Lehrstuhl für Informatik 12
Cauerstraße 11
91058 Erlangen

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

5. Exercise of the Laboratory
Entwicklung interaktiver eingebetteter Systeme

1 Introduction

In the following, you will get familiar with the integration of hardware accelerators into SystemC-based virtual prototypes using Transaction-Level Modeling (TLM) as well as real implementations. Please open the prepared workspace by starting eclipse and choosing the directory depicted in Figure 1. The current lab combines the labs `02_sysc`, `03_filter`, and `04_tlm`. Solutions

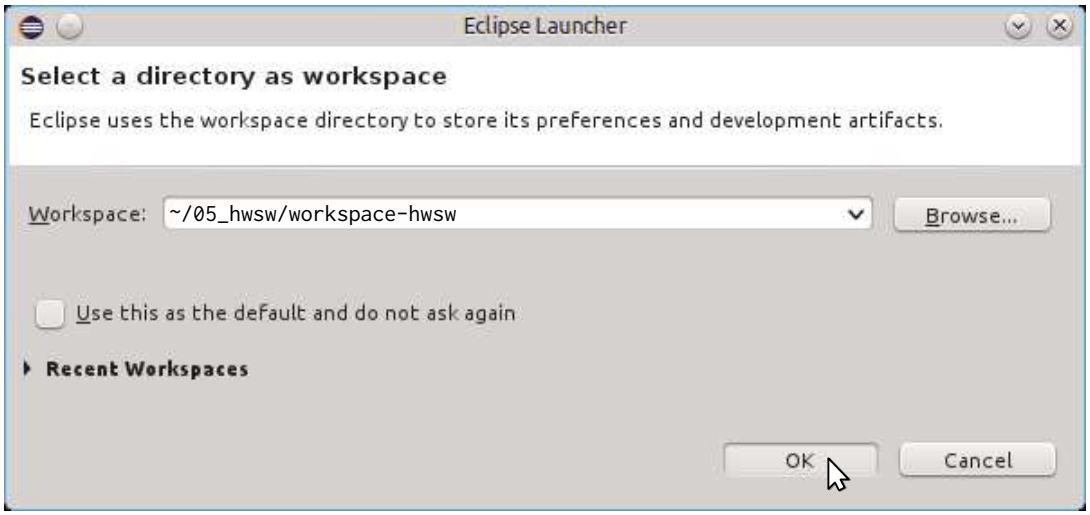


Figure 1: The eclipse workspace for the current lab

for all task of the previous labs are present. If you trust your solutions from these labs, you can copy over the below given files from the specified labs.

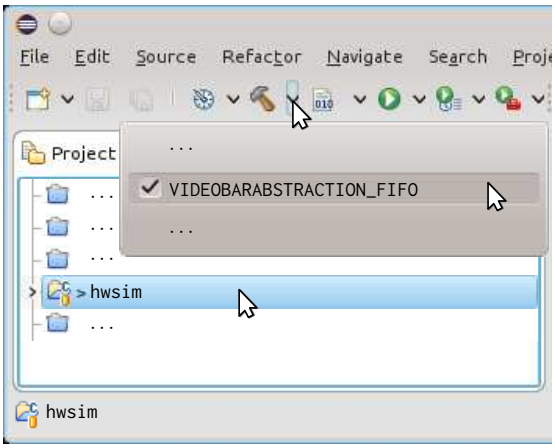
- From *~/02_sysc/workspace-sysc*
  `hwsim/src/hwsim/cpp/VideoAdapterFIFOToRTL.hpp`
  `hwsim/src/hwsim/cpp/VideoAdapterFIFOToRTL.cpp`
  `hwsim/src/hwsim/cpp/VideoAdapterRTLToFIFO.hpp`
  `hwsim/src/hwsim/cpp/VideoAdapterRTLToFIFO.cpp`
- From *~/03_filter/workspace-filter*
  `hwVideoFilterSkinColorDetector/src/module/cpp/VideoFilterSkinColorDetector.hpp`
  `hwVideoFilterSkinColorDetector/src/module/cpp/VideoFilterSkinColorDetector.cpp`
  `hwVideoFilterIntegral/src/module/cpp/VideoFilterIntegral.hpp`
  `hwVideoFilterIntegral/src/module/cpp/VideoFilterIntegral.cpp`
- From *~/04_tlm/workspace-tlm*
  `hwsim/src/hwsim/headers/hwsim/SimpleMem.hpp`
  `hwsim/src/hwsim/cpp/SimpleMem.cpp`

```
hwsim/src/hwsim/cpp/Bus.hpp
hwsim/src/hwsim/cpp/Bus.cpp
```

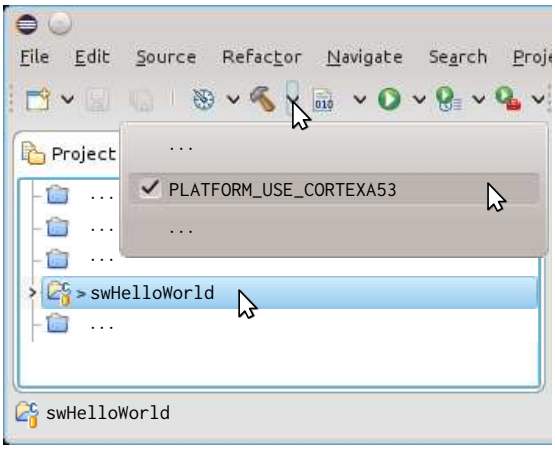
To test the basic functionality of the simulation environment, compile the `hwsim` simulator and simulated the "Hello World"-program with the steps given in Figure 2.

Briefly, the following projects are of relevance for understanding the lab:

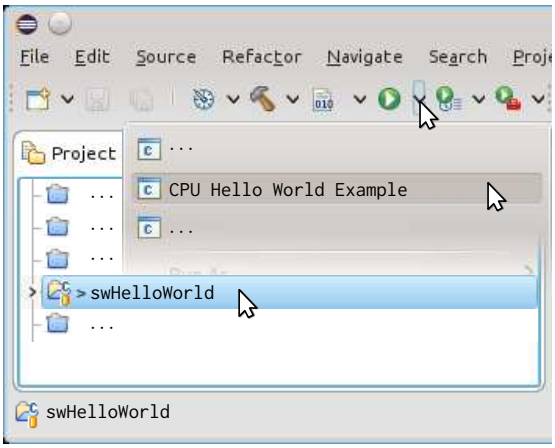
1. `hwsim` – A SystemC simulation environment containing the SystemC module `ZynqUS`, which wraps an Instruction Set Simulator (ISS) from Imperas™ for a dual core ARM™ Cortex A9, the `SimpleMem` SystemC module for memory modeling, the `Bus` module to connect multiple memories or hardware accelerators to the CPU, as well as different video sinks, sources, and filters with and without connection to the *system bus*, i.e., the bus that is also used to connect the memories.
2. `hwVideoSourceFromMem` – A video source that can convert a raw RGB image from memory into a video stream. The dimension of the image as well as the memory address from where to retrieve the image have to be given to the hardware module via its *register file*. Register files are accessible from the CPU via the system bus.
3. `hwVideoSinkToMem` – A video sink that writes a frame from the video stream as a raw RGB image to memory. The address where the hardware module will write the raw RGB image to memory must be specified via its *register file*. Moreover, you also have to specify the size of the buffer that is reserved to store the incomming image. Finally, the dimension of the image can be read from the register file as well as a counter how many images have already been stored to memory. Here, new frames will overwrite the old image in memory.
4. `hwVideoFilterIntegral` – The integral area sum filter from the `03_filter` lab.
5. `hwVideoFilterSkinColorDetector` – The skin color filter from the `03_filter` lab.
6. `hwVideoFilterSkinColorDetectorWithCtrl` – A point filter for skin color detection that can be parameterized via its *register file*.



(a) Compiling the *hwsim* simulator



(b) Compiling the "Hello World"-program



(c) Run the "Hello World"-program

Figure 2: Simulating the "Hello World"-program with the TLM-based MPSoC simulator `hwsim`

7. `hwVideoSourceColorCheckBoardWithCtrl` – A video source to generate a test image. Properties of the test image can be parameterized via its *register file*.
8. `swHelloWorld` – A simple "Hello World" program that should be executed on the simulated ARM CPU provided by the `hwsim` simulator.
9. `swParticleFilter` – The *libParticleFilter* library and corresponding test program to detect skin color blobs in the video input by applying a particle filter on a preprocessed video stream derived by using the filter `hwVideoFilterSkinColorDetectorWithCtrl` as well as the `hwVideoFilterIntegral` filter as preprocessing steps.
10. `swVideoFilterSkinColorDetectorWithCtrl` – The *libCalibrateSKCD* library and corresponding test program to calibrate the `hwVideoFilterSkinColorDetectorWithCtrl` filter to detect a color specified during calibration.
11. `swVideoSourceColorCheckBoardWithCtrl` – A library and corresponding test program to parameterize the `hwVideoSourceColorCheckBoardWithCtrl` image source to generate different test images.
12. `swVideoSourceFromMem` – The *libVideoSourceFromMem* library and corresponding test program to use the `hwVideoSourceFromMem` video source to display images from memory.
13. `swVideoSourceToSinkOverMem` – The *libVideoSinkToMem* library and corresponding test program to use the `hwVideoSinkToMem` video sink to store input images in the memory.

The goal of the current lab is to learn about *register-file*-based HW/SW communication (See Figure 3). All SystemC modules ending with the postfix *WithCtrl* can be assumed to have a register file. Moreover, the SystemC modules `VideoSourceFromMem` and `VideoSinkToMem` also have a register file containing their control registers. For *register-file*-based HW/SW communication, we use *memory mapped I/O*, i.e., the register file is connected to the system bus and given a normal address. Then, these registers can be access no different than accesses to values in the main memory.

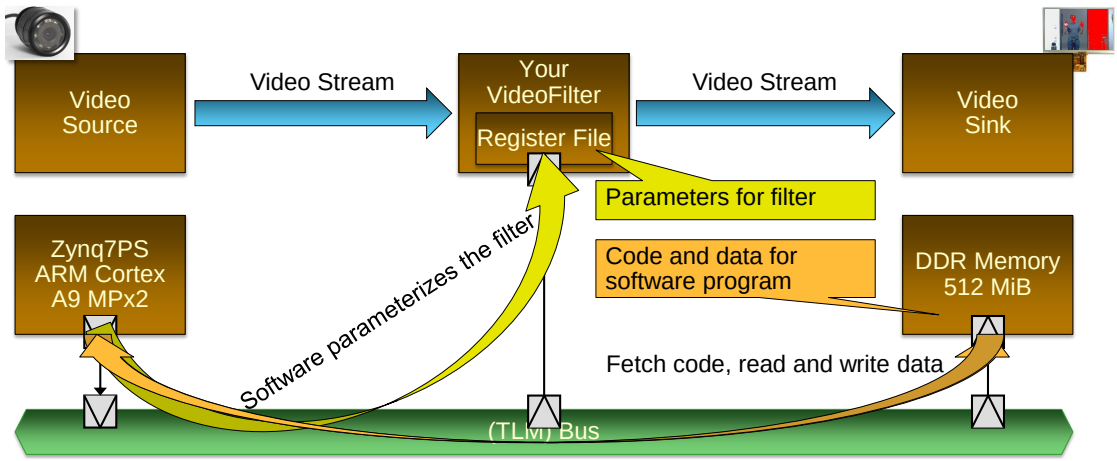


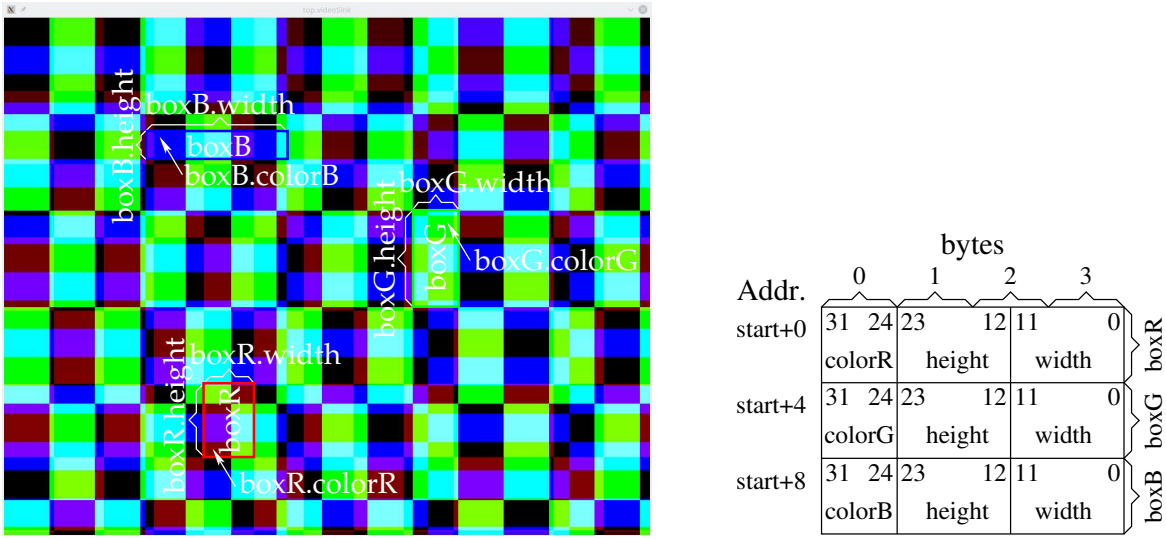
Figure 3: System containing a video filter parameterizable via a register file.

Apart from register-file-based communication, which is mainly used for low bandwidth control tasks, there is also high bandwidth DMA-based HW/SW communication supported by the `hwsim` simulator. The DMA-based HW/SW communication is used by the SystemC modules `VideoSourceFromMem` and `VideoSinkToMem` (see Figures 19 and 23c) to read or write the video data from or to the main memory, respectively. Briefly, in register-file-based communication, the hardware accelerator module is only required to implement a bus slave interface and is, thus, relatively easy and cheap to realize. In DMA-based HW/SW communication, a DMA engine acts as a bus initiator that can transfer data to and from main memory without using the CPU

to transfer the data. While this enables high performance, it also is costly in terms of hardware resources.

1.1 Register-File-Based HW/SW Communication

In order to introduce you to register-file-based HW/SW communication, we have extended the `VideoSourceColorCheckBoard` from the *filter* lab with a register file that controls the color and dimensions of the three different check board patterns generated by this hardware module. See Figure 4a for an example of the resulting effect. This extended hardware module has been renamed to `VideoSourceColorCheckBoardWithCtrl` and can be found in the project `hwVideoSourceColorCheckBoardWithCtrl`. What registers the register file contains and where these are located relative to the *start address* of the register file, e.g., the register file layout information shown in Figure 4b, is controlled by a control structure. As can be seen in Figure 4b, the



(a) Resulting color check board

(b) Layout of the register file

Figure 4: Parameterizing the color check board via its register file

register file consists of 12 bytes that are partitioned into three 32-bit integers in *big endian* format, i.e., high byte first. Each integer is responsible for one check board pattern. To specify the three values, i.e., width, height, and color, in the 32-bit integer, the integer itself is partitioned into bit ranges. The bit range 24 to 31 controls the color of the check board pattern, the bit range 12 to 23 controls the height of the boxes in the check board pattern, and the bit range 0 to 11 controls the width of the boxes in the check board pattern.

As mentioned before, the register file layout is defined via a control structure. For the considered example, this is the `VideoSourceColorCheckBoardWithCtrl::Control` structure defined in the *sw/VideoSourceColorCheckBoardWithCtrl.hpp* header contained in the *src/module/headers* directory of the `hwVideoSourceColorCheckBoardWithCtrl` project. The control structure is declared as follows:

```
struct Control {
  union {
    Register<0,  12, int16_t, BIG_ENDIAN, 32> width;
    Register<12, 12, int16_t, BIG_ENDIAN, 32> height;
    Register<24,  8, uint8_t, BIG_ENDIAN, 32> colorR;
  } boxR;
```

```
    union {
      Register<0,  12, int16_t, BIG_ENDIAN, 32> width;
      Register<12, 12, int16_t, BIG_ENDIAN, 32> height;
      Register<24,  8, uint8_t, BIG_ENDIAN, 32> colorG;
    } boxG;
    union {
      Register<0,  12, int16_t, BIG_ENDIAN, 32> width;
      Register<12, 12, int16_t, BIG_ENDIAN, 32> height;
      Register<24,  8, uint8_t, BIG_ENDIAN, 32> colorB;
    } boxB;
  };
```

The `unions` `boxR`, `boxG`, and `boxB` correspond to the three 32-bit integers in big endian format. The bit ranges within a 32-bit integer are represented using the `Register` helper class defined in the header *sw/Register.hpp* from the `hwsim` project. In more detail, the `boxR.width`, `boxR.height`, and `boxR.colorR` members all access the same 32-bit big endian integer as they are contained in the same union, i.e., the `union` `boxR`. However, the three members are distinct in what bit ranges of the 32-bit big endian integer they will return, e.g., `width.get()` will return the bit range 0 to 11, or modify, e.g., `width.set(...)` will set the bit range 0 to 11. In general, an instance of `Register<N,M,T,E,R>` represents a subfield of type T starting at bit position N with bit length M contained in an R-bit integer of endianness E.

