

The Script for the Lab FPGA Design

Dear students,

Welcome to the Lab FPGA Design! We hope you will have an enjoyable experience. In this semester, your supervisor is Vida Sobhani. You can reach her via email or by arranging personal meetings on her office hours:

Email: sobhani@ids.rwth-aachen.de

Office hours: Wednesdays 15:00 – 18:00

Address: IDS, Mies-van-der-Rohe Str. 15, 3rd floor, room 302.

We suggest that you install Quartus Lite version 17 on your PC in case you want to do the experiments at home. This software is free for students. Besides working from home, you can work at the institute in room 105. The password for PCs is **pw4!fpga** and the user names are the same as the respective PC: **fpga01-fpga15**. As soon as you decide you want to work with one of the PC from the institute, please contact your supervisors. So, we can provide you with one of the Lab PCs and you do not have to share the same PC with other lab participants.

Although the lab is open every day, the FPGA boards can only be borrowed **Wednesdays from 15:00 to 17:00**. Please make sure that you return the board to your supervisor by **18:00**. To pass the lab, students in a group should present their designs for all 6 experiments described in this script. You do not need to provide written reports for the presentations, however, each individual in the group should be able to explain the design and demonstrate the functionality on the board. Please call the supervisor to check your work at the end of each experiment.

We are looking forward to supporting you in the lab.

Wish you the best of luck and success for the semester ahead,

Your Lab_FPGA_Design Team



Introduction

In this lab, our main goal is to help you learn how to program modern Field Programmable Gate Arrays (FPGAs) using the associated development tools. We aim to achieve two key objectives: first, to give you a solid understanding of the basic architecture of modern FPGAs, and second, to help you practice implementing digital circuits on FPGAs through group projects with various examples (6 following experiments). We will use a development platform provided by the Chair of Integrated Digital Systems and Circuit Design (IDS) to validate your results.

FPGA Design Process. FPGAs are a specialized type of integrated circuit that offers post-manufacturing programmability. It comprises an array of reconfigurable logic blocks interconnected to perform diverse digital tasks. FPGAs find extensive use in scenarios necessitating adaptability, rapid processing, and parallel computing capabilities. FPGA configurations are primarily specified using a hardware description language (HDL), e.g., Verilog (will be used in this lab, [1]–[5]), and VHDL. Alternatively, this description can also be created graphically, in the form of a circuit diagram (.bdf). There is no uniform format for these graphical descriptions, as it is usually a visualization of HDL codes.

The design process can be divided into several consecutive steps shown in Figure 1: **FPGA Design Flow**. As part of the lab, the design process takes place in all steps with the integrated development environment Quartus II by Altera. In general, the design process begins with a "top-down" approach. After design description, the HDL code is then subjected to a syntax analysis. If the description is syntactically correct, the programmer performs a verification of the programmed algorithm using a functional simulation (function analysis). The pure HDL code is simulated, regardless of the underlying hardware. Timing information is not yet considered.

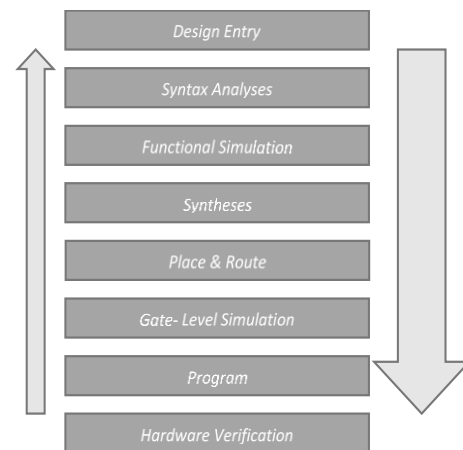


Figure 1: FPGA Design Flow.

After the functionality has been verified, the extraction of a gate-based netlist, which already considers information about the underlying technology, takes place in several synthesis and optimization steps. This netlist specifies how many logic cells are required, how they are programmed, and how they are interconnected. However, it is not yet clear where the logic cells are located on the FPGA and which routing resources will be used.

In the so-called place-and-route step, the extracted netlist is mapped onto the FPGA. With this, the required logic elements are placed on the FPGA and the logic elements are interconnected via the available connection structures (routing resources). After this step, a timing analysis is usually performed, which determines the maximum clock frequency of the design. The maximum clock frequency is determined by the transit times of the logic elements and the delays of the routing.

With this netlist, which contains the mapping information, a so-called gate-level simulation can be performed. This simulation should lead to the same results as the functional simulation. However, changes in functionality are possible, for example, due to races and hazards. This issue can be resolved by revising the HDL description or changing the optimization steps (for example, optimizing for maximum speed or minimum number of logic cells). If the results of the gate-level simulation meet the target specifications, a configuration of the FPGA can be made with the appropriate hardware and

software. Here the correct function on the FPGA can be verified, for example with the help of a logic analyzer, or integrated directly in the target system.

Coding Guidelines. No matter what you do – **be consistent**. Write your code in a tabular manner. Align begin & end parts of your code. Use spaces instead of tabs because tabs might be interpreted differently on different systems, spaces not. Write only one statement per line (plus optional comment). Use explicit port mapping during module instantiation. Use comments and meaningful names to improve readability. Use functions and tasks (instead repeating the same code sections). Use constants & parameters (instead of hardcoded numbers no one understands). Furthermore, your code is easier to maintain if you suddenly need to change something like a bus width. Stick to synchronous designs. Follow consistent naming conventions, e.g., IO signals are extended with `_i`, `_o`. Reset registers only if necessary. Be very careful when using loops because they are very expensive in hardware. Finally, avoid mixed clock edges and manual clock changes and combinational feedback.

Debugging. For Debugging, first, you can eliminate the error messages. The warning messages can be helpful but are not necessarily bad. You can write a testbench to check the functionality of your module. Then you can try different inputs to simulate edge cases. Further debugging can be done by [SignalTap](#). It is a live diagnostic tool for on-board signal analysis with an output of a waveform like Modalism, but with signal trace live on FPGA on-board. However, debug as much as possible in simulation – only move to on-board debugging with SignalTap when out of options.

Experiment 1: Getting started with Quartus II

There are many different FPGA vendors and EDA suppliers on the market. Here, in the lab, we use DE0 Nano Boards [6] (shown in Figure 2) and Quartus II [7]. The main goal of the first experiment is to familiarize yourself with this tool and understand the individual steps for hardware design. We start our first project with the tutorial provided by Altra (pages 45 to 81, chapter 6.3 to chapter 6.10). You can find it on moodle > Materials > Quartus Tutorial. This tutorial creates a design that causes LEDs on the development board to blink at two distinct rates. For the LED design, you will write Verilog HDL code for a simple 32-bit counter, add a phase-locked loop (PLL) megafunction as the clock source, and add a 2-input multiplexer megafunction. When the design is running on the board, you can press an input switch to multiplex the counter bits that drive the output LEDs. Please note that since we use Quartus 17.0 Version, this tutorial needs one modification on page 55-57. This modification is described in the following.



Figure 2: DE0 Nano Board for Experiment 1 to 5.

Modification to Quartus II Tutorial: Adding a Megafunction to the Schematic. Megafunctions, such as the ones available in the library of parameterized modules (LPM), are pre-designed modules that you can use in FPGA designs. These Altera-provided megafunctions are optimized for speed, area, and device family. You can increase efficiency by using Intellectual Property (IP) (in case of Altera: megafunctions) instead of writing the function yourself. Altera also provides more complex functions, called MegaCore functions, which you can evaluate for free but require a license file for use in production designs. This tutorial design uses a PLL clock source to drive a simple counter. A PLL uses the on-board oscillator (DE0-Nano Board: 50 MHz) to create a constant clock frequency as the input to the counter. To create the clock source, you will add a pre-built LPM megafunction named ALTPLL.

Double click in the IP Catalog on (Library > Basic Function > Clocks > PLL > ALTPLL), name it PLL and then click **OK** (Figure 3).

Task 1.1: implementation of a 4-bit counter with different speed

How many logical elements and registers does your design use? (Hint: have a look in the Compilation Report.)

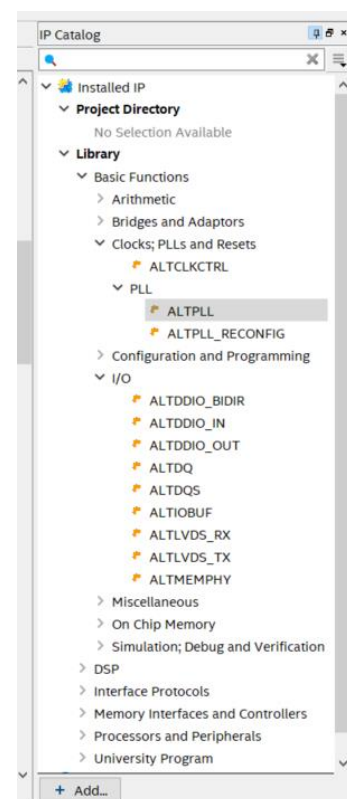


Figure 3: Adding the Megafunction with the IP Catalog.

Experiment 2: Creating a synchronous 8-bit counter

This experiment is a basic start for parallel programming using an example of a synchronous counter. We first describe clock management and then, discuss how counters can be used to generate sub-clocks. Please note that although sub-clocks are derived from the system's global clock, they may be no longer in sync with it due to the necessary switching logic and the poorer wiring over non-specialized clock lines. This shift relative to the system clock is called clock skew (Figure 4) and can lead to unwanted switching states in the circuit. This chapter describes how to create sub-clocks and how to solve the resulting timing issues.

Clock Management. The global clock on an FPGA is usually generated by an external circuit, e.g., a quartz crystal. The global clock is connected to the specific clock inputs to provide a proper (i.e. fast) distribution of clock signal across the entire FPGA. However, sometimes, it may be necessary to provide further, slower clock signals on the chip.

