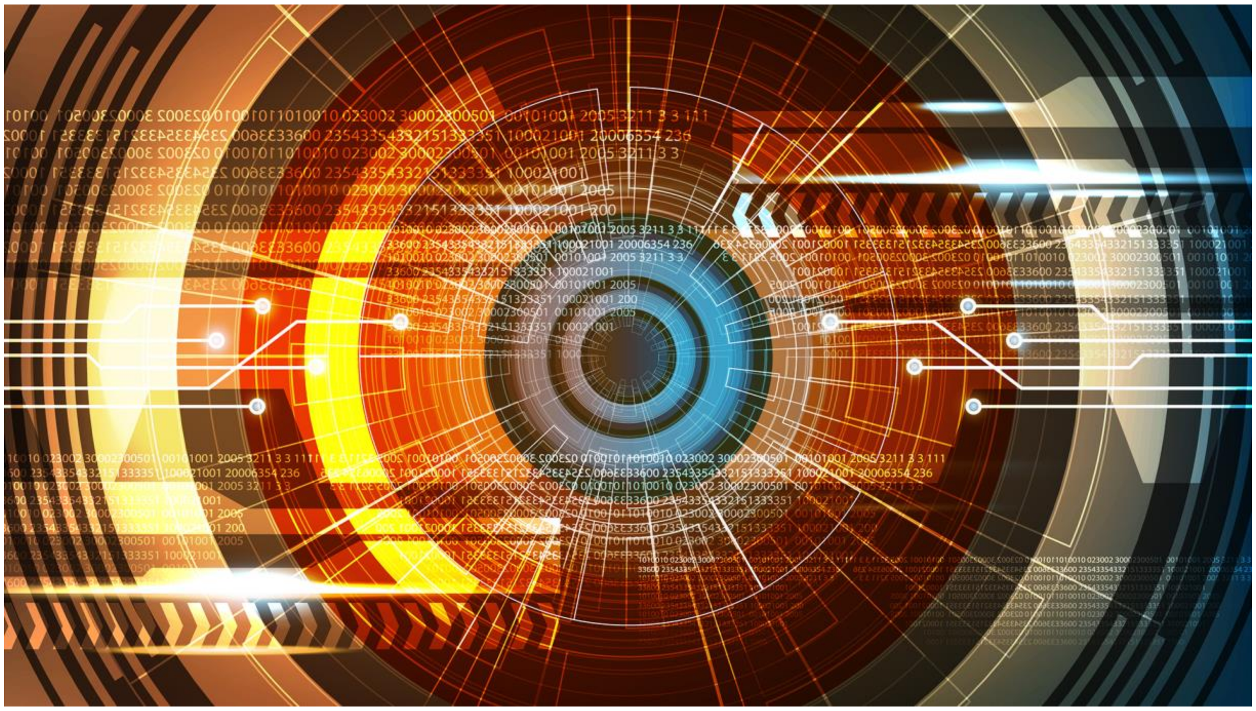


VIDEO PROCESSING WITH INTEL FPGAS WEBINAR SERIES – Q4'2020

<u>Session 4 – On Screen Display overlay</u>

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

Contents

1. Introduction

1.1. Introduction

The VIP suite contains +20 different IP cores installed by default. The collection provides modules for basic and required manipulations in almost every video processing pipeline: crop, scale, color space conversion, gamma correction, deinterlace, mix, … But, how can you add graphics elements to the screen? What if you want to add an operation menu, or insert some text string to show information or display values? What if you need to superimpose some alignment markers, or bounding boxes, …

In this lab manual we are adding a Frame Reader to our VIP pipeline that is continuously reading data from a memory area in the DDR3 bank. This memory block is used by the Nios2 processor, which is running a lightweight graphic library, as video frame memory, to draw text and graphical content to be blended with the live video in an additional layer enabled in the mixer.

The Frame Reader is configured to retrieve pixel information from a specific memory address and generates AvalonST-Video compliant packets that are routed to a layer in the mixer added for that purpose. This layer has been configured to support alpha blending, so we can play with different transparency levels on the graphic content.

In order to generate the graphic elements, we are using a lightweight library in the processor that allows us to draw simple geometrical shapes and insert text strings with configurable font, size and color along the complete video output resolution.

One of the benefits of the FPGAs is that you can, effectively, decouple your control and data planes and, partition your application into blocks to be implemented in software programs or hardware logic in the FPGA fabric. Here, we have implemented a frame reader component able to read back from memory a 1080p60 frame or even a 4K60p frame at the required refresh rate for the application to overlay graphic content on top of full screen video. Once configured, this Frame Reader acts autonomously, without any CPU intervention. The CPU is then in charge of refreshing the graphic content at its own pace, rendering information in the memory space allocated as graphic frame.

Although the Nios2 CPU can address and update the full resolution of the frame, don't consider render the complete frame in real time, but certainly, smaller areas can be updated to sustain the required frame rate. We will see later in the exercise how we can implement a real time frame counter displayed on the screen.

In this session, we are following the steps to modify the VIP pipeline to include and configure the different hardware modules in the **vip_pipeline**, as well as add and use the graphic library in the software flow.

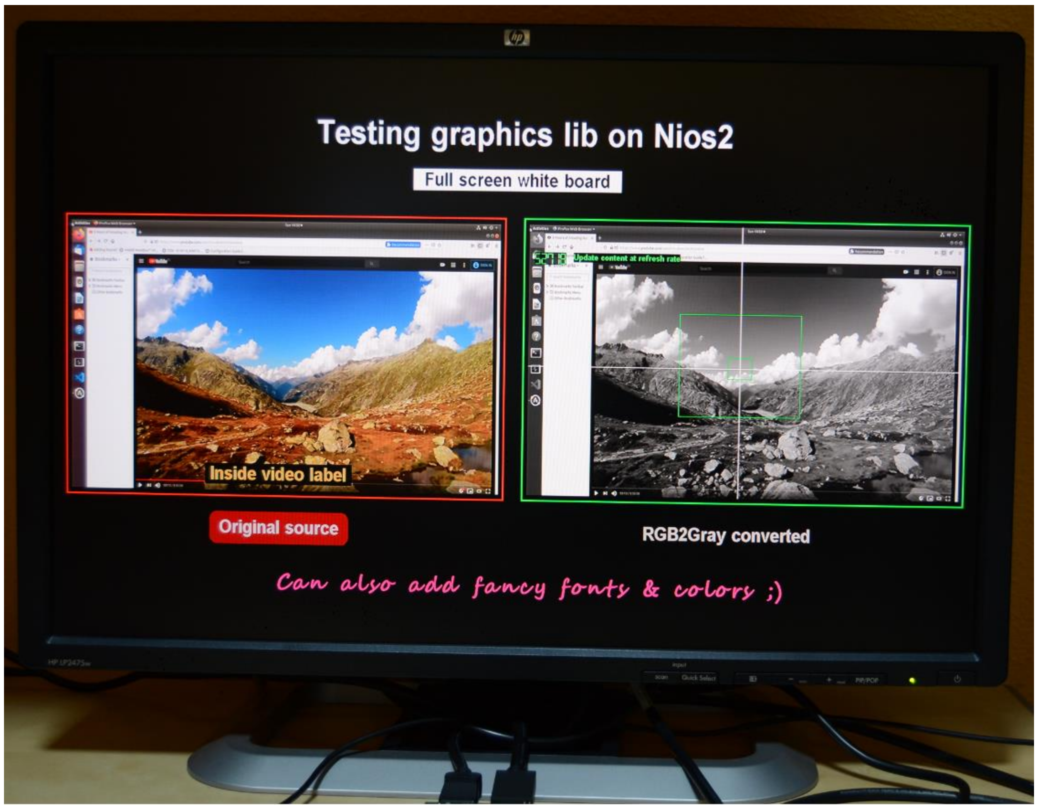
We have also duplicated the video data path to allow simultaneous visualization of the video source in 2 independent picture-in-picture windows: original source and gray scale converted version for comparison.

We are providing an archived package, with all the project files needed, to follow along the instructions here in the manual.

Download and extract the archived project "**4_OSD_Overlay_v1.tar.gz**" located in the Github repository:

https://github.com/perezfra/VIP_webinars_Intel_FPGA

See in the picture below the expected result we will get after this session.


Testing graphics lib on Nios2
Full screen white board
Update content at refresh rate
Inside video label
Original source
RGB2Gray converted
Can also add fancy fonts & colors ;)

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

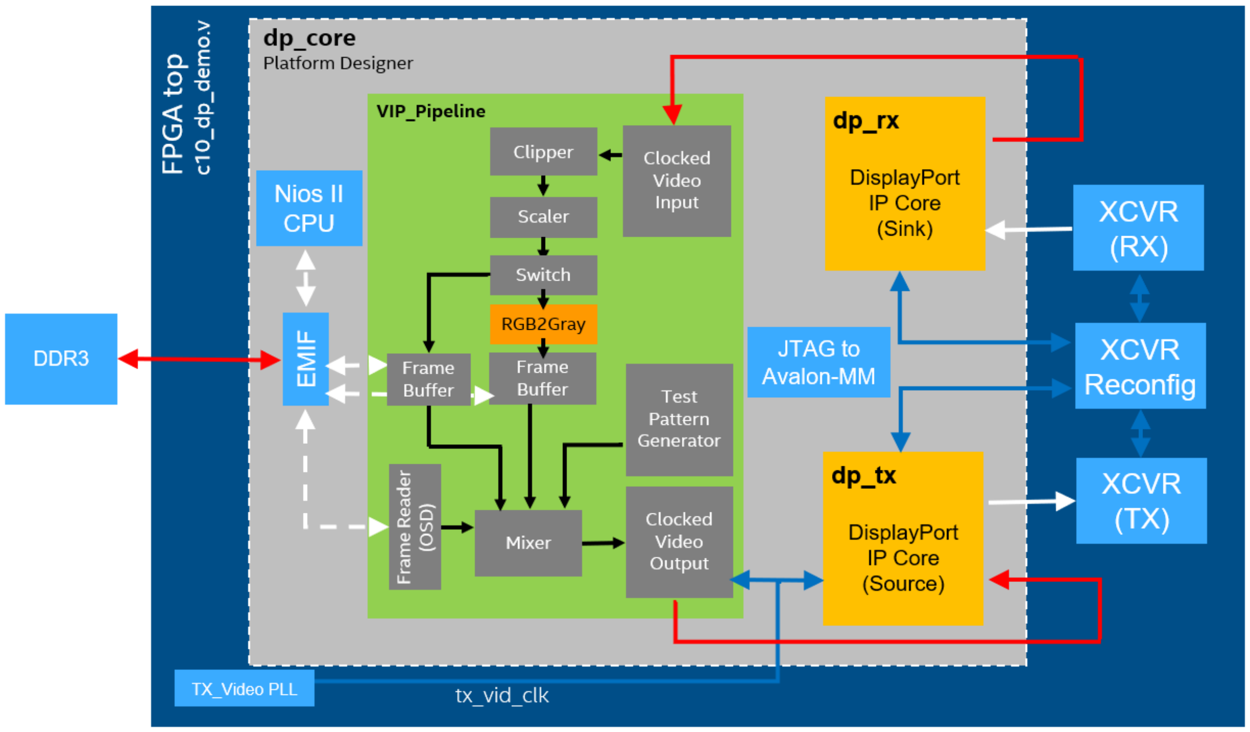
The purpose of this document is to guide you through the process of creating the different building blocks and pulling all together to assembly a working application. For more detailed information about all the potential combinations and settings, you can use the following resources:

- 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
- Quartus Prime Pro: Platform Designer User Guide
  https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-qpp-platform-designer.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). We are using a **Switch** VIP core to duplicate the video stream and feed our custom developed **RGB2Gray** in parallel with our original flow. Both are connected to a **Frame_Buffer** prior get mixed in the **Mixer**. The final composition is enabling 2 PiP (Picture in Picture) windows, over a generated plain background. By using an additional **Frame_Buffer** configured as “Frame_Reader_only”, that retrieves graphics information generated by the Nios2 processor running a lightweight graphic library, we are incorporation graphic information in the upper layer. These graphics are over imposed to the final layout using an additional Mixer layer before sent to the **Clocked_Video_Output** component connected to the DisplayPort TX interface.



In this design we have also connected the **Nios2** processor to the EMIF to write the graphical content onto a designated memory area, mapped as video buffer, also accessible by the **Frame_Reader** component.

The **Nios2** processor is writing this buffer with graphical content information and the **Frame_Reader** component is reading, at the specified display refresh rate (60Hz), the video frame memory to get it finally overlaid on top of our live video in the **Mixer**.

2. Building the hardware pipeline

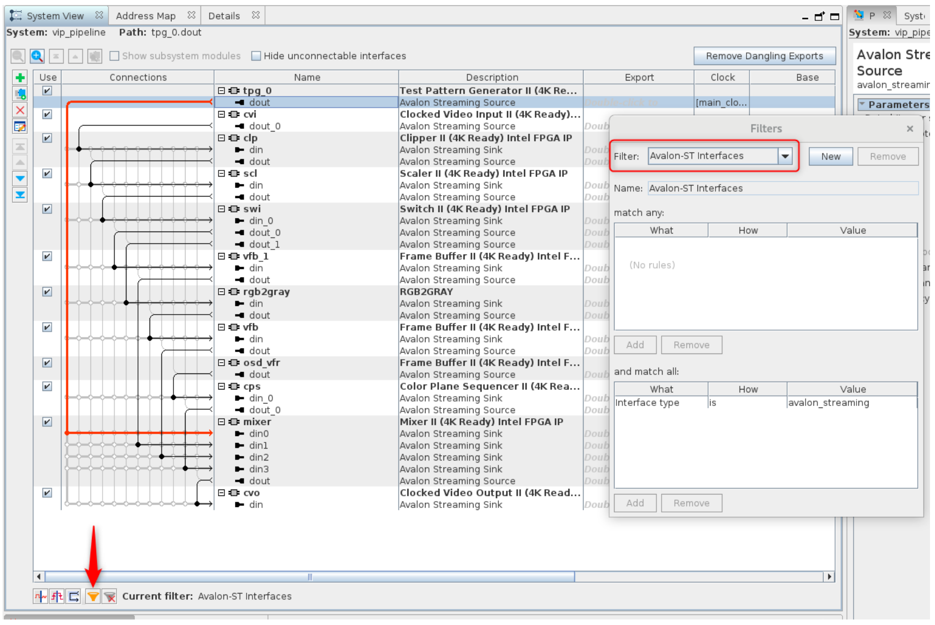
2.1. Setting up the Quartus project

Extract the files provided in the **4_OSD_Overlay_v1.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.
- **custom_ip**: contains our custom module RGB2Gray developed in the previous session

2.2. Understanding the pipeline

Once the Quartus project is open, launch Platform Designer and select `<extracted_folder>/rtl/core/vip_pipeline.qsys` to open. Filter by **Avalon-ST Interfaces** to get a cleaner view on the video data path. In the figure below we have applied the filter and highlighted in red the connection from the test pattern generator (`tpg_0`) to the **din0** `mixer` layer. This connection generates our background for the whole final composition.



System: vip_pipeline **Path:** tpg_0.dout

Use	Name	Description	Clock
✓	tpg_0	Test Pattern Generator II (4K Re...	
	dout	Avalon Streaming Source	[main_clo...
✓	cvi	Clocked Video Input II (4K Ready)...	
	dout_0	Avalon Streaming Source	
✓	clp	Clipper II (4K Ready) Intel FPGA IP	
	din	Avalon Streaming Sink	
	dout	Avalon Streaming Source	
✓	scl	Scaler II (4K Ready) Intel FPGA IP	
	din	Avalon Streaming Sink	
	dout	Avalon Streaming Source	
✓	swi	Switch II (4K Ready) Intel FPGA IP	
	din_0	Avalon Streaming Sink	
	dout_0	Avalon Streaming Source	
	dout_1	Avalon Streaming Source	
✓	vfb_1	Frame Buffer II (4K Ready) Intel F...	
	din	Avalon Streaming Sink	
	dout	Avalon Streaming Source	
✓	rgb2gray	RGB2GRAY	
	din	Avalon Streaming Sink	
	dout	Avalon Streaming Source	
✓	vfb	Frame Buffer II (4K Ready) Intel F...	
	din	Avalon Streaming Sink	
	dout	Avalon Streaming Source	
✓	osd_vfr	Frame Buffer II (4K Ready) Intel F...	
	dout	Avalon Streaming Source	
✓	cps	Color Plane Sequencer II (4K Rea...	
	din_0	Avalon Streaming Sink	
	dout_0	Avalon Streaming Source	
✓	mixer	Mixer II (4K Ready) Intel FPGA IP	
	din0	Avalon Streaming Sink	
	din1	Avalon Streaming Sink	
	din2	Avalon Streaming Sink	
	din3	Avalon Streaming Sink	
	dout	Avalon Streaming Source	
✓	cvo	Clocked Video Output II (4K Read...	
	din	Avalon Streaming Sink	

Current filter: Avalon-ST Interfaces

Filter: Avalon-ST Interfaces

Name: Avalon-ST Interfaces

match any:

What	How	Value
(No rules)		

and match all:

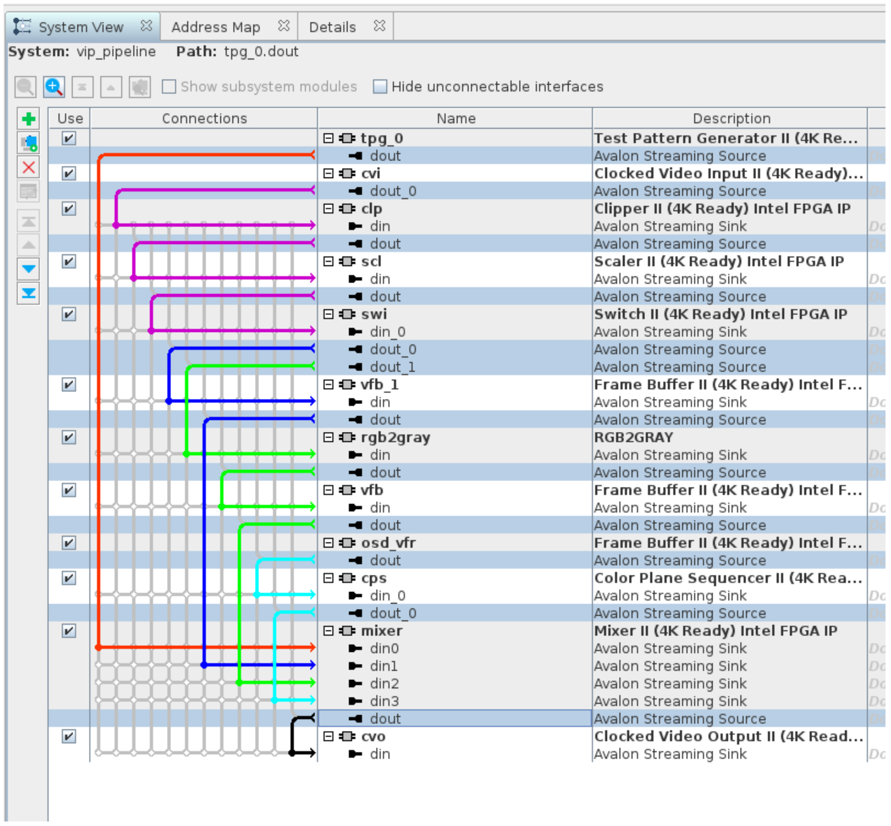
What	How	Value
Interface type	is	avalon_streaming

System: vip_pipe

Avalon Stre
Source
avalon_streamin

In the figure below we have coloured the different branches used in our pipeline implementation:

- **Red**: Direct connection from **tpg_0:dout** to **mixer:din0**
    - This generates our full screen plain colour background
- **Purple**: Input ingest video from **cvi->clp->scl->swi**
    - We capture the live video from the DisplayPort source and scale down until reach the switch for duplication
- **Dark Blue**: Original video source from **swi:dout_0** to **vfb_1** to **mixer:din1**
    - We will enable a window to display (PiP) the original and scaled captured video
- **Green**: Duplicated source video stream that goes through our custom RGB2Gray.
  swi:dout_1 to **rgb2gray** to **vfb** to **mixer:din2**
    - We use a secondary video stream connected to an additional mixer layer to generate another video window (PiP) to display the original/scaled video source but processed by our custom component which converts it to grayscale.
    - We will have a side-by-side positioning of both original and processed flows.
- **Cyan**: On Screen Display information retrieved from the external DDR3 bank. This graphic content has been previously written by the Nios2 CPU.
    - **vfr_osd** is a frame buffer component configured as frame reader only. It creates Avalon-ST video stream from memory content and feeds a Color Plane Sequencer block to re-arrange pixel color and alpha components. We will see more on this later.