The register file described above will be used by software to control the hardware module `VideoSourceColorCheckBoard` as outlined in Figure 5. In the following, this design will be called

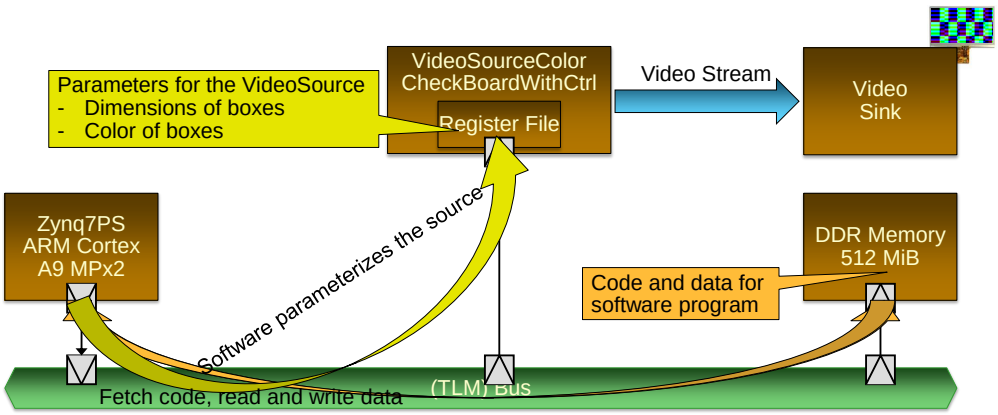


Figure 5: System containing a parameterizable color check board video source

the Color Check Board with ConTRoL (CCBCTRL) design. In detail, the following files and projects comprise the CCBCTRL design:

Project: `swVideoSourceColorCheckBoardWithCtrl`
Files: *src/main/cpp/main.cpp*
src/libVideoSourceColorCheckBoardWithCtrl/headers/libVideoSourceColorCheckBoardWithCtrl.hpp
src/libVideoSourceColorCheckBoardWithCtrl/cpp/libVideoSourceColorCheckBoardWithCtrl.cpp
Project: `hwVideoSourceColorCheckBoardWithCtrl`
Files: *src/module/headers/sw/VideoSourceColorCheckBoardWithCtrl.hpp*
src/module/cpp/VideoSourceColorCheckBoardWithCtrl.hpp
src/module/cpp/VideoSourceColorCheckBoardWithCtrl.cpp

In order to control the hardware, the software has to know where the register file is mapped into the address space of the CPU the software is running on. This address is given by the define `VIDEOSOURCECOLORCHECKBOARDWITHCTRL_CTRLADDR` declared by the project `hwVideoSourceColorCheckBoardWithCtrl` in the header *sw/VideoSourceColorCheckBoardWithCtrl.hpp*. This address, i.e., `0x43C30000`, corresponds to the `start` offset shown in Figure 4b and specifies where the register file resides in the address space of the CPU. The software uses this defined to declare the reference `ctrlCCB` of type `VideoSourceColorCheckBoardWithCtrl::Control` that refers to the memory given by the `VIDEOSOURCECOLORCHECKBOARDWITHCTRL_CTRLADDR` address. The relevant code in the *libVideoSourceColorCheckBoardWithCtrl.hpp* header is replicated below:

```
static VideoSourceColorCheckBoardWithCtrl::Control &ctrlCCB =
  *reinterpret_cast<VideoSourceColorCheckBoardWithCtrl::Control *>
    (VIDEOSOURCECOLORCHECKBOARDWITHCTRL_CTRLADDR);
```

With the above declaration, the software can use statements as follows, e.g., as used by the software *src/main/cpp/main.cpp* of the `swVideoSourceColorCheckBoardWithCtrl` project, to parameterize the hardware module:

```
...
// Assign random box size
ctrlCCB.boxR.width.set(256.0 * rand() / RAND_MAX + 1);
ctrlCCB.boxR.height.set(256.0 * rand() / RAND_MAX + 1);
// Assign red to be at full brightness
ctrlCCB.boxR.colorR.set(255);
...
```

These statements will modify the contents of the memory region from `0x43C30000` to `0x43C30003`, e.g., `ctrlCCB.boxR.colorR.set(255)` will set the byte at address `0x43C30000` to `0xFF`. However, the modified bytes do not reside in the DDR memory but are instead represented by registers in the `VideoSourceColorCheckBoardWithCtrl` hardware module. Thus, modifying the contents of the memory region from `0x43C30000` to `0x43C3000B` (12 bytes) will alter the registers in the hardware module and, hence, influence the behavior of the module.

Remember, there exists two different modes for the video signal, i.e., (i) FIFO abstraction and (ii) RTL abstraction as guarded by `#if VIDEOBARABSTRACTION == VIDEOBARABSTRACTION_FIFO` and `#if VIDEOBARABSTRACTION == VIDEOBARABSTRACTION_RTL`, respectively. Moreover, in this lab, you can also choose between two abstraction levels for the memory bus, i.e., (i) *TLM* and (ii) the *AXI-Lite bus protocol*, which is used at RTL and also the bus protocol used on the real hardware to connect the *Zynq UltraScale+ MPSoC* to the hardware module. These abstraction levels for the memory bus are guarded by `#if BUSABSTRACTION == BUSABSTRACTION_TLM` and `#if BUSABSTRACTION == BUSABSTRACTION_RTL_AXILITE`, respectively. Which abstraction levels are used for the video signal and memory bus depends on the compilation target you choose for the hardware accelerator, e.g., as depicted in Figures 6b and 7 where a compilation at high level respectively at RTL is chosen.

First, we examine how the register file is represented in the module `VideoSourceColorCheckBoardWithCtrl` when using the TLM bus for connection. There, the `RegisterFileTLM<Control> memCtrl` member variable is used to represent the register file, i.e., when `BUSABSTRACTION == BUSABSTRACTION_TLM`. Internally, the `RegisterFileTLM<T>` module is derived from the `SimpleMem` module you developed in the previous lab. Thus, the `VideoSourceColorCheckBoardWithCtrl` module defines the TLM socket `ctrl_socket` that is forwarded to the `m_memory_socket` that the `RegisterFileTLM<T>` module inherits from its `SimpleMem` base class. The relevant parts of the `VideoSourceColorCheckBoardWithCtrl` module for integrating the register file with a TLM connection into the module are shown below:

```
class VideoSourceColorCheckBoardWithCtrl
  : public sc_core::sc_module
  , public SW::VideoSourceColorCheckBoardWithCtrl
{
public:
  ...
  // The TLM socket "ctrl_socket" used to connect the
  // register filte of the module to the rest of the system.
  tlm::tlm_target_socket<>              ctrl_socket;

  VideoSourceColorCheckBoardWithCtrl(sc_core::sc_module_name name) {
    : sc_module(name)
    ...
    // Name the TLM socket
    , ctrl_socket("ctrl_socket")
    // Name the register file module
    , memCtrl("memCtrl")
    ...
  {
    ...
    // Forward the ctrl_socket of the module to the m_memory_socket
    // of the register file memCtrl.
    ctrl_socket(memCtrl.m_memory_socket);
    ...
  }

protected:
  // Register file for the module
  RegisterFileTLM<Control> memCtrl;
  ...
  // Getter functions to access the different registers
  // of the register file.
  sc_dt::sc_int<12> getBoxRWidth();
  sc_dt::sc_int<12> getBoxRHeight();
  sc_dt::sc_uint<8> getBoxRColor();

  sc_dt::sc_int<12> getBoxGWidth();
  sc_dt::sc_int<12> getBoxGHeight();
  sc_dt::sc_uint<8> getBoxGColor();

  sc_dt::sc_int<12> getBoxBWidth();
  sc_dt::sc_int<12> getBoxBHeight();
  sc_dt::sc_uint<8> getBoxBColor();
  ...
};
```

Moreover, the module uses the following getter functions to access the different registers of the register file:

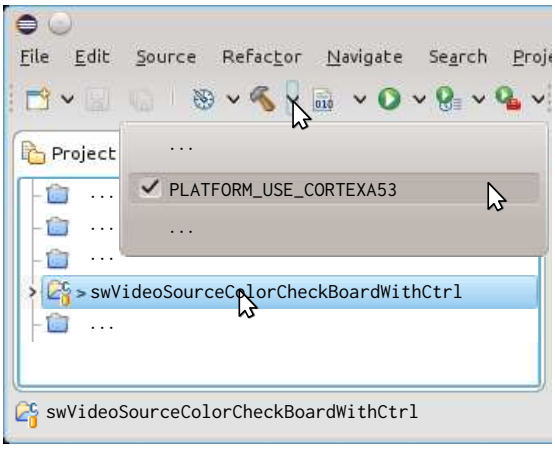
```
sc_dt::sc_int<12> VideoSourceColorCheckBoardWithCtrl::getBoxRWidth()
  { return memCtrl.getCtrl().boxR.width.get(); }
sc_dt::sc_int<12> VideoSourceColorCheckBoardWithCtrl::getBoxRHeight()
```

```
  { return memCtrl.getCtrl().boxR.height.get(); }
sc_dt::sc_uint<8> VideoSourceColorCheckBoardWithCtrl::getBoxRColor()
  { return memCtrl.getCtrl().boxR.colorR.get(); }

sc_dt::sc_int<12> VideoSourceColorCheckBoardWithCtrl::getBoxGWidth()
  { return memCtrl.getCtrl().boxG.width.get(); }
sc_dt::sc_int<12> VideoSourceColorCheckBoardWithCtrl::getBoxGHeight()
  { return memCtrl.getCtrl().boxG.height.get(); }
sc_dt::sc_uint<8> VideoSourceColorCheckBoardWithCtrl::getBoxGColor()
  { return memCtrl.getCtrl().boxG.colorG.get(); }

sc_dt::sc_int<12> VideoSourceColorCheckBoardWithCtrl::getBoxBWidth()
  { return memCtrl.getCtrl().boxB.width.get(); }
sc_dt::sc_int<12> VideoSourceColorCheckBoardWithCtrl::getBoxBHeight()
  { return memCtrl.getCtrl().boxB.height.get(); }
sc_dt::sc_uint<8> VideoSourceColorCheckBoardWithCtrl::getBoxBColor()
  { return memCtrl.getCtrl().boxB.colorB.get(); }
```

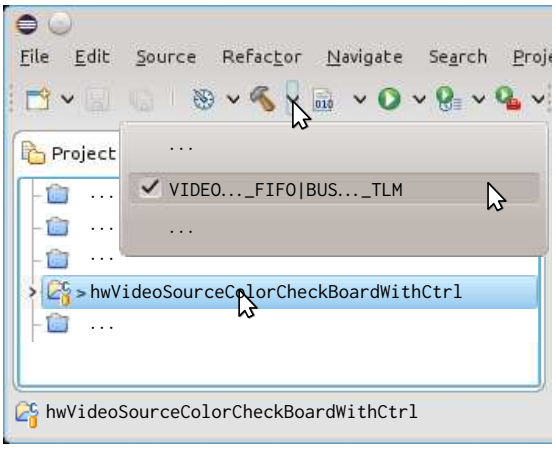
If you want to run a simulation of the CCBCTRL design, compile according to Figures 6a to 6c. Then, select (see Figure 6d) *Video source color check board with control Example* to run the simulation. This should result in color check boards similar to Figure 4a. The color check board configurations are randomly generation and, thus, will only appear similar to the result given in the figure.

However, in the real hardware, the AXI-Lite bus protocol is used to connect the *Zynq UltraScale+ MPSoC* to the hardware module. Thus, we will switch over to the `RegisterFileAXILite` template to represent the register file in case of `BUSABSTRACTION == BUSABSTRACTION_RTL_AXILITE`. The required changes to the `VideoSourceColorCheckBoardWithCtrl` module are as follows:

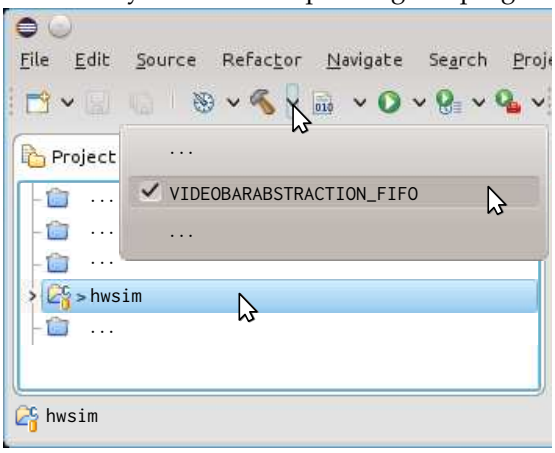
```
class VideoSourceColorCheckBoardWithCtrl
  : public sc_core::sc_module
  , public SW::VideoSourceColorCheckBoardWithCtrl
{
public:
  ...
  // AXI-Lite bus protocol ports to connect the
  // register filte of the module to the rest of the system.
  BUS_RTL_AXILITE_TARGET_SOCKET_PORTS(S);

  VideoSourceColorCheckBoardWithCtrl(sc_core::sc_module_name name) {
    : sc_module(name)
    ...
    // Name the AXI-Lite bus protocol ports
    , BUS_RTL_AXILITE_NAME_TARGET_SOCKET(S)
    // Name the register file module
    , memCtrl("memCtrl")
    ...
  {
    ...
    // Forward the AXI-Lite bus protocol ports to
    // the register file memCtrl.
    BUS_RTL_AXILITE_SIGNAL_FORWARD(memCtrl.,S,,S);
    // Connect signals to the register output ports
```



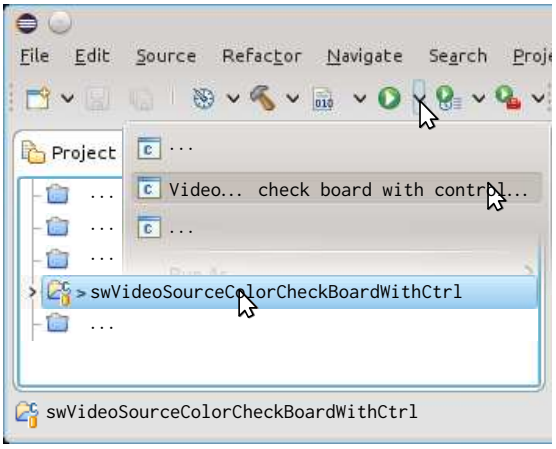
(a) Compile the *libVideoSourceColorCheckBoardWithCtrl* library and its corresponding test program



(b) Compile the SystemC model for the hardware accelerator `VideoSourceColorCheckBoardWithCtrl`



(c) Compiling the *hwsim* simulator



(d) Run the test program for the color check board on the `hwsim` simulator

Figure 6: Running a simulation using the software controlled hardware module `VideoSourceColorCheckBoardWithCtrl` to generate varying color check board test images

```
    // of the register file.
    memCtrl.mem_out[0](mem[0]);
    memCtrl.mem_out[1](mem[1]);
    memCtrl.mem_out[2](mem[2]);
    memCtrl.mem_out[3](mem[3]);
    memCtrl.mem_out[4](mem[4]);
    memCtrl.mem_out[5](mem[5]);
    memCtrl.mem_out[6](mem[6]);
    memCtrl.mem_out[7](mem[7]);
    memCtrl.mem_out[8](mem[8]);
    memCtrl.mem_out[9](mem[9]);
    memCtrl.mem_out[10](mem[10]);
    memCtrl.mem_out[11](mem[11]);
    // Instruct VivadoHLS to realize the mem array as registers, i.e.,
    // allowing parallel access.
    #pragma HLS array_partition variable=mem complete dim=0
    ...
}
```

```
protected:
  // Register file for the module
  RegisterFileAXILite<12,4> memCtrl;
  // Signals for connecting to the register outputs
  // ports of the register file.
  sc_core::sc_signal<sc_dt::sc_uint<8> > mem[12];
  ...
};
```

As can be seen, the `RegisterFileTLM<Control>` type was replaced by a register file that supports the AXI-Lite bus protocol, i.e., `RegisterFileAXILite<12,4>`. Here, 12 indicates the number of bytes contained in the register file and 4 is the required number of address bits required to address this number of bytes, i.e., $4 = \lceil \log_2 12 \rceil$. Moreover, instead of a `getCtrl()` method, the output ports `memCtrl.mem_out[0]...memCtrl.mem_out[11]` of the AXI-Lite register file are used to make its contents available. These ports are connected to the signals `mem[0]...mem[11]` of the `VideoSourceColorCheckBoardWithCtrl` and must be used to access the register file. Thus, the getter functions have to be reworked as follows to use these signals:

```
sc_dt::sc_int<12> VideoSourceColorCheckBoardWithCtrl::getBoxRWidth() {
  sc_dt::sc_int<12> retval;
  retval.range( 7, 0) = mem[ 3].read();
  retval.range(11, 8) = mem[ 2].read().range(3,0);
  return retval;
}

sc_dt::sc_int<12> VideoSourceColorCheckBoardWithCtrl::getBoxRHeight() {
  sc_dt::sc_int<12> retval;
  retval.range( 3, 0) = mem[ 2].read().range(7,4);
  retval.range(11, 4) = mem[ 1].read();
  return retval;
}

sc_dt::sc_uint<8> VideoSourceColorCheckBoardWithCtrl::getBoxRColor() {
  return mem[ 0].read();
}

sc_dt::sc_int<12> VideoSourceColorCheckBoardWithCtrl::getBoxGWidth() {
  sc_dt::sc_int<12> retval;
  retval.range( 7, 0) = mem[ 7].read();
  retval.range(11, 8) = mem[ 6].read().range(3,0);
  return retval;
}

sc_dt::sc_int<12> VideoSourceColorCheckBoardWithCtrl::getBoxGHeight() {
  sc_dt::sc_int<12> retval;
  retval.range( 3, 0) = mem[ 6].read().range(7,4);
  retval.range(11, 4) = mem[ 5].read();
  return retval;
}

sc_dt::sc_uint<8> VideoSourceColorCheckBoardWithCtrl::getBoxGColor() {
```

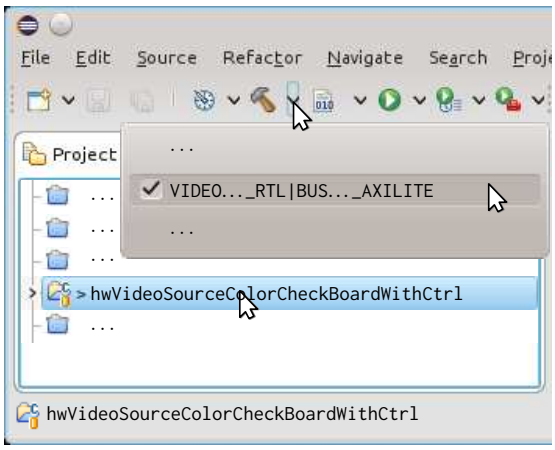


Figure 7: Compile the SystemC model to use the AXI-Lite bus protocol

```
  return mem[ 4].read();
}

sc_dt::sc_int<12> VideoSourceColorCheckBoardWithCtrl::getBoxBWidth() {
  sc_dt::sc_int<12> retval;
  retval.range( 7, 0) = mem[11].read();
  retval.range(11, 8) = mem[10].read().range(3,0);
  return retval;
}

sc_dt::sc_int<12> VideoSourceColorCheckBoardWithCtrl::getBoxBHeight() {
  sc_dt::sc_int<12> retval;
  retval.range( 3, 0) = mem[10].read().range(7,4);
  retval.range(11, 4) = mem[ 9].read();
  return retval;
}

sc_dt::sc_uint<8> VideoSourceColorCheckBoardWithCtrl::getBoxBColor() {
  return mem[ 8].read();
}
```

If you want to run a simulation using the AXI-Lite bus protocol to access the register file as well as use RTL abstraction for the generated video stream, compile the video filter as depicted in Figure 7 and start the simulation by selecting the run target *Video source color check board with control Example* as shown in Figure 6d. The *hwsim* simulator and your test program should still be compiled with the targets shown in Figures 6a and 6c, respectively. If you run the simulation at this abstraction level, the VCD file *videoSourceColorCheckBoardWithCtrlWrapper@0x43c30000-size-0xc.vcd* should be created inside the `swVideoSourceColorCheckBoardWithCtrl` project. This file contains the wave forms of the inputs and outputs of the `VideoSourceColorCheckBoardWithCtrl` module. The file can be view by opening it with `gtkwave` as shown in Figure 8.

