ETH zürich

Institut für Integrierte Systeme
Integrated Systems Laboratory

Department of Information Technology and Electrical Engineering

VLSI I:
From Architectures to VLSI Circuits and FPGAs

227-0116-00L

Exercise 8

FPGA Optimization

Prof. L. Benini
F. Gürkaynak
M. Korb

Last Changed: 2022-12-09 16:55:57 +0100

1 Introduction

In this exercise, you will learn the fundamentals of FPGA Optimization.

When developing for ASICs, the hardware designer has the freedom to add exactly the hardware needed for the target application – and nothing more. A key difference when developing for FPGAs is that the hardware is already pre-implemented there. FPGAs usually have optimized hardware for common operations, such as Digital Signal Processing (DSP) applications. The choice of making use of this hardware is up to the designer – even if it requires changing the architecture so that it maps to such optimized blocks of the FPGA.

In this exercise we will focus on the optimization of a simple convolutional image filter with a small 3×3 kernel. We will reuse the video processing infrastructure you are familiar with from previous exercises.

During this exercise we will guide you through the following tasks:

1. Doing pen-and-paper estimations of computational requirements of an algorithm.
2. Changing an architecture to make better use of available FPGA hardware.
3. Changing an algorithm to make better use of available FPGA hardware.

As always the assistants are at your disposal to help you with the task and discuss solutions.

2 Preparation

To get access to materials required for the exercise please follow the task below:

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

```
sh> /home/vlsi1/ex8/install.sh
sh> cd ex8
```

3 Image filtering

Basic image processing can be done by convolving a small kernel K with an image I:

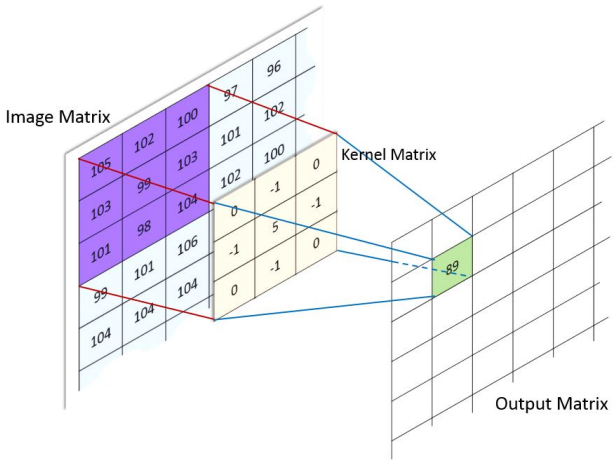


Figure 1: Convolution with a 3×3 kernel.

Convolution kernels are usually small in size. Consider the following convolution between a kernel K and a slice of the red channel (R) of an image I[1]. This produces one pixel of the red channel of the output image O.

$$O_{R,0,0} = \begin{bmatrix} K_{0,0} & K_{0,1} & K_{0,2} \\ K_{1,0} & K_{1,1} & K_{1,2} \\ K_{2,0} & K_{2,1} & K_{2,2} \end{bmatrix} * \begin{bmatrix} I_{R,0,0} & I_{R,0,1} & I_{R,0,2} \\ I_{R,1,0} & I_{R,1,1} & I_{R,1,2} \\ I_{R,2,0} & I_{R,2,1} & I_{R,2,2} \end{bmatrix} = K_{0,0}I_{R,0,0} + K_{0,1}I_{R,0,1} + \ldots + K_{2,2}I_{R,2,2} \quad (1)$$

Identical convolutions are done for the blue (B) and green (G) channels.

Student Task 2 (Computational requirements of a convolution): Consider the convolution of an image with a 3×3 kernel. How many multiplications and additions are required to evaluate one pixel of the output image?

***Hint**: You might want to expand the convolution equation written above. Don't forget we have three channels (RGB).*

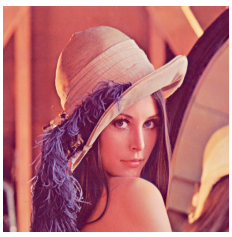
In computer vision literature, it is customary to compare results when applied to default images such as Lenna:

Figure 2: Lenna.

Several common image filters are implemented as simple convolutional kernels. In this exercise we consider four of them: an identity kernel, a Gaussian blur, a sharpen filter and a Sobel filter. Let us compare the result of these four kernels when applied to Lenna.

