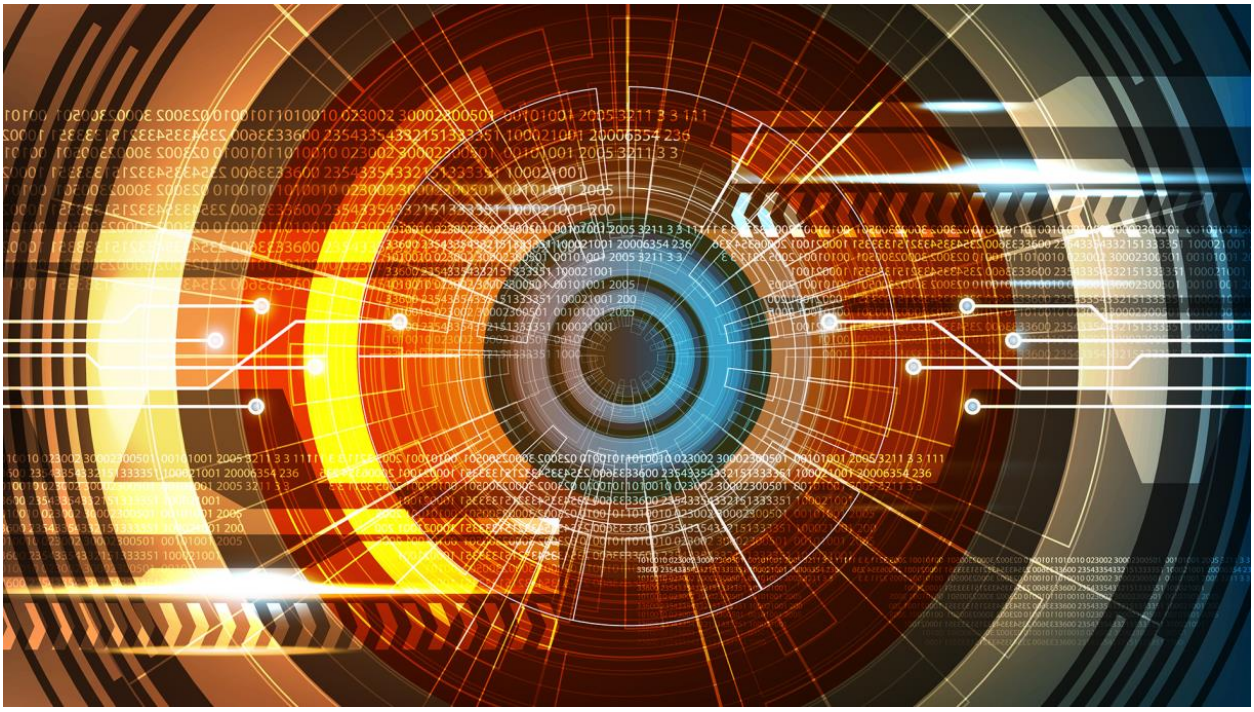


VIDEO PROCESSING WITH INTEL FPGAS WEBINAR SERIES – Q4'2020

Session 3 – Custom VIP Component

Francisco Perez
Intel FPGA Field Applications Engineer
v.1 – November 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, etc… But, how can you add your "secret sauce" to the application? What if you need to add proprietary processing into the pipeline?

In this lab manual we are developing a custom component to be integrated in a video processing pipeline built using VIP cores.

This developed core will be fully compliant with Avalon-ST Video protocol and will manage control and video packets, as well as handles backpressure signals generated while processing live video.

In this session, we are following the steps to develop a custom component in Platform Designer and integrate it into the VIP pipeline to test. This module is an RGB to Grayscale converter.

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

Download and extract the archived project "**3_Develop_Custom_Component_v1.tar.gz**" located in the Github repository:

https://github.com/perezfra/VIP_webinars_Intel_FPGA

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:

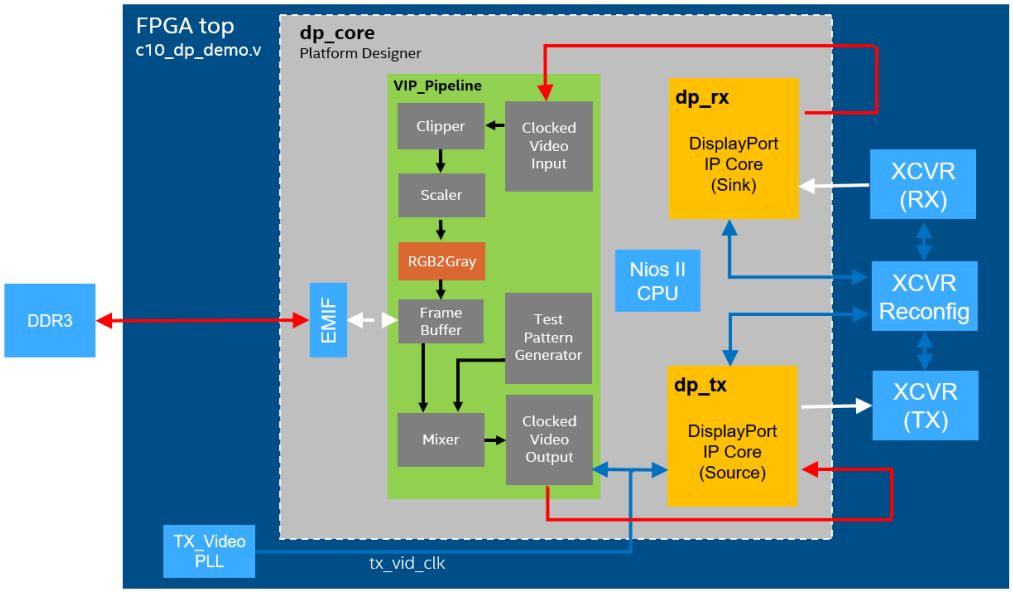
- Intel FPGA DisplayPort IP User Guide
  https://www.intel.com/content/www/us/en/programmable/products/intellectual-property/ip/interface-protocols/m-alt-displayport-megacore.html
- Cyclone 10 GX DisplayPort Design Example User Guide
  https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-dex-dp-c10gx.pdf

- 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), until get it finally mixed with a test pattern background before sent to the **Clocked_Video_Output** component connected to the DisplayPort TX interface.



In this design we 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.

Original Image

Grayscale

To perform this transformation we use the Weighted or Luminosity method, according to the following formula:

```
New grayscale pixel = ((0.3 * R) + (0.59 * G) + (0.11 * B))
```

2. Building the RGB2Gray component

2.1.Setting up the Quartus project

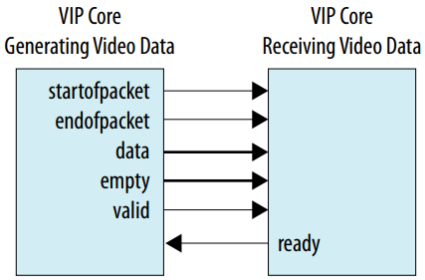
Extract the files provided in the **3_Develop_Custom_Component_v1.tar.gz** package and open the Quartus project located at `<extracted_folder>/quartus/c10_dp_demo.qpf`

- **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 Verilog provided files to build our component compliant with Avalon-ST Video protocol
- **custom_ip_sol**: contains completely develop component ready to use or for reference while building yours

2.2.Avalon-ST Video protocol review

In the previous session (**2_Debug_VIP_pipeline**) we covered, in details, the Avalon-ST Video protocol: which is the transport method used to get information flowing through the VIP Suite cores.

This streaming protocol supports different types of packets: video, control and user packets and sits on top of the Intel Avalon Streaming standard.



This is how 2 VIP cores are connected together. They use typical packet control signals like start and end of packet. The flow control is handled by the ready and valid signals: the downstream core can apply backpressure to the pipeline, by de-asserting ready, when cannot process more data. This stalls the pipeline and we need to manage it properly (with adequate intermediate buffering) to avoid overflow conditions and loss of video pixel data.

Avalon-ST video can be configured to support many different resolutions and pixel formats.

The Video and Image Processing Suite IP cores do not continuously process data. Instead, they use flow-controlled Avalon-ST interfaces, which allow them to stall the pipeline while they perform internal calculations.

During data processing, the IP cores generally process one input or output symbol per clock cycle. There are, however, some stalling cycles. Typically, these are for internal calculations between rows of image data and between frames/fields. When stalled, an IP core indicates that it is not ready to receive or produce data. The time spent in the stalled state varies between IP cores and their parameterizations. In general, it is a few cycles between rows and a few more between frames.

If data is not available at the input when required, the IP cores stall and do not output data.

Avalon streaming video protocol supports different types of packets. Most important and used, even mandatory, are the Control packet and the video packet.

In the table shown we see the different packet type identifiers. This information is included in the lowest nibble of the first symbol transmitted, coincident with the assertion of the start of packet signal.

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

Most VIP compliant cores require a control packet to be received before any video packet. This is used to configure the line buffers, counters, and other internal logic according to the frame will be next processed.

When a VIP core receives a control packet, decodes the height, width and interlacing format to correctly interprets the following video packets, until it receives another control packet with different parameters. VIP cores send always a control packet before a video packet.

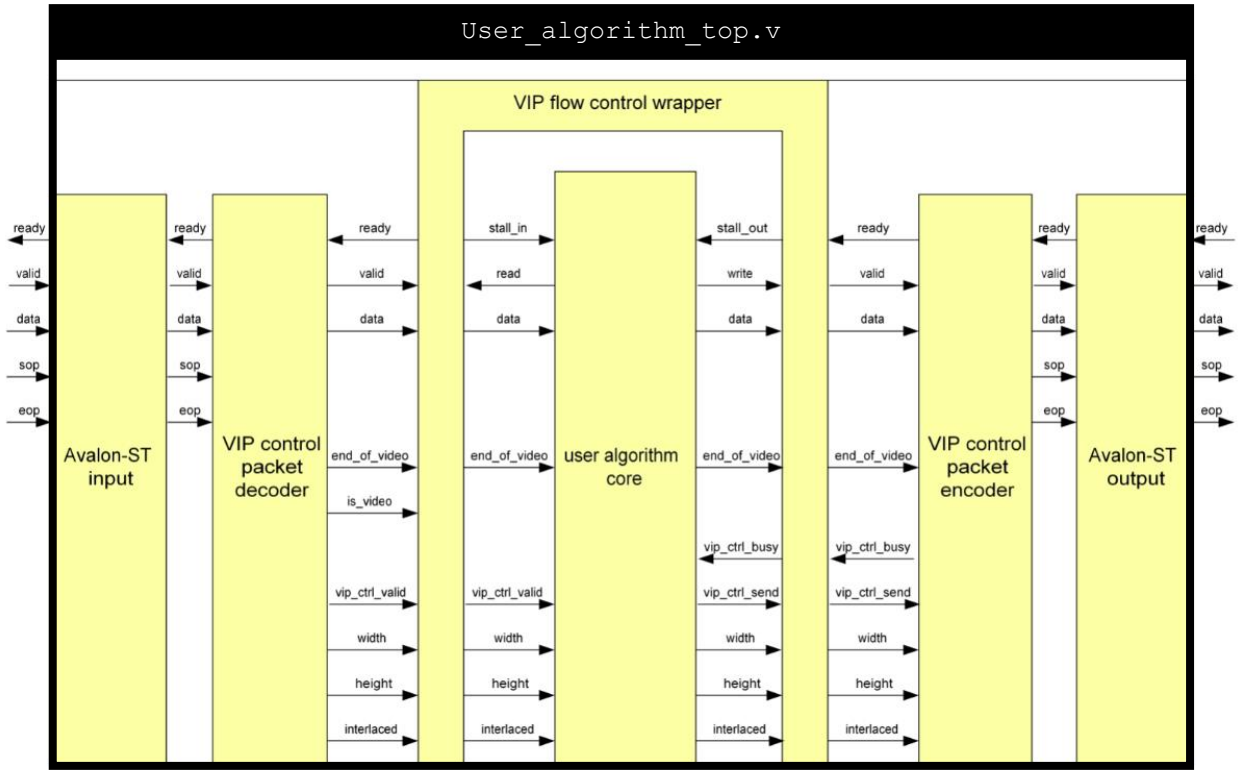
For a more detailed view on Avalon-ST Video protocol check the Video and Image Processing suite user guide: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_vip.pdf

2.3.Avalon-ST video template files

To facilitate the task of the developer to build its component, we provide a collection of files to decode/generate the control and video packets and handle the backpressure mechanism to stall data flow in the pipeline. This allows the user to fully focus on the development of the proprietary algorithm.