Reduction of the clock rate. The simplest way to generate a reduced clock is to implement a counter and use the most significant bit of this counter as the new clock. This code snippet shows an alternative implementation for a reduced clock frequency. In this example, clock50 is the global 50 MHz clock, and a 4-bit counter is used to drive a slower clock called clock1 with the frequency of 50/16 MHz. Of course, with simple modification, any specific divider factor (≤ 16 , for this example) can be set.

```
reg [3:0] counter;
reg clock1;
if (reset == 0) begin
    counter <= 'b0;
end else if (posedge clock50) begin
    counter <= counter + 1;
if (counter == 0) begin
    clock1 = 1'b1;
end else begin
    clock1 = 'b0;
end
end
```

Clock skew and timing violations. To ensure the correct functionality of a circuit, the data must be stable for a certain time before and after the rising clock edge at the inputs of the registers. These time conditions are called *setup* and *hold*. The setup time is the time before the rising edge of the clock in which the data signal at the register input must be stable. The hold time is the time after the rising clock edge in which the data signal must be kept stable. A *hold violation* happens e.g. if the data changes too early. This may be the case when the clock is delayed by its own logic circuits against the data.

In Figure 4, the clock on the second register is delayed by a logic circuit. This can lead to a hold violation. Figure 5 shows a logic circuit with one clock. Hold violations do not occur, violations of the setup time are avoided by maintaining the maximum clock frequency.

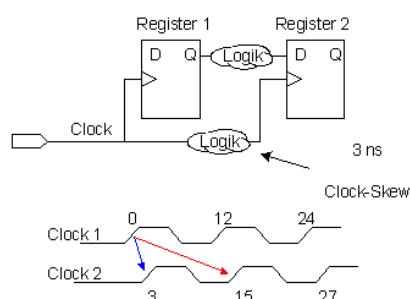


Figure 4: Clock-Skew. SetupRequirement = 15ns, HoldRequirement = 3ns.

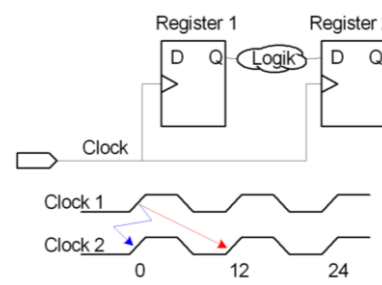


Figure 5: Direct Clock. SetupRequirement = 12ns, HoldRequirement = 0ns.

Control of the Clock Signal - Gated Clock and Clock Enable. This section deals with the control of the clock signal. The simple, but for the reasons already mentioned unfavorable method is the so-called clock gating (see Figure 6). The clock signal is linked with a control signal via an AND gate. However, the delay of the AND gate can lead to a hold violation.

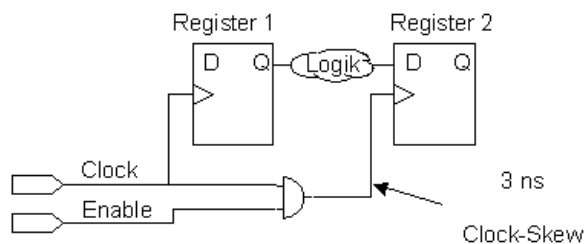


Figure 6: Gated Clock.

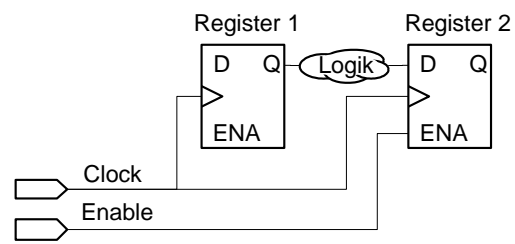


Figure 7: Clock Enable.

Generate a Clock Enable Signal. One way to generate the enable signal is to count a counter up to a certain number, then, set it to zero and at the same time, set the Enable signal (Figure 7) to '1' for one clock cycle. The following example shows how a lower frequency enable signal can be generated from the clock signal. **Hint:** if you struggle with the counter, use 'reg [31:0] count' and compare it with the value `freq_divider = 24'd10000000`. You can choose another clock, but then beware to adapt the `freq_divider` (`freq_divider` is the constant indicating the factor by which the clock signal should be reduced). The simulation results are shown in Figure 8 for the case in which a 1 MHz enable signal is generated from a 10 MHz clock signal (`freq_divider = 10`).

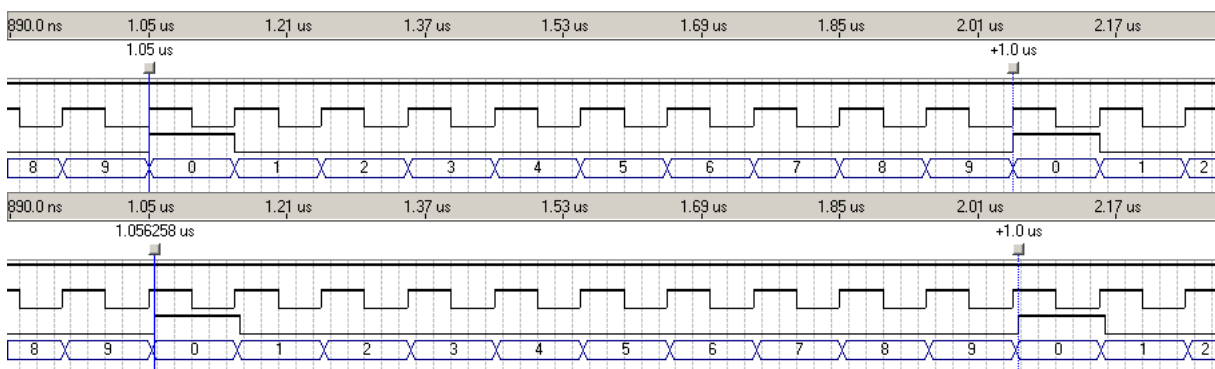


Figure 8: Enable Signal in functional simulation (upper picture) and timing simulation (lower picture).

TASK 2.1: Creating an 8-bit-counter using .bdf.

In the previous experiment, you used a PLL Megafunction to create a counter. The wiring was done via name association on the wires in the .bdf. Now you will learn how to implement this as a fully wired solution without an additional multiplexer. This will be done by using a clock enable signal, since the PLL is just sampling the clock signal of 50MHz down to 5MHz. With this, the signal would be still too fast to display any visual changes in the binary LED counter. In addition to this, you will also build an up-down counter to count either 8 bits up or down in binary.

Therefore, you can copy the previous Quartus files, except the db folders, into a new folder (experiment 2) and then, open Quartus. Now you can create a new project and name it 8_bit_updown_counter.

Here, you should add the module *updown_counter* from the template folder to your project. Modify it and implement a counter with the three following options. The first is to enable and disable the counting process according to input *enable* and the latter is to count up and down, as the user chooses by setting the input *up_down*, and the last is to be able to reset the counting using input *reset*. The result of your counting process has to be assigned to the output *count_out* and later to the LEDs (see Figure 9). The next module you will need is *clkEnable*. Please import this from the template folder and modify it to implement a frequency reduced clock so you can control the *enable* of the module *updown_counter* (the clock enable method illustrated in Figure 7). After you compiled your file, you can create a .bdf symbol and include this in your .bdf file, as shown in Figure 9.

```
module updown_counter(
input up_down,
input clk,
input enable,
input reset,
output [7:0] count_out
);
```

```
module clkEnable(
input clock_5,
input reset,
output enable_out
);
...
always @(posedge clock_5)
begin
if (!reset ) begin
// your code
end else begin
// your code
end
endmodule
```

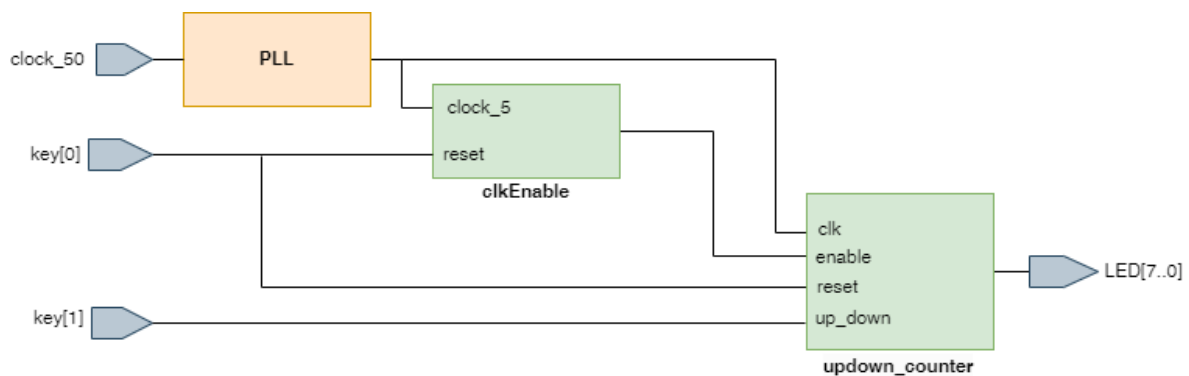


Figure 9: 8_bit_updown_counter.

The assignment should be done like in

Table 1. You can refer to the Datasheets: [8], [9], [10]. Please keep in mind that key[0] and key[1] represent logic high '1' when not pressed and '0' when pressed.

Table 1: Pin Assignment

Signal	Value
LED[0]	PIN_A15
LED[1]	PIN_A13
LED[2]	PIN_B13
LED[3]	PIN_A11
LED[4]	PIN_D1
LED[5]	PIN_F3
LED[6]	PIN_B1
LED[7]	PIN_L3
clock_50	PIN_R8
key[0]	PIN_J15
key[1]	PIN_E1

TASK 2.2: Creating an 8-bit-counter without .bdf.

For more complicated scenarios, the .bdf setup is a bit tedious. It is therefore recommended to do the assignment in a Verilog file. Additionally, this file can then be used in ModelSim to simulate the expected behavior [1]–[4], [11].


For the next version of the 8-bit counter, you can create a new folder and copy the files of the previous project. **Then you can import the files:** *updown_counter.v*, *clkEnable.v*, *top.v*, and *my_first_fpga.sdc*. In the Verilog file *top.v* you should assign the LED to the output of the updown_counter. You can use Key[1] for reset. Please note this code is incomplete and you should instantiate the other modules as well.

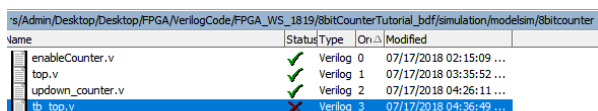
```
module top (
    input clock_5,
    input reset,
    input key1,
    output wire [7:0] LED
);
    wire enable;
    wire [7:0] count;

    clkEnable enb(
        .clock_5(clock_5),
        .reset(reset),
        .enable_out(enable)
    );
endmodule
```

TASK 2.3: Validation of the design via simulation.

