

Department of Information Technology and Electrical Engineering

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

227-0116-00L

Exercise 2

HDL Coding: Combinational Circuits

Prof. L. Benini
F. Gürkaynak
M. Korb

Last Changed: 2023-10-01 15:54:05 +0200

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 Introduction

In this exercise, you will use the basic concepts of the SystemVerilog hardware description language (HDL) for *purely combinatorial* circuits. In particular, you will learn:

- how to describe logic functions such as AND and OR,
- how to handle data types of SystemVerilog,
- how to implement combinatorial blocks such as a multiplexer,
- how to create hierarchical architectures with the aid of modules, and
- how to implement simple arithmetic functions as multiplication and addition.

The exercise is designed to be solvable within *two afternoons*. Together we will implement a real-time HDMI video processor on your Zybo Z7 development board. We assume you already know the FPGA design and implementation flow, but it might be useful to consult the Exercise 1 sheet. During the exercise you will also get in touch with the Vivado simulator, which can be used for functional verification. You will use it with a given testbench (how to develop your own testbench will be the topic of Exercise 4, and we do not expect you to fully know the simulator after this exercise).

As always we encourage you to discuss any questions that arise during the exercise with the assistants.

2 Preparation

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

```
sh> /home/vlsi1/ex2/install.sh
sh> cd ex2
```

3 Video processing

Before you start with the implementation of your video processor we provide you with some background knowledge about video processing. In particular, we will briefly go through the basics of the red-green-blue (RGB) color model and the High-Definition Multimedia Interface (HDMI). First of all, it is important to note that a video is just a sequence of images (called *frames* in the context of video processing) and that an image is just an array of pixels.

3.1 RGB

A pixel is the smallest addressable point in a raster image and, for the RGB model, consists of three components (also known as *channels*) which can change in intensity. The more pixels you have, the higher the resolution of your image is. RGB is a simple and common additive color model: *additive* means that the channels (red, green, and blue in this case) get added to depict the color of a pixel.

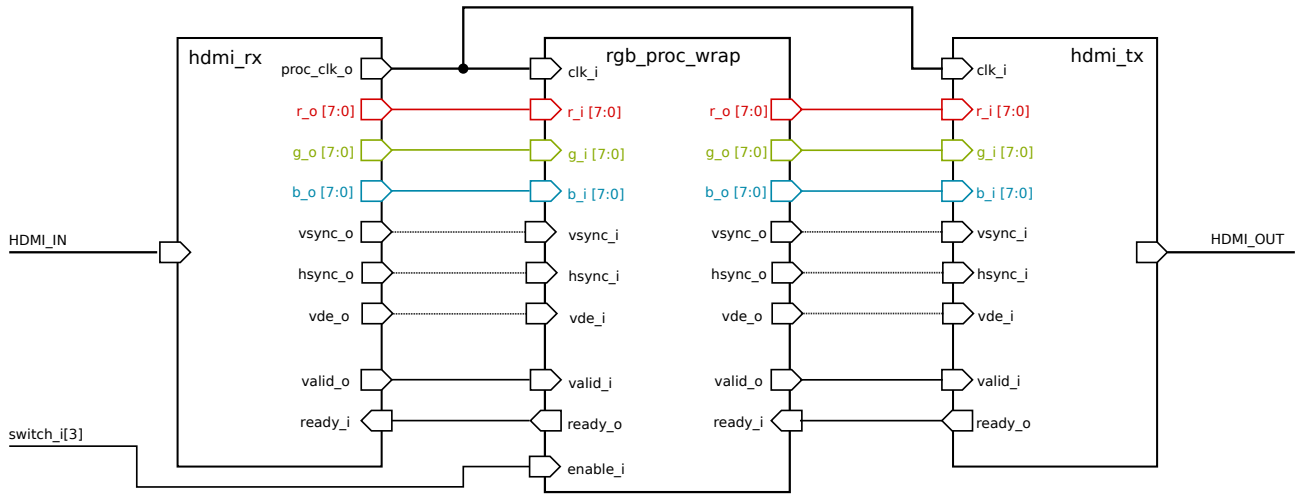


Figure 1: High Definition Multimedia Interface

3.1.1 Color depth

Like the pixel density, the color depth is a measure of the quality of an image. It defines how many different colors can be represented by a pixel. As already mentioned, each of the three components of a pixel has its own intensity; in the exercise, we will directly play with them: each RGB channel has 8 bits hence 256 different intensities are available for each of the three colors, ranging from 0 (no portion of this color) to 255 (maximum intensity of this color). Through the combination of the three colors we are able to depict $2^8 \cdot 2^8 \cdot 2^8 = 16\,777\,216$ different colors. Thus a black pixel is