Let's have a look at the files included in the template. Go to the `<extracted_folder>/custom_ip/hdl` to examine the Verilog modules that are part of the template.

The top-level user algorithm ("`user_algorithm_top.v`") that is used to generate the component in Platform Designer consists of six main blocks, organized as per the following diagram:



1. The **algorithm core** that can be replaced by the HDL template user ("`user_algorithm_core.v`")
2. A **VIP flow control wrapper** for the user algorithm ("`alt_vip_common_flow_control_wrapper.v`")
3. The **VIP Control Packet Decoder** ("`alt_vip_common_control_packet_decoder.v`")
4. The **VIP Control Packet Encoder** ("`alt_vip_common_control_packet_encoder.v`")
5. An **Avalon-ST input** block ("`alt_vip_common_avalon_stream_input.v`")
6. An **Avalon-ST output** block ("`alt_vip_common_avalon_stream_output.v`")

In addition to the HDL source code, the `custom_ip_sol` folder contains the file "`user_algorithm_top_hw.tcl`", which was generated by the Component Editor of Platform Designer, when we created the component and we provide as part of the solution.

This `.tcl` file makes sure that the user algorithm module ("`user_algorithm`") within the design can be found when opening Platform Designer. Therefore, the `custom_ip_sol` folder may need to be included in the Platform Designer IP search path.

These HDL template module files have the following features:

- VIP Control Packet Decoder:

This block decodes the VIP control packets from the data stream and sends the decoded data (width, height, interlaced) as separate signals to the algorithm.

All Avalon-ST signals are simply passed through without any latency.

The signal '`vip_ctrl_valid`' goes high for one cycle as soon as the control data is valid for the next video frame.

The signal '`is_video`' is high when active video data are output to the algorithm. This signal is used in the flow control wrapper and not visible to the user algorithm.

The signal '`end_of_video`' is high when the last pixel of the active video data is output to the algorithm.

- VIP Control Packet Encoder:

This block encodes the VIP control packets based on the received width, height and interlaced information and inserts them into the data stream.

When '`vip_ctrl_send`' goes high, a VIP control packet is inserted.

However, '`vip_ctrl_send`' can only be asserted if '`vip_ctrl_busy`' is low. This is to make sure that control packets are not inserted inside video data packets.

Only active video data and an '`end_of_video`' signal are input into the encoder via the Avalon-ST interface.

The output stream always contains one VIP control packet, followed by one video data packet.

The signal '`vip_ctrl_busy`' gets set to low only when the last active pixel has been received by the encoder.

There is no latency between respective Avalon-ST signals at input and output of the encoder.

- VIP Flow Control Wrapper:

This block wraps around the user algorithm and deals with the flow control conversion from Avalon-ST to/from a simple read/write interface.

At the same time, the VIP control packets are removed from the data stream as they have already been decoded, and only the active video data is sent to the algorithm.

Other control signals are simply passed through.

- Avalon-ST Input:

This is an Avalon-ST sink interface with READY_LATENCY = 1, which registers the Avalon-ST signals and provides backpressure support at the input.

- Avalon-ST Output:

This is an Avalon-ST source interface with READY_LATENCY = 1, which registers the Avalon-ST signals and provides backpressure support at the output.

- User algorithm core:

This block can be replaced by the HDL template user to include their own algorithm.

To request input data, the `'read'` signal is asserted, and incoming data is valid when `'read'` is high and `'stall_in'` is low.

To send valid output data, the `'write'` signal is asserted, and the data needs to be held while `'stall_out'` is high.

Only when `'stall_out'` is low while `'write'` is high, the data is captured.

2.4. User algorithm core

NOTE: For the rest of the exercise we will work with the `custom_ip` folder, if you want to skip the flow, and use directly the finished component you can just directly use `custom_ip_sol` instead.

The place where we should define our custom processing is within the `user_algorithm_core.v` file. Inside the file there is a placeholder dedicated for that purpose.

The `user_algorithm_core.v` file is divided into 2 main blocks:

- Data processing of user algorithm.
  - This is our target block and where we will add our stuff
- 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.

This is the port mapping of the block showing which signals are used to connect to the VIP control packet decoder and VIP control packet encoder via the VIP flow control wrapper.

```
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);
```

As part of the module architecture, there is a labelled section destinated to include the user algorithm logic. This is where we will be adding the logic to implement our proprietary function.

```
// internal flow controlled signals
wire [BITS_PER_SYMBOL * SYMBOLS_PER_BEAT : 0] data_int;
wire input_valid;
reg data_available;
reg [BITS_PER_SYMBOL * SYMBOLS_PER_BEAT : 0] data_int_reg;
reg [BITS_PER_SYMBOL * SYMBOLS_PER_BEAT : 0] data_out_reg;


/*****************************************************************************/
/* Data processing of user algorithm starts here                             */
/*****************************************************************************/

/*****************************************************************************/
/* End of user algorithm data processing                                     */
/*****************************************************************************/

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

In the flow control processing section, we see the generation of the different flow control signals used to pass video data through the module and along the pipeline under Avalon-ST video specs.

