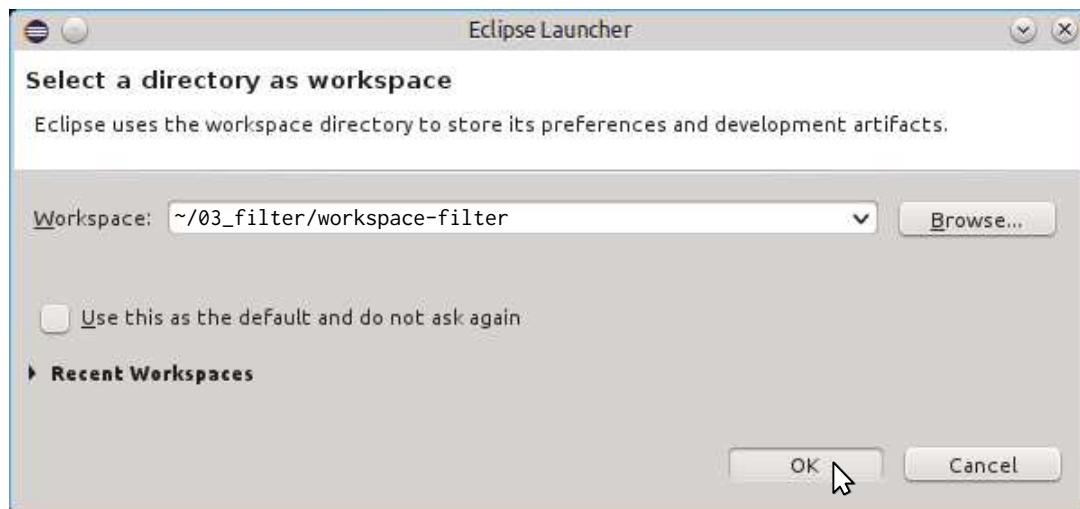


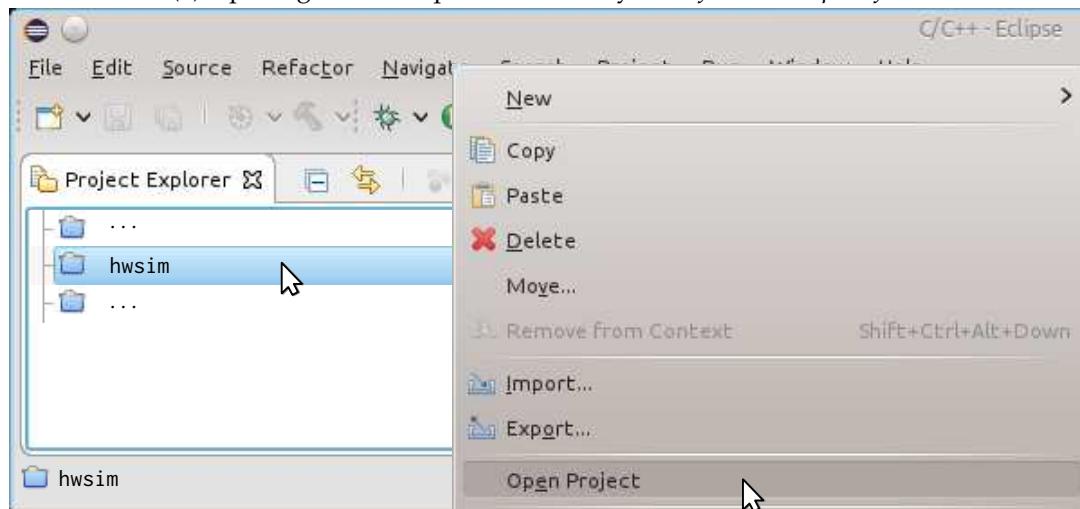
3. Exercise of the Laboratory Entwicklung interaktiver eingebetteter Systeme

1 Introduction

In the following, we will start to work with video filters. For this purpose, an eclipse workspace has been prepared for you as shown in Figure 1.



(a) Opening the workspace in directory `~/03_filter/workspace-filter`



(b) Opening the `hwsim` project in the workspace

Figure 1: The eclipse workspace for the current lab

1.1 Eclipse Workspace of the Lab

Please open the workspace by starting eclipse and choosing the directory depicted in Figure 1a as your workspace directory. Every simulation you perform during this lab will be performed by starting the `hwsim` simulator, which is a parameterizable version of the `hwsim` project from the previous lab. Here, we already provided implementations for the video stream type adapters from the previous lab for you. Consult Section 1.3 for more details on how they are used. If you trust your solutions, you might copy your adapters to this workspace. If you want to, copy over these adapters from workspace `~/02_syst/workspace-syst` to this workspace replacing the below given files.

```
hwsim/src/hwsim/cpp/VideoAdapterFIFOToRTL.hpp  
hwsim/src/hwsim/cpp/VideoAdapterFIFOToRTL.cpp  
hwsim/src/hwsim/cpp/VideoAdapterRTLToFIFO.hpp  
hwsim/src/hwsim/cpp/VideoAdapterRTLToFIFO.cpp
```

Moreover, the `hwsim` simulator has been extended in such a way that video filter chains can be specified via command line, e.g., as depicted in Figures 2 to 3. As a first test, you will compile the simulator at FIFO abstraction level (see Figure 4a) and execute a simple `VideoSource` → `VideoSink` filter chain (see Figure 4b). The filter chain is specified using the following command line arguments for the `hwsim` simulator.

```
--video-filter VideoSource  
--video-filter VideoSink
```

These arguments are specified as shown in Figure 3 for the *run configuration* called *Video source to sink Example*. As a test, execute this run configuration by following the steps given in Figure 4.

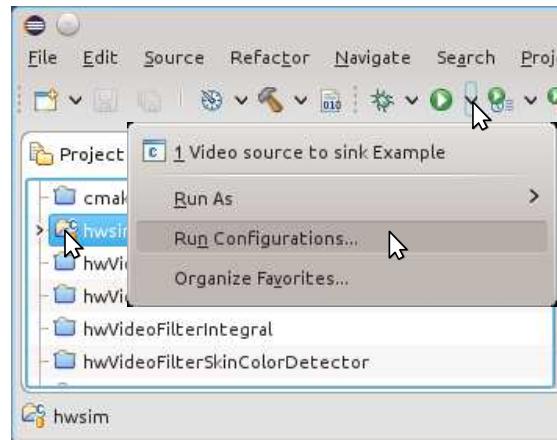


Figure 2: Opening the run configurations

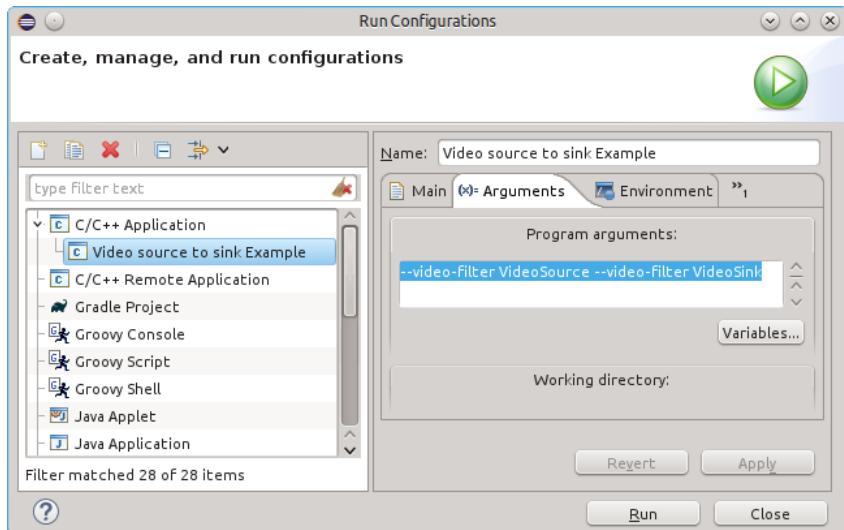
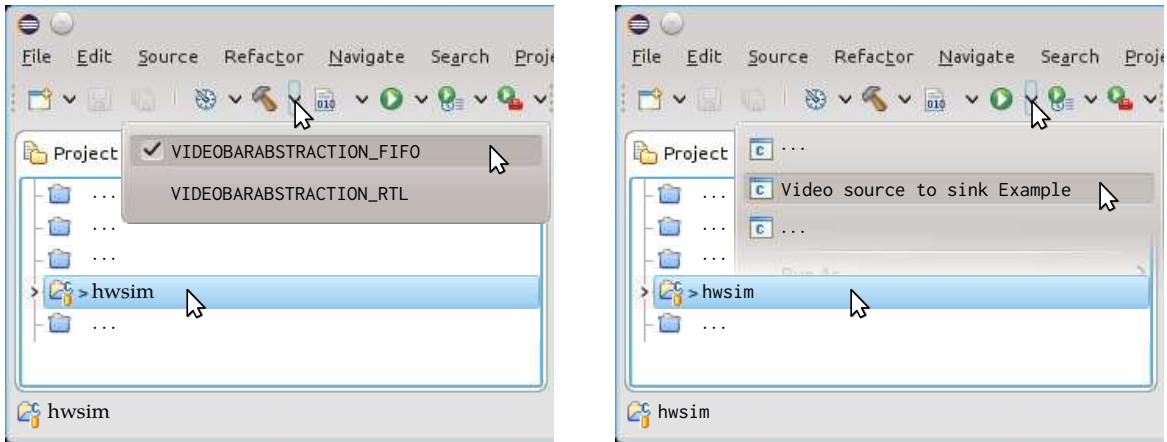


Figure 3: Examining the run configuration *Video source to sink Example* in more detail



(a) Compiling the simulator at FIFO abstraction level (b) Starting the VideoSource —> VideoSink example

Figure 4: Compiling the `hwsim` simulator at different levels of abstraction for the communication and starting the example video source to sink filter chain

Finally, not only the `hwsim` simulator is contained in the workspace, but six other projects representing various filters and video sources. These filter can be loaded as plugins by the `hwsim` simulator and used to build the filter chain. In detail, the `hwsim` simulator as well as two example projects are given to you for testing purpose and to base your own filters on.

1. `hwsim` – A SystemC simulation environment that contains a `VideoSource` that captures images from a USB camera, which will be given to you during the lab, and a `VideoSink` that displays the video stream in a window.
2. `hwVideoSourceColorCheckBoard` – A video source that generates a Color Check Board (CCB) test image.
3. `hwVideoFilterSlidingWindow` – A local filter, e.g., Sobel edge detection.

Moreover, there are four more video filters that will be completed by you as part of this lab.

4. `hwVideoFilterSkinColorDetector` – A filter that should detect skin color.
5. `hwVideoFilterIntegral` – A filter that calculates summed area information of marked pixels. Pixels should be marked by a preceding skin color detection filter.
6. `hwVideoFilterGrayscaleConversion` – A gray scale filter to be used as part of a filter chain in the `hwsim` simulator.
7. `hwVideoFilterStudent` – Your own video filter that will hopefully become part of the final demonstrator of the whole lab.

1.2 Plugin Mechanism of the `hwsim` Simulator

The `hwsim` simulator itself provides exactly two video filters¹ `VideoSource` and `VideoSink`. All other filters, e.g., `VideoSourceColorCheckBoard`, `VideoFilterSlidingWindow`, etc., are provided by *plugins* that the other projects in your workspace will compile into. To exemplify, use Figure 5 to compile `VideoSourceColorCheckBoard`. This will result

¹To be exact it is a source and a sink and not filters.

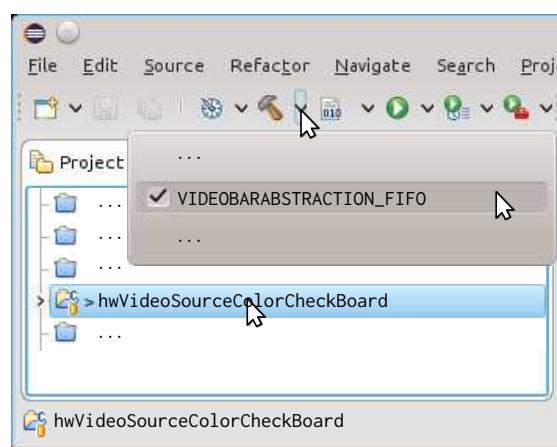
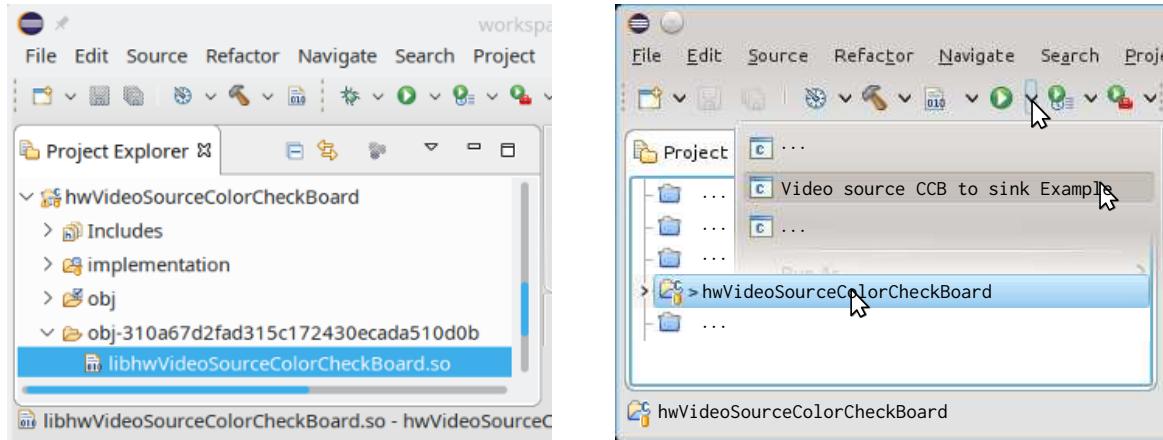


Figure 5: Compile the CCB in FIFO abstraction

in the plugin library *libhwVideoSourceColorCheckBoard.so* shown in Figure 6a. If the object directory *obj-310a...* does not appear in eclipse, then there is a differences between what files are present in the file system and what files Eclipse knows of. To notify Eclipse that new files have appeared in the project, you have to press *F5* while the *hwVideoSourceColorCheckBoard* project is selected in the project explorer.



(a) Resulting plugin library *libhwVideoSourceColorCheckBoard.so* (b) Starting the *VideoSourceColorCheckBoard* → *VideoSink* example

```

workspace-filter - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Problems Tasks Console Properties Call Graph
<terminated> (exit value: 0) Video source color check board to sink Example [C/C++ Application]
Registering module VideoSink
Registering module VideoSource

SystemC 2.3.1-Accellera --- May 7 2018 11:30:25
Copyright (c) 1996-2014 by all Contributors,
ALL RIGHTS RESERVED
Gtk-Message: 17:06:02.610: Failed to load module "overlay-scrollbar"
Loading plugin "...rCheckBoard/obj/libhwVideoSourceColorCheckBoard.so"
Registering module VideoSourceColorCheckBoard
top.videoSink: enableReception @0 s

Info: (I702) default timescale unit used for tracing: 1 ps (videoSourc

```

(c) Output of the *VideoSourceColorCheckBoard* → *VideoSink* example

Figure 6: An example of the *hwsim* simulator loading a plugin

Now, start the *Video source CCB to sink Example* run configuration as depicted in Figure 6b. As depicted in Figure 6c and underlined in green, the *hwsim* simulator loads the shared library (i.e., the plugin library) *libhwVideoSourceColorCheckBoard.so* that provides the *VideoSourceColorCheckBoard* video source. This video source should generate the check board pattern depicted in Figure 7. In general, the *hwsim* simulator will load all shared libraries matching the <workspace

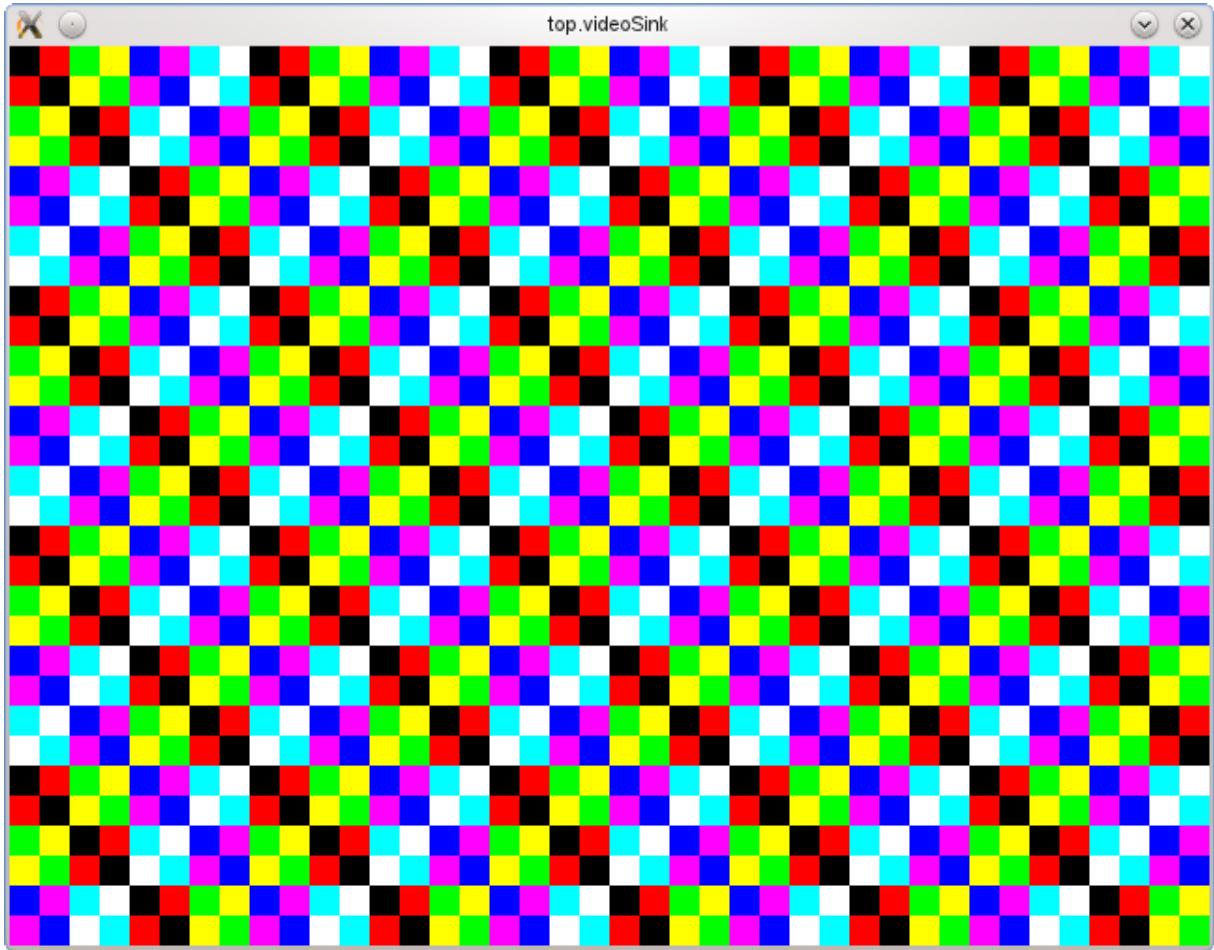


Figure 7: Color check board generated by VideoSourceColorCheckBoard

path>/hw/obj/libhw*.so glob*, i.e., all shared libraries matching *libhw*.so* in the object directory *obj* of all eclipse projects starting with *hw*. The convention here is that projects starting with *hw* contain SystemC code that will be synthesized into Intellectual Property Blocks (IPBs).

You might have noticed that the `hwsim` simulator loads the plugin library from the *obj* directory and not directly from the *obj-310a...* directory. This feature enables the `hwsim` simulator to pick up the plugin library providing the filter at the abstraction level you last compiled the filter at. For this purpose, the *obj* directory is a symbolic link to the real object directory produced by the *build configuration* – `VIDEOBARABSTRACTION_FIFO`, `VIDEOBARABSTRACTION RTL`, etc. – you last used to compile the project. If you last used `VIDEOBARABSTRACTION_FIFO` to compile, then the project should look as follows:

```
[...test@codesigni165:~]$ cd ~/03_filter/workspace-filter
[...65:workspace-filter]$ cd hwVideoSourceColorCheckBoard
[...urceColorCheckBoard]$ ls -lad obj*
1rwxrwxrwx 1 test stud 36 Mar 1 14:56 obj -> obj-310a67d2fad315c172430ecada510d0b
drwxr-xr-x 4 test stud 4096 Mar 1 14:56 obj-310a67d2fad315c172430ecada510d0b
```

To exemplify an update of the *obj* symbolic link, compile the `VideoSourceColorCheckBoard` at RTL abstraction, i.e., as seen in Figure 5 but with `VIDEOBARABSTRACTION_RTL`. Then, list again what happened with the object directories in the `hwVideoSourceColorCheckBoard` project.

```
[...urceColorCheckBoard]$ ls -lad obj*
1rwxrwxrwx 1 test stud 36 Mar 4 08:56 obj -> obj-97f701c5b126235a4f0285c667d9cc76
```

```
drwxr-xr-x 4 test stud 4096 Mar 1 14:56 obj-310a67d2fad315c172430ecada510d0b
drwxr-xr-x 4 test stud 4096 Mar 4 08:56 obj-97f701c5b126235a4f0285c667d9cc76
```

As can be seen, compiling with the VIDEOBARABSTRACTION_RTL build configuration resulted in the new object directory *obj-97f7...* as well as an update of the *obj* symbolic link to this directory.

