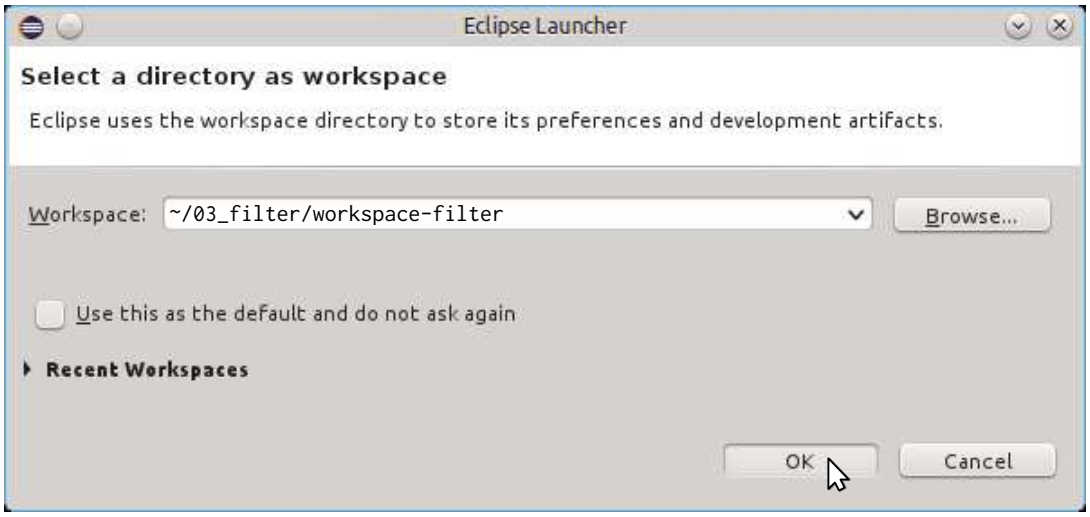
Lehrstuhl für Informatik 12
Cauerstraße 11
91058 Erlangen

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

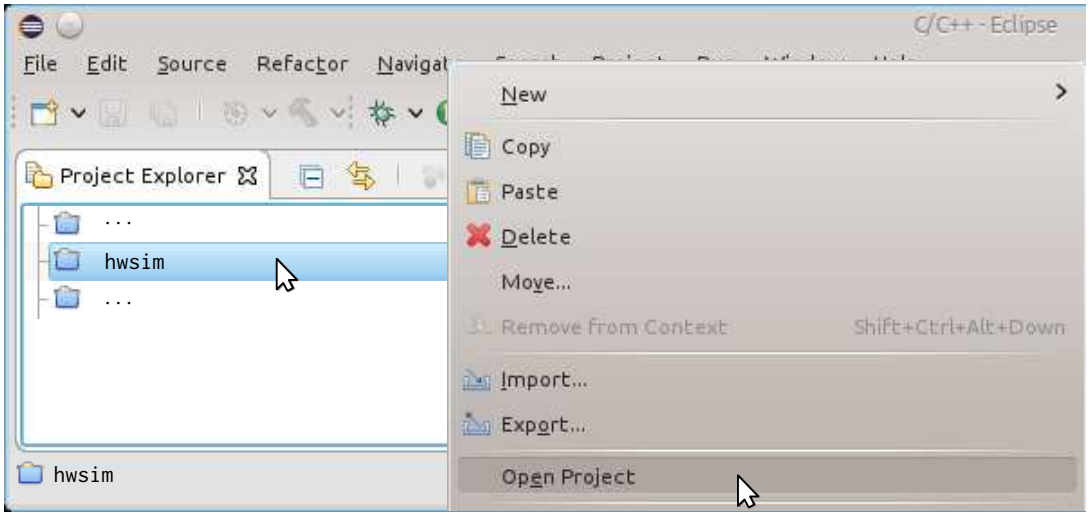
3. Exercise of the Laboratory
Entwicklung interaktiver eingebetteter Systeme

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_sysc/workspace-sysc* 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
```

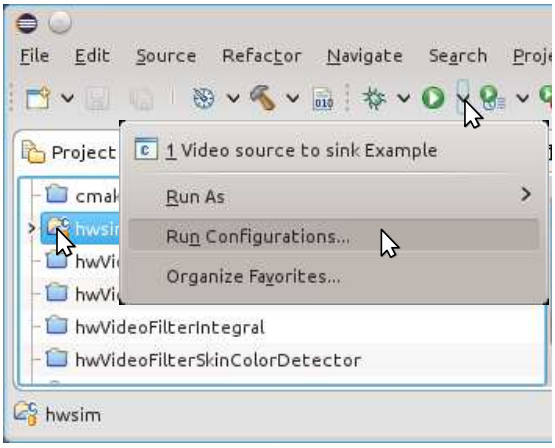


Figure 2: Opening the run configurations

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.

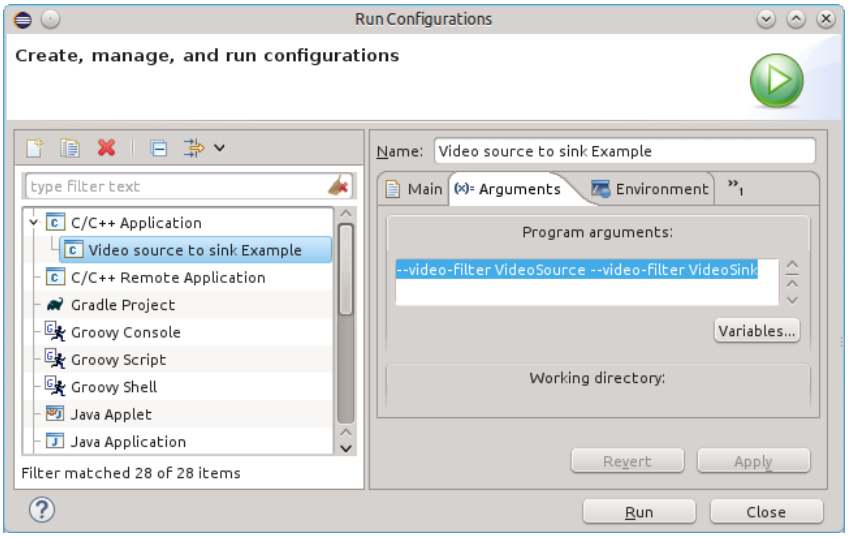
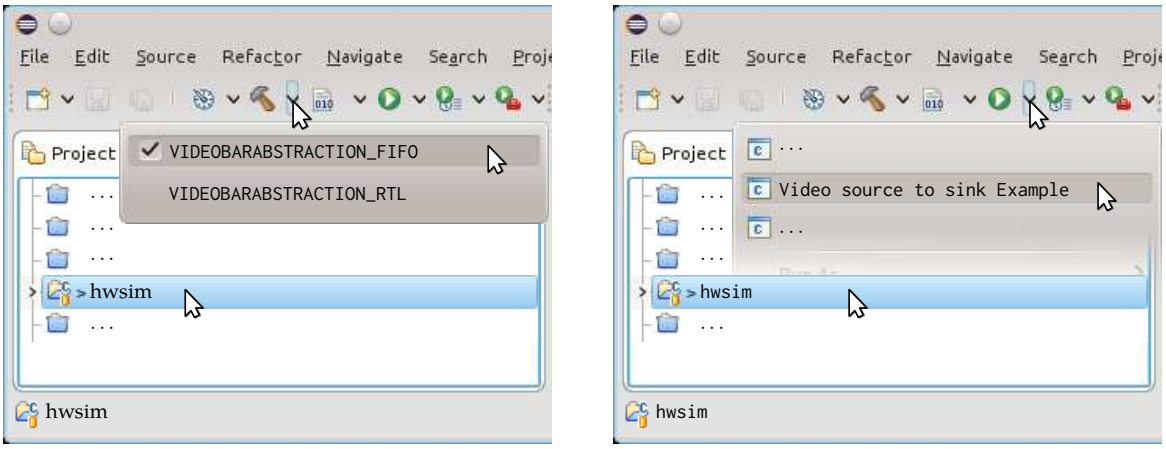


Figure 3: Examining the run configuration *Video source to sink Example* in more detail



(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 filters[1] `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

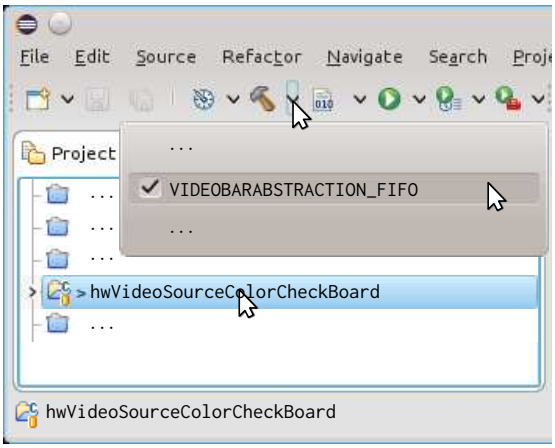
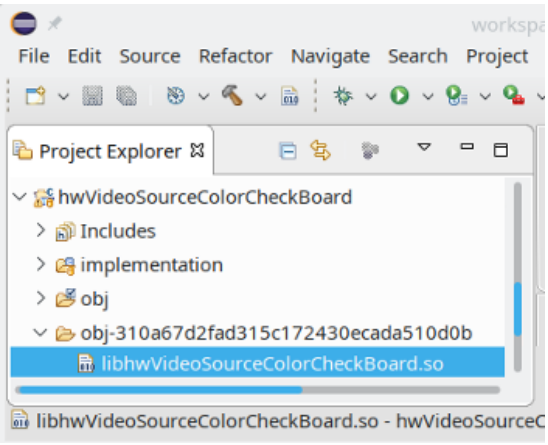


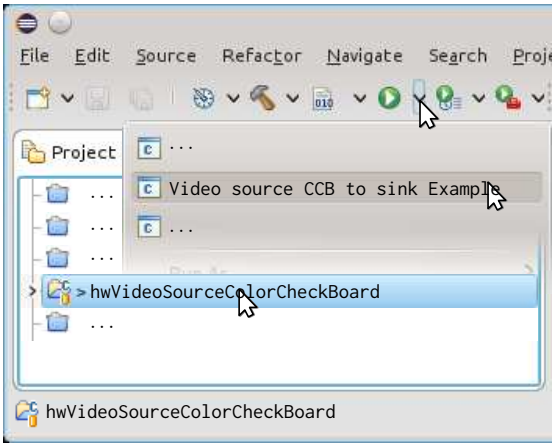
Figure 5: Compile the CCB in FIFO abstraction

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

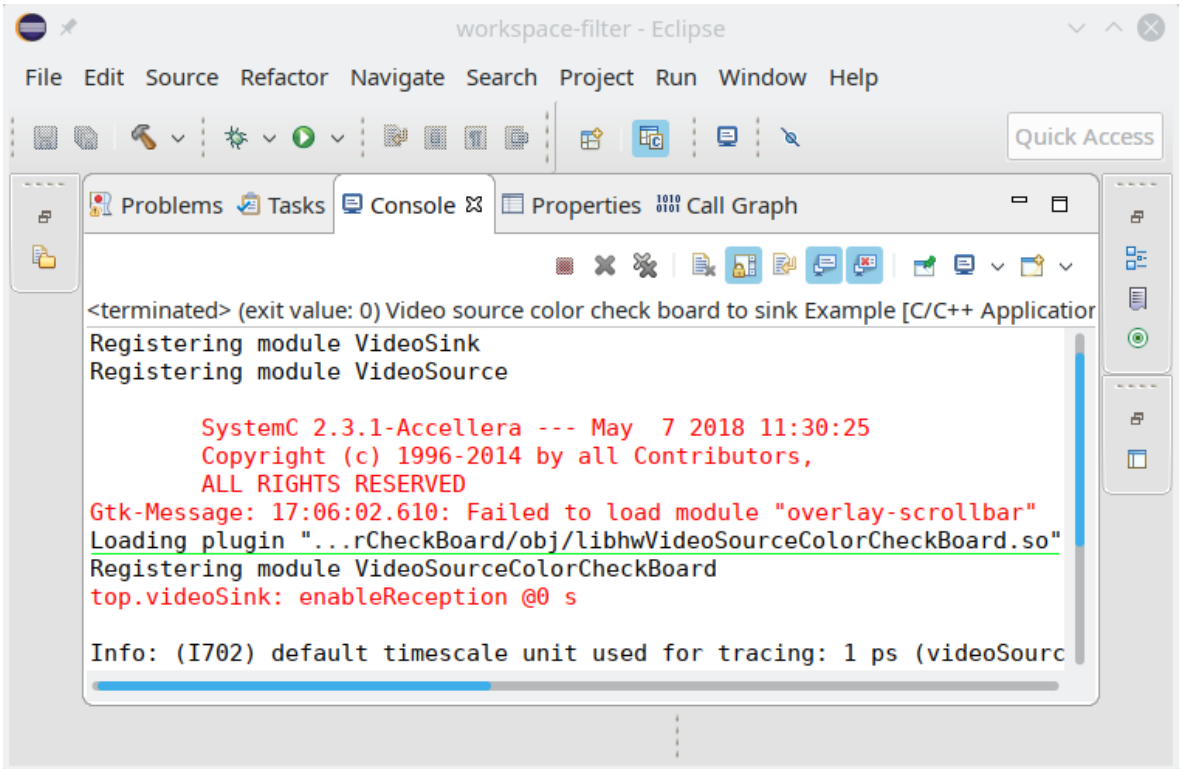
in the plugin library *libhwVideoSourceColor-CheckBoard.so* shown in Figure 6a. If the object directory *obj-310a...* does not appear in eclipse, then there is a differences between what files are present in the file system and what files Eclipse knows of. To notify Eclipse that new files have appeared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



(a) Resulting plugin library *libhwVideoSourceColor-CheckBoard.so*



(b) Starting the `VideoSourceColorCheckBoard` ⟶ `VideoSink` example



(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 `VideoSourceColorCheck-Board` 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*

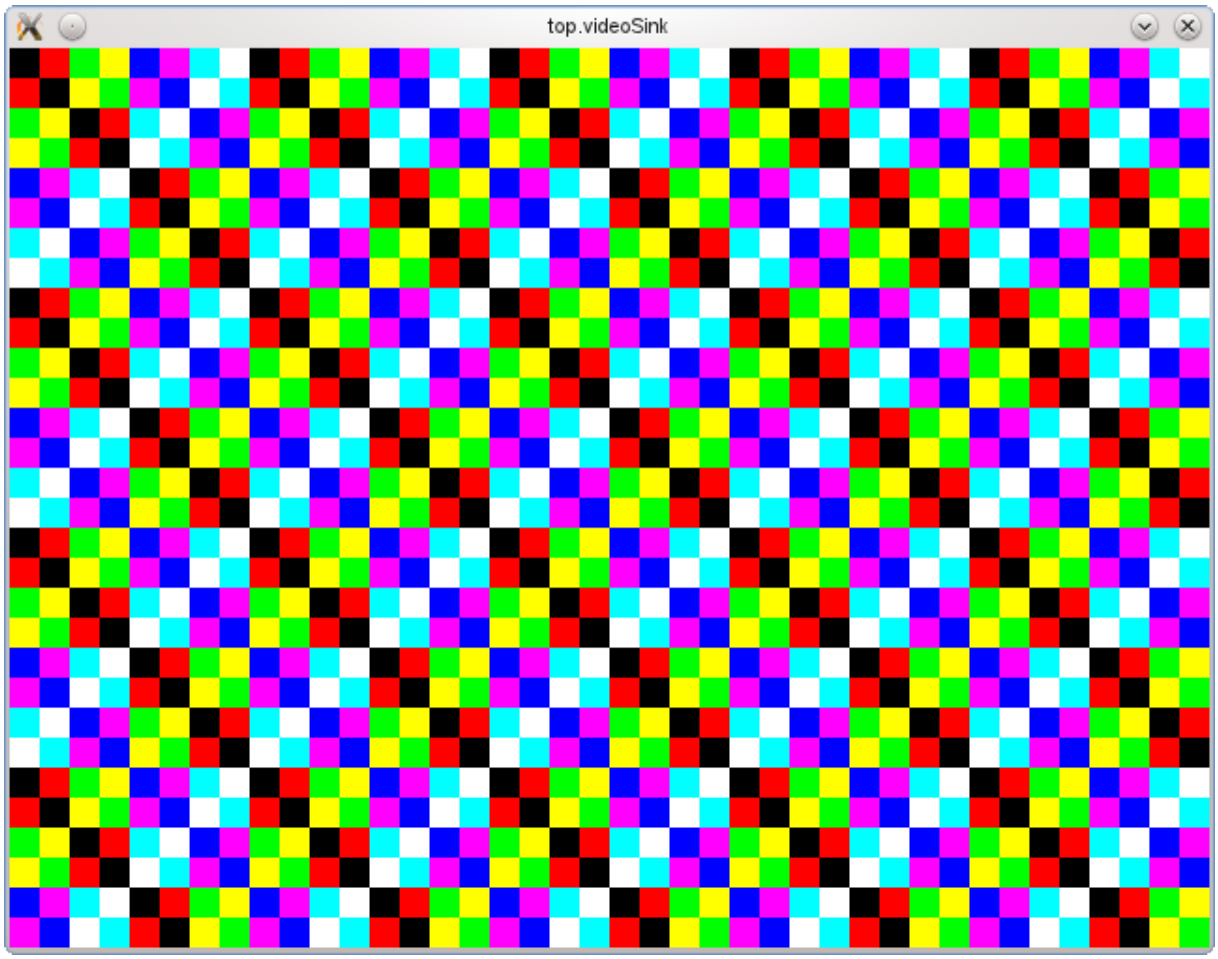


Figure 7: Color check board generated by `VideoSourceColorCheckBoard`

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*
lrwxrwxrwx 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*
lrwxrwxrwx 1 test stud   36 Mar 4 08:56 obj -> obj-97f701c5b126235a4f0285c667d9cc76
```

