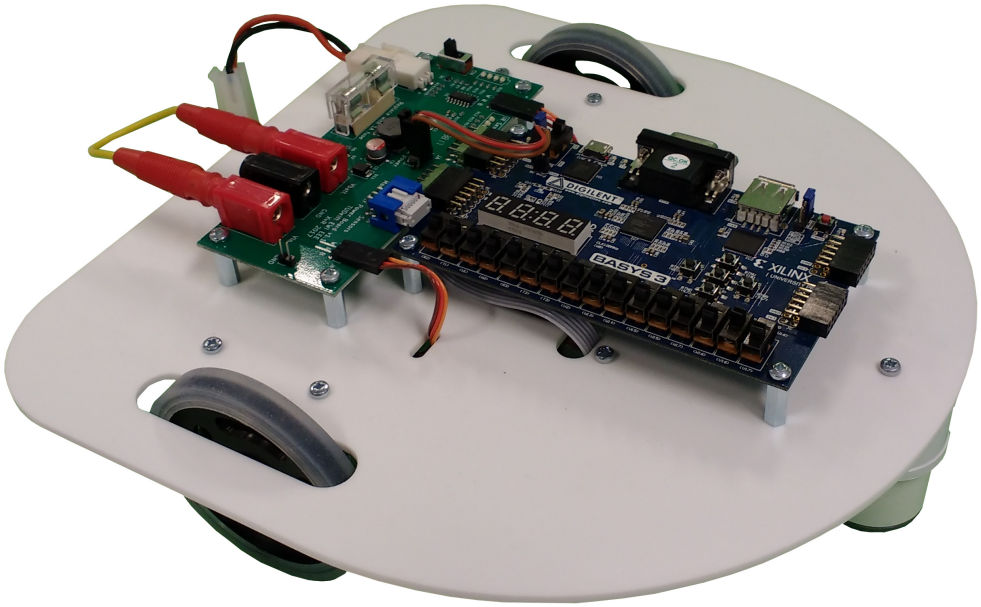
Code EE3130TU

Practical Manual

Mars Rover Project

Delft University of Technology



ing. X. van Rijnsoever, B. Jacobs, ing. T. Slats, Dr. J. Hoekstra

TU Delft Delft University of Technology

Challenge the future

Lab Manual V2.3 (Friday 8th November, 2019)

This manual is typeset with LaTeX in Nimbus Roman 10 pt. The diagrams are created with Dia, the photos and screenshots are edited with GIMP and ImageMagick. The VHDL-code is written in Vim and tested with GHDL and GtkWave and Xilinx Vivado-2017.2.

Preface

Welcome!

First of all, welcome to the Mars Rover project! You're currently reading the lab manual of the Mars Rover project. The manual will guide you through the design steps requried to design a complex system. The method used is the so-called top-down approach: we will initially focus on the complete system as a whole and then fill in the details of each required module. Each block you create, whether it is a VHDL module or an analog circuit, must be separately simulated and tested, before it is used in the larger system.

The manual covers a lot of theory, we have tried to find a good balance between short and concise treatment of this material. We hope you find that we have succeeded, but if not, then founded criticism is always welcome!

Delft, November 2019

X. van Rijnsoever
B. Jacobs
T. Slats
J. Hoekstra

The Lab

In this lab you will work on a mobile robot (the Mars Rover) in a group of two students. The lab consists of two parts:

- Autonomous Robot
- Energy Management

Autonomous Robot

In the first part of the lab you will design the control of a mobile robot (the Mars Rover). This robot has to be able to search for a line and then track the line. Some additional intelligence has to be added as well. Although there are no lines to track on Mars, the line tracking sensors could be replaced with e.g., a wireless communication system between the Mars Rover and a satellite.

The design steps are not only useful for this lab, but also for complex systems in general. First you focus on the problem as a whole, and then the problem is divided into ever smaller parts which are then implemented. This is the so-called top-down

approach. In the design you have to deal with the readout of input signals (sensors), the operation of actuators (the motors) and the stucture of a controller.

The robot is programmed in VHDL. VHDL is a Hardware Description Language (HDL), a language which is used to help design digital circuits. Maybe you'll wonder why not opt for a normal programming language such as C or C++? The reason for choosing VHDL is that we want to teach you the design of digital systems. This is only possible with an HDL such as VHDL.

In order to test the Mars Rover, a complex track is available. Your Mars Rover first has to find the start line to the track and then find the fastest way out. As an added challenge, the track contains instructions for the Mars Rover for making sharp turns that allow a faster traversal of the track.

Energy Management

The second part of the lab focusses on the power supply of the Mars Rover. In the first part of the lab, the robot will use batteries as its power supply. If the batteries run out of power, you can easily replace them. However, on Mars this would pose a problem. In order to be able to use the Mars Rover for an extended amount of time, an alternative energy source is needed. On Mars it is possible to use solar energy to power the Mars Rover. In the second part of the lab you will examine a solar cell and design additional circuits in order to use the solar energy as the power supply of the Mars Rover.

Note that some of the components of the power supply chain can potentially be dangerous. For some steps it is required to have your circuits checked by one of the supervisors before they are powered on. Failure to do so can have you expelled from the lab!

The Manual

The first part of the manual, Autonomous Robot, consists of Chapters 1 through 10. Chapter 1 gives an overview of the Mars Rover and tests the different hardware components. Chapter 2 introduces the software tools required for this part of the lab. Chapters 1 through 3 are written to be finished in a single lab session each. Chapters 4 through 9 must be completed in sequence, but according to your own timetable. The instructions up to and including Chapter 7 are required to complete the lab with a sufficient grade. The Chapters 8 and 9 improve the performance of the robot. Finally, Chapter 10 contains a number of optional refinements to the controller.

Each chapter includes a brief overview of the goals of that session followed by some theory on the subject. You should read this material in advance to start the lab well-prepared. The lab is much more interesting and fun if you understand what you are doing!

The theory is followed by an assignment in which you work on a part of the robot. For some assignments intermediate steps have to be checked by the supervisors. Sometimes this is because incorrect functioning of a component can damage equipment or parts of the robot, sometimes it's because the answer is extended later on and an error in an earlier section can cause annoying problems. So make sure that the intermediate results are checked!

The second part of the manual, Energy Management, consists of Chapters 11 through 13. Chapter 11 gives an overview of the power supply chain. You will also test some

of the components. In Chapter 12 you will work on the switch that is used to select one of the available supply sources. Chapter 13 focusses on the problem of matching the power supplies to the loads.

You will find a number of different frames throughout the manual. These frames contain additional information not directly necessary for the lab, but it may be useful to know.

Warning!
This framework warns against common errors and problems.

Tip!
This framework provides a tip that can be useful in addressing a problem of assignments.

Background information!
This framework provides background information on such as practical applications, or additional functionality that is not necessary for this exercise, but perhaps could be applicable to other labs.

Software

During the lab you have to use several different software packages. The lab PCs run Windows and Linux, both systems can be used for this lab. Sometimes there are differences in the software on these operating systems, this is indicated with the following frames:

Windows
This frame contains information only of importance when using Windows.

Linux
This frame contains information only of importance when using Linux.

Grading

The lab is done in groups of two students and graded with a group grade according to the following table:

Requirement	Topics	Grade
Chapter 1 – 7	Line tracker, Line finder	6
Chapter 8 – 9	Turn signals left and right	7
Chapter 10 – 11	Stop at finish, Solar power component testing	8
Chapter 12	Power supply switch	9
Chapter 13	Power converters	10

Questions, remarks

Questions or remarks can be addressed to:

Dr. J. Hoekstra
Address: Mekelweg 4, room LB01.250
Phone: 015-2783836
Email: J.Hoekstra@TUDelft.nl
or
ing. X. van Rijnsoever
Address: Mekelweg 4, room LB01.260
Phone: 015-2785743
Email: X.vanRijnsoever@TUDelft.nl

Contents

Part I

Autonomous Robot

Chapter 1

Introduction of the Project and Testing of the Components

Objectives

The learning objectives for this session are:

- to know the overall structure of the project
- to understand the operation of the sensors
- to understand the operation of the driving motors
- to know the components that control the robot

1.1 Introduction

Welcome again to the Mars Rover project! This first chapter is intended as an overview. After this chapter you should have a pretty good idea of the final outcome of the first part of the lab. You will also test the hardware to better understand the components that make up the robot. The only thing you will not work on during this session is the FPGA board. That is the topic of the next chapter.

1.2 Overview of the *Autonomous Robot* Part of the Lab

The lab is done in groups of two students and runs throughout the second quarter. In the first part of the lab, Autonomous Robot, you will design the control of a robot. In chapters 1 – 6 you will design the control of a line following robot. A complex track is used to test the robot (see Appendix D). Upon completion of these chapters the robot should be able to:

1. find the track line when placed outside of the track
2. find the exit by tracking the line

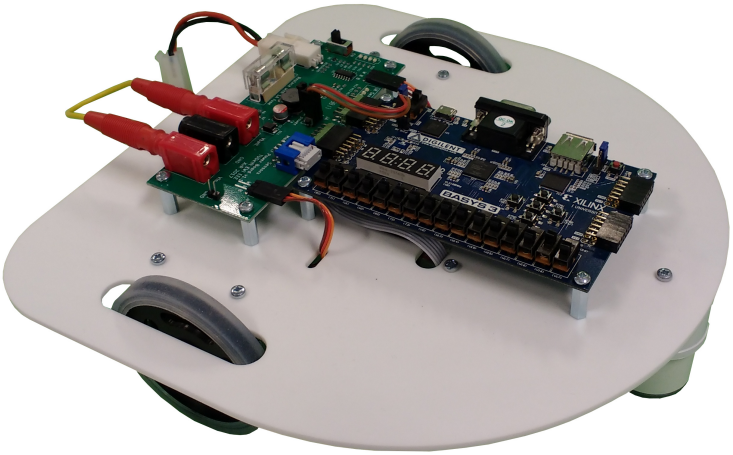


Figure 1.1: Photo of the Mars Rover robot

In the remaining chapters of the Autonomous Robot part you will further enhance the control of the robot. After completing these chapters the robot should be able to more quickly traverse the track by using turn-signals.

1.3 The Robot

For the assignments of the Autonomous Robot part of the lab you need:

- 1 $\times$ robot
- 1 $\times$ micro-USB cable for programming

This chapter discusses the robot as a whole and focusses in more detail on the functioning of the the motors and the sensors. The FPGA board is discussed in the next chapter.

1.3.1 Using the Robot

A photo of the robot is shown in Figure 1.1. The robot is powered by a 7.2 V rechargable NiMH battery. This battery has to be mounted on the bottom side of the robot. There are two pieces of velcro placed onto the battery. These will connect to two mating pieces on the bottom side of the robot. The battery can then further be secured by strapping it in with the piece of hook-and-loop fabric. Figure 1.2 is a photo of a properly mounted battery.

With the battery mounted properly, it can be connected to the robot. The robot consists of two main PCBs (Printed Circuit Board):

- Mars Rover Power Board
- Basys 3 FPGA Board

The Power Board contains circuitry that will convert the 7.2 V battery voltage into 5 V to power the servo motors and the FPGA board. It also contains a battery level indication, an IR sensor indication, and some protection circuitry.

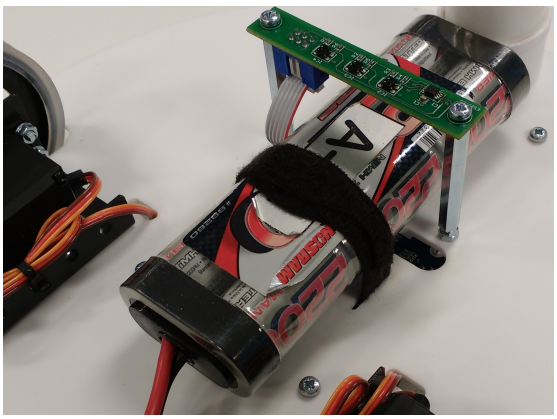


Figure 1.2: Photo of the bottom side of the robot with the battery properly mounted

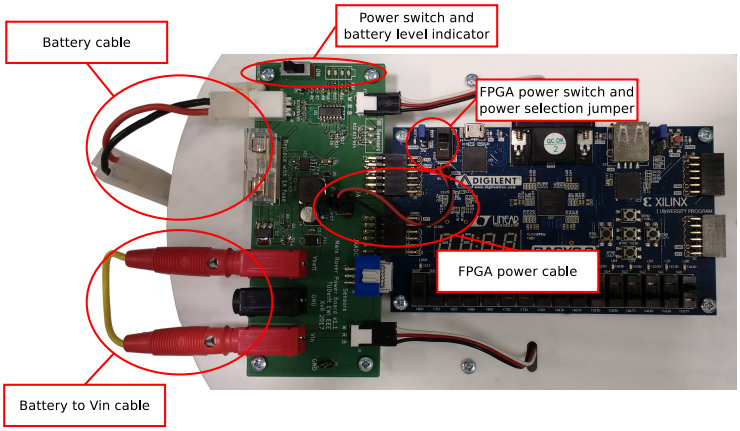


Figure 1.3: Annotated photo of the top side of the robot

Battery Levels
Replace the battery immediately if only a single level indicator led is left.

Figure 1.3 shows an annotated photo of the top side of the robot, indicating some important connectors and switches. Before you can power up the robot, make sure that:

- the two red banana connectors on the Power Board are connected;
- the FPGA power cable is present and connected properly;
- the FPGA power selection jumper `JP2` is in the `EXT` position;
- the FPGA power switch is in the `ON` position.

If all connectors, switches, and jumpers are in the correct posision you can switch on the robot with the `ON/OFF` switch on the Power Board. You should now see the battery indicator as well as the FPGA power-on led light up. If not, please check the list above and also check if the fuse is not blown. Ask an instructor if the fuse is blown or if problems persist.

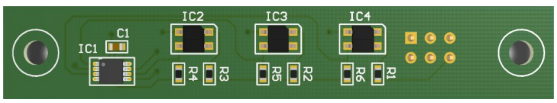


Figure 1.4: Bottom side of the IR sensor showing the three reflective object sensors

1.3.2 Only Using the FPGA Board

When you only need to use the FPGA board, it is easier to use the USB cable for powering the board. To that end you need to modify the jumpers. Make sure that:

- the FPGA power selection jumper `JP2` is in the `USB` position;
- the FPGA power switch `SW6` is in the `ON` position.
- the USB A/micro-B cable is present and connected properly;

Using the board this way is recommended for working through the tutorials in Chapter 2.

1.3.3 Theory of Line-tracking Sensor

The line-tracking sensor is consists of three photosensitive sensors (so-called reflective object sensors) that are placed in a row. The bottom side of the sensor board is shown in Figure 1.4. Each sensor circuit has a digital output that represents the reflectiveness of the surface. With a black background (poor reflection) the sensor circuit has a '0' output, with a white background (good reflection) a '1'. The outputs are connected to the FPGA and to leds on the Power Board.

Each sensor circuit consists of three parts:

- the sensor and peripheral components
- the signal processing
- led output indication

Each sensor consists of an infra-red (IR) led and a (photosensitive) phototransistor. When the sensor is powered the IR led emits light that will be reflected by the surface (a white surface reflects more light than a black). The more light is reflected back, the better the phototransistor conducts.

The signal processing is provided by a so-called inverting Schmitt Trigger [1]. This converts the analog input signal while filtering out noise. It also buffers the output signal which will prevent the leds influencing the output of the sensor. A diagram of a single sensor is shown in Figure 1.5.

In the diagram the intersection point between T1 and R2 is node 1. If no light is reflected onto the phototransistor the resistance of the phototransistor is much greater than R2. In that case there is 5 V on node 1. As more light is reflected, the resistance of the phototransistor decreases, and thus the voltage at node 1. At one point, the resistance of the phototransistor is much smaller than R2. Then there is 0 V on node 1.

The voltage at node 1 is inverted by the Schmitt trigger. In case of a white background (high reflectance) the output voltage of the sensor circuit is 5 V, and the indicator led is on.

[1] `http://en.wikipedia.org/wiki/Schmitt_trigger`

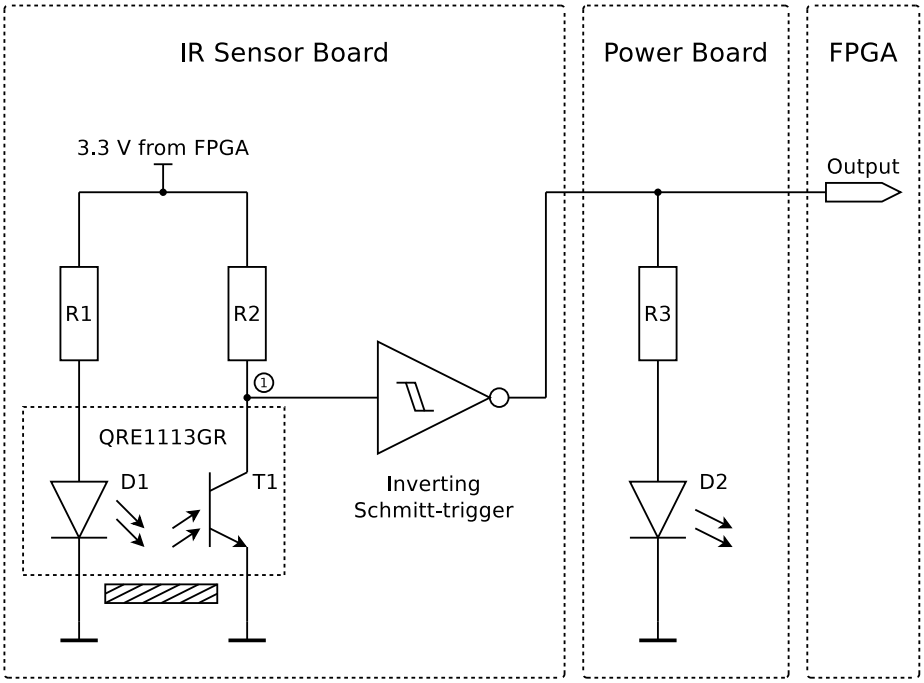


Figure 1.5: Diagram of a sensor

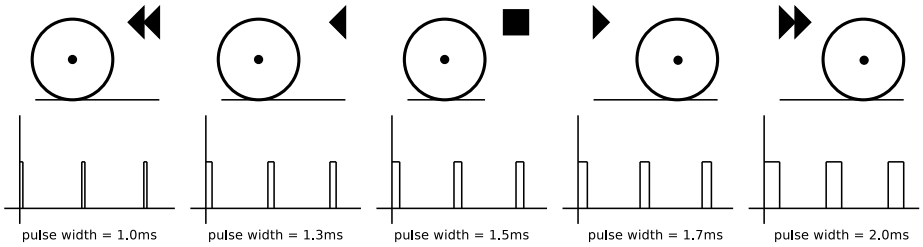


Figure 1.6: Effects of PWM on the servo motor

1.3.4 Theory Motors

Servo motors are available to move the robot. Normal servo motors can only rotate over a limited angular range and are used to operate for example valves or wing flaps of RC planes. The servo motors used in the Mars Rover are special continous rotation servos. Like normal servos, these motors are controlled by pulse width modulation (PWM), but instead of controlling the angular position, the PWM signal controls the rotational speed. PWM is a technique in which the information is encoded in pulses of variable width. The operation in this case is quite simple: every 20 milliseconds the PWM generator sends a pulse. When the pulse is narrower than 1.5 milliseconds, the engine runs counterclockwise; when the pulse is wider than 1.5 milliseconds, the motor turns clockwise. Also the speed can be influenced. If the pulse width is around 1.5 milliseconds, the motor will run slowly or even stop. If the pulse width significantly differs from 1.5 milliseconds, the motor will run faster. This is schematically shown in Figure 1.6.

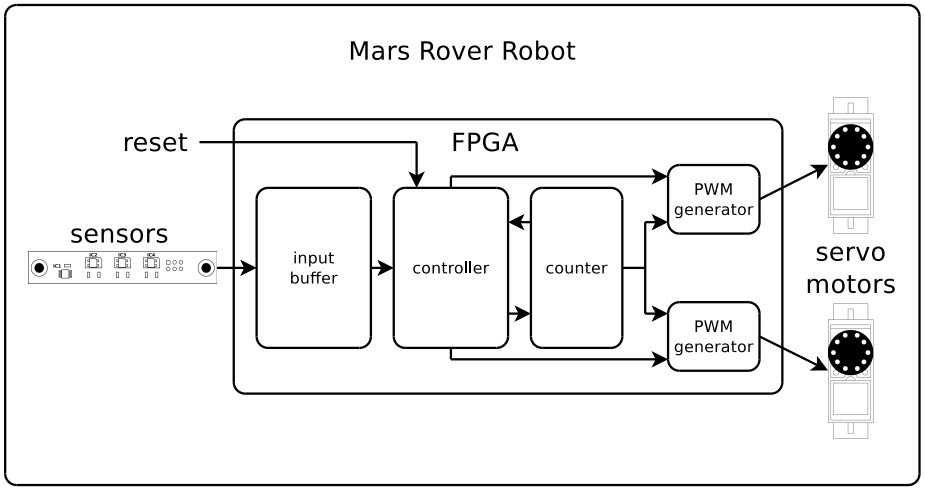


Figure 1.7: Overview of the FPGA modules

1.4 VHDL Modules

The sensors provide input for a number of modules, which in turn control the motors. These modules are the control of the robot and it is your job to design those. There are four VHDL modules, namely:

- the input buffer
- the counter
- the controller
- the PWM generator

The input buffer performs a synchronization step to ensure that the sensor circuit is properly read out. The output values of the sensor circuit are then passed to the controller. Based on these values the controller calculates which direction the robot should go, and directs the PWM generators and the counter. The PWM generators generate the PWM signals to control the motors. The counter is used to provide a timing reference for both the controller and the PWM generators. All this is summarized in the diagram in Figure 1.7.

1.5 Testing the Components

In order to learn the operation of the sensors and the motors, you have to perform some experiments with those components.

1.5.1 Testing the sensors

The input of the robot is provided by three light-sensitive sensors. These sensors and some additional components are already soldered on a PCB (Printed Circuit Board). The sensors should be tested in order to understand their behaviour. The sensor board is shown schematically in Figure 1.8.

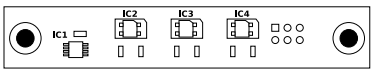


Figure 1.8: Diagram of the sensor board

The sensors are powered directly from the FPGA board, the status of the sensors is shown on the power board. Take the following steps to test the sensors:

1. Select USB power on the FPGA board
2. Use the USB cable to connect the FPGA board to the computer
3. Move the sensor board over the test pattern in Figure 1.9 and measure the effect on each of the individual sensors.

Figure 1.9: Test pattern to test the sensor board

1.5.2 Testing the motors

As explained in the theory, the motors work on the basis of pulse width modulation. To test the motors such a pulse width modulated signal must be presented to the signal input of the servo. We will use the signal generator Tektronix AFG3021 to generate such a signal. In Appendix F you can find a short manual of the signal generator.

1. Switch on the function generator and set the following:
   - Press TOP MENU→OUTPUT MENU→LOAD IMPEDANCE→HIGH Z to specify a high load impedance;
   - Under FUNCTION select wave form PULSE;
   - Use the FREQUENCY/PERIOD button to set a frequency of 50 Hz;
   - Use the AMPLITUDE/HIGH button to set the amplitude to 5 V_{pp};
   - Use the OFFSET/LOW button to set the offset to 2.5 V.
2. Connect the function generator OUTPUT jack with a coaxial cable to the oscilloscope channel 1. Enable the output of the function generator with the ON button above the output connector.

3. Set up the oscilloscope such that the signal is clearly visible and measure the frequency and the amplitude of the signal (using the MEASURE screen). Adjust the function generator if necessary. If you have difficulties setting up the oscilloscope properly, read Appendix E

If the amplitude seems off, make sure you set the output impedance to HIGH Z as described in step 1!

4. See if you can notice a difference between AC and DC coupling on the oscilloscope (CH1 menu). Which setting shows the correct signal? Use that setting.

5. The control signal to the servos must have a duty-cycle (which is the ratio between the time the signal is active and the period [2]) between 5 and 10%. This can be set using the DUTY/WIDTH button next to the PULSE function selection.

6. Set the duty cycle to 5% and verify this on the oscilloscope.

Make sure the signal from the signal generator has the correct properties before you proceed! Offering a wrong signal to the control input of the servo will cause irreparable damage to the servos!
Make sure you do not soley rely on the display of the function generator, but you've measured the output signal on the oscilloscope as well!

7. Switch off the output of the function generator (use the ON button).

8. Disconnect one of the servos from the power board and connect it to a servo tester board. Make sure to match up the colors, the brown wire has to be connected to GND.

9. Power on the Tektronix PWS4205 power supply. Specify a voltage limit of 5 V and a current limit of 1 A. Connect the black and red connectors of the power supply to the GND and 5 V connectors on the servo tester using cables with a banana plug and a clip. Make sure the output is off (use the ON/OFF button). If you have difficulties setting up the power supply properly, read Appendix G.

10. Connect a coaxial cable ending in two small clips with a coaxial T-splitter to the coaxial connector of the signal generator. Connect a coaxial cable from the oscilloscope to the other side of the splitter. This way, you can monitor the shape of the control signal on the oscilloscope. Connect the red terminal of the coaxial cable the CONTROL connector of the servo tester. Connect the black terminal of the cable to the GND connector of the servo tester.

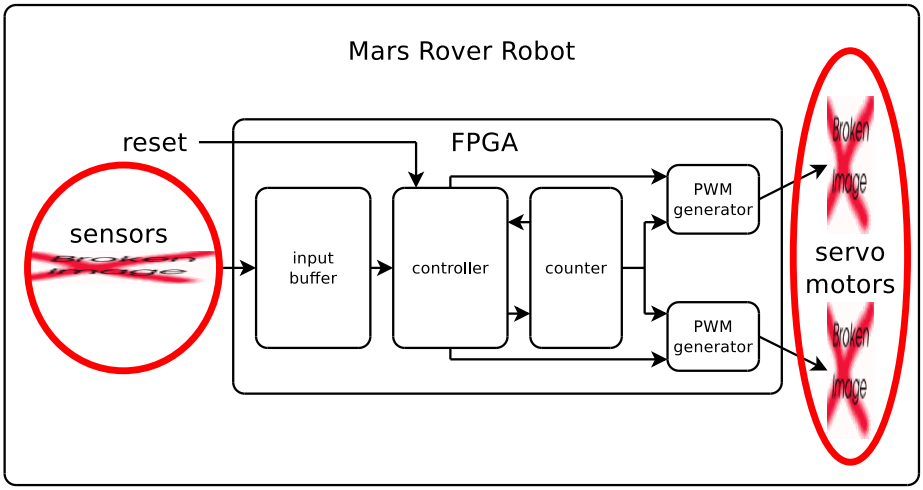
11. Switch the output of the power supply back on, then switch the output of the function generator on and test the effect of varying the duty cycle to the servo motor.

[2] http://en.wikipedia.org/wiki/Duty_cycle

1.6 Overview of the Robot

This session you have familiarized yourself with the components of the robot. By measuring and testing the sensors and the motors, you learned how these components behave. You need this knowledge in the coming weeks to properly control the robot.

At the end of each chapter in which you work on the robot, an overview is given of the different components of the robot. Each session the part you have worked on will be highlighted. In this way, the relationship between the different components and the big picture becomes clear.



Chapter 2

Introduction to the FPGA board and Xilinx Vivado-2017.2

Objectives

During this session you will:

- be introduced to the Basys3 FPGA board
- verify that the FPGA board works well
- familiarize yourself with Xilinx Vivado-2017.2 software

2.1 Introduction

This session you will follow a tutorial on using the FPGA board and the development software: Xilinx Vivado-2017.2. The main goal is to familiarize yourself with the hardware and software. In addition to this, the tutorial is used as a test to verify that all hardware and software works well. In this chapter you have to work with given tutorial code, in the following chapters you will have to write your own code.

First we give a brief overview of the FPGA development board then you will test the software and hardware with the help of two example programs given.

2.2 Hardware: The Basys3 FPGA development board

The Basys3 FPGA development board is a printed circuit board (PCB) designed by Digilent[1] that features the Atrix-7 FPGA by Xilinx[2]. The board features many components that make testing the chip easier. Some are necessary for the operation of the chip, such as components that provide the power, the clock and memory to program the chip. Others are to facilitate the integration of the chip in a system. Think of ways

[1] http://store.digilentinc.com/
[2] https://www.xilinx.com/

of communication (RS232 or CAN), a PS/2 port for a mouse or keyboard, or a VGA connector for a video screen.[3]

A photo of the board is shown in Figure 2.1.

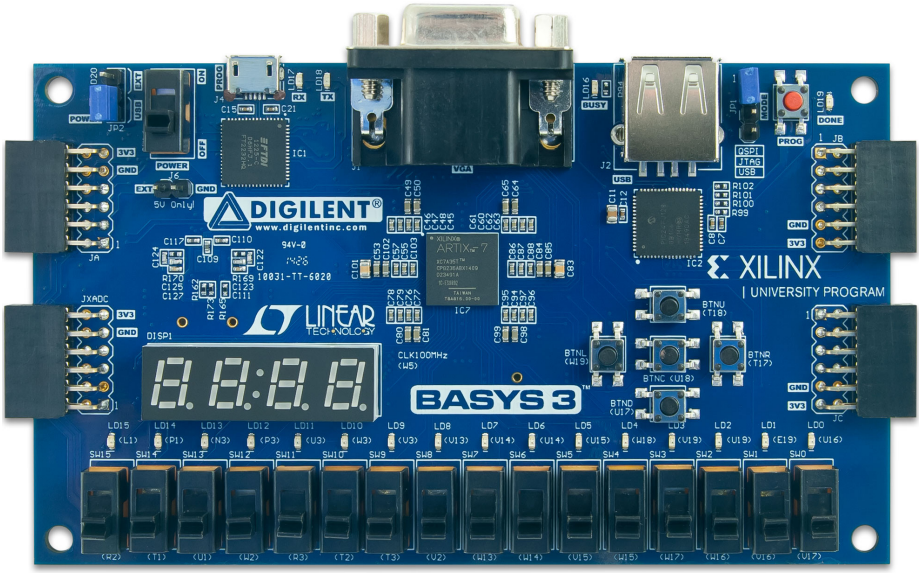


Figure 2.1: Top view of the Basys3 FPGA development board

The large square IC in the middle is the Atrix-7 FPGA. You can also see peripherals like switches, buttons, LEDs, and 7-segment displays. These are simple inputs and outputs you can use to test your design. On both sides of the board you can see the expansion connectors. These connectors allow the FPGA to communicate with the outside world. On the top left you can see the micro-USB connector that is used to program the FPGA. The clock is provided by a 100 MHz crystal mounted on the backside of the board.

2.3 Software: Xilinx Vivado-2017.2

The FPGA is programmed from the Vivado-2017.2 integrated development environment (IDE). You can use this program to write VHDL code, simulate the design and synthesize it so that it can run on the FPGA. The best way to learn how to use this software is by trying it yourself. To that end the remainder of this chapter consists of two examples that you will have to work through. Take the time to properly go through all the steps! During the practical you will need to modify and test your design many times and it is of great help when you're familiar with the software.

[3] Don't worry if the names RS232, CAN or PS/2 don't ring a bell, they're only used here as examples

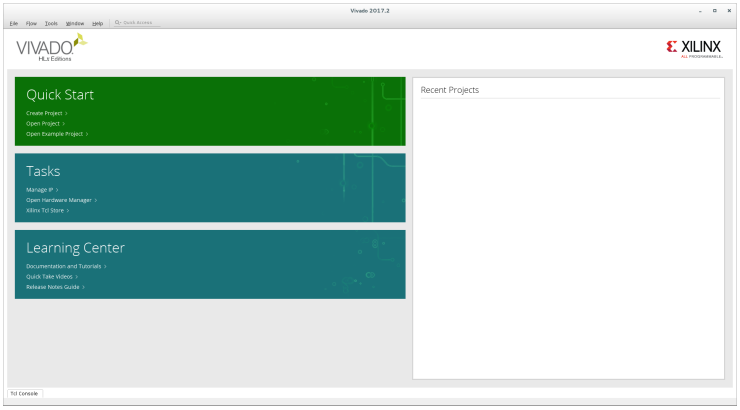


Figure 2.2: Screenshot of Vivado 2017.2

2.3.1 Example 1: Nightrider ft. David Hasselhoff[4]

In the first example you will implement a looping led with the leds on the FPGA board. In a looping led the leds light up one by one, so it looks like an illuminated led running back and forth.

Open Vivado 2017.2 on a PC and make sure that you have downloaded and unpacked the file `nightrider.zip` from Brightspace.

You can find Vivado in the Start menu: Engineering → Vivado-2017.2. Or you can open the start menu and just type in vivado.

You can start Vivado from the terminal with the command vivado-2017.2.

This should open a window that looks like the screenshot of Figure 2.2.

If your window does not look like Figure 2.2 you might have started Vivado HLS 2017.2 instead of Vivado 2017.2!

Create a new project

The first step is creating a new project. This is explained in Appendix B. Follow the steps in the appendix to create the new project with the files from `nightrider.zip`.

[4]Curious students may look up the title of this example here on the Internet, if they do not know where it comes from...

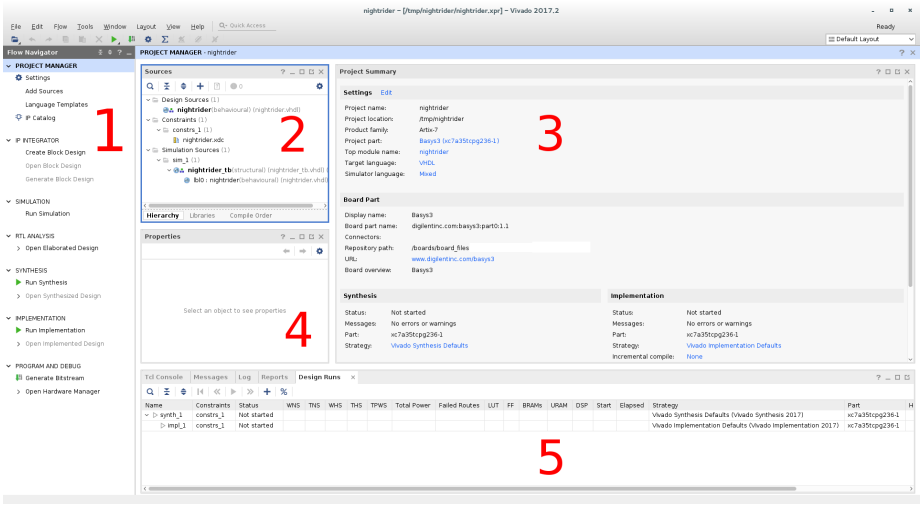


Figure 2.3: Screenshot of Vivado with the opened project

The Vivado user interface

You will now see the Vivado user interface with your newly created project opened. This should look similar to the screenshot of Figure 2.3.

Frame 1 is the Flow Navigator frame. This shows an overview of the steps you can take in your project. This includes simulation, synthesis, and programming of the FPGA. The view and information in all other frames depend on the selection made in the Flow Navigator. The following text describes the default frames from the Project Manager, the manual will further describe the other framesfor other selection when appropriate.

Frame 2 is the Sources frame and shows a hierarchical view of the source files that make up the project.

Frame 3 currently shows the Project Summary. This frame is tabbed and can also show the editor by simple double-clicking a file in the Sources frame.

Frame 4 shows properties of the currently selected source file. You can enable or disable certain files here, specify the VHDL library, and whether or not the file should be included in simulation and/or synthesis.

Frame 5 contains the error logs and messages produced by simulation, synthesis, and implementation.

Tab size
In the files you can download from Brightspace the standard tab size is 8. You have to change the settings of the Vivado editor accordingly in order to view the files with the correct indentation. You can find the tab settings via Tools→Settings and then the dialog via Tool Settings→Text Editor→Tabs.

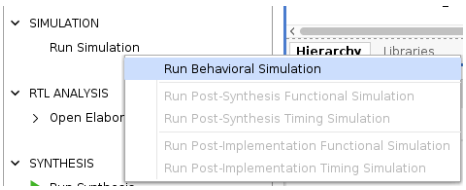


Figure 2.4: Screenshot of the options under Run Simulation

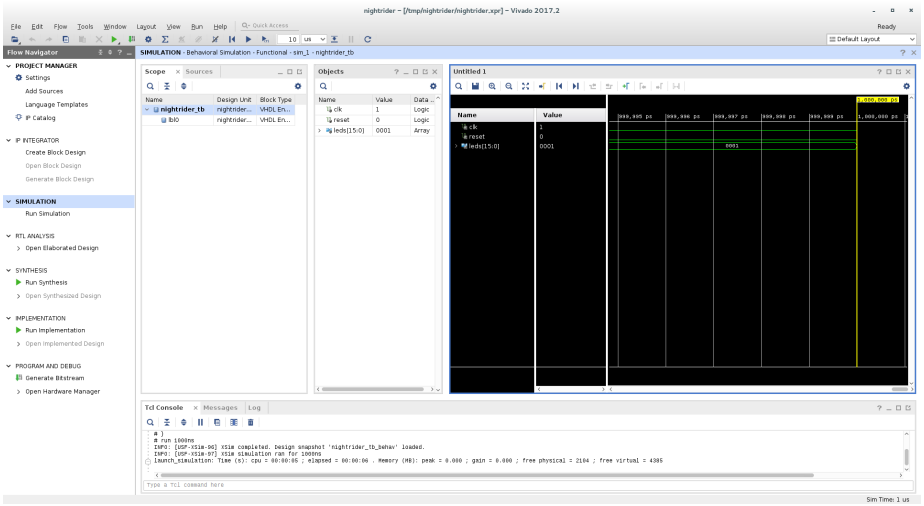


Figure 2.5: Screenshot of the simulator

Make sure that under Design Sources you see `nightrider` as top-level entity (indicated with the stacked boxes icon). If you see `nightrider_tb` as top-level entity, you've added that file for both simulation and synthesis, instead of simulation only (see Appendix B). In that case select the file and in frame 4, Source File Properties, untick the box Synthesis. Vivado will now rebuild the hierarchy and should appoint `nightrider` as the new top-level entity.