When your design is compiled, it is recommended to simulate it in ModelSim or Qsim. Here, one can write a testbench and in it, create the input signals for the design under test (DUT). This will be also instantiated in the testbench module. **Important:** It has neither inputs nor outputs. It is a closed system to elaborate your design.

To start ModelSim please open: **Tools > Run Simulation Tool > RTL Simulation**. Then create a new project and add all Verilog files you are examining. You can also import the tb_top from the template folder. Otherwise you can add a new file, define it as Verilog and create a complete new testbench. After this please save and compile all. If your files contain errors these are shown in the transcript window. By double clicking you can read the message. Please do a right click in the window with this symbol:  and click on **project settings**. Figure 10.a. There you can click on **display compiler output**. With a double click on the error message you should be directed to the error. When your files are compiled, you can click on **Simulate > Start simulation** and the window as in Figure 10.b opens.



Name	Status	Type	On	Modified
enableCounter.v	✓	Verilog	0	07/17/2018 02:15:09 ...
top.v	✓	Verilog	1	07/17/2018 03:35:52 ...
updown_counter.v	✓	Verilog	2	07/17/2018 04:26:11 ...
tb_top.v	✗	Verilog	3	07/17/2018 04:36:49 ...

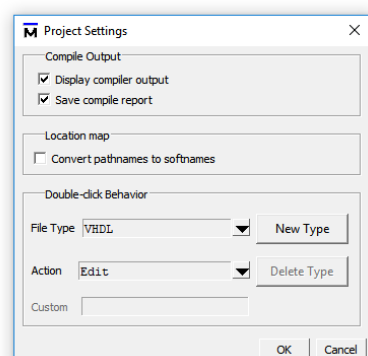


Figure 10.a: ModelSim Project Settings.

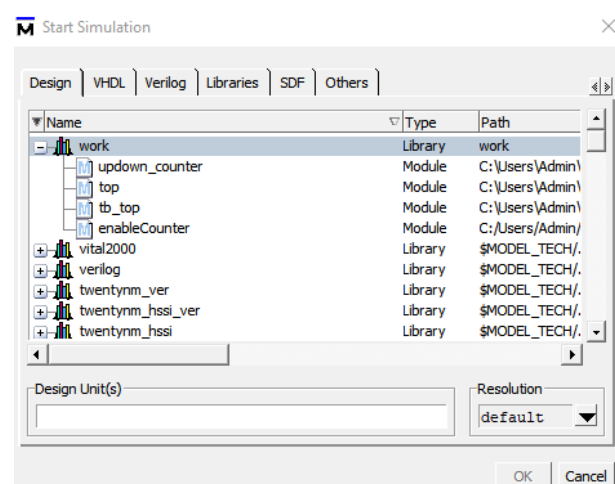




Figure 10.b: Start Simulation.

Now select the testbench file. Then you can select all the input and output signals you want to have a look at. Then you can click **add to wave**. You will see a window as in Figure 11. In the upper corner you can decide how long you want to simulate . Then you can start the simulation by clicking . If you want to restart your simulation, you can click:  You receive a simulation of your design by means of a waveform file (Figure 12).

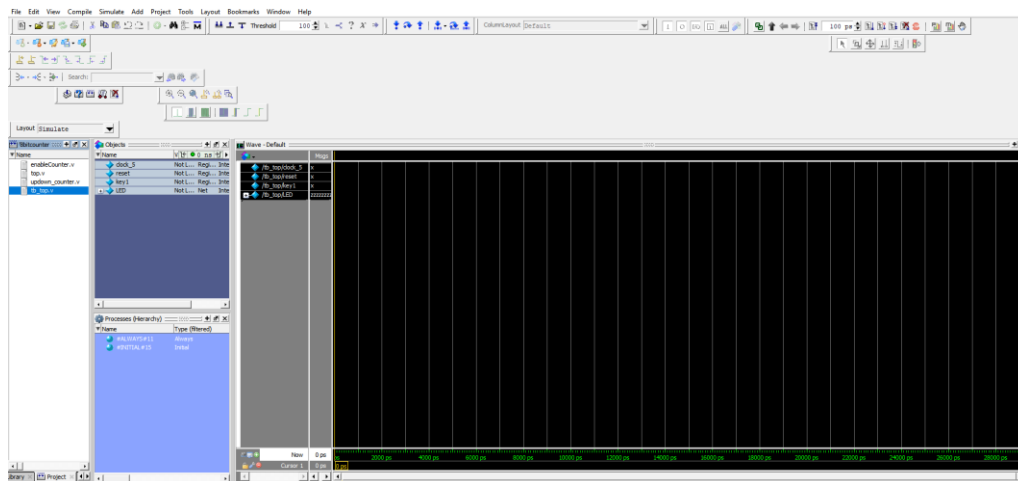


Figure 11: Waveform.

In this scenario, it is recommended to simulate 500 ns. Therefore, you should take care that the frequency_divider is set accordingly. Keep in mind that the board has a 50MHz clock oscillator.

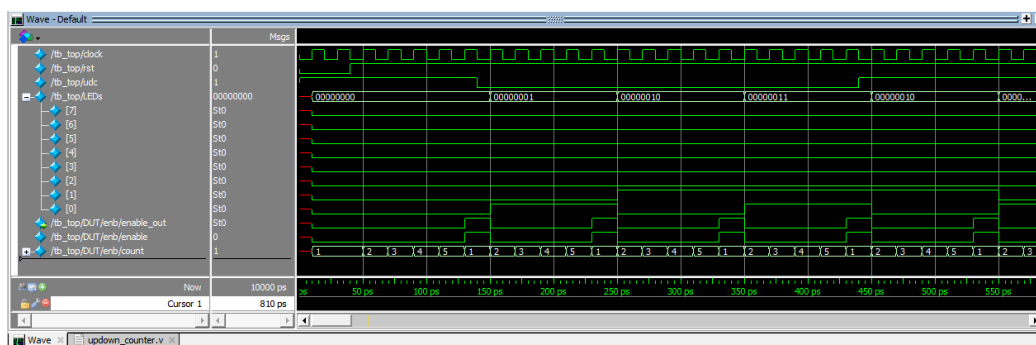


Figure 12: Waveform File.

If you need more information about [ModelSim](#) [12]. If your design is working, you can program the FPGA after informing your supervisor.

Experiment 3: Clockwork

In this and 2 other following experiments (experiments 3-5), a radio clock according to the coding scheme of the longwave time signal DCF77 will be created. The overall system of the radio clock to be realized is shown in Figure 13. Please note that experiment 3 is the foundation code of the clockwork, it will be then extend in experiment 4 and finalize it in experiment 5, therefore, always build upon your previous code.

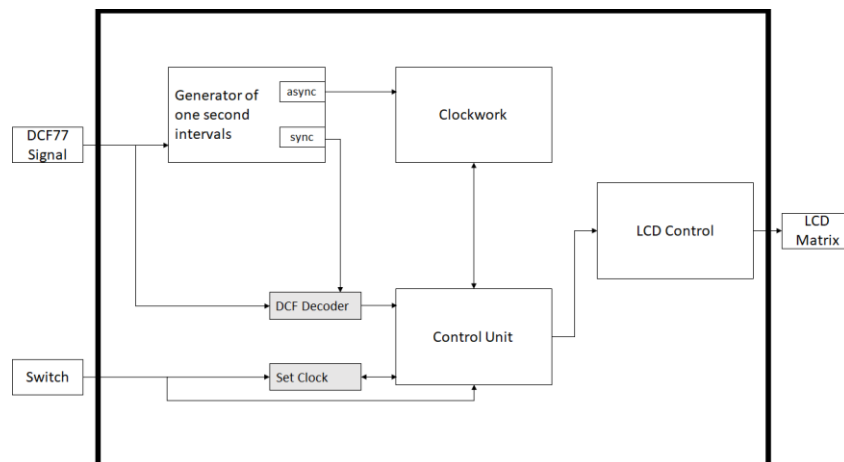


Figure 13: Block Design circuit of clockwork with DCF 77 receiver; thick black box represents the FPGA.

The radio clock is based on a separate clockwork which will be developed in this experiment. Whenever the DCF decoder correctly receives a time signal, the clock is readjusted to the received time (experiment 4). This out-of-the-loop control has the advantage that the clock also works when no DCF signal is received. For this reason, a functional block will be created that allows to set the clock by hand (starts in experiment3 and be completed in experiment 5). A central control unit manages the interaction of the individual blocks (experiment 5). This also gives the information to the block output control. The output control block controls the time display on the LCD matrix. The second-clock generator block generates two clock-enable signals with a clock rate of 1 Hz each (experiment 4). One signal is synchronized with the DCF77 signal. This cannot be used for the clockwork, as a clean signal quality of the received signal cannot produce a clean 1 Hz signal. For this reason, an asynchronous clock enable signal with 1 Hz is generated as well.

Experiment instructions. In this experiment, the clockwork of the radio clock is to be implemented in Verilog. The clockwork should have a date function, i.e., days, weekdays, months and years. Due to the format of the DCF77 signal, it makes sense to implement a separate counter for the individual digits of the clock and to transfer the individual digits to the following blocks. Furthermore, the clockwork should offer the possibility to be set to a certain time. One example module that meets these requirements is given below:

```
module timeAndDateClock(clk, clkEn1Hz, nReset,
    setTimeAndDate_in, timeAndDate_In, timeAndDate_Out);
input clk, clkEn1Hz, nReset, setTimeAndDate_in;
input [43:0] timeAndDate_In;
output [43:0] timeAndDate_Out;
reg [43:0] timeAndDate_Out;
```

```
module setFixTimeAndDate(timeAndDate_Out);

output [43:0] timeAndDate_Out;
reg [43:0] timeAndDate_Out;

endmodule
```

The signals `timeAndDate_in` and `timeAndDate_out` contain the individual digits of the clock. The mapping of the individual signals is given in the following table and can be reproduced in the Verilog code. Brackets `[]` can be used to index the according vector.

Table 2: Projection of the timing information on the output signals.

