Intel FPGA VIP webinar series

Session 3
Develop a custom VIP component

Francisco Perez
Intel FPGA Field Applications Engineer

v.1 – November 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

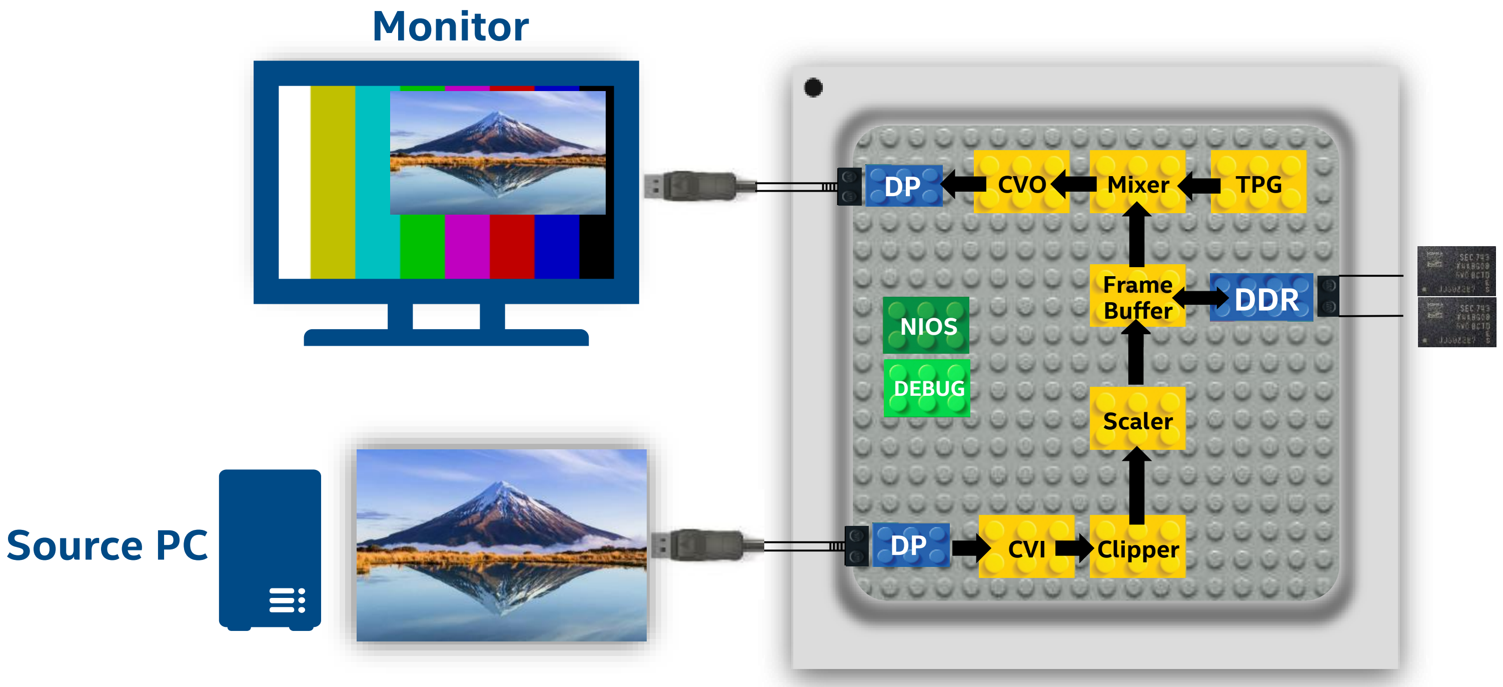
1. Building a video processing pipeline (Oct, 14th and 21st)
   - 1.1- DisplayPort loopback example design implementation
   - 1.2- Building our first video pipeline using VIP suite cores
   - 1.3- Complete End2End VIP pipeline
2. Strategies to debug a VIP pipeline (Nov, 4th)
   - Overview of system 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 sessions

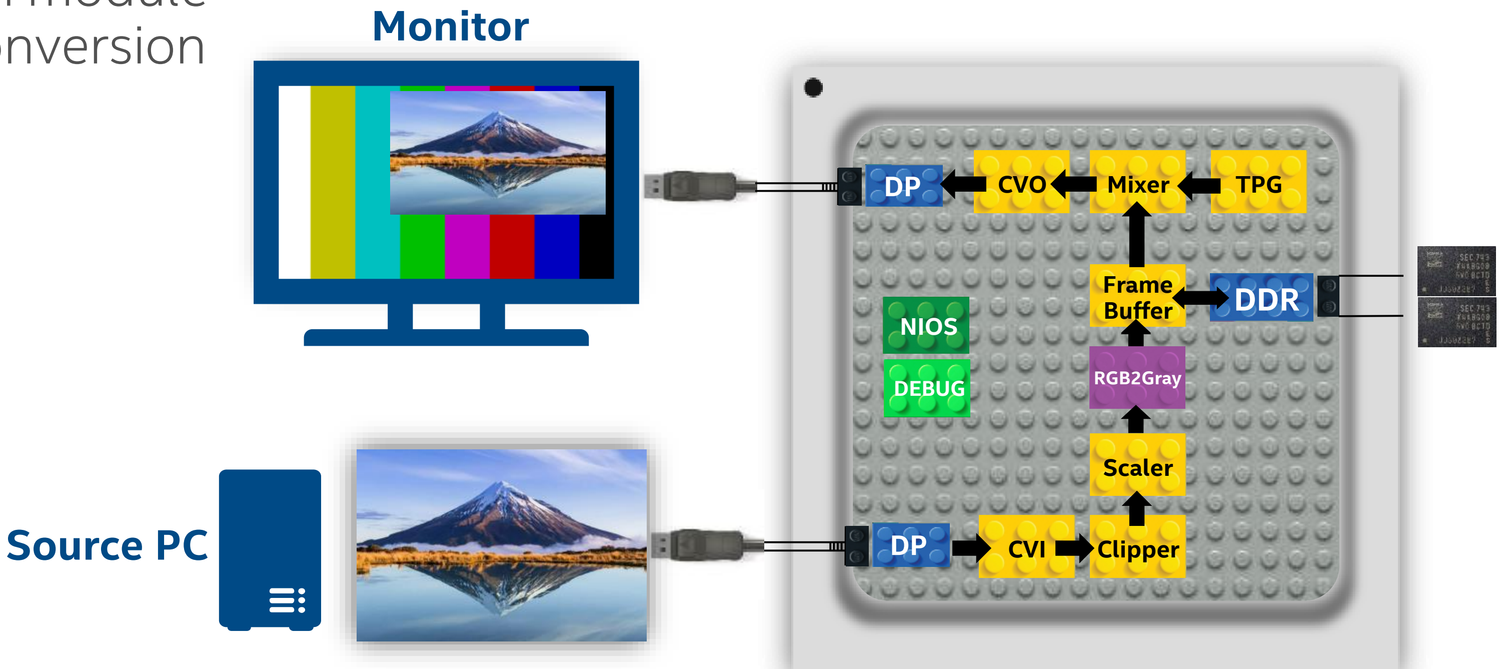
1. Build a video pipeline using VIP cores (cropping, scaling and mixing with a background) and debug



Building a Video Processing Pipeline

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

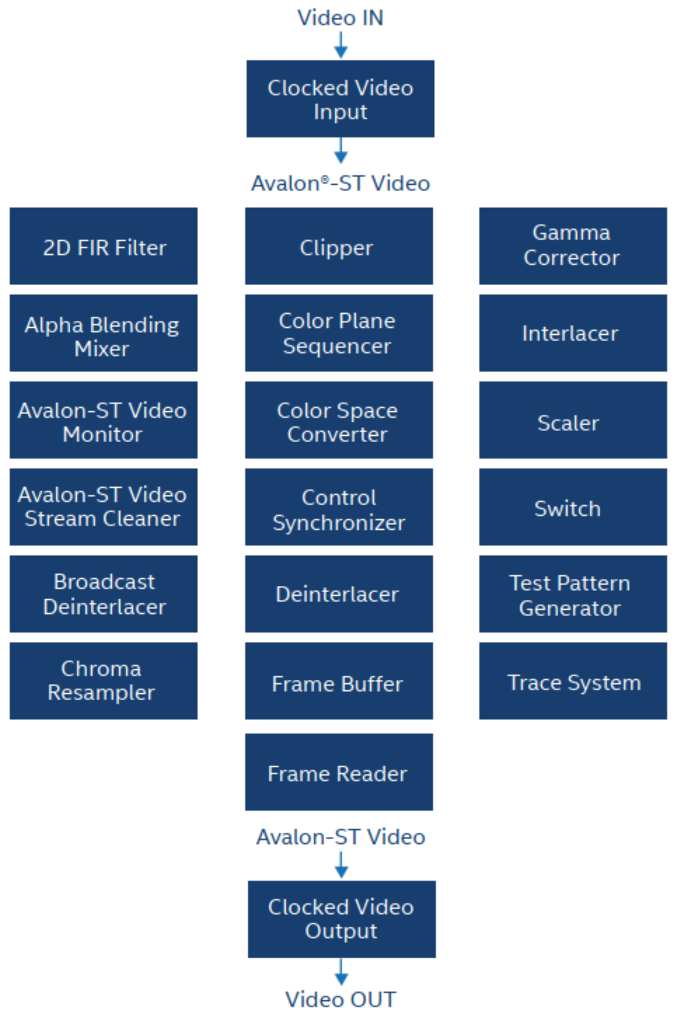


Session 3 – Create custom VIP modules

RGB2Gray converter

Video and Image Processing Suite

- Collection of 20+ IP functions with support for all Intel FPGAs
- Covers most of the basic infrastructure of a typical video pipeline
- How you can differentiate, or add your own blocks for your application?
  - Develop a custom component



https://www.intel.com/content/www/us/en/programmable/products/intellectual-property/ip/dsp/m-alt-vipsuite.html

RGB2Gray converter

- In this design we adding to the `vip_pipeline.qsys` subsystem, a newly created block called **RGB2Gray**. This will be a custom module we are developing to convert our colour video to a monochrome grayscale version.
- To perform this conversion we use the **Weighted** or **Luminosity** method