$$\text{rgb}_{\text{black}} = (0 \ 0 \ 0), \quad (1)$$

and a white pixel is

$$\text{rgb}_{\text{white}} = (255 \ 255 \ 255). \quad (2)$$

The number of bits is not fixed but depends on the technology. If even more bits would be available we would have more colors, resulting in an higher color depth.

3.2 HDMI

The *High Definition Multimedia Interface* was introduced in 2002 for the digital transmission of audio and video signals and has been further developed since then. With a simple passive adapter it is backwards-compatible with DVI (Digital Video Interface). Although the interface offers many modes and communication channels, for our purposes the functioning is quite simple to understand.

In Figure 1, you can see all the signals that you have to consider for your implementation. The most important ones are the RGB wires, which define the color of the pixel as described before. The three signals `hsync`, `vsync`, and `vde` are used by the receiver (e.g., the display) to synchronize the horizontal and vertical position of a pixel. In this exercise we will always just feed them through our processor and will not work with them, although it is very important to make sure that these signals are kept aligned in time with the RGB signals. The `ready` and `valid` signals are used for handshake communication between the modules: the receiver of a data packet signals the transmitter that it is ready to receive a new packet through the `ready` signal, and the sender signals the receiver through the `valid` signal that new data is available.

4 SystemVerilog

With the knowledge of lecture 3 you should be able to write all source code in this exercise. A good overview over the most important concepts can be found on the Wiki page of the IIS ¹. For a better understanding, please also refer to Chapter 4.3 of the textbook *Top-Down Digital VLSI Design*, by Prof. Kaeslin.

4.1 Coding Guidelines

Using a common naming convention and coding style offers the possibility to collaborate on IC projects around the world. We highly recommend you to adhere to the guidelines of any project you are working on. You can find the guidelines used at our institute at <https://github.com/lowRISC/style-guides/blob/master/VerilogCodingStyle.md>.

4.2 Editor Support for SystemVerilog

In general we do not really care which text editor you use to solve the exercises but you can make your life a lot easier by using an editor that actually supports the syntax of SystemVerilog. While you might be used to modern Integrated Development Environments (IDEs) the sad truth is that there currently is no IDE (at least to the authors knowledge) that provides adequate support for SystemVerilog development. Also, since HDL developers are dealing with a plethora of different scripting languages and text files in general, a powerful general-purpose text editor is definitely the weapon of choice. Although the more powerful text editors like vim, Emacs etc. are definitely not as beginner friendly as Gedit or Notepad++ they can, with some practice, provide a tremendous efficiency boost to all forms of text manipulation. Gaining some proficiency in at least one keyboard-driven advanced text editor is time well spent for every engineer in general.

4.2.1 Emacs

At IIS we maintain a heavily customized, “batteries-included” configuration set for the free and open-source Texteditor Emacs. It comes with some additional custom developed packages for automatic syntax checking, a modern GUI and some more beginner friendly default settings so you can just start using it instead of climbing the steep initial learning curve of vanilla Emacs. You can install the customizations by executing the following command in your terminal:

```
sh> /home/vlsi1/emacs-systemverilog/install.sh
```

The installation can take a couple of minutes. Towards the end of the installation process, Emacs will start itself in the terminal to run some initialization scripts. **Wait until emacs terminates itself!** Ignore possible warnings such as *Error while dumping Spacemacs* and just wait for Emacs to close on its own. The installation is done only when you see the message **All done** in your terminal. After the end of the installation procedure, make sure to **open a new terminal**; here you can run:

```
sh> emacs <some-file-to-open>
```

If everything went fine, you should be greeted by a quickstart guide on how to use the customized Emacs version. Should you find yourself lost in space(emacs), remember you always have the quickstart guide and the cheatsheet under *IIS Students Menu* in the top-left corner.

Note 1: The customization set for students and the installation flow are very new. Any kind of feedback on bugs during installation and usability would be well appreciated.