Name	Places in the signal
Seconds, low-order digit	3 to 0
Seconds, high-order digit	6 to 4
Minutes, low-order digit	10 to 7
Minutes, high-order digit	13 to 11
Hours, low-order digit	17 to 14
Hours, high-order digit	19 to 18
Day, low-order digit	23 to 20
Day, high-order digit	25 to 24
Month, low-order digit	29 to 26
Month, high-order digit	30
Year, low-order digit	34 to 31
Year, high-order digit	38 to 35
weekday	41 to 39
time zone	43 to 42

TASK 3.1: Implementation of a clockwork with seconds

First, create a new project as described in experiment 1. Copy the `timeAndDateClock.v` file from the experiment 3 template to the project directory and add the file to the project. It contains the module above. Initially, implement the function of a clockwork only for seconds. The signal `reset` (active low) should trigger an asynchronous reset, which sets all registers to the value zero.

If the signal `setTimeAndDate` equals '1', all registers should synchronously accept the value at the corresponding inputs.

Simulate your hardware description. Create a testbench to ensure the highest possible test coverage. The enable signal should permanently have the value '1' to reduce simulation time. The clock signal should be clocked at 10 MHz.

TASK 3.2: Extending the clockwork to show the complete time.

Extend your code and add the functionality of a minute and hour clock. Test your design and use the `set` signal to test the cases new minute, new hour.

TASK 3.3: Adding the date function.

Now add a date function to your design. To simplify matters, assume that each month has 31 days. Again, check the function by means of simulation.

TASK 3.4: Creating a complete clockwork.

To test the clock on the FPGA, copy the files `setFixTimeAndDate.v`, `FIFO.vhd`, `LCDsteuerung.vhd` and `lcddriver.vhd`, as well as the clock generator `clkGen_verilog.v` from the experiment 3 templates into your project directory. Please do not edit any of the files without consulting your supervisor. Add the files to the project.

This experiment needs two clocks. One with 1 Hz and one with 10000000 Hz. The first one will be generated in the `clkGen_verilog.v`. The latter is the Quartus Megafunction PLL (Tools/ IP Catalog/ Library/ Basic Functions / Clocks, PLLs and Reset/PLL/ ALTPLL). You may choose Verilog or VHDL, as you prefer. The function blocks have the following function:

- clkGen_verilog.v: One-Hertz generator
- setFixTimeAndDate: This function block sets the clock (using the set parameters) to Tuesday the 31st of July '19, 23:59 minutes and 45 seconds.
- timeAndDateClock: The clockwork created in task 3.1 – 3.3.
- LCDSteuerung: This module sets the time applied to the input pins using the module lcddriver on the LCD matrix. Since this module is also used later for the radio clock, the inputs DCF_Enable_in, minute_start_in and states_in must be connected to GND. Since the states_in signal is a bus, a separate GND signal is needed.
- lcddriver: Control logic for the LCD matrix.

The module lcddriver has a generic port with which the applied clock frequency can be specified. Since this is also to be stated in other modules later in the lab, a so-called *parameter* can be introduced here. These are generic ports for .bdf files. Insert a parameter (<Quartus directory> \ libraries \ primitives \ other \ param) into the .bdf file with the Symbol tool and name the parameter clk_freq. This parameter receives the value 10000000.

Now select the properties of the block lcddriver and assign the value clk_freq to the parameter clk_frequency in the menu 'Parameters'. The required inputs and outputs as well as the associated pin assignment for this project are shown in Table 3.

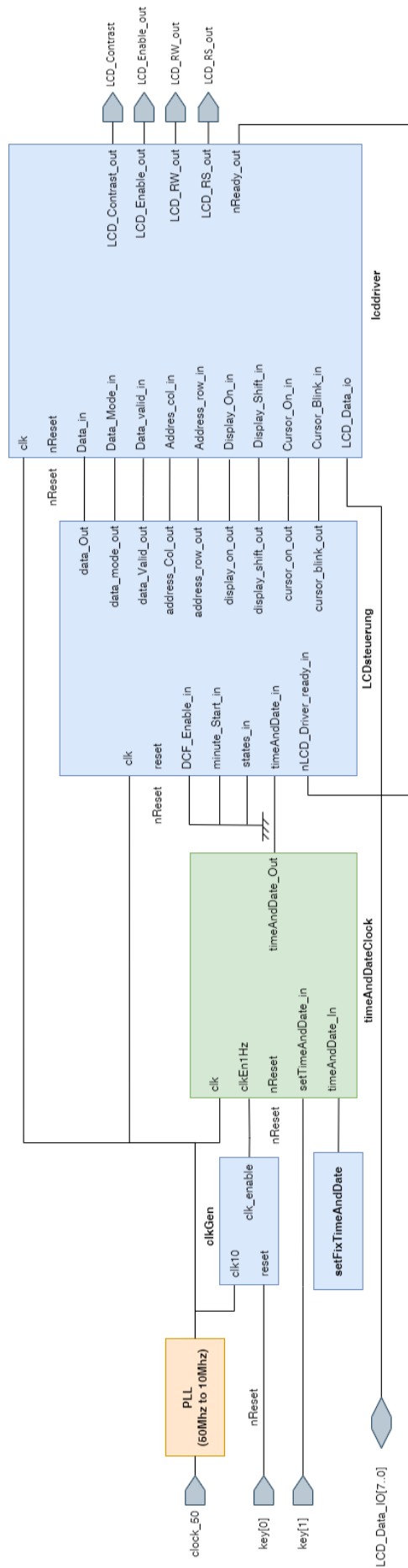
Note that the pins LCD_Data_IO[7..0] are bidirectional pins. These are required because data is exchanged between the lcddriver unit and the LCD matrix. The primitives for bidirectional pins can be found in the Quartus library under the name 'bidir'. One key should be configured to drive a reset signal, the other should drive setTimeAndDate_in (which triggers the timeAndDate module to set the date to the one specified in setFixTimeAndDate).

Table 3: Pin Assignment clockwork.

Pins	Value
LCD_Data_IO[7]	PIN_B7
LCD_Data_IO[6]	PIN_D6
LCD_Data_IO[5]	PIN_A7
LCD_Data_IO[4]	PIN_C6
LCD_Data_IO[3]	PIN_C8
LCD_Data_IO[2]	PIN_E6
LCD_Data_IO[1]	PIN_E7
LCD_Data_IO[0]	PIN_D8
LCD_RS_out	PIN_F9
LCD_RW_out	PIN_F8
LCD_Enable_out	PIN_E8
KEY[1]	PIN_E1
KEY[0]	PIN_J15
CLOCK_50	PIN_R8
LCD_Contrast (only for red FPGA board)	PIN_E9

Bonus Task: Including of functionalities for a realistic calendar.

For the date function, add that not every month has 31 days, and include days of the week and leap years. The coding of the weekdays will be described in the next experiment.



Experiment 4: DCF77 receiver

The goal of this experiment is to receive a real-world DCF77 signal using the attached antenna on the FPGA board, and decode it with the preceding code. First, a general introduction into the DCF77 signal is presented. Then, your tasks will be explained.

DCF77 time signal. The DCF77 is a coded time signal and is broadcasted on the standard frequency 77.5kHz. The transmission power is about 50kW with a radiated power of about 30kW. Within a range of about 2000km in the area around Frankfurt am Main or more specifically, the transmitter location, Mainflingen, this signal can also be received using extremely simple and inexpensive antennas.

In the DCF77 signal, the time information of the next minute is coded and transmitted via long wave. The 77.5 kHz carrier frequency of DCF77 is a highly stable normal frequency, which can be used for the tracking standard frequency oscillators. The carrier is modulated with second marks, time and date are transmitted in coded form. From the calendar year, however, only the last two digits are sent out. The call sign DCF77 is sent three times an hour, as a Morse code, during the minutes 19, 39 and 59.

Using the time information transmitted by DCF77, radio-controlled clocks can be kept for more than one millisecond in accordance with the PTB time. The times of the radio and television stations and the clocks of Deutsche Bahn AG and the time announcement service of Telekom are controlled by DCF77 as many tariff time switches, traffic control devices and traffic lights. In industry and science, process sequences are controlled and monitored by DCF77 radio-controlled clocks. For private use, various radio clock models are commercially available.

Generation of the DCF77 signal and transmitter location. The DCF77 time signal is generated by the PTB using three atomic clocks in the transmitter building (Mainflingen). The carrier frequency of the transmission (77.5kHz) is derived from these atomic clocks, so that the relative deviation of the carrier frequency from the nominal value is on average over one day at 10⁻¹².

The transmission time is, with small exceptions, a 24h continuous operation. Only for the duration of thunderstorms, the transmission of the signal must be temporarily switched off. In summer months, this can very often be the reason for interference in the reception of the signal. For maintenance or malfunctions, it is necessary to switch to a reserve transmitter or reserve antenna. Depending on the type of work, the interruption may then take several hours. However, since nowadays three independently operating control units generate the signal, the transmission of the signal during repairs can usually be continued with the help of reserve antennas.

For safety reasons, there are three control units which generate the DCF77 signal independently of one another. Before the signal is released for transmission, a check of the three signals takes place against each other. As soon as the signal from the main control unit deviates from one of the other two "reserve" control units, the output is determined by the reserve units. For safety reasons, the signal transmission is totally interrupted when the signal is completely "false", i.e. no match of the three signals generated is observed.

Reliability of the DCF77 signal. As mentioned above, the three signals are checked against each other and only then broadcasted. There are also measures against a power failure: A UPS (Uninterruptible power supply) and an emergency generator also maintain the operation in the broadcasting center in Mainflingen upright. The DCF77 signal has extremely high reliability and temporal availability. According to the law on time determination, 99.7% of the time availability is contractually agreed; in 2002, almost 99.95% was achieved for this service.

Coding. The bit information is embedded amplitude-modulated in the highly stable normal carrier frequency 77.5 kHz of the DCF77 signal. One bit is modulated by lowering the carrier wave to 25% amplitude. During each one second, a so-called second mark is sent, with a 100ms long reduction corresponding to a 0 bit and a 200ms long reduction to a 1 bit. Every second mark that is sent over the course of a minute has a specific meaning: seven bits encode the minute, six the hour, three the day of the week, and so on.