```
/*****************************************************************************/
/* 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                                            */
/*****************************************************************************/
```

So, let's now add the logic needed to convert RGB pixels into grayscale representation according to the Weighted or Luminosity method:

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

In `user_algorithm_core.v` file, we go to the Data processing of user algorithm section to add the logic that implements the conversion.

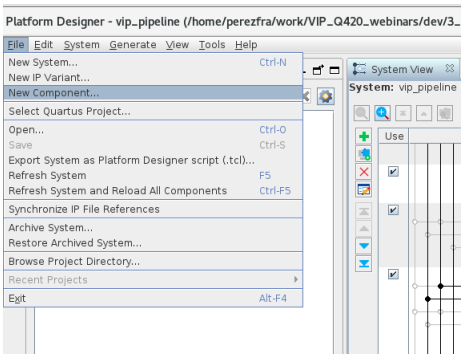
```
/*****************************************************************************/
/* Data processing of user algorithm starts here                             */
/*****************************************************************************/
/*********************************************/
/* this example: 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                                     */
/*****************************************************************************/
```

2.5. Using the component editor

Now that we have modified our `user_algorithm_core.v` file we can start the generation of our module using the component editor available in Platform Designer.

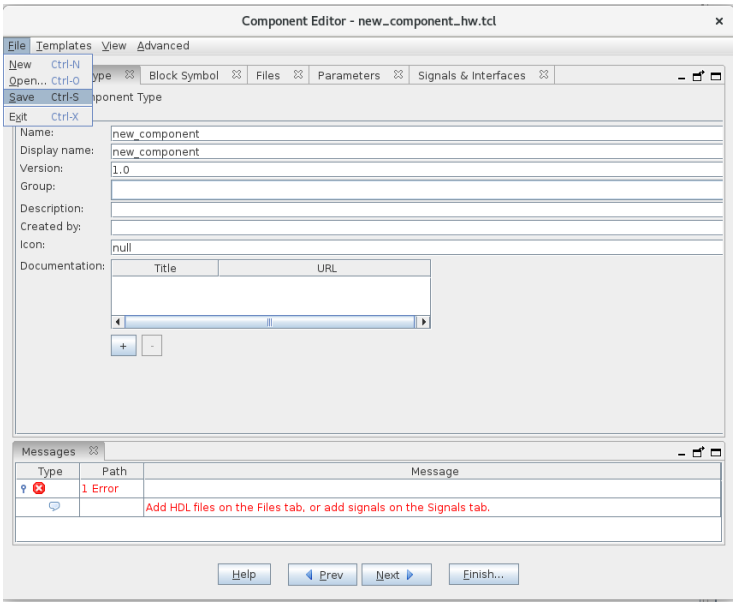
NOTE: in this session we are assuming that you have simulated and validated the operation of your custom algorithm and it's functionally correct. How to simulate and do functional validation of your component is out of the scope.

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



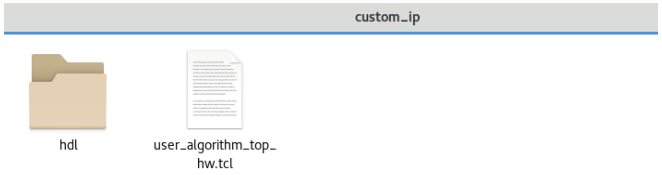
4. Component Editor allows the creation of new components only in the folder where the *.qsys file has been opened. In this case, we have opened the `<extracted_folder>/rtl/core/vip_pipeline.qsys` file, hence this is the root directory.