1.2 Running the CCBCTRL Design on the ZCU102 Board

To start design on the ZCU102 board, we require a description of the hardware design as well as the software binary. Here, we already pre-built the `ccbctrl` block design and provided the corresponding files *ccbctrl_wrapper.bit*, *psu_init.c*, *psu_init.h*, and *psu_init.tcl* in the project `swVideoSourceColorCheckBoardWithCtrl`. As mentioned in the *filter* lab, the *ccbctrl_wrapper.bit*

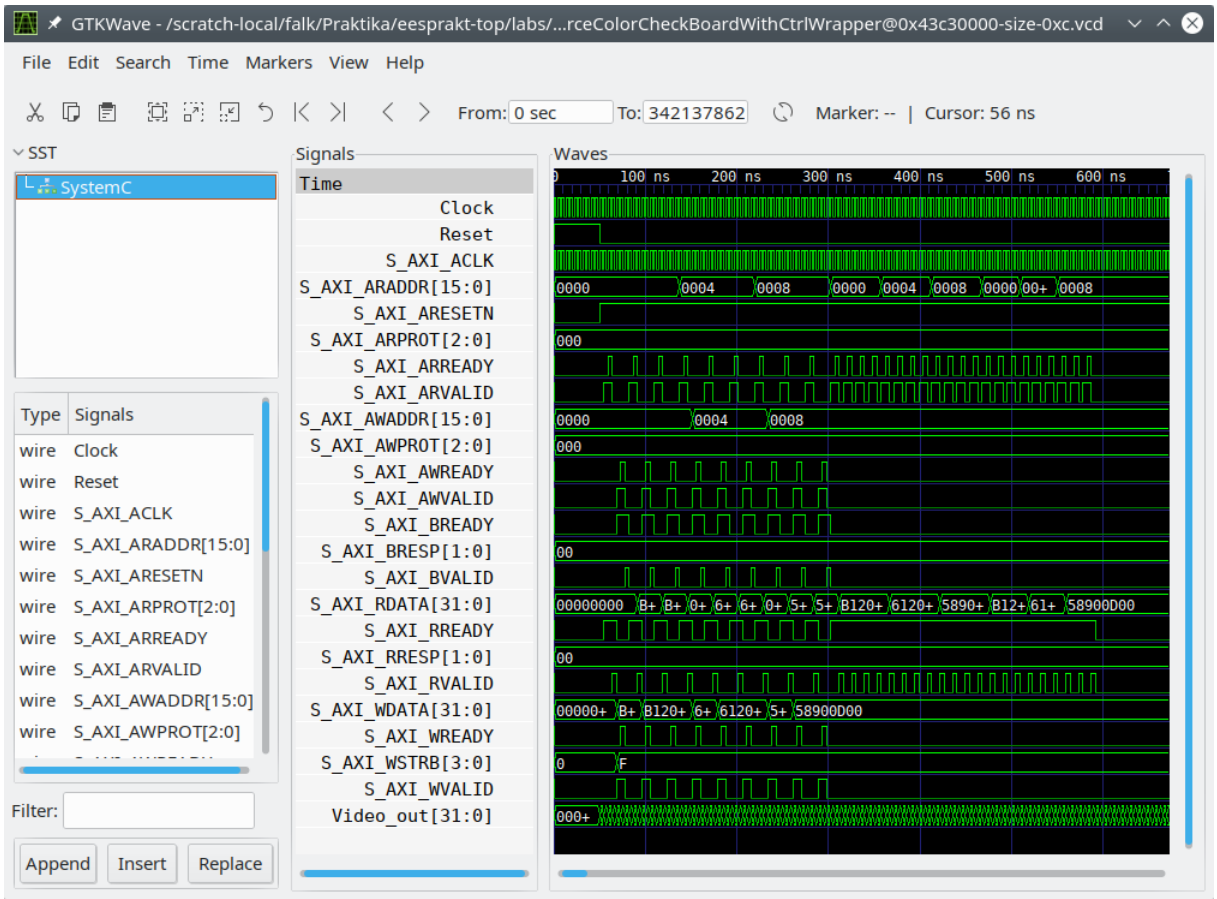


Figure 8: Viewing the input and output signals of the `VideoSourceColorCheckBoardWithCtrl` module with GTKWave

bit file contains the configuration for the Programmable Logic (PL) of the ZCU102 MPSoC. The files *psu_init.c*, *psu_init.h*, and *psu_init.tcl* describe the configuration of the *Zynq UltraScale+ MPSoC* Intellectual Property Block (IPB). Here, the file *psu_init.tcl* is used to configure the Processing System (PS) via the Xilinx debugger, which will be used by the `ZCU102Init.sh` command, while the files *psu_init.c* and *psu_init.h* are required to build a *boot image* used to boot from microSD card or QSPI flash. A boot image can be created by the `ZCU102Flash.sh` command contained in the project. For further discussions of the different boot modes, please consult Section "The Xilinx ZCU102 Board" in the description of the filter lab.

To obtain the software binary, compile the `swVideoSourceColorCheckBoardWithCtrl` project as shown in Figure 6a. This will result in the file *obj/src/main/main.elf*.

Next, to prepare the ZCU102 board to run your program, switch turn the board off and on again. Then, execute the "`ZCU102Init.sh`" script in the `swVideoSourceColorCheckBoardWithCtrl` project to run it on the ZCU102 board:

```
...]$ cd ~/05_hwsw/workspace-hwsw/swVideoSourceColorCheckBoardWithCtrl
...]$ ./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!
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
Start program (y/n)? y
Downloading Program -- .../obj/src/main/main.elf
        section, .text: ...
        section, .init: ...
        ...
100%    0MB   0.6MB/s  00:01
Setting PC to Program Start Address 0x00100000
Successfully downloaded .../obj/src/main/main.elf
Stop program (y/n)? n
...]$
```

To see the output of the program running on the ZCU102 board, you have to connect to its serial console. For this purpose, use the command `picocom -b 115200 /dev/ttyUSB1`. Note that you may connect to the board any time after powering it via the USB connection to your computer. However, only after programming the FPGA and downloading your program via `ZCU102Init.sh` will you get an output on the serial console.

1.3 System Integration and BIT File Creation

First, start behavioral synthesis via VivadoHLS as depicted in Figure 9 to create the IPB for Vivado. Then, open the Vivado project `workspace-vivado-hwsw` (see Figure 10). Next, enable the `ccbctrl` block design and refresh the IP repositories. Moreover, activate the *ccbctrl.xdc* constraint file, i.e., activate the *ccbctrl* constraint set. Then, open the `ccbctrl` block design – a depiction of which is given in Figure 11a – and upgrade the `VideoSourceColorCheckBoardWithCtrl` IPB to the version you have just synthesized via VivadoHLS. If you do not know how to perform these steps, please consult the Section "System Integration and BIT File Creation" in the filter lab description. Furthermore, you can examine where the hardware design will place the register file of the `VideoSourceColorCheckBoardWithCtrl` module in the address space of the Zynq UltraScale+ PS by consulting the address mapping as shown in Figure 11b. It is of utmost importance that the address mapping is consistent with the register

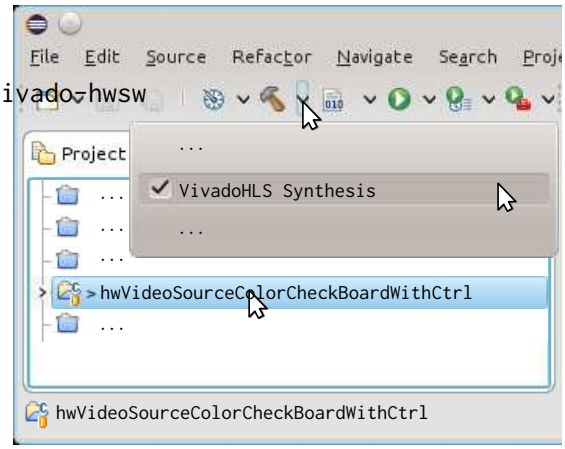
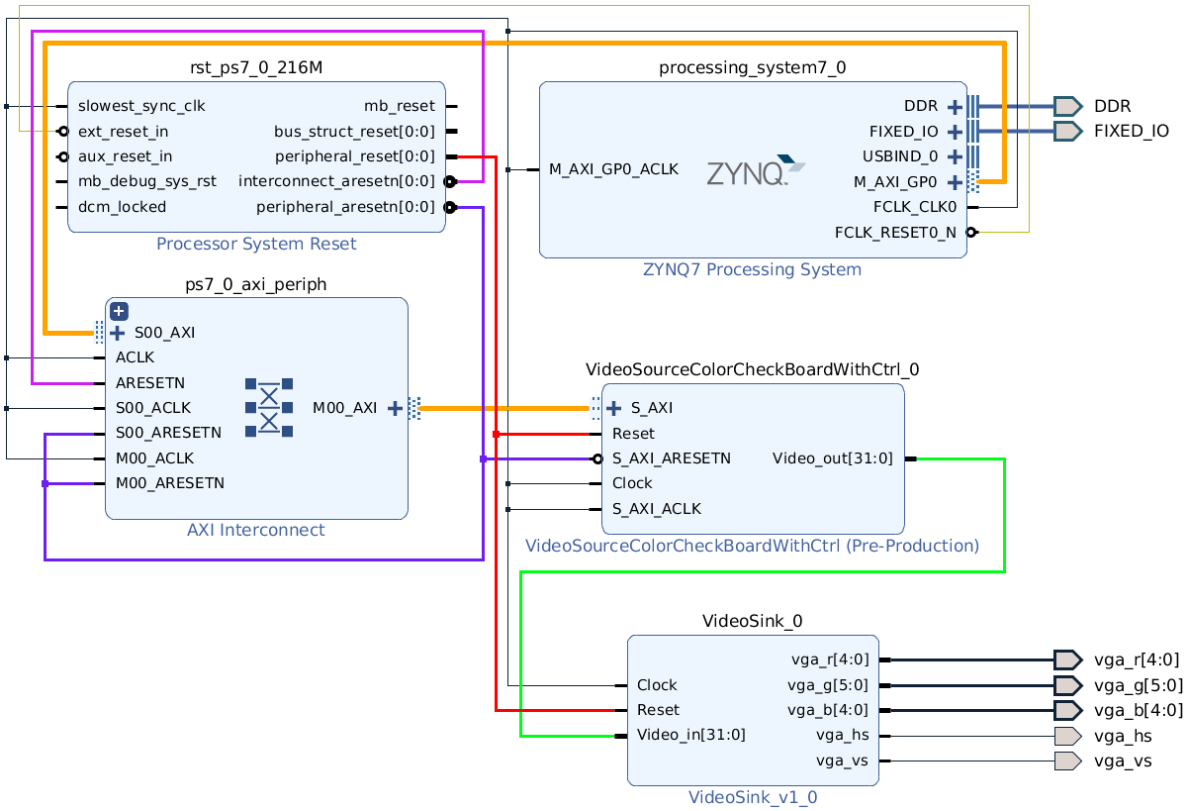


Figure 9: Start behavioral synthesis of the hardware accelerator by using VivadoHLS

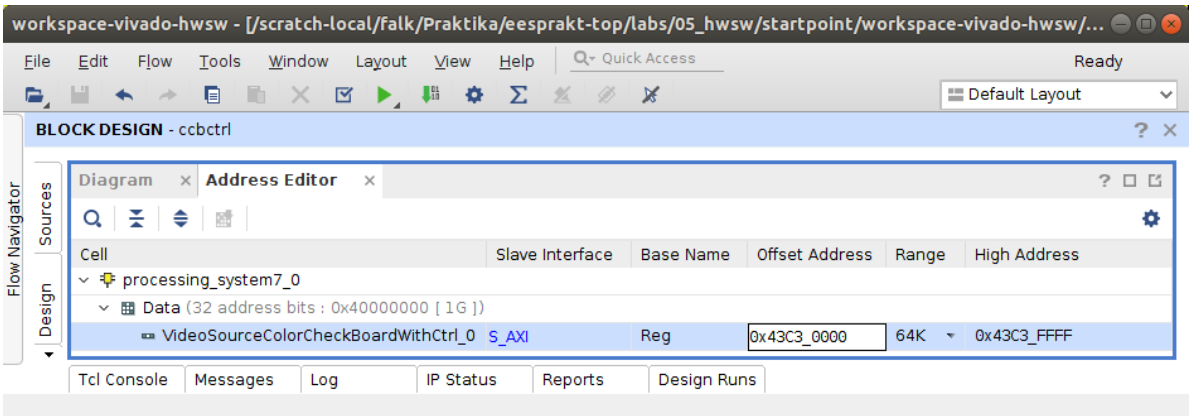
Figure 10: Open the `workspace-vivado-hwsw` Vivado project

file addresses given by the define `VIDEOSOURCECOLORCHECKBOARDWITHCTRL_CTRLADDR`. Otherwise, the hardware you are currently creating and the software that will run on it do not agree on the location of the register files that are controlling the hardware modules contained in the design. Next, create a HDL wrapper (see Figure 12) around the `ccbctrl` block design and set the wrapper as top module (see Figure 13) for synthesis. If the *Set as Top* command is grayed out, then the wrapper is already the top module. Then, start bit file generation as shown in Figure 14.

This will result in a bit file called *ccbctrl_wrapper.bit* that will be used by the scripts `ZCU102Init.sh` and `ZCU102Flash.sh` to either program or flash the ZCU102 board. Don't forget to also copy *psu_init.h*, *psu_init.c*, and *psu_init.tcl initialization files*. For this purpose, export the design to xSDK via "File > Export > Export Hardware for SDK" as depicted in Figure 15 and launch xSDK (see Figure 16) to create the initialization files. Then, copy (see Figure 17) and paste (see Figure 18) these files to the `swVideoSourceColorCheckBoardWithCtrl` Eclipse project. Finally, test if the steps in Section 1.3 still produce a running design on the ZCU102 board.