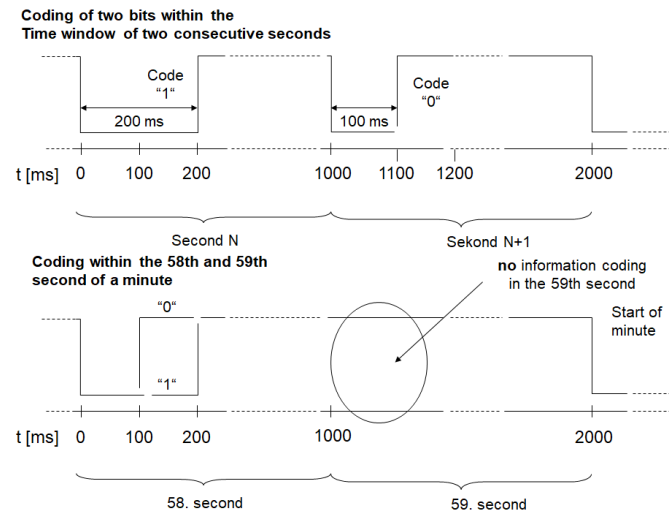


Figure 14: Time Coding of DCF77 Signal.

Within one minute 59 bits are transmitted, coded to contain the information of the following minute. The transmission takes place synchronously to the second rhythm, whereby an exact second beginning is specified. The absence of the last (59th) second within one minute indicates that a new minute begins with the next superfluous second. The transmitted signal has (after digitization) the time waveform shown in Figure 14. Table 4 shows the meaning of the individual bits in the DCF77 coding scheme and shows an example of the encoding of a date. The days of the week are numbered from Monday to Sunday from one to seven.

Table 4: Meaning of individual bits of DCF 77 signal.

Sec.	Bit	Value	Meaning	Example	
0	0		Minute start	0	
1	0/1		Started at Coding as needed, e.g. operating information; in general, '0' is sent	0	
.				.	
.				.	
.				.	
14	0/1			0	
15	0/1		1: Reserve antenna on	0	
16	0/1		1: Hourly jump follows (for example, summer time)	0	
17	0/1	2^1	Time zone: deviation from the	1	
18	0/1	2^0	world time in hours (CET = 1, CEST = 2)	0	MESZ
19	0/1		1: leap second follows	0	
20	1		Start of time coding	1	
21	0/1	2^0	Minutes - Low-order digit in decimal representation	1	
22	0/1	2^1	- see above -	0	
23	0/1	2^2	- see above -	1	
24	0/1	2^3	- see above -	0	5
25	0/1	2^0	Minutes - High-order digit in decimal representation	1	

26	0/1	2 ¹	- see above -	0	
27	0/1	2 ²	- see above -	0	1
28	0/1		Parity bit for 21. . . 27 (even)	1	
29	0/1	2 ⁰	Hours - Low-order digit in decimal representation	1	
30	0/1	2 ¹	- see above -	1	
31	0/1	2 ²	- see above -	1	
32	0/1	2 ³	- see above -	0	7
33	0/1	2 ⁰	Hours - Higher-value digit in decimal representation	1	
34	0/1	2 ¹	- see above -	0	1
35	0/1		Parity bit for 29. . . 34 (even)	0	
36	0/1	2 ⁰	Calendar Day - Low-order digit in decimal representation	1	
37	0/1	2 ¹	- see above -	0	
38	0/1	2 ²	- see above -	0	
39	0/1	2 ³	- see above -	1	9
40	0/1	2 ⁰	Calendar Day - Higher digit in decimal representation	1	
41	0/1	2 ¹	- see above -	0	1
42	0/1	2 ⁰	weekday	0	
43	0/1	2 ¹	- see above -	0	
44	0/1	2 ²	- see above -	1	4 (Th)
45	0/1	2 ⁰	Calendar month - Low-order digit in decimal representation	1	
46	0/1	2 ¹	- see above -	1	
47	0/1	2 ²	- see above -	1	
48	0/1	2 ³	- see above -	0	7
49	0/1	2 ⁰	Calendar month - Highest digit in decimal representation	0	0
50	0/1	2 ⁰	Calendar year - Low-order digit in decimal representation	0	
51	0/1	2 ¹	- see above -	0	
52	0/1	2 ²	- see above -	0	
53	0/1	2 ³	- see above -	1	8
54	0/1	2 ⁰	Calendar year - Highest digit in decimal representation	1	1
55	0/1	2 ¹	- see above -	0	
56	0/1	2 ²	- see above -	0	
57	0/1	2 ³	- see above -	0	0
58	0/1		Parity bit for 36. . . 57 (even)	1	
59	-		Marker is missing, neither 0 nor 1 is sent		

Experiment instructions. In this experiment, the clockwork created in experiment 3 will be expanded to include a DCF77 receiver. To do this, copy the project from experiment 3 and the files from experiment 4 template into a new directory. Add the new files to the Quartus project. Please note that you are only supposed to modify *dcf77_decoder.v*. It specifies an empty module for the DCF77 decoder.

TASK 4.1: Creating a DCF 77 decoder.

First, implement a counter that is zeroed one second after the minute start signal (*minute_start_in*) and incremented based on the synchronous enable signal (*clk_en_1hz*). The minute signal is generated by the *genClockDCF* unit, which detects the beginning of a minute based on the separate encoding of the 59th second in the DCF signal. The periodic, asynchronous enable signal is independent of the DCF signal and can be used as part of the clock in the context of the clockwork. Your task here is to decode the dcf signal (*dcf_Signal_in*) based on Table 4 and form the output signal *timeAndDate_out*.

When designing the DCF decoder, you can simulate the design at any time with the test vectors in the *tb_dcf77signal.v* file. For this, add the testbench file, your dcf decoder file and *GenClockDCF.v* to your ModelSim project. You should simulate around 140000ps. (Note: 1ns in simulation represents 1s in real time – 140000ps will therefore simulate 140s). Check your design after small additions to detect any errors early.

TASK 4.2: Including a parity check.

The next functionality to add is the parity bit checks. For each of the three parity bits (see Table 4), generate an error signal that takes the value '1' if a wrong signal was received.

Now generate the module's valid signal (*data_valid*). The signal shall assume the value '1' for one second during the first second of the DCF77 signal if the signal was received correctly. This is the case if all 59 values were received, and no parity check was erroneous.

Verify the correct function of the receiver based on the given test vectors. It should be received in the first minute of the test vectors - the time is Thursday 19.07.18 17:15 (summer time). In the second minute, no valid DCF77 signal is received.

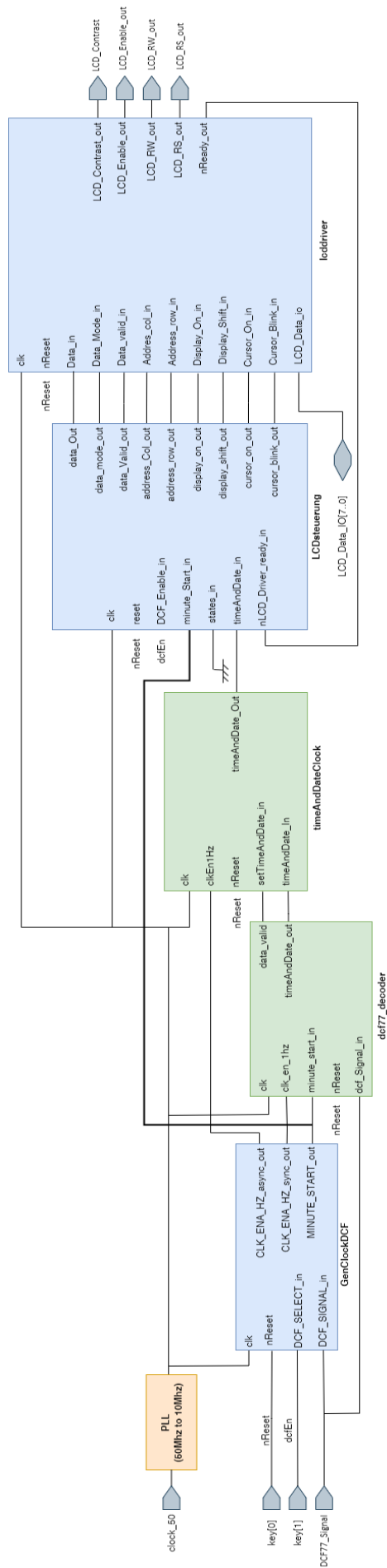
TASK 4.3: Creating the complete clockwork including the DCF decoder.

Integrate the created component into the existing system. To do this, replace the component *setFixTimeAndDate* with the module *dcf77_decoder*. Since *GenClockDCF* is generating the necessary 1Hz clock there is no need for *clkGen* and it can be removed. The pin assignments from the previous task, given in Table 3, remain valid, but additional pins have to be assigned, see Table 5. Note that the key assignments (KEY0 and KEY1) stay the same, but their functionality should change. While one key should still be used for resetting, the other should drive the *DCF_Select_In* signal – while the key is pressed, the DCF77 signal will not be used for setting the clock. Please make sure that you update incoming signals to *LCDsteuerung*. If necessary, you may update your *timeAndDateClock* considering the possible differences in comprehending *setTimeAndDate_in* input.

After assigning the pins, test the design on the FPGA board. The display should show the current time as soon as a complete DCF77 signal has been received (> 1 minute). If more than 3-4 minutes have passed without the display switching to the current time, you have either made a mistake in your implementation or no correct DCF77 signal has been received. You can try placing your development board somewhere else, e.g., closer to a window, to get a clearer signal. To evaluate the source of the problem, you can use the LEDs on the board for debugging, for example by having them show the DCF signal.

Table 5: Pin Assignment for new pins of experiment 4.

Name of pins in the project	Name on FPGA	Description
DCF77_Signal	PIN_N15	DCF77 signal
LED[0]	PIN_A15	Optional, can be used for debugging
LED[1]	PIN_A13	""
LED[2]	PIN_B13	""
LED[3]	PIN_A11	""
LED[4]	PIN_D1	""
LED[5]	PIN_F3	""
LED[6]	PIN_B1	""
LED[7]	PIN_L3	""



Experiment 5: Control Unit of Clockwork

The control unit of the radio-controlled clock should connect and control the various units of the clock (Figure 13). The clock should work as a radio clock, as well as a normal clock. It should be possible to set the clock by hand. The control unit is to be created using a switching mechanism. First, a general introduction to state-machines is presented. Then, the experiment and the task are described.