Simulating the design

The file `nightrider.vhdl` contains the complete nightrider design. To see whether the design works well, you need to *simulate* it. This is done using a *testbench*. A testbench is a VHDL file that exposes the design to a number of inputs. The simulation shows how the design responds to those inputs. To start the simulation, select Flow Navigator→Simulation→Run Simulation and click on Run Behavioural Simulation (see Figure 2.4). This will open the simulation window and the screen should look similar to the screenshot of Figure 2.5. The bottom frame now shows the Tcl Console. You can type in simulator commands in the console. Run the simulation for 2 s by using the command:

```
run 2000 ms
```

The simulation time for this simulation is large, so it may take a while (around 5 minutes) to complete the simulation. A dialog with a progress indicator will be opened.

Very slow simulation time?
If your simulation is very slow or stops with an error, make sure you created the tutorial project under the directory `C:\Users\<netid>\` (Windows) or under `/data/` (Linux), as indicated in Appendix B.

When the simulation has finished, right-click in the frame with the waveforms and select Full View. Do the simulation results match your expectations?

The `leds[15:0]` signal is a so-called vector: it consists of a set of grouped bits. You can expand the signal by clicking on the small > left of the signal name in the waveform frame.

Pin Assignment

In the simulation you have seen what happens when the entity nightrider is fed with a clock and a reset: in turn the various bits of the signal *leds* are on. The signals *clk* and *reset* are controlled from outside the FPGA: they are the inputs for nightrider. Similarly, the signal *leds* is an output of nightrider.

The software needs to know how to handle the signals *leds*, *clk* and *reset*. This information is provided in a constraints file, in this case `nightrider.xdc`. In the sources frame you should see this file under Constraints. Double-click the file to open it in the text editor.

The constraints file contains a number of setting for the clock and the flash memory and contains all available board peripherals (such as buttons and switches) and connectors in commented lines. The names used in the constraints file can also be found on the Basys3 board. In order to enable a peripheral, the corresponding lines have to be uncommented and the port name (in the `get_ports` function) has to match the names used in the top level entity. In Chapter 3 you will find more information on entities and the port names. The port names for this design are:

- *clk*
  The *clk* signal is already enabled and set up properly in line 7–9 of the constraints file.
- *leds[15..0]*
  The *leds[15..0]* signal is already enabled as well. Find these lines and take note of the way the separate bits of vector *leds* are addressed.
- *reset*
  The *reset* signal has to be connected to a push button. Determine which button on the Basys3 board you want to assign. Next to the buttons you can find the name of the button. These names can be found in the constraints file. Uncomment the corresponding lines and adjust the port name.

Do not use comments *after* a constraints line!
You can only use comments (text preceded by a # character) on a separate line! If you use comments after a constraints line, the Generate Bitstream process will fail. Unfortunately the syntax highlighting in the editor suggests that these comments are OK.

Save the file when you've made your modifications.

Programming the FPGA

With the VHDL design and the constraints file you have all the necessary components to create a programming file for the FPGA. The programming file (bitstream file) can be programmed into the FPGA. There are two options to program the FPGA:

1. loading the design as a .bit file in the volatile memory of the FPGA
2. loading the design as a .bin file in external non-volatile memory

The first option is quite simple: directly load the programming file you created into the memory of the FPGA. However, if the FPGA loses power, the volatile memory loses its information and therefore must be reprogrammed next time you want to use it.

By loading the programming file in an external non-volatile memory, the FPGA is programmed from this memory upon powering up. You can then disconnect your FPGA from the power supply without loosing the programmed information.

Before you can load your design on the FPGA, you have to generate bitstream files. By default, only the .bit file for the FPGA is generated. Enable the .bin file generation via Tools→Settings and then in the dialog Project Settings→Bitstream and tick the option -bin_file.

Now click on Flow Navigator→Program and Debug→Generate Bitstream to generate the .bin and .bit files. This also runs the RTL analysis, synthesis, and implementation steps. During these steps a number of dialogs will popup:

- No Implementation Results Available → press Yes;
- Launch Runs → default settings are OK, just press OK;
- Bitstream Generation Completed → simply press Cancel.

Now that the bitstream files are created, you can program the FPGA and the flash memory. In order to specify the flash device, Vivado has to connect to the Basys3 board. Connect the board using the micro-USB cable, the Power led on the FPGA board should now light up.

No power?
If the power led does not light up, check if jumper `J2` is in the `USB` position and the power switch `SW6` is set to `ON`. See also §1.3.2.

With the board powered up you can open the connection to the board in Vivado:

- Click on Flow Navigator→Open Hardware Manager;
- Click Open Target and select Auto Connect;

The connection should now be established. Now click Flow Navigator→Add Configuration Memory Device→x7a35t_0:

- For Manufacturer select Spansion;
- For Density select 32;
- Now select from the list s25fl032p-spi-x1_x2_x4;
- Click OK to confirm the selection;

- You will now be asked if you want to program the device, press OK. You have to specify the location of the .bin file. This file is placed in:
  `<projectdir>/<projectname>.runs/impl_1/`
  Select the file and program the flash memory.

You can also program the FPGA or reprogram the flash memory:

FPGA Right-click on the FPGA device name (xc7a35t_0) and select Program Device...;

Flash memory Right-click on the flash device (s25fl032p-spi-x1_x2_x4) and select Program Configuration Memory Device....

Cannot program the FPGA?
If you cannot program the FPGA this way, check if jumper `JP1` is in the `QSPI` position.

Interpreting the results

You should now see the output of the program on the Basys3 board. Pressing the reset button will restart the looping led. Do the results of the simulation and the running program match?

2.3.2 Example 2: The Calculator

The second example is a simple hex calculator. The calculator can add two (hexadecimal) digits, subtract, multiply, and shift. In this example the buttons, the switches and the 7-segment displays are tested too. A big difference from the first example is that this example consists of several files.

Operation of the Calculator

The slide switches (`SW0-SW7`) are used to input data (in binary) to the calculator, the first four switches set the first digit, the last four switches set the second digit. The input numbers are displayed on the two 7-segment displays on the left. Pushing down a button will perform an arithmetic function. The buttons perform the following functions:

`BTNU` addition

`BTND` subtraction

`BTNL` multiplication

`BTNR` division/modulo

Button `BTNC` is used to reset the calculator. The division/modulo operation calculates the quotient and the remainder of the division of the first digit by the second digit.

While pressing a button the result is displayed on the right two 7-segment displays. If an invalid button combination is used, the display only shows the input numbers. The numbers are displayed in hexadecimal, you can use Table 2.1 to convert the numbers.

Table 2.1: Hexadecimal to decimal conversion table (eg: 4E = 78)

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Structure of the Calculator

The calculator consists of a large number of components, each of which is contained in a separate VHDL file. A schematic overview of the calculator is shown in Figure 2.6.

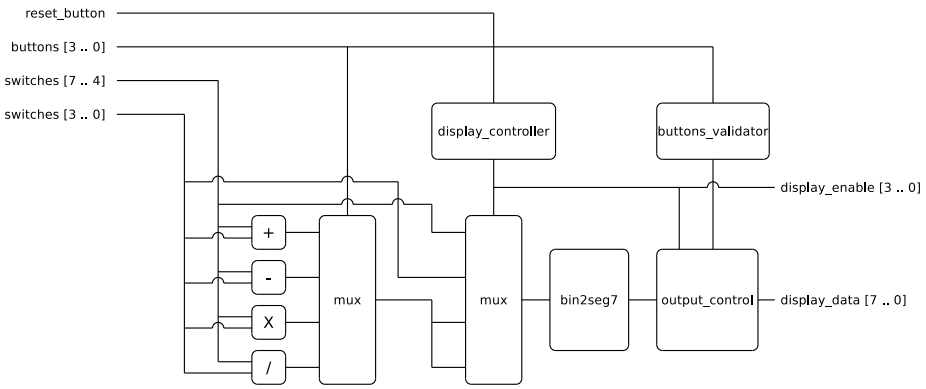


Figure 2.6: Diagram of the construction of the calculator

The calculator uses combinational logic (there is no memory), only the display controller is implemented as an FSM. The FSM is described later in Chapter 5. The inputs of the arithmetic blocks are directly connected to the slide switches. A change in the switches directly results in an update of the outputs of the blocks (the result of the calculations). The left mux then ensures that, depending on which button is pressed, only the required result can pass. All blocks after that mux ensure that the result is shown correctly on the 7-segment displays. The block called `bin2seg7` is a segment decoder. This block converts a binary input code to a correct value to show on the 7-segment displays.

Download the VHDL-files

On Brightspace you can find a .zip file called `calculator.zip`. Download this file and unzip it. This gives you all source files of the calculator and also a testbench to test the correct operation.

Create a new project

Create a new project and add all the files (including the constraints file) to the project. Make sure to specify that the testbench (`calculate_tb.vhdl`) is only used for simulation. Refer to Appendix B for more information on creating a Vivado project.

Perform a behavioural level simulation

In order to see whether and how the calculator works, you can run a behavioural simulation of VHDL code. Click on Flow Navigator→Simulation→Run Simulation→Run Behavioral Simulation.

Because the displays need a special form of control, it is difficult to check the operation of the calculator on the basis of the output signal *display_data*. As shown in Figure 2.6, the results of the various computational blocks can be seen on the output of the leftmost mux. You can add this signal to the waveform by clicking the > sign in the leftmost window of the simulator, tab Scope. This will reveal the structure of the calculator. Now you can find the correct mux (mux_8bit_4input_4sel) under lbl_mux_8bit_4input_4sel. If you click on that line, the input and output signals of that mux appear in the Objects frame in the middle. Right-click on the signal *output* and select Add to Wave Window. This signal is now added to the waveform.

To make it easier to interpret the values on the outputs, you can change the radix (the base) of the signals in the waveform into:

- *buttons* binary
- *switches* hexadecimal
- *output* signed decimal

You can set the radix by right-clicking the appropriate signal and then selecting the required radix under the option radix.

Simulate the calculator for at least 50 ms (use the command `run 50ms`), and check the proper functioning of the calculator.

Assign the pins

If the project is successfully simulated, the pins of the calculator have to be assigned. The pins should be assigned as follows:

- *clk* must be connected to the clock;
- *reset_button* must be connected to the center button `BTNC`;
- *buttons* must be connected to the other push buttons according to the description of the operation of the calculator;
- *switches* must be connected to the slide switches (*switches(0)* is LSB);

- *display_data* must be connected to the segments of the displays (*display_data(0)* is LSB and must be connected to segment A);
- *display_enable* should be connected to the enable signals of the displays (*display_enable(0)* is LSB, default name of these signals in the constraints file is *an[..]*).

Select Flow Navigator→Project Manager and open the constraints file in the Sources window. Modify the constraints file as required.

Brackets and Braces
In the VHDL code a bit of a vector is addressed with braces:
switches(0)
In the constraints file a bit of a vector is addressed with brackets:
switches[0]
Using the wrong type in either the VHDL code or the constraints file will result in an error during synthesis or implementation.

Generate Bitstream

If the pins are assigned you can generate the bitstream. Use the same procedure as in the first example.

Program the Calculator on the FPGA

Now the bitstream file is available, the calculator can be programmed on the FPGA. Use the same procedure as in the first example

Test the Calculator

To test the calculator, the calculator may need to be reset ny pressing `BTNC`. When the calculator has been reset, you can test the calculations by displaying the effect of different input values using the switches.

Chapter 3

VHDL: Introduction and Structural Design

Objectives

During this session you will:

- learn about some important elements in VHDL:
  - comments
  - libraries
  - entities
  - architectures
  - components
  - signals
  - port maps
- describe your design in VHDL as a top-level architecture

3.1 Introduction

In this chapter you will take your first steps into the wonderful world of VHDL. You've already seen some VHDL in the previous chapters, but then you could just assume that everything worked and you did not have to worry about entities, architectures, etc. Now that is past and it's time that you get started working with VHDL yourself. The first section of this chapter is about VHDL as a whole, what the characteristics are, who uses it, where it is being used, etc. Then a section follows on how to translate a diagram of a logic circuit into VHDL. The manual discusses the theory required for this. Lastly are some more examples to clarify the theory and you will have to do some assignments to become familiar with the matter.

The structural description of the robot
The last exercise gives the basis for the robot and the result will be used in the coming weeks. The results of your work forms a kind of blueprint.

3.2 VHDL

VHDL stands for 'Very High Speed Integrated Circuit Hardware Description Language'. The last three words, 'Hardware Description Language', are fairly obvious: VHDL is a language for describing hardware. From 'Very High Speed Integrated Circuit' we see that VHDL was designed as a tool for the design of digital ICs. The answer to the question "What is VHDL?" is: VHDL is a Hardware Description Language, a powerful tool used to help designing hardware. [1]

Concurrent vs. sequential
There are important differences between designing hardware and designing software.
When designing hardware you define connections between various independent elements. This means that each element may be active simultaneously. So if you have several systems programmed in an HDL, they can be active all at the same time: an HDL is a concurrent programming language.
When designing software, you create a stream of instructions for the processor. The processor can only execute one instruction at a time. So if you have programmed several systems in a programming language suitable for the processor, only one system can use the processor at the same time: this is called a sequential programming language.

3.2.1 Who uses VHDL and to what end?

VHDL is used by electrical engineers, programmers and mathematicians, to help to design digital systems. These systems can then be programmed into an FPGA [2] or put onto silicon (as chips). Programmers and mathematicians sometimes use hardware designs, as hardware designs are potentially much faster than conventional CPU systems (hardware designs are concurrent). This makes it possible to crack heavier types of encryption or work on other computationally intensive tasks.

VHDL: Simulation and synthesis
VHDL is a language with different applications. The language can be used for the simulation of designs for a better understanding of a circuit. Another possibility is to use the language to generate a circuit. This is called synthesis and is done with a synthesizer. For each target technology (CMOS, FPGA, etc) another synthesizer is required. In general, a synthesizer converts a design into a netlist (structural description) of the circuit composed of basic cells of the target technology. Only part of the VHDL specifications can be synthesized. The code in this manual consists only of synthesizable VHDL, except for testbenches.

[1] http://en.wikipedia.org/wiki/Hardware_description_language
[2] http://en.wikipedia.org/wiki/Field-programmable_gate_array

3.2.2 Complex systems and VHDL

A top-down approach is a common way to design complex systems. It means that the overall design of a project is designed first. The steps that follow are to fill in the details. In VHDL it is possible to maintain such a hierarchical structure. A higher hierarchical layer describes how different subsystems in a lower hierarchical layer are connected. Such a structure in a higher layer is called a *structural* description. Besides a structural descriptions, there are *behavioral* descriptions. In a behavioral description you describe the behaviour of a lower hierarchical layer instead of its structure.

For example, consider a logic circuit. Such a circuit consists of logic gates and a number of connections. The circuit could be seen as a structural description, while the logic gates themselves are described in a behavioral manner.

Different flavors in VHDL
There are different versions of VHDL: VHDL-87, VHDL-93, VHDL-2002 and VHDL-2008. This guide uses VHDL-93. This version of VHDL is more consistent than its predecessor and is widely supported by compilers and synthesizers. This does not (yet) hold for VHDL-2002 and VHDL-2008.

3.3 VHDL: Structural design

So, now that you know a little about VHDL and why you would want to learn it, it's time for the real thing: the translation of a logic circuit into VHDL. In order to do that, it is important that you realize what components are used to build a logic circuit. A logic circuit has a number of ports (inputs and outputs), a number of components (which also have inputs and outputs) and a number of interconnections. In VHDL we need descriptions of all these things.

The first VHDL element you have to deal with is an *entity*. An entity just defines a black box with a certain name and possibly a number of ports. An entity has one or more *architectures*. An entity and an architecture together form a system. Within a system you can use other systems by declaring them as components. A component is a reference to a subsystem. *Signals* can be used to transport information within one system. This is all summarized in Figure 3.1.

3.3.1 Comments

A comment is a piece of text in the code with which nothing happens. The text is there to make the code more readable and understandable. In VHDL you create a comment by typing -- in front of some text, as shown in the following example. The text after the -- till the end of the line is then treated as comment.

```
1 -- This is a comment.
```

You can also use comments to out-comment a part of your code. You can use this to make some pieces of the VHDL code inactive without removing them.

Use enough comments
No matter how structured your VHDL code is, always use enough comments!

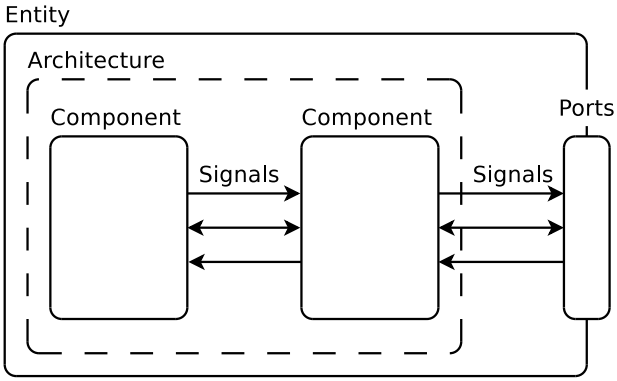


Figure 3.1: Overview of a structural description

3.3.2 Libraries

VHDL comes with a number of libraries. These are pieces of VHDL that contain definitions of functions and data types. Libraries contain packages. If you want to use functions or data types from a particular package, you must specify the library the package comes from. Then you must indicate which package you want. You can use the `use` statement to this end.

The library *std_logic_1164* contains the `std_logic` types, while the library *numeric_std* contains the arithmetic types.

`std_logic_arith` and `std_logic_(un)signed`
In many (older) VHDL projects you can find the `use` statements:

```
use IEEE.std_logic_signed.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
```

These are *not* official IEEE libraries and include, moreover, implementation errors. Use the new official IEEE library instead:

```
use IEEE.numeric_std.all
```

3.3.3 Entity declaration

In VHDL you use entities to name systems and to describe the inputs and outputs. An important entity is the *top-level* entity, also called the *system entity*. This entity describes the system as a whole.

In VHDL you declare an entity as follows:

```
1 entity <name> is
2         port (
3                 <name_port_0>    : <direction>   <type>;
4                 <name_port_1>    : <direction>   <type>;
5                 ...
6                 <name_port_n>    : <direction>   <type>
7         );
8 end entity <name>;
```

Most of the code is quite clear. You specify the name of the entity in `<name>`. The input and output ports are specified with `port();`. Inside the parentheses of `port();`

you name the ports, while the port declarations are separated by a semicolon (note: there is no semicolon after the last port declaration!). Each port of an entity has a name, a direction (`in` or `out`) and a type (see also Section 3.3.5), you must fill in in the right place.

Names
Make sure to use descriptive names for the ports! For example, `c_s` is a bad name, while `clock_slow` is better. In this case `clock_125khz` would be the best option.

Multiple entities in a single VHDL file
Although it is discouraged in the lab, it is possible to have multiple entities in a single VHDL file. Please note that you have to specify the libraries used in the entity before *each* new entity!

3.3.4 Architecture definition

Another important element in VHDL is the *architecture*. Architectures describe how entities work inside. Each entity has at least one architecture. There are three types of architectures, namely: *structural*, *behavioral* and *mixed* architectures. A structural architecture defines how subsystems are interconnected. By contrast, a behavioral architecture describes the behavior of a system, such as a logic function or an FSM. In mixed architectures both structural and behavioral statements are used.

Multiple architectures per entity
In VHDL, it is possible to have multiple architectures for each entity. This is necessary because with hardware design you first make a functional design and then you synthesize it. This synthesized design also contains parts that are used for implementation, such as timing information. To see if the behavior of both matches, you must simulate both architectures. In this case, *configuration files* are used to specify which architecture should be simulated.

In VHDL you declare an architecture as follows:

```
1 architecture <name architecture> of <name entity> is
2
3 begin
4
5 end architecture <name architecture>;
```

Here you have to replace `<name architecture>` by the name of the architecture. This name is in principle free to choose. In practice you usually call an architecture `structural` or `behavioral`, depending on the type of architecture you're designing. The second name, `<name entity>`, must be replaced by the name of the entity the architecture belongs to.

The section before `begin` is called: *declarative part*. You use this section to specify the components that your architecture is built of. In the following sections you'll see that in this section includes component and signal declarations. Nothing actually happens here!

After the keyword `begin` the *statement part* begins. In this part you define structure or the behaviour of the architecture in terms of the elements that were declared before.

Depending on the type of architecture (structural or behavioral), you can use different statements.

Mixed architectures
Although mixed architectures are allowed in the VHDL standard, they are not always supported by synthesis software. It is better not to use them.

3.3.5 Signal declarations and type definitions

In VHDL, *signals* are used to represent information. A signal can be used to store information or to transport information from one place to another. Information is a broad concept: it can be a binary value, 0 or 1, but it can also be an integer value, like 7 or 374. A binary value can be represented with a digital signal. For almost all other forms of information a *vector* of digital signals is necessary.

Each digital signal in VHDL has a type. This type determines what kind of signal it is, what values you can store in it, and what you can do with it. In a signal of type bit, for instance, you can store the values `'1'` and `'0'`. Different signals can be combined into a new signal by using *functions* and *operators*. More information can be found in Chapter 4.

Before you can use a signal in VHDL, you have to declare it first. For the declaration you need: the keyword `signal`, one or more unique names and the type. The syntax is shown below:

```
1 signal  <name_signal_0>, <name_signal_1>, ..., <name_signal_n>     : <signal_type>;
```

There are two types of signals, namely *scalars* and *vectors*. The difference is the number of elements available to store information. A scalar has only one element that can contain information, while a vector consists of several elements. An example of a scalar signal is the type `bit`. You can store the values `'1'` and `'0'` in it, but not `"1101"`. This is because `"1101"` consists of 4 elements. For storing `"1101"`, a bitvector is needed. The size of a vector is the number of elements that can contain information. The bitvector `"1101"` therefore has size 4.

In the case of a scalar signal the declaration is simple. Using the above syntax, the declaration of a signal with *a* as the name and `bit` as type is like this:

```
1 signal  a       : bit;
```

The declaration of a vector is slightly different than the declaration of a scalar as a vector has a size. To declare a vector, you can use the following syntax[3]:

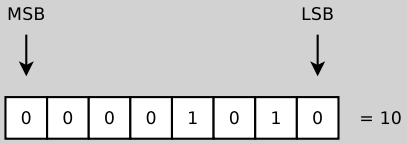
```
1 signal  <name_signal_0> : <signal_type> (<size - 1> downto 0);
```

Using this syntax the declaration of a signal with *number* as name, `unsigned` as type and a size of 4, is like this:

```
1 signal  number  : unsigned (3 downto 0);
```

[3] Other syntaxes are possible for declaring vectors.

MSB and LSB
When specifying the size of a vector the first number, `3` in the example, is the *index* of the *MSB* (most significant bit). The last number is the index of the *LSB* (least significant bit). The following picture illustrates the concept of MSB and LSB:



The type `std_logic`

For connecting the logic blocks you can use the type `std_logic`. This type is designed to model interconnects on a chip. In a signal of type `std_logic` you can store the values `'0'`, `'1'`, `'H'`, `'L'`, `'W'`, `'Z'`, `'U'`, `'X'` and `'-'`. All these values represent situations that are commonly used on chips. These situations are schematically shown in Figure 3.2 and will be briefly discussed.

The values `'1'` and `'0'` These values correspond to the logical values `TRUE` and `FALSE`. In practice (ie in an IC), this represents a connection to V_{cc} or `GND` respectively.

The values `'H'`, `'W'` and `'L'` The values `'H'` and `'L'` are weaker versions of `'1'` and `'0'` respectively. In this case weak says something about the impedance level: a weaker signal has a lower conductivity (and therefore a higher impedance). The diagram with the `'H'` as output contains both an impedance and a switch. When the switch is open, the output is equal to the weaker `'H'`. When the switch is closed, the output is equal to the stronger `'0'`. A similar reasoning can be done for the schematic with the `'L'` output.

The value `'W'` represents a situation for which the output voltage unknown, the only thing that is certain is that it is a weak signal.

The value `'Z'` It is possible that a port is both an input and an output. If a port is used as input, then this value should not affect the signal (no loading may occur). The degree of influence depends on the input impedance of the gate: the higher the impedance, the less this impedance influences the signal level. The value `'Z'` stands for a high input impedance.

The values `'U'`, `'X'` and `'-'` The values `'U'` and `'X'` are signal values that are usually undesirable. `'U'` stands for *uninitialized* and the value `'X'` stands for *undefined*. The accompanying diagrams explain their function.

Sometimes it does not matter what the value of a signal is. This can be indicated with a `'-'`, the *Don't care*.

The type `std_logic_vector`

The vector version of type `std_logic` is `std_logic_vector`. The following example declares a `std_logic_vector` with five elements:

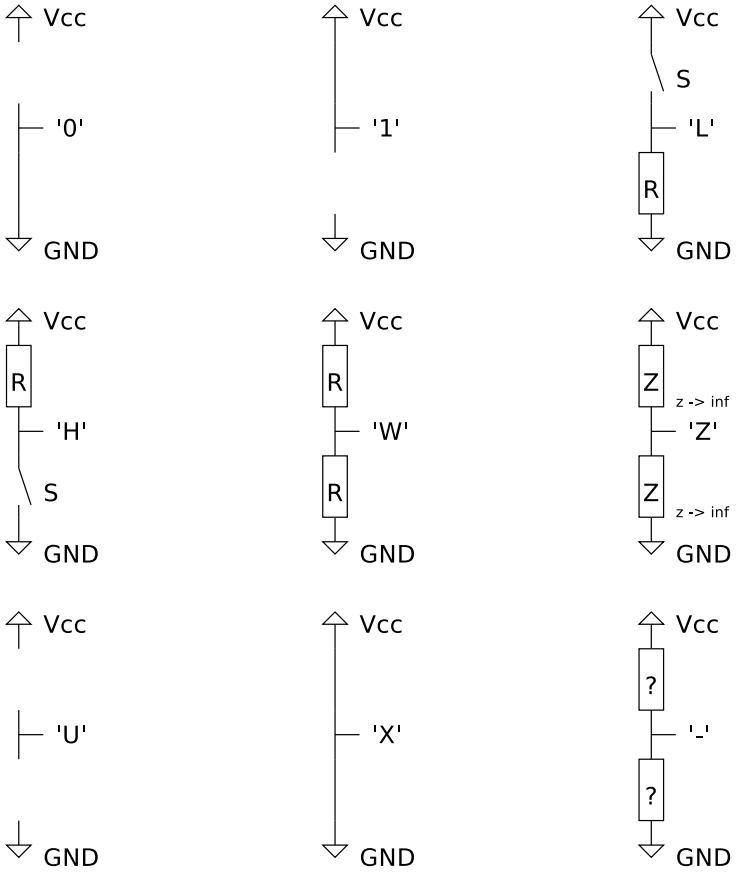


Figure 3.2: The possible values of type `std_logic`

```
signal number : std_logic_vector (4 downto 0);
```

The types `bit` and `boolean`

The simplest type of signal that VHDL knows, is the type `bit`. In a signal of type `bit` you can store the values 0 or 1. The type `bit` is intended to model a bit in a logic circuit. For this purpose a signal of type `std_logic` can also be used. The type `bit` models a more abstract concept than the type `std_logic`.

The type `bit` is a scalar. The associated vector type is `bit_vector`. As an example in VHDL a byte looks like this:

```
signal bit8 : bit_vector (7 downto 0);
```

Besides the type `bit` there is also the type `boolean`. In a signal of type `boolean`, the values `'TRUE'` and `'FALSE'` are stored. The type `boolean` is mainly used in logical expressions, which are discussed in Chapter 4.

The type `unsigned`

The type `unsigned` is intended to model positive integers. In principle, a signal of type `unsigned` is a bit vector where the bits have been given a meaning. Signals of type `unsigned` are thus vectors and model a more abstract concept than signals of type `bit_vector`.

The number of values that can be stored in a signal of type `unsigned` depends on the size of the signal. The maximum value that can be stored in a signal of type `unsigned` is $2^n - 1$, where n is the size of the vector. The largest number that can be stored in an `unsigned` signal of size 4 is equal to $2^4 - 1 = 15$. The minimum value that can be stored in a signal of type `unsigned`, is 0.

The type `signed`

The type `signed` is intended to model both positive and negative integers. Signals of type `signed` works in a way similar to signals of type `unsigned`: they are bitvectors where the bits have been given a particular meaning. Signals of type `signed` are thus also vectors and model a more abstract concept than signals of type `bit_vector`.

The number of values that can be stored in a signal of type `signed` depends on the size of the signal. Because the values are stored in two's complement, the maximum value that can be stored in a signal of type `signed` is $2^{n-1} - 1$, where n is the size of the vector. The minimum value that a signal of type `signed` can store is -2^{n-1}, where n is again the size of the vector. In a signed signal with size 4 a minimum value of $-2^{4-1} = -8$ and a maximum value of $2^{4-1} - 1 = 7$ can be stored.

The type `integer`

The type `integer` is used to represent positive and negative integer numbers. Integer types are mostly used internally and not as type of ports. Literal numbers are of type `integer` as well. Integers are at least capable of storing numbers in the range $2^{-31} - -2^{31} - 1$ but can be limited in range using the `range` keyword.

```
1        signal  my_number       : integer;
2        signal  months          : integer range 1 to 12;
```

User-defined types

Besides all existing types, VHDL also provides a way to define types yourself. This is, for example, commonly used in the design of FSMs (see Chapter 5). The syntax for this is shown below:

```
1        type <type_name> is (<value1>, <value2>, ...);
```

The following example creates a type `decimal` in which the values `'0'`, `'1'`, `'2'`, `'3'`, `'4'`, `'5'`, `'6'`, `'7'`, `'8'` and `'9'` can be stored:

```
1 type decimal is ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');
```

3.3.6 Constants

Constants are useful to improve readability of your VHDL code by preventing the use of literals. Constants in VHDL are denoted with the `constant` keyword instead of the `signal` keyword. Just like signals, constants have a type. Since the value of a constant cannot be modified, it should be specified during initialization as follows:

```
constant max_number : integer := 10;
```

This creates a constant *max_number* of type `integer` and value 10.

3.3.7 The ordinary assignment operator

In order to assign values to signals you can use the *normal assignment operator*, whose syntax is given below:

```
<signal0> <= expression;
```

The value of `expression` is assigned to <*signal0*>. An expression may be a signal, a literal constant or the result of a logical or arithmetic operation[4]. Obviously <*signal0*> must be able to store the value of the expression.

Assignment of scalar values

To assign a value to a signal immediately, you can assign a constant, as the example below illustrates. Examples of scalar constants are `'1'`, `'H'` and `'Z'`. In the following example *signal0*, *signal1* and *signal2* are of type `std_logic`. Note the single quotes around the constants!

```
signal0 <= '1';
signal1 <= 'H';
signal2 <= 'Z';
```

Assignment of vector values

You can also assign constants to vectors. The following example illustrates this:

```
vector1 <= "XXXXXX";
vector2 <= "001111";
vector3 <= (others => '0');
```

Note that vectors are enclosed in double quotes.

others
It is possible to give every element in a vector the same value at once (independent of the size of the vector). For this you can use *others*, as the above example illustrates.

Assignment of signals

It is also possible assign a signal to a signal of the same type. In the following example, *input* and *intermediate* are signals of type `std_logic`:

```
intermediate            <= input;                  -- scalar assignment
vector1                 <= vector2;                -- vector assignment
vector1 (2 downto 0)    <= vector3;                -- vector assignment
vector1                 <= vector4 (3 downto 0);   -- vector assignment
```

[4] logic and arithmetic operations are discussed in Chapter 4

Signed and unsigned literals

Signals of types signed and unsigned are vectors that represent numbers. For storing a number in a signal it must first be converted into a binary representation. In the following example, the number -4 is stored in a 4-bit `signed` and 7 in a 4-bit `unsigned`.

```
number1 <= "1100"; -- -4
number2 <= "0111"; -- 7
```

Converting numbers to a binary representation is cumbersome and time-consuming. Therefore VHDL provides functions to do this: `to_signed ();` and `to_unsigned ();`. The following example has the same results as the previous example and illustrates the use of `to_signed ();` and `to_unsigned ();`:

```
number1 <= to_signed (-4, 4);
number2 <= to_unsigned (7, 4);
```

Both functions expect two arguments. The first argument (`-4` and `7` in the example) is the value that you want to assign to the numeric type. The second argument (in both examples `4`) stands for the number of bits of the resulting vector.

Type casting

Type casting is converting from one data type to another. Type casting is necessary for actions such as storing an input value of type `std_logic_vector` in a signal of type `unsigned`. Type casting is also required for the use of functions and operators that expect a certain type of input signals.

The syntax for type casting is very simple: `new_type (old_type)`. The following example illustrates this:

```
signal  test_unsigned           : unsigned (7 downto 0);
signal  test_std_logic_vector   : std_logic_vector (7 downto 0);
...
-- convert an unsigned signal to a std_logic_vector:
test_unsigned           <= to_unsigned (255, 8);
test_std_logic_vector   <= std_logic_vector (test_unsigned);

-- convert the result of the function to a std_logic_vector
test_std_logic_vector   <= std_logic_vector (to_unsigned (255, 8));

-- other way around: convert a std_logic_vector to an unsigned:
test_unsigned           <= unsigned (test_std_logic_vector);
```

3.3.8 Component declaration

In VHDL you use *component*s to declare subsystems in structural architectures. A component within an architecture basically has the same function as an entity in a system. The declarations of both are exactly the same, except that `entity` has been replaced by `component`. You declare a component in VHDL as follows:

```
component <name component> is
        port (
                <name_port_0>   : <direction>   <type>;
                <name_port_1>   : <direction>   <type>;
                ...
                <name_port_n>   : <direction>   <type>
        );
end component <name component>;
```

Data hiding and code re-use
Components implement two important concepts in VHDL, namely data hiding and code re-use. Data hiding means that you have several separate levels of abstraction: the data from lower abstraction levels is invisible to higher levels of abstraction. Code re-use allows you to re-use pieces of VHDL code by putting it into a component. Especially if you want to make changes in your code later on, this can be a great advantage. You only have to change your code in one place now, instead of all the places where you use that code.

3.3.9 Component instantiations and port mappings

When you create an instance of a component in VHDL, this is called: component instantiation. The brand new component must then be connected to a set of signals. For this you use the command `port map ();`. In VHDL this looks like:

```
1 <lbl_name> : <cmp_name> port map        (       <cmp_port_0>    => <signal_0>,
2                                                 <cmp_port_1>    => <signal_1>,
3                                                 ...
4                                                 <cmp_port_n>    => <signal_n>
5                                         );
```

This piece of code creates a component named `<cmp_name>` and connects `<cmp_port_0>` with *<signal_0>*, `<cmp_port_1>` with *<signal_1>*, etc. Please note that the label is mandatory!

```
1 -- <label> : <component name> port map ( <signal of port 1>, <signal of port 2>,
      ... );
2 -- <signal of port x> can be an input or an output of the main entity or an
      internal signal.
3 lbl0: black_box_1 port map (input1 => input1,          output1 => internal_1_4);
4 lbl1: black_box_2 port map (input2 => input2,          output2 => internal_2_4);
5 lbl2: black_box_3 port map (input3 => input3,          output3 => internal_3_4);
6 lbl3: black_box_4 port map (inputs => internal_4_4,    output => outputs(3));
```

Declarations and instantiations
If you need a certain thing in VHDL, you must declare it first. Declaring something means that you indicate that something exists and what it looks like. Once something is declared, you can make one or multiple *instances* of it. For example, if you want to use an AND gate, you must first indicate that there exists an AND gate and that the AND gate has two inputs and one output. You can then, for each AND-gate that you need, create an instance and connect the inputs and outputs to the right signals.

It is possible for components to be used more than once (see the info-block on the concepts: data hiding and code re-use). Labels are used to differentiate between the various instances of components. It is therefore important that each component used, has a unique label.

3.4 Example: from schematic to VHDL

The elements in the previous section are all you need to describe the structure of a circuit in VHDL. In this section you will see how this process works: using the material covered, we will describe the diagram in Figure 3.3.