(a) Block design `ccbctrl`



(b) Address mapping

Figure 11: Architecture from Figure 5 realized in Vivado for the ZCU102 board

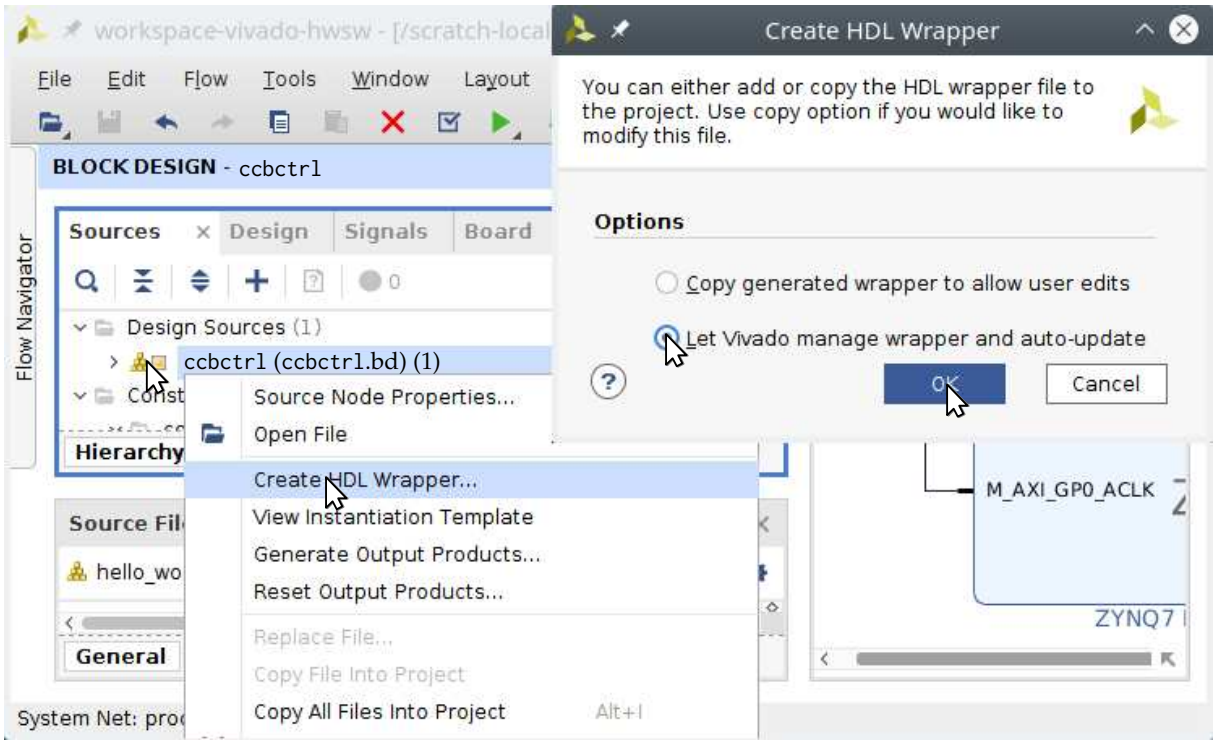


Figure 12: Create a HDL wrapper around the block design

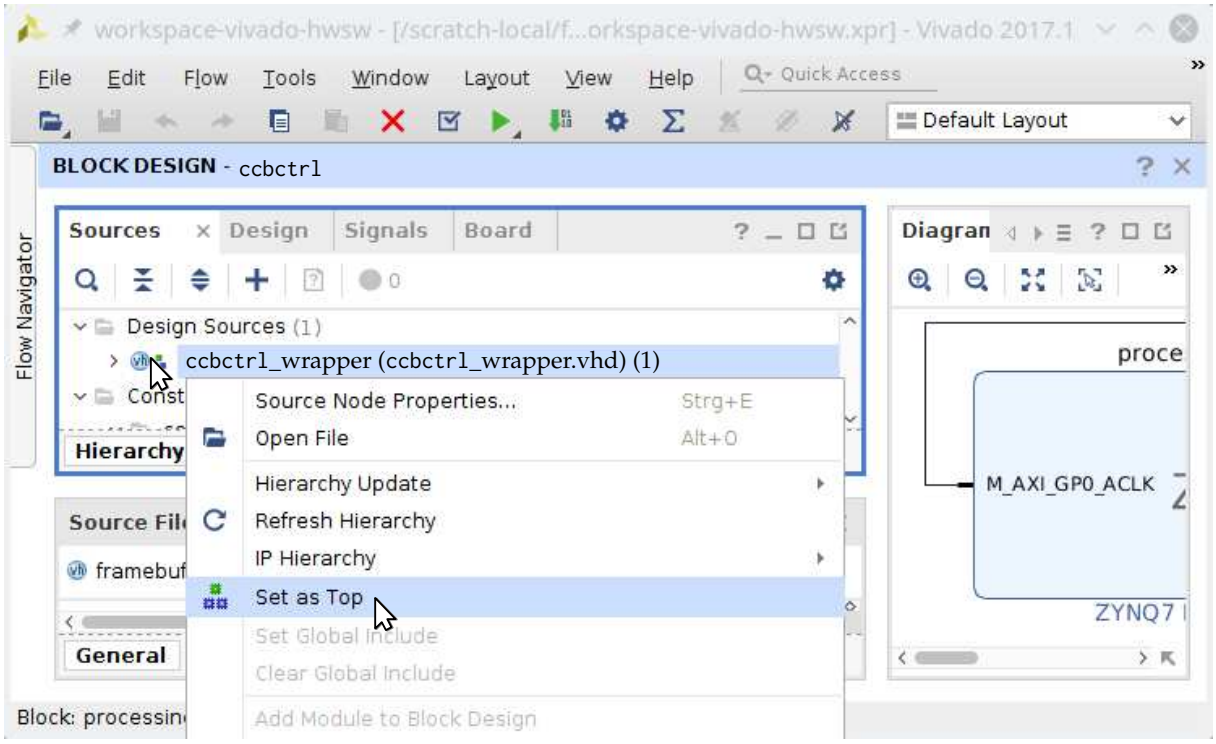


Figure 13: Set the wrapper as top module for synthesis

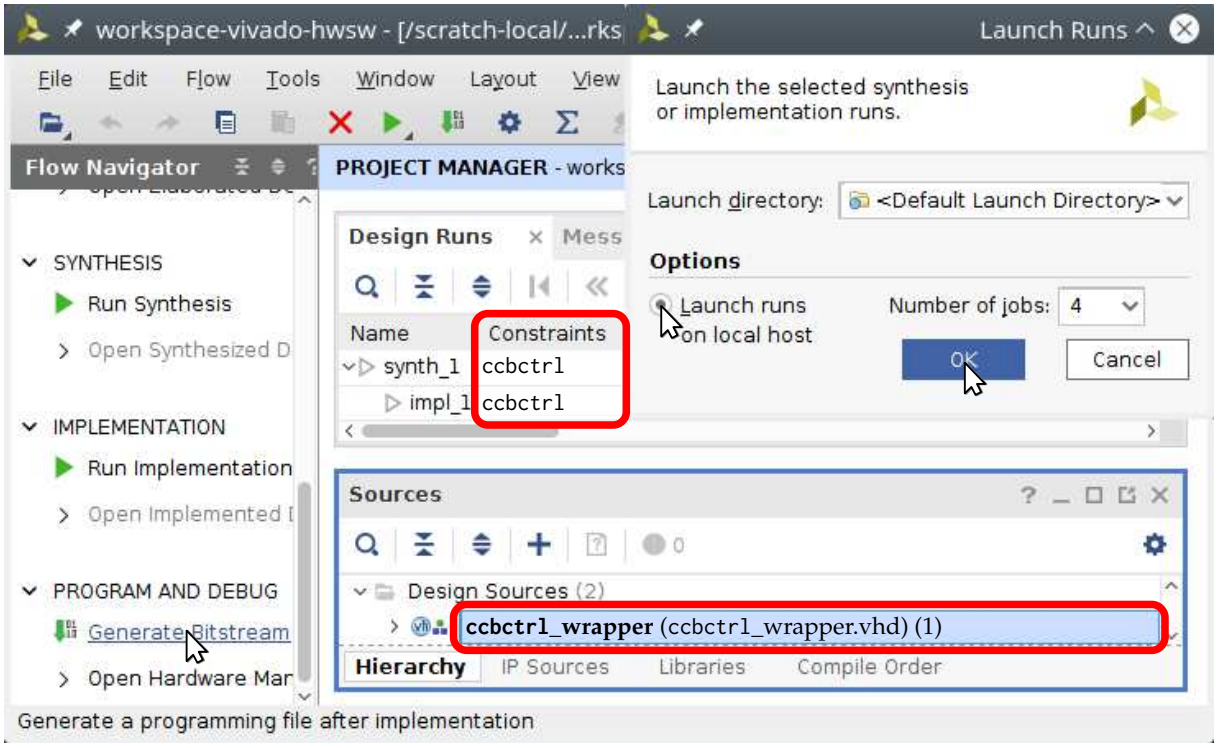


Figure 14: Start bit file generation for the `ccbctrl` block design

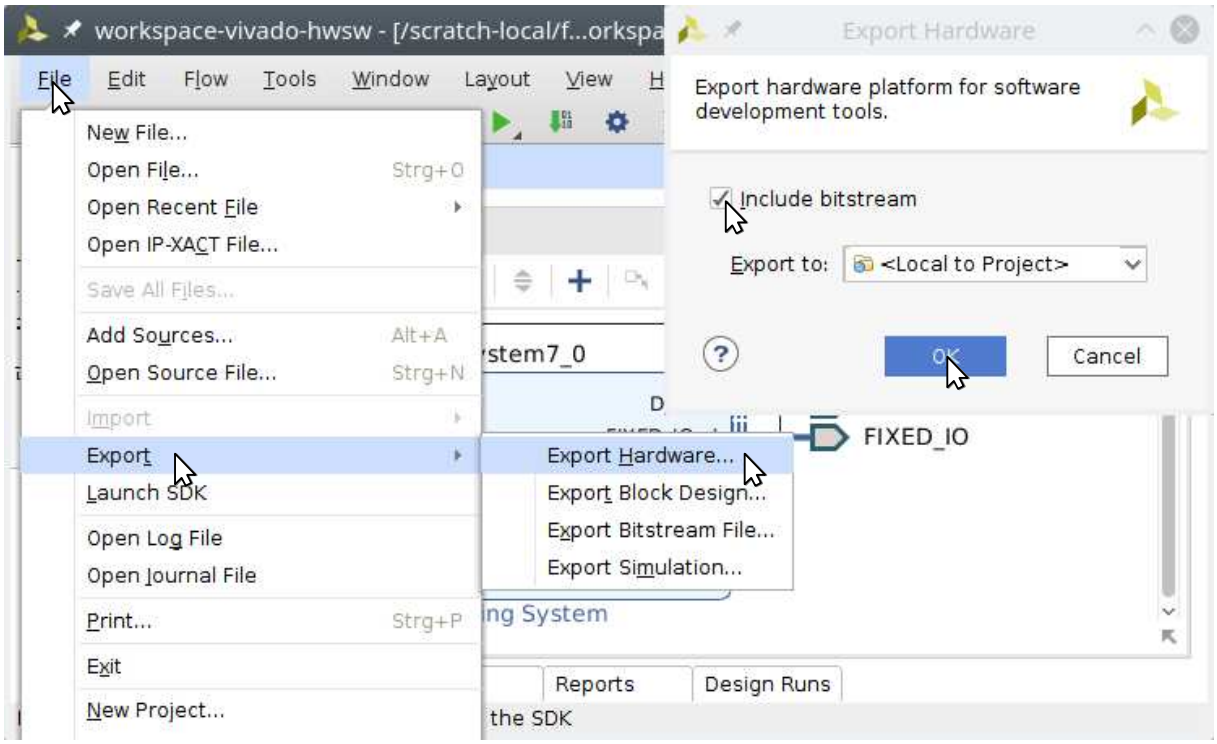


Figure 15: Export hardware

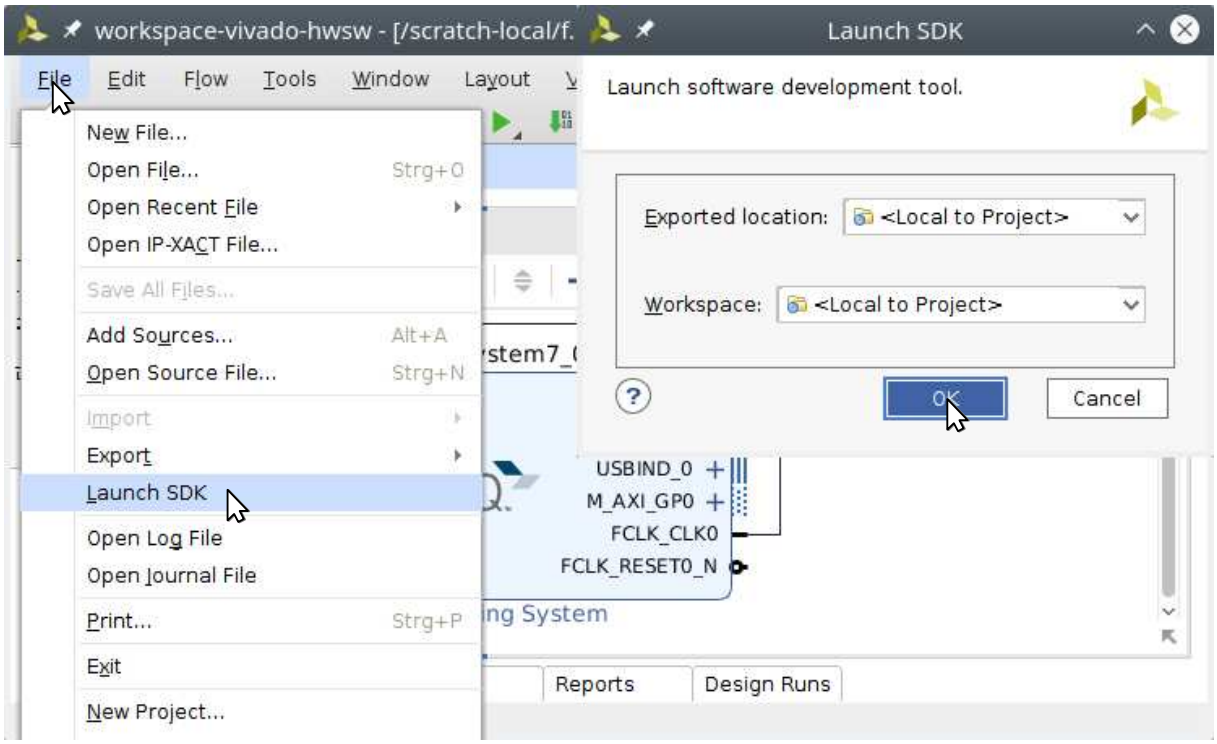


Figure 16: Launch SDK

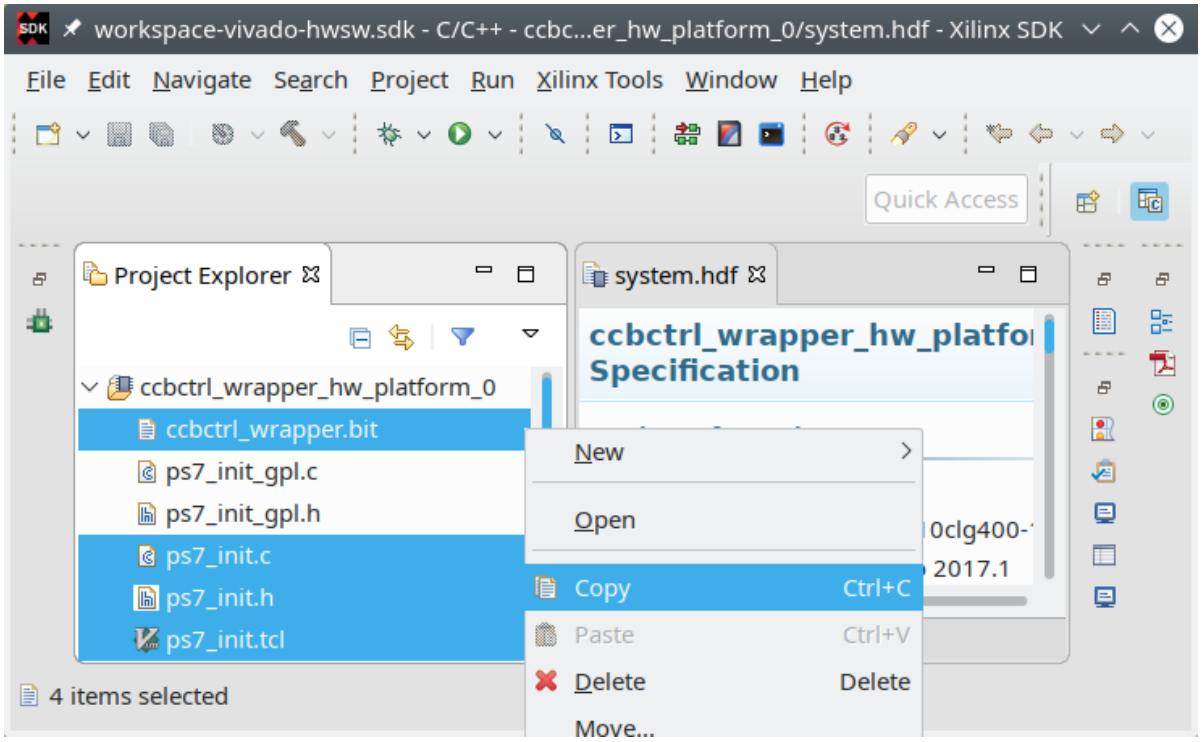


Figure 17: Copy the initialization files for the "Zynq UltraScale+ MPSoC" and the bit file. Use Ctrl-C to copy the selected files marked in blue.

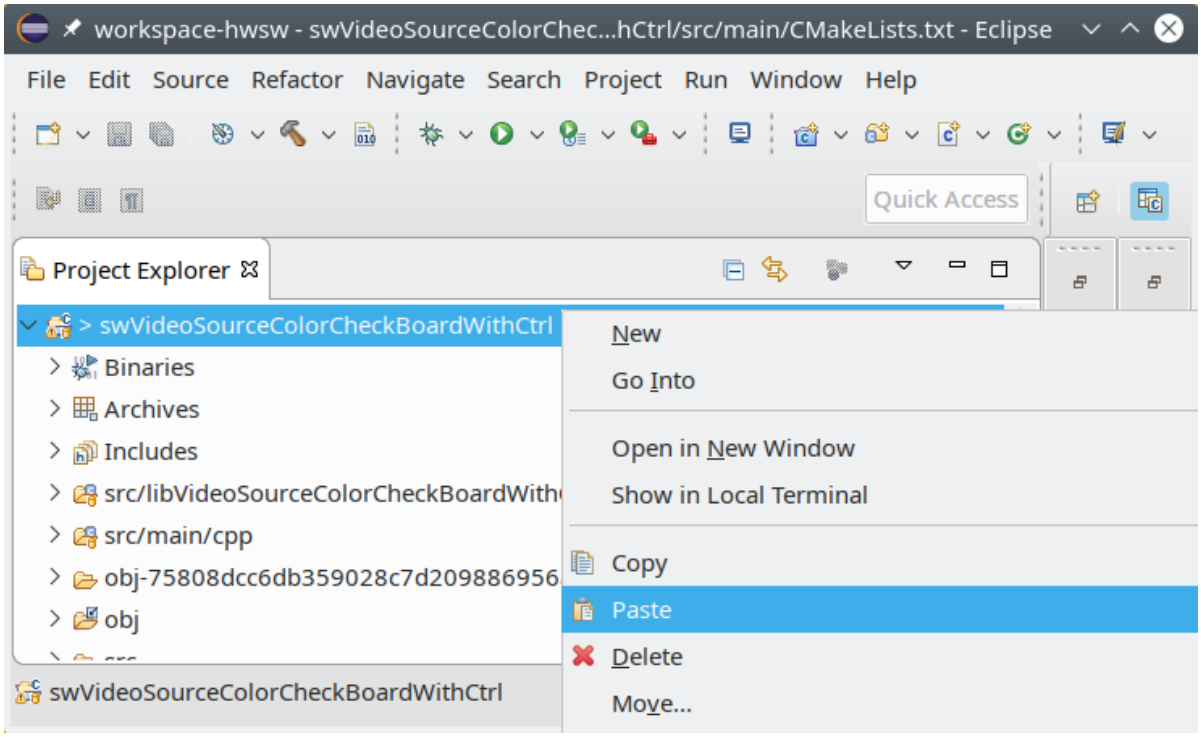


Figure 18: Paste the initialization files for the "Zynq UltraScale+ MPSoC" and the bit file by using Ctrl-V while the `swVideoSourceColorCheckBoardWithCtrl` project is selected.

1.4 DMA-Based HW/SW Communication

In the following, we will look at the `VideoSourceFromMem` hardware module. This module uses DMA to fetch images from memory and translate them into a video stream as shown in Figure 19. As always, this video stream can then be modified by video filters and converted to

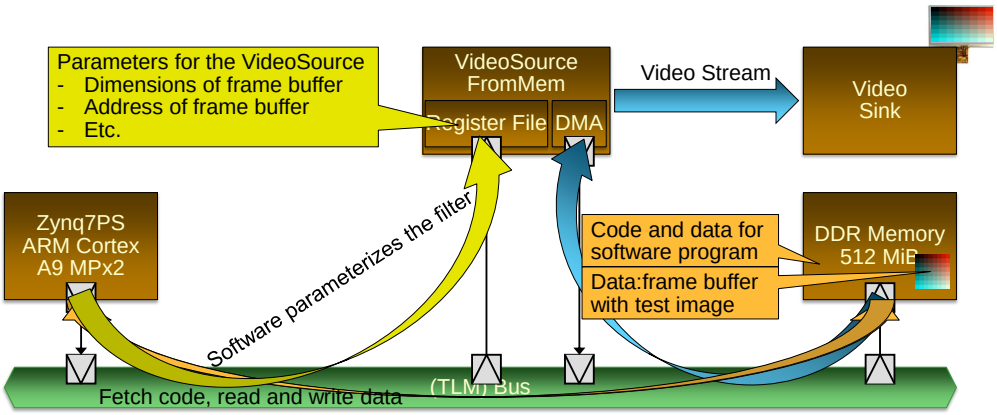


Figure 19: Architecture for the display of test images

DisplayPort via the `VideoSink` module in order to display it. The `VideoSourceFromMem` hardware module is contained in the `hwsim` project. This hardware module is controlled via usage of a register file (see Figure 19). The layout of the register file is provided by the header *sw/VideoSourceFromMem.hpp* in the `hwsim` project. The relevant `VideoSourceFromMem::Control` structure is replicated below:

```
struct Control {
  union {
    Register<00, 12, uint16_t, BIG_ENDIAN, 32> width;
    Register<12, 12, uint16_t, BIG_ENDIAN, 32> height;
    Register<24, 8,  uint8_t,  BIG_ENDIAN, 32> count;
  } dim;
  struct {
    Register<00, 32, unsigned char *, BIG_ENDIAN, 32> addr;
    union {
      // fmt == 1 is gray scale
      //   ( 8-bit pixels, d[ 7: 0]=r, d[ 7: 0]=g, d[ 7: 0]=b).
      // fmt == 2 is r5g6b5
      //   (16-bit pixels, d[15:11]=r, d[10: 5]=g, d[ 4: 0]=b).
      // fmt == 3 is b8g8r8
      //   (24-bit pixels, d[ 7: 0]=r, d[15: 8]=g, d[23:16]=b).
      // fmt == 4 is a8b8g8r8
      //   (32-bit pixels, d[31:24]=a, d[23:16]=b, d[15:8]=g, d[7:0]=r)
      Register<00,  4, uint8_t      , BIG_ENDIAN, 16> fmt;
    };
  } buf;
  union {
    Register<00,  1, bool,            BIG_ENDIAN, 8>  reqovf;
    Register<01,  1, bool,            BIG_ENDIAN, 8>  datauvf;
    Register<02,  1, bool,            BIG_ENDIAN, 8>  rderr;
    Register<03,  1, bool,            BIG_ENDIAN, 8>  interr;
  } dma;
```

```
    char _pad[3];
  };
```

The fields `dim.width` and `dim.height` specify the width and height of the images that should be streamed. This fields must be set by the software controlling the `VideoSourceFromMem` IPB. In contrast, `dim.count` is a counter (modulo 256) that will be incremented by the IPB each time it has streamed one image to its `Video_out` port. Where and how an image is stored in memory is controlled by the fields `buf.addr` and `buf.fmt`, respectively. The `VideoSourceFromMem` IPB will only stream images if `buf.fmt` is set to a valid format, i.e., 1 to 4, and `buf.addr` points to a non-zero address.

In order to control the `VideoSourceFromMem` hardware module, the `swVideoSourceFromMem` project is provided. The relevant files are as follows:

Project:	`swVideoSourceFromMem`
Files:	*src/libVideoSourceFromMem/cpp/libVideoSourceFromMem.cpp*
	src/libVideoSourceFromMem/headers/libVideoSourceFromMem.hpp
	src/main/cpp/main.cpp

The `swVideoSourceFromMem` project provides the *libVideoSourceFromMem* library, which has three goals: (a) define the global structure `ctrlMEM2VS` that maps to the register file of the hardware module `VideoSourceFromMem`, (b) provide the function `dumpRegs` to dump the contents of the register file, and, finally, (c) provide the function `testVideoSourceFromMem` that uses the register file to display a test image. The test program *src/main/cpp/main.cpp* simply uses the `testVideoSourceFromMem` function to display the sequence of test images shown in Figure 20.