Introduction to switching mechanisms in Verilog. The switching mechanism consist of a switching network and storage elements (see Figure 15). In the switching network, depending on the input vector X_k and the state vector Z_k (k is a discrete time variable), the state vector Z_{k+1} is determined using the transfer function f ($Z_{k+1} = f(X_k, Z_k)$). The output function g determines from X_k and Z_k the output vector Y_k ($Y_k = g(X_k, Z_k)$). The memory elements serve to store the states of the switching mechanism.

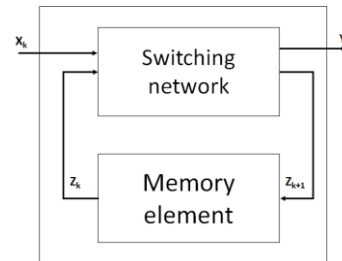


Figure 15: Simple model of a switching mechanism.

In general, switching mechanisms can be divided into static and dynamic state machines, whereby only static machines are important here. These can be subdivided into Mealy and Moore machines, Figure 16. In a Mealy machine, the output vector depends on the state vector and the input vector ($Y_k = g(X_k, Z_k)$), whereas in a Moore machine, the output vector depends only on the state vector.

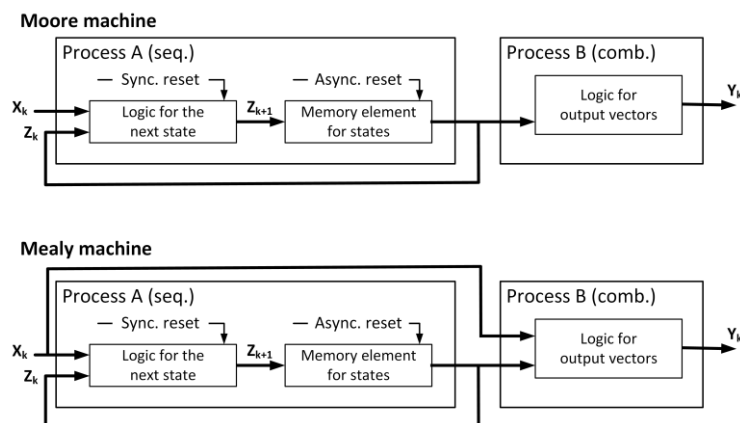


Figure 16: Mealy and Moore machines.

The following code describes Figure 17. The module declaration that belongs to this switch consists of the input input, the output output and the signals clk, reset_syn and reset_asyn. First, the states are defined as localparam.

```
module simpleState (clk, reset_syn, reset_asyn, input, output);
input    clk;
input    reset_syn;
input    reset_asyn;
input    input;
output   output;

localparam state_A = 1'b0;
localparam state_B = 1'b1;
```

1

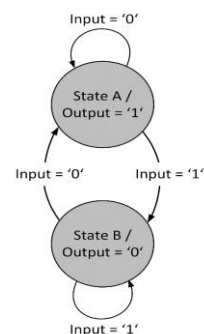


Figure 17: Principal of a state machine.

The process describes the transfer function f and the required registers. In this example, there is an asynchronous (reset_asyn) and a synchronous reset (reset_syn). The asynchronous reset is executed immediately; the synchronous reset does not take effect until the next rising edge of the signal clk.

The combinatorial process B describes the output function g . Since process B, and thus the output function g , depends only on the state of the switching mechanism, the code describes a Moore machine.

```
always @(*)
begin
case(current_state)
state_A:
begin
output <= 1'b1;
end
state_B:
begin
output <= 1'b0;
end
default:
begin
output <= 1'b0
end
endcase
end
```

3

```
reg [1:0] current_state;
if (reset_asyn) begin
current_state <= state_A
end
else if(posedge clk) begin
if (reset_asyn) begin
current_state <= state_A
end
else if begin
case(current_state)
state_A:
begin
if(input == 1)
current_state <= state_B;

else begin
current_state <= state_A;
end
state_B:
begin
if(input == 1)
current_state <= state_B;

else
current_state <= state_A;
end
default:
begin
current_state <= 1'b0;
end
endcase
end
end
```

2

It should be noted for processes that for all signals to which values are assigned in this process, this happens in any case (all possibilities of an IF or CASE expression must be taken into account). For this reason, it is good practice to assign a default value to all signals to be described at the beginning of a combinatorial process. Also an ELSE-part should always follow for an IF-statement, as well as a DEFAULT part for a CASE query.

Experiment instructions. In this experiment, create the control unit of the radio-controlled clock. [5] explains how to create finite state machines in Verilog. To do this, copy the project from the previous task to a new directory. In the experiment 5 template, you will find the files *control_unit.v* and *setClock.v*. You may modify *control_unit* to develop the requested functionality. Copy these files into the new project directory. The signal in the Verilog files have the addition *_out* or *_in* to indicate whether the signal is an output or an input in the module. However, the name before the “_in” or “_out” should be compared with the script. This is helping you to build the control unit.

TASK 5.1: Creating a state-machine to control DCF 77 live decoder and pre-set time.

The output function of the state machine is described by Table 6 and Figure 18. Implement the machine as shown in Figure 18 in Verilog. Use the same names for the states as shown in Figure 18. For the first

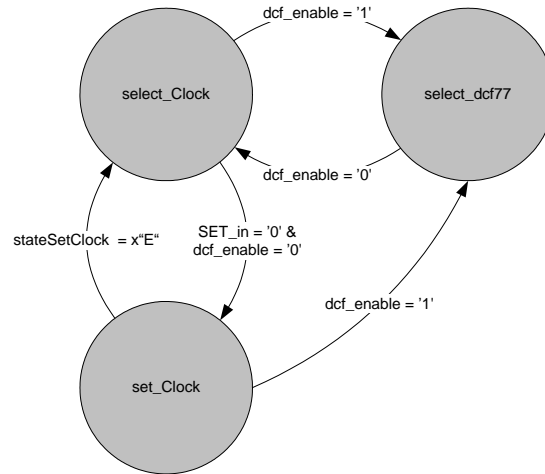


Figure 18: State machine of clockwork with DCF77 receiver.

part of this experiment, neglect the output of the machine. Simulate the state machine. Try to achieve as complete a test coverage as possible by selecting the appropriate test vectors.

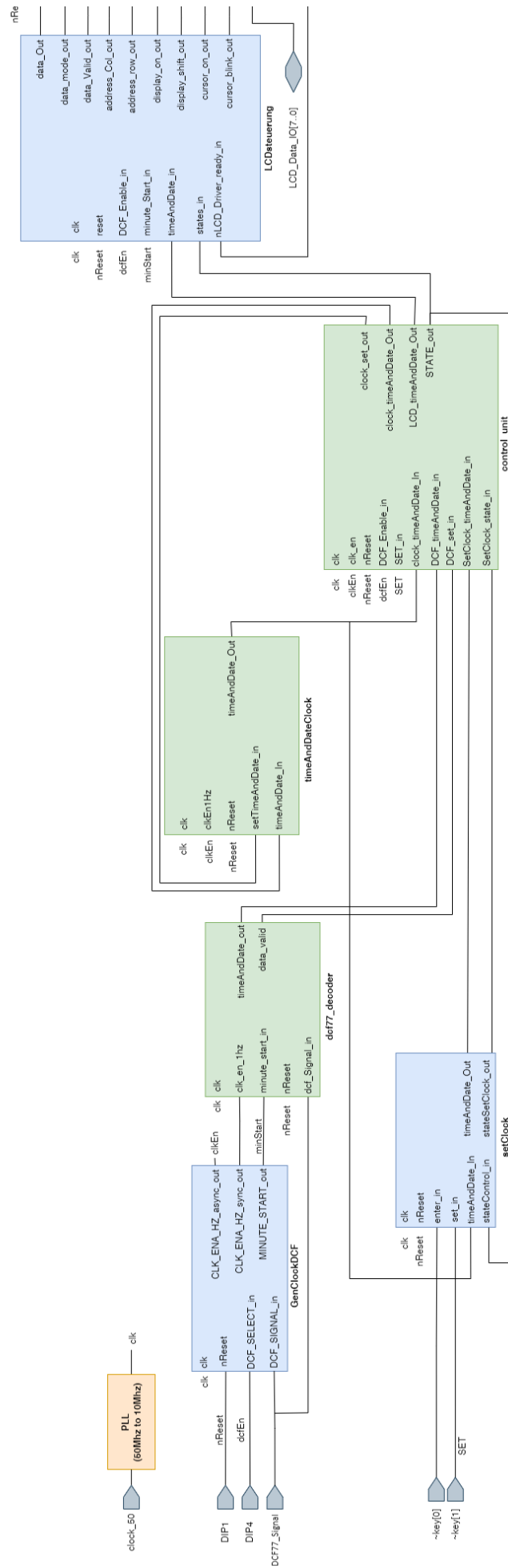
Table 6: Output of switching mechanism (Output signal with an undefined state are supposed to be set on a default value).

State	Output	Value
select_Clock	STATE_out	"000000"
	LCD_timeAndDate_Out	clock_timeAndDate_In
	clock_timeAndDate_Out	SetClock_timeAndDate_in
	clock_set_out	'0'
select_DCF	STATE_out	"110000"
	LCD_timeAndDate_Out	clock_timeAndDate_In
	clock_timeAndDate_Out	DCF_timeAndDate_in
	clock_set_out	DCF_set_in
set_clock	LCD_timeAndDate_Out	SetClock_timeAndDate_in
	clock_timeAndDate_Out	SetClock_timeAndDate_in
	clock_set_out	'1', if stateSetClock = "D" or stateSetClock = "E", else '0'
	State_out (5 downto 4) State_out (3 downto 0)	"10" SetClock_state_in

The next step is to create the output function as described in Table 6. To do this, program a second process (combinatorial). Again, check the functionality with the help of a simulation. Before testing the design on the FPGA board, update the pin assignments according to Table 7. This time, the reset and select functionality have been mapped to two of the DIP switches onboard (check the board datasheet). The previously used buttons KEY0 and KEY1 will now be used for switching into the set_clock state and setting the clock. Now create the entire system so that it corresponds to the schematic overview in Figure 13. Test the design on the FPGA board.

Table 7: Pin Assignment for new pins of experiment 5.

