Intel FPGA VIP webinar series

Session 4
Adding OSD overlay

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

Video processing on FPGAs made easy

Objectives

- What resources are available to develop Video Processing solutions?
- From the basics: step through an incremental series of example designs
- End2End flow demonstration: hardware architecture design, software development & debug
- Sessions will be recorded. Exercise manuals and project files will be available for on-demand consumption

Content available in the GitHub repo:

https://github.com/perezfra/VIP_webinars_Intel_FPGA

Webinar Series

Soft start -> Increasing Complexity

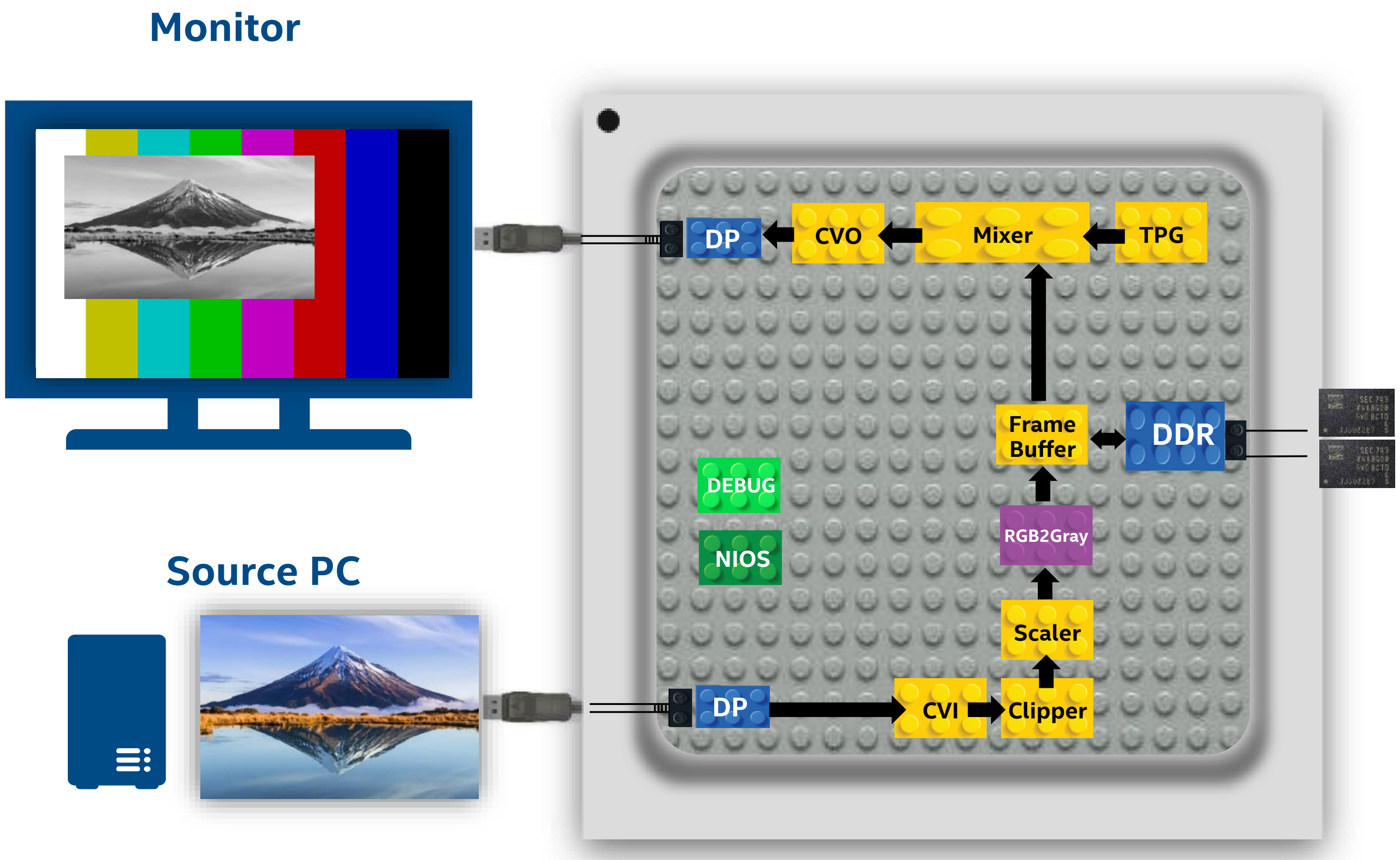
1. Building a video processing pipeline (Oct, 14th and 21st)
    - 1.1- DisplayPort loopback example design implementation
    - 1.2- Building our first video pipeline using VIP suite cores
    - 1.3- Complete End2End VIP pipeline
2. Strategies to debug a VIP pipeline (Nov, 4th)
    - Overview of system level considerations and key video concepts
    - Overview of Avalon-ST Video protocol
    - Bring up your pipeline using System Console
3. Integrate a simple custom component (Nov, 19th)
    - How to add your "secret sauce" to the application
    - Step flow on how to develop a custom component compliant with VIP
4. Adding On Screen Display overlay (Dec, 16th)
    - Use a lightweight graphic library with Nios
    - Add text and graphic content overlay on top of your live video

https://github.com/perezfra/VIP_webinars_Intel_FPGA

Building a Video Processing Pipeline

Previous Session 3 – Nov, 19th

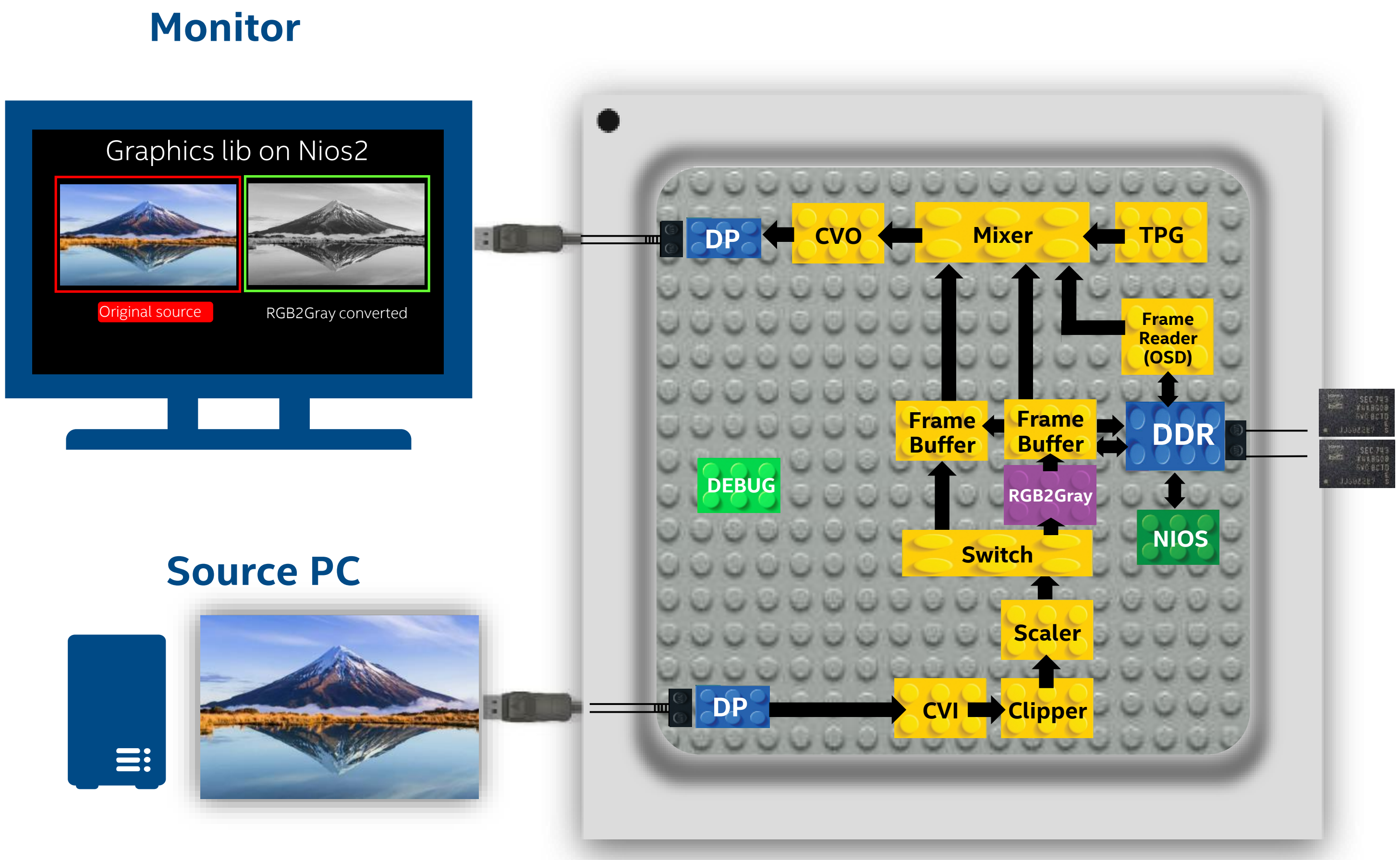
1. Build a video pipeline using VIP cores (cropping, scaling and mixing with a background) and debug
2. Develop and integrate a custom module to perform RGB to Grayscale conversion



Adding OSD overlay

Session 4 – Dec, 16th

1. Build a video pipeline using VIP cores (cropping, scaling and mixing with a background) and debug
2. Develop and integrate a custom module to perform RGB to Grayscale conversion
3. Duplicate video stream with switch and create PiP
4. Add a Frame reader to retrieve graphic frame from DDR3
5. Add new layers to mixer
6. Use a graphic lib on Nios2 to generate shapes and text



Adding OSD overlay

Motivations

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

Adding OSD overlay

Flow

- Insert a Frame Reader component to continuously read data from a specific memory space in the DDR3 bank.
- A Nios2 processor writes, on that same memory space, text and graphical content using a lightweight library
- The Frame Reader is configured to retrieve pixel information from the specific memory address and generates AvalonST-Video packets that are routed to a layer in the mixer.
- This layer has been configured to support alpha blending, so we can play with different transparency levels on the graphic content.

Adding OSD overlay

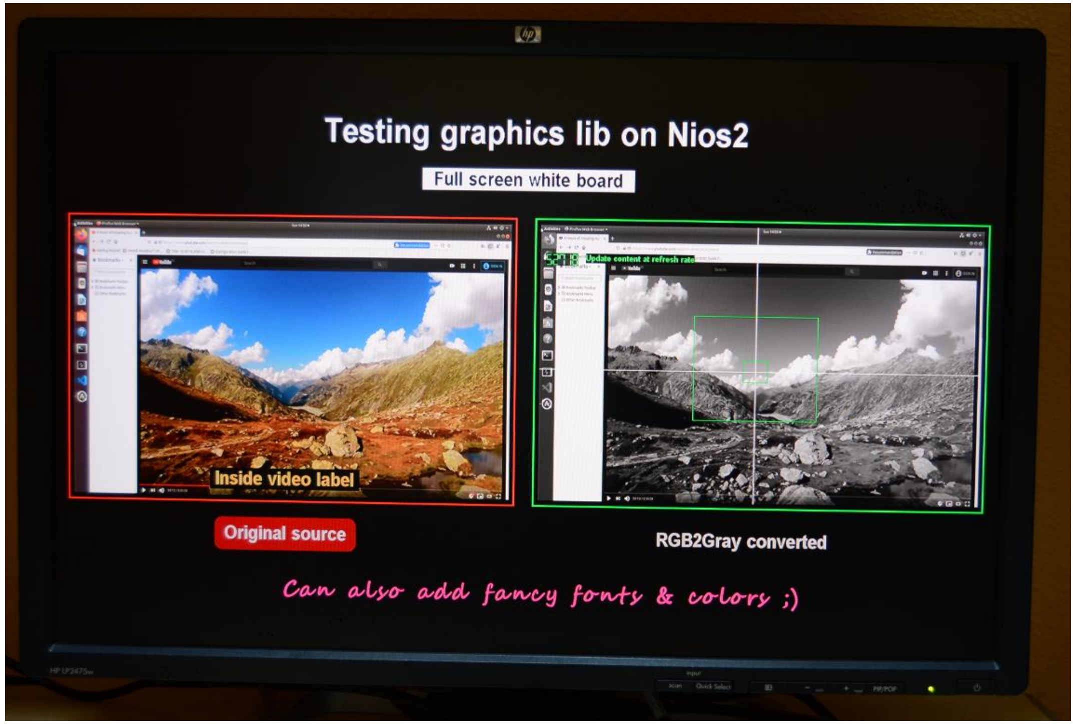
FPGA partitioning

- With FPGAs you can 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.
- We have implemented a frame reader component able to read back from memory at the required refresh of 1080p60 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 updating the graphic content at its own pace, rendering information in the memory space allocated as graphic frame.