1.3 Automatic Adapter Insertions

Depending on the build configurations chosen for the `hwsim` simulator and the `VideoSourceColorCheckBoard` module, there might be either a direct connection between the `VideoSourceColorCheckBoard` and `VideoSink` modules or an appropriate adapter must be inserted to enable communication. However, for all combinations (cf. Figure 8), the filter chain is parameterized via the following arguments contained in the *Video source CCB to sink Example* run configuration:

```
--video-filter VideoSourceColorCheckBoard
--video-filter VideoSink
```

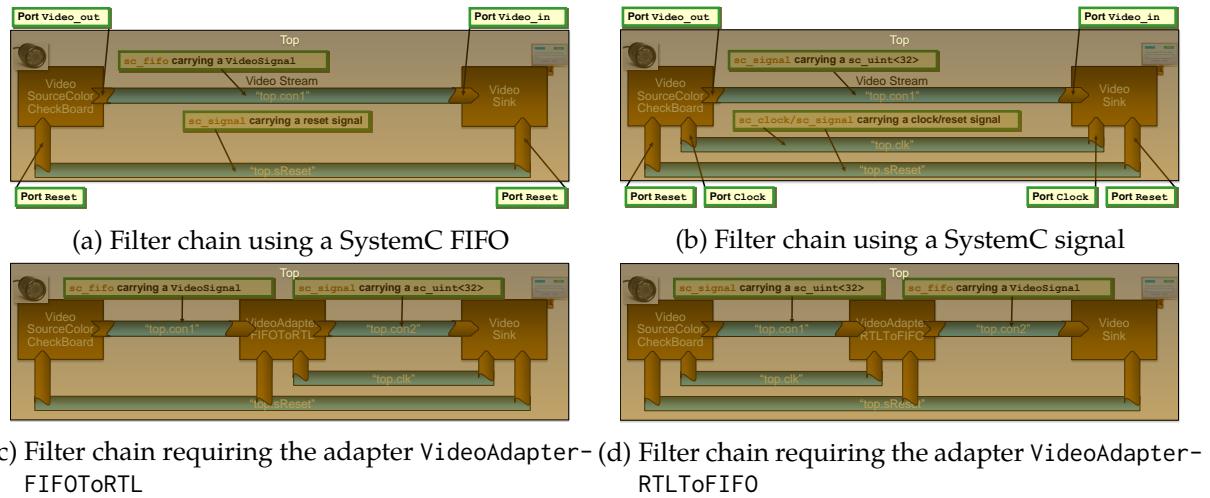


Figure 8: Various combination abstraction levels used for the `VideoSourceColorCheckBoard` → `VideoSink` filter chain using either a direct connection (a and b) or requiring the automatic insertion (c and d) of adapters

The need for an additional adapter to connect `VideoSourceColorCheckBoard` to `VideoSink` is inferred automatically from the types of the `Video_in` and `Video_out` ports of these modules. If they are compatible, the respective channel for connecting them is used directly. To exemplify, this is a SystemC FIFO channel (as depicted in Figure 8a) when `Video_out` and `Video_in` are of type `sc_fifo_out<VideoSignal>` and `sc_fifo_in<VideoSignal>`, respectively. If these ports are of type `sc_out<sc_uint<32>` and `sc_in<sc_uint<32>`, respectively, then a SystemC signal (cf. Figure 8b) is used to connect them. Otherwise, an adapter is needed to connect both modules as is depicted in Figures 8c and 8d. This automatic inference and insertion of needed adapters works for all specified filter chains so that all combinations of build configurations should result in a valid simulation. **However, this assumes that the adapters work correctly. If you copied in your solutions, subtle bugs might happen. Please keep this in mind as a source of errors.**

1.4 The Zybo Zynq-7000 Board

The `hwsim` simulator stands in for a real hardware platform, the *Zybo Zynq-7000 ARM/FPGA Programmable System-on-Chip (PSoC) Board* depicted in Figure 9. In this lab, you will use the PSoC

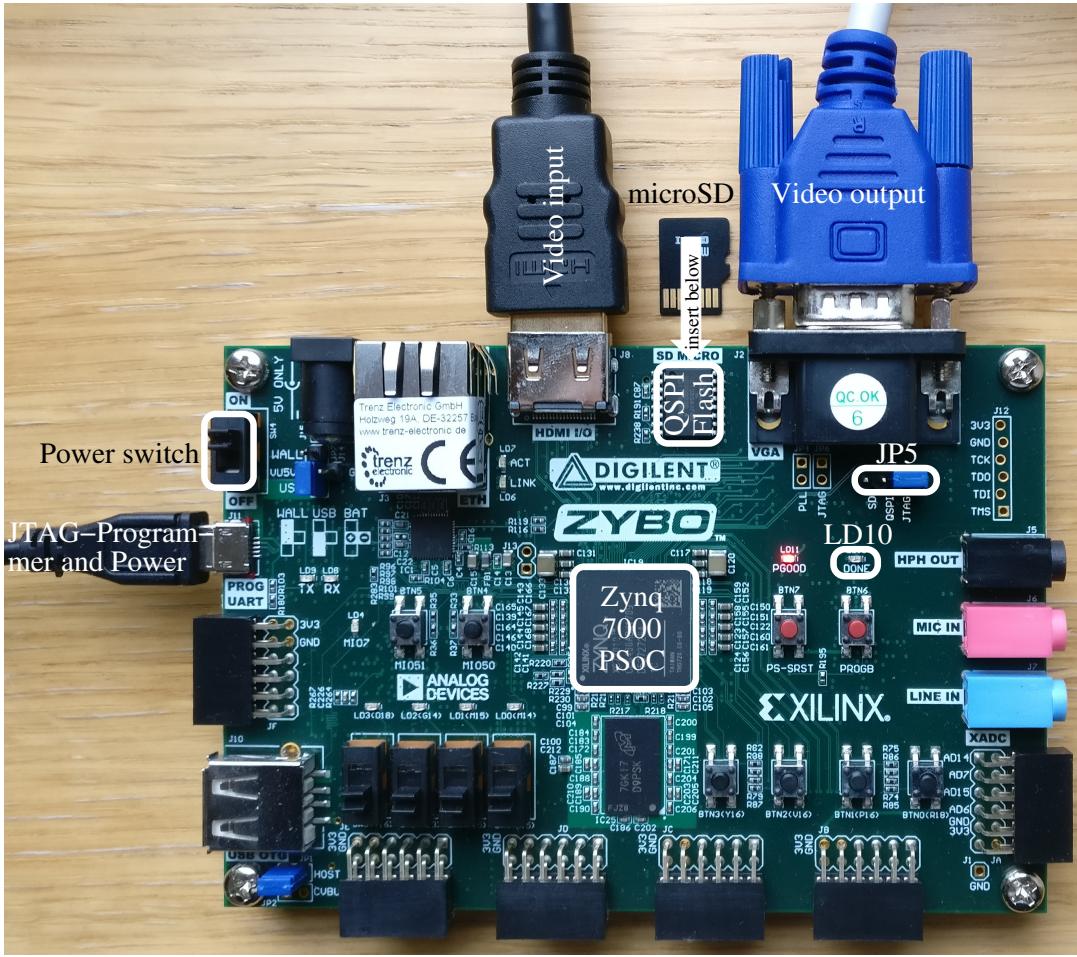


Figure 9: The Zybo Zynq-7000 ARM/FPGA PSoC Board

to perform video processing. For this purpose, you need a video input to get the video stream into the PSoC and a video output to display the result on your monitor. For the video input, you will use the HDMI input provided by the board (not the USB connection). For the video output, you will use the VGA output provided by the board. Furthermore, for power and to program the board, you will need a USB cable and connect it to the *PROG UART* port.

Moreover, the architecture of the Zynq-7000 PSoC is shown in Figure 10. As can be seen, the architecture can be split into two main parts: (i) the Processing System (PS) containing a *dual-core ARM Cortex™ A9 CPU* and (ii) the Programmable Logic (PL) that will *host your developed IPBs*. In this lab, we will concentrate on the PL part of the Zynq-7000 PSoC. In the next lab, you will learn Transaction-Level Modeling (TLM) that can be used to model the PS of the chip and, finally, in the last lab we will integrate both parts to obtain a virtual prototype of the Zybo board.

However, currently you will focus on programming the PL part – also called a Field-Programmable Gate Array (FPGA) – of the chip. For this purpose, you will generate *bit files* by using the Vivado tool from Xilinx. A bit file contains the configuration for the PL of the chip. You will use Vivado to stitch together IPBs into a design for the PL and use the tool to generate a bit file for each of your designs.

Of further interest is the jumper *JP5* – the *programming mode jumper*. This jumper selects from where the design for the PSoC is loaded. There are three different sources for the design: (i) JP5 to the left boots from the microSD card, (ii) JP5 in the middle boots from the QSPI flash, and (iii) JP5 to the right waits for download via JTAG-programmer. In the current lab, you will use

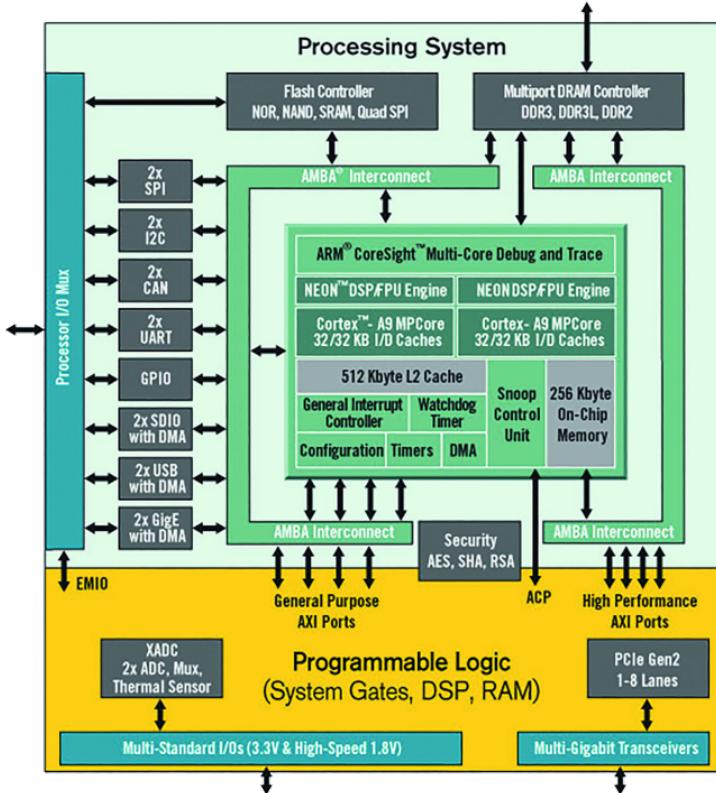


Figure 10: Architecture of the Zynq-7000 ARM/FPGA PSoC

the JTAG-programmer for programming. Thus, leave jumper JP5 in the position it is depicted in Figure 9. In later labs, you will copy your design onto a microSD card and insert this into the Zybo board used by the demonstration hardware. In any case, *LD10* – the *done led* – will turn on (it is a green led) if your design has loaded successfully.

Finally, [~/03_filter/documentation/ZYBO_RM_B_V6.pdf](#) provides further documentation for the board if you are interested.

1.5 Testing the Video Output

To test the video input and output independent from errors you might make in the video filters that you will realize in this lab, we provided two example designs. The first example design – *VideoSourceColorCheckBoard* – only outputs a test image, e.g., as seen in Figure 7. You can use the CCB test design to test if you have correctly connected the Zybo board and it can output a video stream to your monitor. For this purpose, the *hwVideoSourceColorCheckBoard* contains the pre-generated *ccb_wrapper.bit* bit file and the *ZyboInit.sh* script to program the PSoC with this bit file.

If you want to test the Zybo board and how to connect it to your monitor, use the *Micro USB cable* (see Figure 11a) to connect your computer with the Zybo board and the *VGA cable* (see Figure 11b) to connect the Zybo board to your monitor. Then, execute the *ZyboInit.sh* script contained in the *hwVideoSourceColorCheckBoard* project. This should result in output similar to the one provided below:

```
[...test@codesigni165:~]$ cd ~/03_filter/workspace-filter
[...65:workspace-filter]$ cd hwVideoSourceColorCheckBoard
[...urceColorCheckBoard]$ ./ZyboInit.sh
```



(a) Micro USB cable to connect the Zybo board to your computer to facilitate power provisioning and JTAG programming of the PSoC
 (b) VGA cable for connecting the Zybo board to your monitor to facilitate video output

Figure 11: Cables required to connect the Zybo board to facilitate video output

Before I can program the Zybo board with your executable and bitstream, you have to turn the Zybo board off and change JP5 to JTAG mode. Then, turn the board on again. Even if JP5 is already set to JTAG mode, you must turn the Zybo board off and on again to guarantee a clean programming. If you don't, the programming might or might not work properly.

Press enter when ready!

Make sure that JP5 is in JTAG mode before you program the board. If JP5 is in the same position as shown in Figure 9, you can continue by pressing enter.

```
attempting to launch hw_server
```

```
***** Xilinx hw_server v2017.1
**** Build date : Apr 14 2017-19:01:52
** Copyright 1986-2017 Xilinx, Inc. All Rights Reserved.
```

```
INFO: hw_server application started
INFO: Use Ctrl-C to exit hw_server application
```

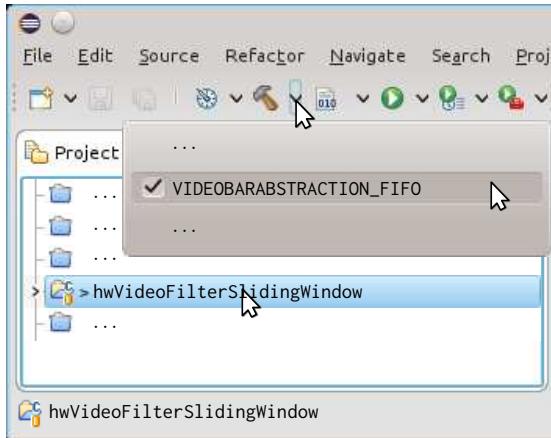
```
INFO: To connect to this hw_server instance use url: TCP:127.0.0.1:3121
```

```
100% 1MB 1.9MB/s 00:01
```

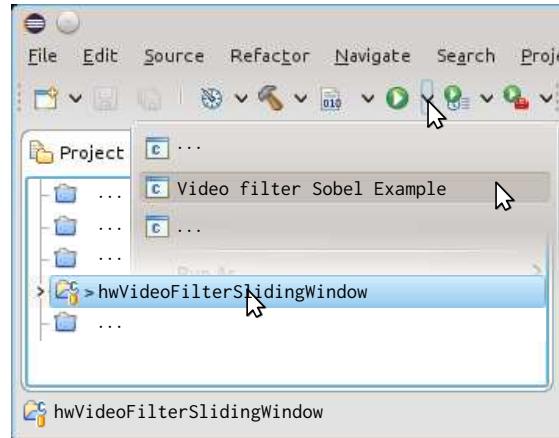
After this, the done led (LD10) should have turned green. Now, change your monitor configuration to display its VGA input. This should result in the test image from Figure 7 being displayed on the monitor.

1.6 Testing the Video Input

For testing purpose, you can use the Sobel video filter chain. A simulation of this filter chain can be started by following the steps in Figure 12 to compile and run the simulation. This should result in an output similar to the one shown in Figure 13.



(a) Compiling the Sobel filter at FIFO abstraction



(b) Running the simulation

Figure 12: Running the Sobel video filter chain at FIFO abstraction



Figure 13: Example output of the Sobel video filter chain

Compiling and running the simulation is not necessary to test on the Zybo board as you can use the pre-generated *sobel_wrapper.bit* bit file and the *ZyboInit.sh* script contained in the *hwVideo-FilterSlidingWindow* project to program the Zybo board.

To provide video input to the Zybo board, the USB camera shown in Figure 14a is *not connected to the Zybo board* but to your computer. Thus, if you want to test the Zybo board and how to run a video filter chain on it, you have to connect your computer's DisplayPort output (see Figure 14c) to the Zybo board's HDMI input by using a *DisplayPort to HDMI adapter* and a *HDMI cable* (see Figure 14b). The connections introduced in Section 1.5 are – of course – still required for video output to your monitor. Then, execute the *ZyboInit.sh* script contained in the *hwVideo-FilterSlidingWindow* project.



(a) USB camera for your computer to act as video source for the Zybo board



(b) DisplayPort to HDMI adapter and HDMI cable to connect computer and Zybo board to facilitate video input



(c) Connection for the DisplayPort to HDMI adapter on your computer

Figure 14: Additional cable, adapter, and camera required to feed video input to the Zybo board

```
[...test@codesigni165:~]$ cd ~/03_filter/workspace-filter
```

```
[...65:workspace-filter]$ cd hwVideoFilterSlidingWindow  
[...FilterSlidingWindow]$ ./ZyboInit.sh
```

Make sure that JP5 is in JTAG mode before you program the board. If JP5 is in the same position as shown in Figure 9, you can continue by pressing enter. After this, the done led (LD10) should have turned green. As a result, a new monitor should appear in your desktop session. You can check what video outputs your session knows of by using the `xrandr` command. The output below should resemble the state when nothing is connected to the DisplayPort connection, i.e., the *DP-0/DP-1 disconnected* lines below.

```
[...test@codesigni165:~]$ xrandr  
Screen 0: minimum 8 x 8, current 1920 x 1200, maximum 16384 x 16384  
DVI-I-0 disconnected  
DVI-I-1 connected primary 1920x1200+0+0 518mm x 324mm  
    1920x1200      60.00**+  
    ...  
DP-0 disconnected  
DP-1 disconnected
```

The output shows that your primary monitor is connected to *DVI-I-1* output of the graphics card and driven with resolution 1920x1200 and refresh rate of 60.00 Hz. After you have programmed the Zybo board, the output of the `xrandr` command should change to something similar to the one provided below.

```
[...test@codesigni165:~]$ xrandr  
Screen 0: minimum 8 x 8, current 3200 x 1200, maximum 16384 x 16384  
DVI-I-0 disconnected  
DVI-I-1 connected primary 1920x1200+0+0 518mm x 324mm  
    1920x1200      60.00**+  
    ...  
DP-0 connected 1280x720+1920+0 510mm x 290mm  
    1280x720      60.00**+  
    ...  
DP-1 disconnected
```

Now, the Zybo board is connected to the DisplayPort DP-0 and driven with a resolution of 1280x720 with a refresh rate of 60 Hz. You can use the following commands to switch the resolution of the output.

```
[...test@codesigni165:~]$ xrandr --output DP-0 --mode 1280x720  
[...test@codesigni165:~]$ xrandr --output DP-0 --mode 1024x768  
[...test@codesigni165:~]$ xrandr --output DP-0 --mode 800x600  
[...test@codesigni165:~]$ xrandr --output DP-0 --mode 640x480
```

Replace DP-0 with the output where the Zybo board is connected to the graphics card. This might differ with the used graphics card in your computer. In any case, resolutions above 1280x720 are not supported with the configuration the VideoSource IPB currently uses. Moreover, you can use commands like

```
[...test@codesigni165:~]$ xrandr --output DP-0 --left-of DVI-I-1  
[...test@codesigni165:~]$ xrandr --output DP-0 --right-of DVI-I-1  
[...test@codesigni165:~]$ xrandr --output DP-0 --above     DVI-I-1  
[...test@codesigni165:~]$ xrandr --output DP-0 --below     DVI-I-1
```

to place the video output to the Zybo board to the left, right, above, or below your primary monitor, e.g., the one connected to output *DVI-I-1*. Of course, you can also use the tools your desktop environment provides to manage the monitor output.

In any case, you can use the `mplayer` command given below to capture footage from the USB camera.

```
[... test@codesigni165:~]$ mplayer -framedrop tv://device=0 -tv width=1280:height=720
```

To close the video filter chain, simply move the window of the `mplayer` command to the position where your output to the Zybo board is located on your desktop and press **f** to enter *full screen mode* for the `mplayer` window.

1.7 Creating IPBs from SystemC

The Vivado tool needs IPBs for stitching them together into a system design from which a bit file can be derived. In this lab, a system design is represented by a *block design* in Vivado, e.g., as depicted in Figure 15. Vivado sources IPBs from Intellectual Property (IP) repositories. On

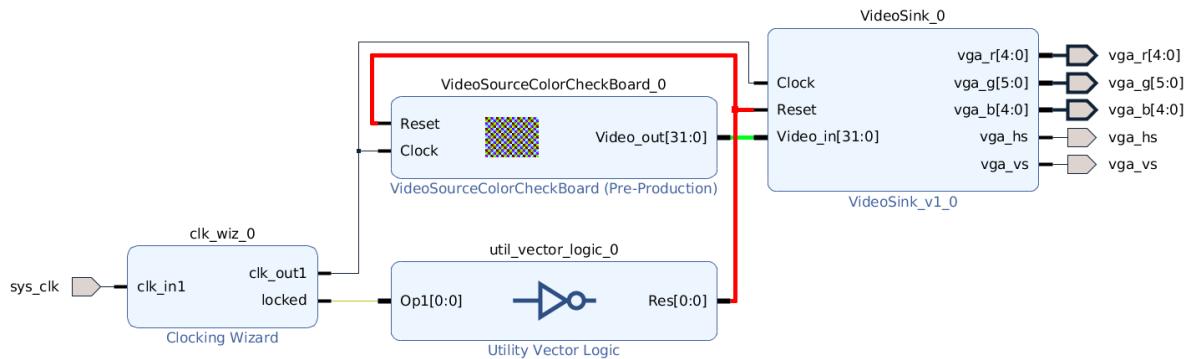


Figure 15: Vivado block design for the CCB test design

the one hand, there is a Vivado IP repository containing IPBs from Xilinx that are shipped with Vivado, e.g., the Clocking Wizard and Utility Vector Logic IPBs are from the Vivado repository. On the other hand, there are user IP repositories where custom IPBs can be sourced from. In this lab, the following two user IP repositories are used:

- `~/03_filter/ip_repo/digilent`
This repository contains IPBs from Digilent – the company producing the Zybo board – that are used internally by the IPBs from the hscd IP repository.
- `~/03_filter/ip_repo/hscd`
This repository contains the custom IPBs for this lab, e.g., the VideoSourceColorCeckBoard and VideoSink IPBs amongst others.