¹ http://eda.ee.ethz.ch/index.php/SystemVerilog_Examples

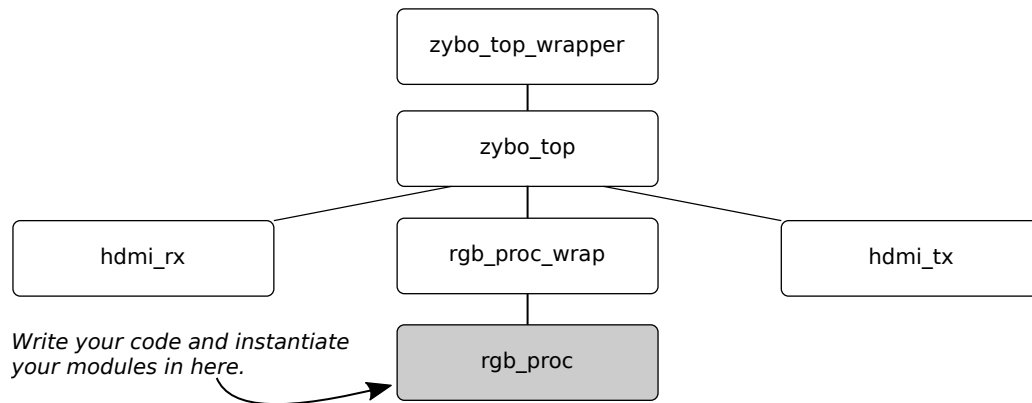


Figure 2: Overview of the hierarchy of blocks used for video processing in these exercises.

4.2.2 Sublime Text

In case you already use SublimeText, a commercial general-purpose text editor, we recommend using the SublimeText SystemVerilog plugin. At home you can install it as a regular package via PackageControl. Here in the lab you need to manually clone it:

```
sh> cd ~/.config/sublime-text-3/Packages
sh> git clone https://github.com/TheClams/SystemVerilog
sh> cd ~/ex2/
```

Afterwards you can invoke SublimeText by typing the following command in your terminal:

```
sh> sublime_text
```

Student Task 2 (Editor Setup):

- Choose one of the above editors, install the customizations for SystemVerilog and make yourself familiar with the user interface.
- Open one of the SystemVerilog files in `part1/sourcecode` and verify that your editor highlights the syntax of the HDL sourcecode.

5 Getting familiar with the design

In the last exercise, you learned how to set up a new project in Vivado. We mentioned that it is good practice to always create a script to speed up the initialization process of a new project. This time we already prepared the necessary script files for you. The scripts will create a new project called `ex2-part1` and will import all design sources that have already been created. They also establish the connectivity between the modules to create a fully functioning circuit.

Student Task 3 (Setting up the project): Go to the `part1/vivado` folder and execute the Makefile

```
sh> cd part1/vivado
sh> make
```

After a few seconds, you will see the block design of the video processor with the three top modules as presented in Figure 1. If you click 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 2.

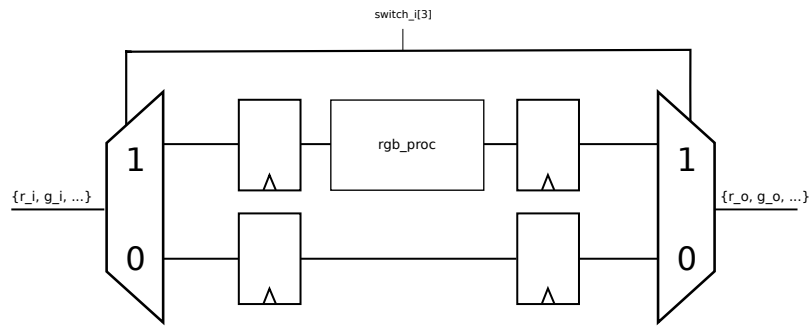


Figure 3: Schematic of the buffer and bypass functionality in `rgb_proc_wrap`.

Throughout this exercise you will not have to care about the modules `hdmi_rx`, `hdmi_tx` and `zybo_top`. They are used to communicate with the physical HDMI interfaces and to expose the HDMI signal in an easy modifiable format.

