

Department of Information Technology and Electrical Engineering

**VLSI I:  
From Architectures to VLSI Circuits and FPGAs**

227-0116-00L

Exercise 7

---

**High-Level Synthesis**

---

Prof. L. Benini  
F. Gürkaynak  
M. Korb

Last Changed: 2023-12-04 14:40:11 +0100

**Reminder:**

With the execution of this training you declare that you understand and accept the regulations about using CAE/CAD software installations at the ETH Zurich. These regulations can be read anytime at

<http://eda.ee.ethz.ch/index.php/Regulations>.

# 1 Overview

High-level synthesis (HLS) enables automated RTL generation from a behavioral model coded in a high-level description language such as C or C++. Such a high-level description of the module's functionality allows for a higher design productivity compared to regular RTL design and an easy adaption to different timing requirements. In this exercise we will demonstrate a high-level synthesis flow and its benefits using a stencil image filter that will be integrated into our video processing pipeline.

After setting up an HLS project within Vivado HLS the filter algorithm is modelled in C++ followed by the high-level synthesis step in which RTL code is automatically generated. The functional correctness of the initial C++ code, and for some designs also of the generated RTL, is checked within the Vivado HLS tool using a provided C++ testbench. After a first HLS synthesis has been successfully concluded, the high-level synthesis flow enables a fast adaption of the stencil filter to different application scenarios. As an example, a second stencil filter with a different area / throughput trade-off will be generated. In the final part of the exercise, the high-level-synthesized stencil filter is integrated into the video filter pipeline. The quality of the automatically generated RTL is evaluated by comparing the FPGA utilization of the manual and HLS-generated filter.

## 2 Getting Started

Initially, the provided design environment and source files need to be copied to the user's home directory.

**Student Task 1:** Setup the working directory for this exercise by calling our install script and changing to the newly created directory:

```
sh> /home/vlsil/ex7/install.sh
sh> source ~/.bashrc
sh> cd ex7
```

Note the `source ~/.bashrc` line - as this exercise uses Python scripts with package dependencies, the `install.sh` script installs a Python virtual environment manager<sup>a</sup> and modifies the bash environment setup script, `~/.bashrc`. In order for those changes to take effect, this script needs to be read again after the changes are made.

**If you have issues related to installing miniconda due to the disk quota being exceeded:** Have a look at where the disk space is being used by running the following command:

```
sh> du -h --max-depth=1 ~ | sort -h
```

Most likely, the folders "eating" the most disk space will be related to previous exercises - delete the biggest ones, and try the install process again!

<sup>a</sup> Conda: <https://docs.conda.io/en/latest/>

Your current working directory has the following structure:

```
workspace
├── cpp ..... Directory holding all the C++ code
│   ├── filter_app.cpp ..... C++ testbench
│   ├── filter_hls.cpp ..... C++ model of the stencil filter that is synthesized to RTL
│   └── filter_hls.h ..... Header file of the stencil filter model
├── modelsim ..... RTL testbench
├── simvectors ..... Stimuli and expected responses
├── sourcecode ..... RTL code for video processing pipeline and additional modules
├── sourcecode_hls ..... Directory used for HLS-generated RTL
├── vivado ..... Vivado working directory
│   ├── constraints
│   └── scripts
└── vivado_hls ..... Vivado HLS working directory
```

All C++ project files can be found in the `cpp` directory including the C++ model of the stencil filter and its configuration (`filter_hls.h`, `filter_hls.cpp`) as well as a C++ testbench (`filter_app.cpp`). The testbench reads an entire video frame from the `simvectors` directory, runs the filter on it, and stores the output to disk.