Adding OSD overlay

Final result

- In this session, we modify the VIP pipeline to include and configure the different hardware modules, as well as, add and use the graphic library in the software flow.
- We have 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 have created some basic text strings and rectangular boxes added on top of video.



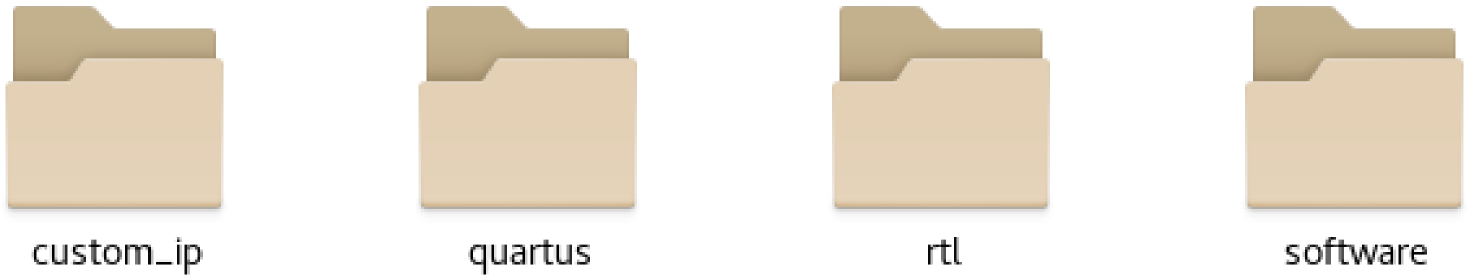
Session 4 – Adding OSD overlay

HW implementation

Adding OSD overlay – HW flow

Setting up the Quartus project

- Extract the files in the **4_OSD_Overlay_v1.tar.gz** package and open the Quartus project located at <extracted_folder>/quartus/c10_dp_demo.qpf

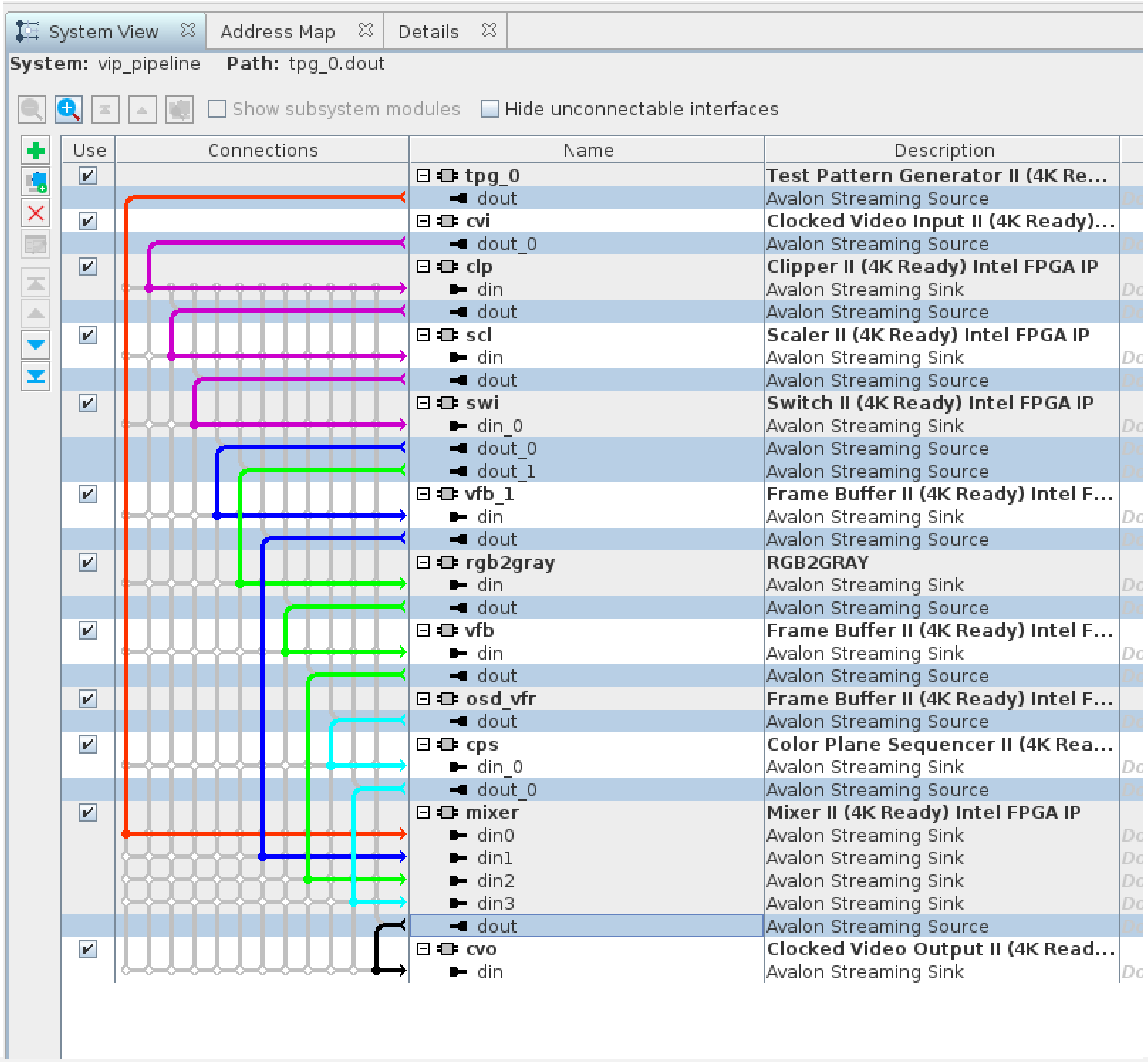


- Download the file from the github repo
  - https://github.com/perezfra/VIP_webinars_Intel_FPGA

Adding OSD overlay – HW flow

Understanding the pipeline 1/3

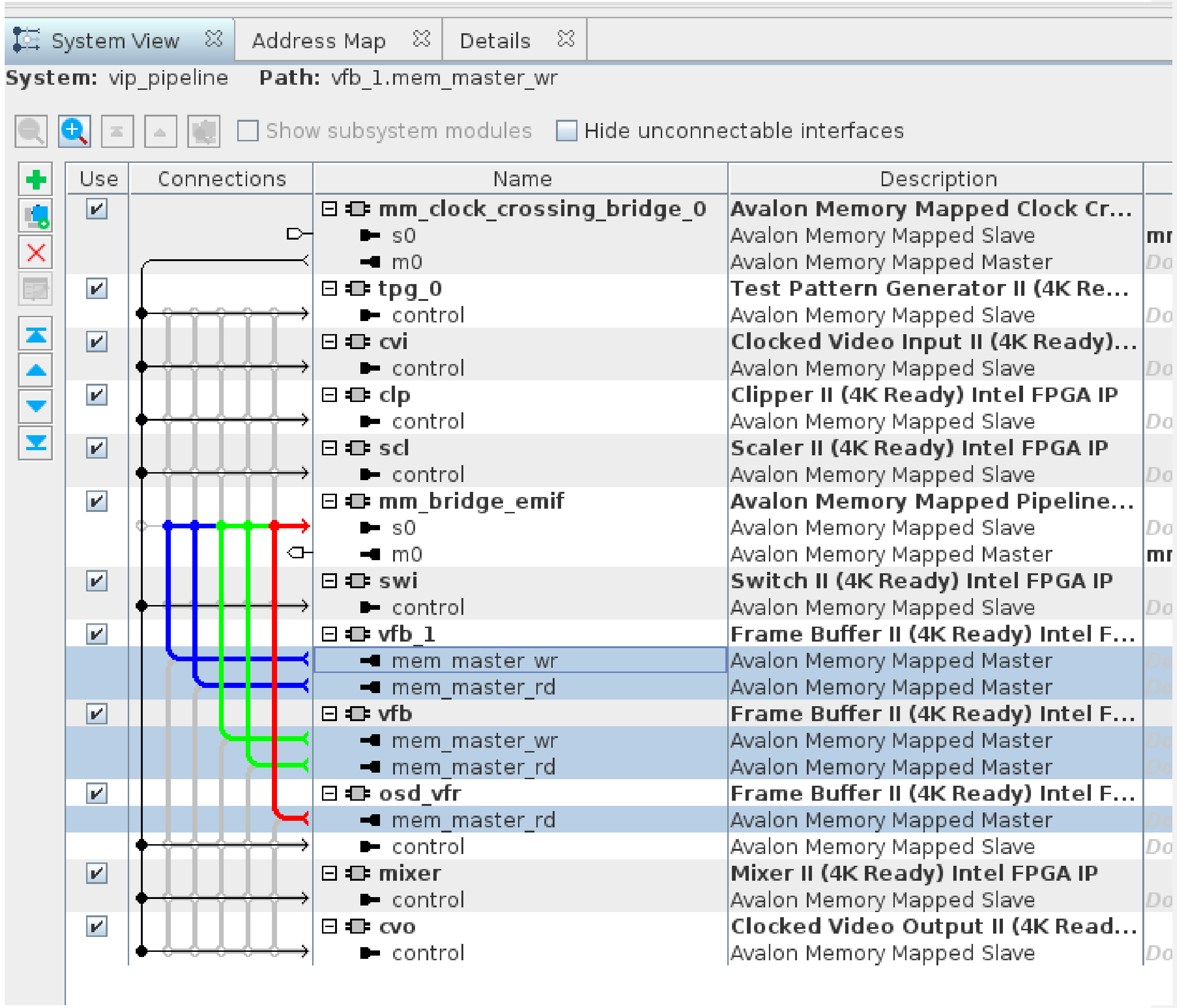
- Launch Platform Designer and select <extracted_folder>/rtl/core/vip_pipeline.qsys to open
- Filter by **Avalon-ST Interfaces** to get a cleaner view on the video data path.



Adding OSD overlay – HW flow

Understanding the pipeline 2/3

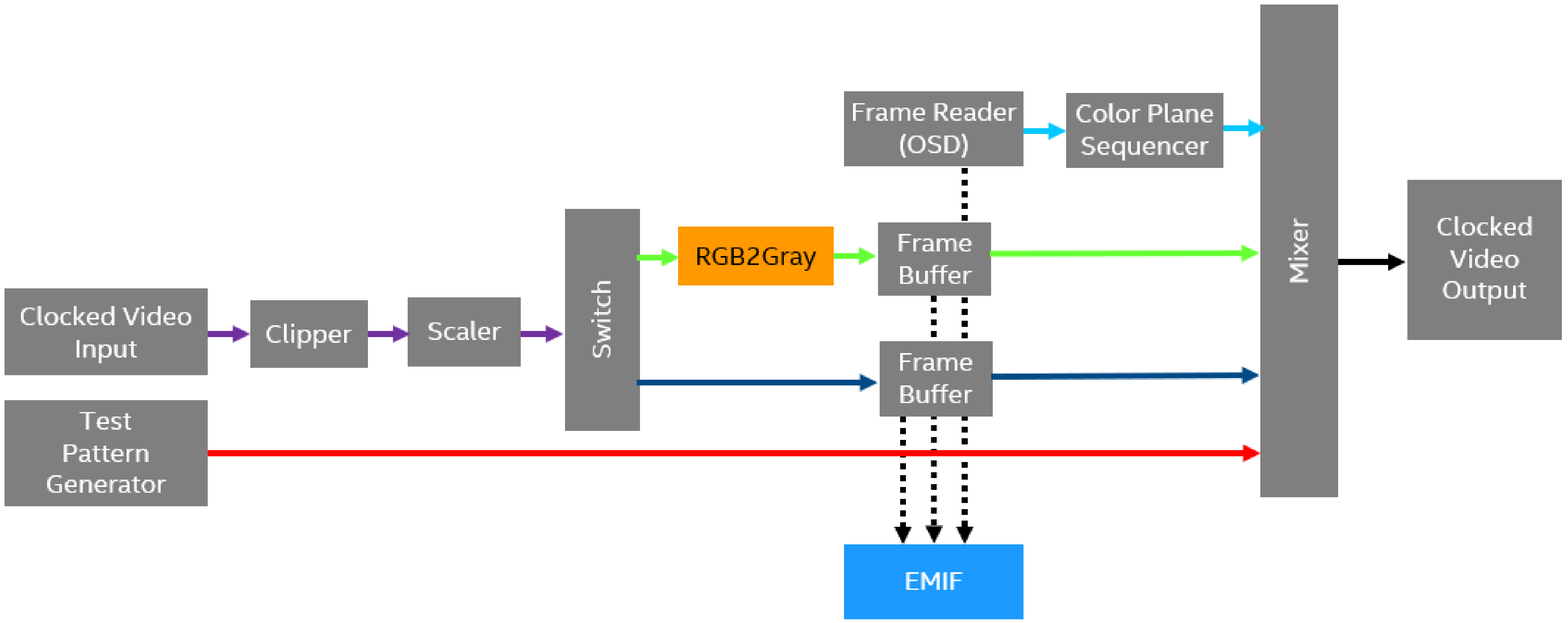
- Filter by **Avalon-MM Interfaces**
- Different Frame Buffer components share the same DDR3 bank
- Platform Designer automatically generates arbitration logic