It might seem strange that so many files are involved when the architecture consists only of the three modules presented in Figure 1, but most of them are used inside `rgb_proc_wrap`. The `rgb_proc_wrap` module is a *wrapper* around the actual RGB processing module, `rgb_proc`, which you will write. A wrapper can be loosely defined as a module that makes the module inside it compatible with the circuit surrounding it, but that adds no functionality. `rgb_proc_wrap` fulfills three tasks:

1. It provides two independent signal paths, one through your `rgb_proc` and one that bypasses it. As shown in Figure 3, the bypass is controlled by the `enable_i` input, which is connected to switch 3 (the leftmost one on the board). In other words, you have to enable switch 3 to test your `rgb_proc`; if your module makes the output unreadable, you can always bypass it by disabling switch 3. This way, you do not have to unplug the FPGA from the monitor for (most) faults that corrupt the output.
2. It buffers all signals coming from `hdmi_rx` and going into `hdmi_tx`, so that your `rgb_proc` is sequentially decoupled and you have a full clock cycle for your longest path (you will learn more about timing and longest paths in Exercise 3).
3. It is a Verilog module that instantiates the `rgb_proc` SystemVerilog module internally, because Vivado cannot directly instantiate SystemVerilog modules in a block diagram.

We will now investigate the functionality of `rgb_proc`.

Student Task 4 (Reverse engineering):

- Open the sourcecode of `rgb_proc` and `rgb_grayscale_primitive` in your editor. Alternatively you can also inspect the sourcecode directly in Vivado by clicking the `Sources` tab and clicking yourself through the design hierarchy.
- Inspect their source code and draw a schematic of the combined functionality of the two modules (confine yourself to the most important signals).

- How would you describe the functionality?

To check whether you described the right functionality, you will now program the FPGA. The following tasks should guide you through this process. The synthesis and implementation strategies have already been set by the scripts.

Student Task 5 (Generating bitstream and programming the FPGA):

- 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 and power it on. Make sure to connect to the correct USB port of the board and that the two jumpers are positioned as stated in last week's exercise.
- Start the implementation, which should again take no longer than 2 minutes.
- Generate the bitstream and program your device.

Now the FPGA is ready; in the next task you are going to tap into the HDMI connection between your PC and the display. The aim is to intercept the HDMI signals and force them through `rgb_proc`, before reaching the display.

Student Task 6 (Placing the FPGA between PC and display):

- Unplug the DVI cable between the PC and the display at both sides.
- Plug the HDMI-to-HDMI (or HDMI-to-DVI, depending on what you got) cable to the display interface of your PC and to the HDMI RX port of the FPGA.
- Plug the HDMI-to-DVI cable to the HDMI TX port of the FPGA and to the DVI port of the display.

Hint: Together with the FPGA you should have received either two HDMI-DVI or one HDMI-HDMI and HDMI-DVI cable. If you accidentally got two HDMI-HDMI cables please ask a TA to get your cable replaced.

You should now see the same image as before except for a little change of the resolution. The Zybo board processes data at a slightly higher resolution than the monitors, which the monitors cannot display perfectly. Due to the resolution change, Vivado present an “Internal Exception”, which you can safely ignore and hit `Continue`. The monitor may also presents a warning message, which you can discard by pressing any button on the monitor. If anything else does not work as expected, please ask an assistant for help.

Hint: In case you need to kill Vivado you can use the handy Linux command `killall Vivado`

Hint: You can stick to this configuration for the whole duration of the exercise, so that you do not have to plug and unplug all the cables. The monitor will only be unusable during FPGA programming. If you have problems with the output, you can toggle switch 3 of your board and forward the unmodified HDMI signal to the monitor.

5.1 Explore the functionality

Student Task 7 (Exploring the functionality): Switch `SW0` and `SW3`. When is the grayscale active?

5.2 Refine the functionality

Our solution for a gray image is quite primitive and we are sure you can do this better. Experiments showed that humans prefer a gray image which consists of all three RGB channels but with different weights. The choice of these weights slightly changes the luminance of the image. We propose you to do it with the values in Table 1.

Channel	Weight	transformed weight
red	0.30	
green	0.59	
blue	0.11	

Table 1: Weights used for converting the color channels to one gray channel.

Each of the three channels gets multiplied with the corresponding weight. The three weighted colors are then added and the result is forwarded to every output channel. Therefore, we need three multiplications and two additions to implement our desired functionality.

