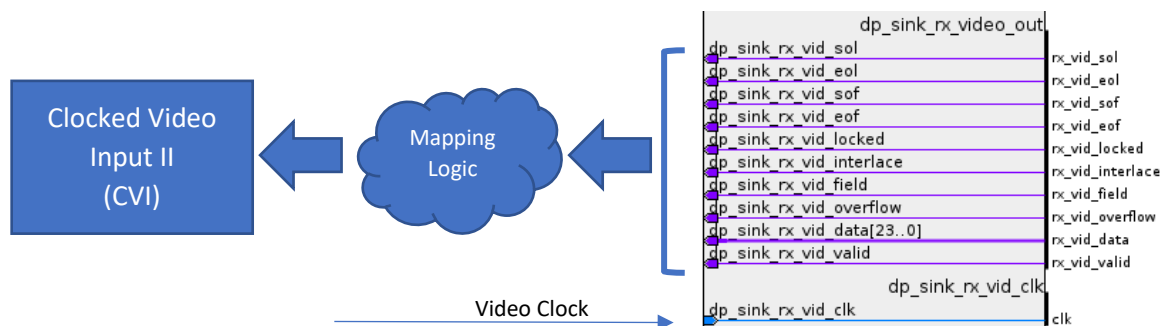
The `dp_source_tx_video_in` interface allows glue-less connection with Clock Video Output component.

- **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.



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 section where we are covering input video capturing

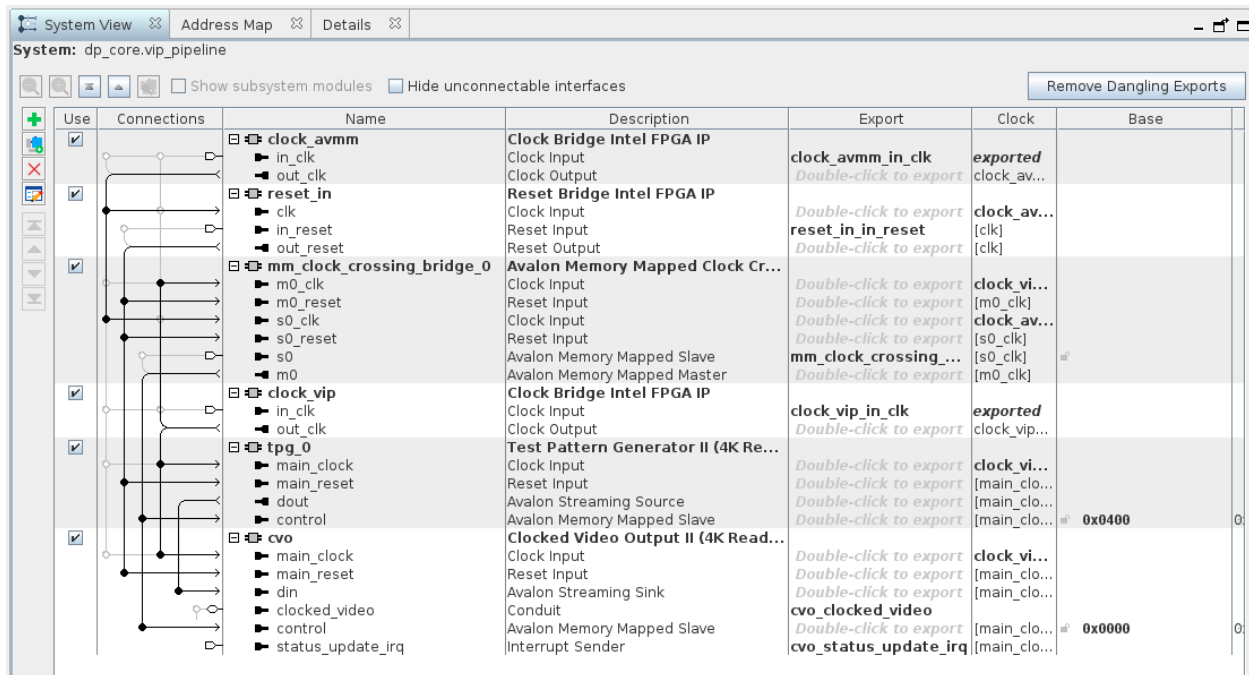


We will do all of these connections in the top level file `c10_dp_demo.v`

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 video processing chain to capture DisplayPort incoming video, process it and display on a DisplayPort external monitor.

1. Open Platform Designer
2. Go to **File->Open** in the main toolbar and select
`<project_dir>/rtl/core/vip_pipeline.qsys`



3. In the current subsystem, we are using a **Test Pattern Generator (tpg)** and **Clocked Video Output (cvo)** cores to generate a static pattern on the output display. Let's now add what is necessary to capture, process and display live incoming video.
4. To capture incoming video we need a **Clocked Video Input** core. Go to **IP Catalog->Library->DSP->Video and Image Processing->Clocked Video Input II** and click **Add**. The dialog box to parameterize the core will open
 - Enable the **Use control port** to export an Avalon-MM slave where we will connect our Nios CPU and keep the rest of parameters by default.
 - 8 bits per color
 - 3 color planes (RGB)
 - 1920x1080 as initial resolution

Note in the block diagram the `clocked_video` interface. We will export this interface to connect with the `dp_sink_rx_video_out` interface on the DisplayPort receiver IP core.

dout_0 interface is an Avalon Video-ST compliant bus dedicated to stream out the captured video to downstream VIP cores in the chain

Clocked Video Input II (4K Ready) Intel FPGA IP
alt_vip_cl_cvi

Block Diagram
Show signals

Avalon-ST-Video Image Data Format
 Bits per pixel per color plane: 8 bits
 Number of color planes: 3
 Color plane transmission format: ☒ Parallel
 Number of pixels in parallel: 1
 Field order: Field 0 first
☐ Enable matching data packet to control by clipping
☐ Enable matching data packet to control by padding
☐ Overflow handling

Clocked Video Parameters
 Sync signals: ☐ Embedded in video ☒ On separate wires
☐ Support 6G & 12G-SDI
☐ Allow color planes in sequence input
☐ Extract field signal
☐ Use vid_std bus
 Width of vid_std bus: 1 bits
☐ Extract ancillary packets
 Depth of the ancillary memory: 1 words
☒ Extract the total resolution
 Enable HDMI Duplicate Pixel Removal: No duplicate pixel removal

Avalon-ST-Video Initial Control Packet
 Interlaced or progressive: ☒ Progressive ☐ Interlaced
 Width: 1920 pixels
 Height - frame/field 0: 1080 pixels
 Height - field 1: 480 pixels

General Parameters
 Pixel FIFO size: 2048 pixels
☐ Video in and out use the same clock
☒ Use control port

Block Diagram Signals:
 control_address[4..0], control_read, control_readdata[31..0], control_write, control_writedata[31..0], control_byteenable[3..0], control_waitrequest, clocked_video_vid_clk, clocked_video_vid_data[23..0], clocked_video_vid_de, clocked_video_vid_datavalid, clocked_video_vid_locked, clocked_video_vid_f, clocked_video_vid_v_sync, clocked_video_vid_h_sync, clocked_video_vid_color_encoding[7..0], clocked_video_vid_bit_width[7..0], clocked_video_sof, clocked_video_sof_locked, clocked_video_refclk_div, clocked_video_clipping, clocked_video_padding, clocked_video_overflow, main_clock_clk, main_reset_reset, status_update_irq, status_update_irq_irq, data, valid, startofpacket, endofpacket, ready, dout_0_data[23..0], dout_0_valid, dout_0_startofpacket, dout_0_endofpacket, dout_0_ready.

Annotations:
 Connect to next VIP module internally
 Connect to DisplayPort Sink

5. Rename it as **cvi**

6. Next, we are adding a clipper to select and crop a Region of Interest of our incoming video. Go to **IP Catalog->Library->DSP->Video and Image Processing->Clipper II** and click **Add**.

Clipper II (4K Ready) Intel FPGA IP
alt_vip_cl_clp

Block Diagram
Show signals

Video Data Format
 Maximum input frame width: 1920
 Maximum input frame height: 1080
 Bits per color sample: 8
 Number of color planes: 3
 Number of pixels in parallel: 1
☒ Color planes transmitted in parallel

Clipping Options
☒ Run-time control
 Clipping method: RECTANGLE
 Left Offset: 0
 Top Offset: 0
 Right Offset: 0
 Bottom Offset: 0
 Width: 32
 Height: 32

Optimisation
☐ Add extra pipelining registers
☐ Reduced control register readback
 How user packets are handled: ☐ No user packets allowed ☐ Discard all user packets received ☒ Pass all user packets through to the output

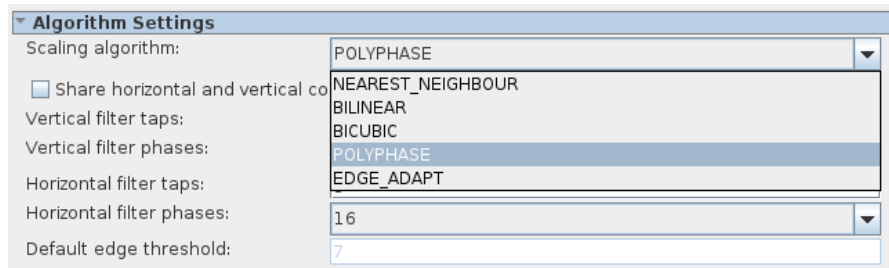
Block Diagram Signals:
 main_clock_clk, main_reset_reset, din_data[23..0], din_valid, din_startofpacket, din_endofpacket, din_ready, control_address[2..0], control_byteenable[3..0], control_write, control_writedata[31..0], control_read, control_readdata[31..0], control_readdatavalid, control_waitrequest, data, valid, startofpacket, endofpacket, ready, dout_data[23..0], dout_valid, dout_startofpacket, dout_endofpacket, dout_ready.

Annotations:
 IN/OUT Avalon ST-Video

- Enable the Use control port to export an Avalon-MM slave where we will connect our Nios CPU
- 8 bits per color
- 3 color planes (RGB)
- 1920x1080 as maximum resolution
- Clipping method: Rectangle

Note in the block diagram that this core has 2 Avalon Video-ST interfaces: `din` and `dout`. This module is intended to be connected internally to other modules and don't export externally.

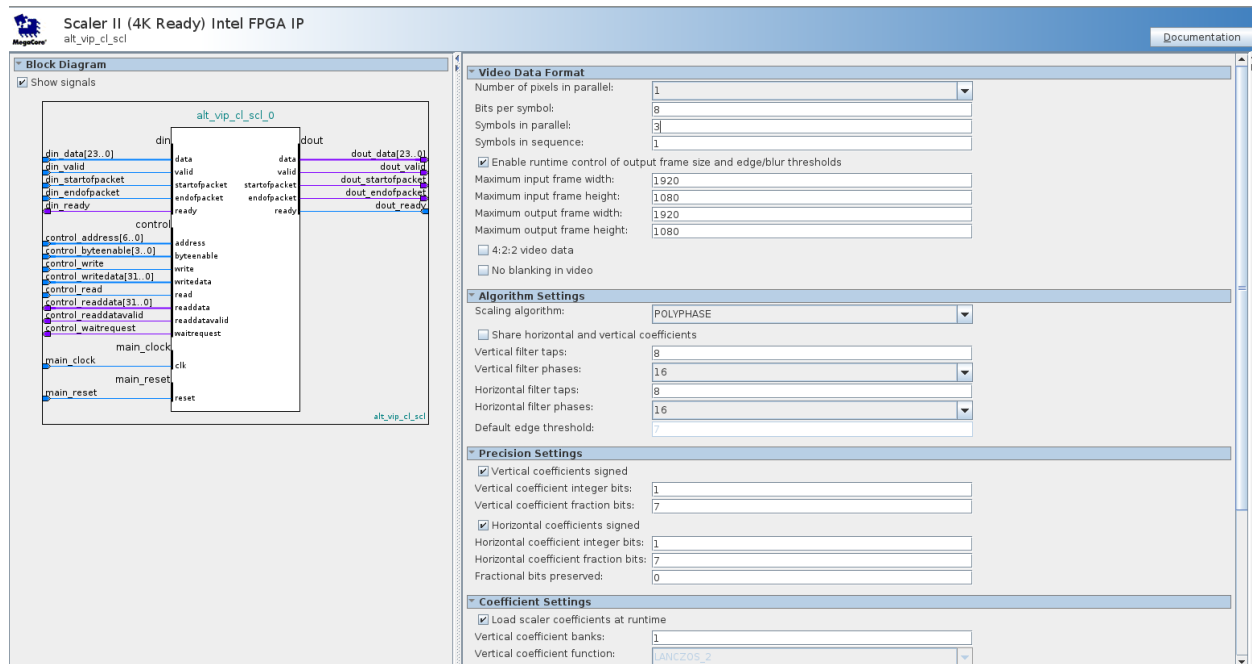
7. Rename it as **clp**.
8. Then, let's add a Scaler IP, that way we can resize the input video to any other arbitrary resolution. Go to **IP Catalog->Library->DSP->Video and Image Processing->Scaler II** and click **Add**. The Scaler allows different configurations from low to a very high quality, making a tradeoff with regards to resource utilization. From a simple **NEAREST_NEIGHBOUR** to a sophisticated **EDGE_ADAPT**, can be selected in the `Scaling algorithm` field:



Select the following parameters:

- Bits per symbol: 8
- Symbols in parallel: 3
- Enable runtime control on output frame size: Yes
- Maximum frame width/height: 1920/1080
- Scaling algorithm: POLYPHASE
- Load scaler coefficients at runtime: Yes

The VIP C++ API is providing some methods to allow dynamic generation of coefficients sets according to the input/output dimensions set at the scaler. So, in run time we are able to calculate new coefficients either when the input resolution change or we apply a different scaling factor. We will practice this in the software development part of this tutorial.



Same as the Clipper module, this core has 2 Avalon Video-ST interfaces: `din` and `dout`. This module is intended to be connected internally to other modules and don't export externally.

9. Rename as `scl`

10. Due to the nature of the Avalon ST-Video protocol and how the data is transferred (frame wise packetized), we need to add an intermediate memory to buffer frames and avoid pipeline stalls (will study this in more detail in upcoming sessions). This frame buffer can also be used to compensate for non-synchronous video input/output clocks and to perform frame rate conversion.

Go to **IP Catalog->Library->DSP->Video and Image Processing->Frame Buffer II** and click on **Add**

- Select frame size as: 1920/1080
- Bits per color sample: 3
- Number of color planes: 3
- Use separate clock for the Avalon-MM master interfaces: Yes
- Avalon-MM master local ports width: 256
 - This is because we are using an external DDR3 mem of 32bits data bus in quarter rate configuration: $32b \times 2(DDR) \times 4(Quarter\ rate) = 256b$ local bus
- FIFO depth Write/Read: 128
- Av-MM burst target Read/Write: 64
 - Video is, by nature, a sequential data pattern on the memory that benefits for large bursts. In the DDR3 memory controller we will add (later on this tutorial) supports a max burst length of 64.
 - FIFO should be set at 2x larger that the burst length, that way we can use "double buffer" mode to increase efficiency.

- Enable Frame dropping and repetition to allow for frame rate conversion and asynchronous clocks
- Enable Drop invalid frames

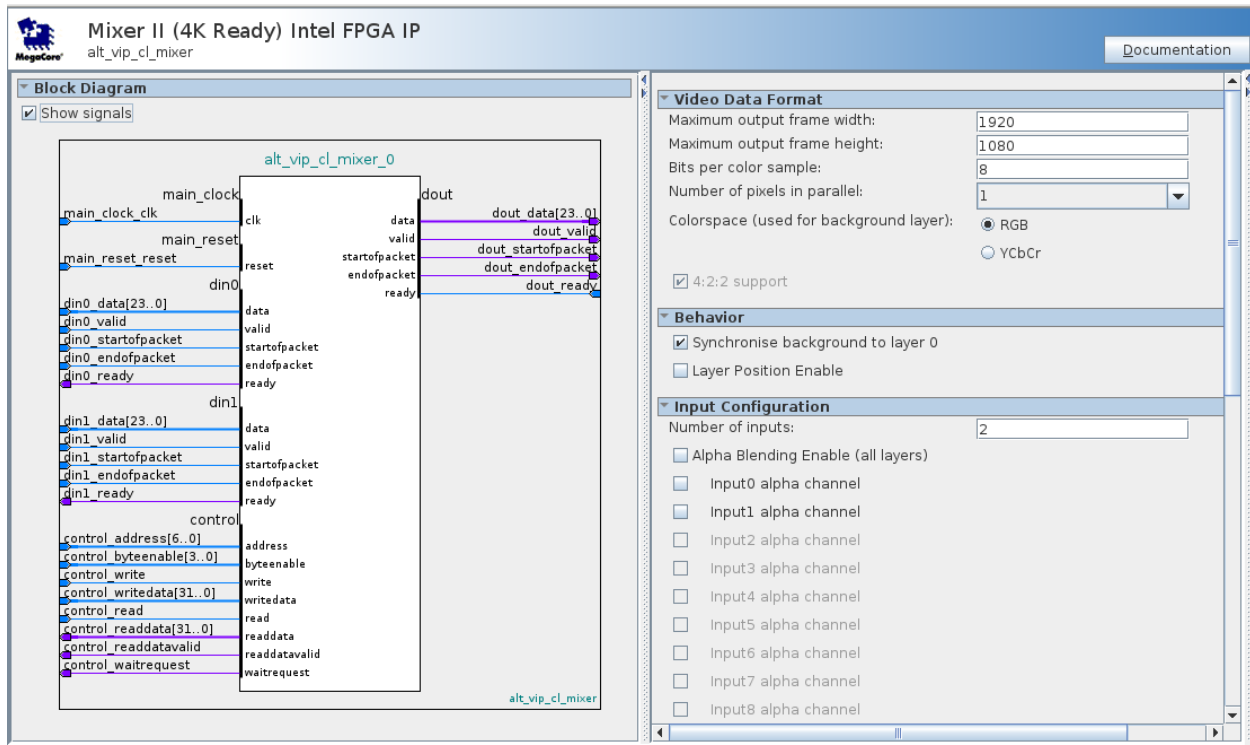
The Frame Buffer has a 2x Avalon-ST ports for video in/out (`din`, `dout`) and 2x Avalon-MM master ports for writing and reading back frames from the external DDR memory bank (`mem_master_wr`, `mem_master_rd`). Having 2 different clocks for video and memory ports allows clock-crossing functionality and to run the video streams and memory banks at different frequencies.

11. Rename it as **vfb**

12. In order to make a final output composition of our processed video along with the static pattern generation, we need to include a Mixer core in our pipeline.