**Student Task 2:** Prepare the stimuli for the C++ testbench by converting the `simvectors/stim.png` file to the PNM format accepted by the testbench:

```
sh> make -C simvectors
```

Check that the `simvectors/stim.ppm` was generated and that you can view it with the system image viewer:

```
sh> tkjpeg simvectors/stim.ppm
```

## 2.1 Setting up the HLS project

**Student Task 3:**

1. In this exercise, Vivado HLS needs to run in an environment with a compatible GCC compiler version. This is needed for each terminal you open. Setup the environment:

```
sh> start-dz-env
```

Note that this will (temporarily) change your prompt to `Singularity>`. Run the following command to restore your standard bash prompt:

```
sh> source /home/vlsil/ex7/env.sh
```

2. Start the Vivado HLS user interface:

```
sh> cd vivado_hls
sh> vivado-2017.2 vivado_hls &
```

3. Then, set the HLS project up as follows:

- After starting Vivado HLS a new project needs to be created which can either be done using the “Create New Project” button in the quick start menu or in the pull down menu under “File→New Project...”.
- Chose “filter” as project name.
- Specify the working directory as `~/ex7/vivado_hls`.
- Add the design files `~/ex7/cpp/filter_hls.cpp` and `~/ex7/cpp/filter_hls.h` and choose the top-level function `filter_hls`.
- Add testbench file `~/ex7/cpp/filter_app.cpp`.
- Keep the default solution name “solution1”.
- Choose a clock period of 8ns. Note that the unit of measurement (ns) is implicitly set in the Vivado window.
- Choose the same FPGA as used in exercise 6 (under Parts, select xc7z020clg400-1).

A screenshot of the Vivado HLS GUI after setting up the project as described is depicted in section 2.1.

The HLS project is now set up correctly and the stencil algorithm can be modelled in C++ next. The better parts of both the testbench and filter C++ model are already provided.