As an IC designer you seldom work with floating-point numbers; you rather mainly employ integers. Each integer number is composed of many bits, each representing one binary digit. The task is therefore to find a representation of decimal numbers using integers of a finite range, while keeping a reasonable precision. We can steer this precision by choosing an appropriate interval of integers. In this exercise we will work with 8-bit precision, hence mapping to the integer range $[0, 255]$.

Our initial weights from table 1 are real values in the interval $[0, 1]$. We can map them to the integer range $[0, 255]$ with a multiplication by 255 and an arithmetic rounding. This will be performed off-line: think about it as a characterization of your system. With this step, both the weights and the channels are integer values in $[0, 255]$. Consequently, the channels can be multiplied by the weights; the result (which will be in the interval $[0, 65535]$) needs a normalization (i.e. division by 256) to be mapped back to $[0, 255]$. Such a reshaping by a power of 2 is simple to implement in hardware as it corresponds to a fixed bit-shift to the right. To put this into a formula:

$$r_o = g_o = b_o = \sum_{c=r,g,b} c_i \cdot \text{weight}_c \approx \sum_{c=r,g,b} c_i \cdot \left\lfloor \frac{\text{weight}_c \cdot 255}{256} \right\rfloor \quad (3)$$

As mentioned, the sum of the intermediate result of the multiplications should be in $[0, 65535]$ (i.e. on a 16-bit range). To understand why we need at most 16 bits for the intermediate sum, consider the following example: assuming the maximum input value, a white pixel

$$\text{rgb}_i = (r_i \quad g_i \quad b_i) = (255 \quad 255 \quad 255), \quad (4)$$

and with weights as described before,

$$\text{weight} = \begin{pmatrix} \text{weight}_r \\ \text{weight}_g \\ \text{weight}_b \end{pmatrix} \approx 255 \cdot \begin{pmatrix} 0.30 \\ 0.59 \\ 0.11 \end{pmatrix}, \quad (5)$$

the weighted sum can be written as

$$\text{sum} = \text{rgb}_i \cdot \text{weight} = 255 \cdot 255 \cdot \underbrace{(0.30 + 0.59 + 0.11)}_{=1} = 65\,025. \quad (6)$$

In conclusion, 65 025 is the highest sum you can obtain, hence it can be stored as a 16-bit unsigned integer.

In some cases you need to be careful on how the SystemVerilog standard² determines the bit lengths of expressions (see Section 11.6 of the standard). SystemVerilog uses the bit length of the operands to determine how many bits to use while evaluating an expression. The bit length rules are given in 11.6.1. In the case of the addition and multiplication operator, the bit length of the largest operand, including the left-hand side of an assignment, shall be used.

Student Task 8 (Transforming the weights): Transform the weights to the interval $[0, 255]$ by using arithmetic rounding and insert the values in Table 1. What should be the sum of the integers weights?

To define local constants in SystemVerilog use the `localparam` qualifier. It can be combined with the data type `logic` in the following way:

```
localparam logic [7:0] MYCONSTANT = 8'd0;
```

This code snippet defines an 8-bit wide constant `MYCONSTANT` (according to the naming convention we capitalize constants) and initializes it with the decimal number 0.

To write your SystemVerilog code, you can use any text editor you want. For this exercise, you can start exploring one of the options described in section 4.



² <https://ieeexplore.ieee.org/document/8299595>

Student Task 9 (Writing the source code):

- Open the source code of the file `rgb_grayscale_refined.sv`.
- The first step for a new module is always to write its port list. This means you have to declare what type of inputs and outputs you expect and how you will name them inside your module. This time we already did it for you.
- Define constants called `RWEIGHT`, `GWEIGHT`, `BWEIGHT` and initialize them with the weights mapped to `[0, 255]`.
- Define four intermediate 16-bit wide signals `int_r`, `int_g`, `int_b`, `int_sum` and an 8-bit wide signal `sum`.
- Assign the intermediate signals with the products of the corresponding multiplications.
- Calculate the sum of the three intermediate signals (`int_sum`) and assign the bit-shifted version of it (`sum`) to every output. **Hint:** Consult your lecture notes to see how to implement a bit shift.
- Do you know another way to assign a part of a signal to another?