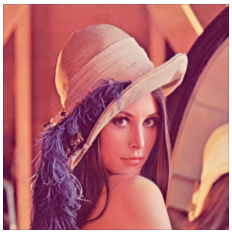
The identity kernel, as hinted by its name, produces an output image identical to the input. The Gaussian blur kernel can be used to blur an image by the application of (an approximation) of a Gaussian function. The sharpen kernel does, in a way, the opposite effect of a Gaussian blur filter. The sharpening filter increases the contrast between dark and white areas of an image. Finally, the Sobel filter is used to detect the edges of an image. Here we are considering the vertical Sobel filter, which tends to find vertical edges in the input image. There exists an equivalent horizontal Sobel filter and both of them are used in edge detection algorithms.

[1] You might note that this is actually a cross-correlation instead of a convolution. This is fine: most kernels are actually symmetrical.



$$K_{\text{Identity}} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

(a) Identity filter.



$$K_{\text{Gaussian blur}} = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

(b) Gaussian blur filter.

$$K_{\text{Sharpen}} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

(c) Sharpening filter.

$$K_{\text{Sobel (vertical)}} = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

(d) Vertical Sobel filter.

Figure 3: Kernels.

4 Fixed-point

As previously discussed in Exercise 7, to implement the convolution kernels in hardware, we need to find a representation of floating-point numbers with integers of a finite range thereby keeping a reasonable precision.

In this exercise we will represent the kernel coefficients as signed integers 12-bit wide, which corresponds to the integer range $[-2048, 2047]$. We are reserving four of those bits to the decimal part, which means that the kernel coefficients must be multiplied by $2^4 = 16$ (and rounded) to find their 12-bit integer representation.

Student Task 3 (Bit widths): Considering that the kernels coefficients are 12-bits wide and each pixel is represented with 8-bits, how many bits are necessary to represent *one* single product $k_i I_i$

Taking into consideration your previous answer, how many bits are necessary to represent the addition of the $3 \times 3 = 9$ multiplications?

Given that one RGB channel here is only 8-bit wide, what would you need to do to convert the result after

addtion to the corresponding RBG channel?

5 Getting familiar with the design

After warming-up, it is time to get hands-on with image filtering.

Student Task 4 (Setting up the project): Go to the `vivado` folder and execute the Makefile

```
sh> cd vivado
sh> make
```

After a few seconds, you will see the block design of the video processor – this is the same interface you used in Exercise 2, as presented in Figure 4.

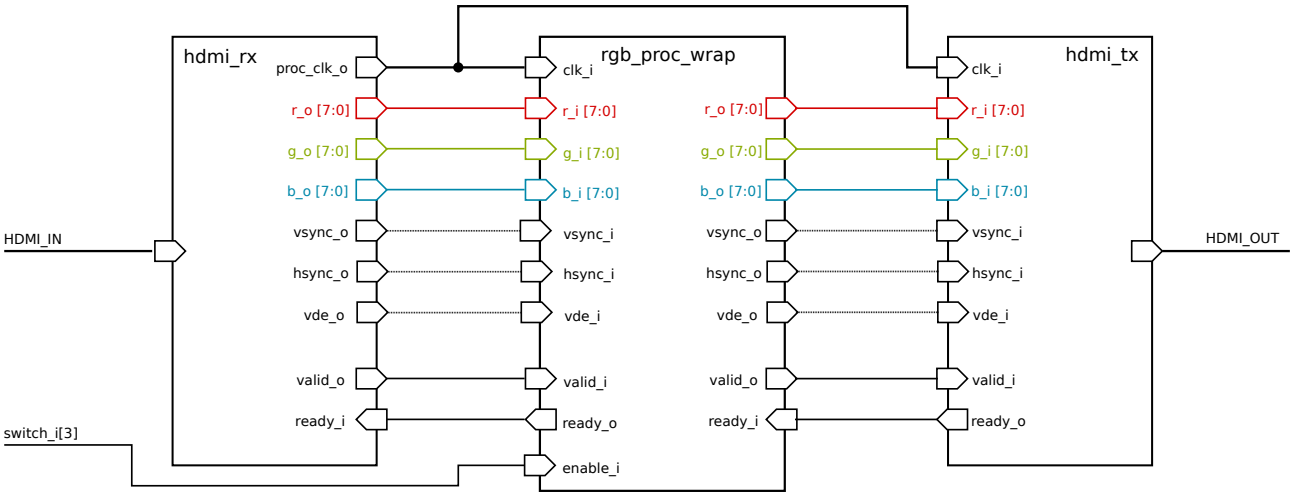


Figure 4: High Definition Multimedia Interface.