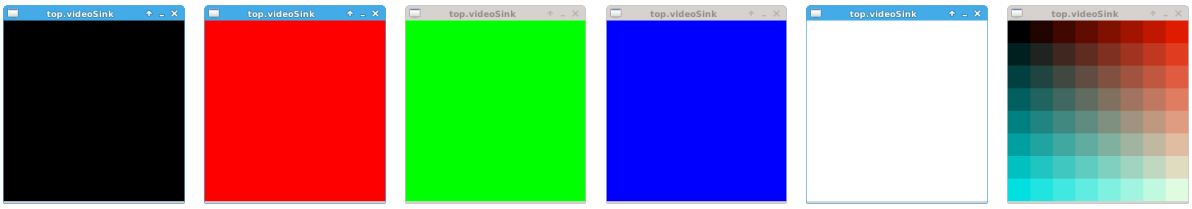
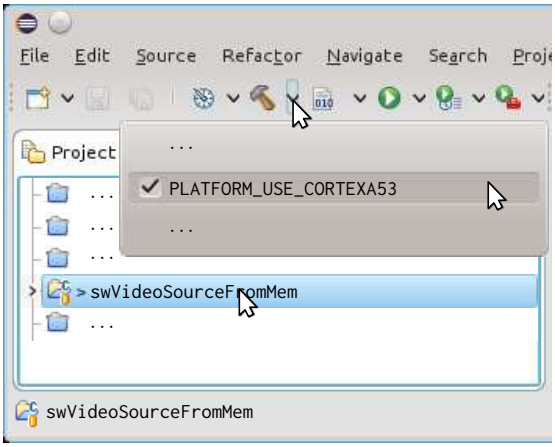
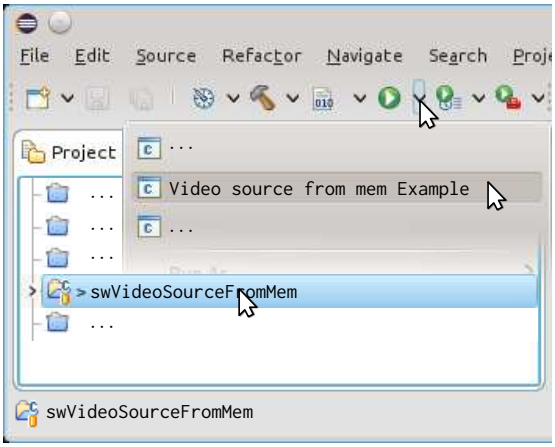


Figure 20: Expected sequence of test images

To check the functionality of the library, you can execute the test program in the simulation environment by following the steps given in Figure 21.



(a) Compile the *libVideoSourceFromMem* library and its corresponding test program



(b) Run the test program for the *libVideoSourceFromMem* library on the `hwsim` simulator

Figure 21: Use VideoSourceFromMem for Test Image Display

Finally, you can execute the program on the ZCU102 board by using the block design depicted in Figure 22. For this purpose, we already pre-built the `framebuffer` block design and provided

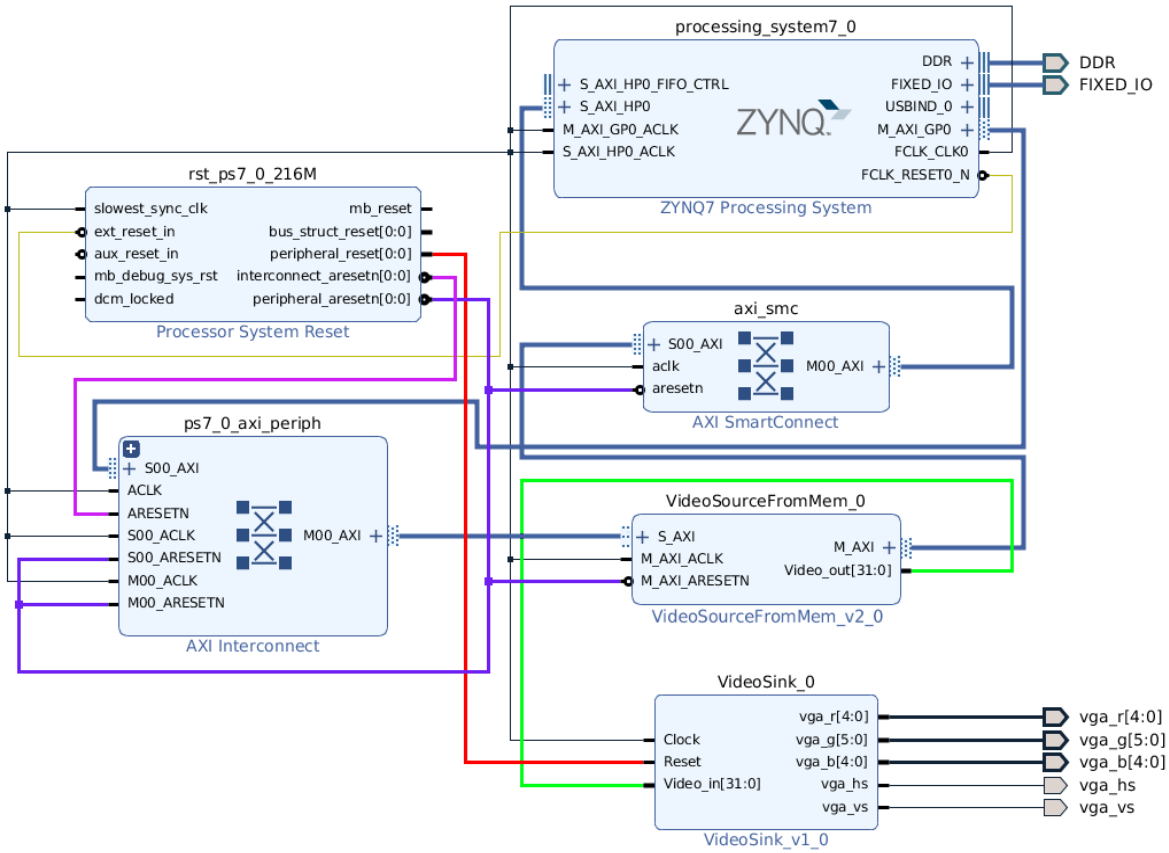


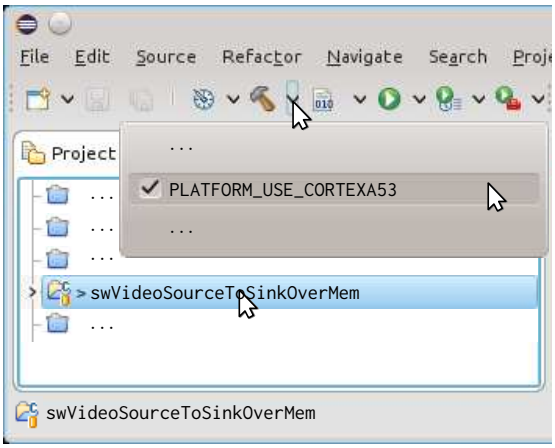
Figure 22: Architecture from Figure 19 realized in Vivado for the ZCU102 board

the corresponding files *framebuffer_wrapper.bit*, *psu_init.c*, *psu_init.h*, and *psu_init.tcl* in the project `swVideoSourceFromMem`. To prepare the ZCU102 board to run your program, turn the board off and on again. Then, run it on the ZCU102 board by executing the "`ZCU102Init.sh`" script in the `swVideoSourceFromMem` project.

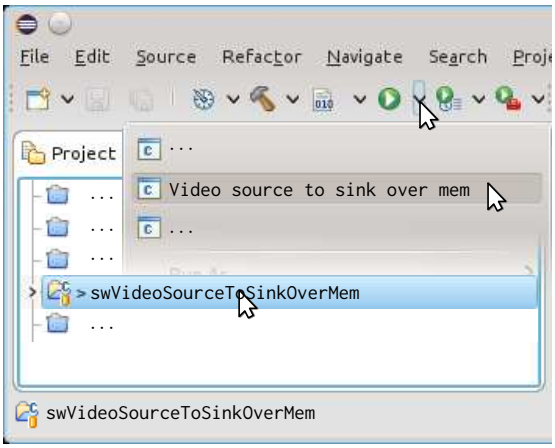
Task 1 (Pass the Video Stream Over the Memory)

Project:	`swVideoSourceToSinkOverMem`
Files:	*src/libVideoSinkToMem/cpp/libVideoSinkToMem.cpp*
	src/libVideoSinkToMem/headers/libVideoSinkToMem.hpp
	src/main/cpp/main.cpp

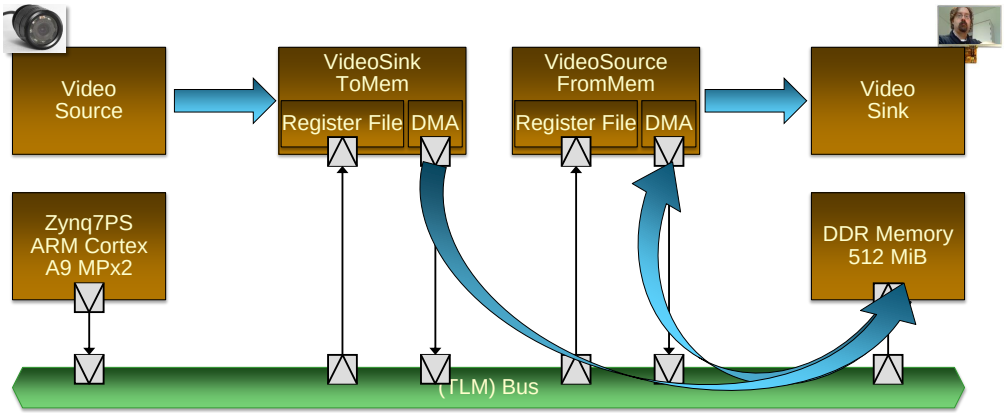
In the following, you will use the `VideoSinkToMem` and `VideoSourceFromMem` hardware modules (see Figure 23) to write the video images to memory and display them again via the previously introduced `VideoSourceFromMem` IPB. First, in the simulation environment and then on the ZCU102 board. The `VideoSinkToMem` hardware module is controlled via usage of a register file. The layout of the register file is provided by the header *sw/VideoSinkToMem.hpp* in the `hwVideoSinkToMem` project.



(a) Compile the *libVideoSinkToMem* library and its corresponding test program



(b) Run the test program for the *libVideoSinkToMem* library on the `hwsim` simulator



(c) System architecture

Figure 23: Use the `VideoSinkToMem` and `VideoSourceFromMem` hardware modules to pass a video stream over the memory

The `swVideoSourceToSinkOverMem` project provides the *libVideoSinkToMem* library, which has three goals: (a) define the global structure `ctrlVS2MEM` that maps to the register file of the `VideoSinkToMem` module, (b) provide the function `dumpRegs` to dump the contents of the register file, and, finally, (c) provide the function `testVideoSinkToMem` that uses the register file to determine the dimensions of the incoming video stream, allocate a corresponding buffer and return its address. The test program *src/main/cpp/main.cpp* simply uses the `testVideoSourceFromMem` and `testVideoSinkToMem` functions to setup a situation where, first, the test sequence given in Figure 20 is displayed and, second, the video input is streamed over the memory to the video

output. It is your task to complete the implementation of the *libVideoSinkToMem* library. For this purpose, the following modifications are necessary:

a) Add a declaration for `ctrlVS2MEM` to the *libVideoSinkToMem.hpp* header file. Take care to use the default address for the register file the hardware module `VideoSinkToMem` specified in the header *sw/VideoSinkToMem.hpp* from the `hwVideoSinkToMem` project. Take care to use the default address for the register file of the hardware module `VideoSourceFromMem` specified in the header *sw/VideoSourceFromMem.hpp* from the `hwVideoSinkToMem` project, e.g., use code like `Something::Control &ctrlST = *reinterpret_cast<Something::Control *>(SOMETHING_CTRLADDR);`. This ensures that `ctrlST` is a reference that points to the register file of the `Something` hardware module. Thus, modifying `ctrlST` will modify the register file provided by the `Something` hardware module hence, controlling this module.

b) Add the `ctrl` parameter to the function `dumpRegs` that specifies the register file that should be dumped. Use the `Control` structure from the *sw/VideoSinkToMem.hpp* header contained in the `hwVideoSinkToMem` project.

c) Add the `ctrl` parameter, as well as `width` and `height` as call by reference parameter to the `testVideoSinkToMem` function. The `width` and `height` parameter should be output parameters that are updated to the detected dimensions of the incoming video stream. The `ctrl` parameter must be a reference so that `testVideoSinkToMem` function can modify and read from the real control structure provided by the register file of the hardware module `VideoSinkToMem`. Otherwise, only a copy is modified and old values read and the *libVideoSinkToMem* library and the hardware module can't communicate with each other. Finally, the `testVideoSinkToMem` function should return the address of a buffer it will allocate to hold the video images that are stream in from the `VideoSinkToMem` hardware module.

d) Implement the `dumpRegs` function. You can take notes from the implementation of the function for the `VideoSourceFromMem` hardware module.

Next, you will implement the `testVideoSinkToMem` function. For this purpose, the following steps have to be performed:

e) Activate the `VideoSinkToMem` hardware module by writing a nonzero address for the receiving buffer into its register file. Take note that the buffer size should still be zero. Otherwise, the hardware module will overwrite the memory specified via the address.

f) Now try to determine the dimensions of the incoming video stream. Assume you have the right dimensions only if these dimensions stay stable for at least five received video images. Hint: You can use `dim.count` from the register file of the `VideoSinkToMem` hardware module to determine if a new image has been received.

g) Allocate memory for the RGB frame buffer according to the detected dimensions and parameterize the register file given by the `ctrl` parameter accordingly to facilitate reception of the video stream to the allocated frame buffer. Make sure to return the address of the allocated buffer at the end of this function.

Subsequently, you should realize the architecture from Figure 23c for the ZCU102 board and execute the program on the real hardware. You should aim to replicate the `memstream` block design shown in Figure 24. For this purpose, perform the following steps:

h) Create the `memstream` block design (see Figure 25). In the end, this will result in a bit file called *memstream_wrapper.bit* used by the scripts `ZCU102Init.sh` and `ZCU102Flash.sh` to either program or flash the ZCU102 board. If you choose a different name, you will have to modify *config.sh* and *src/main/CMakeLists.txt* in `swVideoSourceFromMem`.

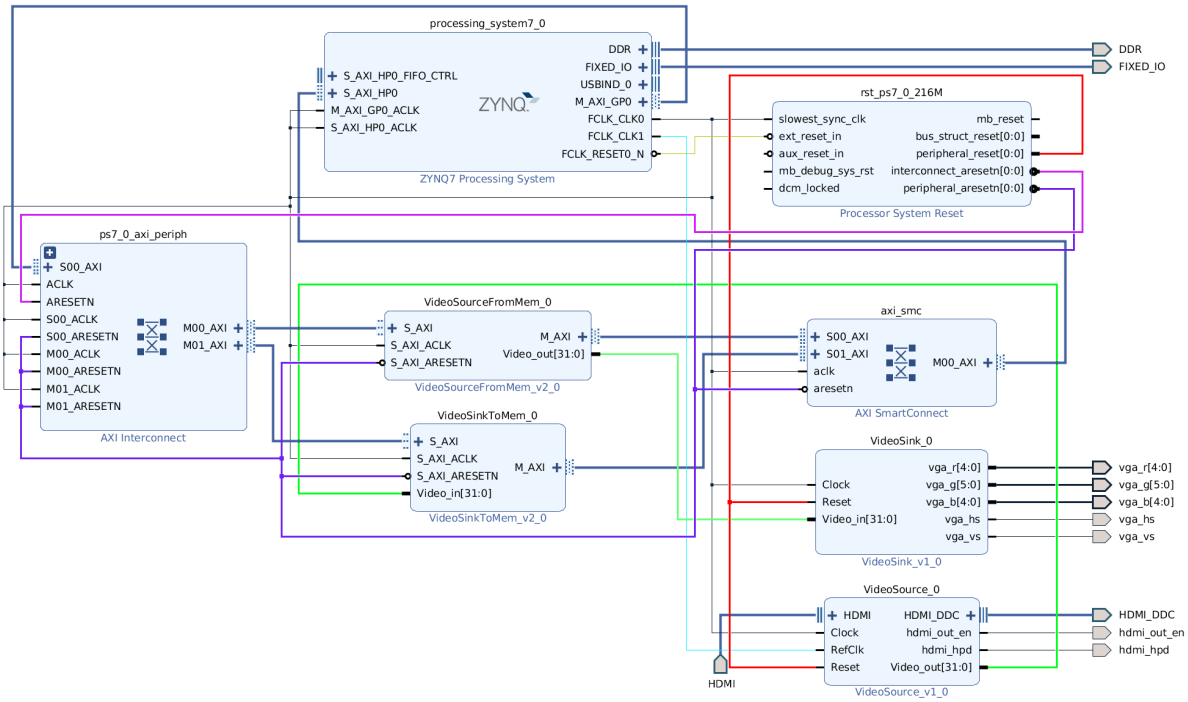


Figure 24: Architecture from Figure 23c realized in Vivado for the ZCU102 board

i) Add the IPBs "VideoSourceFromMem", "VideoSinkToMem", and "Zynq UltraScale+ MPSoC" to the empty block design (see Figure 26).

j) Re-customize the "Zynq UltraScale+ MPSoC" to be valid for the ZCU102 board, i.e., by applying the *ZCU102_zynq_def.tcl* preset to the block (see Figure 28). Moreover, you have to customize (see Figure 29) your "Zynq UltraScale+ MPSoC" to have a "AXI HP Slave" port to allow the IPBs "VideoSourceFromMem" and "VideoSinkToMem" access to the DDR memory. Additionally, the "VideoSourceFromMem" IPB expects a clock frequency of 216 MHz and the "VideoSinkToMem" IPB requirs a 200 MHz clock on its `RefClk` input port. Thus, customize (see Figure 30) your "Zynq UltraScale+ MPSoC" to generate clocks with these frequencies on the `FCLK_CLK0` and `FCLK_CLK1` ports, respectively.

k) Run *Connection Automation* (see Figure 31) to connect the AXI ports of the video modules to the "Zynq UltraScale+ MPSoC". Vivado should automatically add "Processor System Reset" and a "AXI SmartConnect" IPBs. Check your connections by investigating how the clk, reset and AXI ports are connected, i.e., compare the design after connection automation has run with the one given in Figure 24.

l) Add "VideoSink" and "VideoSource" IPBs for HDMI input and DisplayPort output and complete the connections according to Figure 24.

m) Edit the *ZCU102_Master.xdc* constraint file to specify the FPGA-pin locations for the DisplayPort signals. Make sure that the `constrs_1` constraint set containing the *ZCU102_Master.xdc* constraint file is activated.

n) Specify where the register files of the "VideoSourceFromMem" and "VideoSinkToMem" IPBs should be located in the address space of the CPU (see Figure 32). There, we see that the register file of the "VideoSourceFromMem" IPB can be accessed by the CPU starting from address `0x43C00000` by using the `S_AXI` target (i.e., bus slave) interface of the hardware module. Moreover, the register file of the "VideoSinkToMem" IPB can be accessed by the CPU starting from address `0x43C10000` by using the `S_AXI` target (i.e., bus slave) interface of the hardware module. The addresses which are assigned here must be identical to