Next, you can examine what is already present in the hscd IP repository by opening the console and entering following commands:

```
[... test@codesigni165:~]$ cd ~/03_filter/ip_repo/hscd
[... t@codesigni165:hscd]$ ls -l ip
drwxr-xr-x 5 test stud 4096 Dec 19 13:02 VideoSink
drwxr-xr-x 5 test stud 4096 Dec 19 13:02 VideoSinkHDMI
drwxr-xr-x 6 test stud 4096 Mar  1 12:42 VideoSinkToMem
drwxr-xr-x 5 test stud 4096 Mar  1 12:42 VideoSource
drwxr-xr-x 4 test stud 4096 Mar  1 12:42 VideoSourceFromMem
```

As you can see, present are the IPBs VideoSink, VideoSinkHDMI, VideoSinkToMem, VideoSource, and VideoSourceFromMem. For these IPBs, the repository contains Very High Speed Integrated Circuits (VHSIC) Hardware Description Language (VHDL) implementations, e.g., as seen below:

```
[...t@codesigni165:hscd]$ ls -l ip/VideoSink/hdl
-rw-r--r-- 1 test stud 2479 Dec 19 13:02 VideoSink.vhd
```

Thus, the equivalent SystemC modules contained in the hwsim project are only simulation models that model the functionality realized by the real VHDL hardware implementations provided by this IP repository.

In contrast, the VideoSourceColorCeckBoard IPB is not initially present in the repository and will be generated using High Level Synthesis (HLS) from the VideoSource-ColorCheckBoard SystemC module present in the hwVideoSourceColorCheckBoard project. For this purpose, the **VivadoHLS** tool from Xilinx is employed. You can start VivadoHLS for the CCB module by selecting VivadoHLS Synthesis in the compile menu as shown in Figure 16. This will start HLS and then open the VivadoHLS GUI (see Figure 17) so that you can examine the synthesis results in terms of obtained timing, resource usage, throughput, etc. In this process three steps will be performed as can be seen in the console windows of Eclipse. First, HLS is started with `vivado_hls -f VivadoHLS.tcl` producing the following output in the Eclipse console window:

```
...
vivado_hls -f VivadoHLS.tcl
=====
Vivado(TM) HLS - High-Level Synthesis from C, C++ and SystemC
Version 2017.1
Build 1846317 on Fri Apr 14 19:19:38 MDT 2017
Copyright (C) 1986-2017 Xilinx, Inc. All Rights Reserved.
=====
```

Next, Xilinx Vivado is used to generate the VideoSourceColorCeckBoard IPB from the VHDL files generated by VivadoHLS for the VideoSourceColorCeckBoard SystemC module. For this purpose, the `run_ipack.tcl` script is run by Vivado. This step corresponds to the following output in the Eclipse console window:

```
vivado -notrace -mode batch -source run_ipack.tcl

***** Vivado v2017.1 (64-bit)
**** SW Build 1846317 on Fri Apr 14 18:54:47 MDT 2017
**** IP Build 1846188 on Fri Apr 14 20:52:08 MDT 2017
```

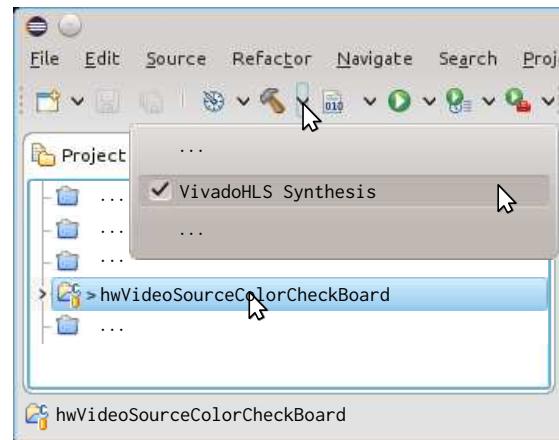


Figure 16: Starting VivadoHLS to synthesize the CCB into an IPB

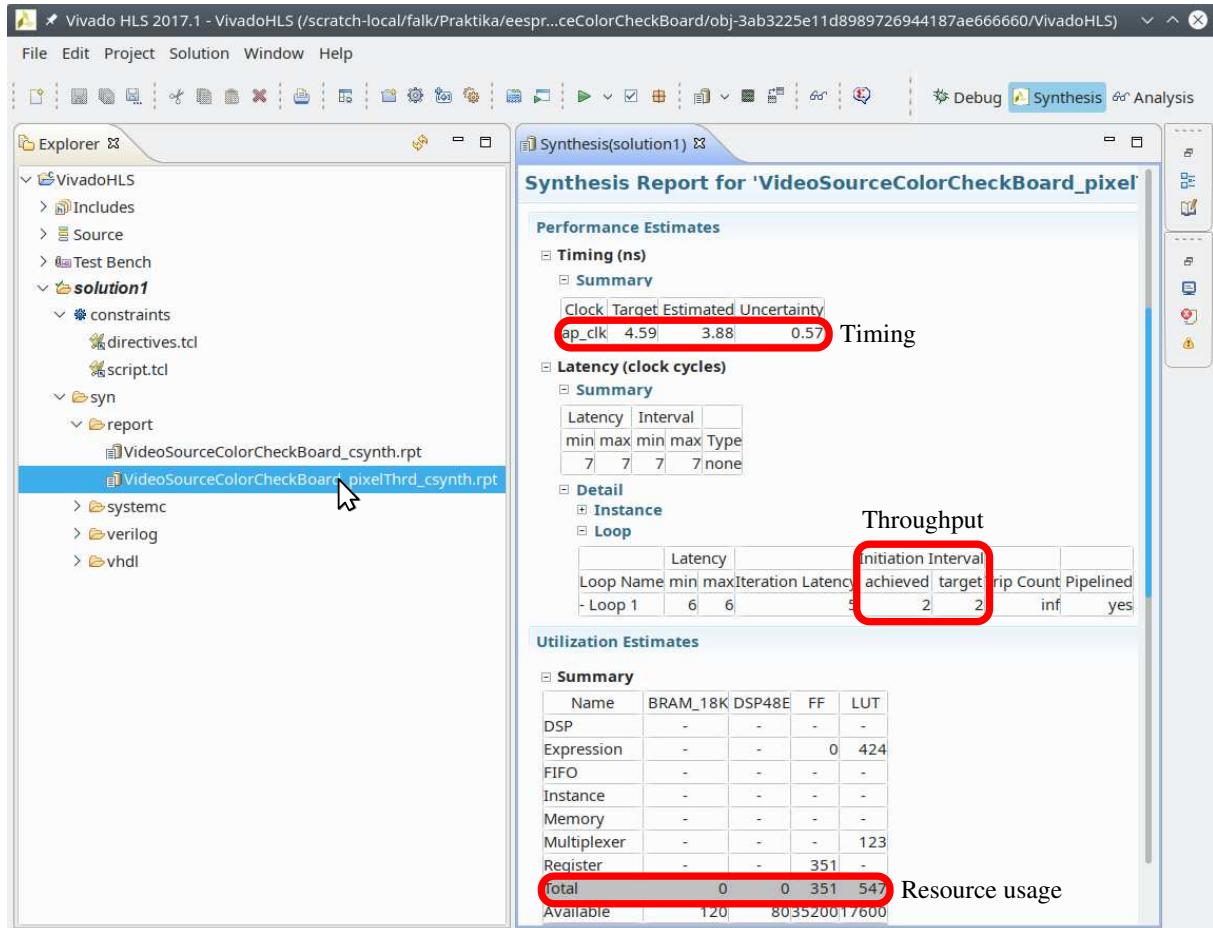


Figure 17: Synthesis report for the CCB IPB

** Copyright 1986–2017 Xilinx, Inc. All Rights Reserved.

```
source run_ipack.tcl -notrace
Found active high reset Reset
Found clock Clock with reset Reset
Found signal Video_out of width 32 and direction out
INFO: [IP_Flow 19-234] Refreshing IP repositories
INFO: [IP_Flow 19-1704] No user IP repositories specified
INFO: [IP_Flow 19-2313] Loaded Vivado IP repository '.../vivado/2017.1/data/ip'.
INFO: [Common 17-206] Exiting Vivado at Thu Mar 21 15:45:04 2019...
...
```

You can check that an IPB for VideoSourceColorCeckBoard has been generated by listing the contents of the hscd IP repository. For this purpose, you can enter the following commands in a console:

```
[...test@codesigni165:~]$ cd ~/03_filter/ip_repo/hscd
[...t@codesigni165:hscd]$ ls -l ip
drwxr-xr-x 5 test stud 4096 Dec 19 13:02 VideoSink
drwxr-xr-x 5 test stud 4096 Dec 19 13:02 VideoSinkHDMI
drwxr-xr-x 6 test stud 4096 Mar  1 12:42 VideoSinkToMem
drwxr-xr-x 5 test stud 4096 Mar  1 12:42 VideoSource
drwxr-xr-x 5 test stud 4096 Mar 21 15:45 VideoSourceColorCheckBoard
```

```
drwxr-xr-x 4 test stud 4096 Mar  1 12:42 VideoSourceFromMem
[...t@codesigni165:hscd]$ ls -l ip/VideoSourceColorCheckBoard/hdl/vhdl
-rw-r--r-- 1 test stud 29871 Mar 21 15:44 VideoSourceColorCheckBoard_pixelThrd.vhd
-rw-r--r-- 1 test stud 3271 Mar 21 15:44 VideoSourceColorCheckBoard.vhd
```

Finally, the VivadoHLS GUI is started with `vivado_hls -p VivadoHLS` so that you can examine the synthesis results in terms of obtained timing, resource usage, throughput, etc. This should result in the following output in the Eclipse console window:

```
vivado_hls -p VivadoHLS
=====
Vivado(TM) HLS - High-Level Synthesis from C, C++ and SystemC
Version 2017.1
Build 1846317 on Fri Apr 14 19:19:38 MDT 2017
Copyright (C) 1986-2017 Xilinx, Inc. All Rights Reserved.
=====
...
INFO: [HLS 200-10] Bringing up Vivado HLS GUI ...
```

After this, the VivadoHLS GUI will open. To see a report for the synthesis result of the `pixelThrd` process, you can open its report under **VivadoHLS** \triangleright **solution1** \triangleright **syn** \triangleright **report** by double clicking on **VideoSourceColorCeckBoard_pixelThrd_csynth.rpt**. This should display the *synthesis report* shown in the right half of Figure 17. There, three things are highlighted in red: (i) the Timing that indicates an estimation of VivadoHLS how fast the IPB can be clocked, e.g., here with a frequency of $\frac{1}{3.88\text{ns}}$, (ii) how many resources the IPB requires, e.g., here 351 Flip Flops (FFs) and 547 Look-Up Tables (LUTs), and (iii) the throughput of the loops in a process, e.g., here the sole while (true) $\{\dots\}$ loop achieving an Initiation Interval (II) of two clock cycles, i.e., the loop body is started every second clock cycle.

A more detailed look into the schedule of the `pixelThrd` process can be taken by opening the *analysis* perspective by clicking **Analysis** on the toolbar (see Figure 18) and selecting **Video-SourceColorCeckBoard_pixelThrd** in the *module hierarchy* opening the *Performance(solution1)* tab where the schedule of the `pixelThrd` process is depicted. There are five clock cycles depicted in the schedule C_0 to C_5 . What operation are run in what cycle can be examined in the *Operation\Control Step* column of the *Performance(solution1)* tab. The schedule of the while (true) $\{\dots\}$ loop is depicted in orange from clock cycle C_1 to C_5 . Hovering the mouse over this orange bar will trigger a pop-up with an overview of the scheduling of the loop. There, you will see the same details for the while (true) $\{\dots\}$ loop (**Loop1**) you found under **Detail** \triangleright **Loop** \triangleright **Loop1** in the *synthesis report* shown in the right half of Figure 17. To exemplify, the while (true) $\{\dots\}$ loop will start at every second clock cycle (II is two clock cycles) and that the loop body takes five clock cycles to complete (iteration latency is five clock cycles). Thus, executions of the loop will overlap, i.e., the loop is *pipelined*. How the loop is scheduled by VivadoHLS is controlled by *pragmas*, e.g., the pragma `#pragma HLS pipeline II=2` in the first line of the body of the while (true) $\{\dots\}$ loop instructs VivadoHLS to find a schedule of the loop where an execution of the loop body starts at every second clock cycle. **Note that VivadoHLS might not find such a schedule and, thus, you have to check if the instructions to VivadoHLS for the scheduling of loop have been satisfied for your designs.** An overview of all VivadoHLS pragmas can be found in `~/03_filter/documentation/ug1253-sdx-pragma-reference.pdf` in chapter **HLS Pragmas**.

1.8 System Integration and BIT File Creation

In this step, you can integrate the IPBs, you might have generated in the previous step, in a system that gets input images from HDMI and outputs the resulting video stream on the VGA

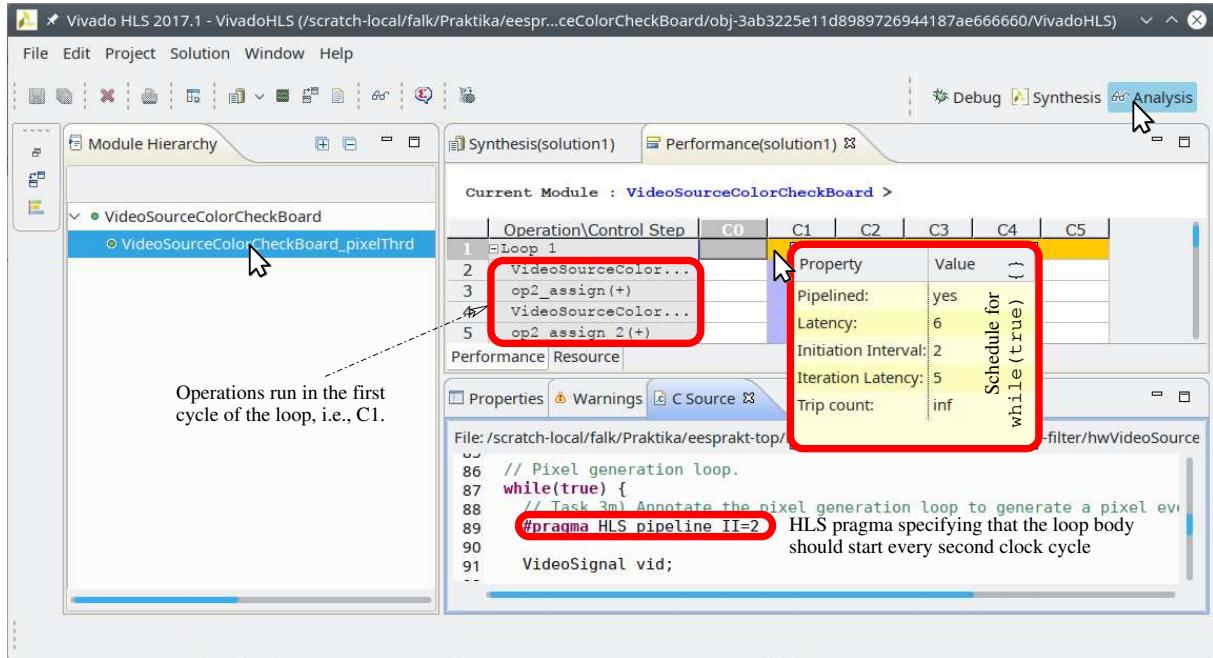


Figure 18: Examining the schedule of the `while (true) { ... }` loop in more detail

connector. Then, you can generate an FPGA *bitstream*, i.e., the configuration file for the Programmable Logic (PL) part of the Zynq-7000 as shown in Figure 10, and run the design on the Zybo board. For this purpose, you should use the Xilinx **Vivado** tool, an Electronic Design Automation (EDA) tool for *synthesis* and *implementation* of hardware designs.

To start, open a console and type `vivado`. Then, select the *workspace-vivado-filter.xpr* project file by following the steps in Figure 19. When opened, you can see that two block designs `ccb` and `sobel` (marked red in Figure 20) have been prepared for you as examples. However, both block designs are still disabled. We will first generate the bit file for the `ccb` block design. Thus, enable the block design by following the steps depicted in Figure 20. Next, open the *IP catalog* view in Vivado by clicking on **Window > IP Catalog** as shown in Figure 21. This should result in a view similar to the one in Figure 22. There, you can see the two *user IP repositories* (marked with a red box) as well as the *Vivado IP repository* containing IPBs from Xilinx that are shipped with Vivado. To notify Vivado the new IPBs might have appeared in the user IP repositories, you have to refresh the repositories as indicated in Figure 22. Next, open the `ccb` block design (see Figure 23) and run **Tools > Report > Report IP Status** as shown in Figure 24 to instruct Vivado to check if newer version of IPBs are available for the `ccb` block design. The result of this check is depicted in Figure 25. As can be seen, there is indeed a newer version of the `VideoSourceColorCheckBoard` present, i.e., the one you created via the HLS you started in Figure 16. The version of this IPB (in the red box) is determined from the date and time the HLS finished creating the IPB, e.g., the new IPB as depicted in Figure 25 has been created on the 21.3.2019 at 15:45 o'clock. Thus, you can check the version of IPBs used in a block design to determine if this block design already contains the latest changes you made to the SystemC sources of the IPB. To upgrade to this new version, check mark the IPB and click the **Upgrade Selected** button as depicted in Figure 25.

Next, you can examine the block design in Figures 15 and 26. As can be seen, the block design has inputs and outputs (marked with red boxes). Thus, information needs to be provided how these inputs and outputs should be connected to the Zybo board. Consulting the user guide `~/03_filter/documentation/ZYBO_RM_B_V6.pdf` for the Zybo board, we find following pertinent information duplicated in Figures 27 and 28.