   Just click on **File->Save** to get the `new_component_hw.tcl` file generated and close the editor

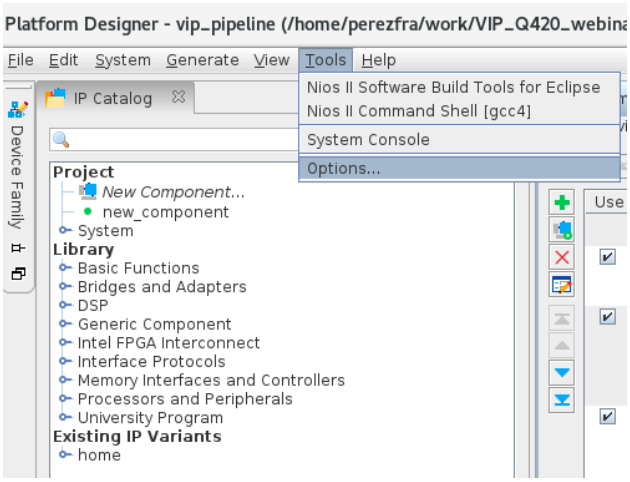


5. Copy the `new_component_hw.tcl` file from `<extracted_folder>/rtl/core/` to the target `<extracted_folder>/custom_ip` folder and rename as `user_algorithm_top_hw.tcl`

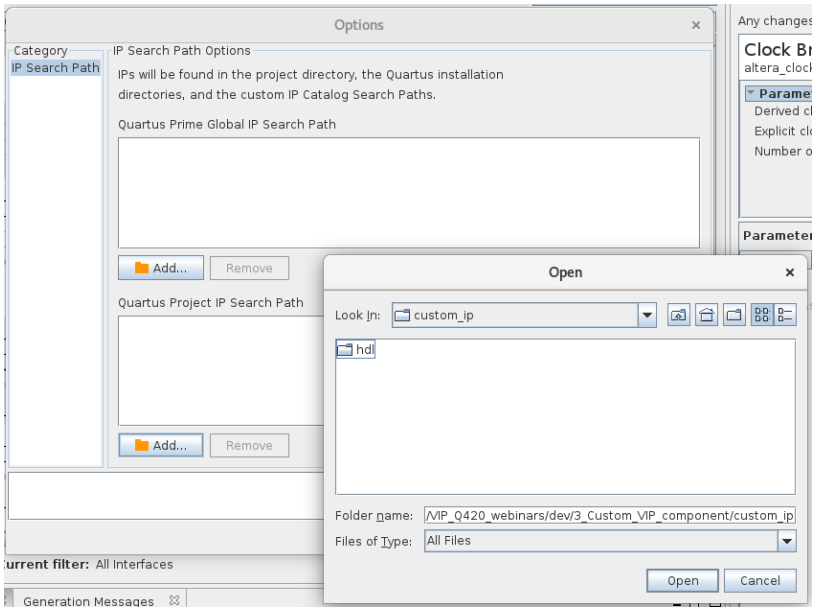
   You should end up in the following



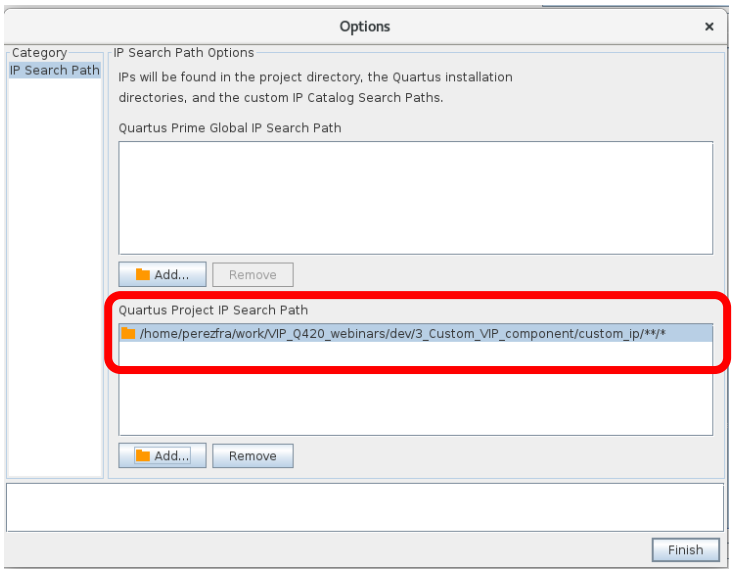
6. Now we need to allow Platform Designer to look for specific paths to find new components. In the main toolbar of Platform Designer select **Tools->Options**…



   In the dialog box, click on the **Add** button under *Quartus Project IP Search Path* and select `<extracted_folder>/custom_ip` to be included



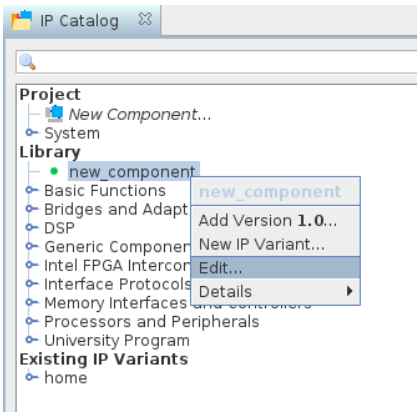
Once done, you should see the folder added to the *IP Search Path*



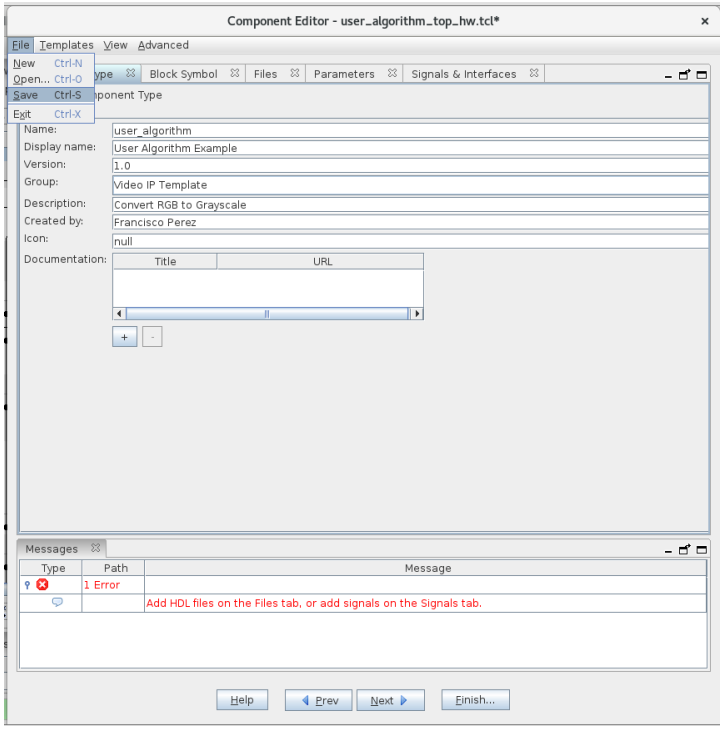
Click **Finish** to close the dialog

7. After closing the editor, the Platform designer system gets refreshed and we can notice a `new_component` under the IP Catalog Library

8. Right click on `new_component` to start editing

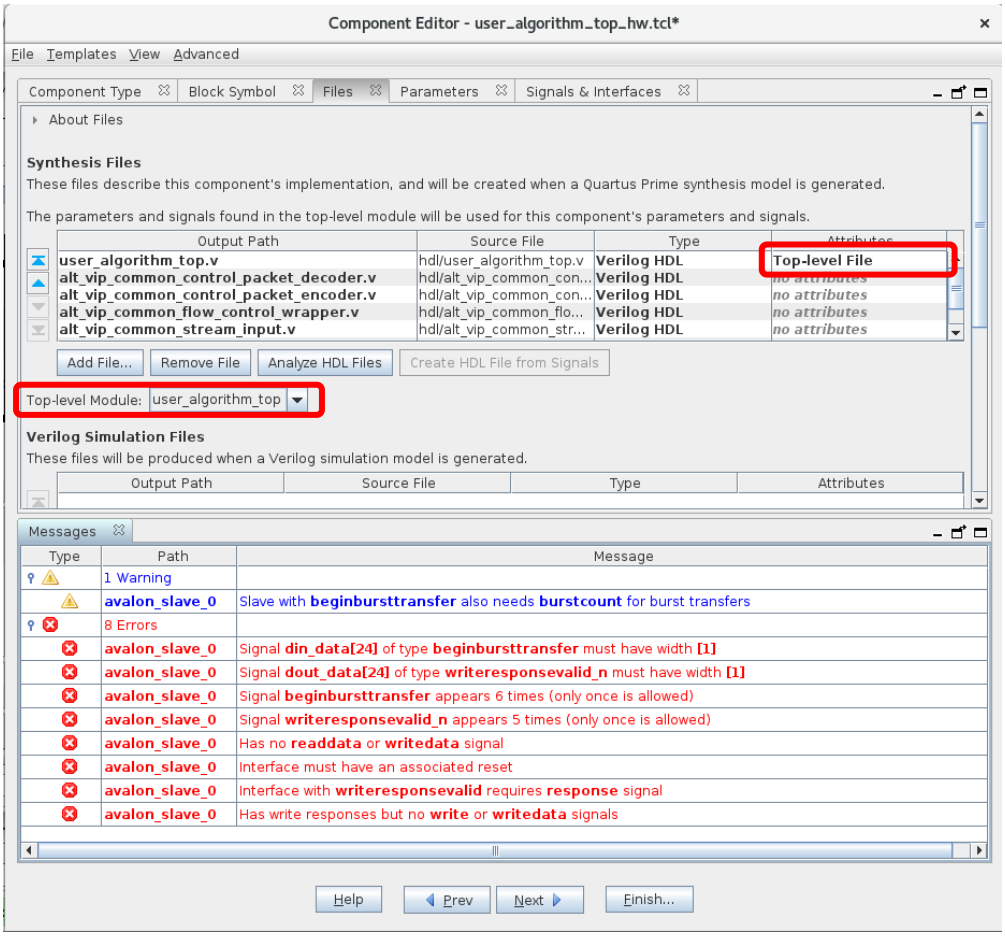


9. A dialog box editor will appear. Type in the fields the information you want as Name, Description, etc… and select **File->Save** in the main toolbar



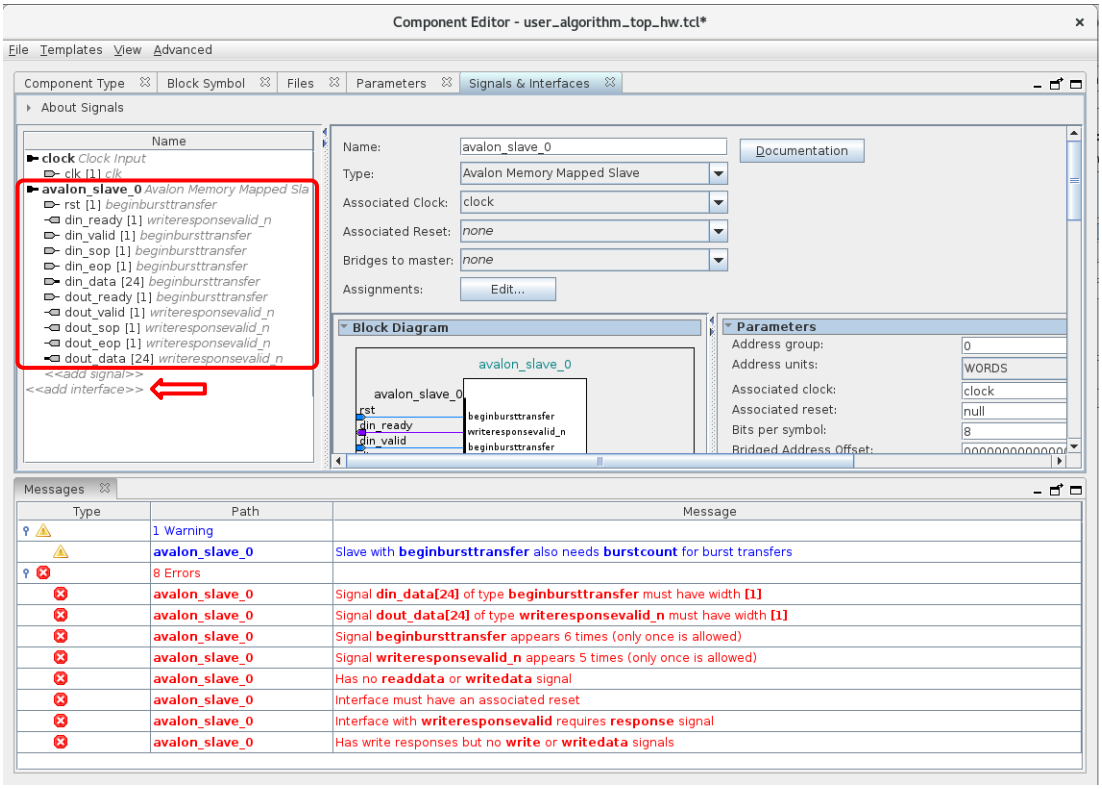
10. Next go to the *Files* tab and click on **Add** File… under *Synthesis Files*. In the dialog box select all the Verilog .v files in the `custom_ip/hdl` folder

11. 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



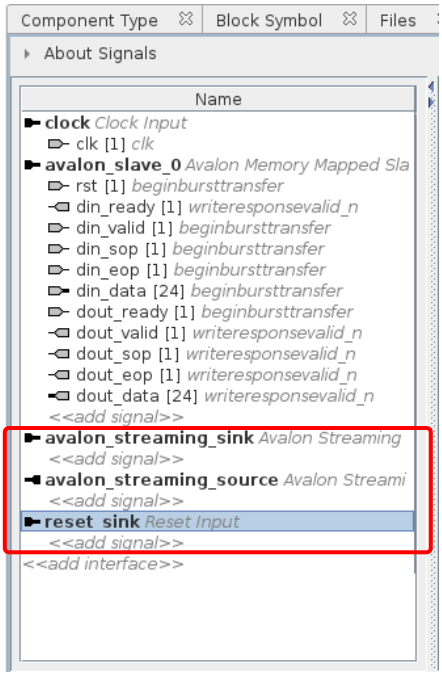
You will notice some error messages that we are fixing next in the *Signals & Interfaces* tab

12. In the *Signal & Interfaces* tab we see that the Avalon-ST video in and out interfaces have been wrongly assigned to an Avalon MM Slave interfaces. Let's modify this.



In the *Name* pane, click 3 times in `<<add interface>>` to add the following interfaces:

- *1x Avalon Streaming Sink*
- *1x Avalon Streaming Source*
- *1x Reset Input*

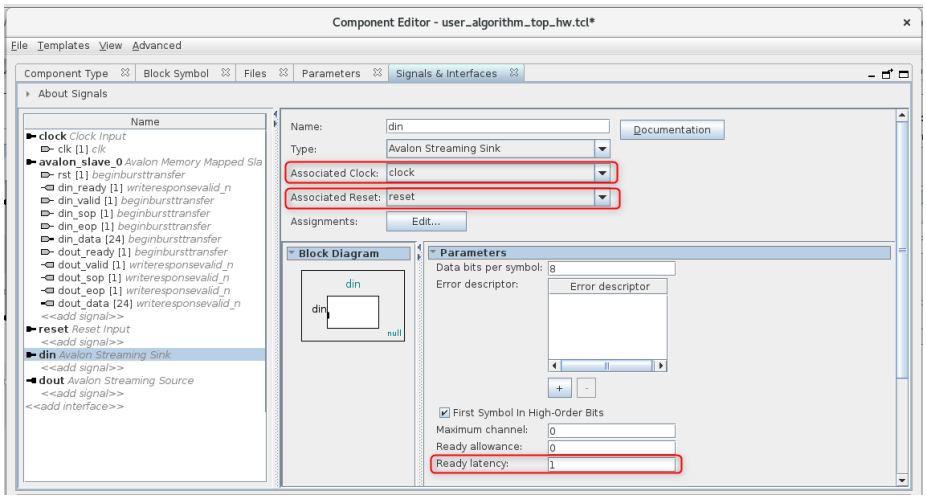


Select each of the newly created interfaces and rename as follows:

- reset_input >> `reset`
- avalon_streaming_sink >> `din`
- avalon_streaming_source >> `dout`

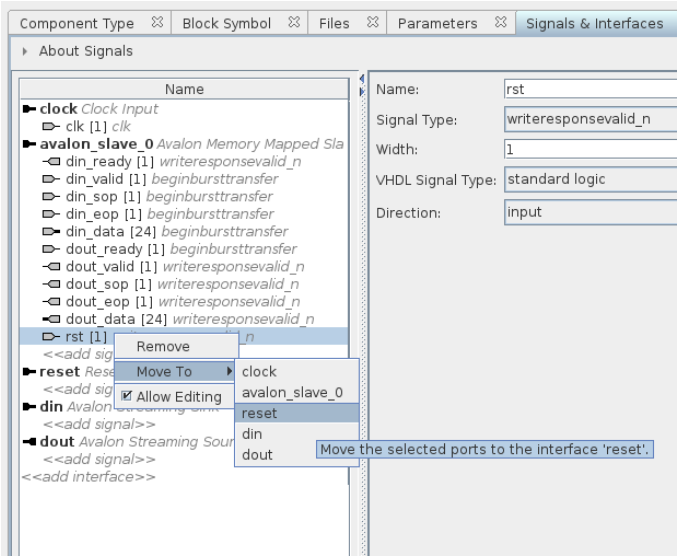
Make sure you set for both `din` & `dout` interfaces:

- Associated Clock >> clock
- Associated Reset >> reset
- Ready latency >> 1 (to be compliant with Avalon-ST latency requirements)

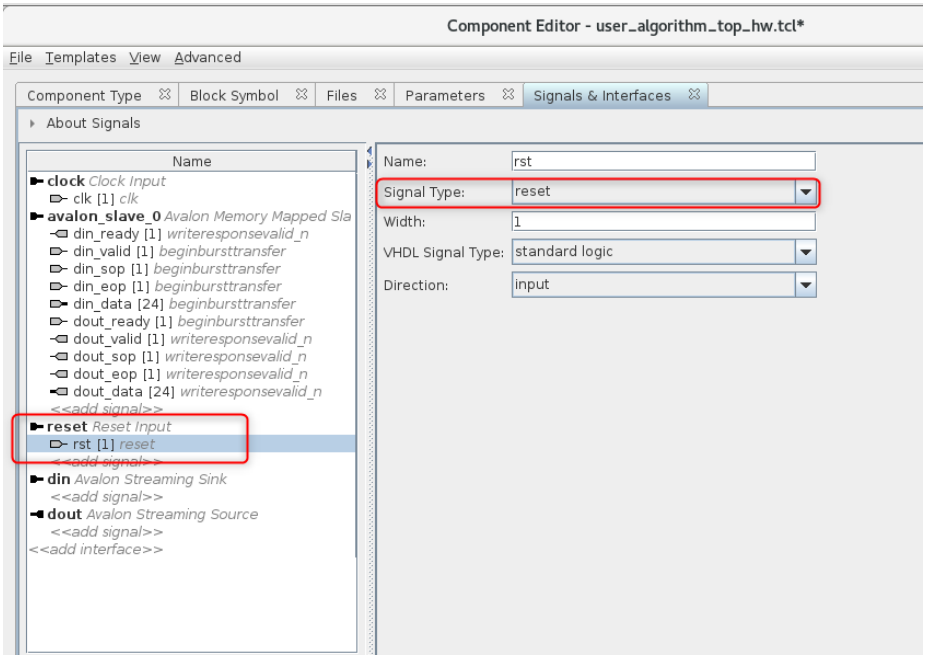


13. Now we need to assign the different signals to the appropriate interface and select the right Signal Type

Select the `rst` signal and right click. **Move To ->** `reset` to assign to the interface



Once there, expand the Signal Type list to select `reset`.

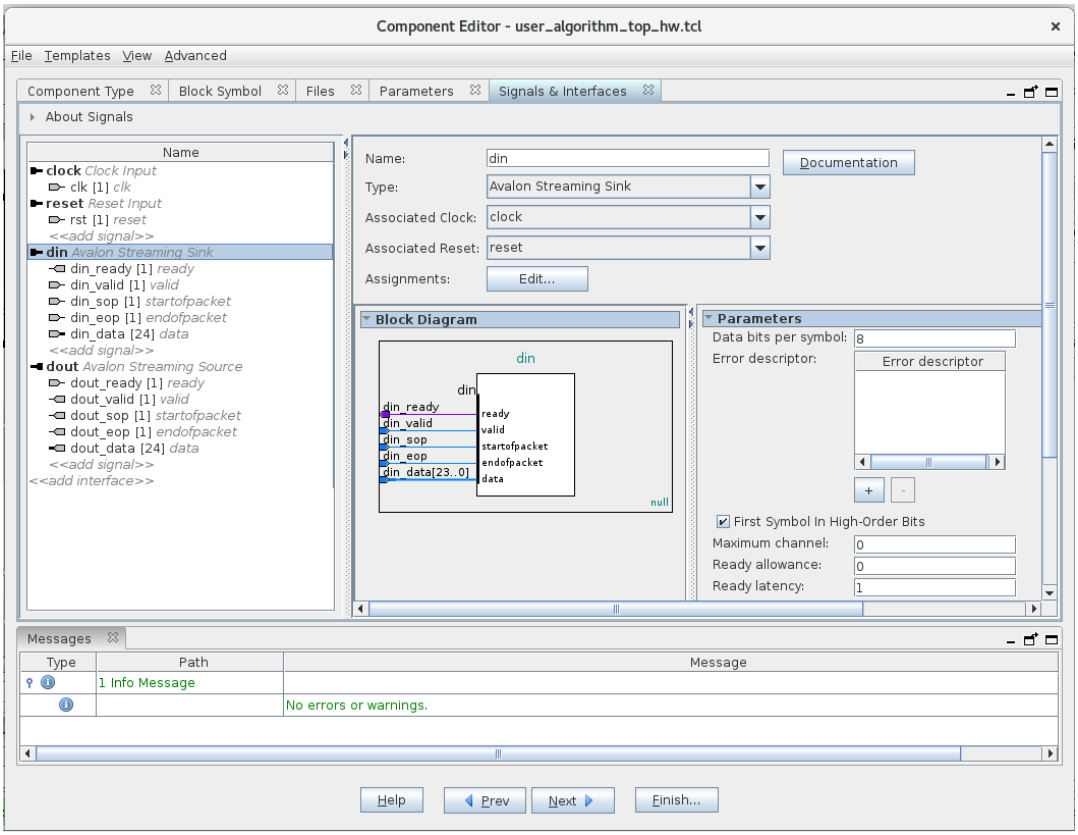


Do all the signals as per the table below

Signal	Interface	Type
rst[1]	reset	reset
din_ready[1]	din	ready
din_valid[1]	din	valid
din_sop[1]	din	startofpacket
din_eop[1]	din	endofpacket
din_data[24]	din	data
dout_ready[1]	dout	ready
dout_valid[1]	dout	valid
dout_sop[1]	dout	startofpacket
dout_eop[1]	dout	endofpacket
dout_data[24]	dout	data

Right click on the **avalon_slave_0** interface and select *Remove*

You should have a component like the one below. Make sure you don't have any error or warning message.



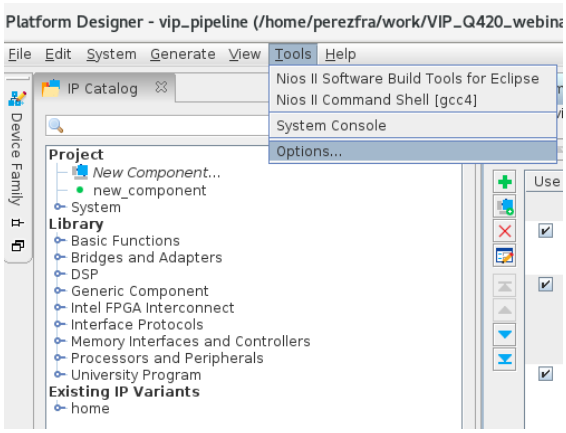
Save the component and click on **Finish** to close the editor.

2.6. Using the provided solution

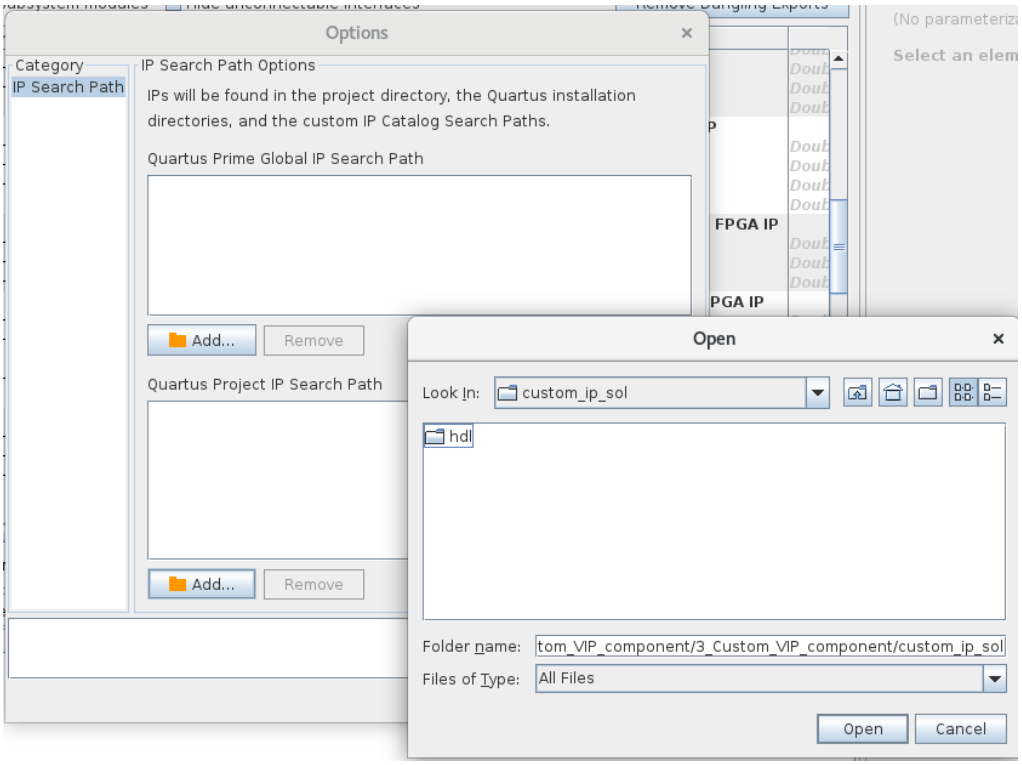
If you want to use the component already finished, provided in the `<extracted_folder>/custom_ip_sol` folder, you just need to add this component to the platform designer IP search path and go to next step: **2.7. Include the component in the pipeline.**

NOTE: If you have followed all the lab exercise, skip this step and move directly to the next one.

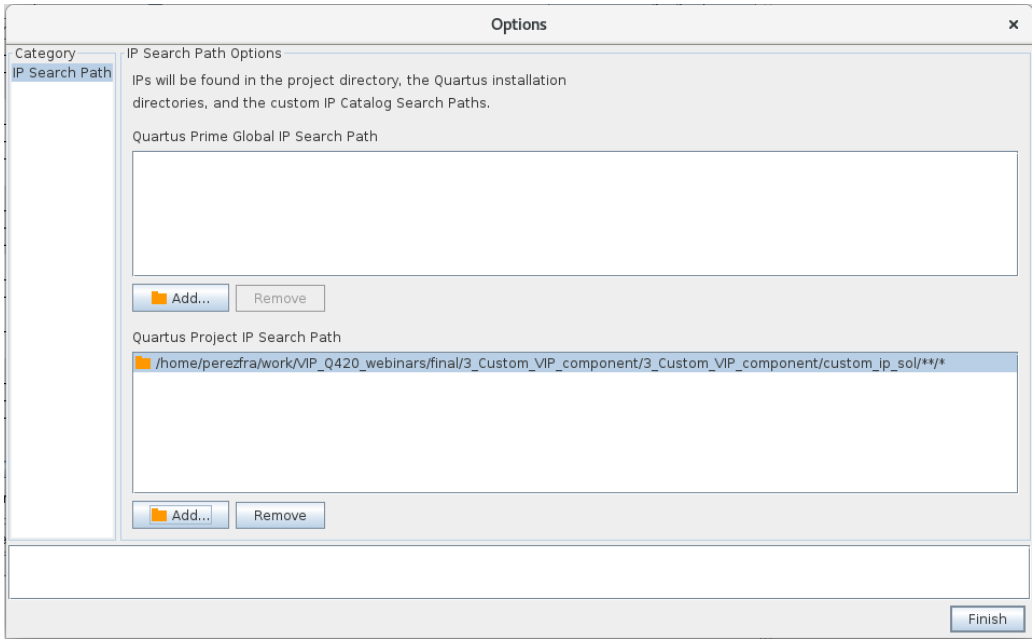
