

Department of Information Technology and Electrical Engineering

VLSI I: From Architectures to VLSI Circuits and FPGAs

227-0116-00L

Exercise 1

Introduction to FPGAs

Prof. L. Benini
F. Gürkaynak
M. Korb

Last Changed: 2022-09-26 18:10:25 +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 learn the basics of working with a field-programmable gate array (FPGA) development board. First, you will get an overview of what it takes to describe a hardware design on the FPGA. Second, you will go through the steps of deploying a basic design on the FPGA, explore its functionality, and understand its components. Finally, you will check your understanding by taking another design that adds some functionality but also contains some flaws, deploying it, and getting it to work correctly.

Most questions in this exercise are phrased very openly because they are about principles and concepts rather than closed-form solutions. To ensure the best learning experience, please, discuss the emerging questions and your answers to the problems with one of the assistants every now and then. Active communication with teaching assistants is your key to a successful exercise.

2 Preparation

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/vlsil/ex1/install.sh
sh> cd ex1
```

3 Describing Hardware on an FPGA

Before we dive deep into the tools, we start with a short summary describing hardware on an FPGA.

3.1 Resources of an FPGA

Each FPGA is composed of a fixed set of basic elements of different types. By configuring and connecting these elements in a custom way, you, as a hardware designer, can implement custom designs on an off-the-shelf FPGA.

The Xilinx Zynq-7020 System-on-Chip (SoC) in front of you features five different types of basic FPGA elements: configurable logic blocks (CLBs), block random access memories (BRAMs), digital signal processing (DSP) slices, input/output blocks (IOBs), and clocking resources. Fig. 1a shows the top-level layout of a Xilinx 7 Series FPGA, which share the same elements with the Zynq-7000 family. The FPGA is organized in multiple clock regions (the six tiles in that figure), each of which is 50 CLBs tall. The main processing workload is performed by the CLBs and DSPs operating on the BRAMs, but understanding I/O and clocking resources is also important. The following gives a short summary of each element type.

3.1.1 Configurable logic block (CLB)

Each CLB¹ is composed of two logic slices. A simplified diagram of a CLB slice is shown in Fig. 1b. Each slice contains:

- Four lookup tables (LUTs) with six independent inputs and two independent outputs. A LUT can implement any Boolean function of its inputs by storing and evaluating the truth table of the function.
