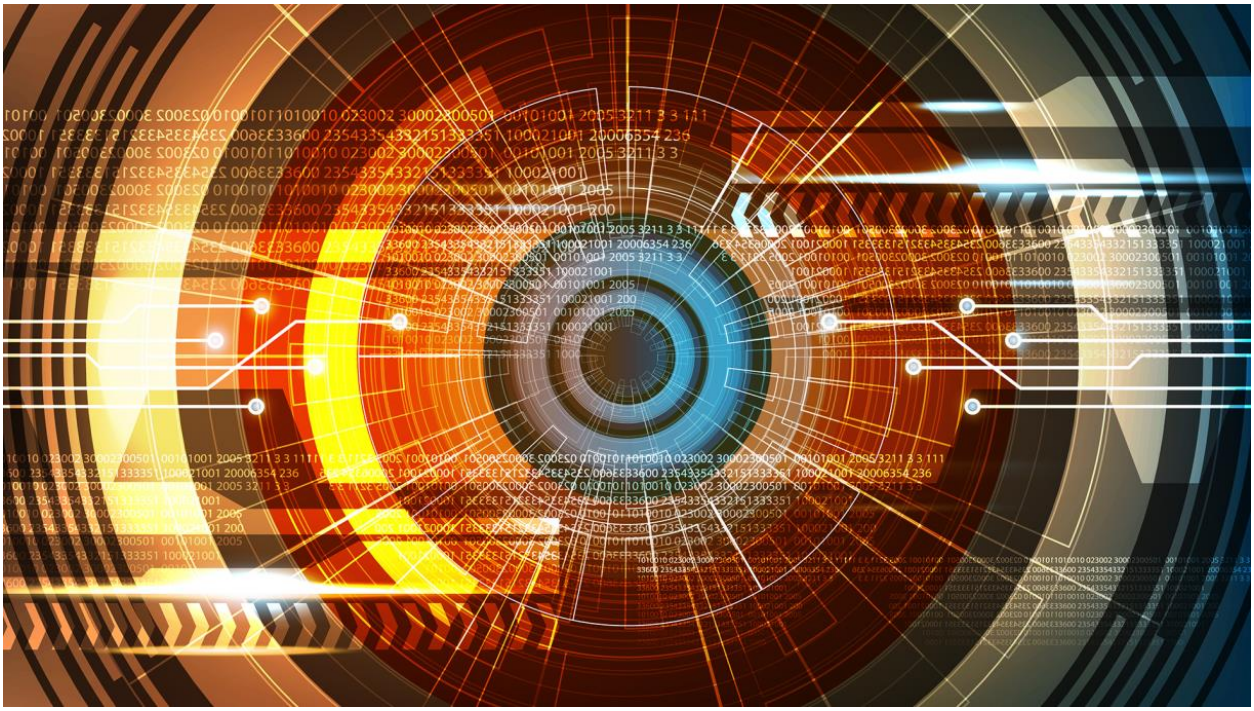


VIDEO PROCESSING WITH INTEL FPGAS WEBINAR SERIES – Q4'2020

Session 2 – Debug VIP pipeline

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

Contents

1. Introduction

1.1. Introduction

In this lab manual we are using the design generated in the previous session: "**Session 1.3 – Complete VIP pipeline**" by adding a *JTAG to Avalon-MM master* module to use **System Console**. This will enable us to perform read/write transactions over the Avalon-MM interfaces of our VIP cores at a very preliminary stage: when a CPU is not available in the system or the software hasn't been developed yet.

That way we can modify, in run time, the parameters of our `vip_pipeline` to bring up the execution, apply different use case scenarios and/or debug potential problems.

This session will cover how to add the JTAG to Avalon-MM master component and how to use the services available through System Console debug application.

NOTE: Download and extract the archived project "**2_System_Console_v1.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.
- **Test flow**: Follow the steps in the chapter 3 "Using System Console" to learn the usage flow and options available.

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:

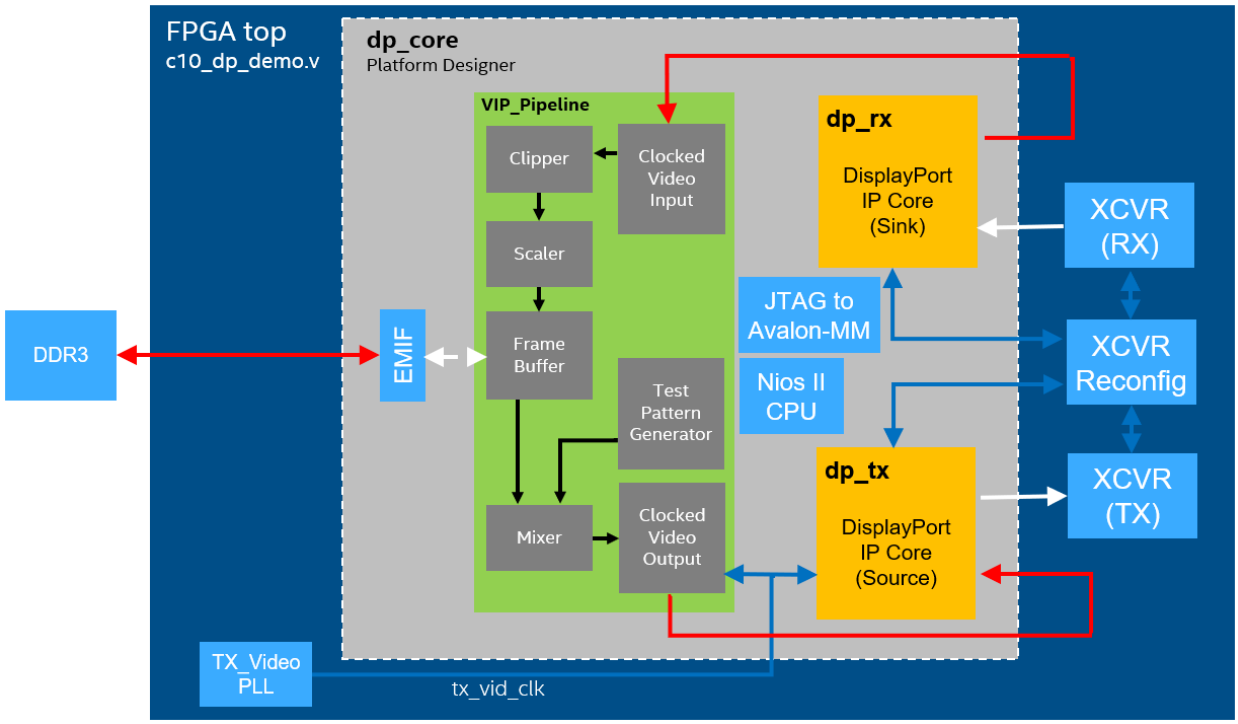
- Intel FPGA DisplayPort IP User Guide
  https://www.intel.com/content/www/us/en/programmable/products/intellectual-property/ip/interface-protocols/m-alt-displayport-megacore.html
- Cyclone 10 GX DisplayPort Design Example User Guide
  https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-dex-dp-c10gx.pdf

- 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

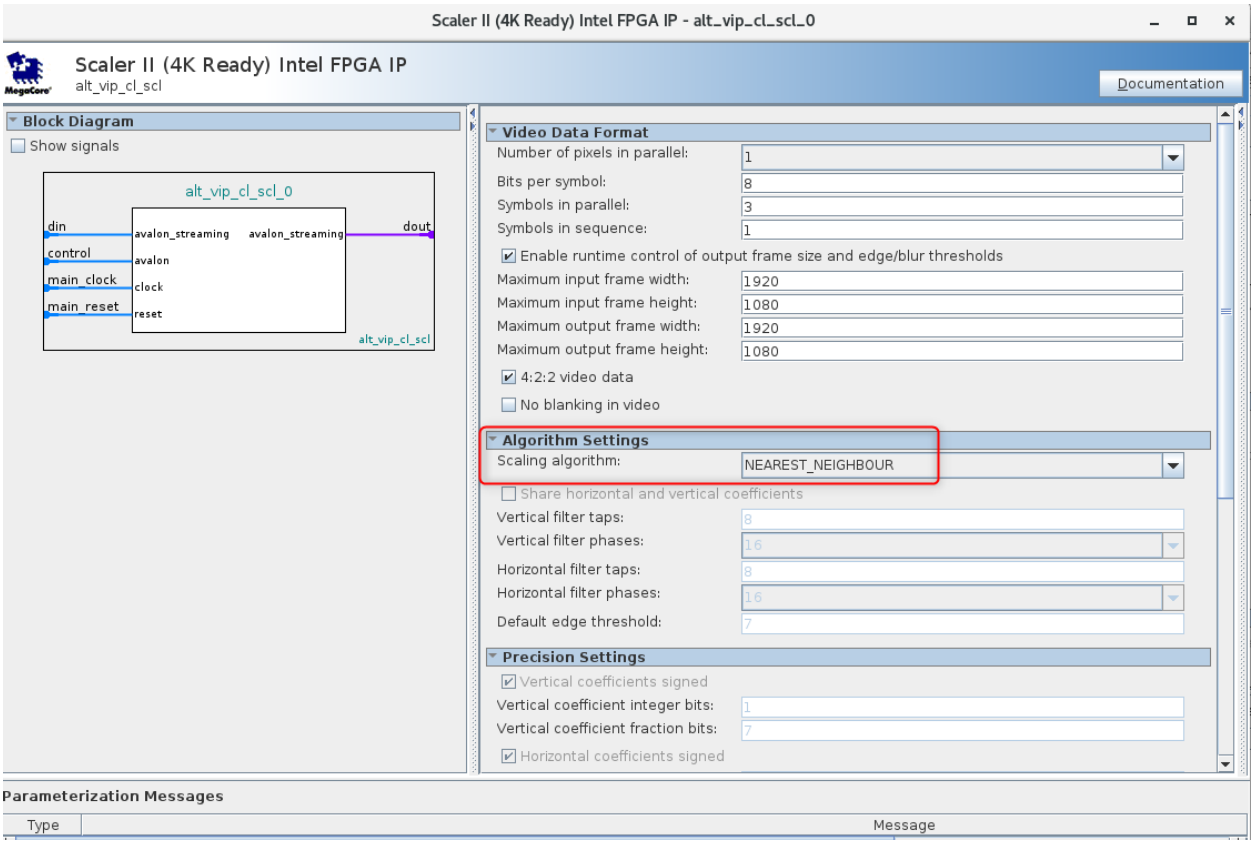
Find, in the below figure, a high-level block diagram with the hardware implementation. Inside the FPGA, we are configuring a set of high-speed transceivers to receive and transmit the DisplayPort video streams in serialized form, acting as the **physical layer**. Attached to them, we have the DisplayPort 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 have modified the `dp_core.qsys` subsystem by adding a **JTAG to Avalon-MM** master module, that is connected to the Avalon-MM bridge in the `vip_pipeline` subsystem