In the main toolbar of Platform Designer select **Tools->Options**…



In the dialog box, click on the **Add** button under *Quartus Project IP Search Path* and select `<extracted_folder>/custom_ip_sol` to be included



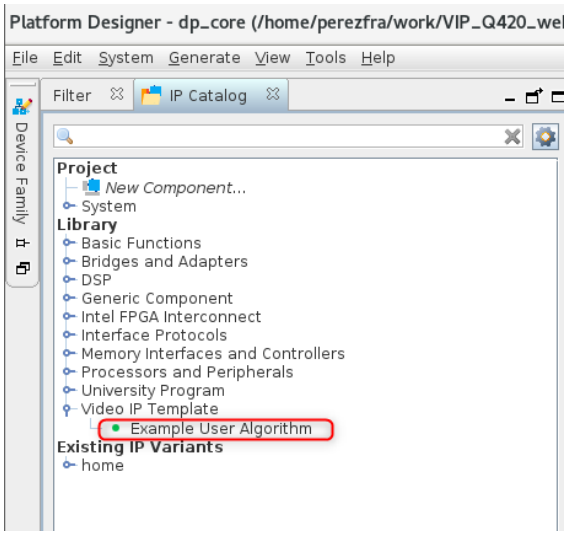
Once done, you should see the folder added to the **IP Search Path**



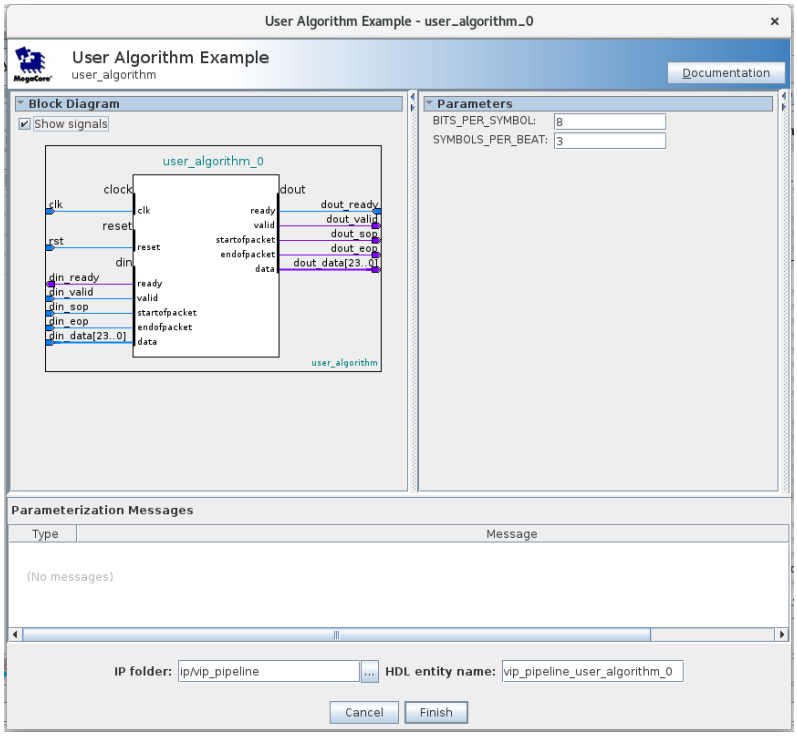
Click **Finish** to close the dialog

2.7.Include the component in the pipeline

After closing the editor, the Platform Designer system is get refreshed and you'll notice a new category under the **IP Catalog->Library->Video IP Template**, where we have available our **User Algorithm Example** component ready to use.



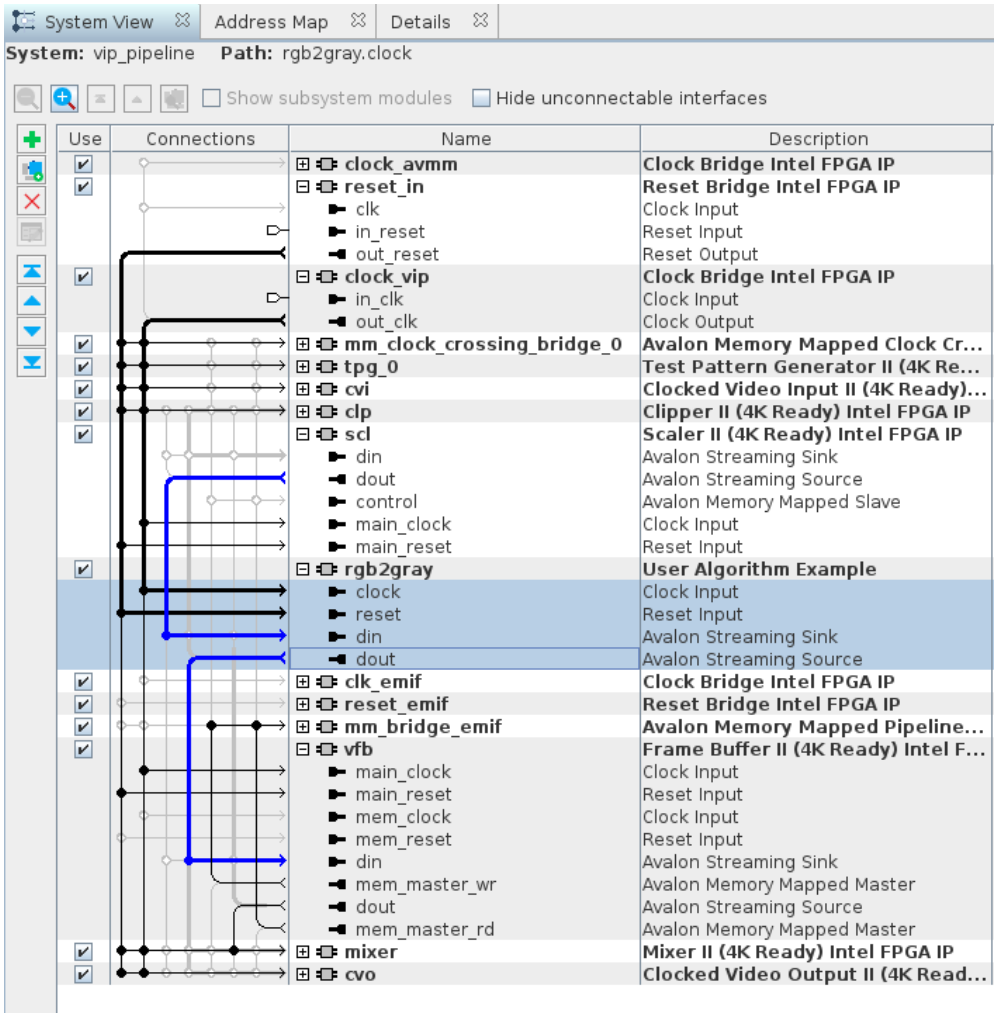
Click on **Add** to open the parameters dialog box. Make sure you select 8 bits per symbols and 3 symbols per beat, as this is the color depth and planes (RGB) we are using in our `vip_pipeline` subsystem



You can see that our custom component has an input Avalon-ST video interface to input data from the upstream module and another one to deliver downstream. Also has the mandatory clock and reset interfaces. So, let's add it and connect to the rest of the pipeline.

Rename the component as **rgb2gray** and move it until right below the Scaler block. Connect the interfaces as follows:

- **clock_vip**: out_clk to
  - **rgb2gray:** clock
- **reset_in**: out_reset to
  - **rgb2gray:** reset
- **scl**: dout to
  - **rgb2gray:** din
- **rgb2gray**: dout to
  - **vfb:** din

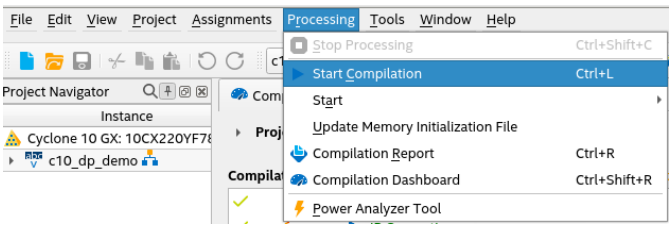


Save the `vip_pipeline.qsys` subsystem and open the higher hierarchy level `dp_core.qsys` and Generate HDL files.

2.8.Generate the bitstream

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.

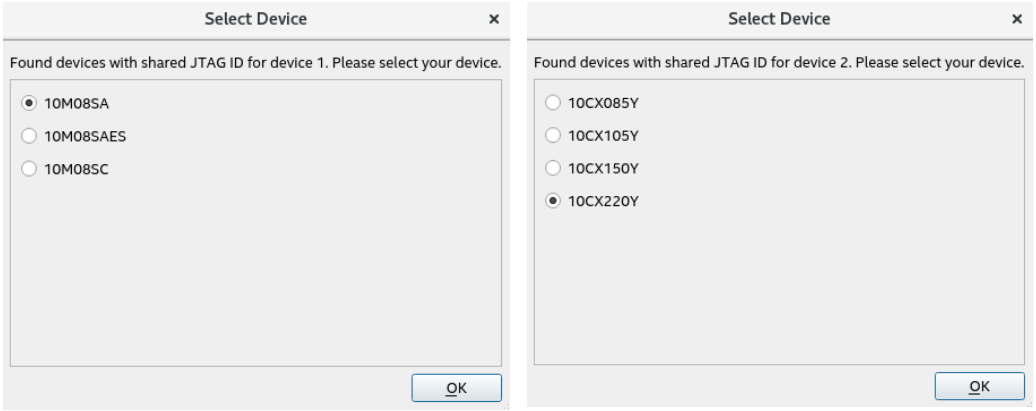
It will take ~12 minutes, depending on machine configuration.



2.9.Configuring the FPGA device

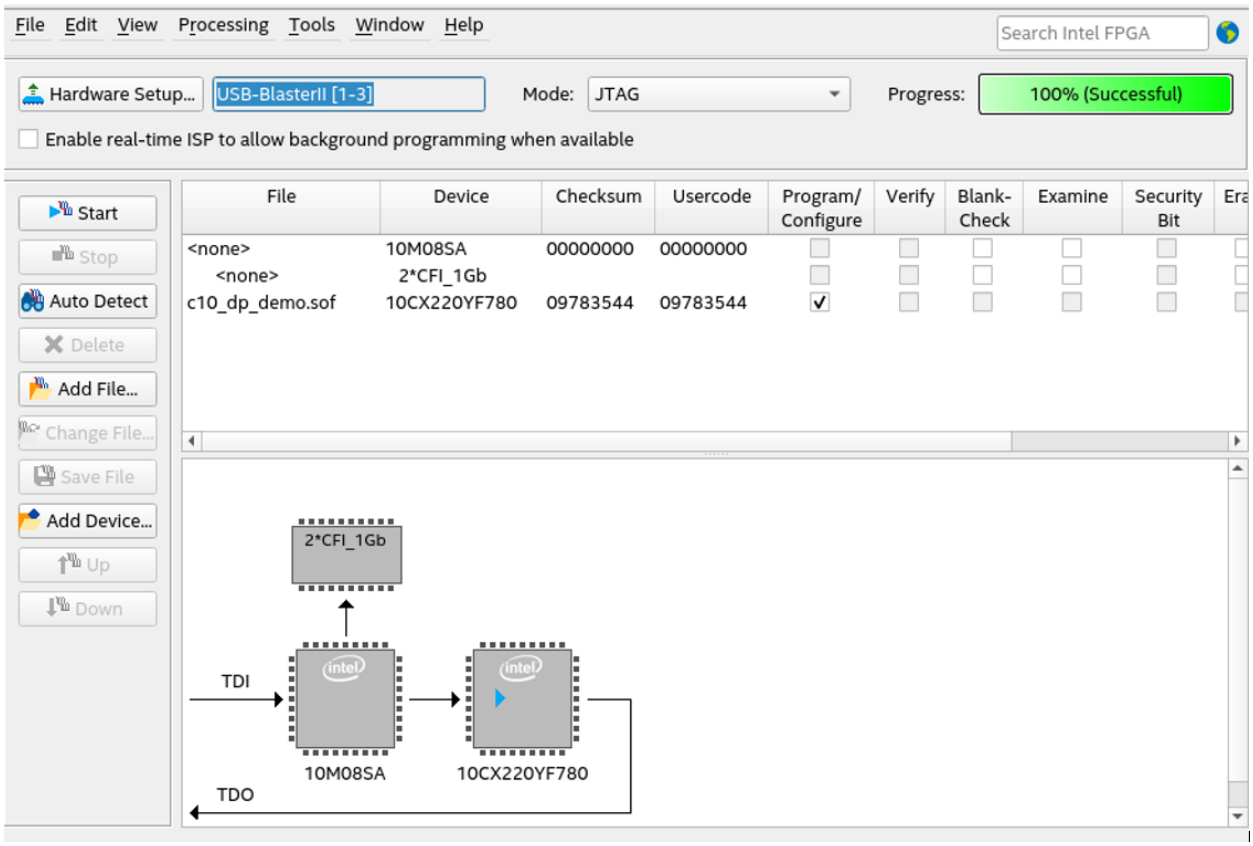