Figure 19: Open the workspace-vivado-filter Vivado project

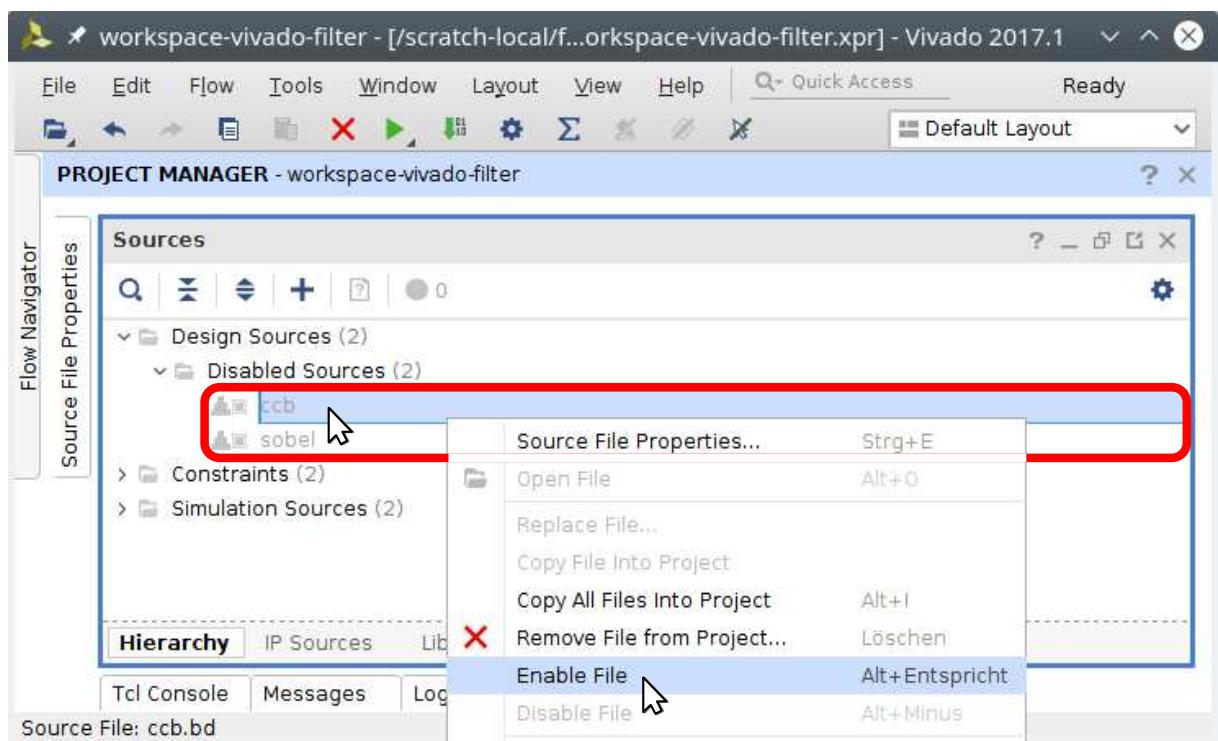


Figure 20: Enabling the block design ccb

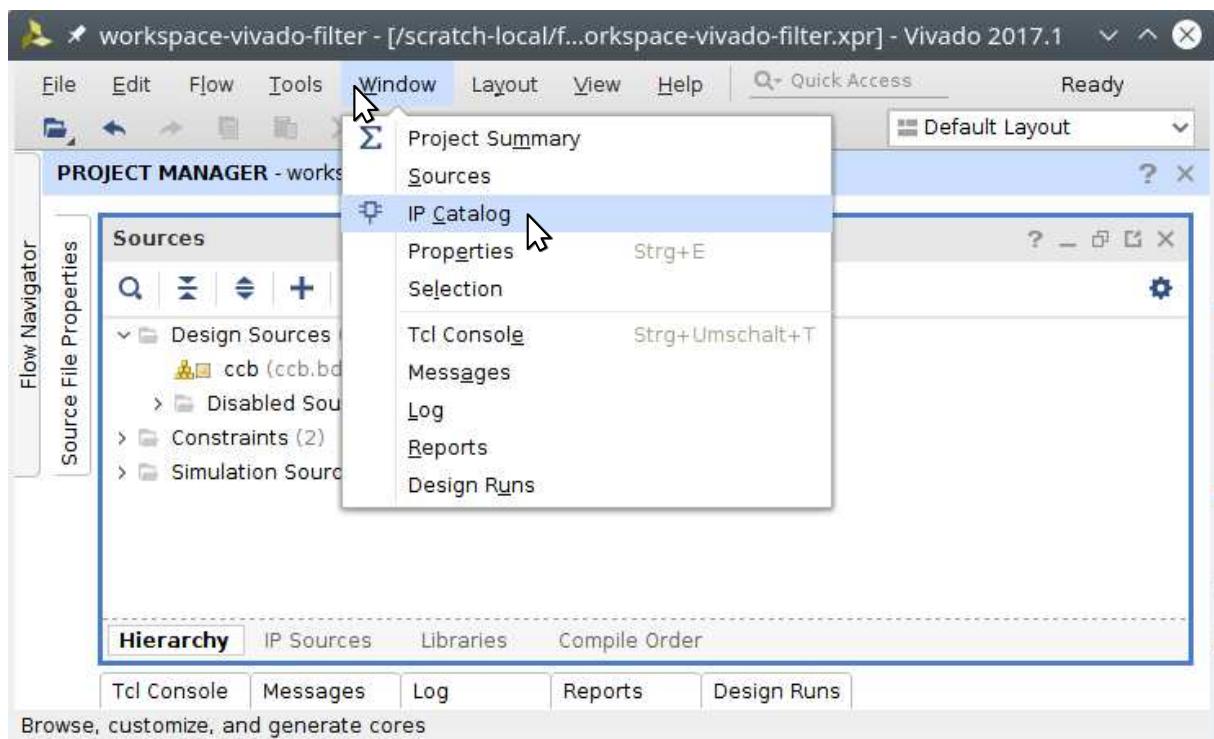


Figure 21: Open the *IP catalog* view in Vivado

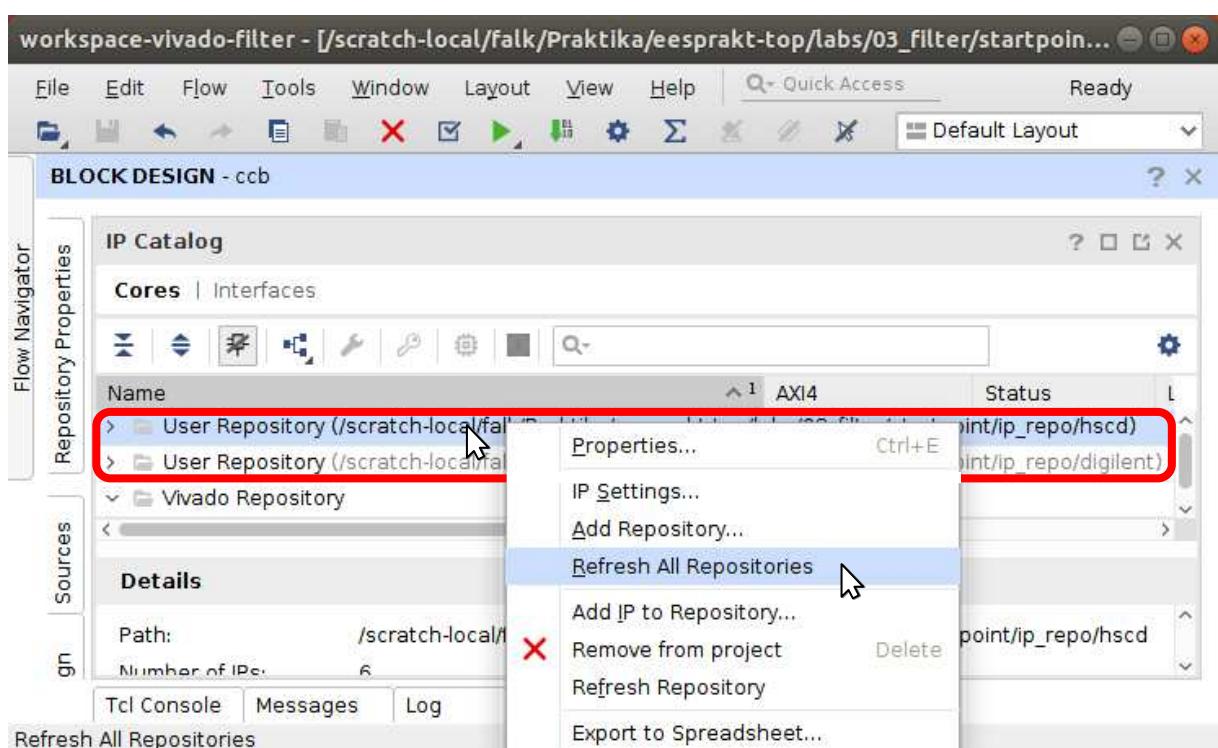


Figure 22: Refresh *IP repositories*

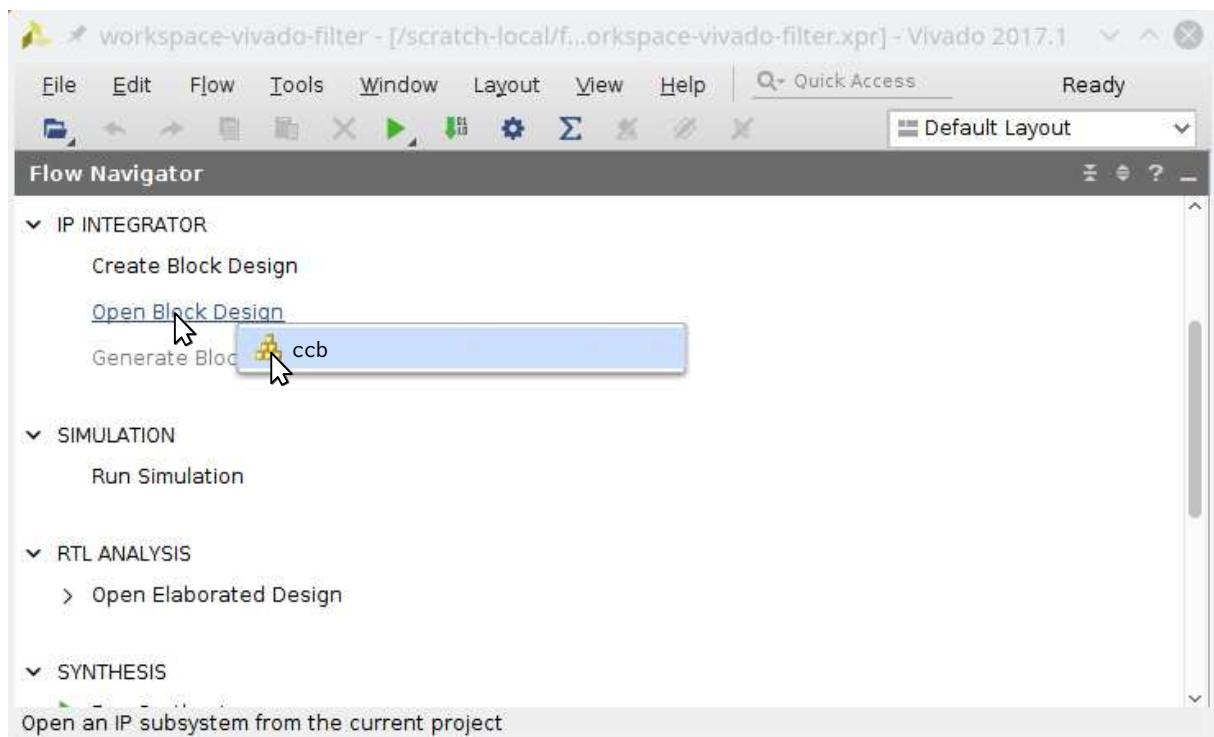


Figure 23: Open the ccb block design

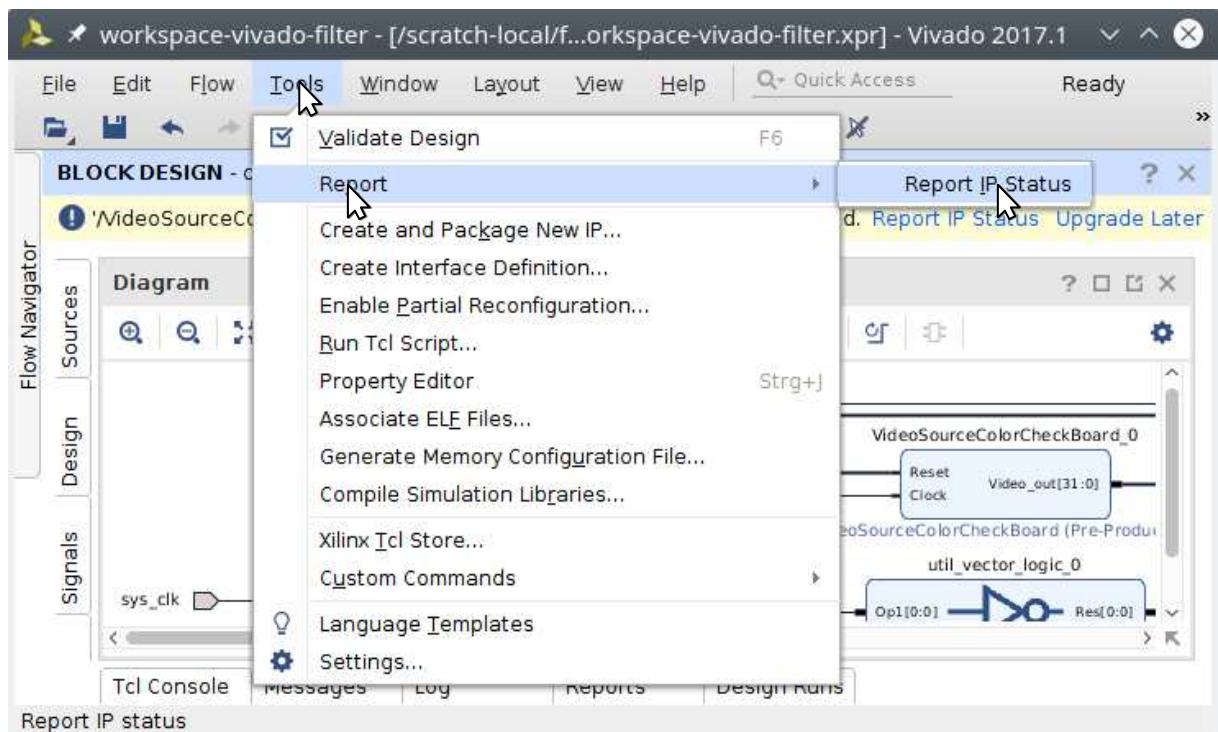


Figure 24: Check for newer version of IPBs by running *Report IP Status*

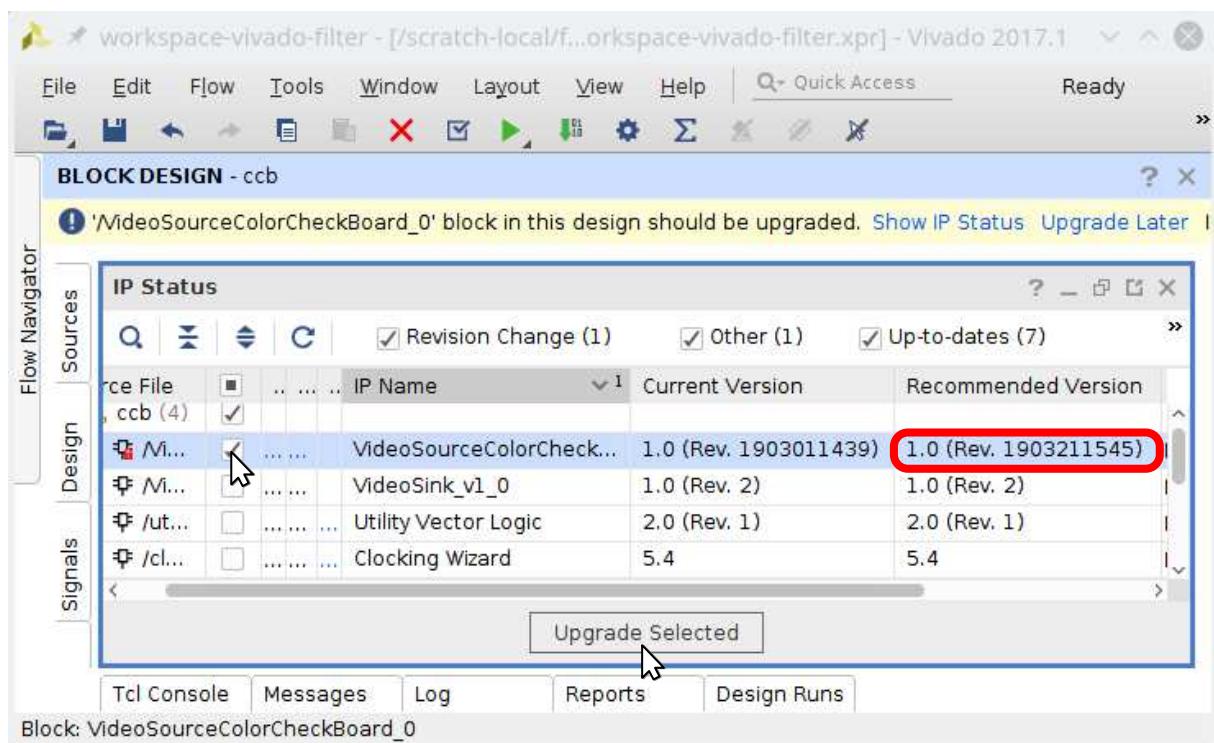


Figure 25: Upgrade IPBs to latest version

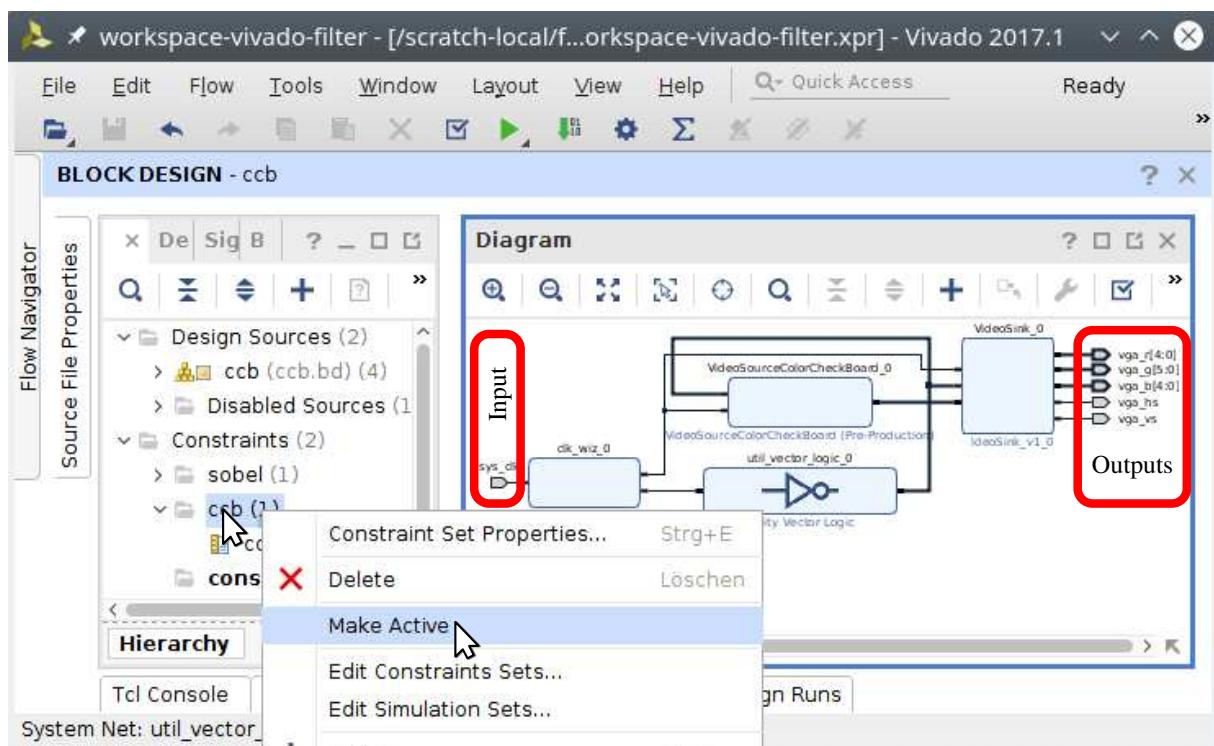


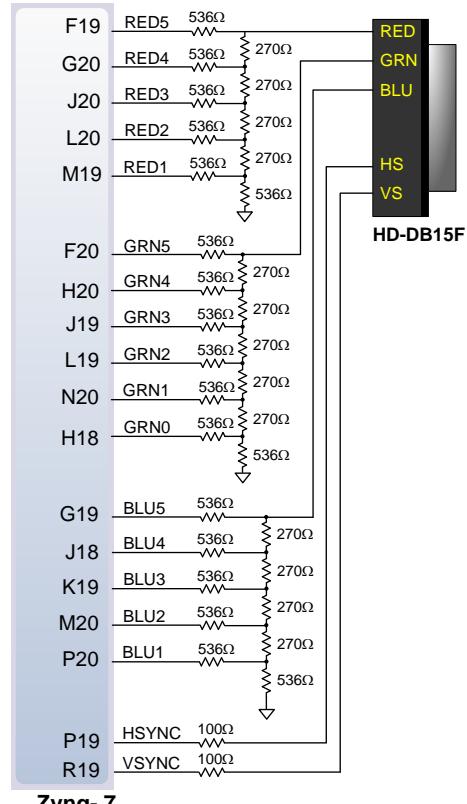
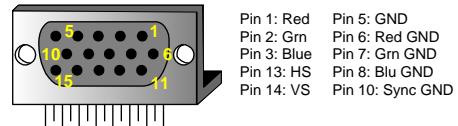
Figure 26: Activate the *ccb.xdc* constraints for the ccb block design

11 VGA Port

The ZYBO board uses 18 programmable logic pins to create an analog VGA output port. This translates to 16-bit color depth and two standard sync signals (HS – Horizontal Sync, and VS – Vertical Sync).

The digital-to-analog conversion is done using a simple R-2R resistor ladder². The ladder works in conjunction with the 75-ohm termination resistance of the VGA display to create 32 and 64 analog signal levels red, blue, and green VGA signals. This circuit, shown in Fig. 8, produces video color signals that proceed in equal increments between 0V (fully off) and 0.7V (fully on). With 5 bits each for red and blue and 6 bits for green, 65,536 ($32 \times 32 \times 64$) different colors can be displayed, one for each unique 16-bit pattern.

A video controller circuit must be created in programmable logic to drive the sync and color signals with the correct timing in order to produce a working display system



Zynq-7

Figure 8. ZYBO VGA circuit.

² http://en.wikipedia.org/wiki/Resistor_ladder

Copyright Digilent, Inc. All rights reserved.
Other product and company names mentioned may be trademarks of their respective owners.

Page 17 of 26

Figure 27: Information about the VGA connector in ZYBO_RM_B_V6.pdf

12 Clock Sources

The ZYBO provides a 50 MHz clock to the Zynq PS_CLK input, which is used to generate the clocks for each of the PS subsystems. The 50 MHz input allows the processor to operate at a maximum frequency of 650 MHz and the DDR3 memory controller to operate at a maximum of 525 MHz (1050 Mbps). The ZYBO Base System Design configures the PS to work properly with this input clock, and should be used as a reference when creating custom designs.

The PS has a dedicated PLL capable of generating up to four reference clocks, each with settable frequencies, that can be used to clock custom logic implemented in the PL. Additionally, The ZYBO provides an external 125 MHz reference clock directly to pin L16 of the PL. The external reference clock allows the PL to be used completely independently of the PS, which can be useful for simple applications that do not require the processor.

Figure 28: Information about clock sources in ZYBO_RM_B_V6.pdf

Thus, the sys_clk input has to be connected to pin L16 on the Zynq-7000 PSoC and the VGA outputs to the pins given in Figure 27. To specify this information, we have to provide constraints to the synthesis process. These constraints are contained in the *ccb.xdc* constraints file that can be activated by following the steps given in Figure 26. Opening the *ccb.xdc* constraints file will show the following relevant constraints:

```
...
#Clock signal
set_property -dict { PACKAGE_PIN L16 IOSTANDARD LVCMOS33 } [get_ports { sys_clk }];
create_clock -add -name sys_clk -period 8.00 -waveform {0 4} [get_ports { sys_clk }];
...
#VGA Connector
set_property -dict { PACKAGE_PIN M19 IOSTANDARD LVCMOS33 } [get_ports { vga_r[0] }];
set_property -dict { PACKAGE_PIN L20 IOSTANDARD LVCMOS33 } [get_ports { vga_r[1] }];
set_property -dict { PACKAGE_PIN J20 IOSTANDARD LVCMOS33 } [get_ports { vga_r[2] }];
set_property -dict { PACKAGE_PIN G20 IOSTANDARD LVCMOS33 } [get_ports { vga_r[3] }];
set_property -dict { PACKAGE_PIN F19 IOSTANDARD LVCMOS33 } [get_ports { vga_r[4] }];
set_property -dict { PACKAGE_PIN H18 IOSTANDARD LVCMOS33 } [get_ports { vga_g[0] }];
set_property -dict { PACKAGE_PIN N20 IOSTANDARD LVCMOS33 } [get_ports { vga_g[1] }];
set_property -dict { PACKAGE_PIN L19 IOSTANDARD LVCMOS33 } [get_ports { vga_g[2] }];
set_property -dict { PACKAGE_PIN J19 IOSTANDARD LVCMOS33 } [get_ports { vga_g[3] }];
set_property -dict { PACKAGE_PIN H20 IOSTANDARD LVCMOS33 } [get_ports { vga_g[4] }];
set_property -dict { PACKAGE_PIN F20 IOSTANDARD LVCMOS33 } [get_ports { vga_g[5] }];
set_property -dict { PACKAGE_PIN P20 IOSTANDARD LVCMOS33 } [get_ports { vga_b[0] }];
set_property -dict { PACKAGE_PIN M20 IOSTANDARD LVCMOS33 } [get_ports { vga_b[1] }];
set_property -dict { PACKAGE_PIN K19 IOSTANDARD LVCMOS33 } [get_ports { vga_b[2] }];
set_property -dict { PACKAGE_PIN J18 IOSTANDARD LVCMOS33 } [get_ports { vga_b[3] }];
set_property -dict { PACKAGE_PIN G19 IOSTANDARD LVCMOS33 } [get_ports { vga_b[4] }];
set_property -dict { PACKAGE_PIN P19 IOSTANDARD LVCMOS33 } [get_ports { vga_hs }];
set_property -dict { PACKAGE_PIN R19 IOSTANDARD LVCMOS33 } [get_ports { vga_vs }];
...