Original image

Monochrome image

```
Gray = ((0.3 * R) + (0.59 * G) + (0.11 * B))
```

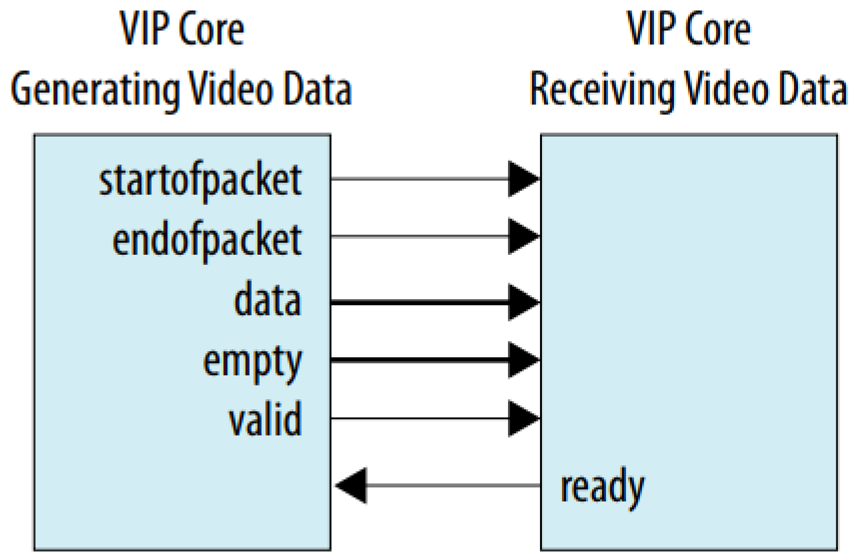
Session 3 – Create custom VIP modules

Recap Avalon-ST video protocol

Avalon-ST Video - Recap

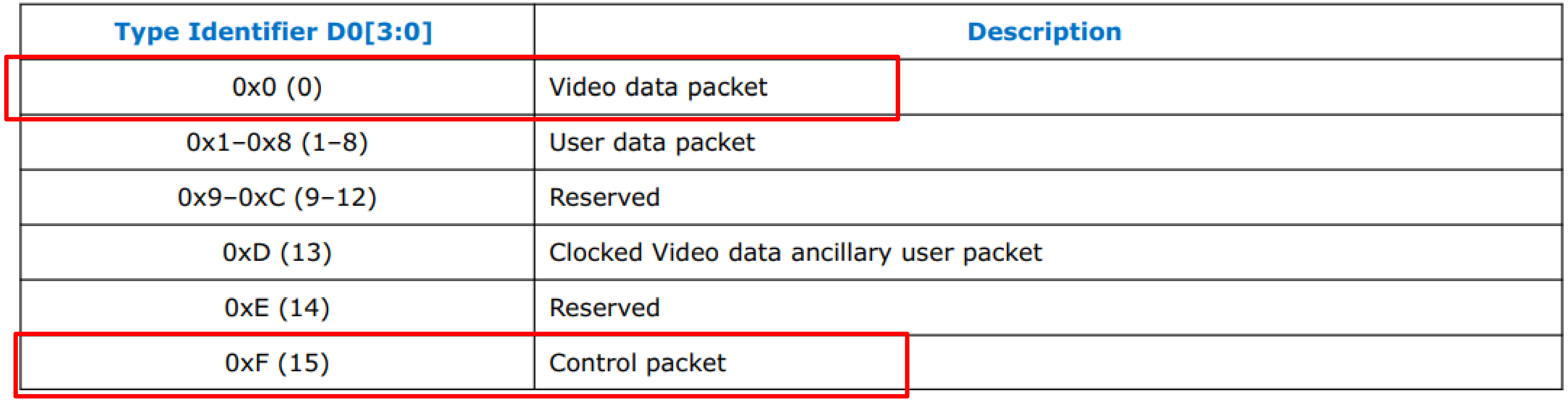
Introduction

- In the previous session we covered, in details, the Avalon-ST Video protocol: which is the transport method used to flow information between the VIP Suite cores.
- Flow control is handled by the `ready` and `valid` signals. The downstream core can apply backpressure and stalls the pipeline.
- Can be configured to support many different resolutions and pixel formats



Avalon-ST Video - Recap

Avalon-ST Video Packet Type Identifiers



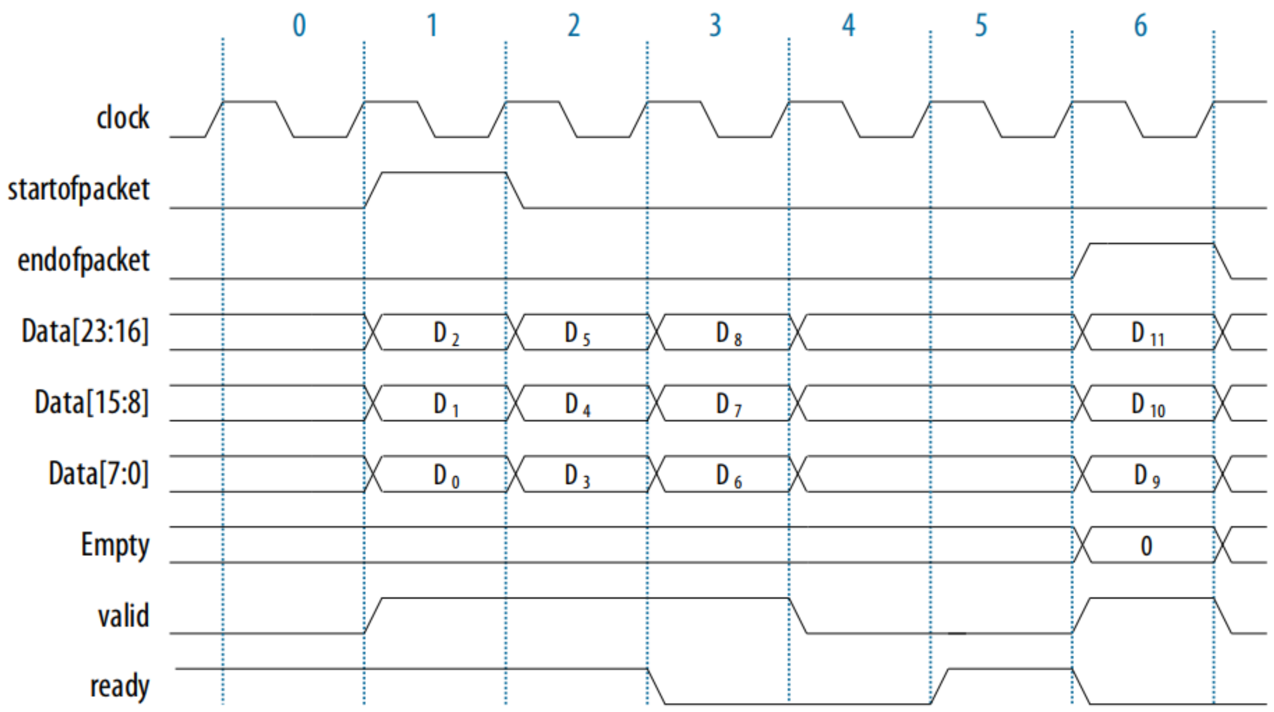
Type Identifier D0[3:0]	Description
0x0 (0)	Video data packet
0x1–0x8 (1–8)	User data packet
0x9–0xC (9–12)	Reserved
0xD (13)	Clocked Video data ancillary user packet
0xE (14)	Reserved
0xF (15)	Control packet

The type of packet is determined by the lowest 4 bits of the first symbol transmitted.

Avalon-ST Video - Recap

Avalon-ST Video packet transmission

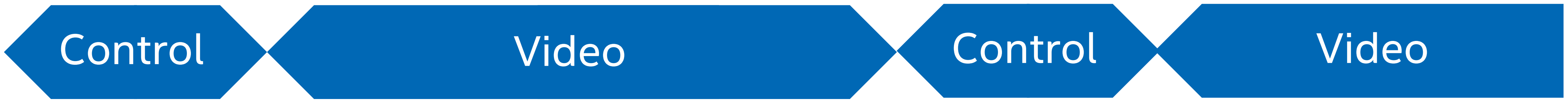
- A “ready latency” of 1 is used for Avalon streaming video.
- The receiving video sink drops its ready signal in cycle 3, to indicate that it is not ready to receive any data in cycles 4 or 5.
- As the ready signal returns high in cycle 5, the video source data in cycle 6 is safely registered by the sink



Avalon-ST Video - Recap

Operation

- Most Avalon-ST Video compliant VIP IP cores require an Avalon-ST control packet to be received before any video packets, so that line buffers and other sub-components can be configured.
- When a VIP IP core receives an Avalon-ST Video control packet, the IP core decodes the height, width, and interlacing information from that packet and interprets any following Avalon-ST Video packets as being video of that format until it receives another control packet.



Session 3 – Create custom VIP modules

Custom VIP HDL template

Custom VIP HDL template

Focus on your algorithm

- The template is a collection of files included in the ../custom_ip/hdl folder of the extracted archive file
- Decodes & generates control and video packets
- Handles the backpressure mechanism with ready/valid
- It’s provided in clear text verilog files
- Allows you to focus on your “secret sauce”

alt_vip_common_control_packet_decoder.v

alt_vip_common_control_packet_encoder.v

alt_vip_common_flow_control_wrapper.v

alt_vip_common_stream_input.v

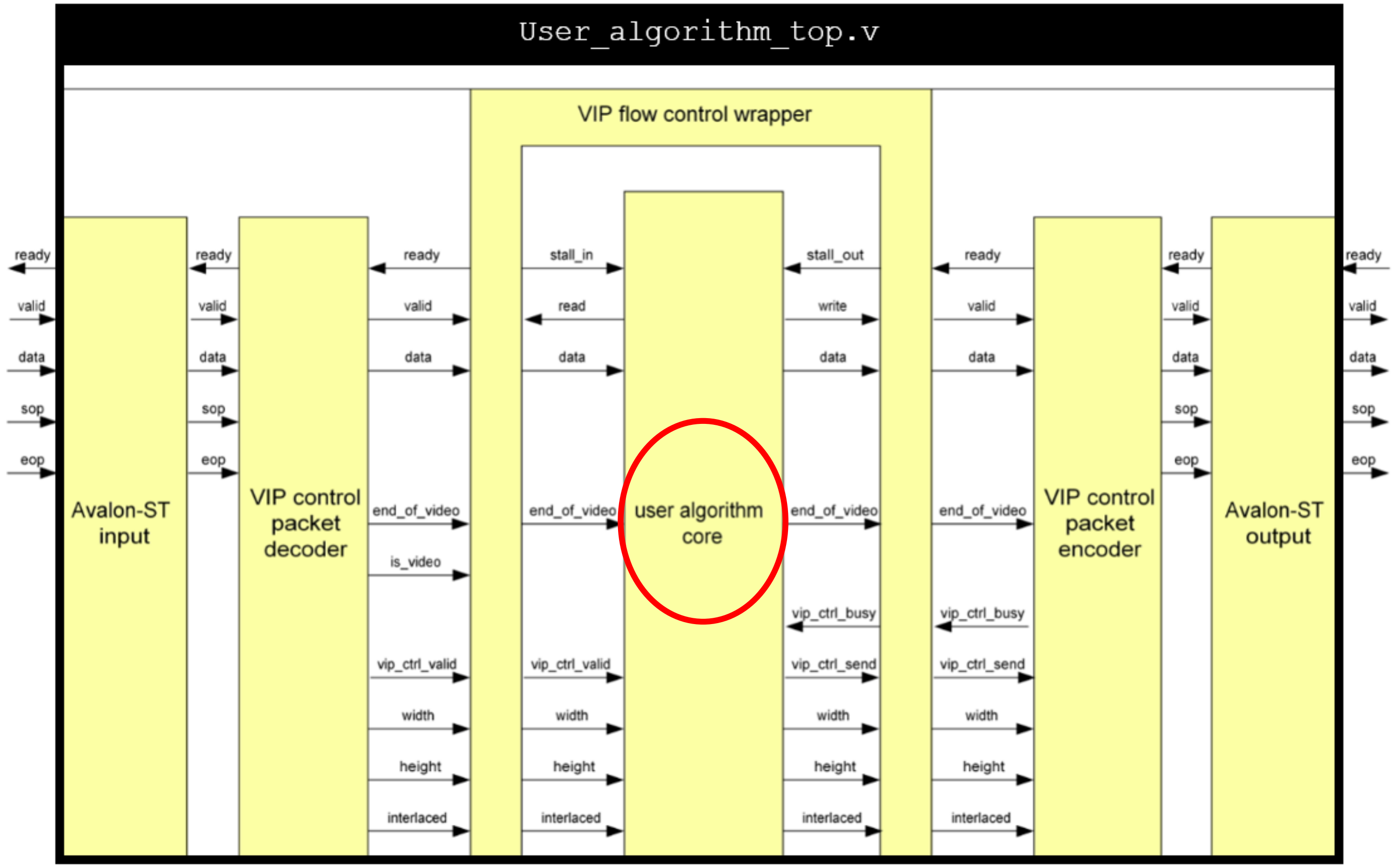
alt_vip_common_stream_output.v

user_algorithm_core.v

user_algorithm_top.v

Custom VIP HDL template

User Algorithm top anatomy



Custom VIP HDL template

User Algorithm Core

- The user_algorithm_core.v file is divided into 3 main blocks:
  - Module ports to connect with the rest of template modules
  - Flow control processing
    - The block needed to connect the relevant flow control signals that allows the user algorithm block to operate according to Avalon-ST video protocol, being compliant with packet management and handling backpressure.
  - Data processing of user algorithm.
    - This is our target block and where we will add our stuff