Adding OSD overlay – HW flow

Understanding the pipeline 3/3

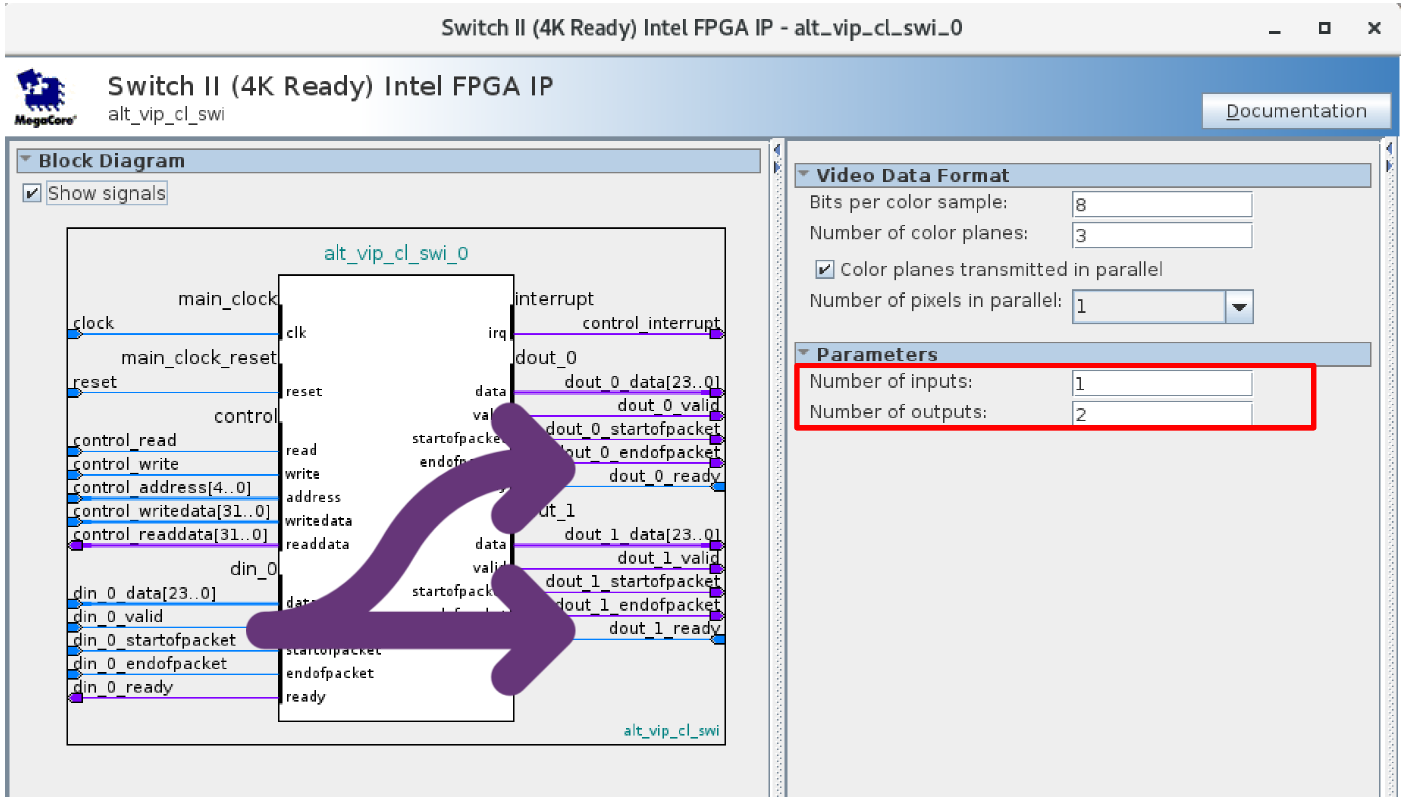
- Final diagram representation of the `vip_pipeline.qsys` subsystem



Adding OSD overlay – HW flow

Switch

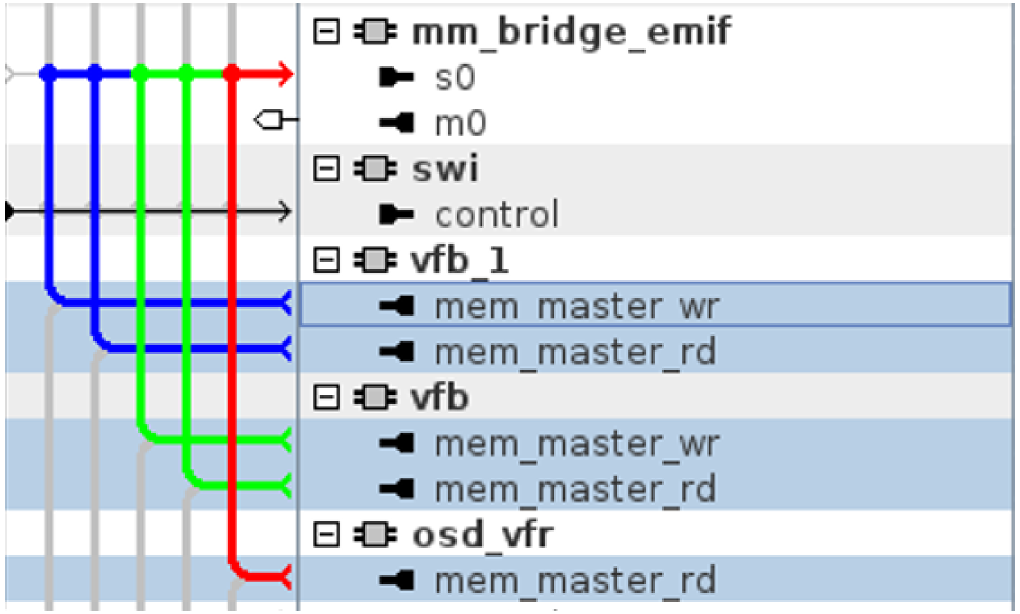
- Duplicate a video stream



Adding OSD overlay – HW flow

Memory Map – 1/7

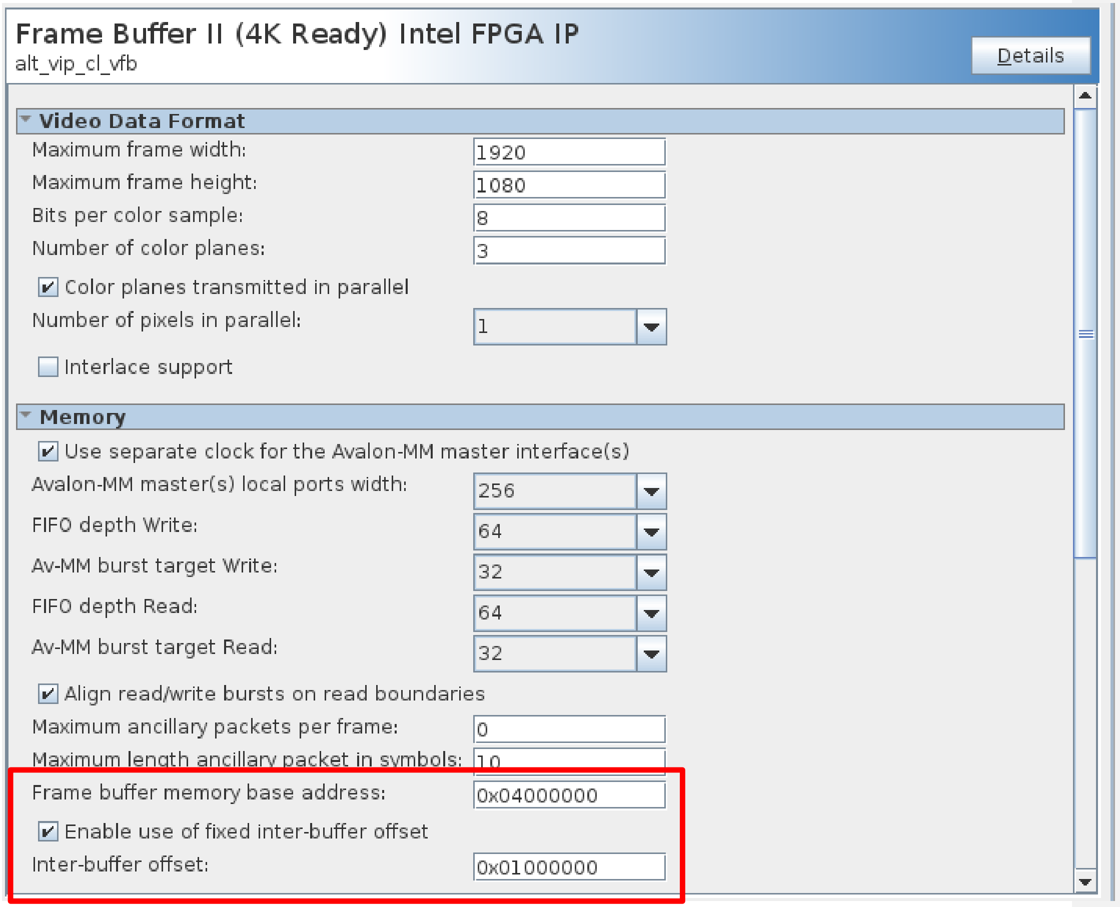
- In the Cyclone10 GX devkit we have 1GB or DDR3: 0x0 <> 0x3fff_ffff
- In **`vip_pipeline.qsys`** we have 3 components connected to the pipeline bridge, that will be finally connected to the DDR3 emif in the **`dp_core.qsys`** subsystem
- Let's see how to configure each Frame Buffer



Adding OSD overlay – HW flow

Memory Map – 2/7

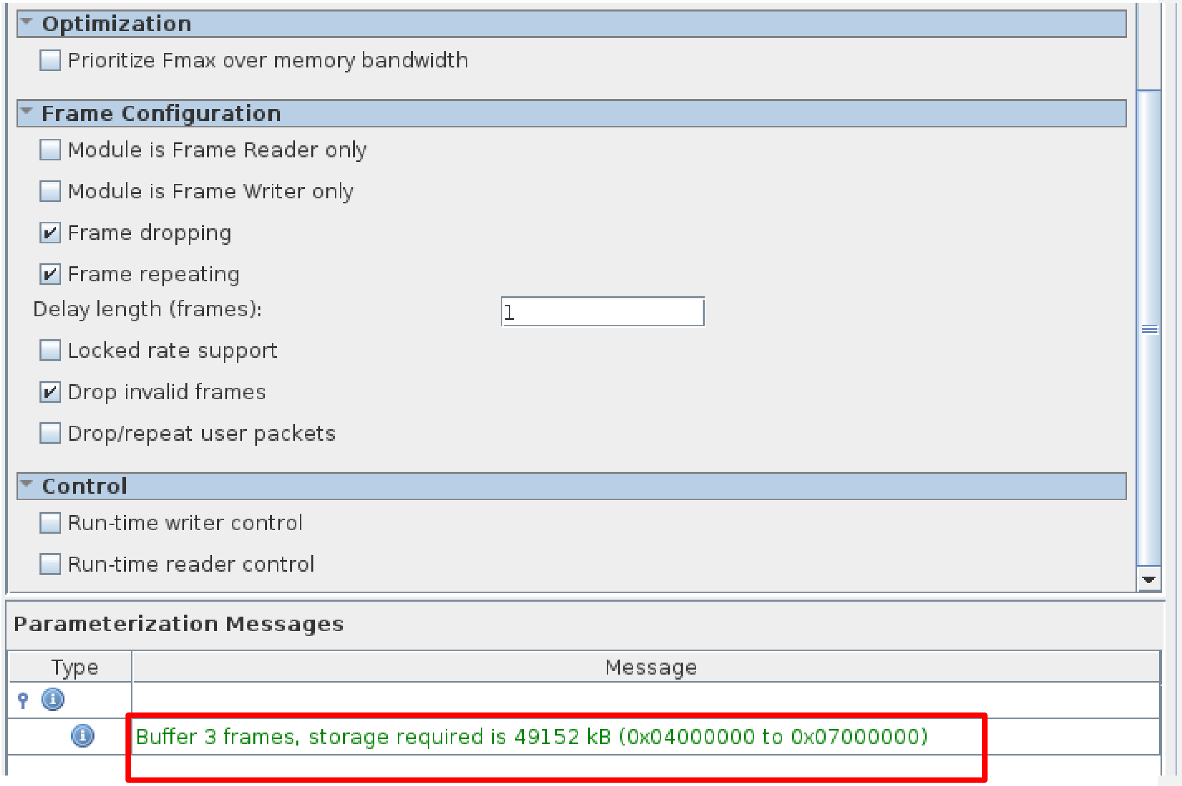
- If you need a refresh consider go back to the session **1.3_Building_Complete_Pipelines** where we explain all the details about Frame Buffer configuration
- We need to set the Frame buffer memory base address to be exclusive
- Enabling fixed inter-buffer offset has advantages in software development
- This is **vfb_1** used for original video source: 0x04000000