```

Next, you have to create a HDL wrapper around the ccb block design by following the steps in Figure 29, set this wrapper as the top module for synthesis as depicted in Figure 30, and subsequently start the bit file generation as shown in Figure 31. Then, wait for synthesis to complete and open the hardware manager as shown in Figure 32. Finally, you have to program the Zynq-7000 PSoC by following the steps in Figure 33. This should again result in the test image depicted in Figure 7 to be displayed on your monitor. If not, test again with the instructions given in Section 1.5 to check if your board connections are correct.

If you also want to generate the bit file for the sobel block design, start *Vivado HLS Synthesis* for the *hwVideoFilterSlidingWindow* project. Then, disable the ccb block design and enable the sobel block design instead, i.e., the step in Figure 20 but for the sobel block design. Moreover, activate the *sobel.xdc* constraint file, i.e., the step in Figure 26 but for the *sobel* constraint set. Subsequently, open the sobel block design, refresh all IP repositories, run **Tools > Report > Report IP Status** and upgrade the *VideoFilterSlidingWindow* IPB with the one you just created via HLS. Next, create a HDL wrapper around the sobel block design, set the wrapper as top module for synthesis, start bit file generation, and program the device with the resulting bit file.

For further information, please consult the guide *~/03_filter/documentation/ug896-vivado-ip.pdf* for general help on Vivado.

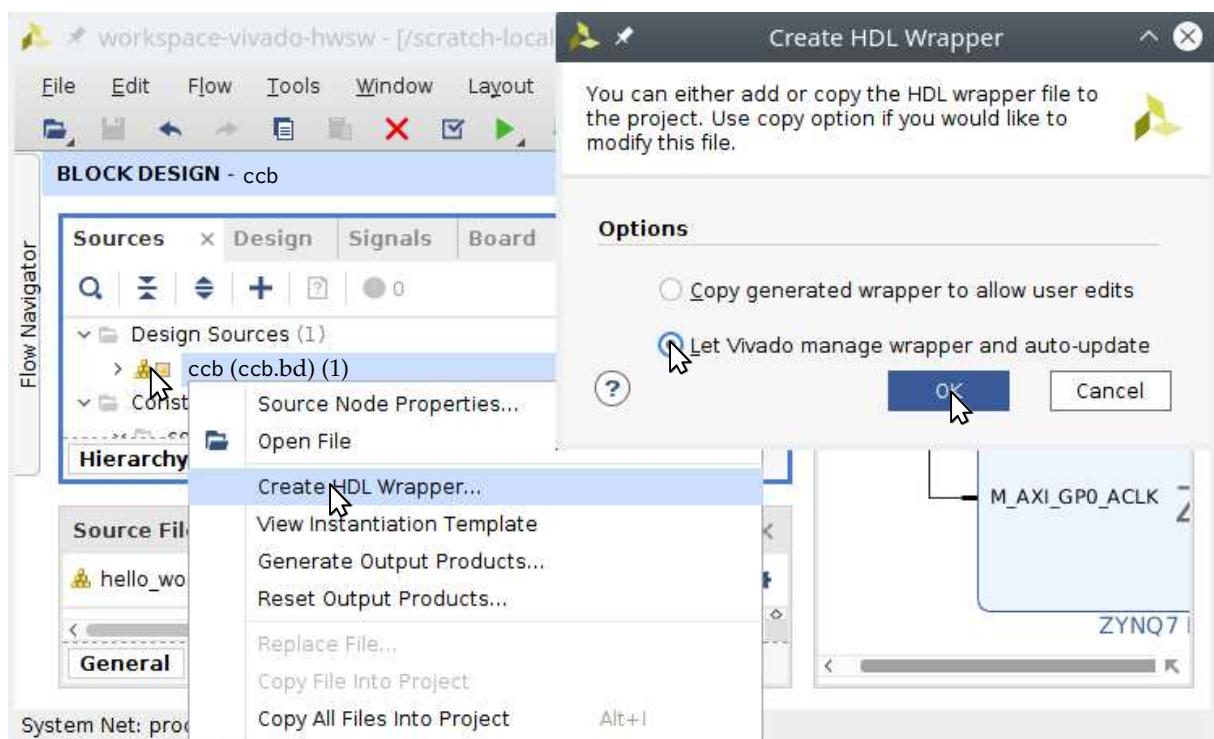


Figure 29: Create a HDL wrapper around the ccb block design

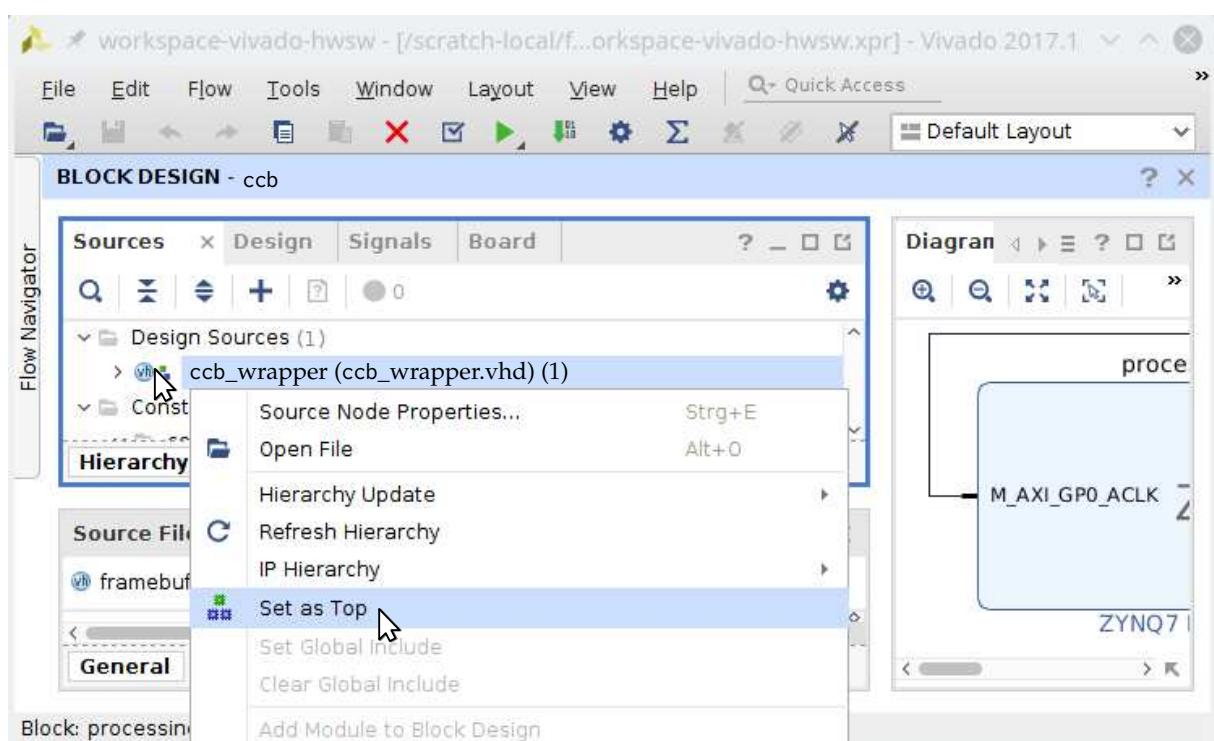


Figure 30: Set the wrapper as top module for synthesis

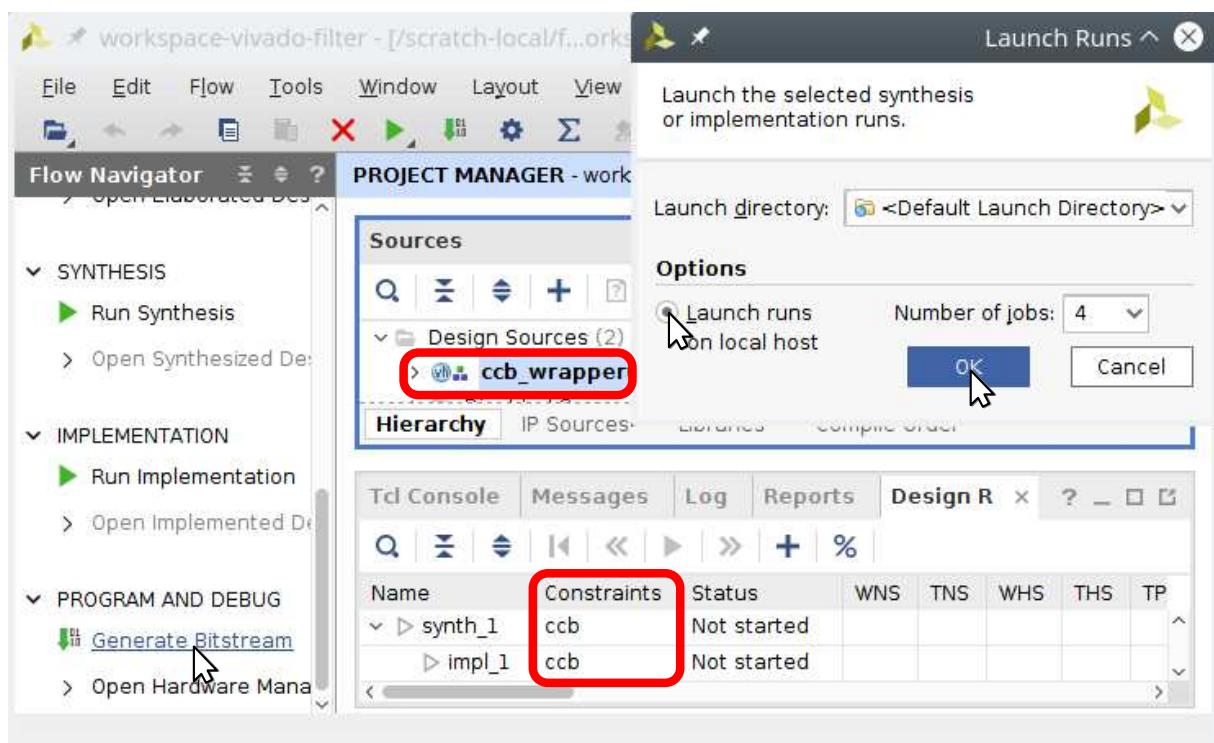


Figure 31: Start bit file generation for ccb block design

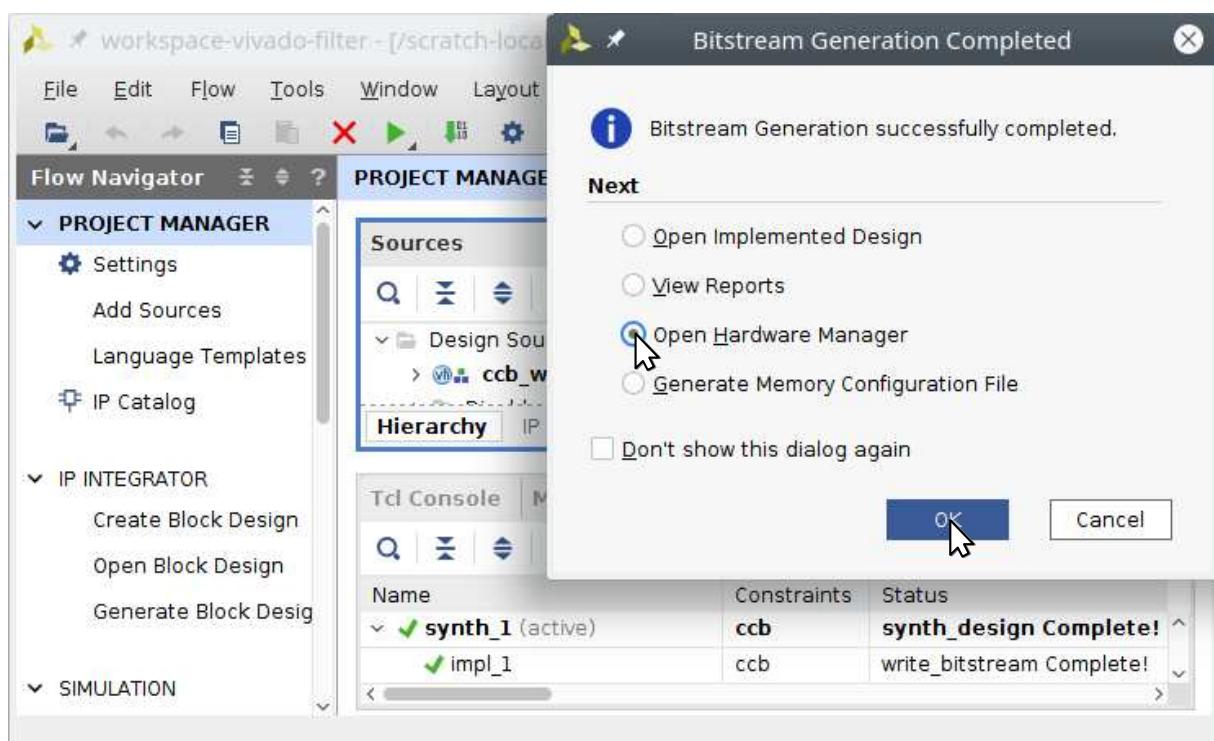


Figure 32: Wait for synthesis to complete and open the hardware manager

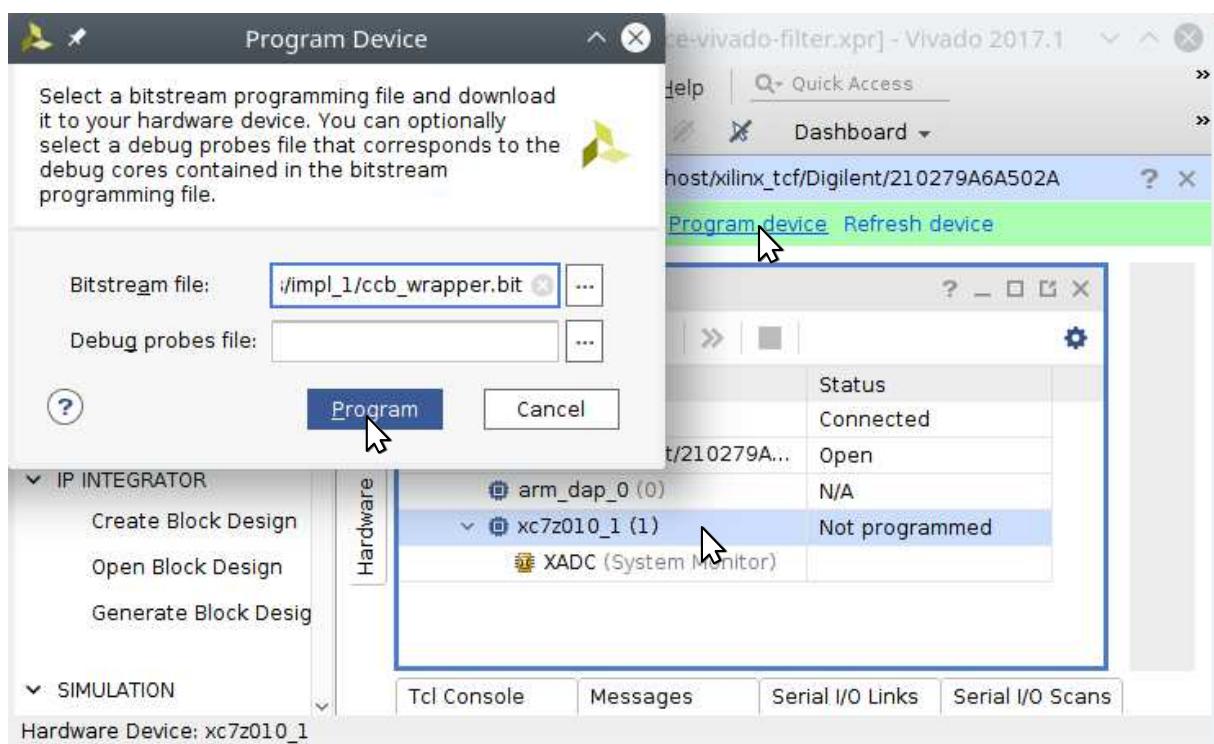


Figure 33: Program the Zynq-7000 PSoC with the bit file for the ccb block design

Task 1 (Simple Point Filter Implementation)

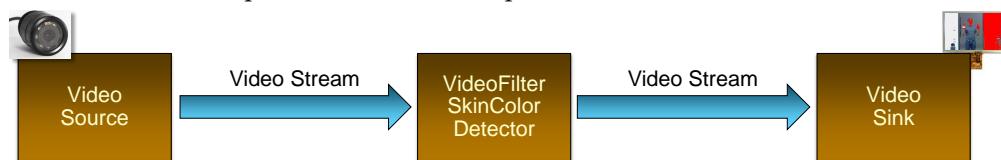
Project: hwVideoFilterSkinColorDetector

Files: *src/module/cpp/VideoFilterSkinColorDetector.hpp*
src/module/cpp/VideoFilterSkinColorDetector.cpp

In the following, we will realize a simple skin color detection filter (see Figure 34b for the simple filter chain). This filter is called a *point filter* because it operates only on one pixel to determine



(a) Expected result for a simple skin color detection filter



(b) Video filter chain for a simple skin color detection design

Figure 34: Architecture and expected result for a simple skin color detection filter

an output pixel. For this purpose, we open the project `hwVideoFilterSkinColorDetector`. The `hwsim` project should be opened automatically as a referenced project.

All filters can be compile in two different modes: (i) FIFO abstraction and (ii) RTL abstraction level. These levels are guarded by `#if VIDEOBARABSTRACTION == VIDEOBARABSTRACTION_FIFO` and `#if VIDEOBARABSTRACTION == VIDEOBARABSTRACTION_RTL`. In the following, you will implement functionality for the FIFO abstraction level.

- Define the Reset input port for the VideoFilterSkinColorDetector module. This port must be connectable to a Boolean (bool) signal. Take care to also name the signal accordingly in the constructor.
- Define the Video_in and Video_out FIFO ports for the module. Take care to also name these ports accordingly in the constructor.
- Register the pixelThrd method as a threaded process (SC_THREAD) of the module. Take care to specify that the VideoFilterSkinColorDetector module can have processes. This process will simply forward the data. Thus, check that the module can now be used as a pass through filter module at the FIFO abstraction level. For this purpose, first compile the hwsim project at FIFO abstraction level, i.e., VIDEOBARABSTRACTION_FIFO as seen in Figure 4a. Then, compile this project at FIFO abstraction level (see Figure 35a) and, finally, start the simulation (see Figure 35b)
- Change the pixelThrd process to detect skin color pixels. Mark these pixels by setting the skin color bit and turning the pixel bright red. Check that you have correctly realized skin color detection (cf. Figure 35 to run and Figure 34a for the expected result).

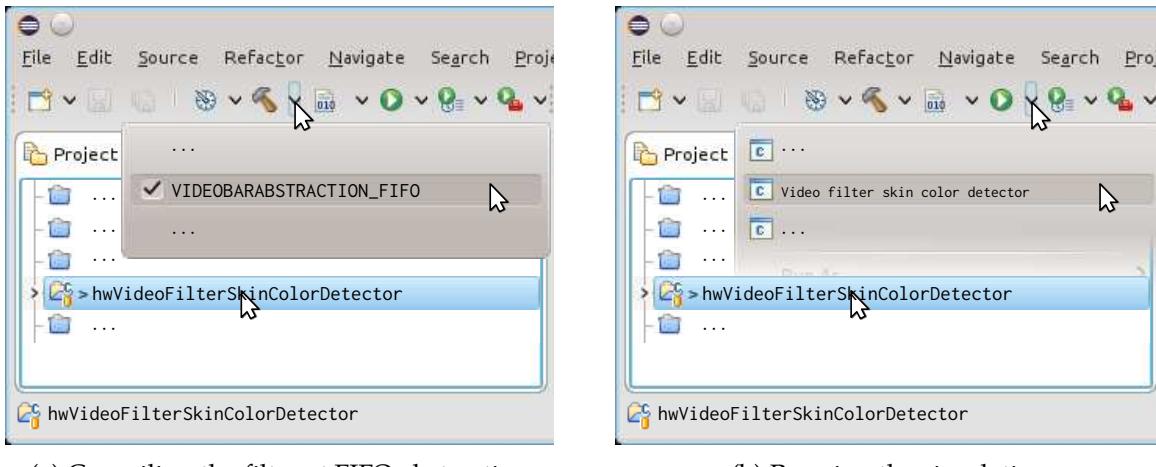


Figure 35: Check the functionality of your skin color filter at FIFO level.

In the following, you will switch the filter from FIFO abstraction to RTL abstraction. However, please guarantee that the FIFO abstraction level still remains functional. Hence, you have to define the Clock and the Video_in ports as well as the Video_out port to be signal input, respectively, output ports in case of RTL abstraction. The pixelThrd process has to be adjusted accordingly and converted to a clocked and threaded process.