System View | Address Map | Details

System: vip_pipeline **Path:** tpg_0.dout

☐ Show subsystem modules ☐ Hide unconnectable interfaces

Use	Connections	Name	Description
☑		⊟ tpg_0	Test Pattern Generator II (4K Re...
		dout	Avalon Streaming Source
☑		⊟ cvi	Clocked Video Input II (4K Ready)...
		dout_0	Avalon Streaming Source
☑		⊟ clp	Clipper II (4K Ready) Intel FPGA IP
		din	Avalon Streaming Sink
		dout	Avalon Streaming Source
☑		⊟ scl	Scaler II (4K Ready) Intel FPGA IP
		din	Avalon Streaming Sink
		dout	Avalon Streaming Source
☑		⊟ swi	Switch II (4K Ready) Intel FPGA IP
		din_0	Avalon Streaming Sink
		dout_0	Avalon Streaming Source
		dout_1	Avalon Streaming Source
☑		⊟ vfb_1	Frame Buffer II (4K Ready) Intel F...
		din	Avalon Streaming Sink
		dout	Avalon Streaming Source
☑		⊟ rgb2gray	RGB2GRAY
		din	Avalon Streaming Sink
		dout	Avalon Streaming Source
☑		⊟ vfb	Frame Buffer II (4K Ready) Intel F...
		din	Avalon Streaming Sink
		dout	Avalon Streaming Source
☑		⊟ osd_vfr	Frame Buffer II (4K Ready) Intel F...
		dout	Avalon Streaming Source
☑		⊟ cps	Color Plane Sequencer II (4K Rea...
		din_0	Avalon Streaming Sink
		dout_0	Avalon Streaming Source
☑		⊟ mixer	Mixer II (4K Ready) Intel FPGA IP
		din0	Avalon Streaming Sink
		din1	Avalon Streaming Sink
		din2	Avalon Streaming Sink
		din3	Avalon Streaming Sink
		dout	Avalon Streaming Source
☑		⊟ cvo	Clocked Video Output II (4K Read...
		din	Avalon Streaming Sink

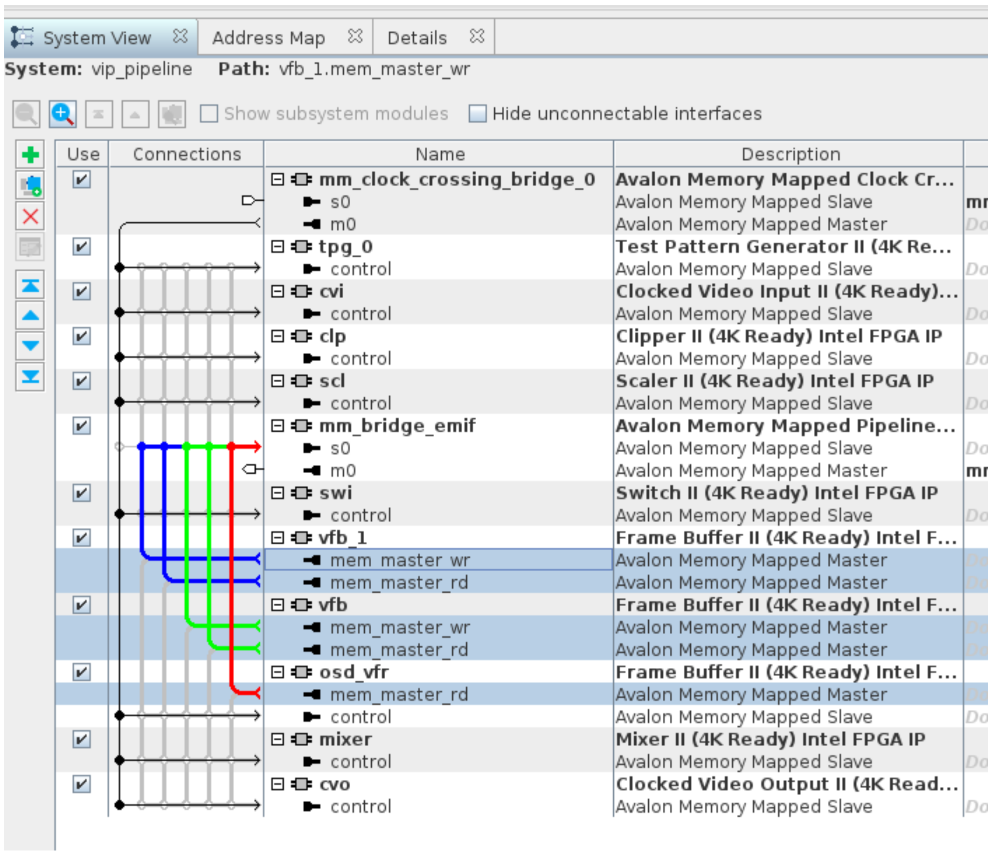
By filtering by Avalon-MM interfaces in Platform Designer, we can observe how the different Frame Buffer components are connected to the pipeline bridge (`mm_bridge_emif`) to share the External Memory Interface controller to get access to the external DDR3 bank for video buffering.

Please note that Platform Designer automatically generates arbitration logic to, effectively, allow this time sharing bandwidth among all the components.

In this architecture we have 2 Frame Buffer components that are writing and reading video frames, as they are dealing with the input frames on one side to write into the memory and reading back those frames as requested by the mixer.

- **Dark Blue**: Original video source from **swi:dout_0 -> vfb_1 -> mixer:din1**
- **Green**: Duplicated source video stream that goes from **swi:dout_1** through our custom **rgb2gray->vfb->mixer:din2**
- **Red**: Avalon-MM master used by the Frame Buffer, configured as "reader_only". Its purpose is to read back video frames from memory and generate Avalon-ST stream to be connected to the **mixer:din3**

Please note that the Frame Reader exposes only 1 Avalon-MM master interface for reading as doesn't require to write. We will see a more detailed configuration next.



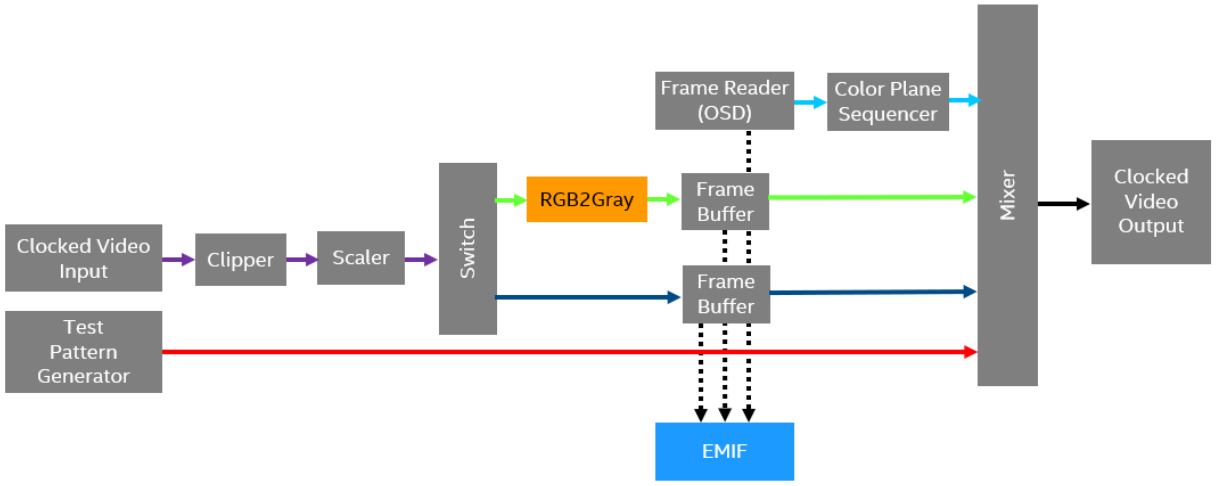
System View | Address Map | Details

System: vip_pipeline **Path:** vfb_1.mem_master_wr

☐ Show subsystem modules ☐ Hide unconnectable interfaces

Use	Connections	Name	Description	
✓		**mm_clock_crossing_bridge_0**	**Avalon Memory Mapped Clock Cr...**	
		s0	Avalon Memory Mapped Slave	mr
		m0	Avalon Memory Mapped Master	Do
✓		**tpg_0**	**Test Pattern Generator II (4K Re...**	
		control	Avalon Memory Mapped Slave	Do
✓		**cvi**	**Clocked Video Input II (4K Ready)...**	
		control	Avalon Memory Mapped Slave	Do
✓		**clp**	**Clipper II (4K Ready) Intel FPGA IP**	
		control	Avalon Memory Mapped Slave	Do
✓		**scl**	**Scaler II (4K Ready) Intel FPGA IP**	
		control	Avalon Memory Mapped Slave	Do
✓		**mm_bridge_emif**	**Avalon Memory Mapped Pipeline...**	
		s0	Avalon Memory Mapped Slave	Do
		m0	Avalon Memory Mapped Master	mr
✓		**swi**	**Switch II (4K Ready) Intel FPGA IP**	
		control	Avalon Memory Mapped Slave	Do
✓		**vfb_1**	**Frame Buffer II (4K Ready) Intel F...**	
		mem_master_wr	Avalon Memory Mapped Master	
		mem_master_rd	Avalon Memory Mapped Master	
✓		**vfb**	**Frame Buffer II (4K Ready) Intel F...**	
		mem_master_wr	Avalon Memory Mapped Master	
		mem_master_rd	Avalon Memory Mapped Master	
✓		**osd_vfr**	**Frame Buffer II (4K Ready) Intel F...**	
		mem_master_rd	Avalon Memory Mapped Master	
		control	Avalon Memory Mapped Slave	Do
✓		**mixer**	**Mixer II (4K Ready) Intel FPGA IP**	
		control	Avalon Memory Mapped Slave	Do
✓		**cvo**	**Clocked Video Output II (4K Read...**	
		control	Avalon Memory Mapped Slave	Do

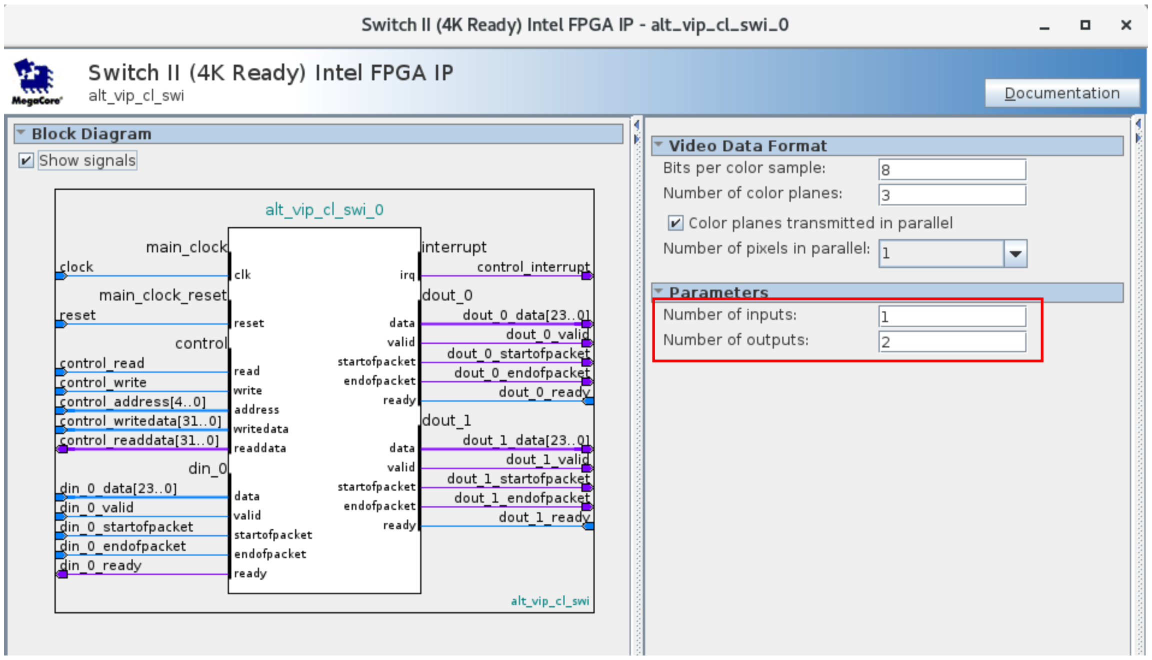
Final diagram representation of the `vip_pipeline.qsys` subsystem



2.3. Switch

In this application we want to do some fancy (and easy to implement) effect. We are duplicating a video stream to apply different processes and finally get them mixed together in a window over the final background. In the VIP suite we have a component (**switch**) able to do this duplication, among other things.

In Platform Designer, open `vip_pipeline.qsys` and go to **IP Catalog->Library->DSP->Video and Image Processing** to select **Switch (4K Ready) Intel FPGA IP**



Video Data Format

Bits per color sample: 8

Number of color planes: 3

☑ Color planes transmitted in parallel

Number of pixels in parallel: 1

Parameters

Number of inputs: 1

Number of outputs: 2

For our application, we parameterize the switch to have **1 input** and **2 outputs**. Again, the Video Data Format is 8 bits per color sample and 3 color planes to be compliant with the rest of the pipeline.

The Switch component comes with a C++ API that we will use while developing our software application that allows us to create your internal input-to-output connection points by writing internal registers. We will route the single input to both output to, actually, create a stream replication.

2.4. DDR3 memory map

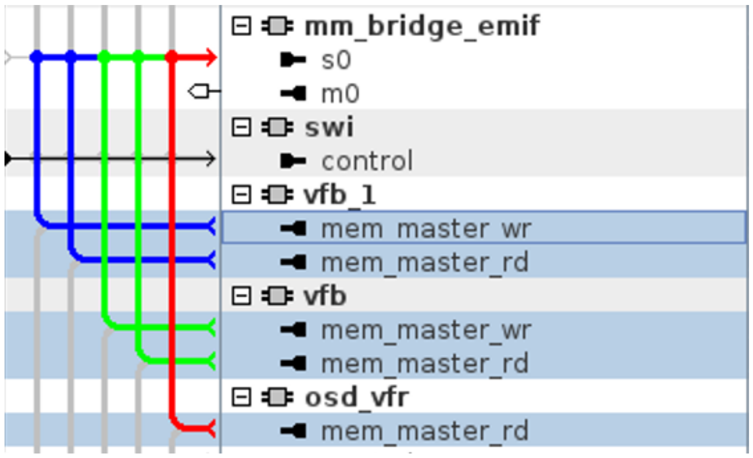
In the Cyclone10 GX Devkit we have 1GB of DDR3 memory space in the available bank. Address range goes from **0x0** to **0x3fff_ffff**. This bank is used by different components so, we need to plan ahead about memory mapping and distribution to ensure non-overlap at run time and proper functionality.

Let's see which are the different components accessing the DDR3, their specific requirements and the memory map implemented in our application.

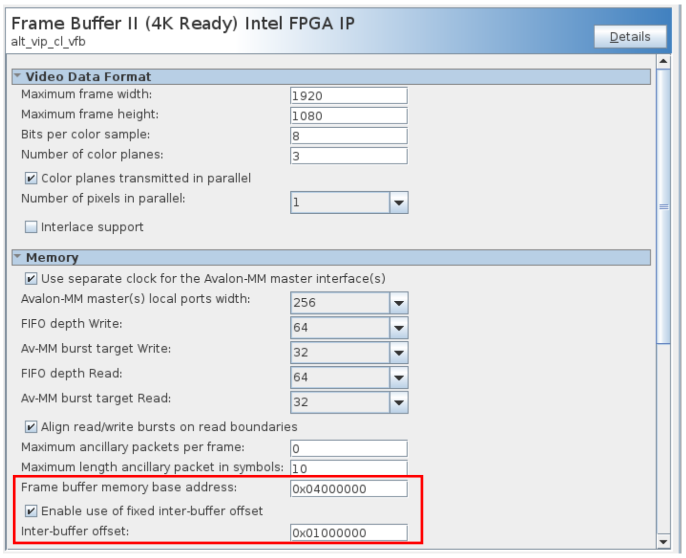
We are analyzing the different blocks we have connected within `vip_pipeline.qsys` subsystem, as well as in the higher hierarchy level `dp_core.qsys`, where we are connecting our Nios2 processor to write and execute programs in the same DDR3 physical bank.

Vip_pipeline.qsys conections

Internally in `vip_pipeline.qsys` we have 3 components connected to the pipeline bridge, that will be finally connected to the DDR3 emif in the `dp_core.qsys` subsystem



- **vfb_1**: this is a frame buffer component, configured in triple buffer mode and it's responsible to buffer and feed data to the `din_1` in the mixer to display the scaled down version of the live video source. Let's have a look at the configuration:


Frame Buffer II (4K Ready) Intel FPGA IP
alt_vip_cl_vfb

Video Data Format

Maximum frame width:	1920
Maximum frame height:	1080
Bits per color sample:	8
Number of color planes:	3
☑ Color planes transmitted in parallel	
Number of pixels in parallel:	1
☐ Interlace support	

Memory

☑ Use separate clock for the Avalon-MM master interface(s)	
Avalon-MM master(s) local ports width:	256
FIFO depth Write:	64
Av-MM burst target Write:	32
FIFO depth Read:	64
Av-MM burst target Read:	32
☑ Align read/write bursts on read boundaries	
Maximum ancillary packets per frame:	0
Maximum length ancillary packet in symbols:	10
Frame buffer memory base address:	0x04000000
☑ Enable use of fixed inter-buffer offset	
Inter-buffer offset:	0x01000000

Recall from previous sessions, that we are using a pixel transport format of 8 *bits per color sample* and 3 *color planes* (RGB). We discussed also how to adjust the Avalon-MM master local ports width to match our configuration and the values for target burst and FIFO depths to maximize efficiency while allowing double internal buffering. If you need a refresh consider go back to the session **1.3_Building_Complete_Pipelines** where we explain all the underlying details about this configuration.

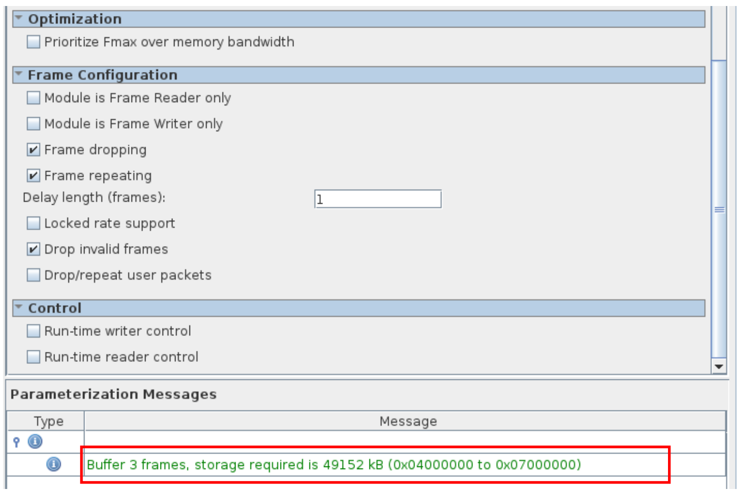
Focus now on the value we set in Frame buffer memory base address setting: **0x04000000**. We need to assign to the frame buffer a base address to use for accessing the memory, and you are responsible of providing addresses which are not conflicting together with other components.

We have also enabled the usage on fixed inter-buffer offset of **0x01000000**. This gives us an advantage: the frame buffer typically uses 2 or 3 frames (depending on configuration) that are swapped in real time while writing incoming video into the memory and reading back to deliver to downstream module. Once a frame is processed, the read and write pointers are exchanged to guarantee that there are no aberrations in a single frame. But depending on the dimensions of the frame, the total memory space required might be different. There is nothing wrong with let the frame buffer calculates dynamically the addresses for the 2nd or 3rd frame but, if for any reason, you want to access from another component to that specific area for some custom processing, you might benefit of knowing, in advance, the base addresses used by each intermediate frame. In our case, we are setting this Inter-buffer offset to 0x01000000 which is enough for our largest/worst case of 1920x1080 frame.

As the remaining part of the frame buffer configuration, we have enabled **Frame Dropping** and **Frame Repeating** which creates a triple buffer implementation.

You can see in the parameterization messages that 3 buffers have been allocated with the following base addresses: **0x04000000**, **0x05000000** and **0x06000000** (which ends in **0x07000000**).

That way, we have constant base addresses for all the frames regardless the actual resolution running through.