User Algorithm Core

Module ports

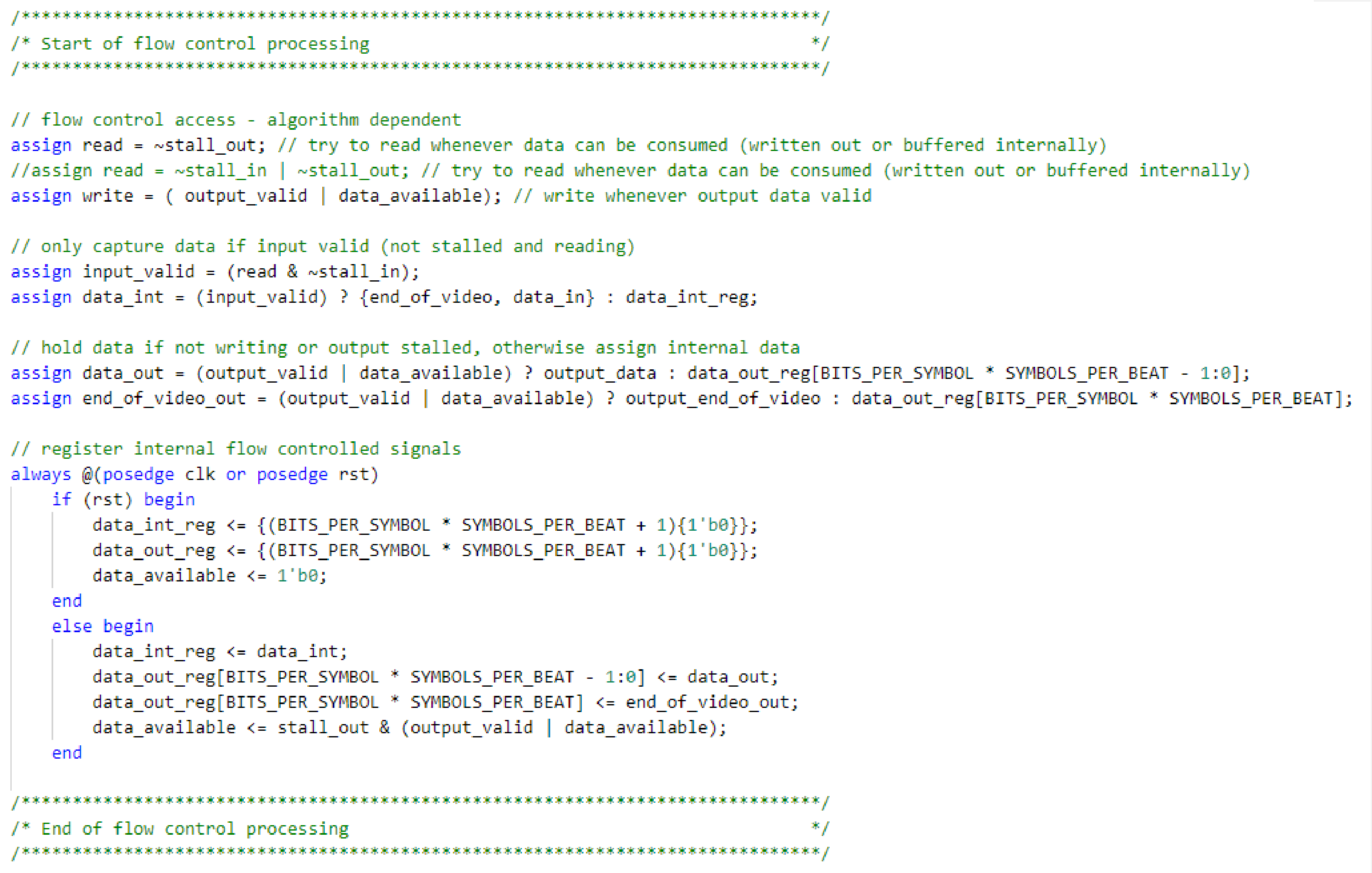
- Control signals from Avalon-ST decoder and flow wrapper (sink)
- Control signals to Avalon-ST encoder and flow wrapper (source)

```
module user_algorithm_core

        #(parameter BITS_PER_SYMBOL = 8,
                parameter SYMBOLS_PER_BEAT = 3)

        (       input           clk,
                input           rst,

                // interface to VIP control packet decoder via VIP flow control wrapper
                input           stall_in,
                output          read,
                input           [BITS_PER_SYMBOL * SYMBOLS_PER_BEAT - 1:0] data_in,
                input           end_of_video,

                input           [15:0] width_in,
                input           [15:0] height_in,
                input           [3:0] interlaced_in,
                input           vip_ctrl_valid,

                // interface to VIP control packet encoder via VIP flow control wrapper
                input           stall_out,
                output          write,
                output          [BITS_PER_SYMBOL * SYMBOLS_PER_BEAT - 1:0] data_out,
                output          end_of_video_out,

                output  reg     [15:0] width_out,
                output  reg     [15:0] height_out,
                output  reg     [3:0] interlaced_out,
                input           vip_ctrl_busy,
                output  reg     vip_ctrl_send);
```

User Algorithm Core

Flow control processing

- Generate and register internal control flow signals

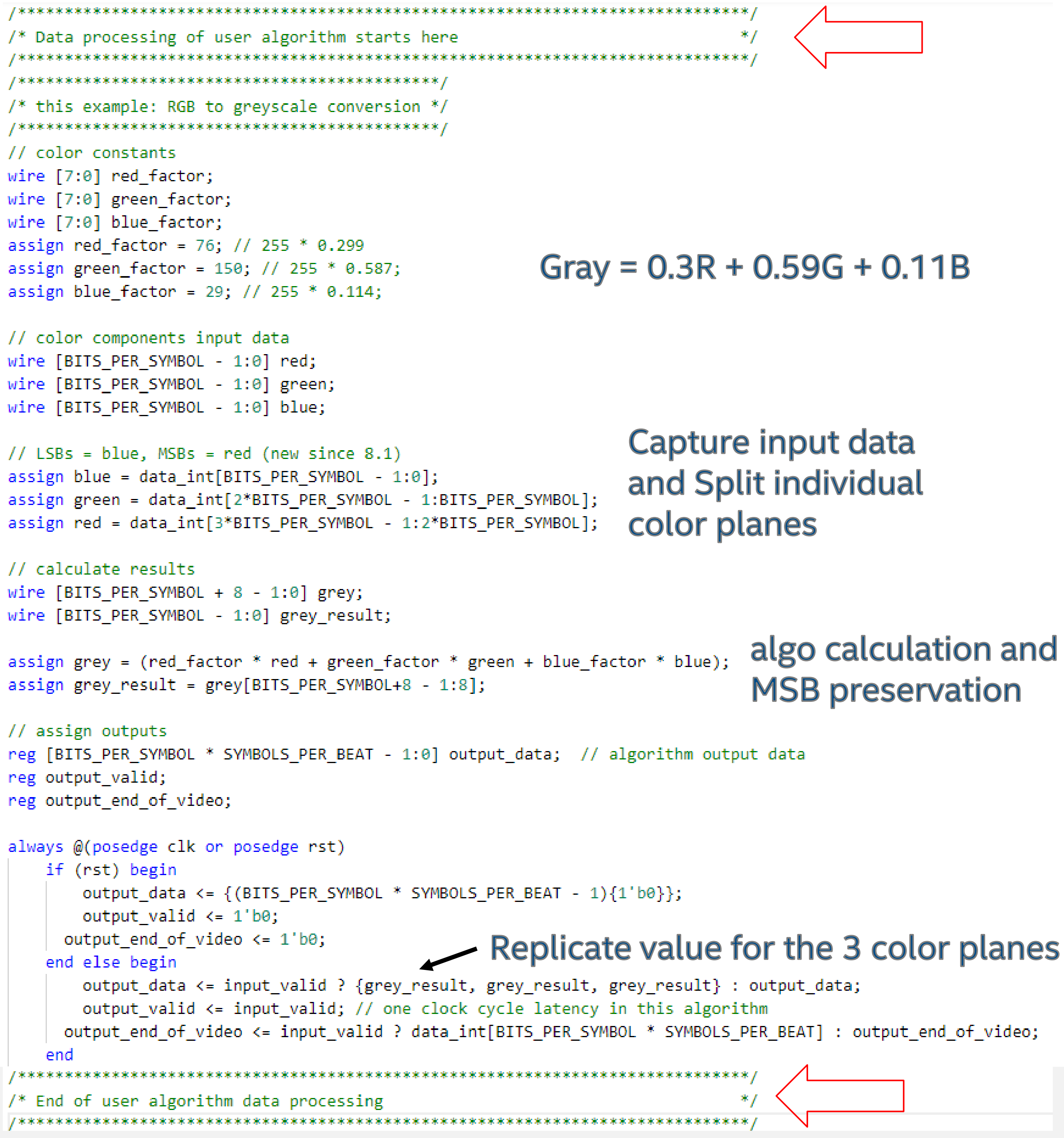


```
/*****************************************************************************/
/* Start of flow control processing                                         */
/*****************************************************************************/

// flow control access - algorithm dependent
assign read = ~stall_out; // try to read whenever data can be consumed (written out or buffered internally)
//assign read = ~stall_in | ~stall_out; // try to read whenever data can be consumed (written out or buffered internally)
assign write = ( output_valid | data_available); // write whenever output data valid

// only capture data if input valid (not stalled and reading)
assign input_valid = (read & ~stall_in);
assign data_int = (input_valid) ? {end_of_video, data_in} : data_int_reg;

// hold data if not writing or output stalled, otherwise assign internal data
assign data_out = (output_valid | data_available) ? output_data : data_out_reg[BITS_PER_SYMBOL * SYMBOLS_PER_BEAT - 1:0];
assign end_of_video_out = (output_valid | data_available) ? output_end_of_video : data_out_reg[BITS_PER_SYMBOL * SYMBOLS_PER_BEAT];

// register internal flow controlled signals
always @(posedge clk or posedge rst)
    if (rst) begin
        data_int_reg <= {(BITS_PER_SYMBOL * SYMBOLS_PER_BEAT + 1){1'b0}};
        data_out_reg <= {(BITS_PER_SYMBOL * SYMBOLS_PER_BEAT + 1){1'b0}};
        data_available <= 1'b0;
    end
    else begin
        data_int_reg <= data_int;
        data_out_reg[BITS_PER_SYMBOL * SYMBOLS_PER_BEAT - 1:0] <= data_out;
        data_out_reg[BITS_PER_SYMBOL * SYMBOLS_PER_BEAT] <= end_of_video_out;
        data_available <= stall_out & (output_valid | data_available);
    end

/*****************************************************************************/
/* End of flow control processing                                           */
/*****************************************************************************/
```

User Algorithm Core

Process the user algo

- User algorithm boundaries clearly defined
- Assign conversión weights
- Do the calculation and keep the MSBs of the result
- Replicate the gray value for all 3 color planes to generate monochrome representation



```
/*******************************************************************************/
/* Data processing of user algorithm starts here                               */
/*******************************************************************************/
/*********************************************/
/* this example: RGB to greyscale conversion */
/*********************************************/
// color constants
wire [7:0] red_factor;
wire [7:0] green_factor;
wire [7:0] blue_factor;
assign red_factor = 76; // 255 * 0.299
assign green_factor = 150; // 255 * 0.587;
assign blue_factor = 29; // 255 * 0.114;

// color components input data
wire [BITS_PER_SYMBOL - 1:0] red;
wire [BITS_PER_SYMBOL - 1:0] green;
wire [BITS_PER_SYMBOL - 1:0] blue;

// LSBs = blue, MSBs = red (new since 8.1)
assign blue = data_int[BITS_PER_SYMBOL - 1:0];
assign green = data_int[2*BITS_PER_SYMBOL - 1:BITS_PER_SYMBOL];
assign red = data_int[3*BITS_PER_SYMBOL - 1:2*BITS_PER_SYMBOL];

// calculate results
wire [BITS_PER_SYMBOL + 8 - 1:0] grey;
wire [BITS_PER_SYMBOL - 1:0] grey_result;

assign grey = (red_factor * red + green_factor * green + blue_factor * blue);
assign grey_result = grey[BITS_PER_SYMBOL+8 - 1:8];

// assign outputs
reg [BITS_PER_SYMBOL * SYMBOLS_PER_BEAT - 1:0] output_data;  // algorithm output data
reg output_valid;
reg output_end_of_video;

always @(posedge clk or posedge rst)
    if (rst) begin
        output_data <= {(BITS_PER_SYMBOL * SYMBOLS_PER_BEAT - 1){1'b0}};
        output_valid <= 1'b0;
      output_end_of_video <= 1'b0;
    end else begin
        output_data <= input_valid ? {grey_result, grey_result, grey_result} : output_data;
        output_valid <= input_valid; // one clock cycle latency in this algorithm
      output_end_of_video <= input_valid ? data_int[BITS_PER_SYMBOL * SYMBOLS_PER_BEAT] : output_end_of_video;
    end
/*******************************************************************************/
/* End of user algorithm data processing                                       */
/*******************************************************************************/
```