Adding OSD overlay – HW flow

Memory Map – 3/7

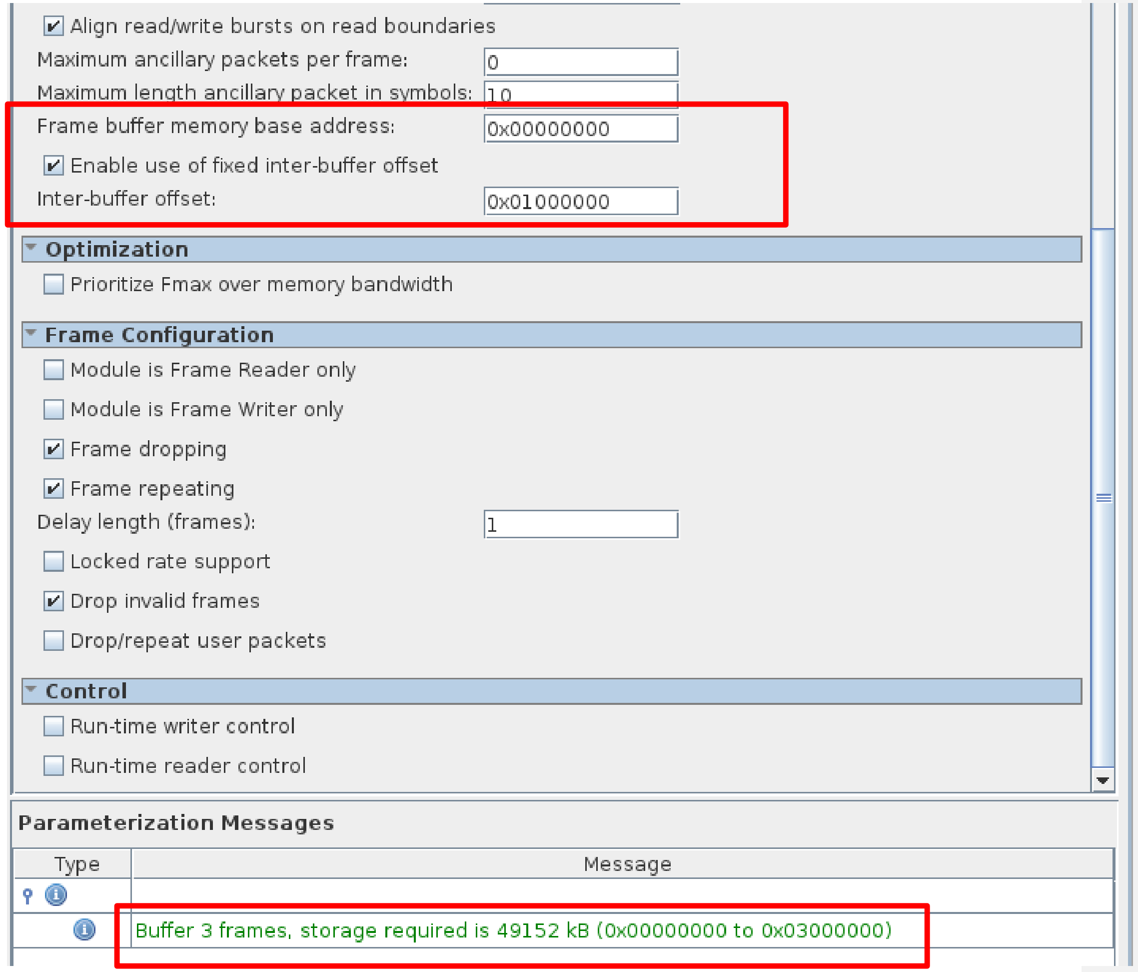
- Enable Frame Dropping and Repeating
- We have now 3 video frames allocated to fixed addresses
  - 0x04000000
  - 0x05000000
  - 0x06000000
- Let’s do the same with the other video Frame Buffer



Adding OSD overlay – HW flow

Memory Map – 4/7

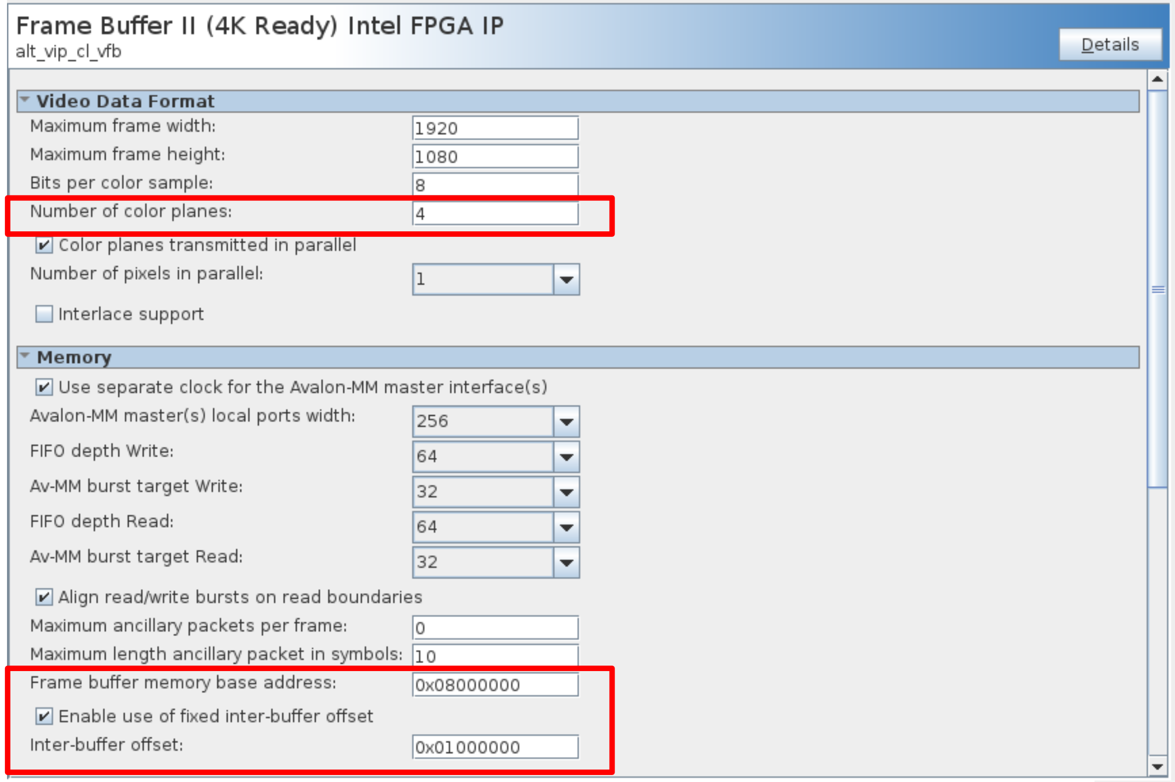
- We use **vfb** component for our **rgb2gray** stream flow
- We set base address to 0x0
- Also enable fixed offset
- We have now 3 video frames allocated to fixed addresses
  - 0x00000000
  - 0x01000000
  - 0x02000000



Adding OSD overlay – HW flow

Memory Map – 5/7

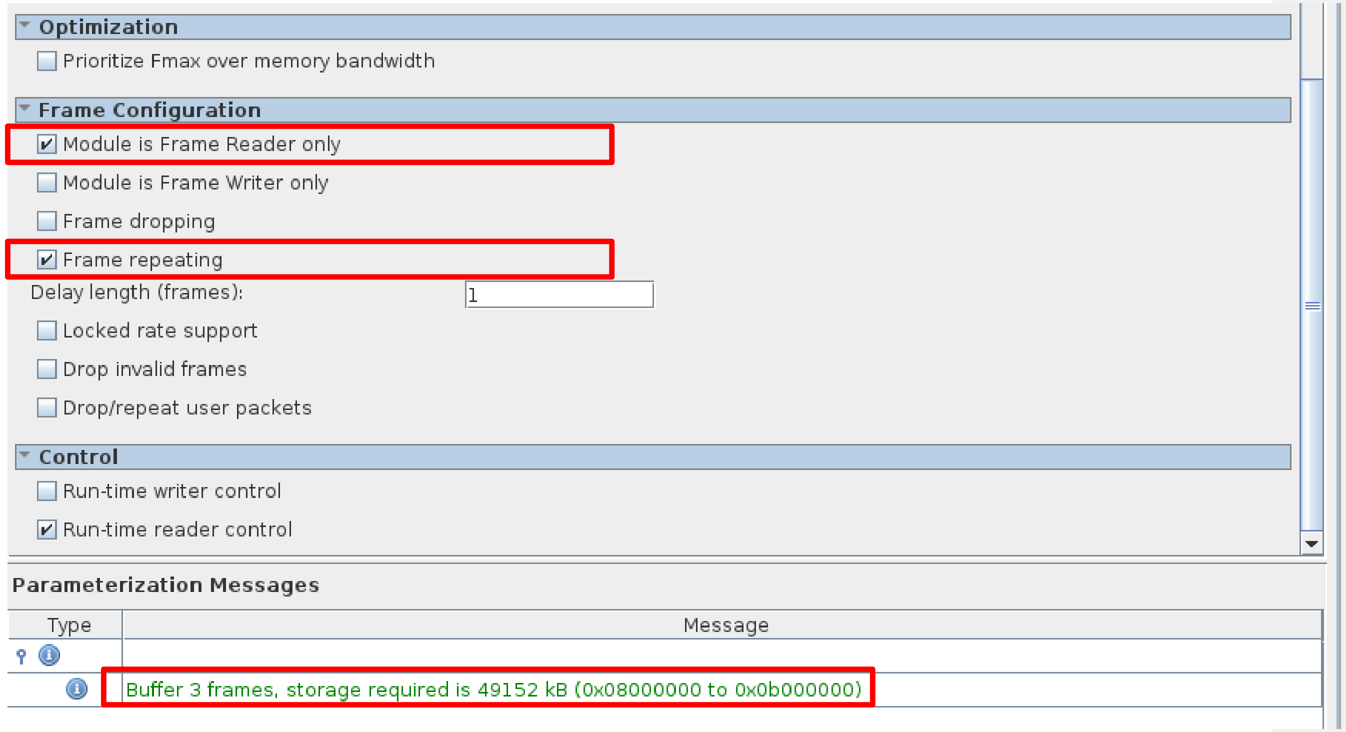
- **osd_vfr** has a special configuration as Frame Reader only
- 4 color planes to enable **ARGB**
- Base address to 0x08000000