Go to **IP Catalog->Library->DSP->Video and Image Processing->Mixer II** and click on **Add**

- Select Frame Width/Height to: 1920/1080
- Bits per color sample: 8
- Synchronize background to layer 0: Enable
- Number of inputs: 2



The symbol for the mixer shows 2x Avalon-ST input ports (**din0**, **din1**) and 1 Avalon-ST output with the final mixing (**dout**).

By using the Avalon-MM control port and the provided C++ API, we will be able to enable/disable the visualization on the layers (Background layer cannot be disabled) and to position each layer with a relative offset to background (0, 0) as long as we are not exceeding the boundaries.

13. Rename it to **mixer**

We are adding the DDR3 EMIF controller outside the **vip_pipeline** subsystem. We are, indeed, adding the DDR3 EMIF controller in the higher hierarchy level **dp_core**.

We are doing this because, in following sessions, we are sharing the DDR3 memory with the Nios CPU to run the executable program, as well as to implement the buffer memory to add text and graphic content on top of our live video.

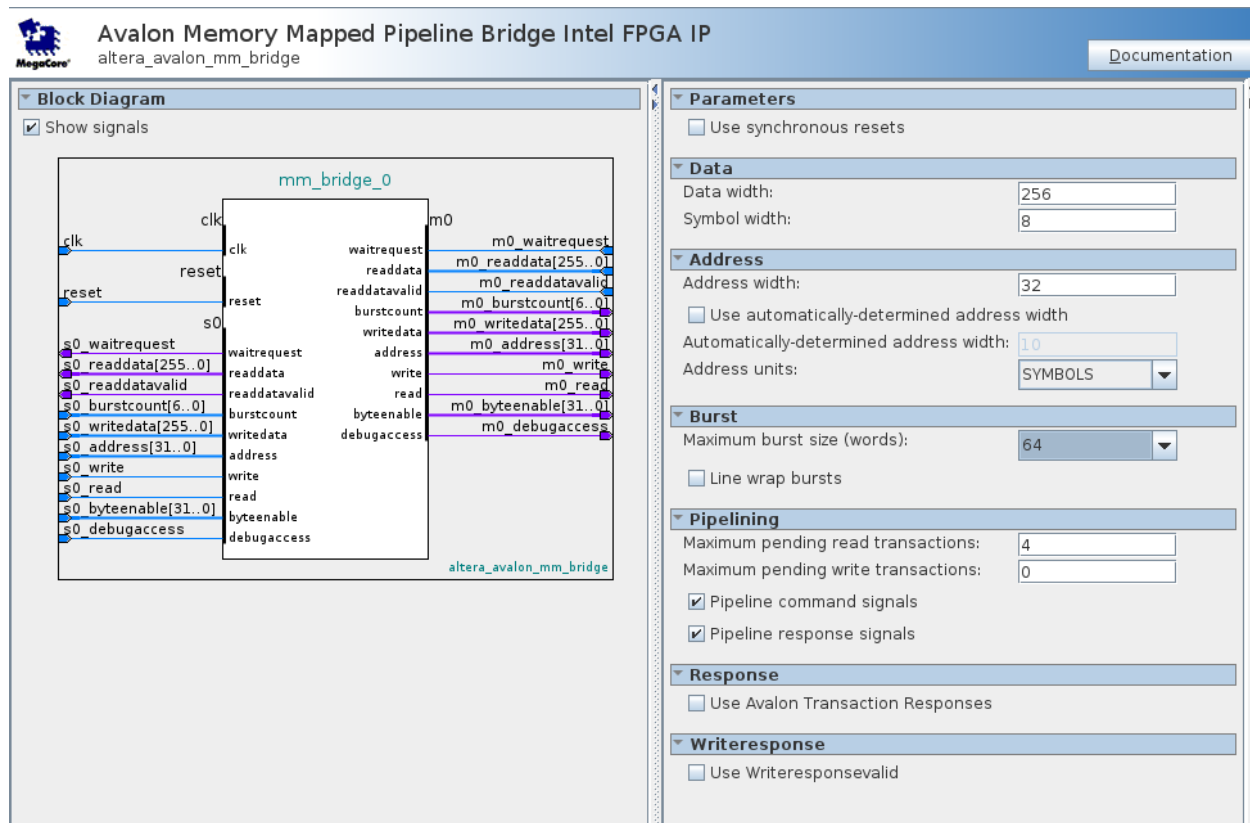
In order to access the DDR3 EMIF from the **vfb** in the **vip_pipeline**, we need to export the 2x Avalon-MM generated. We are doing this using an Avalon-MM Pipeline Bridge, whose functionality is twofold:

- Consolidates the 2x Avalon-MM masters (read/write) from the **vfb** into a single one
- Allows better control over the Avalon-MM architecture topology, by creating branches, to ease timing closure in congested systems

14. Go to IP Catalog->Library->Basic Functions->Bridges and Adaptors->Memory Mapped->Avalon Memory Mapped Pipeline Bridge and click on Add

- Data width: 256 (same as **vfb**)

- Address width: 32 (to be consistent with what is generated in the av-mm for **vfb**)
- Maximum burst size: 64 (same as **vfb**)



15. Rename it as **mm_bridge_emif**

The DDR3 EMIF controller we are adding later will generate its own user clock and reset lines, and we will need to connect these lines to our Video Frame Buffer. To allow this, let's add some additional Clock and Reset bridges to export the interfaces and allow external connections.

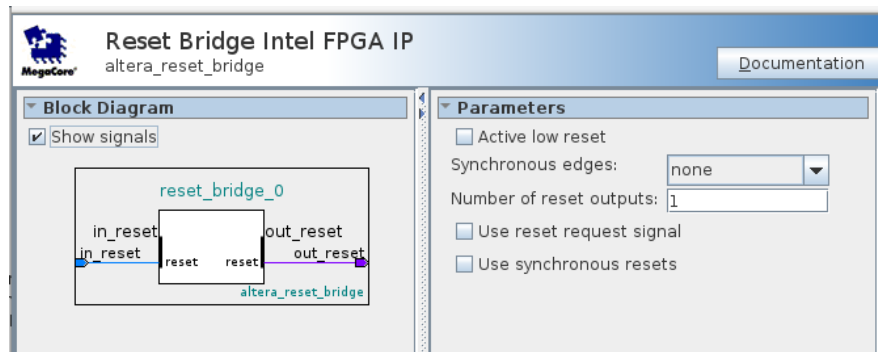
16. Go to **IP Catalog->Library->Basic Functions->Bridges and Adaptors->Clock->Clock Bridge** and click on **Add**

17. Rename it as **clk_emif**

18. Go to **IP Catalog->Library->Basic Functions->Bridges and Adaptors->Reset->Reset Bridge** and click on **Add**

The generated reset signal is already synchronized at source (in the EMIF controller), so don't need to sync again here.

- Synchronous edges: none



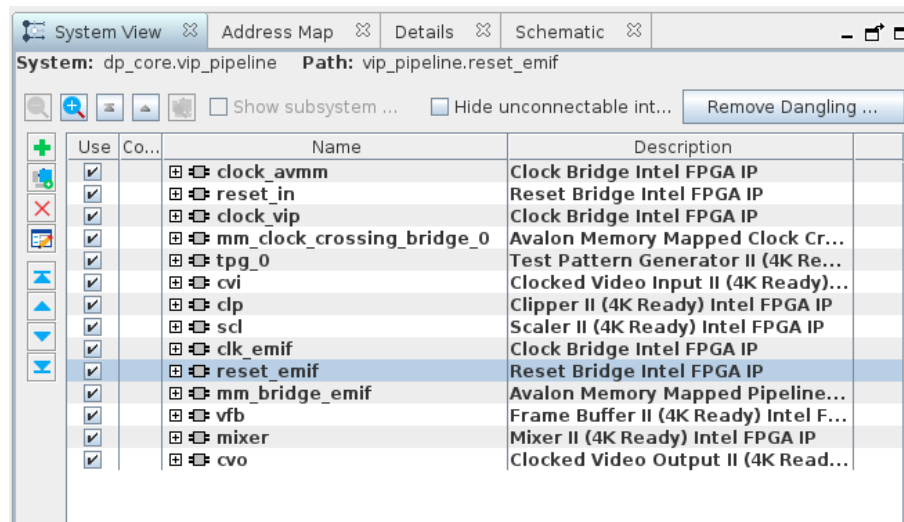
19. Rename as **reset_emif**

We have already finished to add all the necessary components for our application. Let's now re-arrange them for better connection readability, export the interfaces needed to be used externally and make the internal connections within the subsystem.

20. Re-arrange components

Right-click on the System View pane and select: Collapse All

Use the UP/Down blue arrows in the side toolbar to organize the order of the different components, according to the following picture:



21. Exporting interfaces


Let's export the following conduits and signals (in addition on what we have already exported before)

- **cvi**: cvi_status_update_irq
- **cvi**: cvi_clocked_video
- **clk_emif**: in_clk
- **reset_emif**: in_reset
- **mm_bridge_emif**: m0

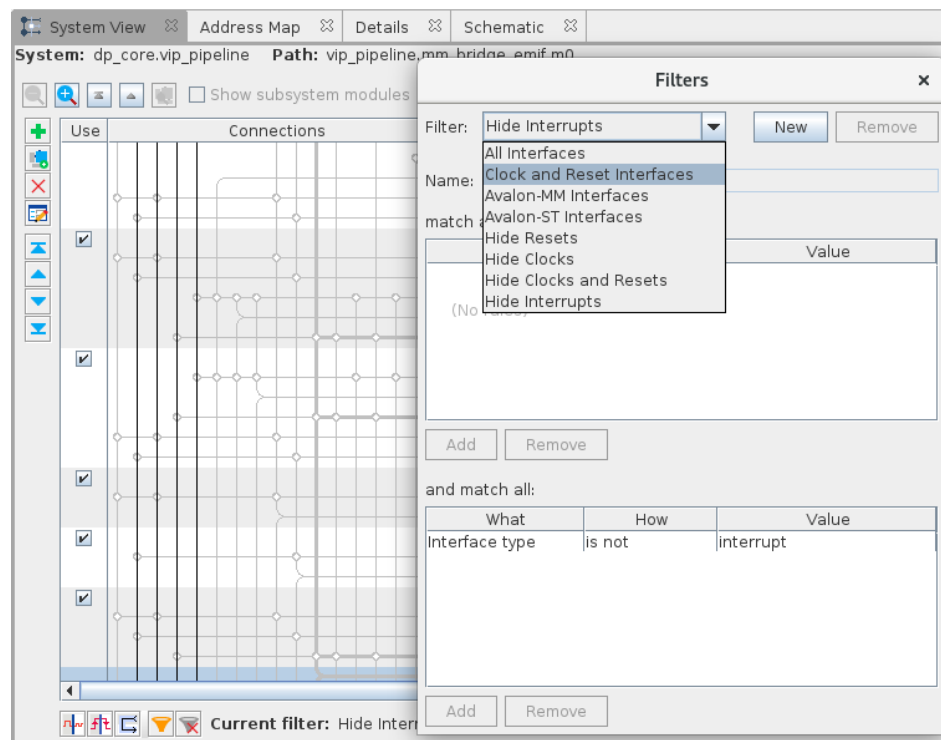
22. Making the connections.

We will use the filtering capabilities of Platform Designer to make easier the different connections in crowded subsystems.

- Connect Clock and Resets

At the bottom of the System View pane, you can find the click on Select Filter Button 

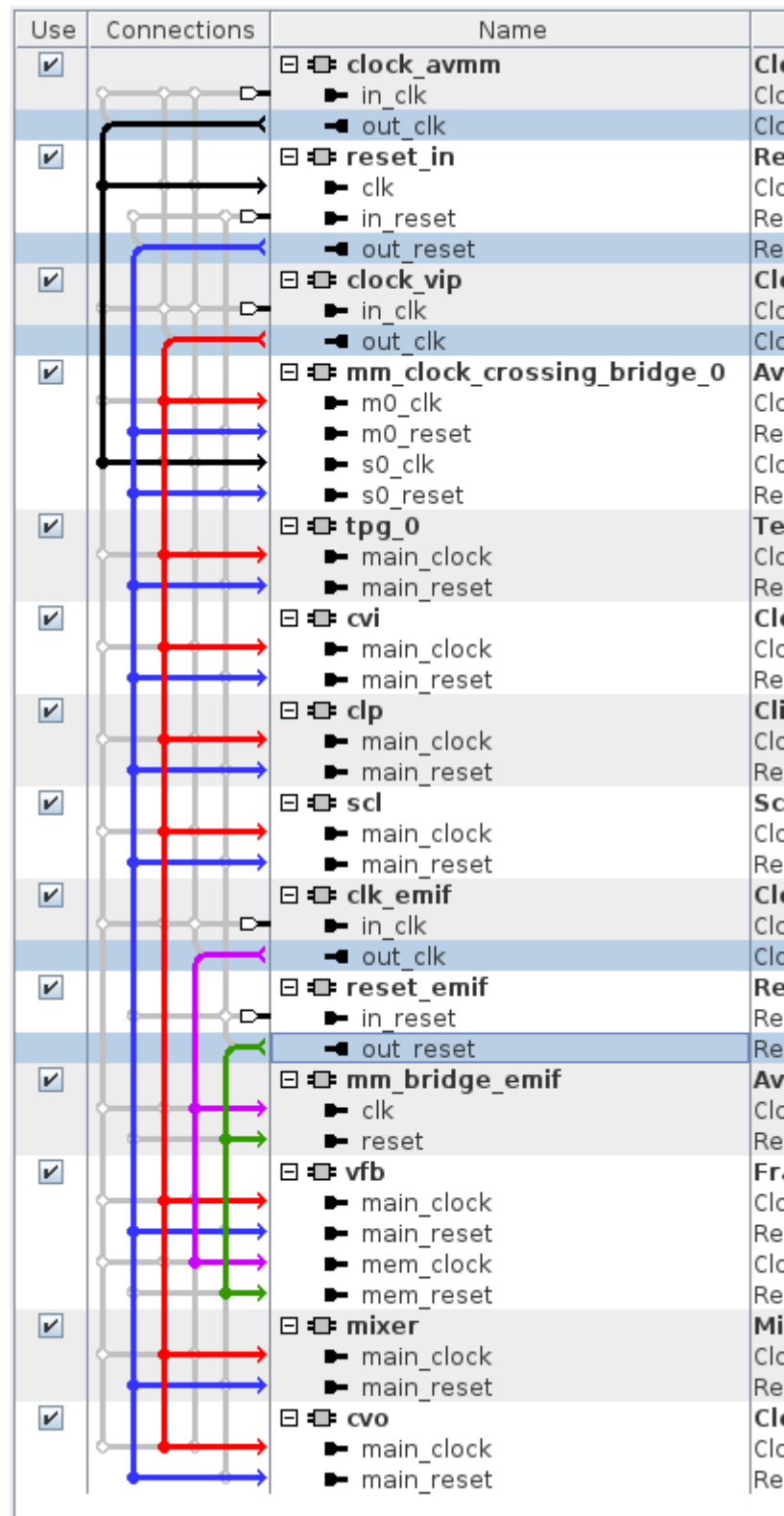
The Filters Dialog appears: Filter by Clock and Reset Interfaces



Do the following connections:

- **Clock_avmm:** out_clk to
 - **Reset_in:** clk
 - **Mm_clock_crossing_bridge_0:** s0_clk
- **Reset_in:** out_reset to
 - **Mm_clock_crossing_bridge_0:** m0_reset
 - **Mm_clock_crossing_bridge_0:** s0_reset
 - **Tpg_0:** main_reset
 - **Cvi:** main_reset
 - **Clp:** main_reset
 - **Scl:** main_reset
 - **Vfb:** main_reset
 - **Mixer:** main_reset

- **Cvo:** main_reset
- **Clock_vip:** out_clk to
 - **Mm_clock_crossing_bridge_0:** m0_clk
 - **Tpg_0:** main_clock
 - **Cvi:** main_clock
 - **Clp:** main_clock
 - **Scl:** main_clock
 - **Vfb:** main_clock
 - **Mixer:** main_clock
 - **Cvo:** main_clock
- **Clk_emif:** out_clock to
 - **Mm_bridge_emif:** clk
 - **Vfb:** mem_clock
- **Reset_emif:** out_reset to
 - **Mm_bridge_emif:** reset
 - **Vfb:** mem_reset



- Avalon-MM

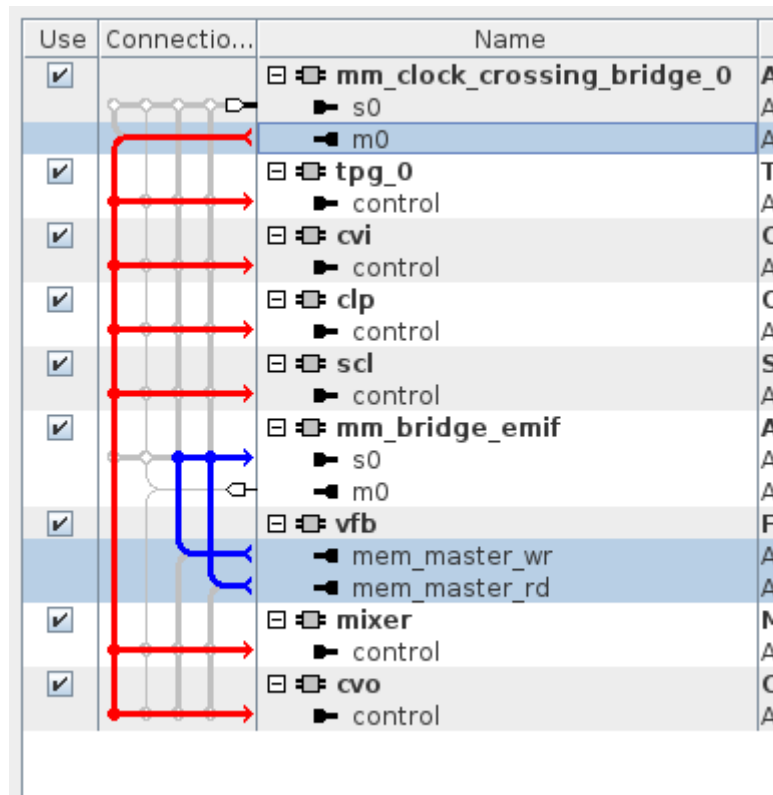
Now, let's filter by Avalon-MM Interfaces and make the following connections

Connect the **mm_clock_crossing_bridge** master interface to all the VIP cores slave ports

- **mm_clock_crossing_bridge_0**: m0 to
 - **tpg_0**: control
 - **cvi**: control
 - **clp**: control
 - **scl**: control
 - **mixer**: control
 - **cvo**: control