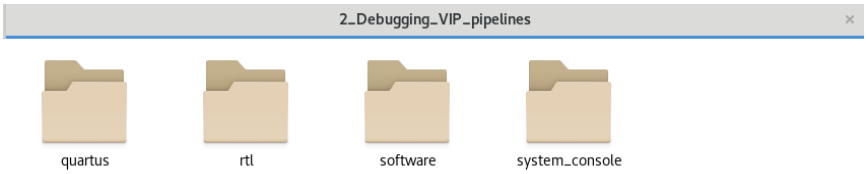
We have also modified the scaler core in the `vip_pipeline.qsys` subsystem to set the scaling algorithm to NEAREST_NEIGHBOUR. We aren't supposed to have software running on the NIOS processor, hence no ability to generate and load coefficients set on the scaler's registers. NEAREST_NEIGHBOUR allows running the scaler without any further parameterization. Output visual quality will be degraded, compared with POLYPHASE, but will allow us to bring up our pipeline more easily.



2. Generating the hardware pipeline

2.1. Setting up the Quartus project

Extract the files provided in the **2_Debugging_VIP_pipelines.tar.gz** package and open the Quartus project located at `<extracted_folder>/quartus/c10_dp_demo.qpf`

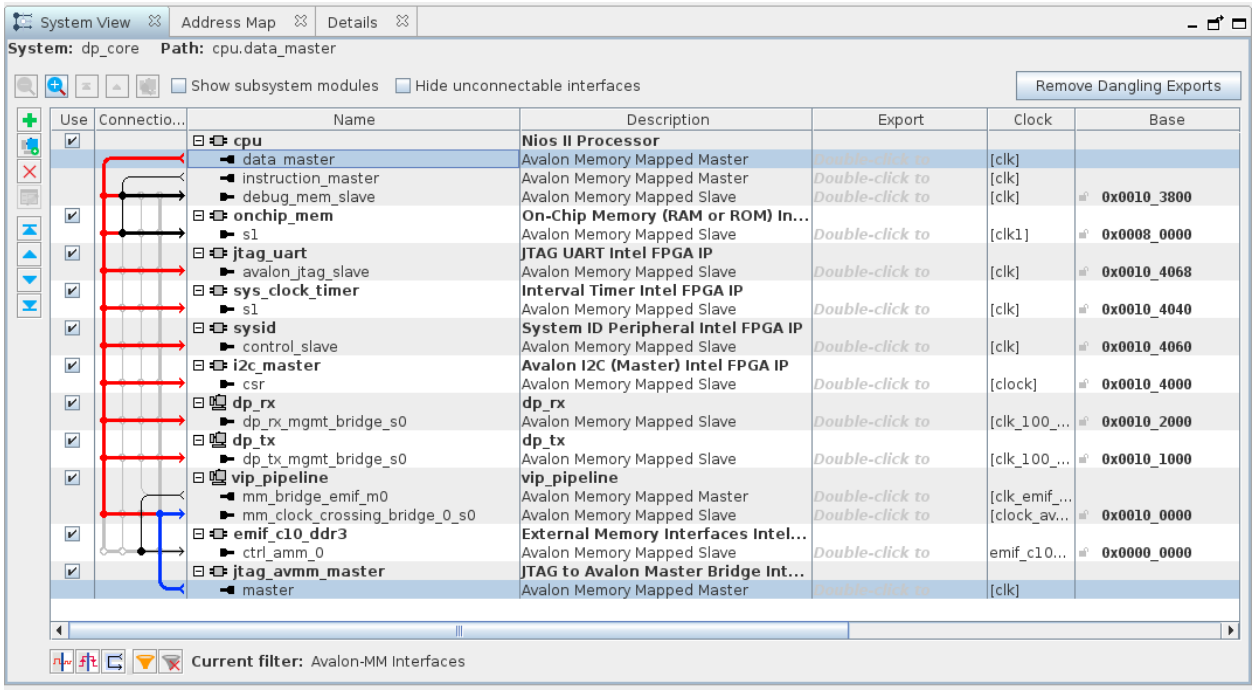


- **quartus**: *contains the project and settings file for the project*
- **rtl**: *contains all the hardware building blocks for the complete pipeline*
- **software**: software application and bsp for the Nios II processor. Please note that we are using the CPU to initialize the DisplayPort RX and TX modules, as well as control link training and EDID management, but NOT to control any of the VIP cores. In this session, we are using System Console.
- **system_console**: collection of scripts we are using to define some address spaces, register offsets and procedures to facilitate the interaction with the VIP cores.

2.2. Examining the project

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

Filter by the Avalon-MM Interfaces to facilitate the view and understand the connection we have done.

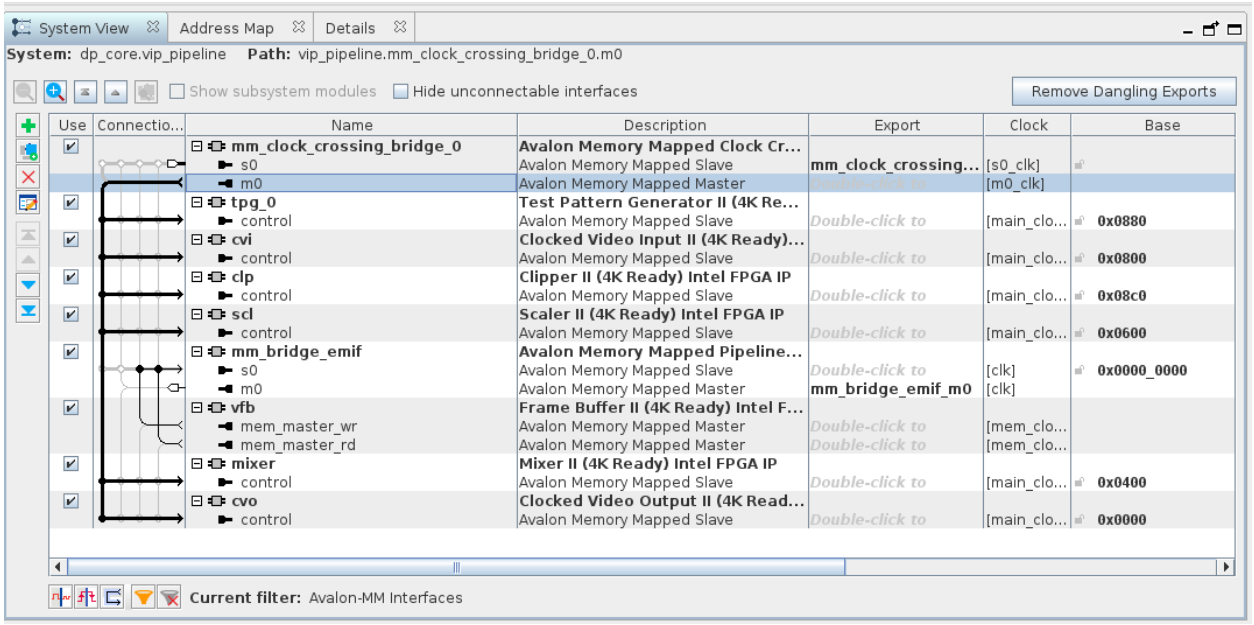


We have added a JTAG to Avalon Master Bridge to the system and connected to the Avalon MM slave port of the clock crossing bridge, used in the **vip_pipeline** to get access to all the VIP cores inside.

Please note that we have kept the connection with the `data_master` interface of the Nios CPU. This will allow us to access to all the VIP cores from two different masters: JTAG and NIOS. Platform Designer is adding all the necessary arbitration logic to handle any potential simultaneous access, although on this application only JTAG Master will be used.

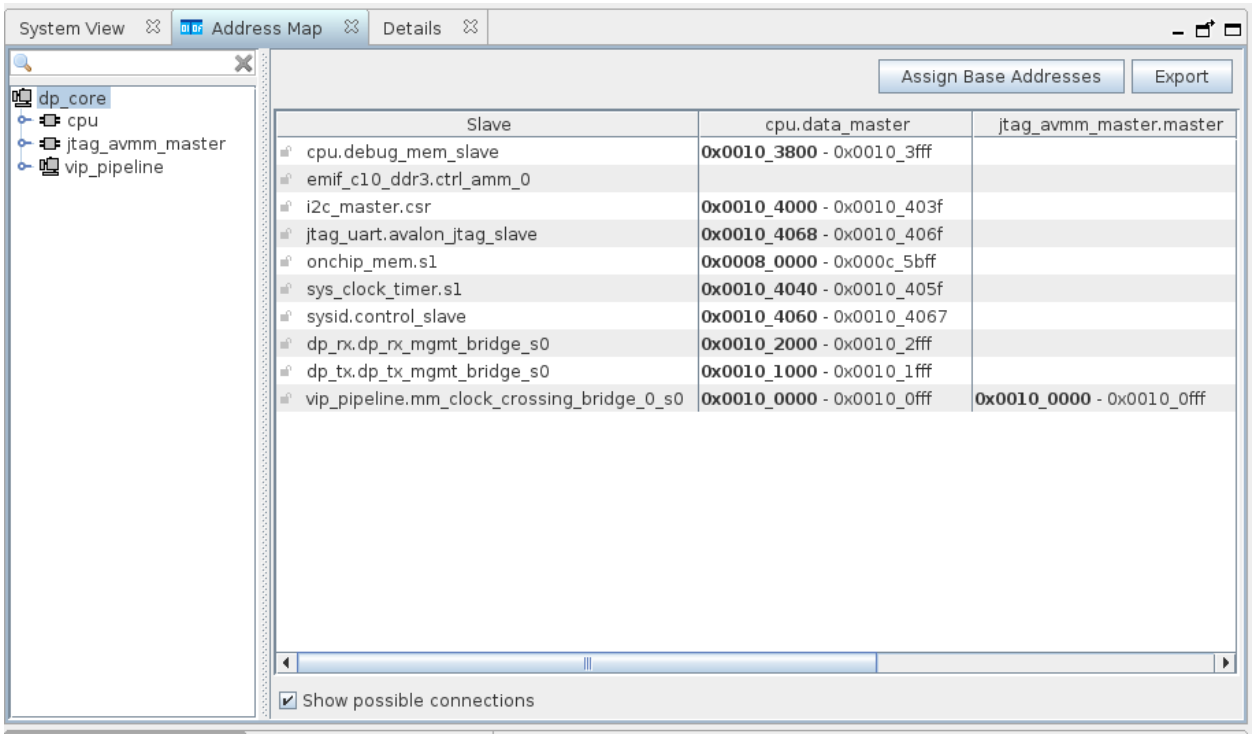
Having a look inside **vip_pipeline** susbsytem, we see how the different VIP cores are connected to the `avmm_clock_crossing_bridge` and the Base Address assigned to each one

We will need these address details later on, while accessing to each individual component from System Console



2.3. Understanding address space

Back in dp_core.qsys module, let's go to the Address Map tab to understand the created address space.