- Eight flip-flops (FFs), which can store the outputs of the LUTs.
- One 4-bit arithmetic carry chain.
- Four 1-bit function multiplexers.

Between 25 and 50 % of all slices can also use each of their LUTs as one 64×1 -bit distributed RAM, as 32-bit shift register, or as two 16-bit shift registers. This minority of slices is of type *SLICEM* (optimized for memory), while the majority is of type *SLICEL* (optimized for logic). Synthesis tools are aware of the properties and individual capabilities of each slice and automatically try to use them in an optimal way.

¹ See [Xilinx' 7 Series FPGAs Configurable Logic Block user guide \(UG474\)](#) for more information.

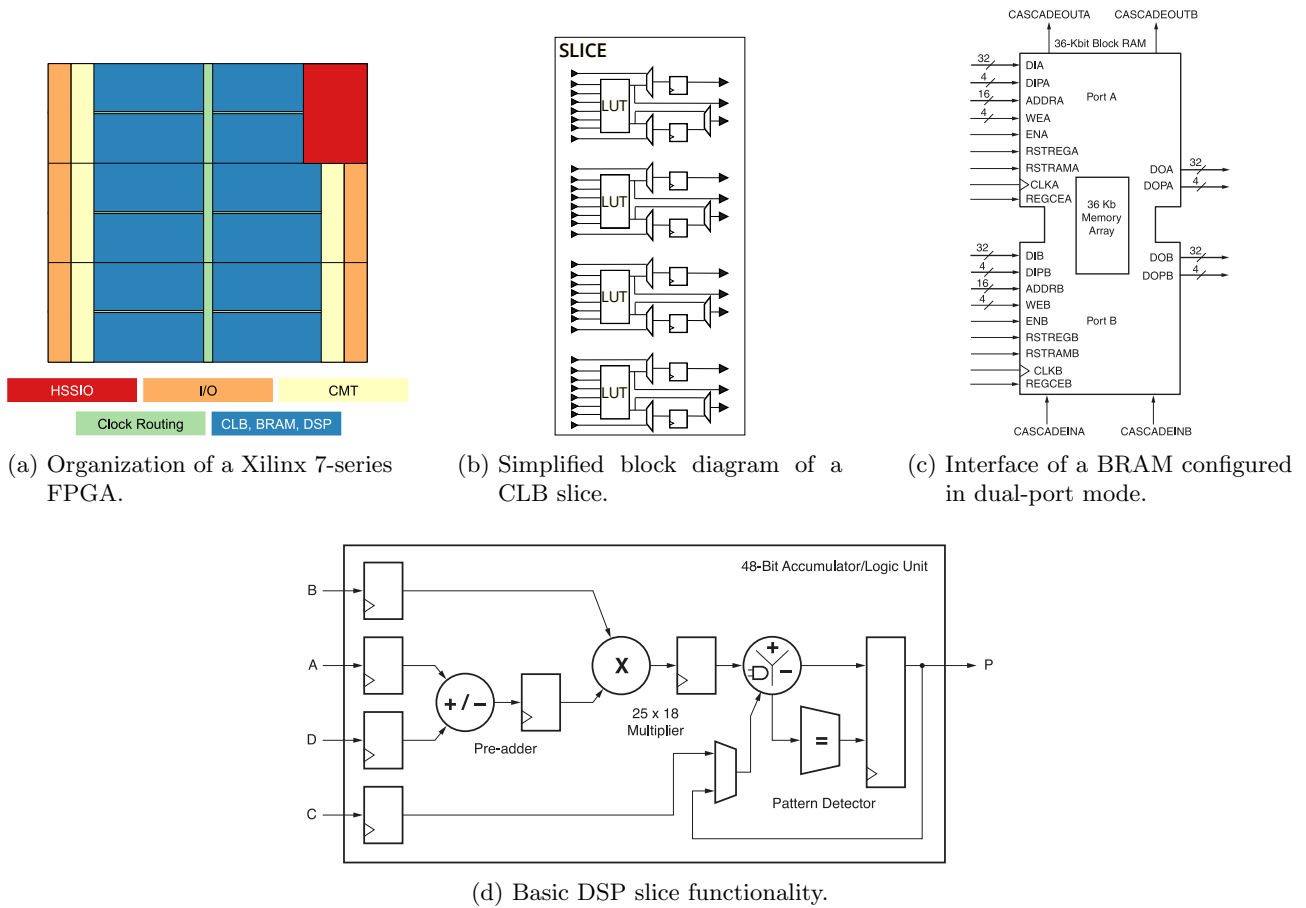


Figure 1: Essentials of the functional units in Xilinx 7 Series FPGAs.

3.1.2 Block random access memories (BRAMs)

Each BRAM² stores up to 36 Kibit of data. In a common configuration, a BRAM holds 1024 entries, each of which is 36-bit (32 bit plus one parity bit per byte) wide. In general, BRAMs are highly configurable in their number of entries and width per entry. In the dual-port mode—the interface of which is shown in Fig. 1c—each BRAM has two independent ports with (32+4)-bit data input, byte-wise write enable, 16-bit address³, (32+4)-bit data output, and separate clock, reset, and enable signals.

3.1.3 Digital signal processing (DSP) slice

The basic functionality of a DSP slice is shown in Fig. 1d. Each slice⁴ includes one 25×18 -bit two's complement multiplier, one 48-bit accumulator, one 25-bit pre-adder, an optional logic unit that can generate any of ten different logic functions on two operands, and a pattern detector for rounding. Additionally, each DSP slice includes optional input, pipeline, and output registers. In fact, at least three pipeline registers are required for multiply and multiply-accumulate operations to run at full speed.

3.1.4 Input/output blocks (IOBs)

There are two different types of IOBs⁵: high-performance (HP) and high-range (HR). The HP IOBs are designed to meet the performance requirements of high-speed memory and other chip-to-chip interfaces with

² See [Xilinx' 7 Series FPGAs Memory Resources user guide \(UG473\)](#) for more information.

³ Addressing 1024 32-bit entries byte-wise would only require 12 address bits, but the BRAMs also support narrower, deeper RAM configurations. The deepest configuration supported is 64 Ki entries of 1 bit each, which requires 16 bits to address.

⁴ See [Xilinx' 7 Series DSP48E1 Slice user guide \(UG479\)](#) for more information.

⁵ See [Xilinx' 7 Series FPGAs SelectIO Resources user guide \(UG471\)](#) for more information.