Make the connections of the Frame Buffer with the EMIF pipeline bridge

- **mm_bridge_emif**: s0 to
 - **vfb**: mem_master_wr
 - **vfb**: mem_master_rd

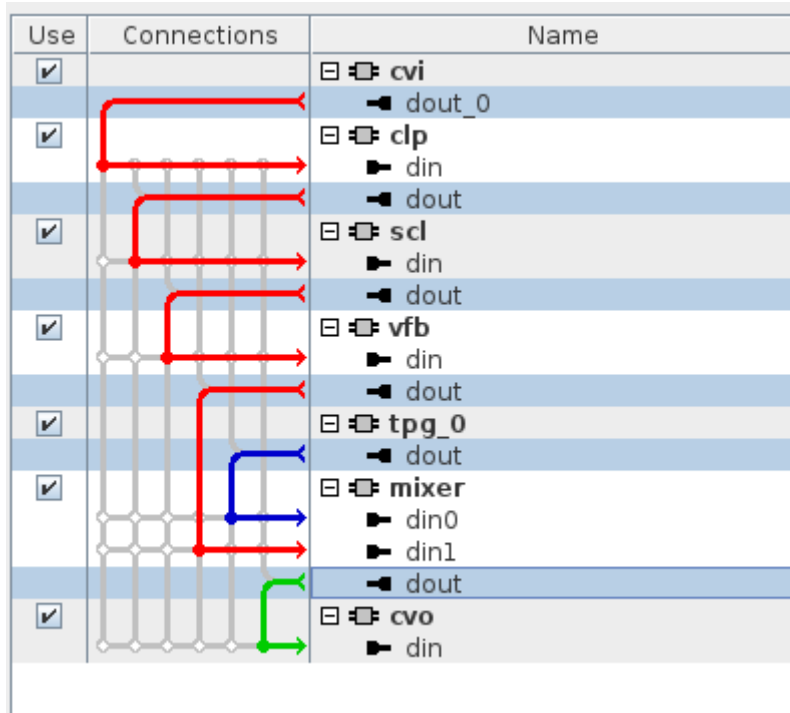


- Avalon-ST

And now let's filter by Avalon-ST Interface to, actually, build the pipeline topology where our video packets will flow through

- **Cvi**: dout_0 -> **clp**: din
- **Clp**: dout -> **scl**: din
- **Scl**: dout -> **vfb**: din
- **Vfb**: dout -> **mixer**: din1

- **Tpg_0**: dout -> **mixer**: din0
- **Mixer**: dout -> **cvo**: din



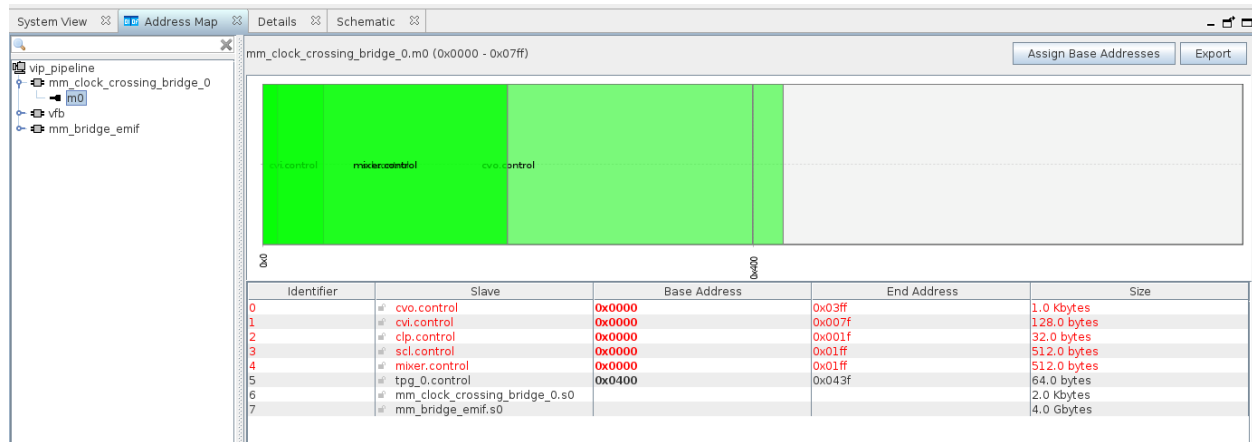
This concludes the internal connection of all the modules within the `vip_pipeline` subsystem

23. Solving Memory Mapped address conflicts

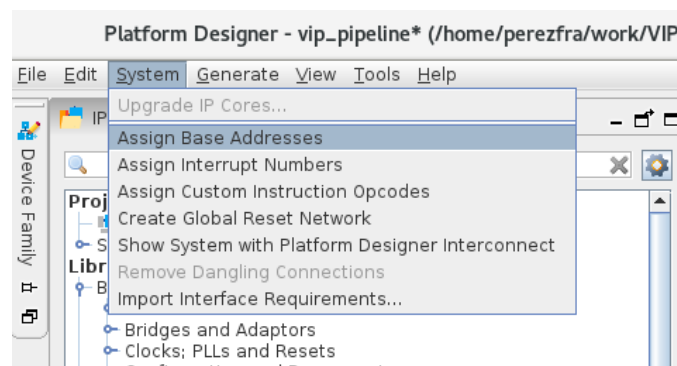
You might have noticed some error messages in the System Messages tab highlighting Memory Mapped overlapping errors

| Type | Path | Message |
|------------------------------|---|--|
| 4 System Connectivity Errors | | |
| Platform Designer Tip | | Please Sync All System Infos before attempting to resolve the following error messages |
| | <code>vip_pipeline.mm_clock_crossing_bridge_0.m0</code> | <code>cvi.control (0x0..0x7f)</code> overlaps <code>cvo.control (0x0..0x3ff)</code> |
| | <code>vip_pipeline.mm_clock_crossing_bridge_0.m0</code> | <code>clp.control (0x0..0x1f)</code> overlaps <code>cvi.control (0x0..0x7f)</code> |
| | <code>vip_pipeline.mm_clock_crossing_bridge_0.m0</code> | <code>scl.control (0x0..0x1ff)</code> overlaps <code>clp.control (0x0..0x1f)</code> |
| | <code>vip_pipeline.mm_clock_crossing_bridge_0.m0</code> | <code>mixer.control (0x0..0x1ff)</code> overlaps <code>scl.control (0x0..0x1ff)</code> |

We have connected 6 slaves (CVI, CLP, SCL, TPG_0, MIXER and CVO) to the same master `mm_clock_crossing_bridge:m0`, and some of them 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



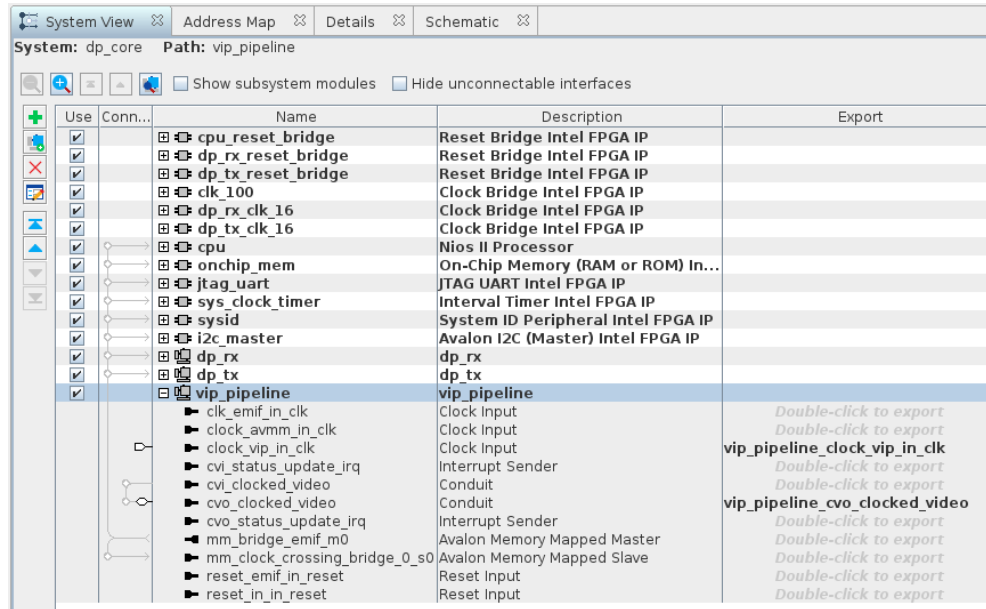
After execution, each slave will have its exclusive address space



24. 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. As we had already instantiated the `vip_pipeline` in the design, as part of the loading process of `dp_core`, `vip_pipeline` gets refreshed to account for the changes we have made. `vip_pipeline` shows now the new ports and interfaces added in the previous step. Let's connect them.



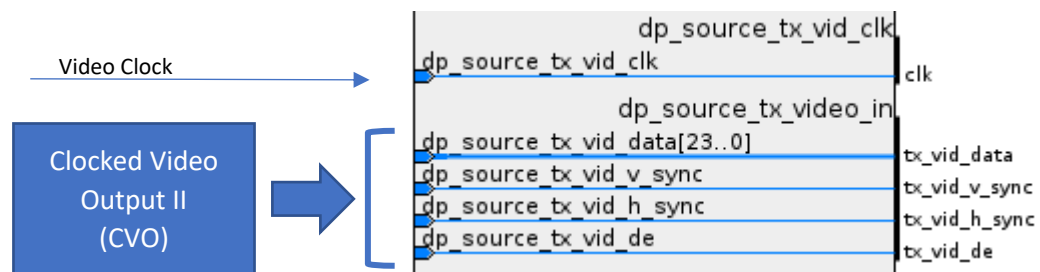
System View Address Map Details Schematic

System: dp_core Path: vip_pipeline

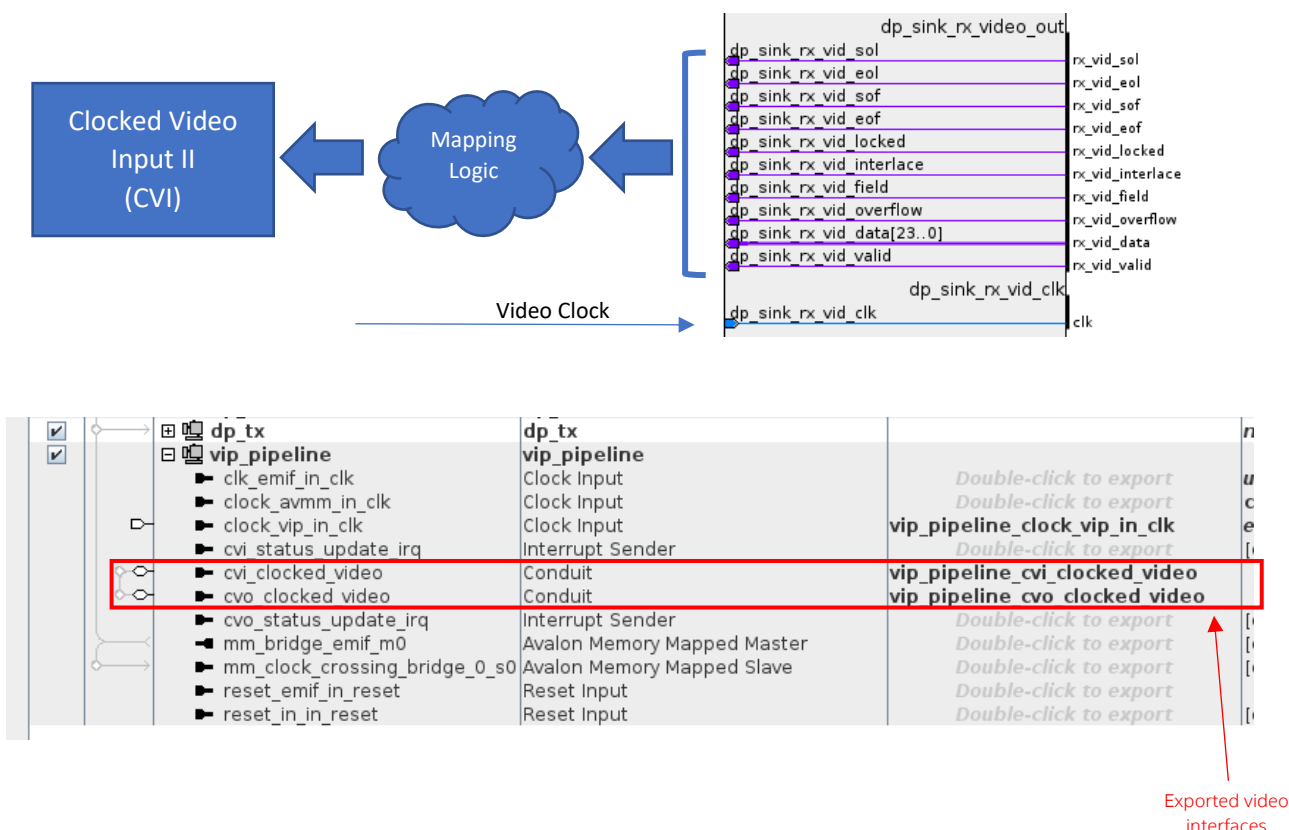
Show subsystem modules Hide unconnectable interfaces

| Use | Conn... | Name | Description | Export |
|-------------------------------------|---------|-------------------------------|------------------------------------|------------------------|
| <input checked="" type="checkbox"/> | | cpu_reset_bridge | Reset Bridge Intel FPGA IP | |
| <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 | |
| <input checked="" type="checkbox"/> | | dp_rx_clk_16 | Clock Bridge Intel FPGA IP | |
| <input checked="" type="checkbox"/> | | dp_tx_clk_16 | Clock Bridge Intel FPGA IP | |
| <input checked="" type="checkbox"/> | | cpu | Nios II Processor | |
| <input checked="" type="checkbox"/> | | onchip_mem | On-Chip Memory (RAM or ROM) In... | |
| <input checked="" type="checkbox"/> | | jtag_uart | JTAG UART Intel FPGA IP | |
| <input checked="" type="checkbox"/> | | sys_clock_timer | Interval Timer Intel FPGA IP | |
| <input checked="" type="checkbox"/> | | sysid | System ID Peripheral Intel FPGA IP | |
| <input checked="" type="checkbox"/> | | i2c_master | Avalon I2C (Master) Intel FPGA IP | |
| <input checked="" type="checkbox"/> | | dp_rx | dp_rx | |
| <input checked="" type="checkbox"/> | | dp_tx | dp_tx | |
| <input checked="" type="checkbox"/> | | vip_pipeline | vip_pipeline | |
| | | clk_emif_in_clk | Clock Input | Double-click to export |
| | | clock_avmm_in_clk | Clock Input | Double-click to export |
| | | clock_vip_in_clk | Clock Input | Double-click to export |
| | | cvi_status_update_irq | Interrupt Sender | Double-click to export |
| | | cvi_clocked_video | Conduit | Double-click to export |
| | | cvo_clocked_video | Conduit | Double-click to export |
| | | cvo_status_update_irq | Interrupt Sender | Double-click to export |
| | | mm_bridge_emif_m0 | Avalon Memory Mapped Master | Double-click to export |
| | | mm_clock_crossing_bridge_0_s0 | Avalon Memory Mapped Slave | Double-click to export |
| | | reset_emif_in_reset | Reset Input | Double-click to export |
| | | reset_in_in_reset | Reset Input | Double-click to export |

3. Our `vip_pipeline` module already had the `clock_vip` and `cvo_clocked_video` interfaces exported. We were using `vip_pipeline: cvo_clocked_video` conduit to connect to the `dp_tx: dp_source_tx_video_in` conduit of the DisplayPort transmitter IP core.



In order to connect our CVI with the DisplayPort RX interface, we would need to export the `vip_pipeline: cvi_clocked_video` input conduit that we will use to connect to the `dp_rx: dp_sink_rx_video_out` to ingest the video captured by the DisplayPort receiver IP core. Note that we will need to add some simple mapping logic to adapt the signal behavior. We will be doing this next in the top level file `c10_dp_demo.v`



4. Add the DDR3 EMIF controller.

As we are buffering incoming video signal, we need to access to the on-board DDR3 memory bank in the C10 GX Devkit. To get more details about the implemented memory bank consult the user guide: https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/cyclone-10-gx-development-kit.html

Follow the next steps to implement a DDR3 controller compliant with the target devkit.

- Go to **IP Catalog->Library->Memory Interfaces and Controllers->External Memory Interfaces Intel Cyclone 10 FPGA IP** and click **Add**. The IP parameter editor for the module will open
- In **General** tab modify the parameters as shown in the picture below. In the Clocks section do these modifications:
 - Memory clock frequency: 932.203 MHz
 - Deselect Use recommended PLL reference clock frequency
 - Expand the PLL reference clock frequency list and select 21.186 MHz

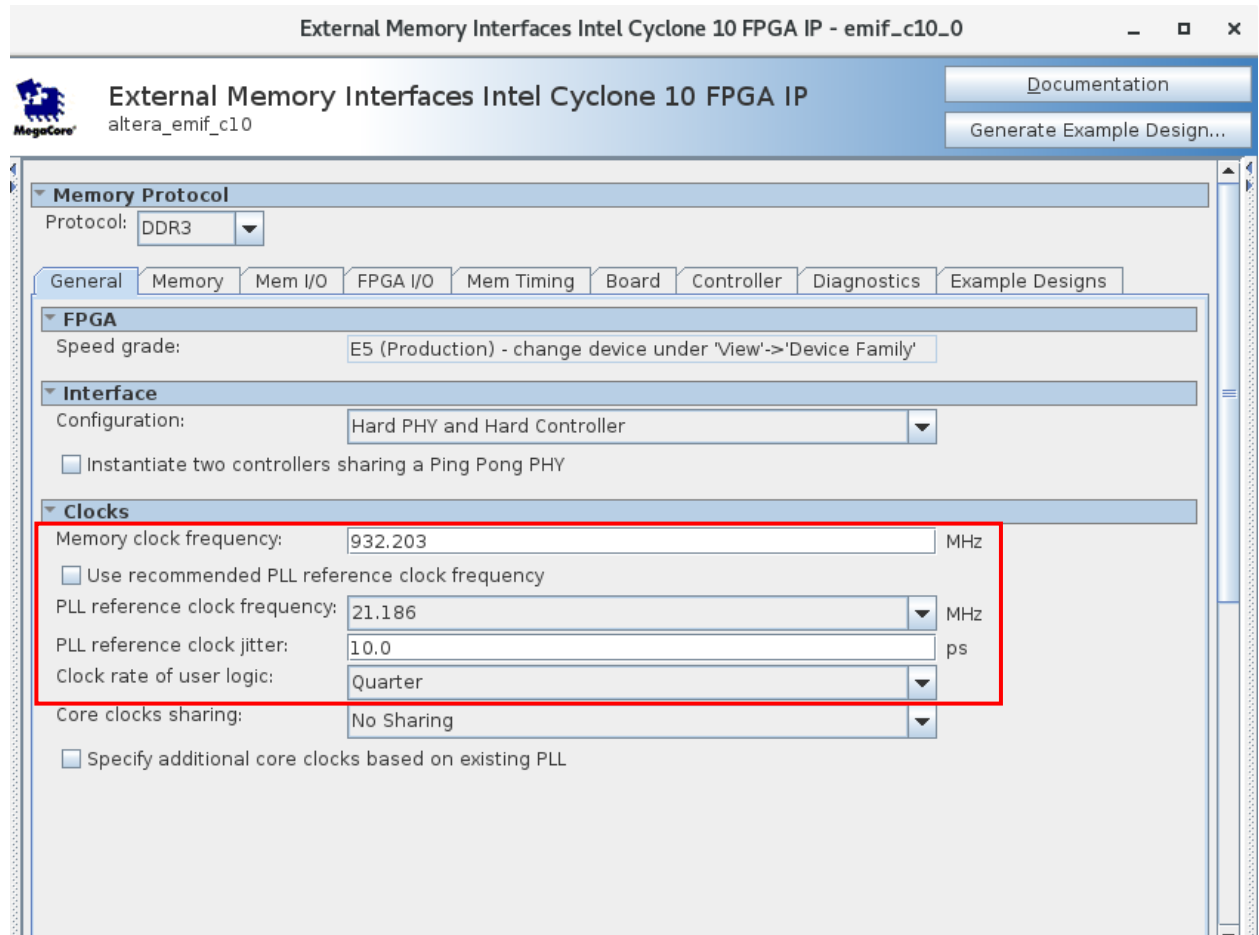
In the C10 GX devkit we have a dedicated oscillator to supply the memory reference clock and it's rated at 21.186 MHz.