Optimization

☐ Prioritize Fmax over memory bandwidth

Frame Configuration

☐ Module is Frame Reader only
☐ Module is Frame Writer only
☑ Frame dropping
☑ Frame repeating
Delay length (frames): 1
☐ Locked rate support
☑ Drop invalid frames
☐ Drop/repeat user packets

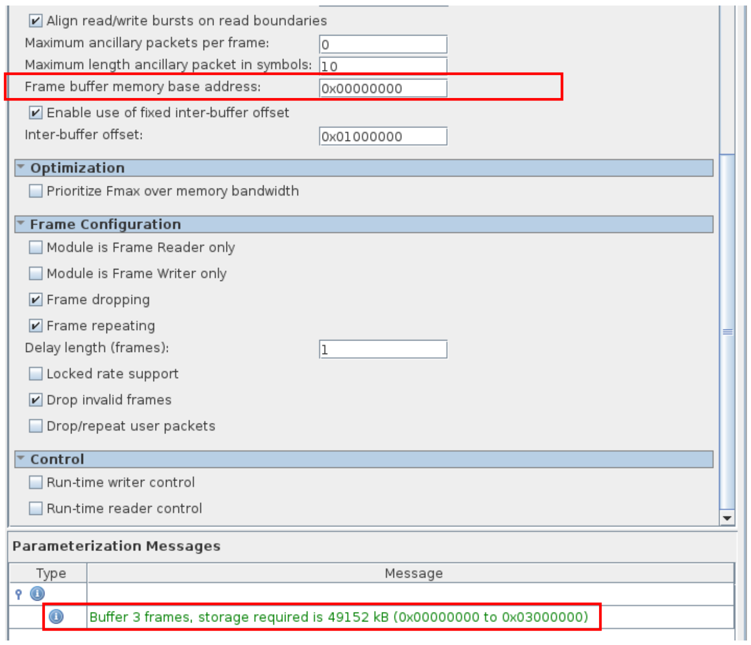
Control

☐ Run-time writer control
☐ Run-time reader control

Parameterization Messages

Type	Message
ⓘ	Buffer 3 frames, storage required is 49152 kB (0x04000000 to 0x07000000)

- **vfb**: this is another frame buffer component, configured in triple buffer mode and it's responsible to buffer and feed data to the **din_2** in the mixer to display **RGB2Gray** conversion applied by our custom module. We are placing them side-by-side in the final layout for comparison. The configuration is exactly the same as the previous **vfb_1** module but with only one modification: The memory base address which is **0x00000000** instead of 0x04000000



☑ Align read/write bursts on read boundaries
Maximum ancillary packets per frame: 0
Maximum length ancillary packet in symbols: 10
Frame buffer memory base address: 0x00000000
☑ Enable use of fixed inter-buffer offset
Inter-buffer offset: 0x01000000

Optimization

☐ Prioritize Fmax over memory bandwidth

Frame Configuration

☐ Module is Frame Reader only
☐ Module is Frame Writer only
☑ Frame dropping
☑ Frame repeating
Delay length (frames): 1
☐ Locked rate support
☑ Drop invalid frames
☐ Drop/repeat user packets

Control

☐ Run-time writer control
☐ Run-time reader control

Parameterization Messages

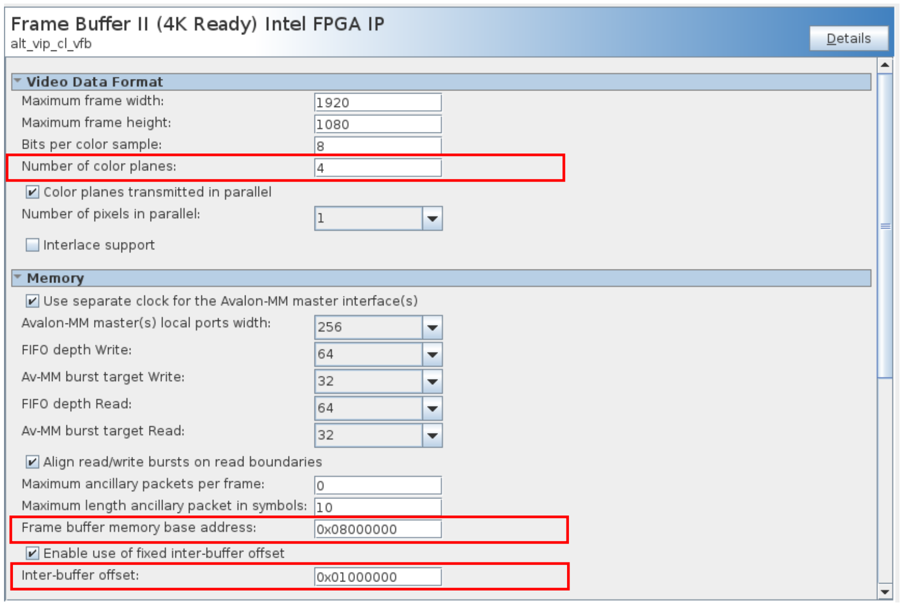
Type	Message
ⓘ	Buffer 3 frames, storage required is 49152 kB (0x00000000 to 0x03000000)

That way, our frames for this component will be buffered at **0x00000000**, **0x01000000** and **0x02000000** which are completely independent on the previous one -> no overlap.

- **osd_vfr**: this is a frame buffer component but configured in a slightly different way as before. In this case we are using it only as a frame reader as the purpose is to retrieve information from a certain memory area to convert to Avalon-ST protocol and feed the **din_3** in the mixer. The information will be written by our Nios2 processor that is using a graphic library to include text strings overlay and simple geometrical shapes on top of live video.

  Please note that this component only has 1 Avalon-MM master interface for reading instead of having 2 for read/write as a regular frame buffer.

  Let's have a look at the configuration:



Frame Buffer II (4K Ready) Intel FPGA IP
alt_vip_cl_vfb

Video Data Format

Maximum frame width:	1920
Maximum frame height:	1080
Bits per color sample:	8
Number of color planes:	4
☑ Color planes transmitted in parallel	
Number of pixels in parallel:	1
☐ Interlace support	

Memory

☑ Use separate clock for the Avalon-MM master interface(s)	
Avalon-MM master(s) local ports width:	256
FIFO depth Write:	64
Av-MM burst target Write:	32
FIFO depth Read:	64
Av-MM burst target Read:	32
☑ Align read/write bursts on read boundaries	
Maximum ancillary packets per frame:	0
Maximum length ancillary packet in symbols:	10
Frame buffer memory base address:	0x08000000
☑ Enable use of fixed inter-buffer offset	
Inter-buffer offset:	0x01000000

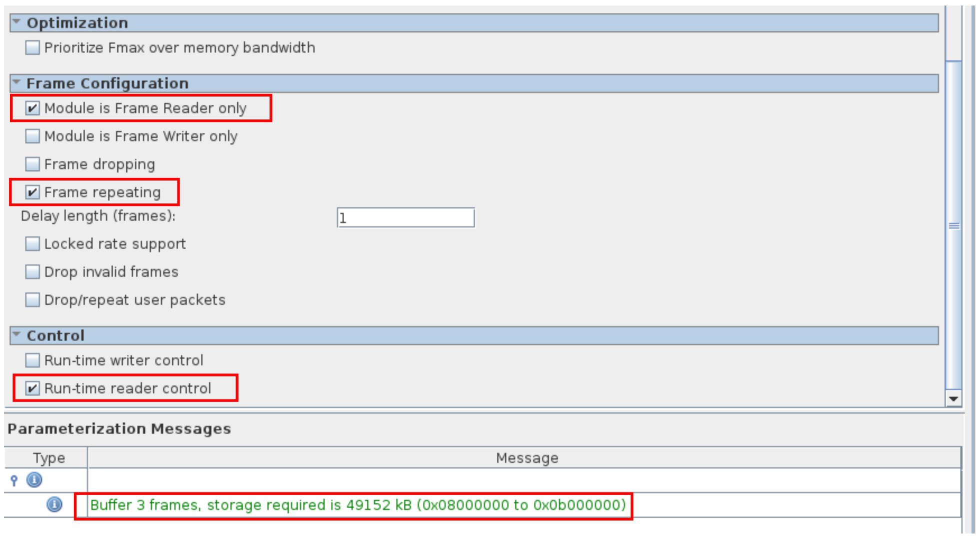
First thing we noticed is the Number of color planes is 4 instead of 3, why? We will have a longer discussion later but, as a sneak view, is because the graphic library we use in Nios2 has transparency capabilities and manages an alpha channel, so the pixels written into the memory are of the format **ARGB** being Alpha the MSB. To leverage this Alpha component we need to read back the full 4 bytes from the memory and handle the A (Alpha) component in the proper way in the mixer. We will discuss over this later.

We have also set the base address to **0x08000000** to avoid any overlap with previous frame buffer components.

The remaining part of the configuration shows how it's configured as Frame Reader only and we have enabled the Frame Repeating behavior. *This is important as will not require us to write in the control registers that a new frame is available to display.* This will create a huge burden in the host running application and, most of the times, the graphical content displayed is static.

If you configure the frame buffer component as Frame Reader or Frame Writer only modes, it's mandatory to enable the corresponding run-time control to allow the processor setup the run time parameters accordingly.

Note that the 3 allocated frames for this component are at addresses **0x08000000**, **0x09000000** and **0x0a000000**



Optimization

- [ ] Prioritize Fmax over memory bandwidth

Frame Configuration

- [✓] Module is Frame Reader only
- [ ] Module is Frame Writer only
- [ ] Frame dropping
- [✓] Frame repeating

Delay length (frames): 1

- [ ] Locked rate support
- [ ] Drop invalid frames
- [ ] Drop/repeat user packets

Control

- [ ] Run-time writer control
- [✓] Run-time reader control

Parameterization Messages

Type	Message
	Buffer 3 frames, storage required is 49152 kB (0x08000000 to 0x0b000000)

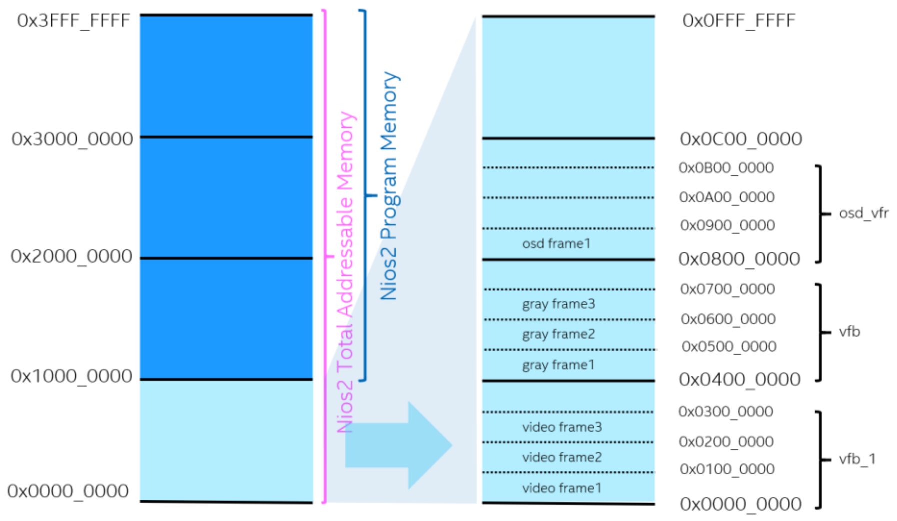
Memory map

In the figure below we show how the total DDR3 memory space is sliced into the different masters. We will see shortly how the Nios2 processor can address the full space (with the data master) but we limit the program memory (instruction master) to a region non occupied by the video frame buffers.

Indeed, we can see how the 2 frame buffers and 1 reader are set to non-overlap regions.

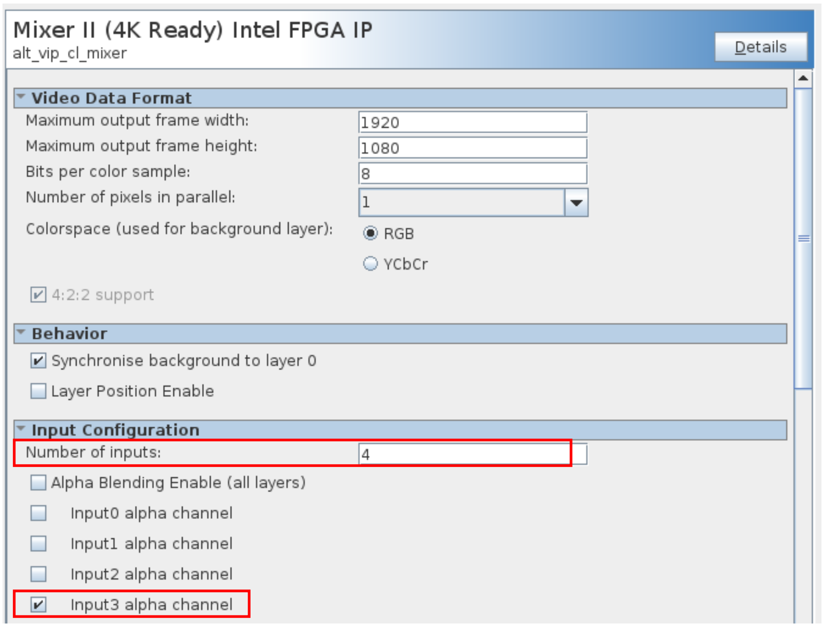
vfb_1 and **vfb** are components completely isolated that works standalone for reading and writing frames, so they have their dedicated memory area.

The **osd_vfr** component only reads frame information from the **osd_frame_1** located **@0x08000000**. This address is also accessible by the Nios2 CPU that will write graphic content to be mixed with the video. We will see later how the processor can access here.



2.5. Mixer configuration

We discussed already that our mixer has to mix now up to 4 layers and there are additional features supported like alpha channel in the OSD generated stream. Let’s dive into the mixer configuration to enable that functionality.


Mixer II (4K Ready) Intel FPGA IP
alt_vip_cl_mixer

Video Data Format

Maximum output frame width:	1920
Maximum output frame height:	1080
Bits per color sample:	8
Number of pixels in parallel:	1
Colorspace (used for background layer):	[✓ RGB] YCbCr

☑ 4:2:2 support

Behavior

☑ Synchronise background to layer 0
☐ Layer Position Enable

Input Configuration

Number of inputs:	4

☐ Alpha Blending Enable (all layers)
☐ Input0 alpha channel
☐ Input1 alpha channel
☐ Input2 alpha channel
☑ Input3 alpha channel

In the above capture we see the enablement of 4 input layers but notice we have also activated the Input3 alpha channel. Let’s go to the VIP user guide to understand the meaning.
https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_vip.pdf

Each input of the mixer (*N*) has an **Input*N* alpha channel** parameter that enables the extra per pixel alpha value for input *n*. When you enable either of the Alpha blending modes, bit [3:2] in the `Input control N` registers control which one of these alpha values is used:

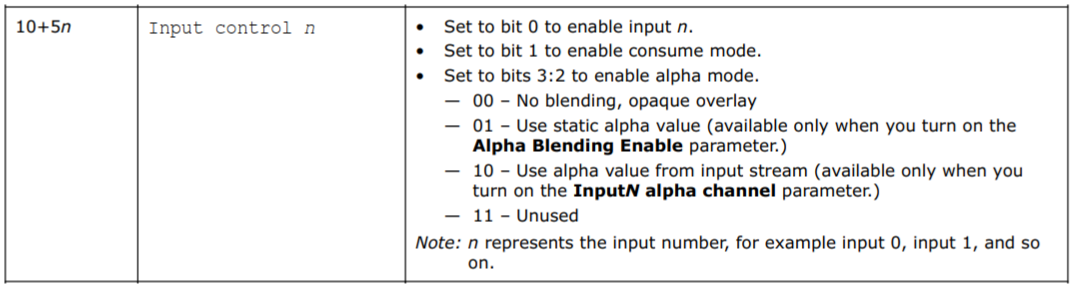
- No blending, fixed opaque alpha value

- Static (run-time programmable) value (only when you select the Alpha Blending Enable parameter)
- **Per-pixel streaming value (only when you select the InputN alpha channel parameter for InputN)**

Note: When you turn the InputN alpha channel parameter, the IP core adds an extra symbol per-pixel for that input. **<u>The least significant symbol is the alpha value</u>**. The control packet is composed of all symbols including alpha.

The valid range of alpha coefficients is 0 to 1, where 1 represents full translucence, and 0 represents fully opaque. The Mixer II IP core determines the alpha value width based on your specification of the bits per pixel per color parameter. For n-bit alpha values, the coefficients range from 0 to 2*n*−1. The model interprets (2*n*− 1) as 1, and all other values as (Alpha value/2*n*. For example, 8-bit alpha value 255>1, 254>254/256, 253>253/256, and so on.

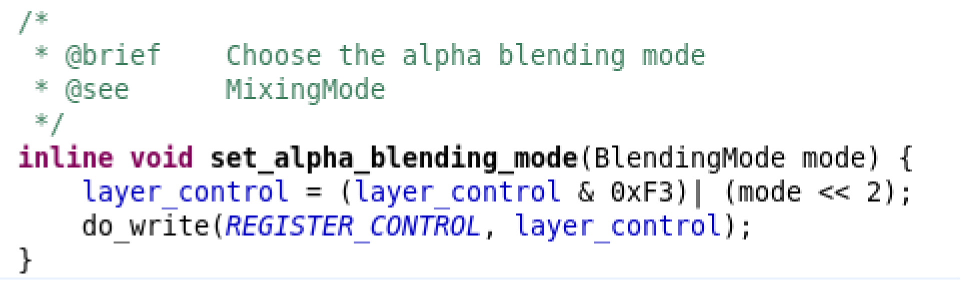
Looking at the mixer register map:



10+5*n*	`Input control n`	• Set to bit 0 to enable input *n*. • Set to bit 1 to enable consume mode. • Set to bits 3:2 to enable alpha mode. — 00 – No blending, opaque overlay — 01 – Use static alpha value (available only when you turn on the **Alpha Blending Enable** parameter.) — 10 – Use alpha value from input stream (available only when you turn on the **Input*N* alpha channel** parameter.) — 11 – Unused *Note:* *n* represents the input number, for example input 0, input 1, and so on.

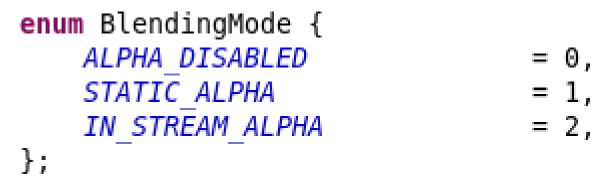
In order to enable the alpha value coming from input stream (that way we are reading a per-pixel alpha value previously written by the Nios2 in the graphic memory) we need to set bits[3:2] in the Input control register corresponding to the layer #3, which is where we have connected our Frame Reader to feed with overlay information.

The VIP C++ API provides access functions to set the run time parameters. Specifically we have them in **MixerLayer.hpp**, the function `set_alpha_blending_mode(BlendingMode mode)`



```
/*
 * @brief    Choose the alpha blending mode
 * @see      MixingMode
 */
inline void set_alpha_blending_mode(BlendingMode mode) {
    layer_control = (layer_control & 0xF3)| (mode << 2);
    do_write(REGISTER_CONTROL, layer_control);
}
```

Takes as parameter an `enum` type `BlendingMode` as per follows:

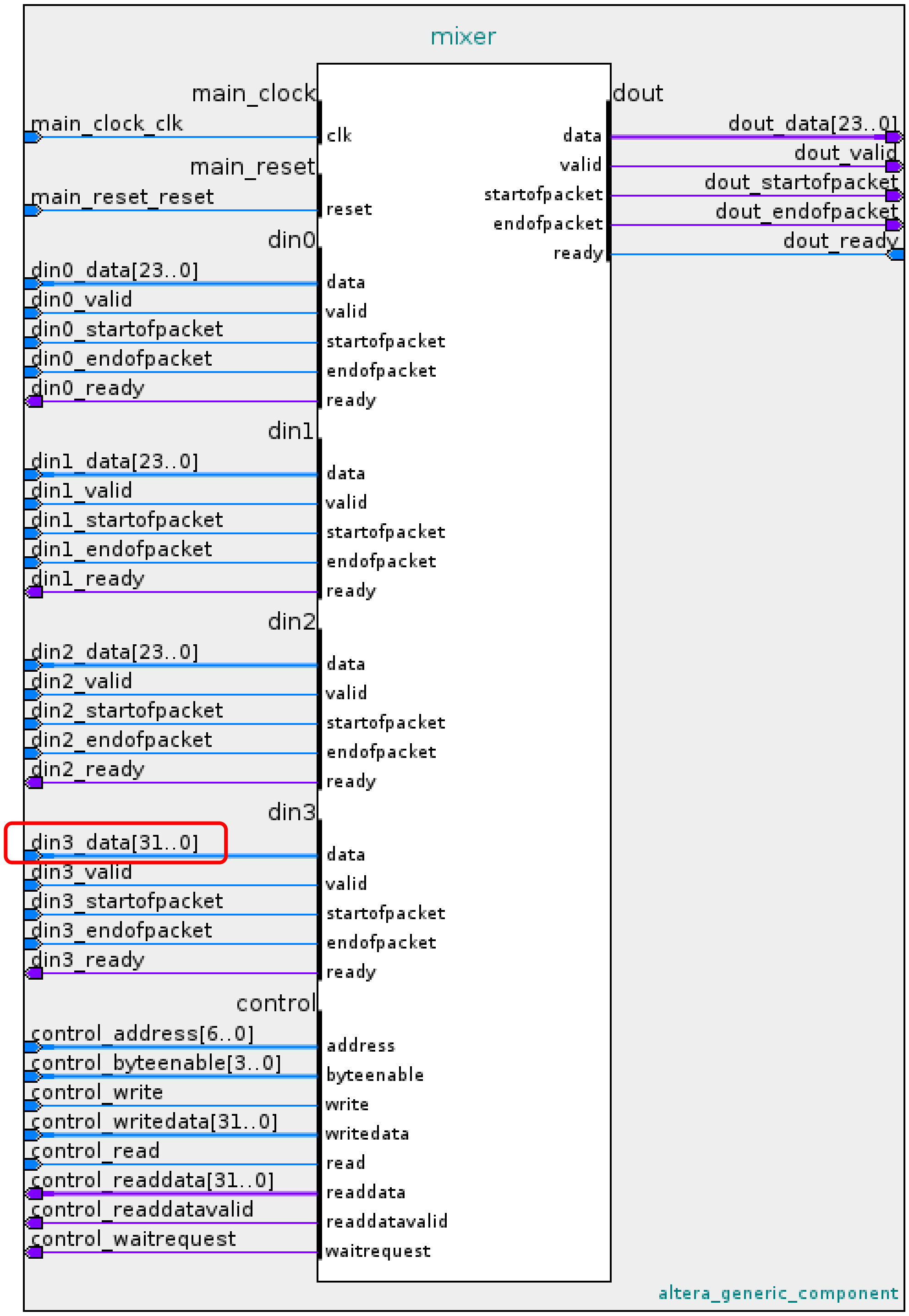


```
enum BlendingMode {
    ALPHA_DISABLED          = 0,
    STATIC_ALPHA            = 1,
    IN_STREAM_ALPHA         = 2,
};
```

So, in our program we can enable the IN_STREAM_ALPHA behavior in the **din_3** of the mixer by issuing:

```
mixer[3].set_alpha_blending_mode(MixerLayer::IN_STREAM_ALPHA);
```

The mixer component we are adding has 4 inputs to mix into a 1 single output. Please notice that **din0**, **din1**, **din2** and **dout** has 3 color planes of 8bpc which means [23:0] data width. However, as we have enabled 4 color planes (to allow an alpha channel), **din3** has [31:0] as data width.