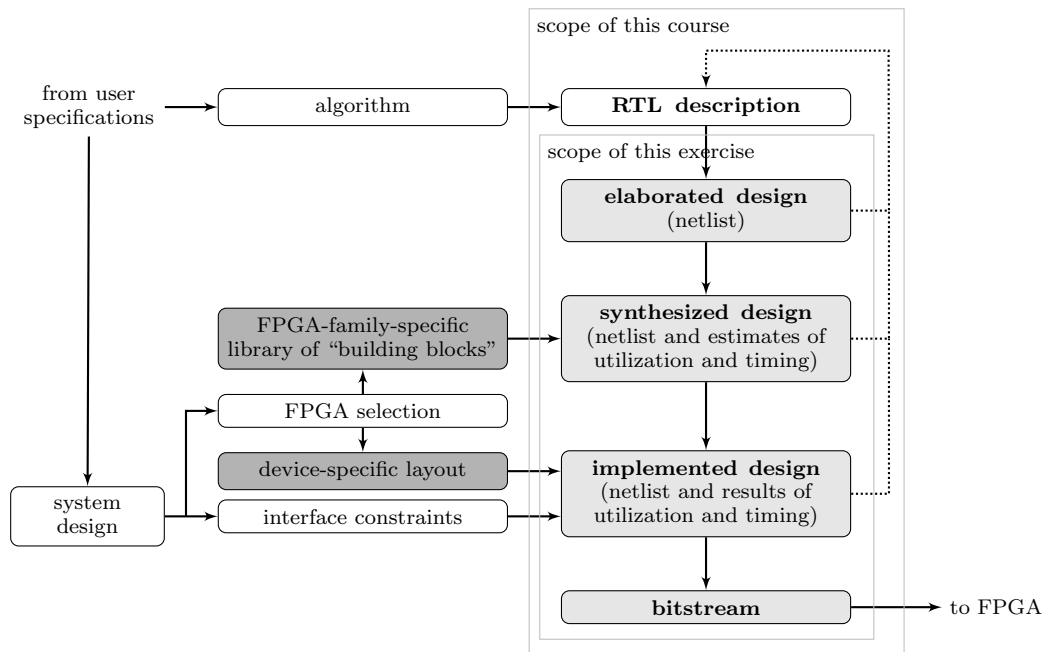


Figure 2: FPGA design and implementation flow. Blocks in white are parts of your design, blocks in light gray are produced from your design using electronic design automation (EDA) tools, and blocks in dark gray are static data supplied by the FPGA vendor.

voltages up to 1.8 V. The HR IOBs are designed to support a wider range of I/O standards with voltages up to 3.3 V. In addition to programmable voltages, IOBs feature a programmable input or output delay, dedicated serializing/parallelizing converters, and programmable reference voltages.

3.1.5 Clocking resources

There are three layers of clocking⁶ on the FPGA: clock-capable inputs and global and regional clock resources. An external clock must be brought into the FPGA on dedicated inputs, which provide balanced high-speed access to the global clock resources via dedicated routing. Global clock resources comprise the clock management tile (CMT) and global clock buffers. These buffers are used to create multiple copies of the input clock signal and drive several loads, while achieving low jitter and fast rise/fall time. From CMT, the clock is distributed with a low skew over the regional clock trees to all sequential resources in each clock region.

3.2 The FPGA design and implementation flow

Fig. 2 shows the FPGA design and implementation flow from user specifications to the configuration of the FPGA.

You will spend a large part of this course in the step from algorithm to hardware architecture described at the register-transfer level (RTL). You will learn that for a given algorithm there are many different hardware architectures, each of which comes with its own trade-off of design objectives. Once the hardware is described, there is little room for changing the properties of the final hardware.

The light gray boxes are the steps your design takes before it can be deployed to the FPGA. In this exercise, you will go through these steps. While the steps are mainly automated, it is nonetheless crucial that you understand what they do and how this influences the implementation of your design. We will therefore start with an overview of each step.

⁶ See [Xilinx' 7 Series FPGAs Clocking Resources user guide \(UG472\)](#) for more information.

3.2.1 Elaboration

During elaboration, a netlist⁷ of behavioral logic and arithmetic elements is generated from the RTL description. For example, an addition of two signals is transformed into an adder with the corresponding inputs. This step is independent of the technology: the aforementioned adder is not directly a logic element on the FPGA. You can use the elaborated netlist to check that your hardware is consistent with your intentions. (For example: Is the adder in a 10-bit counter that you might have described really just 10 bits wide? How is the enable/disable functionality of that counter implemented?) Elaboration usually takes a few minutes at most and completes within seconds for the designs we consider in the exercises.