Open the **Quartus Programmer**, select your USB-Blaster cable in the Hardware Setup and click on **Auto Detect** to retrieve the JTAG chain on the Cyclone10 GX Devkit.

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



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

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


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

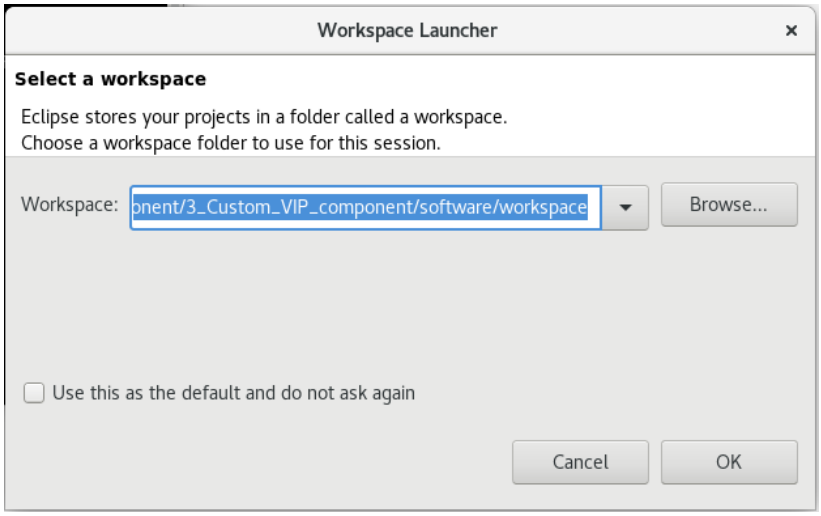
3. Building the software application

3.1.Setting up the Eclipse for Nios project

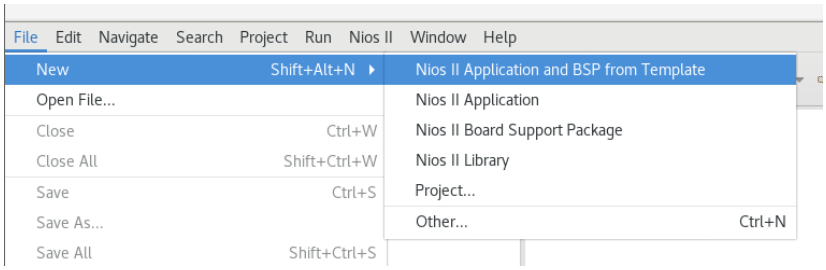
1. Creating Eclipse project for Nios II application and BSP.

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

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



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

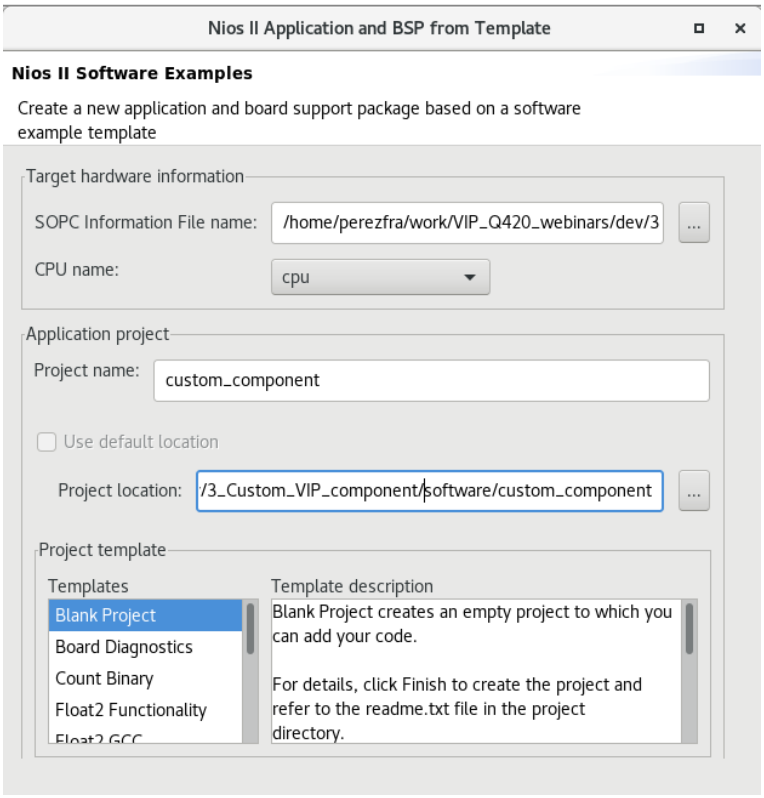


- Under **Target hardware information->SOPC Information File name**, you need to select the `*.sopcinfo` file that contains your Nios CPU. In our case is located in *<project_dir>/rtl/core/dp_core/dp_core.sopcinfo*

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

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

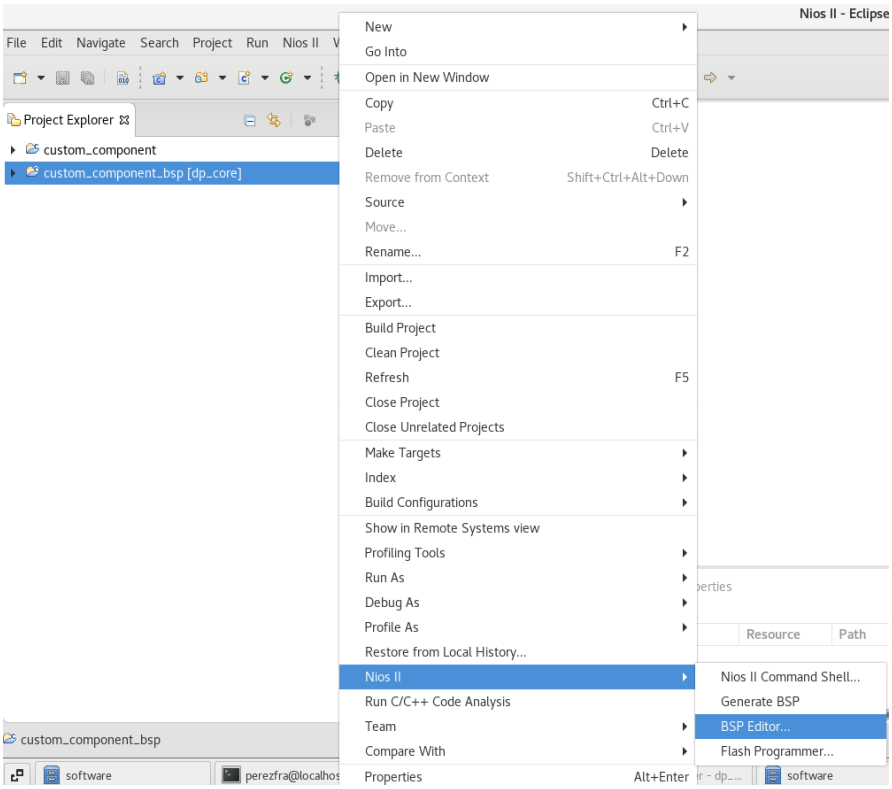
- Select **Blank Project** as **Templates** and Click **Finish**



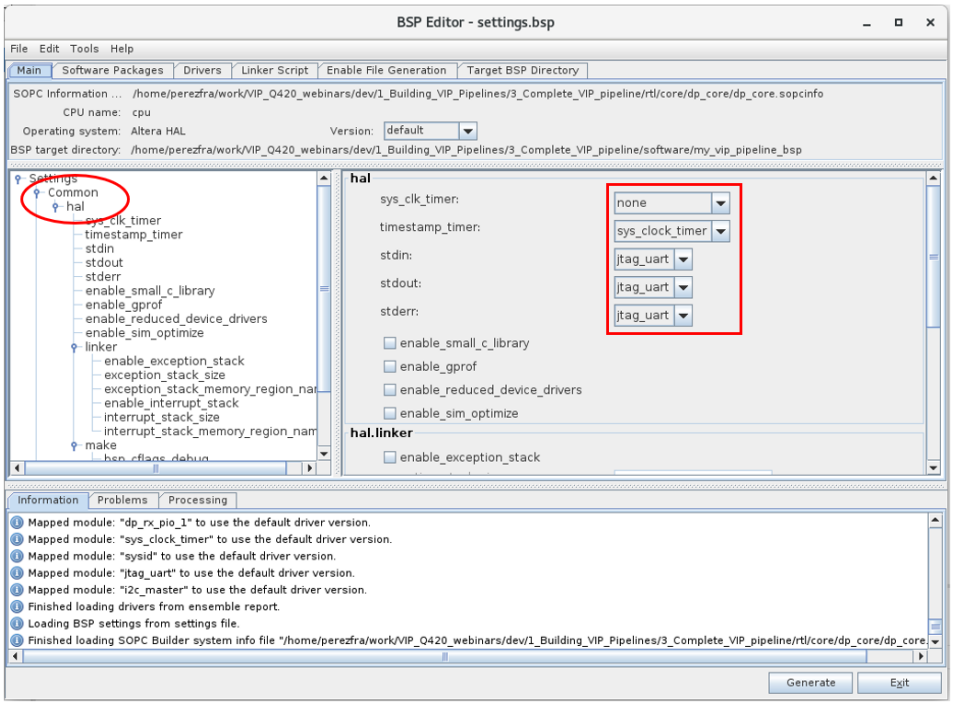
You will end up in a software folder structure as follows:

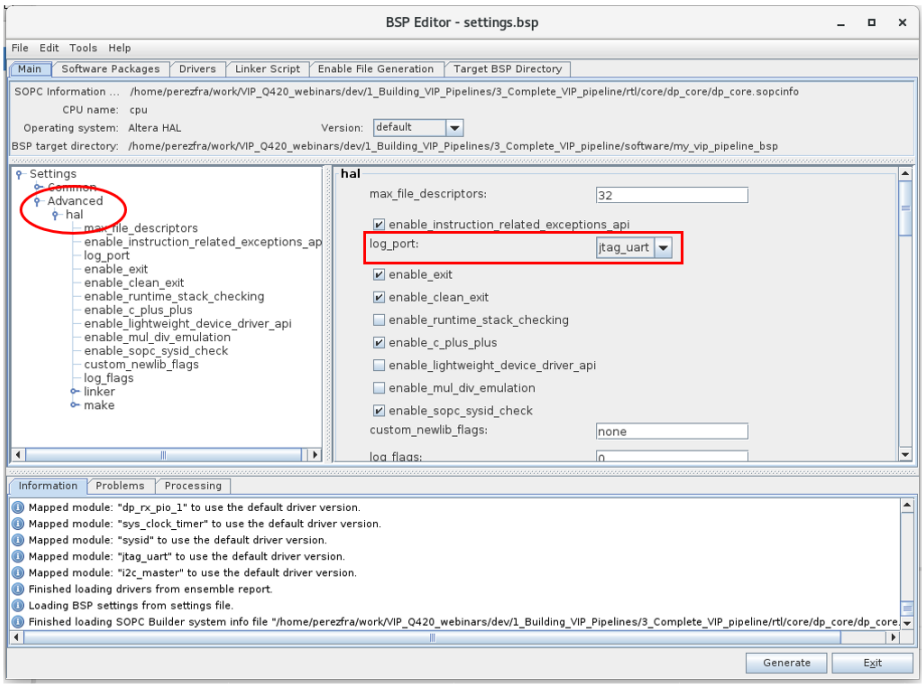
2. **Configure the bsp**.

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

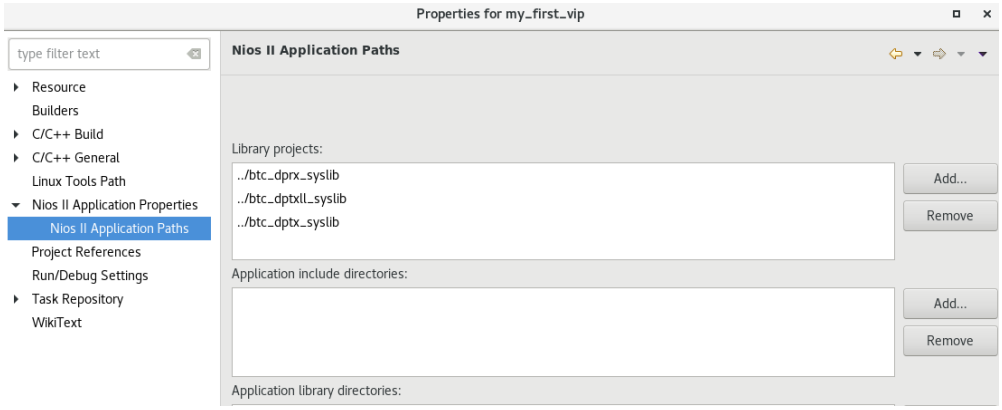




Then click on **Generate** and **Exit**.

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

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

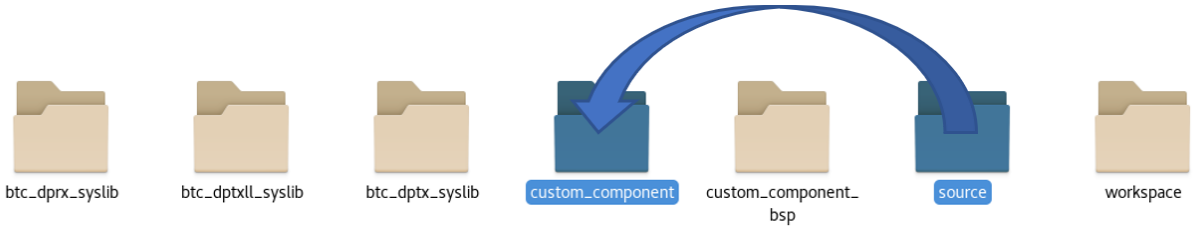


Then click **Apply** and **OK**

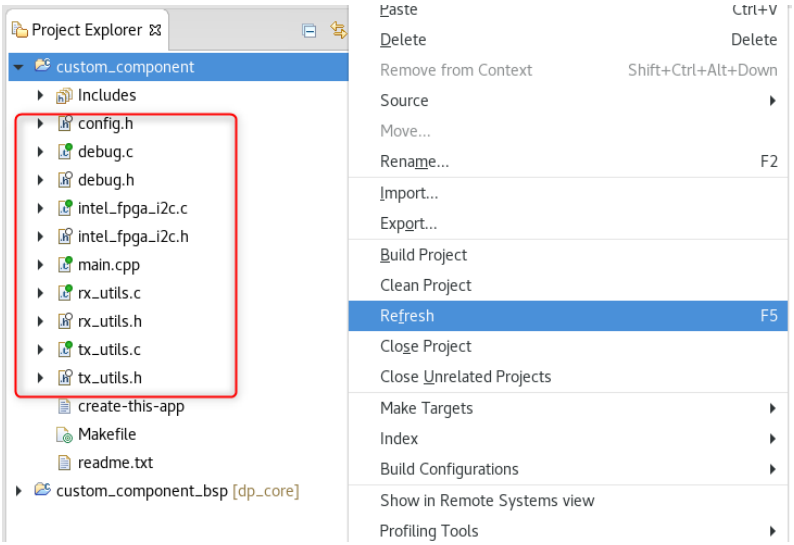
3.2.Importing the code

Copy all the source files provided in the `source` folder into `custom_component` and you can jump directly to **Building and launching program execution,** we don't need to do any modification in the code