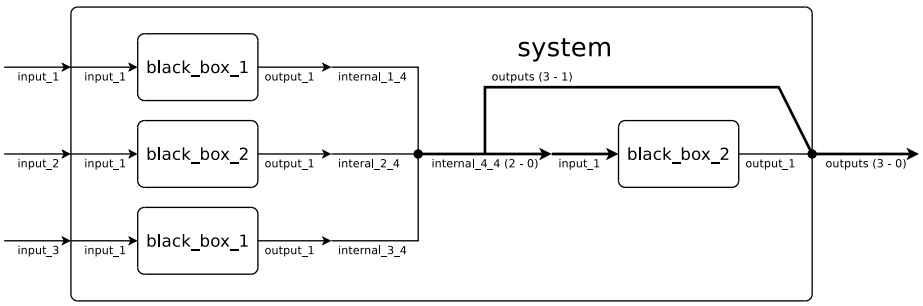


Figure 3.3: An example circuit

3.4.1 The libraries

As you have seen, to use the standard data types and functions, you must indicate that you want to use the associated libraries. In VHDL it looks like this:

```
1 -- Libraries for the commonly used datatypes and functions
2 library IEEE;
3 use IEEE.std_logic_1164.all;
```

3.4.2 The top-level entity and architecture

The next step is to create the top-level entity and the top-level architecture. The top-level entity contains the name and the inputs and outputs of the system. The corresponding architecture remains empty for the moment, beside the keyword `begin`.

```
1  -- The top-level entity
2  entity system is
3          port (  input_1 : in     std_logic;
4                  input_2 : in     std_logic;
5                  input_3 : in     std_logic;
6                  outputs : out    std_logic_vector (3 downto 0)
7          );
8  end entity system;
9
10 -- The top-level architecture
11 architecture structural of system is
12         -- Declarations should be put here:
13
14 begin
15         -- Statements should be put here:
16
17 end architecture structural;
```

3.4.3 The top-level architecture - component declarations

Now you have to write the component declarations for the black boxes used in the circuit. As you can see in Figure 3.3, in this example there are two different black boxes with the following component declarations:

```
1          -- The first black-box
2          component black_box_1 is
3                  port (  input_1 : in     std_logic;
4                          output_1: out    std_logic
5                  );
6          end component black_box_1;
7
8          -- The second black-box
9          component black_box_2 is
```

```
10              port (  input_1 : in    std_logic;
11                      output_1: out   std_logic
12              );
13      end component black_box_2;
```

3.4.4 The top-level architecture - signal declarations

All signals required for the connections should be declared. Figure 3.3 shows the different signals with all the names provided. Signal declarations for these signals are as follows:

```
1       -- The scalar internal signals
2       signal internal_1_4, internal_2_4, internal_3_4: std_logic;
3
4       -- The vector internal signals
5       signal internal_4_4: std_logic_vector (2 downto 0);
```

3.4.5 The top-level architecture - component and signal instantiations

The last thing still to be done to finish the description is connecting all signals and components in the right way. The following listing first creates the node, followed by the component instantiations.

```
1       -- The node
2       internal_4_4 (0)        <= internal_1_4;
3       internal_4_4 (1)        <= internal_2_4;
4       internal_4_4 (2)        <= internal_3_4;
5       outputs (3 downto 1)    <= internal_4_4 (2 downto 0);
6
7       -- Component instantiations
8       lb10:   black_box_1     port map (      input_1         => input_1,
9                                               output_1        => internal_1_4
10                              );
11
12      lb11:   black_box_2     port map (      input_1         => input_2,
13                                              output_1        => internal_2_4
14                              );
15
16      lb12:   black_box_1     port map (      input_1         => input_3,
17                                              output_1        => internal_3_4
18                              );
19
20      lb13:   black_box_2     port map (      input_1         => internal_4_4
            (0),
21                                              output_1        => outputs (0)
22                              );
```

3.4.6 The result

The complete description of the diagram in Figure 3.3 will be as follows:

```
1  -- Libraries for the commonly used datatypes and functions
2  library IEEE;
3  use IEEE.std_logic_1164.all;
4
5  -- The top-level entity
6  entity system is
7       port (  input_1 : in    std_logic;
8               input_2 : in    std_logic;
9               input_3 : in    std_logic;
10              outputs : out   std_logic_vector (3 downto 0)
11      );
12 end entity system;
```

```
13
14 -- The top-level architecture
15 architecture structural of system is
16         -- Declarations should be put here:
17         -- The first black-box
18         component black_box_1 is
19                 port (  input_1 : in     std_logic;
20                         output_1: out    std_logic
21                 );
22         end component black_box_1;
23
24         -- The second black-box
25         component black_box_2 is
26                 port (  input_1 : in     std_logic;
27                         output_1: out    std_logic
28                 );
29         end component black_box_2;
30
31         -- The scalar internal signals
32         signal internal_1_4, internal_2_4, internal_3_4: std_logic;
33
34         -- The vector internal signals
35         signal internal_4_4: std_logic_vector (2 downto 0);
36
37 begin
38         -- Statements should be put here:
39         -- The node
40         internal_4_4 (0)        <= internal_1_4;
41         internal_4_4 (1)        <= internal_2_4;
42         internal_4_4 (2)        <= internal_3_4;
43         outputs (3 downto 1)    <= internal_4_4 (2 downto 0);
44
45         -- Component instantiations
46         lbl0:   black_box_1     port map (      input_1         => input_1,
47                                                 output_1        => internal_1_4
48                                 );
49
50         lbl1:   black_box_2     port map (      input_1         => input_2,
51                                                 output_1        => internal_2_4
52                                 );
53
54         lbl2:   black_box_1     port map (      input_1         => input_3,
55                                                 output_1        => internal_3_4
56                                 );
57
58         lbl3:   black_box_2     port map (      input_1         => internal_4_4
               (0),
59                                                 output_1        => outputs (0)
60                                 );
61
62 end architecture structural;
```

3.5 Assignments

3.5.1 A simple circuit

This exercise is about translating a simple circuit to VHDL code. The circuit that you need to translate, is shown in Figure 3.4.

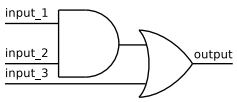


Figure 3.4: A simple circuit

Translate this circuit into VHDL code. Please observe the following:

1. You do not have to create the entities and architectures of the components, it is about the structural description!
2. You can use the example in the text.
3. Note the names and the indentation!

3.5.2 Top-level description of the Mars Rover Robot

The final assignment consists of designing the top-level entity and the top-level architecture of the robot. Before you begin, download the file `entities.zip` from Brightspace.

The top-level entity

The top-level entity of a system describes the black box model.

1. Which signals serve as inputs for control?
2. Which serve as output signals for control?
3. Determine the types of the input and output signals.
4. Explain why it is a bad idea to place the output signals of the sensors in a vector.

Design, based on the answers given, the top-level entity.

The top-level architecture

The top-level architecture of a system describes how the various modules are connected.

The skeleton of the top-level architecture The skeleton of an architecture is an "empty" architecture. There are no declarations or instances in it yet.

1. What is the type of the top-level architecture: structural or behavioral?

Fill in the top-level architecture of the skeleton according to the answers given.

The component declarations Components can be used to include subsystems in a structural description. The entities of the component files can be found in `entities.zip`. Note: the vector sizes are not correct. This must be adjusted according to your own understanding later on.

1. Which subsystems make up the robot?
2. Design the component declarations using the given entity descriptions.
3. Add the component declarations in the correct place in the top-level architecture.

The signals Signals are the means to transport information from one port to another.

1. Describe in your own words how you can determine all required signals in a structured way.
2. What is the type of the required intermediate signals?
3. Design the declarations of the intermediate signals.
4. Add the declarations of the signals in the right place in the top-level architecture.

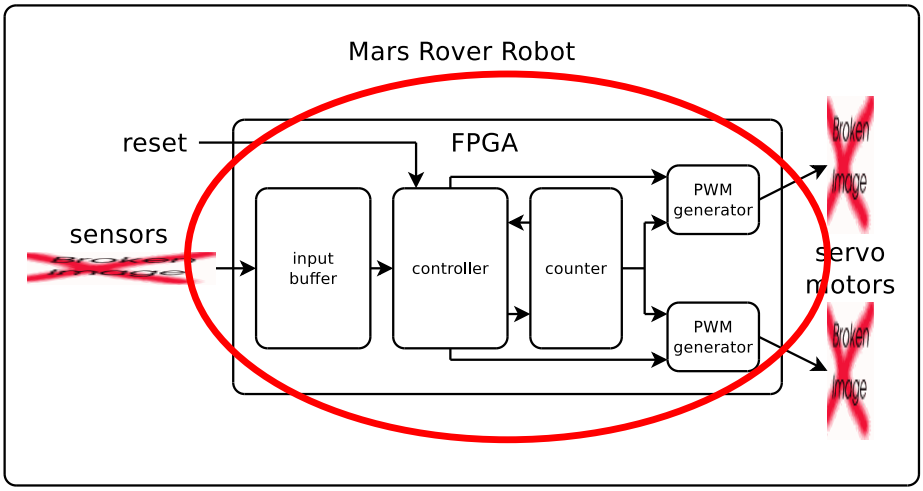
The component instantiations After a component is declared, you can make instantiations of it. This process is called component instantiation. You can use the command `port map ()` to connect the ports to the right signals.

1. For each component, specify the amount of instances required for the robot.
2. For each component, create the required number of instances.
3. Connect all ports to the right signals by using `port map ();`.

The top-level entity and the top-level architecture are now ready. In the next chapters the architectures of the modules will be implemented.
In Appendix A you can find an enlarged version of the overview of the robot. Annotate this overview with the names and sizes of the signals connecting the different blocks. Add any missing signals.

3.6 Overview of the robot

This session you've created the structural description of the hardware modules that have to be put in the FPGA. In the coming sessions, these modules are implemented separately.



Chapter 4

VHDL: Behavioural Design and Clocked Circuits

Objectives

During this session, you will:

- learn to describe behavioural circuits in VHDL
- learn to describe clocked circuits in VHDL
- learn to create a testbench to test your design
- design, implement, simulate and synthesize a counter

4.1 Introduction

In the previous session, you've implemented the *structural* architecture of the top-level entity. A structural architecture is used to combine several smaller parts into one larger design. In some cases, however, it is neccessary to describe the behaviour of a design instead of the structure. In VHDL this can be done using the *behavioural* architecture. VHDL has a rich set of statements and constructs to describe behavioural architectures. The most important of those are described in this chapter, see eg. Ashenden [1] for more information.

To test the theory, you have to modify an existing behavioural design first. This also requires the use of testbenches to verify the functionality of the system. You will also work on a component of the robot: the counter. This counter provides the time base for the robot and is required for the motor control.

4.2 Concurrent Statements

In the previous session on structural architectures, it was already mentioned that VHDL has concurrent statements. Two of those are already used: the signal assignment (<=) and the `port map`. These statements should be placed in the statement part of the architecture. Concurrent statements are very well suited for describing the behaviour

of combinational circuits. In such circuits, the outputs directly depend on the input values and all components of the circuit work concurrently.

VHDL has several operators to describe these type of circuits.

4.2.1 Operators

An operator establishes a relationship between (usually) two expressions. The result is stored (using the assignment operator) or used in a comparison or more complex expression. An expression is a (more or less) mathematical formula and may include constants, operators and other expressions. The result of an expression is an expression again. In VHDL most operators are binary (describe a relationship between two expressions), some operators are unary (work on a single expression). An example of a binary operator is the + used to calculate the sum of two expressions. An example of a unary operator is the unary `NOT` used to calculate the inverse value of an expression.
A binary expression usually looks like this:

```
expression1 binary_operator expression2
```

A unary expression looks like this:

```
unary_operator expression
```

Assignment Operator

The assignment operator is already discussed in the previous session. There are three distinct versions of the assignment operator:

Normal assignment operator The normal assignment operator unconditionally assigns a value to a signal.

```
1 a_signal <= value;
```

Conditional assignment operator The conditional assignment operator conditionally assigns a value to a signal.

```
1 a_signal <= value when choice; -- choice is a boolean expression
2 a_signal <= value when choice else value;
```

Selection assignment operator The `with ... select` version of the assignment operator can be used to easily implement a lookup table.

```
1 with expression select a_signal <= value when choice [, value when choice];
```

Logical Operators

The following logical operators are available: `not` (unair), `and`, `or`, `nand`, `nor`, `xor`, `xnor`. The operator `&` is used for concatenation (combining two signals into a single larger signal).

Comparison Operators

Comparison operators can be used inside a conditional assignment operator. The following comparison operators are available: =, <, >, <=, >=, /= (unequal to).

Arithmetic Operators

Arithmetic operators are only defined for signals of the type `unsigned` and `signed`, but not for the type `std_logic`. The following arithmetic operators are available: `+` (unary and binary), `-` (unary and binary), `*`, and `/`.

4.2.2 Examples

In order to clarify the theory a few examples of concurrent circuits follow.

2-input AND Gate

This 2-input AND gate shows the implementation of a simple logic gate using an assignment operator and a logical operator.

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity and_2_input is
5          port (  input_0 : in    std_logic;
6                  input_1 : in    std_logic;
7                  output  : out   std_logic
8          );
9  end entity and_2_input;
10
11 architecture behavioural of and_2_input is
12 begin
13
14         output  <= input_0 and input_1;
15
16 end architecture behavioural;
```

Binary-to-7-segment Decoder

A binary-to-7-segment decoder maps a 4-bit binary input to an 8-bit code suited for displaying on a 7-segment display (such as those on the FPGA-board). See the Basys3 Reference Manual (you can find this on Brightspace) for more information on the encoding of a 7-segment display.

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity bin2seg7 is
5          port (  bin     :       in      std_logic_vector (3 downto 0);
6                  seg7    :       out     std_logic_vector (7 downto 0)
7          );
8  end entity bin2seg7;
9
10 architecture behavioural of bin2seg7 is
11
12 begin
13         with bin select
14                 seg7 <= "11000000" when "0000", -- 0
15                         "11111001" when "0001", -- 1
16                         "10100100" when "0010", -- 2
17                         "10110000" when "0011", -- 3
18                         "10011001" when "0100", -- 4
19                         "10010010" when "0101", -- 5
20                         "10000010" when "0110", -- 6
21                         "11111000" when "0111", -- 7
22                         "10000000" when "1000", -- 8
23                         "10010000" when "1001", -- 9
24                         "10001000" when "1010", -- A
25                         "10000011" when "1011", -- b
```

```
26                          "11000110" when "1100", -- C
27                          "10100001" when "1101", -- d
28                          "10000110" when "1110", -- E
29                          "10001110" when "1111", -- F
30                          "11111111" when others; -- Weird condition
31
32 end architecture behavioural;
```

Combining Signals

The following code uses two methods to combine two 4-bit vectors into a single 8-bit vector. This shows the use of the `&` operator. Both output signals are assigned the same value. If, for example, *input0* has the value "0010"and *input1* has the value "1101", both output signals have the value "11010010".

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity combine is
5          port (  input_0         : in    std_logic_vector (3 downto 0);
6                  input_1         : in    std_logic_vector (3 downto 0);
7
8                  output_0        : out   std_logic_vector (7 downto 0);
9                  output_1        : out   std_logic_vector (7 downto 0)
10         );
11 end entity combine;
12
13 architecture behavioural of combine is
14 begin
15
16         -- Combine two input signals into one output signal
17         -- by assigning a subset of the bits of the output signal
18         output_0 (3 downto 0)   <= input_0;
19         output_0 (7 downto 4)   <= input_1;
20
21         -- Combine two input signals into one output signal
22         -- using the concatenation operator
23         output_1                <= input_1 & input_0;
24
25 end architecture behavioural;
```

Clock Signal

You can use `after <time>` in order to execute an assignment after the specified delay. This can not be synthesized in hardware, but is of great importance for simulations. The following code generates a clock signal with a period of 10 ns and a 50% duty cycle.

```
1  clk             <=      '1' after  0 ns,
2                          '0' after 5 ns  when clk /= '0' else '1' after 5 ns;
```

4.3 Sequential Statements

A third concurrent statement in VHDL is the `process` statement. Different `process` statements are executed concurrently. The code *inside* a `process` block however, is not executed concurrently, but sequentially (the code is executed line after line).

4.3.1 The `process` Block

A `process` block looks like this:

```
process (signal1, signal2)
begin
        ...
end process;
```

The signals *signal1* and *signal2* are in the *sensitivity list*. Each time the value of one of the signals in this list changes, the code inside the process block is executed again.

4.3.2 Operators

You can use the same operators inside a `process` block as you can use for concurrent statements (logical, comparison and arithmetic). The assignment operator is an exception: inside a `process` block you can only use the normal version of the assignment operator. The conditional assignment operator can be implemented using `if` statements, while the `with ... select` version can be implemented with a `case` statement.

4.3.3 Statements

There are several additional statements available inside a `process` statement. Using these statements, it is relatively easy to describe the behaviour of a circuit.

If Statement

The `if` statement can be used to execute code conditionally. An `if` statement has a number of optional parts, a complete version could look like the following code:

```
if boolean-expression then
        -- if-body
elsif boolean-expression1 then
        -- elsif-body1
elsif boolean-expression2 then
        -- elsif-body2
else
        -- else-body
end if;
```

Code inside the if-body will only be executed if the `boolean-expression` evaluates to `TRUE`. Such a `boolean-expression` is normally written in the form of a comparison between two signals. The optional `elsif` part allows for testing another boolean expression. Multiple `elsif` parts are allowed within an `if` statement. The code inside the optional else-body is executed only if the code inside the `if` or `elsif` bodies isn't executed. The `end if` is required and closes the `if` statement.

Case Statement

An `if` statenment can be used to conditionally execute a piece of code based on the result of a complex boolean expression. In many cases, however, an option has to be selected from a large list of fixed values. A common example is the lookup table: every input maps onto exactly one output value. It is possible of course to code such an option-list using `if` statements. This can, however, be implemented much easier using the `case` statement. Such a `case` statement is as follows:

```
case signal_name is
        when option1    =>
                -- option1_body
        when option2    =>
                -- option2_body
        when others     =>
                -- option_others_body
end case;
```

In this example, `option1` has to be a constant value, eg. "0000". The special option `others` used here is equivalent with the `else` part in an `if` statement: the code is only executed if no other option applies.

The use of `others`
In order to synthesize a `case` statement in the FPGA, it is neccessary to handle *all* possible values that the signal can assume. This may not always be clear: for example a 1-bit signal of the type `std_logic` doesn't have two possible values (0 and 1), but 9! (0, 1, H, L, Z, -, U, X). You can use `others` to handle these possible situations.

4.3.4 Examples

In order to clarify the theory, a few examples of simple circuits described using sequential statements follow.

2-input Mux

This 2-input mux is described using an `if ... then ... else ...` statement.

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity mux_2_input is
5          port (  input_0 : in    std_logic;
6                  input_1 : in    std_logic;
7                  sel     : in    std_logic;
8                  output  : out   std_logic
9          );
10 end entity mux_2_input;
11
12 architecture behavioural of mux_2_input is
13 begin
14
15         process (sel, input_0, input_1)
16         begin
17                 if (sel = '0') then
18                         output  <= input_0;
19                 else
20                         output  <= input_1;
21                 end if;
22         end process;
23 end architecture behavioural;
```

4-input Mux

This 4-input mux is described using a `case` statement. This is simpler than using an `if` statement for a large number of choices.

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity mux_4_input is
5          port (  input_0 : in    std_logic;
6                  input_1 : in    std_logic;
7                  input_2 : in    std_logic;
8                  input_3 : in    std_logic;
9                  sel     : in    std_logic_vector(1 downto 0);   -- 2-bit selection
10                 output  : out   std_logic
11         );
12 end entity mux_4_input;
13
14 architecture behavioural of mux_4_input is
```

```
15 begin
16         process (sel, input_0, input_1, input_2, input_3)
17         begin
18                 case sel is
19                         when "00" => output <= input_0;
20                         when "01" => output <= input_1;
21                         when "10" => output <= input_2;
22                         when "11" => output <= input_3;
23
24                         when others => output <= '0';
25                 end case;
26         end process;
27 end architecture behavioural;
```

4.4 Clocked Circuits

Using the behavioural VHDL elements described thus far, you can build any combinational circuit. However, you can't describe clocked circuits, such as flipflops, timers and counters yet. A flipflop is a basic memory element capable of storing a single bit. It is the basic element of all clocked circuits. A flipflop stores its input value only after a clock edge. This means we need a way to describe a clock edge, in order to implement a flipflop in VHDL. To this end, VHDL provides `rising_edge (signal)`. By combining this `rising_edge` with an `if` statement, one can easily describe a D-flipflop:

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity d_flipflop is
5          port (  clk     : in    std_logic;
6                  D       : in    std_logic;
7                  Q       : out   std_logic
8          );
9  end entity d_flipflop;
10
11 architecture behavioural of d_flipflop is
12 begin
13         -- clk in sensitivity list
14         process (clk)
15         begin
16                 -- Copy input value to output on a rising edge of the clock
17                 if (rising_edge (clk)) then
18                         Q <= D;
19                 end if;
20         end process;
21 end architecture behavioural;
```

Most clocked circuits not only contain flipflops, but also a part combinational logic. For example in a counter, an addition is done using combinational circuitry, but the result is stored using flipflops.

Instead of `rising_edge (clk)`, one can also see (especially in older code) the code `clk'event and clk='1'`. A `clk'event` occurs when the clock changes (this detects the clock edge), the test for `clk='1'` determines if there was a rising edge or a falling edge. Both constructs are functionally almost identical, but `rising_edge` is clearer.

4.4.1 Examples

In order to clarify the use of `rising_edge` for describing clocked circuits, a couple of examples follow.

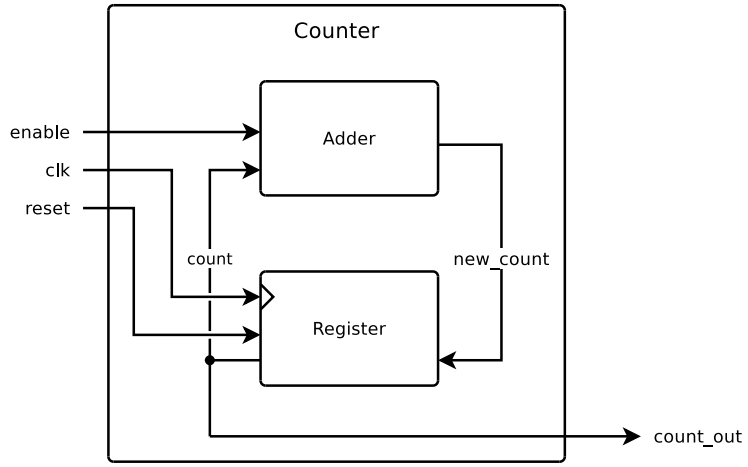


Figure 4.1: Counter with enable

D-flipflop with Synchronous Reset

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity d_flipflop_reset is
5          port (  clk     : in    std_logic;
6                  reset   : in    std_logic;
7                  D       : in    std_logic;
8                  Q       : out   std_logic
9          );
10 end entity d_flipflop_reset;
11
12 architecture behavioural of d_flipflop_reset is
13 begin
14         process (clk, reset)
15         begin
16                 -- The code inside the following if statement is executed
17                 -- on a rising edge of the clock. The value of the reset
18                 -- signal is tested only when there is a rising edge of the
19                 -- clock. This means this is a synchronous reset
20                 if (rising_edge (clk)) then
21                         if (reset = '1') then
22                                 Q       <= '0'; -- reset value of the output
                                       signal
23                         else
24                                 Q       <= D;   -- Copy input to output
25                         end if;
26                 end if;
27         end process;
28 end architecture behavioural;
```

4-bit Counter with Enable

A counter contains a register and a combinational block (the adder), see Figure 4.1. For this adder with enable, the following holds:

- The value of the signal *count* must be initialised with *reset*
- The value of the signal *count* can only change on a rising clock edge
- The counter may only count when the signal *enable* equals '1'

We can now define two processes that are executed concurrently:

1. A process that implements the register and runs on changes of the clock. This makes sure the value of *count* only changes on a rising clock edge.
2. A process that implements the combinational logic and runs on changes of *enable* (if *enable* equals '1' the counter may count) or a change of *count*.

We need the signals *count* and *new_count*, with *count* containing the current value of the counter and *new_count* containing the next value of the counter. This results in the following code:

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.numeric_std.all;
4
5  entity counter is
6          port (  clk             : in    std_logic;
7                  reset           : in    std_logic;
8                  enable          : in    std_logic;
9                  count_out       : out   std_logic_vector (3 downto 0)
10         );
11 end entity counter;
12
13 architecture behavioural of counter is
14
15         signal  count, new_count        : unsigned (3 downto 0);
16
17 begin
18         -- This process generates the register
19         process (clk)
20         begin
21                 if (rising_edge (clk)) then
22                         if (reset = '1') then
23                                 -- Reset the count value to 0
24                                 count   <= (others => '0');
25                         else
26                                 count   <= new_count;
27                         end if;
28                 end if;
29         end process;
30
31         -- This process calculates the new count value
32         process (count, enable)
33         begin
34                 if (enable = '1') then
35                         -- only increment if enable = '1'
36                         new_count       <= count + 1;
37                 else
38                         new_count       <= count;
39                 end if;
40         end process;
41
42         -- This line connects the internal count value to the outside world
43         count_out       <= std_logic_vector (count);
44
45 end architecture behavioural;
```

4.5 Testbenches

As described earlier, you have to test your VHDL design using a testbench. A testbench is like an enveloping black box of the entity under testing. Within the testbench values are presented to the inputs of entity, while the outputs are monitored. This is schematically depicted in Figure 4.2. As shown in the diagram, the testbench itself has neither inputs nor outputs. This means its entity description is empty. In VHDL this entity may look like this:

```
1 entity testbench is
2 end entity testbench;
```

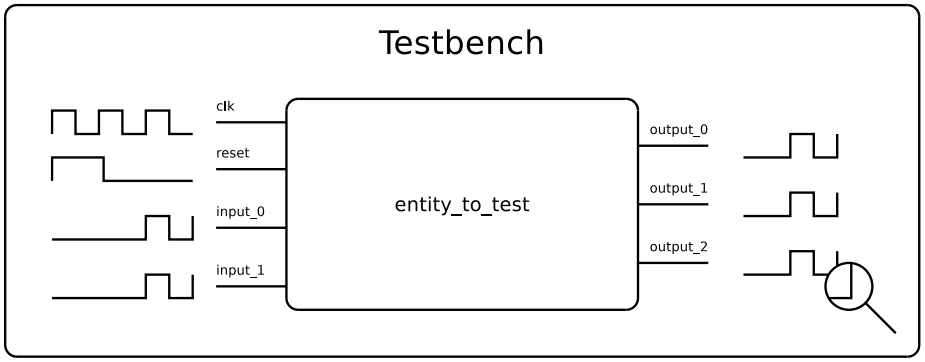


Figure 4.2: Diagram of a testbench

Since the testbench has to generate input values for the entity undergoing testing, its architecture contains a portmap to correctly map the signals to the entity and signal assignments to provide proper values to the signals. The testbench from the diagram could look as follows in VHDL:

```
1  architecture structural of testbench is
2
3          component entity_to_test is
4                  port (  clk      : in     std_logic;
5                          reset    : in     std_logic;
6                          input_0  : in     std_logic;
7                          input_1  : in     std_logic;
8                          output_0: out    std_logic;
9                          output_1: out    std_logic;
10                         output_2: out    std_logic
11                 );
12         end component entity_to_test;
13
14
15         signal  clk      : std_logic;
16         signal  reset    : std_logic;
17         signal  input_0  : std_logic;
18         signal  input_1  : std_logic;
19         signal  output_0: std_logic;
20         signal  output_1: std_logic;
21         signal  output_2: std_logic;
22
23 begin
24
25         clk              <=      '1' after  0 ns,
26                                  '0' after  5 ns  when clk /= '0' else '1' after 5
                                        ns;
27         reset            <=      '1' after  0 ns,
28                                  '0' after 11 ns;
29         input_0          <=      '0' after  0 ns,
30                                  '1' after 20 ns,
31                                  '0' after 25 ns,
32                                  '1' after 30 ns;
33         input_1          <=      '0' after  0 ns,
34                                  '1' after 20 ns,
35                                  '0' after 25 ns,
36                                  '1' after 30 ns;
37
38 lbl0:    entity_to_test port map (       clk      => clk,
39                                          reset    => reset,
40                                          input_0  => input_0,
41                                          input_1  => input_1,
42                                          output_0=> output_0,
43                                          output_1=> output_1,
44                                          output_2=> output_2);
45 end architecture structural;
```

By simulating the testbench as toplevel entity in a VHDL simulator, it is possible to quickly execute a complete and complex simulation. A testbench also provides a means to verifiy whether changes in the VHDL code influenced existing functionality (regression testing).

4.6 Assignments

In order to do some practice with behavioural elements of VHDL, the first assignment is extending an existing behavioural design: the calculator from the tutorial. You will also work on the counter during this lab session. This counter is a key component of the PWM generators and the controller.

4.6.1 Extend the Calculator with New Functions

Currently, the calculator provides the following functions:

- addition
- subtraction
- multiplication
- division/modulo

Each of these functions is assigned to a button. In this assignment, you need to add a couple of functions to the calculator. Since there are no more buttons available one of the slide switches will be used similar to the `2ND` button found on many calculators. The functions should be implemented according to Table 4.1.

Table 4.1: New functions of the calculator and the corresponding button combinations

Function	Button combination
`and` and `or`	`SW15` and `BTNU`
`nand` and `nor`	`SW15` and `BTND`
`xor` and `xnor`	`SW15` and `BTNL`
comparison	`SW15` and `BTNR`

The logical functions don't change the number of bits. This means we can show the result of two logical functions on the two result displays. In case of the `and/or` function, the left display should show the result of the `and` function and the right display should show the result of the `or` function. The `nand/nor` and `xor/xnor` functions can be implemented in a similar way. For the comparison function, the result is defined as follows:

Comparison	Display
input_0 < *input_1*	FF
input_0 = *input_1*	00
input_0 > *input_1*	11

In order to implement these new functions, a number of components of the calculator need to be modified. For clarities sake, the diagram of the calculator is depicted again in Figure 4.3.

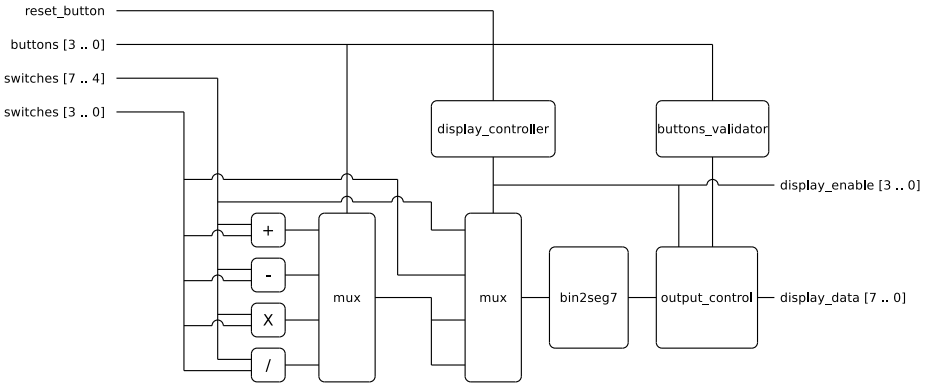


Figure 4.3: Diagram of the construction of the calculator

Download the VHDL Files

If you no longer have the VHDL files of the calculator, you can download the file `calculator.zip` from Brightspace.

Analyze the Construction of the Calculator

Use the code to determine which signals connect the different blocks in Figure 4.3. Also determine the number of bits of these signals.

Implement the `and/or` Function

The first step is implementing the new function. The function blocks are of course very similar to existing functions. For example, the in- and output signals have the same size. Create the file `and_or.vhdl` for the `and/or` function. Use the examples and the code provided to implement this block. Remember, the result of the `and` function should be shown on the left display, the result of the `or` function should appear on the right display. The following table contains a detailed example of how these functions should work.

Input				Output	
Switches	Display	Value	Function	Result	Display
input_0 	A	1010	and	1000	8
input_1 	C	1100			
input_0 	A	1010	or	1110	E
input_1 	C	1100			

Make sure the Buttons Work Correctly

When the `and/or` function is implemented, you should be able to select that function with `BTNU` and switch `SW15`. You will have to modify the mux used to select the required function. This mux is implemented in the file `mux_8bit_4input_4sel.vhdl`. Modify the VHDL code in such a way that the combination of `SW15` and button `BTNU` puts the result of the `and/or` function on the output of the mux. Don't forget to update the sensitivity list of the `process` block.

Add the New and Modified Blocks to the Top-level Architecture

Now that all blocks are implemented or modified, they can be added or modified in the top-level architecture. This top-level architecture is located in the file `calculator.vhdl`. Add the component descriptions of the new blocks and modify the descriptions of the mux and the buttons-validator. Also update the port maps.

Update the Testbench and Simulate the System

The provided testbench only tests the original four valid button combinations. Modify the testbench in order to able to test the new `and/or` function. Simulate the system and verify the result. Show the working system to a supervisor.

Implement the `xor/xnor` Functions

Now implement the `nand/nor` function in the VHDL file `nand_nor.vhdl` and the `xor/xnor` function in the VHDL file `xor_xnor.vhdl`. Take the similar steps as for the `and/or` function.

Implement the Compare Function

The last new function to add is the compare function. You should implement this function in the file `compare.vhdl`. This function has to be implemented with a `process` block and `if` statements.

Implement the Modified Calculator in the FPGA

If all functions are correctly implemented and simulated, the modified calculator can be tested on the FPGA board. Update the constraints file, synthesize the modified calculator, and generate a bitstream file. Load the bitstream file into the FPGA and test the result.

4.6.2 Design and Implement a Counter

Now that you have gained some experience with behavioural VHDL elements, it is time to implement a component of the robot: the counter. The counter is an important component of the robot: it is the core of the PWM generators that control the motors. The counter provides a time reference to enable the PWM generators to generate an accurate PWM signal.

Overview of the Counter

A diagram of the counter is shown in Figure 4.4.

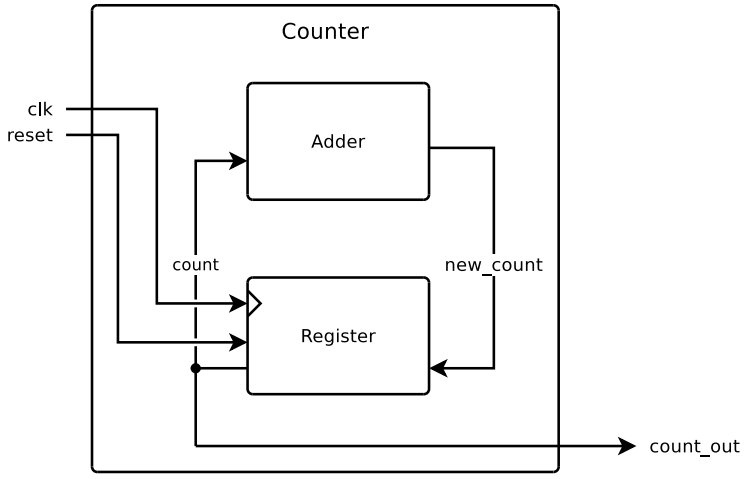


Figure 4.4: Diagram of the counter

The input and output signals have the following meaning:

- *clk* is the 100 MHz system clock.
- the *reset* signal is used to reset the counter to 0.
- the *count_out* signal is output of the counter and shows the current value of the counter.

As discussed earlier, the counter is used to generate a PWM signal. This means the counter should at least be able to count the period of one PWM pulse. Furthermore, the counter can be reset by a synchronous reset.

Implement the Entity of the Counter

Create a VHDL file `counter.vhdl` that contains the entity of the counter with the input and output signals as shown in Figure 4.4. The signal *count_out* contains the current value of the counter. Since the counter counts further than 1, this signal should be of the type `std_logic_vector`. Find the answer to the following questions in order to find out the required number of bits of this vector:

- Up to which number should the counter be able to count at least? (Hint: the system clock runs at 100 MHz)
- How many bits are required to represent this number?

Implement the Architecture of the Counter

As shown in Figure 4.4, the counter consists of two elements:

1. A piece of memory to keep the count value
2. An adder to determine the new count value

Implement the Register The required memory must be implemented using flipflops. This means you need a `process` block to implement a register. As usual for a register, the value of the signal *new_count* is copied to the signal *count* at each rising edge of the clock. When the *reset* signal has the value `'1'`, the counter must be reset to the value 0. Implement such a register using a behavioural VHDL description.

Implement the Adder The addition can be implemented with the `+` operator. This operator is, however, not defined for signals of the type `std_logic`. It is defined for signals of the type `unsigned`. This means that the signals *count* and *new_count* should be of the type `unsigned`. The adder should be implemented in a `process` block. Carefully think about which signals should be placed in the sensitivity list of this `process` block. The counter *should not* automatically restart itself, this should only be done after a reset!