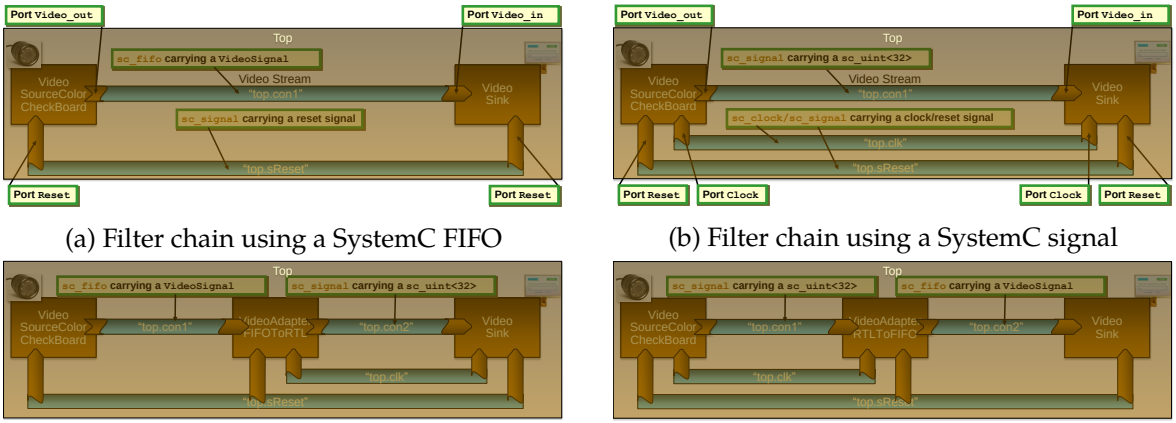
```
drwxr-xr-x 4 test stud 4096 Mar 1 14:56 obj-310a67d2fad315c172430ecada510d0b
drwxr-xr-x 4 test stud 4096 Mar 4 08:56 obj-97f701c5b126235a4f0285c667d9cc76
```

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
```



(a) Filter chain using a SystemC FIFO

(b) Filter chain using a SystemC signal

(c) Filter chain requiring the adapter `VideoAdapter-FIFOToRTL`

(d) Filter chain requiring the adapter `VideoAdapter-RTLToFIFO`

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 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 ZCU102 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 *ZCU102Init.sh* script to program the Programmable System-on-Chip (PSoC) with this bit file.

If you want to test the ZCU102 board and how to connect it to your monitor, use the *Micro USB cable* (see Figure 9a) to connect your computer with the ZCU102 board and the *DisplayPort cable* (see Figure 9b) to connect the ZCU102 board to your monitor. Then, execute the *ZCU102Init.sh*

(a) Micro USB cable to connect the ZCU102 board to your computer to facilitate power provisioning and JTAG programming of the PSoC

(b) DisplayPort cable for connecting the ZCU102 board to your monitor to facilitate video output

Figure 9: Cables required to connect the ZCU102 board to facilitate video output

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]$ ./ZCU102Init.sh
============================================================
Before I can program the ZCU102 board with your executable
and bitstream, you have to turn the ZCU102 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!
```

You can continue by pressing enter.

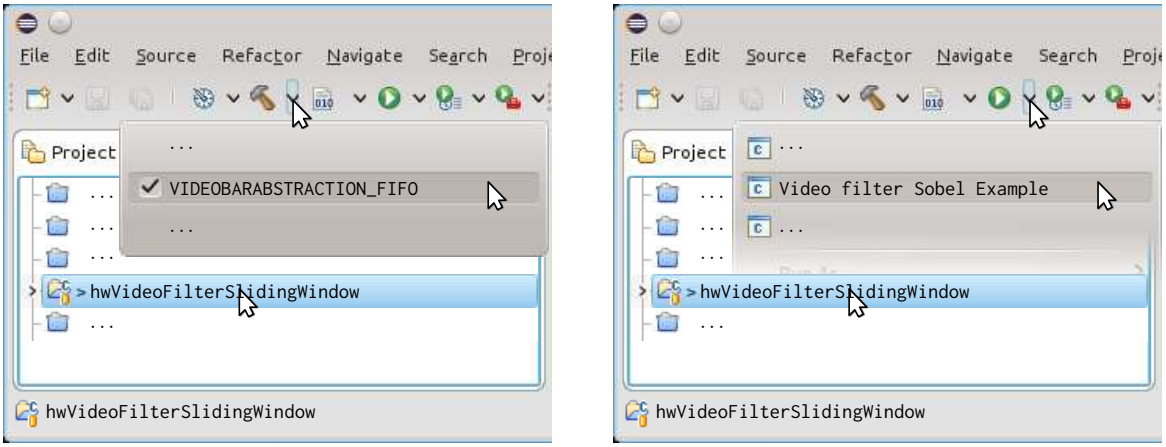
```
attempting to launch hw_server

****** 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 DisplayPort input. This should result in the test image from Figure 7 being display on the monitor.

1.5 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 10 to compile and run the simulation. This should result in an output similar to the one shown in Figure 11.



(a) Compiling the Sobel filter at FIFO abstraction

(b) Running the simulation

Figure 10: Running the Sobel video filter chain at FIFO abstraction

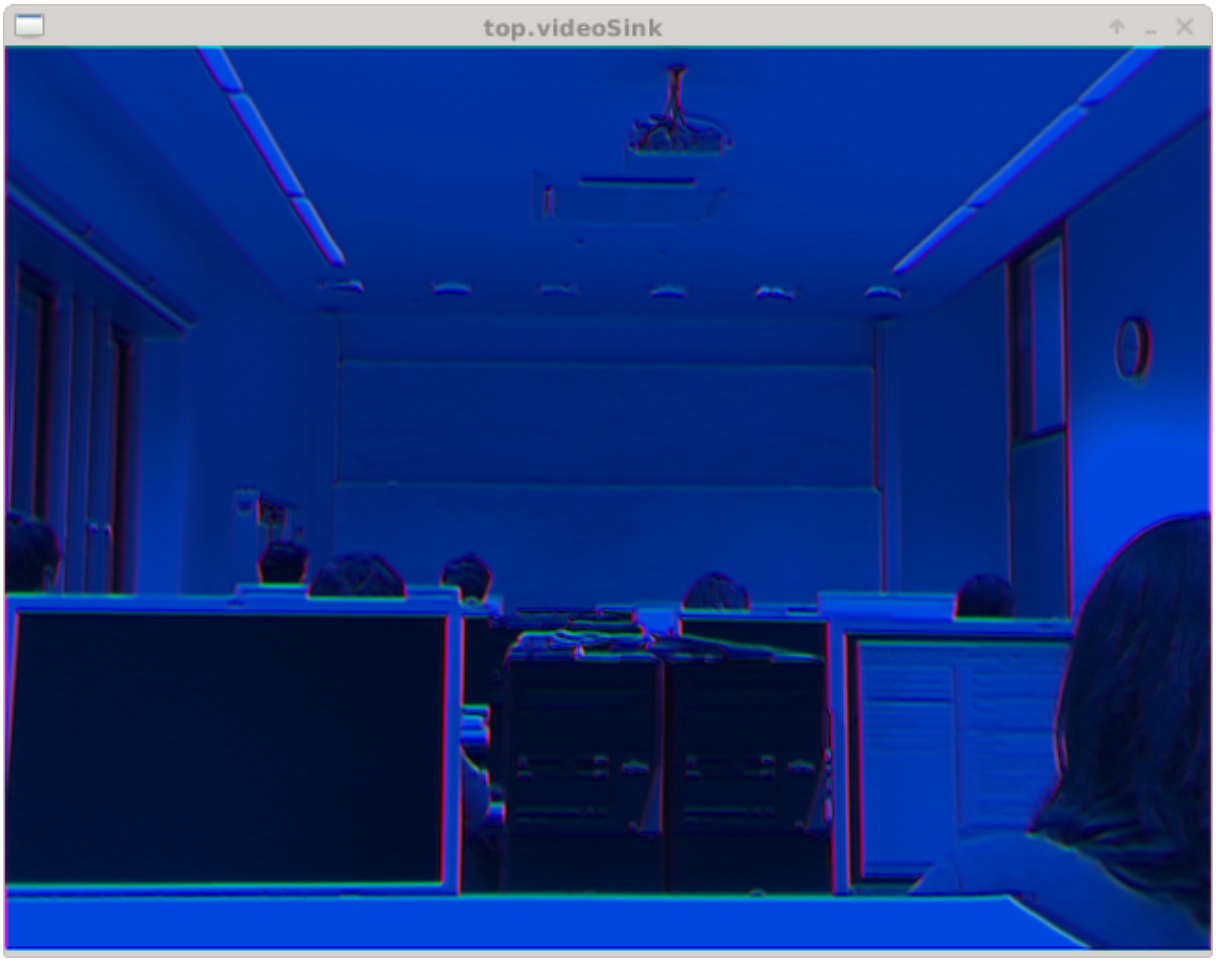


Figure 11: Example output of the Sobel video filter chain

Compiling and running the simulation is not necessary to test on the ZCU102 board as you can use the pre-generated *sobel_wrapper.bit* bit file and the *ZCU102Init.sh* script contained in the `hwVideoFilterSlidingWindow` project to program the ZCU102 board.

To provide video input to the ZCU102 board, the USB camera shown in Figure 12a is *not connected to the ZCU102 board* but to your computer. Thus, if you want to test the ZCU102 board and how to run a video filter chain on it, you have to connect your computer's DisplayPort output (see Figure 12c) to the ZCU102 board's HDMI input by using a *DisplayPort to HDMI adapter* and a *HDMI cable* (see Figure 12b). The connections introduced in Section 1.4 are – of course – still required for video output to your monitor. Then, execute the *ZCU102Init.sh* script contained in the `hwVideoFilterSlidingWindow` project.