Now it is time to test your design. Since you rely a lot on a correct working display, it is a good idea to verify the correctness of your circuit before reprogramming your FPGA. We will do this using Vivado's simulator with a testbench we wrote for you.

Student Task 10 (Verifying the functionality of your module):

- In the `Sources` window of Vivado's Project Manager, under *Simulation Sources*, right-click on the `rgb_grayscale_tb` module and click *Set as Top*. If that is already the case, the menu entry is grayed out and your module is marked by a  icon.
- Run the simulation by clicking `Run Simulation` → `Run Behavioral Simulation` in the left-hand menu.
- Under the tab 'Untitled 1', you should now see the signals of your circuit. You should make sure that the output signals match with the expected ones for all applied inputs. The testbench actually helps you with this: after execution has completed, you can find a summary of passing and failing tests in the Tcl console at the bottom of the Vivado window, which you can read by scrolling up. If the test fails, you will find additional information on where the actual output of your circuit did not match the expected output.
- Debug your code **in case anything does not work** as expected. To test your fixes, relaunch the simulation with . If you are curious, you can insert deliberate errors in your code to see how the simulation behaves.
- Close the simulation once the test passes.

After having ensured that your new module is functionally correct, we will insert it into our design. To allow direct comparison between the primitive and the refined grayscale, we will let them both operate on the input and multiplex the output with `switch_i[0]`. This will allow you to compare the two filters in real-time by switching `SW0` on the board.

Student Task 11 (Instantiating the new grayscale):

- Open the source code file `rgb_proc.sv`.
- Declare internal signals for the three color outputs of the refined grayscale similarly to how they are declared for the primitive grayscale.
- Instantiate the refined grayscale after the primitive grayscale. **Hint:** Both port lists are identical, but the module and instance names differ.
- Adapt the code for the multiplexer such that it forwards the output of the refined grayscale when `switch_i[0] == 1'b0`.

Student Task 12 (Bringing your design to the FPGA):

- If you are unsure whether you correctly executed the last steps you can ask an assistant to look through your code.
- Synthesize your code, implement it, generate the bitstream, and program your device. **Hint:** You may get timing violations on the Worst Pulse Width Slack^a. They are unrelated to your design and may be safely ignored in this project. Any other timing violation may not be ignored, though!
- Check whether the circuit meets your expectations (e.g. have a look at the highlighted line in your Vivado code editor).

^a Worst Pulse Width Slack (WPWS): Corresponds to the worst slack of the min low/high pulse width, min/max period and max skew checks when using both min and max delays; see https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug906-vivado-design-analysis.pdf.



You have reached the end of the first afternoon.
Please discuss your answers and any open questions with an assistant.



6 Implementing an advanced design

Last week we have implemented a circuit that uses a switch to toggle between two modes. By employing a second switch, we can expand the functionalities of our video processor selecting among four different configurations.

In the folder `ex2/part2` you will find all the necessary files for this week's exercise.

Student Task 13 (Setup):

```
sh> cd ex2/part2/vivado
sh> make
```

The first functionality we would like to implement is a *quantizer*. The idea is to use only a subset of the provided input bits. In your design you should use only the first two most significant bits of each RGB channel.

Student Task 14 (First considerations):

- How many colors can be represented with only two bits per channel?

- Considering the red channel: Will you still be able to differentiate between the initial intensity levels 0 and, for example, 20 if you use only the most significant two bits of the channel?

With these considerations you should understand our choice of the name *quantizer*. Now we want that you understand how to invert colors.

- Assume 8-bit channels. What is the opposite of

$$\text{rgb}_{\text{white}} = (255 \ 255 \ 255) ?$$

- In general, how can you invert an 8-bit color?

To switch between the four functionalities, we will use the following encoding:

switch_i[1]	switch_i[0]	functionality
0	0	feedthrough
0	1	quantizer
1	0	inverter
1	1	grayscale_refined



Table 2: Encoding scheme for the different functionalities

Student Task 15 (Adapting the top module):

- Open the source code of `rgb_proc.sv`.
- We already provided you with all necessary signals and the grayscale module.
- Using the encoding in Table 2, write the multiplexer by changing the `if .. else ..` clause to a `case` statement.
- Write the quantizer functionality. You do not have to write a new module but you can write it directly in the source code of `rgb_proc`. *Hint: Although we only use 2 bits per channel, we still have to define a value for all the 8 bits of each color, due to how we defined the interface of our modules. Use the concatenation operator (`{ .., .., .. }`) or the bit-wise AND operator (`&`).*
- Write the inverter functionality. Again, you can write your source code directly in `rgb_proc`.
- Finally, for the refined grayscale module, use the one with the same configuration as last time.