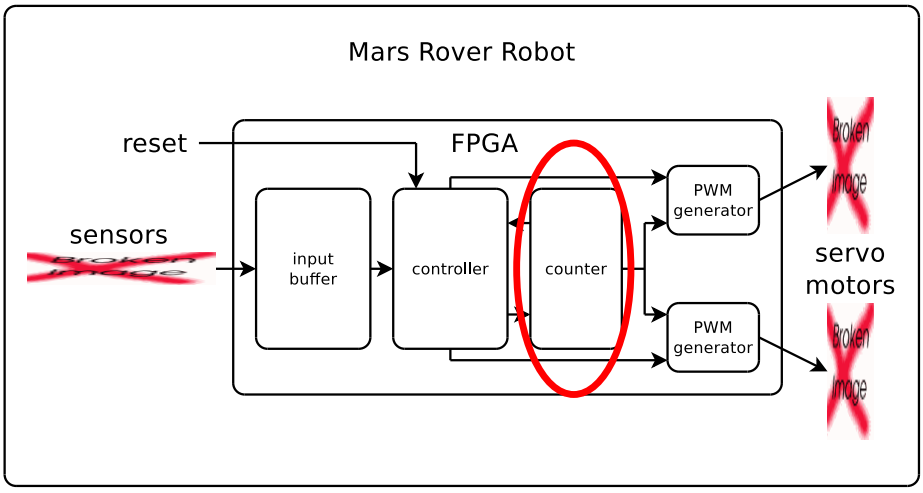
Connect the Signal *count* to the Output Signal *count_out* You should be able to read the currect value of the counter from the output signal *count_out*. To make this possible, you should connect the internal signal *count* to the output signal *count_out*.

Create a Testbench

You need a testbench in order to verify the behaviour of the counter. Create a testbench and use it to simulate the counter. Show the result to a supervisor and explain how you can deduce from the simulation results that the counter works correctly.

4.7 Overview of the Robot

This session you've created the counter. Next session you will implement the PWM generators and, combined with the counter, control the servo motor.



Chapter 5

VHDL: FSM as Controller

Objectives

During this session you will:

- learn what an FSM is
- learn about the differences between the Mealy and Moore style FSM's and the advantages and disadvantages of both.
- learn to design an FSM using state diagrams
- learn to implement an FSM in VHDL
- implement an input buffer
- implement the PWM generators

5.1 Introduction

The PWM generators are an example of controllers. In hardware design, such controllers are generally implemented as an FSM. This chapter teaches you how an FSM works and how to design and implement an FSM. This session you will implement the input buffer and the PWM generators.

5.2 FSM

5.2.1 What is an FSM?

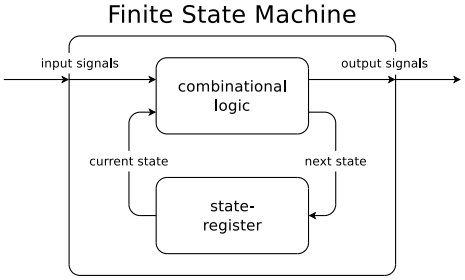
FSM is an abbreviation for **F**inite **S**tate **M**achine, a machine with a finite number of states. An FSM is generally applicable programming technique in which the required behaviour is modelled into a number of states. But what is such a state? A state represents a step in the required behaviour. Under certain conditions, a state is entered and under certain other conditions the state is left.

To clarify this principle, the FSM of a simple robot is given. The robot can only move forward and is equiped with a single sensor. The robot has to move forward

while the sensor outputs the value '1'. The motor of the robot runs when the value '1' is written to it, and stops when it receives the value '0'. This behaviour can be described in two states: one state in which the robot moves and one state in which the robot stands still. The states are changed based on the output values of the sensor.[1]

In order to keep the robot moving while the sensor outputs '1' it is neccessary to keep track of the current state. This means the FSM contains memory to store the value of the current state. The values of the output signals (in this case: the control signal to the motor) depend on this current state and possibly the input values. A new state is selected based on the current state and the values of the input signals. This means we need a formula or description to determine the output values and the new state.

In a software implementation of an FSM, the state is stored in a variable and the values of the output signals are determined by a piece of program code. In a hardware implementation of an FSM, the state is stored in memory (flipflops) and the values of the output signals are determined using combinational logic. The general structure of a hardware implementation of an FSM is as follows:



Every clock cycle the state is recalculated, this means the system always stays in a given state at least for one clock cycle, even if that state is left unconditionally. This way, it is possible to wait for a number of clock cycles.

5.2.2 Moore and Mealy machines

There are two different types of FSMs: Moore and Mealy style. The difference between both is, at least in theory, very simple:

- In Moore style FSMs, the values of the output signals only depend on the current state.
- In Mealy style FSMs, the values of the output signals depend on both the current state and the values of the input signals.

In practice, it can be difficult to determine the type of FSM you're dealing with.

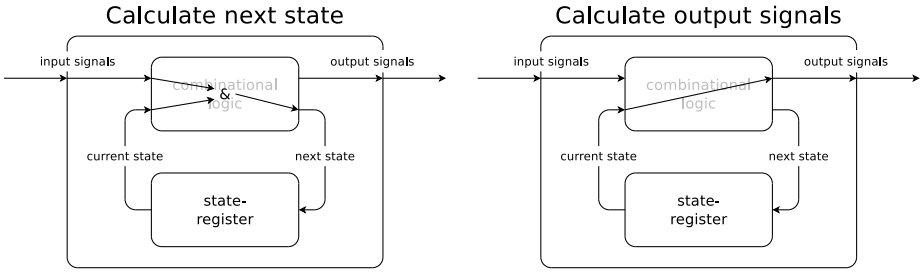
Both designs have advantages and disadvantages:

	Moore	**Mealy**
Advantages	Simple design	Compact design
	Robust design	Sometimes faster than Moore
Disadvantages	Larger design than Mealy	Sensitive to glitches on input signals
	May be slower than Mealy	Can quickly become complex

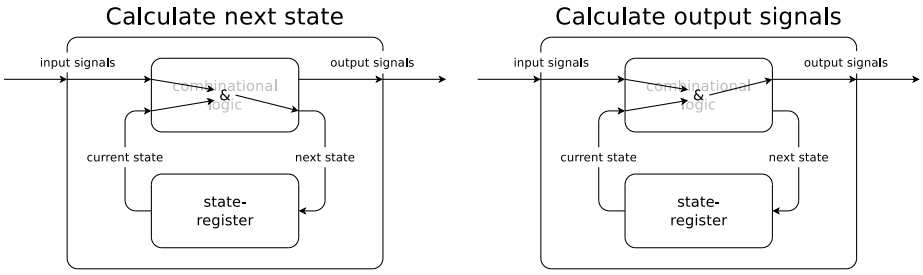
[1] Of course this is a over-simplified example that can also be implemented without an FSM

Although Mealy machines generally have fewer states than Moore machines, it is very difficult to control the disadvantages of a Mealy design. During this lab, you will have to implement Moore machines.

A hardware implementation of a Moore style FSM looks like this:



A Mealy style FSM looks like this:



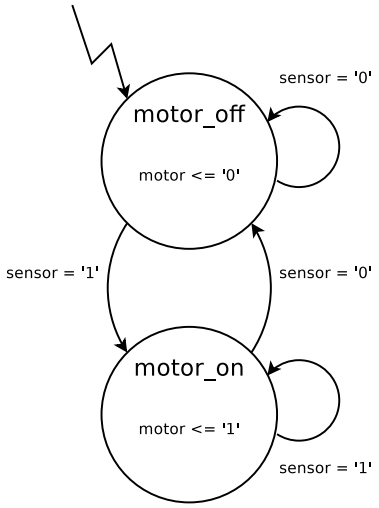
5.2.3 FSM as a state diagram

An important tool in designing an FSM is the state diagram or bubble diagram (named after the shape). The bubbles in the diagram represent the states. Each bubble (state) contains at least the following information:

- Name of the state
- The values of *all* output signals
- One or more inward-pointing arrows indicating under what conditions this state is entered
- One or more outward-pointing arrows indicating under what conditions this state is left

Additionally, a state diagram *always* has an indication of the *reset state*. This is the state in which the system resides after a reset. This reset state in indicated with a “lightning arrow” pointing towards the relevant state. It goes without saying that there can only be one reset state.

In the example of the robot, the state in which the motor is turned off will be named `motor_off`, while the state in which the motor is turned on is named `motor_on`. The state `motor_off` is the most logical choice as the reset state (why?). The complete state diagram will look as follows:



This state diagram meets the requirements. Every state has a name that reflects the effect of the state. This makes it much easier to understand the operation of the FSM than if the states were named `state0` and `state1`.

All bubbles provide the values of all output signals (in this case there is just one output signal).

Every bubble (and thus every state) has two inward-pointing arrows annotated with the conditions under which the state is entered (in this case the values of the input signal *sensor*). One arrow originates from the state itself, this indictes that the state isn't left under certain conditions.

Every bubble has one outward-pointing arrow. This indicates that the states are changed when the sensor values change.

5.2.4 Constructing a state diagram

Now that you know what a state diagram looks like, the question remains how one should design such a diagram (and an FSM). This requires that the required behaviour is described in a number of states. How that happens is basically up to the designer, but by limiting the choice to a Moore type FSM, one option is fixed immediately. In a Moore style FSM, the value of the output signals only depends on the current state, which means that you need at least a separate state for every output combination.

Make sure to use a descriptive name for the different states. This makes it much easier to understand the function of a state. In the example the states had clear names: in one state the motor was turned on, while in the other state the motor was turned off.

When the (basic) states are determined (this includes of course the reset state), you need to decide how the different states interact. Which state can be reached from a given state and under what conditions. In the example, this was also quite easy: if the motor is turned off (state `motor_off`) and the sensor has the value `'1'`, the motor should turn on. This means the state machine should change to state `motor_on`. The reverse argument can be used to turn the motor off.

If it seems impossible to draw a correct state diagram, make sure you're not implementing a Mealy machine instead of a Moore machine! If, for example, the design has 4 output combinations but only 3 states, this is a Mealy machine (Why?).

5.2.5 From state diagram to VHDL description

When the state diagram is correctly designed, it can be implemented in VHDL. This is very easy for a properly designed state diagram. As described earlier, an FSM contains flipflops to store the state and a combinational part to determine the new state and the values of the output signals. The memory is implemented in a way similar to D-flipflops. It is common practice to use a large `case` statement to determine the new state and the output values. This `case` statement selects the current state. You can define both the values of all output signals and the new state within that state selection. To make it easier to describe an FSM, you can use user-defined types. These types may contain the names of the states. By defining the signals *state* and *new_state* as this user-defined type, you can easily refer to the names of the states in the `case` statement.

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity motor_controller is
5  port (  clk     : in    std_logic;
6          reset   : in    std_logic;
7          sensor  : in    std_logic;
8          motor   : out   std_logic);
9  end entity motor_controller;
10
11 architecture behavioural of motor_controller is
12
13         -- the user-defined type motor_controller_state contains
14         -- the names of the states.
15         type    motor_controller_state is (     motor_off,
16                                                  motor_on);
17
18         signal  state, new_state:       motor_controller_state;
19
20 begin
21
22 -- State transitions take place in this process-statement:
23         process (clk)
24         begin
25                 if (rising_edge (clk)) then
26                         if (reset = '1') then
27                                 state   <= motor_off;   -- reset-state
28                         else
29                                 state   <= new_state;
30                         end if;
31                 end if;
32         end process;
33
34 -- The functionality of the described FSM is implemented in this
35 -- process-statement. The sensitivity-list contains the signals
36 -- 'state' (the current state) to determine the output signals and
37 -- 'sensor' (combined with 'state') to determine the new state
38         process (state, sensor)
39         begin
40                 case state is
41                         when motor_off  =>
42                                 -- Determine the outputsignals:
43                                 motor           <= '0';
44
45                                 -- Determine the new state:
46                                 if (sensor = '1') then
47                                         new_state       <= motor_on;
```

```
48                                  else
49                                          new_state       <= motor_off;
50                                  end if;
51
52                          when motor_on   =>
53                                  motor           <= '1';
54
55                                  if (sensor = '0') then
56                                          new_state       <= motor_off;
57                                  else
58                                          new_state       <= motor_on;
59                                  end if;
60                  end case;
61          end process;
62  end architecture behavioural;
```

Since there is a clear relationship between an FSM and a state diagram, there are several pieces of software available to automate the conversion from one description to the other. The Xilinx software provides a way to graphically design an FSM, the Quartus software from Altera is able to generate a state diagram from synthesized code.

5.2.6 Design considerations of FSMs

Finally, a number of things that are easily forgotten and may cause problems with a sometimes difficult to identify cause.

Only use a reset FSM

On startup of the hardware, the flipflops have arbitrary values. This could mean that the FSM is in a state other than the reset state. If, however, you assume the FSM is in the reset state, this could lead to undefined behaviour of the design. To prevent these problems, make sure you always reset the FSM first, eg. by connecting it to the system reset.

Specify all output values in every state

It is important to specify the output values of *all* output values in every state. Failing to do so forces the FSM to store the last assigned value. This is done using a latch. In larger designs, there can be multiple states from which a certain state can be entered. This means that the value stored in the latch could be different from what you would expect. The presence of a latch in the design also renders the timing tools of the synthesis software useless. The sysnthesizer will generate a latch if your VHDL code contains an `if` statement without an `else` block![2] If this happens, you will recieve a warning in ISE. Make sure to read these warnings!

[2] An `if` statement with a `rising_edge()/clk'event` is an exception

Default values
It is possible to assign default values in an FSM. This way it is not necessary to specify the value of a signal in each state separately. These default values are specified in the same `process` block, but should be placed before the `case` statement in which the states are handled. For example:

```
1 process (state)
2 begin
3         -- default value
4         testvalue       <= '0';
5
6         -- state handling
7         case state is
8                 ...
```

It is still possible to assign a different value to such a signal in another state. This works because the statements within a `process` block are parsed sequentially: the signal is assigned the default value first, which is overwritten later on. This is not possible outside a `process` block, since assignments are concurrent in that case.
In general it is wise to use default values only in specific occasions, it is much clearer to assign the values in each state separately.

Specify all relevant input signals in the sensitivity list

All input signals, including the state signal, should be placed in the sensitivity list of the `process` statement. Failing to do so will lead to a situation in which the hardware works generally works fine (the synthesizer is unable to generate hardware for this situation), but in the simulation a reference to a signal that isn't placed in the sensitivity list of the `process` statement is ignored.

Synchronise all input signals with the clock of the FSM

As described earlier, the new state is determined with combinational logic, based on the currect state and the current values of the input signals. As you know from the practical digital systems, the delay in combinational logic can differ per bit. An FSM changes the state on a rising edge of the clock, at that specific moment, the result of the combinational block is used as new state. In this case, due to the difference in delay, it is possible for one one bit of the new state vector to be calculated *after* a rising edge of the clock has occurred, the state machine may not end up in the required state. This is schematically shown in the Figure 5.1.

In this example, the signals *state* and *new_state* consist of two bits. The separate bits of *new_state* are shown as *new_state(0)* (LSB) and *new_state(1)* (MSB). The situation on the left is correct, the sensor output changes and *new_state* has the correct new value before the next rising edge of the clock. In the situation on the right, the sensor value changes at a bad moment: just before a rising edge of the clock signal. Due to the differences in delay in the separate bits of the signal *new_state*, the one bit doesn't hold the correct value yet when the rising edge of the clock occurs. This causes the FSM to end up in a wrong state, namely state 1 instead of state 3. In this specific case, the situation is fixed the next clock cycle, but that depends on the state transition logic in state 1.

This problem can be prevented by synchronising the input signals with the clock of the FSM. This can be done by feeding the input signals through a flipflop (running on the system clock) first. This way, the output of the flipflops changes synchronous

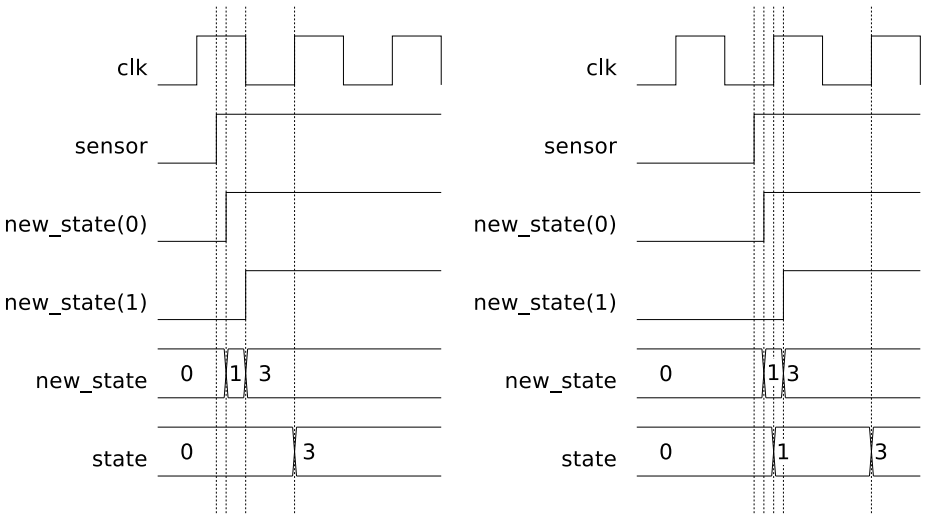


Figure 5.1: Effects of unsynchronized inputs of the FSM

with the system clock. Unfortunately, this is not sufficient to prevent all problems. In order to understand why this might still cause problems, we need to dive deeper into the inner workings of a flipflop. A flipflop needs a certain amount of time in order to copy the input values. This is known as the *setup time*. In order to correctly copy the input values, the input signal should be stable during this setup time. If the input changes during the setup time, the flipflop is said to be in a *metastable* state. During this metastable state, the output of the flipflop is undefined: it will toggle between 1 and 0 or float in the middle of the output voltage range. It is obvious that under these conditions, the inputs of the FSM still aren't synchronised properly. The question remains how one can prevent this problem. The answer is, unfortunately, that this problem can never be prevented 100%. It is, however, possible to minimize the probability of these problems. This can be done by placing a second flipflop directly behind the first flipflop. If in this case the first flipflop enters a metastable state, it has an additional clockcycle to produce a stable output. During this time the second flipflop will output a stable output value. In practice, this is an adequate solution. We will implement such an input buffer in the robot as well.

5.3 Assignments

5.3.1 Implement the Input Buffer

As is discussed in the theory on FSMs, input signals that are used to determine the state need to be stable during a clock cycle. For signals from the outside world, as in this case the sensor values, this can not be guaranteed. A buffer can be used to synchronise these input signals with the FSM. This buffer consists of two cascaded registers working on the same clock signals as the FSM itself. The effect of an input buffer consisting of a single register is shown in Figure 5.2.

Answer the following questions:

- The use of the flipflops has a disadvantage for the reaction rate of the system.

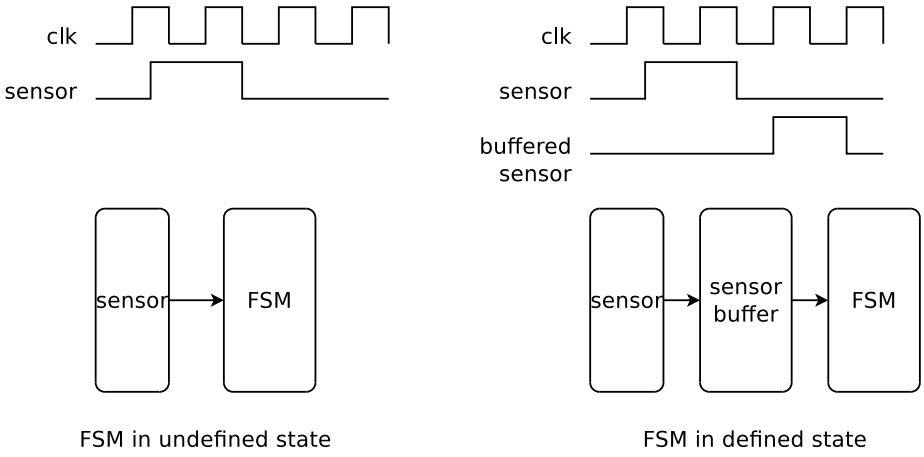


Figure 5.2: Diagram of the effect of the usage of the input buffer

What is that disadvantage and why does it occur?

- Is it likely this will be a problem in the robot?

Implement a 3-bit register

The input buffer consists of two 3-bit registers. This is depicted in Figure 5.3. Implement a 3-bit register, create a testbench and simulate the design.

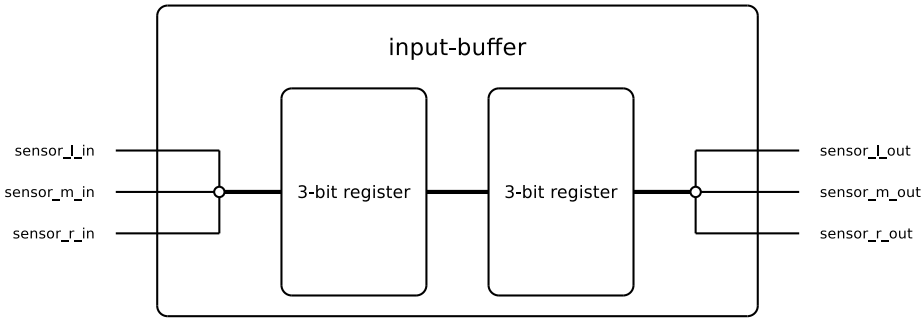


Figure 5.3: Diagram of the structure of the input buffer

Implement the input buffer

Implement the input buffer using a structural architecture. Create a testbench and simulate the input buffer. Does it work according to specifications?

5.3.2 Design and implement the PWM generator

The motor control should be able to switch the motor on or off and make it turn left or right. As described earlier, the servo motors must be controlled with a PWM signal. This signal must meet a number of requirements:

- The frequency of the signal must be 50 Hz
- The duty-cycle must be between 5 and 10%
- The pulse voltage must be between 3 and 5 V
- If the pulse duration is shorter than 1.5 ms, the motor turns left
- If the pulse duration is longer than 1.5 ms, the motor turns right

If the motor should be stopped, there should be no signal coming from the PWM generator.

Design the state diagram of the PWM generator

The PWM generator generates a single PWM pulse that meets the specifications. The entity of the PWM generator should look like this:

```
1 entity pwm_generator is
2         port (  clk             : in    std_logic; -- clock-signal
3                 reset           : in    std_logic; -- reset-signal
4                 direction       : in    std_logic; -- '0' == left, '1' == right
5                 count_in        : in    std_logic_vector (? downto 0); -- value of
                        the counter
6
7                 pwm             : out   std_logic  -- the PWM signal
8         );
9 end entity pwm_generator;
```

The signal *direction* is used to specify the direction. When the PWM generator should be off, the signal *reset* must be kept high.

The counter is used to provide the time reference required to generate the PWM signal. The PWM generator is implemented as a state machine. In one state the signal *pwm* is '0', in another state, that signal is '1'. Based on the value of the signal *count_in* the state machine changes to another state. There is, of course, a separate reset-state in which the output signal is always '0'. After generating a single PWM pulse, the PWM generator should stop automatically, later on the controller will be used to restart the PWM generator when needed.

Design a state diagram of a PWM generator with the given entity description that meets these requirements. Carefully think about the conditions under which the state machine should change its state.

Implement the state diagram in VHDL

Convert the state diagram into VHDL code. Also create a testbench to test the design. Make sure to test at least all possible input combinations in the testbench. Do not forget to reset the system! Simulate the design and verify it works properly. Show the results and the state diagram to a supervisor and explain how you can deduce from these results that the motor control works as required.

You can use the VHDL function `to_unsigned (value, number_of_bits)` to make it easier to compare the value of a signal with a given number. This is discussed in §3.3.7.

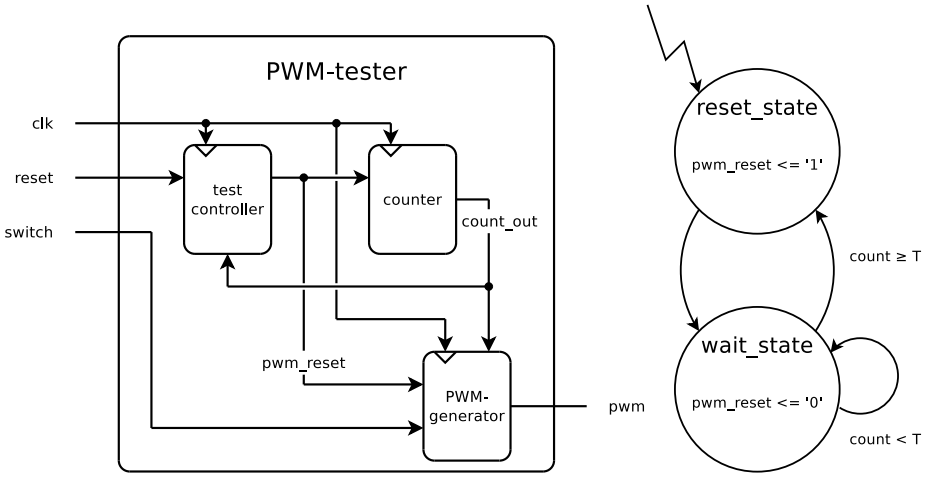


Figure 5.4: PWM tester, on the left a diagram of the structure of the system, on the right the state diagram of the test controller

The input signals of the PWM generators that originate from the controller and the counter aren't synchronised using flipflops. This is not needed, since these signals are already synchronous (same clock domain).

5.3.3 Test the PWM generator on the FPGA board

Now that the PWM generator works correctly in simulation, it can be placed on the FPGA board to do a field test. You can use an oscilloscope to verify if the PWM signals meet the requirements. You can also connect a servo motor when the PWM signals are correct.

Testing the servos requires continuous PWM pulses. Since the PWM generator only generates a single pulse, you need a (simple) controller to restart the generator after one period. Both this controller and the PWM generators use the counter as a time reference. This means the test system consists of three components:

- PWM generator
- counter
- test controller

A diagram of the structure of the test system and a state diagram of a very simple test controller are shown in Figure 5.4.

A switch is used to set the direction of the motor.

Implement the test controller

Convert the state diagram of the test controller into VHDL code. Also create a test-bench and verify the workings.

Implement the PWM tester

Combine the counter, the PWM generator and the test controller into the test system depicted in Figure 5.4. This should be done using a structural architecture. The signal *pwm_reset* from the test controller is used to reset both the counter and the PWM generator.

- implement the structural architecture
- create a suitable testbench to simulate the PWM tester

Explain how you can deduce from the simulation results that the test system and the PWM generators work properly.

Implement the test system on the FPGA board

When the PWM tester works properly in simulation, it can be implemented on the FPGA board. Connect the *reset* signal of the system (the reset of the test controller) to a push button and connect the *switch* to one of the slide switches. Download the constraints file `mars_rover.xdc` from Brightspace and edit the file to connect one of the servos to your servo tester.

Power the FPGA board from the USB connector and *don't* switch on the power board. This will allow you to measure the output of the PWM generator without the servo moving.

The signal *switch* isn't synchronised! Since this is a manual test, this shouldn't be a problem in this case. If the system locks up, it can be reset manually. In a production system, however, make sure to synchronise all input signals!

Verify the System with the Oscilloscope

If the design is synthesised succesfully and the correct pins are assigned, the design can be uploaded to the FPGA board. Use the Basys3 Reference Manual (you can find this on Brightspace) to find out to which physical pin the output of the PWM controller is connected. Connect an oscilloscope to that pin. Do not forget to connect the GND-connection of the probe as well! You can use GND pins of connector `J5` on the power board to that end (the rightmost jumper pins). Use Appendix E if you have difficulties operating the oscilloscope. Verify the workings of the system by checking if the PWM signal meets the requirements. Test both positions of the slide switch.

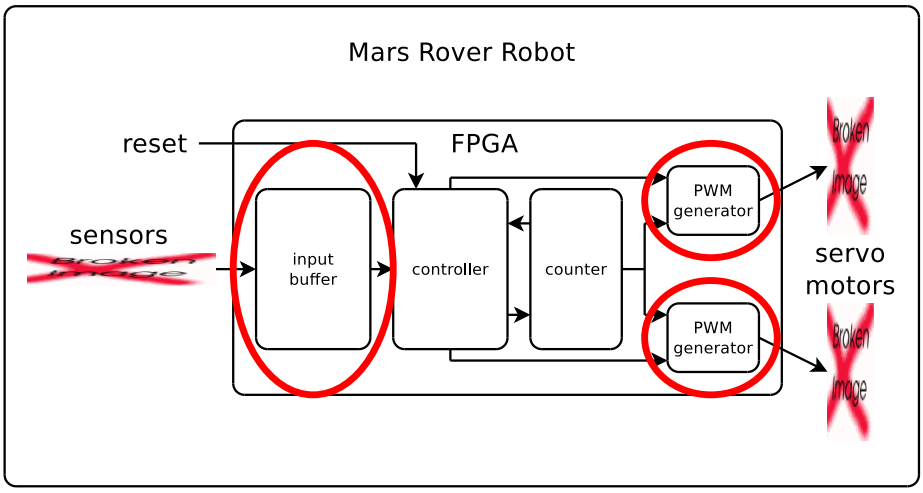
Discuss the results with the supervisor. Only proceed when the supervisor has verified your results!

Verify the System with a Servo Motor

Now that the motor control seem to work correctly, it's time to see if it can actually control a servo motor. Disconnect the FPGA from the USB cable and put the jumper cap on FPGA power selection jumper `JP2` back on `EXT`. Connect a battery to the robot, switch on the power board, and test the servo.

5.4 Overview of the robot

This session you've implemented the input buffer and the PWM generator. This means that all hardware modules, except for the controller, have been implemented. The controller is the brain of the robot and is the component you will work on in the coming sessions. Next session you will implement a very simple controller. With this controller you can test the complete robot.



Chapter 6

Line Tracker

Objectives

In this session you will:

- design, implement, simulate, synthesise and test the controller of the line tracker
- combine the VHDL components of the line tracker and simulate, synthesise and test the line tracking robot

6.1 Introduction

In the previous sessions all the components of the robot have been designed: the input buffer, the counter and the PWM generators. The last remaining part is the controller. The controllor is the "brain" of the robot and largely determines the effectiveness of the robot. When this component is implemented and tested, the different VHDL components can be combined into the first complete version of the robot.

This session you will work on the line tracker, the part of the controller that is used to track the line. Later on you will extend this controller in order to find a line as well.

This chapter doesn't present any new theory, a controller is implemented as an FSM and the required theory was already discussed in the previous chapter. Since designing a controller can be a difficult problem, you will first implement a very simple controller. This enables you to quickly test the complete robot. If the robot functions correctly, this simple controller can be extended to a complete line tracking controller.

6.2 Assignments

6.2.1 Design and Implement a Simple Controller

To avoid losing sight immediately, the first assignment is to design a simplified controller. Equiped with this simple controller the robot should be able to move forward when the sensors see white – black – white and stop when it detects a different input. The controller is a Moore-style FSM, so it follows immediately that you need at least two states (why?).

The first step in implementing the controller is designing the state diagram. Next, this diagram can be implemented in VHDL code and be simulated. When the controller works as required, all components of the robot are available. Next you can use the structural architecture of Chapter 3 as a basis to combine all these components. When this works properly, the code can be programmed into the FPGA. Then the complete robot can be tested for the first time.

Design the state diagram of the simple controller

The simple controller consists of two states:

- one state in which the input of the sensors is mapped onto the next action of the motors
- one state in which the motors are driven

Since after a reset, the controller must determine the next actions, the first state can double as a reset state. In the state in which the motors are driven, exactly one PWM pulse must be generated. The counter is used as a time reference for the period of the pulse.

Design a state diagram of an FSM that meets these requirements. Carefully think about the conditions of a state transition.

Implement and the State Diagram in VHDL Code

Convert the state diagram into VHDL code. Design a testbench and verify if the design indeed meets the requirements.

Combine the Components of the Robot in a Structural Architecture

Create a structural architecture to combine all components of the robot. Design a testbench for the complete system and verify its workings.

Assign the pins, Synthesise the Design, and Program the FPGA

When the system works properly in simulation, you can do a pin assignment of the input and output pins of the system. Next you can synthesise the system and program the FPGA. If all goes well, you should have a robot that is able to track a straight line.

Pin assignments
It is important that you always assign *all* input and output signals to pins. Failing to do so will cause those signals to be connected to random pins. This can lead to undesirable side effects and, at worst, yield damage to the FPGA board.

6.2.2 Design and Implement the Line Tracker Controller

The simple controller is able to track a straight piece of line, but it is not able to track a turn. To this end, the controller needs to be extended.

Table 6.1: Output values for different robot actions

	Motor Left		Motor Right	
Action	**reset**	**direction**	**reset**	**direction**
Forward	0		0	
Turn left				
Turn right				
Sharp turn left				
Sharp turn right				

Design the State Diagram of the Line Tracker Controller

When designing a state diagram of a Moore machine the minimum number of states is equal to the number of output combinations. The line tracker controller controls the inputs of the counter and the two PWM generators and thus basically controls the possible movements of the robot.

The simple controller consists of one state that determines the next state based on the sensor input and one state in which the robot moves forward. You can still use the same design for the complete line tracker by using one state that determines which movement should be selected, based on the sensor input, while the other states control the motors. Each motor can turn forward, turn backward or stop. This means that two motors allow for the following nine robot actions:

- stop (both motors turned off)
- forward or backward (both motors moving)
- turn left or right (one motor standing still, one moving forward)
- turn left or right (one motor standing still, one moving backward)
- sharp turn left or right (one motor moving forward, the other in reverse)

The moving backward or turning backward actions are not used in the line tracker controller, the stop action is not used yet, that leaves the five actions in Table 6.1. Fill in Table 6.1 and create a bubble in the state diagram for each of the possible actions.

In order to complete the diagram, you need to know what sensor values requires which robot action. As a help, fill in Table 6.2, a number of entries have already been filled in. Complete the state diagram by drawing the (annotated) state transition arrows. The robot should be able to track a line with turns with a diameter of approximatly 20 cm. This requires the use of sharp turns. If, however, you only use sharp turns, the robot won't be fast. Carefully think which action to take for each set of sensor values.

Implement and simulate the state diagram in VHDL code

Implement the design into VHDL code and create a testbench to test the controller. Simulate the controller and verify if it meets the requirements.

Table 6.2: Robot actions for different sensor values

Sensors			
left	**middle**	**right**	**Robot Action**
black	black	black	forward
black	black	white	
black	white	black	forward
black	white	white	
white	black	black	
white	black	white	forward
white	white	black	
white	white	white	forward

Combine the components of the robot using a structural architecture

You've already combined the components of the robot with the simple controller, now replace the simple controller with the complete line tracking controller. Simulate the complete robot.

Assign the pins, synthesis the design and program the FPGA

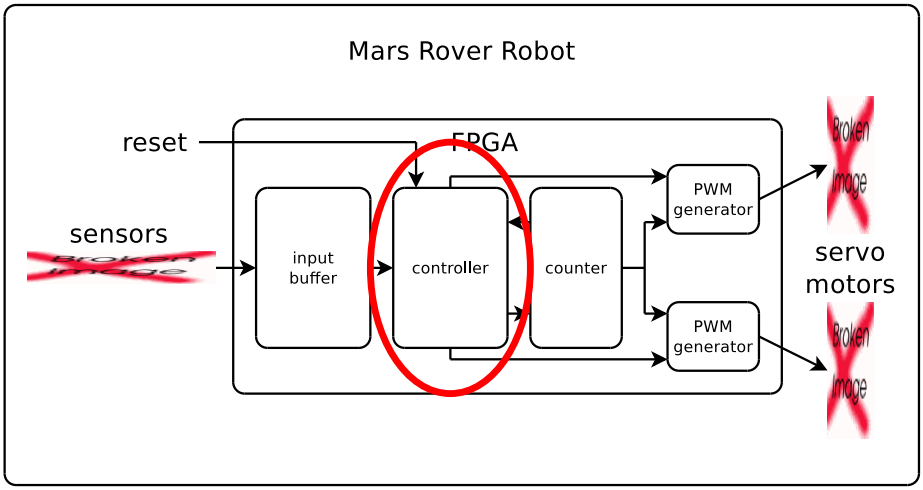
When the system works properly in simulation, you can assign the input and output pins of the system. Next you can synthesise the system and program the FPGA.

Test the robot

Use the test patterns from Appendix C and black tape to test the robot. The robot should be able to succesfully track the test lines 1 – 4.

6.3 Overview of the robot

This session you've implemented the last VHDL module: the controller. With all parts available, they are combined into a complete working robot. The current controller is sufficient to succesfully track a line and thus finish the entire track. In order to find a line as well, the controller needs to be extended. Next session, you will add the line finder to the controller.


Mars Rover Robot
reset
FPGA
sensors
input buffer
controller
counter
PWM generator
PWM generator
servo motors

Chapter 7

Line Finder

Objectives

During this session you will:

- extend the controller with a line finder

7.1 Introduction

The current controller is able to succesfully track a line and thus finish the track. There is, however, no code written to find a line. This session you will extend the controller with a module that will succesfully find a line. After this session, you should have a complete, working controller that will let you pass the practical with a sufficient grade.

7.2 The search for the line

The current controller is able to *track* the line, but it can't *find* the line.

Depending on the angle that the robot crosses the line with, the line is detected with one or more sensors. A number of possible situations is depicted in Figure 7.1. When the line is detected with all three sensors (this is the case when the line is crossed under 90°), it isn't clear what direction the robot should take. In order to keep tracking the

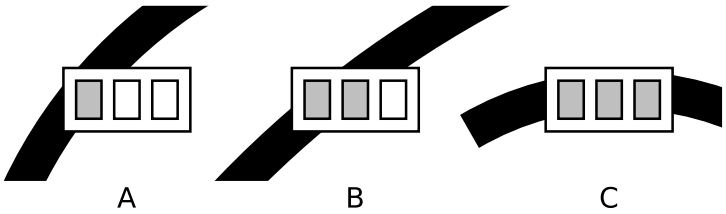


Figure 7.1: Several possible ways the sensors can detect the line: A) with one sensor only, B) with two sensors, C) with three sensors

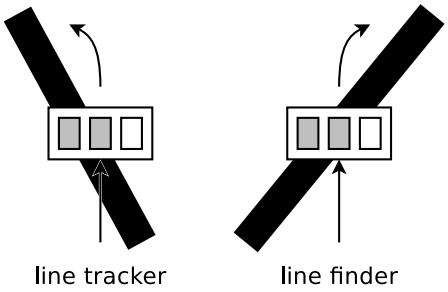


Figure 7.2: Two situations in which the sensors detect black – black – white

line, the robot must make a turn of (somewhat more than) 90° left or right. The choice for going left or right is up to you.

7.2.1 First attempt: re-use of the line tracker

Although it may seem logical to use the line tracking controller once the line is detected, this turns out to be problematic. One of the reasons is the situation described before in which the line is detected with three sensors. Another reason is shown in Figure 7.2. This figure shows two situation in which the sensors detect black – black – white, on the left is the situation of the line tracker, on the right the situation of the line finder. In case of the line tracker, the robot is too far to the right and must turn *left* in order to properly track the line. In case of the line finder, the robot comes from the right and then crosses the line. In order to track the line now, the robot must turn *right*.

A possible solution could be to adjust the line tracker. Unfortunately, this is not easily achievable. One problem occurs when the line is detected with one sensor, for example the left. In order to get the line in the middle of the sensor board, the robot has to steer to the right, but in that case the sensors moves away from the line. If, in that case, the controller has already switched to the line tracker, the robot will steer left and also moves the robot away from the line.

7.2.2 Second attempt: a new approach

In order to easily implement the line finder, we need a new method. This is shown in Figure 7.3. This algorithm consists of four steps:

A Search the line: move in a straight line as long as the sensor detects white – white – white

B Pass the line: move in a straight line as long as the sensors detects the line

C Store the sensor values: store the last sensor values *before* the line was passed

D Turn towards the line: use the stored sensor values to turn towards the line via the shortest path

There is one exception in this system, namely if the robot already starts on the line. In that case, the line is already found immediatly. This new method can easily be implemented and will correctly find the line no matter the angle of incidence.

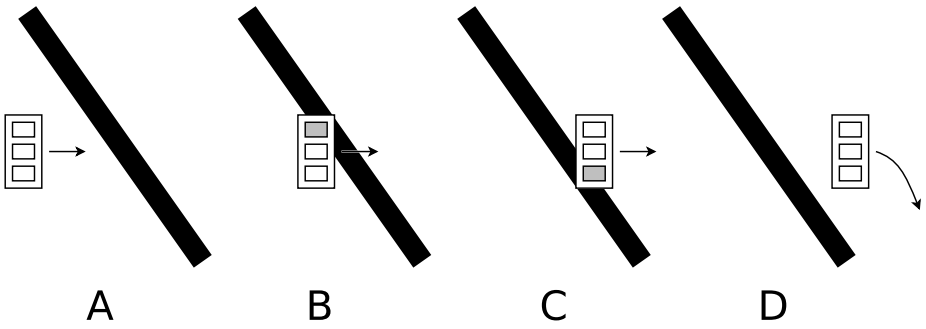


Figure 7.3: A different method for the line finder in four steps: A) find the line, B) pass the line, C) store sensor values, D) turn towards the line

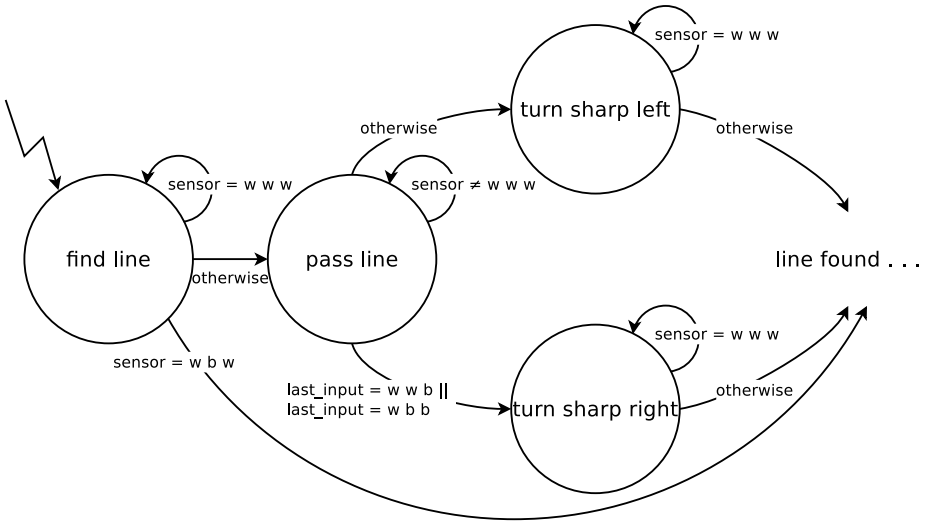


Figure 7.4: Simplified state diagram of the line finder

7.3 Assignments

7.3.1 Design and Implement the Line Finder

This assignment you will use the second method for finding the line. The controller is (of course) implemented as an FSM. A simplified state diagram of the controller is shown in Figure 7.4. Step C of the method requires storing the sensor values. The easiest way to implement this, is by adding additional register in the controller. Then the `process` block working on the clock not only needs to determine the new state, but also updates the register. This `process` block will look more or less as follows:

```
1  process (clk)
2  begin
3          if (rising_edge (clk)) then
4                  if (reset = '1') then
5                          state           <= reset_state;
6                          last_input      <= ???; -- reset-value
7                  else
8                          state           <= new_state;
9                          last_input      <= new_last_input;
10                 end if;
11         end if;
```

```
12 end process;
```

It isn't a good idea to continuously store the current values of the sensors. In that case the important infomation, the values of the sensors *before* the line was crossed, will be overwritten with the value white – white – white. By only storing the values of the sensor when they're not equal to white – white – white, you will store the required information.

Design the State Diagram of the Line Finder

The simplified state diagram of Figure 7.4 can't be used in its current state. Refer for example to Figure 5.4 to see how the motors should be controlled. Extend the simplified state diagram of the line finder into a complete state diagram.

Implement the Register in VHDL

Implement the register in VHDL. The new value of the register (in the example indicated with *new_last_input*) must be defined in every state. Since the value depends on the current value of the sensors (remember we don't want to store the value white – white – white), this could be a lot of work to add to the state machine. This is an example of a situation in which it is easier to use default values (see the hint on page 65).

Implement the State Diagram in VHDL and Simulate the Design

Now that the register is implemented, the state diagram can be implemented in VHDL. Also create a testbench and simulate the line finder.

7.3.2 Combine the Line Finder and the Line Tracker

When the line finder works properly, it can be combined with the line tracker. After a reset, the system should start in the line finder. When the line finder has found the line, the line tracker should take over. There are multiple options to combine the controllers, we will discuss two possible solutions:

- Combining the states of the two controllers into a single large controller;
- Keeping two separate controllers and use an additional controller to select between the two.

Create a single large controller

The simplest solution is to simply combine the states of the line tracker and the line finder into a single large controller. Simply create a `state_type` enumeration type that contains all the state designators of both controllers and handle each state in the combinational part of the state machine VHDL description. Add proper state transition logic between the states of the line finder and line tracker controllers.

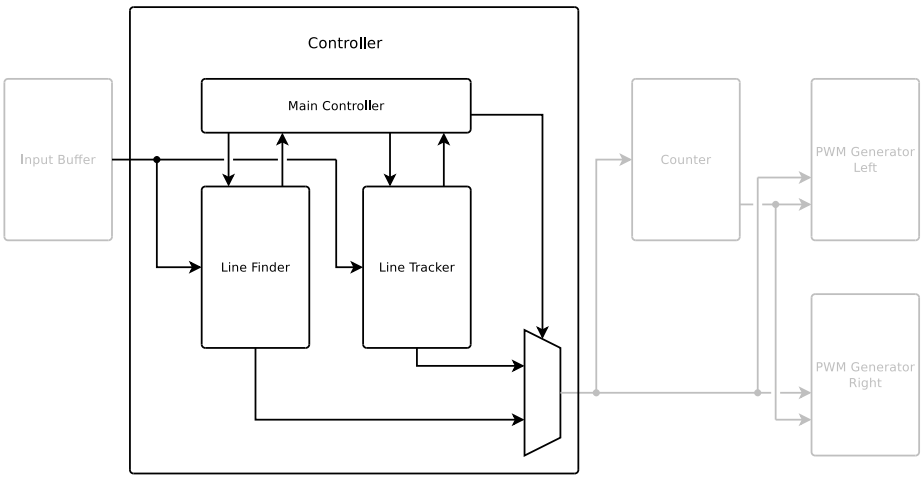


Figure 7.5: Using two separate controllers with a mux and a main controller

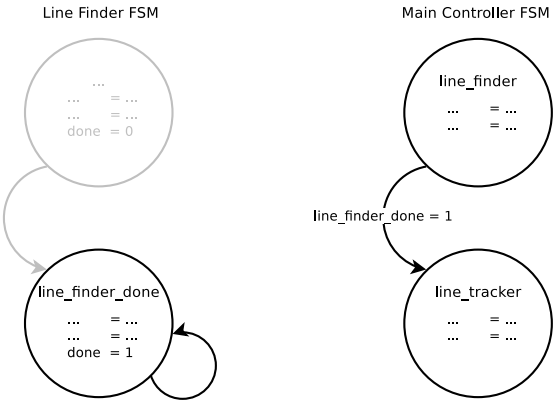


Figure 7.6: Simplified partial FSMs of the line finder and the main controller

Multiple separate controllers

Another option is to keep the two separate controllers. Just like the calculator from the tutorial and the robot itself, the controller can be build out of serveral different blocks. Each controller then uses the inputs of the sensors and writes the same set of outputs (counter and PWM-generator inputs). Of course only a single set of outputs can be used at a time. A mux can be used to select the output of only one of the controllers. An additional controller can now be used to activate one controller (eg. by keeping the other controllers in reset) and controlling the selection input of the mux. A simplified schematic is shown in Figure 7.5. The main controller in this case has at minimum two states: one in which the line finder is active and one in which the line tracker is active (which state could be used as the reset state?). The line finder and line tracker controller now need some end-state and a signal indicating the controller has finished. This signal is used in the main controller to switch between the two controllers. This principle is shown in the simplified partial state diagrams of the line finder FSM and the main controller FSM in Figure 7.6.

By encapsulating the different controllers and the mux into a structural design the

controller has the same interface as the original line tracker controller.

Advantages and disadvantages

Both of these options will work but there are advantages and disadvantages to both. Some remarks on the single controller approach:

- Creating a single controller is simple and involves almost no additional work;
- Adding too many states into a single controller may result in a very large and opague design;
- Simulating a single large controller can be difficult and take a very long time;
- Placement and routing is an NP-complete[1] problem, meaning larger designs take considerably more time to place and route.

Some remarks on the two separate controllers approach:

- Using separate controllers and a mux to select the proper set of outputs gives a very general and extensible solution;
- Implementing initially requires additional work: you need to implement an additional controller and a mux;
- Extending the design is simple and quick;
- The design can easily and quickly be simulated in parts;
- Placement and routing are simpler.

Which option should you choose? In general, the second option leads to a better design: it is conceptually simple, the system is easily testable and extensible, even when the total system complexity increases. The first solution is only feasible for simple situations. In case of the Mars Rover both options will work.

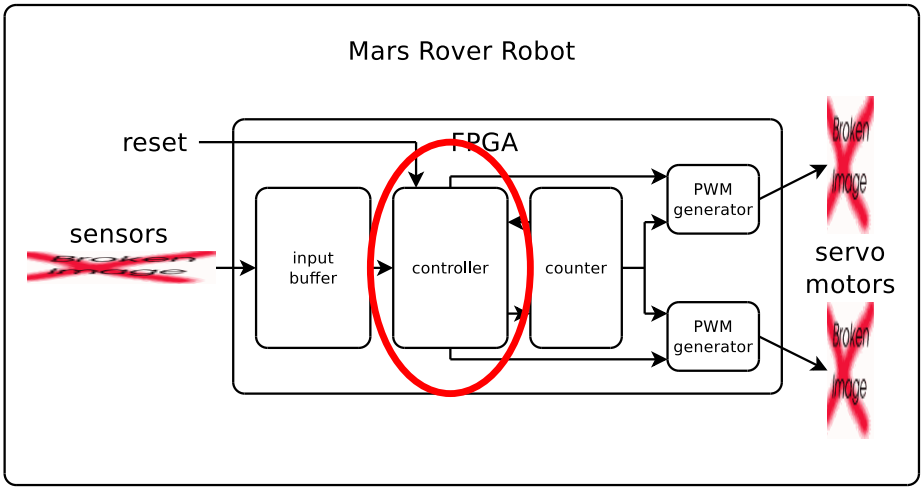
Select the option you prefer and combine the line finder and line tracker controllers.

Combine the components of the robot and program the new version of the controller with both the line finder and the line tracker into the FPGA. Test the system in different situations, such as different angles of incidence. Also make sure the functionality of the line tracker remains unchanged.

7.4 Overview of the robot

This session you've also worked on the controller. This is the last mandatory component of the robot. In order to let the robot perform better, the controller can be extended with support for turn signals.

[1] https://en.wikipedia.org/wiki/NP-completeness


Mars Rover Robot
reset
FPGA
sensors
input buffer
controller
counter
PWM generator
PWM generator
servo motors

Chapter 8

Turn-signals for a Right Turn

Objectives

During this session, you will:

- extend the controller with support for turn-signals for a right turn

8.1 Introduction

The assignments in this chapter aren't required to finish the lab, but they enable the robot to finish the track more quickly. For a greater challenge, the explanation in this chapter is minimal.

As discussed in the introduction of Chapter 1, the track contains extra information to find the exit more quickly. This information is provided in the form of *turn-signals*. Such a turn-signal is code that indicates the robot that a sharp (90°) turn follows.

8.2 Turn-signals for a Right Turn

To unambiguously indicate a turn-signal, we need to use a combination of sensor values that cannot occur normally. In this case, this is the combination black – white – black. If the robot encounters this combination, then the next combination of black – black – black indicates a 90° turn right. A scaled down example of this situation is show in Figure 8.1. The full-size image is included in Appendix C.

When the robot encounters the combination black – black – black, you need to know whether or not a turn-signal was seen in order to decide if you should take a 90° turn right, or just go straight on.

8.3 Assignments

Extend the controller with support for turn-signals. The implementation consists of two steps:

- storing the turn-signal information

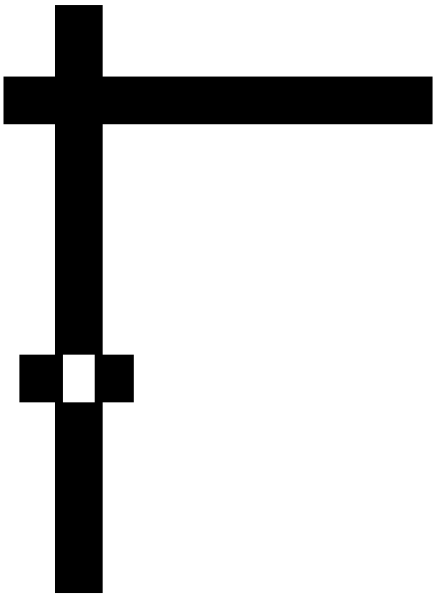


Figure 8.1: Turn-signal for a 90° turn right

- being able to take a right turn of 90°

Storing the Turn-signal Information

Storing the information that a turn-signal had been seen, is comparable with storing the sensor values in the line finder. Use a similar method to store the turn-signal information. Carefully think in which states the signal has to change its value.

90° turn right

Although taking the 90° turn right seems similar to the situation of the line finder, it is not a good idea to implement the same method here. In case of the line finder, the robot kept on turning right until the sensor values white – black – white were found. In case of the turn-signal this can pose a problem when the robot just crosses the crossing. In that case it will also find the code white – black – white over there. Try to imagine yourself why this problem could not occur in the line finder and how you can circumvent this problem for the turn signals.

You will also notice that the robot will not be able to make the 90° turns properly using the turning method of the line finder. Try to come up with a robust method for taking those 90° turns.

- If you've implemented the robot with multiple separate controllers, you can implement the steps to turn 90° right as another controller. You can design and simulate the controller separately. In order to combine the existing controllers and this controller you have to enlarge the mux and extend the main controller to support the new controller.

- If you've implemented the robot with a single large controller, you have to implement the steps as additional states in the controller. You have to simulate the complete controller in order to test these steps.

When this is all implemented, you can test the result using the test patterns in Appendix C.

Chapter 9

Turn-signals for a Left Turn

Objectives

During this session, you will:

- extend the controller with support for turn-signals for a left turn

9.1 Introduction

So far, the turn-signals were used to indicate a 90° turn right. In order to finish the track even quicker, the track also contains information to indicate a 90° turn left.

9.2 Turn-signals for a left turn

There is only one possible sensor value that cannot normally occur: black – white – black. This code, however, is already used to indicate a turn-signal for a 90° turn right. In order to be able to indicate a 90° turn left, we therefore add an additional requirement: when two turn-signals appear in sequence, the next situation black – black – black indicates a 90° turn left.

9.3 Assignments

Extend the controller with support for turn-signals for a 90° turn left. Carefully think about how to implement this: it is less simple than it seems. When you've implemented this, you can test the controller using the test patterns in Appendix C. Carefully check if the controller can differentiate between turn-signals for a turn right and turn-signals for a turn left.

Chapter 10

Improving the Controller: Stop at Finish

10.1 Introduction

After this chapter the robot should be able to stop after reaching the finish of the track. Before you start working on this assignment, you should implement the turn-signal support (Chapter 8 and Chapter 9) first.

10.2 Assignments

10.2.1 Automatically halt on the code white – white – white

The line tracker robot (Chapter 6) was required to drive straight when the sensor readout was white – white – white. This obviously means that the robot just keeps going straight, even after it has reached the finish of the track. It would be nice if the robot would stop automatically at the finish. Just stopping the robot on the sensor readout white – white – white won't work since there are some spots on the track where the robot has to cross a small white – white – white part. In order to make the robot stop at the end of the track, you must be able to differentiate between this situation and the end of the track.

Implement a solution that will stop the robot after it has finished the track.

Part II

Energy Management

Chapter 11

Introduction and Testing

Objectives

During this session, you will:

- familiarize yourself with the power supply chain of the Mars Rover
- determine the power requirements of the Mars Rover
- examine the solar panel
- examine the buffer

11.1 Introduction

The current Mars Rover is able to navigate Mars[1]. There is, however, one big problem: you have to replace the batteries every now and then. In order to be able to use the Mars Rover for an extended time, an alternative power supply is needed. For the Mars Rover one can use a solar cell. The solar cell and the robot work on different voltage levels and the output of the solar panel is highly dependent on the intensity of the incident light. This requires some additional hardware in order to match the voltage levels, allow for some buffering and the possibility of switching to a backup power supply.

11.2 Overview of the *Energy Management* Part of the Lab

In its simplest form, the complete power supply consists of two power supplies, the solar cell (with support electronics) and the backup power, a switch to select the either of these supplies and a control circuit to drive the switch. This is shown in Figure 11.1. In this chapter you will first examine the power requirements of the different parts of the robot, then you will examine the battery and the solar panel with the buffer. In Chapter 12 you will build the controlled switch. In Chapter 13 you will work on

[1] Well, actually, it can find and track a line

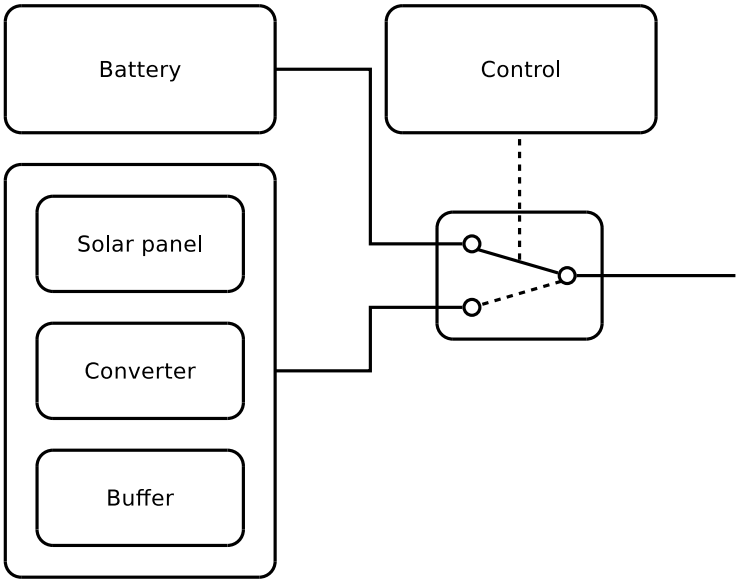


Figure 11.1: Overview of the power supply of the Mars Rover

different designs for the converter, including a linear regulator and a so-called buck converter.

11.3 Power Board

The Power Board that is used in the Mars Rover robot is designed to support the Energy Management part of the lab. The schematic of the Power Board can be found on Brightspace. The board contains a buck converter that will convert the 7.2 V battery voltage to a 5 V output voltage. The board contains two red connectors, VBATT and VIN. VBATT is the 5 V buck converter output. In addition the black GND connector provides a ground connection. The FPGA board and the servos are powered from VIN. Thus, when running on batteries VBATT is connected to VIN. An alternate power source can be connected to the VIN and GND connectors.

When both the solar panel and the batteries are used the Power Board additional hardware is required. In case of a design error, the Power Board tries to prevent potential danger:

- In case of an overcurrent a so-called crowbar circuit[2] is activated to quickly blow the fuse;
- In case of a high input voltage on the VBATT connector diode D11 (see schematic) will block current.

Of course these safety measures should only be a last resort! If you have any doubts on the safety of your own circuit have it checked by a supervisor before you connect it to the robot.

[2] https://en.wikipedia.org/wiki/Crowbar_(circuit)

Batteries
Be carefull when working on the Energy Management part of the lab! The batteries are quite powerful and shorting the batteries may result in severe burns or worse.

11.4 Power Requirements

The solar panel is used to power the robot, but can it actually power the robot? In order to answer that question we start with measuring the power requirements of the robot. Since the robot is powered from a source present on connector VIN we can use this connector to measure power consumption of the robot with multimeter.

What is the open clamp voltage present on the VBATT connector?

Answer:_____

Assuming the voltage is constant, the power consumption can be determined by measuring the average current. Use the Fluke 177 multimeter to measure this average current. Use the following steps:

- Use the 10 A current measurement connectors;
- Use the dial to select DC current measurements for the appropriate scale;
- Press the MINMAX button in order to measure the minimum, maximum and average value.

With the multimeter connected have the robot follow the track. When the robot has finished you can press the HOLD button to freeze the display. Repeatedly press the MINMAX button until the display shows AVG in order to display the measured average voltage.

What is the average current the robot draws while tracking the line?

Answer:_____

When is the average power consumption of the robot?

Answer:_____

If you want to model the robot with an equivalent circuit consisting of a single resistor, what would be the value of that resistor?

Answer:_____

11.5 Power supply

Now that the power requirements are clear, it's time to look into the power supply. As shown in Figure 11.1, the power supply actually consists of two separate supplies: the

batteries and the solar panel. First, you will examine the batteries and then the solar panel supply. The solar panel supply consists not only of the solar panel itself, but also a converter and a buffer.

11.5.1 Batteries

Up until now, the robot was powered by the batteries. Batteries are an example of a non-ideal voltage source that can be modelled by an ideal voltage source with a non-linear source resistor. However, the robot does not use the 7.2 V battery voltage directly. Instead a buck converter is used to decrease the output voltage to 5 V. This will also significantly lower the effect of the source resistor. The output voltage will however be influenced by the protection diode `D11` of the power board.

Measure the open clamp voltage of the battery voltage on connector `VBATT`:

Answer:_____

Use the schematic of the Power Board to find the type of diode `D11`:

Answer:_____

Use the datasheet of the diode. Given the average current measured in §11.4 what voltage would you expect on `VBATT` under average load?

Answer:_____

In §11.4 you've calculated the value of the resistor in the single resistor equivalent circuit of the robot. Connect a power resistor of this value to `VBATT` and `GND` and measure the voltage over this load resistor. What voltage do you measure? Does this match your expected voltage from the previous question?

Answer:_____

This non-ideal behaviour of the batteries will complicate some of the circuits you have to design later on.

11.5.2 Solar Panel

Basically, a solar cell is a diode constructed such that photons (of light) can increase the reverse current I_s; that is, the current through the diode when i and v are both negative. The solar cell operates in a so-called photovoltaic mode, that is, it is unbiased and connected to a load impedance. The energy of the incoming light will cause a current through the diode in additon to the current caused by the voltage across the diode. The solar cell is assumed to have an ideal diode characteristic.

The total iv-characteristics of a solar cell under illumination is simply a summation of the dark current I_s (current in the dark) and the photocurrent given as:

$$I = I_s \left[exp\left(\frac{eV}{kT}\right) - 1 \right] - I_{SC}$$

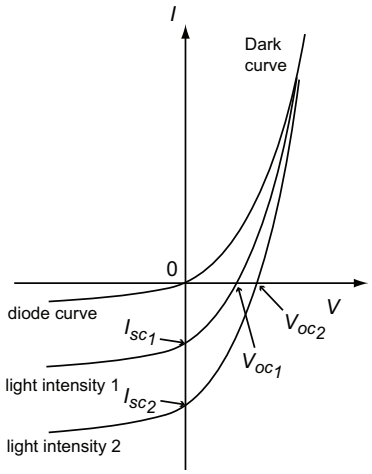


Figure 11.2: *iv*-characteristic of a solar cell showing short-circuit current and open-circuit voltage as the light intensity increases. In the fourth quadrant the power is negative, meaning that we have the behaviour of an active nonlinear resistor that generates energy.

The result is shown in Figure 11.2.

As $V = 0$ on the vertical axis, the intersections with this axis are called short-circuit currents and are indicated by I_{SC}. The intersections with the horizontal axis ($I = 0$) are the open-circuit voltages V_{OC} representing the terminal voltage across the solar cell in absence of a load (also called photovoltaic voltage). The shifted *iv*-characteristic shows that in the 4^{th} quadrant the current *in* the solar cell is negative (passive sign convention) while the voltage across the solar cell is positive; the solar cell is delivering energy to a load and can be used as an energy source for a subcircuit (the solar cell behaves as an active nonlinear resistor in this quadrant). In general, the short-circuit current is a linear function of the illumination, while the open-circuit current is a logarithmic function of the illumination. The major increase in V_{OC} occurs for lower-level increases in illumination, thereby increasing the power capabilities. However, further increasing the voltage across the solar cell decrease the power until the voltage reaches the open-circuit voltage where the solar cell cannot provide any energy anymore. For a silicon single solar cell element the maximum voltage is typically 0.6 V. A series arrangement of solar cells permits a voltage beyond that of a single element.

IV-curve

Before you can optimally use the solar panel, you have to determine the behaviour of the panel under different circumstances. On the back of the solar panel, there are some specifications. Answer the following questions:

- What is the rated nominal output voltage of the panel?

  Answer:_____

- What is the rated nominal output current of the panel?

  Answer:_____

- What is the rated nominal output power of the panel?

  Answer:_____

These specifications are only met in ideal situations. Characterize the solar panel as used in the lab setup by doing measurements. Since this lab takes place in winter, a lamp will be used as light source.

Lamps
There are two types of lamps available:

- Halogen lamps
- Metal-halide lamp

The halogen lamps are high wattage (500 W) and produce a lot of heat and will also significantly heat up the solar panel. Switch off these lamps as soon as you're done with a measurement!
The metal-halide lamp is a gas discharge lamp. This lamp uses a lot less energy (150 W) but has a cooldown period: it cannot be switched on and off too fast. It needs to cool down before being turned on again, so do *not* switch off this lamp untill you've done all your measurements!
Both lamps have a comparable effect on the generated power.

Draw the IV-curve of the solar panel. How can you do the measurements needed to draw the IV-curve? When you've done the measurements and drawn the IV-curve, answer the following questions:

- Does the solar panel behave like voltage source or a current source?

  Answer:_____

- What is the open-circuit voltage?

  Answer:_____

- What is the short-circuit current?

  Answer:_____

- What is the maximum output power of the panel?

  Answer:_____

The point at which the maximum output power is reached, is called the Maximum Power Point (mpp). When does the panel work in the mpp? What are the voltage V_{mpp}, the current I_{mpp} and the power P_{mpp} at which the panel works in the mpp?

Answer:_____

Adjust the amount of incident light and figure out the influence of the amount of incident light on the IV-curve and the mpp.

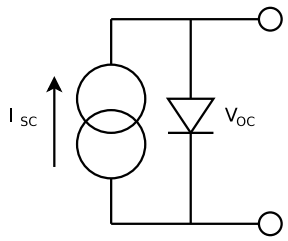


Figure 11.3: Simplified equivalent circuit of a solar cell under illumination.

Equivalent Circuit

The measurements show that for small voltages the solar cell behaves as a current source. Of course, for too high voltages this no longer holds (why?). The solar panel can be modelled with the circuit of Figure 11.3. The constant current source that provides the short-circuit current is connected in parallel with a diode that has a forward voltage equal to the open-circuit voltage.

First Evaluation of the Solar Power Supply

Now that you've done measurements on the solar panel and the robot, can you explain whether or not the robot will be able to run soley on the solar power supply?

Answer:_____

11.5.3 Buffer

The solar panel requires incident light in order to work. Although there are no clouds on Mars, the shade from rocks etc could block or at least dim the light. In order to compensate for these short interruptions in the power supply, we could add a buffer. This buffer will charge when there is a surplus of energy and will discharge when there is an energy shortage. A possible solution could be to add a rechargable battery, but batteries generally can't properly handle the quick charge/discharge cycles we could expect for this application. Instead we will use a capacitor bank, consisting of a series connection of three 50 F, 2.3 V super capacitors. A photo of such a super capacitor is shown in Figure 11.4.

For safety reasons, the super capacitors are placed in a box. The capacitors are placed in a circuit that will protect against mild abuse, but be carefull when you handle these components. Keep the following warnings in mind:

Capacitors can be dangerous!
Capacitors are used to store energy. When unconnected, capacitors can stay charged for extended periods of time. When you short the pins of a charged capacitor, all this energy will be released. This can result in a dangerous electrical shock! Always be careful when handling these devices! Make sure you discharge any used capacitor by connecting it to a discharge circuit (ask a supervisor).

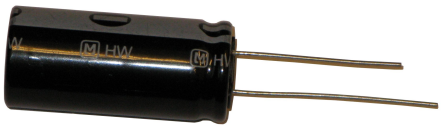


Figure 11.4: Photo of a super capacitor: on top the minus-sign is visible, the long pin is the plus

Observe polarity!
The capacitors used in the buffer are electrolytic capacitors. These capacitors have a polarity, the minus is shown on the capacitor itself, and (on a new capacitor) the plus pin is slightly longer. This is shown in Figure 11.4. Be careful to observe this polarity! If you reverse the polarity, the capacitor may explode! The electrolyte in the capacitor isn't nice material, you don't want to get it in your face, hair or eyes!

Cables
When you test the capacitors, large currents will run through the connecting cables. Do not use the thin cables with the miniature clips for testing, these cables can't handle these currents and will melt!

Super Capacitors
The buffers need to be able to store a large amount of energy. Normal capacitors have a very limited capacitance (order of magnitude several 100 µF – 1 mF), but there exist special so-called super capacitors with a much larger capacitance (1 F - 100 F). This comes at a price: the maximum allowed voltage across these capacitors is low (several V, for normal capacitors this is between 20 V to 400 V). Exceeding this maximum voltage could result in an explosion of the capacitor! Also, the previous warnings on capacitors hold even more strongly for these super capacitors!

Capacitance and Maximum Energy Storage of the Buffer

The three super capacitors are connected in series. A 6 V, 5 W zener-diode[3] is added as protection against overvoltage and incorrect polarity. The circuit is shown in Figure 11.5. The 47 kΩ resistors are in place to ensure the voltage is distributed evenly over the capacitors. Just as combining resistors in a series connection will alter the resistance of the circuit, combining the capacitors in a series connection will have an influence on the total capacitance. Determine the capacitance of the series connection of the three capacitors.

Answer:_____

How much energy can be stored inside this capacitor bank?

Answer:_____

[3] http://en.wikipedia.org/wiki/Zener_diode

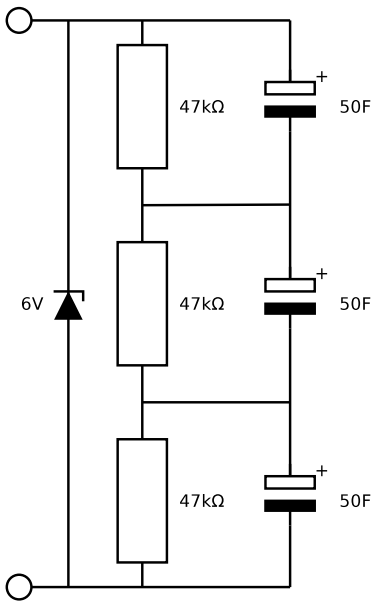


Figure 11.5: Circuit of the buffer

Assume the capacitors are fully charged initially. Assuming the robot uses average power, for how long can it work on the capacitors if it can deplete them completely?

Answer:

Charge and Discharge time

Charging the capacitors takes time. Calculate the time required to charge the capacitors when the capacitors are charged from a current source with a current of 1 A. Use [2] for formulas on the charging and discharging capacitors.

Answer:

The output current of the solar panel is less than 1 A, and the voltage is also different. Assume you are able to convert the output of the solar panel to the correct voltage with 100% efficiency. How long would it take then to charge the capacitors?

Answer:

The robot cannot work when the voltage is too low. The voltage over the capacitors will decrease when they are discharging. Assume the capacitors are fully charged initially (what is the voltage over the capacitors in that case?) and the robot can only work when the supply voltage is at least 5 V. Use a power balance to determine how long can the robot work when running on the capacitors, assuming it draws average power?

Answer:

The maximum power consumed by the robot depends on the supply voltage (why?). This supply voltage will change when the capacitors discharge. Again, assume the capacitors are fully charged initially and the robot can only work when the supply voltage is at least 5 V. Use the equivalent resistance model of the robot and determine how long the robot can work.

Answer:

Can you explain the difference between the answers to the last two questions?

Answer:

Measurements

Do some measurements on the capacitors to see how the calculations match reality.

Use a lab power supply to charge the capacitors. A lab power supply behaves like a voltage source, you should not try to charge the capacitors by directly connecting them to the power supply. You have to use a resistor in series with the capacitors. The maximum voltage over the capacitors is fixed with the zener diode, the current has to be limited to a maximum of 1 A using the resistor. Use a voltage of 6 V and a 6.8 Ω resistor. The capacitor has to be charged to 5.9 V (why not to 6 V?). First calculate the time required to charge the capacitors.

Answer:

Switch off the power supply and build the charging circuit. Connect a multimeter to measure the voltage over the capacitors. Have a supervisor verify your circuit before switching on the power supply! Measure the time required to charge the capacitors to 5.9 V. Do the calculations and measurements match?

Answer:

Connect the super capacitors to the robot (charged to 5.9 V). Use a voltmeter to measure the voltage over the capacitors and measure the time the robot can work on the buffer. What is the minimum voltage level at which the robot can still function?

Answer:

Calculate the time required to charge the capacitors back to 5.9 V using the solar panel.

Answer:

Chapter 12

Controllable Power Supply Switch

Objectives

During this session, you will:

- Determine the characteristics of different types of MOSFETs
- Design and build the power supply switch
- Build a comparator with hysteresis

12.1 Introduction

The output power of the solar cell will depend on the amount of incident light. Under certain conditions (night, shade) the output power of the cell won't be sufficient to drive the Mars Rover. For a brief power shortages, the buffer will power the Mars Rover. The buffer only has a limited amount of energy stored, it cannot supply enough power for extended periods of time. In that case the Mars Rover has to fall back on the battery pack as its power supply. This means we need a controllable switch to select the required power supply. You will first build the switch itself and then implement the control signal.

12.2 Power Supply Switch

The switch should be able to select between the two power supplies: the solar panel (with the additional circuitry) and the battery. The switch should be controllable by a digital input in order to switch power supplies when certain criteria are met. This is depicted in Figure 12.1.

Ideally, the switch should have the following properties:

- Resistance of the switch in on-state should be very low ($\rightarrow 0\,\Omega$)
- Resistance of the switch in off-state should be very high ($\rightarrow \infty\Omega$)

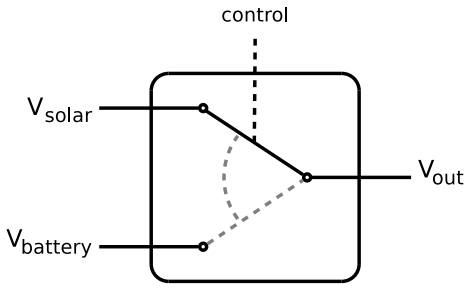


Figure 12.1: Simplified representation of the switch

- Resistance between the two inputs should be very high ($\rightarrow \infty\Omega$)
- Power required for control should be very low ($\rightarrow 0\,\mathrm{W}$)
- Switching should be very fast

There are several options for implementing the switch:

- Mechanical relay
- Solid state relay
- Transistors

The traditional mechanical relay uses an electromagnet to move a physical switch. It has a very low on-resistance and an almost infinite off-resistance, but it requires a current through the electromagnet to hold the switch in the required position (control power >> 0). Due to the mechanical nature of the system, switching is relatively slow. Other disadvantages, especially for the Mars Rover, are that the switch is relatively large and sensitive to bounces (a large enough shock could (temporarily) toggle the switch). Many of these disadvantages are overcome by the use of a solid-state relay. Such a relay internally consists of transistors (MOSFETs). In this lab we will implement the switch with MOSFETs.

12.2.1 Different MOSFETs

There are several different types of transistors. For the application of the switch, the *MOSFET* is the most suitable choice. When used as on/off switch MOSFETs have the following properties:

- The on-resistance is $< 1\,\Omega$
- The off-resistance is $>> 100\,\mathrm{M}\Omega$
- Switching times < 1 ms (can even be improved by using MOSFET drivers)

There are four different types of MOSFETs, these are shown in Figure 12.2. Before you can build the switch, you will need to know the characteristics of the different devices. Those of you who did the lab “Electronic Circuits for Aerospace Engineering” (code ET3604LRP) have already done some measurements on an enhancement type NMOS. In this lab we will do some simulations using the Spice simulator and the Spice-models provided by the MOSFET manufacturers.

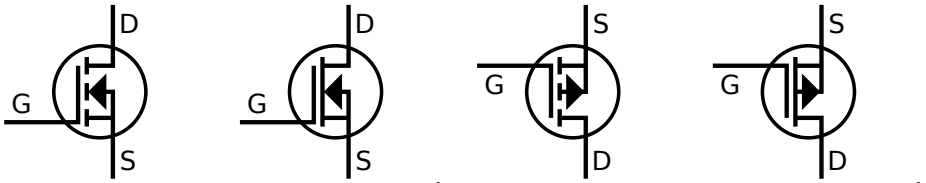


Figure 12.2: Different types of MOSFETs: a. NMOS enhancement, b. NMOS depletion, c. PMOS enhancement, d. PMOS depletion

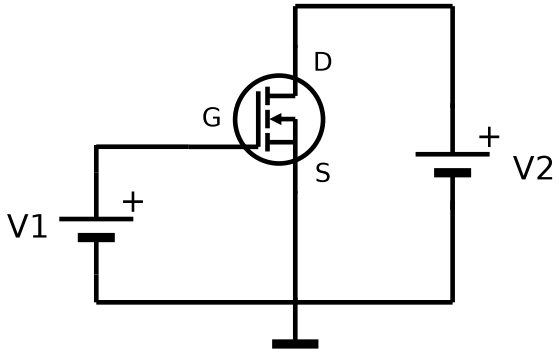


Figure 12.3: Simulation circuit for NMOS

NMOS enhancement MOSFET operation

The most common type of MOSFET is the NMOS enhancement MOSFET. Such a MOSFET has three terminals[1]: the gate (G), the drain (D) and the source (S). The voltage V_{GS} provides a way to control the current through the drain I_d. If the voltage V_{GS} is larger than a certain value ($V_{GS,\,th}$, the gate-source threshold voltage), a current I_{DS} will start flowing from the drain to the source. Depending on the magnitude of the voltage V_{DS}, the drain-source voltage, the MOSFET behaves differently. Using a simulator, you can analyze the MOSFET and determine the different working regions of the MOSFET.