Also, looking at the VIP user guide we need to pay attention to the following:

***Note**: When you turn the InputN alpha channel parameter, the IP core adds an extra symbol per-pixel for that input. **The least significant symbol is the alpha value**. The video packet is composed of all symbols including alpha.*

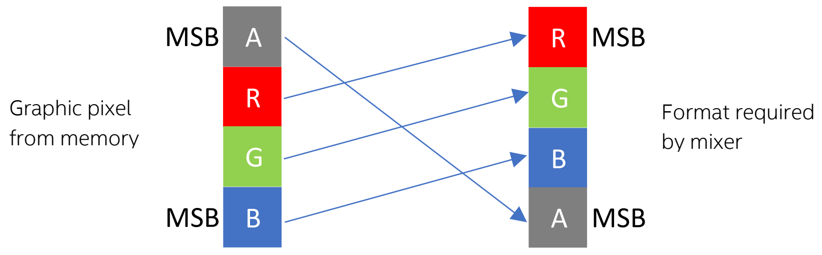
This indicates us that, for din3[31:0] the color plane mapping is the following:

Din3[31:0] = Red[31:24], Green[23:16], Blue[15:8], Alpha[7:0]

There is nothing wrong with this association, however this needs to match with the pixel information the graphic library running on Nios2 is writing on the frame memory. In this case (we will see more details later), the graphic library uses the following color scheme to write pixels in the memory:

Pixel_into_mem[31:0] = Alpha[31:24], Red[23:16], Green[15:8], Blue[7:0]

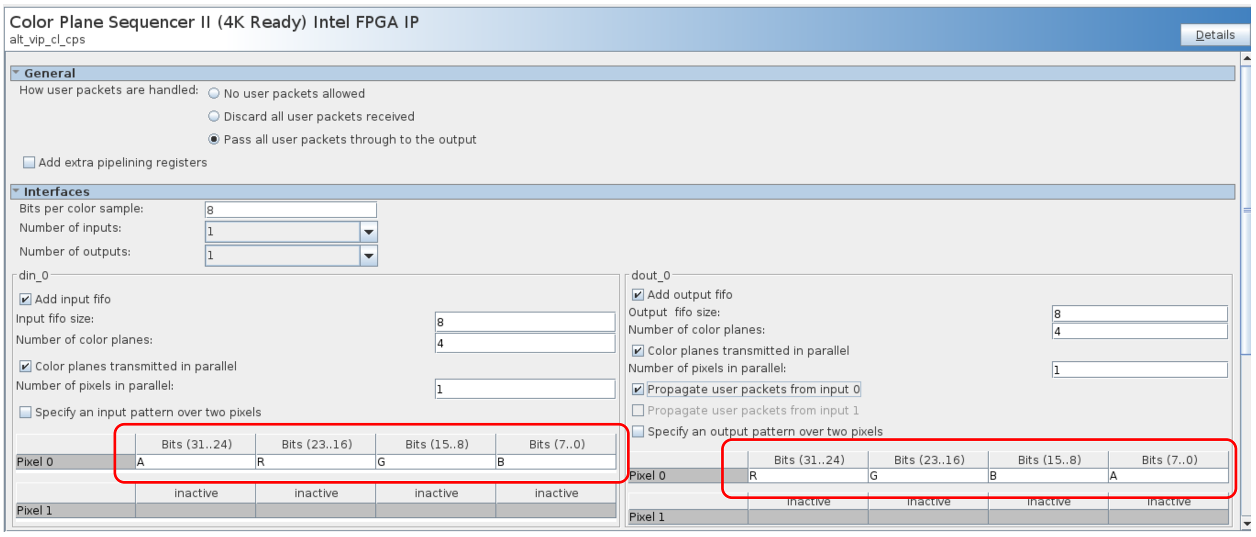
So, we need to make some arrangements to align what was written in the graphic memory to correctly process and blend with the other layers in the mixer.



2.6.Color Plane sequencer

Fortunately, in the VIP suite we have a component that can implement, specifically, this color components re-arrangement: the **Color Plane Sequencer**

We configure the CPS as having 1 input and 1 output and we declare it as of 4 color planes. Then we can rename the different color planes by meaningful names (ARGB) and just do the mapping from the pixels coming into the input (from the **osd_vfr:dout**) to the **mixer:din3** as we described in the previous paragraph.



Color Plane Sequencer II (4K Ready) Intel FPGA IP
alt_vip_cl_cps

Details

General

How user packets are handled: No user packets allowed / Discard all user packets received / [✓ Pass all user packets through to the output]

☐ Add extra pipelining registers

Interfaces

Bits per color sample: 8
Number of inputs: 1
Number of outputs: 1

din_0

☑ Add input fifo
Input fifo size: 8
Number of color planes: 4
☑ Color planes transmitted in parallel
Number of pixels in parallel: 1
☐ Specify an input pattern over two pixels

	Bits (31..24)	Bits (23..16)	Bits (15..8)	Bits (7..0)
Pixel 0	A	R	G	B
	inactive	inactive	inactive	inactive
Pixel 1				

dout_0

☑ Add output fifo
Output fifo size: 8
Number of color planes: 4
☑ Color planes transmitted in parallel
Number of pixels in parallel: 1
☑ Propagate user packets from input 0
☐ Propagate user packets from input 1
☐ Specify an output pattern over two pixels

	Bits (31..24)	Bits (23..16)	Bits (15..8)	Bits (7..0)
Pixel 0	R	G	B	A
	inactive	inactive	inactive	inactive
Pixel 1				

The output of the **cps** component can be directly connected to the mixer as it's, now, compliant with the expected color plane alignment.

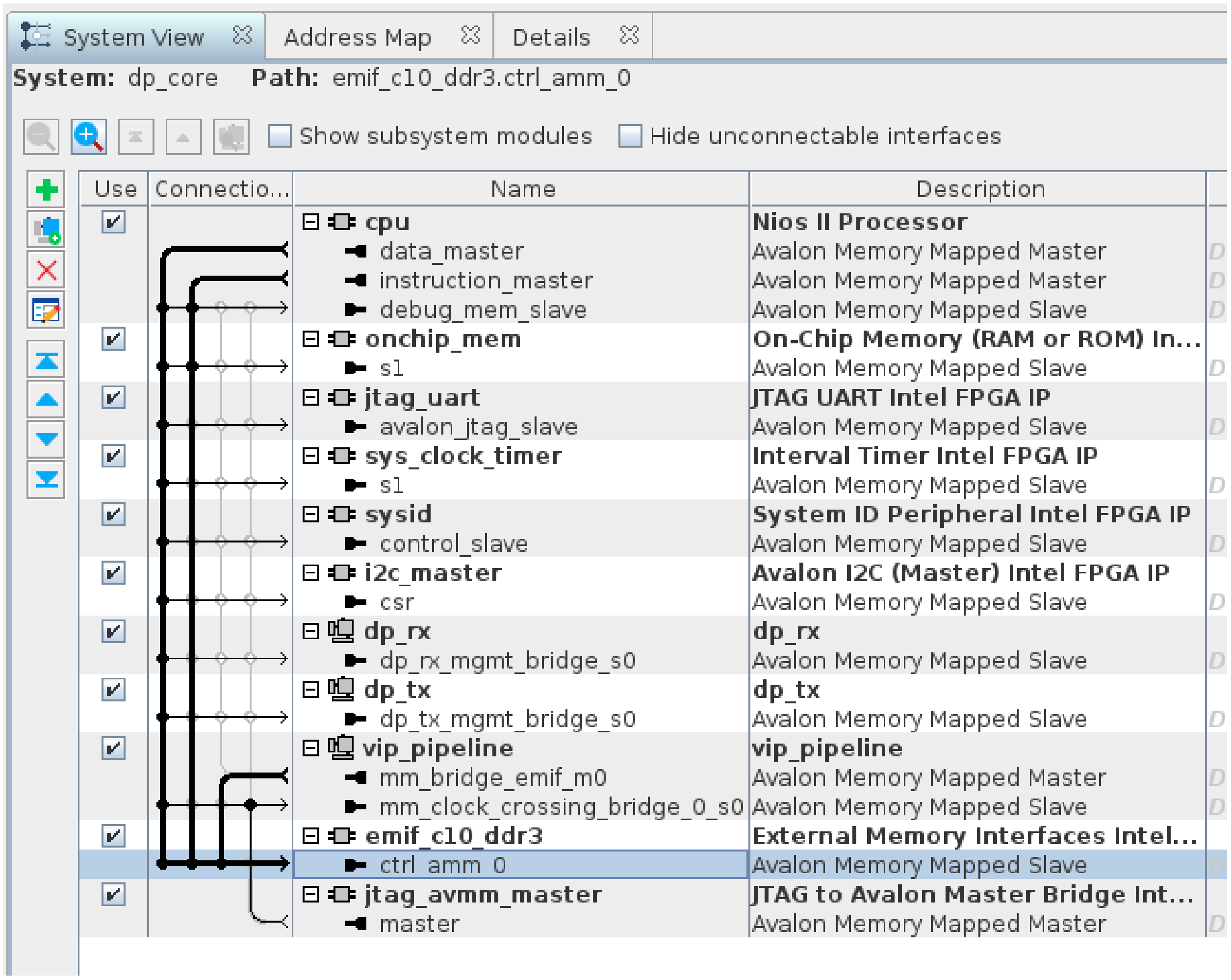
With this we have finished the modifications in the `vip_pipeline.qsys` subsystem. You can now Save & Close.

2.7.Connecting our Nios2 processor to the map

Let's open the higher hierarchy level `dp_core.qsys` to finalize the connections and memory address space assignment.

If we filter by Avalon-MM in the System View on Platform Designer, we see the masters connected to the **emif_c10_ddr3** slave:

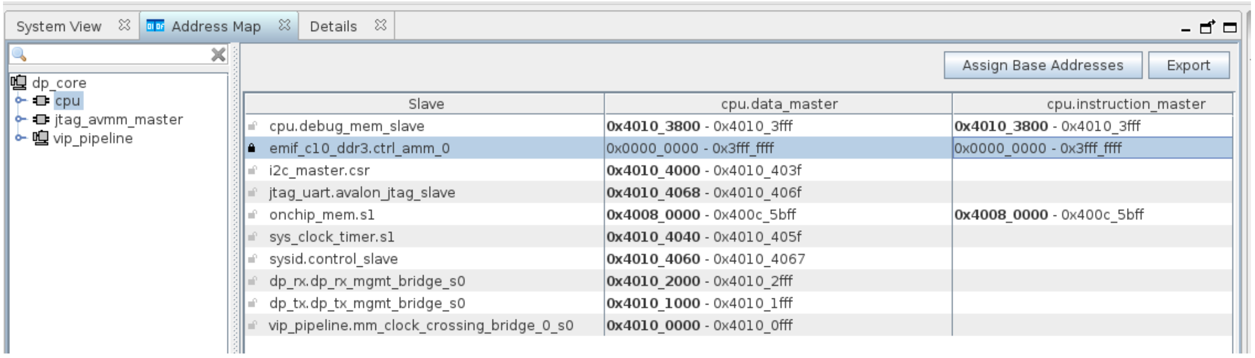
- We have the **vip_pipeline** (all the frame buffers and frame readers explained before)
- The **CPU**: to execute the program and to access to a specific DDR3 region (pointed by **osd_vfr** component 0x08000000



System: dp_core **Path:** emif_c10_ddr3.ctrl_amm_0

Use	Name	Description
✓	**cpu**	**Nios II Processor**
	data_master	Avalon Memory Mapped Master
	instruction_master	Avalon Memory Mapped Master
	debug_mem_slave	Avalon Memory Mapped Slave
✓	**onchip_mem**	**On-Chip Memory (RAM or ROM) In...**
	s1	Avalon Memory Mapped Slave
✓	**jtag_uart**	**JTAG UART Intel FPGA IP**
	avalon_jtag_slave	Avalon Memory Mapped Slave
✓	**sys_clock_timer**	**Interval Timer Intel FPGA IP**
	s1	Avalon Memory Mapped Slave
✓	**sysid**	**System ID Peripheral Intel FPGA IP**
	control_slave	Avalon Memory Mapped Slave
✓	**i2c_master**	**Avalon I2C (Master) Intel FPGA IP**
	csr	Avalon Memory Mapped Slave
✓	**dp_rx**	**dp_rx**
	dp_rx_mgmt_bridge_s0	Avalon Memory Mapped Slave
✓	**dp_tx**	**dp_tx**
	dp_tx_mgmt_bridge_s0	Avalon Memory Mapped Slave
✓	**vip_pipeline**	**vip_pipeline**
	mm_bridge_emif_m0	Avalon Memory Mapped Master
	mm_clock_crossing_bridge_0_s0	Avalon Memory Mapped Slave
✓	**emif_c10_ddr3**	**External Memory Interfaces Intel...**
	ctrl_amm_0	Avalon Memory Mapped Slave
✓	**jtag_avmm_master**	**JTAG to Avalon Master Bridge Int...**
	master	Avalon Memory Mapped Master

Let's open the Address Map tab to understand the architecture

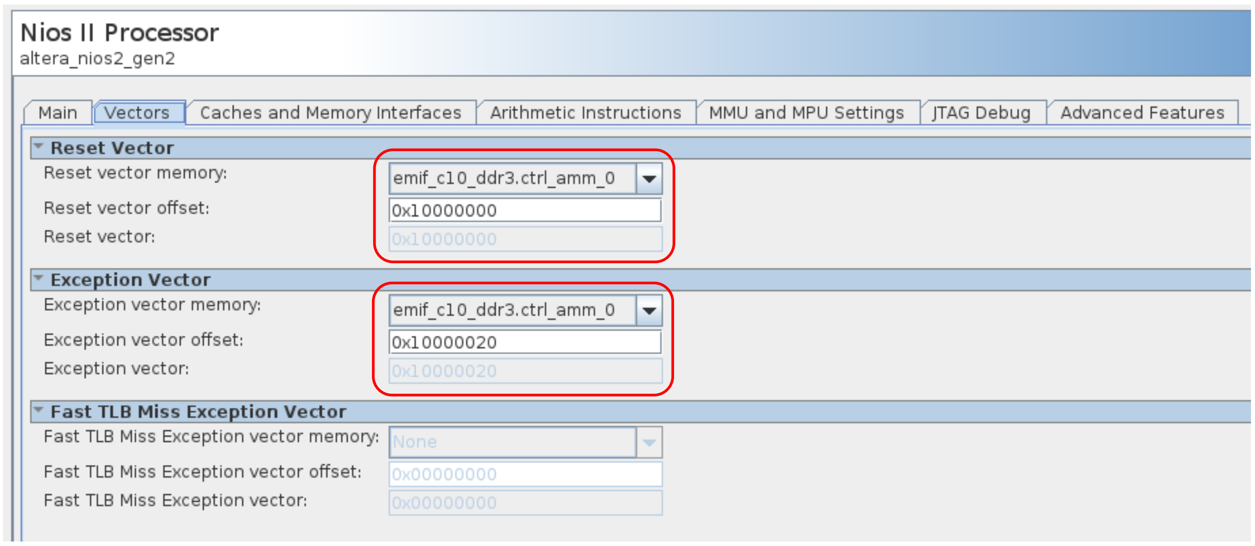


Slave	cpu.data_master	cpu.instruction_master
cpu.debug_mem_slave	**0x4010_3800** - 0x4010_3fff	**0x4010_3800** - 0x4010_3fff
emif_c10_ddr3.ctrl_amm_0	0x0000_0000 - 0x3fff_ffff	0x0000_0000 - 0x3fff_ffff
i2c_master.csr	**0x4010_4000** - 0x4010_403f	
jtag_uart.avalon_jtag_slave	**0x4010_4068** - 0x4010_406f	
onchip_mem.s1	**0x4008_0000** - 0x400c_5bff	**0x4008_0000** - 0x400c_5bff
sys_clock_timer.s1	**0x4010_4040** - 0x4010_405f	
sysid.control_slave	**0x4010_4060** - 0x4010_4067	
dp_rx.dp_rx_mgmt_bridge_s0	**0x4010_2000** - 0x4010_2fff	
dp_tx.dp_tx_mgmt_bridge_s0	**0x4010_1000** - 0x4010_1fff	
vip_pipeline.mm_clock_crossing_bridge_0_s0	**0x4010_0000** - 0x4010_0fff	

The **cpu.data_master** can address the complete 1GB DDR3 memory space. However, we must ensure that our program memory doesn't fall into the memory allocated for the video frame buffers.

Otherwise, our application program will get corrupted once the frame buffer components start to write video frames through. We do this in the **vectors cpu** configuration.

Back in the System View, edit the parameters of the **cpu** by opening the parameterization GUI and move to the **Vectors** tab



Nios II Processor
altera_nios2_gen2

Main | Vectors | Caches and Memory Interfaces | Arithmetic Instructions | MMU and MPU Settings | JTAG Debug | Advanced Features

Reset Vector

Reset vector memory:	emif_c10_ddr3.ctrl_amm_0
Reset vector offset:	0x10000000
Reset vector:	0x10000000

Exception Vector

Exception vector memory:	emif_c10_ddr3.ctrl_amm_0
Exception vector offset:	0x10000020
Exception vector:	0x10000020

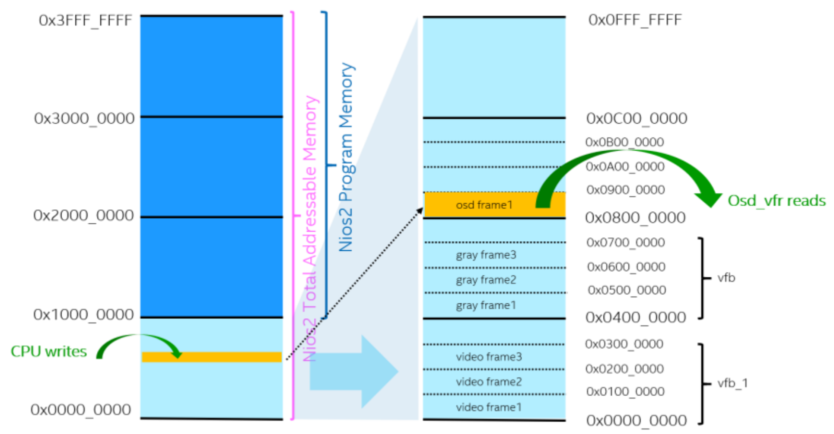
Fast TLB Miss Exception Vector

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

In our case, we set the **Reset vector memory offset** to **0x10000000** and the **Exception vector memory offset** to **0x10000020** to delimit the starting address allocation for our code memory. This is well above the highest address used by the frame buffers for video. In our case, it's the **osd_vfr** that is set to **0x08000000**, **0x0a000000** and **0x0b000000** giving us a worst case of **0x0c000000** for max limit in the upper buffer.

With these directives the linker responsible to map the program we will build for our Nios2 processor will ensure to use addresses higher than **0x10000000** for program symbols (code and data)

Please note that the addressable space for the **cpu:data.master** still remains to the full space (including the address space for the hardware frame buffers) and we will later (in software) create a pointer to get read/write access to the address used by our **osd_vfr**. The purpose is to have graphic content written by the **cpu** in a memory space accessible by the **osd_vfr** to be consumed and converted to Avalon-ST video stream to get mixed (in the **mixer**) with the other live video channels.

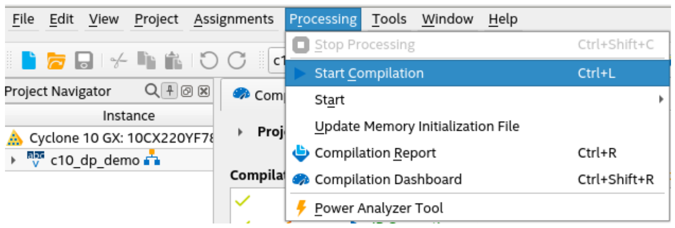


2.8.Generate the bitstream

Save the `dp_core.qsys` subsystem and Generate HDL files.

We don't need to make any modification in the `quartus c10_dp_demo.v` top level file, so we can directly go to **Processing->Start Compilation** in the main toolbar to trigger bitstream compilation.