Gray = 0.3R + 0.59G + 0.11B

Capture input data and Split individual color planes

algo calculation and MSB preservation

Replicate value for the 3 color planes

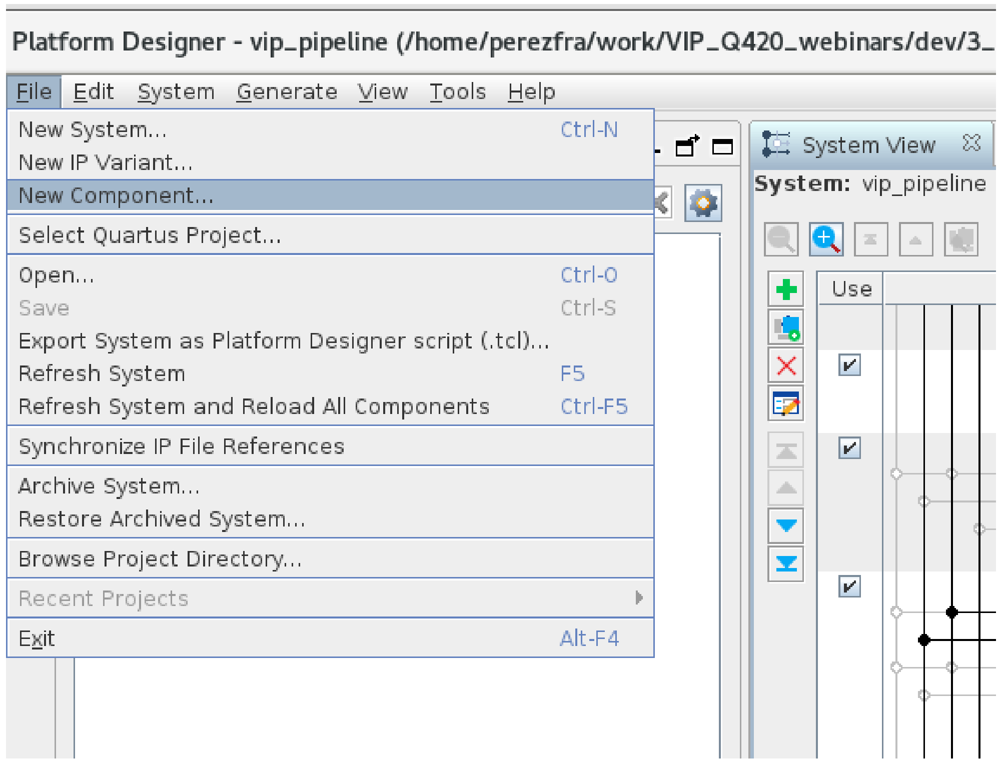
Session 3 – Create custom VIP modules

Using the component editor

Opening Component Editor

Create the user_algorithm_top_hw.tcl

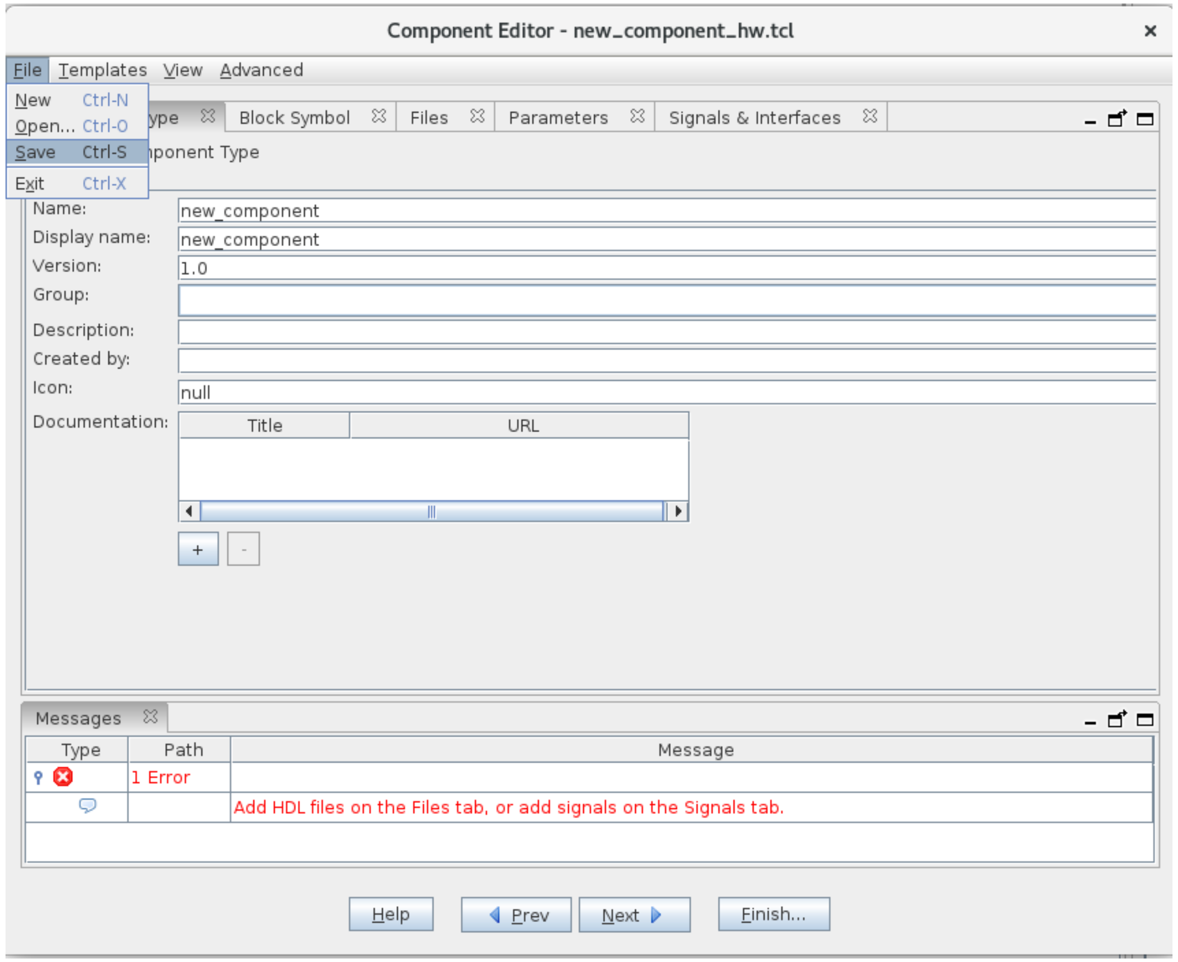
- Open the Quartus project located at `<extracted_folder>/quartus/c10_dp_demo.qpf`
- Open `<extracted_folder>/rtl/core/vip_pipeline.qsys` in Platform Designer
- In the main toolbar select **File->New Component** to open the Component Editor GUI


Platform Designer - vip_pipeline (/home/perezfra/work/VIP_Q420_webinars/dev/3_
File Edit System Generate View Tools Help
New System... Ctrl-N
New IP Variant...
New Component...
Select Quartus Project...
Open... Ctrl-O
Save Ctrl-S
Export System as Platform Designer script (.tcl)...
Refresh System F5
Refresh System and Reload All Components Ctrl-F5
Synchronize IP File References
Archive System...
Restore Archived System...
Browse Project Directory...
Recent Projects
Exit Alt-F4
System View
System: vip_pipeline
Use

Opening Component Editor

Create the user_algorithm_top_hw.tcl

- Component Editor allows the creation of new components only in the folder where the `*.qsys` file has been opened.
- In this case, we have opened the `<extracted_folder>/rtl/core/vip_pipeline.qsys` file, hence this is the root directory.
- Just click on **File->Save** to get the `new_component_hw.tcl` file generated and close the editor



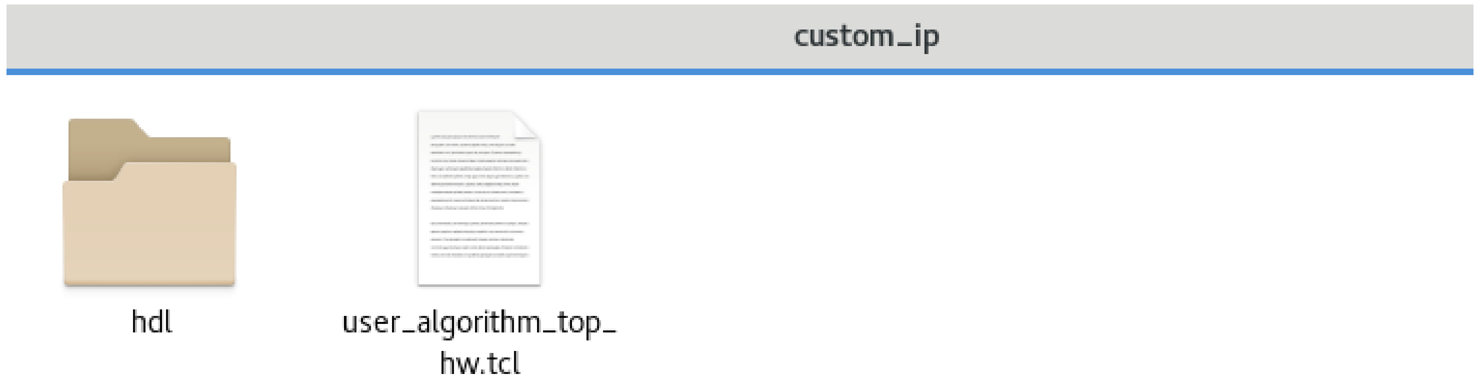
Name: new_component
Display name: new_component
Version: 1.0
Group:
Description:
Created by:
Icon: null
Documentation:

Type	Path	Message
	1 Error	
		Add HDL files on the Files tab, or add signals on the Signals tab.

Opening Component Editor

Create the user_algorithm_top_hw.tcl

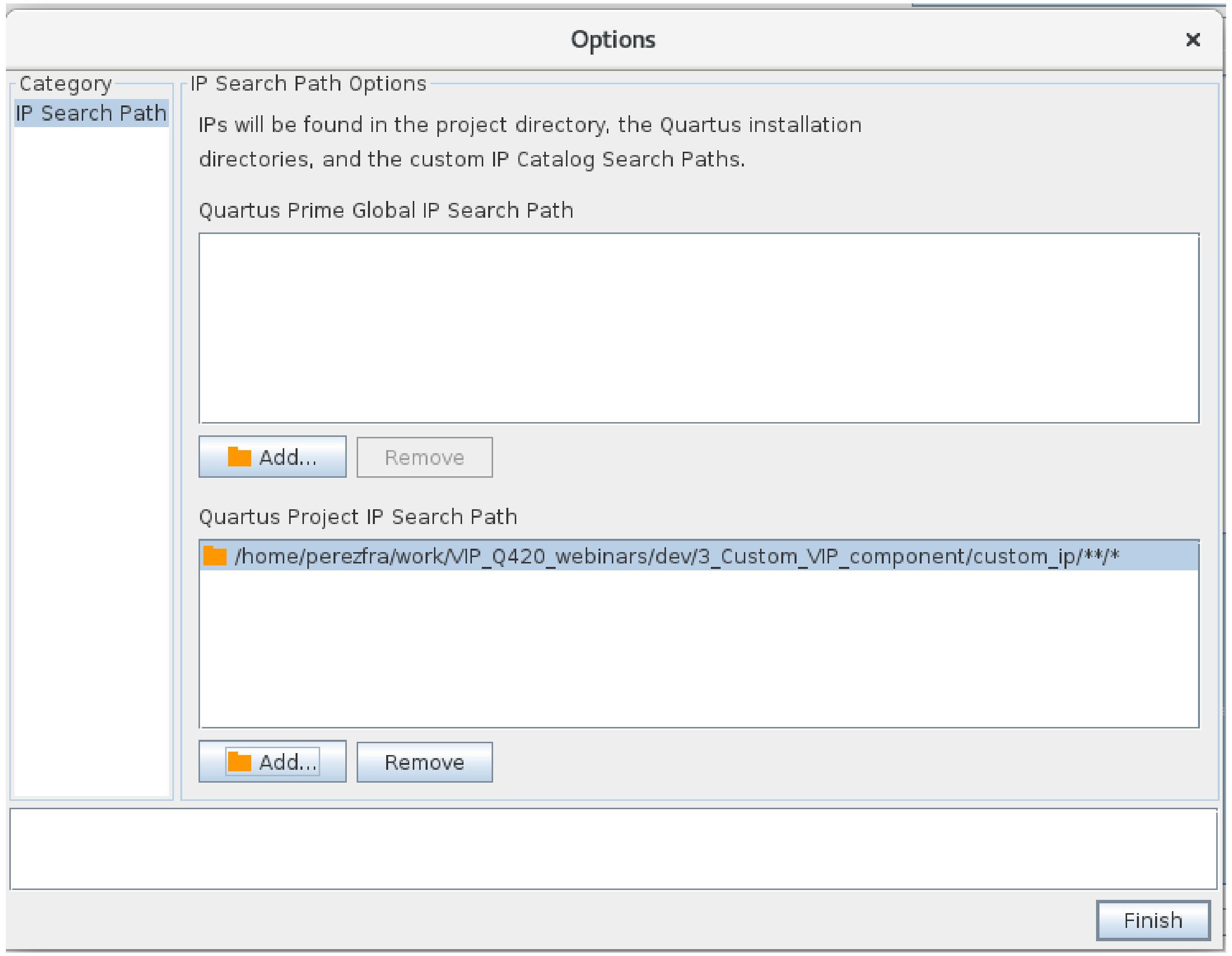
- Copy the `new_component_hw.tcl` file from `<extracted_folder>/rtl/core/` to the target `<extracted_folder>/custom_ip` folder and rename as `user_algorithm_top_hw.tcl`
- You should end up in the following