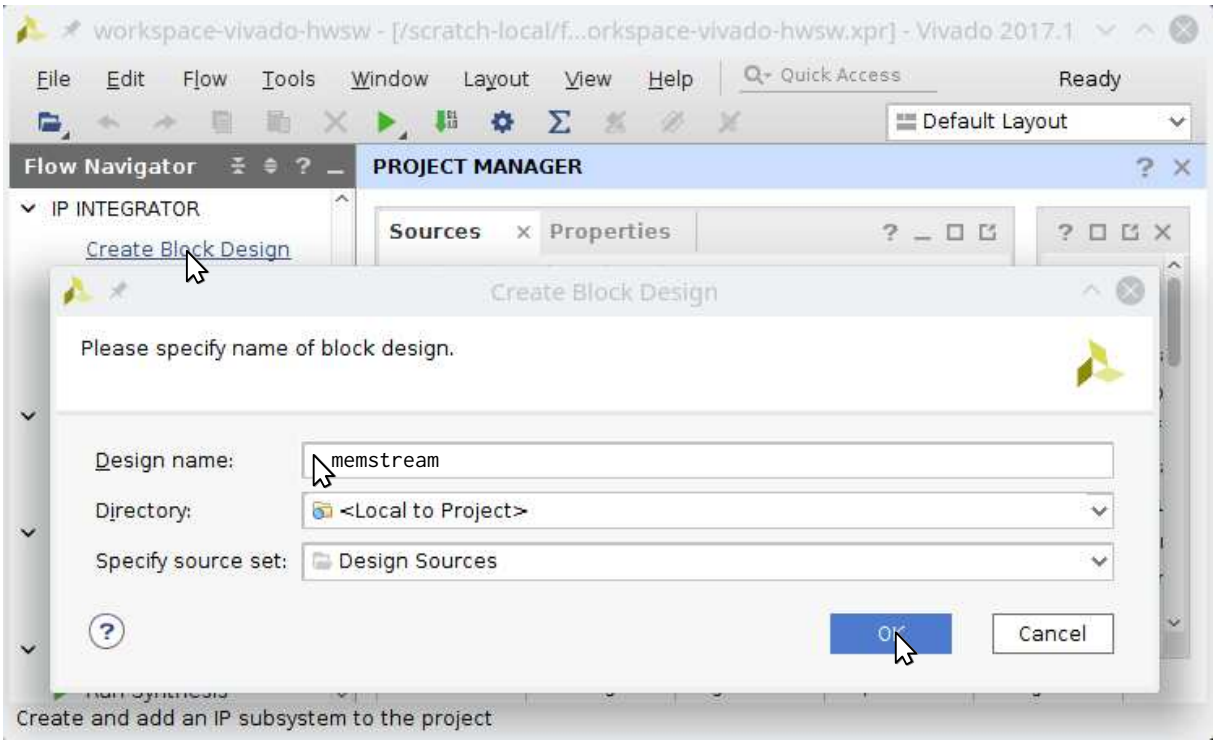


Figure 25: Create the `memstream` block design

the addresses given by the `VIDEOSOURCEFROMMEM_CTRLADDR` and `VIDEOSINKTOMEM_CTRLADDR` defines that are used to declare the `ctrlMEM2VS` and `ctrlVS2MEM` references. Otherwise, the hardware you are currently creating and the software that will run on it do not agree on the location of the register files that are controlling the hardware modules `VideoSourceFromMem` and `VideoSinkToMem`. Thus, your software will write to a location where no memory is present in the real system and, thus, your program will die.

Moreover, we see that the "VideoSourceFromMem" and "VideoSinkToMem" IPBs can access the DDR RAM by using their `M_AXI` initiator (i.e., bus master) interface. This interface will be used by the IPBs to facilitate their DMA transfers. From the perspective of these IPBs, the DDR RAM is accessible from address 0x00000000 to address 0x1FFFFFFF, i.e., all 512 MiB of DDR RAM. This is the same address range under which the CPU sees the DDR RAM and, thus, no special address transformations have to be performed.

o) Create a HDL wrapper around the block design and set it as the top module (see Figures 12 and 13 but for the `memstream` block design). If the *Set as Top* command is grayed out, then the wrapper is already the top module.

p) Start the synthesis as seen in Figure 14 but for the `memstream` block design and using the `constrs_1` constraint set.

In order to copy to bit file and the initialization files for the "Zynq UltraScale+ MPSoC" to the `swVideoSourceToSinkOverMem` project, adapt the steps from Figures 15 to 18. Here, the bit file should be named *memstream_wrapper.bit* and the destination project you will paste the files into is `swVideoSourceToSinkOverMem`. Remember, if you have chosen a different name for your block design, the bit-file name will change according and you have to rename it or modify *config.sh* and *src/main/CMakeLists.txt* to accomodate your changed bit-file name. Finally, use the scripts `ZCU102Init.sh` or `ZCU102Flash.sh` to either program or flash the ZCU102 board.

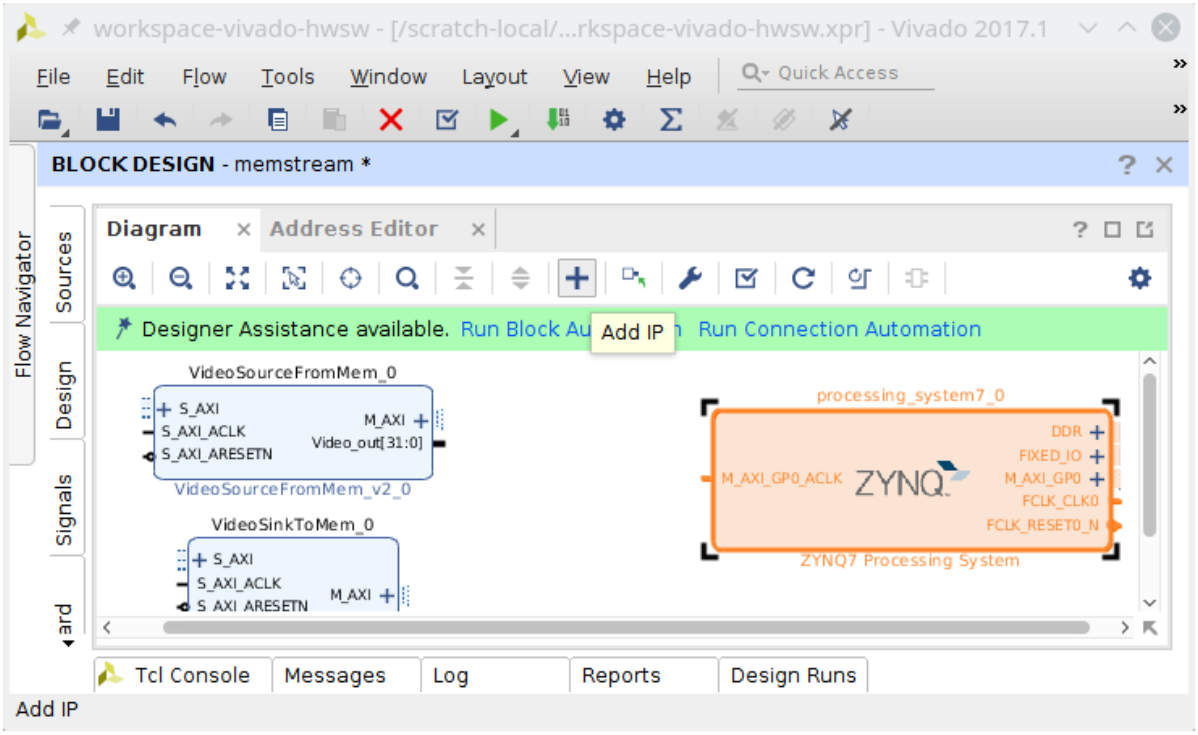


Figure 26: Block design with IPBs "VideoSourceFromMem", "VideoSinkToMem", and "Zynq UltraScale+ MPSoC"

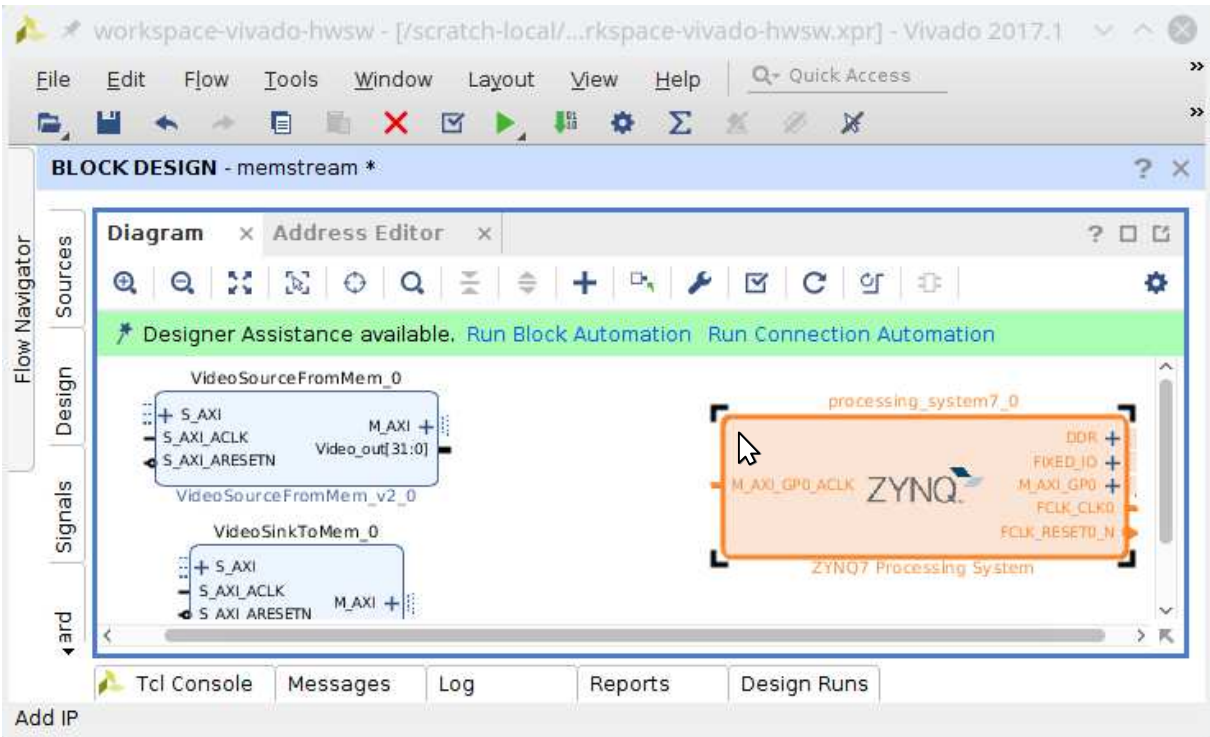


Figure 27: Select "Zynq UltraScale+ MPSoC" for Re-customization

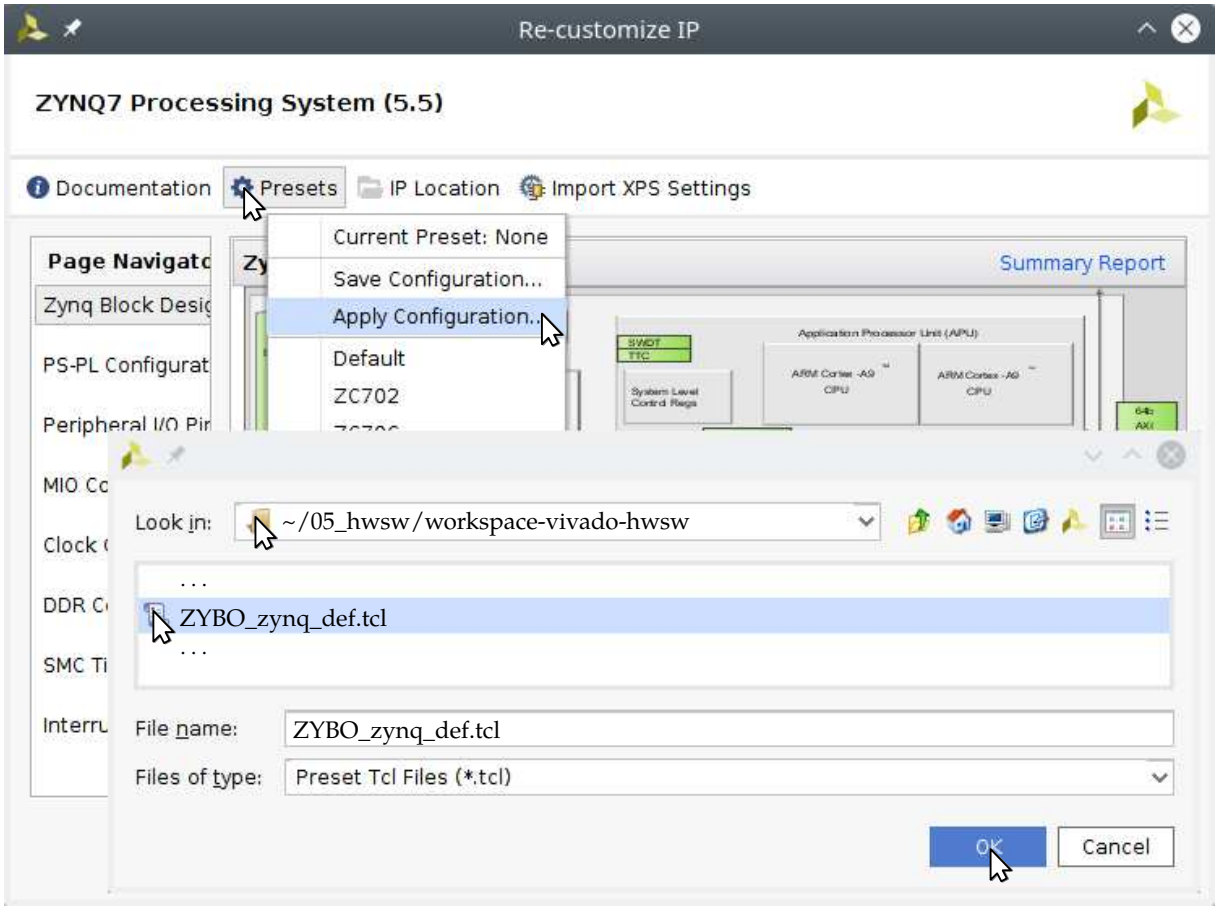


Figure 28: Apply the *ZCU102_zynq_def.tcl* preset for the "Zynq UltraScale+ MPSoC"

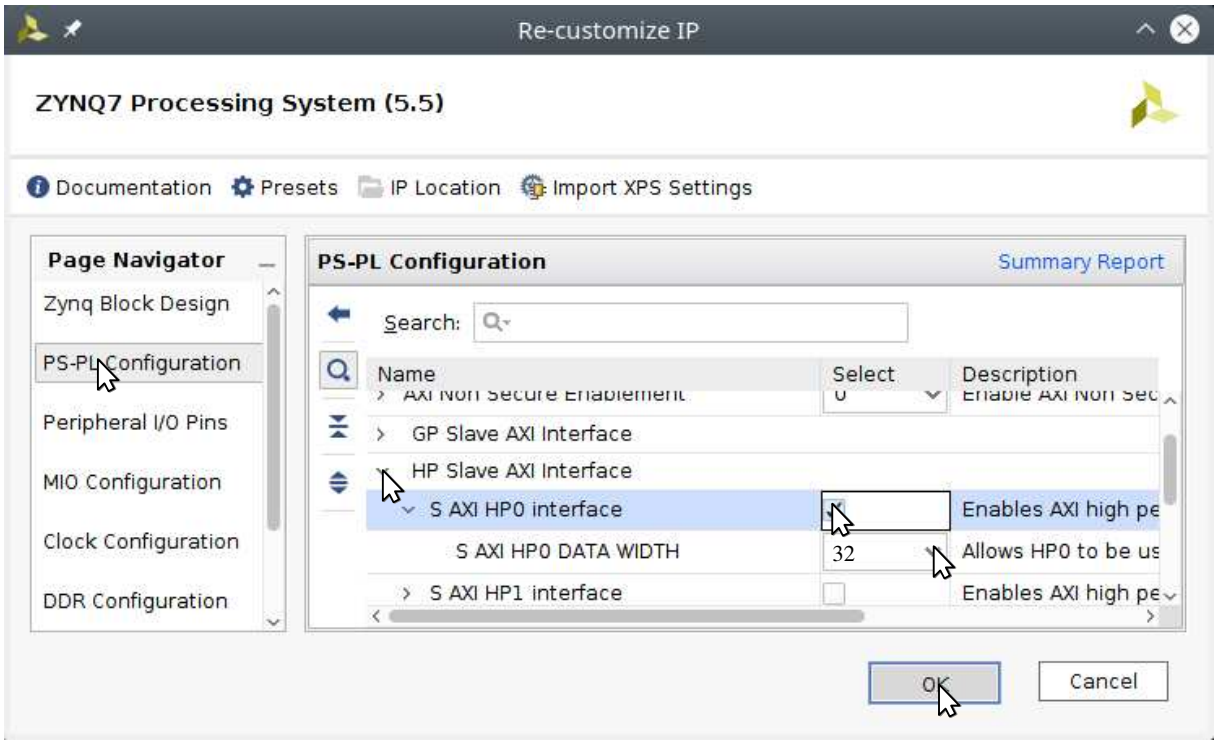


Figure 29: Add a "AXI HP Slave" port

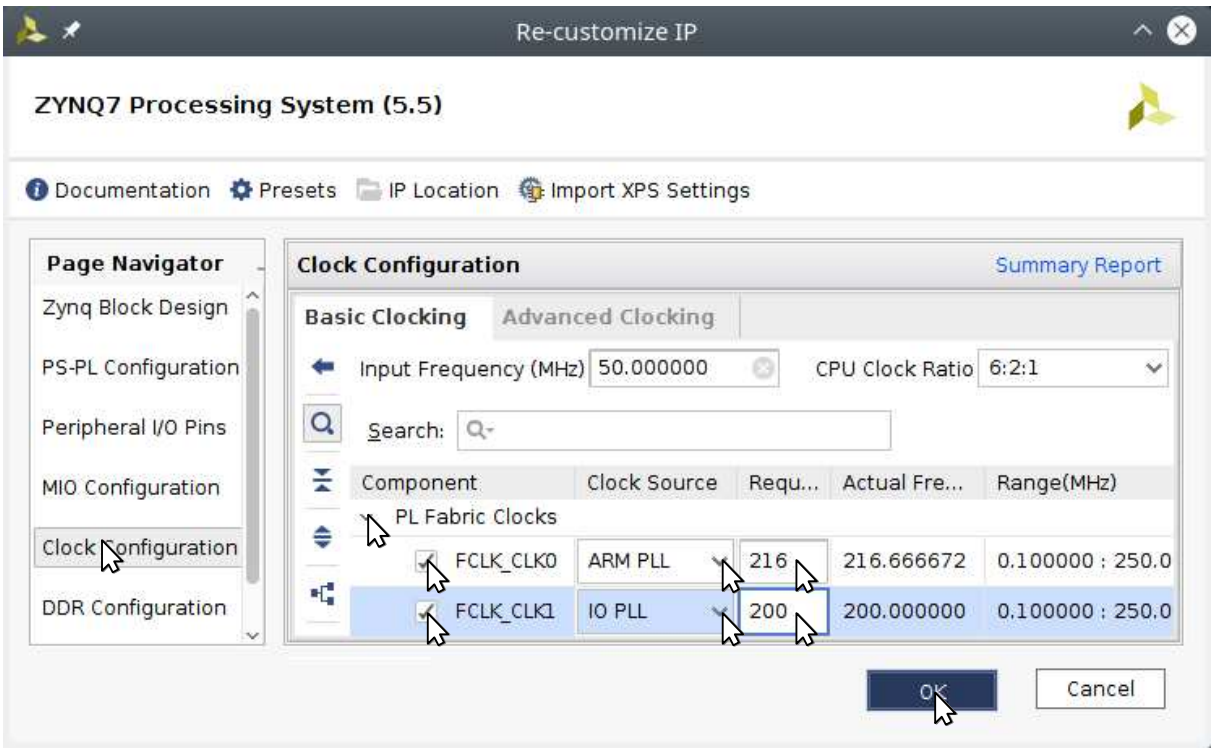


Figure 30: Add 216 MHz and 200 MHz clocks on ports `FCLK_CLK0` and `FCLK_CLK1`, respectively