3.2.2 Synthesis

During synthesis, the elaborated netlist is mapped onto elements available in the target FPGA family. In the resulting netlist, the used elements are configured and directly connected in a way that is functionally equivalent to the elaborated netlist. For example, the aforementioned adder might be mapped onto a couple of logic slices (if it is very simple, e.g., part of a counter) or onto a DSP slice (if it is more complex, e.g., part of a digital filter). From this netlist, the tools can derive a first estimate of the utilization and timing of the design. These estimates (especially utilization) are accurate enough to necessitate changes in the RTL description if they significantly deviate from requirements. Synthesis can take up to a few hours for complex design, but will be around one minute for the designs we consider in the exercises.

3.2.3 Implementation

During implementation, the synthesized netlist is mapped onto the target FPGA device. Implementation is divided into two steps called place and route. During **placement**, each element in the synthesized netlist is assigned to one physical element on the FPGA. The tools try to place functionally-coupled elements near to each other, but this might not always be possible. During **routing**, the signal flow between the placed elements is implemented using the routing resources of the FPGA. The tools try to balance routing delays such that timing requirements can be met, but this task might be challenging on densely utilized FPGAs. The resulting netlist fully defines the configuration-time state of the FPGA: which elements are used in which configuration, how routing resources connect elements, how I/O cells get signals to and from the FPGA, which memory elements are used and what their initial state is, and so on. The utilization results of the implemented design are fully accurate, and its timing results are as close as you can get to reality when taking statistical variations into account. If the implemented design does not fit onto the target FPGA or does not meet your timing requirements, you have to adapt your circuit by changing its RTL description. Implementation can take up to a day for complex designs, but will be less than five minutes for the designs we consider in the exercises.

3.2.4 Bitstream generation

A *bitstream* is a sequence of bits that configure transistors on the FPGA. During bitstream generation, the bitstream is obtained from the implemented netlist through an injective mapping. Bitstream generation can take some minutes, but completes within seconds for the designs we consider in the exercises.

3.3 Getting data to and from FPGAs

FPGAs communicate with peripherals through input/output (I/O) ports, of which they have quite many. For example, even the relatively small device you have in front of you has 100 user-configurable I/O ports. On the development board, most of the ports are wired to one of the many peripheral devices. There are simple peripherals, such as push buttons, switches, and LEDs, and there are more complex devices, such as memories, other ICs, high-speed data ports (e.g., Ethernet, USB), and video ports (one HDMI input and one output).

These vastly different peripherals all come with different electrical requirements on current drive and voltage as well as timing. Moreover, a single FPGA must be able to work with different devices and PCBs.

⁷ A *netlist* describes the connectivity of an electronic circuit by listing the components of the circuit and the connections between the components.

To achieve this flexibility, the I/O resources of an FPGA are highly configurable (as briefly discussed in § 3.1.4). This configuration is specified at design time in a so-called *constraints file*. That file also defines the internal names for the signals on the I/O pins.

4 Programming the FPGA Development Board

Most FPGA development boards feature *jumpers* to make or break connections on the PCB. To avoid damage to a board, you should always check if its jumpers are set correctly before connecting it to anything.

On the Zybo, jumpers are used to select the power source (options: wall plug, programming USB port, or battery) and the source of the FPGA bitstream (options: SD card, flash via QSPI, or JTAG). We will power the Zybo via the programming USB port and load its bitstream from our workstation via the programming USB port that is connected to the JTAG.

Student Task 2 (Controlling and setting the jumpers): Before connecting the FPGA board, make sure that the jumpers are set as follows:

- The Power Select jumper, JP6 (at the top left), must be set to USB, i.e., it must connect VU5V0 to USB.
- The Programming Mode jumper, JP5 (at the top right), must be set to JTAG, i.e., to the rightmost position.

Student Task 3 (Connecting the board): Plug the USB cable into the PROG UART connector of the FPGA board and into one of the front USB ports of your workstation. Switch the board on and make sure that the PGOOD LED lights up and constantly glows.

The FPGA is now ready to be programmed. We will use Vivado, Xilinx' FPGA development environment, to load the introductory design.

Student Task 4 (Starting Vivado): In your `ex1` directory, change to the `intro/vivado` directory and start XILINX VIVADO 2017.2 with

```
sh> vivado-2017.2 vivado
```

You are now on Vivado's welcome screen. The tool has a menu bar on the top, a large graphical working area in the middle (which currently shows common entry points and recently used projects), and a console at the bottom.