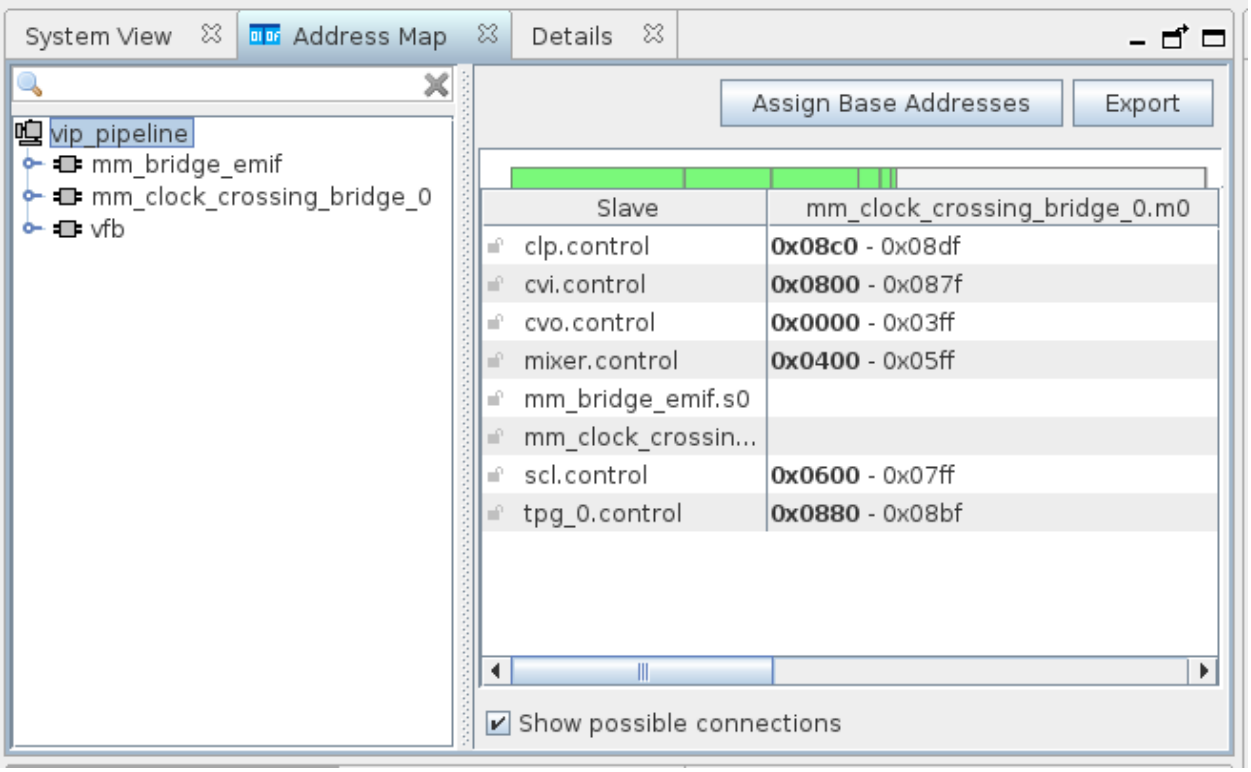


Here we see how the `cpu.data_master` has access to all the slaves on the system, while the `jtag_avmm_master.master` has only access to the `vip_pipeline.mm_clock_crossing_bridge_0_s0` slave.

Notice also, that the `vip_pipeline.mm_clock_crossing_bridge_0_s0` is addressed by 2 independent masters (arbitration logic required) and the address location is the same on both cases.

The Base Address assigned to the `vip_pipeline.mm_clock_crossing_bridge_0_s0` is **0x10_0000**

Drilling into the **vip_pipeline** subsystem, we can identify the downstream Address Map of the `mm_clock_crossing_bridge_0.m0` which manages all the VIP cores



The final Base Address of each VIP core can be calculated as follows:

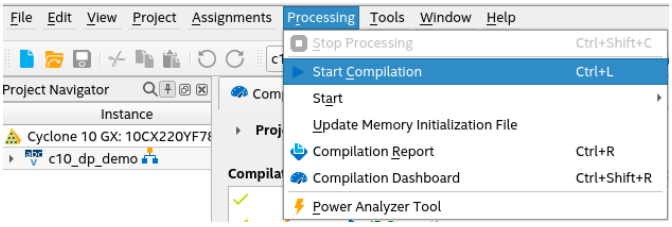
`vip_pipeline.mm_clock_crossing_bridge_0_s0` + `m_clock_crossing_bridge_0.m0`

Hence the table below

VIP core	Address
cvi	0x10_0800
clp	0x10_08c0
scl	0x10_0600
mixer	0x10_0400
cvo	0x10_0000
tpg	0x10_0880

2.4.Generate the programming file

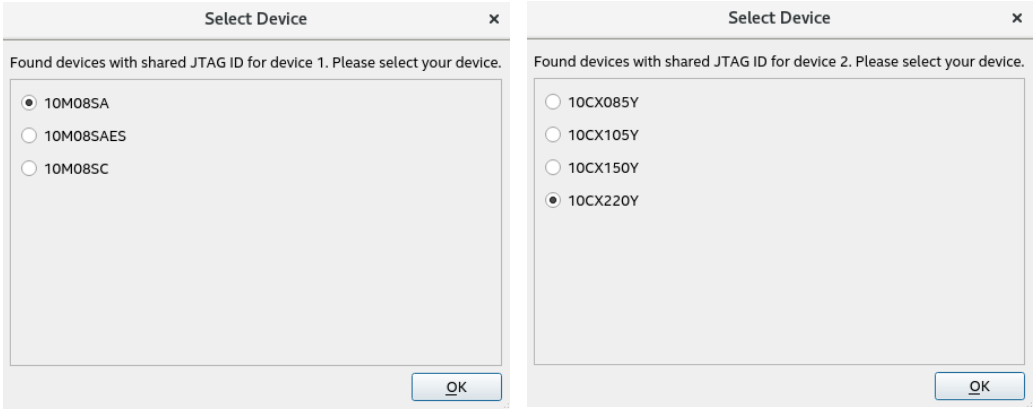
Once you are familiar with the design setup, can now compile the design and generate the FPGA bitstream. Go to **Processing-> Start_Compilation** to trigger the process. It will take ~12 minutes, depending on machine configuration.



2.5.Configuring the FPGA device

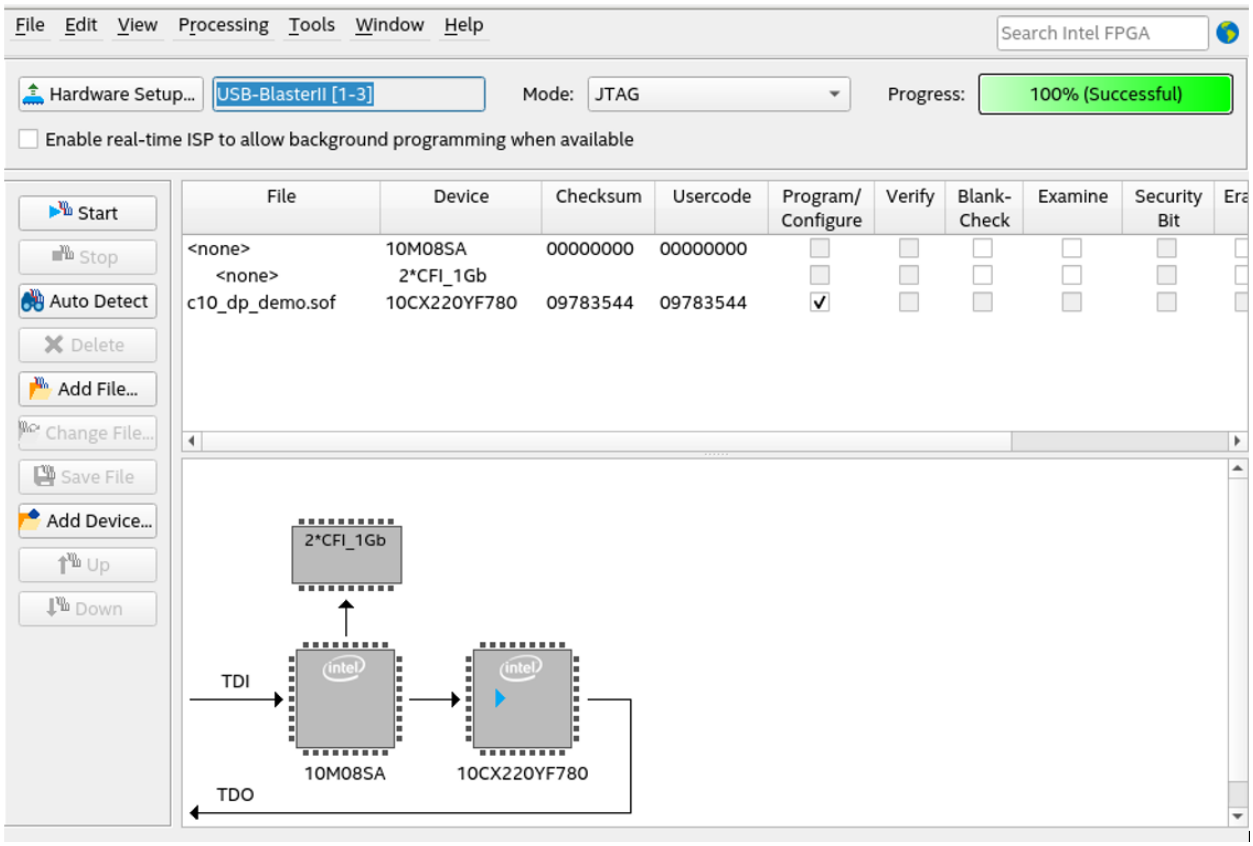
Open the **Quartus Programmer**, select your USB-Blaster cable in the Hardware Setup and click on **Auto Detect** to retrieve the JTAG chain on the Cyclone10 GX Devkit.