Use a Spice simulator (you can find an introduction to PSpice on Brightspace) to simulate the circuit of Figure 12.3. Besides PSpice, you can use other spice simulators, e.g. LTSpice (Windows) or NGSpice (Linux).

Add GND symbol
Do not forget to add the GND symbol, otherwise simulation will fail!

Operating regions The results of the simulation show different behaviour of the MOSFET for a different value of the voltage V_{DS}. For low values of V_{DS}, the MOSFET operates in the *Ohmic* region, for high values of V_{DS} in the *saturation* region. In

[1] actually, a MOSFET is a four-terminal device, but in most configurations the fourth terminal, the bulk, or body, is internally connected to the source

the Ohmic region, the current I_D depends on both the voltage V_{DS} and the voltage V_{GS}. In the saturation region, I_D mainly depends on the voltage V_{GS}. In this application the MOSFETS are used as on/off switches, which region would be preferred for this purpose?

Answer:

Testing the circuit You can now test the circuit, but you'll have to limit the current I_D. For this purpose several low-ohmic, high wattage resistors are available. Use two separate power supplies and make sure the resulting circuit has a single ground potential. Have a supervisor check the circuit!

Power MOSFET MOSFETs can be used as logical switches, but special versions, so-called power MOSFETs, can be used to switch large currents. In order to be able to handle these large currents, power MOSFETs are build a bit differently than normal MOSFETs[2]. One consequence of this is the presence of a so-called *intrinsic diode* from drain (anode) to source (cathode). This diode will conduct when the voltage V_{SD} is positive, no matter the value of V_{GS}. Test the presence of this diode by applying a positive voltage V_{SD} for different values of V_{GS}.

Power MOSFET in the circuit simulator
The simulator from `http://www.falstad.com/` does not have a model for a power MOSFET with an intrinsic diode, however, you can easily use the normal MOSFET model and add the diode manually. The presence of this diode affects the circuit you need to build, so make sure you do add it in your simulations!

PMOS enhancement MOSFET operation

You can now also perform the same simulations and tests on a PMOS MOSFET. Make sure to test the PMOS MOSFET in simulation first!

12.2.2 Design and implement the switches

Now that you've examined the building blocks available, you can design the controlled switch. The switch is controlled by a digital input signal. It is up to you to decide the meaning of '0' and '1' of the control signal. You don't have to implement the control signal yet. Simulate the circuit with the circuit simulator and discuss the results with a supervisor before building the circuit.

Intrinsic diode
Make sure to add the intrinsic diodes to the MOSFETs in the simulation!

What type of MOSFETs do you need to use in this circuit and why?

Answer:

[2] See `http://en.wikipedia.org/wiki/Power_MOSFET`

How do the intrinsic diodes influence the way the circuit works? What can you do to compensate?

Answer:

12.3 Control signal

The switch has to be controlled with a digital control signal. The switch has to switch between the batteries and the solar supply based on their relative voltage levels. We will use a comparator to compare these voltage levels.

12.3.1 Comparator

In the lab the LM239 comparator is available. Download the datasheet of this component from Brightspace and study it before proceeding.

Testing the comparator: output

Test the functionality of the comparator by building a simple test-circuit. Use two power supplies for the different input voltages. Can you get both a low and a high output value? If the high output value seems wrong, carefully study the output stage of the equivalent circuit of the comparator (see the datasheet). What happens to the output when the the voltage on the base of the output transistor is high? What happens to the output when this base voltage is low?

Answer:

Testing the comparator: similar inputs

What happens to the output value of the comparator when both sources have more or less the same voltage level? Is this a problem when this signal is used to switch the MOSFETs?

Answer:

Testing the comparator: supply voltage and input levels

The comparator requires a supply voltage to operate. Which supply is the best suited for this task? How does this choice influence the voltage levels that can be compared?

Answer:

12.3.2 Comparator with hysteresis

When the result of a comparison of the supply voltages is used to switch between the two sources, the robot will very quickly alternate between running on batteries and running on the buffer (why?). Instead of switching based on the result of a single comparison, we would like to have different levels for switching to the solar supply and for switching back to the batteries.

You've done measurements on the charge time of the capacitors, the voltage levels the robot can work on etc. Based on these measurements, answer the following questions:

- When running on the batteries, for what voltage level of the buffer would you switch to the solar supply?

  Answer:

- When running on the buffer, for what voltage level of the buffer would you switch back to the batteries?

  Answer:

In this case the output of the comparator should not only depend on the voltage level of the buffer, but also on the internal state of the system. This is called *hysteresis*. A graphical representation is shown in Figure 12.4. When the output value is initially high, the output will change to low for a certain input voltage $V_{th,high}$, the high threshold value. Then, when the output is low, it will change back to high for a different (lower) input voltage $V_{th,low}$.

Implementing such a system can be done in different ways, here two possible implementations are described in more detail:

- A complete analog implementation known as *Schmitt trigger*
- A mixed analog/digital implementation using a state machine

Either solution will work, you can choose which one you want to implement. The Schmitt trigger is described in §12.3.3, the mixed mode implementation is described in §12.3.4.

12.3.3 Hysteresis with a Schmitt trigger

Hysteresis can be implemented with a single comparator by connecting the output to the non-inverting input of the comparator. This is schematically shown in the circuit

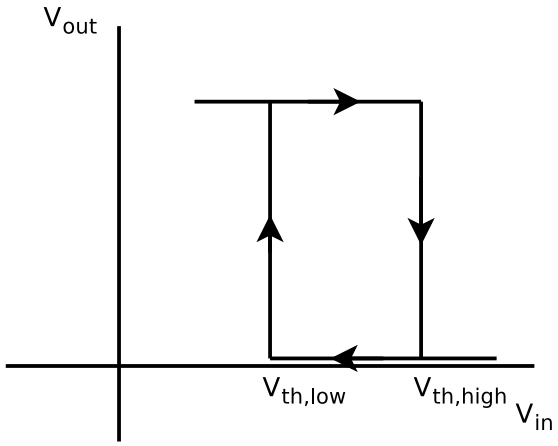


Figure 12.4: Graphical representation of the hysteresis effect

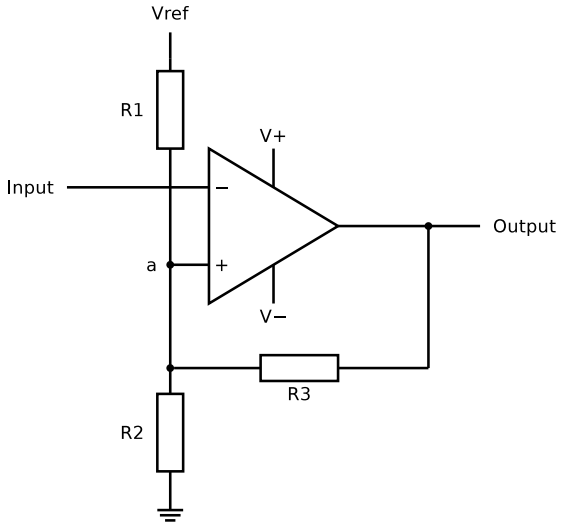


Figure 12.5: Inverting Schmitt trigger circuit

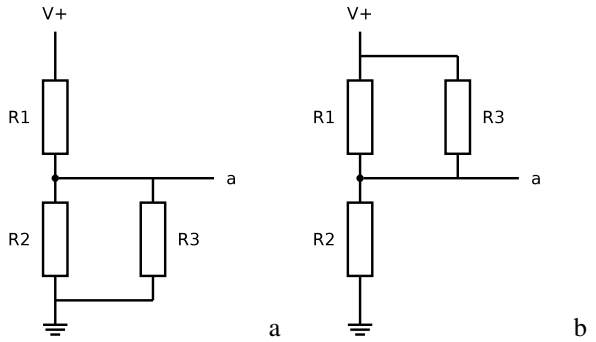


Figure 12.6: Schmitt trigger circuits when $V_{\text{ref}} = V+$ and $V- = 0\,\text{V}$: a. low output voltage, b. high output voltage.

of Figure 12.5. By coupling the output to the non-inverting input, the output will very quickly clip to the supply voltage, so the output can only be high (V+) or low (V-).

Op-amp used as comparator
When an op amp is used as a comparator, the assumption that there is no voltage drop over the inputs no longer holds!

Analyzing the Schmitt trigger

When analyzing the circuit of Figure 12.5, you can write down the KCL (Kirchhoff Current Law) for node a as follows:

$$\frac{V_{\text{ref}} - V_{\text{a}}}{R_1} + \frac{V_{\text{Output}} - V_{\text{a}}}{R_3} = \frac{V_{\text{a}}}{R_2}$$

With a lot of algebra, this can be rewritten as:

$$V_{\text{a}} = \frac{R_1||R_2||R_3}{R_1} V_{\text{Ref}} + \frac{R_1||R_2||R_3}{R_3} V_{\text{Output}}$$

Since the output can have two different values (in this case V+ or V-), the voltage V_{a} will have two distinct values, depending on the output voltage. This implements the hysteresis.

A much simpler analysis can be done when $V_{\text{ref}} = V+$ and $V- = \text{GND} = 0\text{V}$. In that case, the circuit can be replaced with two circuits, one with the low output voltage and one with a high output voltage. This is shown in Figure 12.6. For a numerical example, take $R_1 = R_2 = 10\text{k}\Omega$, $R_3 = 20\text{k}\Omega$ and $V+ = 5\text{V}$. Simple calculation shows that $V_{th,low} = 2\text{V}$ and $V_{th,high} = 3\text{V}$ (check this!).

Comparator in a simulator
If the simulator you use does not have the comparator component, you can implement it with an op amp. Make sure to adjust the supply voltages to match the actual situation!

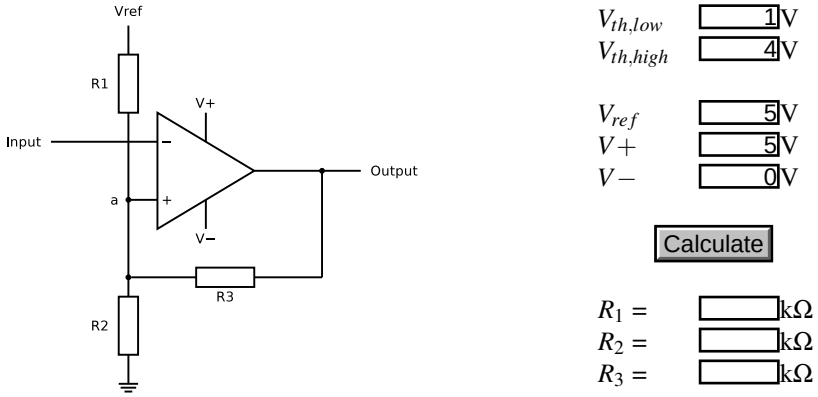


Figure 12.7: Inverting Schmitt trigger calculator for E12-series

Synthesizing the Schmitt trigger

Since there are three unknown variables (the resistors), but only two equations, it is impossible to calculate the resistor values analitically. However, it can easily be seen from the formulas that the value of V_a does not depend on the exact values of the resistors, but rather their ratio. Since only this ratio is important, one of the registers can be set to a fixed value, for example $R_2 = 1$. That leaves only two variables, so now it is possible to calculate the ratio of the resistors. These ratios have to be implemented with the available resistors. This requires testing a lot of combinations and is thus very time consuming. Fortunately, these calculations can be automated. If you're reading this manual as a digital version in a reader that supports PDF forms and JavaScript [3], you can use the calculator of Figure12.7. Otherwise you can use an online calculator, for example:

`http://www.random-science-tools.com/electronics/inverting-schmitt-trigger-calculator.htm`

Use one of the options to calculate the values of the resistors and test the resulting circuit in a simulator. Does it work correctly?

Resistor values available
In the lab most of the resistors in the E12 series (although with 5% accuracy) are available.

Output stage of the comparator

The output stage of the comparator will influence the way the Schmitt trigger works. What can you do to minimize this effect?

Problems due to nonideal power supply

Unfortunately, the protection circuitry of the Power Board affects the output voltage VBATT. In §11.5.1, you've measured the effect of the protection diode D11 the output

[3] At the moment of writing these include Adobe Acrobat Reader and Foxit Reader (Windows version). For Linux currently the only option is Adobe Acrobat Reader.

voltage.

The problem Due to the protection diode, the output voltage differs when the load is connected (when the robot is using the battery power).

- What was the open clamp voltage of the batteries?

  Answer:_____

- What was the voltage over the load resistor?

  Answer:_____

The correct functioning of the Schmitt trigger depends not only on the resistors, but also on the reference voltage level V_{ref} and the output voltage level V_{output}. Given the answers to the previous questions and the chosen values of the resistors, answer the following questions (use circuits similar to those in Figure 12.6 to answer the questions):

Assume the Mars Rover has just switched to running on the capacitors.

- What is the output voltage of the Schmitt trigger?

  Answer:_____

- What is the battery voltage?

  Answer:_____

- What is the low threshold voltage level $V_{th,low}$?

  Answer:_____

Now assume the capacitors have been discharged to the minimum voltage level and the Mars Rover has switched to the batteries.

- What is the output voltage of the Schmitt trigger?

  Answer:_____

- What is the battery voltage?

  Answer:_____

- What is the high threshold voltage level $V_{th,high}$ now?

  Answer:_____

Solving the problem As shown in the previous paragraph, the Schmitt trigger doesn't work properly due to variations in both V_{ref} and V_{output}. To solve this problem, we need to generate a stable voltage out of the battery voltage. The easiest option available is the 3.3 V voltage of the FPGA board.

Redesign the Schmitt trigger using the 3.3 V voltage provided by the FPGA as the voltage level for V_{ref} and V_{output}.

Power supply of the comparators
The output voltage of the comparator does not depend on the supply voltage, so you can just use the battery to power the comparators.

Increasing the output voltage swing The Schmitt trigger should work properly now, however, the output voltage is both a bit too low to switch the MOSFETs. An easy way to increase the voltage swing of the output is the use of another comparator: the output configuration of the comparator allows for a difference between the voltage swing at the input and the voltage swing at the output. Design this part of the circuit and combine it with the Schmitt trigger.

Build the Schmitt trigger

When the simulation results are satisfactory, build the circuit with actual hardware. Test the circuit. When the circuit works properly, proceed with §12.4.

Use the multimeter to determine the threshold levels
You can use the multimeter to measure the threshold levels using the `MINMAX` button in a similar fashion as in §11.4.

12.3.4 Hysteresis with a state machine

The Wikipedia page on hysteresis[4] states:

> Hysteresis is the dependence of a system not only on its current environment but also on its past environment. This dependence arises because the system can be in more than one internal state.

You're already familiar with a method to implement multiple internal states: the FSM. Since the FPGA is an integral part of the Mars Rover, why not utilize it for implementing the hysteresis as well? By combining an analog circuit with the digital components in the FPGA, you can implement the comparator with hysteresis.

The digital components can be implemented as a separate block in the top level entity. The FPGA then contains two main blocks:

- The line finder and line tracker controller
- The comparator with hysteresis

This is schematically shown in Figure 12.8. There are two digital components that make up the comparator with hysteresis:

- An input buffer
- A FSM

[4] `http://en.wikipedia.org/wiki/Hysteresis`

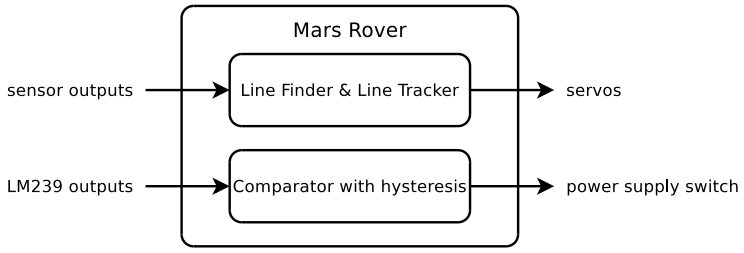


Figure 12.8: The line tracker controller and the comparator with hysteresis as two blocks in the top level entity

Digital components: Input buffer

The FSM uses the outputs of one or more comparators as inputs. These inputs must be synchronized with the clock of the FSM. Implement an input buffer that will perform this synchronization.

This input buffer has the same functionality as the input buffer used in the robot controller.

Digital components: FSM

The FSM is used to implement the required different internal states. Since there are two different output values, (at least) two states are required. The hysteresis is implemented by using different conditions for the state transitions to and from one state to another. Draw a state diagram for the FSM. What conditions do you need for the state transitions? How can you implement these conditions?

The LM239 contains four comparators in a single package.

Digital components: Testing

Combine the input buffer and the FSM into the comparator with hysteresis and simulate your design. If the design works correctly, you can implement it in the FPGA. If you connect the digital inputs to buttons or switches, you can test verify the output signal with a multimeter or an oscilloscope. Does the output level follow the specifications of your MOSFET circuit?

If the system works correctly and follows the specification of the MOSFET circuit, you can connect the MOSFET circuit to the FPGA. Can you correctly switch between the two power supplies?

If everything is OK, you can combine the robot controller and the comparator with hysteresis according to Figure 12.8.

Analog components: Comparator circuit

The FSM implements hysteresis based on digital inputs. These inputs are generated with the comparator. Design and implement the analog circuit. Think about the following:

- The FPGA has to read the input, so make sure to match the comparator output(s) with the pin assignments of the digital components
- Make sure you take into account your findings from §12.3.1
- Make sure you take into account your findings from §11.5.1
- Do not forget the ground connection

Watch voltage levels!
Note that the FPGA input voltage should be limited to a maximum of 3.3 V! Overvoltage can damage the FPGA board! Think about how to restrict the output voltage of the comparator within 0 V to 3.3 V.

Potentiometers
You are advised to implement the compare level with a potentiometer. This way, you can adjust the level later on.

Analog components: Testing

You can test your analog design for the inputs with a power supply an a multimeter. Does it work correctly? Make sure to check if the output of the comparators does not exceed the 3.3 V!

Use the multimeter to determine the threshold levels
You can use the multimeter to measure the threshold levels using the `MINMAX` button in a similar fashion as in §11.4.

Analog components: Increasing the Output Swing

The output signal(s) of the system are the outputs of the FPGA. These outputs are 0 V to 3.3 V. This is a bit too low to properly switch the MOSFETs. An easy way to increate the output voltage swing of the system is to use a comparator: the output configuration of the comparator allows for a difference between the voltage swing at the input and the voltage swing at the output. Design and integrate this part of the system.

Causality problems
Your solution might end up having a causality problem: the FPGA decides which power supply to use, but in order to do that, the FPGA itself should already be powered on. You can solve this problem by adding a button to force the switch signal to high or low. Make sure to add a resistor in series to prevent a short circuit!

12.4 Implement the complete switch

Now that you've implemented the switches and the control signal, you can combine the two parts to implement the controllable power supply switch. Connect the completed circuit to the robot and the super capacitors and test the funcionality.

Chapter 13

Converting the Solar Panel Output Voltage

Objectives

During this session, you will:

- Design different power converters:
  - resistor-based voltage divider
  - linear regulator
  - buck converter
- Examine the behaviour of these designs

13.1 Introduction

At this moment you have all the required components for solar/battery power supply:

- solar panel
- capacitor bank
- battery pack
- switch (with a comparator with hysteresis)

There is, however, one problem: the output voltage of the solar panel is around 20 V, while the capacitor bank and the Mars Rover itself can only handle 6 V. The capacitor bank is protected with a high-current 6 V zener-diode that will make sure the voltage drop over the capacitors will not exceed the 6 V, but this is of course more of a safety measure than a real solution. In this chapter you will examine different options to lower the output voltage of the solar panel.

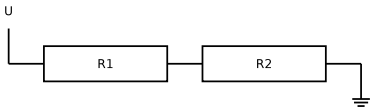


Figure 13.1: Simple voltage division using two resistors

13.2 Voltage division using resistors

One of the first options that could come into mind, is using voltage division. As you know, the voltage drop over a single resistor in a series connection of two resistors is proportional to the relative magnitude of the resistance of that resistor. This is depicted in Figure 13.1. In this case, the voltage drop U_{R2} over resistor $R2$ is equal to:

$$U_{R2} = U \frac{R2}{R1 + R2}$$

13.2.1 Determine the ratio of $R1$ and $R2$

Since the required voltage U_{R2} and the supply voltage U are known, the ratio of $R1$ and $R2$ is fixed. Determine this ratio.

Answer:

13.2.2 Determine the values of $R1$ and $R2$

The exact values of $R1$ and $R2$ depend on the available resistors and the current that should flow through the network. In many cases you want to minimize the current flowing through the resistors, so the actual values are in the order of 100 kΩ. In this case, the situation is somewhat different: we want to use the voltage over $R2$ as the input voltage of the Mars Rover. What does this mean for the order of magnitude of the resistors? Verify your findings with the circuit simulator.

Answer:

13.2.3 Evaluate the usability of the system

The efficiency η of the system is determined as:

$$\eta = \frac{\text{output power}}{\text{input power}}$$

Determine the efficiency for this system.

Answer:

Depending on whether or not the motors need to run, the load on the voltage divider varies. How does this influence the output voltage?

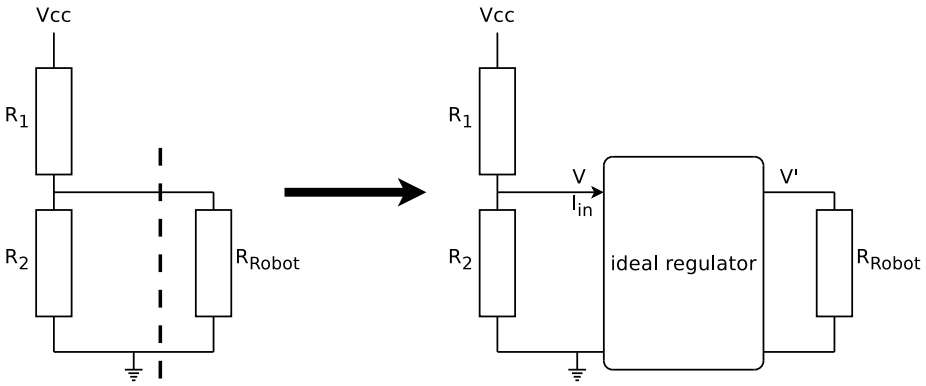


Figure 13.2: From voltage division with resistors to linear regulator

Answer:

Is this voltage divider a useable solution for the Mars Rover? Discuss the results with a supervisor.

Answer:

13.3 Linear voltage regulator

Although the voltage division option is simple and cheap, it is far from efficient. Even worse: the system is also sensitive to changes in the value of the load resistor. Another option to limit the output voltage of the solar panel is using a *linear voltage regulator*. Such a regulator limits its output voltage to the required level for a wide range of loads. The excess energy is dissipated as heat. Linear regulators come in handy prefabricated packages for a wide range of fixed or configurable output voltages. The most well-known regulators are the 78XX range, in which the XX represents the output voltage. Thus the 7805 is a 5 V linear regulator. The output voltage of these regulators is (almost) independent of the load and the input voltage. In the lab, we're not going to use such a prefabricated voltage regulator, instead we will build are own version out of discrete transistors and some additional passive components.

13.3.1 Basic idea

In order to get an idea of the workings of a linear regulator, we start with the circuit of Figure 13.2. In this circuit, the two resistors R_1 and R_2 are used as voltage dividers. When the equivalent resistance of the robot is constant, one can choose the values of these two resistors in such a way that the voltage drop over the robot is equal to the required 6 V. The equivalent resistance of the robot is very low, this means that a change in this resistance will greatly affect the voltage division (which depends on the equivalent resistance of the parallel combination of R_2 and R_{Robot}).

In order to make this circuit less sensitive to changes in the equivalent resistance of the robot, the circuit should be split in two (across the dashed line). If it is possible to

connect the two parts using a component with the following two properties:

- $I_{in} = 0$
- $V = V'$

the circuit would be insensitive to changes in R_{Robot}. This component is the ideal regulator. You will examine two possible implementations of the regulator.

13.3.2 Op amp

A unity gain buffer is a component for which the two stated properties hold. Such a buffer can be build with an op amp. The input voltage of the buffer can (again) be obtained from a voltage divider, but the resistor values no longer need to be very low (why?).

Use a simulator to design and simulate a regulator based on an op amp. Does the circuit behave well for different values of the load resistor?

Answer:

Although this circuit should work as required, it is a simplification of the real world. If you've correctly implemented the circuit, the current through the load resistor is much larger than the current through the resistors in the voltage divider. From where does this current originate?

Answer:

What is the current the op amp has to supply to the Mars Rover? What is the maximum current a simple general purpose op amp[1] can provide?

Answer:

13.3.3 Bipolar transistors

Implementing the linear regulator with a buffer build with an op amp makes the circuit insensitive to changes in the load resistor. However, the op amp is not able to provide enough power to the Mars Rover. We need an alternative solution that:

- makes the circuit insensitive to changes in the load transistor
- can provide enough power to the Mars Rover

In Chapter 12 MOSFETs were used to implement the switch. A MOSFET is very well suited for use as a on/off switch. For the voltage regulator, we don't need an on/off switch, but we want a controllable source. This can be implemented with a bipolar transistor.

A bipolar transistor works differently than a MOSFET. The device still has three terminals, but they are now called *base*, *collector* and *emitter*. There are two different types of bipolar transistors, the NPN and the PNP. Both versions are depicted in Figure 13.3.

[1] Examples include U741 and LF356

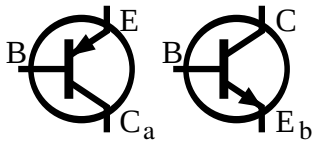


Figure 13.3: Different types of bipolar transistors: a. PNP, b. NPN

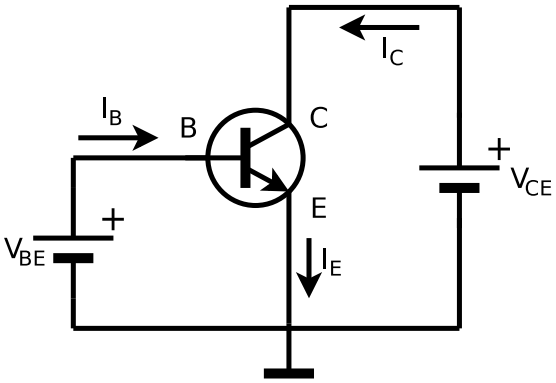


Figure 13.4: Test circuit of a NPN transistor

A properly biased bipolar transistor behaves as a voltage controlled current source. A simple test circuit of a NPN transistor is shown in Figure 13.4.

The voltage V_{be} will cause the base current I_b and the voltage V_{ce} will cause an output current I_c. The collector current I_c and the base current I_b are related according to the following equation:

$$\beta = \frac{I_c}{I_b}$$

The β is the DC gain of the transistor. This characteristic is also known as h_{FE}. The value of β of a specific transistor can be found in the datasheets and is typically in the range 20 – 200.

Just like MOSFETs, bipolar transistors can be used in different configurations, one of which is a voltage follower. In case of the bipolar transistor, this is called an *emitter follower*. This circuit is shown in Figure 13.5. In this circuit the resistors R_1 and R_2 determine the voltage over the load resistor R_{load}. Just as in the case of the op amp regulator, the values of the resistors R_1 and R_2 can be much larger than R_{load}.

You will simulate the emitter follower and study its behaviour by varying different components of the circuit. The supply voltage V_{cc} should of course have the value of the open-circuit voltage of the solar panel.
Before you start, answer the following questions:

- What is the value of β of the NPN transistor used? Adjust the value in the simulator accordingly.

  Answer:_____

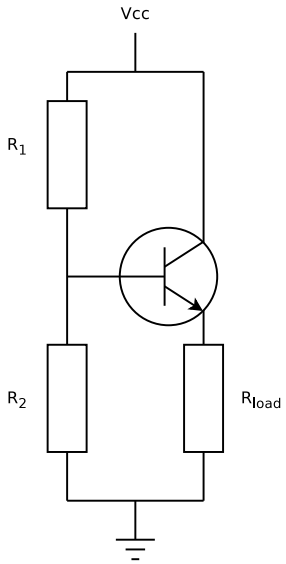


Figure 13.5: Emitter follower

- What is the ratio of the resistors R_1 and R_2?

  Answer:

Now build the emitter follower in the simulator (make sure to adjust β!). Choose values for R_1 and R_2 just below 1 kΩ (you don't have to build yet, so you can choose arbitrary values) and take a 100 Ω resistor as R_{load}. You will see that the voltage V_B will be slightly different from the voltage calculated with the voltage divider of R_1 and R_2. The transistor will influence this division. The resulting relationship is very complex, but in the simulation you can easily adjust the resistors to compensate. Answer the following questions:

- The voltage drop over the load resistor differs from the base voltage V_B. What is the difference? Can you explain this difference? Adjust R_1 and R_2 to compensate.

  Answer:

- Vary the value load resistor (± 20%). Is the circuit sensitive to changes in this load resistor?

  Answer:

- Use the same ratio for the resistors R_1 and R_2, but now choose a value in the 100 kΩ range. What happens? Can you explain this effect? Hint: test what happens when you change the value of the load resistor to 10 kΩ.

  Answer:

Now use the resistor model of the Mars Rover for the load resistor R_{load} and adjust the resistors R_1 and R_2 in order to have the correct output voltage.

- Vary the value load resistor ($\pm$ 20%). Is the circuit sensitive to changes in this load resistor?

  Answer:

- What is the total power dissipated in the resistors R_1 and R_2?

  Answer:

- What is the efficiency η of this regulator circuit?

  Answer:

You should have seen that this circuit is not very sensitive to variations in the load resistor, but it is possible to improve on this situation even further. This can be done by making the voltage V_B independent of the voltage division of R_1 and R_2. This can be done with a *zener diode*. Such a diode has a device specific zener voltage, a voltage level for which reverse breakdown occurs. In contrast to normal diodes, this is intended behaviour. The zener voltage is defined for a specific reverse current, this can be found in the datasheet.

- Search in the datasheet for the test current and the maximum allowed current of the zener diode.

  Answer:

- Rebuild the emitter follower in the simulator, but now use a resistor and a zener diode to specify the voltage V_{be}. Does the circuit work?
- What is the total power dissipated in the resistor and the zener diode?

  Answer:

- What can you say about the efficiency η of this regulator circuit? What are your conclusions when you compare this regulator to the one with the resistor voltage division?

  Answer:

If the circuit works correctly in simulation, build it and test it with a lab power supply and the solar panel. If this circuit also works correctly, you can combine it with the other components of the power supply and test it.

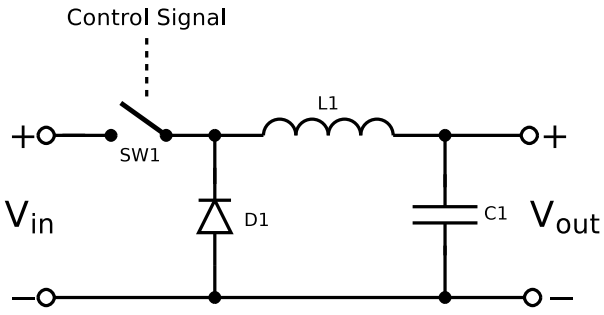


Figure 13.6: General topology of the buck converter

13.4 Buck Converter

A third way to step down the input voltage is using a so-called buck converter. Such a converter is a type of switching power supply and can be more efficient than a linear regulator. The analysis of such a buck converter is already discussed in the course "Electronic Power Conversion" (code ET3051TU). The general topology of the buck converter is shown in Figure 13.6. By periodically opening and closing the switch, a high input voltage can be stepped down. Both the frequency and the duty cycle of the control signal can be adjusted, in many situations, however, the frequency is fixed and only the duty cycle is varied. The efficiency of the buck converter stems from the fact that it doesn't load the power source continuously. The output voltage of a buck converter is always lower than the input voltage and by not loading the power source, for example a battery, during the moments the switch is open, the circuit can run on batteries for (much) longer. In the Mars Rover the buck converter is not used with a battery but with a solar panel; this affects the way the buck converter should work.

Before diving into the details of dimensioning the buck converter, we start with a re-evaluation of the voltage regulators used thus far and discuss the effect of the conclusions of this analysis on the buck converter.

13.4.1 Analysis of the different voltage converters

Analyzing the resistor division and the linear regulator

As you should have seen, using a resistor division or a linear regulator doesn't work well for this system. Can you explain why? There are a number of reasons:

- What is the effect of the electrical behaviour of the solar panel (I/V curve) on the output voltage?

  Answer:_____

  _____

- What is the effect of the protection circuit in the super capacitor bank on the output voltage?

  Answer:_____

- What is the effect of the complete system (the solar power supply (panel and super capacitor bank), the battery supply, the power switch, and the robot together) on the output voltage?

  Answer:

First look at the buck converter

As stated before, a buck converter can be very efficient since it doesn't load the power source continuously.

- Will the buck converter in this configuration work well for this system, when you consider the maximum power delivered by the solar panel and the power requirements of the robot? Explain!

  Answer:

Intermediate conclusions and recommendations

Based on the above analysis, we can draw the following conclusions:

- The electrical nature of the solar panel does not lend itself well to the use of a voltage converter;
- The complete system will make it possible to use the solar panel without voltage converter;
- The general structure of buck converter will not work as intended for this system.

Although these conclusions do not seem very promising, it is possible to make a properly working buck converter for this system.

13.4.2 Using the buck converter efficiently with the solar panel

In order to have the converter work properly in the complete Mars Rover setup, two things need to change:

- The solar panel needs to work in or as close as possible to its mpp (maximum power point);
- The converter needs to act as a DC transformer: the voltage should be stepped down, while the current should be stepped up (and power is conserved).

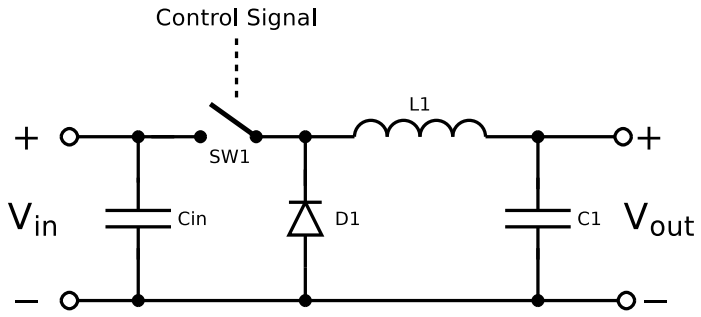


Figure 13.7: Buck converter with input capacitor

Using the solar panel in the mpp

The output voltage of the buck converter is controlled by periodically opening (during time t_{off}) and closing (during time t_{on}) the switch. This is specified as the duty ratio D:

$$D = \frac{t_{on}}{T_s}$$

with

$$t_{on} + t_{off} = T_s = \frac{1}{f_s}$$

In continuous-conduction mode[2], the input voltage and the output voltage of the buck converter are related as:

$$\frac{V_{out}}{V_{in}} = D$$

By choosing D such that V_{out} has the required value for a value $V_{in} = V_{mpp}$, the solar panel will operate at its mpp. Calculate this duty ratio D (see your answers at page 102):

Answer:

Stepping up the current

The current at the mpp (I_{mpp}) is too low to power the robot. Since in the buck converter the switch is periodically opened and closed, the average current supplied to the load will be even lower (why?). By adding an input capacitor, the buck converter can be used to simultaneously step down the voltage and step up the current. The circuit with this added input capacitor is shown in Figure 13.7.