- Define the Clock input port. This port must be connectable to a Boolean signal.
- Define the Video_in and Video_out signal ports for the module.
- Change the pixelThrd process to a clocked and threaded process (SC_CTHREAD) for RTL abstraction level. The pixelThrd process should be sensitive to the rising edge of the Clock signal.
- Define the Reset signal as an active-high reset for the pixelThrd process.
- Change the pixelThrd process to also support SC_CTHREAD mode as well as reading and writing from the signal ports Video_in and Video_out. Then, check that the module can perform skin color detection at RTL abstraction level. For this purpose, first compile the hwsim project at RTL abstraction level. Then, compile this project at RTL abstraction level (see Figure 36a) and, finally, start the simulation (see Figure 35b)

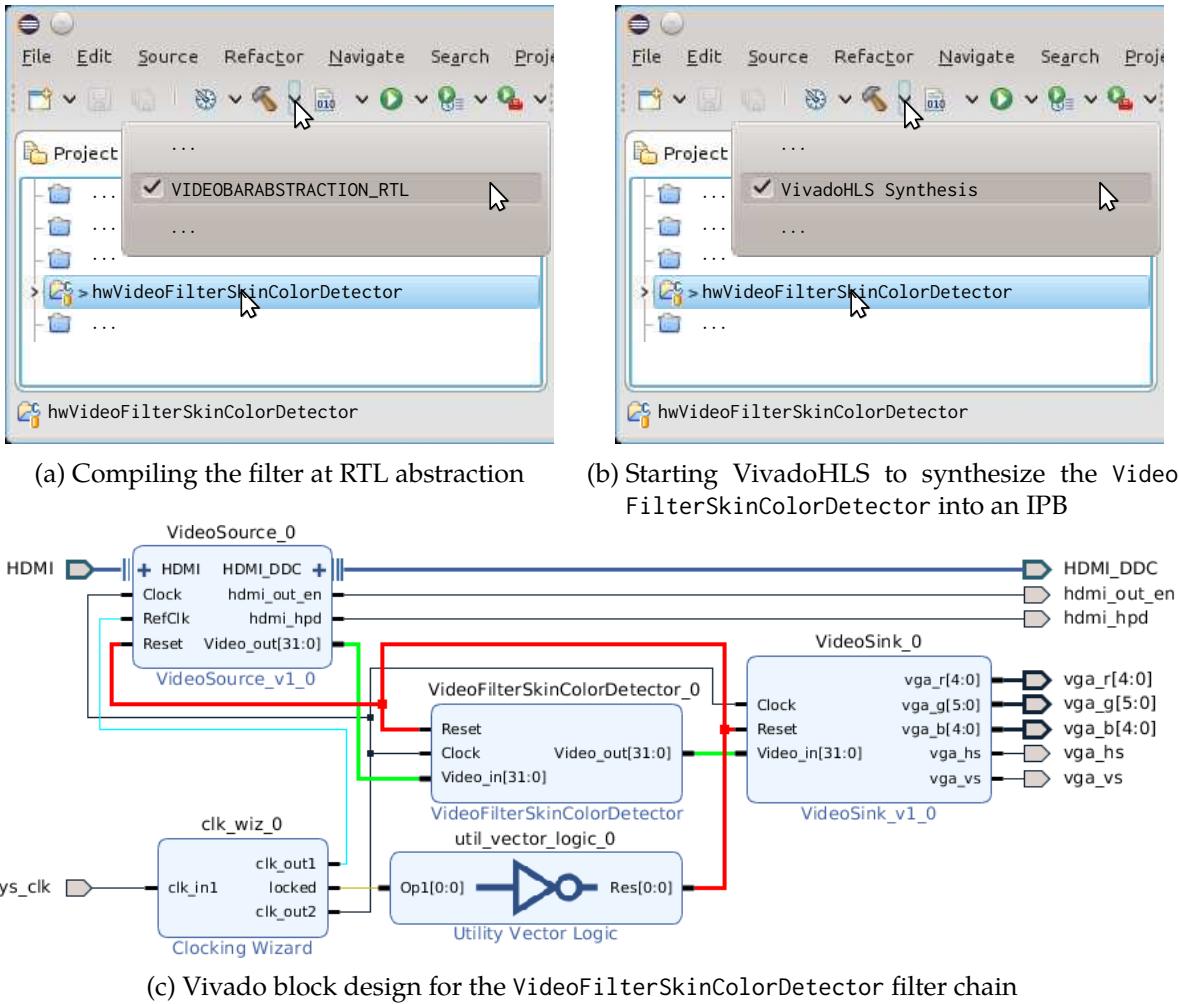


Figure 36: Realizing the skin color detection filter chain on the Zybo board

Next, you will realize the skin color detection filter chain given in Figure 34b as the skcd block design on the Zybo board. To distinguish a module at RTL abstraction level from one undergoing synthesis via VivadoHLS, the define `__SYNTHESIS__` will be defined in the HLS *synthesis case*.

- j) First, add `#pragma HLS pipeline II=1` as the first statement inside the `while (true) { ... }` loop. This *pragma* instructs VivadoHLS to try to find a schedule for the loop that starts a loop body execution every clock cycle. However, this might fail depending on the code in the loop body.
- k) Then, start VivadoHLS synthesis for the VideoFilterSkinColorDetector filter as shown in Figure 36b. Check the *analysis* perspective (see Figure 18) to ensure that the `while (true) { ... }` loop really has a schedule that starts a loop body execution every clock cycle. You also might run a simulation again (see Figure 35b) to check that the synthesized result works.
- l) Next, start Vivado and refresh the *IP repositories* as shown in Figure 22. This is needed for Vivado to pick up the VideoFilterSkinColorDetector IPB you just created via HLS.
- m) Then, create the skcd block design (see Figure 37).
- n) Add the IPBs “VideoFilterSkinColorDetector”, “VideoSource”, “VideoSink”, “Clocking Wizard”, and “Utility Vector Logic” to the empty skcd block design (see Figure 38). This should result in a block design similar to the one depicted in Figure 39. However, the IPBs

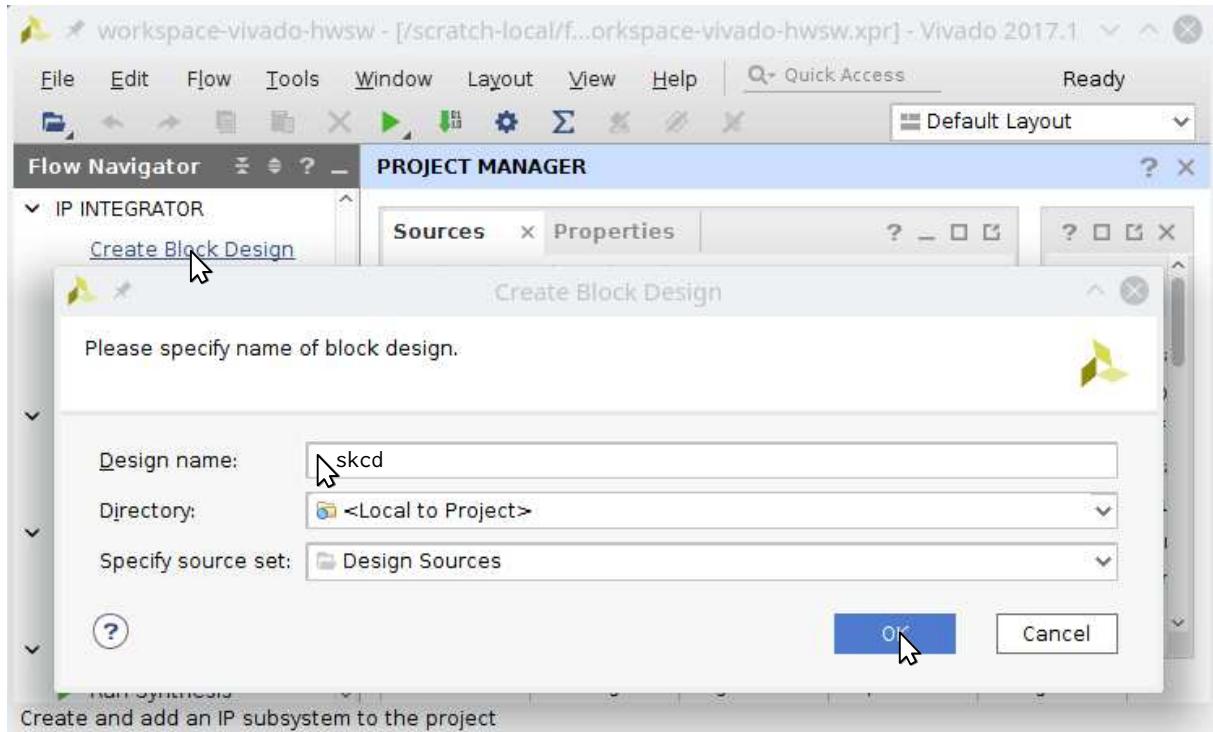


Figure 37: Create the skcd block design

surrounded by the red boxes need to be re-customized.

- o) Thus, double click on the “Utility Vector Logic” IPB to bring up its customization dialog as shown in Figure 40. Change **C_SIZE** to “1” and **C_OPERATION** to “not” to change the “Utility Vector Logic” IPB to be a one bit inverter.
- p) The block design needs two different clocks. It requires a 200 MHz clock to be connected to the RefClk input port of the VideoSource IPB as well as a 216 MHz clock at which your IPBs will run and, thus, has to be connected to the Clock input ports of the IPBs. Available is the 125 MHz clock input pin L16 described in Figure 28. Thus, you have to customize the “Clocking Wizard” IPB to convert from this 125 MHz clock to the 200 and 216 MHz clocks. For this purpose, you customize the input of the “Clocking Wizard” to be 125 MHz as shown in Figure 41. Then, switch to the **Output Clocks** tab (see Figure 42) and select a 200 MHz clock as output on port clk_out1 and a 216 MHz clock as output on port clk_out2. Please check that the “Clocking Wizard” will really generate these clock frequencies by checking the **Actual Output Freq** column (boxed red in Figure 42). Moreover, disable the reset input pin.
- q) The locked output port of the “Clocking Wizard” will be low until the output clocks on clk_out1 and clk_out2 become stable whereas it will turn high. Thus, we have to invert the locked signal to derive an active high reset signal. Hence, connect the reset derived by inverting the locked output port of the “Clocking Wizard” to the Reset input ports of the IPBs as shown in Figure 43. Connections can be made by clicking and dragging an output port of an IPB to an input port of an IPB. Then, connect clk_out1 to the RefClk input port of the VideoSource IPB as well as clk_out2 to the Clock input ports of the other IPBs.
- r) After connecting these ports, the block design will look as shown in Figure 44. Subsequently, the remaining input and output ports have to be *made external* as these ports represent the interface to the Zybo board. Right click on the remaining input and output

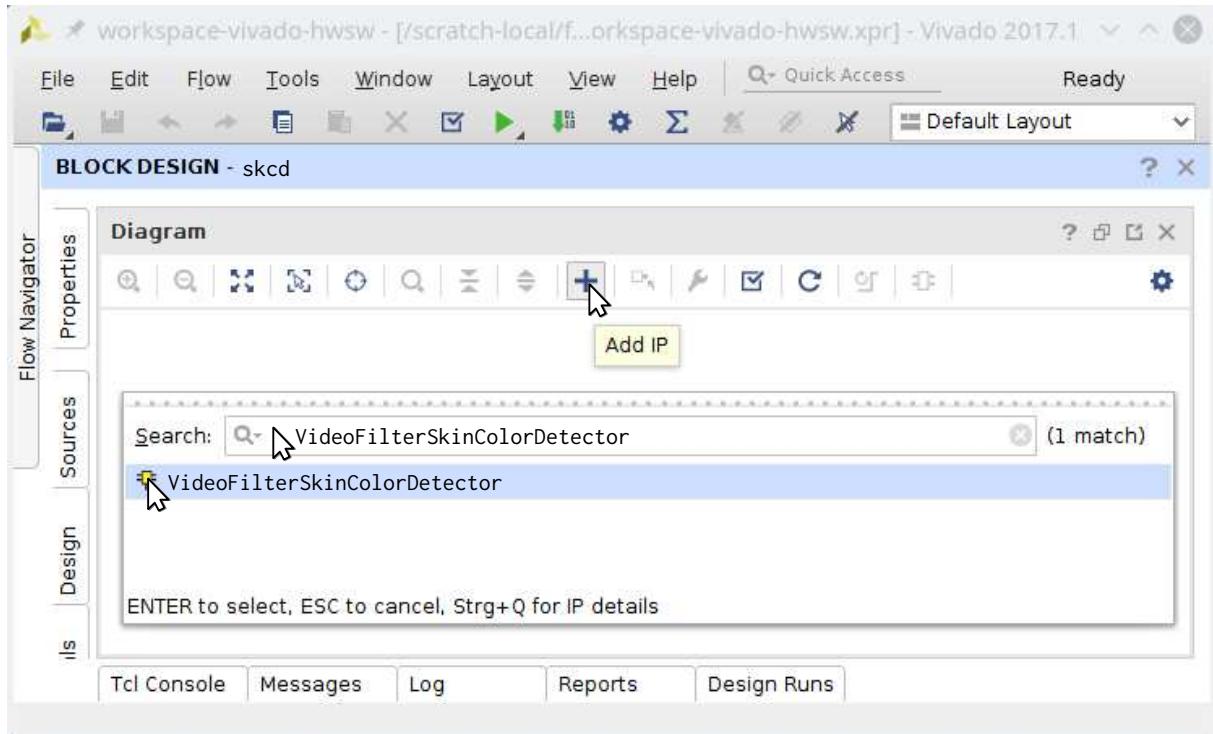


Figure 38: Add the IPB “VideoFilterSkinColorDetector” to the skcd block design

ports and select **Make External** from the pop-up menu. This should result in a block design similar to the one depicted in Figure 15 with the exception that the sys_clk port is still named clk_in1.

- s) Thus, rename (see Figure 45) the clk_in1 input port to sys_clk by clicking on the port and changing its name in the **Name** text field of the **External Port Properties** window. Then, save the block design by pressing **Ctrl+S**.
- t) Activate the **constrs_1** constraint set by adapting the steps from Figure 26. This constraint set contains the *ZYBO_Master.xdc* *xdc constraint file* (its absolute path is *~/03_filter/workspace-vivado-filter/ZYBO_Master.xdc*). This file contains commented out constraints for a significant fraction of the periphery the Zybo board has connected to the Zynq-7000 PSoC.
- u) Enable the required constraints in the *ZYBO_Master.xdc* to give the pin location for each input and output port of the skcd block design. There are three groups of input and output ports in the block design: (i) the clock input sys_clk, (ii) the HDMI video input interfaces, and (iii) the VGA output port. First, let us consider the sys_clk port. We know (see Figure 28), that this port must be connected to the L16 pin of the Zynq-7000 PSoC. Thus, enable the following constraints in the *ZYBO_Master.xdc* constraint file.

```
# Clock signal
set_property -dict { PACKAGE_PIN L16 IO_STANDARD LVCMOS33 } [get_ports { sys_clk }];
create_clock -add -name sys_clk -period 8.00 -waveform {0 4} [get_ports { sys_clk }];
```

This, will place the sys_clk port on the L16 pin and tell synthesis that this is a 125MHz clock input, i.e., $\frac{1}{8\text{ns}}$ is 125MHz. Next, there are the *HDMI video input interfaces*, i.e., HDMI interface, HDMI_DDC interface, hdmi_out_en output port, and hdmi_hpd output port. Behind the HDMI interface, there are the following ports and pin placements:

```
# HDMI interface
set_property -dict { PACKAGE_PIN H17 IO_STANDARD TMDS_33 } [get_ports HDMI_clk_n];
set_property -dict { PACKAGE_PIN H16 IO_STANDARD TMDS_33 } [get_ports HDMI_clk_p];
```

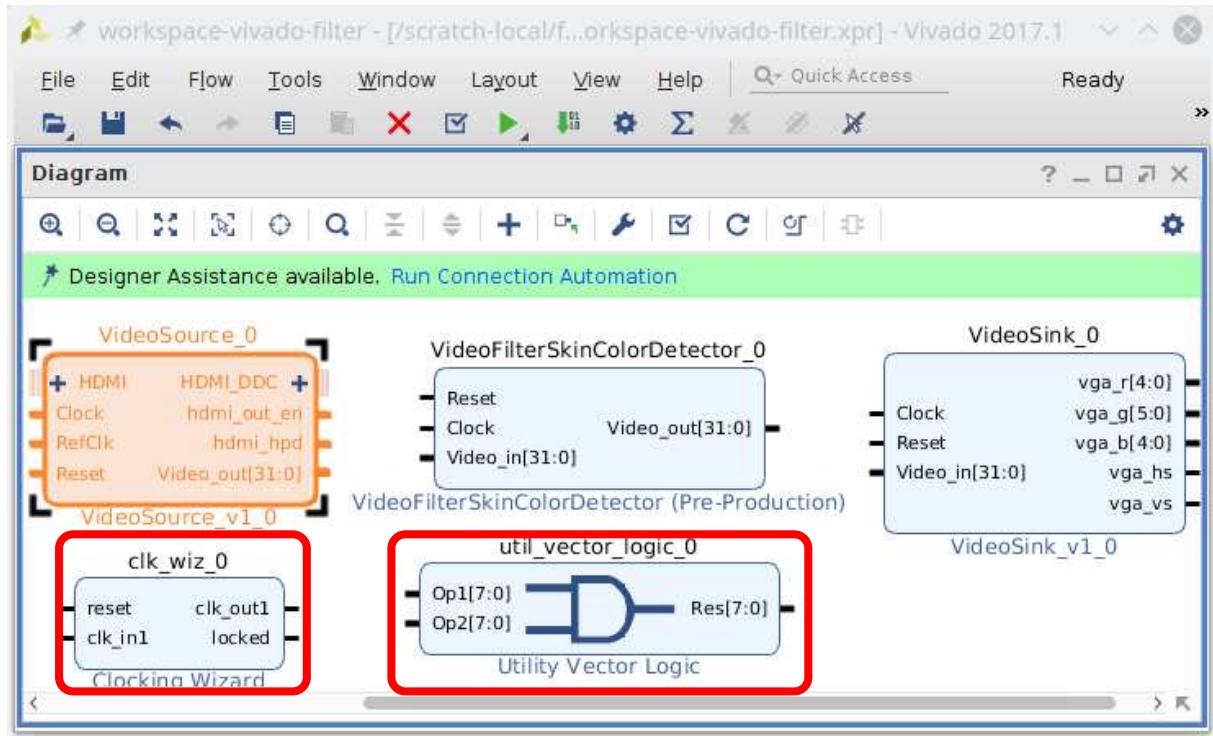


Figure 39: All IPBs for the skcd block design

```
set_property -dict { PACKAGE_PIN D20 IOSTANDARD TMDS_33 } [get_ports { HDMI_data_n[0] }];
set_property -dict { PACKAGE_PIN D19 IOSTANDARD TMDS_33 } [get_ports { HDMI_data_p[0] }];
set_property -dict { PACKAGE_PIN B20 IOSTANDARD TMDS_33 } [get_ports { HDMI_data_n[1] }];
set_property -dict { PACKAGE_PIN C20 IOSTANDARD TMDS_33 } [get_ports { HDMI_data_p[1] }];
set_property -dict { PACKAGE_PIN A20 IOSTANDARD TMDS_33 } [get_ports { HDMI_data_n[2] }];
set_property -dict { PACKAGE_PIN B19 IOSTANDARD TMDS_33 } [get_ports { HDMI_data_p[2] }];
```

Behind the HDMI_DDC interface, there are the following ports and pin placements:

```
# HDMI_DDC interface
set_property -dict { PACKAGE_PIN G17 IOSTANDARD LVC MOS33 } [get_ports hdmi_ddc_scl_io];
set_property -dict { PACKAGE_PIN G18 IOSTANDARD LVC MOS33 } [get_ports hdmi_ddc_sda_io];
```

Here, the pins G17 and G18 are input/output pins, e.g., they can act as an input as well as an output. Thus, each pin is controlled by three signals <prefix>_i, <prefix>_o, and <prefix>_t, e.g., hdmi_ddc_scl_i, hdmi_ddc_scl_o, and hdmi_ddc_scl_t. There, the signal <prefix>_t controls if the pin is an input or an output. If it is an output, then the signal <prefix>_o will drive the pin. Otherwise, the signal <prefix>_i will carry the signal from the pin while it is acting as an input. Then, we need to place the conventional output ports hdmi_out_en and hdmi_hpd needed to control the HDMI connector. To place these, use the following pin placement constraints:

```
# hdmi_cec, hdmi_hpd, and hdmi_out_en outputs
#set_property -dict { PACKAGE_PIN E19 IOSTANDARD LVC MOS33 } [get_ports hdmi_cec];
set_property -dict { PACKAGE_PIN E18 IOSTANDARD LVC MOS33 } [get_ports hdmi_hpd];
set_property -dict { PACKAGE_PIN F17 IOSTANDARD LVC MOS33 } [get_ports hdmi_out_en];
```

Finally, pin placement constraints for the VGA output connector are as follows:

```
#VGA Connector
set_property -dict { PACKAGE_PIN M19 IOSTANDARD LVC MOS33 } [get_ports { vga_r[0] }];
set_property -dict { PACKAGE_PIN L20 IOSTANDARD LVC MOS33 } [get_ports { vga_r[1] }];
```