Adding OSD overlay – HW flow

Memory Map – 6/7

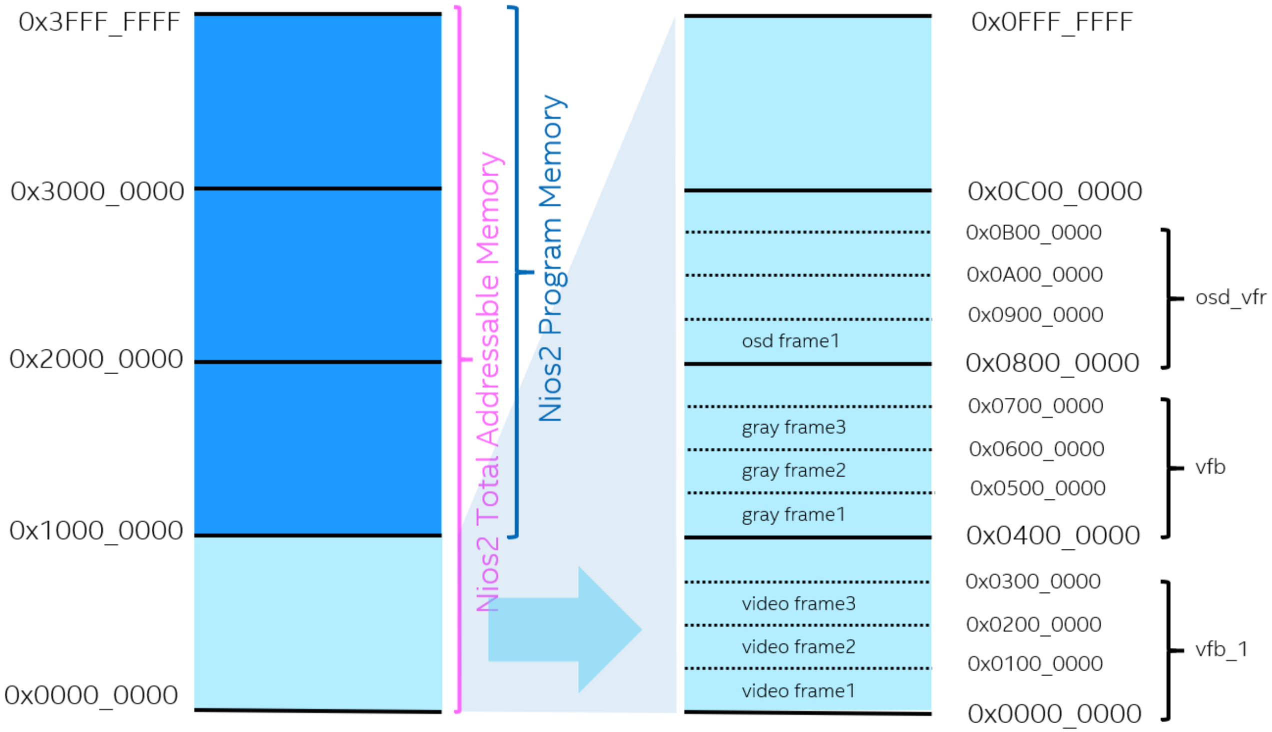
- Frame Reader only
- Frame repeating
- Run-time reader control
- We have now 3 video frames allocated to fixed addresses
  - 0x08000000
  - 0x09000000
  - 0x0a000000



Adding OSD overlay – HW flow

Memory Map – 7/7

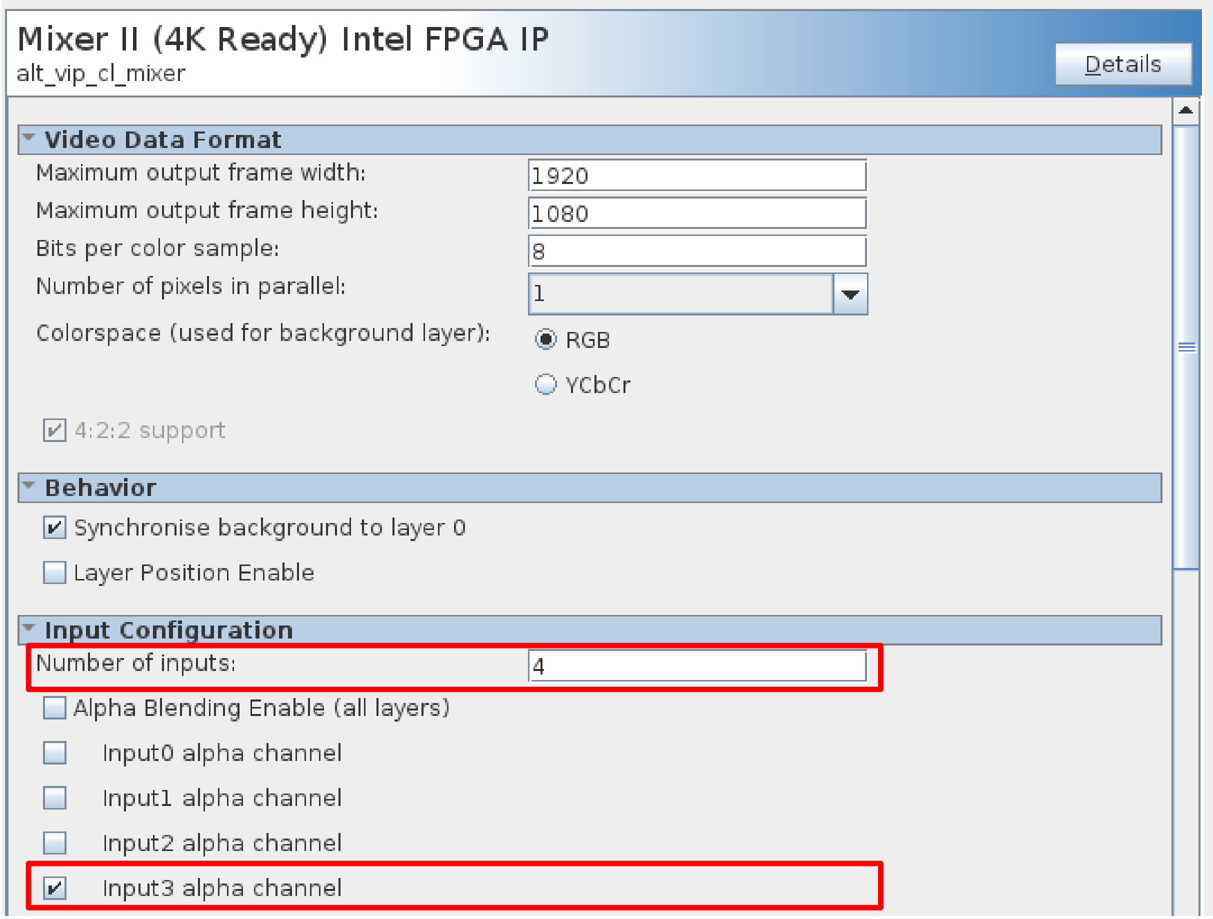
- DDR3 memory space sliced into different masters
  - Non-overlapping regions
- We allocate up to 0x1000_0000 for video memory
- Nios2 can address the total space, however program memory is set above 0x1000_0000



Adding OSD overlay – HW flow

Mixer – 1/3

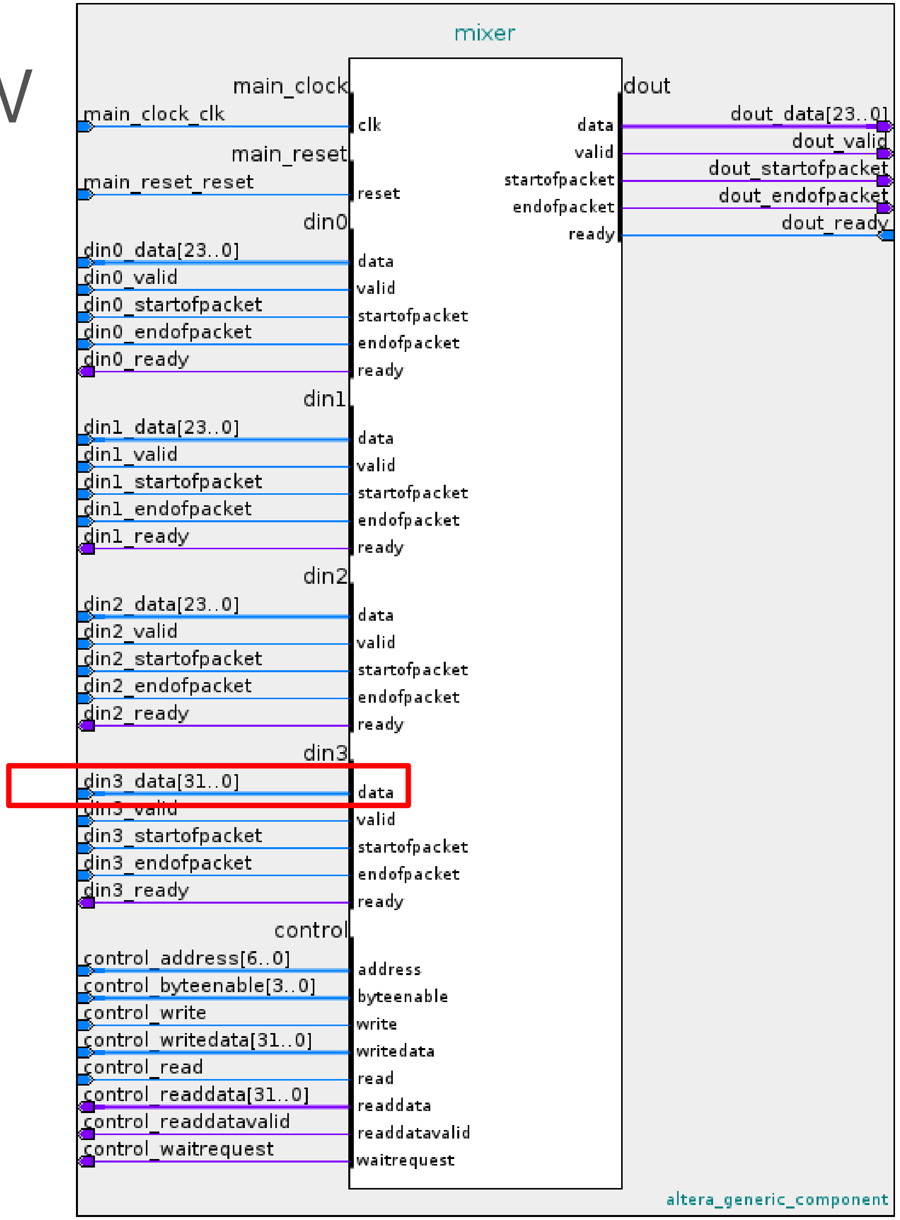
- We enable 4 layers
  - Tpg, original, rgb2gray, osd
- We activate osd Alpha channel
- 3 Alpha modes
  - No blending, opaque
  - Static, run-time prog value
  - **Per-pixel streaming**
- More details in VIP User Guide
  - https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_vip.pdf



Adding OSD overlay – HW flow

Mixer – 2/3

- 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 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 the bits per pixel per color parameter.
- For 8-bit alpha values
  - 255 full transparency
  - 0 fully opaque



Adding OSD overlay – HW flow

Mixer – 3/3

- 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.
- 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]

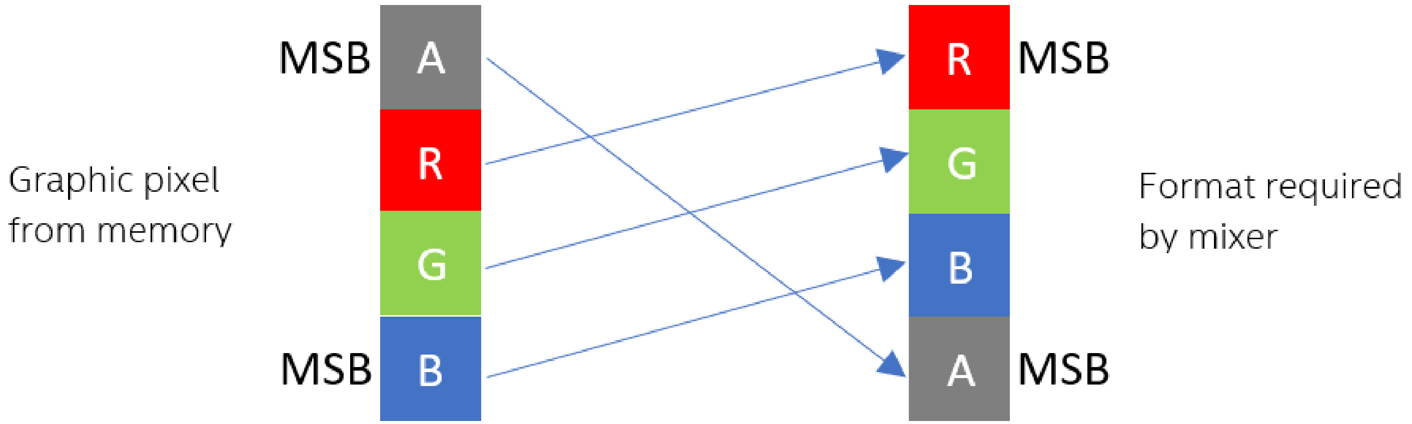
- But the graphic lib running on Nios writes pixels in the format:

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

Adding OSD overlay – HW flow

Color Plane Sequencer – 1/2

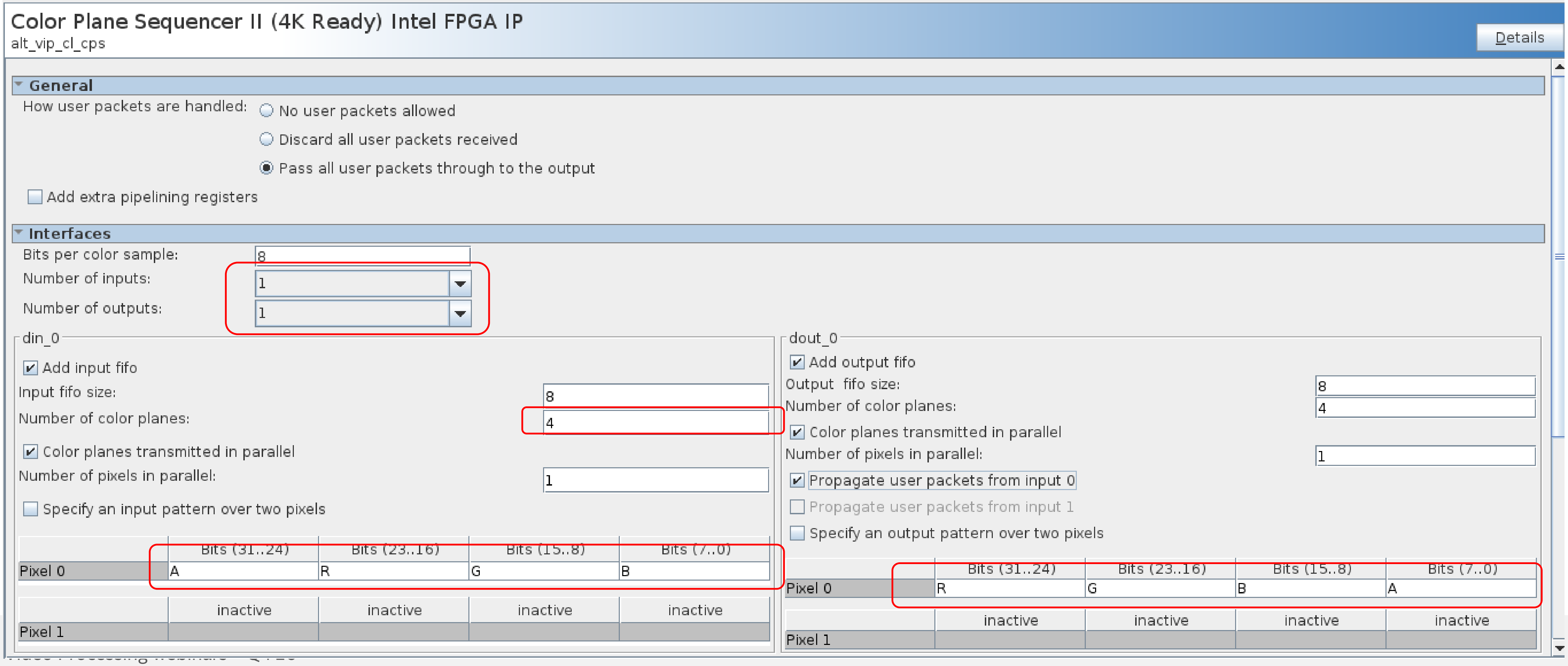
- We need to make some color arrangements
- Use Color Plane Sequencer



Adding OSD overlay – HW flow

Color Plane Sequencer – 2/2

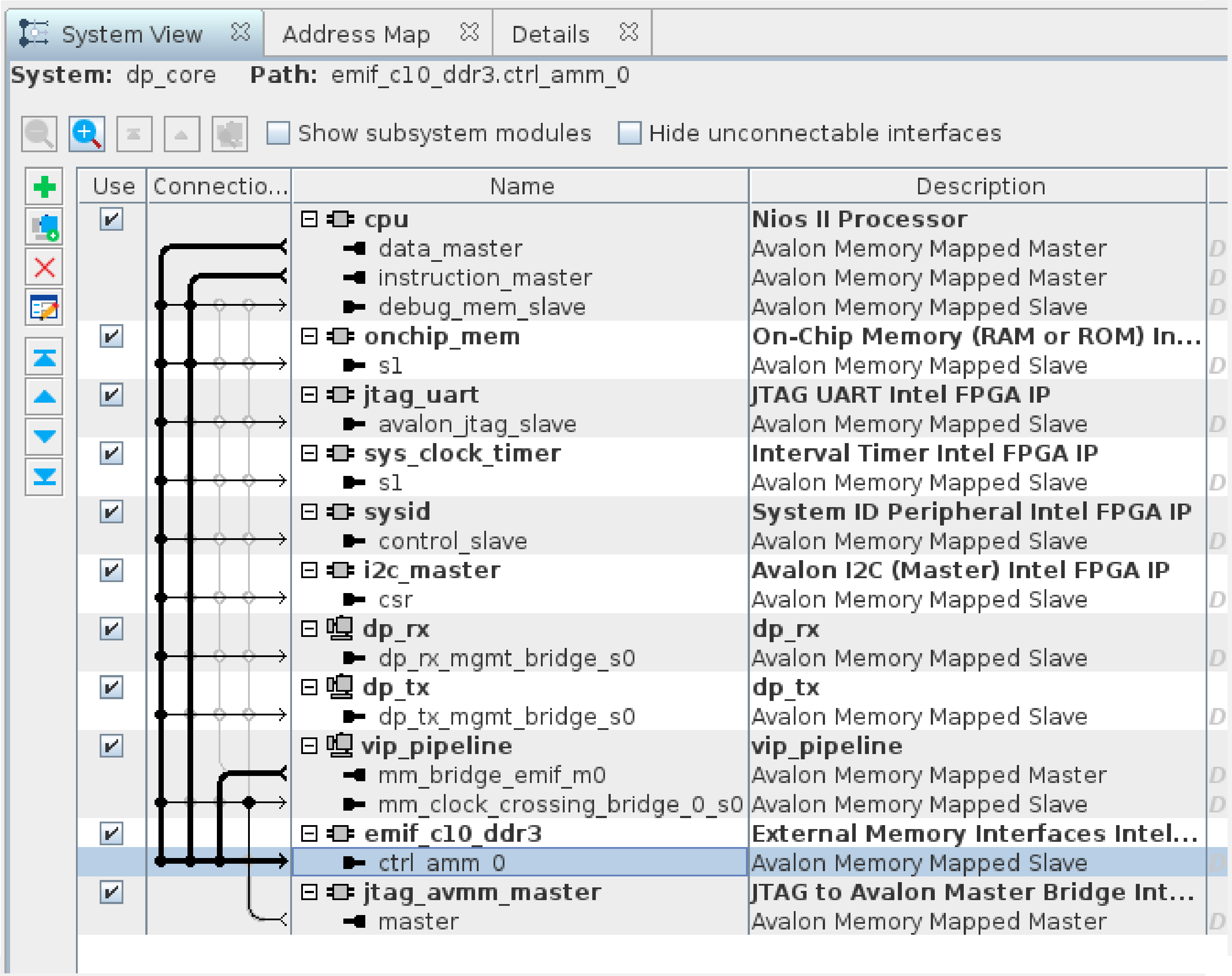
- 1 IN -> 1 OUT, 4 color planes >> Reassing components



Adding OSD overlay – HW flow

Connecting the Nios2 CPU – 1/4

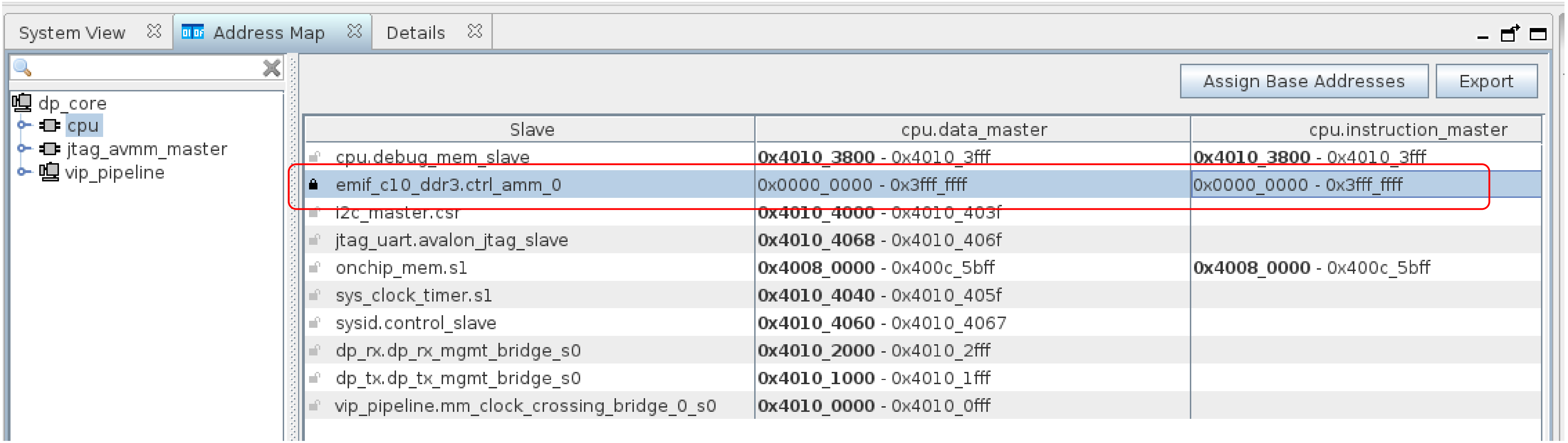
- Open **`dp_core.qsys`**
- Filter by Avalon-MM
- The **cpu** data and instruction masters are connected to **EMIF**
  - To execute the program
  - To write into graphic memory block (addressed by **osd_vfr**)



Adding OSD overlay – HW flow

Connecting the Nios2 CPU – 2/4

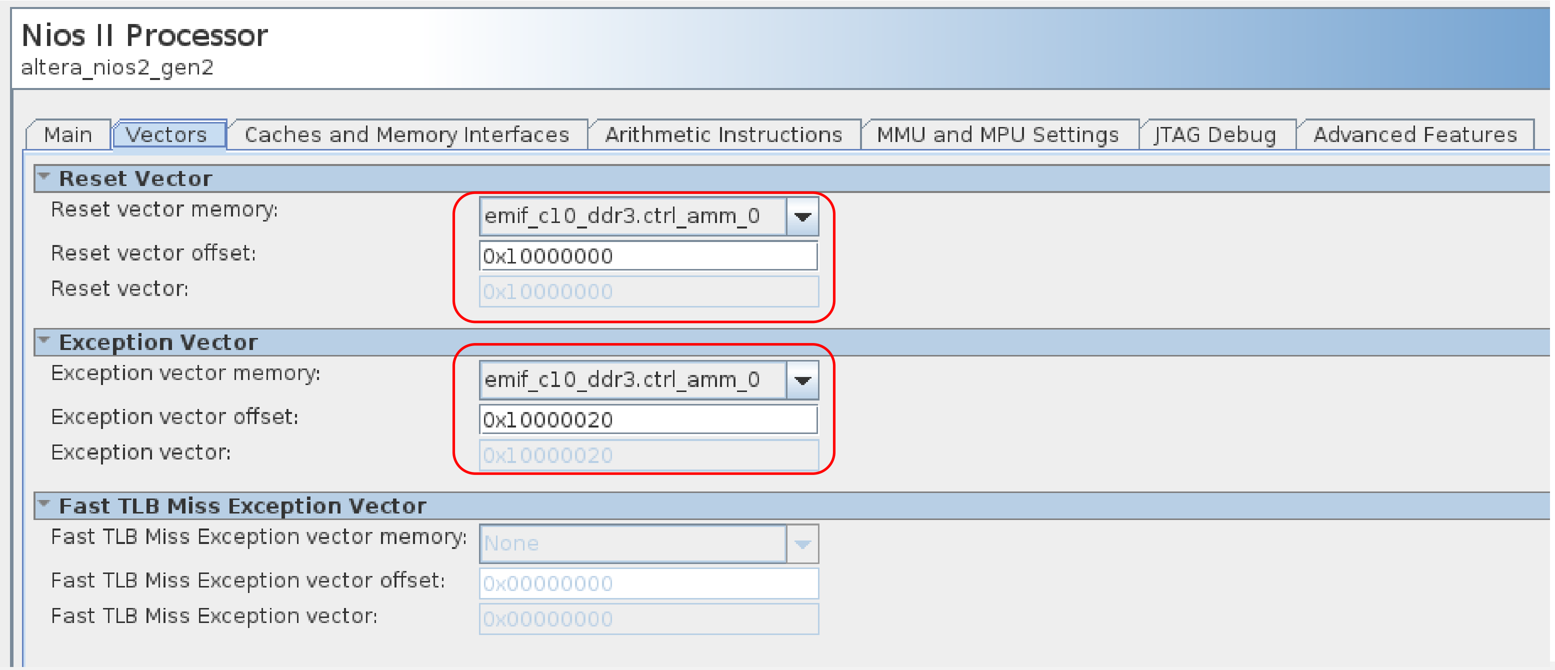
- CPU can address the complete 1GB DDR3 space
- We need to map the program memory out of the video space
  - Configure cpu vectors



