

Code EE1D21

# Manual

## CourseLab Digital Systems B

### Line-follower

13<sup>th</sup> February, 2025

Ing. X. van Rijnsoever, Ing. T. Slats, Dr. J. Hoekstra  
Dr. ir. A.J. van Genderen, Ing. B.M. Verdoes

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Getting to know the Robot</b>	<b>2</b>
2.1	Introduction . . . . .	2
2.2	Overview of the robot . . . . .	2
2.2.1	Only Using the FPGA Board . . . . .	4
2.2.2	Theory of Line-tracking Sensor . . . . .	5
2.2.3	Theory Motors . . . . .	5
2.3	Hardware: The Basys3 FPGA development board . . . . .	6
<b>3</b>	<b>Overview of Line-Follower</b>	<b>8</b>
<b>4</b>	<b>The Time base</b>	<b>10</b>
4.1	Introduction . . . . .	10
4.2	Overview of the time-base . . . . .	10
4.3	Complete the interface of the time-base . . . . .	11
4.4	Implement the contents of the time-base . . . . .	11
4.5	Make a testbench . . . . .	12
<b>5</b>	<b>Motor control</b>	<b>13</b>
5.1	Introduction . . . . .	13
5.1.1	Moore and Mealy machines . . . . .	14
5.1.2	FSM in a state-diagram . . . . .	15
5.1.3	The construction of a state-diagram . . . . .	16
5.1.4	From state-diagram to SystemVerilog description . . . . .	17
5.1.5	Considerations when designing FSM's . . . . .	18
5.2	Exercise: Design and implement the motor control . . . . .	18
5.2.1	Simulating the motor control . . . . .	20
<b>6</b>	<b>Input buffer</b>	<b>22</b>
6.1	Introduction . . . . .	22
6.2	Exercise: Implement the input buffer . . . . .	23
<b>7</b>	<b>The Controller</b>	<b>26</b>
7.1	Introduction . . . . .	26
7.2	Exercise: Design and implement the controller . . . . .	26
7.2.1	Design a state-diagram for the controller of the line-follower . . . . .	26
7.2.2	Implement and simulate the state-diagram in SystemVerilog . . . . .	27

<b>8 Putting everything together</b>	<b>28</b>
8.1 Introduction . . . . .	28
8.2 Exercise: Assemble and simulate the robot . . . . .	29
8.2.1 The top-level module . . . . .	29
8.2.2 Simulation of the line-follower . . . . .	29
<b>9 Test the Robot</b>	<b>32</b>
9.1 Go through the Vivado tutorial . . . . .	32
9.2 Program the FPGA of the robot with your SystemVerilog code . . . . .	32
9.3 Test the robot . . . . .	33
<b>A Overview of the robot</b>	<b>34</b>
<b>B Test lines</b>	<b>35</b>
<b>C Vivado 2023.2 Tutorial</b>	<b>39</b>
C.1 Introduction . . . . .	39
C.2 Step-by-Step Tutorial . . . . .	41
<b>D Overview of the FPGA pins</b>	<b>48</b>
D.1 Selected IO connections . . . . .	48
D.2 Full list of connections . . . . .	49
<b>E Working with an oscilloscope</b>	<b>54</b>
E.1 Overview . . . . .	54
E.2 Basic Operation . . . . .	54
E.2.1 Vertical Position . . . . .	55
E.2.2 Horizontal Position . . . . .	55
E.2.3 Trigger Level . . . . .	56
E.2.4 Halting Acquisition and Single Sequence . . . . .	56
E.2.5 AUTOSET . . . . .	56
E.2.6 AUTORANGE . . . . .	57
E.3 Menus . . . . .	57
E.3.1 MEASURE menu . . . . .	57
E.3.2 CURSOR menu . . . . .	57
E.3.3 ACQUIRE menu . . . . .	58
E.3.4 DISPLAY menu . . . . .	58
E.4 Advanced Options . . . . .	58
E.4.1 Storing Screenshots and Data . . . . .	58
E.4.2 FFT . . . . .	59
E.5 Probes . . . . .	59
<b>Bibliography</b>	<b>60</b>

# Chapter 1

## Introduction

This manual will guide you step by step to write the SystemVerilog code for an FPGA that controls a line follower robot. In the first chapter, the hardware of the robot is described. Next, a subdivision of the design is proposed. The implementation of the different components of this design is described in the subsequent chapters. Finally, the last chapter describes how all components are put together and how the line follower can be tested.

In the manual, there are a number of frames showing a different icon. These frames contain additional information that is not directly necessary for the lab, but it is useful to know.



### **Attention**

The frame with this icon warns on common errors and problems.



### **Suggestion**

The frame with this icon provides a tip that can be useful in addressing a problem of the assignment.



### **Background information**

The frame with this icon provides background information about, for example practical applications, or additional functionalities that might not be required for the exercise, but might be applicable to other labs.

## Chapter 2

# Getting to know the Robot

### Learning Objectives

At the end of this session you will have knowledge of:

- the structure of the robot,
- the power source of the robot,
- the type and functionality of the sensors,
- the servo motors of the robot,
- the software to realize digital control.

### 2.1 Introduction

In this chapter you will familiarize with the various elements of the robot. You will first examine all the hardware components of the robot, then the software to control the robot.

### 2.2 Overview of the robot

A photo of the robot is shown in Figure 2.1. 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 2.2 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):

- 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.

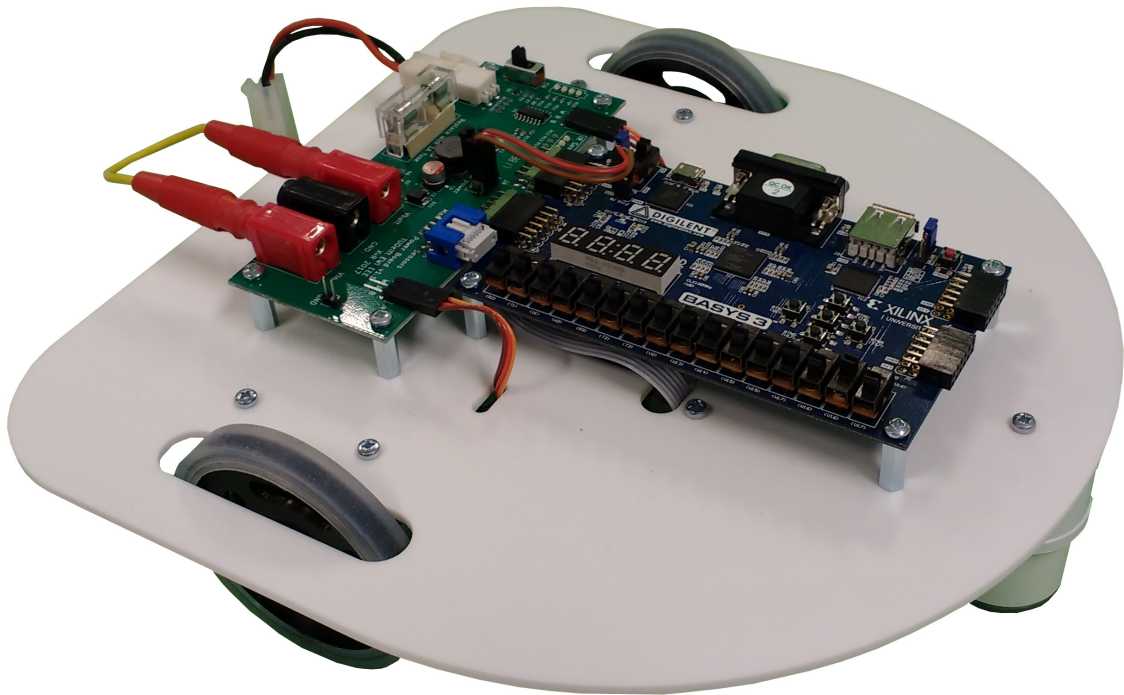


Figure 2.1: Photo of the robot

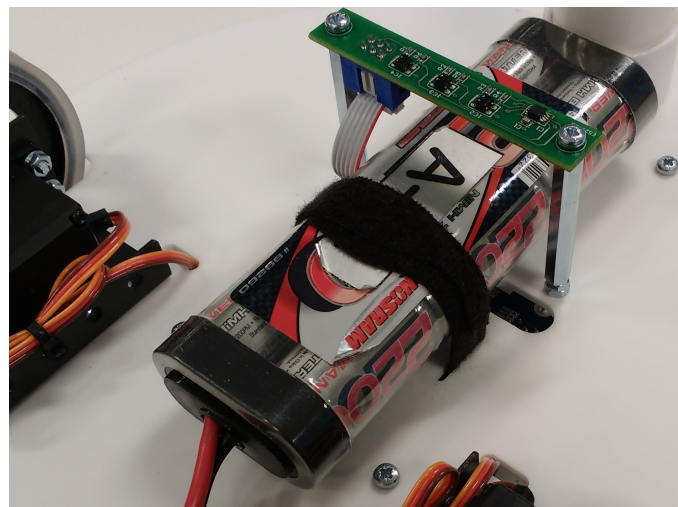
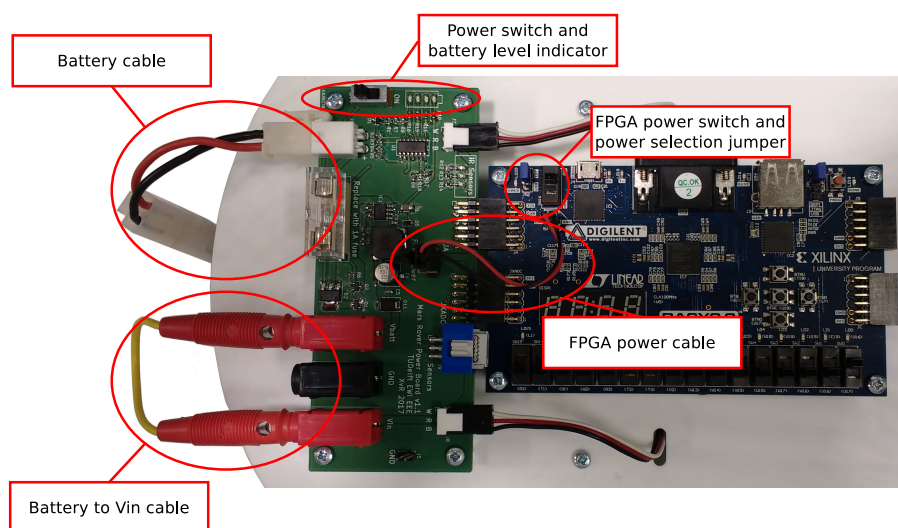


Figure 2.2: Photo of the bottom side of the robot with the battery properly mounted



**Figure 2.3:** 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 2.3 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 position 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.