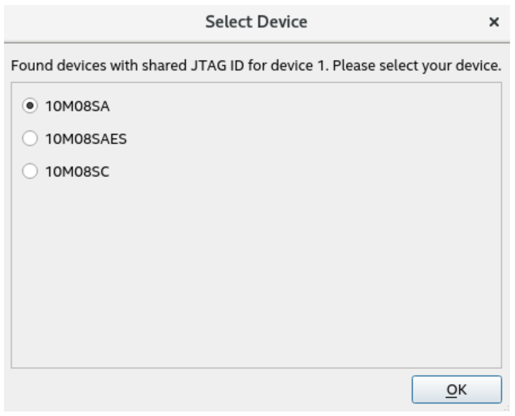
It will take ~15 minutes, depending on machine configuration.


File
Edit
View
Project
Assignments
Processing
Tools
Window
Help
Stop Processing Ctrl+Shift+C
Start Compilation Ctrl+L
Start
Update Memory Initialization File
Compilation Report Ctrl+R
Compilation Dashboard Ctrl+Shift+R
Power Analyzer Tool
Project Navigator
Instance
Cyclone 10 GX: 10CX220YF78
c10_dp_demo

2.9.Configuring the FPGA device

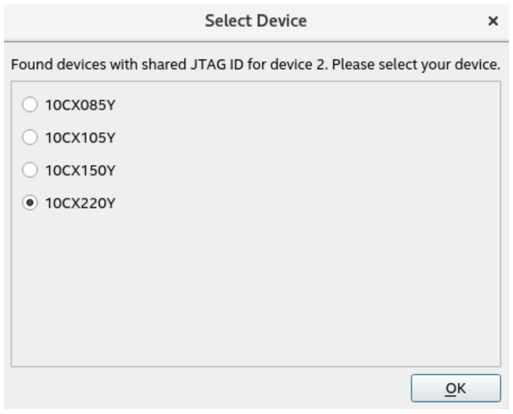
Open the **Quartus Programmer**, select your USB-Blaster cable in the Hardware Setup and click on **Auto Detect** to retrieve the JTAG chain on the Cyclone10 GX Devkit.

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


Select Device

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

[✓ 10M08SA] 10M08SAES 10M08SC

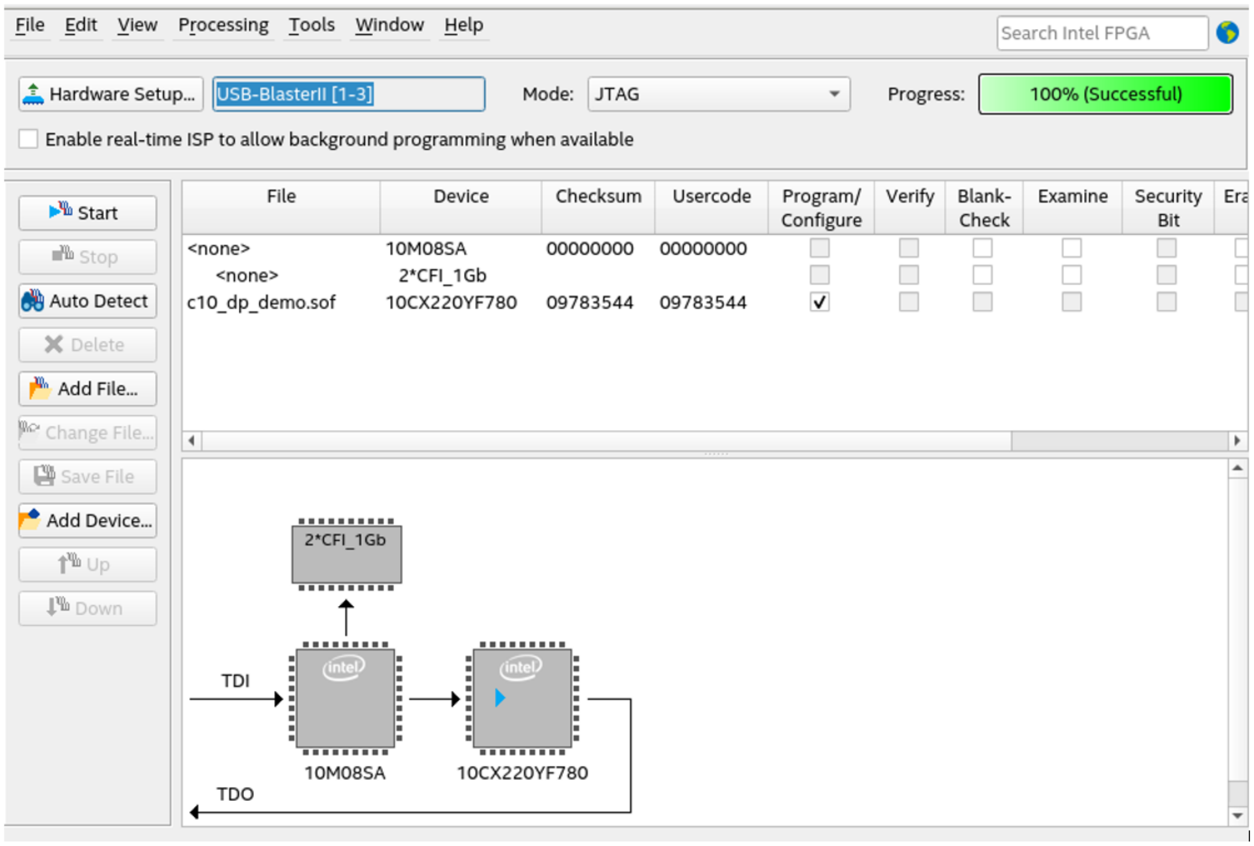

Select Device

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

10CX085Y 10CX105Y 10CX150Y [✓ 10CX220Y]

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

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



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

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

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

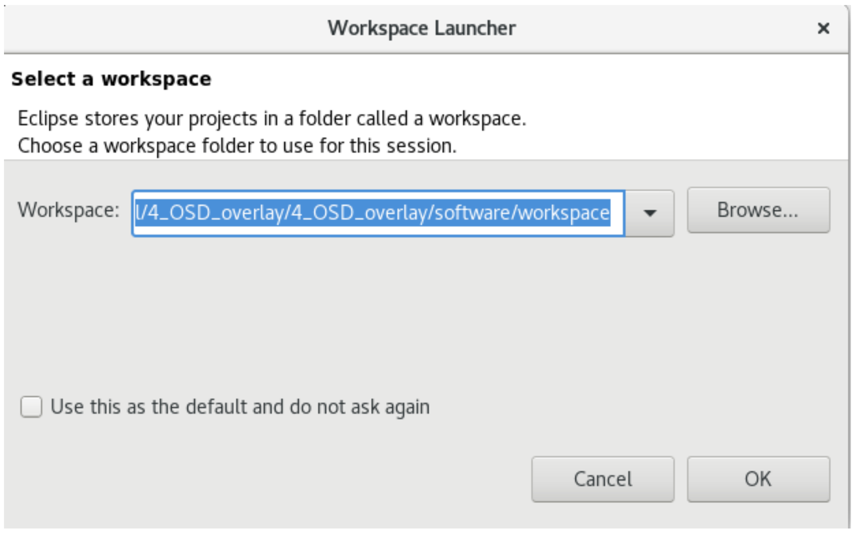
3. Building the software application

3.1.Setting up the Eclipse for Nios project

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


Workspace Launcher

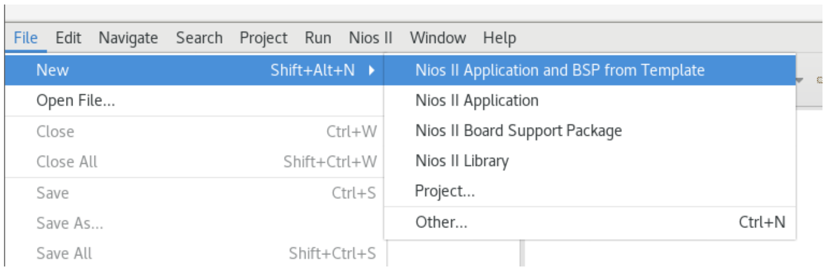
Select a workspace

Eclipse stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: l/4_OSD_overlay/4_OSD_overlay/software/workspace

Use this as the default and do not ask again

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

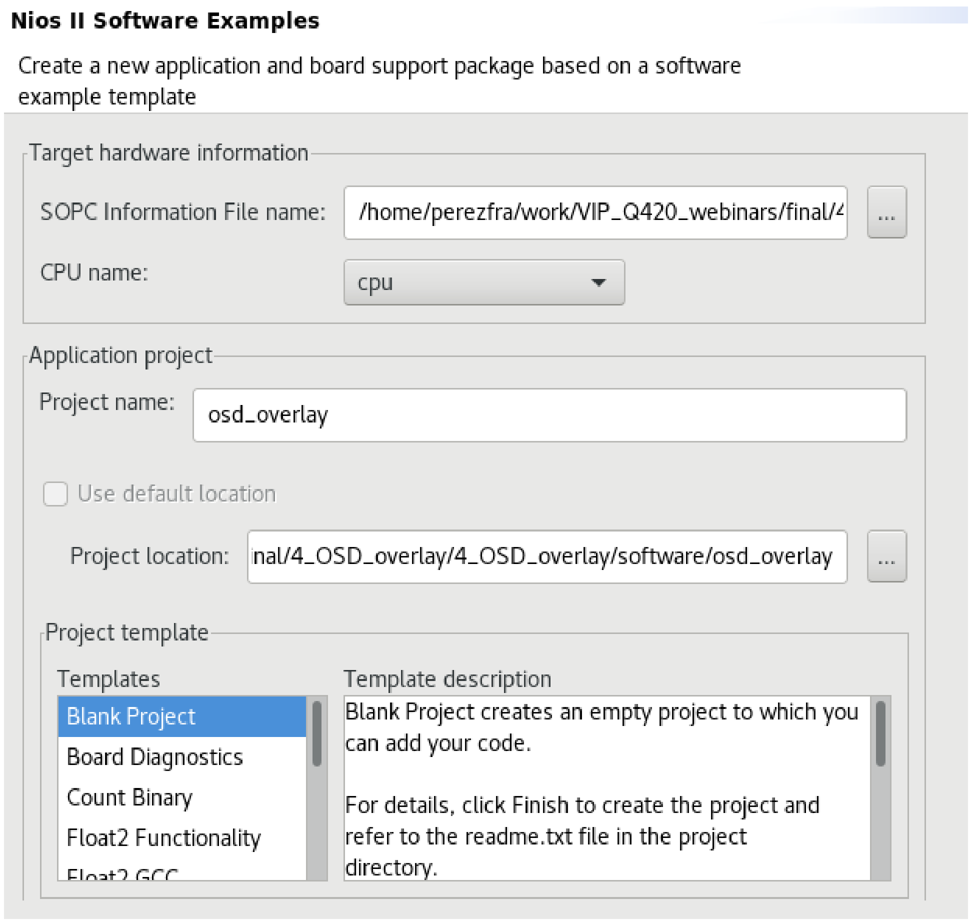


➢ Under **Target hardware information->SOPC Information File name**, you need to select the `*.sopcinfo` file that contains your Nios CPU. In our case is located in *<project_dir>/rtl/core/dp_core/dp_core.sopcinfo*

➢ Under **Application project->Project name** : `osd_overlay`

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

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



Nios II Software Examples

Create a new application and board support package based on a software example template

Target hardware information

SOPC Information File name: /home/perezfra/work/VIP_Q420_webinars/final/4

CPU name: cpu

Application project

Project name: osd_overlay

Use default location

Project location: nal/4_OSD_overlay/4_OSD_overlay/software/osd_overlay

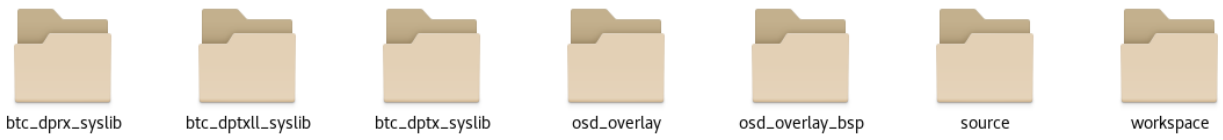
Project template

Template description

Blank Project creates an empty project to which you can add your code.

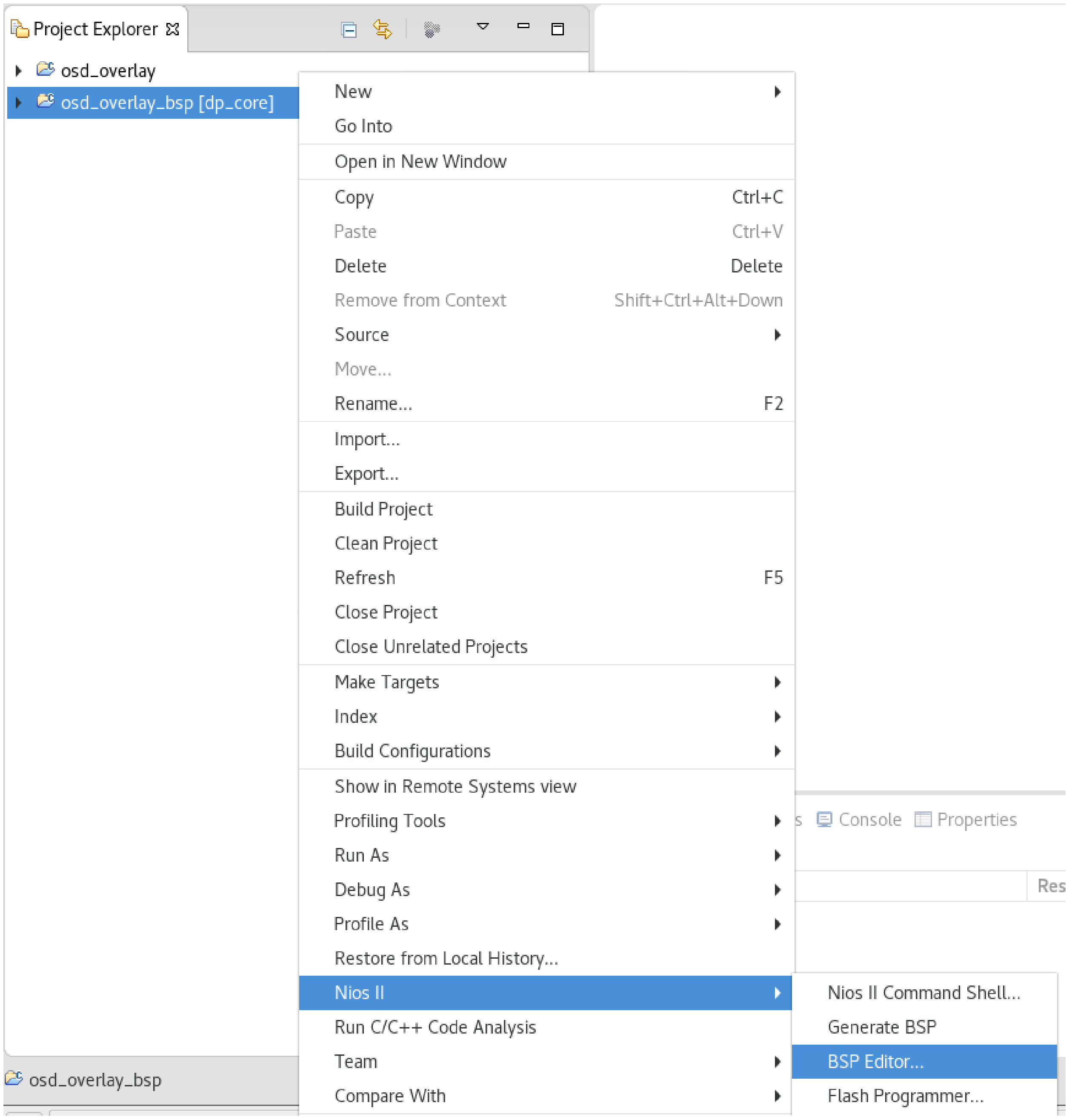
For details, click Finish to create the project and refer to the readme.txt file in the project directory.

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

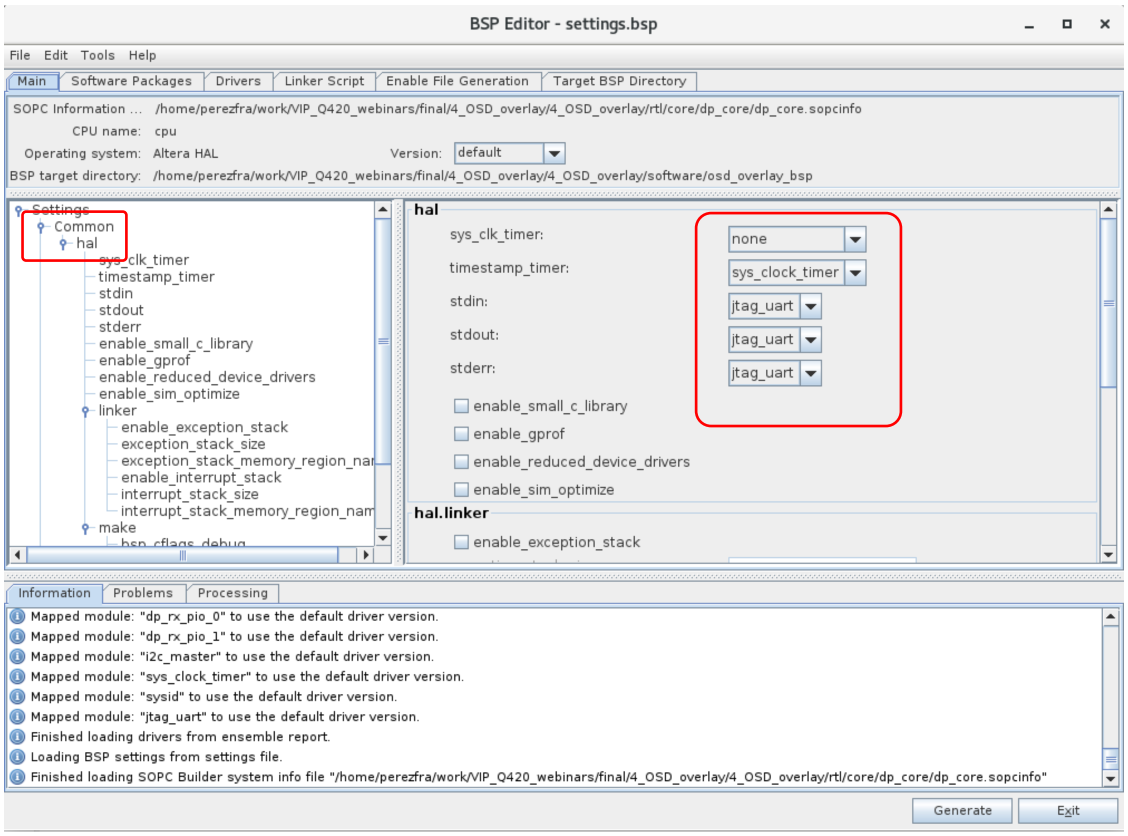


2. **Configure the bsp**.