Can you explain why adding this capacitor solves the problems? Consider both the situation in which the switch is open and in which the switch is closed. Use the Falstad simulator to build and simulate the circuit of Figure 13.8. Start the simulation with switch `sw1` closed and `sw2` open until the capacitor is charged. Now close switch `sw2` and verify your answer by simulating with switch `sw1` both opened and closed.

[2]This condition may not always hold for this specific application!

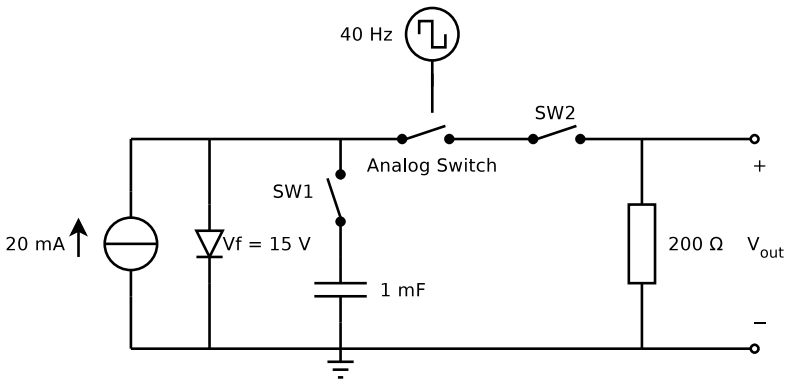


Figure 13.8: Test circuit for testing the effect of the input capacitor

Answer:

13.4.3 Components the buck converter

Now that you know how to use the buck converter efficiently with the solar panel, there are still a lot of parameters that can be changed:

- Switching frequency f_s,
- Values of the components,
- The type of switching element,
- The control circuitry of the switching element

Switching element and control signal

One of the main components of the buck converter is the switching element. There are multiple options to implement this switch, but since in the general structure, the input voltage (high side) must be switched, the most obvious switching element is the PMOS MOSFET (why?).

The control signal of the switch can be generated with the FPGA. The output of the FPGA cannot directly be used to switch the PMOS (why?). A possible solution is to use the FPGA to switch an NMOS that in turn switches the PMOS MOSFET. Such a circuit is shown in Figure 13.9:

- When the control signal is high (5 V), the NMOS will be conducting and the gate of the PMOS will be pulled to 0 V and thus the PMOS will conduct;
- When the control signal is low (0 V), the NMOS will shut off and the gate of the PMOS will be pulled up to the voltage V_{Solar} via resistor R.

Resistor R is dimensioned to limit the maximum current through the NMOS transistor in conducting state and in the order of 100 Ω for a small NMOS.

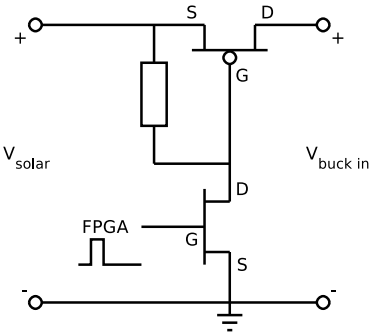


Figure 13.9: Switching a PMOS with the FPGA via an NMOS

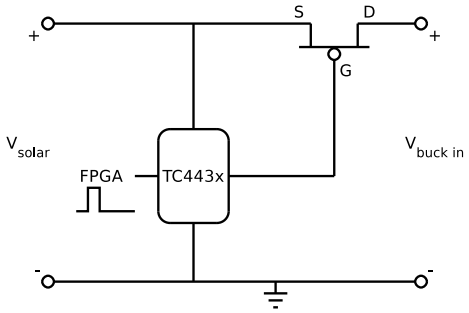


Figure 13.10: Switching a PMOS with the FPGA via a MOSFET driver

The problem with the circuit of Figure 13.9 is that it switches quite slowly. The switching speed of a MOSFET is limited due to the *gate capacitance*. This gate capacitance is present simply due to the way a MOSFET is constructed. When the state of the MOSFET is changed, the gate capacitance has to be charged or discharged. In the circuit of Figure 13.9 the gate capacitance can be quicly discharged when the control signal from the FPGA is high (5 V). The conducting NMOS connects the gate of the PMOS to ground via a low-ohmic path and the gate capacitance is quickly discharged. However, when the control signal from the FPGA is low (0 V) the NMOS will close and the gate capacitance will be charged via the much larger resistor R. During the time the capacitor is charging, the MOSFET acts as a resistor with a resistance $R >> R_{ds,on}$ and thus dissipates a lot of energy. In the buck converter the MOSFET needs to switch fast (order of kHz), so we need to improve on the circuit of Figure 13.9.

In order to have the MOSFET switch much faster, a special circuit can be used: the *MOSFET driver*. Such a MOSFET driver is able to very quickly charge the gate capacitance and accordingly switch the MOSFET states very fast. In the lab the Microchip TC4431 and TC4432 are available (datasheets can be found on Brightspace). This is schematically shown in Figure 13.10.

Switching frequency

The output voltage of the buck converter depends on the duty cycle, but of course the switching frequency is important as well. Choosing a high switching frequency will generally reduce ripple effects. Another reason for choosing the frequency not too low is that this might produce audible noise. Choosing the switching frequency $f_s \geq 20\,\text{kHz}$

will eliminate this noise.

The MOSFET driver will allow you to use frequencies up to several 100 kHz without any problems, although very high frequencies will cause excessive dissipation in the MOSFET driver.

The choice of the switching frequency is important for dimensioning the other components of the buck converter. Build a test circuit with the MOSFET driver and a PMOS MOSFET. Use a lab power supply with a voltage level equal to V_{mpp}. Use a 100 Ω power resistor as load.

Use a high power load resistor
Do not use a regular, low power, 100 Ω resistor, these can only handle 0.25 W!

MOSFET driver
Do not directly solder the MOSFET driver onto the PCB, but use a DIP socket instead. This way the driver will not be damaged by the heat of soldering and can easily be re-used or replaced.

Use a function generator to generate the control signal with the required duty cycle.

Control signal wave form
The control signal should not be negative, this will damage the MOSFET driver! Before connecting the output of the function generator to the MOSFET driver, verify the wave form on the oscilloscope.

If the control signal is correct, connect it to the MOSFET driver and test the circuit with different frequencies. Based on your measurements, decide on the switching frequency f_s of the buck converter.

Dimensioning input capacitor C_{in}

The capacitance of input capacitor C_{in} determines the voltage ripple in V_{in} when switch SW_1 is closed. For a capacitor, the following equation holds:

$$i = C\frac{dv}{dt}$$

We can easily approximate the value of the capacitance under the following conditions:

- Assume the converter runs in continuous conduction mode;
- Assume the voltage changes linearly.

What should be the capacitance of C_{in} when the maximum allowable voltage ripple is 1%?

Answer:

Choose a suitable capacitor based on this calculated value.

Maximum capactitor voltage
An electrolytic capacitor has a maximum voltage rating. This is usually shown on the casing. Make sure the capacitor can handle the maximum voltage present in your circuit!

Output capacitor C_1

The capacitance of output capacitor C_1 determines the ripple in the output voltage. In this case, the output capacitor is taken as the super capacitor bank. The (very) large capacitance means that for virtually all purposes, the output voltage can be assumed constant.

Dimensioning inductor L_1

The inductance of inductor L_1 determines the ripple in the inductor current. For an inductor, the following equation holds:

$$v = L\frac{di}{dt}$$

Assume the output voltage V_{out} over the super capacitor bank C_1 to be constant[3]. With a properly dimensioned input capactor, the input voltage V_{in} can also be assumed to be constant. Again assuming continuous conduction mode, the ripple current should not exceed the maximum current that the solar panel and input capacitor C_{in} can provide. Based on these assumptions, determine the inductance of inductor L_1:

Answer:

Build and test the buck converter

With all the components dimensioned, build the buck converter and test the circuit. Use a function generator to generate the control signal. Pre-charge the super capacitor bank to half a volt below the optimal output voltage of the buck converter. Does the converter work correctly?

[3] Of course this is not true, since the voltage will change while charging or discharging the capacitor bank. However, for a small time step Δt we can assume $V_{C,out}$ to be constant

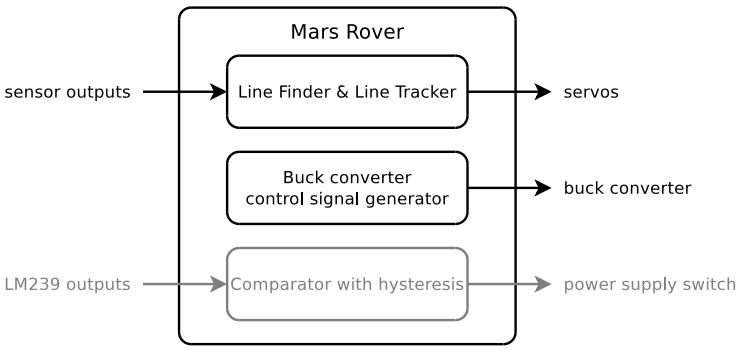


Figure 13.11: All current blocks of the Mars Rover combined into a new top level entity

Intrinsic diode
The MOSFET contains an intrinsic diode. This diode is conducting when drain voltage is higher than the source voltage. Make sure the solar panel is providing a stable output voltage (lamp is switched on) before connecting the charged capacitor bank, otherwise a current will flow *into* the solar panel, heating it up and possibly cause damage!

Generating the control signal

The control signal of the buck converter can be generated with the FPGA. Design a block diagram and a state diagram of your FSM. Simulate the design and verify that the period and the duty cycle are according to your specifications. Your current Mars Rover VHDL code and the block that generates the control signal of the buck converter operate independently. Create a new top level entity and combine all the blocks in this new entity. Depending on your choice for the control signal of the power switch, you end up with either two or three blocks in the top level entity:

- The line finder and line tracker
- The control signal generator of the buck converter
- Optionally the comparator with hysteresis[4]

This is schematically shown in Figure 13.11.

Assign a pin to the control signal and create a programming file. Upload the program and connect the control input of the buck converter to the FPGA. Does it still work correctly?

Jumper pin
Solder a single jumper pin to the control input wire of the buck converter to easily connect it to the FPGA.

[4] You should not have this component if you've implemented the control signal of the power switch with an analog schmitt trigger

13.4.4 Implement the complete buck converter

Now that you've implemented and tested all components of the buck converter, you can integrate the buck converter in the Mars Rover and test the complete system. Does it work according to your expectations?

13.4.5 Final thoughts on the buck converter

The buck converter should provide a serious improvement in the charging time of the capacitor bank. However, the buck converter design is not optimal due to some of the assumptions and design simplifications made:

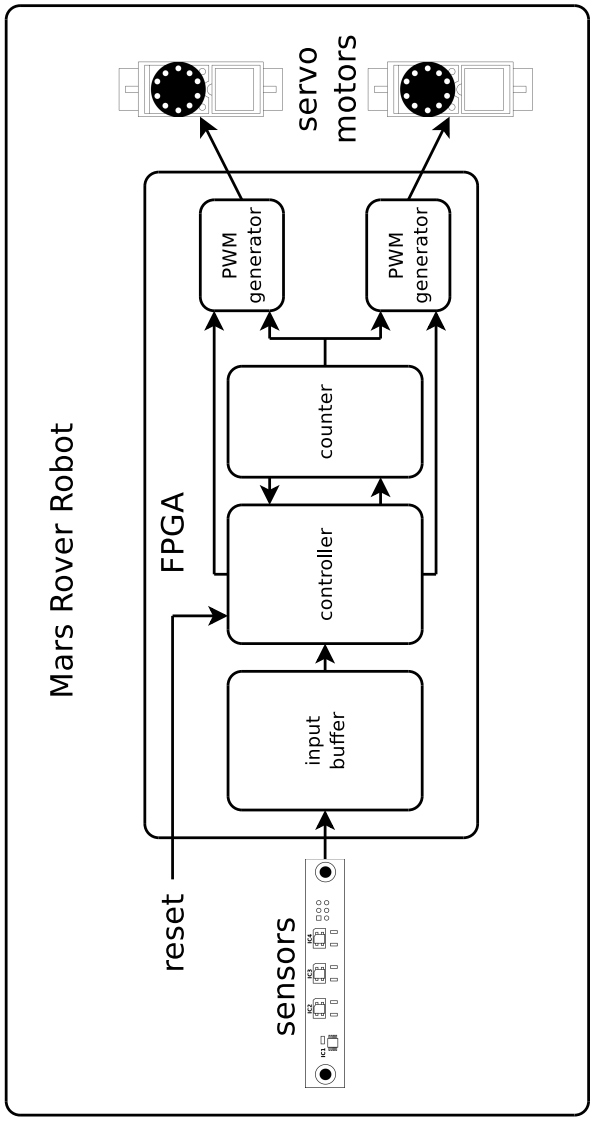
- With a fixed duty cycle the converter only works optimally (eg. in the mpp) for a single output voltage. Since the output voltage changes during charging and discharging, the duty cycle should be changed accordingly. This requires monitoring of the output voltage, which is difficult to implement in the current setup.
- The converter was assumed to work in continous conduction but it is very likely it actually runs in discontinuous mode. This will complicate the design considerably.

13.5 The end

This was the last assignment of the Mars Rover project. We hope you've enjoyed doing the project and that you've learned a lot. The project and the courses in the minor have shown you some of the possibilities of electrical engineering. If doing the minor has convinced you to start a MSc program in electrical or computer engineering, please consult one of the study counselors. Good luck with last exams of the minor!

Appendix A

Overview of the robot



Appendix B

Create a Vivado Project

B.1 Introduction

This appendix explains the steps required to create a Vivado project. In this example the tutorial project `nightrider` is created, modify file names if your creating another project. For the tutorial project, download the zip-file `nightrider.zip` from Brightspace and extract the file.

B.2 Creating a New Project

Open Vivado 2017.2 and select File→New Project.... This opens the New Project wizard intro page (see Figure B.1). Press Next to continue.

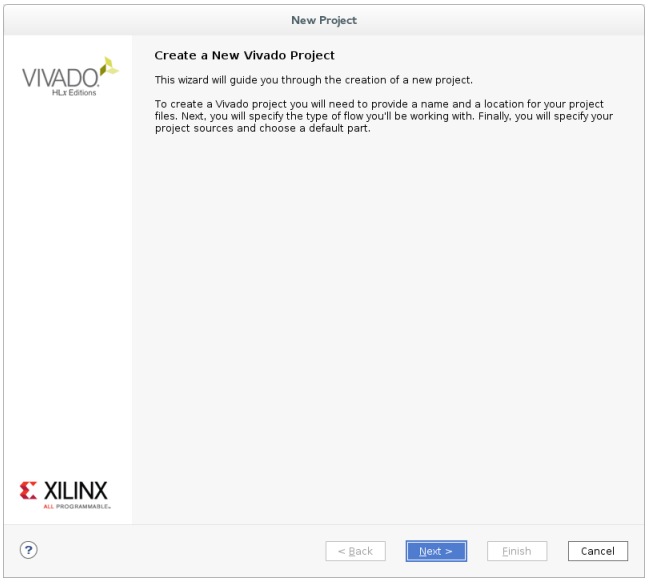


Figure B.1: Screenshot of the intro page of the New Project wizard

The Project Name page (see Figure B.2) allows you to specify the name and location of the project. Enter a project name and specify a valid location and then press Next to continue.

For the nightrider tutorial, please use a location under `C:\Users\<netid>\` (Windows) or `/data` on (Linux) since the simulation will generate very large files that will not fit in your home directory!

Do not use non-SSD flash storage for your project
Do not specify a location on a thumb drive, SD-card, or similar flash memory based storage! Vivado will create a lot of small files during compilation and synthesis. This will both take a long time to complete and quickly wear out the flash memory. Instead, specify a location in your home-directoy or under `C:\Users\<netid>\` (Windows) or `/data` on (Linux). Note that the data disk locations are local storage, you cannot access the files remotely!

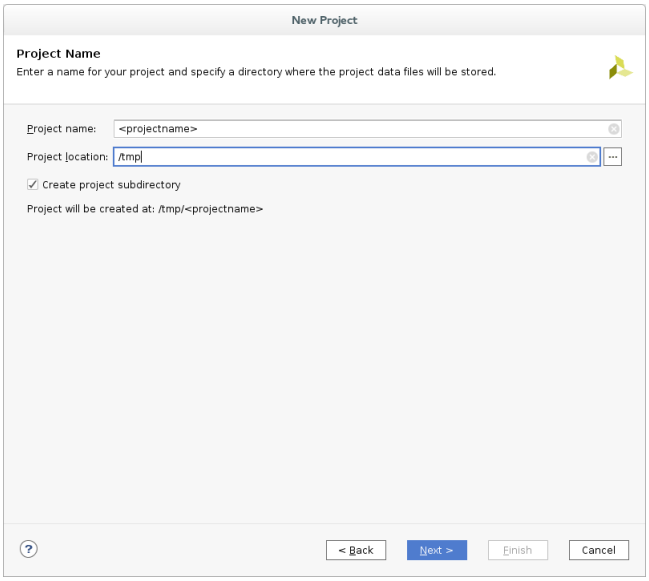


Figure B.2: Screenshot of the Project Name page of the wizard

The Project Type page (see Figure B.3) is used to specify the type of project. For normal projects you need to the RTL Project option. Select that option and press Next to continue.

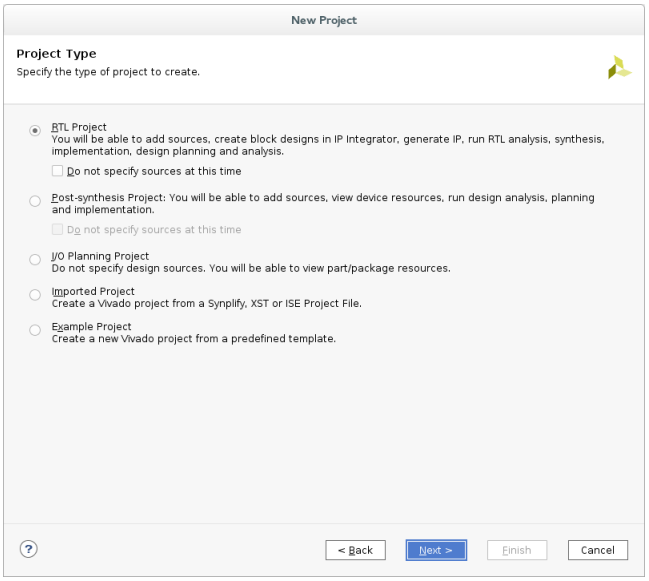


Figure B.3: Screenshot of the Project Type page of the wizard

The Add Sources page (see Figure B.4) is used to add sources or create new source files. You can specify if added sources should be used during both simulation and synthesis or during simulation only. For the tutorial project add the files `nightrider.vhdl` (the actual source file) and `nightrider_tb.vhdl` (the test bench file). The latter file is used only in simulation, so adjust the column labeled HDL Source For of this file to Simulation only.

Link to sources or copy sources
By default, files added in this step are *linked* into the project. This means that the project contains a reference of the original files and editing such a file will modify the original. You can also *copy* a file locally into the project. This means that editing the file will only modify the copy and leave the original intact. You can copy sources by ticking the box Copy sources into project.
Carefully choose the option you want, otherwise changes to the source files might have unintended side effects for other projects!

In addition to adding files, this dialog is also used to specify the target HDL language and the HDL language used in simulation. Set Target language to VHDL and Simulator Language to either VHDL or Mixed. Now press Next to continue.

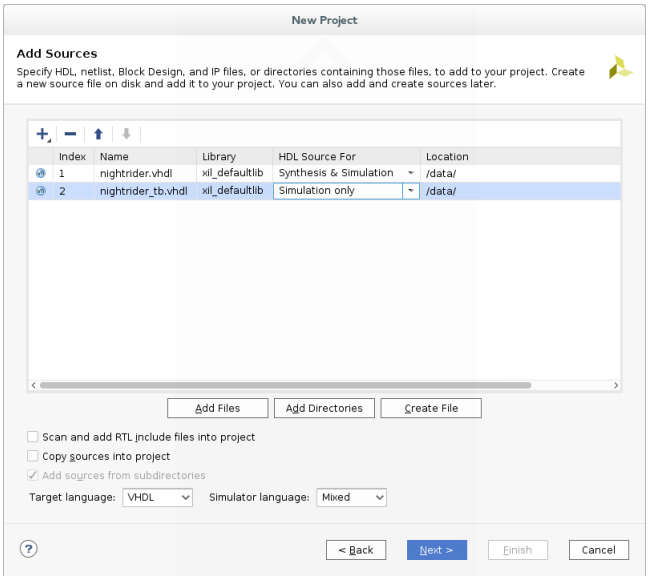


Figure B.4: Screenshot of the Add Sources page of the wizard

The Add Constraints page (see Figure B.5) can be used to add constraint file(s). These files describe pin settings, clock settings, and flash settings. For the `nightrider` project add the file `nightrider.xdc`. Press Next to continue.

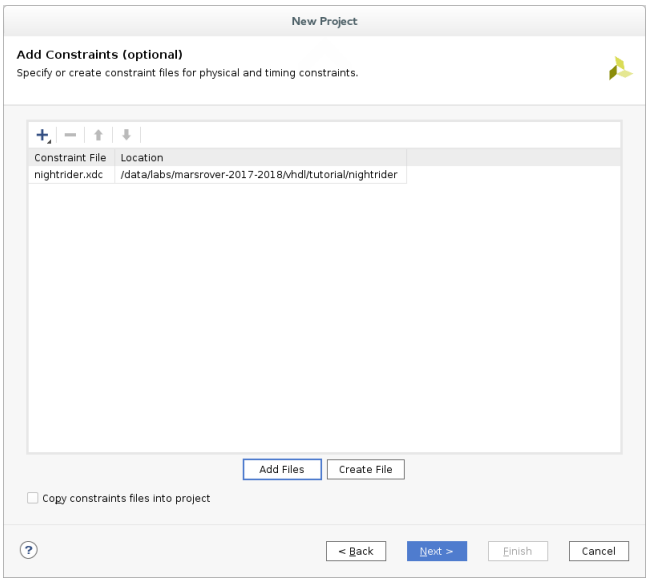


Figure B.5: Screenshot of the Add Constraints page of the wizard

The Default Part page (see Figure B.6) is used to specify the board or the FPGA. For the tutorial, take the following steps:

- Select the Boards tab;

- For Vendor, select digilent.com;
- For Display Name, select Basys3;
- For Board Rev, select Latest;
- Now you should only have a single choice left, the Basys3 board, revision C.0, FPGA xc7a35tcpg236-1;

Press Next to continue to the summary page of the wizard.

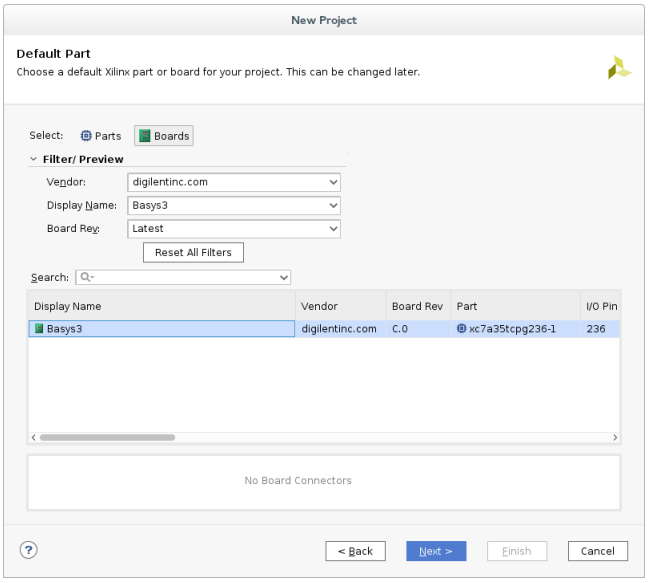


Figure B.6: Screenshot of the Default Part page of the wizard

The New Project Summary page (see Figure B.7) briefly describes the steps that will be taken on project creation. Review the information and press Finish to complete the wizard and create the new project.

Changing settings later on
Almost all settings can be changed later on using the various entries in the settings dialog (Tools→Settings...). However, files are copied or linked on project creation, so make sure you select the correct option when adding files!

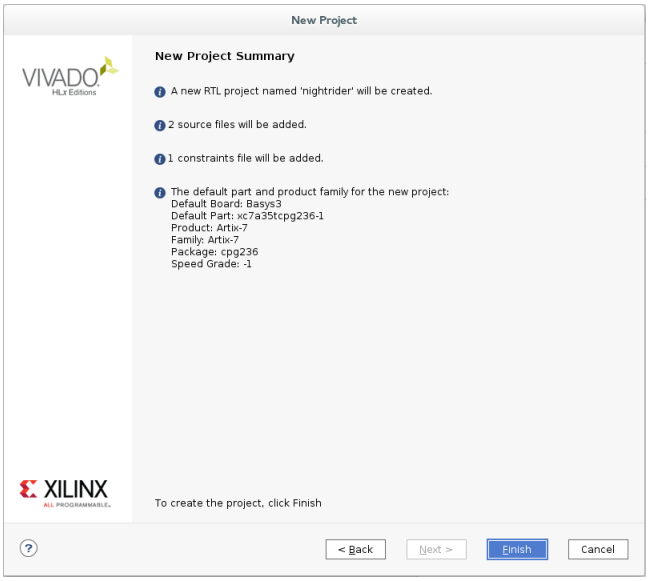
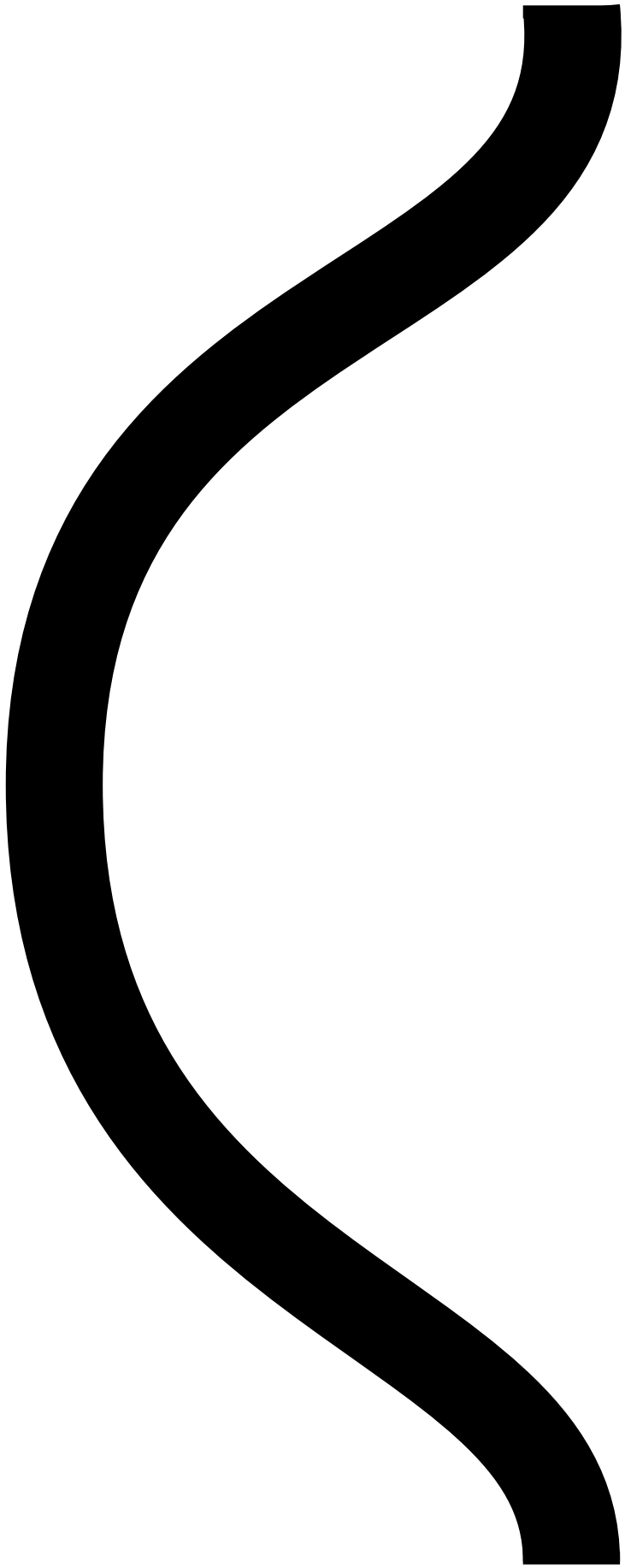


Figure B.7: Screenshot of the Summary page of the wizard

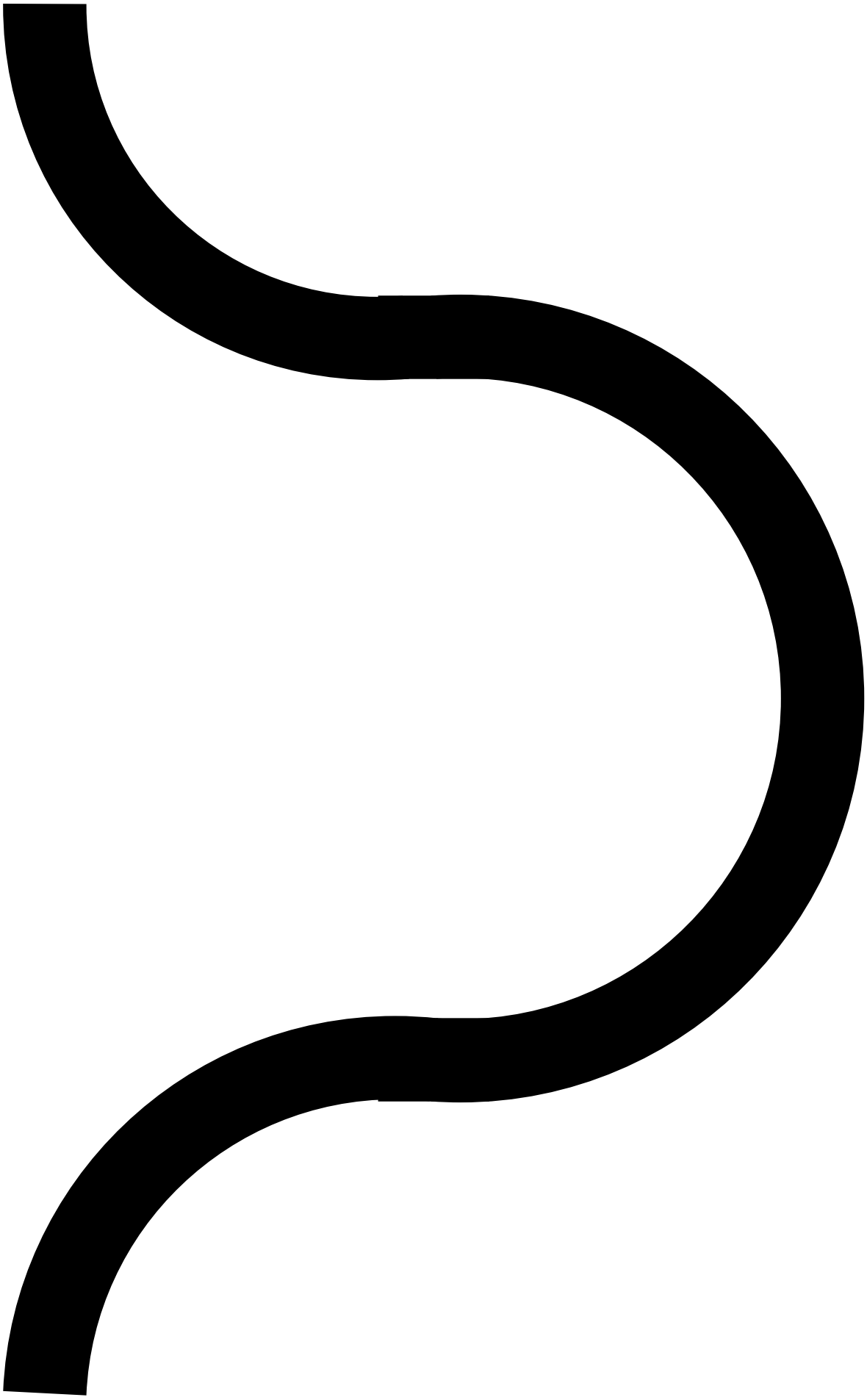
Appendix C

Test lines

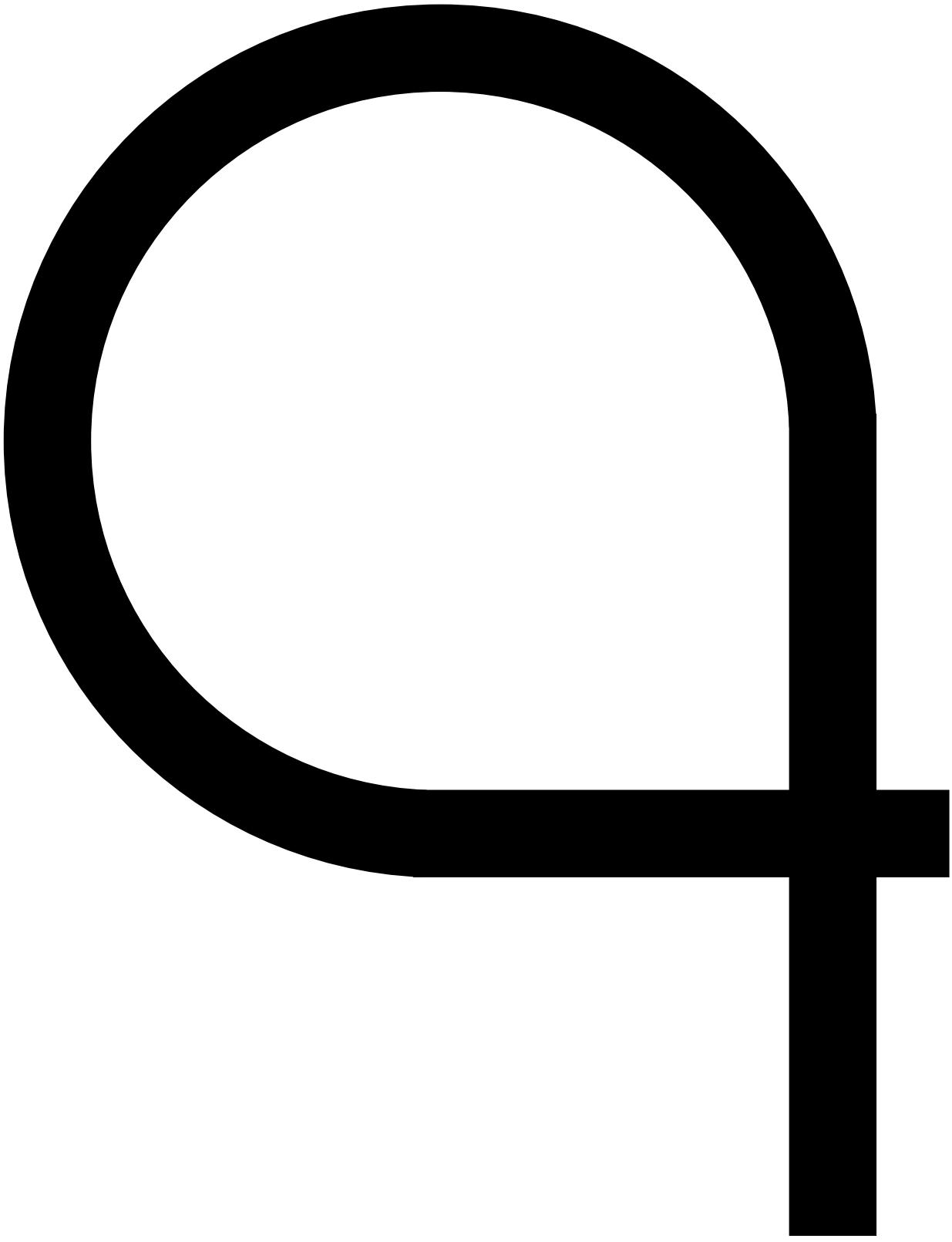
Line 1: straight line



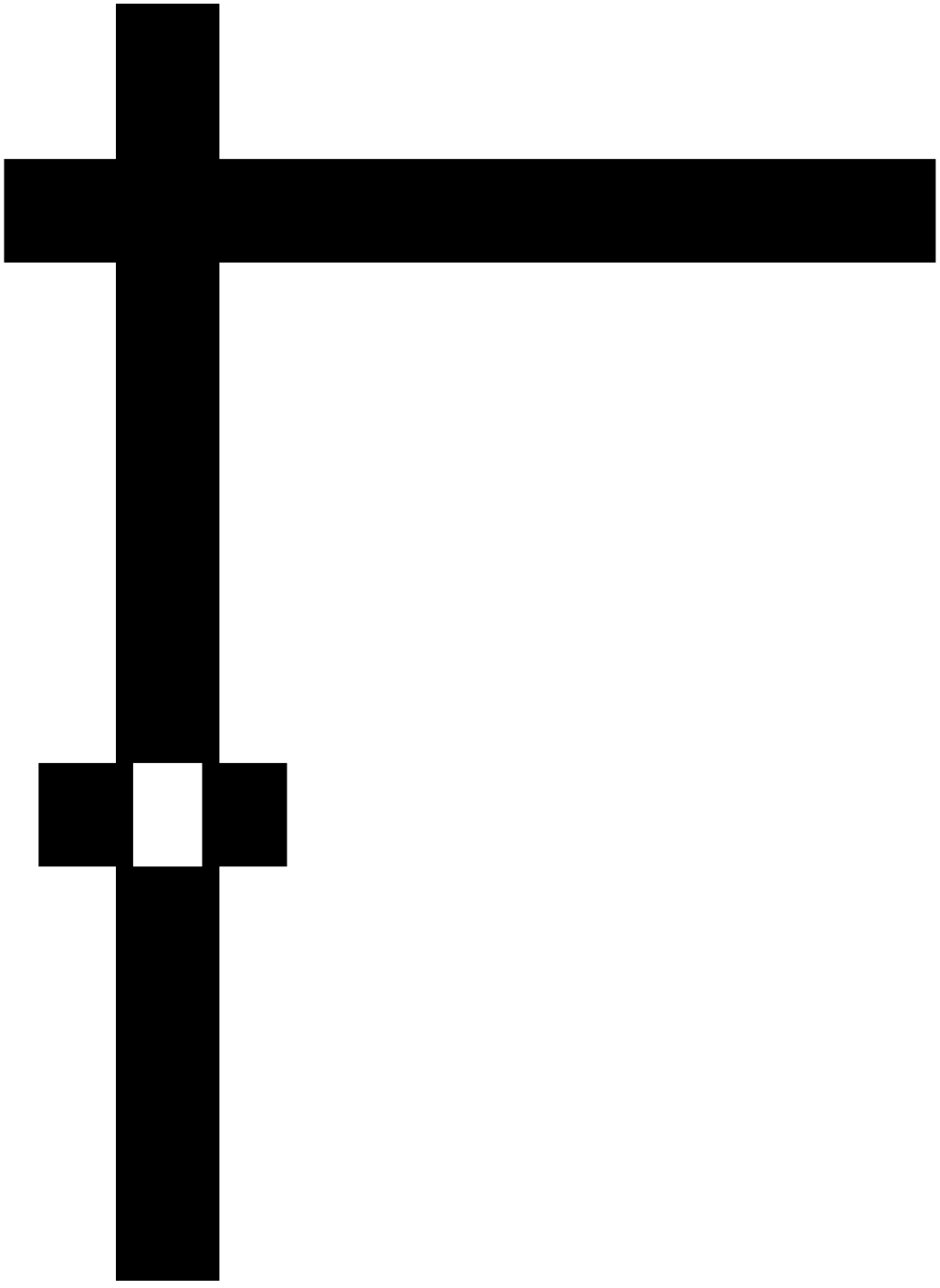
Line 2: bend



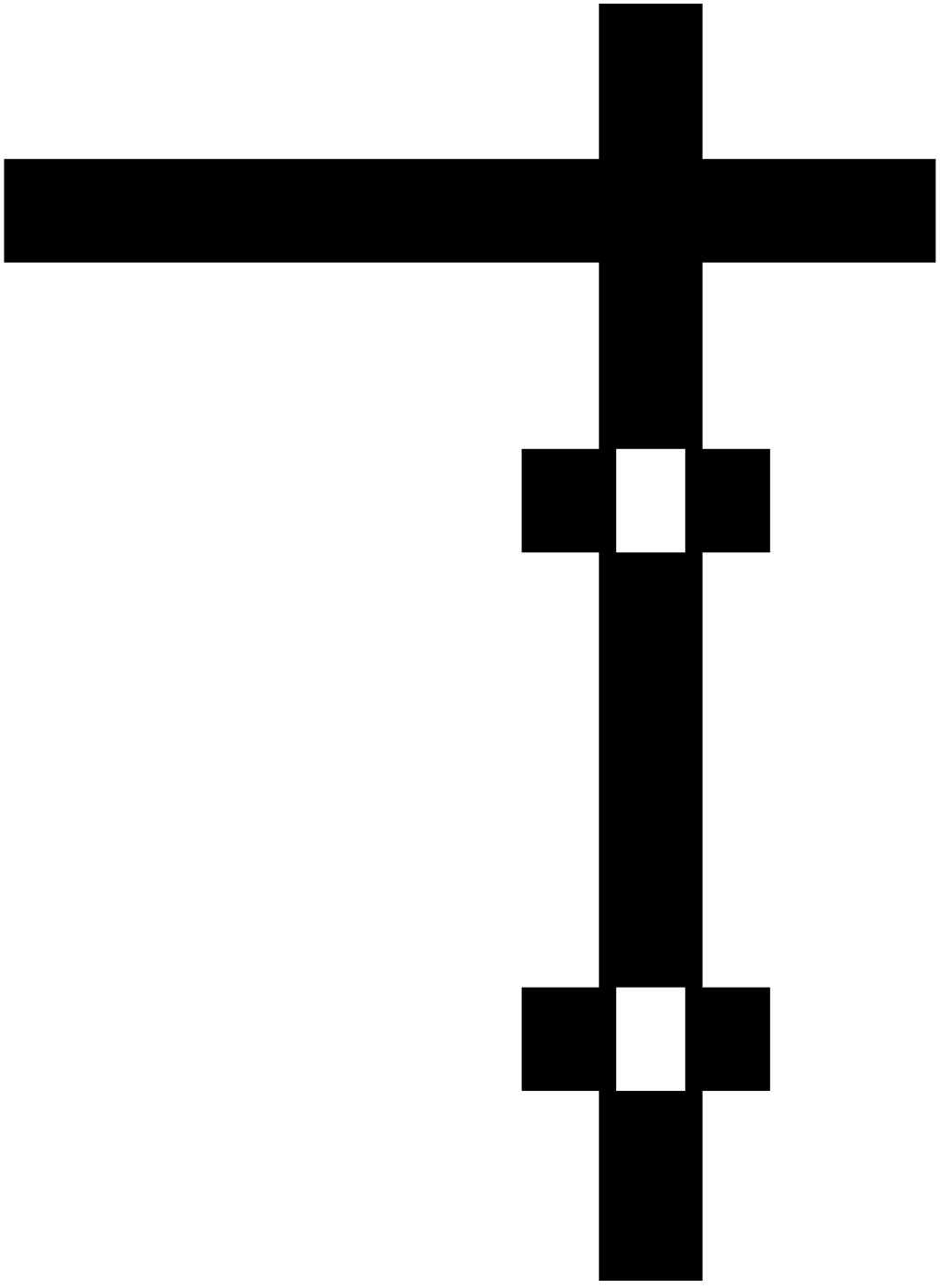
Line 3: sharp turn



Line 4: 90° turn



Line 5: turn signal right

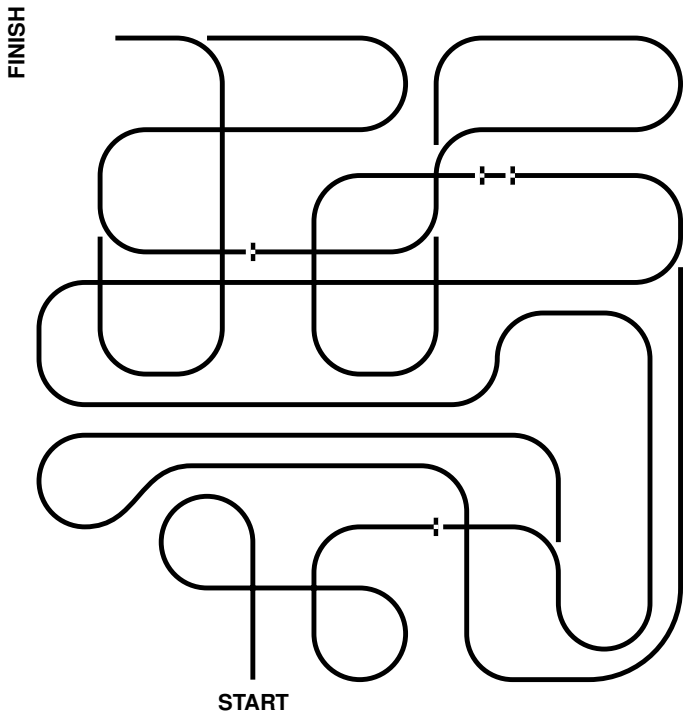


Line 6: turn signal left

Appendix D

Test Track

This is a small-scale version of the test track. Note that the actual track is somewhat different.



Appendix E

Using the Oscilloscope TDS 2022C

Introduction

An oscilloscope is a commonly-used electrotechnical measuring instrument. It is used to display variation in electrical voltage as a function of time. Whereas in the past analog oscilloscopes were primarily suitable for qualitative measurements (amplitude and shape of a signal), modern digital oscilloscopes can also be used to carry out quantitative measurements (determining the value of a signal). The lab is equipped with the Tektronix TDS 2022C digital storage oscilloscope. This appendix will briefly discuss the operation of this device. For more information, please consult the use manual that is available in the lab room.

E.1 Overview

A diagram of the TDS 2022C is depicted in Figure E.1. The buttons on the devices are grouped, based on functionality.

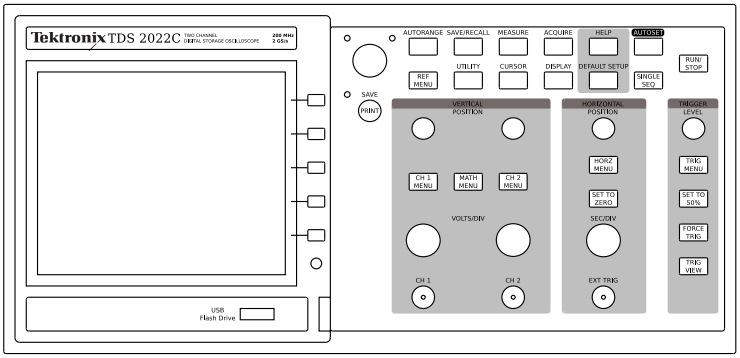


Figure E.1: The Tektronix TDS 2022C Digital Storage Oscilloscope

E.2 Basic Operation

The basic operating elements of this oscilloscope are more or less the same as those of an analog oscilloscope. The most important part of an oscilloscope is, of course, the display. The waveforms of the input signals are visualized on this display. The screen is divided into squares, in this case: 8 vertical and 10 horizontal squares. These squares are called *divisions*, the abbreviation of which is *divs*. The time is shown on the horizontal axis, the amplitude of the signal on the vertical axis.

E.2.1 Vertical Position

The vertical position group of knobs and buttons are used to modify the vertical scale and position of the waveform. The two large knobs are used to change scale of their respective channels. This is expressed in VOLTS/DIV, the actual setting is displayed in the bottom left corner of the screen, the first channel being preceded by the text CH1, the second channel by CH2.

The two smaller knobs are used to move the 0 V reference level. This can be useful when displaying two signal simultaneously. The actual setting is shown at the bottom of the screen while turning the knobs, and indicated with a small arrow and the channel number on the left of the screen.

CH1 and CH2 menu

The CH1 and CH2 menus are used to set a number of functions for the two channels. A particular channel can be switched on or off by pressing the channel menu button for that channel twice.
The menu comprises the following options:

- Coupling (GND, DC, AC)

  This is used to read out the input signal as GND (handy for adjusting the oscilloscope), DC (this is used to measure a DC component in the signal along with the rest) or AC (this is used to filter out a DC component).

- BW Limit (OFF(200MHz), ON(20MHz))

  This is used to limit the maximum bandwidth of the oscilloscope.

- Volts/Div (Coarse, Fine)

  This is used to select fine or coarse tuning using the volts/div button.

- Probe ($1\times$, $10\times$, $100\times$, $1000\times$)

  This is used to compensate for the state of attenuation of the probe.

- Invert (OFF, ON)

  This is used to invert the selected channel.

MATH menu

Calculations relating to the input signal can be carried out with the aid of the MATH menu. This menu can be used to determine the difference between and the sum of signals.

E.2.2 Horizontal Position

The large knob is used to change the horizontal scale (time axis). This is expressed in SEC/DIV, the actual value is shown at the bottom right of the right. It is possible to set two different vertical scales, but only a single horizontal scale, so both channels use the same time base setting.

The small rotary knob for the horizontal position is used to horizontally move the 0 s reference level. The SET TO ZERO button can be used to quickly reset the horizontal position.

E.2.3 Trigger Level

The trigger level is used to determine when to start data acquisition and when to display the waveform. The trigger level is a voltage level indicated by a small arrow on the right of the screen. The waveform is shown starting on the intersection point of the trigger level and the 0 s horizontal position. The trigger level can be changed with the small rotary knob.

TRIG MENU

The trigger menu allows you to change trigger settings. The following options are available:

- Type (Edge, Video, Pulse)

  This is used to select the type of trigger event. Usually Edge is the correct setting.

- Source (CH1, CH2, Ext, Ext/5, AC Line)

  This is used to select the source signal. Normally you should only use CH1 or CH2.

- Slope (Rising, Falling)

  Select the type of edge used to trigger.

- Coupling (DC, Noise Reject, HF Reject, LF Reject, AC)

  Allows filtering of the input signal of the trigger circuitry.

SET TO 50%

This button sets the trigger level to the midpoint of the peaks in the trigger input signal.

FORCE TRIG

Force a trigger to start the data acquisition.

TRIG VIEW

While this button is pressed down, the display shows the trigger input signal. This can be useful when trigger coupling differs from the channel settings.

E.2.4 Halting Acquisition and Single Secquence

Data acquisition can be stopped by pressing the RUN/STOP button. This will also freeze the display. Note that the functions of the MEASURE menu (see E.3.1) still work.

In order to capture a single-shot signal, press the SINGLE SEQ button. After the input has triggered the oscilloscope, it will automatically halt and shown the captured waveform on the display.

E.2.5 AUTOSET

This oscilloscope is equipped with functionality to automatically select the correct settings for the time base, voltage scale and the coupling on the basis of the input signals. This takes place by means of the AUTOSET button. Although this may be very useful, the autoset functionality does not work well if the signal to be measured has a very low frequency.

E.2.6 AUTORANGE

This oscilloscope can also automatically set both the horizontal and vertical scales as well as the vertical signal positions. This is done by pressing the AUTORANGE button. If the autorange function is active, the indicator light on the left of the button will be on. Pressing the button again will turn the autorange function off.

E.3 Menus

The various functions of the oscilloscope can be operated by means of menus. If you wish to call up a particular menu, simply press the button with the relevant text. You can then select the settings you want from the menu using the five buttons next to the screen.

E.3.1 MEASURE menu

As already mentioned, a digital oscilloscope can also be used to take measurements. On this digital oscilloscope, the MEASURE menu is used for this purpose. Various measurements can be taken at the same time, the results being displayed on the menu.

The topmost menu selector button switches between `Source` and `Type`. The option selected will affect how the other menu buttons subsequently work. If `Source` has been selected, the channel to be measured is set using the other buttons. The `Type` button selects the type of measurement. The following options are available:

None no measurement;

Freq displays the frequency of the signal;

Period displays the time period of the signal;

Mean displays the average value of the signal;

Pk-Pk displays the peak-peak value of the signal;

Cyc RMS displays the RMS (root mean square) value of the signal;

Min displays the minimum value of the signal;

Max displays the maximum value of the signal;

Rise Time displays the time between 10% and 90% of the first rising edge;

Fall Time displays the time between 90% and 10% of the first falling edge;

Pos Width displays the time between the 50% level of the first rising edge the next falling edge;

Neg Width displays the time between the 50% level of the first falling edge and the next rising edge

Note that at least one full period of the signal must be visible in order for these measurements to be accurate.

E.3.2 CURSOR menu

Besides the option in the Measurement menu, you can use the cursors from the CURSOR menu. Cursors can be horizontal (voltage) or vertical (time). Vertical cursors can track the active channel and show the voltage level at a specific time instance. They can also be used as a visual reference level, which can be useful to examine the behaviour of a circuit when it is subjected to a sweep input.

The first button in the CURSOR menu can be used to select the type of cursor:

- If you select `Off`, no cursor will be shown;
- If you select `Voltage`, horizontal cursors will be shown;
- If you select `Time`, vertical cursors will be shown.

The second button enables you to change between different sources:

- `Ref A`
- `Ref B`
- `CH1`
- `CH2`
- `Math`

The third menu option shows the absolute value of the difference between the two cursors. This option is not selectable and cannot be changed.

The fourth and fifth menu options show the position of the cursors. The active cursor is highlighted and can be moved by means of the general purpose rotary knob on the top left of the buttons.

E.3.3 ACQUIRE menu

The acquire menu can be used to modify the data that is used to draw the signals on the screen. The following options are possible:

Sample this is the default, it shows directly the sampled data;

Peak Detect this can be useful when measuring a noisy signal, low amplitude noise is slightly dimmed so spikes are easier to see;

Average the displayed signal is the average of an adjustable number of samples. This can be useful to filter out noise

In the upper left corner of the screen is an icon indicating the type acquire option.

E.3.4 DISPLAY menu

The display menu can be used to modify the way the signals are visualized.

E.4 Advanced Options

This digital storage oscilloscope has some advanced options that can be useful.

E.4.1 Storing Screenshots and Data

The oscilloscope can store and recall data from a USB flash drive. Insert a flash drive into the USB port[1] and wait while the oscilloscope examines the drive.

Press the SAVE/RECALL button to setup the action you wish to perform. The following actions are the most useful:

Save Image this option will store a screenshot image in a selectable data format;

Save Waveform this option will store the acquired data points of the selected channel in a CSV spreadsheet file;

Save All this option will store a screenshot, the acquired data points of both channels, and the settings of both channels

Saving the data will take some time, do not remove the flash drive while the oscilloscope is still writing data!

You can also couple an action to the PRINT button in order to quick access to a save action.

E.4.2 FFT

The oscilloscope also features a low frequency digital spectrum analyzer. This option can be found in the MATH menu by selecting FFT as type of Operation. You can specify the source signal and horizontal and vertical resolution with the VOLTS/DIV and SEC/-DIV knobs. The horizontal position knob can be used to select the center frequency on the display.

[1] The oscilloscope can only flash drives up to 64 GB capacity, and only when formatted with FAT32. The newer exFat (FAT64) format will not work.

E.5 Probes

Measurement on an oscilloscope are preferable done with a probe. Probes that attenuate the input signal contain a circuit that has to be tuned before the probe is used. Tuning can be done by connecting the probe to the Probe Comp connector. This output will provide a 5 V, 1 kHz square wave input. If the output on the display is distorted, the probe can be adjusted by turning the small screw in the probe (either near the probe tip, or near the BNC connector). Note that probes with an adjustable attenuation factor bypass this circuitry in the $1\times$ position.

Appendix F

Using the Function Generator Tektronix AFG 3021C

Introduction

The lab is equiped with the Tektronix AFG 3021C arbitrary function generator. This appendix will briefly discuss the operation of this device. For more information, please consult the user manual that is available in the lab room.

F.1 Overview

A diagram of the AFG 3021C is depicted in Figure F.1. The buttons on the devices are grouped, based on functionality.

F.1.1 Run Mode

On the top, there are four buttons grouped as Run Mode. These perform the following functions:

Continuous continous operation, selected signal is continously outputted;

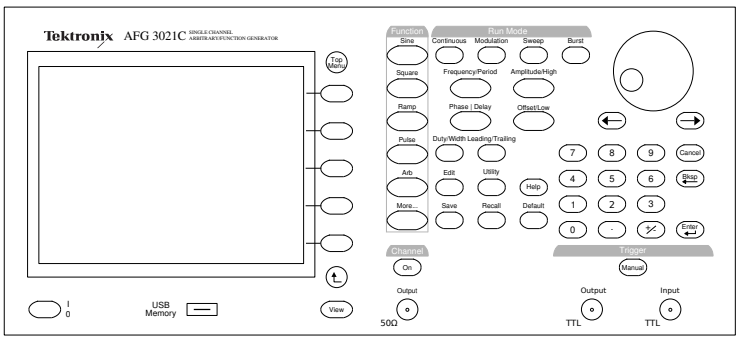


Figure F.1: The Tektronix AFG 3021C Arbitrary Function Generator

Modulation perform AM, FM, PM, of FSK modulation;

Sweep sweep output over a range of frequencies;

Burst output one or more bursts of a signal

F.1.2 Function

Directly left from the Run Mode group is the Function selection. These allow quick a choice between the following waveforms:

Sine a sine wave with adjustable frequency, amplitude, and phase;

Square a square wave with adjustable frequency, amplitude, and phase;

Ramp a ramp wih adjustable frequency, amplitude, phase, and symmetry;

Pulse a pulse with adjustable frequency, amplitude, delay, and duty cycle;

Arb an arbitrary waveform, fully adjustable;

More... select sinc, Guassian, Lorentz, Exponential Rise or Decay, or Haversine functions or white noise

F.1.3 Signal Setup

Under the Run Mode buttons are buttons to set up the selected waveform. Each button has two (related) functions, repeatedly pressing the button will alternate between the functions.

Frequency/Period select the frequency or period of the signal;

Amplitude/High select the amplitude or the high voltage level;

Phase|Delay select the phase shift or the delay;

Offset/Low select the offset or the low voltage level

When the Pulse waveform is selected, two additional buttons are available:

Duty/Width select the duty cycle or pulse width

Leading/Trailing select the leading and trailing time

Pressing one of these buttons highlights the corresponding setting on the screen. The value can now be changed by either typing in the numerical value on the keypad and pressing Enter to confirm, or using the general purpose rotary knob. The arrow keys directly under the general purpose knob move the cursor and thus the digit changed when turning the general purpose knob.

The screen shows also shows a preview of the signal annotated with all the important voltage levels and frequencies. These levels are only valid when the output impedance is properly set up, as described in §F.2.

F.1.4 Additional Signal Setup

The buttons directly next to the screen are context-sensitive and are used to modify additional signal parameters. Changing the parameters is identical as descibed in §F.1.3.

F.2 Output and Output Impedance

The output of the generator is the leftmost coax connector labeled Output. The output impedance of this output is adjustable. Changing the output impedance will have an influence on the settings of the generator. The amplitude settings on the generator are only valid when the actual output impedance is identical to the set output impedance. By default, the output impedance is set to 50 Ω. The input impedance of, for example the oscilloscope, op amp inputs and almost any digital IC input is very high. In order to have the actual voltage levels match the settings on the function generator, the output impedance has to be changed to high impedance. You can use the following steps to set the output impedance to high impedance.

- Top Menu (this is the upper button next to the screen)
- Output Menu
- Load Impedance
- High Z

The lower button can be used to return to the main screen.

Note that the output settings are only valid for a fixed load impedance; if the load is highly variable, make sure to verify the actual output on an oscilloscope or use a buffer circuit.

The output signal is enabled or disabled by pushing the On button in the block Channel just above the output connector.

Appendix G

Using the Tektronix PWS 4205 Power Supply

G.1 Introduction

The lab is equipped with the Tektronix PWS 4205 power supply. This appendix will briefly discuss the operation of this device. For more information, please consult the user manual that is available in the lab room.

G.1.1 Overview

The PWS 4205 is a 0–20 V programmable power supply, capable of delivering a current of 5 A. A diagram of the PWS 4205 is shown in Figure G.1.

G.2 Basic operation

Turn on the device with the green POWER button. The PWS 4205 has a double-line display that is used to show both the voltage and current settings and the actual values. The top line shows the actual output voltage and output current, the bottom line shows the maximum voltage and current settings.
Always specify both the voltage and current settings:

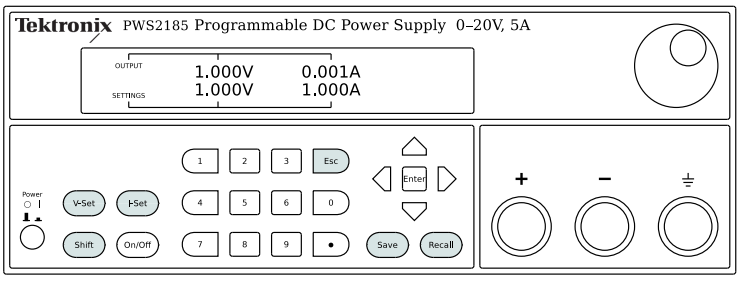


Figure G.1: Overview of the PWS 4205 power supply

- Press the V−SET button to specify the maximum voltage. Use the arrow keys, the rotary knob, or the number keys to change the maximum voltage. Confirm with the ENTER key.
- Press the I−SET button to specify the maximum current. Use the arrow keys, the rotary knob, or the number keys to change the maximum current. Confirm with the ENTER key.
- Press the ON/OFF to enable the output.

G.3 Display

The following status text can appear in the display:

OFF The output is disabled. Use the button ON/OFF to enable the output.

CV The power supply runs in constant voltage mode. This means the requested voltage is equal to or higher than the voltage specified with the V−SET button.

CC The power supply runs in constant current mode. This means the requested current is equal to or higher than the current specified with the I−SET button.

Appendix H

Simulating VHDL code with GHDL and GtkWave

H.1 Introduction

The student edition of the VHDL simulator ModelSim can be downloaded for free for users of Windows. If, however, you're using a Unix-like operating system such as Linux, *BSD or Solaris, this is not a viable option. The open source tools GHDL and GtkWave provide an easy way to perform VHDL simulations.

H.2 GHDL

GHDL (`http://ghdl.free.fr/`) is an open source VHDL compiler. GHDL can be used to simulate VHDL code. Contrary to ModelSim, GHDL is commandline tool, there is no graphical interface. The result of the simulation is a waveform file (`.ghw`) that can be viewed with GtkWave.

H.2.1 Analyse

The first step of performing a simulation, is analysing the VHDL file. GHDL will compile the files and creates a database that contains the available entities. The analyse command can be used as follows:

```
ghdl -a file.vhdl
```

If all goes well, the analyse command has generated a file `file.o`. If there was a problem, GHDL will provide a error message.

H.2.2 Elaborate

When all required VHDL files are processed with the analyse command, GHDL can create an executable from the top-level entity. This top-level entity has to be a test-bench. The elaborate commands expects the name of the top-level entity as a parameter, not a filename. The elaborate command is used as follows:

```
ghdl -e top-level_entity
```

If the command is succusfull, you should have an executable with the name of the top-level entity.

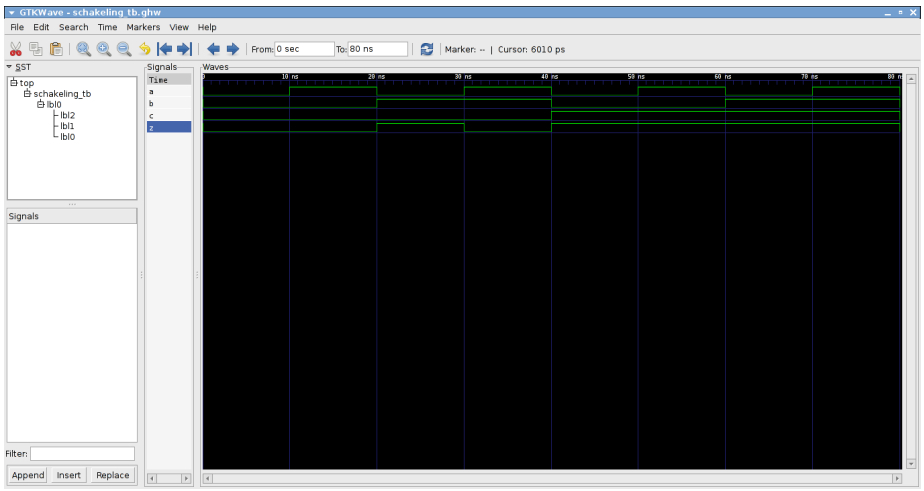


Figure H.1: Screenshot of GtkWave

H.2.3 Simulate

The executable created with the elaborate command can now be used to perform the simulation. This can also be done via GHDL:

```
ghdl -r top-level_entity flags
```

The flags are used to provide an end time and the filename of the waveform file:

```
ghdl -r top-level_entity --wave=filename.ghw --stop-time=100ns
```

After a succesful simulation, the waveform file `filename.ghw` is created. You can use GtkWave to view this file.

H.3 GtkWave

GtkWave (`http://gtkwave.sourceforge.net/`) is a waveform viewer and can be used to view the result of a GHDL simulation. GtkWave must be started with the name of the waveform file as a paramter: `gtkwave waveform.ghw`
By providing the option `--save=savefile.sav` it is possible to save the current configuration (zoom-level, signals shown etc). This can save a lot of time during consecutive simulations.

The user interface of GtkWave is very similar to that of ModelSim and easy to use. A screenshot of GtkWave is depicted in Figure H.1. This screenshot shows the results of the simulation of a simple circuit. The largest part of the GtkWave window is used for displaying the waveform. On the left there is hierarchical view of the system. This allows you to select a part of the design. The corresponding signals are shown left below. By dragging the name of such a signal to the waveform, the signal is added. This is different from working with ModelSim: in ModelSim you need to re-simulate the design when you add a signal after the initial simulation. GtkWave uses the waveform result from GHDL, this contains the simulation result of *all* signals in the design.

Appendix I

Circuit Simulator from `http://www.falstad.com/`

Introduction

When designing circuits, a simulator can be a very helpful tool. Probably the most well-known circuit simulator is Spice. There are many different variants of Spice simulators available, both open source or commercial closed source versions. Spice has a relative steep learning curve and isn't interactive, one runs a simulation and then verifies the results. The falstad simulator is much simpler but easier to use than Spice. It is an interactive tool, meaning you can directly see the results of you altering your circuit.

I.1 Running the simulator

The simulator is written as a Java applet and be run directly from within a browser. Just point a Java-enabled browser to: `http://www.falstad.com/`. It is also possible to run the simulator offline. Just download and unpack the file `circuit.zip` from the website. You can run the program by double-clicking the file `circuit.jar`.

Permissions
Under Linux, you should make the file executable in order to run it!

When started, the simulator should look like the screenshot of Figure I.1. In the large area, you can build your circuit, the controls on the right are used to control the simulator.

I.2 Building and simulating a simple circuit

I.2.1 Building the circuit

In order to familiarize yourself with the simulator, we will build and simulate the simple circuit from Figure I.2.

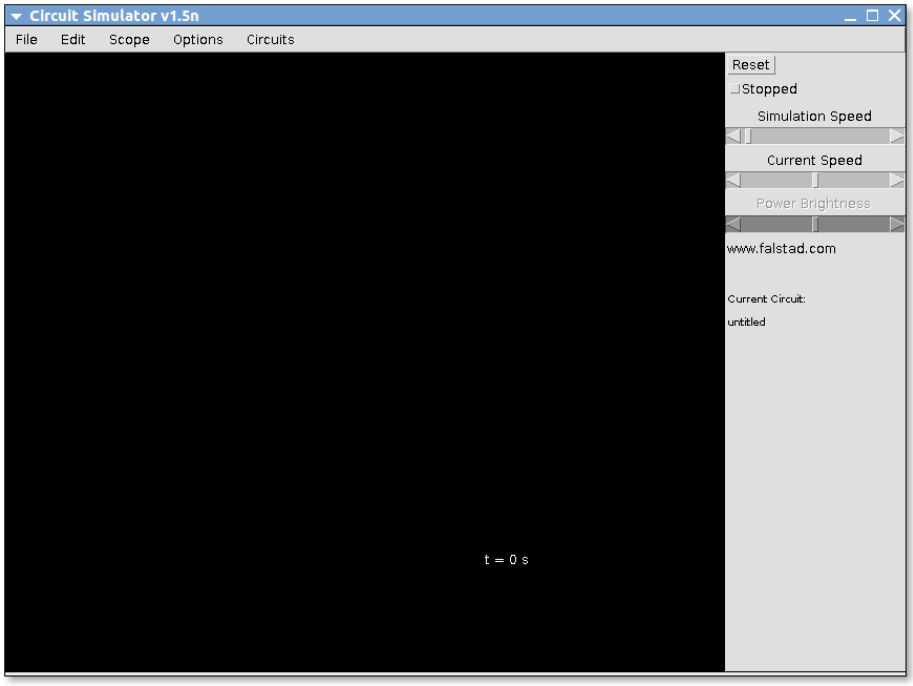


Figure I.1: Screenshot of the circuit simulator

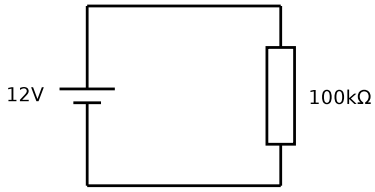


Figure I.2: Simple example circuit

You can add components by right-clicking on an empty part of the circuit area. Select a DC voltage supply with Inputs/Outputs → Add Voltage Source (2-terminal) and place it on the circuit area by dragging the mouse over the preferred location. Moving or changing the orientation of a component can be done by clicking and dragging the component or its terminals while holding a key:

- holding Shift ⇧ will move the component
- holding Ctrl will move a terminal

Next, add a resistor to the circuit (right click, Add Resistor). Now connect the voltage source and the resistor with wires (right click, Add Wire).

European resistors
If you want your resistors to show up in European style, select Options→European Resistors.

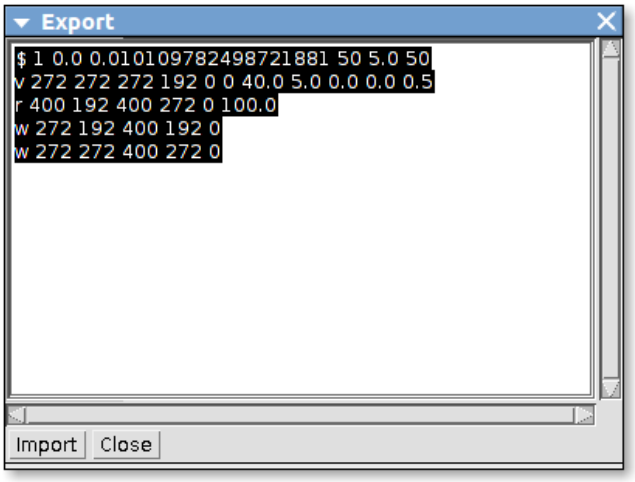


Figure I.3: Export dialog from the circuit simulator

Now we have the topology of the circuit, but the values of the resistor and the voltage supply are not correct yet. Select the voltage supply by hovering the mouse over it, the color should change to blue. Now right click and select Edit. You can now set the correct value (12V). Repeat this procedure for the resistor. The simulator accepts the value 100k.

I.2.2 Simulating the circuit

Now that the circuit is complete, you can start the simulation by clicking the ticked box with the label Stopped on the right. The current is shown as moving yellow blocks, the speed is related to the magnitude of the current. You can verify the voltage and current levels, the power dissipated in components etc by hovering the mouse over the component of interest.

I.2.3 Adding a scope

You can easily add a scope to circuit components, simply hover over the component, right click and select View in Scope. This will open a scope view at the bottom of the simulator. You can adjust the properties of the scope by hovering over the scope view and right clicking. Of course this isn't of much interrest for the example circuit.

I.3 Saving and loading a circuit

The simulator doesn't have a save or load option, but you can load and store circuits using the import and export options in the File-menu. To save the example circuit, select File→Export. This will open a dialog similar to the one in Figure I.3. By selecting the text and using copy and paste actions to a texteditor (Notepad, gedit etc), you can save the circuit. In order to load a circuit, open the saved file in a texteditor, copy all the text and past it in the import-dialog (File→Import) of the circuit simulator.

Unable to copy
In some Linux distributions, the web-version of the simulator won't allow you to import or export something. You can of course manually type over the required information, but that is only an option for very simple circuits. If you run into this problem, download the simulator from the website and try running it locally. Make sure to start it from within the directory where the file `circuit.jar` is extracted or you will miss some functionality!

I.4 Tips and warnings

Finally some tips and warnings on the use of this simulator:

- Always connect each and every component with a separate wire. Failure to do so could result in the message <number> bad connections. These connections are shown in red.
- If the simulator gives unexpected results, try the reset button (top-right).
- You can adjust the speed of the simulator and the current with the sliders on the right.
- Under Circuits you can find many example circuits.

Bibliography

[1] P. J. Ashenden, *The student's guide to VHDL*, Morgan Kaufmann Publishers, Inc, San Francisco, 1998

[2] N. Storey, *Electronics; A Systems Approach*, fourth edition, Addison-Wesley, 2009

[3] N. Mohan, T.M. Undeland, W.P. Robbins, *Power Electronics; Converters, Application and Design*, John Wiley & Sons, Inc, 2003