### 2.2.1 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 Vivado tutorial in the appendix.

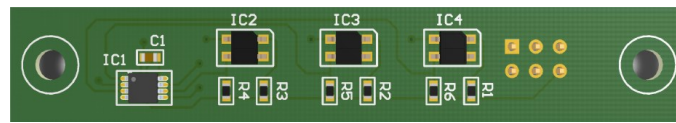


Figure 2.4: Bottom side of the IR sensor showing the three reflective object sensors

### 2.2.2 Theory of Line-tracking Sensor

The line-tracking sensor 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 2.4. 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<sup>1</sup>. 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 2.5.

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.

### 2.2.3 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 robot are special continuous 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 2.6.

<sup>1</sup>[http://en.wikipedia.org/wiki/Schmitt\\_trigger](http://en.wikipedia.org/wiki/Schmitt_trigger)



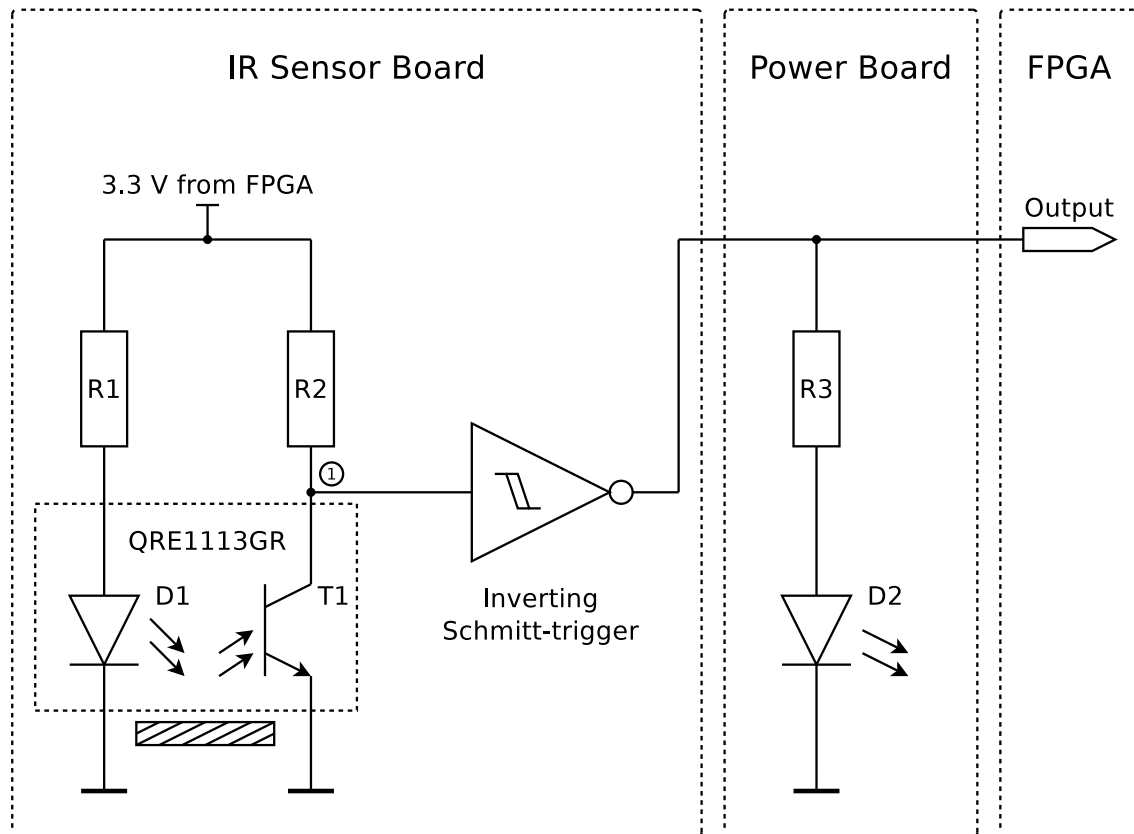


Figure 2.5: Diagram of a sensor

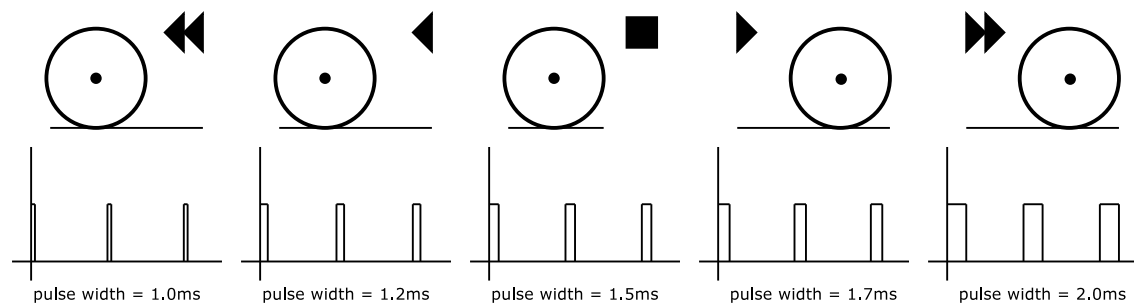


Figure 2.6: Effects of PWM on the servo motor

## 2.3 Hardware: The Basys3 FPGA development board

The Basys3 FPGA development board is a printed circuit board (PCB) designed by Digilent<sup>2</sup> that features the Atrix-7 FPGA by Xilinx<sup>3</sup>. 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 of communication (RS232 or CAN), a PS/2 port for a mouse or keyboard, or a VGA connector for a video screen.<sup>4</sup>

A photo of the board is shown in Figure 2.7.

<sup>2</sup><http://store.digilentinc.com/>

<sup>3</sup><https://www.xilinx.com/>

<sup>4</sup>Don't worry if the names RS232, CAN or PS/2 don't ring a bell, they're only used here as examples

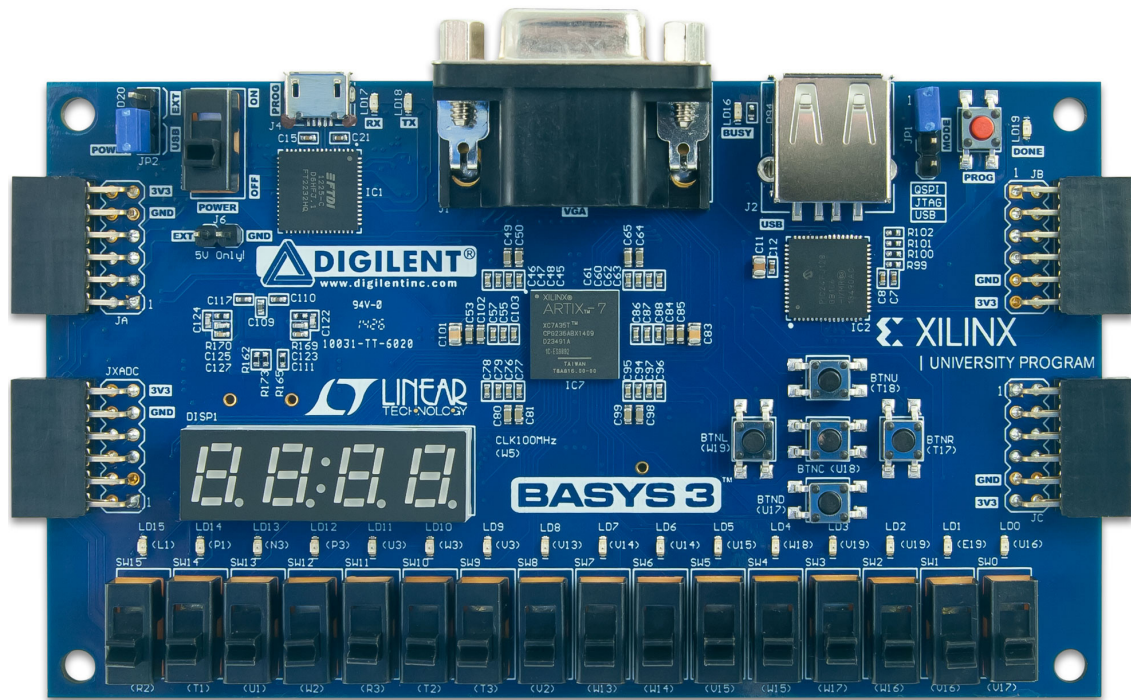


Figure 2.7: Top view of the Basys3 FPGA development board

The large square IC in the middle is the Artix-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.

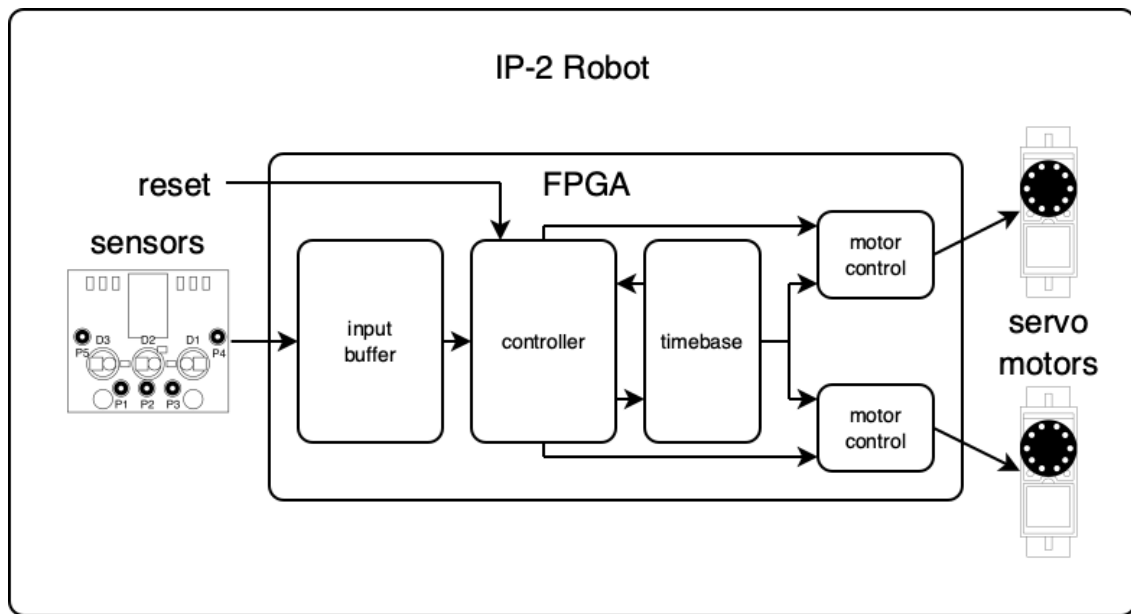
## Chapter 3

# Overview of the modules for the line-follower

In this manual, we will make a distinction between four different hardware modules in the design of the digital control for the linefollower. These are (see also the diagram on the FPGA in Figure 3.1):

- **The input-buffer.** The input-buffer ensures that the signals from the light sensors are synchronized with the clock and hence are read in a correct manner.
- **The controller.** The output values of the sensor circuit are then passed on to the controller. The controller calculates, on the basis of these values, in which direction the robot should drive, and also manages the motor control and time base.
- **The time base.** The time base is a counter from which a time can be measured.
- **The motor control (2 instances are used).** Each motor control instance drives a motor by generating a PWM signal. The time base is used to create the correct timing for the PWM signals.