Please note that we are using Quarter rate as Clock rate of user logic.

The memory interface in the board has 40 bits of data width, but we are enabling ECC on the interface resulting in 32 bits user data width. How we translate the bus data width from the memory to the (internal to user logic in the FPGA) local bus?

32 bits (external data memory) x 2 (DDR interface) x 4 (quarter rate logic) = 256 bits (local bus)

Remember we set before our Video Frame Buffer component and the associated Pipeline Bridge to 256bit data width. This matches the physical local bus available.



- c. Move to Memory tab and select:
- Memory format: Component
 - DQ width: 40 (32 bit user data + 8 ECC)
 - Keep the rest as default

Memory Protocol

Protocol: DDR3

General **Memory** Mem I/O FPGA I/O Mem Timing Board Controller Diagnostics Example Designs

Topology

| | |
|-------------------------|------------------------|
| Memory format: | Component |
| DQ width: | 40 |
| DQ pins per DQS group: | 8 |
| Number of DQS groups: | 5 |
| Number of clocks: | 1 |
| Number of chip selects: | 1 |
| Row address width: | 15 |
| Column address width: | 10 |
| Bank address width: | 3 |

☒ Enable DM pins

Latency and Burst

| | |
|--------------------------------------|-----------------------|
| Memory CAS latency setting: | 14 |
| Memory write CAS latency setting: | 10 |
| Memory additive CAS latency setting: | Disabled |

Mode Register Settings

☒ Hide advanced mode register settings

d. Move to Mem I/O tab

- In Memory I/O settings->ODT Rtt nominal value select: RZQ/4
- Keep the rest as default

Memory Protocol
Protocol: DDR3

General | **Memory** | Mem I/O | FPGA I/O | Mem Timing | Board | Controller | Diagnostics | Example Designs

Memory I/O Settings
Output drive strength setting: RZQ/7
ODT Rtt nominal value: RZQ/4
Dynamic ODT (Rtt_WR) value: RZQ/4

ODT Activation
☒ Use Default ODT Assertion Tables

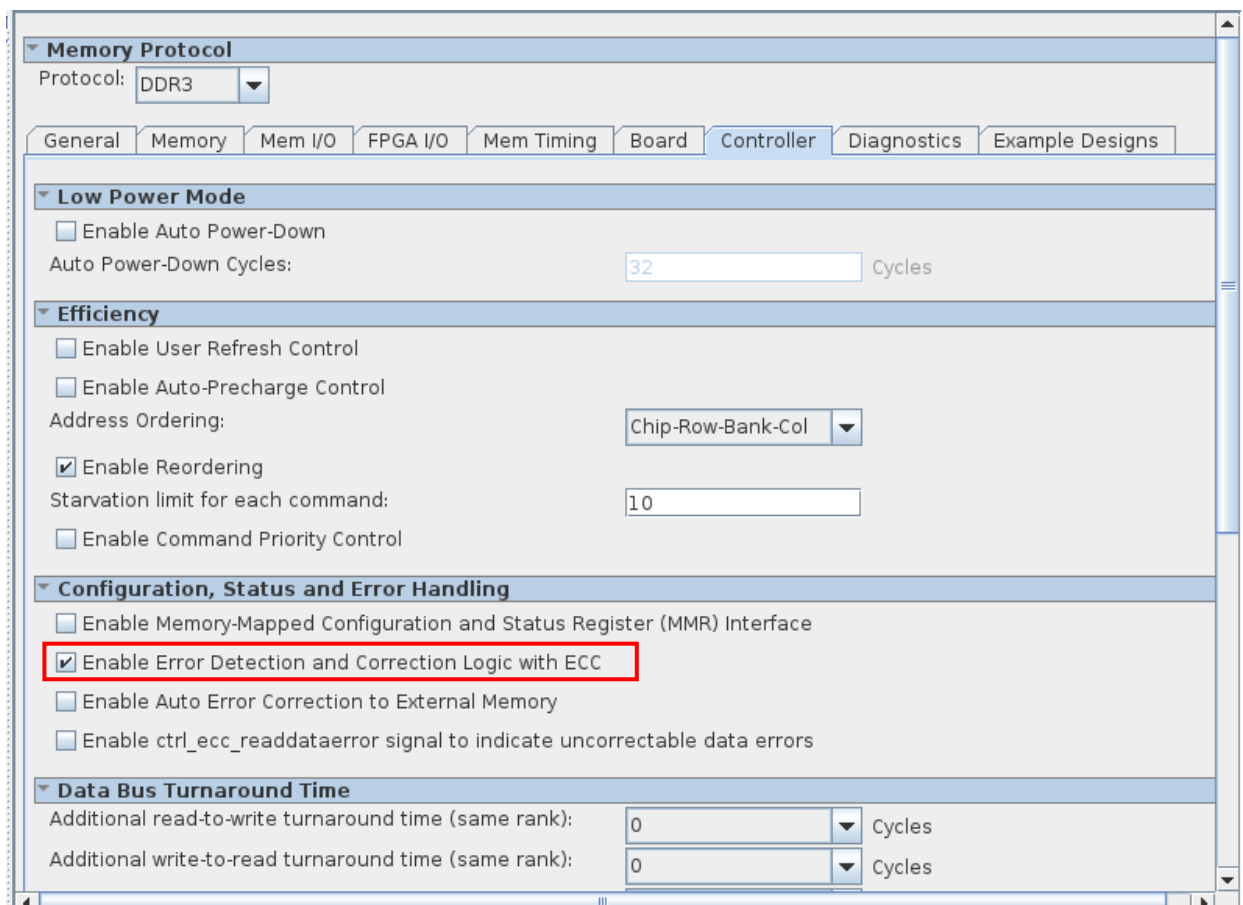
Derived ODT Matrix for Read Accesses
This matrix is derived based on settings for RTT_DRV, RTT_NOM and the read ODT assertion table.

| Read Target | ODT0 Value | ODT1 Value | ODT2 Value | ODT3 Value |
|-------------|---------------|------------|------------|------------|
| Rank 0 | (Drive) RZQ/7 | - | - | - |
| - | - | - | - | - |
| - | - | - | - | - |
| - | - | - | - | - |

Derived ODT Matrix for Write Accesses
This matrix is derived based on settings for RTT_WR, RTT_NOM and the write ODT assertion table.

| Write Target | ODT0 Value | ODT1 Value | ODT2 Value | ODT3 Value |
|--------------|-----------------|------------|------------|------------|
| Rank 0 | (Dynamic) RZQ/4 | - | - | - |
| - | - | - | - | - |
| - | - | - | - | - |

- e. Keep as default the settings in Mem I/O, FPGA I/O, Mem Timing, Board and Diagnostics tabs.
- f. Go to the Controller tab and enable ECC
 - o Configuration, Status and Error Handling -> Enable Error Detection and Correction Logic with ECC



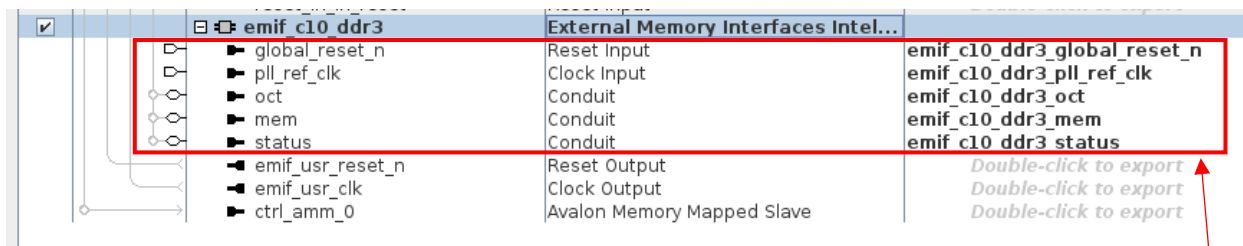
g. Press Finish button to close the IP parameter editor and add the DDR3 EMIF controller to the system

h. Rename as emif_c10_ddr3

i. Export interfaces

Let's export the external interfaces we need to connect to FPGA pins to access to the external DDR3 memory bank on the board. Double click in the Export column to export:

- o emif_c10_ddr3: global_reset_n
- o emif_c10_ddr3: pll_ref_clk
- o emif_c10_ddr3: oct
- o emif_c10_ddr3: mem
- o emif_c10_ddr3: status



j. Connect the signals

Exported memory interfaces

As mentioned in previous steps, the EMIF controller generates its own user clock and reset signals we need to connect to our logic driving the interface. These are the `emif_usr_reset_n` and `emif_usr_clk` signals. We need to connect as well the Avalon-MM slave interface of the memory controller to the master we created in the `vip_pipeline` (driven by the Frame Buffer through a Pipeline Bridge).

- **emif_c10_ddr3**: `emif_usr_reset_n` -> **vip_pipeline**: `reset_emif_in_reset`
- **emif_c10_ddr3**: `emif_usr_clk` -> **vip_pipeline**: `clk_emif_in_clk`
- **emif_c10_ddr3**: `ctrl_amm_0` -> **vip_pipeline**: `mm_bridge_emif_m0`

- k. This finalizes the integration of the DDR3 EMIF controller in our system.
5. Finalize the **vip_pipeline** connections. There is one (optional) missing connection here.
 - **vip_pipeline**: `cvi_status_update_irq` -> **cpu**: `irq`

Note: We are not using interrupt driven program in this webinar series, but let's get the CVI and CVO interrupt connected as we might probably generate additional examples, on how to use interrupts, in future tutorials.

6. Then, save the system and generate it 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.

We need to add the newly generated DDR3 EMIF controller and connect the Clocked Video Input interface with the DisplayPort Receiver IP core.

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 **cvi**. Copy all the complete lines:

```
.dp_tx_xdash_out_port      (connected_to_dp_tx_xdash_out_port_),          // output,
.vip_pipeline_clock_vip_in_clk_clk  (connected_to_vip_pipeline_clock_vip_in_clk_clk_),    // input,
.vip_pipeline_cvi_clocked_video_vid_clk  (connected_to_vip_pipeline_cvi_clocked_video_vid_clk_),    // input,
.vip_pipeline_cvi_clocked_video_vid_data  (connected_to_vip_pipeline_cvi_clocked_video_vid_data_),    // input,
.vip_pipeline_cvi_clocked_video_vid_de  (connected_to_vip_pipeline_cvi_clocked_video_vid_de_),    // input,
.vip_pipeline_cvi_clocked_video_vid_datavalid  (connected_to_vip_pipeline_cvi_clocked_video_vid_datavalid_),    // input,
.vip_pipeline_cvi_clocked_video_vid_locked  (connected_to_vip_pipeline_cvi_clocked_video_vid_locked_),    // input,
.vip_pipeline_cvi_clocked_video_vid_f  (connected_to_vip_pipeline_cvi_clocked_video_vid_f_),    // input,
.vip_pipeline_cvi_clocked_video_vid_v_sync  (connected_to_vip_pipeline_cvi_clocked_video_vid_v_sync_),    // input,
.vip_pipeline_cvi_clocked_video_vid_h_sync  (connected_to_vip_pipeline_cvi_clocked_video_vid_h_sync_),    // input,
.vip_pipeline_cvi_clocked_video_vid_color_encoding  (connected_to_vip_pipeline_cvi_clocked_video_vid_color_encoding_),    // input,
.vip_pipeline_cvi_clocked_video_vid_bit_width  (connected_to_vip_pipeline_cvi_clocked_video_vid_bit_width_),    // input,
.vip_pipeline_cvi_clocked_video_sof  (connected_to_vip_pipeline_cvi_clocked_video_sof_),    // output,
.vip_pipeline_cvi_clocked_video_sof_locked  (connected_to_vip_pipeline_cvi_clocked_video_sof_locked_),    // output,
.vip_pipeline_cvi_clocked_video_refclk_div  (connected_to_vip_pipeline_cvi_clocked_video_refclk_div_),    // output,
.vip_pipeline_cvi_clocked_video_clipping  (connected_to_vip_pipeline_cvi_clocked_video_clipping_),    // output,
.vip_pipeline_cvi_clocked_video_padding  (connected_to_vip_pipeline_cvi_clocked_video_padding_),    // output,
.vip_pipeline_cvi_clocked_video_overflow  (connected_to_vip_pipeline_cvi_clocked_video_overflow_),    // output,
.vip_pipeline_cvo_clocked_video_vid_clk  (connected_to_vip_pipeline_cvo_clocked_video_vid_clk_),    // input,
```

2. Open `c10_dp_demo.v` and look for where the **dp_core_i** is instantiated. Scroll right after the last port declared (should be the **cvo** module we added in the previous step) and paste the new

ones. You can re-arrange ports to have **cvi** before **cvo** in the instantiation but doesn't matter. You can add some comments to the code, if you wish.

```
// -----
// VIP Pipeline Sub-System
// -----
// Common Clock
.vip_pipeline_clock_vip_in_clk_clk      (dp_rx_vid_clk),          // input,   width = 1,   vip_pipeline
// Clocked Video Input
.vip_pipeline_cvi_clocked_video_vid_clk  (connected_to_vip_pipeline_cvi_clocked_video_vid_clk_),
.vip_pipeline_cvi_clocked_video_vid_data (connected_to_vip_pipeline_cvi_clocked_video_vid_data_),
.vip_pipeline_cvi_clocked_video_vid_de   (connected_to_vip_pipeline_cvi_clocked_video_vid_de_),
.vip_pipeline_cvi_clocked_video_vid_datavalid (connected_to_vip_pipeline_cvi_clocked_video_vid_datavalid_),
.vip_pipeline_cvi_clocked_video_vid_locked (connected_to_vip_pipeline_cvi_clocked_video_vid_locked_),
.vip_pipeline_cvi_clocked_video_vid_f     (connected_to_vip_pipeline_cvi_clocked_video_vid_f_),
.vip_pipeline_cvi_clocked_video_vid_v_sync (connected_to_vip_pipeline_cvi_clocked_video_vid_v_sync_),
.vip_pipeline_cvi_clocked_video_vid_h_sync (connected_to_vip_pipeline_cvi_clocked_video_vid_h_sync_),
.vip_pipeline_cvi_clocked_video_vid_color_encoding (connected_to_vip_pipeline_cvi_clocked_video_vid_color_encoding_),
.vip_pipeline_cvi_clocked_video_vid_bit_width (connected_to_vip_pipeline_cvi_clocked_video_vid_bit_width_),
.vip_pipeline_cvi_clocked_video_vid_sof     (connected_to_vip_pipeline_cvi_clocked_video_vid_sof_),
.vip_pipeline_cvi_clocked_video_vid_sof_locked (connected_to_vip_pipeline_cvi_clocked_video_vid_sof_locked_),
.vip_pipeline_cvi_clocked_video_vid_refclk_div (connected_to_vip_pipeline_cvi_clocked_video_vid_refclk_div_),
.vip_pipeline_cvi_clocked_video_clipping    (connected_to_vip_pipeline_cvi_clocked_video_clipping_),
.vip_pipeline_cvi_clocked_video_padding     (connected_to_vip_pipeline_cvi_clocked_video_padding_),
// Clocked Video Output
.vip_pipeline_cvo_clocked_video_vid_clk  (dp_tx_vid_clk),          // input,   width = 1,   vip_pipeline_cvo
.vip_pipeline_cvo_clocked_video_vid_data (tx_vid_data),          // output,  width = 24,
.vip_pipeline_cvo_clocked_video_vid_underflow ((), // output, width = 1,
.vip_pipeline_cvo_clocked_video_vid_mode_change ((), // output, width = 1, .vid_m
.vip_pipeline_cvo_clocked_video_vid_std ((), // output, width = 1,
.vip_pipeline_cvo_clocked_video_vid_datavalid (tx_vid_de), // output, width = 1,
.vip_pipeline_cvo_clocked_video_vid_v_sync (tx_vid_vsync), // output, width = 1,
.vip_pipeline_cvo_clocked_video_vid_h_sync (tx_vid_hsync), // output, width = 1,
.vip_pipeline_cvo_clocked_video_vid_f ((), // output, width = 1,
.vip_pipeline_cvo_clocked_video_vid_h ((), // output, width = 1,
.vip_pipeline_cvo_clocked_video_vid_v ((), // output, width = 1,
);
```

- As opposite to CVO can be connect directly to DisplayPort TX core, CVI needs some simple adaption logic to connect to DisplayPort RX core.
This mapping logic is just some delayed versions of the `dp_rx_vid_eol` and `dp_rx_vid_eof` signals to generate `vid_h_sync` and `vid_v_sync` respectively and build a `vid_datavalid` signal using blanking information and `dp_rx_vid_valid` information. See the complete code excerpt below

```
// -----
// DP - CVI II Interface Signals
// -----
// CVI II Hsync and Vsync are derived from delayed versions of DP RX eol and eof signals

// DP - CVI II Interface Signals

reg    vid_h_sync;
reg    vid_v_sync;
reg    h_blanking;
wire   vid_datavalid;

always @(posedge dp_rx_vid_clk)
begin
    vid_h_sync <= dp_rx_vid_eol;
    vid_v_sync <= dp_rx_vid_eof;
end

// CVI II vid_datavalid
always @(posedge cpu_reset or posedge dp_rx_vid_clk)
begin
    if (cpu_reset)
        h_blanking <= 1'b0;
    else
        h_blanking <= dp_rx_vid_eol? 1'b1 :
                      dp_rx_vid_sol? 1'b0 :
                      h_blanking;
end

assign vid_datavalid = (!dp_rx_vid_valid) | h_blanking;
```