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.


File Edit View Processing Tools Window Help
Search Intel FPGA
Hardware Setup...
USB-BlasterII [1-3]
Mode: JTAG
Progress:
100% (Successful)
Enable real-time ISP to allow background programming when available
Start
Stop
Auto Detect
Delete
Add File...
Change File...
Save File
Add Device...
Up
Down
File
Device
Checksum
Usercode
Program/ Configure
Verify
Blank-Check
Examine
Security Bit
<none>
10M08SA
00000000
00000000
<none>
2*CFI_1Gb
c10_dp_demo.sof
10CX220YF780
09783544
09783544
2*CFI_1Gb
TDI
TDO
10M08SA
10CX220YF780

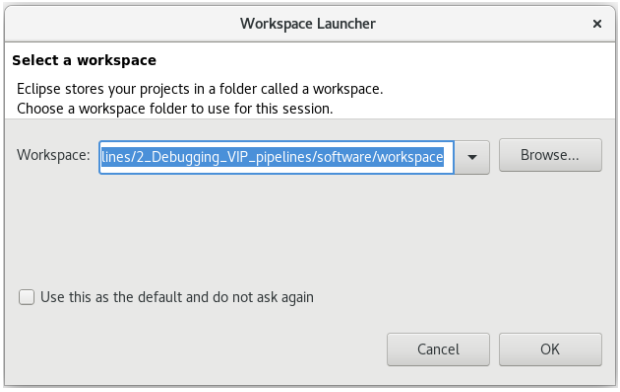
3. Building the software application

3.1.Setting up the Eclipse for Nios project

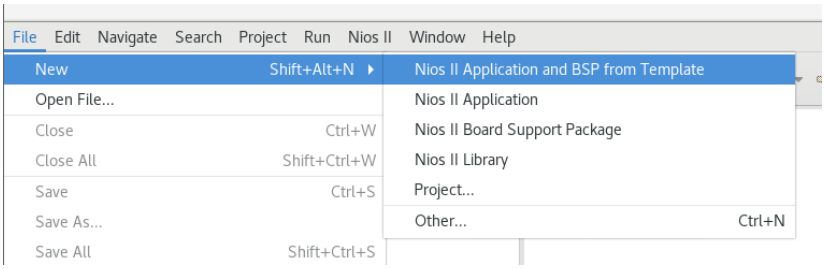
1. Creating Eclipse project for Nios II application and BSP.

- 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

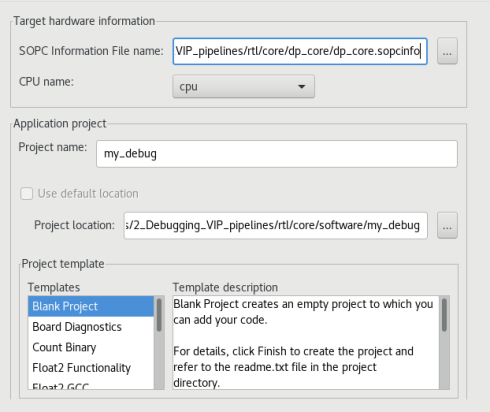


- Under **Target hardware information->SOPC Information File name**, you need to select the `*.sopcinfo` file that contains your Nios CPU. In our case is located in *<project_dir>/rtl/core/dp_core/dp_core.sopcinfo*

- Under **Application project->Project name** : `my_debug`

- Make sure that **Project location** is set to *<project_dir>/software/my_debug*, by default gets located at *<project_dir>/~~rtl/core/~~software/my_debug*

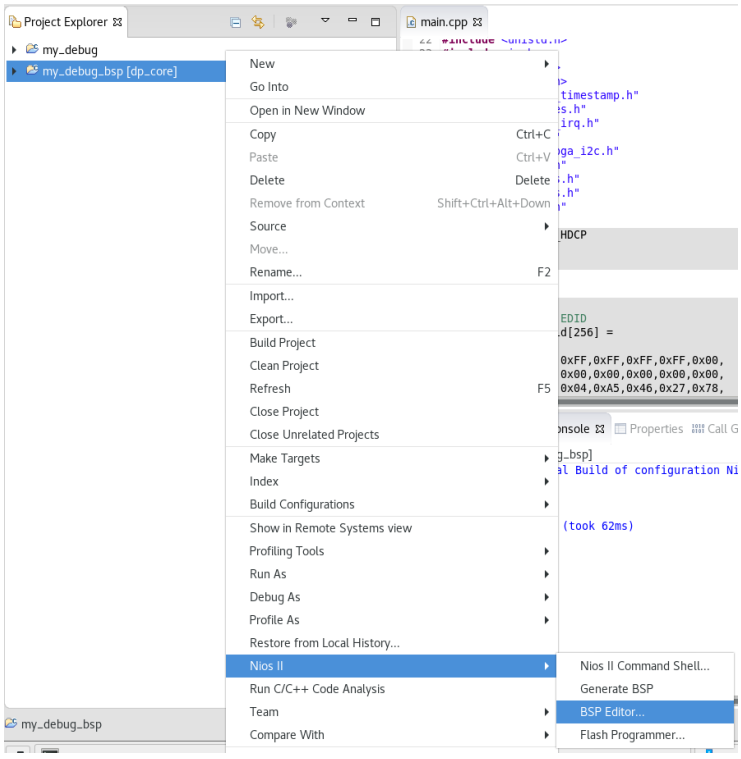
- Select **Blank Project** as **Templates** and Click **Finish**



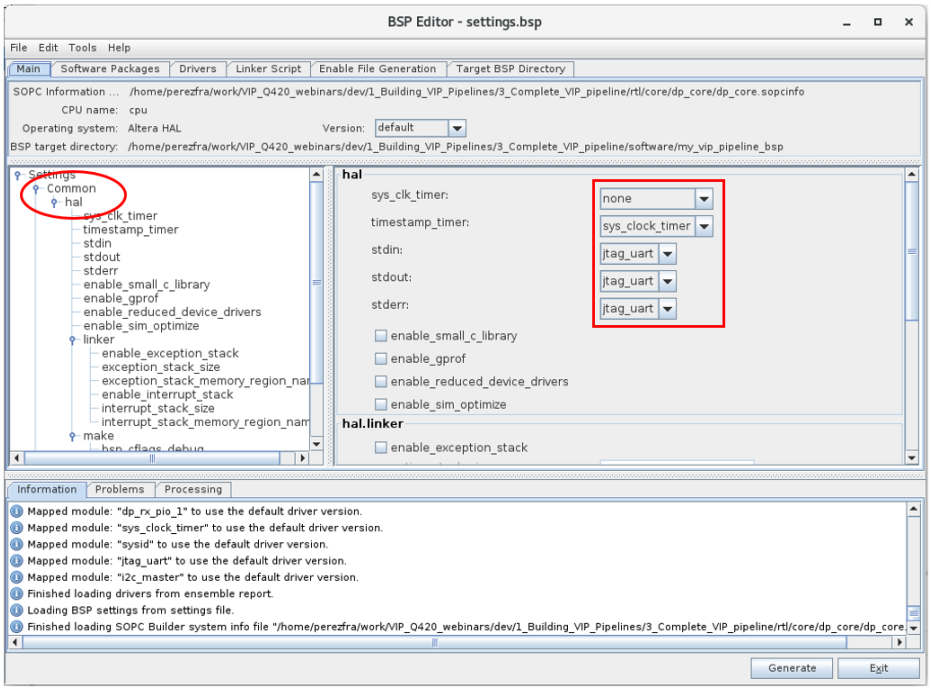
You will end up in a software folder structure as follows:

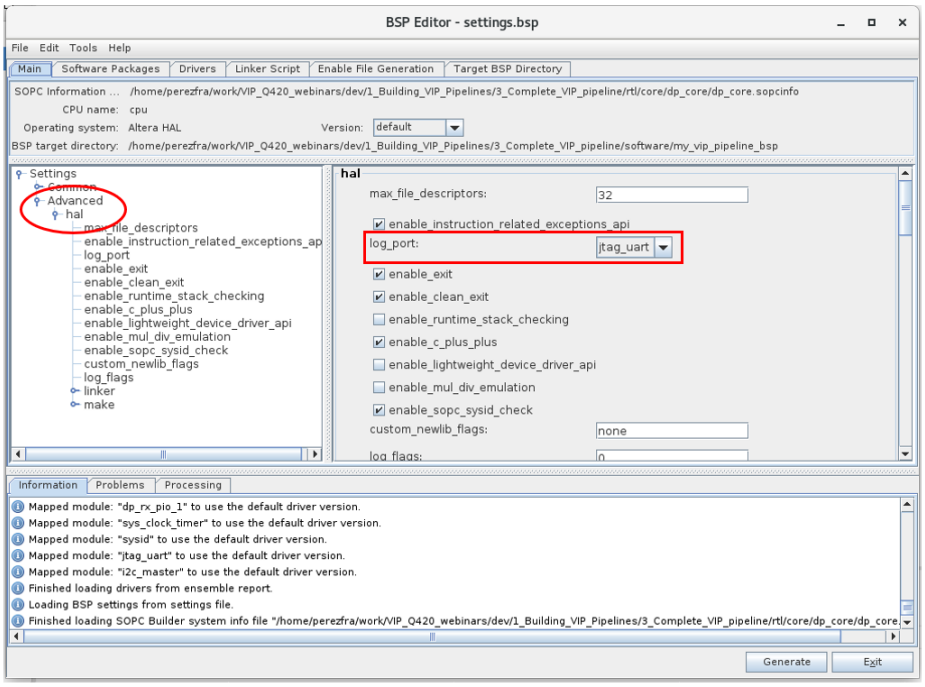
2. **Configure the bsp**.