as the **rgb2gray** component doesn't need to be configured by software to run and execute its function, indeed it doesn't have any *avalon_mm_slave* port.



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

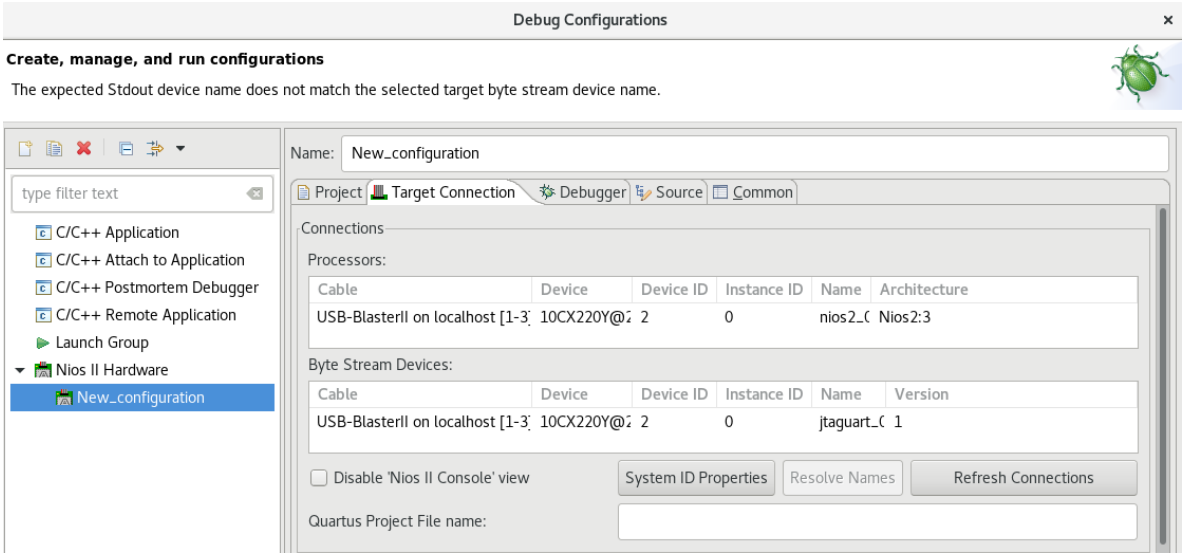


3.3. Building and launching program execution

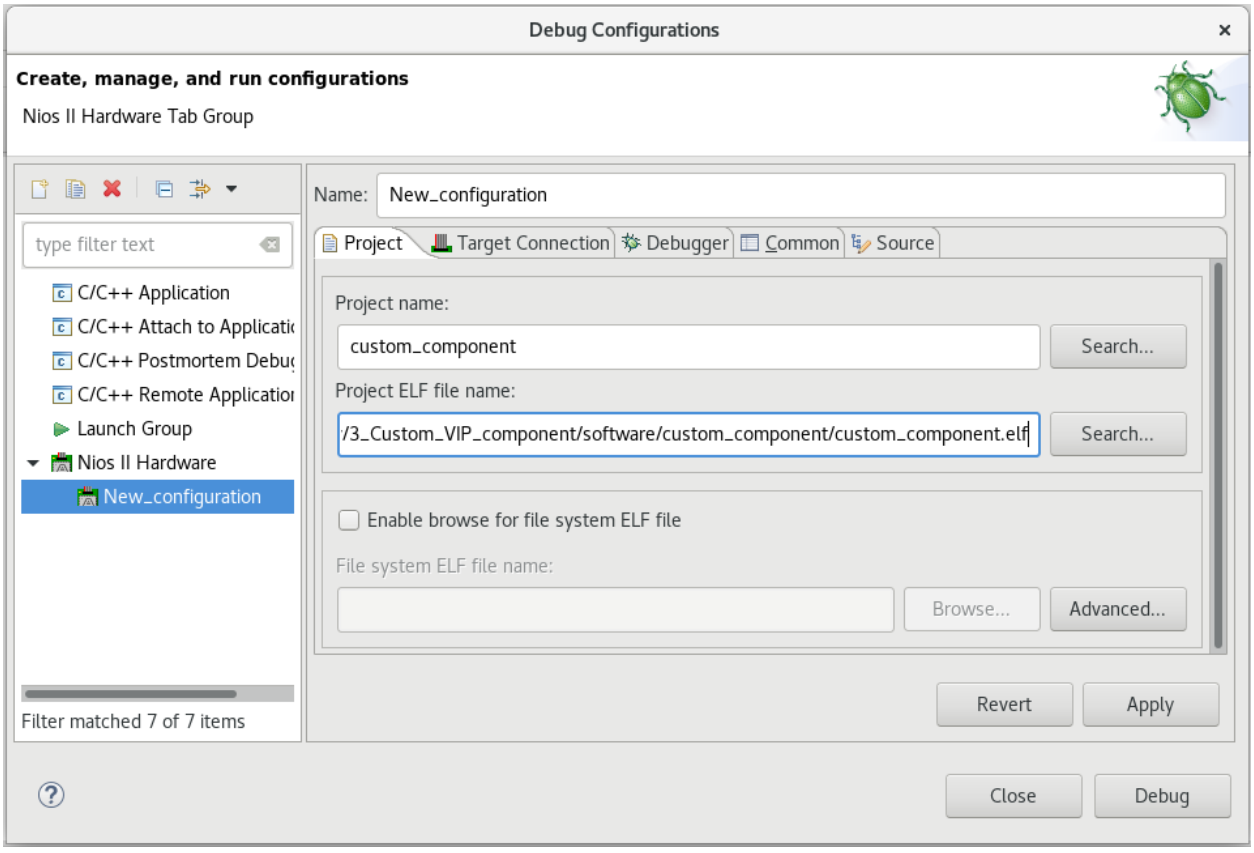
After doing that modifications, we can generate our executable file. In Eclipse IDE, go to **Project->Build All** to compile the *bsp* and generate `custom_component.elf` file.

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

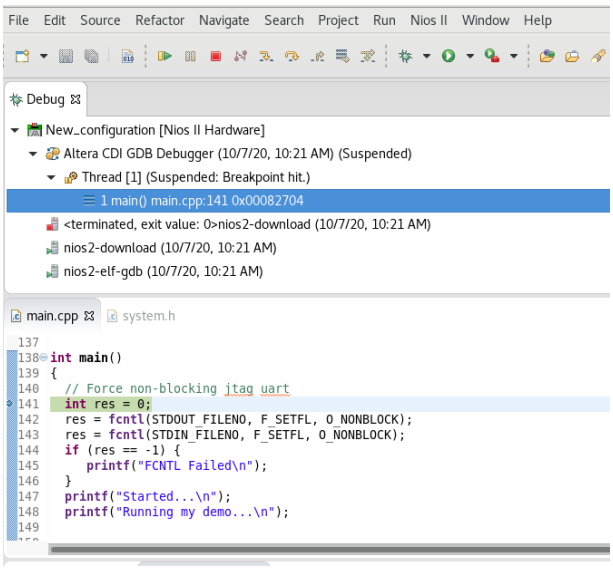
We double-click on **Nios II Hardware** option to create a *New_configuration*. In **Target Connection** tab, click on **Refresh Connections** to select the right *USB-BlasterII* adapter



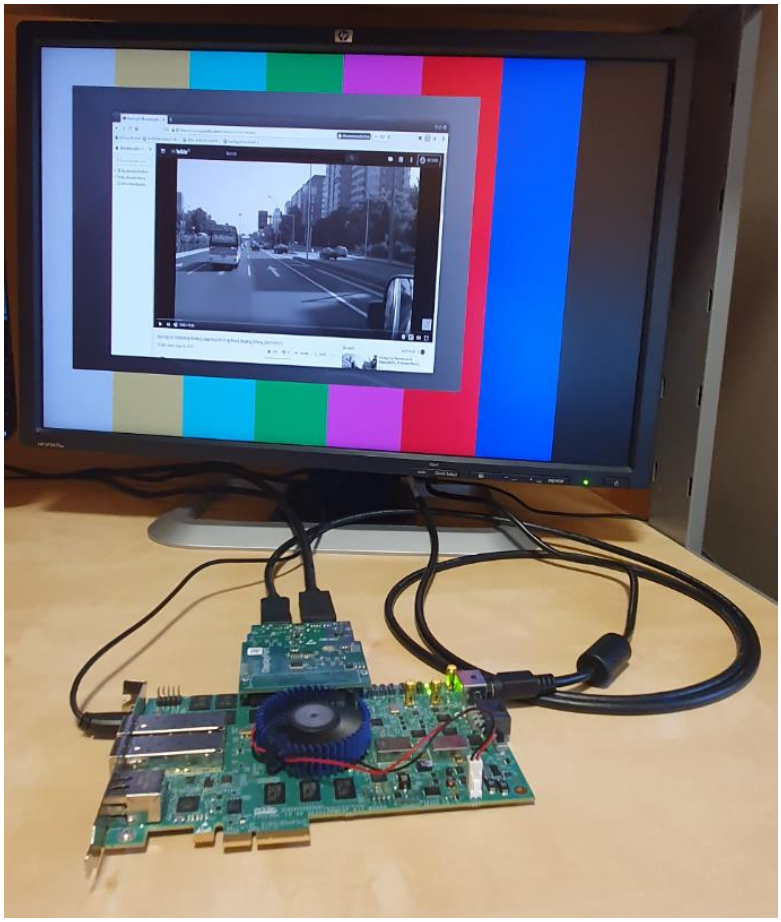
Back to the **Project** tab, make sure the right *Project Name* and *ELF File Name* are selected and just click on **Apply** and **Debug** to launch the session



The executable file is then downloaded 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.



If we have our video source from the external PC connected to the DisplayPort input, we will see the captured image converted to grayscale on top of the color bars generated background.



4. Summary

In this lab, we have exercised with Component Editor in Platform Designer to develop a custom component for VIP suite

- We have followed the steps to build our own proprietary component to implement RGB to grayscale conversion (**rgb2gray**)
- We used the provided HDL template: a collection of hdl files which is facilitating the task to develop a fully compliant VIP component, which can be seamless integrate within a video processing pipeline.
- The template handles all the video and control packets, as well as the backpressure flow control, allowing you to focus on your own `secret sauce´.