Opening Component Editor

Add the IP search path

- Now we need to allow Platform Designer to look for specific paths to find new components.
- In the main toolbar of Platform Designer select **Tools->Options...**
- In the dialog box, click on the Add button under Quartus Project IP Search Path and select `<extracted_folder>/custom_ip` to be included



IPs will be found in the project directory, the Quartus installation directories, and the custom IP Catalog Search Paths.

Quartus Prime Global IP Search Path

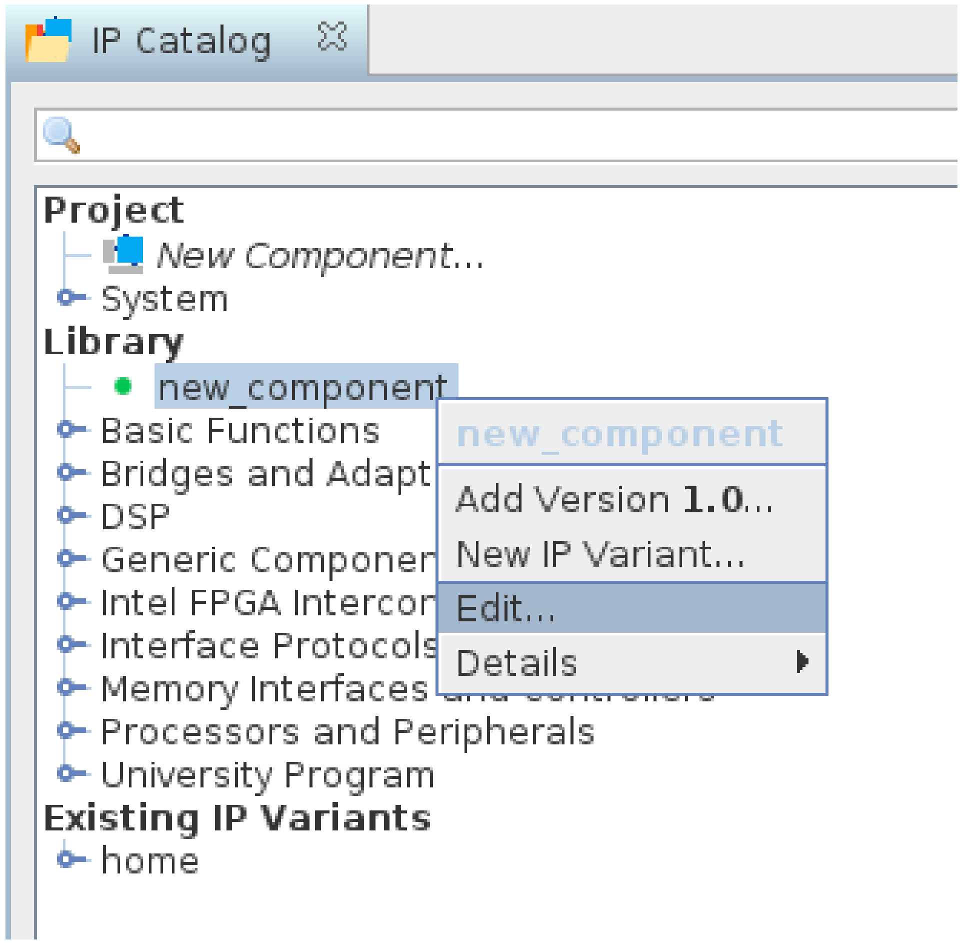
Quartus Project IP Search Path

/home/perezfra/work/VIP_Q420_webinars/dev/3_Custom_VIP_component/custom_ip/**/*

Opening Component Editor

Edit the component

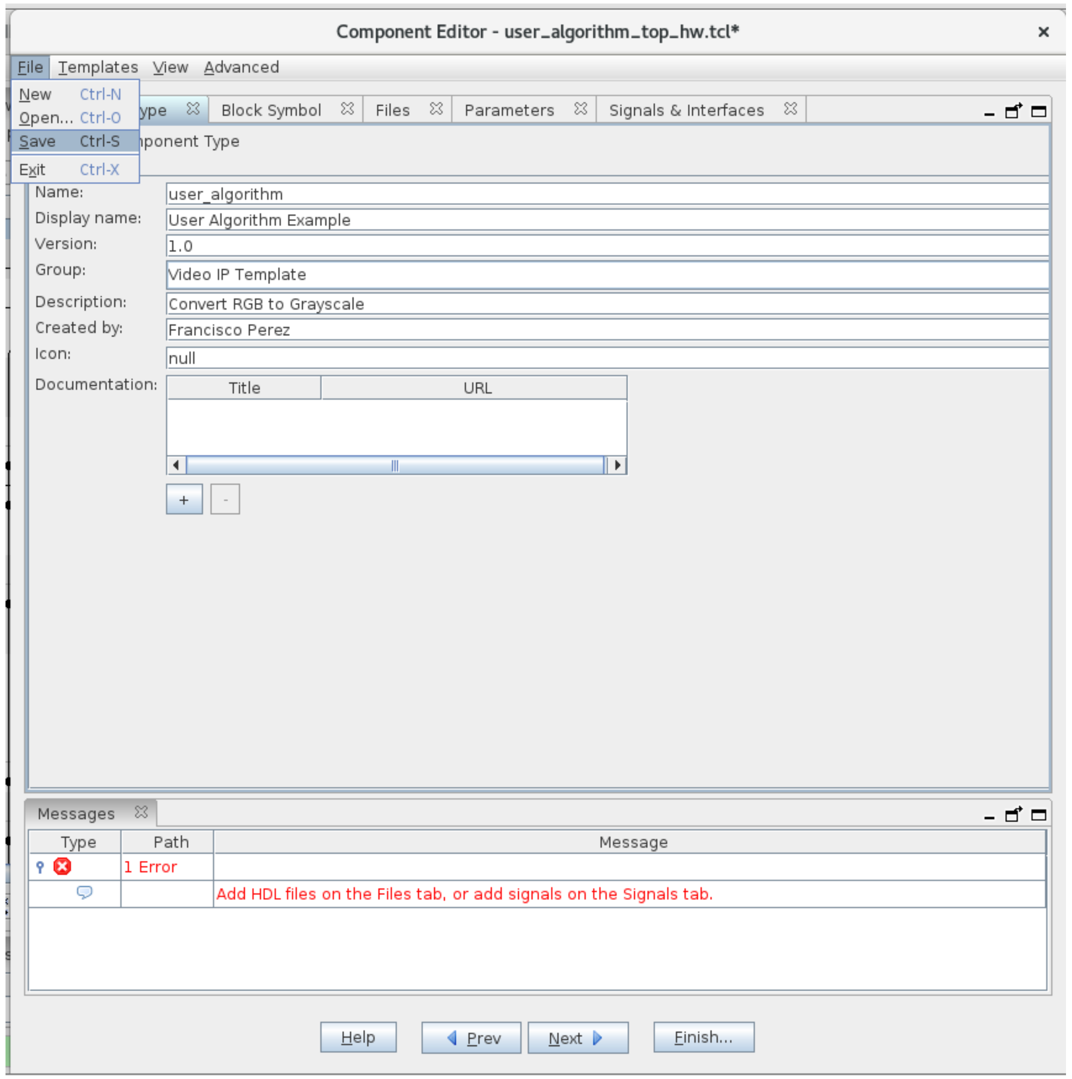
- After closing the editor, the Platform designer system gets refreshed and we can notice a new_component under the **IP Catalog Library**
- Right click on **new_component** to start editing



Opening Component Editor

Component properties

- First information to provide in the component editor are the generic properties
- Then we will supply the hdl files and configure the interfaces.



Name:	user_algorithm
Display name:	User Algorithm Example
Version:	1.0
Group:	Video IP Template
Description:	Convert RGB to Grayscale
Created by:	Francisco Perez
Icon:	null
Documentation:	Title / URL

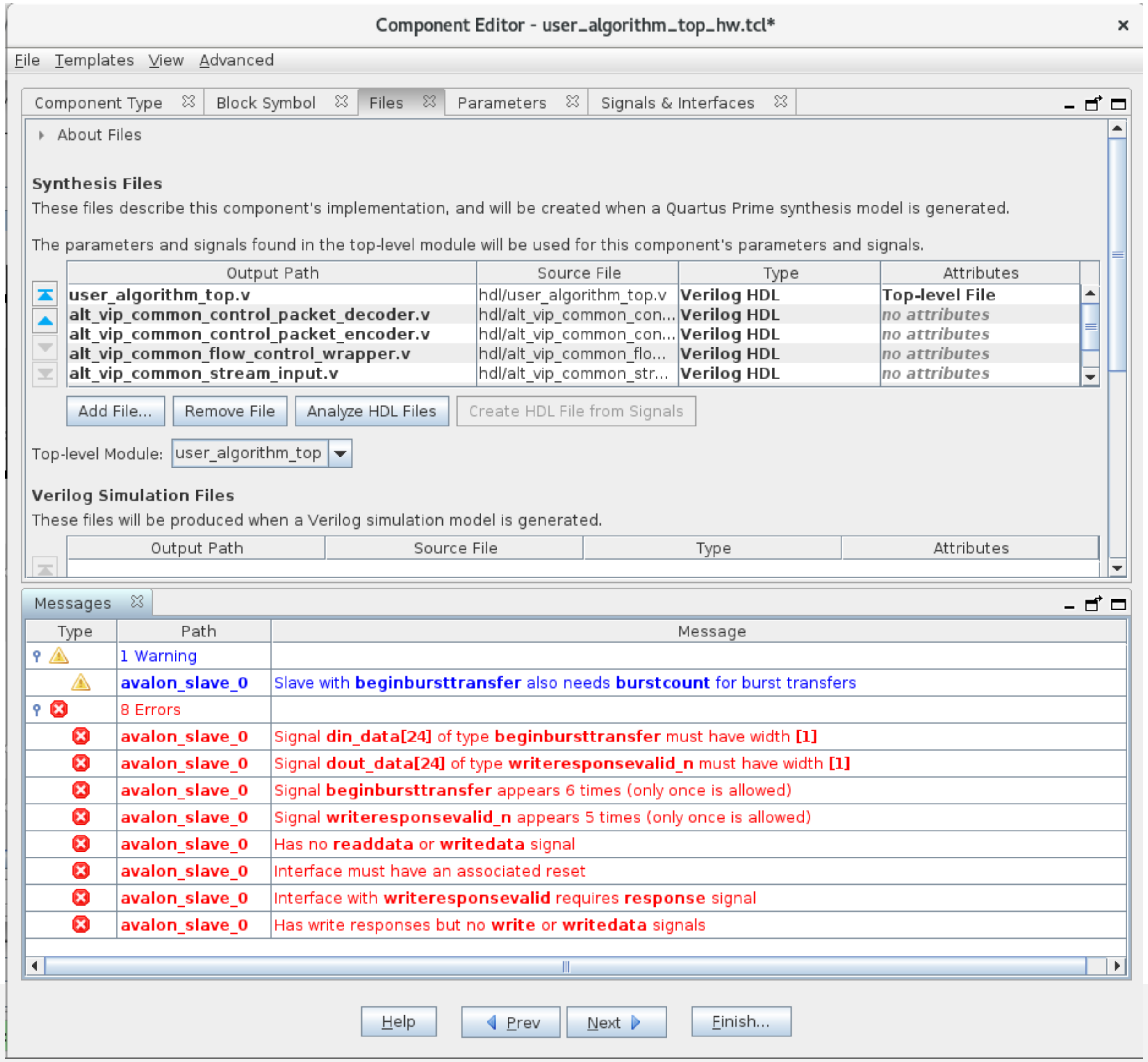
Messages

Type	Path	Message
	1 Error	
		Add HDL files on the Files tab, or add signals on the Signals tab.