4. As recap, we are using a free run generated clock to drive the output. This allows us to get live output video even in the absence of input video signals.
From previous labs, find the PLL instantiation below

```
// -----
// 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 daughter card
// -----

wire tx_vid_clk_freerun;

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 135MHz
    .locked   (),              // output, width = 1, locked.export
    .outclk_0 (tx_vid_clk_freerun) // output, width = 1, outclk0.clk 148.5MHz for freerun output video
);
```

5. Now, we are ready to make the connections between cvi and **dp_rx** modules.
Find the signals connected to the **dp_rx_dp_sink_rx_vid** interface in the **dp_core** module

```
// DisplayPort RX Video Protocol Interface
.dp_rx_dp_sink_rx_vid_clk      (dp_rx_vid_clk),
.dp_rx_dp_sink_rx_vid_sol     (dp_rx_vid_sol),
.dp_rx_dp_sink_rx_vid_eol     (dp_rx_vid_eol),
.dp_rx_dp_sink_rx_vid_sof     (dp_rx_vid_sof),
.dp_rx_dp_sink_rx_vid_eof     (dp_rx_vid_eof),
.dp_rx_dp_sink_rx_vid_locked  (dp_rx_vid_locked),
.dp_rx_dp_sink_rx_vid_interlace (),
.dp_rx_dp_sink_rx_vid_field   (),
.dp_rx_dp_sink_rx_vid_overflow (),
.dp_rx_dp_sink_rx_vid_data    (dp_rx_vid_data),
.dp_rx_dp_sink_rx_vid_valid   (dp_rx_vid_valid),
```

We are using `dp_rx_vid_eol`, `dp_rx_vid_eof`, `dp_rx_vid_sol` and `dp_rx_vid_valid` to generate our mapping logic to **cvi**

We are connecting `dp_rx_vid_clk`, `dp_rx_vid_data`, `dp_rx_vid_valid`, `dp_rx_vid_locked` and `dp_rx_vid_field` directly to **cvi**. See the complete connection below

```
// -----
// VIP Pipeline Sub-System
// -----
// Common clock
.vip_pipeline_clock_vip_in_clk_clk      (dp_rx_vid_clk),
// Clocked Video Input
.vip_pipeline_cvi_clocked_video_vid_clk (dp_rx_vid_clk),
.vip_pipeline_cvi_clocked_video_vid_data (dp_rx_vid_data),
.vip_pipeline_cvi_clocked_video_vid_de  (dp_rx_vid_valid),
.vip_pipeline_cvi_clocked_video_vid_datavalid (vid_datavalid), //
.vip_pipeline_cvi_clocked_video_vid_locked (dp_rx_vid_locked),
.vip_pipeline_cvi_clocked_video_vid_f     (dp_rx_vid_field),
.vip_pipeline_cvi_clocked_video_vid_v_sync (vid_v_sync), //
.vip_pipeline_cvi_clocked_video_vid_h_sync (vid_h_sync), //
.vip_pipeline_cvi_clocked_video_vid_color_encoding (8'h00), // input, w
.vip_pipeline_cvi_clocked_video_vid_bit_width (8'h00), // input,
.vip_pipeline_cvi_clocked_video_sof         (), // ou
.vip_pipeline_cvi_clocked_video_sof_locked (), // output,
.vip_pipeline_cvi_clocked_video_refclk_div  (), // output,
.vip_pipeline_cvi_clocked_video_clipping   (), // output,
.vip_pipeline_cvi_clocked_video_padding    (), // output
.vip_pipeline_cvi_clocked_video_overflow   (), // output,
// Clocked Video Output
```

6. Confirm the **cvo** and **dp_tx** connections as follows:


```
// Clocked Video Output
.vip_pipeline_cvo_clocked_video_vid_clk      (tx_vid_clk_freerun),
.vip_pipeline_cvo_clocked_video_vid_data    (tx_vid_data),          //
.vip_pipeline_cvo_clocked_video_underflow   (),                    // output,
.vip_pipeline_cvo_clocked_video_vid_mode_change (), // output, width
.vip_pipeline_cvo_clocked_video_vid_std     (),                    // output,
.vip_pipeline_cvo_clocked_video_vid_datavalid (tx_vid_de),        // output
.vip_pipeline_cvo_clocked_video_vid_v_sync  (tx_vid_vsync),        //
.vip_pipeline_cvo_clocked_video_vid_h_sync  (tx_vid_hsync),        //
.vip_pipeline_cvo_clocked_video_vid_f       (),                    // output
.vip_pipeline_cvo_clocked_video_vid_h       (),                    // output
.vip_pipeline_cvo_clocked_video_vid_v       (),                    // output
);

// TX Video Signal Interface
.dp_tx_dp_source_tx_vid_clk      (tx_vid_clk_freerun), // 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
```

TX_PIXELS_PER_CLOCK = 1 in this design

- Let's integrate now the generated DDR3 memory controller.
Open the `<project_dir>/rtl/core/dp_core/dp_core_inst.v` file and look for the new ports added related to the **ddr3 emif**. Select and copy them.

```
dp_core u0 (
.clk_100_in_clk      (_connected_to_clk_100_in_clk_),
.cpu_dummy_ci_port   (_connected_to_cpu_dummy_ci_port_),
.cpu_reset_bridge_in_reset_n (_connected_to_cpu_reset_bridge_in_reset_n),
.dp_rx_clk_16_in_clk (_connected_to_dp_rx_clk_16_in_clk_),
.dp_rx_reset_bridge_in_reset_n (_connected_to_dp_rx_reset_bridge_in_reset_n),
.dp_tx_clk_16_in_clk (_connected_to_dp_tx_clk_16_in_clk_),
.dp_tx_reset_bridge_in_reset_n (_connected_to_dp_tx_reset_bridge_in_reset_n),
.emif_c10_ddr3_global_reset_n_reset_n (_connected_to_emif_c10_ddr3_global_reset_n_reset_n),
.emif_c10_ddr3_pll_ref_clk_clk (_connected_to_emif_c10_ddr3_pll_ref_clk_clk_),
.emif_c10_ddr3_oct_oct_rzqin (_connected_to_emif_c10_ddr3_oct_oct_rzqin_),
.emif_c10_ddr3_mem_mem_ck (_connected_to_emif_c10_ddr3_mem_mem_ck_),
.emif_c10_ddr3_mem_mem_ck_n (_connected_to_emif_c10_ddr3_mem_mem_ck_n_),
.emif_c10_ddr3_mem_mem_a (_connected_to_emif_c10_ddr3_mem_mem_a_),
.emif_c10_ddr3_mem_mem_ba (_connected_to_emif_c10_ddr3_mem_mem_ba_),
.emif_c10_ddr3_mem_mem_cke (_connected_to_emif_c10_ddr3_mem_mem_cke_),
.emif_c10_ddr3_mem_mem_cs_n (_connected_to_emif_c10_ddr3_mem_mem_cs_n_),
.emif_c10_ddr3_mem_mem_odt (_connected_to_emif_c10_ddr3_mem_mem_odt_),
.emif_c10_ddr3_mem_mem_reset_n (_connected_to_emif_c10_ddr3_mem_mem_reset_n_),
.emif_c10_ddr3_mem_mem_we_n (_connected_to_emif_c10_ddr3_mem_mem_we_n_),
.emif_c10_ddr3_mem_mem_ras_n (_connected_to_emif_c10_ddr3_mem_mem_ras_n_),
.emif_c10_ddr3_mem_mem_cas_n (_connected_to_emif_c10_ddr3_mem_mem_cas_n_),
.emif_c10_ddr3_mem_mem_dqs (_connected_to_emif_c10_ddr3_mem_mem_dqs_),
.emif_c10_ddr3_mem_mem_dqs_n (_connected_to_emif_c10_ddr3_mem_mem_dqs_n_),
.emif_c10_ddr3_mem_mem_dq (_connected_to_emif_c10_ddr3_mem_mem_dq_),
.emif_c10_ddr3_mem_mem_dm (_connected_to_emif_c10_ddr3_mem_mem_dm_),
.emif_c10_ddr3_status_local_cal_success (_connected_to_emif_c10_ddr3_status_local_cal_success_),
.emif_c10_ddr3_status_local_cal_fail (_connected_to_emif_c10_ddr3_status_local_cal_fail_),
.i2c_master_sda_in (_connected_to_i2c_master_sda_in_),
.i2c_master_scl_in (_connected_to_i2c_master_scl_in_),
```

Open the top level `c10_dp_demo.v` file and paste the new ports somewhere in the **dp_core** instantiation. I have added them just on top of the `vip_pipeline` signals added before. Add some comments to your code

```
.vip_tx_dp_source_tx_cdc_busy          (vip_tx_cdc_busy),
.dp_tx_dp_source_tx_std_clkout        (gxb_tx_clkout),
.dp_tx_dp_source_tx_pll_locked        (dp_txpll_locked),
// -----
// DDR3 EMIF
// -----
.emif_c10_ddr3_global_reset_n_reset_n (connected_to_emif_c10_ddr3_global_reset_n_reset_n),
.emif_c10_ddr3_pll_ref_clk_clk        (connected_to_emif_c10_ddr3_pll_ref_clk_clk),
.emif_c10_ddr3_oct_oct_rzqin          (connected_to_emif_c10_ddr3_oct_oct_rzqin),
.emif_c10_ddr3_mem_mem_ck             (connected_to_emif_c10_ddr3_mem_mem_ck),
.emif_c10_ddr3_mem_mem_ck_n           (connected_to_emif_c10_ddr3_mem_mem_ck_n),
.emif_c10_ddr3_mem_mem_a              (connected_to_emif_c10_ddr3_mem_mem_a),
.emif_c10_ddr3_mem_mem_ba             (connected_to_emif_c10_ddr3_mem_mem_ba),
.emif_c10_ddr3_mem_mem_cke            (connected_to_emif_c10_ddr3_mem_mem_cke),
.emif_c10_ddr3_mem_mem_cs_n           (connected_to_emif_c10_ddr3_mem_mem_cs_n),
.emif_c10_ddr3_mem_mem_odt            (connected_to_emif_c10_ddr3_mem_mem_odt),
.emif_c10_ddr3_mem_mem_reset_n        (connected_to_emif_c10_ddr3_mem_mem_reset_n),
.emif_c10_ddr3_mem_mem_we_n           (connected_to_emif_c10_ddr3_mem_mem_we_n),
.emif_c10_ddr3_mem_mem_ras_n          (connected_to_emif_c10_ddr3_mem_mem_ras_n),
.emif_c10_ddr3_mem_mem_cas_n          (connected_to_emif_c10_ddr3_mem_mem_cas_n),
.emif_c10_ddr3_mem_mem_dqs            (connected_to_emif_c10_ddr3_mem_mem_dqs),
.emif_c10_ddr3_mem_mem_dqs_n          (connected_to_emif_c10_ddr3_mem_mem_dqs_n),
.emif_c10_ddr3_mem_mem_dq             (connected_to_emif_c10_ddr3_mem_mem_dq),
.emif_c10_ddr3_mem_mem_dm             (connected_to_emif_c10_ddr3_mem_mem_dm),
.emif_c10_ddr3_status_local_cal_success (connected_to_emif_c10_ddr3_status_local_cal_success),
.emif_c10_ddr3_status_local_cal_fail  (connected_to_emif_c10_ddr3_status_local_cal_fail),
// -----
// VIP Pipeline Sub-System
// -----
// Common Clock
.vip_pipeline_clock_vip_in_clk_clk    (dp_rx_vid_clk),          // input, width = 1, vip_f
// Clocked Video Input
.vip_pipeline_clk_vip_in_clk_clk      (connected_to_vip_pipeline_clk_vip_in_clk_clk)
```

8. Define top level pins to allow external connections from the FPGA to the DDR3 memory devices on the board. Go the `c10_dp_demo.v` module declaration, on top of the file, to add the mem ports after the existing ones:

```

//User-I0
input  wire [1:0]
output wire [3:0]

//DDR3
output wire [0:0]
output wire [0:0]
output wire [14:0]
output wire [2:0]
output wire [0:0]
output wire [0:0]
output wire [0:0]
output wire [0:0]
output wire [0:0]
output wire [0:0]
inout  wire [4:0]
inout  wire [4:0]
inout  wire [39:0]
output wire [4:0]
input  wire [0:0]

user_pb,
user_led_g,

mem_ck,
mem_ck_n,
mem_a,
mem_ba,
mem_cke,
mem_cs_n,
mem_odt,
mem_reset_n,
mem_we_n,
mem_ras_n,
mem_cas_n,
mem_dqs,
mem_dqs_n,
mem_dq,
mem_dm,
mem_oct_rzqin
);

```

9. We need to add an additional port to connect the reference clock for the memory pll

```

module c10_dp_demo
#(
    parameter RX_MAX_LANE_COUNT = 4,
    parameter TX_MAX_LANE_COUNT = 4
)
(
    //Clocks Inputs
    input wire          cpu_resetn,          //1.8V //CPU Reset Pushbutton (TR=0), assign to user_pb[2]
    input wire          refclk2_p,           //1.8V XCVR reference clock - 100MHz (Programmable Si570) or
    input wire          mem_pll_ref_clk,     //21.186MHz - Reference clock for DDR3

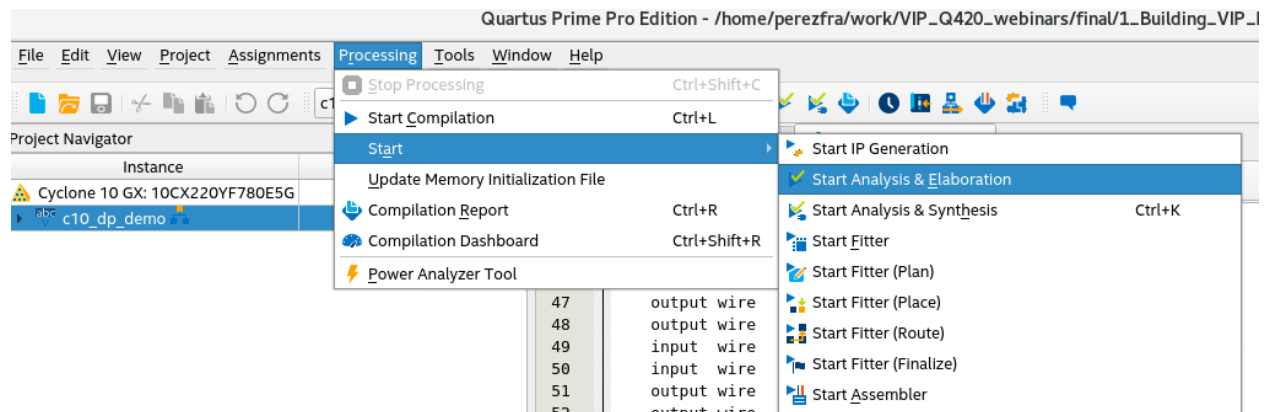
```

10. Make all the connections to the **ddr3_emif** module

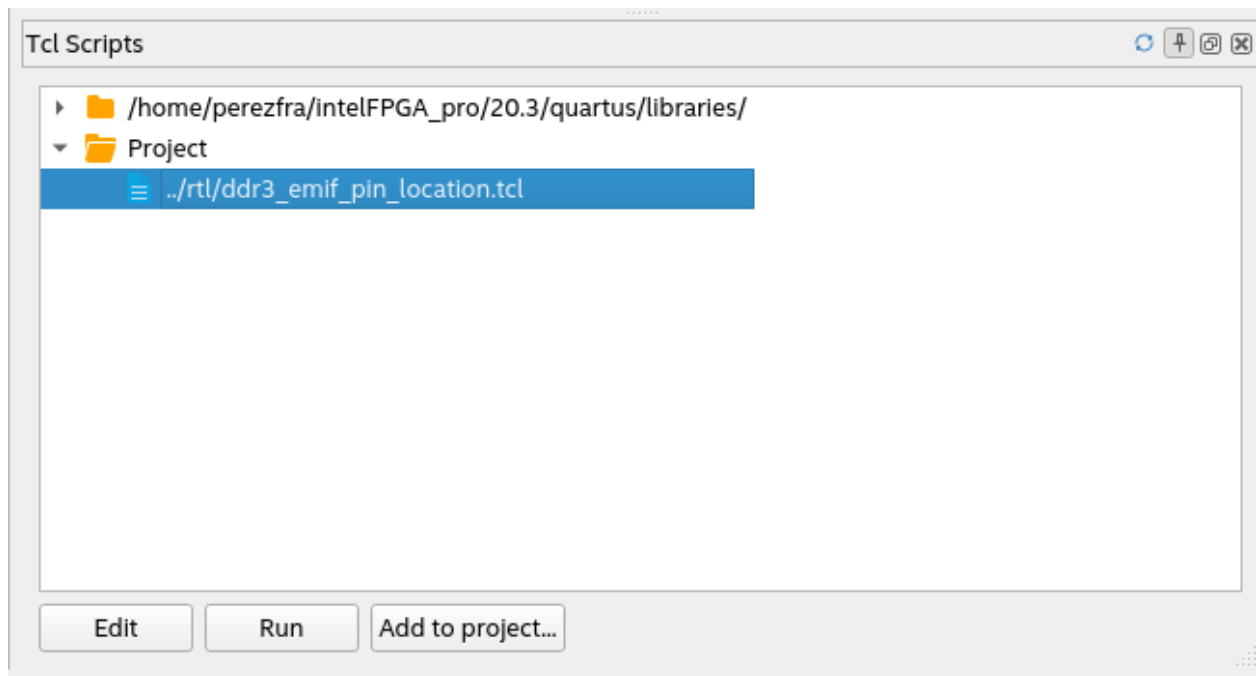
```
// -----
// DDR3 EMIF
// -----

.emif_c10_ddr3_global_reset_n_reset_n      (cpu_reset_n),
.emif_c10_ddr3_pll_ref_clk_clk              (mem_pll_ref_clk),
.emif_c10_ddr3_oct_oct_rzqin                (mem_oct_rzqin),
.emif_c10_ddr3_mem_mem_ck                   (mem_ck),
.emif_c10_ddr3_mem_mem_ck_n                 (mem_ck_n),
.emif_c10_ddr3_mem_mem_a                    (mem_a),
.emif_c10_ddr3_mem_mem_ba                   (mem_ba),
.emif_c10_ddr3_mem_mem_cke                  (mem_cke),
.emif_c10_ddr3_mem_mem_cs_n                 (mem_cs_n),
.emif_c10_ddr3_mem_mem_odt                  (mem_odt),
.emif_c10_ddr3_mem_mem_reset_n              (mem_reset_n),
.emif_c10_ddr3_mem_mem_we_n                 (mem_we_n),
.emif_c10_ddr3_mem_mem_ras_n                (mem_ras_n),
.emif_c10_ddr3_mem_mem_cas_n               (mem_cas_n),
.emif_c10_ddr3_mem_mem_dqs                  (mem_dqs),
.emif_c10_ddr3_mem_mem_dqs_n                (mem_dqs_n),
.emif_c10_ddr3_mem_mem_dq                   (mem_dq),
.emif_c10_ddr3_mem_mem_dm                   (mem_dm),
.emif_c10_ddr3_status_local_cal_success     (),
.emif_c10_ddr3_status_local_cal_fail        (),
```

11. Save `c10_dp_demo.v` file. In the main toolbar, select **Processing->Start->Start Analysis & Elaboration** to create an initial database to update the new pins added to the design.