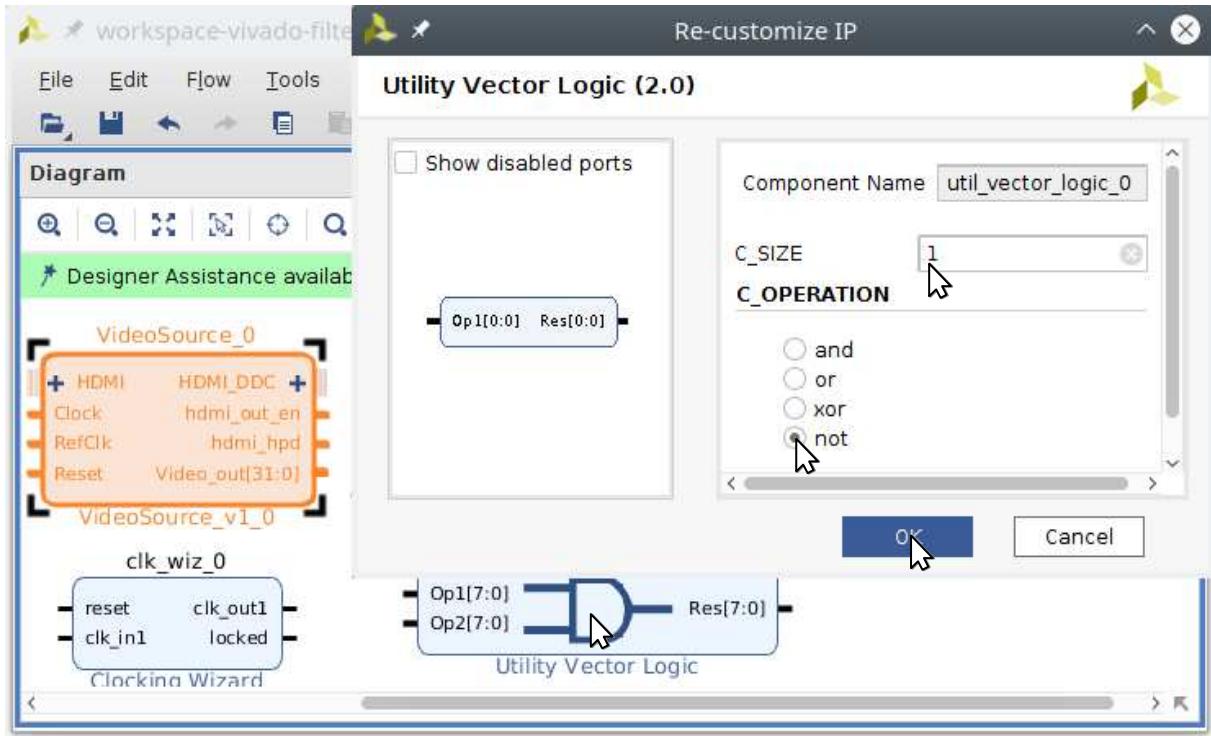


Figure 40: Re-customized “Utility Vector Logic” IPB to be a one bit inverter

```

set_property -dict { PACKAGE_PIN J20 IOSTANDARD LVCMOS33 } [get_ports { vga_r[2] }];
set_property -dict { PACKAGE_PIN G20 IOSTANDARD LVCMOS33 } [get_ports { vga_r[3] }];
set_property -dict { PACKAGE_PIN F19 IOSTANDARD LVCMOS33 } [get_ports { vga_r[4] }];
set_property -dict { PACKAGE_PIN H18 IOSTANDARD LVCMOS33 } [get_ports { vga_g[0] }];
set_property -dict { PACKAGE_PIN N20 IOSTANDARD LVCMOS33 } [get_ports { vga_g[1] }];
set_property -dict { PACKAGE_PIN L19 IOSTANDARD LVCMOS33 } [get_ports { vga_g[2] }];
set_property -dict { PACKAGE_PIN J19 IOSTANDARD LVCMOS33 } [get_ports { vga_g[3] }];
set_property -dict { PACKAGE_PIN H20 IOSTANDARD LVCMOS33 } [get_ports { vga_g[4] }];
set_property -dict { PACKAGE_PIN F20 IOSTANDARD LVCMOS33 } [get_ports { vga_g[5] }];
set_property -dict { PACKAGE_PIN P20 IOSTANDARD LVCMOS33 } [get_ports { vga_b[0] }];
set_property -dict { PACKAGE_PIN M20 IOSTANDARD LVCMOS33 } [get_ports { vga_b[1] }];
set_property -dict { PACKAGE_PIN K19 IOSTANDARD LVCMOS33 } [get_ports { vga_b[2] }];
set_property -dict { PACKAGE_PIN J18 IOSTANDARD LVCMOS33 } [get_ports { vga_b[3] }];
set_property -dict { PACKAGE_PIN G19 IOSTANDARD LVCMOS33 } [get_ports { vga_b[4] }];
set_property -dict { PACKAGE_PIN P19 IOSTANDARD LVCMOS33 } [get_ports { vga_hs }];
set_property -dict { PACKAGE_PIN R19 IOSTANDARD LVCMOS33 } [get_ports { vga_vs }];

```

- v) Subsequently, create a HDL wrapper (see Figure 46) around the block design and set it as the top module (see Figure 47). If the *Set as Top* command is grayed out, then the wrapper is already the top module.
- w) Start the synthesis as seen in Figure 31 and wait for synthesis to complete and open the hardware manager as shown in Figure 32.

Next, you should test the video-in Zybo board video-out setup as outlined in Sections 1.5 and 1.6. If you are comfortable with this setup, you can continue to test your generated *skcd_wrapper.bit* bit file.

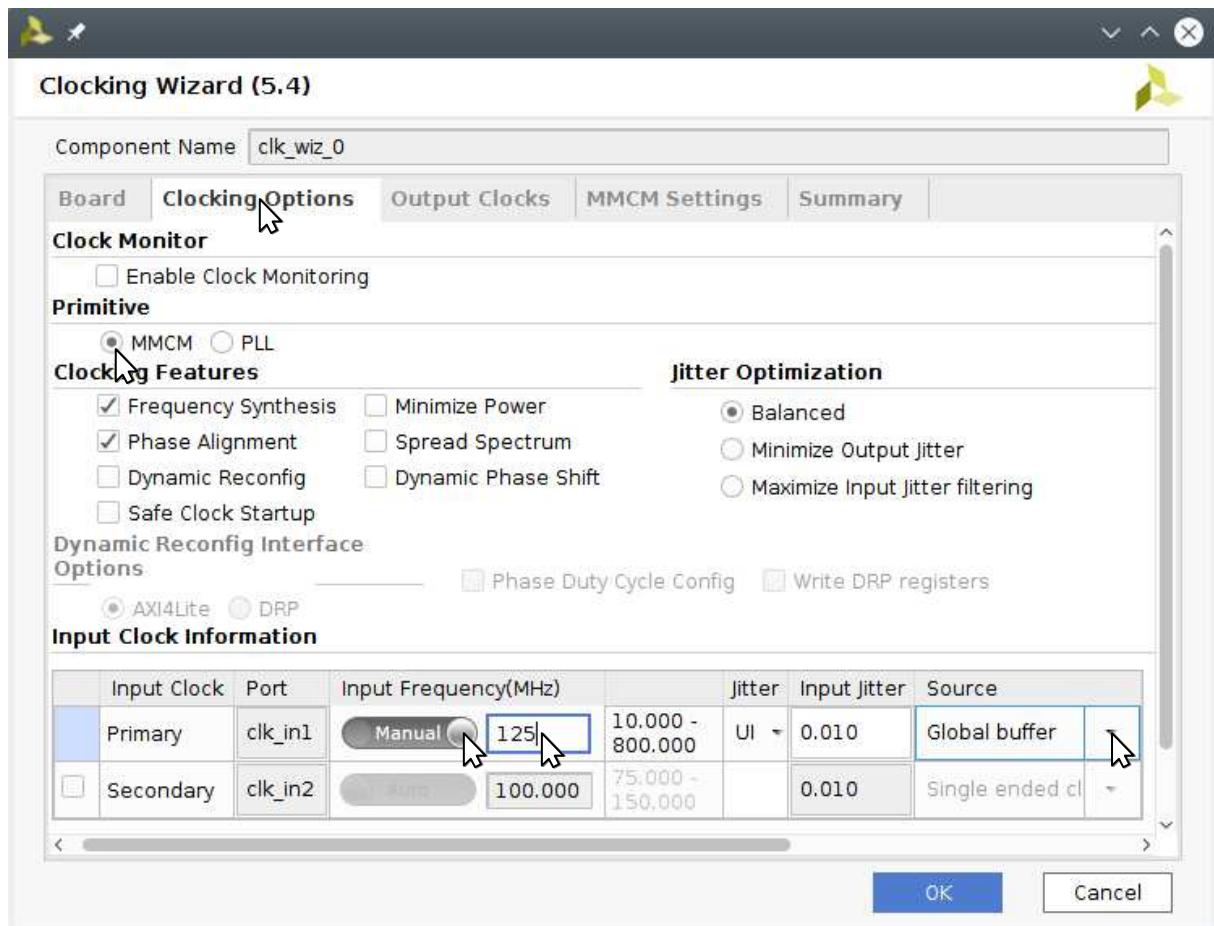


Figure 41: Re-customized “Clocking Wizard” to have a 125 MHz input clock

For this purpose, you have to program the Zynq-7000 PSoC by following the steps in Figure 33 but for the skcd block design. This should result in an output similar to Figure 34a on the monitor you connected to the VGA output of the Zybo board.

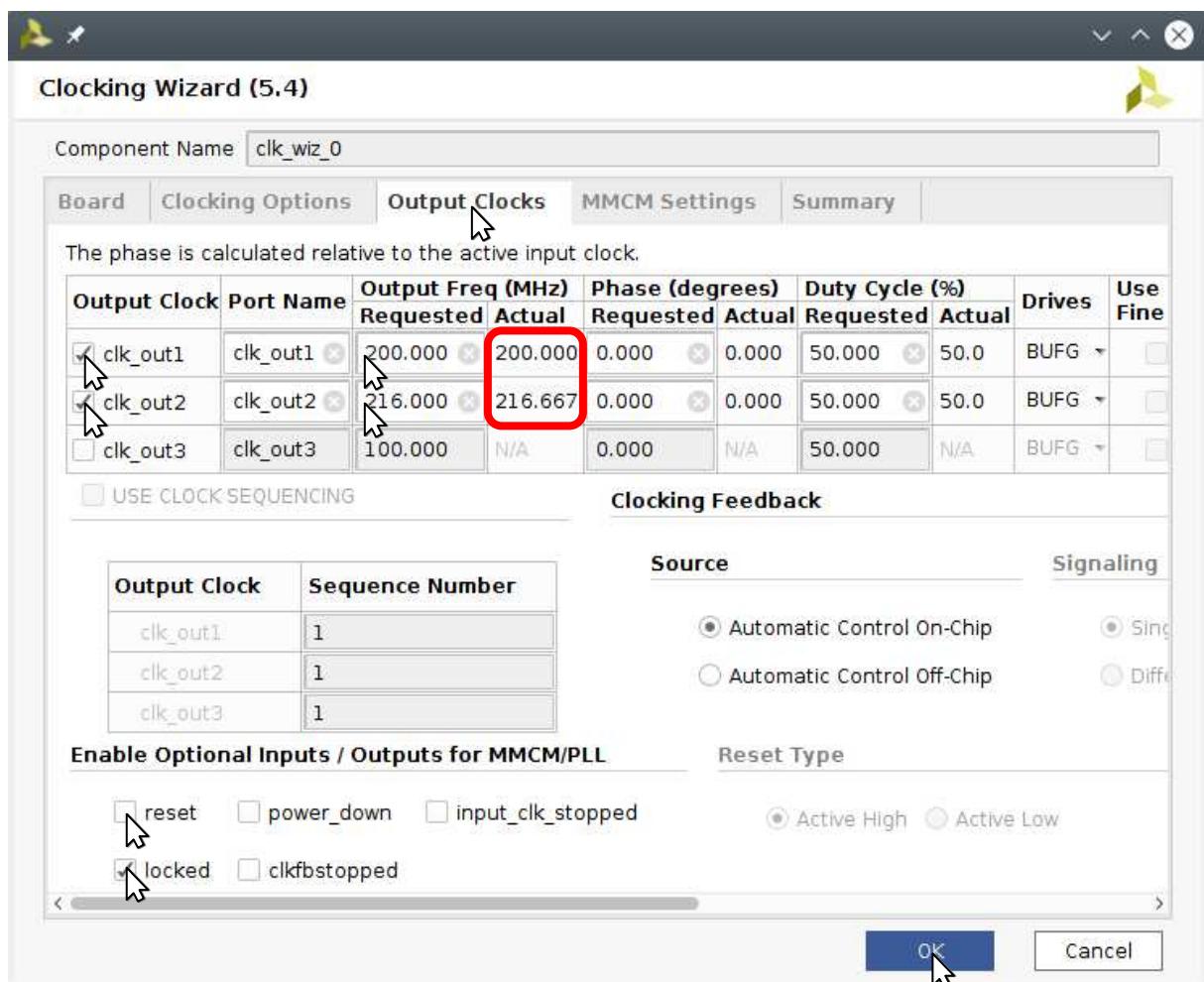


Figure 42: Re-customized “Clocking Wizard” to have 200 MHz and 216 MHz output clocks as well as disable the reset input

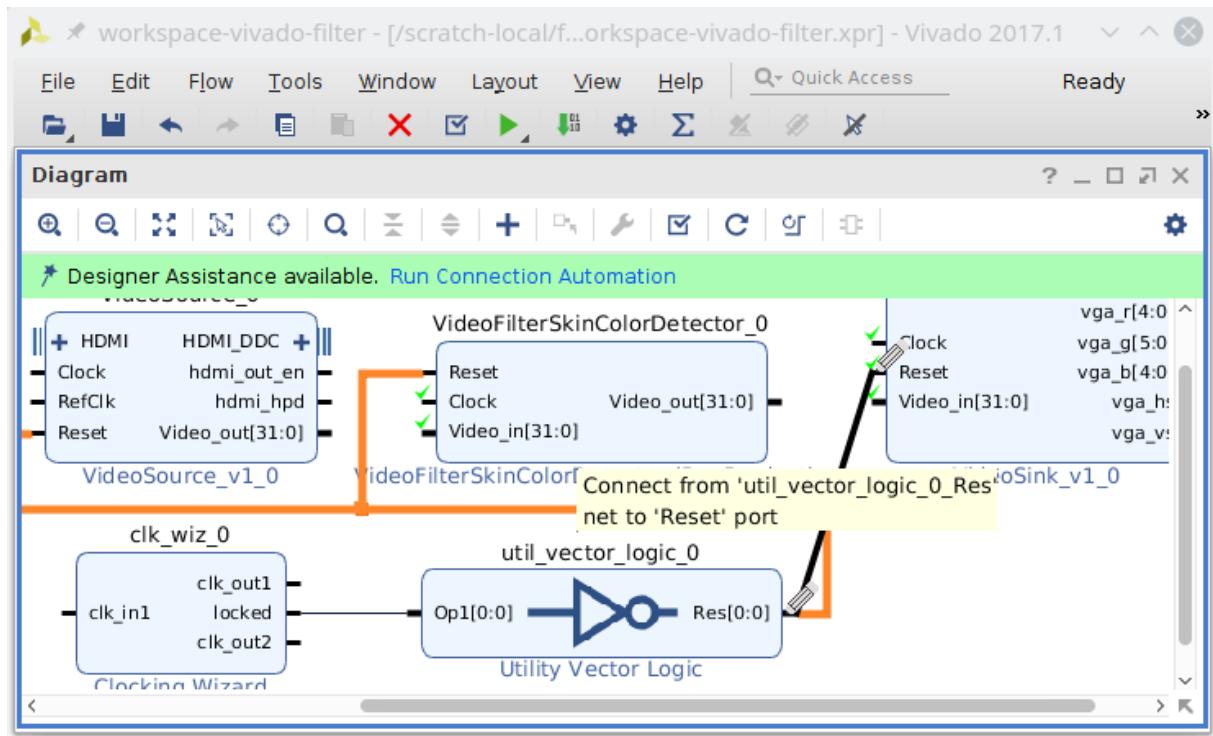


Figure 43: Connect the reset derived by inverting the locked output port of the “Clocking Wizard” to the Reset input ports of the IPBs