```
[...test@codesigni165:~]$ cd ~/03_filter/workspace-filter
[...65:workspace-filter]$ cd hwVideoFilterSlidingWindow
[...FilterSlidingWindow]$ ./ZCU102Init.sh
```

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 ressemble 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 ZCU102 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 ZCU102 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 ZCU102 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 ZCU102 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 ZCU102 board is located on your desktop and press **f** to enter *full screen mode* for the mplayer window.

1.6 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 13. Vivado sources IPBs from Intellectual Property (IP) repositories. On 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 ZCU102 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 `VideoSourceColorCheckBoard` 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 14. This will start HLS and then open the VivadoHLS GUI (see Figure 15) so that you can examine the synthesis results in terms of obtained timing, resource usage,

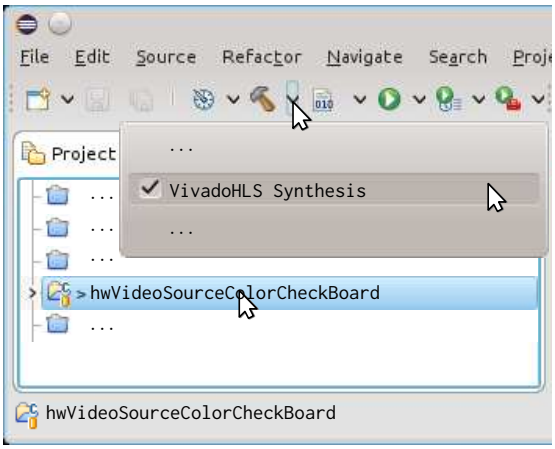


Figure 14: Starting VivadoHLS to synthesize the CCB into an IPB

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_ippack.tcl* script is run by Vivado. This step corresponds to the following output in the Eclipse console window:

```
vivado -notrace -mode batch -source run_ippack.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
    ** Copyright 1986-2017 Xilinx, Inc. All Rights Reserved.

source run_ippack.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** ▷ **solution1** ▷ **syn** ▷ **report** by double clicking on **VideoSourceColorCeckBoard_pixelThrd_csynth.rpt**. This should display the *synthesis report* shown in the right half of Figure 15. 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 $\frac{1}{3.88\,ns}$, (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) {...}` 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 16) and selecting **VideoSourceColorCeckBoard_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 $C0$ to $C5$. 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) {...}` loop is depicted in orange from clock cycle $C1$ to $C5$. 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) {...}` loop (**Loop1**) you found under **Detail** ▷ **Loop** ▷ **Loop1** in the *synthesis report* shown in the right half of Figure 15. To exemplify, the `while (true) {...}` 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) {...}` 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**.

1.7 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 DisplayPort connector. Then, you can generate an FPGA *bitstream*. 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 17. When opened, you can see that two block designs `ccb` and `sobel` (marked red in Figure 18) 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 18. Next, open the *IP catalog* view in Vivado by clicking on **Window** ▷ **IP Catalog** as shown in Figure 19. This should result in a view similar to the one in Figure 20. 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 20. Next, open the `ccb` block design (see Figure 21) and run **Tools** ▷ **Report** ▷ **Report IP Status** as shown in Figure 22 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 23. 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 14. 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 23 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 23.

If you also want to generate the bit file for the `sobel` block design, start *VivadoHLS Synthesis* for the `hwVideoFilterSlidingWindow` project. Then, disable the `ccb` block design and enable the `sobel` block design instead, i.e., the step in Figure 18 but for the `sobel` block design. Moreover, activate the *sobel.xdc* constraint file. 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.

(a) USB camera for your computer to act as video source for the ZCU102 board

(b) DisplayPort to HDMI adapter and HDMI cable to connect computer and ZCU102 board to facilitate video input

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

Figure 12: Additional cable, adapter, and camera required to feed video input to the ZCU102 board

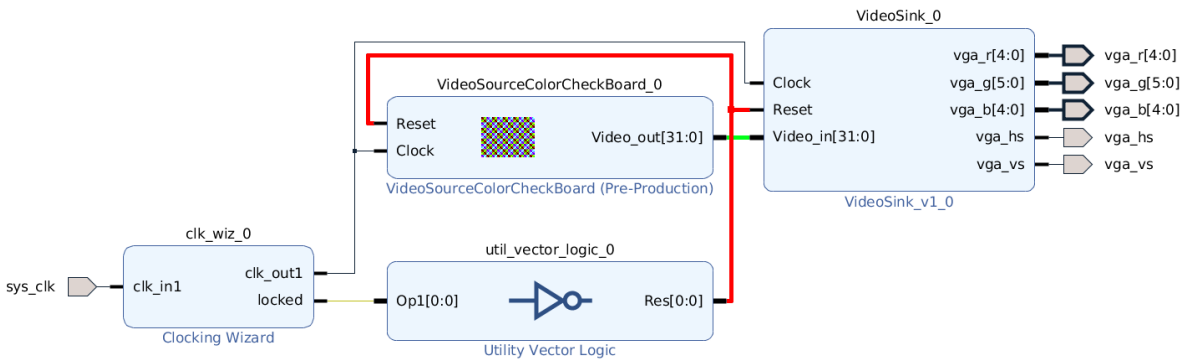


Figure 13: Vivado block design for the CCB test design

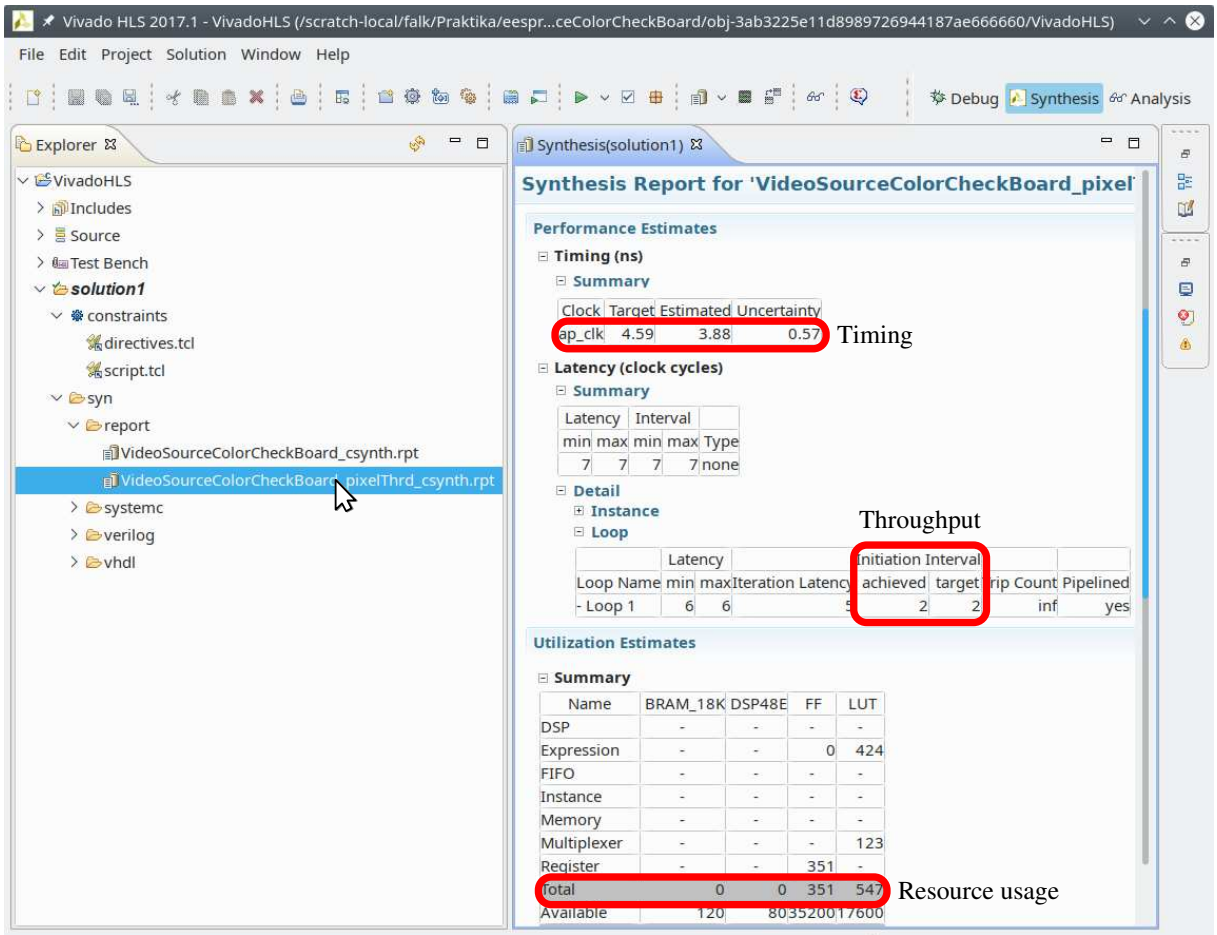


Figure 15: Synthesis report for the CCB IPB

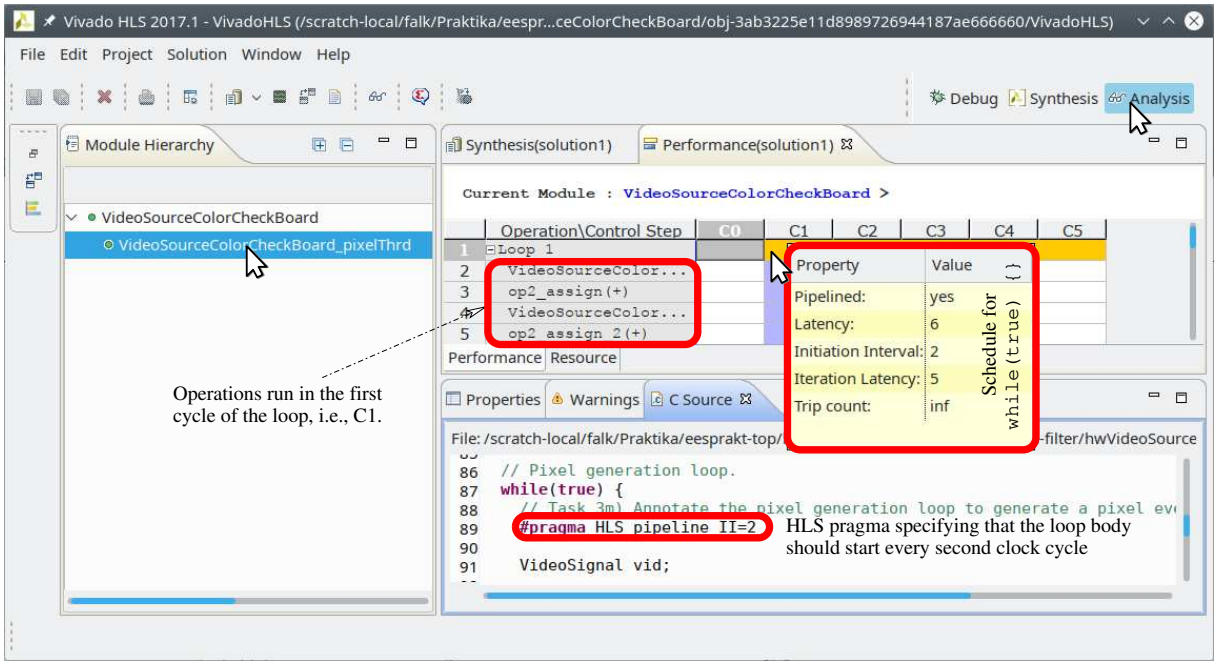


Figure 16: Examining the schedule of the `while (true) {...}` loop in more detail