To test if everything works as desired we provided you with a new testbench.

Student Task 16 (Verifying the functionality of your module):

- In the `Sources` window of Vivado's Project Manager, under *Simulation Sources*, right-click on the `rgb_proc_tb` module and click *Set as Top*. If that is already the case, the menu entry is grayed out and your module is marked by a  icon.
- Run the simulation by clicking `Run Simulation` → `Run Behavioral Simulation` in the left-hand menu.
- Under the tab 'Untitled 1', you should now see the signals of your circuit. You should make sure that the output signals match with the expected ones for all applied inputs. The testbench actually helps you with this: after execution has completed, you can find a summary of passing and failing tests in the Tcl console at the bottom of the Vivado window, which you can read by scrolling up. If the test fails, you will find additional information on where the actual output of your circuit did not match the expected output.
- Debug your code **in case anything does not work** as expected. To test your fixes, relaunch the simulation with . If you are curious, you can insert deliberate errors in your code to see how the simulation behaves.
- Close the simulation once the test passes.

Student Task 17 (Programming the device): Do all the necessary steps to program your device and make sure its functionality matches your expectations.

We still have an unused switch, switch 2, which we now also want to employ in our design. We propose you to introduce an option to modify the brightness of the video signal, but feel free to develop your own idea. At this point it should be obvious that making an image brighter means increasing its RGB-values. To have no timing violations we will not scale the channels with a multiplication but with a simple addition. We will place the module after our multiplexer and use it only if `switch_i[2] == 1'b1`.

Student Task 18 (Writing a brightener):

- Choose `File` → `New File`, create a new file called `rgb_brightener.sv` and save it in the folder `part2/sourcecode`.
- In the source file, write a module called `rgb_brightener` with 3 RGB channels as input and 3 RGB channels as output.
- Inside an `always_comb` block, write for each channel an if-else statement that adds 55 to the channel if it is smaller than 201 and forces it to 255 if it is above this threshold. **Hint:** Make sure your literals are properly typed. Also, do not forget to end the `always_comb` block and the module.
- Save the file and add it to the project scope: right-click in the `Sources` window, click `Add Sources\...` and follow the wizard.

Now we have to adapt our top module to use the brightener. Figure 4 shows an overview of the final architecture of `rgb_proc`.



Student Task 19 (Adapting the top module):

- Open `rgb_proc.sv`.
- Introduce six new internal 8-bit wide signals to connect your new module. **Hint:** Choose meaningful names. The naming should make it easy to distinguish between inputs and outputs of the new module. The question marks in Figure 4 show where you will use them.
- Instantiate your brightener and connect the new internal signals to its ports.
- Change the signals in the case statement. **Hint:** See also Figure 4 to better understand how the signals should be used.
- Distinguish whether you want to use the brightened image through the condition `switch_i[2] == 1'b1`.

Hint: There is a construct in SystemVerilog that lets you write the assignment and the condition on a single line for each channel. Please consult your lecture notes or the IIS wiki to see how to use this construct.

It is again time to test the functional correctness of your circuit.

Student Task 20 (Verifying the functionality of your module):

- In the `Sources` window of Vivado's Project Manager, under *Simulation Sources*, right-click on the `rgb_proc_extended_tb` module and click *Set as Top*. If that is already the case, the menu entry is grayed out and your module is marked by a  icon.
- Run the simulation by clicking `Run Simulation` → `Run Behavioral Simulation` in the left-hand menu.
- Under the tab 'Untitled 1', you should now see the signals of your circuit. You should make sure that the output signals match with the expected ones for all applied inputs. The testbench actually helps you with this: after execution has completed, you can find a summary of passing and failing tests in the Tcl console at the bottom of the Vivado window, which you can read by scrolling up. If the test fails, you will find additional information on where the actual output of your circuit did not match the expected output.
- Debug your code **in case anything does not work** as expected. To test your fixes, relaunch the simulation with . If you are curious, you can insert deliberate errors in your code to see how the simulation behaves.
- Close the simulation once the test passes.