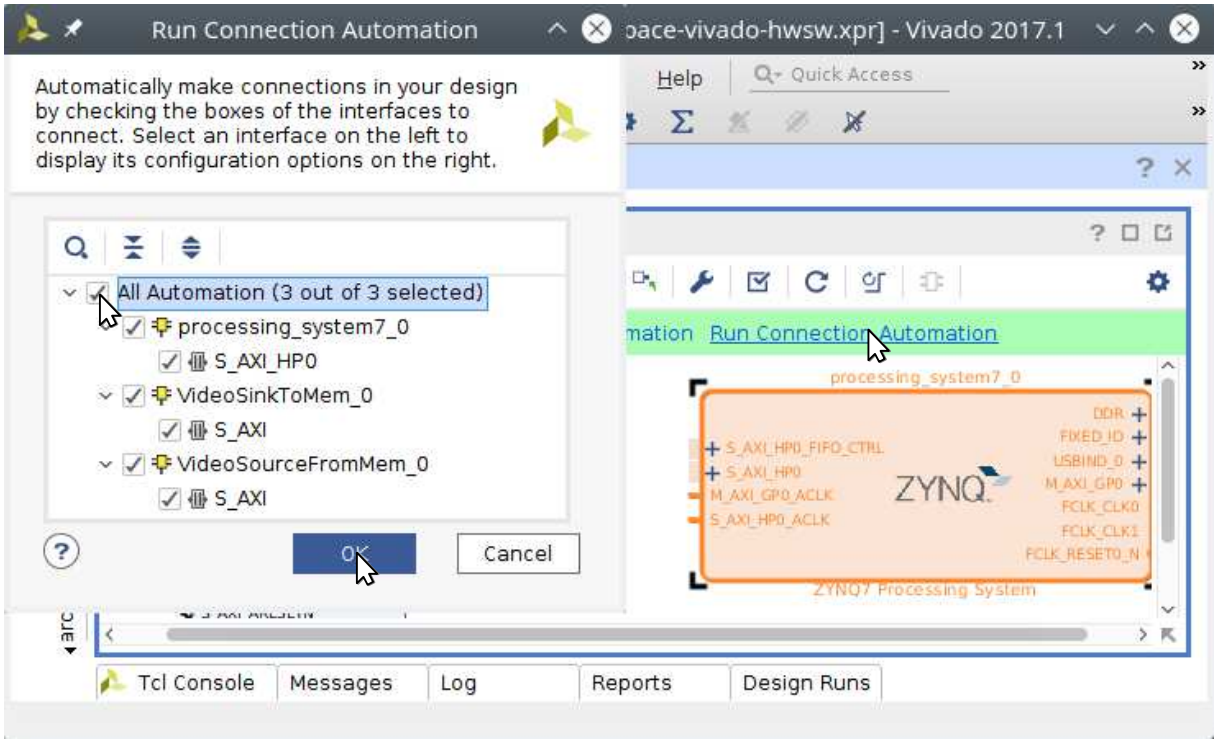


Figure 31: Run *Connection Automation*

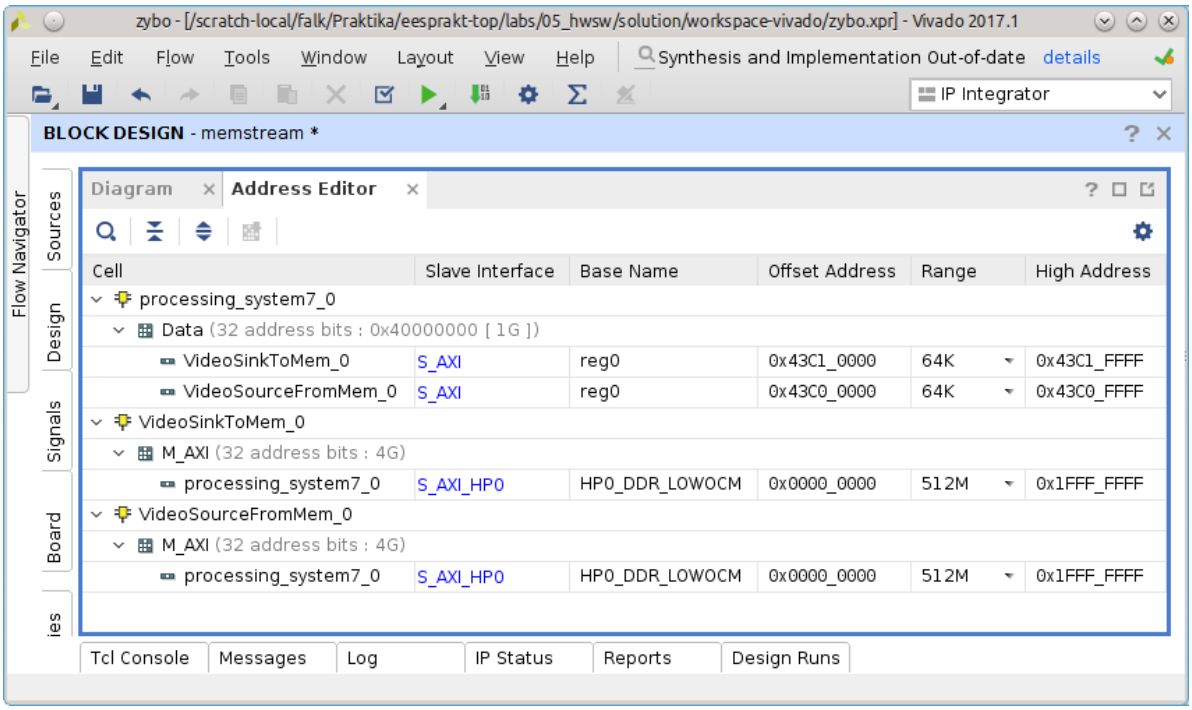


Figure 32: Address mapping for the architecture from Figure 23c

Task 2 (Extend the Skin Color Detector to Detect a Parameterizable Color)

Project:	`hwVideoFilterSkinColorDetectorWithCtrl`
Files:	*src/module/headers/sw/VideoFilterSkinColorDetectorWithCtrl.hpp*
	src/module/cpp/VideoFilterSkinColorDetectorWithCtrl.hpp
	src/module/cpp/VideoFilterSkinColorDetectorWithCtrl.cpp
Project:	`swVideoFilterSkinColorDetectorWithCtrl`
Files:	*src/libCalibrateSKCD/cpp/calibrate.cpp*
	src/libCalibrateSKCD/headers/calibrate.hpp
	src/main/cpp/main.cpp

The `swVideoFilterSkinColorDetectorWithCtrl` project consists of the *libCalibrateSKCD* library and a corresponding test program. The test program *src/main/cpp/main.cpp* simply uses the `calibrate` function to calibrate the `VideoFilterSkinColorDetectorWithCtrl` hardware accelerator to detect certain colors that have to be provided in the *calibrate regions* (see Figure 33a and the red and blue boxes) of the input video stream while the calibration phase is running. At system startup, the test program uses the function `testVideoSourceFromMem` to display the test sequence given in Figure 20. Then, the `testVideoSinkToMem` function is used to setup a situation where the video input is streamed over the memory to the video output (see Figure 33g). This situation allows the test program to subsequently call the `calibrate` function, which accesses the streamed video images that are now available in memory to perform *histogram calibration* on the colors in the calibrate regions. Finally, the detected color ranges derived via histogram calibration are used to parameterize the hardware accelerator to detect the calibrated colors. In the following, it is your task to complete the implementation of the *libCalibrateSKCD* library.

For this purpose, your have to use the register file (see Figure 33g) to communicate between the `VideoFilterSkinColorDetectorWithCtrl` hardware accelerator and the *libCalibrateSKCD* library. This register file is defined in the header file *VideoFilterSkinColorDetectorWithCtrl.hpp* contained in the *src/module/headers/sw* directory of the `hwVideoFilterSkinColorDetectorWithCtrl` project. The video filter hardware accelerator can be operated in two modes: (a) *calibrate mode* (see Figures 33a to 33c) where the video filter should draw a red and blue box signifying the calibrate regions into which the user should place the colors that should later be detected by the filter, and (b) *detection mode* (see Figures 33d to 33f) where the previously calibrated colors that will be parameterized by the *libCalibrateSKCD* library should be detected and signaled by the `skincolor` bits in the video stream.

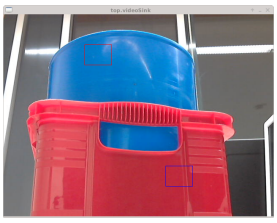
To start your implementation, you have to modify the source file *implementation/VideoFilterSkinColorDetectorWithCtrl.cpp* in the `hwVideoFilterSkinColorDetectorWithCtrl` project to realize the register file in the hardware accelerator. There, you have to make the following implementation steps:

a) Modify the video filter module to have the required TLM socket to connect the control register of this video filter to the rest of the system. Declare the socket in the header of the video filter and also name it accordingly in the constructor of the video filter.

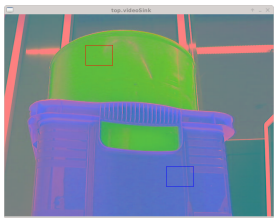
b) Use the `RegisterFileTLM` template to represent the register file for this video filter. The exact layout of the register file is specified in the `Control` structure. Declare the register file as a member variable of the class and also name it accordingly in the constructor of the video filter.

c) Forward the `ctrl_socket` of the video filter to the `m_memory_socket` of the register file `memCtrl`.

d) Implement the functions `isCalibrateMode`, `getColorSpace`, ... `getBVYHigh2`, which are used by the `pixelThrd` process to access the register file. If you only want to detect one color, you might want to stop with `getBVYHigh1`.

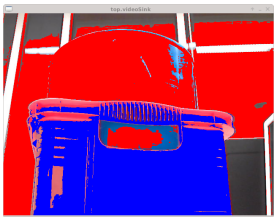


(a) Calibration in the RGB color space

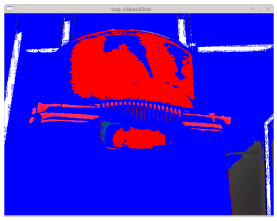


(b) Calibration in the YUV color space

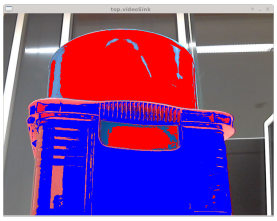
(c) Calibration in the HSY color space



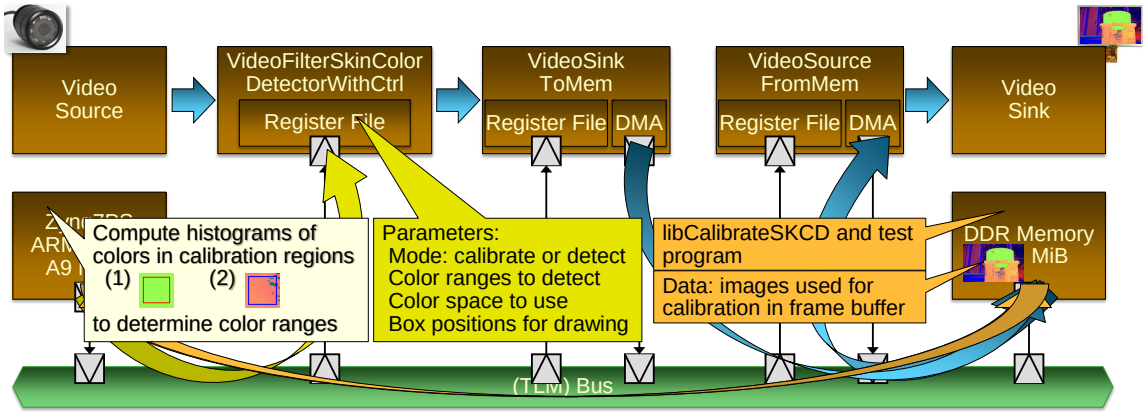
(d) Calibration result in the RGB color space



(e) Calibration result in the YUV color space



(f) Calibration result in the HSY color space



(g) System architecture

Figure 33: Possible steps in the calibration process

e) In detect mode, use `detectViaRGBSettings[0].{low,high}.{r,g,b}`, which defines the first color range, to detect pixels in this range. If you do need to detect a second color, use `detectViaRGBSettings[1].{low,high}.{r,g,b}`, which defines the second color range, to detect pixels in this range.

f) You may also support multiple color spaces by converting to the YUV or HSY color space according to the `colorSpace` parameter in the register file. The benefits of different color spaces can be seen in Figures 33d to 33f.

After the video filter uses the color ranges from the register file for the detection of the correspondingly colored pixel, you can modify the `main` function in *main.cpp* source file to provide some example color ranges to check your implementation of the video filter hardware accelerator. First, however, you will need to make the following modifications in the *calibrate.hpp* header:

g) Add a declaration for `ctrlSKCD` to the *calibrate.hpp* header file. Take care to use the default address for the register file the hardware module `VideoFilterSkinColorDetectorWithCtrl` specified in the header *sw/VideoFilterSkinColorDetectorWithCtrl.hpp* contained in the project `hwVideoFilterSkinColorDetectorWithCtrl`.

h) Add the `ctrl` parameter of the appropriate `Control` structure type to the `dumpRegs` function. This `Control` structure specifies the register file that should be dumped. It is declared by the `hwVideoFilterSkinColorDetectorWithCtrl` project in the header file *sw/VideoFilterSkinColorDetectorWithCtrl.hpp*.

i) Add the parameters `colorSpace`, `ctrlSKCD`, and `ctrlVS2MEM` of the appropriate control structure types to the `calibrate` function.

Now, the `ctrlSKCD` structure and is declared and header usable, you can begin to specify color ranges in the *main* function. You may also want to already implement the `dumpRegs` function to get better debugging support.

j) In the `main` function, switch off calibrate mode and switch to RGB color space in `ctrlSKCD`. Then, use `detectViaRGBSettings[0]` with the color range r: $0-162$, g: $104-254$, and b: $78-237$ to test your color detector. If a second color is needed, use `detectViaRGBSettings[1]` with the color range r: $162-253$, g: $21-212$, and b: $38-108$ to test your color detector. These ranges correspond to the calibration achieved in Figure 33d.

k) Also, wait for five frames or an appropriate delay to pass before calling the calibrate function.

To test your changes, you can start the test program as described in Figures 34a to 34d. Subsequently, to enable drawing of calibrate region boxes at different positions in the video stream, modify the `pixelThrd` method in hardware accelerator as follows:

l) Write code to determine the current x and y position of the pixel from the video stream. Here, you can use the implementation you already realized for the x and y position determination in the video filter `VideoFilterImageColorer` from the previous lab three.

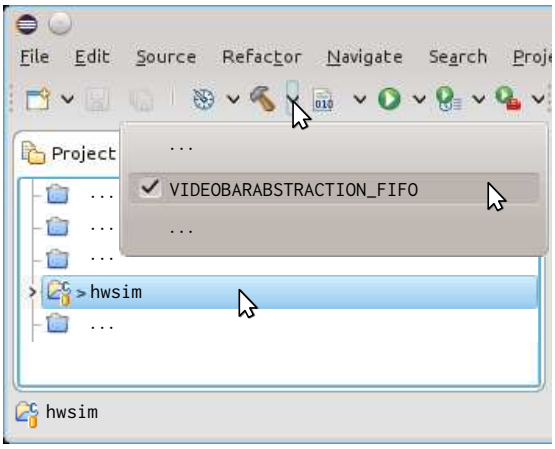
m) In calibrate mode, draw the first box given by `calibrateSettings[0].boxStart.posx` and `posy` and `calibrateSettings[0].boxDims.width` and `height`. If you do need to detect a second color, draw the second box given by `calibrateSettings[1].boxStart.posx` and `posy` and `calibrateSettings[0].boxDims.width` and `height`.

Next, to parameterize the register file of the hardware accelerator to draw the first or second box at different positions in the video stream, modify the *calibrate* function in test *calibrate.cpp* source file as follows: For this purpose, you should modify the `calibrate` function as follows:

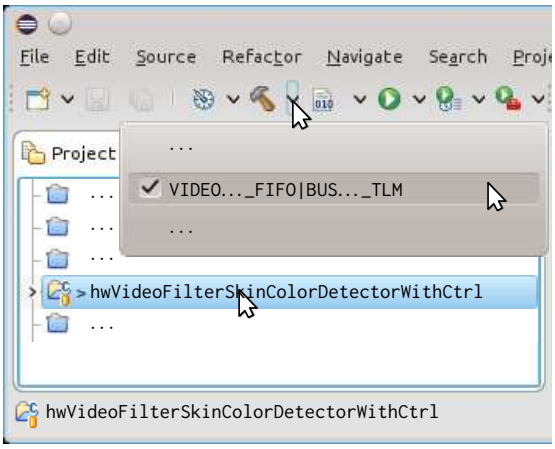
n) Implement the `dumpRegs` function to dump the fields of the control structure that you are interested in for debugging purposes. You can take notes from the implementation of the `dumpRegs` function for the `VideoSourceFromMem` hardware module.

o) In the `calibrate` function, switch on calibrate mode in register file `ctrlSKCD`. Then, use `ctrlSKCD.calibrateSettings[0]` to display a calibrate box of dimensions $10\,\%$ of the input image width and height with the lower right corner at $40\,\%$ width and $25\,\%$ height of the input image. If you need a second color, use `ctrlSKCD.calibrateSettings[1]` to display a calibrate box of the same dimensions but with the upper left corner at $60\,\%$ width and $75\,\%$ height of the input image. An example of such calibration boxes can be seen as the red and blue boxes in the Figures 33a to 33c.