Each of these modules will be designed in the subsequent chapters. Templates for the 4 different modules are already given in the file `entities.zip`. Take a look at these descriptions separately. They need to be completed and merged into a structural description as shown in Figure 3.1.



**Figure 3.1:** Schematic overview of the line-follower.

## Chapter 4

# Designing a counter: the time-base

### 4.1 Introduction

The time-base module is an important part of the robot, since it indicates the time reference for the motor control modules, which in turn control the motors by means of PWM signals. The time-base module will count the time within a period of the PWM signal. The time-base module will be implemented with a counter.

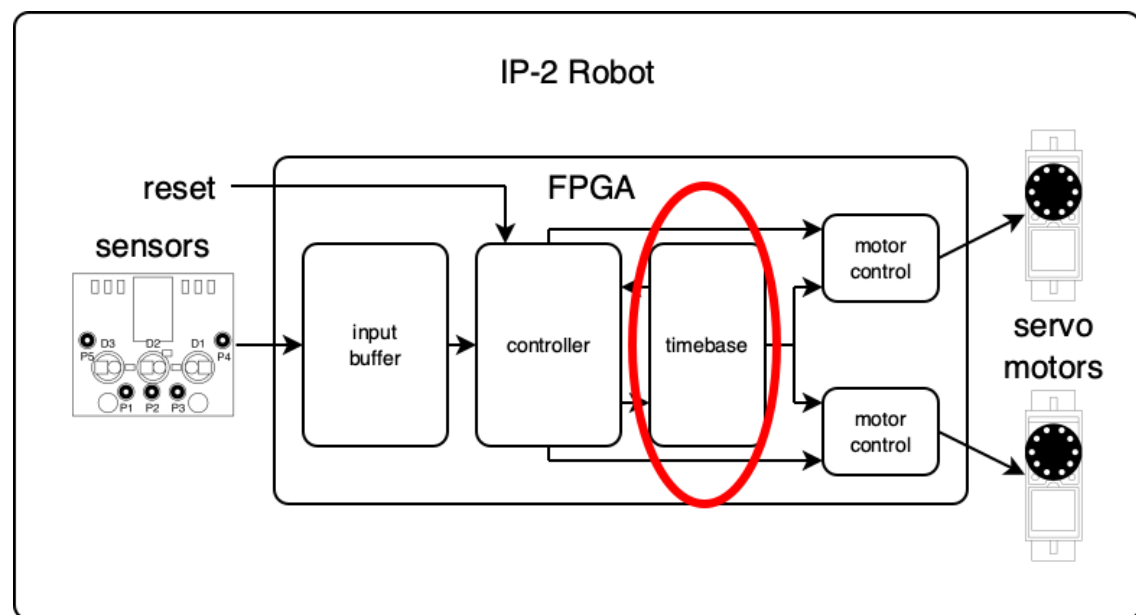


Figure 4.1: Implementing the timebase of the line-follower.

### 4.2 Overview of the time-base

For the counter we examine here, we assume a scheme as shown in Figure 4.2. The in- and output signals have the following meanings:

- *clk* is the 100 MHz system clock
- with the *reset*-signal, the counter can be placed on the null-setting.

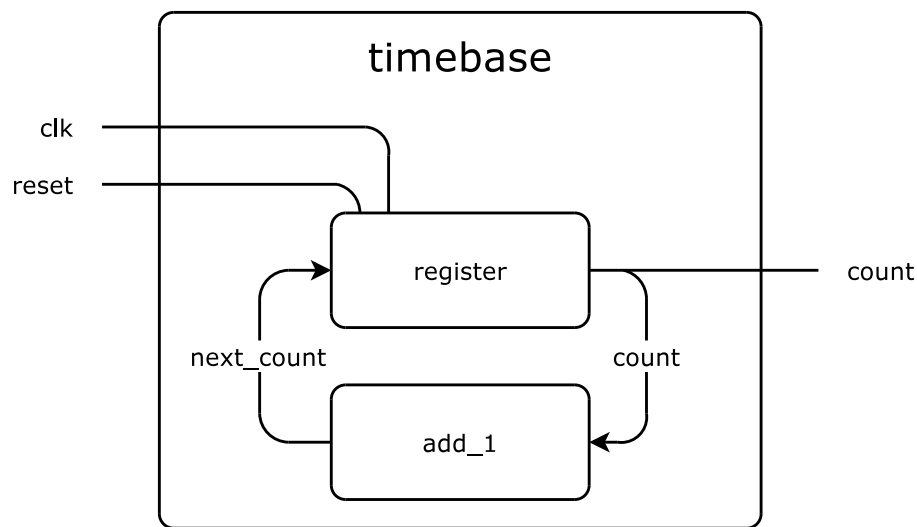


Figure 4.2: Implementing timebase as a counter.

- the *count*-signal is the counters output and contains the current counter condition

Like mentioned before, the counter is used as a time reference in order to generate a PWM signal. This means that the counter should be (at least) able to count the period of time of a PWM pulse. Furthermore, you should be able to reset the counter using a synchronous reset.

### 4.3 Complete the interface of the time-base

Use the SystemVerilog file `timebase.sv` to create a header of the time-base with in- and outputs as shown in Figure 4.2. The signal *count* contains the current counter value. As the counter counts to more than 1, this signal will have to be a vector. Answer the following questions to determine how large the vector should be:

- Given the clock frequency and given the duration of one PWM period, to what number should the counter be able to count at least?
- How many bits are needed to present this number?

### 4.4 Implement the contents of the time-base

As depicted in Figure 4.2, the counter consists out of two parts:

1. A piece of memory to keep track of the counter value.
2. An adder that determines the new counter value.

Please refer to the lecture slides and/or the book, to see how a counter can implemented as described below.

**Implementing the register** The required memory has to be implemented in the form of flipflops, that means a `always_ff`-block is needed that implements the register. The value of *next\_count* is copied to the signal *count* at a rising clock edge, as usual in a register. If the *reset*-signal obtains the value '1', the counter has to be set back to 0.

**Implementing the adder** The addition can be accomplished using the + operator on count. The assignment to *next\_count* can be done using the `assign` statement or using an `always_comb` statement.

## 4.5 Make a testbench

To know whether the counter actually works, it must be tested with a testbench. You may want to use the following testbench for this (first update the range for signal count):.

```
1  `timescale 1ns/1ps
2
3  module timebase_tb();
4
5      logic clk;
6      logic reset;
7      logic [?:0] count;
8
9      timebase test (clk, reset, count);
10
11     always
12         #5 clk = ~clk; // period 10ns (100 MHz)
13     initial
14         clk = 0;
15
16     initial begin
17         reset = 1;
18         #10; reset = 0;
19     end
20
21 endmodule
```

## Chapter 5

# A simple FSM design: motor control

### Learning Objectives

During this session you will:

- learn the differences between Mealy and Moore style FSM's, and the advantages and disadvantages of both
- learn to design a FSM using a state-diagram
- learn to implement a FSM in SystemVerilog
- implement the motor control

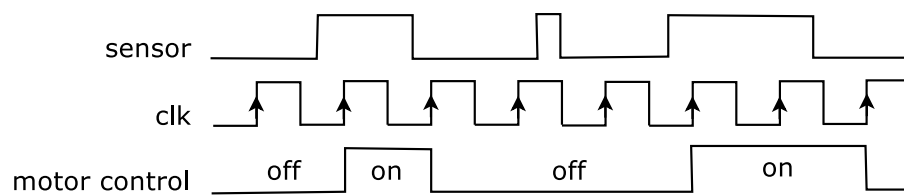
### 5.1 Introduction

The control of the robot and motor are examples of controllers. A controller is typically implemented as a FSM in hardware design. FSM is an abbreviation for **Finite State Machine**, a machine with a finite number of states. A FSM is a general applicable programming technique in which the desired behavior is modeled in a finite number of states. What is a state or condition? A state represents a step in the desired behavior. Under certain conditions, a state is entered, and under certain other conditions the state is departed again. In synchronous circuits (which are most circuits) this is done in a rhythm that is indicated by a clock signal. To illustrate this principle, an example of a FSM is given here. The robot is only able to drive forward and is equipped with a single sensor. The goal is (also see Figure 5.1) that every time during an active clock edge, the sensor value is sampled. Is the value '1', then the FSM should control the motor with a '1' during the next clock edge (to drive the robot forward). The FSM should give an 'o' to let the robot stand still when the sensor value is equal to 'o'. This behavior can be described by two states: one state wherein the robot drives, and another state in which the robot is stationary. Based on the sensor value, one state is left and the other is entered.

In order to ensure that the robot drives (or is stationary) for an entire clock period, it has to remember in what state it is. This means that the FSM has a memory that can store the current state. (Note that if the motor control does not have to stay constant during a clock period, the output signal can directly be derived from the input signal and a combinational circuit can be used).