Like most of the CAD/CAE tools we use, Vivado can be controlled through its GUI and through its built-in Tcl console. *Tcl* (short for "tool command language") is a general-purpose, interpreted ("scripting") programming language, similar to Bash scripts you saw in the previous exercise. The GUI is usually the more intuitive way to use the tool and as such is preferable when doing a task for the first time. Using the console, however, repetitive tasks can be automated easily. Fortunately, creating a script for the console from GUI actions is easy, as most tools print commands that are equivalent to your GUI actions to the console. For example, Vivado prints commands in blue; after start-up, its console shows `start_gui`. You could use this command at some point in a script to start the GUI after a batch initialization sequence, for example.

Note 1 (Recommendation on using GUI and console in CAE/CAD tools): Use GUI and console in the CAE/CAD tools complementarily: When you perform a task for the first time, use the GUI. Commands equivalent to your actions will be printed on the console. If you want to be able to reproduce your actions automatically (e.g., if you have to do the same steps multiple times, or to let others reproduce your work), save the commands into a script file. You can let a tool execute such a script by entering `source path\to/script.tcl` into its console.

We have prepared the source code (RTL description and constraints) of a hardware design for this exercise. To deploy the design to the FPGA, we will now create a Vivado project and import those files.

Student Task 5 (Creating a script file for Vivado): Create the file `intro/vivado/init_proj.tcl` and open it with the editor of your choice. Throughout the following GUI steps of setting up a project, watch the Tcl console in Vivado for commands printed in blue, and paste them into that file. This script will be used later to set up a new project.

Student Task 6 (Creating a new Vivado project):

- In the `Quick Start` box, click on `Create Project`.
- Click `Next`, name the project “intro” and make sure that it will be created at `intro/vivado` (set location accordingly and uncheck the *create project subdirectory* option).
- On the next screen, choose *RTL Project* and make sure *Do not specify sources at this time* is unchecked.
- Click `Next`, then `Add Directories`, select the `intro/sourcecode` folder, and click `Select`.
- Make sure that
 - *Scan and add RTL include files into project* is selected,
 - *Copy sources into project* is unchecked,
 - *Add sources from subdirectories* is selected, and
 - the *Target language* is Verilogand click `Next`.
- In the *Add Constraints* window, click `Add Files`, add the `intro/vivado/constraints/zybo-z7.xdc` file, and uncheck *Copy constraints files into project*.
- Click `Next` to go to the window where you are asked to pick an FPGA for your project.
- Switch to the `Boards` view, scroll down in the list, and select the *Zybo Z7-20* board (Rev. B.2).
- The next window presents the summary of the project to be created.
- Check the presented settings and hit `Finish` if you agree.

Vivado will now create the project and print the executed commands in the console. When it is done, you are presented with the *Project Manager* screen, which shows an overview of the project and its configuration options.

The *Flow Navigator* on the left-hand side is used to navigate between the steps of the FPGA implementation flow (see Fig. 2). The currently active step is highlighted in **bold on a light blue background**. Today, we will be going through the *RTL Analysis*, *Synthesis*, *Implementation*, and *Program and Debug* steps; we will cover the *IP Integrator* and *Simulation* in other exercises.

In the context of this exercise, we are not interested in the fastest, smallest, or most efficient implementation of our circuit. Instead, we want Vivado to generate results as quickly as it can. We will now instruct it to do so by selecting the appropriate synthesis and implementation strategies.

Student Task 7 (Selecting synthesis and implementation strategies):

- Go to the `Settings` of your project (under the `Project Manager` step).
- Switch to the `Synthesis` tab and select `Flow_RuntimeOptimized` as *Strategy*.
- Switch to the `Implementation` tab and select `Flow_RuntimeOptimized` as *Strategy*.
- Hit `OK` to confirm the settings.

Let us bring our design to the FPGA and discuss the steps and details after that.

Student Task 8 (Generating the bitstream): Take the fast track and select `Generate Bitstream\` (under `Program and Debug` step or with the same symbol in the menu bar). In the configuration window if that pops up, set the number of parallel jobs to the maximum available and confirm the default values for the other settings. Without further questions, Vivado will now automatically perform all steps in the flow to generate the bitstream. You can follow the progress under the `Design Runs` tab at the bottom of the screen. What does that window report? How long does bitstream generation take in total?

If the bitstream cannot be generated successfully (it really should), ask an assistant for help.

Student Task 9 (Programming the FPGA):

- Connect to the FPGA by selecting `Open Hardware Manager` (in the pop-up dialog or under `Program and Debug`), clicking on `Open Target`, and finally `Auto Connect`.
- Under the `Hardware` tab of the Hardware Manager, you should now see your `xc7z020_1` FPGA with status *Not programmed*.