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



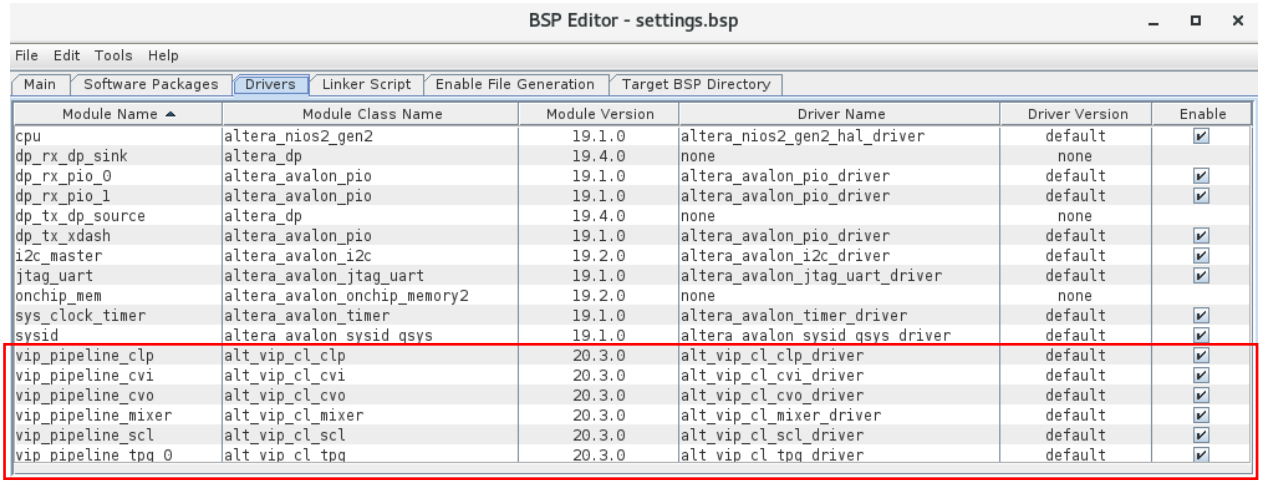
- The **BSP Editor** opens, make the following changes:
  - **Settings->Common->hal->sys_clk_timer**: none
  - **Settings->Common->hal->timestamp_timer**: sys_clock_timer
  - **Settings->Advanced->hal->log_port**: jtag_uart





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

This is because we have connected all the VIP cores to the CPU, even if we are not using them in this example.

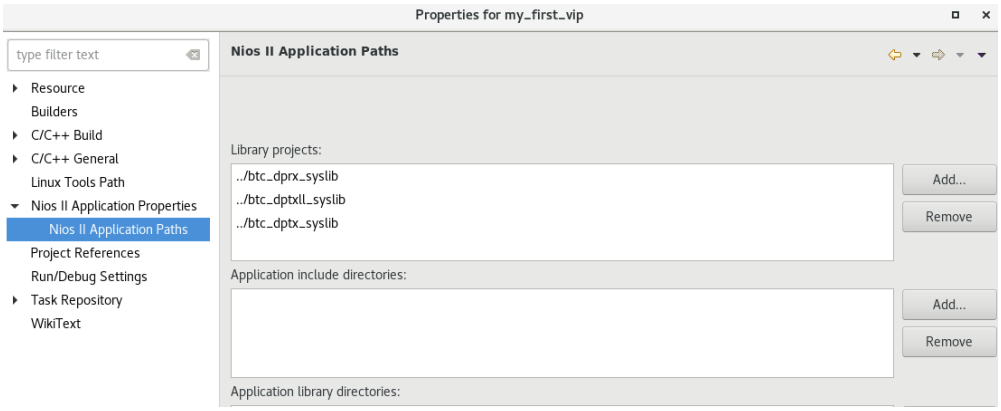


Module Name	Module Class Name	Module Version	Driver Name	Driver Version	Enable
cpu	altera_nios2_gen2	19.1.0	altera_nios2_gen2_hal_driver	default	☑
dp_rx_dp_sink	altera_dp	19.4.0	none	none	
dp_rx_pio_0	altera_avalon_pio	19.1.0	altera_avalon_pio_driver	default	☑
dp_rx_pio_1	altera_avalon_pio	19.1.0	altera_avalon_pio_driver	default	☑
dp_tx_dp_source	altera_dp	19.4.0	none	none	
dp_tx_xdash	altera_avalon_pio	19.1.0	altera_avalon_pio_driver	default	☑
i2c_master	altera_avalon_i2c	19.2.0	altera_avalon_i2c_driver	default	☑
jtag_uart	altera_avalon_jtag_uart	19.1.0	altera_avalon_jtag_uart_driver	default	☑
onchip_mem	altera_avalon_onchip_memory2	19.2.0	none	none	
sys_clock_timer	altera_avalon_timer	19.1.0	altera_avalon_timer_driver	default	☑
sysid	altera_avalon_sysid_qsys	19.1.0	altera_avalon_sysid_qsys_driver	default	☑
vip_pipeline_clp	alt_vip_cl_clp	20.3.0	alt_vip_cl_clp_driver	default	☑
vip_pipeline_cvi	alt_vip_cl_cvi	20.3.0	alt_vip_cl_cvi_driver	default	☑
vip_pipeline_cvo	alt_vip_cl_cvo	20.3.0	alt_vip_cl_cvo_driver	default	☑
vip_pipeline_mixer	alt_vip_cl_mixer	20.3.0	alt_vip_cl_mixer_driver	default	☑
vip_pipeline_scl	alt_vip_cl_scl	20.3.0	alt_vip_cl_scl_driver	default	☑
vip_pipeline_tpg_0	alt_vip_cl_tpg	20.3.0	alt_vip_cl_tpg_driver	default	☑

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

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

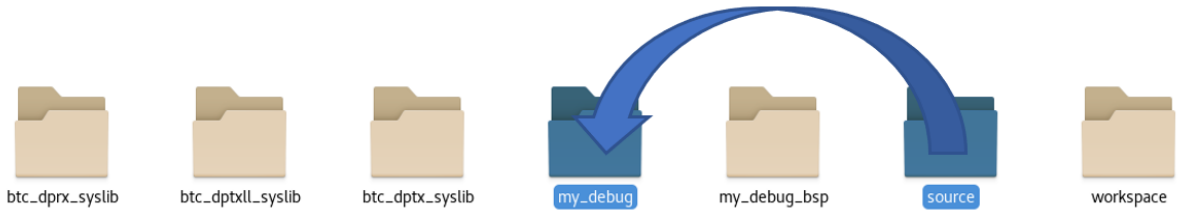
In the dialog box, select **Nios II Application Properties->Nios II Application Paths** and add the Library projects



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

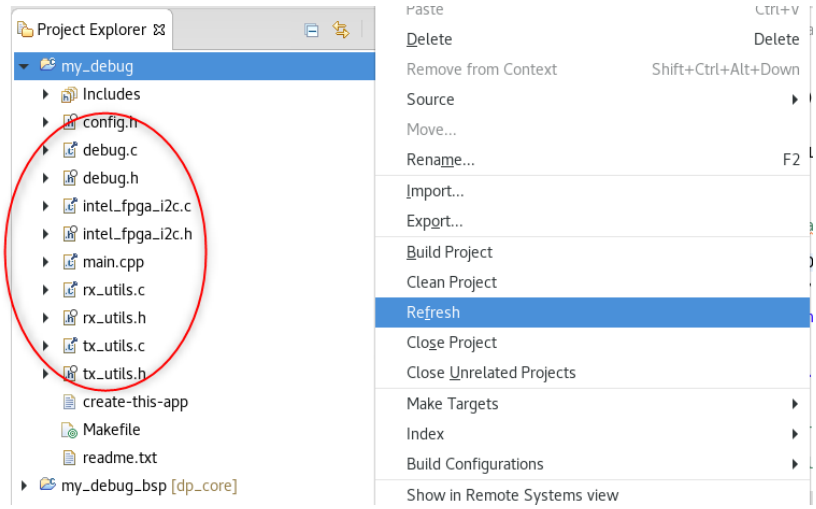
3.2.Importing the code

Copy all the source files provided in the `source` folder into `my_debug` and you can jump directly to **Building and launching program execution,** as we are using the application running in the processor to take care of all related to DisplayPort management, and this is already set in the code.



NOTE: Use `main.cpp` to get an application without any VIP initialization and use **System Console** for the whole operation. You can rename `main_with_vip.cpp_rename` to `main.cpp` to get all the VIP cores initialized by software and still use **System Console** for incremental accesses.