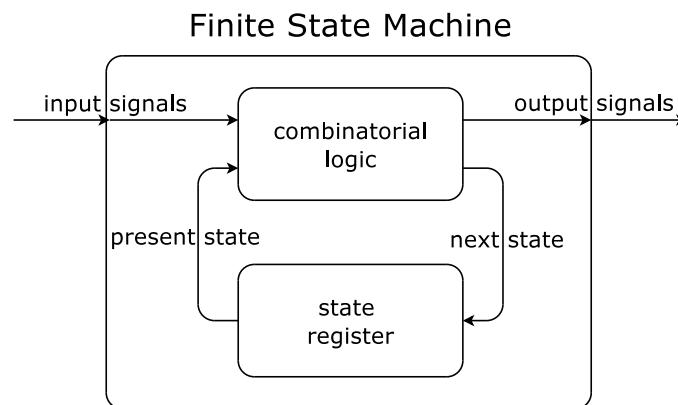
The value of the output signals (in case of the example, the motor control signal) depends on the state, and possibly the input values. The choice of a new state depends generally on the current state and the input values. This means that there has to be a formula or description to determine the output values and the new state.





**Figure 5.1:** The behavior of a simple FSM that controls the motor, based on the sensor value. Each time during an active (rising) clock edge, the value of the sensor determines the value of the motor control during the next clock period

In a software implementation of a FSM, the state is stored in a variable, and the values of the output signals and the new state are determined by a piece of program-code. In a hardware implementation the states are stored in memory (flipflops), while the output values and the new state are determined by combinational logic. The general structure of a FSM in hardware is depicted in Figure 5.2.



**Figure 5.2:** General model for a Finite State Machine

The state is determined every clock-cycle. This means that there is always at least one clock cycle spent in a state, even though there are some states that are unconditionally run. In this way, it is easily possible to wait a number of clock-cycles.

### 5.1.1 Moore and Mealy machines

There are two different types of FSM's: Moore and Mealy. The difference between both is, from a theoretical point of view, very simple:

- In Moore-style FSM's the value of the output signals is only dependent on the current state
- In Mealy-style FSM's the value of the output signals is dependent on the current state and the input signal values

In practice, however, it can sometimes be difficult to see whether you have a Moore- or Mealy-style FSM.

There are advantages and disadvantages to the different designs:

	Moore	Mealy
<b>Advantages</b>	Simple design Robust design	Compact design Sometimes faster than Moore
<b>Disadvantages</b>	Larger design than Mealy Can be slower than Mealy	Sensitive for glitches on the input Instability with feedback loop Can become complex fast

Although Mealy machines often contain fewer states than Moore machines, it is very difficult to keep the disadvantages under control. In the lab we will therefore only use Moore machines.

The hardware implementations of a schematic depicted Moore- and Mealy-style FSM looks like Figure 5.3. Note that in this figure (compared to Figure 5.2) the combinational part is shown separately to indicate the distinction between the Moore and Mealy FSM. Both combinational parts are usually modeled with a single SystemVerilog `always_comb` statement. The state register section is then modeled with an `always_ff` statement.

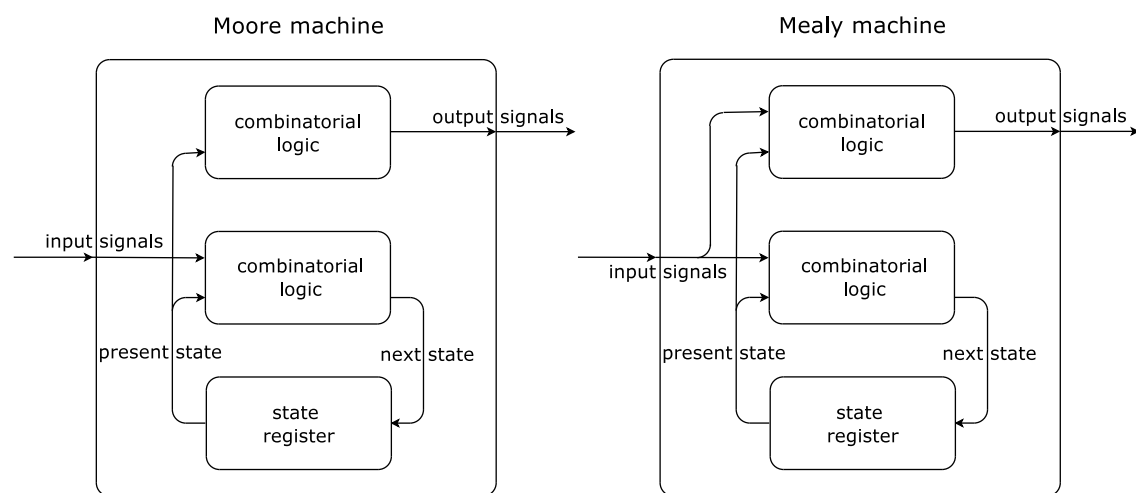


Figure 5.3: Model for a Moore FSM and a Mealy FSM

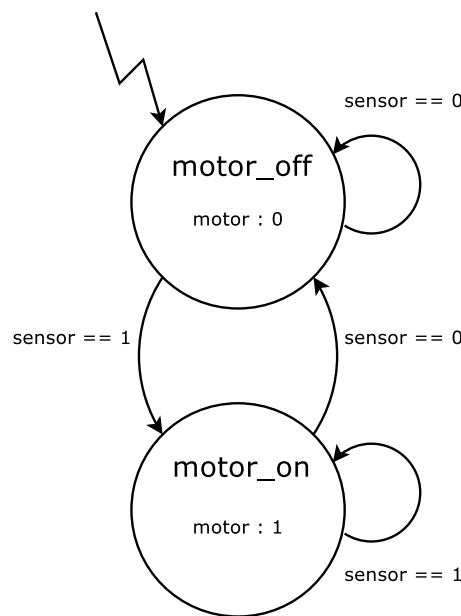
### 5.1.2 FSM in a state-diagram

An important tool in the design of a FSM, is the so-called state-diagram. Because of the shape it is also called the 'bolletjes'-diagram (spheres-diagram). The spheres represent the states in the diagram. Every sphere contains at least the following information:

- Name of the state
- The assigned values of *all* output signals
- One or more inward-pointing arrows that indicate under what conditions it selects this state
- One or more outward-pointing arrows that indicate under what conditions the state is left

In addition the FSM *always* contains a reference to the *reset-state*, the state in which the system is located after a reset. Such a reset state is indicated by a "lightning-arrow" to the relevant state. It goes without saying that there can only be one reset-state.

In the previously showed example, the state in which the motor is turned off is called `motor_off`. The state in which the motor is running, is called `motor_on`. The most logical state of the reset-state is `motor_off` (why?). The complete state diagram looks like this:



This state-diagram complies with the above requirements, as each state has a name. The names provide a clear description of the effect of the states. This makes it easier to comprehend how the FSM works, instead of when we used, for example, `state0` and `state1` as state names.

All the output signals are located within all the spheres, in this case only one.

For every sphere (and thus per state) there are two inward-pointing arrows. Next to these arrows the conditions (the values of the input signal *sensor*) are mentioned, and how to enter the state. One of the arrows comes from the state itself and indicates that the state is not left under certain conditions.

Every sphere has an outward-pointing arrow. This indicates that by a varying sensor-value, another state is selected.

### 5.1.3 The construction of a state-diagram

Now we know what a state-diagram looks like, the question remains how we design such a diagram (and thus also a FSM). To do this, the desired behavior is to be first described in a number of states. How that happens is basically the choice of the designer, however, by choosing a Moore FSM type the choice is established immediately. In a Moore machine the output value is only dependent on the current state, that means that at least every output combination needs a different state.

Make sure you assign descriptive names to the various states, so you can easily see the logical functions of a state. In case of the previously used FSM, the states are clear: in one state the motor is turned on, in the other the motor is turned off.

When the (basic) states are determined (including the reset-state), it must be established how the different states are interrelated: from which state is the other state reachable, and under what conditions. In case of the example FSM, this is also quite evident: When the motor is off (state `motor_off`) and the sensor displays a '1', the motor has to turn on and the next stage must become `motor_on`. The reverse reasoning is used to switch off the motor again.



If it seems impossible to make a good state diagram, check properly if you are making a Moore machine and not a Mealy machine! For example, if you have 4 output combinations and only 3 states, this is a Mealy machine (why?).

### 5.1.4 From state-diagram to SystemVerilog description

If you finished the state-diagram, you can convert this to a SystemVerilog description. This is very easy to do in a properly designed state-diagram. As mentioned previously, a FSM is constructed out of memory, implemented by means of flipflops, and a combinational part that determines the output signals and the next state. Determination of the output signals and the next state is usually described with a large `case` statement, which is selected at the current state. Within the choice of this state, the values of all output signals are described and a next state is also established. In order to make the description of a FSM easier, you can use the user-defined types. Such type can contain the names of the states. By making the *state* and *next\_state* signals of this type, the reference to the names of these states can be easily done in the `case`-statement.

```

1 module motor_controller (
2     input logic clk,
3     input logic reset,
4     input logic sensor,
5     output logic motor
6 );
7
8     // the user-defined motor_controller_state type
9     // contains the names of the states
10    typedef enum logic {motor_off, motor_on} motor_controller_state;
11
12    motor_controller_state state, next_state;
13
14    // In this process-statement, the state transitions take place:
15    always_ff @(posedge clk)
16        if (reset)
17            state <= motor_off; // reset state
18        else
19            state <= next_state;
20
21    // This always_comb statement implements the FSM's functionality,
22    // like described in the state-diagram.
23    always_comb
24    begin
25        case (state)
26            motor_off:
27                begin
28                    // Determine the outputsignals:
29                    motor = 0;
30
31                    // Determine the next state:
32                    if (sensor == 1)
33                        next_state = motor_on;
34                    else
35                        next_state = motor_off;
36                end
37            motor_on:
38                begin
39                    motor = 1;
40
41                    if (sensor == 0)
42                        next_state = motor_off;
43                    else
44                        next_state = motor_on;
45                end
46        endcase
47    end
48 endmodule

```



Since there is a clear and direct relation between a FSM and a state-diagram, there is a variety of software available that can convert one form to another, or visa versa. Xilinx enables the possibility to graphically design a FSM, Quartus from Altera can make a state-diagram from synthesized code.

### 5.1.5 Considerations when designing FSM's

Finally, a number of issues that are easily forgotten and cause problems with a sometimes difficult identifiable cause.

#### Take care that your FSM has a reset


Flipflops get, in principle, random values when booting hardware. This could mean that the FSM is in a different state than the reset state. If you then assume that the FSM is still in the reset state, this could lead to inexplicable behavior of the circuit. Therefore, make sure that your FSM has a reset input and a reset state and that you always reset the FSM at the start its operation, for at least one full clock period. In practice, you can connect the reset input of your FSMs to the system reset.