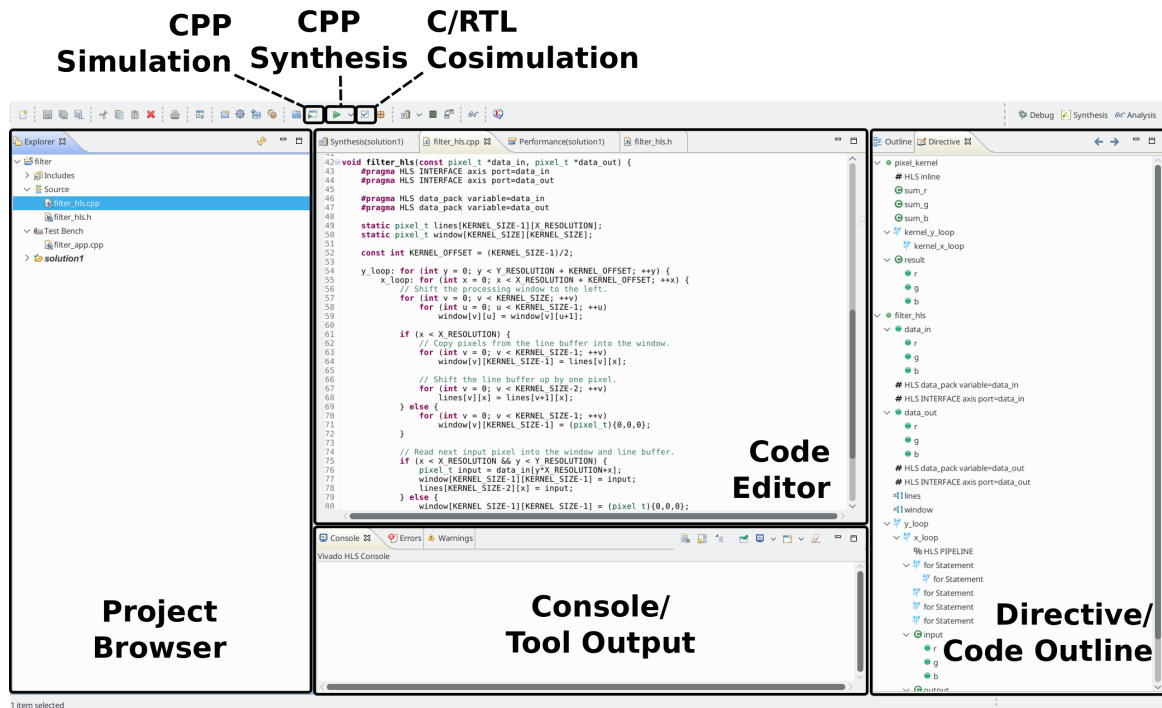


Figure 1: Annotated Screenshot of the Vivado HLS GUI

## 2.2 Stencil Project

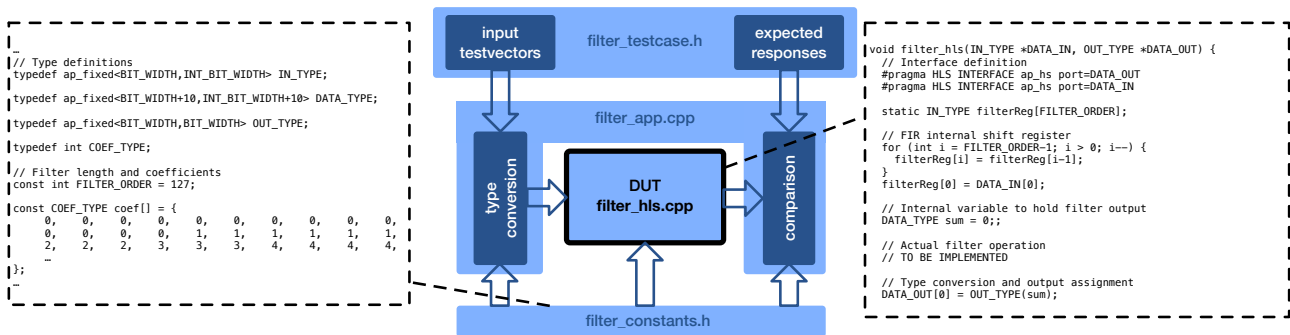
In this exercise we are using an  $n$ -by- $n$  stencil filter. The figure on the next page shows the overall structure of the used C++ project. The key component is the high-level description of the filter algorithm (**filter\_hls.cpp**) representing the DUT and encapsulated in a C++ method.

```
void filter_hls(const pixel_t *data_in, pixel_t *data_out)
```

The filter expects one Full HD video frame ( $1920 \times 1080$  pixels) to be available at `data_in` and will write the filtered result to `data_out`. The C++ testbench loads an input image into memory, prepares some empty memory for the output, and passes pointers to these memory locations to the `filter_hls` function.

**Student Task 4:** Take a look at the main function in **filter\_app.cpp** to see how this is done. You may skip over **struct** `ppm_image` which is a simple implementation of PPM image reading and writing.

Upon completion the testbench stores the filtered image to disk. We use this testbench to verify that the implementation of the filter is functional and to visualize its output. As such you can try out different filter coefficients and see their effect before starting the process of implementing the design on the FPGA.



**Student Task 5:** Get familiar with the project and its initial state by looking at all provided C++ source and header files.

## 2.3 Data Type Definitions

Vivado HLS provides synthesizable template classes for fixed-point types. The name of the fixed-point template class is `ap_fixed` which offers the following template parameters:

```
ap_fixed < DATA_BIT_WIDTH, DATA_INT_WIDTH, TRUNC_STYLE, SAT_STYLE >
```

Besides the overall and integer bit-widths `DATA_BIT_WIDTH` and `DATA_INT_WIDTH`, the truncation behavior as well as the saturation behavior can be chosen. Possible truncation and saturation styles are `TRUNC_STYLE = SC_RND` and `SAT_STYLE = SC_SAT`, for example, in which unused least-significant bits are rounded and values larger than the maximum representable value are saturated. In case of missing truncation or saturation definition as in the initial state of the project, the default configuration `TRUNC_STYLE = SC_TRN_ZERO` and `SAT_STYLE = SC_WRAP` are used which round to zero for both negative and positive values and wrap around in case of an overflow.

## 3 Getting Familiar with the Vivado High-Level Synthesis Flow

### 3.1 High-Level Description of Stencil Algorithm and Functional Verification

The file `filter_hls.cpp` contains the part of the C++ model that will be synthesized into RTL. In its initial state, the top level method `filter_hls` already contains a model of the line buffers necessary for the stencil operation. The buffers are used to keep neighboring lines at hand such that pixels from them can be multiplied with filter coefficients.

To get started the `filter_hls` function already contains a dummy filter which simply inverts the input image's color. This is done by picking the pixel at the center of the current window and reversing its color channels.