Then you can go to the Project Explorer and right-click on Refresh to update the file list with the source code added and update the Makefile

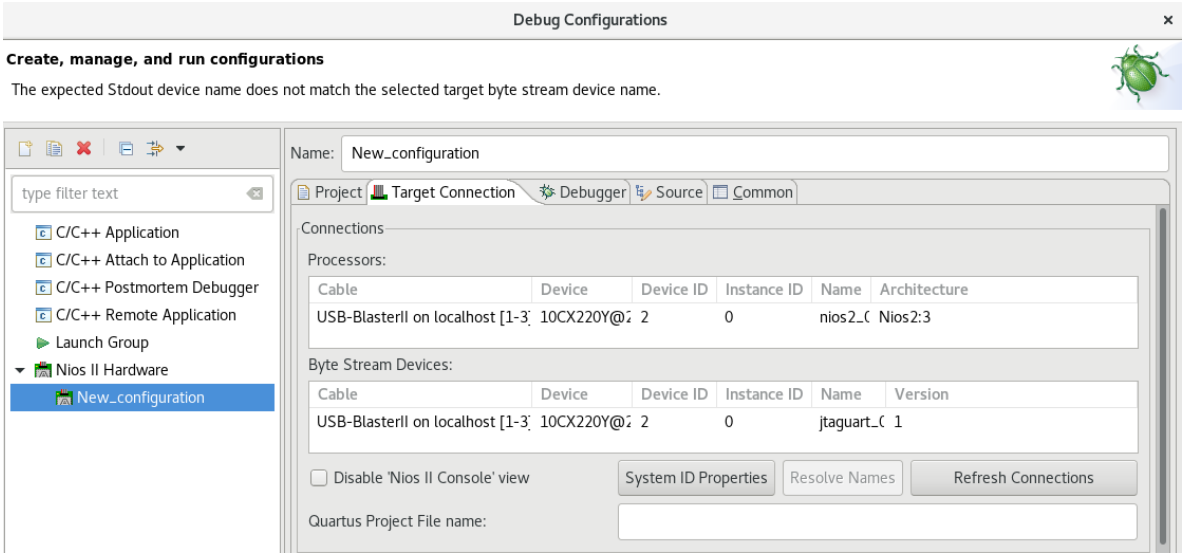


3.3. Building and launching program execution

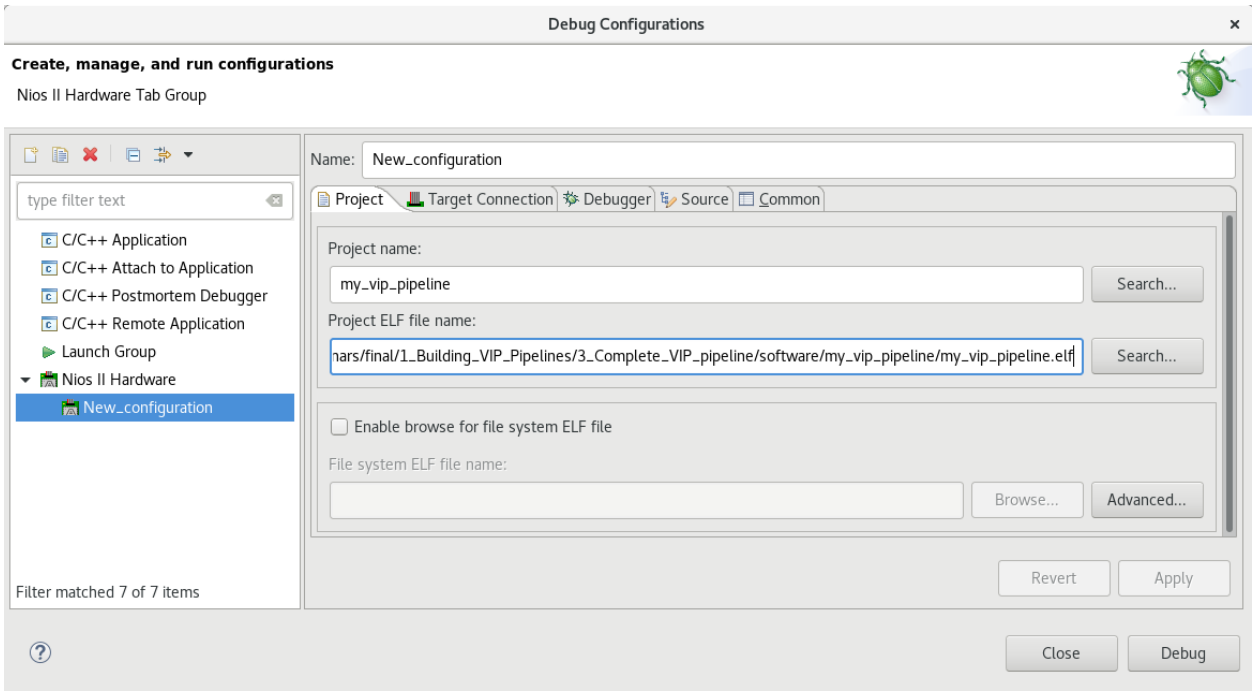
After doing that modifications, we can generate our executable file. In Eclipse IDE, go to **Project->Build All** to compile the *bsp* and generate *`my_debug.elf`* file.

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

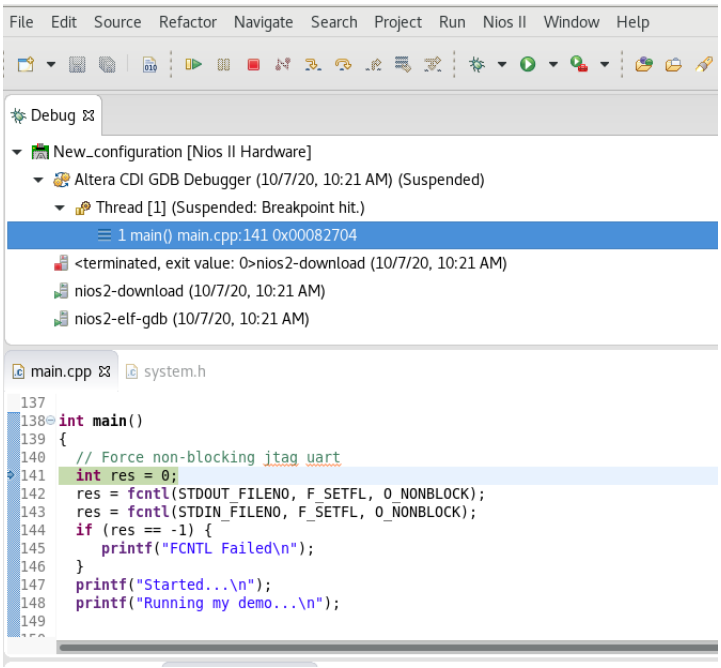
We double-click on **Nios II Hardware** option to create a *New_configuration*. In **Target Connection** tab, click on **Refresh Connections** to select the right *USB-BlasterII* adapter



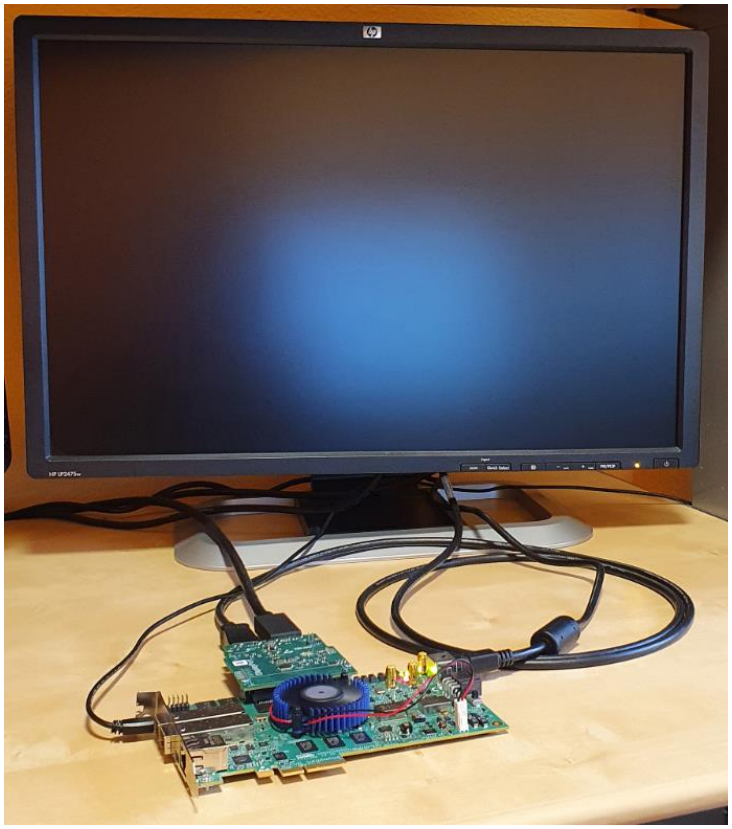
Back to the **Project** tab, make sure the right *Project Name* and *ELF File Name* are selected and just click on **Apply** and **Debug** to launch the session



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 will notice that nothing is happening on the screen, as the VIP cores haven't been initialized. This is what we are doing next with **System Console**.



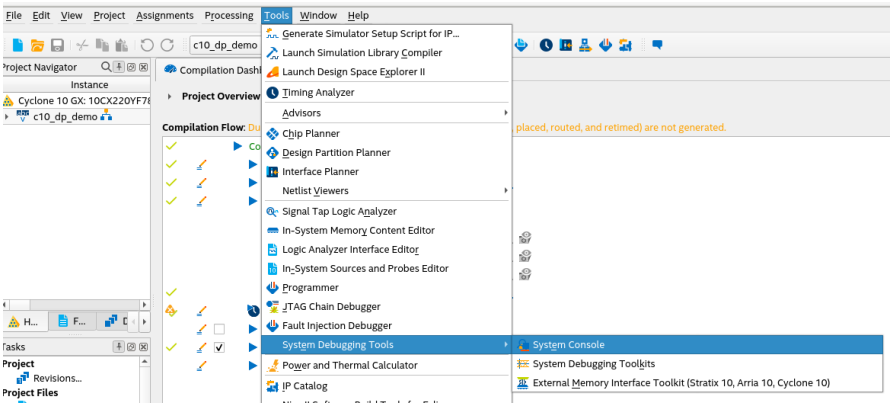
4. Using System Console

4.1. First steps with System Console

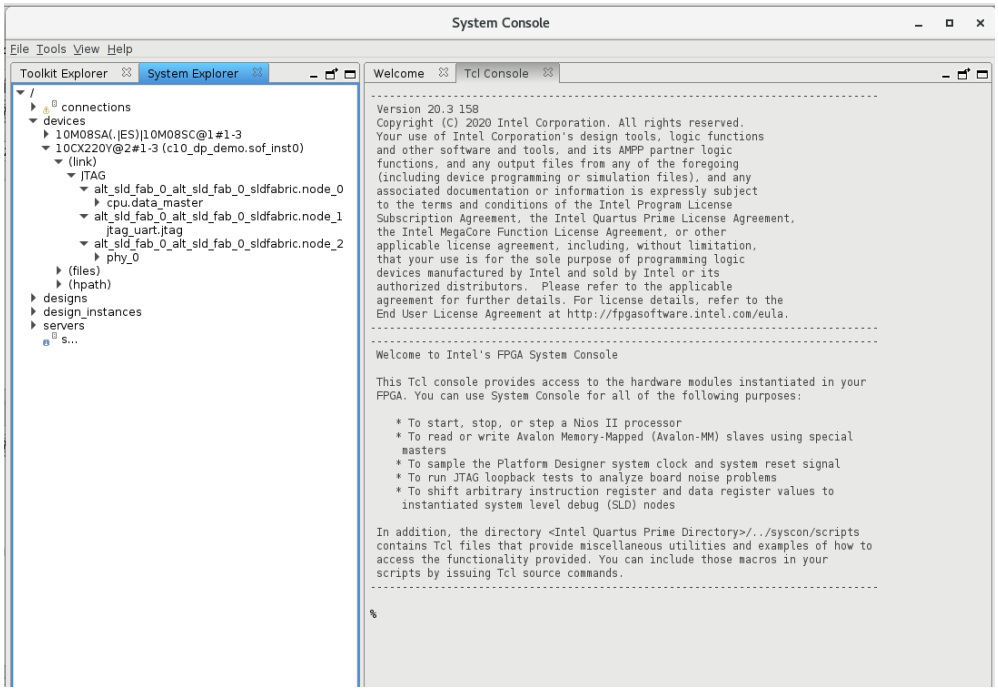
Once we have our FPGA configured with the bitstream (.sof) file and the software application running in the Nios cpu (.elf), it's time to launch System Console.