Pins	Value
SELECT	PIN_M15 (DIP4)
nReset	PIN_M1 (DIP1)



Bonus Task: Creating an advanced output control unit.

Extend the radio clock with the alarm function. Start with the actual alarm clock that awakens at a time that is fixed at the synthesis time. Then expand your design with a module to set the alarm clock. Note that both the control unit and the LCD control need to be adjusted. A possible state machine graph for the extended control unit is shown in Figure 19. The functional unit for setting the alarm clock can be derived from the unit for setting the clock. Include a possibility to show a lettering for the alarm function on the LCD matrix.

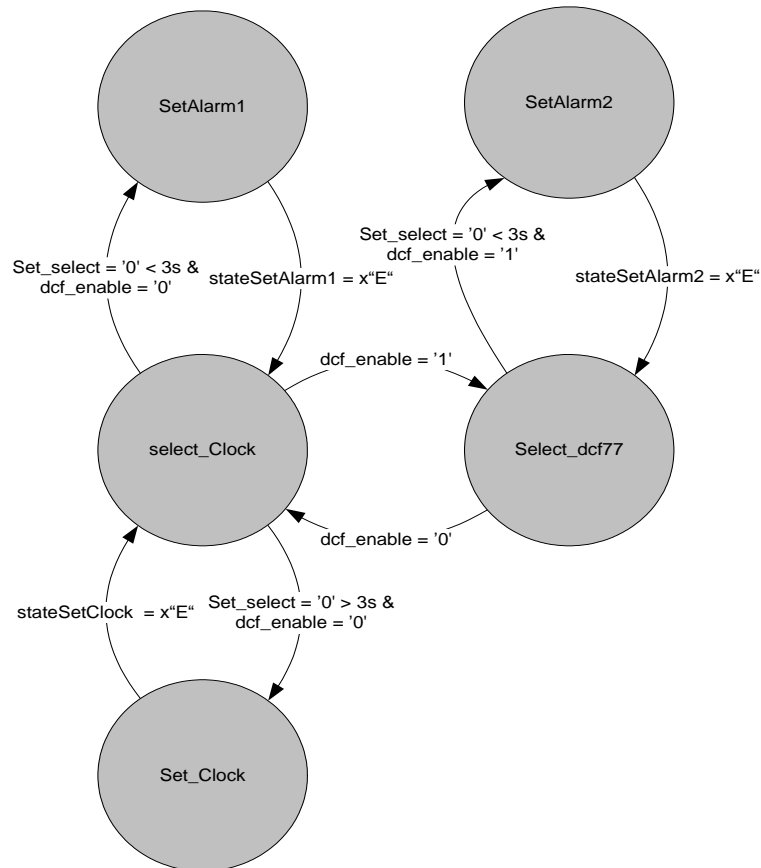


Figure 19: Advanced state machine.

Experiment 6: Image- / video processing

FPGAs are often used in image-/video processing domain to gain acceleration. This exercise presents an embedded system, which displays image/video on the LCD matrix display available on your board (Figure 20). As an example, an implementation of a simple 3x3 Laplacian filter will be practiced. The goal is to understand the underlying hardware architecture and apply this framework to image processing applications.

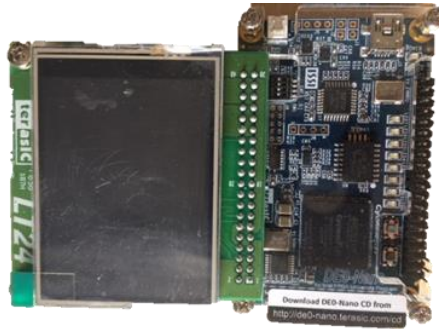


Figure 20: Experiment-6 FPGA-board setup¹.

System and experiment description. Your DEO-Nano development board provides a series of components including storage components such as SDRAM, EEPROM as well as I/O peripherals like LEDs, pushbuttons or GPIO headers. The most important part however is the processing unit. In this case, it is the Altera Cyclone IV E. For more detailed information you can search for the corresponding manuals online.

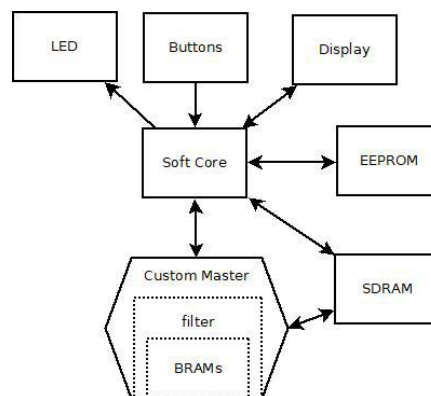


Figure 21: System Architecture.

The general architecture used in this exercise is depicted in the Figure 21. There are 4 main components to run our experiment: the soft microprocessor (Nios II), Display, SDRAM, and the Custom Master.

Nios II manages the different subcomponents. In this case it is used for providing an interface to the attached LT24-display.

SDRAM is used to load or store image or video. It can be accessed using the Read and Write Master (7_Attachment_Avalon_MM_Masters_Readme.pdf, page 5, Figure 1 and 2), and instructed via the “System Interconnect Fabric”, which is solely handled by the Nios II and the Custom Master.

¹ Please make sure that your board looks like this with a display attached at the correct side.

Custom Master Interface (DE0_Nano_SOPC/synthesis/submodules/CI_custom_master.v) provides an interface between Nios II and the Custom Logic (in our example the filter). The Custom Master also instantiates the Read Master and the Write Master to access the SDRAM. This process is realized with a FSM (Figure 22).

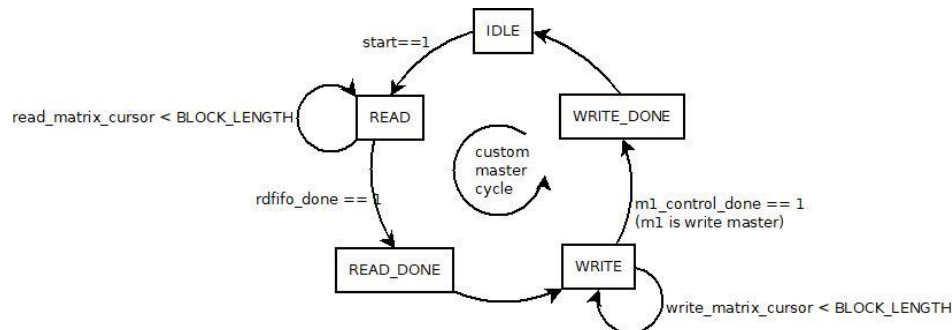


Figure 22: Custom Master Cycle.

During the 'IDLE' state, the Custom Master waits for the 'start' signal. When 'start' equals to '1' (the custom master will automatically receive the start signal from the processor), the state will change to 'READ'. The Read Master passes the amount of BLOCK_LENGTH pixels data to the BRAMs. When the 'READ' state is done, the filter will send the processed (or filtered) data to the Write Master. The Custom Master repeats the step until the whole image is loaded.

Steps to run the experiment on the FPGA. To start the filtering of the image, please follow the following steps:

1. Copy the complete Folder *EX6_Templates* on your Desktop
2. Open Quartus II Project
3. Play *.sof of your compiled project FPGA with **programmer** (do not worry that you have a time-limited *.sof, it should work as good as the other)
4. Click 'windows' + s
5. Open NiosII Command Shell
6. Go now to the .elf file to download the image:

```
$ cd /cygdrive/c/Users/YourGroupAccount/Desktop/EX6_Template/software/LT24
```

7. Enter the command shell: (if you want a quick debugging test use LT24_foto.elf).

```
$ nios2-download -g LT24_foto_video.elf
```

Settings of CI_custom_master.v. In the C- Code, two memory spaces, one for the raw image and one for the filtered image, have been allocated. These are similar in size. Therefore, dataa starts in its allocated space at zero. So would datab. But the filtered image will have one line less data. So, the write address for the filtered image will be datab + 2* 240. The 2 times is due to the 2 bytes per pixel.

Let's have a look in the Custom Master. The starting address of the first pixel in the upper left corner of the three columns to be loaded is dataa. It is incremented in the C-Code, which is loaded on the Nios II by 240 Pixel, as soon as the Write Master sets a done signal and the Custom Master, therefore, sets a done signal. This process will be done 319 times. So, then the next three lines may be read. The cursor data length indicates the amount of pixel, which will be read in one full process cycle: 720.

The start signal (set by the C-Code) marks the start of the complete process. This signal triggers the Read Master. As soon as the Read Master has 720 pixels, it puts the done signal high. The pixels are first read from the SDRAM and then pushed into the FIFO in the Read Master. The

user_buffer_data[n:0] contains the data from the FIFO. The signal user_data_available shows whether the data is valid. These are set by the Read Master. The user_read_buffer is set by the Custom Master if the user_data_available is high. It is important to note, if one does not want to read the data, this signal needs to be low. Otherwise, the data is read, and the read pixel is deleted automatically in the FIFO. When all 720 pixels are read, the control_done signal is set high. This is then triggering the Write Master. The Write Master works as the Read Master, but in the other direction. The user_write_buffer is only high if the FIFO of the Write Master is not full. When all 240 pixels were written, it puts its control_go high. This triggers the C program, and another iteration starts.

The filter gets the sequentially read data from the Read Master, processes it, and hands it back to the Write Master. The signal wren indicates with high (= 1) the Read Master is on and with low (= 0) the Write Master is on. The signal d_rdy is used to indicate if the output data from the filter is valid. If d_rdy is low, the filter is not ready. As d_rdy equals 0, write_buffer equals 0. So, the Write Master needs to halt. This can be useful, if the processing of the input data needs more than one clock cycle. If d_rdy is set high, the output data is valid. In the case of the Read Master: the signal mem_cursor gives the relative address to the first pixel of the current 720 input pixel. It is incremented from 0 to 719. In the case of the Write Master: the signal mem_cursor gives the relative address to the first pixel of the current 240 output pixel. It is incremented from 0 to 239. The input data is the raw input data and the output data is the filtered data.

TASK 6.1: Filtering the input image with a Laplacian Filter.