#### Student Task 6:

- Open up the `filter_hls.cpp` file and take a look at the implementation of the dummy filter.
- Run a C++ simulation with the default settings to verify the functional correctness of the code ( ). This may take a minute. Note that our C++ testbench does not compare the produced output to the desired one - it just produces an image showing the result of the simulation!
- View the output produced by the filter in `vivado_hls/filter/solution1/actual.ppm`. Does it match the expected output (`simvectors/exp_resp_task0.png`)?

One of the key benefits of using high-level synthesis compared to a manual RTL-design style is the improved designer productivity. In manual RTL design, the design of finite state machines is often error prone resulting

in long debugging sessions to assure functional correctness for all possible cases. In a high-level synthesis flow, the design of finite state machines is left to the tool and, thus, correct by construction. This is not only true when designing the DUT. A lot of design / debugging effort is spent on the testbench. A rough estimation of the design effort can be drawn from the required lines of code to model the DUT or the testbench.

## 3.2 Filter Implementation

We will now replace the dummy filter in `pixel_kernel()` with the actual implementation of a convolution. If you are not yet familiar with how convolutions work in image processing, spend a minute researching the basic principles on the internet<sup>1</sup>. The basic ingredient to the convolution is a filter kernel which in our case has `KERNEL_SIZE * KERNEL_SIZE` coefficients. To determine the value of an output pixel, the pixel's neighborhood is multiplied with the filter coefficients and summed up. More precisely, each pixel in the neighborhood window is multiplied with the corresponding coefficient in the filter kernel. As the filter coefficient window is not flipped horizontally nor vertically, the filtering process is technically a cross-correlation, not a convolution. But as the two are mathematically equivalent, the term "convolution" is often used to cover both.<sup>2</sup>

The file `filter_hls.h` defines several filter kernels (and corresponding bias and kernel size constants) that you can comment/uncomment individually to try different results. By default an inversion filter is enabled. This file also defines the `DATA_TYPE` type which you should use for the accumulation variable. It is a fixed-point type that allows higher precision while the color of a pixel is calculated, before finally rounding to an integer value.

The recipe for the filter implementation is as follows:

1. Initialize three accumulators with the `FILTER_BIAS`.
2. Iterate over the x and y axis of the window (`KERNEL_SIZE`) and perform the multiply-accumulate.
3. Clamp the accumulated values to the range `[0, 255]` (optional, but avoids artifacts).
4. Pack your accumulators into a `pixel_t` and return that.

### Student Task 7:

- Open up the `filter_hls.cpp` file and implement the convolution inside `pixel_kernel()`.
- 
- Change the `filter_hls.h` file to enable the  $3 \times 3$  box blur filter. You can comment/uncomment a block of lines with "Ctrl + /"
  - Run the C++ simulation again. Does the output match the expectations (`simvectors/exp_resp_task1.png`)? Show your implementation and resulting image to an assistant.

Convolutions allow a wide range of filters to be implemented simply by adjusting the coefficients. Let's try a different filter which we will then bring to the FPGA.

### Student Task 8:

- Open up the `filter_hls.h` file.
- Comment out the current filter coefficients and parameters.
- Uncomment the "3x3 Discrete Laplace Operator", which implements a kind of spatial derivative that resembles edge detection.
- Run the C++ simulation again. Does the output match the expectations (`simvectors/exp_resp_task2.png`)?

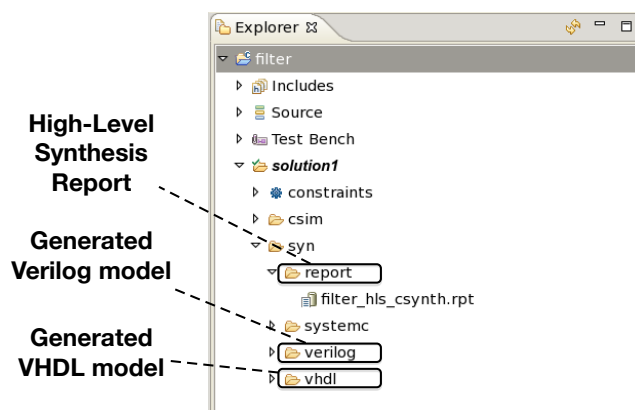
<sup>1</sup> <https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1>

<sup>2</sup> <https://bit.ly/36IfP70>


### 3.3 First High-Level Synthesis Run

Once the functionality of the stencil model has been verified, the actual high-level synthesis can be run. For the small stencil example the high-level synthesis run does finish in less than a minute. In case the synthesis is successful, a report file illustrates the synthesis results. Vivado HLS generates three different models of the synthesized design. Besides a VHDL model and Verilog model, a SystemC model is provided. All of them can be accessed using the file explorer as illustrated in the following figure.