#### Specify every output value in every state

It is important to specify a value for *all* output signals in every state and for every *all* input values. Note that with this, the next state signal is also an output signal. If the above is not done, the FSM is supposed to remember the last set value, and a synthesizer will use a latch for this. Because the larger FSM's have multiple states from which another state can be accessed, the latch value can be quite different from what you would expect. Furthermore, a latch ensures that the timing tool of the synthesis-software no longer works. As an example, a synthesizer will generate a latch when you do not provide an `if`-statement with an `else` block! If latches are generated, you will get a warning about it during synthesis. So, read the warning carefully.

#### Default values

It is possible to give the outputs of the combinational part of an FSM a default value. In this way, it is not necessary to assign a value to one particular signal in every state. These default values are specified in the same `always`-block, but before the `case`-statement, in which the states are handled. For example:



```

1 always_comb
2 begin
3     -- default value
4     testvalue = 0;
5
6     -- state-handling
7     case (state)
8         ...

```

It is possible to assign a different value in another state to such a signal. The reason this is possible and allowed is simple: the statements in a `always`-block are processed sequentially. The signal gets the default-value first, and is then overwritten. This is not allowed outside a `always`-block, because assignments are concurrent.

Moreover, it is wise to use the default values only as an exception, it is much easier (and clear) to provide signals per state of the correct values.

## 5.2 Exercise: Design and implement the motor control

Through the motor control it should be possible to let the motor turn left or right, and to turn it on or off. As explained earlier, the servo-motors are controlled with a PWM-signal, this signal must meet several requirements:

- The frequency of the signal has to be 50 Hz
- The duty-cycle must be between 5 and 10%
- If the duration of the pulse is shorter than 1.5ms, the engine turns left

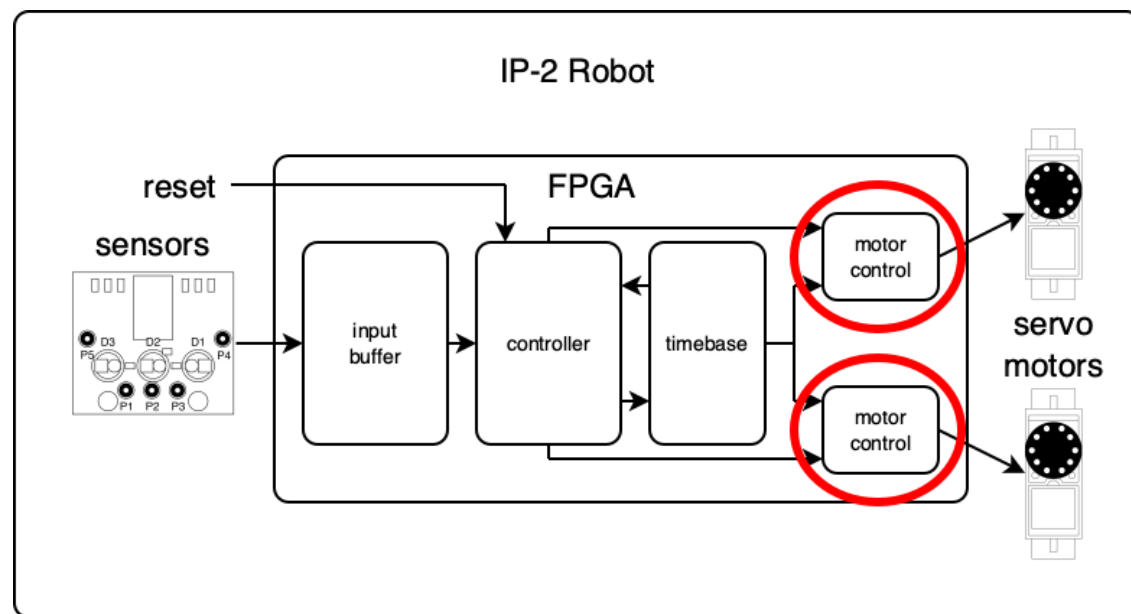


Figure 5.4: Implementing the motor control of the line-follower.

- If the duration of the pulse is longer than 1.5ms, the engine turns right

If the engine must stop, there should be no signal coming from the motor control, so no pulse. A pulse of 1.5ms is supposed to also let stand still the motor, but this more prone to errors because the motor may still rotate slowly a little bit.

The final signal must then have a shape as illustrated in Figure 5.5.

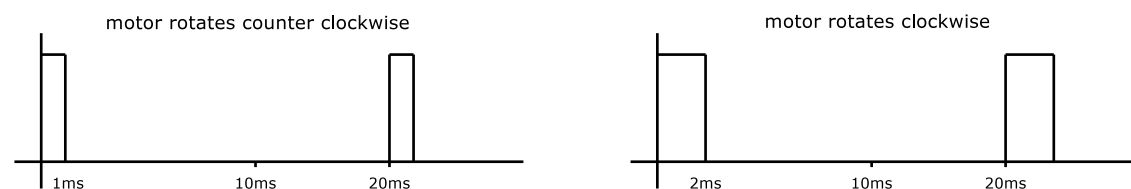


Figure 5.5: PWM-signal for a counter clockwise and clockwise turning motor

### Design a state-diagram for the motor control

The motor control has to generate a single PWM-pulse that complies with the specifications as given in Figure 5.5. The header of the motor control module should look something like this:

```

1 module motorcontrol
2   (input logic clk,
3    input logic reset,
4    input logic direction, // 0 == ccw, 1 == cw
5    input logic [?:0] count_in, // Value of the counter
6    output logic pwm); // The PWM signal

```

The signal *direction* indicates the direction (counter clockwise or clockwise). If the motor control is turned off, the signal *reset* must be kept high. The signal *count\_in* is originating from the counter in the time-base component and provides a reference for the time within a pulse of 20 ms. The idea is that the time-base component, as well as the motor control, receives a reset signal each 20 ms from the controller in the top-level module (the controller determines the

moment, based on the counter value in the time-base component). After that, a new period of the FSM is run through.

The motor control can be implemented as a state machine with 3 states.

One is the reset state. When the *reset* signal is high, the FSM will go to this state. The output signal *pwm* will be low in this state. When the *reset* signal is kept high for a longer period, the motor will consequently be off.

When the *reset* signal goes to a low, the FSM will go to a second state in which the output is high. It should stay in this state as long as is required to generate a pulse of 1 ms or a pulse of 2 ms. (which one is generated depends on the value of input *direction*). The value of input *count\_in* is used to decide when a time of 1 ms or 2 ms has passed.

When the value of input *count\_in* is such that 1 ms or 2 ms has passed, the FSM should go to a third state in which the output signal *pwm* is low. The FSM of motorcontrol should remain in the third state until the *reset* becomes high again, which is done after 20 ms by the controller that will be discussed in the next chapter.

Design a state-diagram of a motor control with the described module that meets these requirements. Since the output signal *pwm* is only dependent on the state, the state-diagram will describe a Moore machine. Think carefully about the conditions under which the state machine goes to a new state.

### Implement the state-diagram in SystemVerilog and simulate it

Convert the state-diagram into SystemVerilog code.



Comparing the signal *count\_in* with a value, can be done as follows: *count\_in* >= N' dV, where N is the number of bits for signal *count\_in* and V is the decimal value.

#### 5.2.1 Simulating the motor control

The motor control should now be simulated to test the correctness of the design. It is convenient to do this together with the timebase since the motor control uses input from the time base.

The following testbench is provided for your convenience to simulate the motor control in conjunction with the time base. It simulates a period of 20 ms in counter clockwise direction (*direction* = 0) and a period of 20 ms in clockwise direction (*direction* = 1). Note that the *reset* is high for one clock period at the start of each period to reset the timebase as well as the motor control. Also note that the upper boundary for signal *count* still has to be specified in the given testbench.

```

1  `timescale 1ns/1ps
2
3  module motorcontrol_tb ();
4
5      logic clk;
6      logic reset;
7      logic direction;
8      logic [?:0] count;
9      logic pwm;
10
11     timebase test1 (clk, reset, count);
12
13     motorcontrol test2 (clk, reset, direction, count, pwm);
14
15     always
16         #5 clk = ~clk; // period 10ns (100 MHz)
17     initial
18         clk = 0;
19
20     initial begin
21         reset = 1; direction = 0;

```

```

22     #10;          reset = 0;
23     #19999990; reset = 1; direction = 1;
24     #10;          reset = 0;
25     #19999990; reset = 1;
26     #10;          reset = 0;
27     end
28
29 endmodule

```

The simulation result should then look as shown as in Figure 5.6

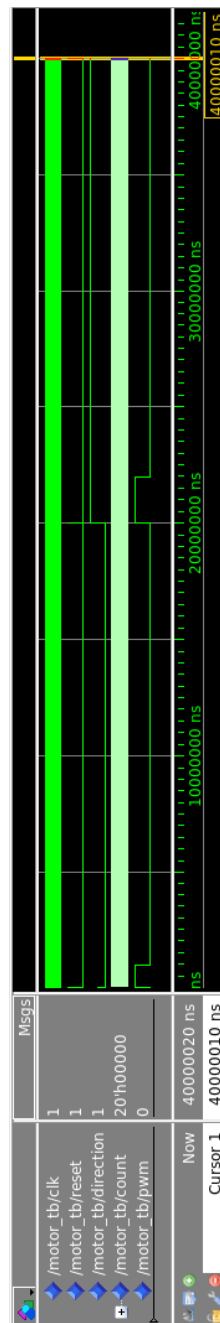


Figure 5.6: Required simulation result for the motor control in conjunction with the timebase.