   In eclipse, right-click on **Project Explorer->custom_component_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

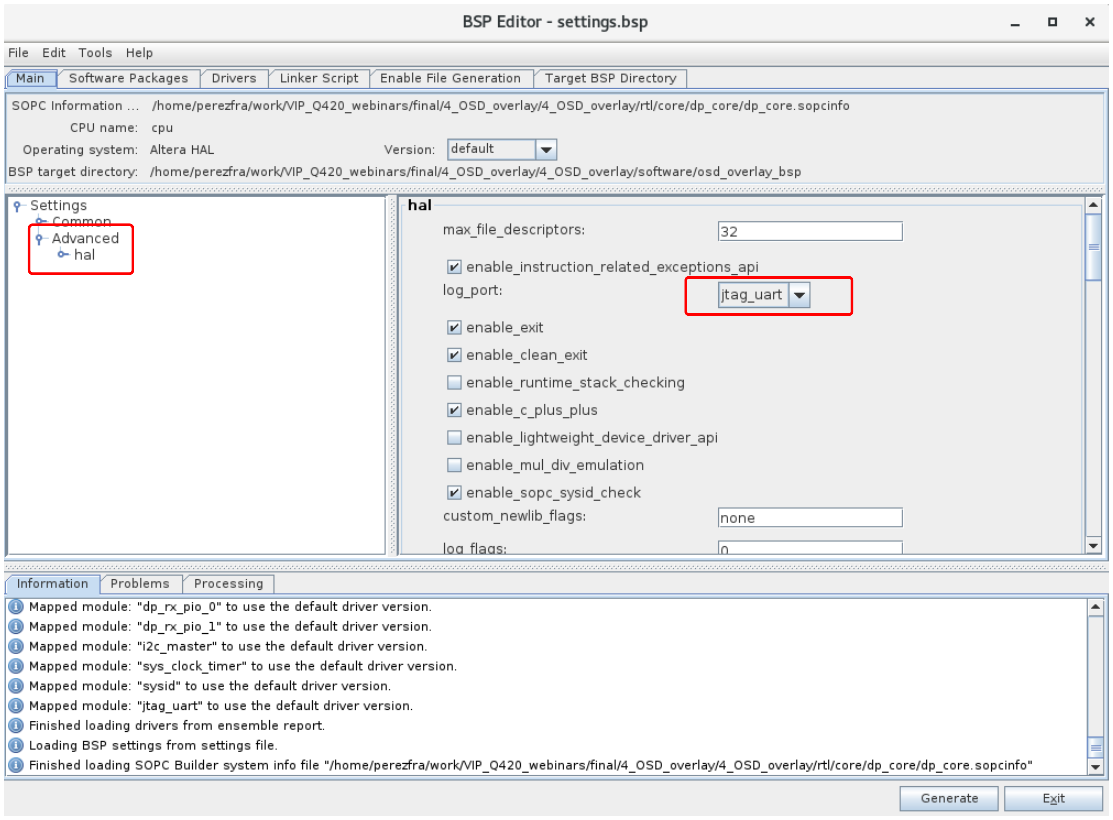


SOPC Information ... /home/perezfra/work/VIP_Q420_webinars/final/4_OSD_overlay/4_OSD_overlay/rtl/core/dp_core/dp_core.sopcinfo
CPU name: cpu
Operating system: Altera HAL Version: default
BSP target directory: /home/perezfra/work/VIP_Q420_webinars/final/4_OSD_overlay/4_OSD_overlay/software/osd_overlay_bsp

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

hal.linker
☐ enable_exception_stack

```
Mapped module: "dp_rx_pio_0" to use the default driver version.
Mapped module: "dp_rx_pio_1" to use the default driver version.
Mapped module: "i2c_master" to use the default driver version.
Mapped module: "sys_clock_timer" to use the default driver version.
Mapped module: "sysid" to use the default driver version.
Mapped module: "jtag_uart" to use the default driver version.
Finished loading drivers from ensemble report.
Loading BSP settings from settings file.
Finished loading SOPC Builder system info file "/home/perezfra/work/VIP_Q420_webinars/final/4_OSD_overlay/4_OSD_overlay/rtl/core/dp_core/dp_core.sopcinfo"
```


BSP Editor - settings.bsp
File Edit Tools Help
Main | Software Packages | Drivers | Linker Script | Enable File Generation | Target BSP Directory
Settings
Common
Advanced
hal
Information | Problems | Processing
Generate
Exit

SOPC Information ... /home/perezfra/work/VIP_Q420_webinars/final/4_OSD_overlay/4_OSD_overlay/rtl/core/dp_core/dp_core.sopcinfo
CPU name: cpu
Operating system: Altera HAL Version: default
BSP target directory: /home/perezfra/work/VIP_Q420_webinars/final/4_OSD_overlay/4_OSD_overlay/software/osd_overlay_bsp

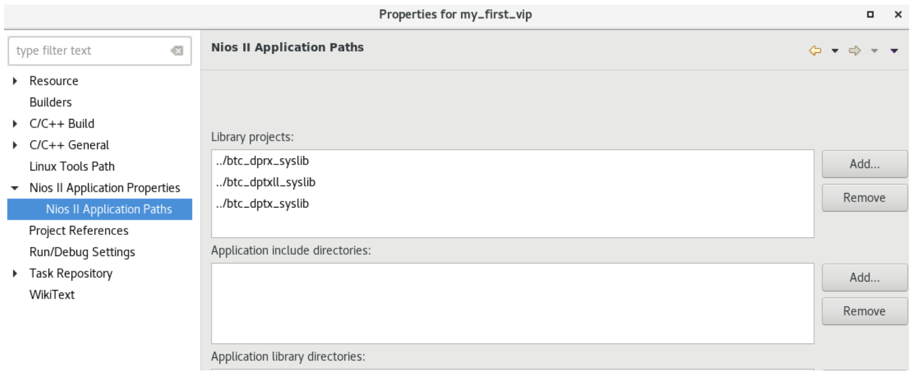
hal
max_file_descriptors: 32
✓ enable_instruction_related_exceptions_api
log_port: jtag_uart
✓ enable_exit
✓ enable_clean_exit
☐ enable_runtime_stack_checking
✓ enable_c_plus_plus
☐ enable_lightweight_device_driver_api
☐ enable_mul_div_emulation
✓ enable_sopc_sysid_check
custom_newlib_flags: none
log_flags: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

```
Mapped module: "dp_rx_pio_0" to use the default driver version.
Mapped module: "dp_rx_pio_1" to use the default driver version.
Mapped module: "i2c_master" to use the default driver version.
Mapped module: "sys_clock_timer" to use the default driver version.
Mapped module: "sysid" to use the default driver version.
Mapped module: "jtag_uart" to use the default driver version.
Finished loading drivers from ensemble report.
Loading BSP settings from settings file.
Finished loading SOPC Builder system info file "/home/perezfra/work/VIP_Q420_webinars/final/4_OSD_overlay/4_OSD_overlay/rtl/core/dp_core/dp_core.sopcinfo"
```

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

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

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



Nios II Application Paths

Library projects:

```
../btc_dprx_syslib
../btc_dptxll_syslib
../btc_dptx_syslib
```

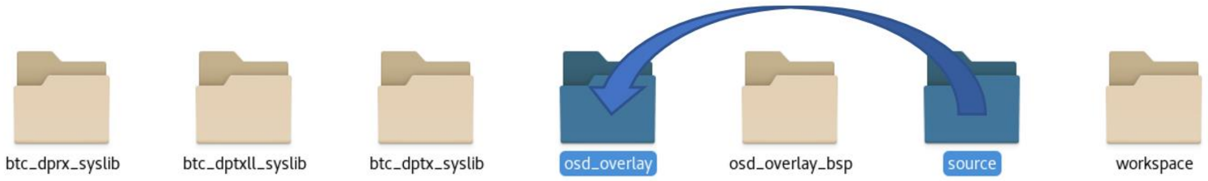
Application include directories:

Application library directories:

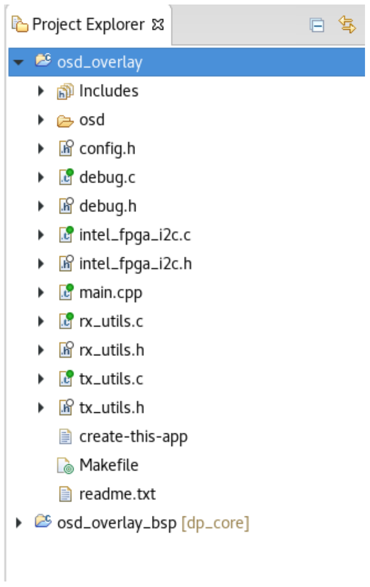
Then click **Apply** and **OK**

3.2.Importing the code

Copy all the source files provided in the `source` folder into `osd_overlay`



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*

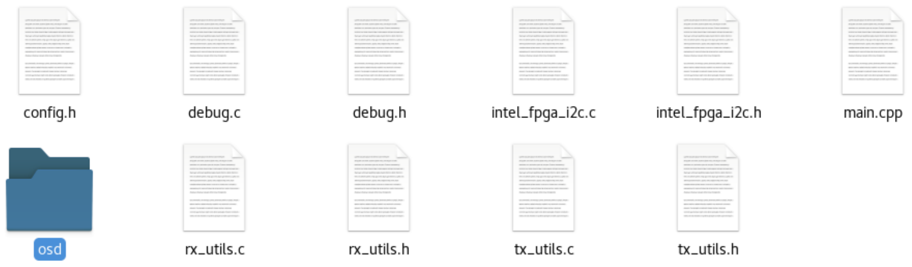


```
osd_overlay
  Includes
  osd
  config.h
  debug.c
  debug.h
  intel_fpga_i2c.c
  intel_fpga_i2c.h
  main.cpp
  rx_utils.c
  rx_utils.h
  tx_utils.c
  tx_utils.h
  create-this-app
  Makefile
  readme.txt
osd_overlay_bsp [dp_core]
```

3.3. Examine the code

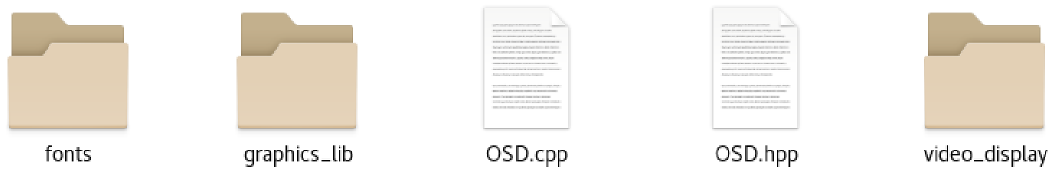
If we open the `source` folder we see the already familiar files we have been using in previous designs: *`debug, config, intel_fpga_i2c, rx_utils, tx_utils`* *and* *`main.cpp`*

But we have now a new folder named **osd**, which contains all the supporting code we need to create graphic content in a memory area that we will convert to a video frame (using the **Frame_Reader** component instantiated in hardware) to superimpose text and graphic shapes on top of live video.



3.4. Exploring the graphic library

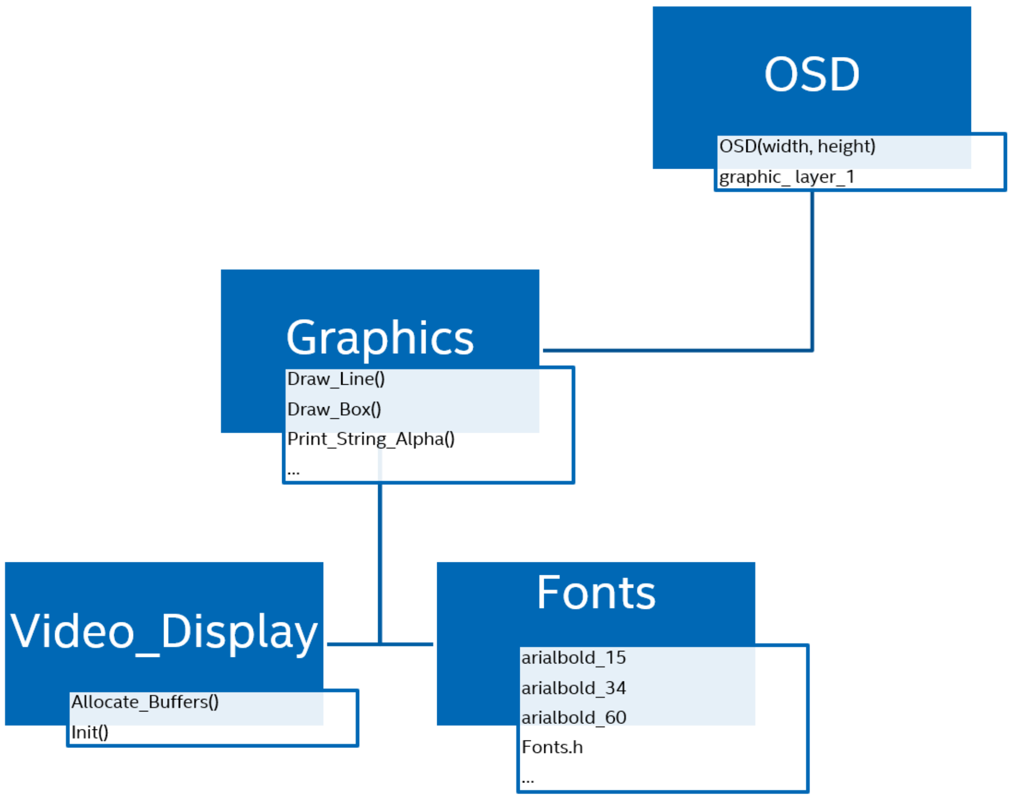
Inside the **osd** folder, we find the following directory tree with the different components which constitute our complete graphic library implementation. The complete on screen display engine is composed by a top level class (**OSD**), which uses a **video_display** class as a mean to abstract a `memory_buffer` and link with the underlying hardware and a **graphic_library** implementing the methods to draw shapes and text strings on that `memory_buffer`. Ultimately, the text strings use the different **fonts** structures provided. Let's have a deep dive!



- **video_display**
  - This is the base class which allocates and maps a certain memory buffer in the system memory (DDR3) and remaps it to the same address location expected by the **osd_vfr** frame reader hardware module.
  - It has methods to reserve, allocate and initialize memory blocks that we will use as frames to draw our overlay content.
- **graphics_lib**
  - This class is based on **video_display** and adds functions and methods to draw graphical content like simple shapes (Lines, Boxes and Rounded_Boxes) and Text Strings and Characters.
  - These functions are represented here as illustration purposes to showcase a simple way to add OSD content to your application, they are not fully optimized versions.

        - For some of the functions we have added a transparency argument to illustrate how you can modify and grow your own library to support more sophisticated alpha blending effects.
    - **fonts**
        - We have included, as a sample, some fonts that can be used with the graphic library to represent text strings. We have added different sizes and font names that can cover many different use cases.
        - Please note that you can print these text strings using your preferred combination of background and foreground colors.
        - NOTE: We have used a specific script with GIMP (https://www.gimp.org/) to extract the fonts. We are planning an upcoming session on how to use gimp to extract and make usable for your application any font installed in your computer.
    - **osd.cpp, osd.hpp**
        - This is the top level class which defines and instantiates an OSD object in the application to allow the generation of graphical content.
        - It has as parameters the width and height of the graphic layer we want to generate. This is typically the same as our output image resolution to have full screen overlay capabilities.

See in the diagram below a logic association among all the classes and components explained above.



3.5.Modifying our main program to configure the new elements

We start adding the corresponding header files of the new components included in the application: **Frame_Reader** and **Switch**.

We also need to make some additional configuration to the mixer to account for the newly added layer for OSD.

In `main.cpp`

```
#include "Frame_Reader.hpp"
#include "Switch.hpp"
```

Configuring the Frame_Reader:

We first create an object of **Frame_Reader** class:

```
Frame_Reader osd_vfr(VIP_PIPELINE_OSD_VFR_BASE, -1);
```

We configure now the resolution of the frame we want to read and the base address where we should start to retrieve data.

```
//osd_vfr.set_frame_information(1920,1080,0);
 IOWR(VIP_PIPELINE_OSD_VFR_BASE, 5, 0x00f00438);  // This is the resolution of our output image
 osd_vfr.set_frame_address(0x8000000);            // Remember in the hardware configuration we set this as
                                                  Frame Buffer Memory Base Address
 osd_vfr.start();                                 // Set the GO bit to start our component
```

NOTE: There is a bug in the current version of the `Frame_Reader` API. The

```
void Frame_Reader::set_frame_information(unsigned int width, unsigned int height, unsigned int interlace_nibble) {
   do_write(VFR_FRAME_INFORMATION, (width & 0x7FF) | ((height & 0x7FF) << 13) | ((interlace_nibble & 0xF) << 26));
}
```

is not correctly implemented in the auto generated `Frame_Reader.cpp` file and produces wrong functionality on the component. It should have been implemented as follows:

```
void Frame_Reader::set_frame_information(unsigned int width, unsigned int height, unsigned int interlace_nibble) {
   do_write(VFR_FRAME_INFORMATION, (width & 0x1FFF) | ((height & 0x1FFF) << 13) | ((interlace_nibble & 0xF) << 26));
}
```

In order to overcome this issue we have 2 options:

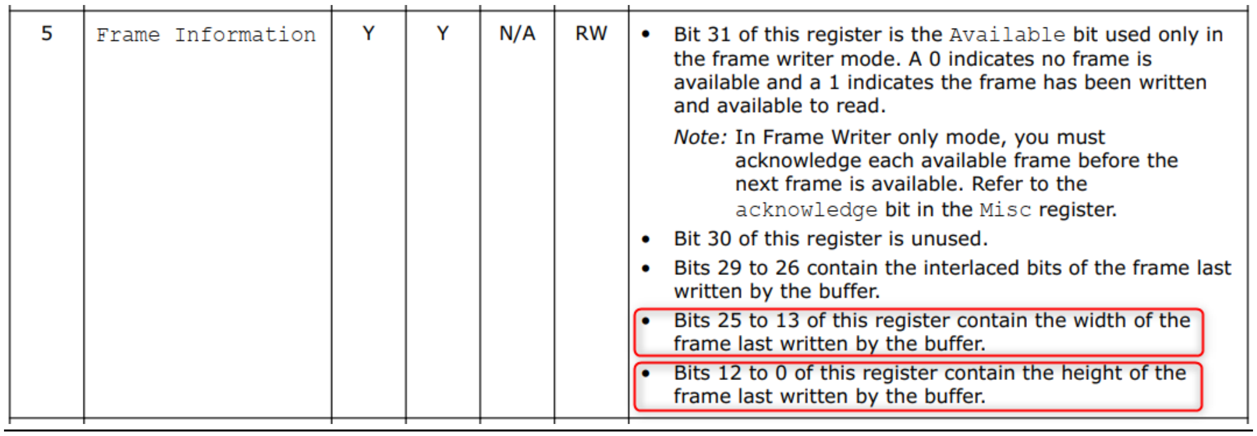
- **Replace the set_frame_information function directly in the bsp:**
  `<project_dir>/software/osdd_overlay_bsp/drivers/vip/src/Frame_Reader.cpp`

  This solution has a caveat, as every time you generate the bsp the file gets automatically reverted to its original configuration and you would need to redo the modification.

- **Use a direct IOWR instruccion:**
  IOWR(VIP_PIPELINE_OSD_VFR_BASE, 5, 0x00f00438);

Please note that in Frame Reader only mode you set the frame dimensions through the `Frame Information` register and the frame start address through the `Frame Start Address` register.

See below the corresponding bits (from the VIP User Guide) within `Frame Information` register to set the parameters.

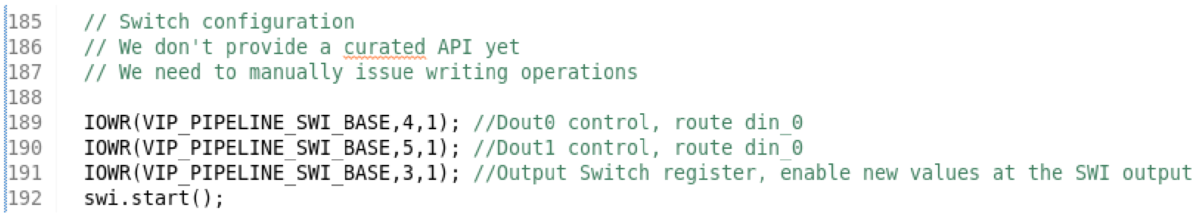


5	`Frame Information`	Y	Y	N/A	RW	• Bit 31 of this register is the `Available` bit used only in the frame writer mode. A 0 indicates no frame is available and a 1 indicates the frame has been written and available to read. *Note:* In Frame Writer only mode, you must acknowledge each available frame before the next frame is available. Refer to the `acknowledge` bit in the `Misc` register. • Bit 30 of this register is unused. • Bits 29 to 26 contain the interlaced bits of the frame last written by the buffer. • Bits 25 to 13 of this register contain the width of the frame last written by the buffer. • Bits 12 to 0 of this register contain the height of the frame last written by the buffer.

Configuring the Switch:

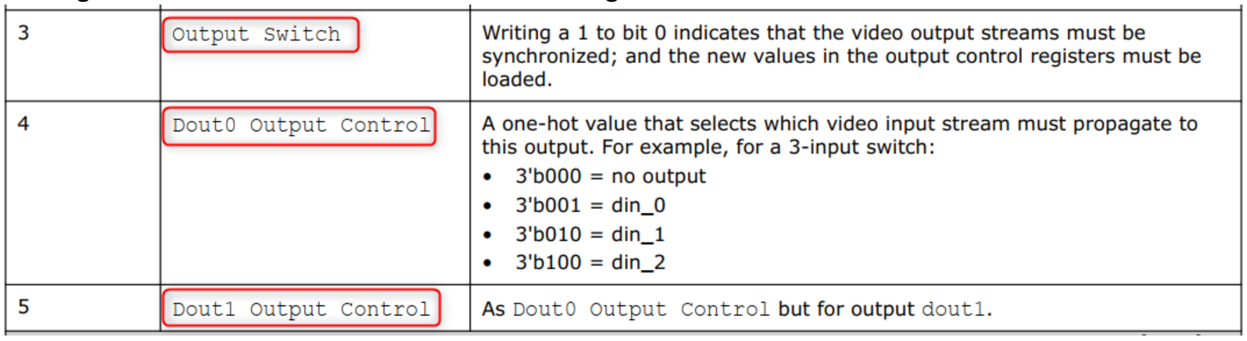
We are using the switch to duplicate the input video stream. We have generated a switch component with 1 input and 2 outputs and we want to route the input flow to both outputs.

The C++ VIP API for the switch is not fully curated yet, so we would need to perform direct IOWR accesses to setup the configuration. The `start()` method is included as all the VIP components are inherited from `VipCore.hpp`



```
// Switch configuration
// We don't provide a curated API yet
// We need to manually issue writing operations

IOWR(VIP_PIPELINE_SWI_BASE,4,1); //Dout0 control, route din_0
IOWR(VIP_PIPELINE_SWI_BASE,5,1); //Dout1 control, route din_0
IOWR(VIP_PIPELINE_SWI_BASE,3,1); //Output Switch register, enable new values at the SWI output
swi.start();
```

Looking at the VIP User Guide we see the Switch registers as follows:



3	`Output Switch`	Writing a 1 to bit 0 indicates that the video output streams must be synchronized; and the new values in the output control registers must be loaded.
4	`Dout0 Output Control`	A one-hot value that selects which video input stream must propagate to this output. For example, for a 3-input switch: • 3'b000 = no output • 3'b001 = din_0 • 3'b010 = din_1 • 3'b100 = din_2
5	`Dout1 Output Control`	As `Dout0 Output Control` but for output `dout1`.