NOTE: We need to have the cpu running the software applications to service the DisplayPort cores to get successful link training and initialization of the connectivity layer, otherwise this wouldn't have been required.

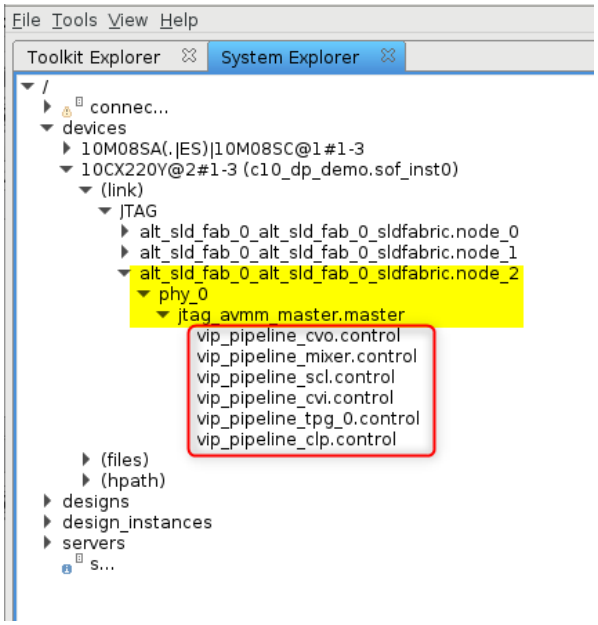
Go to Tools -> System Debugging Tools -> System Console in Quartus to launch the tool.



Once the System Console GUI is open, select the System Explorer tab and expand the devices->10CX220Y->(link)->JTAG to see all the different nodes accessible through the JTAG hub embedded in the FPGA



We can see 3 different types of services available



- **cpu.data_master**
- **jtag_uart.jtag**
- **phy_0** << This is our JTAG to AvalonMM master we will use to drive read/write accesses to the VIP cores.
  If we even expand the **phy_0** instance, we can see the different vip components hanging from the master.

a. Locate a Service Path. In the Tcl Console type

```
#We are interested in master services.
% set service_type "master"
#Get all the paths as a list.
% set master_service_paths [get_service_paths $service_type]
```

We have this response

```
/devices/10CX220Y@2#1-3/(link)/JTAG/alt_sld_fab_0_alt_sld_fab_0_sldfabric.node_0/cpu.data_master
/devices/10CX220Y@2#1-3/(link)/JTAG/alt_sld_fab_0_alt_sld_fab_0_sldfabric.node_2/phy_0/jtag_avmm_master.master
{/devices/10M08SA(.|ES)|10M08SC@1#1-3/(link)/JTAG/(110:130 v3 #0)/jtagmem_0}

#We are interested in the second service in the list, which is our jtag_avmm_master
% set master_index 1
#The path of the selected master.
% set master_path [lindex $master_service_paths $master_index]
#Or condense the above statements into one statement:
% set master_path [lindex [get_service_paths master] 1]
```

As response

```
/devices/10CX220Y@2#1-3/(link)/JTAG/alt_sld_fab_0_alt_sld_fab_0_sldfabric.node_2/phy_0/jtag_avmm_master.master
```

System Console commands require service paths to identify the service instance you want to access. The paths for different components can change between runs of System Console and between versions. Use the `get_service_paths` command to obtain service paths.

After you have a service path to a particular service instance, you can access the service for use. The `claim_service` command directs System Console to start using a particular service instance

```
#Claim a service to access for use:
% set claimed_path [claim_service master $master_path {} {}]
```

Once we have the service claimed, we can perform the desired operations with the service

```
% master_write_32 $claimed_path 0x2000 0x02

% master_read_32 $claimed_path 0x2000 2
```

If we want to change the pattern in the Test Pattern Generator core, we would need to do:

```
% master_write_32 $claimed_path 0x10_089C 1 (to select GrayScale Bars)
```

TPG base address: `0x10_0880` from the address space in Platform Designer

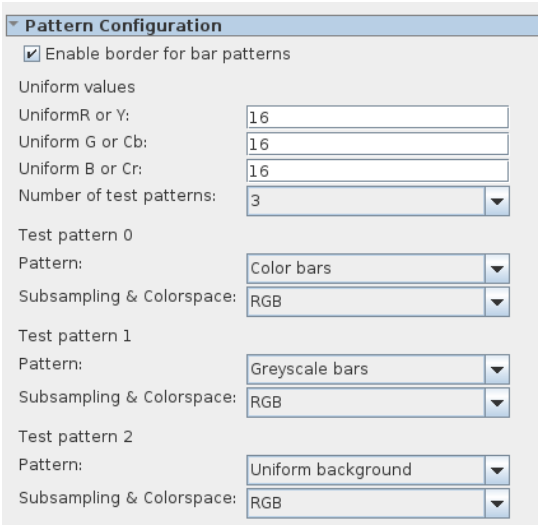
Looking at the VIP User Guide -> Test Pattern Generator Control Register Map, we see that offset 7 is the Pattern Select register.

As we are using 32bit accesses, the equivalent address is 0x07 x 4 = 0x1C

Register to access is: tpg_base (0x10_0880) + offset (0x1C) = **0x10_0890**

Address	Register	Description
0	Control	Bit 0 of this register is the Go bit.. Setting this bit to 0 causes the IP core to stop before control information is read. When you enable run-time control, the Go bit gets deasserted by default. If you do not enable run-time control, the Go is asserted by default. Bit 1 of this register enables or disables the black border in the bars test patterns. This bit gets deasserted (border disabled) at reset. All other bits are unused.
1	Status	Bit 0 of this register is the Status bit, all other bits are unused. The IP core sets this address to 0 between frames. The IP core sets this address to 1 while it is producing data and cannot be stopped.
2	Interrupt	Unused.
3	Reserved	Reserved.
4	Output Width	The width of the output frames or fields in pixels.
5	Output Height	The progressive height of the output frames or fields in pixels. *Note:* Value from 32 up to the maximum specified in the parameter editor.
6	Output interlacing	The output interlacing standard. Set to 0 for progressive, 1 for interlacing with F0 first, and 2 for interlacing with f1 first.
7	Pattern select	The test pattern configuration to enable. Write 0 for configuration 0, 1 for configuration 1, and so on.
8	R/Y value	The value of the R (or Y) color sample when the test pattern is a uniform color background. *Note:* Available only when the IP core is configured to produce a uniform color background and run-time control interface is enabled.
9	G/Cb value	The value of the G (or Cb) color sample when the test pattern is a uniform color background. *Note:* Available only when the IP core is configured to produce a uniform color background and run-time control interface is enabled.
10	B/Cr value	The value of the B (or Cr) color sample when the test pattern is a uniform color background. *Note:* Available only when the IP core is configured to produce a uniform color background and run-time control interface is enabled.

In the hardware configuration for our Test Pattern Generator, we set:



So, writing 0x1 to the register will switch the TPG to generate Grayscale bars.

NOTE: Looking and calculate the final addresses for all the registers in the cores, can be a daunting task. In the next section we will see how to create some scripts to facilitate the run time access.

4.2. Building scripts to simplify accesses

In order to facilitate the task of accessing the VIP cores from System Console, we have included some Tcl scripts that you can use as templates for your own designs.

In the `<project_dir>` directory you can see a folder named **system_console**:

Looking inside there are 3 files:

- `system_base_addr_map.tcl`
  - We have set some variables containing the base addresses of all the VIP cores, extracted directly from `system.h`

```
# set base addresses of VIP IP core according to Qsys address map
set base_address_cvi   0x100800              ;# base address for CVI II IP
set base_address_cvo   0x100000              ;# base address for CVO II IP
set base_address_mixer 0x100400              ;# base address for MIXER II IP
set base_address_tpg   0x100880              ;# base address for TEST_PATTERN_GENERATOR II IP
set base_address_clp   0x1008C0              ;# base address for CLIPPER II IP
set base_address_scl   0x100600              ;# base address for SCALER II IP
```