## Chapter 6

# Design of an input buffer

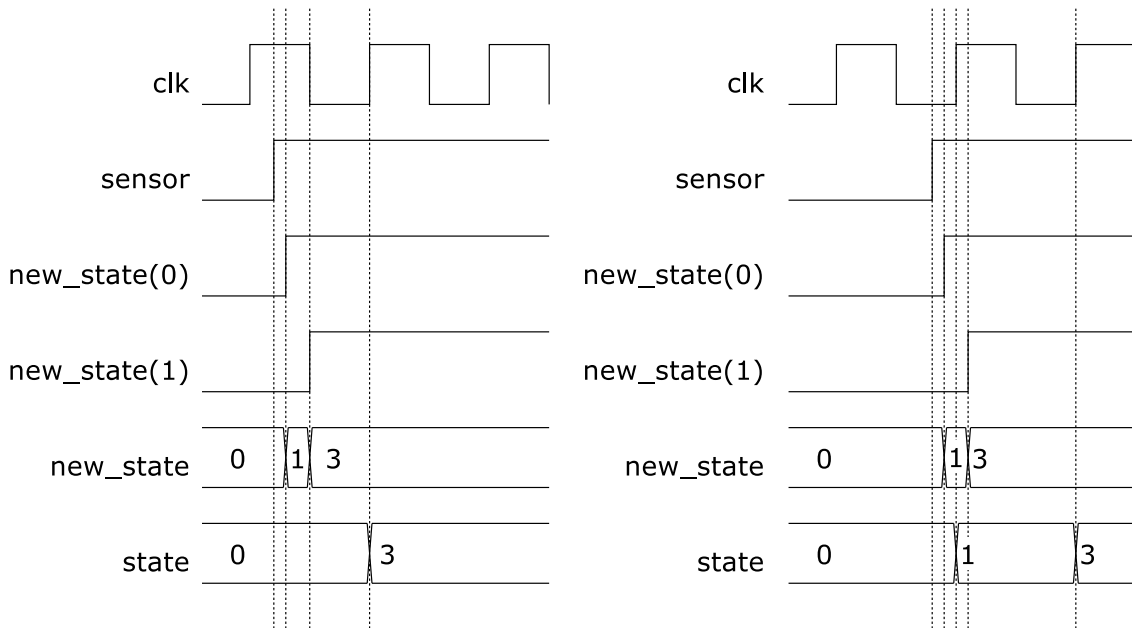
### Learning Objectives

During this session you will:

- learn why external input signals should be buffered
- implement the input buffer

### 6.1 Introduction

As described earlier, the new state of a counter is determined by a piece of combinational logic based on the current state and the current input signal values. The same will be true for a Finite-State Machine (FSM) that will be discussed in the next chapter. From the lecture digital systems it is known that the delay in a piece of combinational logic per bit may be different. A counter or FSM switches states during a rising clock edge, at which time the value coming from the combinational block is used as the new state. In case a bit is calculated *after* a rising edge cycle (because of the different delays), the state machine could fall into a different state than wanted. This is shown in the following Figure:

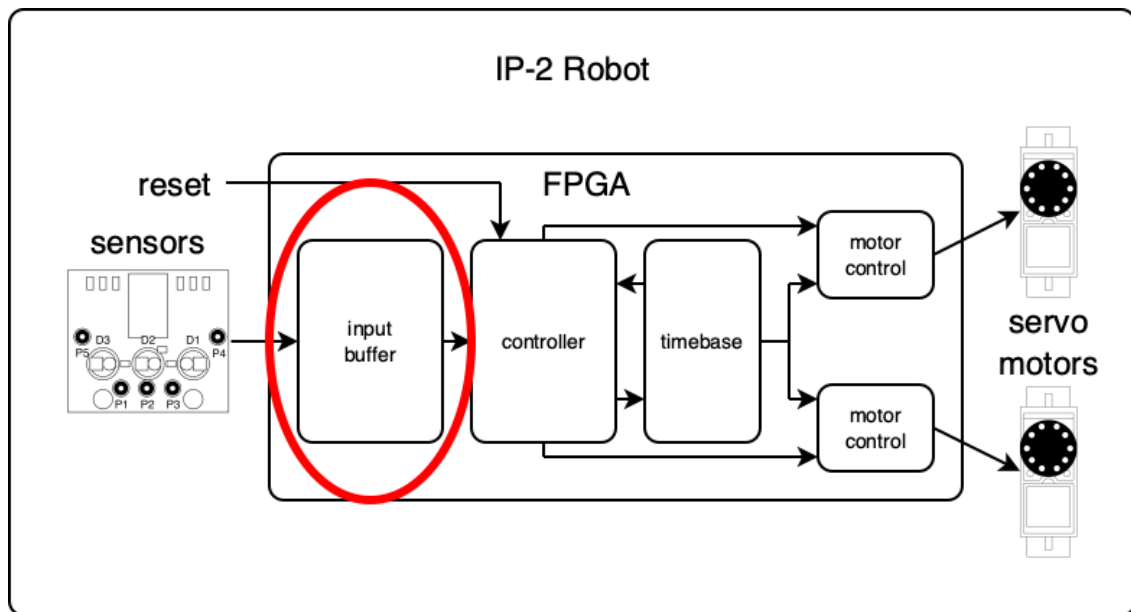


In this example, the signals *state* and *new\_state* are constructed out of two bits. The lose bits *new\_state*, *new\_state(0)* (LSB) and *new\_state(1)* (MSB) are also shown. The left side of the situation is correct, the sensor changes and *new\_state* has the correct new value for the next rising clock edge. In the right situation, however, the sensor-change occurs rather unfortunate, just before a rising clock edge. Due to delay differences, the *new\_state(1)* bit has not yet been calculated before the rising clock edge. As a result, the FSM results in the wrong state. In this case the situation is restored during the next rising clock edge, depending on how the new state is determined in state 1, but the FSM may also get into a wrong state from which it can not recover.

This problem can be avoided by ensuring that the input values are always synchronized with the FSM's clock. This can be achieved by offering the input signals via a flipflop (operating on the system clock) to the inputs of the system. In this way, the output of the flipflops changes synchronized with the system clock. Unfortunately, this is not sufficient to prevent all of the problems. In order to understand why this can still go wrong, we have to go deeper into the operation of a flipflop. A flipflop needs a certain amount of time to take over the input value, this is indicated by the *setup-time*. For proper functioning of the flipflop, the input signal has to be stable for this setup-time. If the input values of the flipflop changes during this time, the output value is undefined or *meta-stable*. The output value of the flipflop will continue to float in the middle of the voltage range (when in meta-stability). It may be clear that if this situation occurs, the input signals of the FSM are still not synchronized. The question is of course how you can avoid this problem. Unfortunately, the answer is that this can never be prevented a 100%. But it is possible to reduce the likelihood of such problems as much as possible. This can be done by placing a second flipflop after the first flipflop. If meta-stability occurs in the first flipflop, then the flipflop has an extra clock cycle to produce a stable output value before this problems occurs in the second flipflop. In reality, this appears to be a satisfactory solution. We will also implement such an input buffer in the robot.

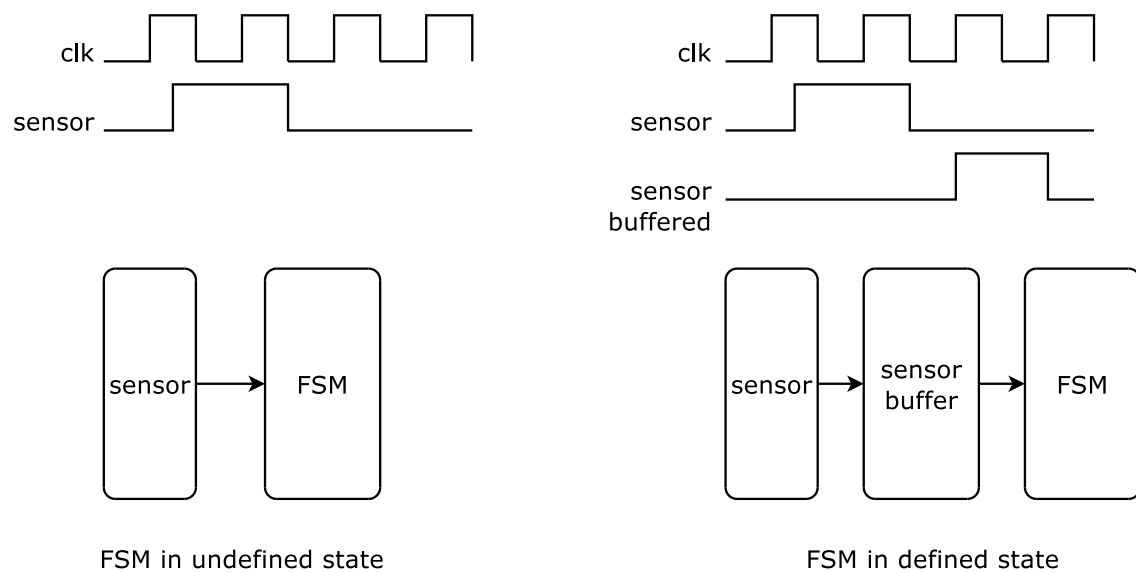
## 6.2 Exercise: Implement the input buffer

As discussed in the theory of the FSM, it is of great importance to make sure the input signals (which determine the state) are not changing during a clock-cycle. For signals originating from the outside, such as the sensor-values in this case, this cannot be guaranteed. In order to ensure the stability of these signals for the FSM, a buffer can be placed between the sensor and the



**Figure 6.1:** Implementing the buffer of the line-follower.

FSM. The buffer consists of two cascaded registers that operate on the same clock as the FSM. For an input buffer consisting of a single register, the effect is shown in Figure 6.2.



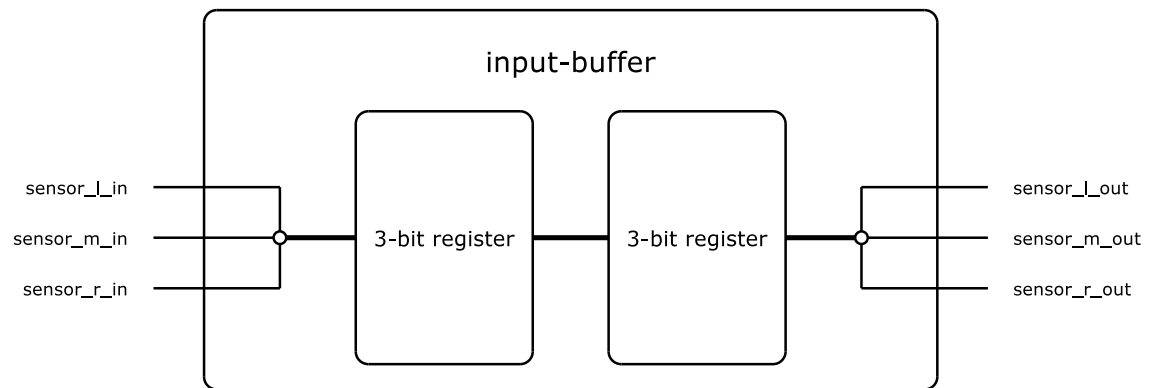
**Figure 6.2:** Schematic representation of the effect on whether or not to use an input buffer

Answer the following questions:

- The use of flipflops has a disadvantage for the reaction speed of the system, why is that?
- Is this a problem in case of the robot?