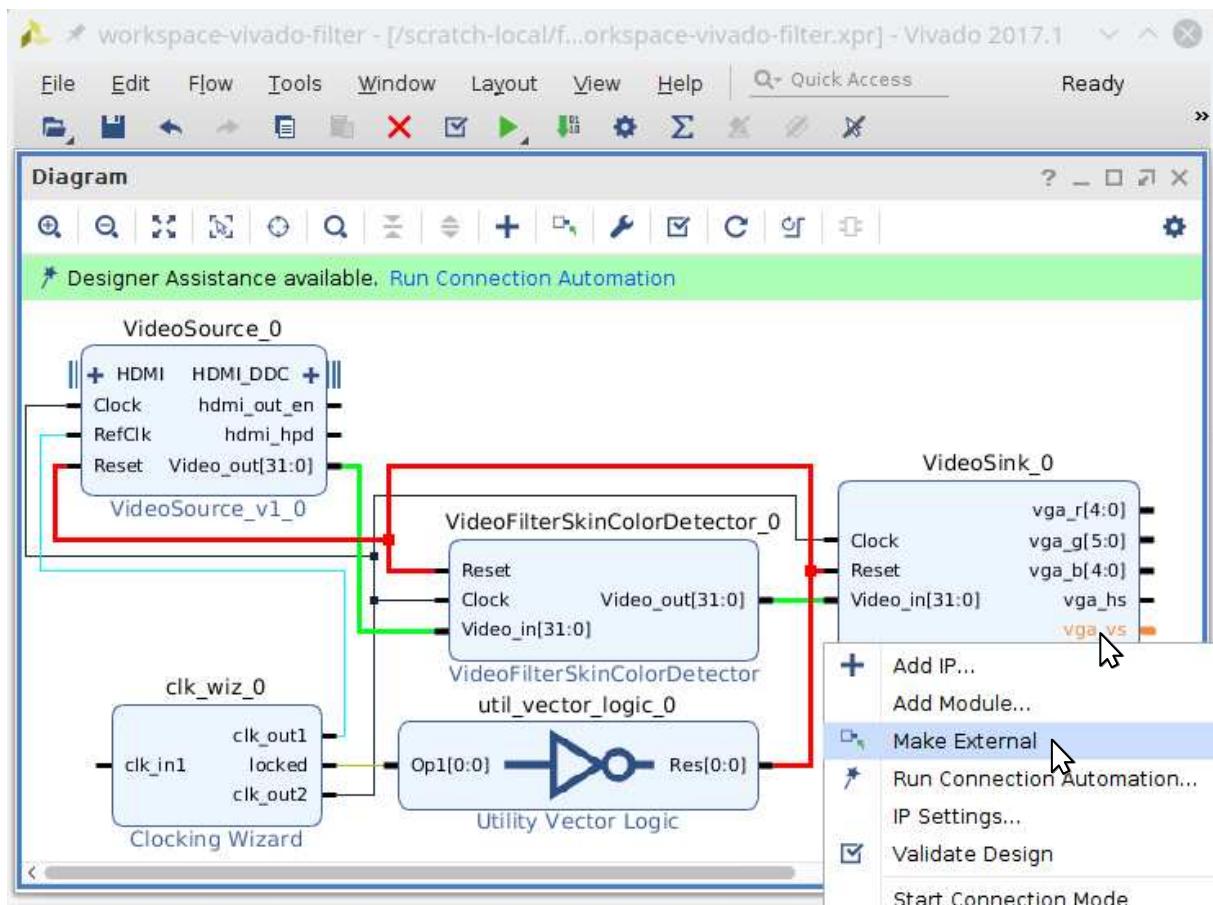


Figure 44: Make the inputs and outputs of the block design external

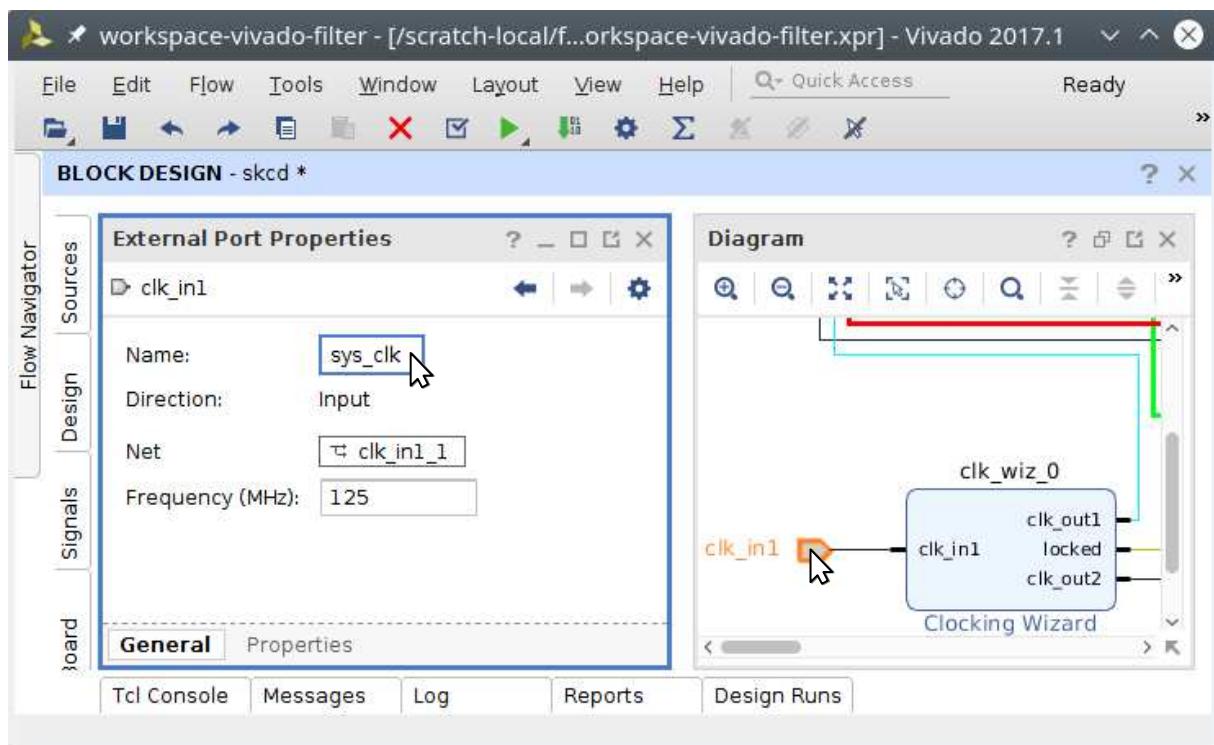


Figure 45: Renaming the clk_in1 input port to sys_clk

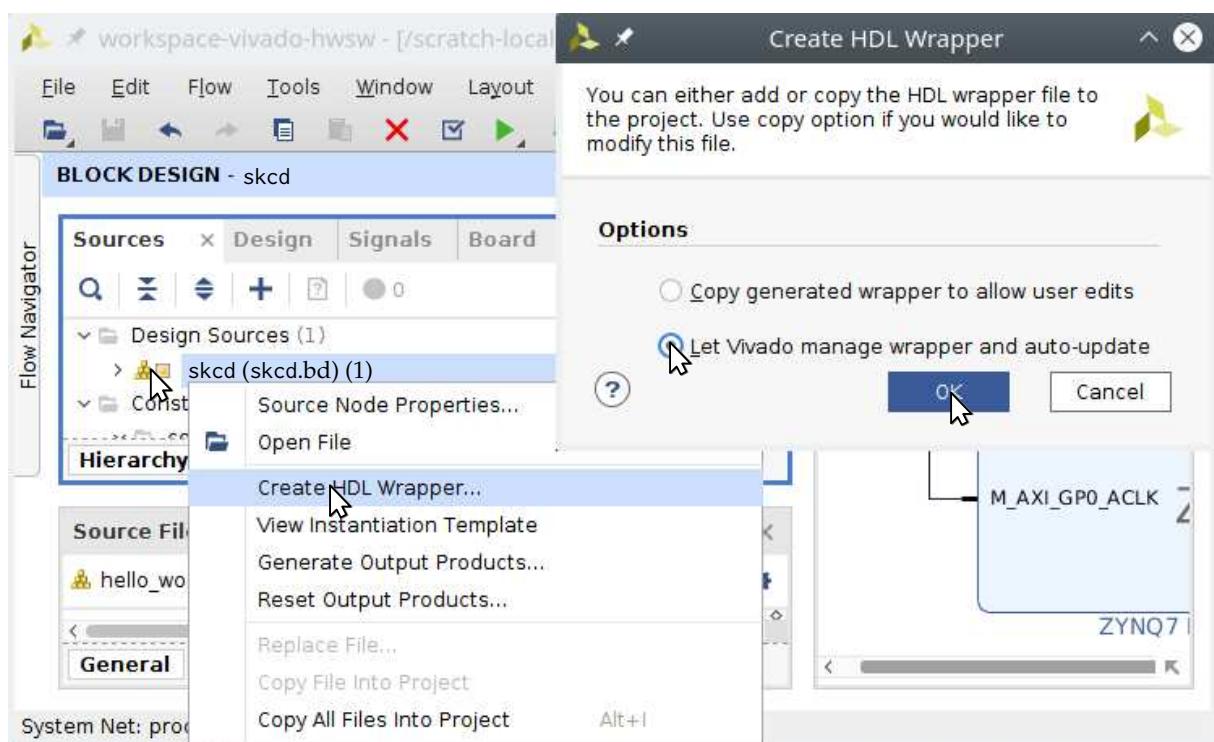


Figure 46: Create a HDL wrapper around the block design

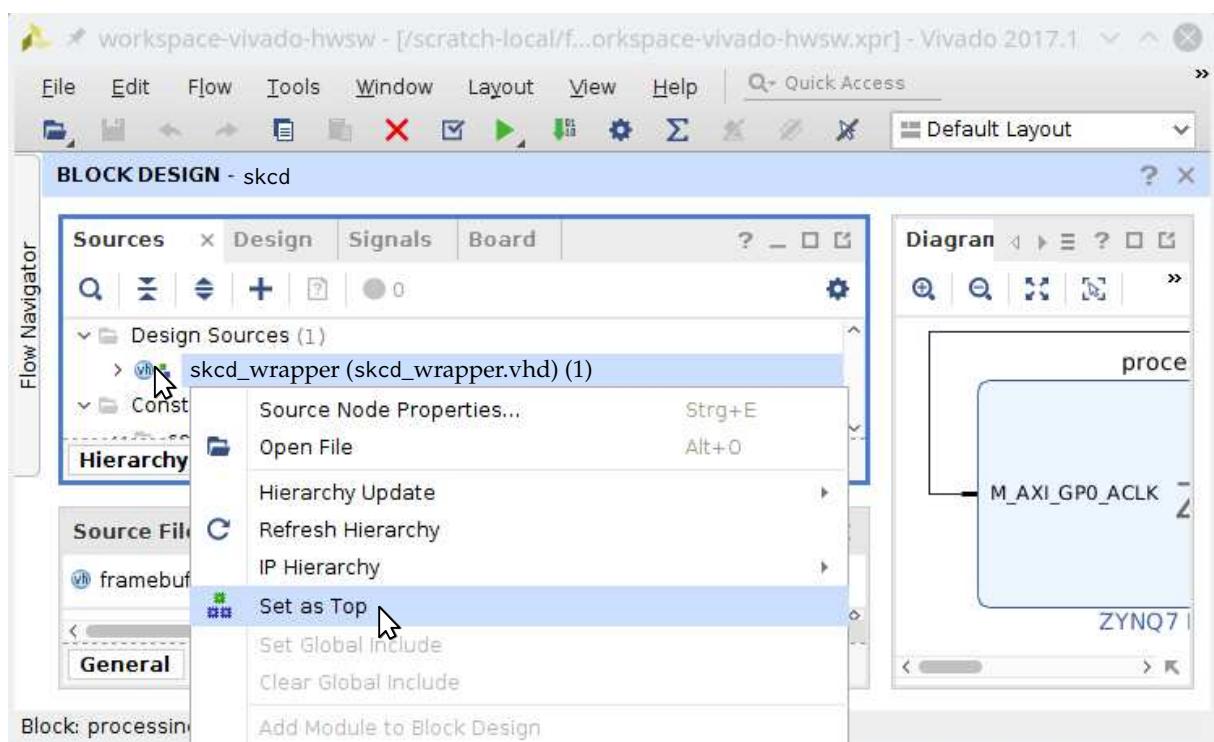


Figure 47: Set the wrapper as top module for synthesis

Task 2 (Fixed-Point vs. Floating-Point)

Project: hwVideoFilterGrayscaleConversion
 Files: src/module/cpp/VideoFilterGrayscaleConversion.hpp
 src/module/cpp/VideoFilterGrayscaleConversion.cpp

In the following, we will realize a simple gray-scale conversion filter (see Figure 48b for the simple filter chain). The multiplications for the conversion can be either performed using the



(a) Expected result for a simple gray scale conversion filter



(b) Video filter chain for a simple gray scale conversion design

Figure 48: Architecture and expected result for a simple skin gray scale conversion filter

float or double data type of C++ – very complex to realize in the FPGA – or using a SystemC fixed-point data type.

- a) Define the ports Clock, Reset, Video_in, and Video_out for FIFO and RTL abstraction levels. Take care to also name the ports accordingly in the constructor.
- b) Register the pixelThrd method as a SC_THREAD or SC_CTHREAD process according to the used abstraction level. Take care to specify that the VideoFilterGrayscaleConversion module

can have processes. Also specify the Reset signal as an active-high reset for the `pixelThrd` process. Check that the module can be used as a pass through filter module at the FIFO abstraction level.

- c) Change the `pixelThrd` process to also support `SC_CTHREAD` mode as well as reading and writing from the signal ports `Video_in` and `Video_out`. Check that the module can be used as a pass through filter module in RTL abstraction level.
- d) Change the `pixelThrd` process to realize gray scale conversion. This can be performed by computing the gray scale value $v_{\text{gray}} = 0.299 \cdot v_{\text{red}} + 0.587 \cdot v_{\text{green}} + 0.114 \cdot v_{\text{blue}}$ and writing it into all three color channels of the output pixel. Finally, check this functionality at both abstraction levels.

In the following, we will prepare our module for HLS. To distinguish a module at RTL abstraction level from one undergoing synthesis via VivadoHLS, the define `__SYNTHESIS__` will be defined in the *HLS synthesis case*.

- e) Annotate the *pixel generation loop* to process a value from the `Video_in` port at each master clock cycle. Note that there will not be a new pixel at every master clock cycle. At maximum, you might receive a pixel at every second cycle. Provide a reason for this observation.
- f) You might need to annotate the *pixel input and output code* to have a fixed timing not modifiable by VivadoHLS. However, code corresponding to statements outside the protocol regions might need to overlap with the statements in the protocol region in order to keep your timing constraints to generate a pixel every master clock cycle. Then, check that the filter still performs gray scale conversion even after HLS. This check should still be performed in the simulation environment.
- g) After HLS, check the used resources of the synthesized module and write them down.
- h) Most likely your gray scale conversion is employing floating-point computations, i.e., you used `float` or `double` for the multiplications. Thus, you will now modify the gray scale conversion to use fixed-point computations. You can use the `sc_dt::sc_fixed` data type for this. Determine the number of bits and the number of fractional bits yourself. Check your modification at various levels of abstraction and also perform HLS.
- i) After HLS of the fixed-point implementation, check the used resources of the synthesized module and write them down and compare them against the version using floating-point computations.

Finally, we will test the filter on the Zybo board by realizing and synthesizing the gray block design shown in Figure 49.

- j) Follow the tutorial in Task 1 to also test the gray scale filter on the Zybo board.

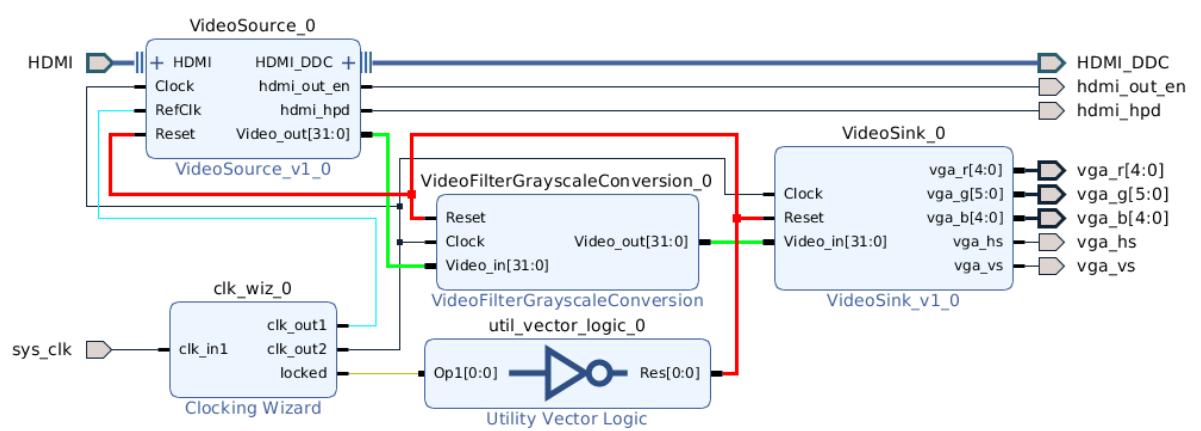
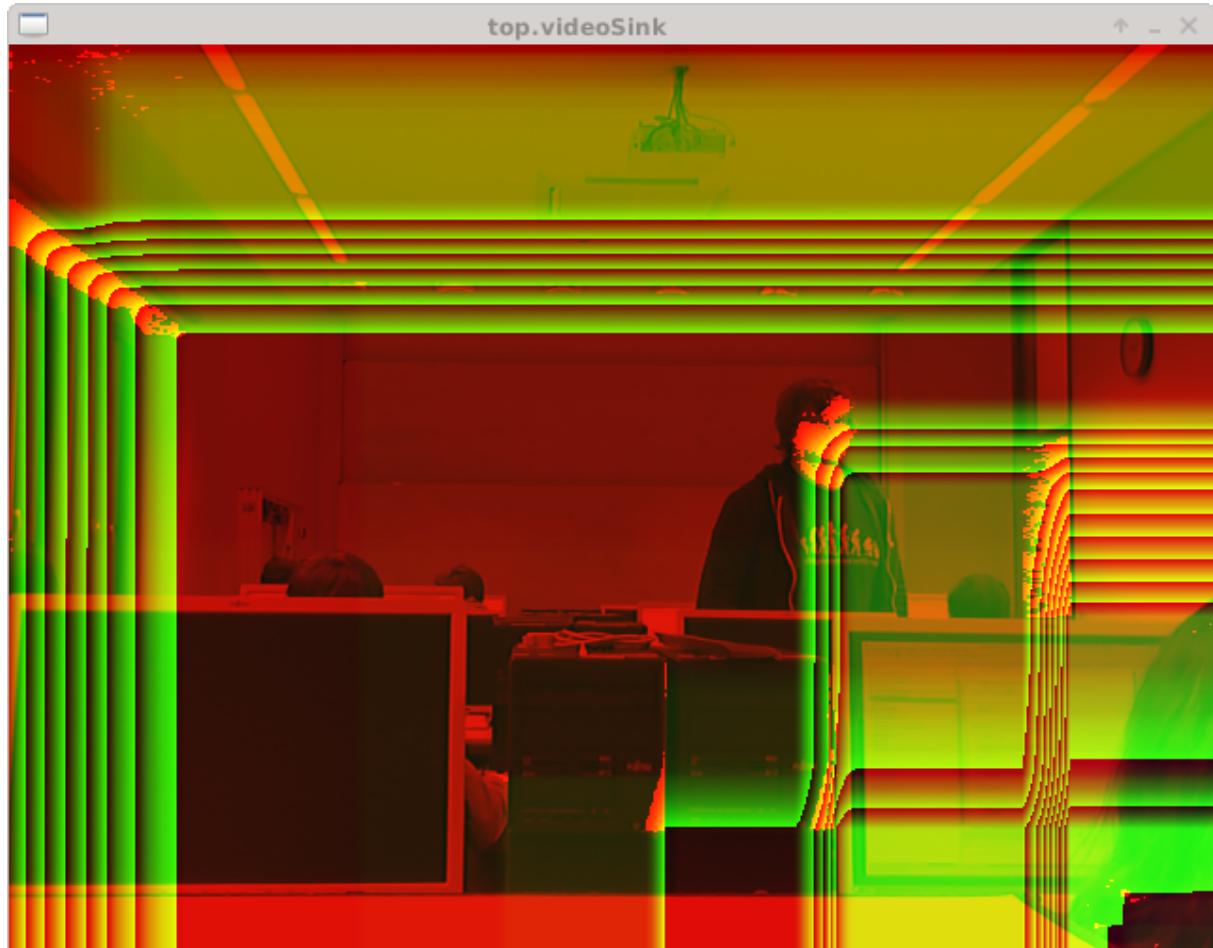


Figure 49: Vivado block design for the `VideoFilterGrayscaleConversion` filter chain

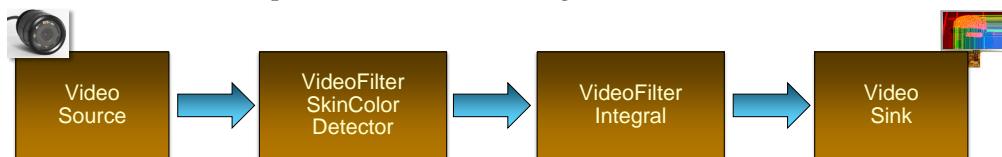
Task 3 (Using Memories in HLS)

Project: hwVideoFilterIntegral
 Files: src/module/cpp/VideoFilterIntegral.hpp
 src/module/cpp/VideoFilterIntegral.cpp

In the following, you should implement an integral filter. This filter should work for video streams up to a resolution of 1280×1024 . However, this filter will not work on its own but use the preceding skin color detector filter (see Figure 50b) to identify skin color pixels. First, you



(a) Expected result for the integral video filter chain



(b) Video filter chain with skin color detector and integral filter

Figure 50: Architecture and expected result for the integral video filter chain

will develop this filter at the FIFO abstraction level.

- Define the ports `Clock`, `Reset`, `Video_in`, and `Video_out` for FIFO and RTL abstraction levels. Take care to also name the ports accordingly in the constructor.
- Register the `pixelThrd` method as a `SC_THREAD` or `SC_CTHREAD` process according to the used abstraction level. Take care to specify that the `VideoFilterGrayscaleConversion` module

can have processes. Also specify the Reset signal as an active-high reset for the `pixelThrd` process.

- c) Generate a *line buffer*, i.e., an array to be realized as memory by VivadoHLS, called `linebuf`. The entry of the line buffer at position x should store the number of skin color pixels in the column at position x starting in the first line until the line, which is currently processed by the filter. Consider what this requirements means for the array size of the line buffer as well as how many bits are needed to store an entry of the line buffer.
- d) To calculate the integral image in the `pixelThrd` process, you must know the x position of the pixel which is currently processed. This will be needed to manipulate the correct line buffer entry.
- e) Moreover, it is interesting to know when the first (visible) line of the image is processed by the filter. For this line, the entries in the line buffer must assumed to be zero, i.e., there are obviously no skin color pixels above the first line of the image.
- f) Additionally, knowledge when a (visible) line has ended will also prove important. After a line has ended, the *summed area* (*sum*) has to be reseted, i.e., there are obviously no skin color pixels to the left of the left boarder of the image.
- g) Then, the integral image function can be realized by manipulating the summer area variable and the corresponding line buffer entry at position x .
- h) Finally, the summed are has to be transported in the *green and blue channel* of the output pixel. Here, you should store the *high bits* in the *blue channel* and the *next lower bits* in the *green channel*. You might not have a sufficient number of bits in the green and blue channels to store the summed area information. Thus, you might throw away some lower bits of the summed area information. Later on, this might trip you up in your particle filter implementation. Keep this problem in mind.
- i) For visual debugging purpose, you might consider storing a gray scale image in the *red channel* of the output pixel. However, store 0xFF if a skin color pixel has been detected. For an example what the result should look like, consult Figure 50a.

In the following, we will prepare our module for HLS. First, you will enable the filter to work at RTL abstraction level. Subsequently, perform HLS and check the result. Here, you might need to perform additional modifications to optimize the memory accesses to the line buffer. Otherwise, you might not be able to satisfy the timing requirements of the filter. As always, to distinguish a module at RTL abstraction level from one undergoing synthesis via VivadoHLS, the define `__SYNTHESIS__` will be defined in the *HLS synthesis case*.

- j) Change the `pixelThrd` process to also support RTL abstraction level. Remember, at RTL, a new pixel on the `Video_out` port is signaled by a *rising edge of the pixel clock*. Check that the filter `VideoFilterIntegral` works at both abstraction levels.
- k) Next, consider which *initiation interval* to choose for the *pixel processing loop*. Annotate your chosen interval with the corresponding HLS pragma.
- l) For HLS, you might need to perform additional modifications to *optimize the memory accesses* to the line buffer to keep your chosen *initiation interval*. For example, to tell VivadoHLS that you do not require values from `linebuf` written in the previous iteration of the loop in this iteration, you might annotate `#pragma HLS dependence variable=linebuf inter false` **into the body of the while loop**. Check your modification at various levels of abstraction.

Finally, we will test the filter on the Zybo board by realizing and synthesizing the `integral` block design shown in Figure 51.

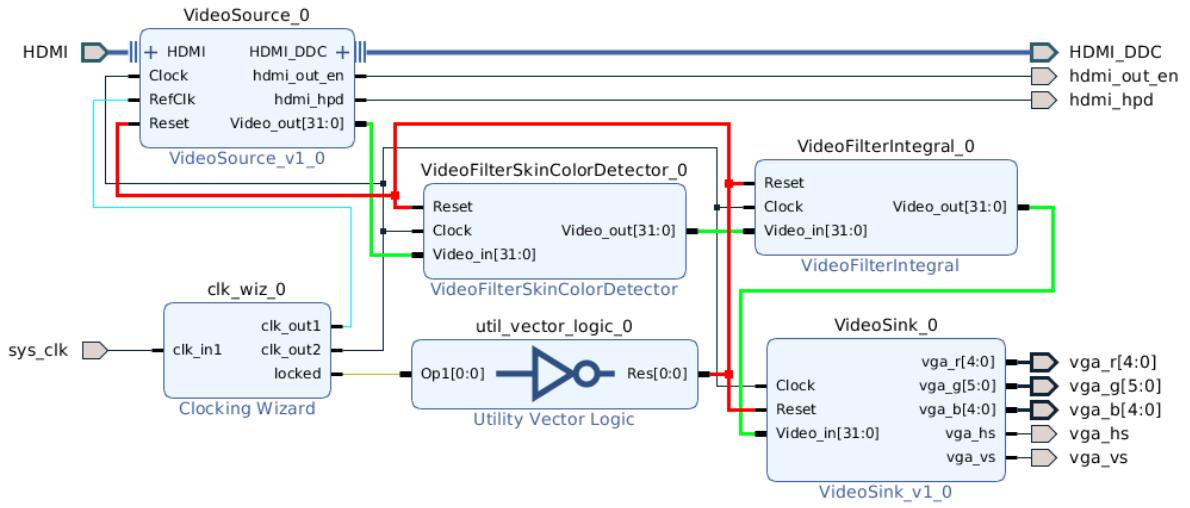


Figure 51: Vivado block design for the skin color detector and integral filter chain

- m) Follow the tutorial in Task 1 to also test the integral filter on the Zybo board. Don't forget that the skin color filter has to be put in front of the integral filter.