- `vip_csr_offset.tcl`
  - Constains variable definitions for the different internal registers of each VIP core, with their relative offset as extracted from the VIP User Guide document.

```
#set CVI II IP core CSR offset
set cvi_csr_control             0x00
set cvi_csr_status              0x04  ; #0x01 * 4

#set TEST_PATTERN_GENERATOR II IP core CSR offset
set tpg_csr_control             0x00
set tpg_csr_status              0x04  ; #0x01 * 4
set tpg_csr_width               0x10  ; #0x04 * 4
set tpg_csr_height              0x14  ; #0x05 * 4
set tpg_csr_select              0x1C  ; #0x07 * 4
set tpg_csr_red                 0x20  ; #0x08 * 4
set tpg_csr_green               0x24  ; #0x09 * 4
set tpg_csr_blue                0x28  ; #0x0A * 4

#set CLIPPER II IP core CSR offset
set clp_csr_control             0x00
set clp_csr_status              0x04  ; #0x01 * 4
set clp_csr_left_offset         0x0C  ; #0x03 * 4
set clp_csr_top_offset          0x14  ; #0x05 * 4
set clp_csr_width               0x10  ; #0x04 * 4
set clp_csr_height              0x18  ; #0x06 * 4

#set SCALER II IP core CSR offset
set scl_csr_control             0x00
set scl_csr_status              0x04  ; #0x01 * 4
set scl_csr_out_width           0x0C  ; #0x03 * 4
set scl_csr_out_height          0x10  ; #0x04 * 4

#set MIXER II IP core CSR offset
set mixer_csr_control           0x00
set mixer_csr_background_width  0x0C  ; #0x03 * 4
set mixer_csr_background_height 0x10  ; #0x04 * 4
set mixer_csr_input0_x          0x20  ; #(0x08+5n) * 4 where n is the input number
set mixer_csr_input0_y          0x24  ; #(0x09+5n) * 4 where n is the input number
set mixer_csr_input0_control    0x28  ; #(0x0A+5n) * 4 where n is the input number
set mixer_csr_input1_x          0x34  ; #(0x08+5n) * 4 where n is the input number
set mixer_csr_input1_y          0x38  ; #(0x09+5n) * 4 where n is the input number
set mixer_csr_input1_control    0x3C  ; #(0x0A+5n) * 4 where n is the input number

#set CVO II IP core CSR offset
set cvo_csr_control             0x00
set cvo_csr_status              0x04  ; #0x01 * 4
set cvo_csr_mode_match          0x0C  ; #0x03 * 4
set cvo_csr_bank_select         0x10  ; #0x04 * 4
```

- `main.tcl`
  - It's a collection of functions to encapsulate certain operations, like master enumeration and service claiming, VIP pipeline initialization and specific functions to allow modify runtime parameters: scaler resolution, mixer layer position, mixer layer enable/disable, …
  - We source the address map files

```
source system_base_addr_map.tcl
source vip_csr_offset.tcl
```

- Claim the service master, in your specific implementation, you might want to modify the master_index variable according to your master enumeration

```
proc init_master {} {
#declare a variable to hold the claimed master resource
    variable claimed_path

#We are interested in master services.
    set service_type "master"
#Get all the paths as a list.
    set master_service_paths [get_service_paths $service_type]
#We are interested in the second service in the list, which is our jtag_avmm_master
    set master_index 1
#The path of the selected master.
    set master_path [lindex $master_service_paths $master_index]
#Claim a service to access for use:
    set claimed_path [claim_service master $master_path {} {}]
}
```

- Define auxiliary functions to perform actions

```
# Enable CVO II to output video data to AV-ST bus
proc cvo_go {} {
   global claimed_path
   global base_address_cvo
   global cvo_csr_control

   puts "Writing CVO control register\n"
   master_write_32 $claimed_path [expr $base_address_cvo + $cvo_csr_control] 0x01
}

# Enable MIXER to output video data to AV-ST bus
proc mixer_go {} {
   global claimed_path
   global base_address_mixer
   global mixer_csr_control

   puts "Writing MIXER control register\n"
   master_write_32 $claimed_path [expr $base_address_mixer + $mixer_csr_control] 0x01
}

# Enable TPG to output video data to AV-ST bus
proc tpg_go {} {
   global claimed_path
   global base_address_tpg
   global tpg_csr_control

   puts "Writing TPG control register\n"
   master_write_32 $claimed_path [expr $base_address_tpg + $tpg_csr_control] 0x01
}

# Control Mixer layer 1
proc enable_mixer_layer1 {enable} {
   global base_address_mixer
   global mixer_csr_input1_control
   global claimed_path

   puts "Writing MIXER enable layer1 register\n"
   master_write_32 $claimed_path [expr $base_address_mixer + $mixer_csr_input1_control] $enable
}

# Set Mixer layer 1 position
proc set_mixer_layer1_position {x y} {
   global base_address_mixer
   global mixer_csr_input1_x
   global mixer_csr_input1_y
   global claimed_path

   puts "Writing MIXER layer1 position registers\n"
   master_write_32 $claimed_path [expr $base_address_mixer + $mixer_csr_input1_x] $x
   master_write_32 $claimed_path [expr $base_address_mixer + $mixer_csr_input1_y] $y
}
```

```
# Change the pattern in TPG
proc change_pattern {pattern} {
   global base_address_tpg
   global tpg_csr_select
   global claimed_path

   puts "Writing TPG select pattern register\n"
   master_write_32 $claimed_path [expr $base_address_tpg + $tpg_csr_select] $pattern
}

# Set Scaler output resolution
proc set_scaler_resolution {width height} {
   global base_address_scl
   global scl_csr_out_width
   global scl_csr_out_height
   global claimed_path

   puts "Writing SCL output registers\n"
   master_write_32 $claimed_path [expr $base_address_scl + $scl_csr_out_width] $width
   master_write_32 $claimed_path [expr $base_address_scl + $scl_csr_out_height] $height
}
```

- Init the pipeline

```
# Enable all VIP IPs
proc go {} {
    cvo_go
    set_mixer_layer1_position 0 0
    mixer_go
    tpg_go
    set_scaler_resolution 1280 720
    scl_go
    set_clipper_area 0 0 1920 1080
    clp_go
    cvi_go
    enable_mixer_layer0 1
}
```

- And then, you are ready to go by typing dedicated commands in the console

4.3.Control your VIP pipeline

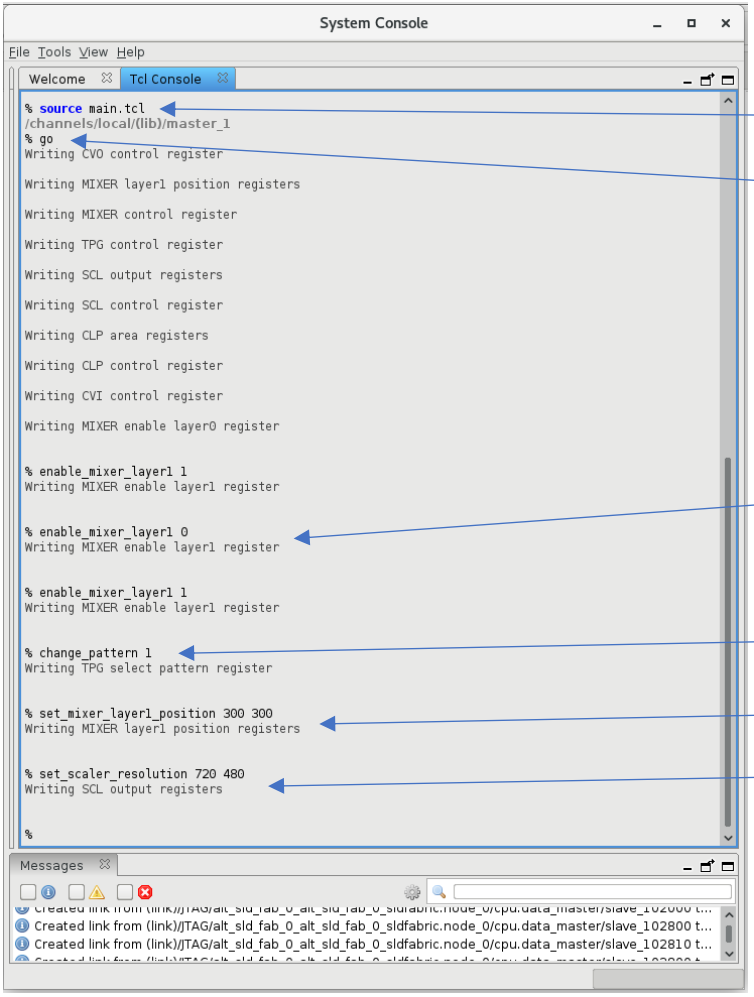
Back in the Tcl Console, you should start by sourcing the `main.tcl` script to initialize all the variables and proc functions.

Then you can start the pipeline, by calling the proc `go`

You will see as a response, what write sequence the go function has performed. After you can issue single commands to the different VIP cores like:

- enable/disable mixer layer
- modify the layer position offsets in the mixer
- change the scaler output resolution
- switch to a different generated pattern
- …

System Console allows you to interact with your `vip_pipeline` and test different scenarios and behavior without writing any single line of code.



source main.tcl script

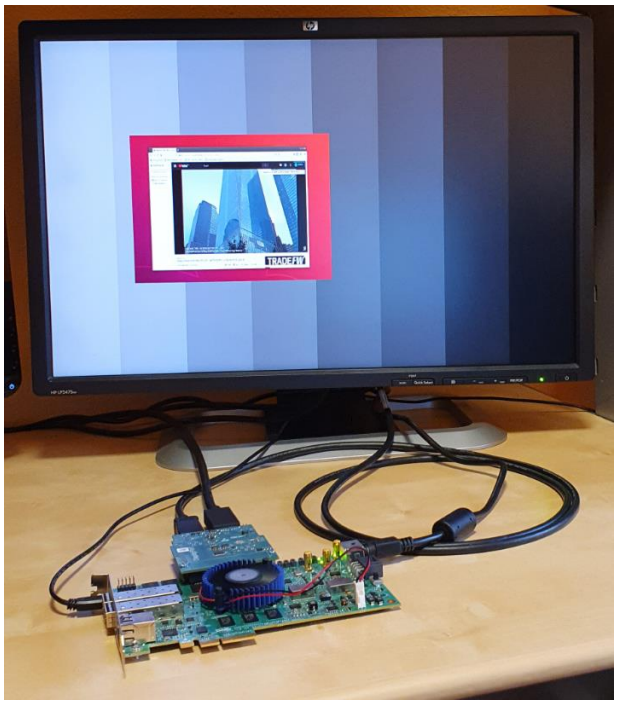
initialize the pipeline

enable/disable mixer layers

change pattern to grayscale bars

modify mixer layer position

Modify scaler resolution



5. Summary

In this lab, we have exercised with System Console to bring up your video pipeline in the absence of software application ready for your processor

- We have added a JTAG to Avalon-MM master bridge connected to all Avalon-MM slave ports of the VIP cores.
- We have used the Tcl Console to discover and claim a master service to allow read/write access to the memory space
- We have prepared some scripts to automate the task of accessing the cores
- We have exercised dynamically with some live actions over the scaler, mixer and test pattern generator to see immediate effects on the screen without software intervention.