**Implement the input buffer using two 3-bit registers**

A template for the input buffer module is given by the file `inputbuffer.sv`. The input buffer should be composed of two 3-bit registers, in which each 3-bit registers is implemented with 3 flip-flops. A schematic representation is shown in Figure 6.3. One 3-bit register can be implemented with one `always_ff` statement acting on a 3 bit vector. The output is another 3-bit vector, that is input to another `always_ff` statement. Assign statements should then be used to connect the relevant vectors to the inputs and outputs of the module.



**Figure 6.3:** Schematic representation of the input buffer layout

## Chapter 7

# Designing a more complex FSM: the controller

### Learning Objectives

During this session you will:

- design the controller for the line-follower

### 7.1 Introduction

Different parts of the robot have been designed in the last few days: the input buffer, the time-base, and the motor control. The last item to be implemented is the controller. The controller is the “brain” of the robot, and largely determines the effectiveness of the robot. When this component has been implemented and tested, the different SystemVerilog modules can be merged and the line follower can be simulated as a whole.

This chapter will not be discussing new theory, a controller will be implemented as a FSM, wherefore the theory is already addressed in the previous chapters.

### 7.2 Exercise: Design and implement the controller

To implement the controller, a state-diagram has to be created first. The state-diagram can then be implemented in SystemVerilog and will implement the controller.

#### 7.2.1 Design a state-diagram for the controller of the line-follower

To make a controller for the line-follower, the idea is that at the beginning of each period of 20 ms, it is decided in which direction the robot will steer, based on the sensor values, and that during the rest of the period of 20 ms this movement is executed.

You should use Table 7.1 to implement the behavior of the motors as a function of the sensor values. Note that since the motors are mirrored on the robot, one wheel has to rotate clockwise and the other wheel has to rotate counter clockwise in order to drive forward.

For deciding the direction at the beginning of the 20 ms period, a separate central state should be used, which is also the reset state. For each direction (each output combination for the motor controls), also a separate state should be used. The controller will stay in such a state until it is detected from the counter value from the timebase that a period of 20 ms has ended. In that case, the controller should go back to the central state and a new direction is determined.

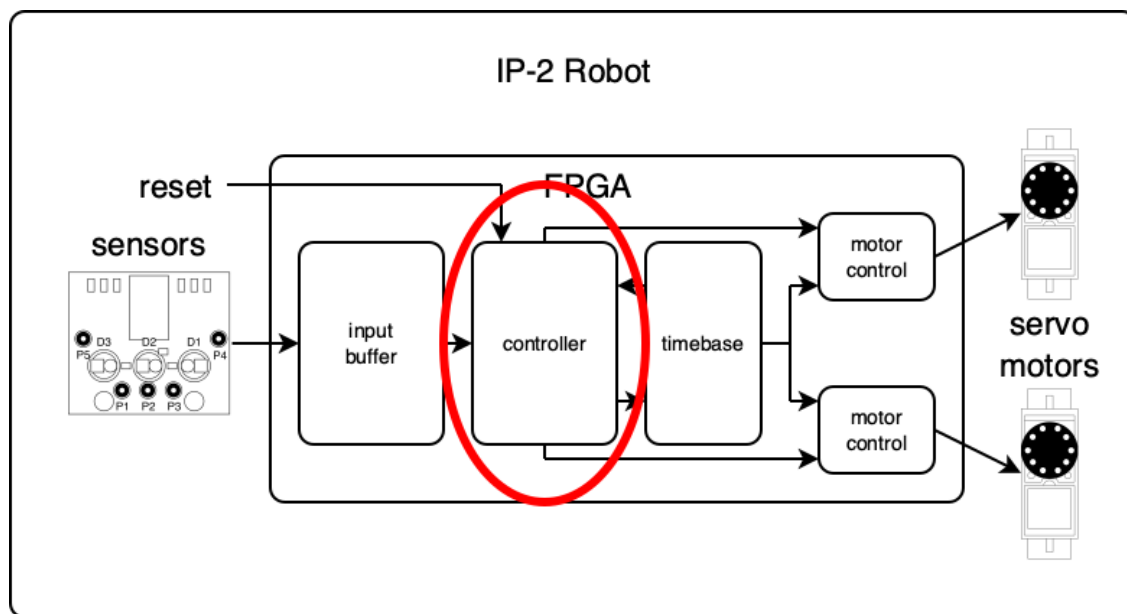


Figure 7.1: Implementing the controller of the line-follower.

Table 7.1: Motor control for different sensor inputs

Sensors			Wheels		Motor pulses		Direction
left	middle	right	left	right	left	right	
black	black	black	forward	forward	2 ms	1 ms	forward
black	black	white	stop	forward	0 ms	1 ms	gentle left
black	white	black	forward	forward	2 ms	1 ms	forward
black	white	white	reverse	forward	1 ms	1 ms	sharp left
white	black	black	forward	stop	2 ms	0 ms	gentle right
white	black	white	forward	forward	2 ms	1 ms	forward
white	white	black	forward	reverse	2 ms	2 ms	sharp right
white	white	white	forward	forward	2 ms	1 ms	forward

Note that the advantage of using a central state, to which all different direction states return after 20 ms, is that the (relatively complex) decision about the direction only has to be made in one state and not at the end of every different direction state.

Design a state-diagram for a controller that works according to the principle described above. Take care that the state-diagram implements a Moore machine by making the outputs direction and reset for both motor controls, and the output reset for the timebase only dependent on the current state. Also, take care that, for the combinational part of the FSM, all outputs (including the next state signal) receive a value for all input combinations and for all states, to prevent the creation of latches, later on, during synthesis.

### 7.2.2 Implement and simulate the state-diagram in SystemVerilog

Use the template file `controller.sv` to implement the finite-state diagram that is described above in SystemVerilog.

## Chapter 8

# Putting everything together

### Learning Objectives

During this session you will:

- Create a top-level module for the line-follower, in which you connect all modules together using a structural architecture description.
- Simulate the top-level module of the line-follower.

### 8.1 Introduction

All modules of the line-follower should now be added together and the top-level module should be created, see Figure 8.1.

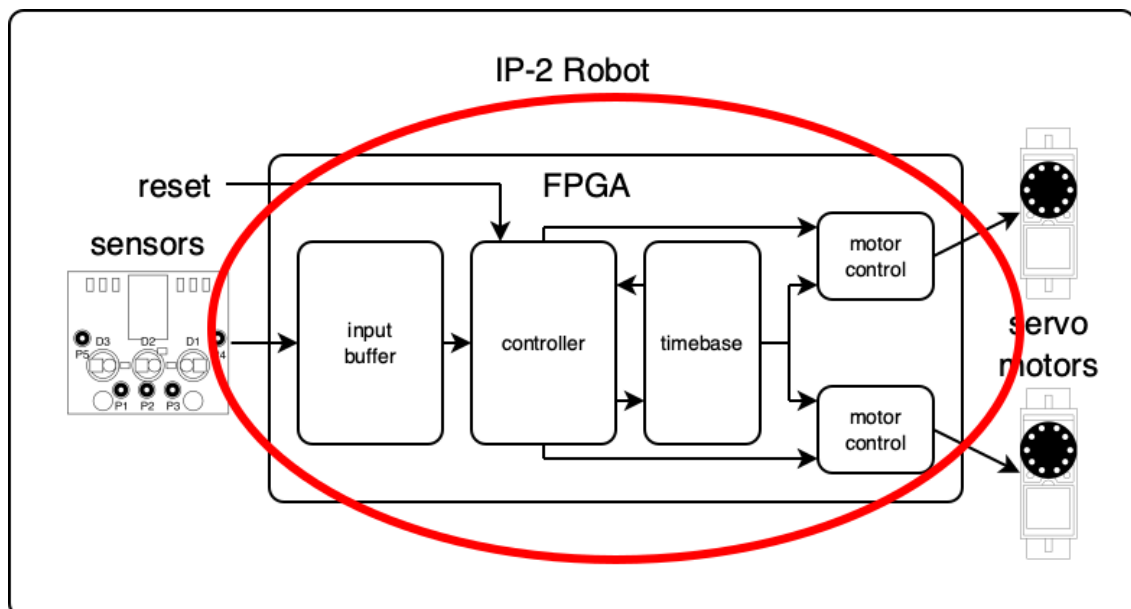


Figure 8.1: Creating the top-level schematic of the line-follower

## 8.2 Exercise: Assemble and simulate the robot

### 8.2.1 The top-level module

The module description gives the name of the system and the various inputs and outputs. *In order to allow automatic verification of the line-follower, we require you to implement the interface of the top-level module as given by the template in the file robot.sv.* This interface is repeated here below.

```

1 module robot
2   (input logic clk,
3    input logic reset,
4
5    input logic sensor_l_in,
6    input logic sensor_m_in,
7    input logic sensor_r_in,
8
9    output logic motor_l_pwm,
10   output logic motor_r_pwm);
11 );

```

The rest of the module should describe how the system is implemented. Here we make a description by means of various modules which are connected to each other according to as shown in Figure 8.1. This is a structural description, containing the following items:

**The signal declarations** These are the declarations for the intermediate signals that are used to make connections between the different modules, e.g. between the timebase and motor control.

**The module instantiations** These are instances of the modules that are part of the robot. For the module motorcontrol, two instances should be used. The signals that were declared above, are used to connect the ports of these components.