We set the input routed to `Dout0` in address=4, and `Dout1` in address=5.

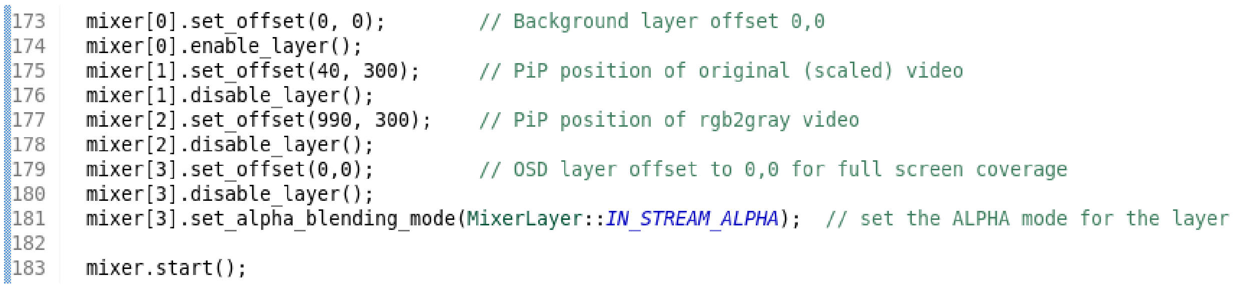
Every time we change the switch configuration we need to write a '1' into the `Bit0` of `Output Switch` register to take effect.

Configuring the Mixer:

As we explained in the hardware section of this manual, we have modified the Mixer to include an additional layer for the video stream coming from the `Frame_Reader` component. We also explained that we have enabled the alpha channel for this layer and we want to have *IN_STREAM_ALPHA* as the stream coming to this port will have 4 color components (**ARGB**). Thus, the required **mixer** configuration is as follows:

Mixer mixer(VIP_PIPELINE_MIXER_BASE, 4); // 4 layers mixer: background, original video, rgb2gray, osd

We need to configure the parameters for the different layers:



```
mixer[0].set_offset(0, 0);        // Background layer offset 0,0
mixer[0].enable_layer();
mixer[1].set_offset(40, 300);     // PiP position of original (scaled) video
mixer[1].disable_layer();
mixer[2].set_offset(990, 300);    // PiP position of rgb2gray video
mixer[2].disable_layer();
mixer[3].set_offset(0,0);         // OSD layer offset to 0,0 for full screen coverage
mixer[3].disable_layer();
mixer[3].set_alpha_blending_mode(MixerLayer::IN_STREAM_ALPHA);  // set the ALPHA mode for the layer

mixer.start();
```

Background layer [0] is always enabled. Video layers [1] & [2] are controlled in run time and enabled when there is active video detected in the DisplayPort input and disabled otherwise. The graphic layer [3] is originally disabled until we get it configured and some graphic content written to display.

NOTE: It's recommended to keep the graphic layer disabled before initialization of the frame reader component, otherwise the frame information is not set and can create corrupted frames to the mixer and create visual artifacts, can even create backpressure in the system making impossible the right visualization. Once the Frame Reader component is set, we can enable the mixer layer and modify "on the fly" the overlay content using the methods in the graphic library.

3.6.Adding graphics to our final image

Once we have all of our video pipeline configured it's time to use the graphic library to generate content to be displayed on top of live video.

We start by creating our OSD object, with the same resolution as our background.

In `main.cpp` we declare an object of OSD class providing the full resolution as parameters for the constructor

OSD **osd**(1920, 1080);

The **OSD** class has some private properties for dimensions and an exposed graphic_layer object from the **Graphics** class.

```
class OSD {
    /**************
     * Properties *
     **************/
    private:
        int width;
        int height;
        long screen_pixel_count;
    public:
        Graphics graphic_layer_1;

    /***********
     * Methods *
     ***********/
    public:
        OSD(int osd_width, int osd_height); // Constructor
        ~OSD(void); // Destructor

};
```

By creating the **osd** object, we are calling the class constructor that is allocating a buffer of dynamic memory and initializing it to be at our desired resolution and get located in an address space accessible by our **Frame_Reader** hardware component.

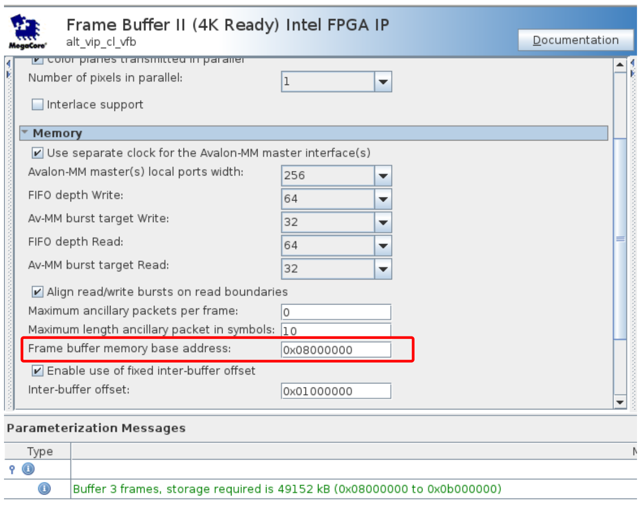
```
/*
 * Constructor
 */
OSD::OSD(int osd_width, int osd_height){

    width = osd_width;
    height = osd_height;

    if(graphic_layer_1.Init(width, height, VIDEO_DISPLAY_COLOR_DEPTH, 0x08000000, 1)){
        printf("Memory allocation error (graphic_layer_1)!");
        while(1);
    }

    screen_pixel_count = width * height;

    // Clear the frame buffer to initial content
    // We set black color and full transparency
    graphic_layer_1.Draw_Box(0,0,width,height,0x000000,1,0xFF);
}
```

VIDEO_DISPLAY_COLOR_DEPTH is declared as 32 (8 bits per ARGB color planes)

0x08000000 is the Frame Reader Memory Base Address we set at hardware configuration time.

Note that we set the same value in the software for consistency:

```
osd_vfr.set_frame_address(0x8000000);
```



Frame Buffer II (4K Ready) Intel FPGA IP
alt_vip_cl_vfb

[✓] Color planes transmitted in parallel
Number of pixels in parallel: 1
[ ] Interlace support

Memory
[✓] Use separate clock for the Avalon-MM master interface(s)
Avalon-MM master(s) local ports width: 256
FIFO depth Write: 64
Av-MM burst target Write: 32
FIFO depth Read: 64
Av-MM burst target Read: 32
[✓] Align read/write bursts on read boundaries
Maximum ancillary packets per frame: 0
Maximum length ancillary packet in symbols: 10
Frame buffer memory base address: 0x08000000
[✓] Enable use of fixed inter-buffer offset
Inter-buffer offset: 0x01000000

Parameterization Messages

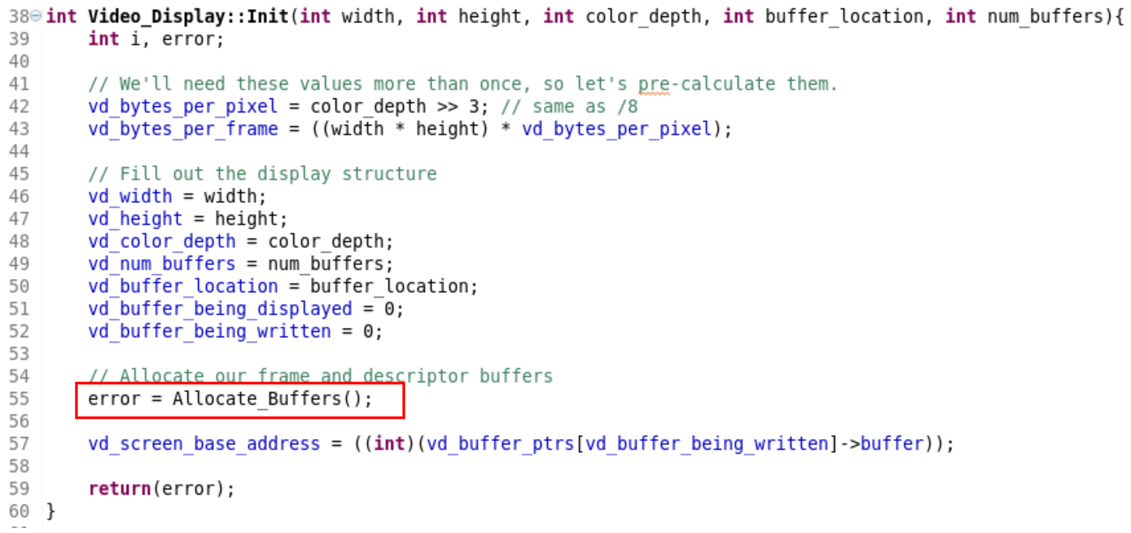
Type	
ⓘ	Buffer 3 frames, storage required is 49152 kB (0x08000000 to 0x0b000000)

Recall the **Frame_Reader** configuration, with a base address of 0x08000000. But we can change this value at run-time by setting a different value using the `set_frame_address` method showed above. If you do this, you might need to modify the argument passed to the method Init in the **OSD** constructor to match.

NOTE: You can also modify the class to accept an additional parameter to the function, being this the address where you want to locate your buffer.

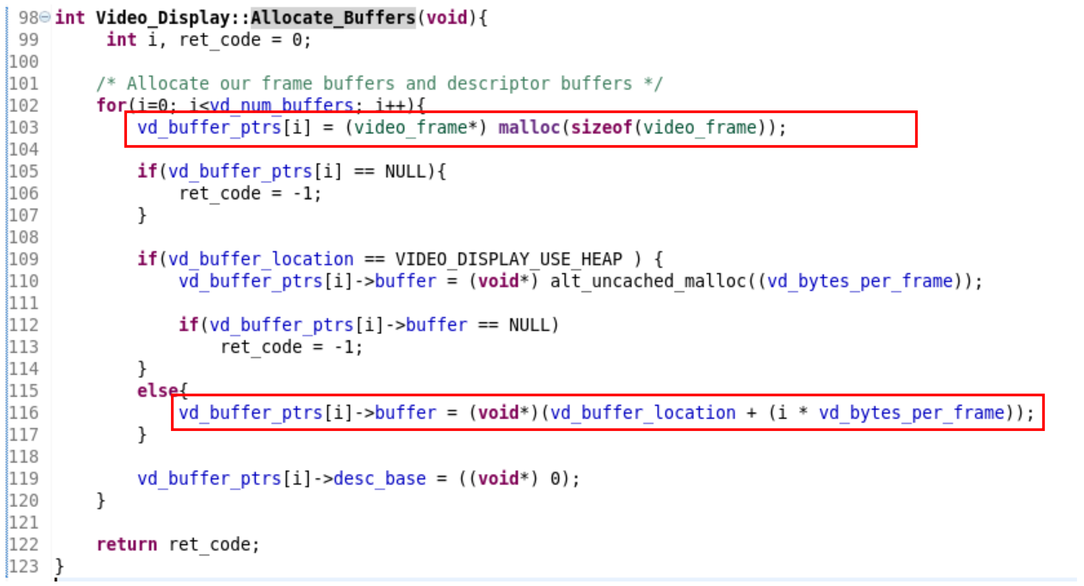
Ultimately, this address is stored in the vd_buffer_location property of the **Video_Display** class and remapped into the `Allocate_Buffers` method.

See below the Init function that sets all the graphic_layer properties, as well as calls `Allocate_Buffers` method.



```
int Video_Display::Init(int width, int height, int color_depth, int buffer_location, int num_buffers){
    int i, error;

    // We'll need these values more than once, so let's pre-calculate them.
    vd_bytes_per_pixel = color_depth >> 3; // same as /8
    vd_bytes_per_frame = ((width * height) * vd_bytes_per_pixel);

    // Fill out the display structure
    vd_width = width;
    vd_height = height;
    vd_color_depth = color_depth;
    vd_num_buffers = num_buffers;
    vd_buffer_location = buffer_location;
    vd_buffer_being_displayed = 0;
    vd_buffer_being_written = 0;

    // Allocate our frame and descriptor buffers
    error = Allocate_Buffers();

    vd_screen_base_address = ((int)(vd_buffer_ptrs[vd_buffer_being_written]->buffer));

    return(error);
}
```

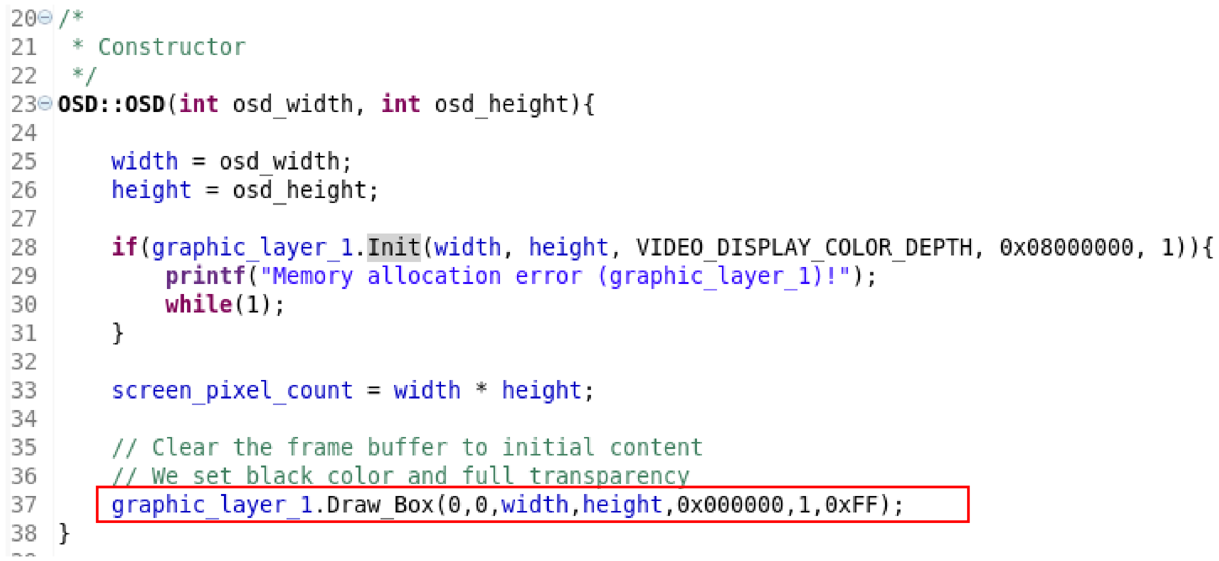
`Allocate_Buffers` creates dynamic memory portions and remaps the address to a specific location to be accessible by our hardware.



```
int Video_Display::Allocate_Buffers(void){
     int i, ret_code = 0;

    /* Allocate our frame buffers and descriptor buffers */
    for(i=0; i<vd_num_buffers; i++){
        vd_buffer_ptrs[i] = (video_frame*) malloc(sizeof(video_frame));

        if(vd_buffer_ptrs[i] == NULL){
            ret_code = -1;
        }

        if(vd_buffer_location == VIDEO_DISPLAY_USE_HEAP ) {
            vd_buffer_ptrs[i]->buffer = (void*) alt_uncached_malloc((vd_bytes_per_frame));

            if(vd_buffer_ptrs[i]->buffer == NULL)
                ret_code = -1;
        }
        else{
            vd_buffer_ptrs[i]->buffer = (void*)(vd_buffer_location + (i * vd_bytes_per_frame));
        }

        vd_buffer_ptrs[i]->desc_base = ((void*) 0);
    }

    return ret_code;
}
```

Finally, the **OSD** constructor uses one of the graphic functions to draw a full screen rectangular box to initialize all the random memory content to color BLACK (0x000000) and with full transparency (0xFF)

We will see details on the graphic library in the next section.

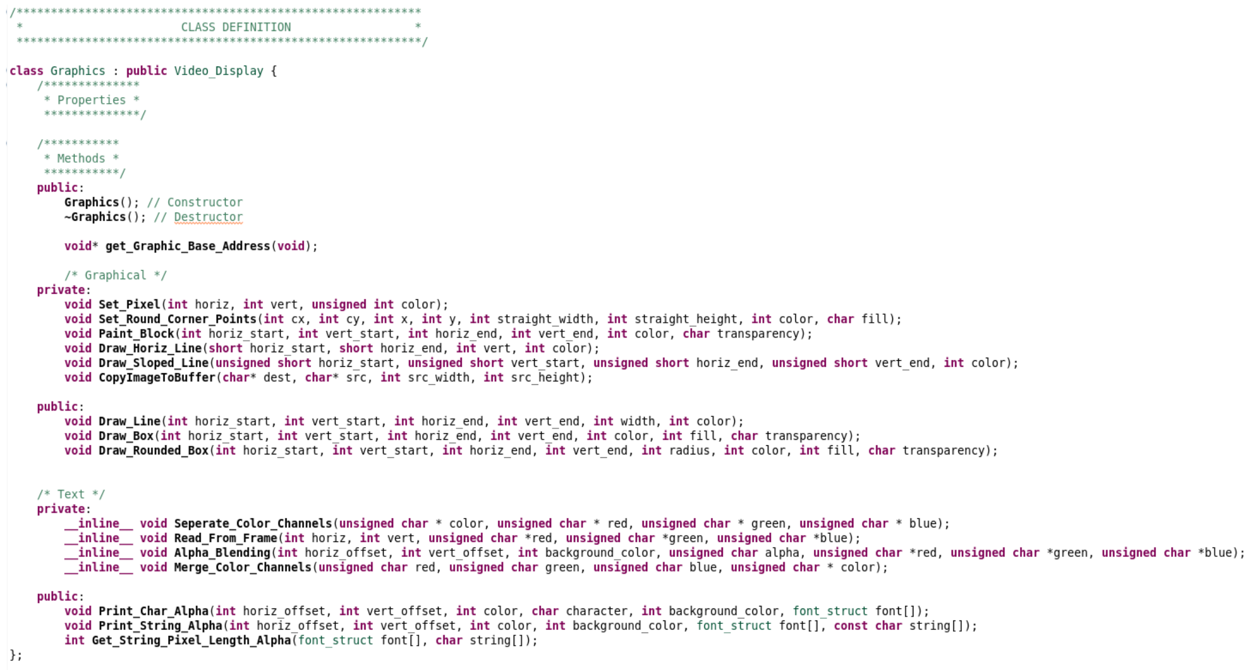


```
/*
 * Constructor
 */
OSD::OSD(int osd_width, int osd_height){

    width = osd_width;
    height = osd_height;

    if(graphic_layer_1.Init(width, height, VIDEO_DISPLAY_COLOR_DEPTH, 0x08000000, 1)){
        printf("Memory allocation error (graphic_layer_1)!");
        while(1);
    }

    screen_pixel_count = width * height;

    // Clear the frame buffer to initial content
    // We set black color and full transparency
    graphic_layer_1.Draw_Box(0,0,width,height,0x000000,1,0xFF);
}
```

3.7.Using the graphic library

Once we have set all the **OSD** class configuration and allocated the memory buffer which constitutes our whiteboard, it’s now time to draw on it. For that, we will use the methods available in the **Graphics** class defined in the `Graphics_lib.hpp` and `Graphics_lib.cpp` files.

The **Graphics** class provides with additional methods to **Video_Display** class, these methods are fundamentally used to print text and draw simple shapes on the allocated buffer memory, and ultimately on screen.



```
/***********************************************************
 *                       CLASS DEFINITION                  *
 ***********************************************************/

class Graphics : public Video_Display {
    /**************
     * Properties *
     **************/

    /***********
     * Methods *
     ***********/
    public:
        Graphics(); // Constructor
        ~Graphics(); // Destructor

        void* get_Graphic_Base_Address(void);

        /* Graphical */
    private:
        void Set_Pixel(int horiz, int vert, unsigned int color);
        void Set_Round_Corner_Points(int cx, int cy, int x, int y, int straight_width, int straight_height, int color, char fill);
        void Paint_Block(int horiz_start, int vert_start, int horiz_end, int vert_end, int color, char transparency);
        void Draw_Horiz_Line(short horiz_start, short horiz_end, int vert, int color);
        void Draw_Sloped_Line(unsigned short horiz_start, unsigned short vert_start, unsigned short horiz_end, unsigned short vert_end, int color);
        void CopyImageToBuffer(char* dest, char* src, int src_width, int src_height);

    public:
        void Draw_Line(int horiz_start, int vert_start, int horiz_end, int vert_end, int width, int color);
        void Draw_Box(int horiz_start, int vert_start, int horiz_end, int vert_end, int color, int fill, char transparency);
        void Draw_Rounded_Box(int horiz_start, int vert_start, int horiz_end, int vert_end, int radius, int color, int fill, char transparency);


    /* Text */
    private:
        __inline__ void Seperate_Color_Channels(unsigned char * color, unsigned char * red, unsigned char * green, unsigned char * blue);
        __inline__ void Read_From_Frame(int horiz, int vert, unsigned char *red, unsigned char *green, unsigned char *blue);
        __inline__ void Alpha_Blending(int horiz_offset, int vert_offset, int background_color, unsigned char alpha, unsigned char *red, unsigned char *green, unsigned char *blue);
        __inline__ void Merge_Color_Channels(unsigned char red, unsigned char green, unsigned char blue, unsigned char * color);

    public:
        void Print_Char_Alpha(int horiz_offset, int vert_offset, int color, char character, int background_color, font_struct font[]);
        void Print_String_Alpha(int horiz_offset, int vert_offset, int color, int background_color, font_struct font[], const char string[]);
        int Get_String_Pixel_Length_Alpha(font_struct font[], char string[]);
};
```

There are public functions to draw lines, boxes and rounded boxes and other ones to print text strings and characters. These functions rely on other private functions to perform all the pixel manipulations needed to build the graphic elements.