Opening Component Editor

Component hdl files

- Next go to the Files tab and click on Add File… under Synthesis Files
- In the dialog box select all the Verilog .v files in the custom_ip/hdl folder
- Make sure you select the user_algorithm_top.v file as Top-level file
- Click on Analyze HDL Files and select user_algorithm_top as Top-level Module



Synthesis Files

These files describe this component's implementation, and will be created when a Quartus Prime synthesis model is generated.

The parameters and signals found in the top-level module will be used for this component's parameters and signals.

Output Path	Source File	Type	Attributes
user_algorithm_top.v	hdl/user_algorithm_top.v	Verilog HDL	Top-level File
alt_vip_common_control_packet_decoder.v	hdl/alt_vip_common_con...	Verilog HDL	no attributes
alt_vip_common_control_packet_encoder.v	hdl/alt_vip_common_con...	Verilog HDL	no attributes
alt_vip_common_flow_control_wrapper.v	hdl/alt_vip_common_flo...	Verilog HDL	no attributes
alt_vip_common_stream_input.v	hdl/alt_vip_common_str...	Verilog HDL	no attributes

Top-level Module: user_algorithm_top

Verilog Simulation Files

These files will be produced when a Verilog simulation model is generated.

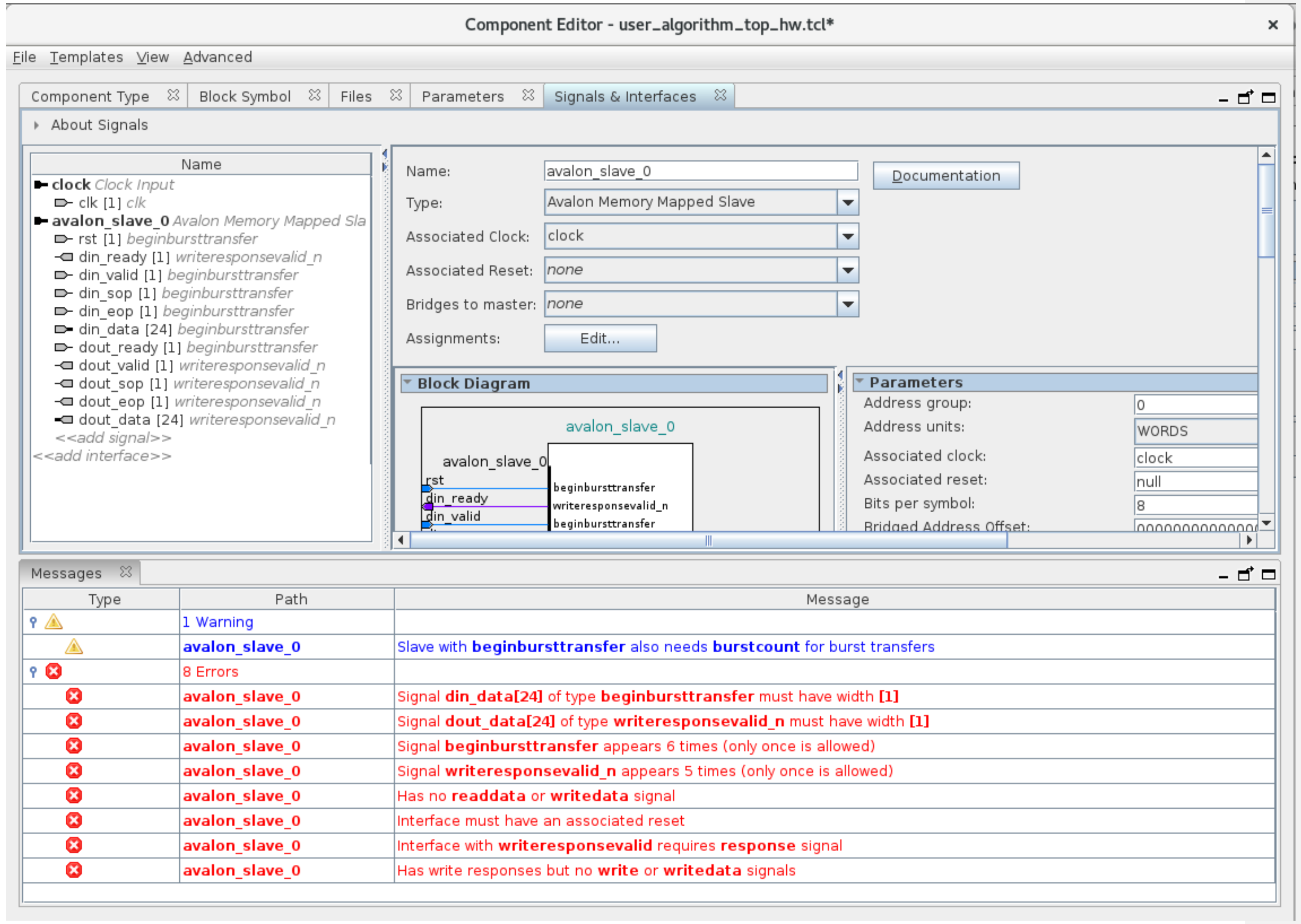
Output Path	Source File	Type	Attributes

Type	Path	Message
⚠	1 Warning	
⚠	avalon_slave_0	Slave with **beginbursttransfer** also needs **burstcount** for burst transfers
✖	8 Errors	
✖	avalon_slave_0	Signal **din_data[24]** of type **beginbursttransfer** must have width **[1]**
✖	avalon_slave_0	Signal **dout_data[24]** of type **writeresponsevalid_n** must have width **[1]**
✖	avalon_slave_0	Signal **beginbursttransfer** appears 6 times (only once is allowed)
✖	avalon_slave_0	Signal **writeresponsevalid_n** appears 5 times (only once is allowed)
✖	avalon_slave_0	Has no **readdata** or **writedata** signal
✖	avalon_slave_0	Interface must have an associated reset
✖	avalon_slave_0	Interface with **writeresponsevalid** requires **response** signal
✖	avalon_slave_0	Has write responses but no **write** or **writedata** signals

Opening Component Editor

Component signals and interfaces

- **Avalon-ST** video in and out interfaces have been wrongly assigned to an **Avalon MM** Slave interface.
- For interface type prefixes to allow automatic signal recognition, checkout the Intel Quartus Prime Edition User Guide: Platform Designer



Name
- **clock** *Clock Input*
  - clk [1] *clk*
- **avalon_slave_0** *Avalon Memory Mapped Sla*
  - rst [1] *beginbursttransfer*
  - din_ready [1] *writeresponsevalid_n*
  - din_valid [1] *beginbursttransfer*
  - din_sop [1] *beginbursttransfer*
  - din_eop [1] *beginbursttransfer*
  - din_data [24] *beginbursttransfer*
  - dout_ready [1] *beginbursttransfer*
  - dout_valid [1] *writeresponsevalid_n*
  - dout_sop [1] *writeresponsevalid_n*
  - dout_eop [1] *writeresponsevalid_n*
  - dout_data [24] *writeresponsevalid_n*
  - *<<add signal>>*
- *<<add interface>>*

Name:	avalon_slave_0
Type:	Avalon Memory Mapped Slave
Associated Clock:	clock
Associated Reset:	*none*
Bridges to master:	*none*
Assignments:	Edit...

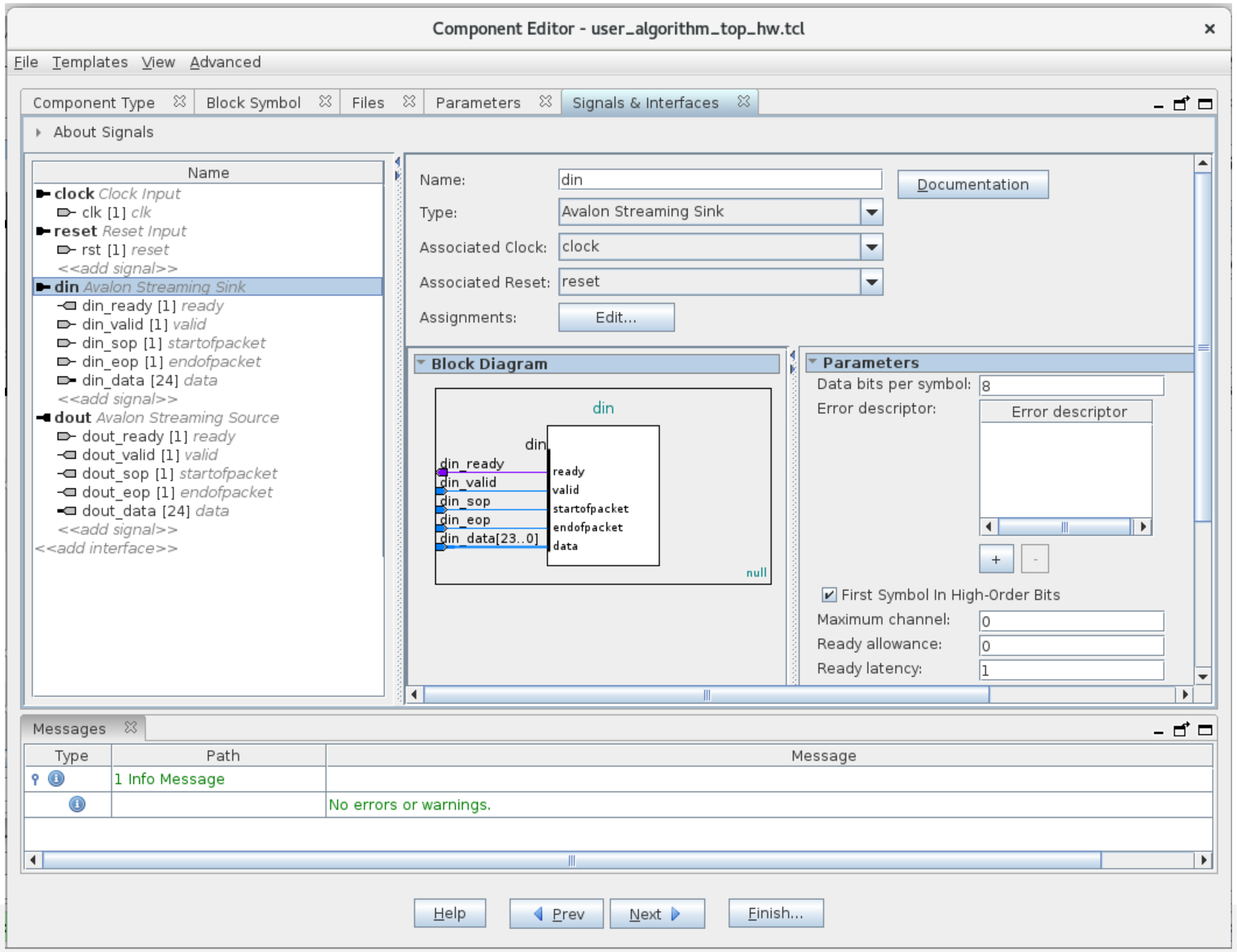
Address group:	0
Address units:	WORDS
Associated clock:	clock
Associated reset:	null
Bits per symbol:	8
Bridged Address Offset:	000000000000

Type	Path	Message
⚠	1 Warning	
⚠	**avalon_slave_0**	Slave with **beginbursttransfer** also needs **burstcount** for burst transfers
✖	8 Errors	
✖	**avalon_slave_0**	Signal **din_data[24]** of type **beginbursttransfer** must have width **[1]**
✖	**avalon_slave_0**	Signal **dout_data[24]** of type **writeresponsevalid_n** must have width **[1]**
✖	**avalon_slave_0**	Signal **beginbursttransfer** appears 6 times (only once is allowed)
✖	**avalon_slave_0**	Signal **writeresponsevalid_n** appears 5 times (only once is allowed)
✖	**avalon_slave_0**	Has no **readdata** or **writedata** signal
✖	**avalon_slave_0**	Interface must have an associated reset
✖	**avalon_slave_0**	Interface with **writeresponsevalid** requires **response** signal
✖	**avalon_slave_0**	Has write responses but no **write** or **writedata** signals

Opening Component Editor

Component signals and interfaces

- Add the required interfaces and group the signals in the name pane.
- Associate the clock and reset interfaces to each Avalon-ST sink and source



Component Editor - user_algorithm_top_hw.tcl

Name
- **clock** *Clock Input*
  - clk [1] *clk*
- **reset** *Reset Input*
  - rst [1] *reset*
  - *<<add signal>>*
- **din** *Avalon Streaming Sink*
  - din_ready [1] *ready*
  - din_valid [1] *valid*
  - din_sop [1] *startofpacket*
  - din_eop [1] *endofpacket*
  - din_data [24] *data*
  - *<<add signal>>*
- **dout** *Avalon Streaming Source*
  - dout_ready [1] *ready*
  - dout_valid [1] *valid*
  - dout_sop [1] *startofpacket*
  - dout_eop [1] *endofpacket*
  - dout_data [24] *data*
  - *<<add signal>>*
- *<<add interface>>*

Name: din

Type: Avalon Streaming Sink

Associated Clock: clock

Associated Reset: reset

Assignments: Edit...