By clicking on the `Sources` tab in the middle column of your screen, you can navigate through the different source files. You can see that it is a hierarchical structure similar to the one in Figure 5. During this exercise, you will only need to care about the modules `rgb_filter` and `rgb_conv`.

5.1 Adding the kernels

We will now investigate the functionality of `rgb_filter`. Open the `rgb_filter.sv` file in Vivado's editor or your preferred text editor.

Besides control signals, this block takes as input three $3 \times 3 \times 8$-bit matrices `r_i`, `g_i` and `b_i` (corresponding to a 3×3 window of the input image in the three color channels) and should produce a pixel through the three 8-bit output signals `r_o`, `g_o` and `b_o`.

Student Task 5 (Reverse engineering): The first lines of the `rgb_filter.sv` define a $3 \times 3 \times 12$ signed signal `k`, which holds the convolution weights. Understand how this variable is assigned a value.

We forgot to add some of the convolution kernels that will be used later in this exercise. Complete the assignment of `k` with the remaining kernels.

***Hint**: Don't forget that the kernels are 12-bits wide and have four bits of fractional part.*

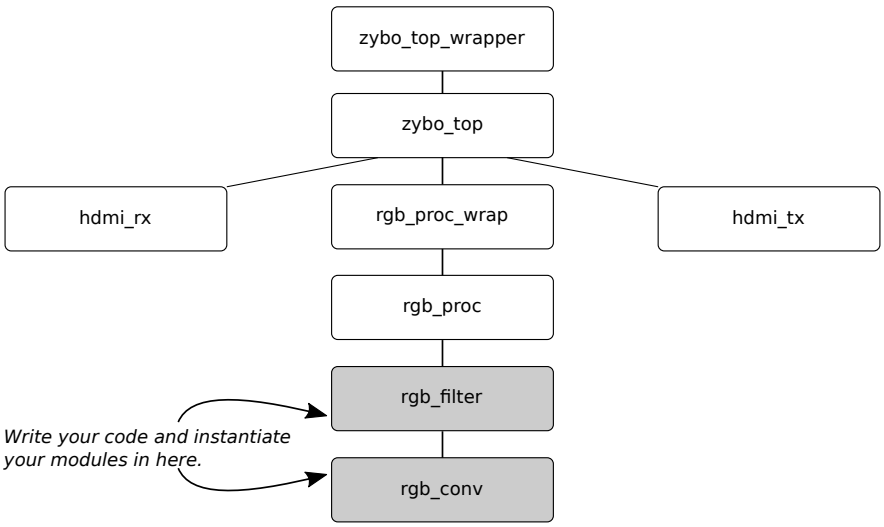


Figure 5: Overview of the hierarchy of blocks used for video processing in these exercises.

***Hint**: The matrix k is indexed as $k[x][y]$.*

5.2 Basic convolution

Now that we have our convolution kernels correctly assigned, it is time to dive into the convolution itself. Open the file `rgb_conv.sv`. There are four sections in this file, "`PARAMETERS`", "`MULTIPLICATIONS`", "`ACCUMULATION`" and "`OUTPUT`".

In the "`PARAMETERS`" section, we define the width of the convolution partial multiplications and of the final accumulator result.

Student Task 6 (Bit-widths): In the file `rgb_conv.sv`, do the widths defined in section "`PARAMETERS`" agree with the minimum widths you calculated before? Would this be a problem?

Fix the widths.

You can now test the behaviour of the convolution.

Student Task 7 (Functional simulation): Run a functional simulation in Vivado with the testbench we gave you. This testbench compares the expected output of the convolution kernel with the four kernels discussed beforehand.

The section "`MULTIPLICATIONS`" declares three $3 \times 3 \times$`MULTIPLICATION_WIDTH` signed matrices, that hold the partial multiplication results of the convolution. These matrices are filled up in the `always_comb` that comes just after.

The partial multiplications are summed together in the `always_comb` of the section "`ACCUMULATION`". The accumulation over the three channels is stored in three `ACCUMULATOR_WIDTH`-bit wide vectors. This accumulation result, with suffix `_d`, is registered and stored in flip-flops (with suffix `_q`), to improve the overall design timing.

Finally, the section "`OUTPUT`" takes the registered accumulator result and outputs only its integer part. This is done by dividing the accumulator value by $2^4 = 16$, which corresponds to a shift-right operation.