We will not go into full details on how those functions have been implemented (you can have a complete view on the file `Graphics_lib.cpp` provided in clear source text form).

Let's anyway go through a couple of examples on what information is required to use them:

void Draw_Rounded_Box(**int** horiz_start, **int** vert_start, **int** horiz_end, **int** vert_end, **int** radius, **int** color, **int** fill, **char** transparency);

This function is used to draw a rectangular box with rounded corners. It accepts the following parameters:

- **int** horiz_start: horizontal offset from the origin
- **int** vert_start: vertical offset from the origin
- **int** horiz_end: where to finish. Width can then be calculated as (horiz_end – horiz_start)
- **int** vert_end: where to finish. Height can then be calculated as (vert_end – vert_start)
- **int** radius: to apply on the rounded corners
- **int** color: color in RGB format (24bit) of the box
- **int** fill: '1' means filled with same color, '0' meand only the outline is drawn
- **char** transparency: 0 means fully opaque, 255 means fully transparent. There are 255 possible transparency levels applicable.

void Print_String_Alpha(**int** horiz_offset, **int** vert_offset, **int** color, **int** background_color, font_struct font[], **const char** string[]);

This function is used to print text strings on screen. It accepts the following parameters:

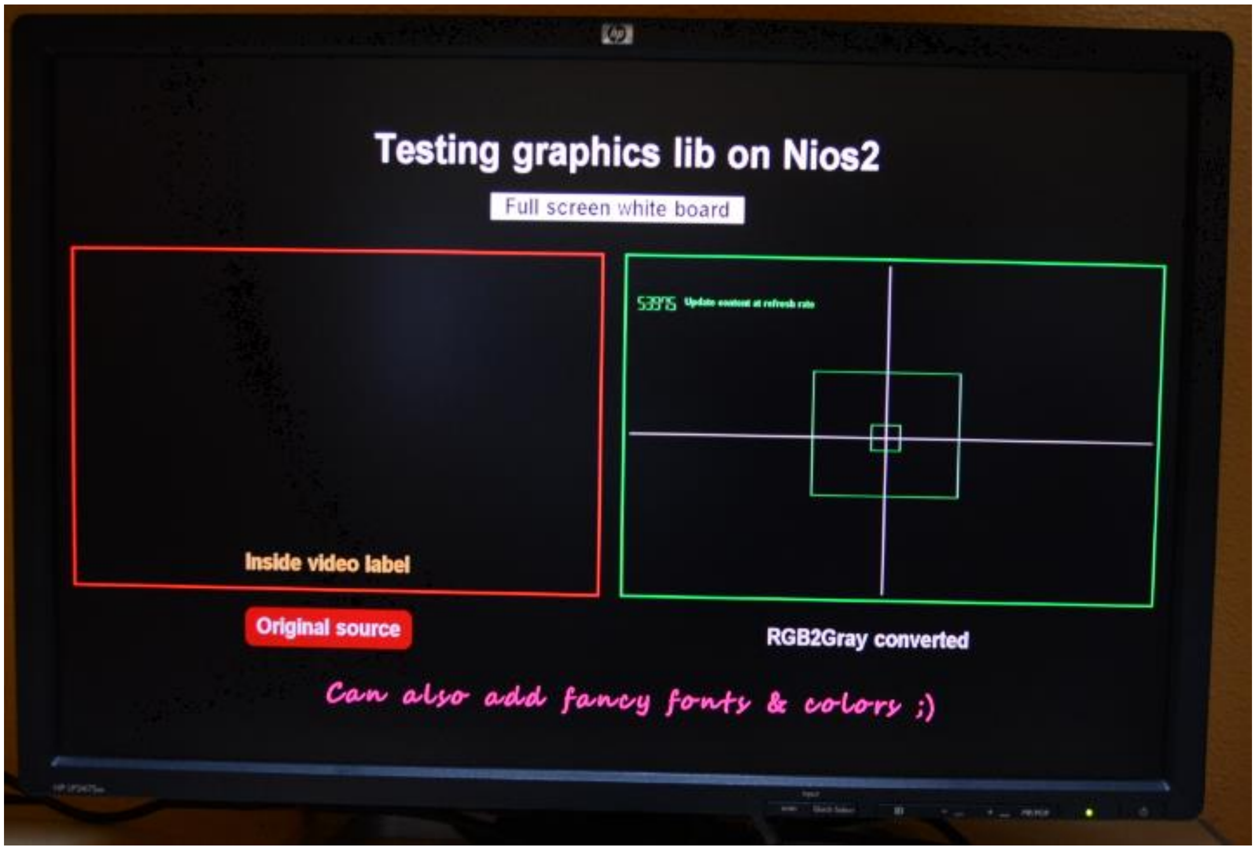
- **int** horiz_offset: horizontal offset from the origin
- **int** vert_offset: vertical offset from the origin
- **int** color: foreground/font color
- **int** background_color: background color
- font_struct font[]: selected font (we are providing some sample fonts in the software application)
- **const char** string[]: text information to print as a string of characters.

By using the functions provided in the **Graphics** class we can enrich the information we show on screen. In this application we have created an example use case as shown in the picture below by using the following instructions:

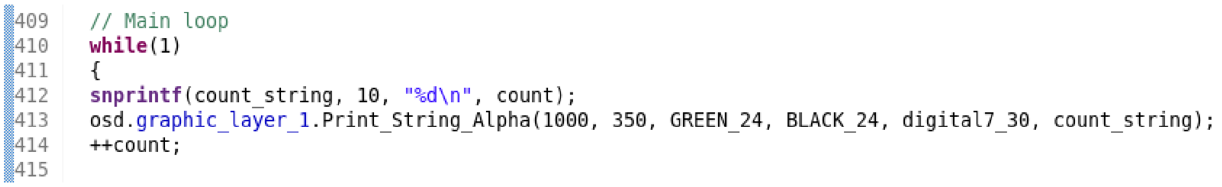
```
osd.graphic_layer_1.Print_String_Alpha(550, 100, WHITE_24, BLACK_24, arialbold_60, "Testing graphics lib on Nios2");
osd.graphic_layer_1.Print_String_Alpha(750, 200, BLACK_24, WHITE_24, arialbold_34, "  Full screen white board  ");

osd.graphic_layer_1.Draw_Box(30,290,940,810,RED_24,0,0);
osd.graphic_layer_1.Draw_Box(29,289,941,811,RED_24,0,0);
osd.graphic_layer_1.Draw_Box(28,288,942,812,RED_24,0,0);

osd.graphic_layer_1.Draw_Box(980,290,1890,810,GREEN_24,0,0);
osd.graphic_layer_1.Draw_Box(979,289,1891,811,GREEN_24,0,0);
osd.graphic_layer_1.Draw_Box(978,288,1892,812,GREEN_24,0,0);


osd.graphic_layer_1.Draw_Rounded_Box(330,840,620,900,15,DARKRED_24,1,0);
osd.graphic_layer_1.Print_String_Alpha(350, 850, SILVER_24, DARKRED_24, arialbold_34, "Original source");

osd.graphic_layer_1.Print_String_Alpha(1230, 850, SILVER_24, BLACK_24, arialbold_34, "RGB2Gray converted");

osd.graphic_layer_1.Print_String_Alpha(320, 750, GOLDENROD_24, BLACK_24, arialbold_34, " Inside video label ");
osd.graphic_layer_1.Print_String_Alpha(1080, 350, GREEN_24, BLACK_24, arialbold_15, "Update content at refresh rate");

osd.graphic_layer_1.Print_String_Alpha(470, 940, DEEPPINK_24, BLACK_24, segoescriptbold_42,"Can also add fancy fonts & colors ;)");

osd.graphic_layer_1.Draw_Box(1400,540,1450,580,GREEN_24,0,0);
osd.graphic_layer_1.Draw_Box(1300,460,1550,650,GREEN_24,0,0);
osd.graphic_layer_1.Draw_Line(1425,300,1425,800,2,WHITE_24);
osd.graphic_layer_1.Draw_Line(990,560,1880,560,2,WHITE_24);

mixer[3].enable_layer();
```

Please note that we are providing definition for a large collection of colors in `Graphics_lib.hpp` file

Don't forget to enable the mixer's layers to see the results on screen.


Testing graphics lib on Nios2
Full screen white board
53975 Update content at refresh rate
Inside video label
Original source
RGB2Gray converted
Can also add fancy fonts & colors ;)

Even this is not a fully optimized graphic library, still allows to update on screen information significantly fast (for a small area), as we are sampling here by placing a counter on top of our **rgb2gray** converted video.



```
// Main loop
while(1)
{
snprintf(count_string, 10, "%d\n", count);
osd.graphic_layer_1.Print_String_Alpha(1000, 350, GREEN_24, BLACK_24, digital7_30, count_string);
++count;

```

This code doesn't represent a frame counter and it isn't bind to any timer for accurate representation, it's just a counter updated every time the main loop executes.

We use **snprintf** function to convert an integer value to string, which is compatible with the parameter expected in the `Print_String_Alpha` function.

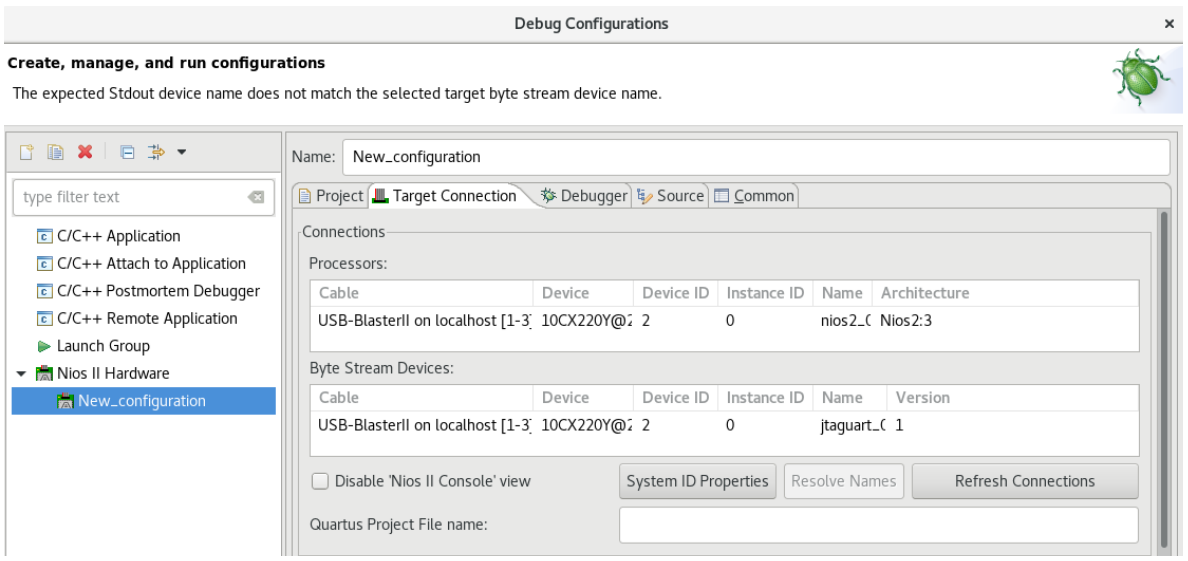
NOTE: In `<project_dir>/software/source/osd/fonts` we have provided some sample fonts to be used with the implemented graphic library. This is just a sample of different sizes and even fancy fonts like **digital7** and **segoescriptbold**. We have used gimp (https://www.gimp.org/) to extract and generate the fonts with a specific script. We are planning a new session with advanced topics on graphic library, including how to create your own fonts.

3.8. 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 *`osd_overlay.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


Debug Configurations

Create, manage, and run configurations

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

Name: New_configuration

Connections

Processors:

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

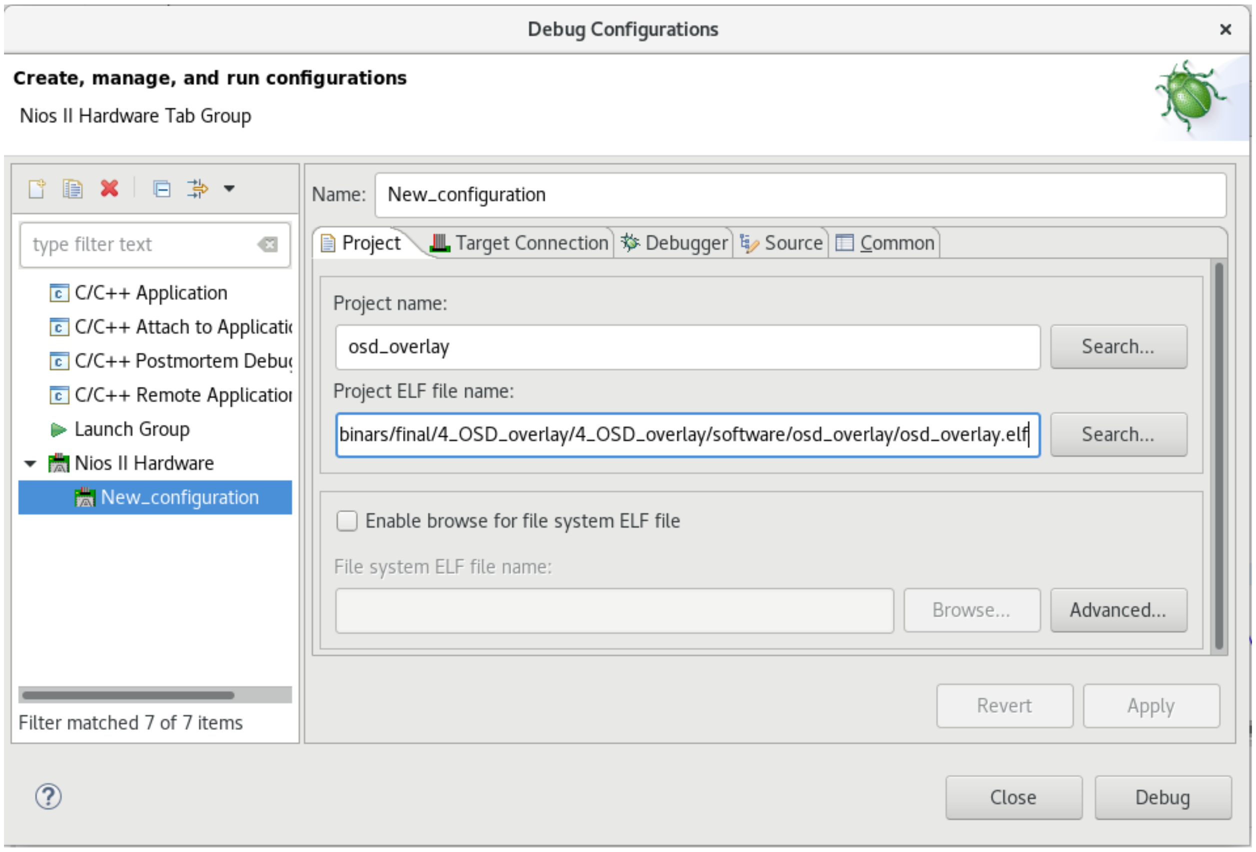
Byte Stream Devices:

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

Disable 'Nios II Console' view

Quartus Project File name:

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


Debug Configurations

Create, manage, and run configurations

Nios II Hardware Tab Group

Name: New_configuration

Project name: osd_overlay

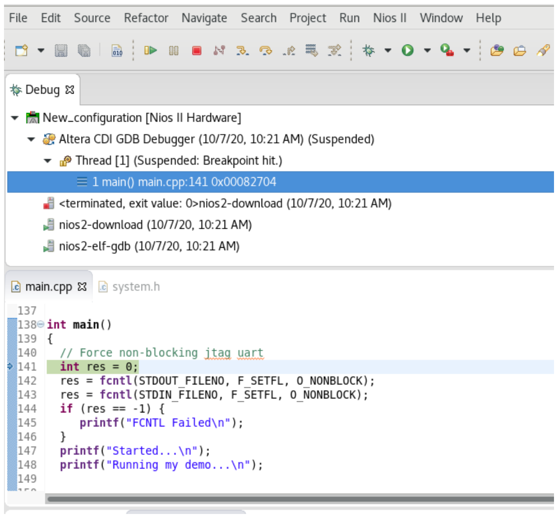
Project ELF file name: binars/final/4_OSD_overlay/4_OSD_overlay/software/osd_overlay/osd_overlay.elf

Enable browse for file system ELF file

File system ELF file name:

Filter matched 7 of 7 items

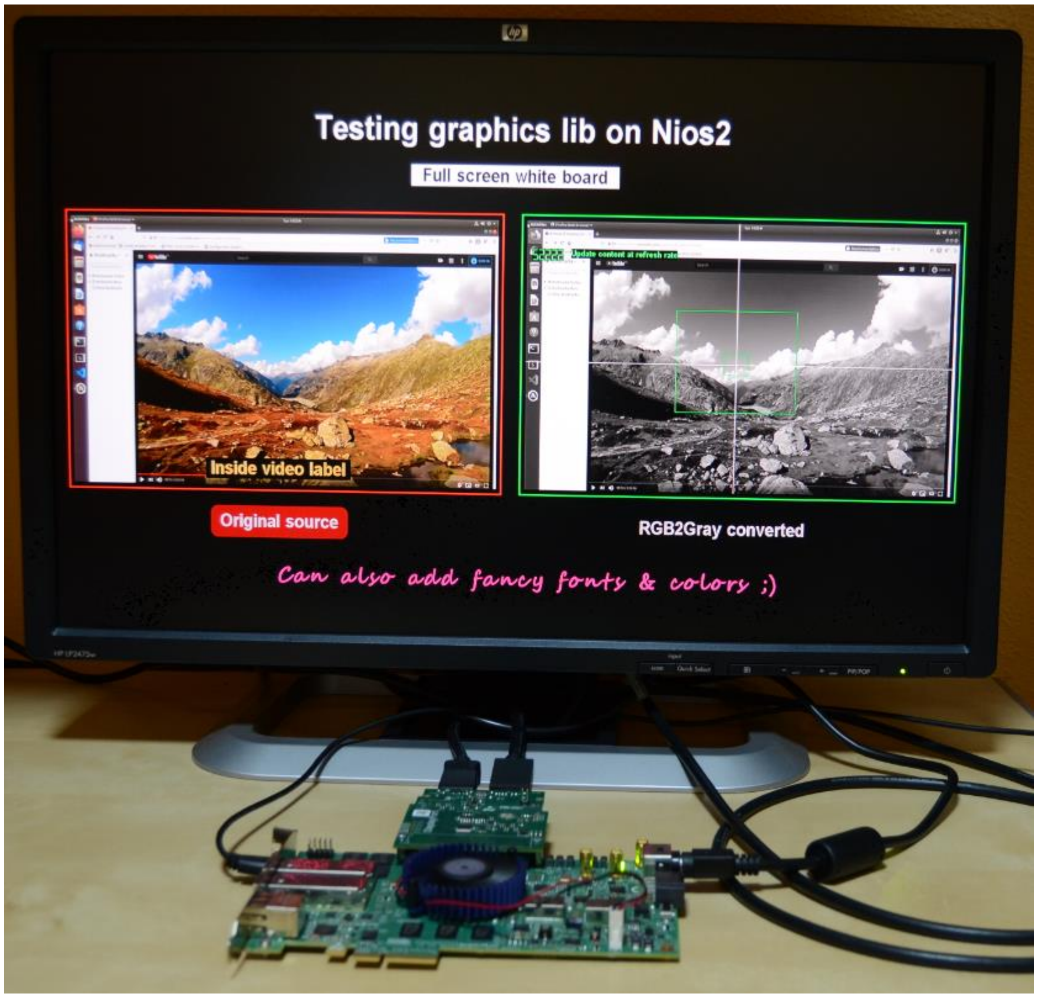
The executable file is then downloaded into the NiosII program memory and the execution is stopped right after main function is called. Just press the **Resume** button in the debugger to launch the complete execution.


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

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

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

```

If we have our video source from the external PC connected to the DisplayPort input, we will see the captured image scaled in one PiP window, side-by-side in another PiP window we have the converted to grayscale by our custom rgb2gray module and on the top layer, all the graphics we have generated with the Nios2 CPU and the graphic library.


Testing graphics lib on Nios2
Full screen white board
Update content at refresh rate
Inside video label
Original source
RGB2Gray converted
Can also add fancy fonts & colors ;)

4. Summary

In this lab, we have exercised with a lightweight graphic library, running on the Nios2 processor, to draw simple shapes and text strings to overlay on top of our processed live video.

- We have followed the steps to create a video pipeline able to duplicate a video stream and apply different processing before getting them mixed in PiP in the global layout
- We have configured a Frame_Reader and the Mixer to allow mixing graphic content from a memory buffer with the live video
- We have learnt how to configure and use the graphic library to enrich the information we show on screen.