Figure 17: Open the `workspace-vivado-filter` Vivado project

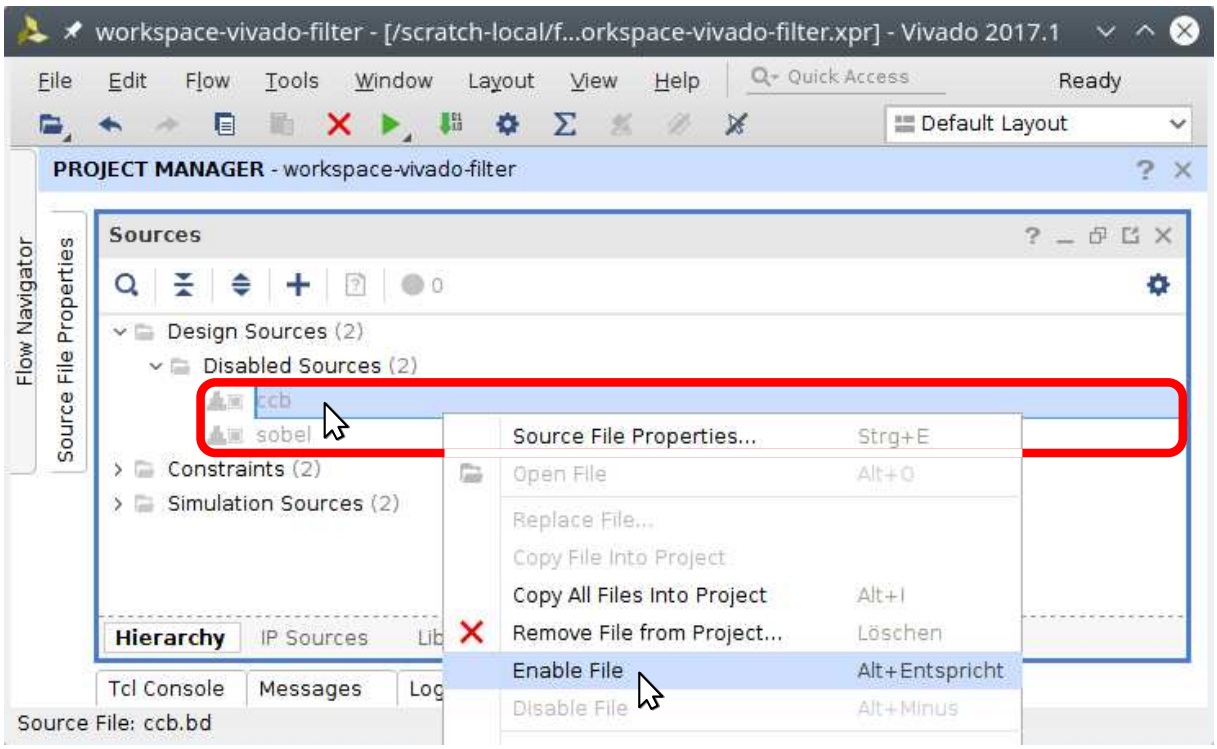


Figure 18: Enabling the block design ccb

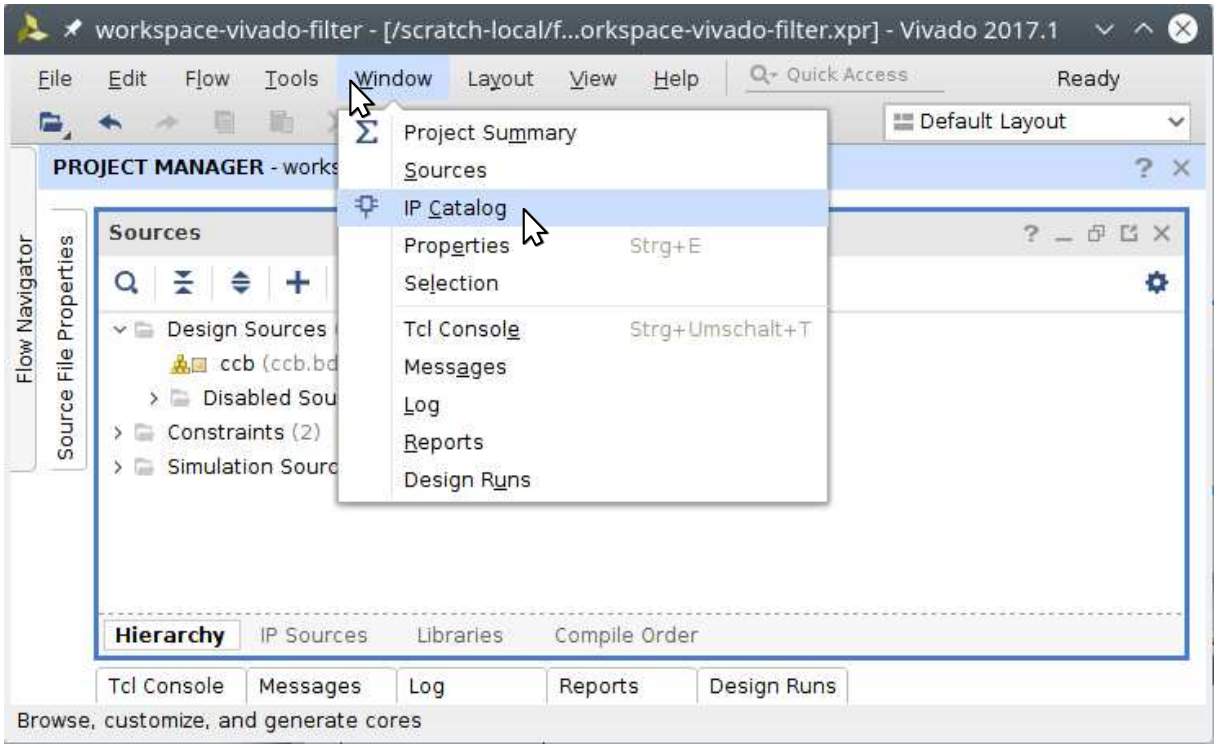


Figure 19: Open the *IP catalog* view in Vivado

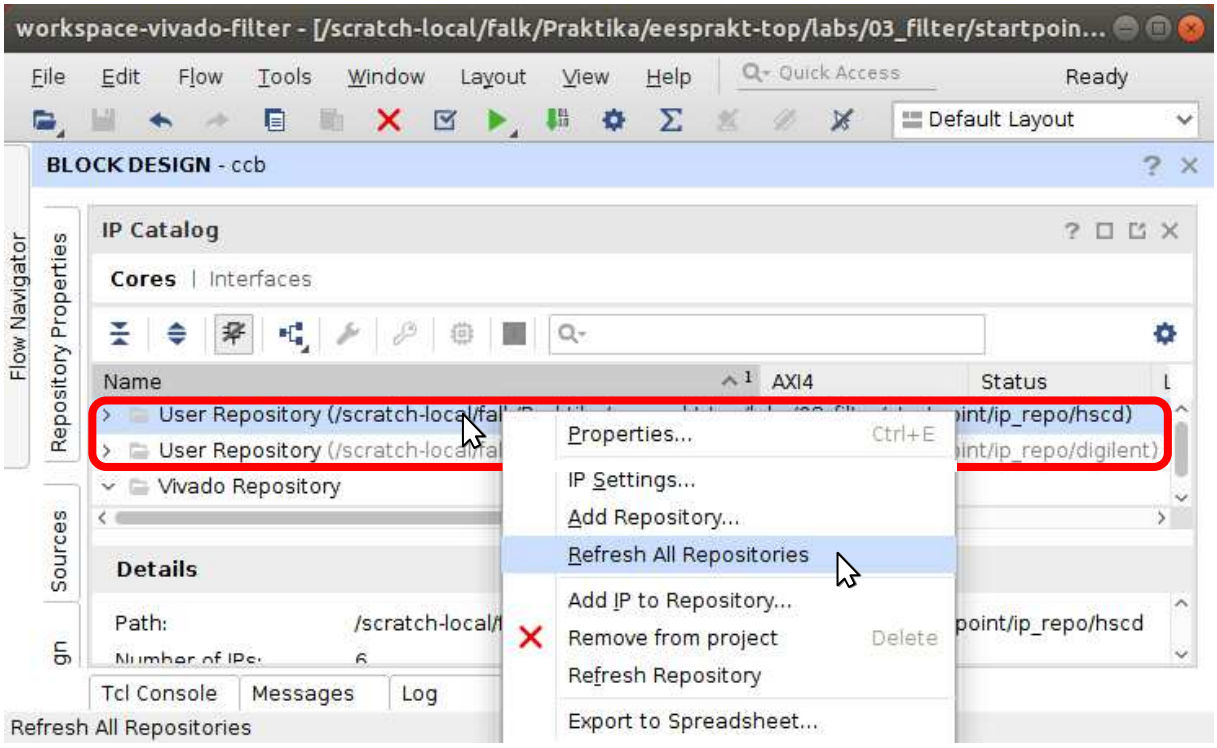


Figure 20: Refresh *IP repositories*

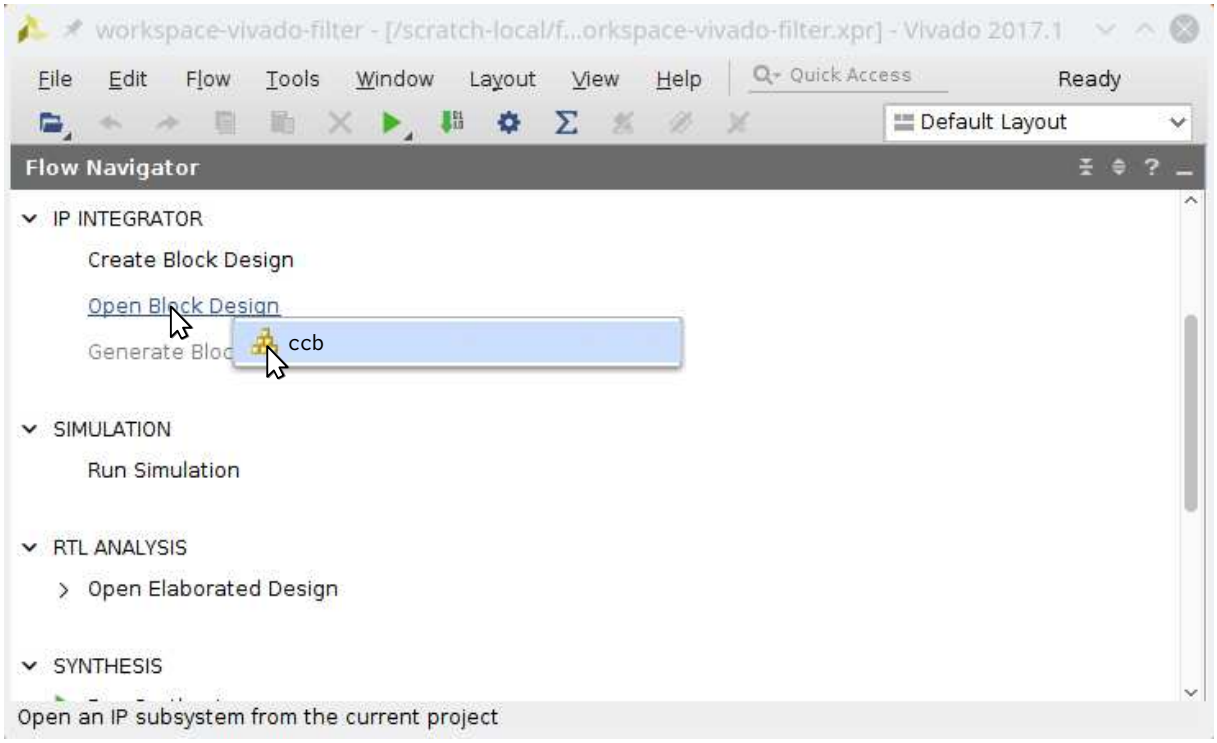


Figure 21: Open the ccb block design

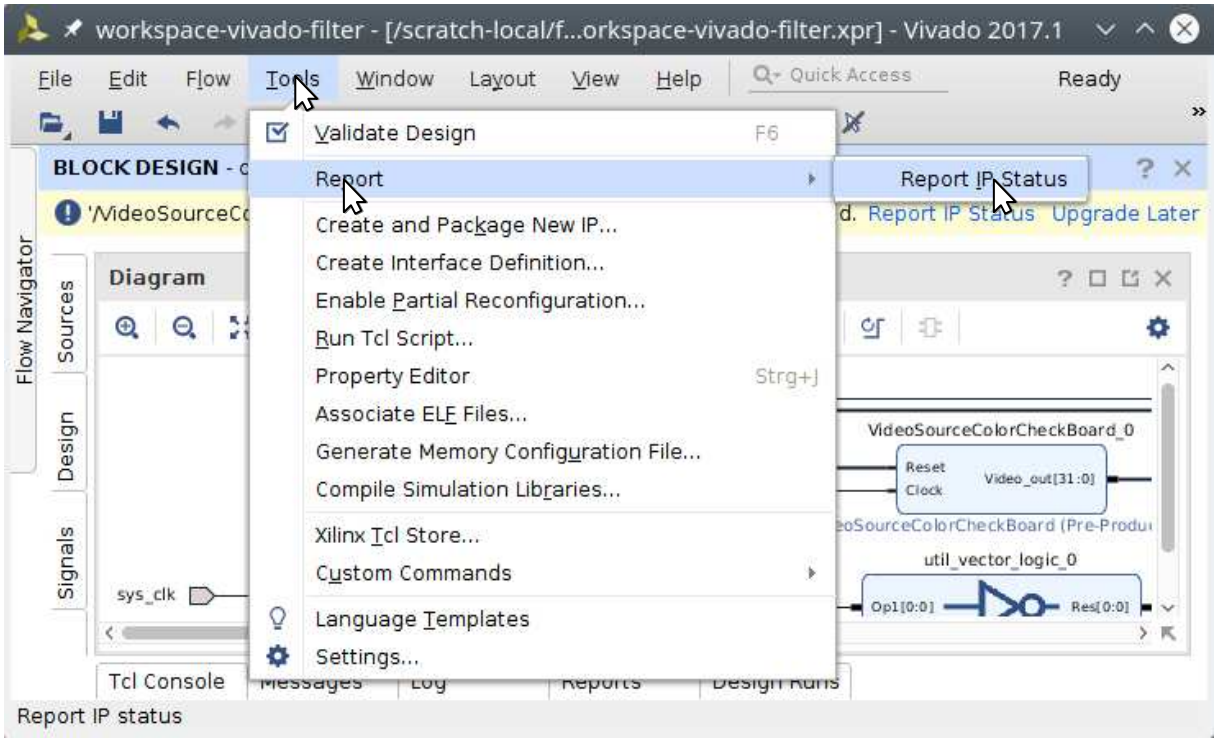


Figure 22: Check for newer version of IPBs by running *Report IP Status*

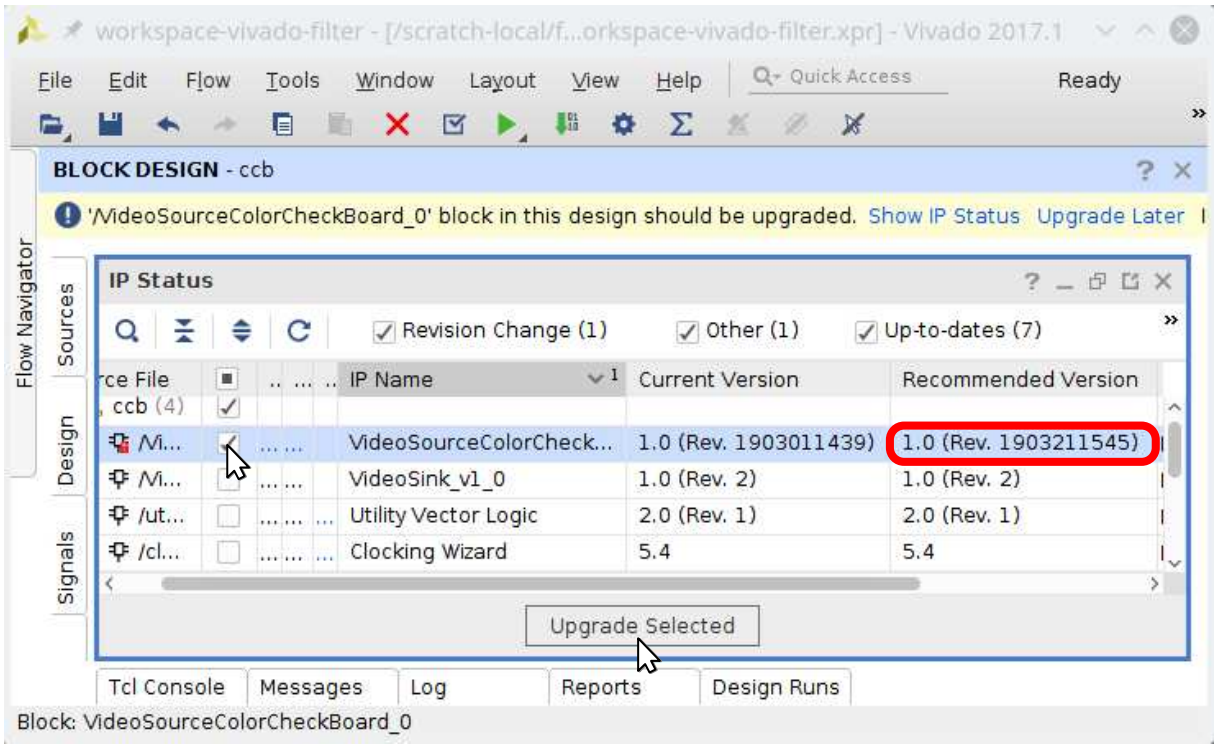


Figure 23: Upgrade IPBs to latest version

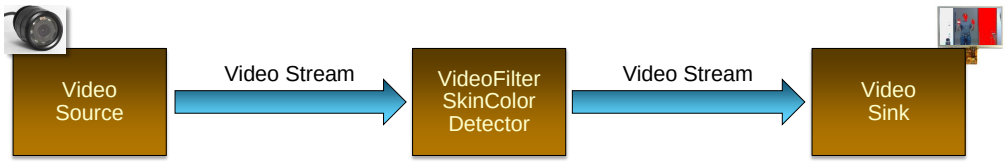
Task 1 (Simple Point Filter Implementation)

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 24b 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 24: 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.

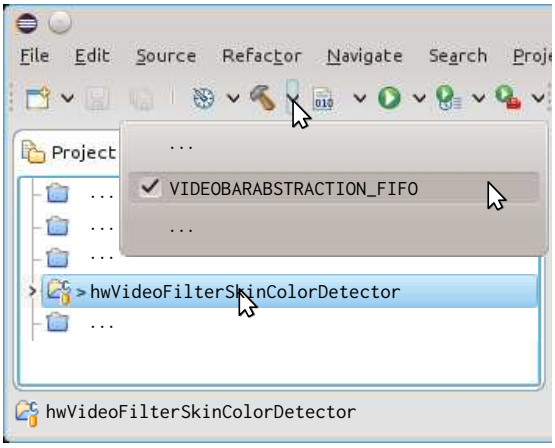
All filters can be compile in two different modes: (i) FIFO abstraction and (ii) RTL abstraction level. These levels are guarded by `#if VIDEOBARABSTRACTION == VIDEOBARABSTRACTION_FIFO` and `#if VIDEOBARABSTRACTION == VIDEOBARABSTRACTION_RTL`. In the following, you will implement functionality for the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

a) 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.

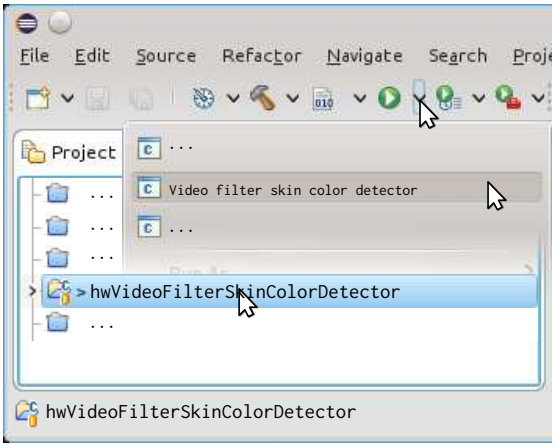
b) Define the `Video_in` and `Video_out` FIFO ports for the module. Take care to also name these ports accordingly in the constructor.

c) 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 25a) and, finally, start the simulation (see Figure 25b)

d) 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 25 to run and Figure 24a for the expected result).



(a) Compiling the filter at FIFO abstraction



(b) Running the simulation

Figure 25: Check the functionality of you 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.

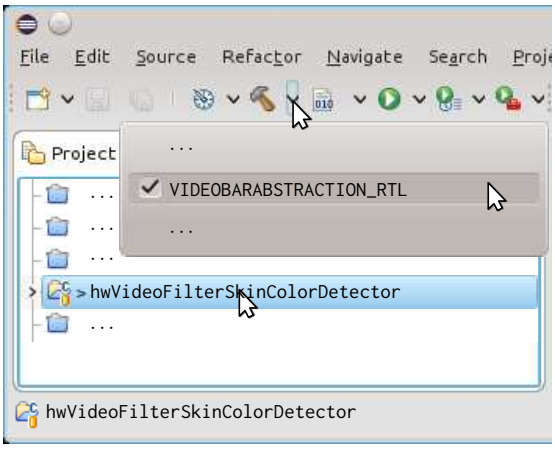
e) Define the `Clock` input port. This port must be connectable to a Boolean signal.

f) Define the `Video_in` and `Video_out` signal ports for the module.

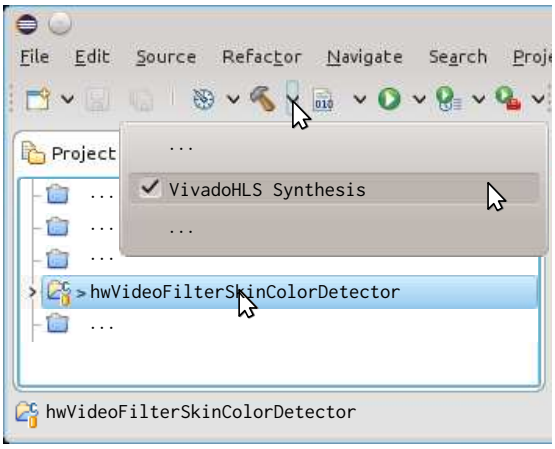
g) Change the `pixelThrd` process to a clocked and threaded process (`SC_CTHREAD`) for RTL abstraction level. The `pixelThrd` process should be sensitive to the rising edge of the `Clock` signal.

h) Define the `Reset` signal as an active-high reset for the `pixelThrd` process.