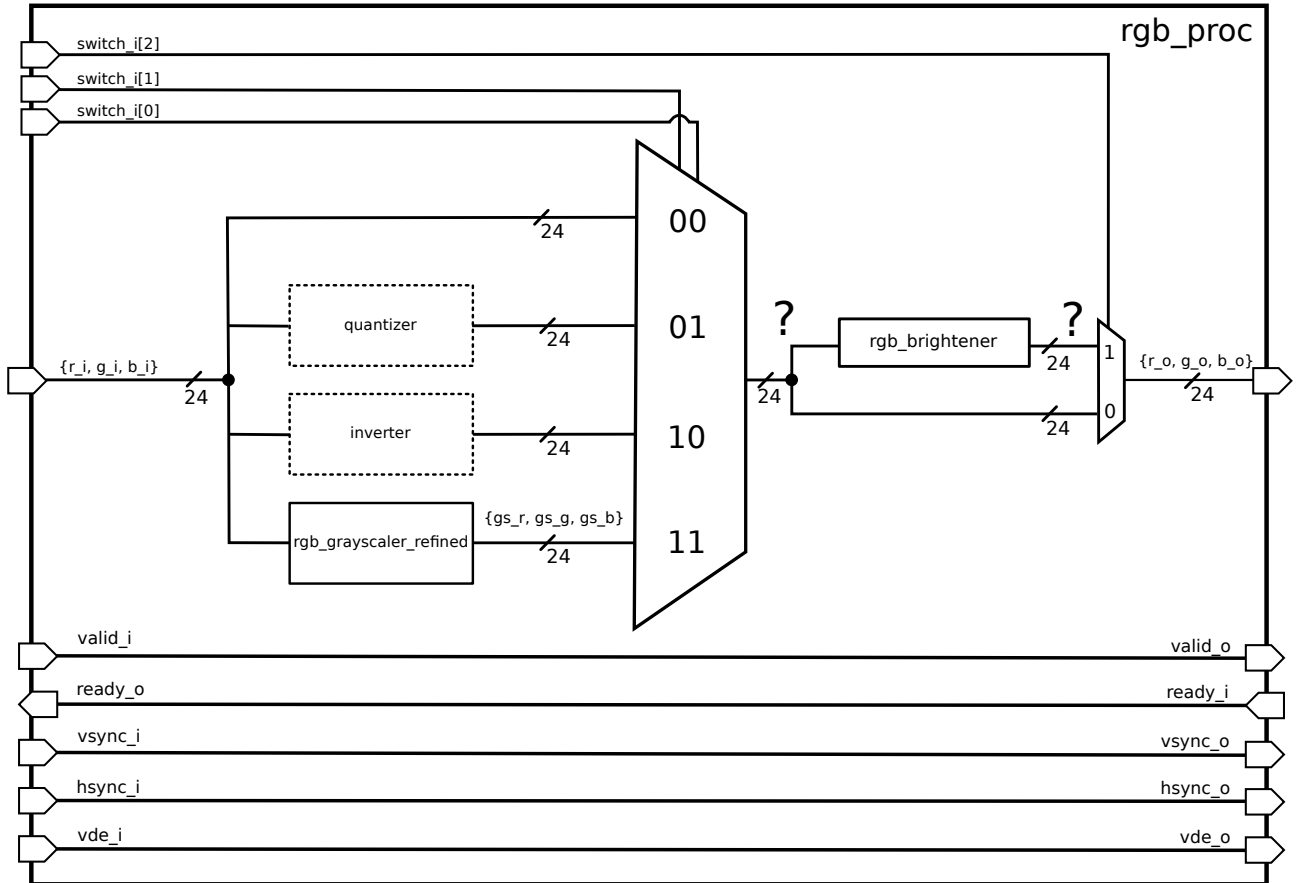


Figure 4: Overview over the circuit including the brightener. Question marks show where intermediate signals with appropriate names have to be defined. Blocks with a dashed outline do not have to be put in standalone modules but can be implemented in-line in `rgb_proc`.

Student Task 21 (Programming the device): Do all the necessary steps to program your device and make sure its functionality matches your expectations.

You have reached the end of the 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