Complete the file “\DE0_Nano_SOPC\synthesis\submodules\filter_3x3_720px.v” to process the data using a 3x3 Laplacian filter. In our experiment the raw image/video data is 720x240 (Figure 23) in RGB565 format. The current setup of the Custom Master provides an interface to fetch (or read) 3 columns of the image from the Read Master (240x3=720 pixels). The 3x3 filter will be performed on these 3 columns. Then, as the filtered or processed data, only 1 column (240 pixels) is sent to the Write Master. This line of image is the center pixels filtered by the 3x3 filter (Figure 25). For example, during the 1st Custom Master cycle, column 1, 2 and 3 are fetched from the Read Master (READ state) and the processed data (column 2) is sent to the Write Master (WRITE state). In the 2nd Custom Master cycle, column 2, 3 and 4 are fetched and the processed column 3 of the image is sent back. This Custom Master cycle runs 718 times until the whole image is processed. Figure 24 summarizes the whole processing procedure.

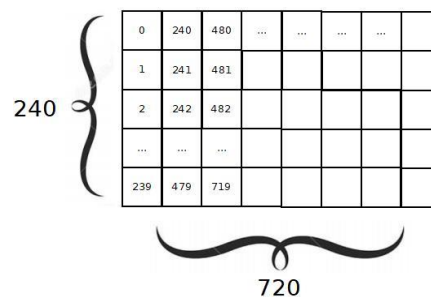


Figure 23: Image resolution.

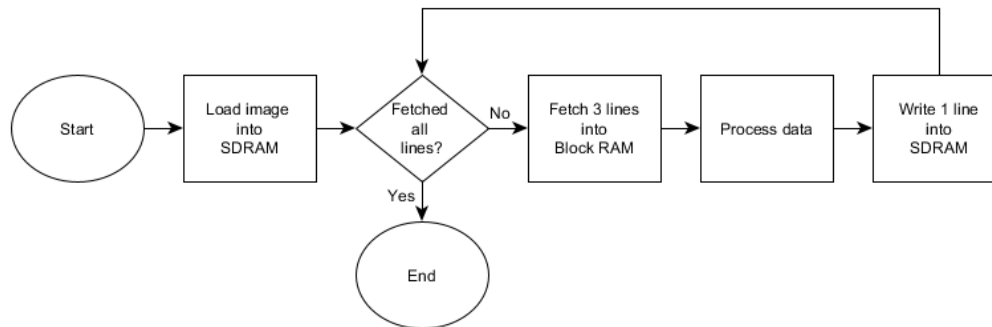


Figure 24: Processing procedure.

The Laplacian filter is popular for edge detection. The values for 3x3 are given in Figure 25. The size of the Laplacian filter is a 3x3 matrix (left). It is convoluted with the input image. The filter moves over the input image, to compute all output pixels. Each pixel is the sum of the element-wise products of the convolution of the filter with the equally sized input pixel fraction. The output matrix has the size of a 2x2 matrix for the example illustrated in the figure.

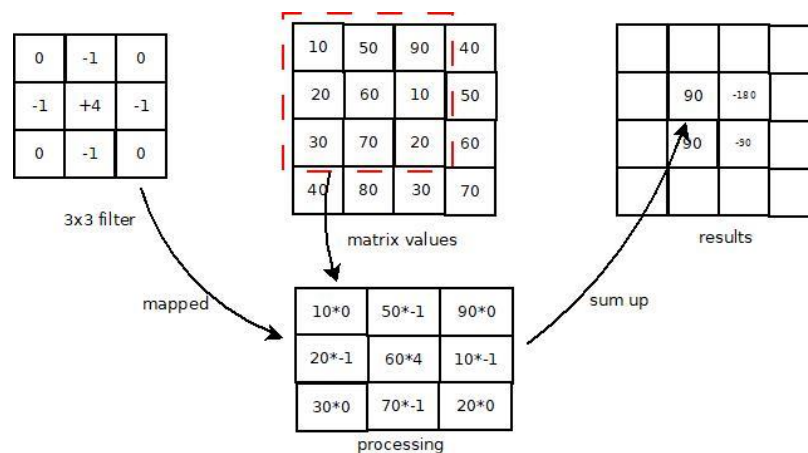


Figure 25: Process of Laplacian Filter.

In the templates, the BRAM can be read on positive edge of the clock when 'wren' is '0' (please note that the latency of the BRAM is 3 clock cycles). On each clock only one word can be read but in order to process a single pixel, the information of the 8 other neighboring pixels is required. Therefore, 9 BRAMs (ram11, ram12, ram13, ram21, ram22, ram23, ram31, ram32, and ram33) are defined to have access to all required information on each cycle.

This template does not provide an efficient design to do image-/video processing. It requires a lot of clock cycle to process an image, this is due to the redundancy of reading two old columns again and only one new column image in each Custom Master cycle. Hence, three columns are read, and one column is written in each Custom Master cycle.

TASK 6.2: Creating a 3x3 filter by reading 240 pixels and writing 240 pixels in each Custom Master cycle.

The given solution has a lot of redundancy. Since data loading from SDRAM needs much more energy than from the on-chip buffers, this design is very inefficient.

Your task is to modify the filter instance (*filter_3x3_240px.v*) to store the incoming data in an appropriate manner and provide a filtered output. The goal is to read 240 pixel and produce 240 filtered data. You can use Figure 26 as a reference for this new dataflow. Here, the colors do not

correspond to the RGB image. The colors are just used to highlight which pixels are processed at the same clock cycle. Each group of three RAMs store the same data, which are pixel 0 to 239, 240 to 479 and 480 to 719. During the 1st clock cycle, the pixels 0, 1, 2, 240, 241, 242, 480, 481 and 482 are read and processed (green color). During the 2nd clock cycle, pixels 1, 2, 3, 241, 242, 243, 481, 482 and 483 are required (red color).

Note:

- By commenting the line `“define USE_3_COLS”` in the module `“CI_custom_master.v”`, the setup will be changed to read and write 240 pixels in each custom master cycle.
- Take care that you implement any changes in the files in the folder: `DE0_Nano_SOPC\synthesis\submodules`
- Also take care to change the correct filter file into the custom master, so it uses the file you want to.

An example of how a RAM is used in the filter is provided in `“\DE0_Nano_SOPC\synthesis\submodules\filter_1RAM_delay.v”`.

You can use nine RAMs to store the data, which is shown below. Please modify the file: `“\DE0_Nano_SOPC\synthesis\submodules\filter_3x3_240px.v”`.

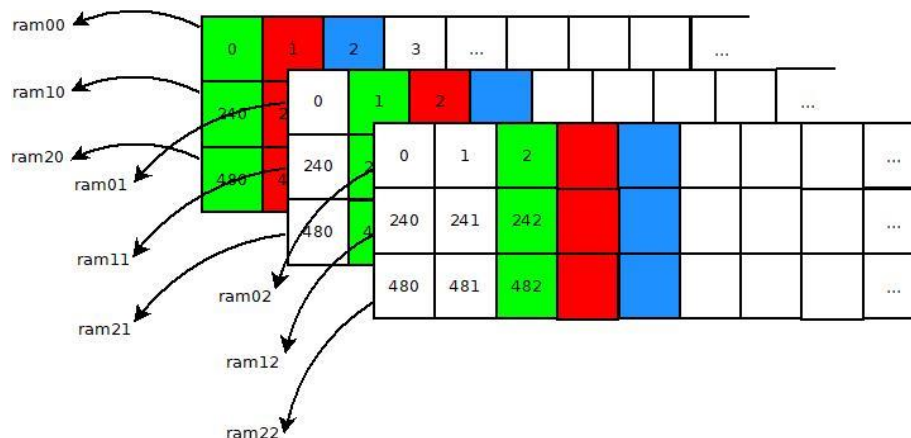


Figure 26: RAM distribution in the filter.

TASK 6.3: Comparing the performance and FPGA resources of `filter_3x3_720px.v` and `filter_3x3_240px.v`.

The question is, how much FPGA resources e.g., RAM do you need and how fast do you process one picture, so what are the frames/s? Please provide a table with your solutions for the supervisor. These questions are very interesting to evaluate any FPGA designs in the real world. You can use SignalTap2 in Quartus to record the signal. For a detailed introduction into SignalTap you can refer to the [user guide](#) and the [video](#) tutorial.

- Choose Clock 50MHz.
- Choose a proper sample depth → trade-off between information and FPGA memory.
- Choose a fitting trigger position, e.g., in CI Custom Master.
- Choose the fitting trigger signal for your analysis, use the presynthesis signal, because postsynthese can change the names of the signals or directly remove some signals.
- Choose a proper pattern.

Do not record master_read and master_write, because these are the main data paths and can significantly increase the needed memory on the FPGA.

Bonus Task: Create a 5x5 Gauß Filter.

How does the performance change for implementation 1.) and 2.) for a 5*5 Filter (Figure 27). Please modify the file: “\DE0_Nano_SOPC\synthesis\submodules\filter_5x5_720px.v”.

Please provide a table with your frames/s and resources.

	1	4	7	4	1
	4	16	26	16	4
$\frac{1}{273}$	7	26	41	26	7
	4	16	26	16	4
	1	4	7	4	1

Figure 27: Gauß Filter 5x5.

Literature

- [1] S. Sutherland and S. Hdl, *Verilog-2001 Quick Reference Guide*. 2001.
- [2] C. E. Cummings, "Nonblocking Assignments in Verilog Synthesis , Coding Styles That Kill!," 2000.
- [3] M. Soc, "Verilog HDL Coding," pp. 1–44.
- [4] "FPGA PROTOTYPING BY VERILOG EXAMPLES."
- [5] C. Science, "EECS150 : Finite State Machines in Verilog The FSM in Verilog," pp. 1–17.
- [6] C. Page, "03 IN / OUT CONTENT Cover Page , Placement , TOP," pp. 1–14, 2012.
- [7] www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/de0_nano_user_manual_v1.9.pdf.
- [8] A. Corporation, "1 . Cyclone IV Device Datasheet," vol. 3, no. July, pp. 1–44, 2010.
- [9] T. F. T. Lcd and S. Chip, "ILI9341," no. 38.
- [10] T. I. Park, "Thin-Film-Transistor LCD Module Model : GWTQ24NPDL1R0," no. 18, 2013.
- [11] Cheatsheet, "Summary of Synthesisable Verilog 2001," *Simulation*, pp. 0–1.
- [12] M. G. Corporation, "ModelSim ® Tutorial," 2012.
- [13] www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_ram.pdf