Student Task 8 (DDG): Based on the convolution equation we analyzed before and on your understanding of the file `rgb_conv.sv`, draw the Data-Dependancy Graph (DDG) for the convolution on one

channel.

***Hint**: You might want to check slide 47 in https://iis-students.ee.ethz.ch/assets/VLSI1/lectures/2022/lec05-arch1.pdf.*

The DDG shows that the critical path in our design is quite long. In fact, optimizing this long path (which is the core of the convolution) will be the main focus of this exercise.

Student Task 9 (FPGA Implementation): Synthesize your code. This should take approximately 2 minutes. During this time you can already connect your FPGA board to the PC with the USB cable.

Start the implementation, which should again take no longer than 2 minutes.

Do you meet the setup timing constraints?

In the Project Summary page, open the `Implemented Timing Report` and inspect the worst failing path. How many logic levels does the longest path have? which components are contained in the longest?

---

6 Optimizing the convolution

We are not able to meet the setup timing constraints of the convolution. Lowering the clock period would require having a lower resolution on our output HDMI port – something we cannot accept. We will try to meet these constraints with architectural optimizations alone!

6.1 Accumulation

Student Task 10 (Transforming the accumulation.): Go back to your DDG. The longest path should pass through a long chain of eight adders. In your VLSI-I class you should have seen some transformations for combinational computations that should improve the timing of this adder chain, making use of the associative property of the addition.

Propose a transformation for the adder chain.

***Hint**: You might want to check https://iis-students.ee.ethz.ch/assets/VLSI1/lectures/2022/lec07-arch2.pdf.*

Implementing this transformation in SystemVerilog is actually simple. You just need to add parenthesis as required to group the operations, so that the adder chain corresponds to what you drew in the previous student task.

Student Task 11: Patch the "`ACCUMULATION`" section of the `rgb_conv.sv` file, so that it corresponds to the transformation you proposed. Synthesize and implement your modified design. By how much your did worst negative slack decrease?

***Hint**: You might get a marginal timing improvement. This is because in our original design (without parenthesis grouping), the long chain of 8 adders is optimized automatically by Vivado synthesizer, in order to meet the time constraints.*

6.2 Multiplication

Multiplications – especially wide ones – are expensive operations. It makes sense for FPGA developers to provide special blocks to optimize this kind of common operations. Those are the DSP Slices, as seen in Figure 6.

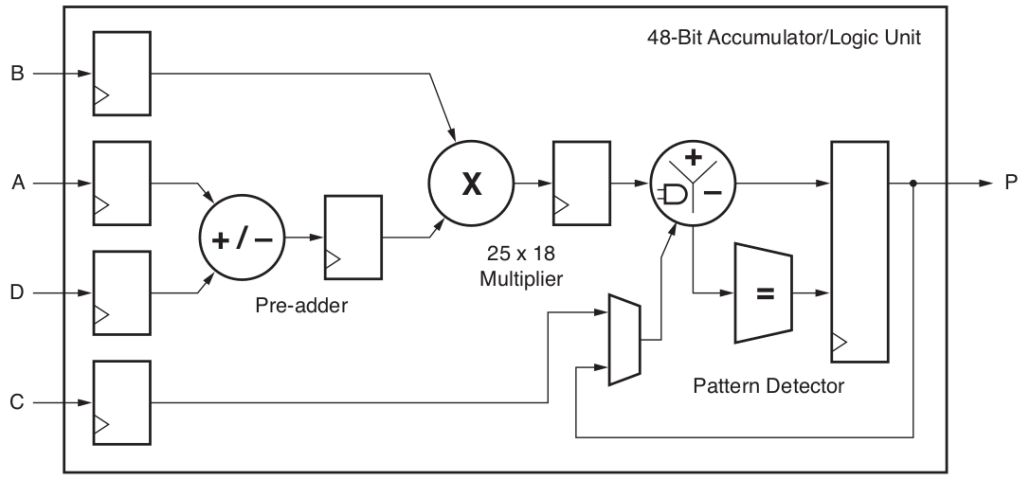


Figure 6: DSP Slice.

Student Task 12 (DSP Slices): In the project summary page, check the post-implementation utilization

report. How many DSP slices are being used? Why?

How many flip-flops are being used?

It turns out that doing a wide multiplication and accumulating the partial results is too much to be done in only one cycle. Currently, we are making use of the DSP Slice's multipliers. Its output is directly sent to outside logic that accumulates the partial multiplication into the final result.