12. We need now to assign all the pin locations to the signals driving the memory interface and the reference clock for the pll.
For this we have provided a tcl script `<project_dir>/rtl/ddr3_emif_pin_location.tcl` to make all these assignments.
In the main toolbar, go to **Tools->Tcl Scripts** to open the Tcl Scripts pane in Quartus.
Select **Project->ddr3_emif_pin_location.tcl** and click on the **Run** button



NOTE: We only need to assign pin locations, as all the other settings related to voltage standard and terminations, are automatically handled by other scripts added by the DDR3 EMIF IP core at generation time, as part of the IP variation file created.

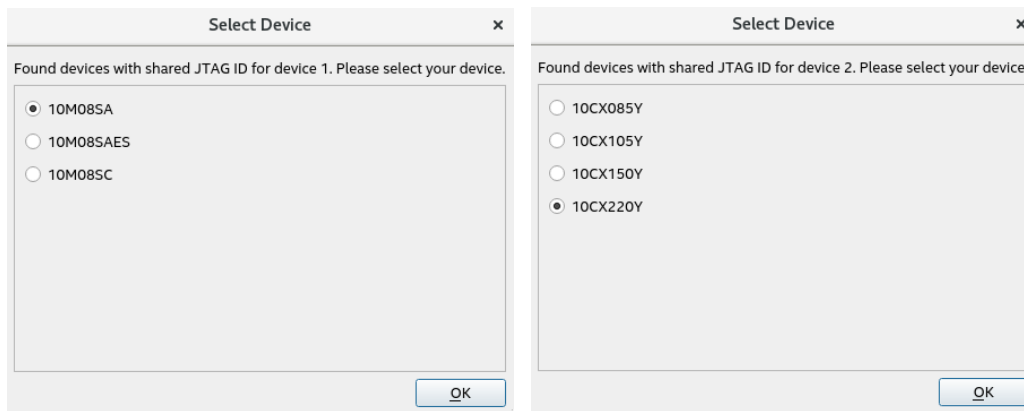
2.7. Generate the programming file

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 ~12 minutes, depending on machine configuration.

2.8. Configuring the FPGA device

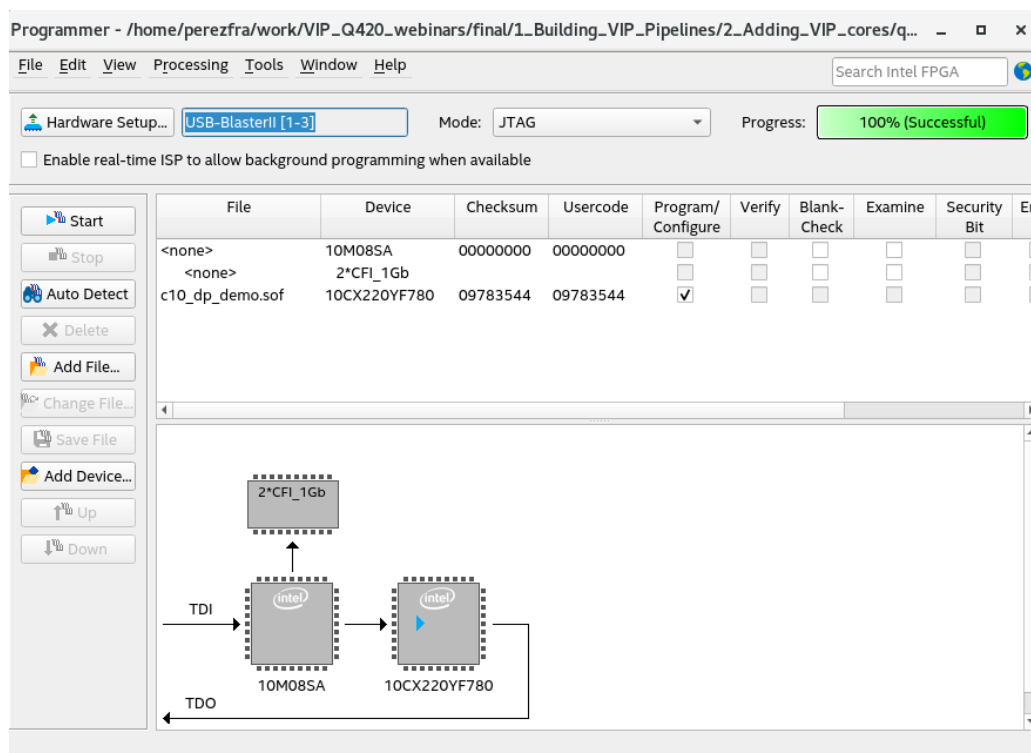
Open the **Quartus Programmer**, select your USB-Blaster cable in the Hardware Setup and click on **Auto Detect** to retrieve the JTAG chain on the Cyclone10 GX Devkit.

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**.

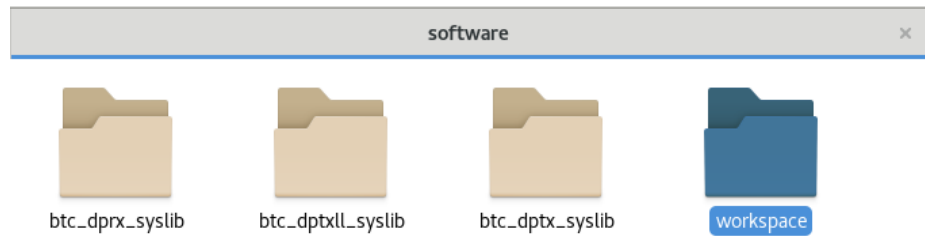


3. Building the software application

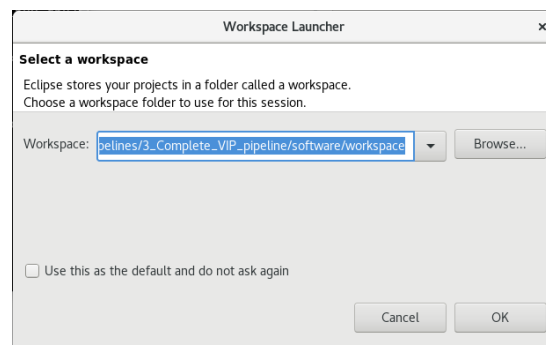
3.1. Setting up the Eclipse for Nios project

1. Creating Eclipse project for Nios II application and BSP.

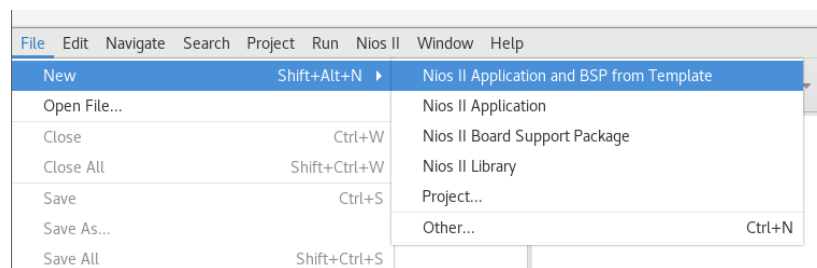
- Create a workspace folder in `<project_dir>/software`



- Launch `eclipse-nios2` from your terminal. When asked for a **Workspace**, select the folder you have just created in the previous step

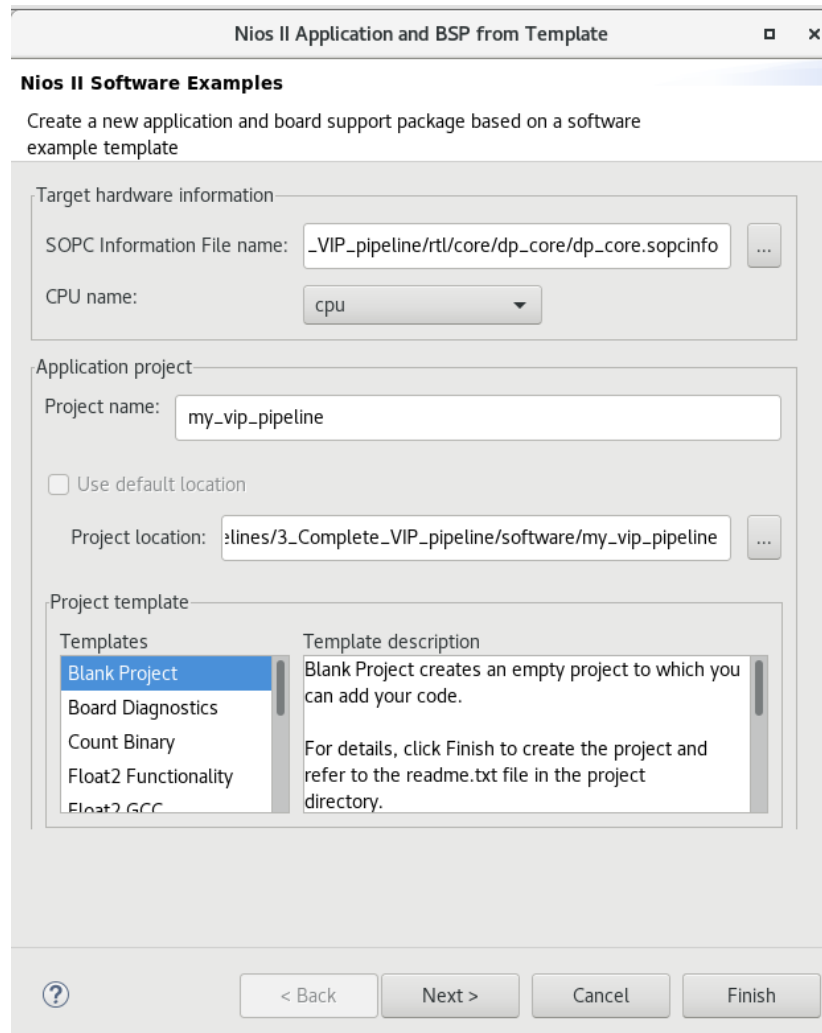


- Use **File->New->Nios II Application and BSP from Template** to create your new project

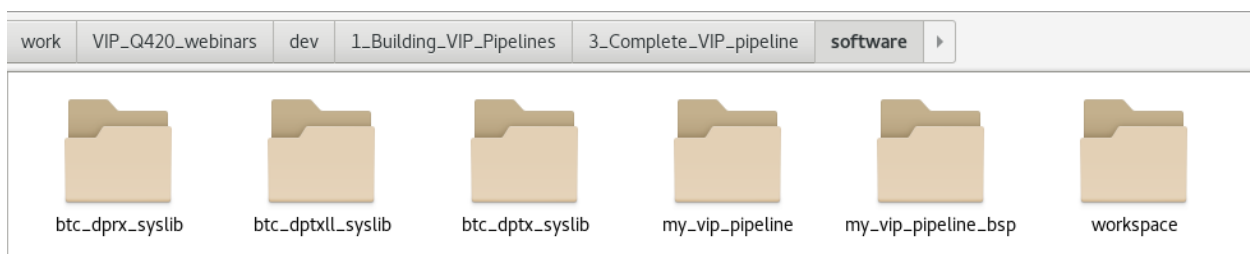


- Under **Target hardware information->SOPC Information File name**, you need to select the `*.sopcinfo` file that contains your Nios CPU. In our case is located in `<project_dir>/rtl/core/dp_core/dp_core.sopcinfo`
- Under **Application project->Project name** : `my_vip_pipeline`

- Make sure that **Project location** is set to `<project_dir>/software/my_vip_pipeline`, by default gets located at `<project_dir>/rtl/core/software/my_dp_demo`
- Select **Blank Project** as **Templates** and Click **Finish**

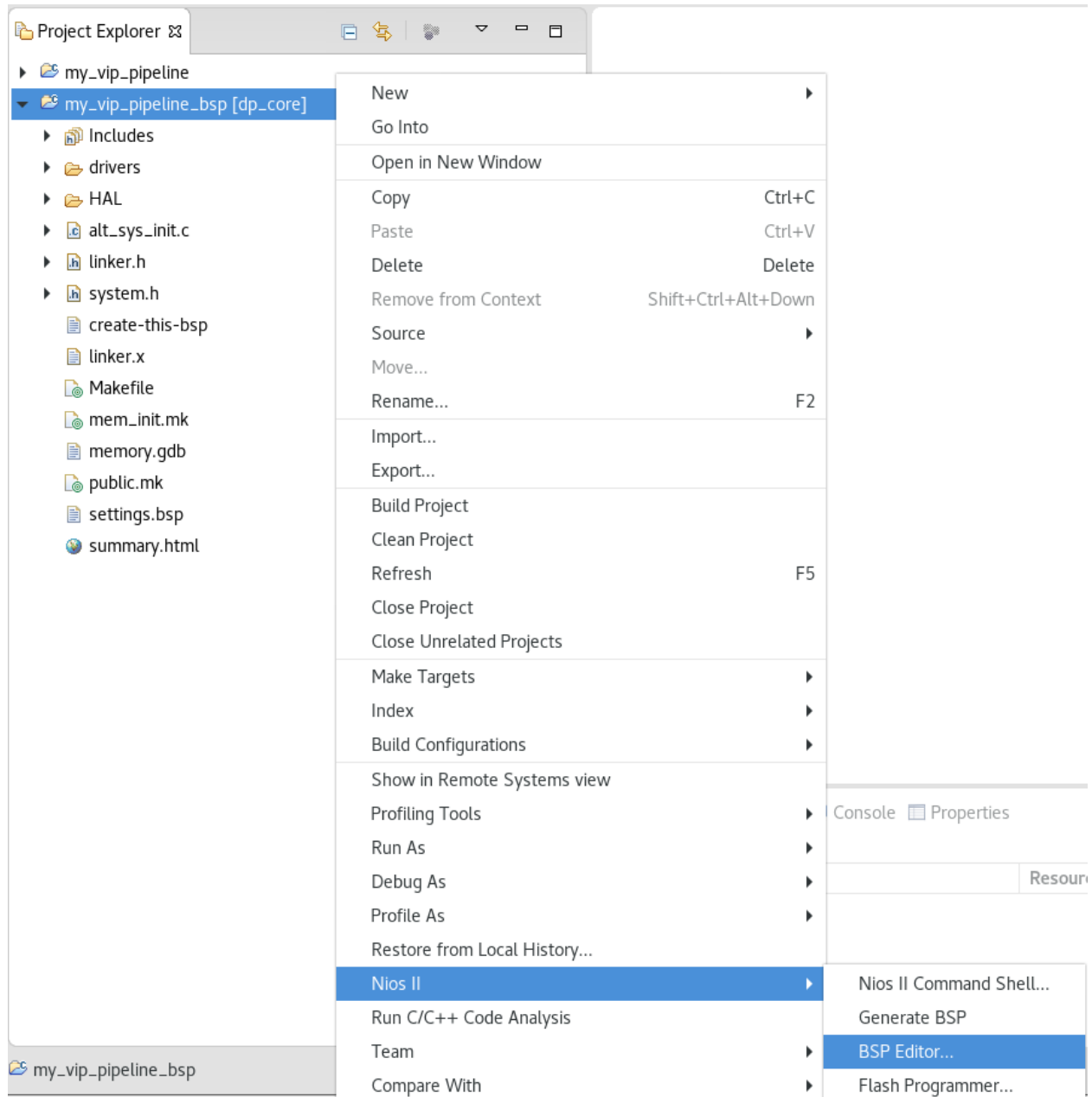


You will end up in a software folder structure as follows:

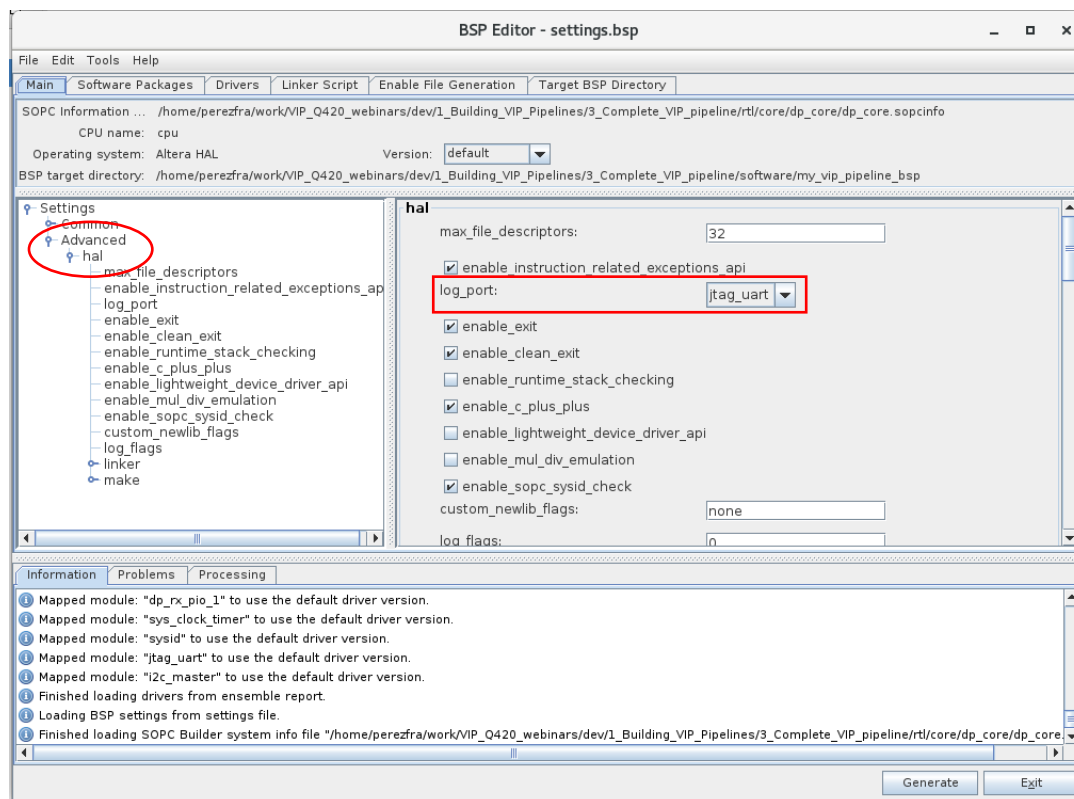
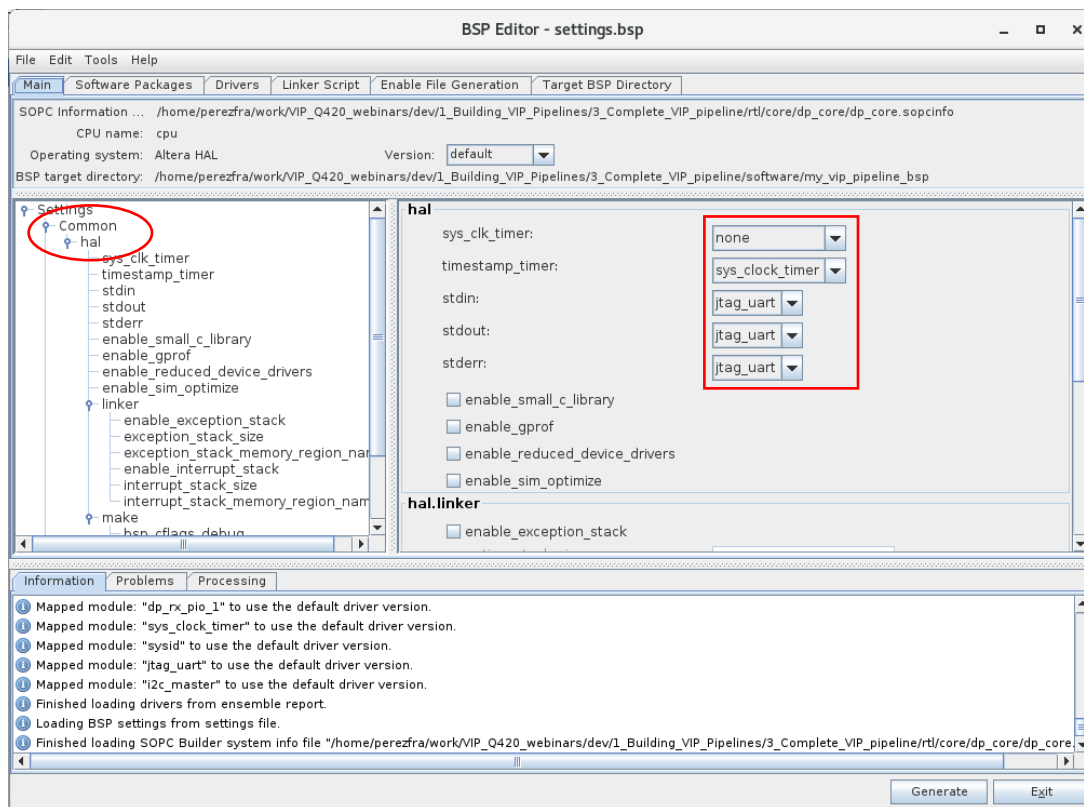


2. Configure the bsp.

In eclipse, right-click on **Project Explorer->my_vip_pipeline_bsp** and select **Nios II->BSP Editor**



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



Please note that in the **Drivers** tab, the bsp generation has already included supporting code for the all VIP components we are using: CVI, CLP, CVO, MIXER, SCL y TPG.

BSP Editor - settings.bsp

File Edit Tools Help

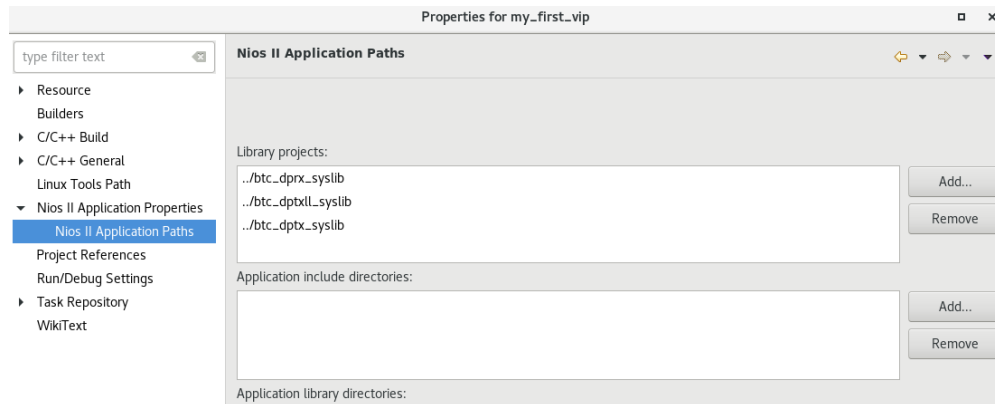
Main Software Packages Drivers Linker Script Enable File Generation Target BSP Directory

| 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_clp | alt_vip_cl_clp | 20.3.0 | alt_vip_cl_clp_driver | default | <input checked="" type="checkbox"/> |
| vip_pipeline_cvi | alt_vip_cl_cvi | 20.3.0 | alt_vip_cl_cvi_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_mixer | alt_vip_cl_mixer | 20.3.0 | alt_vip_cl_mixer_driver | default | <input checked="" type="checkbox"/> |
| vip_pipeline_scl | alt_vip_cl_scl | 20.3.0 | alt_vip_cl_scl_driver | default | <input checked="" type="checkbox"/> |
| vip_pipeline_tpg_0 | alt_vip_cl_tpg | 20.3.0 | alt_vip_cl_tpg_driver | default | <input checked="" type="checkbox"/> |

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

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

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



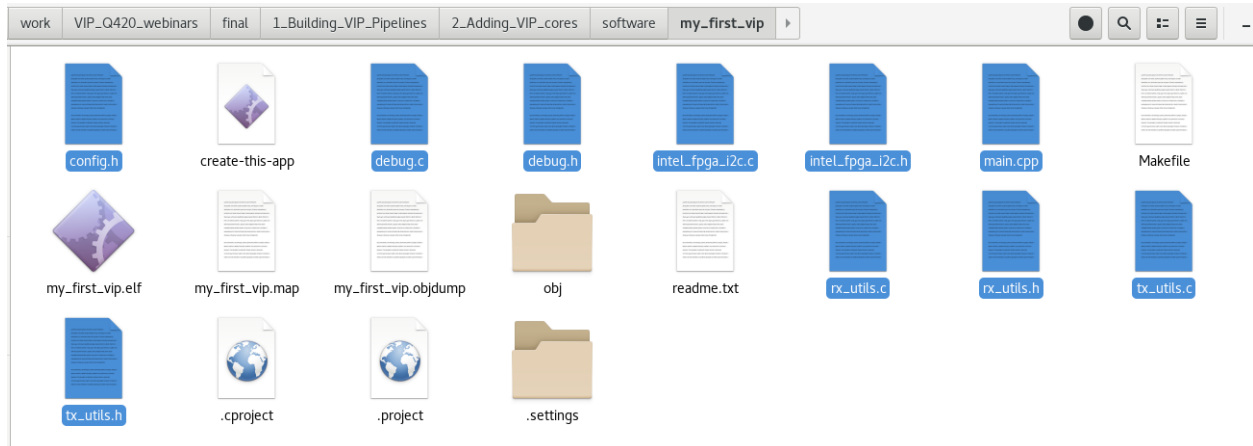
Then click **Apply** and **OK**

3.2.Importing the code

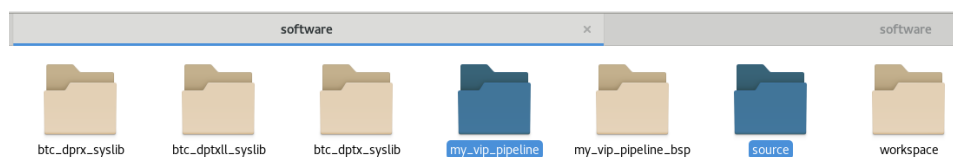
We are reusing all the code we modified in the previous lab “Session 1.2 – Adding VIP cores”. This code has been already adapted to a C++ application and ready to use. What we are doing here is expanding the capabilities for the new cores.

Copy all the source files in

`<session1.2_adding_vip_cores_project_dir>/software/my_first_vip` into the newly created `<project_dir>/software` folder where our application resides.

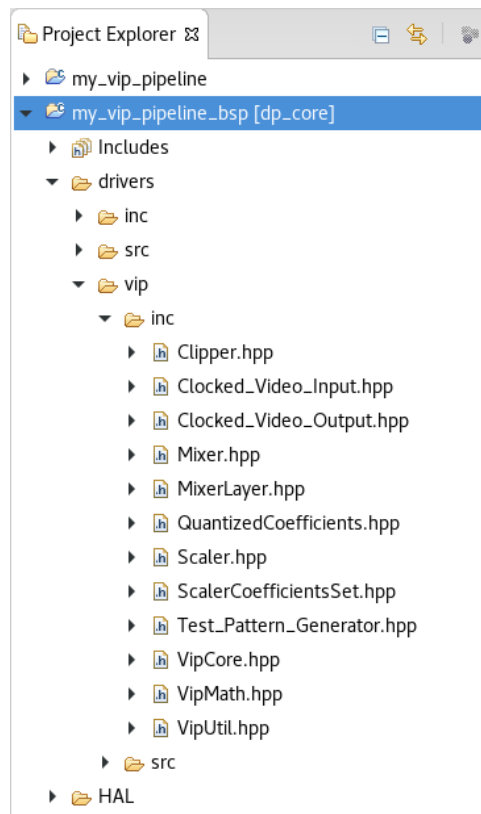


NOTE: If you decide to use the files provided in [1_3_Complete_VIP_pipeline.tar.gz](#) package, you can copy the source files provided in the folder `<project_dir>/software/source` to `<project_dir>/software/my_vip_pipeline` and you can jump directly to the point **3.5. Building and launching program execution**, as all the modifications have been done already.



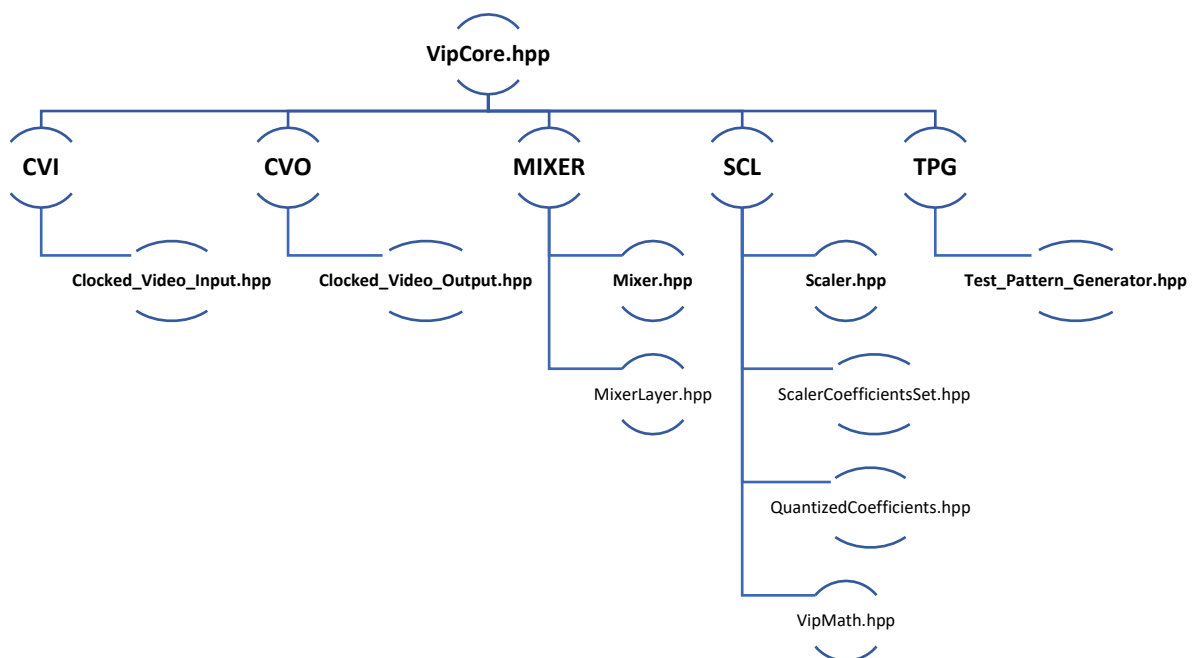
3.3. Getting familiar with VIP C++ API

If we expand the generated `my_vip_pipeline_bsp` in the **Project Explorer**, we can notice that a new set of drivers have been added to control the VIP cores.



Each of the VIP cores has its own *.hpp header file with proprietary methods for the class. Some of the classes use auxiliary header files, like the MIXER and SCALER. All of them have in common VipCore.hpp and VipUtil.hpp which are used to declare generic methods and members.

All VIP classes are inherited from **VipCore**.



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 | Register | Description |
|---------|---------------------------------|---|
| 0 | Bit 0 = Go Bits 31:1 = X | Bit zero of this register is the Go bit, all other bits are unused. Setting Go bit to 0 will cause the core to pause at the end of the frame. Setting Go bit to 1 will resume the core. |
| 1 | Bit 0 = Status Bits 31:1 = X | Bit zero of this register is the Status bit, all other bits are unused. The core sets the Status bit to 1 when it is running, and zero otherwise. |
| 2 | reg 0 | Interrupt register (core specific: check VIP user guide) |
| ... | ... | ... |
| n+1 | reg n-1 | Function dependent (check VIP user guide) |

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.

The first two registers of every control interface have the same functions. The others vary with each core and control interface.

Register **0** is the **Go** register. Bit zero of this register is the **Go** bit. Most VIP cores stop at the beginning of the video frame data packet if the **Go** bit is set to **0**. This allows you to stop the core to program run-time control data before the processing of the video frame begins. A few cycles after the **Go** bit is set, the core begins processing video data. If the **Go** bit is de-asserted while the data is being processed, then the core function stops processing at the beginning of the next video frame data packet and waits until the **Go** bit is asserted.

Register **1** is the **Status** register. Bit zero of this register is the Status bit. The VIP core sets the Status bit to 1 when it is running, zero otherwise. This register can be polled by an external processor or state machine control logic to determine the state of the core.

○ 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()

```
#ifndef __VIP_CORE_HPP__
#define __VIP_CORE_HPP__

#include <system.h>
#include <io.h>
#include <sys/alt_irq.h>
#include <cassert>

class VipCore
{
public:
    enum {
        REGISTER_CONTROL    = 0,
        REGISTER_STATUS      = 1,
        REGISTER_INTERRUPT    = 2
    };

    inline unsigned long get_base_address() const
    inline void start()
    inline void stop()
    inline bool is_running() const

    inline void enable_interrupt(unsigned int interrupt_number)
    inline void disable_interrupt(unsigned int interrupt_number)
    inline bool has_interrupt_fired(unsigned int interrupt_number) const
    inline void clear_interrupt(unsigned int interrupt_number)

    inline void write_control_register(unsigned int new_control_reg)
    inline unsigned int read_status_register() const
    inline void do_write(unsigned int offset, unsigned int value)
    inline unsigned int do_read(unsigned int offset) const
    .
    .
    .
#endif // __VIP_CORE_HPP__
```

Generic functions

Manage interrupts

Access core registers

Let's now explore a couple of VIP cores to see how each has its own members and functions, but all of them leverage the base class Vipcore.

○ Clocked_Video_Output.hpp

```
#ifndef __CLOCKED_VIDEO_OUTPUT_HPP__
#define __CLOCKED_VIDEO_OUTPUT_HPP__

#include "VipCore.hpp"

// Some convenient defines to use when calling the set_output_mode function
//
// interl, width, height, fl_h, h blanking, v blanking, f0 blanking, active_line, field_
#define CVO_720P_MODE false, 1280, 720, 0, 110, 40, 370, 5, 5, 30, 0, 0, 0, 26, 0, ...
#define CVO_1080I_MODE true, 1920, 540, 540, 88, 44, 280, 2, 5, 22, 2, 5, 23, 22, 562, ...
#define CVO_1080I_SDI_MODE true, 1920, 540, 540, 0, 0, 280, 0, 0, 22, 0, 0, 23, 21, 561, ...
#define CVO_1080P_MODE false, 1920, 1080, 0, 88, 44, 280, 4, 5, 45, 0, 0, 0, 42, 0, ...
#define CVO_2160P_MODE false, 3840, 2160, 0, 176, 88, 560, 8, 10, 90, 0, 0, 0, 84, 0, ...
```

```

class Clocked_Video_Output : public VipCore {

public:

// CVO specific registers
enum CVORegisterType {
    CVO_VIDEO_MODE_MATCH           = 3,
    CVO_BANK_SELECT                = 4,
    CVO_MODEX_CONTROL              = 5,
    CVO_MODEX_SAMPLE_COUNT         = 6,
    CVO_MODEX_F0_LINE_COUNT        = 7,
    CVO_MODEX_F1_LINE_COUNT        = 8,
    CVO_MODEX_HORIZONTAL_FRONT_PORCH = 9,
    CVO_MODEX_HORIZONTAL_SYNC_LENGTH = 10,
    CVO_MODEX_HORIZONTAL_BLANKING   = 11,
    CVO_MODEX_VERTICAL_FRONT_PORCH  = 12,
    CVO_MODEX_VERTICAL_SYNC_LENGTH  = 13,
    CVO_MODEX_VERTICAL_BLANKING     = 14,
    CVO_MODEX_F0_VERTICAL_FRONT_PORCH = 15,
    CVO_MODEX_F0_VERTICAL_SYNC_LENGTH = 16,
    CVO_MODEX_F0_VERTICAL_BLANKING   = 17,
    CVO_MODEX_ACTIVE_PICTURE_LINE   = 18,
    CVO_MODEX_F0_VERTICAL_RISING    = 19,
    CVO_MODEX_FIELD_RISING          = 20,
    CVO_MODEX_FIELD_FALLING         = 21,
    CVO_MODEX_STANDARD              = 22,
    CVO_MODEX_SOF_SAMPLE            = 23,
    CVO_MODEX_SOF_LINE              = 24,
    CVO_MODEX_VCO_CLK_DIVIDER       = 25,
    CVO_MODEX Ancillary_Line        = 26,
    CVO_MODEX_F0_Ancillary_Line     = 27,
    CVO_MODEX_HSYNC_POLARITY        = 28,
    CVO_MODEX_VSYNC_POLARITY        = 29,
    CVO_MODEX_VALID                 = 30,
};

Clocked_Video_Output(unsigned long base_address, int irq_number = -1);

inline bool is_producing_data() const
inline bool is_underflowed() const
inline void clear_underflow_flag()
inline bool is_frame_locked() const

void set_output_mode(unsigned int bank_sel, bool interlaced, bool sequential, unsigned
    int sample_count, unsigned int f0_line_count, unsigned int f1_line_count, unsigned int
    h_front_porch, unsigned int h_sync_length, unsigned int h_blanking, unsigned int
    v_front_porch, unsigned int v_sync_length, unsigned int v_blanking, unsigned int
    f0_v_front_porch, unsigned int f0_v_sync_length, unsigned int f0_v_blanking, unsigned
    int active_picture_line, unsigned int f0_v_rising, unsigned int field_rising, unsigned
    int field_falling, unsigned int ancillary_line, unsigned int f0_ancillary_line, bool
    h_sync_polarity, bool v_sync_polarity, unsigned int vid_std = 0, unsigned int
    sof_sample = 0, unsigned int sof_line = 0, unsigned int vco_clk_div = 0);

#endif // __CLOCKED_VIDEO_OUTPUT_HPP__