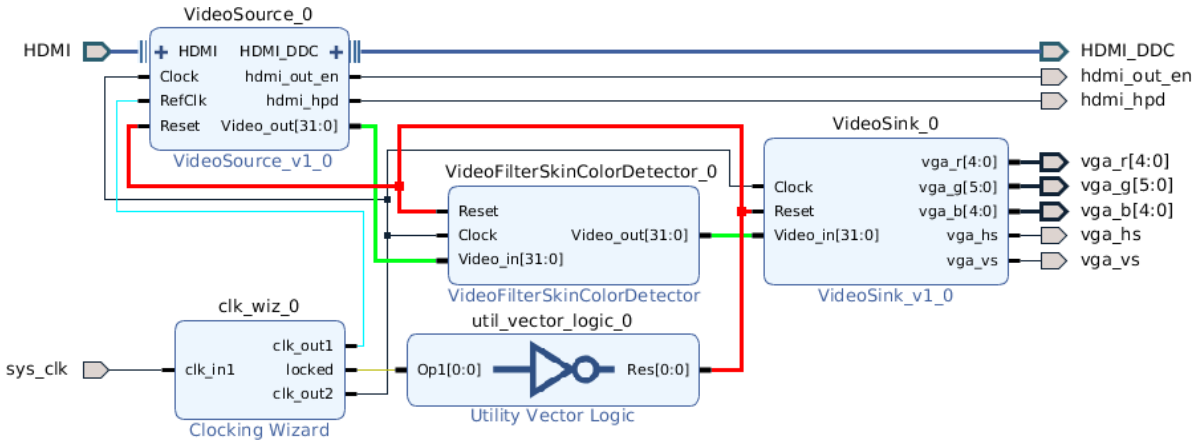
i) Change the `pixelThrd` process to also support `SC_CTHREAD` mode as well as reading and writing from the signal ports `Video_in` and `Video_out`. Then, check that the module can perform skin color detection at RTL abstraction level. For this purpose, first compile the `hwsim` project at RTL abstraction level.Then, compile this project at RTL abstraction level (see Figure 26a) and, finally, start the simulation (see Figure 25b)



(a) Compiling the filter at RTL abstraction



(b) Starting VivadoHLS to synthesize the `VideoFilterSkinColorDetector` into an IPB



(c) Vivado block design for the `VideoFilterSkinColorDetector` filter chain

Figure 26: Realizing the skin color detection filter chain on the ZCU102 board

Next, you will realize the skin color detection filter chain given in Figure 24b as the `skcd` block design on the ZCU102 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 26b. Check the *analysis* perspective (see Figure 16) 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 25b) to check that the synthesized result works.

l) Next, start Vivado and refresh the *IP repositories* as shown in Figure 20. 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 27).

n) Add the IPBs "VideoFilterSkinColorDetector", "VideoSource", "VideoSink", "Clocking Wizard", and "Utility Vector Logic" to the empty `skcd` block design (see Figure 28). This should result in a block design similar to the one depicted in Figure 29. However, the IPBs

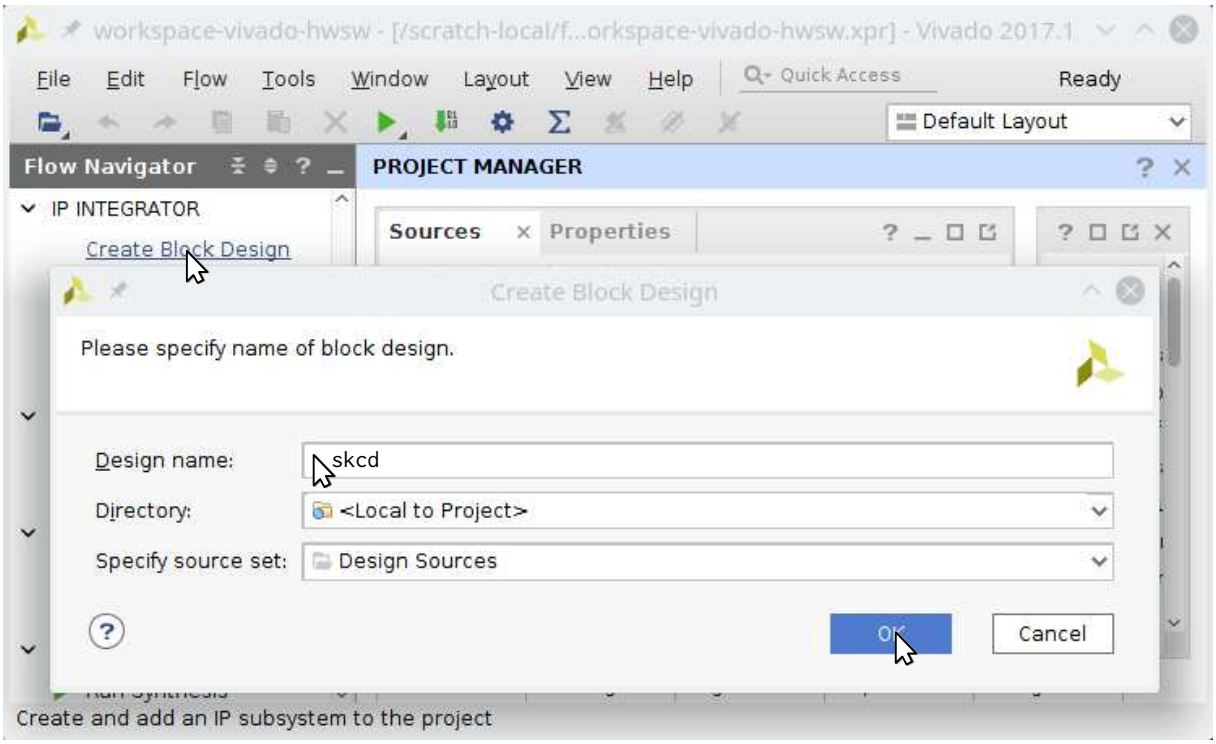


Figure 27: Create the skcd block design

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 30. 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. Thus, customize the "Clocking Wizard" IPB to convert from the system 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 31. Then, switch to the **Output Clocks** tab (see Figure 32) 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 32). 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 hight 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 33. 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 34. Subsequently, the remaining input and output ports have to be *made external* as these ports represent the interface to the ZCU102 board. Right click on the remaining input and output ports and select **Make External** from the pop-up menu. This should result in a block de-