- Click `Program Device`, select the FPGA, select *Enable end of startup check* (the default bitstream file should be fine), and hit `Program`.

After a few seconds of programming, notice that the green `DONE` LED on the FPGA lights up. This means that the FPGA has been programmed successfully and is ready for operation.

5 Understanding the Introductory Design

Let's try to understand the implemented circuit by playing with it on the FPGA board. The circuit features two simple modes, which can be switched with the rightmost push button (labeled `BTN0`). The other three push buttons and the four switches figure as inputs, the four LEDs above the switches are the outputs.

Student Task 10 (Exploring the behavior of the implemented circuit): Use the inputs and outputs described above to explore the behavior of the circuit. How would you describe the two modes? How do the inputs affect the output in each mode?

5.1 The elaborated circuit

Pressing buttons and guessing functionality may be fun, but it is not a very reliable way to characterize the circuit. Instead, let us open the block diagram of the elaborated design in Vivado and derive our own block diagram from that and a description of the blocks.

Student Task 11 (Viewing the schematic of the elaborated design): Expand the `Open Elaborated\` `Design` navigator item and click on `Schematic`. Vivado will now elaborate the design and present you with a netlist on the left and a schematic on the right. Both are hierarchical and can be expanded, but we are interested in the top-level at the moment. Maximize the *Schematic* window.

The inputs of the circuit (the clock, push buttons and switches) are on the left, its outputs (the LEDs) are on the right. The two modes are implemented in two blocks that are multiplexed on the LEDs. The output multiplexer is driven from a register that is toggled by the rightmost push button.

While the switches can directly be used as inputs in this circuit, the push buttons must be *debounced* before they can be used: When a mechanical contact closes or opens, its electrical output oscillates (or “bounces”) for some time before settling on a value. A debouncer is a simple block that switches its output only if its input remains stable for a configurable number of clock cycles. Since we are only interested in the “button pressed” event, the debounced signal is fed through another simple block that outputs a single-cycle pulse when its input changes from 0 to 1.

Student Task 12 (Deriving a top-level block diagram): Based on Vivado’s schematic of the elaborated design and the description above, draw your own top-level block diagram. **Hint:** You can make your diagram more intuitive than Vivado’s by grouping similar blocks, putting closely related blocks together, and drawing the circuit controlling the multiplexer also as a block. You may also analyze the source code files to understand the design better.

You may wonder how Vivado knows what the inputs and outputs of the circuit are and how they are connected. After all, the top-level schematic starts from input ports on the left side and ends in output ports on the right side. These ports are described in the *constraints file*.

Student Task 13 (Understanding the constraints file): Un-maximize the Schematic window and switch to the *Sources* tab in the window to the left of the schematic. Open the `zybo-z7.xdc` file in the `Constraints/constrs_1` folder by double-clicking on it. This file is full of Vivado-specific Tcl commands that define ports. What are the three pieces of information necessary to connect an internal signal to an external one through a pin of the FPGA?

In addition to that information, all non-clock inputs and outputs should be constrained with input and output delay, respectively. In our case, all inputs and outputs are asynchronous, however: neither switches, nor buttons, nor LEDs are related to the clock of the FPGA. As such, no meaningful input or output delay can be specified on them. Instead, they are marked as *false paths* and assigned an arbitrary delay of zero.

The following two tasks are to roughly understand the functionality of the blocks. Please don’t try to figure out the complete behavior of all blocks for the sake of time.

Student Task 14 (Understanding the auxiliary blocks): Maximize the *Schematic* window again and expand the schematic of one of the `rising_edge_pulsers` by clicking on the plus symbol in its top left corner. Which combinatorial and sequential (i.e., state-holding) elements does it use? How are they connected to accomplish the functionality described above?

Expand the schematic of one of the `debouncers`. Which elements form its state and how are they controlled? What drives its output? Explain broadly how it achieves the described functionality. **Hint:** Focus on registers and counters, and do not get distracted by the huddle of multiplexers. We also recommend analyzing the source code to better understand the design.

Estimate how many clock cycles the input must be stable to influence the output. (Calculate the upper limit of this number.) Assuming a clock frequency of 100 MHz, how much wall clock time would that be? **Hint:** Look at the counter.

Student Task 15 (Understanding the main blocks): Expand the schematic of the `led_switcher` block. What does it do?

Expand the schematic of the `led_toggler` block. What does it do? Which elements form its state and how are they controlled? What drives its output?