```

○ Clipper.hpp

```

#ifndef __CLIPPER_HPP__
#define __CLIPPER_HPP__

#include "VipCore.hpp"

class Clipper : public VipCore {
public:
    // Clipper specific registers
    enum ClipperRegisterType {
        CLP_LEFT_OFFSET   = 3,
        CLP_RIGHT_OFFSET  = 4,
        CLP_TOP_OFFSET     = 5,
        CLP_BOTTOM_OFFSET  = 6,
    };

```



```

// In "rectangle" mode, registers 4 and 6 are now the width and
// height instead of right offset and bottom offset
CLP_WIDTH      = 4,
CLP_HEIGHT     = 6,
};

Clipper(const long base_address, bool rectangle_mode);

inline void set_left_offset(unsigned int left_offset) {
    do_write(CLP_LEFT_OFFSET, left_offset);
}

inline void set_right_offset(unsigned int right_offset) {
    assert(!rectangle_mode);
    do_write(CLP_RIGHT_OFFSET, right_offset);
}

inline void set_top_offset(unsigned int top_offset) {
    do_write(CLP_TOP_OFFSET, top_offset);
}

inline void set_bottom_offset(unsigned int bottom_offset) {
    assert(!rectangle_mode);
    do_write(CLP_BOTTOM_OFFSET, bottom_offset);
}

inline void set_width(unsigned int width){
    assert(rectangle_mode);
    do_write(CLP_WIDTH, width);
}

inline void set_height(unsigned int height) {
    assert(rectangle_mode);
    do_write(CLP_HEIGHT, height);
}

void set_offsets(unsigned int left_offset, unsigned int right_offset,
                unsigned int top_offset, unsigned int bottom_offset);
void set_rectangle(unsigned int left_offset, unsigned int top_offset,
                  unsigned int width, unsigned int height);

#endif // __CLIPPER_HPP__

```

3.4. 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

```

// -----
// Include VIP header files
// -----
#include "Clocked_Video_Output.hpp"
#include "Test_Pattern_Generator.hpp"
#include "Mixer.hpp"
#include "Scaler.hpp"
#include "Clocked_Video_Input.hpp"
#include "Clipper.hpp"

```

- **Declare some run time variables.** Inside *main()* function, declare some global variables we can use to hold some information read back from the cores.

```

// -----
// Declare run-time variables
// -----
int active_lines=0;
int active_pixels=0;
bool input_stable=0;

```

```
bool input_overflow=0;
```

- **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

<project_dir>/software/my_vip_pipeline_bsp/system.h, located in the bsp folder.

Base addresses extracted from system.h

```
#define VIP_PIPELINE_CLP_BASE 0x8c0
#define VIP_PIPELINE_CVI_BASE 0x800
#define VIP_PIPELINE_CVO_BASE 0x0
#define VIP_PIPELINE_MIXER_BASE 0x400
#define VIP_PIPELINE_SCL_BASE 0x600
#define VIP_PIPELINE_TPG_0_BASE 0x880
```

Create objects to handle the VIP cores in our application. In addition to create the VIP cores base objects, we have created a couple of “ScalerCoefficientsSet” objects we will use to calculate and store, run time Scaler coefficients according to input/output resolution ratio.

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

ClockVideoOutput cvo(VIP_PIPELINE_CVO_BASE);
TestPatternGenerator tpg(VIP_PIPELINE_TPG_0_BASE);
Mixer mixer(VIP_PIPELINE_MIXER_BASE, 3);
Scaler scaler(VIP_PIPELINE_SCL_BASE, 8, 8, 16, 16, 1, 7, 1, 7);
Clipper clipper(VIP_PIPELINE_CLP_BASE, true);
ClockVideoInput cvi(VIP_PIPELINE_CVI_BASE);

//-- Scaler coefficients
ScalerCoefficientsSet scalard_hcoeff_setup(16, 8);
ScalerCoefficientsSet scalard_vcoeff_setup(16, 8);
```

- **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.
It's recommended to trigger VIP execution from the last VIP core backwards to the beginning to avoid CVO overflow. So, the ideal sequence should be:

CVO -> MIXER -> TPG (background) -> SCL->CLP->CVI

```
// -----
// Initialize objects and start execution
// -----

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

mixer[0].set_offset(0, 0);
mixer[0].enable_layer();
mixer[1].set_offset(0, 0);
mixer[1].disable_layer();
mixer.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();

scalard_hcoeff_setup.lanczos_generate(1920, 1280);
scalard_vcoeff_setup.lanczos_generate(1080, 800);
scaler.apply(scalard_hcoeff_setup, scalard_vcoeff_setup, 0, 0);
scaler.set_output_resolution(1280, 800);
scaler.start();

clipper.set_rectangle(0,0,1920,1080);
clipper.start();

```

But, CVI is not yet initialized, why? and in the mixer the layer [1], connected to the CVI is disabled.

With this configuration, we will only see on the screen the color bars pattern generated by the TPG as background. Let's add some logic to the program to detect when we have a stable and known resolution applied to our DisplayPort input (capture by CVI) and enable the MIXER layer [1] when we have a correct signal and disable otherwise.

Let's find the main loop (while(1)) in the main function and add the following code:

```

// -----
// VIP run time control of CVI and Mixer layer
// -----

if(IORD(btc_dprx_baseaddr(0), DPRX0_REG_VBID) & 0x80)
{
    active_lines = cvi.get_active_line_count_f0();
    active_pixels = cvi.get_active_sample_count();
    input_stable = cvi.is_locked();
    if (input_stable)
    {
        input_overflow = cvi.is_overflowed();
        if (input_overflow)
        {
            cvi.clear_overflow_flag();
        }
        else
        {
            cvi.start();
            mixer[1].enable_layer(); // MSA lock -> Enable DP image
        }
    }
}
else
{
    mixer[1].disable_layer(); // MSA not locked -> Disable DP image
    cvi.stop();
    input_stable = cvi.is_locked();
}

```

With the following instruction we are reading, from the DisplayPort RX core, whether we have connected a stable and known resolution video at the input.

```
if(IORD(btc_dprx_baseaddr(0), DPRX0_REG_VBID) & 0x80)
```

The **DPRX0_REG_VBID** register is declared in

<project_dir>/software/btc_dprx_syslib/btc_dp_rxregs.h as

```
#define DPRX0_REG_VBID          0x2f
```

You can check in the [DisplayPort IP User Guide](#) that corresponds to **DPRX0_VBID**. This is a Sink MSA register in the DPCD address space.

11.4.16. DPRX0_VBID

VB-ID register, DPRX0_VBID.

Address: 0x002f

Direction: RO

Reset: 0x00000000

Table 147. DPRX0_VBID Bits

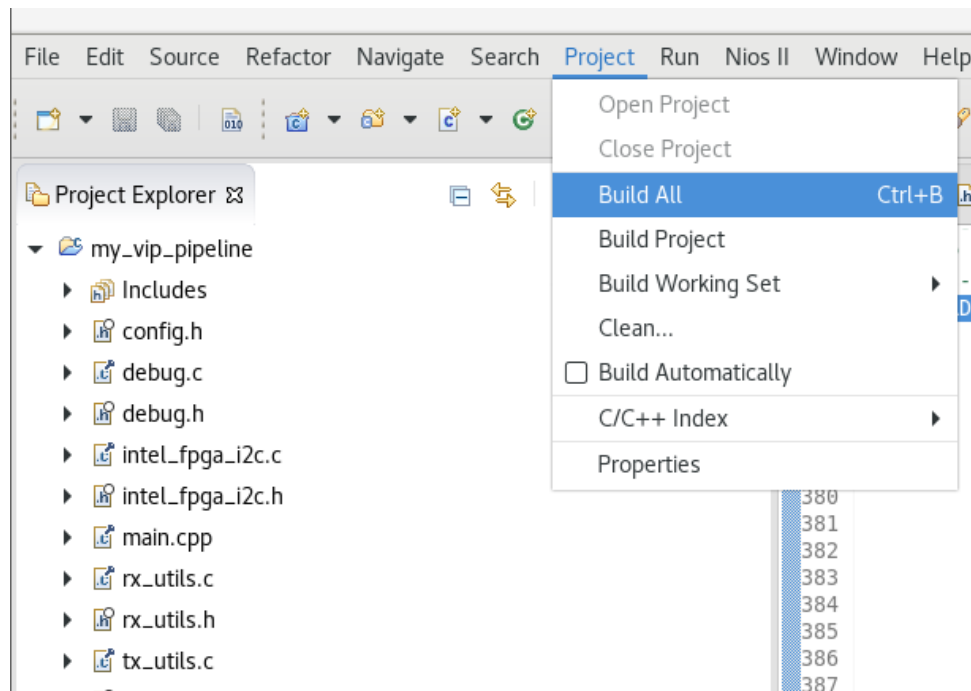
| Bit | Bit Name | Function |
|------|-----------|---|
| 31:8 | Unused | |
| 7 | MSA_LOCK | 0 = MSA unlocked 1 = MSA locked (on all lanes) |
| 6 | VBID_LOCK | 0 = VB-ID unlocked 1 = VB-ID locked (on all lanes) |
| 5:0 | VBID | VB-ID flags (refer to the <i>VESA DisplayPort Standard</i>). |

When **Bit 7 = 1** MSA is locked on all lanes to the input, so we have a stable and known resolution connected that we can process in our pipeline.

If we have MSA locked and CVI has detected a stable video input, we can proceed to start CVI processing and enable the layer in the mixer. Otherwise, we stop CVI from processing video and disable the mixer layer.

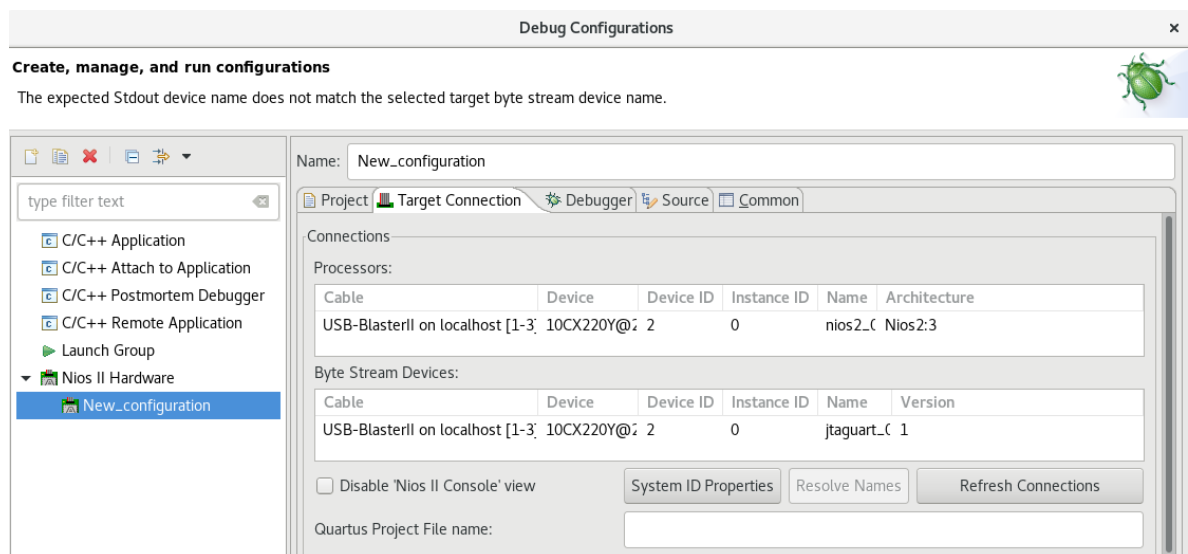
3.5. Building and launching program execution

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

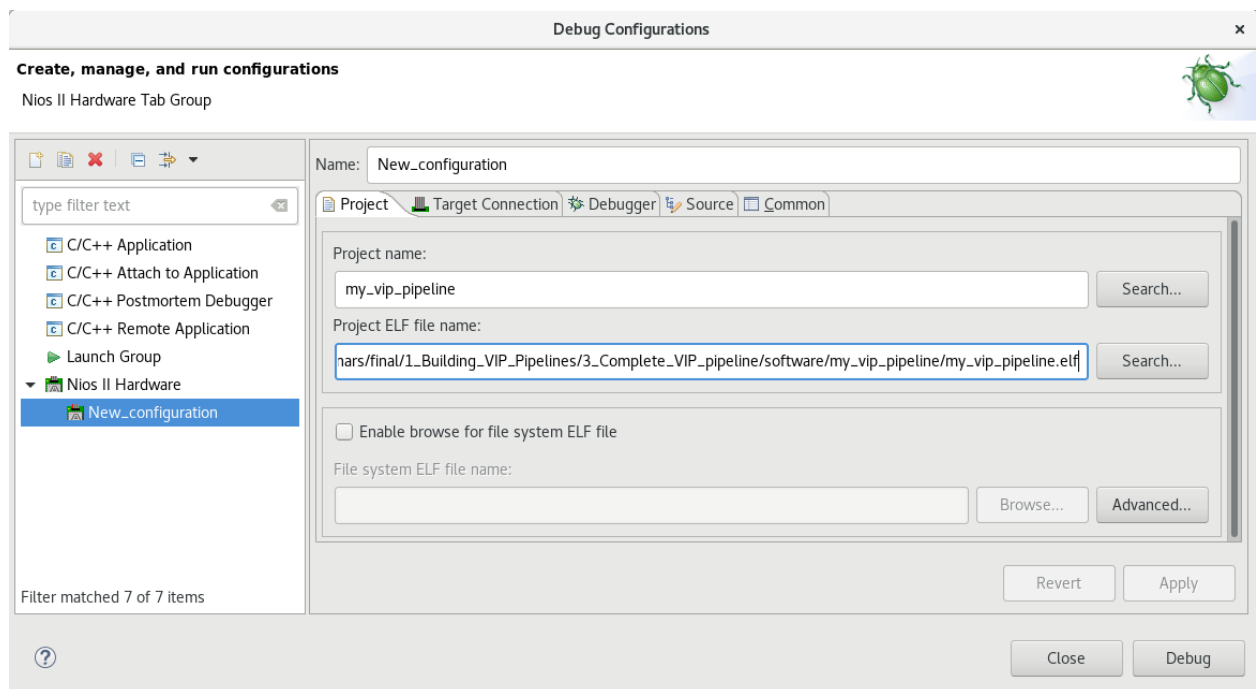


To launch our application, we can go to **Run->Debug Configurations...** in the main toolbar to open *Debug Configuration* dialog.

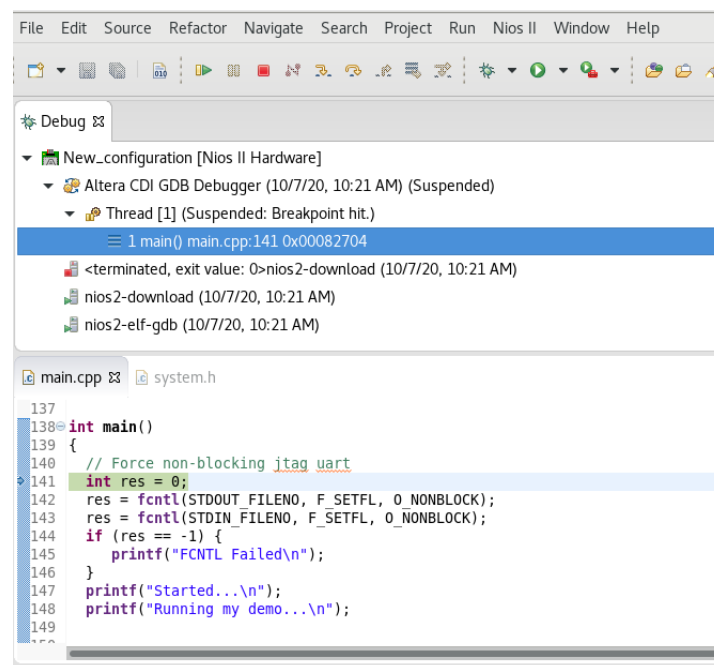
We double-click on **Nios II Hardware** option to create a *New_configuration*. In **Target Connection** tab, click on **Refresh Connections** to select the right *USB-BlasterII* adapter



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.



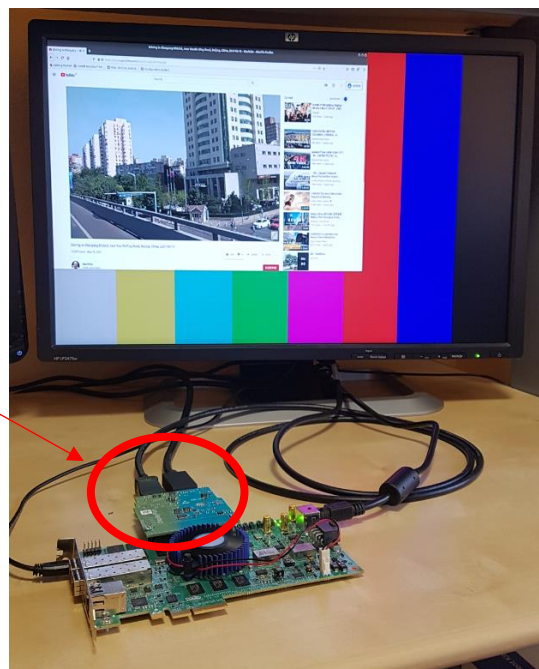
You should see now a nice Color Bars pattern displayed on your attached monitor when the input cable is disconnected.

Disconnected
DisplayPort input



And the live video coming from the computer, on a layer on top of the background, when the input video cable is attached.

Connected
DisplayPort input



4. Summary

In this lab, we have exercised with a complete video pipeline implementation using VIP cores.

- We have developed a Platform Designer subsystem with our video processing pipeline and we have integrated it with the rest of the control modules.
- We have connected the interfaces with the DisplayPort IP cores, building an End2End chain from video capturing, processing, mixing and displaying on an external monitor.
- We have learnt how to build a software application to control the VIP cores from a Nios II Processor.
- We have managed to apply runtime control capabilities to our video chain.
- More to come on upcoming sessions.