Note that HLS interface pragmas are employed to inform the compiler about the type of I/O protocol interfaces the RTL code (Verilog/VHDL) defines. The HLS tool also provides pragmas that can be used to optimize the design: reduce latency, improve throughput performance, and reduce area and device resource usage of the resulting RTL code. These pragmas can be added directly to the source code for the kernel, as in this exercise.



#### Student Task 9:

- Run the high-level synthesis (  ).
- Analyze the generated RTL code
  - Try to find out the respective functionalities of each generated Verilog file and describe them here.  

---

---

---

---

---

---
  - Open up the `filter_hls.v` file in `solutions1/syn/verilog/`.
    - \* What interface has been synthesized for the input and output signals?  

---
    - \* Is the window array mapped to registers or memory banks?  

---
    - \* Is the lines array mapped to registers or memory banks?  

---
- Analyze the HLS report and summarize the results in the following table.

Specified Clock Period [ns]	Estimated Clock Period [ns]	Latency [Cycles]	BRAM_18K	DSP48E	FF	LUT
8						

## 4 Use-Case Specific RTL Generation

Having a working HLS run including C++- and RTL-level verification now enables a fast adaptation of the model to different use cases. In order to adapt the model to those different requirements, the C++ model itself remains unchanged. The adaption is done by adding/modifying directives in the directives window (see section 2.1). Code sections in the current source file for which directives can be set are automatically listed in this window. Note that directives can be implemented by the tool in the following ways:

1. **Directive File:** Vivado HLS inserts the directive as a Tcl command into the file **directives.tcl** in the solution directory.
2. **Source File:** Vivado HLS inserts the directive directly into the C source file as a pragma, which usage has been briefly described previously.

Let us first determine if the generated RTL would actually be able to keep up with the HDMI video data:

### Student Task 10:

- How many pixels are there in a Full HD video frame?  
\_\_\_\_\_  
\_\_\_\_\_
- Assuming a 60 Hz video signal, what is the frequency at which our filter needs to process pixels?  
\_\_\_\_\_  
\_\_\_\_\_
- Open up the synthesis report. What is the target clock frequency?  
\_\_\_\_\_  
\_\_\_\_\_
- The “Interval” of the synthesized design is an important metric. It specifies after how many cycles the filter can be started again. Thus it is a metric for how long the filter takes to perform its computation. What is the **maximum interval** of the synthesized design?  
\_\_\_\_\_  
\_\_\_\_\_
- Remember that each “execution” of the filter computes one entire video frame. Given the interval, how many frames per second can the filter sustain?  
\_\_\_\_\_  
\_\_\_\_\_
- What is the maximum interval at which the filter can sustain a 60 Hz (frames/s) video stream if the operating frequency is kept constant?  
\_\_\_\_\_  
\_\_\_\_\_

Clearly the filter is not fast enough. However since we have not provided any constraint to HLS as to what our



interval requirements are, the tool simply generates any reasonable solution.

#### Student Task 11:

- Have a look at possible directives by selecting code sections and inserting directives (right click on section label in the directives window).
- Set a “PIPELINE” directive on the `x_loop` within the `filter_hls` method (this can be done in the directives window.) This directive forces HLS to arrange the loop body into a pipeline that accepts new items at every cycle (an “initiation interval” of 1.)
- Rerun the high-level synthesis.
- What is the interval of the new filter? Can it sustain our video stream?

- 
- Compare the utilization estimates to your previous results. Which resource changed the most? Why?

- 
- Open up the `filter_hls.v` file in `solutions1/syn/verilog/` again.

- Is the `window` array mapped to registers or memory banks?

- Is the `lines` array mapped to registers or memory banks?