These schematics show the logic structures that Vivado generates from the hardware described by the designer. Behavioral HDL code, however, usually contains much more information on the intentions of the designer. Whenever available, understanding that code is key to understanding the circuit. At this point, you have not yet been trained in reading HDL code. Nonetheless, you can get a first impression and try to understand the outline of the code describing the blocks you already got to know.

Optional Student Task 16 (Browsing the HDL of the blocks): Un-maximize the *Schematic* window and, in the *Netlist* tree on the left, double click on the most simple unit, the `led_switcher`. The new window shows the SystemVerilog code for that block. After an introductory header comment, you can find the *module declaration* of the block. What does it describe?

The rest of the file describes the internals, the *implementation*, of the module. Which statement connects

one signal to another in SystemVerilog?

Next, have a look at the SystemVerilog code of the `rising_edge_pulser`. Which code elements form the register that holds the input of the previous clock cycle? Which statement determines the output?

Finally, open the code of the top-level `zybo_top` unit. You are already familiar with the module declaration, but the implementation contains quite some new constructs. What do they do?

Before, you have seen one statement to connect two signals in SystemVerilog. How are most signals connected in this implementation?

In this code, the delay of the debouncers is defined. What is its value and does it match the value you expected?

You may also have a look at the code of the `debouncer` and the `led_toggler`, of course, but you do not have to understand them yet.

5.2 The synthesized circuit

Let us leave the elaborated circuit and turn to the more FPGA-specific synthesized circuit. (Refer to the introduction if you are unsure about the difference.)

Student Task 17 (Viewing the schematic of the synthesized design): Expand the `Open Synthesized Design` navigator item and click on `Schematic`. How does this schematic compare to the one of the elaborated circuit (similarities, differences)?

What kind of blocks is the circuit made of? How do they implement the functionality of the circuit?

Where has the `led_switcher` gone?

After mapping functionality to real FPGA elements, first estimates on timing and utilization can be made.

Student Task 18 (Understanding the timing summary): Click `Report Timing Summary` and confirm the default options. The timing summary will open in a new tab in the window at the bottom. What is the clock period and frequency?

Does the circuit meet the timing constraints, and could it run even faster? If yes, up to what frequency?

Where in the circuit is its longest path?

What contributes more to that path, logic elements or signal routing?


How much can you trust such detailed results at this point and why?

Post-synthesis timing estimates are usually less constraining than post-implementation timings. Thus, if your circuit does not meet its requirements after synthesis, chances are very low it will do so in implementation.

Student Task 19 (Understanding the utilization report): Click `Report Utilization` and confirm the default options. The utilization report will open in a new tab in the window at the bottom. Click on `Summary`. Which kinds of FPGA resources does the circuit use? How heavily is the FPGA used?

Click on `Hierarchy`. Which block uses how much of which resource type? Which block uses the most resources overall, and would you have thought so when we first described the functionality of the blocks?

As part of synthesizing the design, Vivado has already allocated a few elements on the FPGA. The *Device* view shows the layout of the FPGA and how its elements are used.

Optional Student Task 20 (Understanding the device view): Open the `Devicea` tab of the synthesized design and try to match the shown layout to your knowledge of the internals of FPGAs. You can zoom this view with  + mouse wheel and pan it by holding the middle mouse button and dragging. What are

the six large rectangular zones?

What types of columns are there within each zone?

What kind of “special” columns, which only exist at particular spots, are there?

What are the cells at the right border of the device?

Most cells in the layout are not filled with color, meaning they are not (yet) allocated. A few cells, however, are already allocated. Which cells are that, and why could Vivado place elements of the circuit there before the actual placement phase?

^a If this tab is missing, you can open it from the Window menu.

5.3 The implemented circuit

Finally, we will inspect the implemented design. (Refer to the introduction if you are unsure what happens with the synthesized design during implementation.) Since we last looked at the device view of the synthesized design, we will start with the device view of the implemented design.

Student Task 21 (Understanding the implementation on the device): Click on `Open Implemented\Design` in the flow navigator and switch to the `Device` tab (if it is not already open). A few cells have been filled and they are quite closely grouped in one spot. What are the reasons for this placement?

Staying in the “macroscopic” view for a moment, open the `Netlist` tab (left of the `Device` window). Color the cells that implement the different blocks in different colors: for each first-level entry in the netlist, right-click on the entry and select a distinct color under `Highlight Leaf Cells`. What do you observe?

In the `Netlist` tab, select the three `clk.i*` nets, right click, and highlight them with a color. How does the clock signal come into the FPGA, and how is it distributed to the circuit blocks? Which FPGA element

plays a central role in this, and where is it located?