Parameters

Data bits per symbol: 8

Error descriptor: Error descriptor

[✓] First Symbol In High-Order Bits

Maximum channel: 0

Ready allowance: 0

Ready latency: 1

Messages

Type	Path	Message
	1 Info Message	
		No errors or warnings.

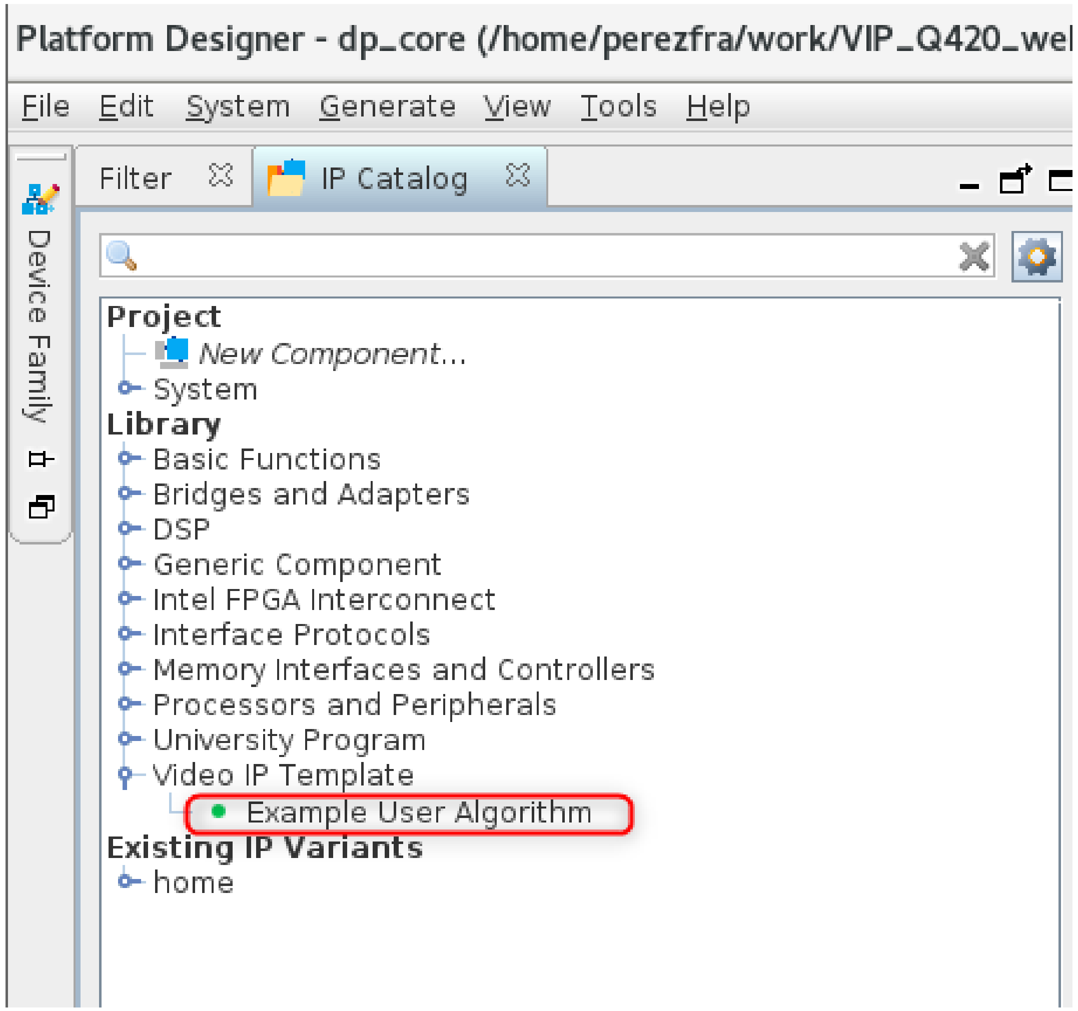
Session 3 – Create custom VIP modules

Adding the custom component to the pipeline

Using the component

Adding it to the pipeline

- Find a new category under the **IP Catalog->Library->Video IP Template**, where we have available our **User Algorithm Example** component ready to use.
- Click on Add to open the parameters dialog box



Platform Designer - dp_core (/home/perezfra/work/VIP_Q420_wel

Project
- *New Component...*
- System

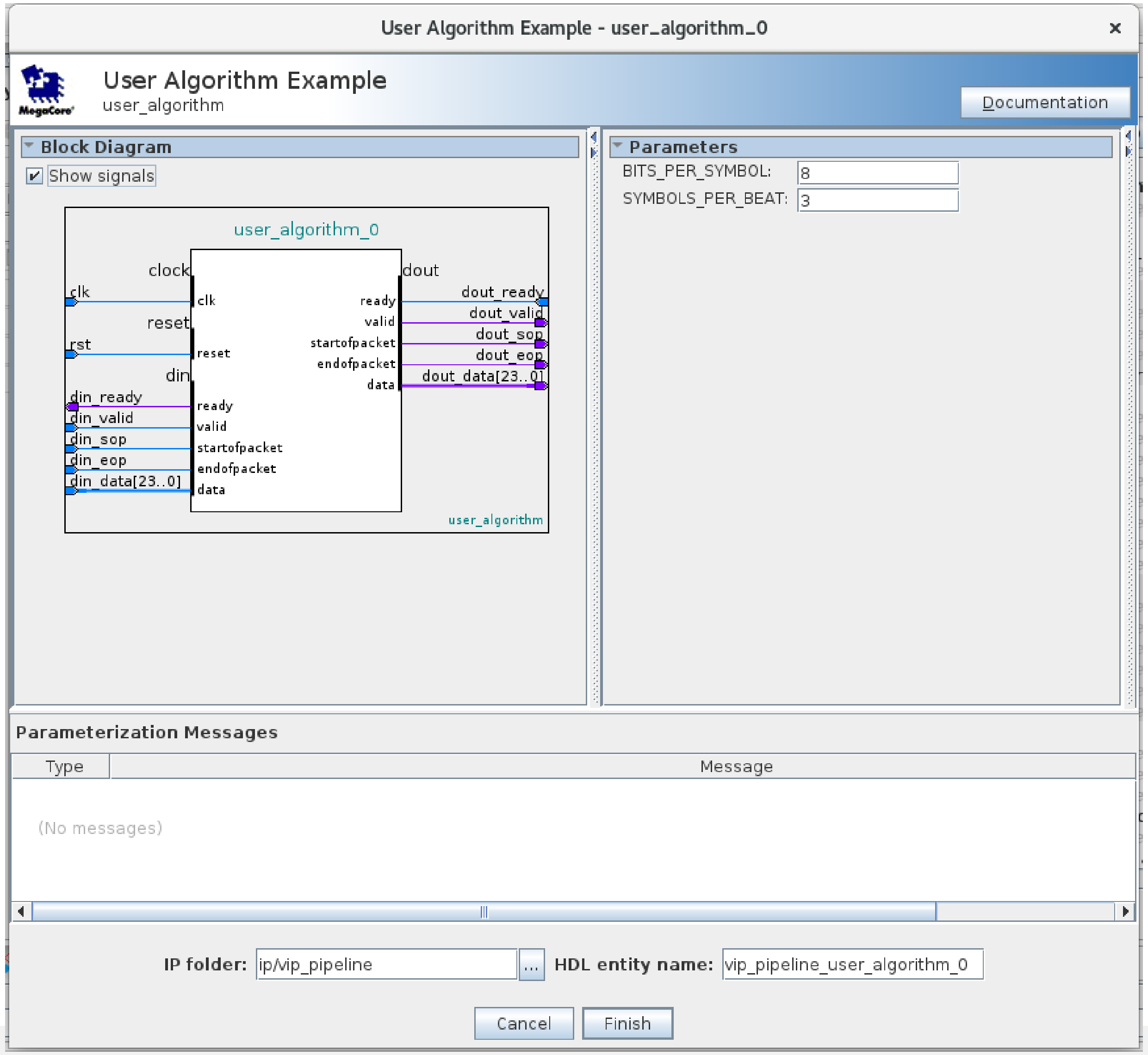
Library
- Basic Functions
- Bridges and Adapters
- DSP
- Generic Component
- Intel FPGA Interconnect
- Interface Protocols
- Memory Interfaces and Controllers
- Processors and Peripherals
- University Program
- Video IP Template
  - Example User Algorithm

Existing IP Variants
- home

Using the component

Adding it to the pipeline

- Make sure you select `8 bits per symbols and 3 symbols per beat`, as this is the color depth and planes (RGB) we are using in our `vip_pipeline` subsystem.
- Notice that our custom component has an input Avalon-ST video interface to input data from the upstream module and another one to deliver downstream.