To solve this issue, we will pipeline the multiplication result. The final DDG of our convolution has the form of Figure 7.

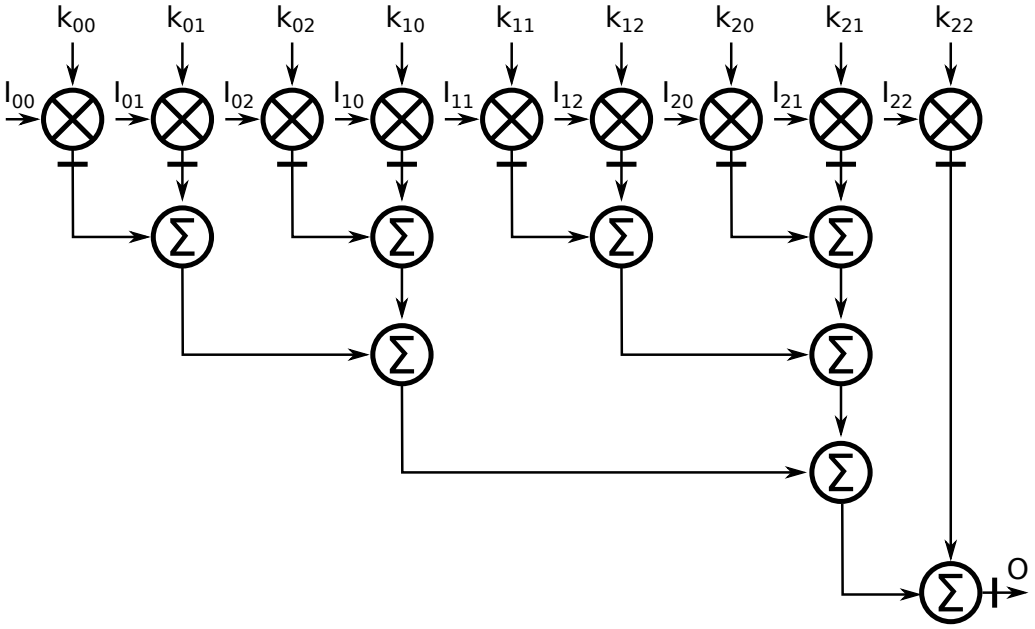


Figure 7: Example DDG after pipelining the multiplier result. Slightly different topologies with the same longest path are possible, too.

Please note that the DSP Slices already have a register at the output of the multiplier, which is not being used (it is being *bypassed*).

Student Task 13 (Registering the multiplier output.): Which effects do you expect from registering the multiplier output in terms of latency, timing and area?

Patch the "`MULTIPLICATIONS`" and "`ACCUMULATION`" sections of `rgb_conv` so that the multiplier output is registered.

Run the testbench and check the result with an assistant.

6.3 Implementation

By now we implemented two optimizations to the convolution kernel: we replaced the adder chain by a tree-based accumulator and we registered the multiplier output. We can now test that the filters are working.

Student Task 14 (Generating bitstream and programming the FPGA): Synthesize and implement your code as before. Do you meet the setup timing constraints?***Hint****: You may get timing violations on the* Worst Pulse Width Slack*. They are unrelated to your design and may be safely ignored in this project. Any other timing violation may* not *be ignored, though!*

Check the device usage in terms of flip-flops and DSP Slices. Compare to what you had before. Discuss the result with an assistant.

Generate the bitstream and program your device.

Now the FPGA is ready to be connected between the PC and the display.

Student Task 15 (Placing the FPGA between PC and display):

- Unplug the DVI cable between the PC and the display at the display side.
- Plug the first HDMI-to-DVI cable to the second display port of your PC and to the HDMI RX port of the FPGA.
- Plug the second HDMI-to-DVI cable to the HDMI TX port of the FPGA and to the DVI port of the Display.

We can finally test that the convolution is working.

Student Task 16 (Testing the filters.): Test the convolution behaviour with the switches `switch[2:0]`. Test the result with Lenna (provided in the `simvectors` directory) – the result should agree with the example images of Figure 3.

7 Final adjustments

Student Task 17 (Further optimizations): Go back to the DSP Slice of Figure 6 and to the kernels we have been using. Can you spot another optimization that can allow us to use a bit fewer multipliers?

Hint*: all four kernels are vertically symmetrical.*

Student Task 18 (Optional): Implement your optimization and compare it with your current design. What changed?

Discuss your results and any unclear part(s) in the exercise with an assistant.

You have reached the end of this exercise.
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!