Finally, zoom in on one of the slices until it fills your screen. What elements does it have and how are they used? Examine different slices that implement different parts of the circuit. Which slice elements are often used, which almost never? Why?

Optional Student Task 22 (Comparing post-implementation to post-synthesis results): Generate the timing report for the implemented design and compare it to the one of the synthesized design. How did the longest path change? Does it still involve the same class of endpoints? Did its total delay change, and if so, which part of the path caused the change?

Generate the utilization report for the implemented design and compare it to the one of the synthesized design. How did overall resource utilization change and why? How did resource utilization by individual blocks change?

Hint: Compare the schematics of a block in the synthesized and the implemented design by right-clicking on the block in the utilization hierarchy and opening its schematic.

6 Deploying and Fixing the Extended Design

It is time to turn our attention to a functionally more interesting design! Unfortunately, we ran out of time implementing this one and could not quite finish it—so we thought we would make it an exercise for you to complete!⁸

Student Task 23 (Setting up a new Vivado project): Close the currently running Vivado, change to the `work_in_progress/vivado` directory, and start Vivado from there. If you have created a script from the last project setup, copy that file to the `scripts/init_proj.tcl` subdirectory, adapt the project name and the paths of source and constraint files, and execute it in Vivado by entering

```
source ./scripts/init_proj.tcl
```

in Vivado's Tcl console. If you did not create such a script, setup a new project with adapted settings.

Hint: You can make your `init_proj.tcl` script a lot more generic by letting it include *all* files in the `sourcecode` directory instead of explicitly listing individual files. To do so, replace the `add_files {`

⁸ Please do yourself and us a favor and take this with a grain of salt before writing articles in electrically charged magazines. ;-)

```
... } statement with add_files [glob ../sourcecode/*].
```

Student Task 24 (Generating and deploying the bitstream): Generate the bitstream for the project. There have been some problems with this design in the past, and our verification team has not approved it yet (although the responsible implementation engineer swore that everything is fine ...). In any case, watch out for warnings and errors in the *Messages* window. Note any serious problems you find and how you fixed them:

Hint: Although the RTL designers are overly confident in their code most of the time, the RTL code really is fine at this point (you might have to debug it later, though ...), `vimdiff` might be a helpful tool here.

Once you have the bitstream, deploy it to the FPGA.

Unfortunately, the design has not been documented yet, but a whiteboard drawing in the coffee room says that it features three modes: the first two modes have been adopted from the first design, but the third mode has been specifically designed for a client from the film industry. The client was really pressed for time to produce the movie, so we had no time to verify it properly. The circuit should work perfectly nonetheless, since we are all masterminds and the functionality of the circuit really is below our intellectual level!⁹

Student Task 25 (Testing and fixing the basic functionality): Use the buttons and switches to test the functionality of the first two modes. Can you confirm that everything works as before? If not, which problems do you notice and how can you fix them?

Hint: There are exactly two distinct problems, one in _____ and one in _____. None of the problems involves code in the subentities. To find the former problem, closely check _____. To find the latter problem, ensure that _____. (Open the `task.pdf` file in your exercise directory and mark the transparent text above to uncover the spoilers.)

Optional Student Task 26 (Exploring the third mode): Once you have verified the basic functionality, you may agree that the team responsible for the third mode did a quite good job. What kind of visual effect did they realize?

Open the schematic of the third mode. It consists of three logical parts: two counters and the output drivers. What is the meaning of the counters? How are the output drivers controlled?

⁹ Whenever you read claims like this, put on your *very* suspicious glasses and expect nasty surprises everywhere.

Draw a simplified block diagram of the block that implements the third mode.

The pulse-width modulator (PWM) that drives the LEDs in the third mode is a central block for driving signals that are interpreted as analog average over time rather than with discrete digital samples. On the LEDs, for example, the human eye does not see a series of digital strobes, but rather a continuously shining light of varying amplitude.¹⁰ Since the human eye perceives brightness logarithmically, the circuit uses logarithmic brightness levels to drive the LEDs.

Optional Student Task 27 (Understanding the implementation of the PWM): Explore the schematic and the RTL code of the `pulse_width_modulator` and the `log_pulse_width_modulator`, respectively. Explain their rather simple implementation:

The logarithmic PWM basically does the same as the standard PWM. How is this implemented without repeating identical logic?

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!**



¹⁰ Have a look at https://en.wikipedia.org/wiki/Pulse-width_modulation if you are entirely unfamiliar with PWM.