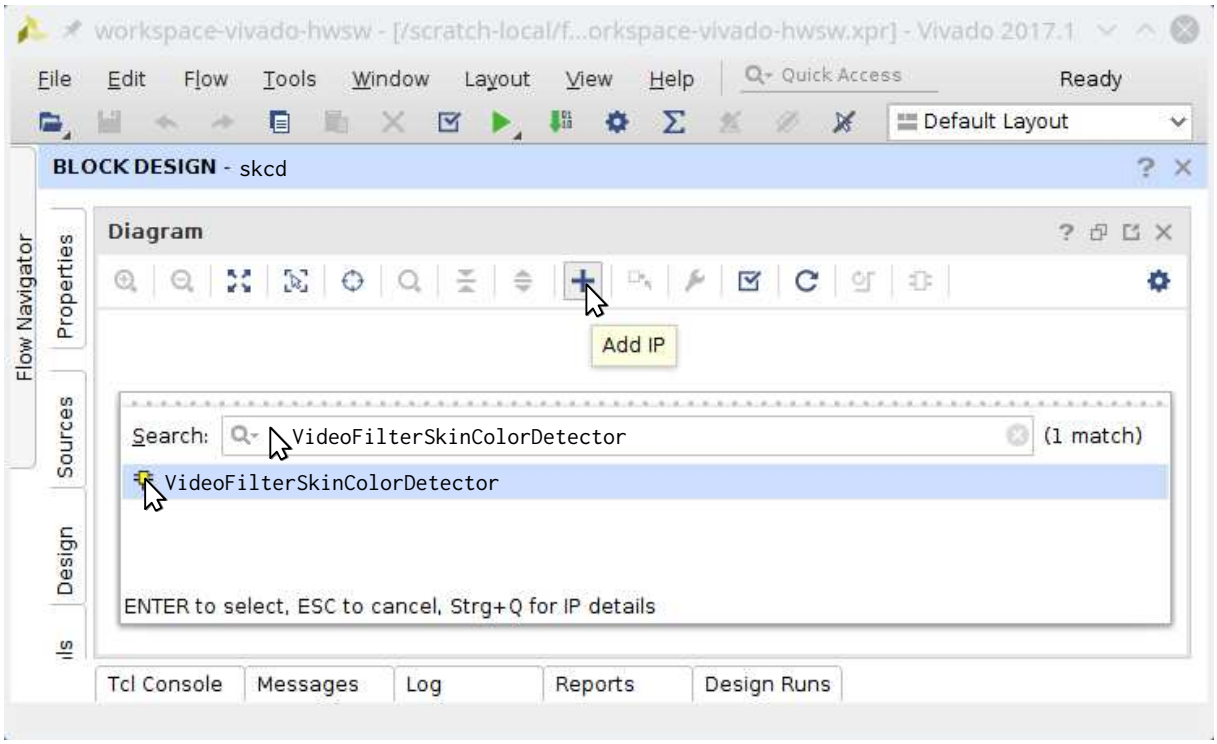


Figure 28: Add the IPB "VideoFilterSkinColorDetector" to the skcd block design

sign similar to the one depicted in Figure 13 with the exception that the `sys_clk` port is still named `clk_in1`.

s) Thus, rename (see Figure 35) 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. This constraint set contains the *ZCU102_Master.xdc xdc constraint file* (its absolute path is *~/03_filter/workspace-vivado-filter/ZCU102_Master.xdc*). This file contains commented out constraints for a significant fraction of the periphery the ZCU102 board has connected to the ZCU102 Multi-Processor System-on-Chip (MPSoC).

u) Enable the required constraints in the *ZCU102_Master.xdc* to give the pin location for each input and output port of the skcd block design.

v) Subsequently, create a HDL wrapper (see Figure 36) around the block design and set it as the top module (see Figure 37). If the *Set as Top* command is grayed out, then the wrapper is already the top module.

w) Start the synthesis and wait for completion to open the hardware manager for programming of the FPGA.

Next, you should test the video-in to video-out setup as outlined in Sections 1.4 and 1.5. If you are comfortable with this setup, you can continue to test you generated *skcd_wrapper.bit* bit file.

For this purpose, you have to program the ZCU102 MPSoC with the skcd block design. This should result in an output similar to Figure 24a on the monitor you connected to the DisplayPort output of the ZCU102 board.