Adding OSD overlay – HW flow

Connecting the Nios2 CPU – 3/4

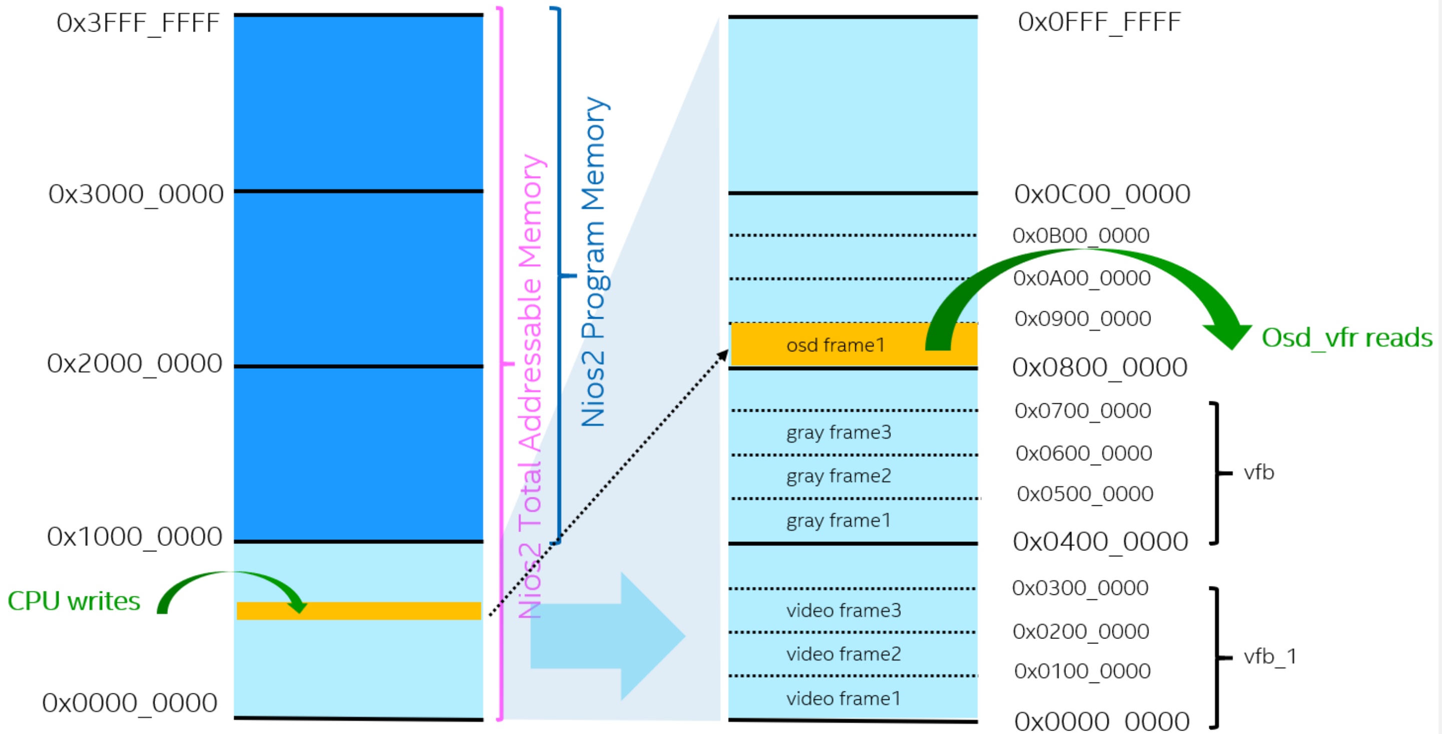
- Add offset to Reset and Exception vectors
- Linker will map program symbols (code & data) starting at 0x1000_0000



Adding OSD overlay – HW flow

Connecting the Nios2 CPU – 4/4

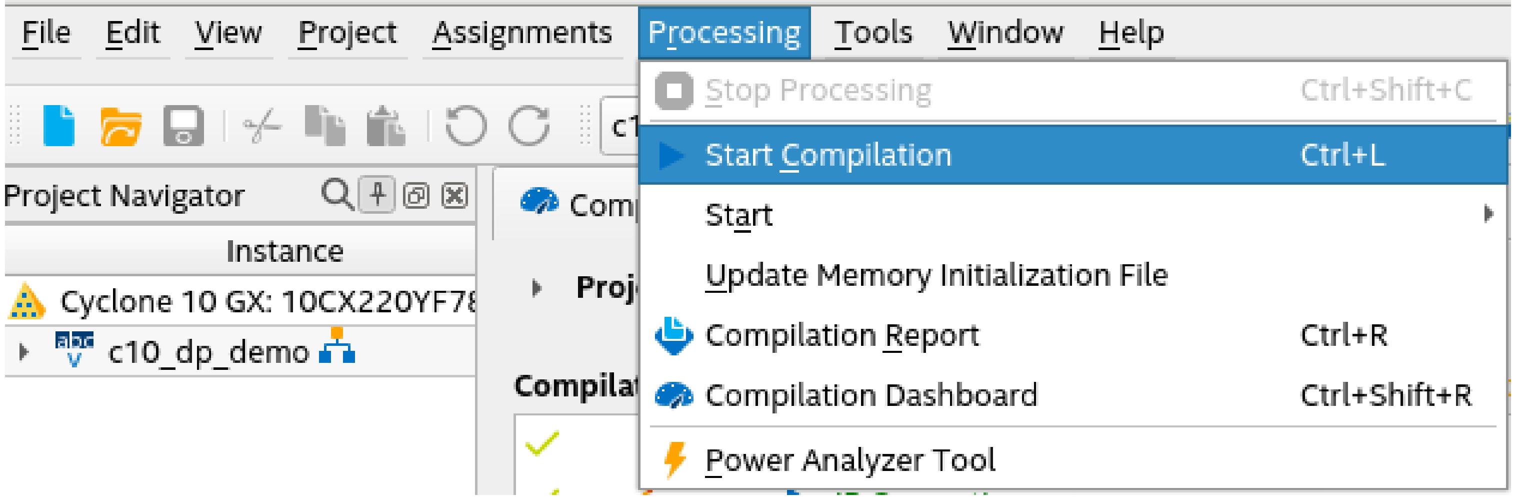
- Program memory is low bounded to 0x1000_0000
- But **cpu:data.master** still can access to full space
- We will create a pointer to 0x0800_0000 to write graphic information
- Then will be consumed by **osd_vfr** and sent to the mixer



Adding OSD overlay – HW flow

Generate bitstream

- Save **`dp_core.qsys`** and generate HDL files
- No need to modify top level file
- Processing->Start_Compilation



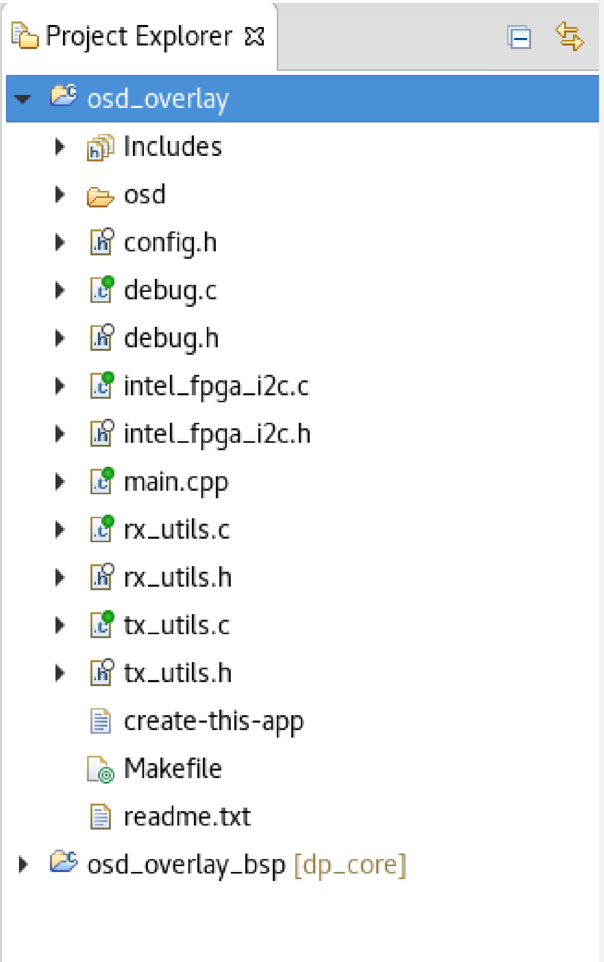
Session 4 – Adding OSD overlay

SW Development

Adding OSD overlay – SW flow

Create application and bsp

- As we did in previous sessions, generate the software application for Nios II processor using the supplied source code files.
  - Create a workspace folder under <project_dir>/software
  - In Eclipse, create a new application and bsp from template (**osd_overlay**)
  - Configure the bsp
  - Add DisplayPort libraries to the application
  - Copy files from **source** folder to **osd_overlay**
- For a comprehensive step guide checkout the `Session4_OSD_Overlay_lab_v1.pdf` manual in the repo
  - https://github.com/perezfra/VIP_webinars_Intel_FPGA



Adding OSD overlay – SW flow

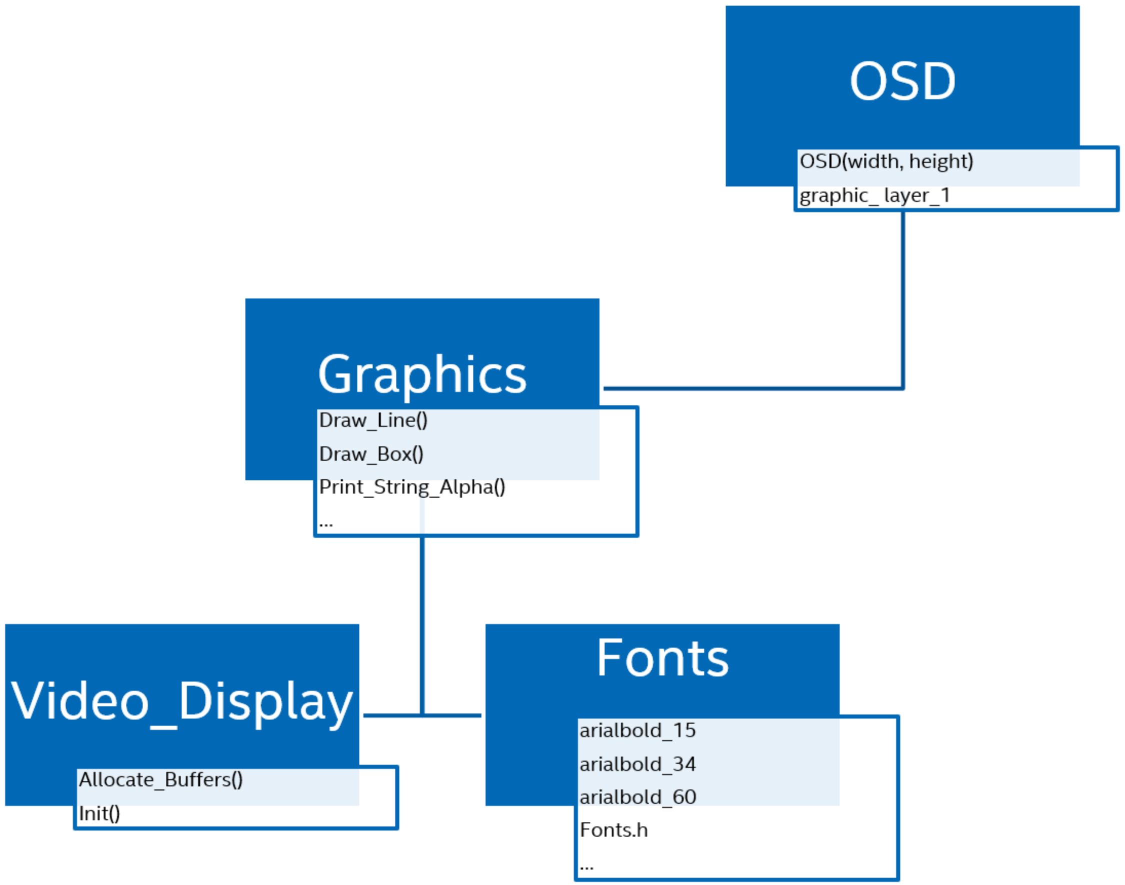
Examine the code

- **source** folder content
- We have already familiar files
- **osd** folder contains the graphic lib implementation

Adding OSD overlay – SW flow

Exploring the graphic library

- **osd** folder content
  - OSD class
  - Graphics class
  - Video_Display class
  - fonts



Adding OSD overlay – SW flow

Configuring components in main.cpp – 1/3

- Include new components
  - #include "Frame_Reader.hpp"
  - #include "Switch.hpp"
- Create a Frame_Reader object
  - Frame_Reader **osd_vfr**(VIP_PIPELINE_OSD_VFR_BASE, -1); // From system.h, no interrupt
- Configure object parameters
  - **osd_vfr**.set_frame_information(1920,1080,0); // This is the resolution of our output image
  - **osd_vfr**.set_frame_address(0x8000000); // Frame Buffer Memory Base Address
  - **osd_vfr**.start(); // Set the GO bit to start the component

Adding OSD overlay – SW flow

Configuring components in main.cpp – 2/3

- Create a Switch object
  - Switch **swi**(VIP_PIPELINE_SWI_BASE);
- Configure object parameters

```
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();
```

- From VIP User Guide

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.