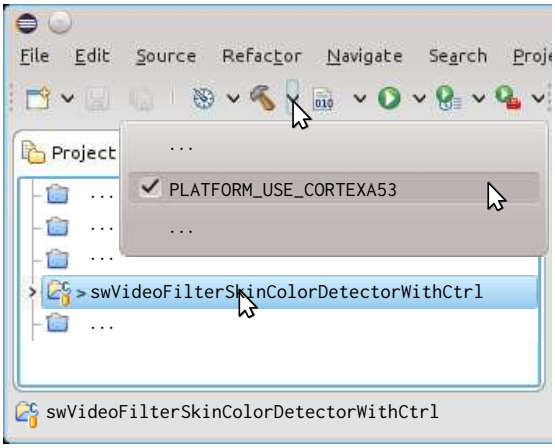
To test your changes, you can start the test program as described in Figures 34a to 34d. Now, we are interested in automatically determining the color ranges via histogram calibration. Here, we have already provided to you the `ChannelRange` and `Histogram` class that represent a color range on one color channel and a histogram for all color channels, respectively. Moreover, there is the `CalibrateRegion` class that represents the position and dimension of a calibrate region in the image, i.e., an instance of such a structure should carry the same information as either `calibrateSettings[0]` or `calibrateSettings[1]`. In detail, the method `countPixelColors` of



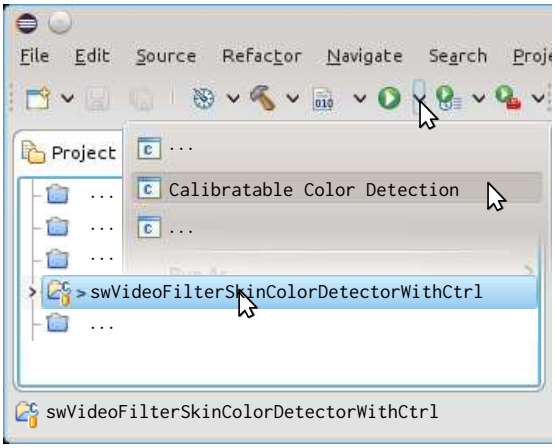
(a) Compiling the *hwsim* simulator



(b) Compile SystemC model for the hardware accelerator `VideoFilterSkinColorDetectorWithCtrl`



(c) Compile the *libCalibrateSKCD* library and its corresponding test program



(d) Run the test program for the *libCalibrateSKCD* library on the `hwsim` simulator

Figure 34: Extend the skin color detector to detect a parameterizable color determined via calibration at program start

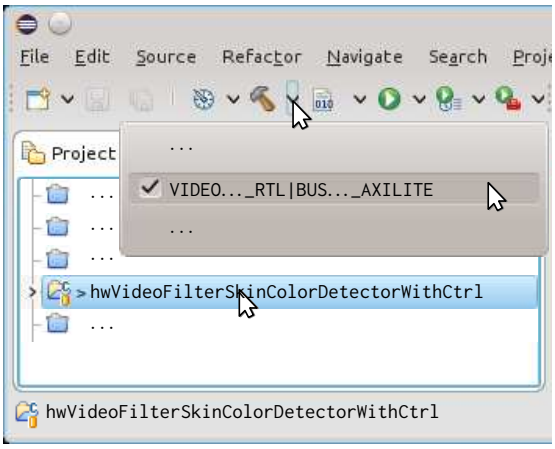
a `Histogram` instance with a `CalibrateRegion` parameter is used to accumulate the histogram values for the three different color channels of the pixels in the calibration region. You should call `countPixelColors` with multiple input images and have two different `Histogram` instances that is one for each calibration region. Finally, you can use `getSpanRGB_{R,G,B}` to get the color ranges for the three different color channels.

p) However, you should not immediately start with histogram calibration in order to give the user time to move the desired colors into the calibrate regions. Here, it is useful to slightly move the position of the calibration boxes during the waiting time to give feedback to the user that calibration has not yet begun. Select an appropriate waiting time an slightly shake the position of the calibration boxes during this time.

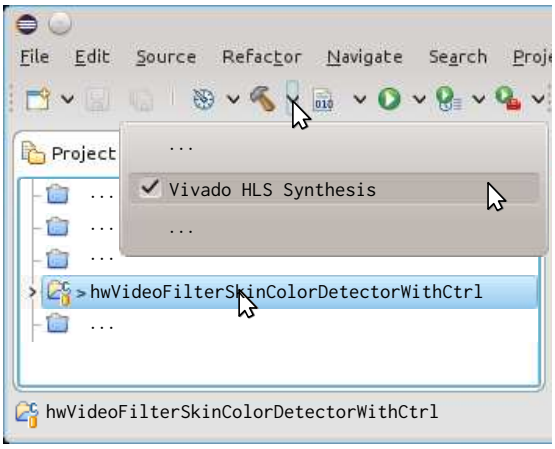
q) Call `countPixelColors` for each calibration region with five different input images.

r) Enable skin color detection with the histogram bounds the `getSpanRGB_{R,G,B}` methods have determined.

Next, you should perform the following changes to your hardware accelerator and guard them to be only active for the AXI-Lite bus protocol:



(a) Compile the SystemC model for the hardware accelerator at RTL



(b) Use Vivado HLS for hardware synthesis of the hardware accelerator

Figure 35: To bring the video filter to hardware first (a) try to get it to work at RTL level and then (b) use Vivado HLS for synthesis

s) Modify the video filter module to have the required AXI-Lite bus protocol ports to connect the control register of this video filter to the rest of the system. Declare the ports in the header of the video filter and also name them accordingly in the constructor of the video filter.

t) Use the `RegisterFileAXILite` template to represent the register file for this video filter. Declare the register file as a member variable of the class and also name it accordingly in the constructor of the video filter.

u) Forward the AXI-Lite bus protocol ports to the `memCtrl` register file.

v) Declare signals for connecting them to the `memCtrl.mem[n]` output ports of the register file and connect them to these ports in the constructor of the video filter.

w) Implement the functions `isCalibrateMode`, `getColorSpace`, ...`getBVYHigh2`, which are by the `pixelThrd` process to access the register file. To retrieve the values, you have to use the signals you have declared in the previous step. If you only want to detect one color, you might want to stop with `getBVYHigh1`.

Check your modifications by compiling your video filter as depicted in Figure 35a and start the simulation by selecting the run target shown in Figure 34d. The *hwsim* simulator and your test program should still be compiled with the targets shown in Figures 34a and 34c, respectively. If you have successfully checked your changes, then you should use Vivado HLS behavioral synthesis as depicted in Figure 35b. If yes, realize the architecture depicted in Figure 33g on the ZCU102 board.

For this purpose, create the `skcdctrl` block design as shown in Figure 36. It is of utmost importance that the address mapping (see Figure 37) is consistent with the register file addresses given by the defines `VIDEOSOURCEFROMMEM_CTRLADDR` and `VIDEOSINKTOMEM_CTRLADDR` as well as the define `VIDEOFILTERSKINCOLORDETECTORWITHCTRL_CTRLADDR`. Otherwise, the hardware you are currently creating and the software that will run on it do not agree on the location of the register files that are controlling the hardware modules contained in the design.

Next, if you have chosen a different name for your block design, you will have to modify *config.sh* and *src/main/CMakeLists.txt* in `swVideoSourceToSinkOverMem`. Don't forget to also copy *psu_init.h*, *psu_init.c*, and *psu_init.tcl*, i.e., adapt the steps in Figures 15 to 18 for the current bit-file name and this destination project. Finally, execute the program on the real hardware

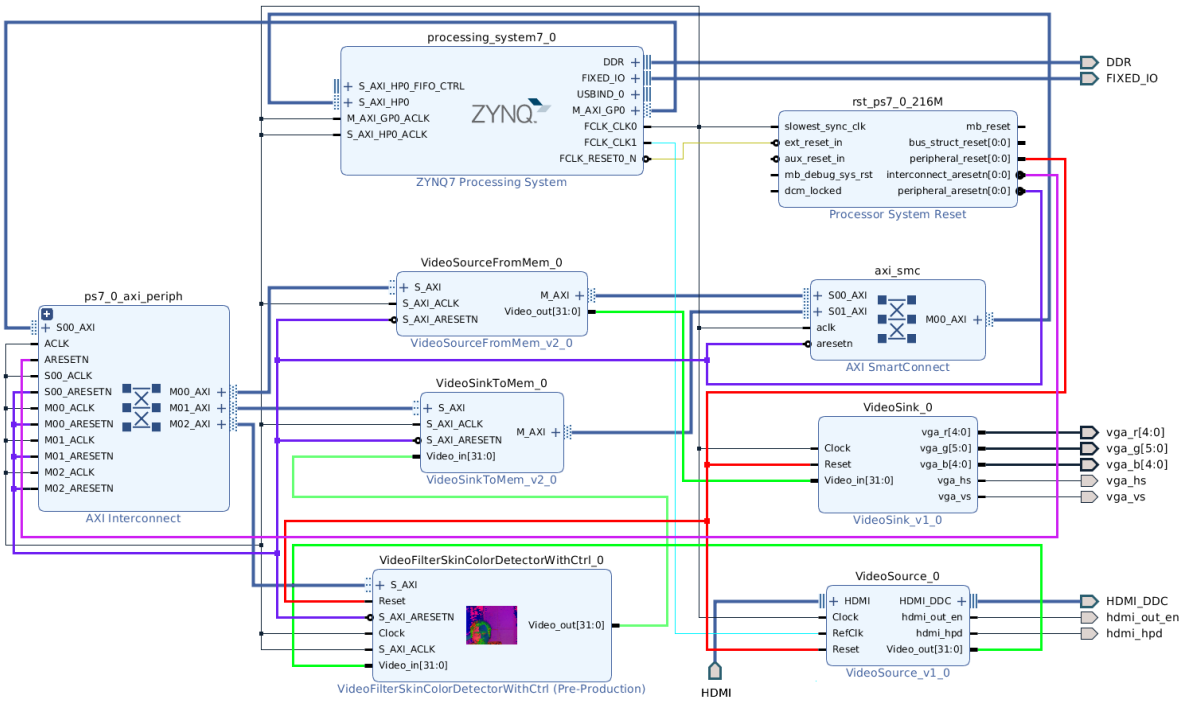


Figure 36: Architecture from Figure 33g realized in Vivado for the ZCU102 board

by using the `ZCU102Init.sh` and `ZCU102Flash.sh` scripts to either program or flash the ZCU102 board.

workspace-vivado-hwsw - [/scratch-local/falk/Praktika/...vivado-hwsw/workspace-vivado-hwsw.xpr] - Vivado 2017.1

File Edit Flow Tools Window Layout View Help Quick Access write_bitstream Complete

Default Layout

BLOCK DESIGN ccbctrl skcdctrl

Address Editor Diagram

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
VideoSourceFromMem_0	S_AXI	reg0	0x43C0_0000	64K	0x43C0_FFFF
VideoSinkToMem_0	S_AXI	reg0	0x43C1_0000	64K	0x43C1_FFFF
VideoFilterSkinColorDetectorWithCtrl_0	S_AXI	Reg	0x43C2_0000	64K	0x43C2_FFFF
VideoSourceFromMem_0					
M_AXI (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF
VideoSinkToMem_0					
M_AXI (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF

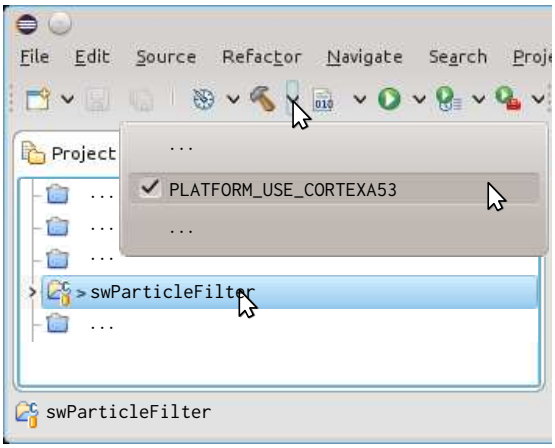
Tcl Console Messages Log Reports Design Runs

Figure 37: Address mapping for the architecture from Figure 33g

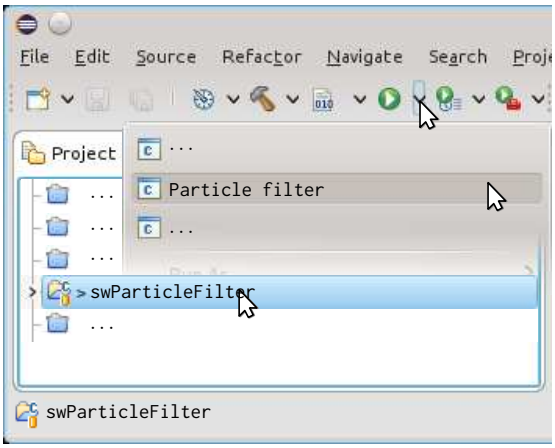
Task 3 (Realize a Particle Filter to Detect Blobs of the Parameterized Color)

Project:	`swParticleFilter`
Files:	*src/libParticleFilter/cpp/ParticleFilter.cpp*
	src/libParticleFilter/headers/ParticleFilter.hpp
	src/main/cpp/main.cpp

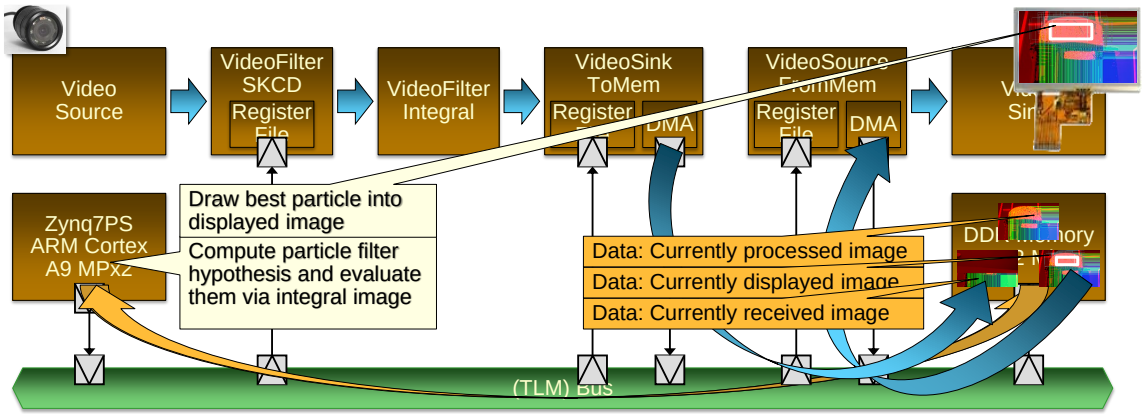
In the following, you will try to track skin color blobs by applying a particle filter on a preprocessed video stream derived by using the filters `VideoFilterSkinColorDetectorWithCtrl` and `VideoFilterIntegral` as preprocessing steps. You might also choose to use the hardware module `VideoFilterSkinColorDetector` in a first try in order to skip the calibration step required for the `VideoFilterSkinColorDetectorWithCtrl` hardware module. In this case, adjust the run configuration used in Figure 38b accordingly, i.e., change the argument `--video-filter VideoFilterSkinColorDetectorWithCtrl` to `--video-filter VideoFilterSkinColorDetector`.



(a) Compile the *libParticleFilter* library and its corresponding test program



(b) Run the test program for the *libParticleFilter* library on the `hwsim` simulator



(c) System architecture

Figure 38: Realize a Particle Filter to Detect Blobs of the Parameterized Color

The `swParticleFilter` project consists of the *libParticleFilter* library and a corresponding test program.

a) Use the `testVideoSourceFromMem` and `testVideoSinkToMem` methods from the libraries *libVideoSourceFromMem* and *libVideoSinkToMem* to setup passing of the video stream over memory.

b) In case you use the `VideoFilterSkinColorDetectorWithCtrl` hardware module, use the `calibrate` method from the *libCalibrateSKCD* library to setup the parameters for this hardware module.

c) Allocate two more frame buffers for the incoming video stream in order to realize triple buffering, i.e., you will have one buffer receiving a new image, one buffer currently being processed by the particle filter, and one buffer currently being displayed on the output.

d) Instantiate the particle filter. Use 256 particles for tracking. The Width and height parameters have to be set according to the input image size.

e) Manage the triple buffering by switching the pointers for incoming, processed, and outgoing image. Take care that you receive a full image into the new image frame buffer. Otherwise, you will have a frame buffer containing two different integral images and the particle filter will not work correctly.
Hint: You can use `dim.count` to detect that a new image starts.
Hint: You can use `buf.syncerror` to detect that you switched the `buf.addr` pointer while the image was still in the process of being stored in the memory.

f) Use the `track` method of the particle filter to try to track the skin color blob.

g) Use the `drawParticles` or `drawBestParticle` method to get a hint of what the particle filter is tracking.

h) Parameterize the `VideoSourceFromMem` hardware module to display the image into which the particles or best particle where just drawn by the particle filter.

i) Implement the weight function for a particle. Assume that the `imageBuffer` contains the integral image you calculated in your `VideoFilterIntegral` hardware module.

Finally, you should realize the architecture as depicted in Figure 38c on the ZCU102 board and execute the program on the real hardware.

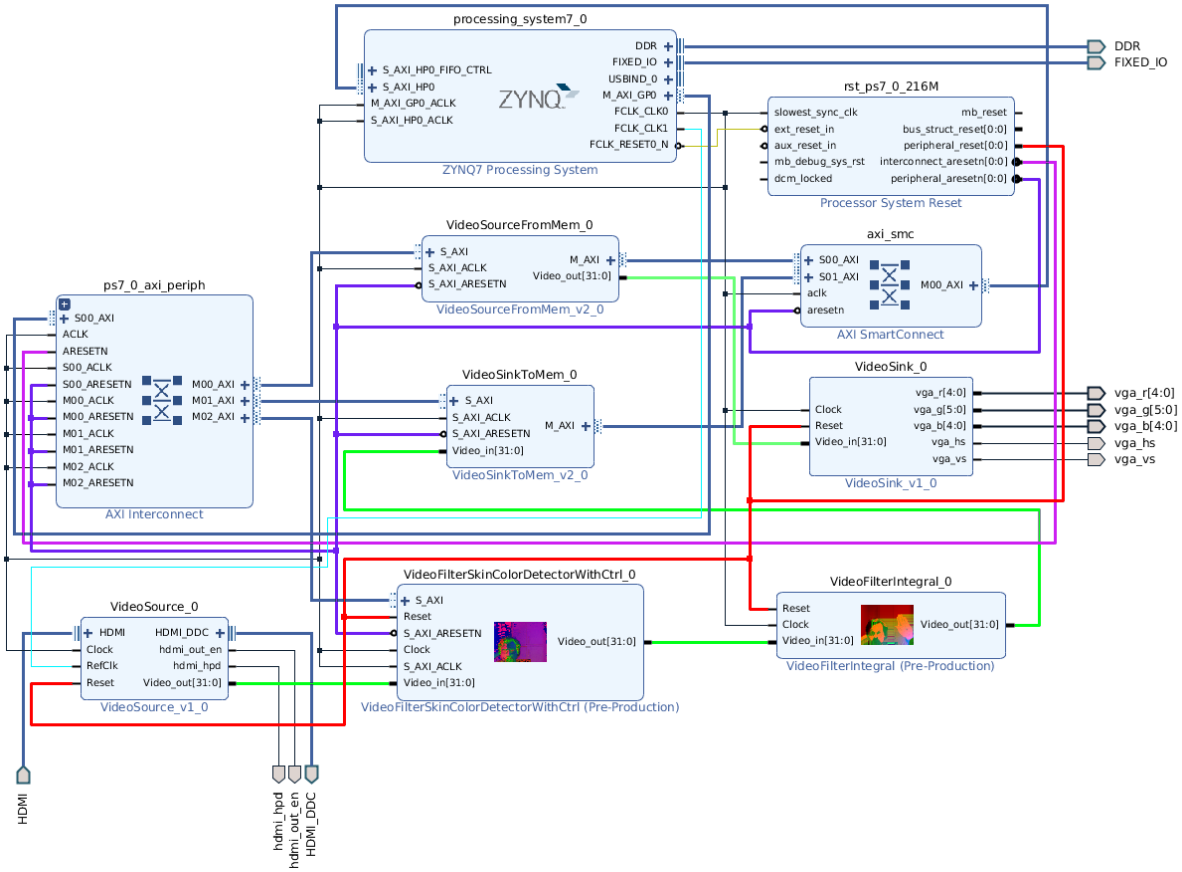


Figure 39: Architecture from Figure 38c realized in Vivado for the ZCU102 board