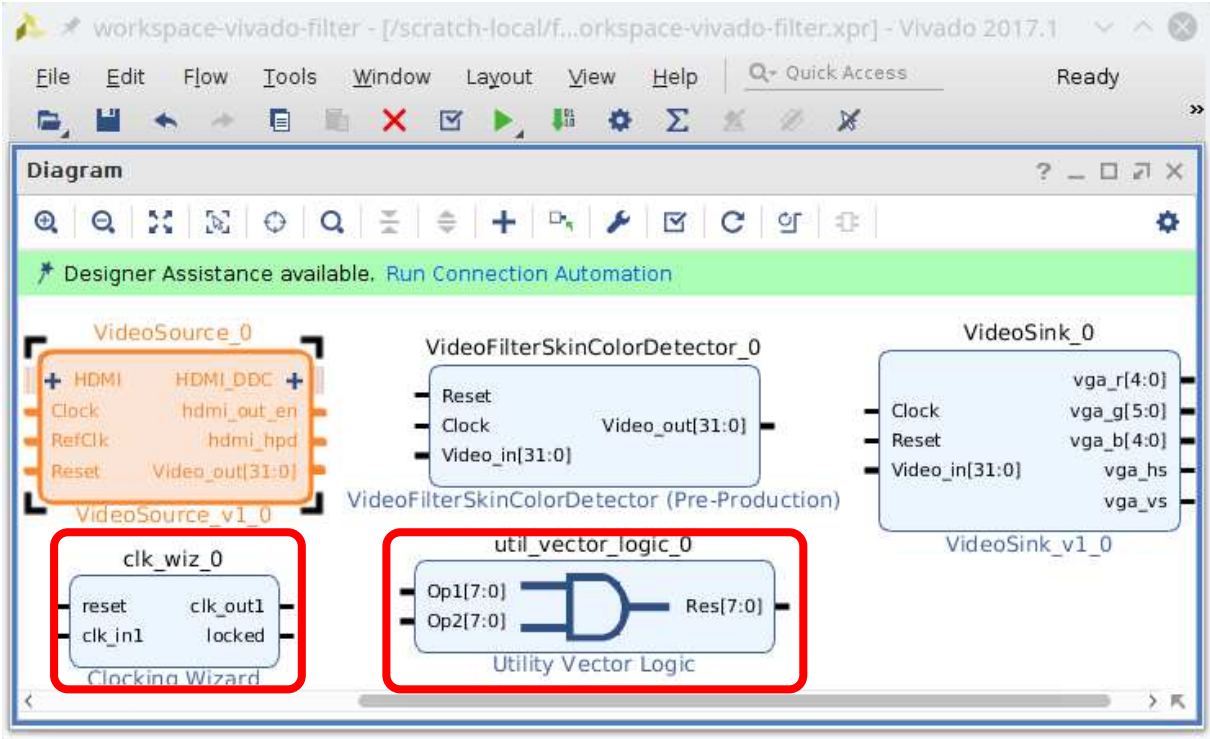


Figure 29: All IPBs for the skcd block design

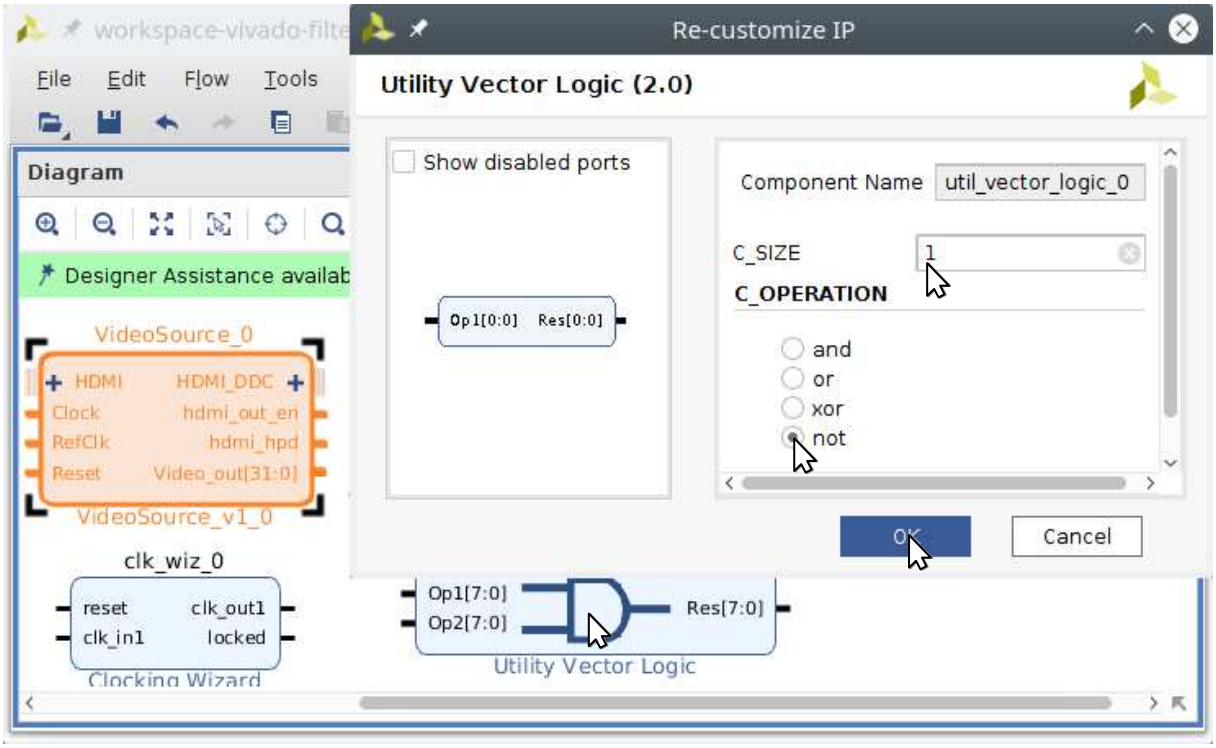


Figure 30: Re-customized “Utility Vector Logic” IPB to be a one bit inverter

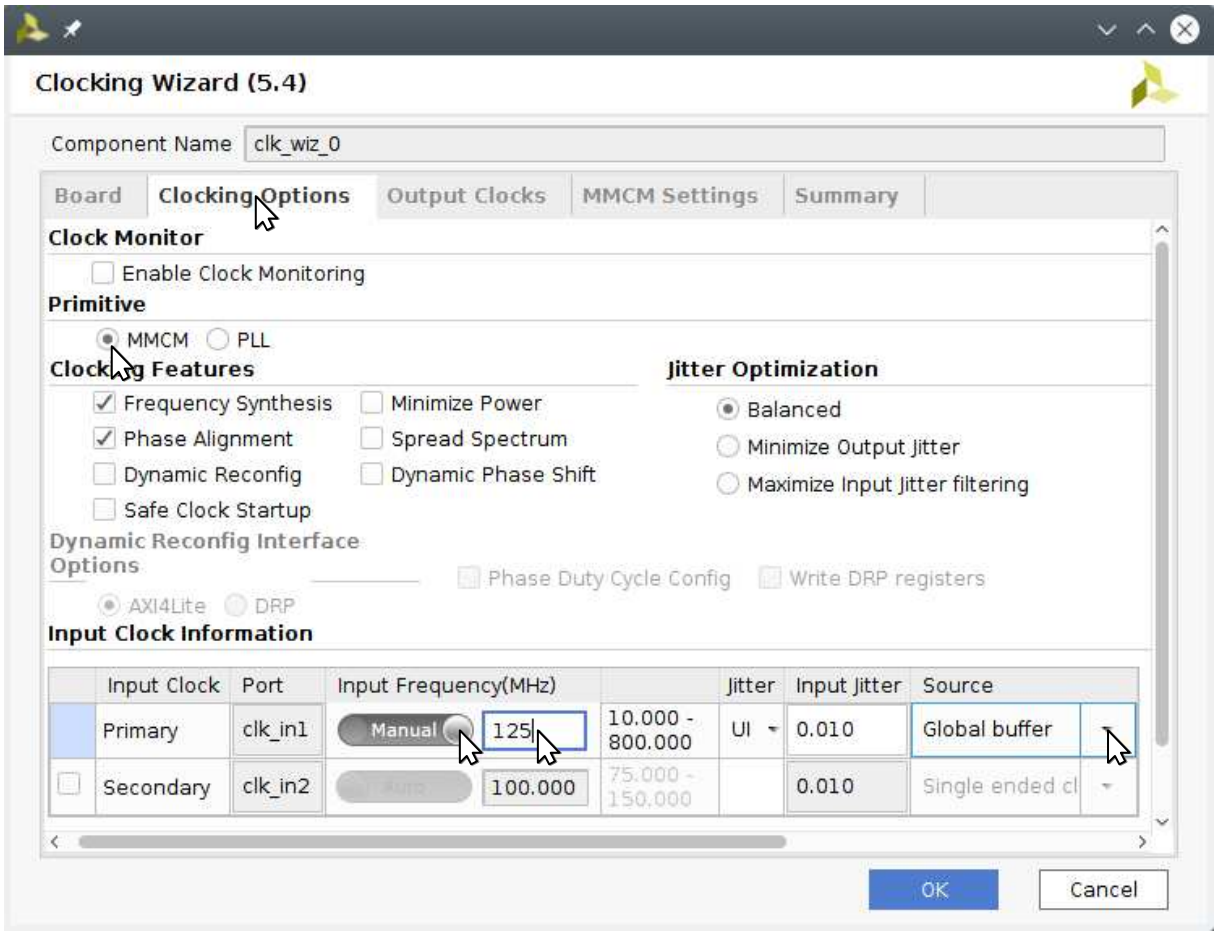


Figure 31: Re-customized "Clocking Wizard" to have a 125 MHz input clock

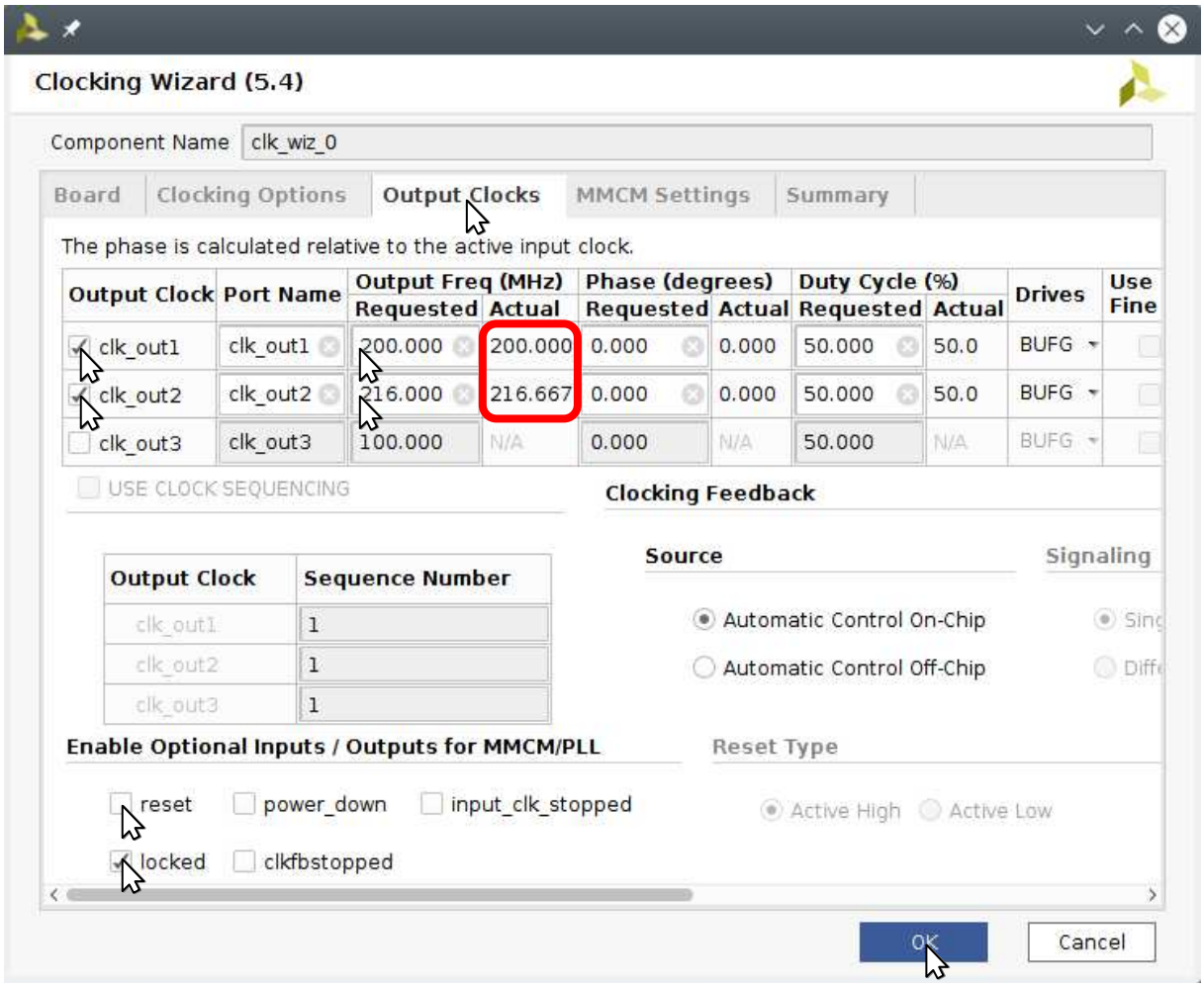


Figure 32: Re-customized "Clocking Wizard" to have 200 MHz and 216 MHz output clocks as well as disable the reset input

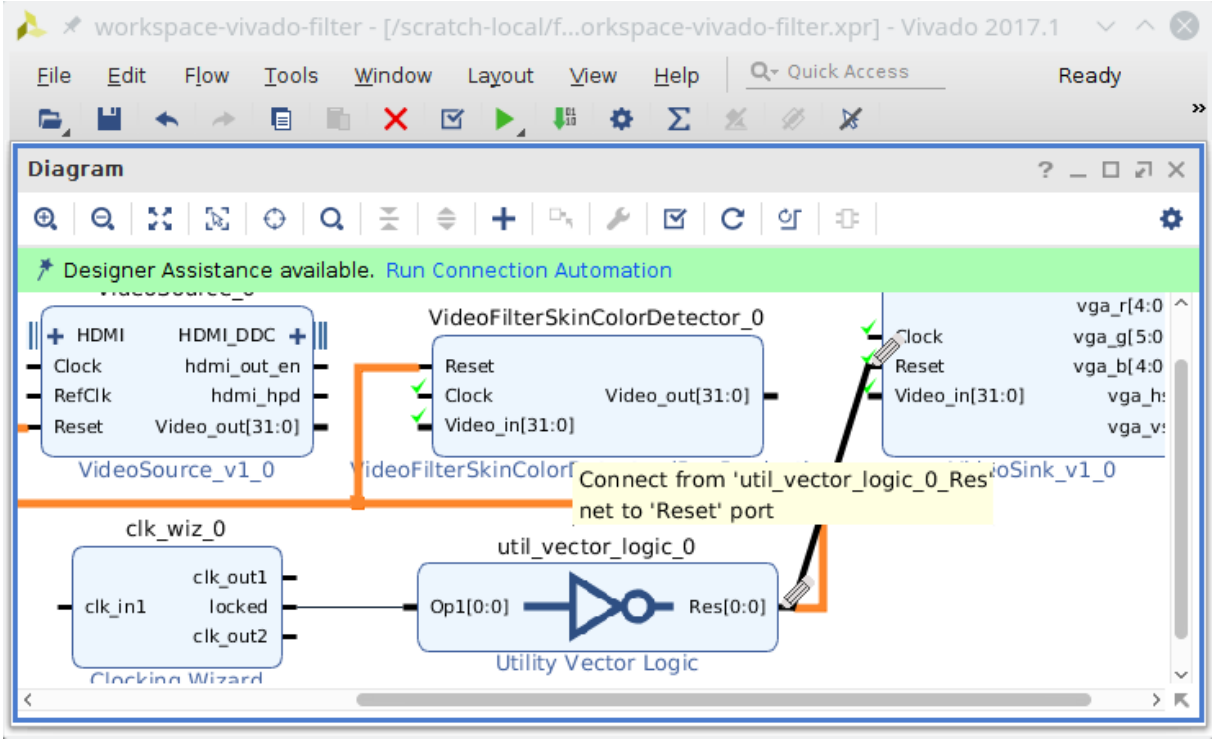


Figure 33: Connect the reset derived by inverting the `locked` output port of the "Clocking Wizard" to the `Reset` input ports of the IPBs