**Note:** The HLS tool may tell you that it cannot meet the target clock frequency. This is just a rough estimate and needs to be confirmed during synthesis (in the RTL-to-gates sense). Feel free to do this if you are interested by clicking the “Export RTL” button in the toolbar, checking “Vivado synthesis” under the “Evaluate Generated RTL” section and continuing the export for the “IP Catalog” format.

Besides adding a pipeline constraint various other adaptations are possible. If there is time at the end of the exercise you might want to try the following modifications:

- Add a latency constraint.
- Change the interface definition.
- Change the bit width of the data types or use float data types instead.

## 4.1 Functional Verification of High-Level Synthesized RTL Code

Before using the generated RTL code on the FPGA, its functional correctness needs to be verified. We use a file-based SystemVerilog testbench to verify the filter. First however we need to generate the input stimuli and expected responses using the provided Python script:

**Student Task 12:** Open a new shell. Change into the `simvectors/` directory and convert the stimuli:

```
sh> cd simvectors
sh> make stim
```

In case you get a Python error related to a missing PIL installation, try the following:

```
sh> make deps
```

Now we need to copy the RTL code generated by HLS into a separate location that is easier to access.

**Student Task 13:** Copy all files from `vivado_hls/filter/solution1/syn/verilog/` into the `sourcecode_hls/` directory.

```
sh> cd ../
sh> cp vivado_hls/filter/solution1/syn/verilog/* sourcecode_hls/
```

Now we are ready to run the testbench:

**Student Task 14:** Run the simulation as follows:

```
sh> cd modelsim
sh> make
```

The testbench setup should be familiar to you. After compilation the script pauses allowing you to check for compilation errors. Type 'exit' to close the compiler, after which the actual simulation will start. Note that the simulation will take some minutes to finish!

The simulation takes some time to complete. Once finished, convert the actual responses to an image (3x3 Discrete Laplace Operator):

```
sh> cd ../simvectors
sh> make resp
```

In case you get a Python error related to a missing PIL installation, try the following:

```
sh> make deps
```

Inspect the generated output. Does it match the output generated by the C++ testbench?

Show your resulting image to an assistant.

## 5 Integration of the High-Level Synthesized Stencil RTL Code into the Video Filter

**Before you start:** Make sure you have copied the RTL code generated by HLS into the `sourcecode_hls` directory as indicated above. If not do that now.

Initially, the Vivado video filter project has to be restored. A Tcl script is provided in this exercise that will automatically create a suitable video filter project.

**Student Task 15:** Source the environment for Vivado as before since you are in a new shell:

```
sh> start-dz-env
sh> source /home/vlsil/ex7/env.sh
```

Start Vivado in the intended directory:

```
sh> cd ~/ex7/vivado
sh> make
```

The design is similar to what you have been working with in previous exercises. If you have followed the previous parts of the exercise successfully you can now verify the functionality of the video filter by mapping the system onto the FPGA and viewing the output on the screen. Note that the design will only work correctly for an input/output resolution of 1920x1080 pixels!

#### Student Task 16:

- Synthesize and implement the design, and make sure that there are no errors or critical warnings. Additionally, check the timing of the implemented design. (As before there are a few endpoints that fail timing which are inside IP blocks, which you may ignore. If in doubt ask an assistant.)
- View the output on the screen. Show your result to an assistant.

#### Student Task 17: Additional ideas if you are up to a challenge at the end of the exercise:

- Pick a different filter from the provided list of kernels. E.g. a 5x5 one.
- Come up with your own kernel size, filter coefficients, and bias.
- Implement a Sobel filter by rewriting the contents of `pixel_kernel`.

## 6 Final Remarks and Further Reading

In this exercise, we have only scratched the surface of HLS design. There are many details and techniques we did not cover. For example, we did not make use of the C/RTL co-simulation capabilities of Vivado HLS, which can save designers a lot of time in testbench development. If you are curious, feel free to ask an assistant. You can also read up on the details in the following documents:

**Vivado HLS User Guide:** [UG902](#)

**Vivado HLS Tutorial:** [UG871](#)

**A handy list of pragmas:** [SDx 2017.2 HTML Documentation](#)

**You have reached the end of this exercise.**

ℰ

Please discuss your answers and any open questions with an assistant.  
Finally, please provide us with feedback by filling out the form on our  
website. Thank you!

ℰ