BITS_PER_SYMBOL: 8
SYMBOLS_PER_BEAT: 3

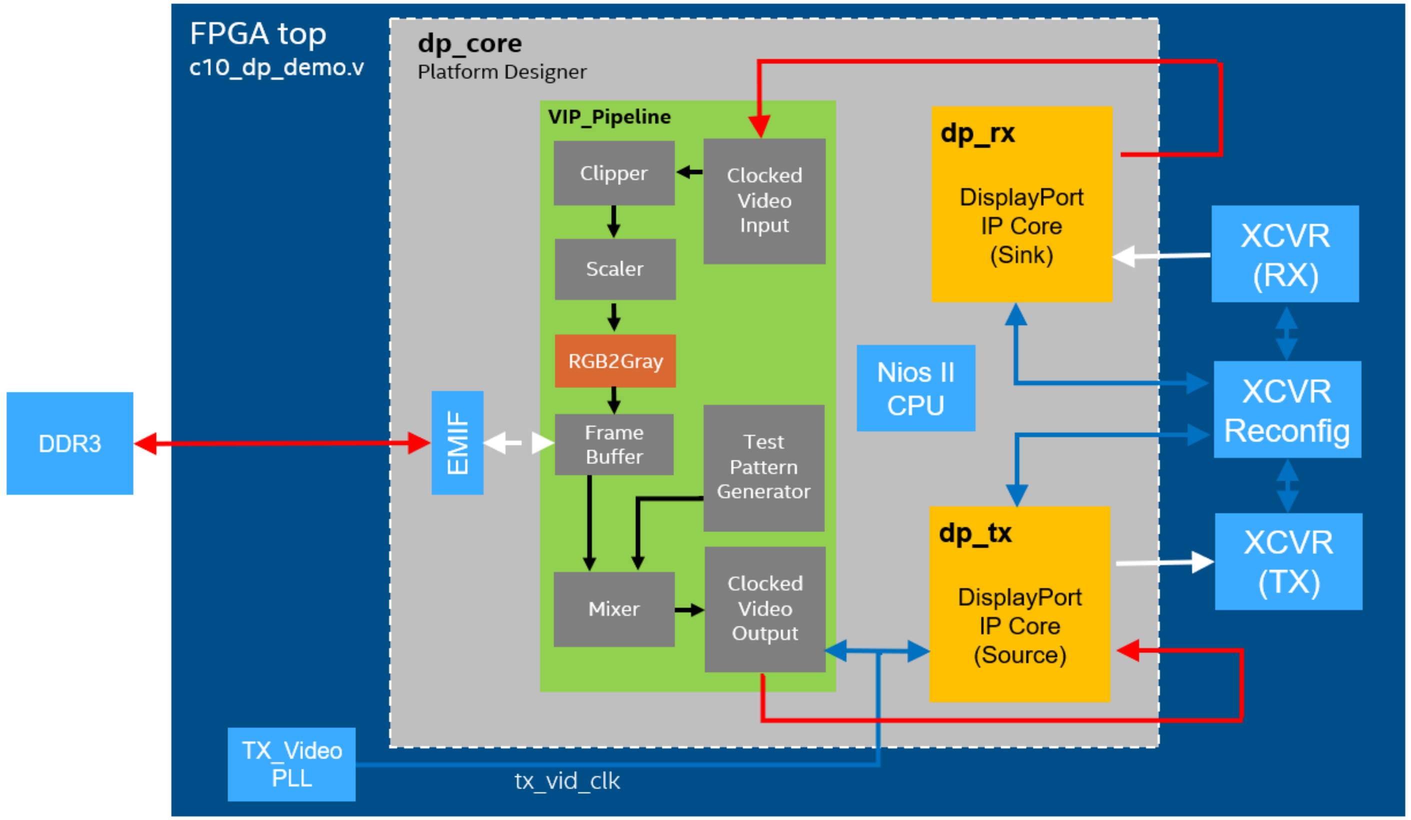
Parameterization Messages

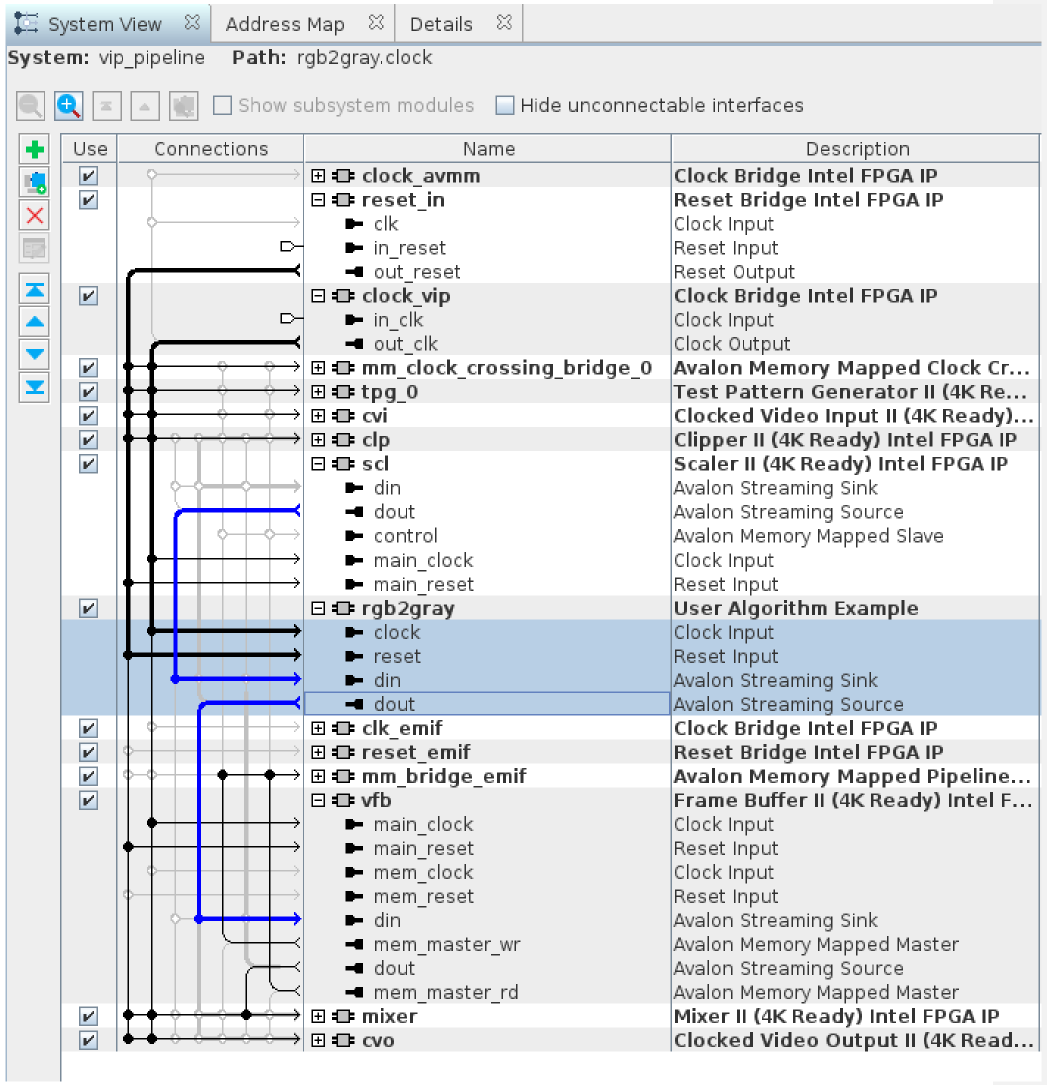
Type	Message
(No messages)	

IP folder: ip/vip_pipeline   HDL entity name: vip_pipeline_user_algorithm_0

Using the component

Adding it to the pipeline





System View | Address Map | Details

System: vip_pipeline **Path:** rgb2gray.clock

☐ Show subsystem modules ☐ Hide unconnectable interfaces

Use	Connections	Name	Description
✓		clock_avmm	Clock Bridge Intel FPGA IP
✓		reset_in	Reset Bridge Intel FPGA IP
		clk	Clock Input
		in_reset	Reset Input
		out_reset	Reset Output
✓		clock_vip	Clock Bridge Intel FPGA IP
		in_clk	Clock Input
		out_clk	Clock Output
✓		mm_clock_crossing_bridge_0	Avalon Memory Mapped Clock Cr...
✓		tpg_0	Test Pattern Generator II (4K Re...
✓		cvi	Clocked Video Input II (4K Ready)...
✓		clp	Clipper II (4K Ready) Intel FPGA IP
✓		scl	Scaler II (4K Ready) Intel FPGA IP
		din	Avalon Streaming Sink
		dout	Avalon Streaming Source
		control	Avalon Memory Mapped Slave
		main_clock	Clock Input
		main_reset	Reset Input
✓		rgb2gray	User Algorithm Example
		clock	Clock Input
		reset	Reset Input
		din	Avalon Streaming Sink
		dout	Avalon Streaming Source
✓		clk_emif	Clock Bridge Intel FPGA IP
✓		reset_emif	Reset Bridge Intel FPGA IP
✓		mm_bridge_emif	Avalon Memory Mapped Pipeline...
✓		vfb	Frame Buffer II (4K Ready) Intel F...
		main_clock	Clock Input
		main_reset	Reset Input
		mem_clock	Clock Input
		mem_reset	Reset Input
		din	Avalon Streaming Sink
		mem_master_wr	Avalon Memory Mapped Master
		dout	Avalon Streaming Source
		mem_master_rd	Avalon Memory Mapped Master
✓		mixer	Mixer II (4K Ready) Intel FPGA IP
✓		cvo	Clocked Video Output II (4K Read...

Using the component

Adding it to the pipeline

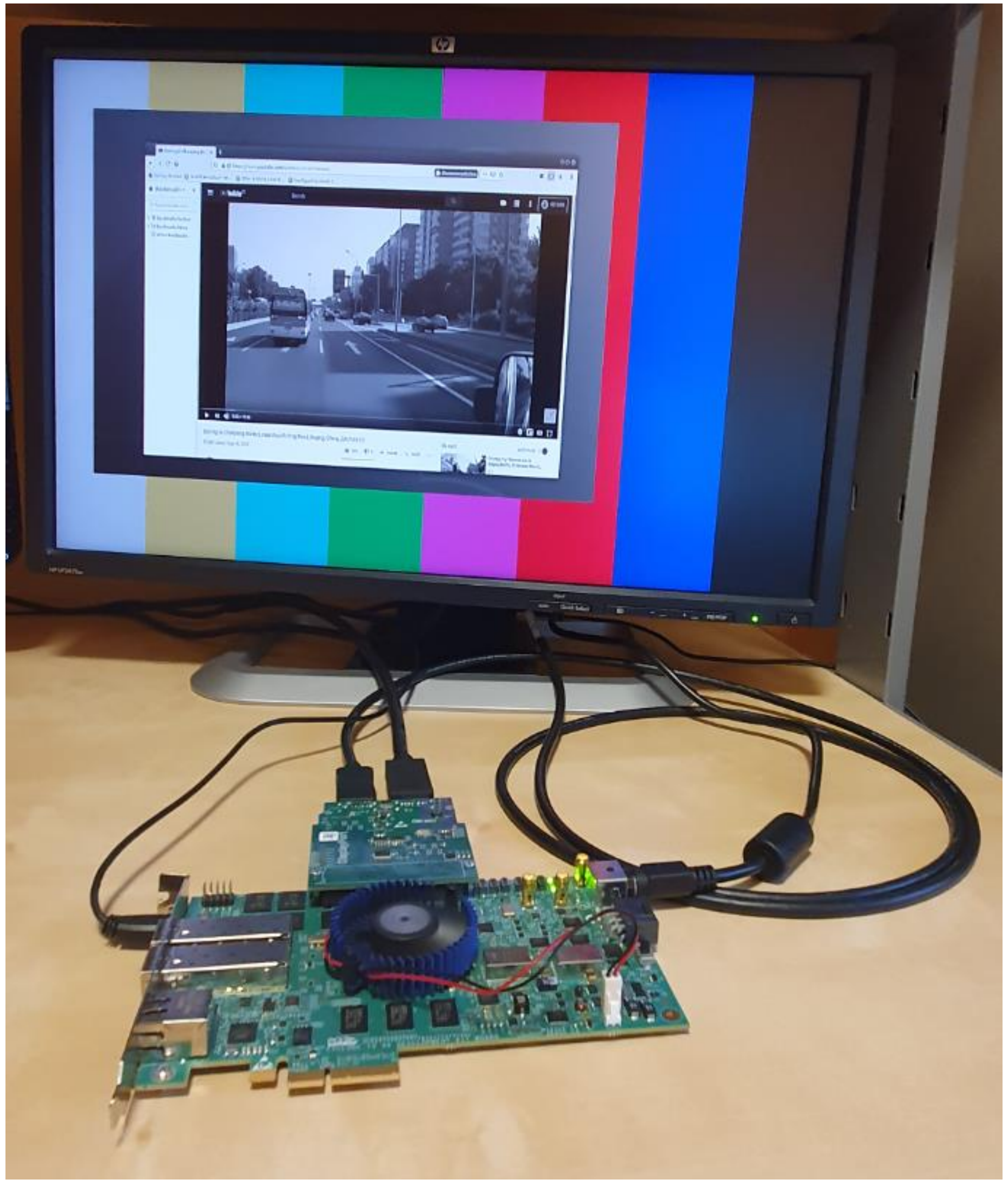
- Save `vip_pipeline.qsys` subsystem and open `dp_core.qsys` higher level
- Generate the Platform Designed `dp_core.qsys` System
- In quartus compile the design to generate the bitstream
- As we did in previous sessions, generate the software application for Nios II processor using the supplied source code files.
- For a comprehensive step guide checkout the `Develop_Custom_VIP_Component_lab.pdf manual`

Session 3 – Create custom VIP modules

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 build our own proprietary component to implement RGB to grayscale conversion
- We used the provided hdl template which is facilitating the task to develop a fully VIP compliant component to be integrated in our pipeline
- The template handles all the video and control packets, as well as backpressure, allowing oyu to focus on your own secret sauce.