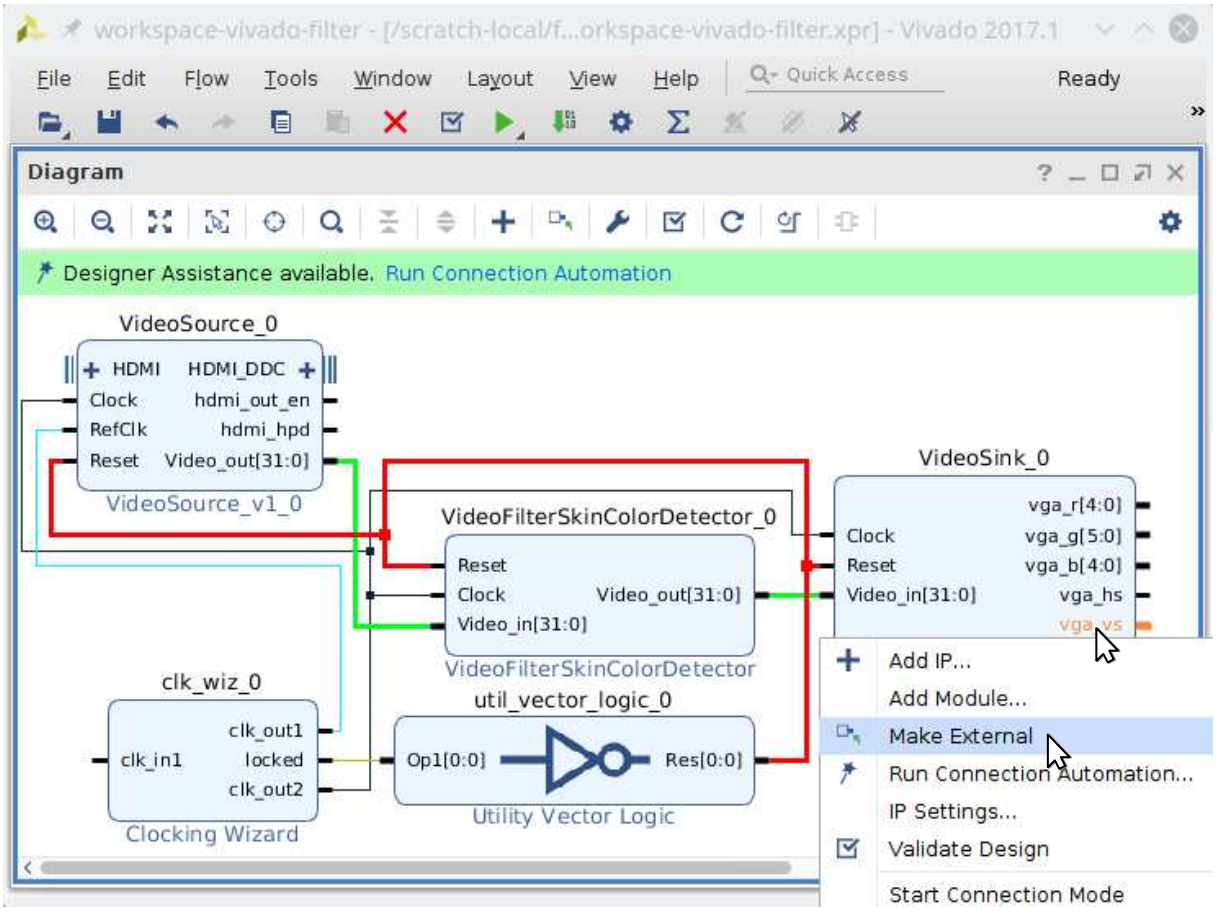


Figure 34: Make the inputs and outputs of the block design external

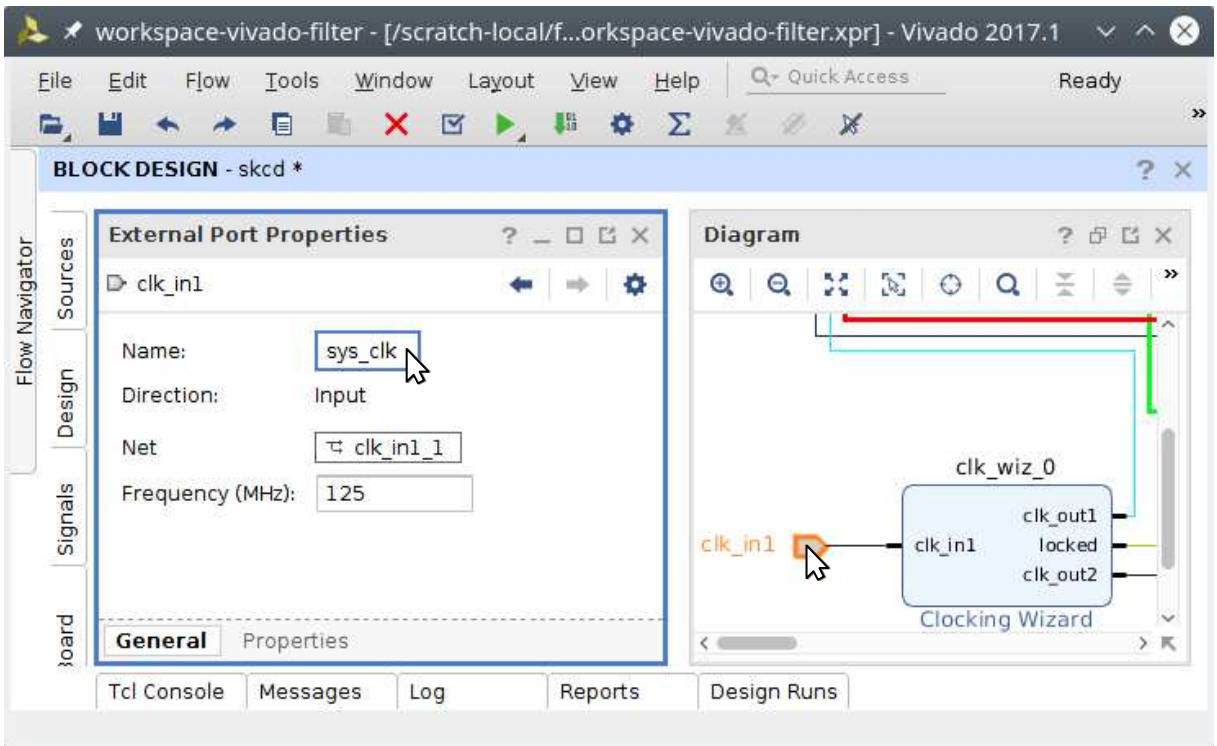


Figure 35: Renaming the `clk_in1` input port to `sys_clk`

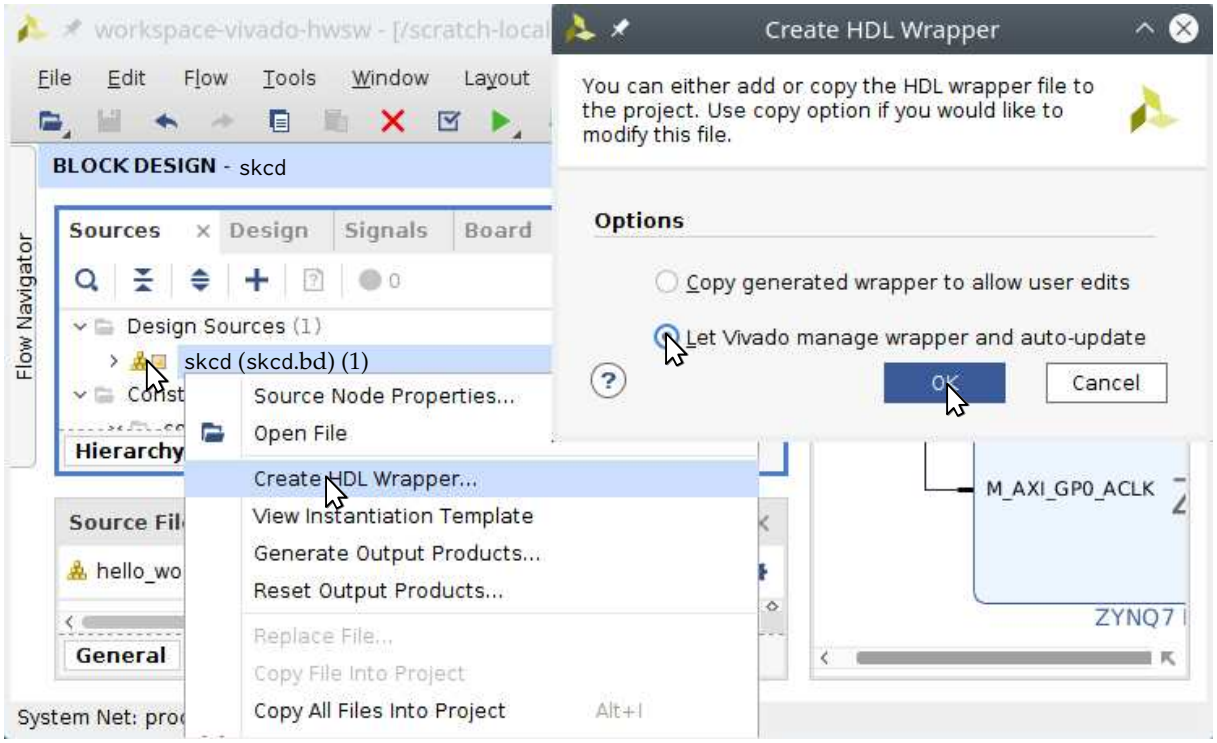


Figure 36: Create a HDL wrapper around the block design

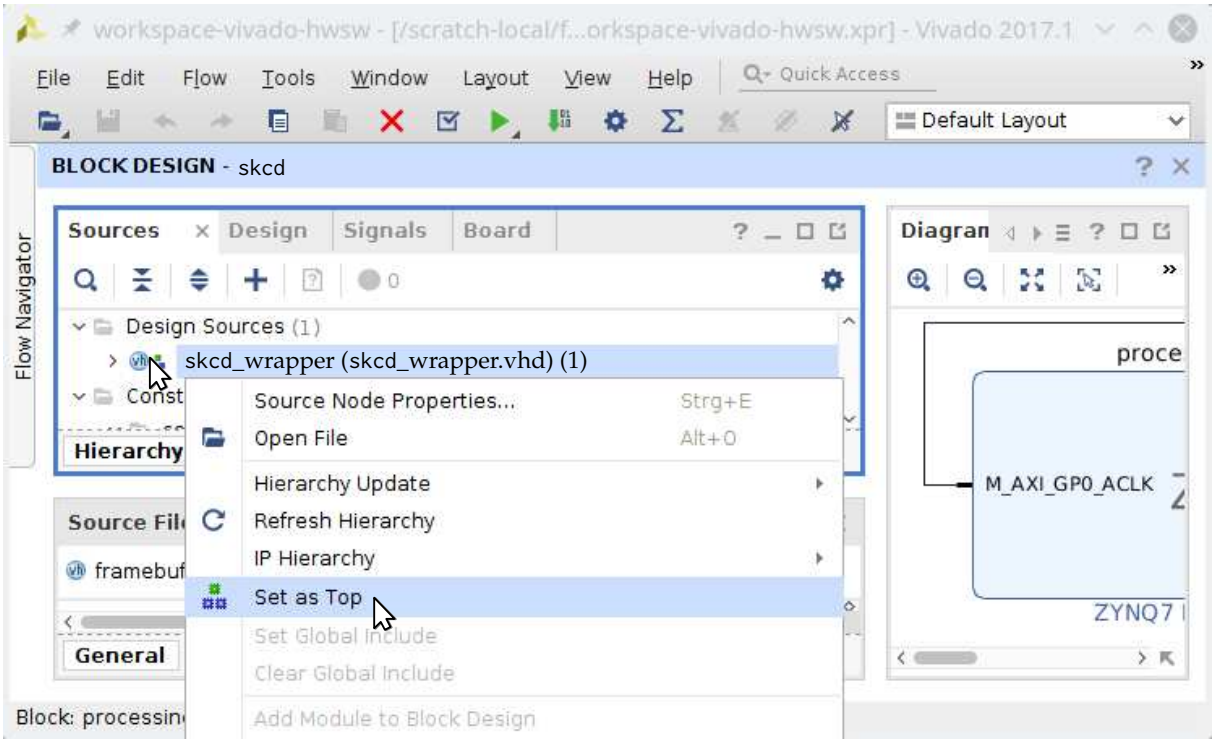


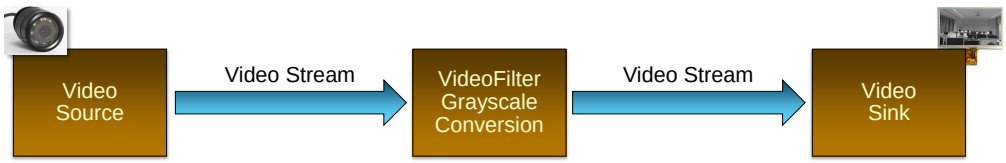
Figure 37: Set the wrapper as top module for synthesis

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 38b 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 38: 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 ZCU102 board by realizing and synthesizing the `gray` block design shown in Figure 39.

j) Follow the tutorial in Task 1 to also test the gray scale filter on the ZCU102 board.

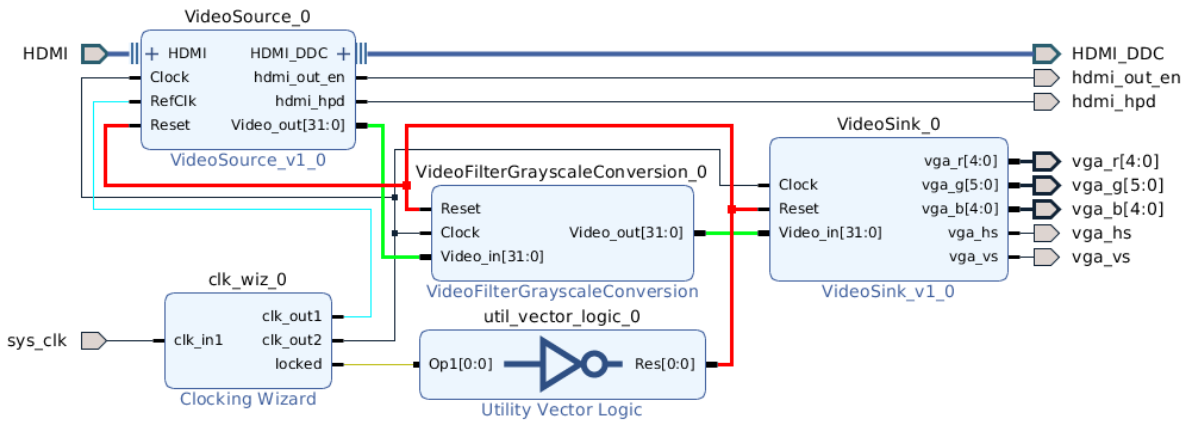
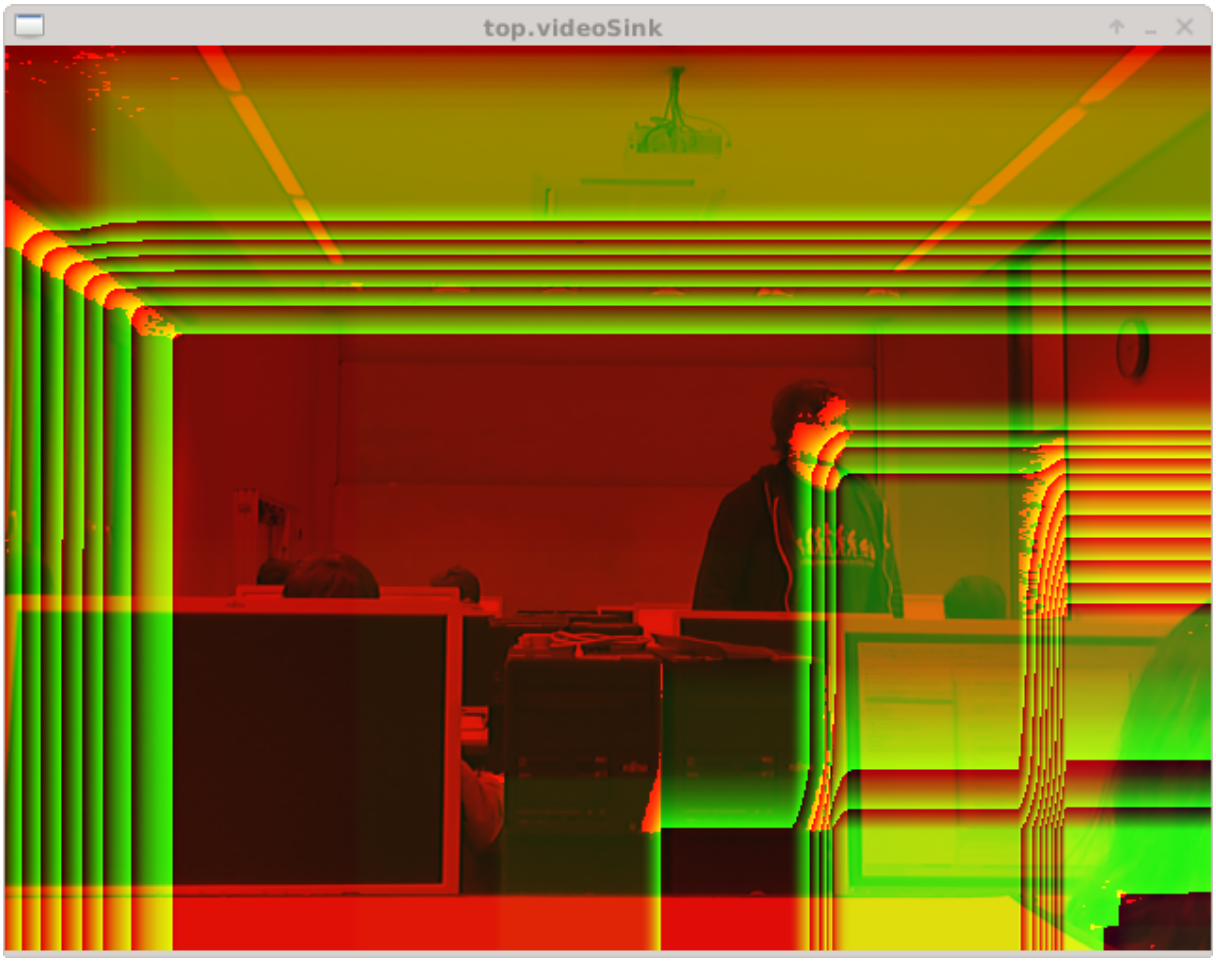


Figure 39: Vivado block design for the `VideoFilterGrayscaleConversion` filter chain

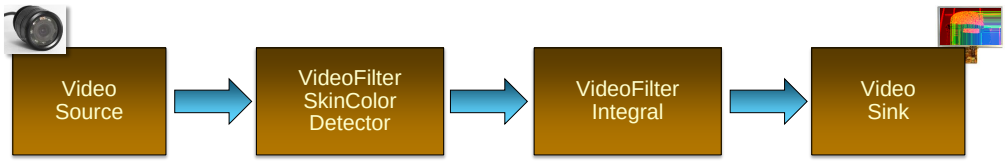
Task 3 (Using Memories in HLS)

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 40b) 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 40: Architecture and expected result for the integral video filter chain

will develop this filter at the FIFO abstraction level.

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.

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 40a.

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 ZCU102 board by realizing and synthesizing the `integral` block design shown in Figure 41.

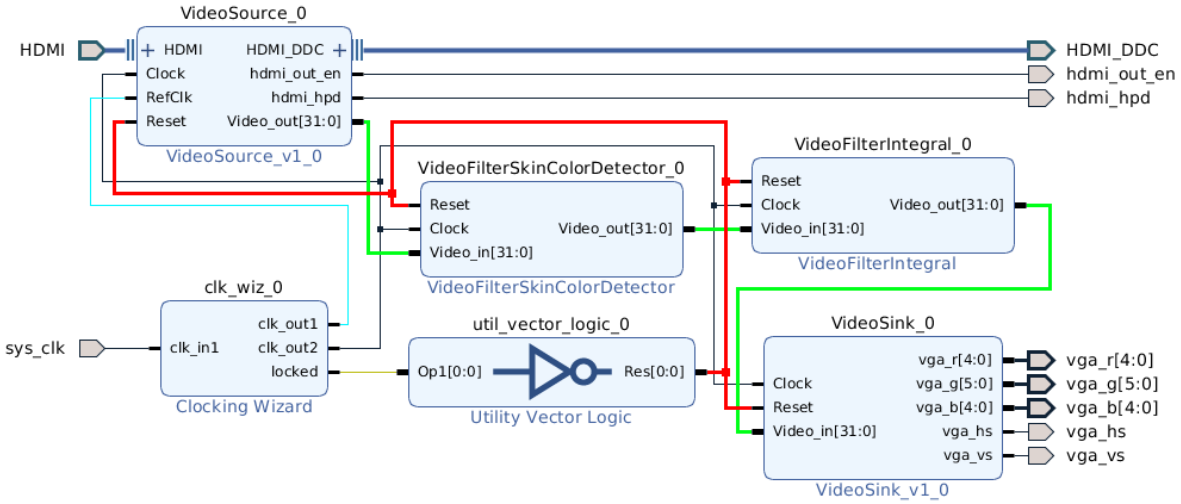


Figure 41: Vivado block design for the skin color detector and integral filter chain

m) Follow the tutorial in Task 1 to also test the integral filter on the ZCU102 board. Don't forget that the skin color filter has to be put in front of the integral filter.