Adding OSD overlay – SW flow

Configuring components in main.cpp – 3/3

- Create a Mixer object with 4 layers
  - Mixer **mixer**(VIP_PIPELINE_MIXER_BASE, 4); // 4 layers mixer: background, original video, rgb2gray, osd
- Configuring the Mixer

```
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();
```

Adding OSD overlay – SW flow

Initialize graphics library – 1/5

- Create a OSD object with full screen resolution
  - OSD **osd**(1920, 1080);
- OSD class declaration

```
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

};
```

Adding OSD overlay – SW flow

Initialize graphics library – 2/5

- OSD constructor

```
/*
 * 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);
```

Adding OSD overlay – SW flow

Initialize graphics library – 3/5

- Graphics initialization

```
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()` uses the internal property vd_buffer_location to remap the pointer.

Adding OSD overlay – SW flow

Initialize graphics library – 4/5

- Buffers allocation

```
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;
}
```

creates a dynamic memory buffer of a video frame's size

uses the internal property vd_buffer_location to remap the pointer

Adding OSD overlay – SW flow

Initialize graphics library – 5/5

- Clear the buffer

```
/*
 * 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);
}
```

We draw a whole frame filled box to initialize random content to a black “transparent” color.

Adding OSD overlay – SW flow

Using the graphics library – 1/5

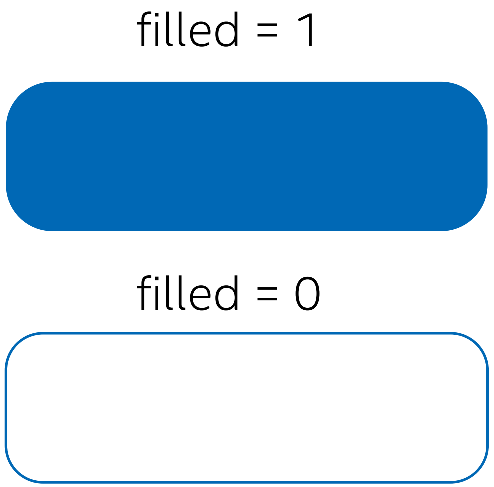
```
/*************************************************************
 *                       CLASS DEFINITION                   *
 *************************************************************/

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[]);
};
```

Adding OSD overlay – SW flow

Using the graphics library – 2/5

- Draw_Rounded_Box() example

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



- 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’ means only the outline is drawn
- char **transparency**: 0 means fully opaque, 255 means fully transparent. There are 255 possible transparency levels applicable.

Adding OSD overlay – SW flow

Using the graphics library – 3/5

- Print_String_Alpha() example

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

- 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.

Adding OSD overlay – SW flow

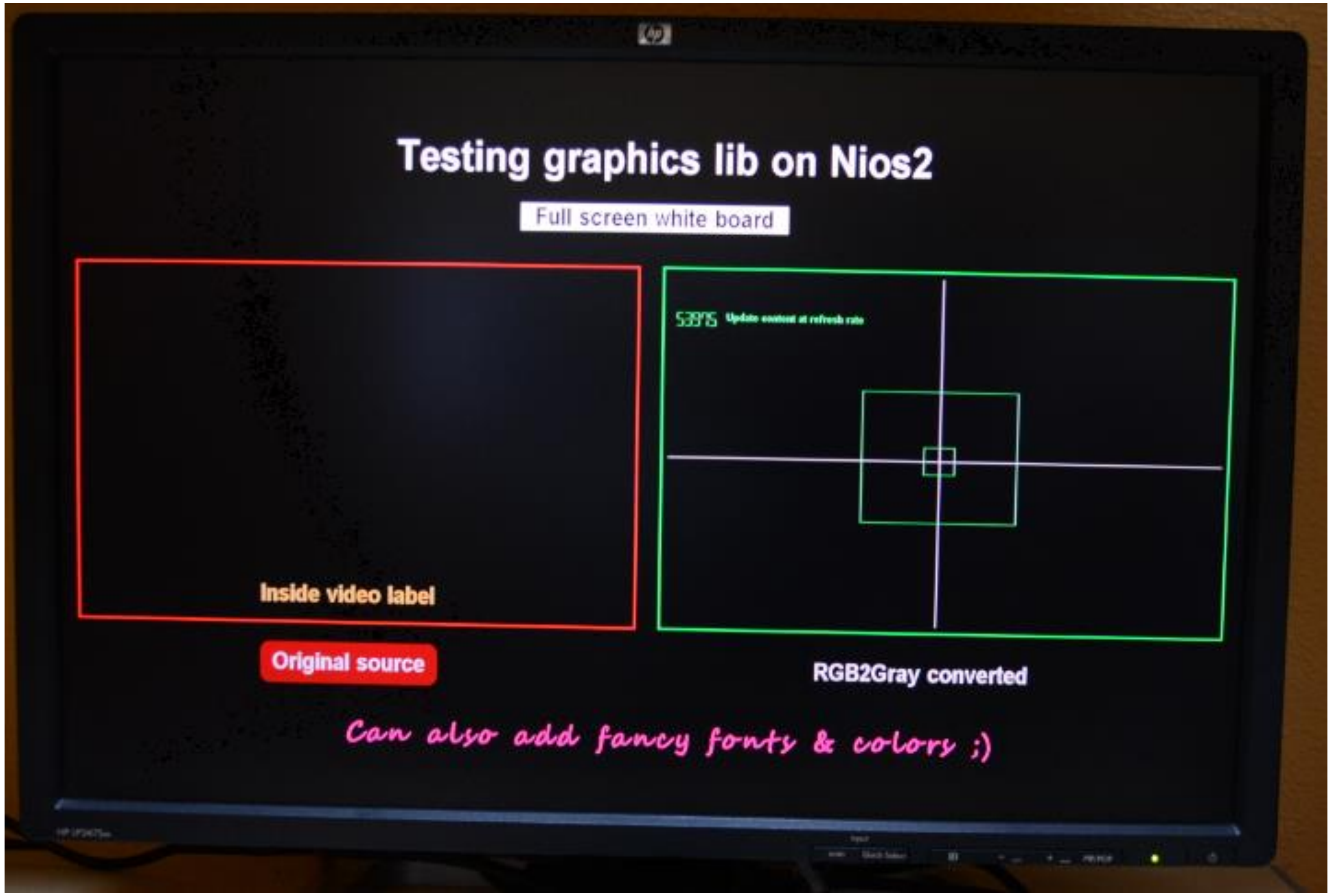
Using the graphics library – 4/5

- Graphic composition example

```
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();
```

Adding OSD overlay – SW flow

Using the graphics library – 5/5

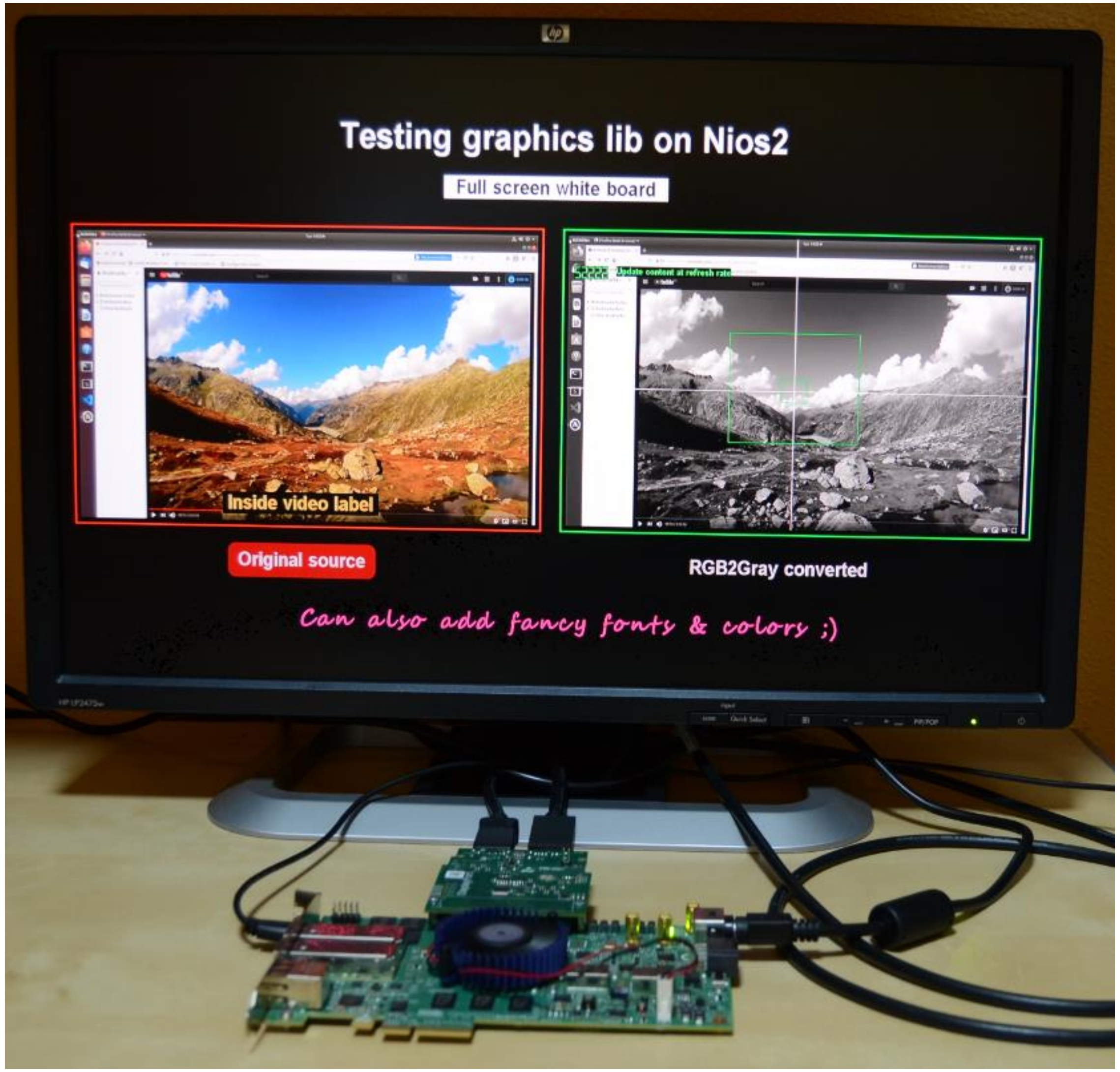


Session 4 – Adding OSD overlay

Running the application

Running the application

- Connect a display port video source and a monitor to the kit
- Open the Quartus programmer and download .sof
- In eclipse, launch the application for the Nios CPU
- See how the live captured video is converted to gray scale and displayed on the monitor



LIVE DEMO

Video processing on FPGAs made easy

Content available in the GitHub repo:

https://github.com/perezfra/VIP_webinars_Intel_FPGA

Summary

- 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 overlay 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.