### 8.2.2 Simulation of the line-follower

You can verify the behavior of the line-follower by simulating it with the following testbench.

```

1 `timescale 1ns/1ps
2
3 module robot_tb();
4
5   logic clk;
6   logic reset;
7   logic sensor_l;
8   logic sensor_m;
9   logic sensor_r;
10  logic [2:0] sensors;
11  logic motor_l_pwm, motor_r_pwm;
12
13  robot test (clk, reset, sensor_l, sensor_m, sensor_r, motor_l_pwm, motor_r_pwm);
14
15  assign {sensor_l, sensor_m, sensor_r} = sensors;
16
17  always
18    #5ns clk = ~clk; // period 10ns (100 MHz)
19  initial
20    clk = 0;
21
22  initial begin
23    #0ms; reset = 1;
24    #40ms; reset = 0;
25  end
26
27  initial begin
28    #0ms; sensors = 3'b000;
29    #70ms; sensors = 3'b001;
30    #40ms; sensors = 3'b010;
31    #40ms; sensors = 3'b011;
32    #40ms; sensors = 3'b100;
33    #40ms; sensors = 3'b101;
34    #40ms; sensors = 3'b110;
35    #40ms; sensors = 3'b111;
36  end

```



```
37  
38 endmodule
```

The simulation result should then look as shown as in Figure 8.2

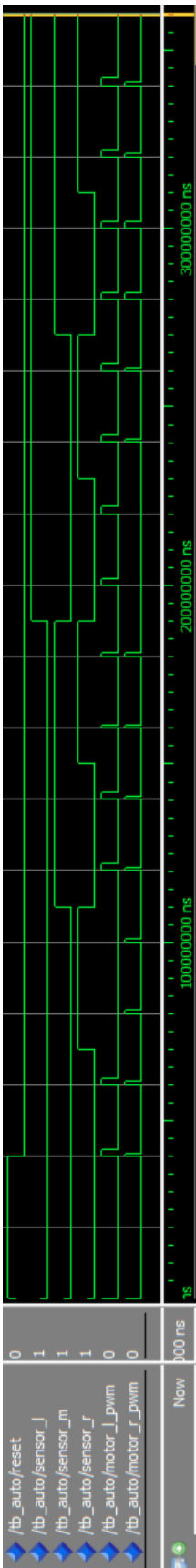


Figure 8.2: Required simulation result for the line-follower robot

## Chapter 9

# Test the Robot

### Learning Objectives

During this session you will:

- program the SystemVerilog code onto the FPGA of the robot and test the robot.

### 9.1 Go through the Vivado tutorial

You are now ready to test the design on the robot. Therefore, first follow the Vivado tutorial in Appendix C to learn how to program the FPGA on the robot. Important: note that Vivado 2023 is currently only installed on the PCs in the Tellegen hall that have a TUD machine number ending on even digit.

### 9.2 Program the FPGA of the robot with your SystemVerilog code

Create a Vivado project, import your SystemVerilog code, assign pins to the different input- and output signals of the system as show in Table 9.1, and synthesize the system.



#### Pin-assignments

It is important that you assign *all* in- and output signals to the pins. If you don't do that, the signals will be connected to randomly chosen pins. This may have various undesirable side effects and even, in worst case, cause damage to the FPGA board.

**Table 9.1:** Pin assignments for the line-follower. The reset will be connected to switch 15.

port	pin
clk	W <sub>5</sub>
reset	R <sub>2</sub>
sensor_l_in	M <sub>2</sub>
sensor_m_in	L <sub>3</sub>
sensor_r_in	J <sub>3</sub>
motor_l_pwm	L <sub>2</sub>
motor_r_pwm	J <sub>1</sub>

If there are no error messages during synthesis and no important warnings like latches being created, you can program the FPGA. In this case, it is best to power the robot, including the FPGA, via the battery pack and use JTAG programming via USB, which only uses the volatile FPGA SRAM memory. After you have programmed the FPGA via USB, you can then unplug the USB cable and let the robot operate using the battery pack. To power the robot and FPGA via the battery pack, set the POWER jumper (highlighted in Figure C.1) to EXT, see also Figure 9.1. To use JTAG programming, set the programming mode jumper to JTAG Programming, see Figure 9.2.

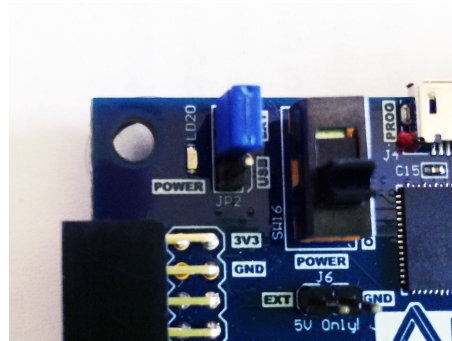


Figure 9.1: Setting the POWER jumper to EXT (battery pack)

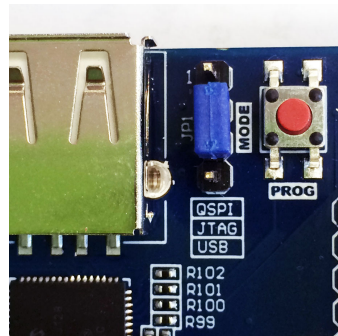


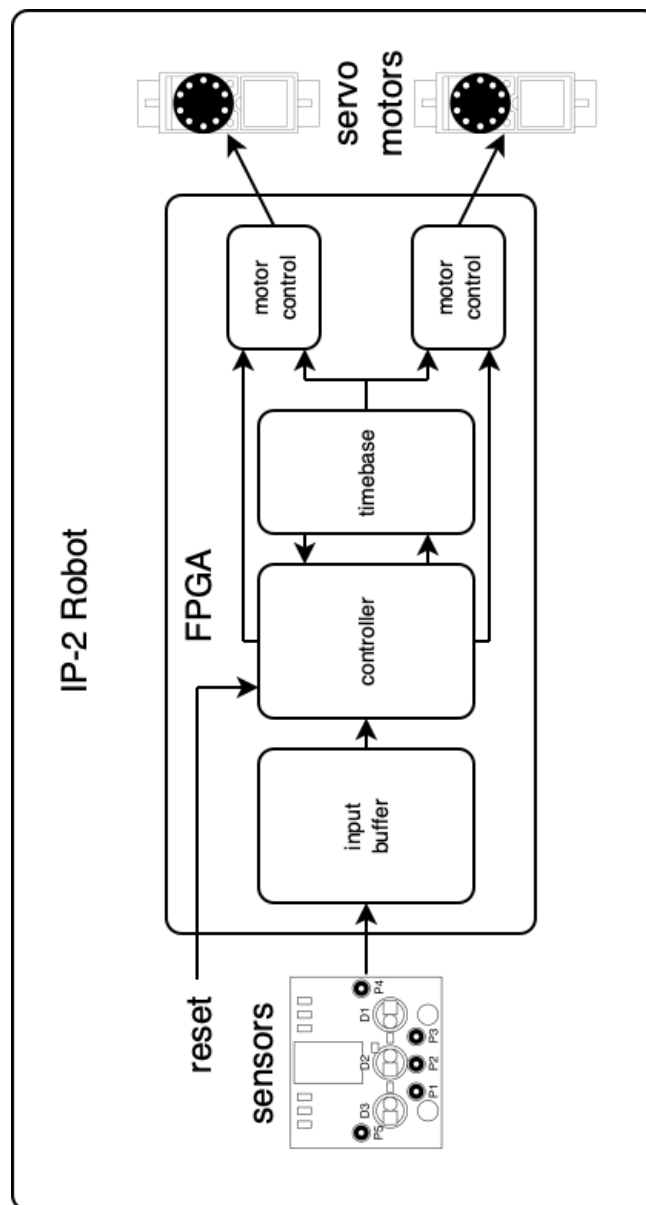
Figure 9.2: Setting the programming jumper to JTAG programming via USB

### 9.3 Test the robot

Use the test patterns in Appendix B and the black tape to test the robot. The robot must be able to follow the test-lines 1 – 4 successfully.

## Appendix A

### Overview of the robot

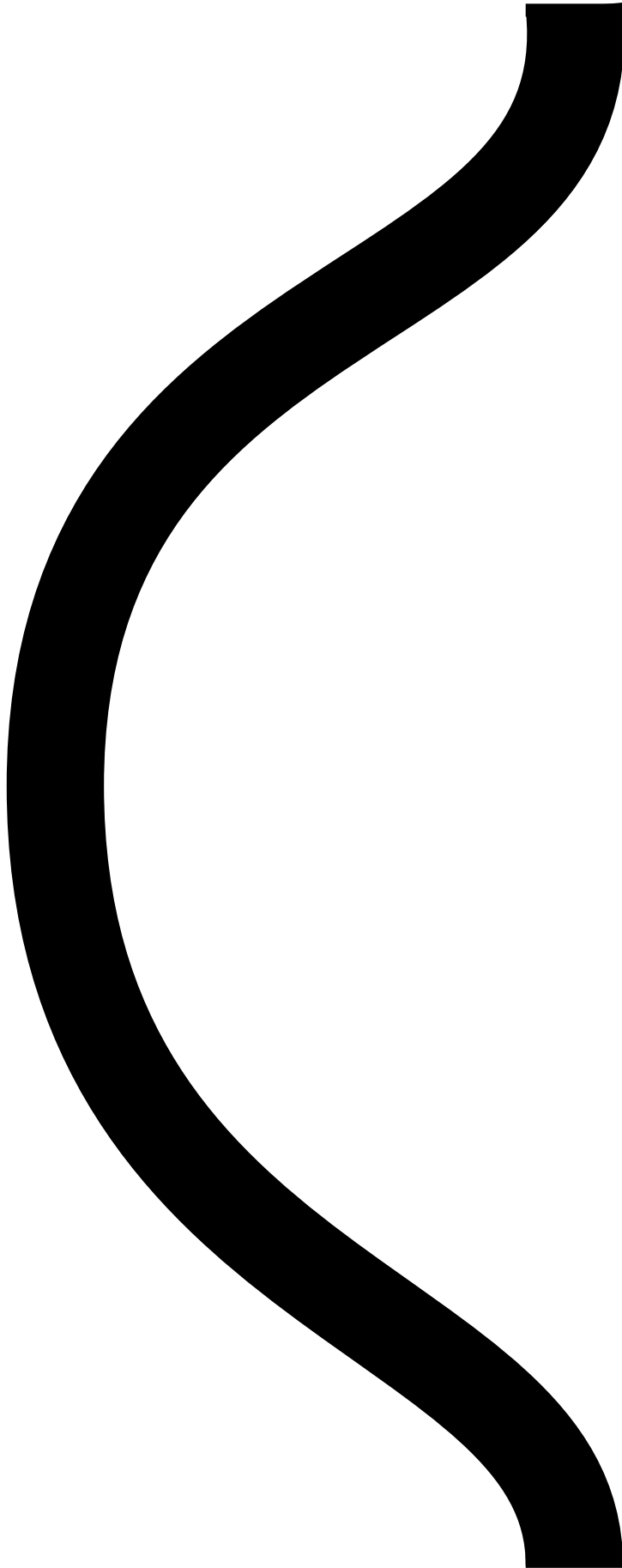


## Appendix B

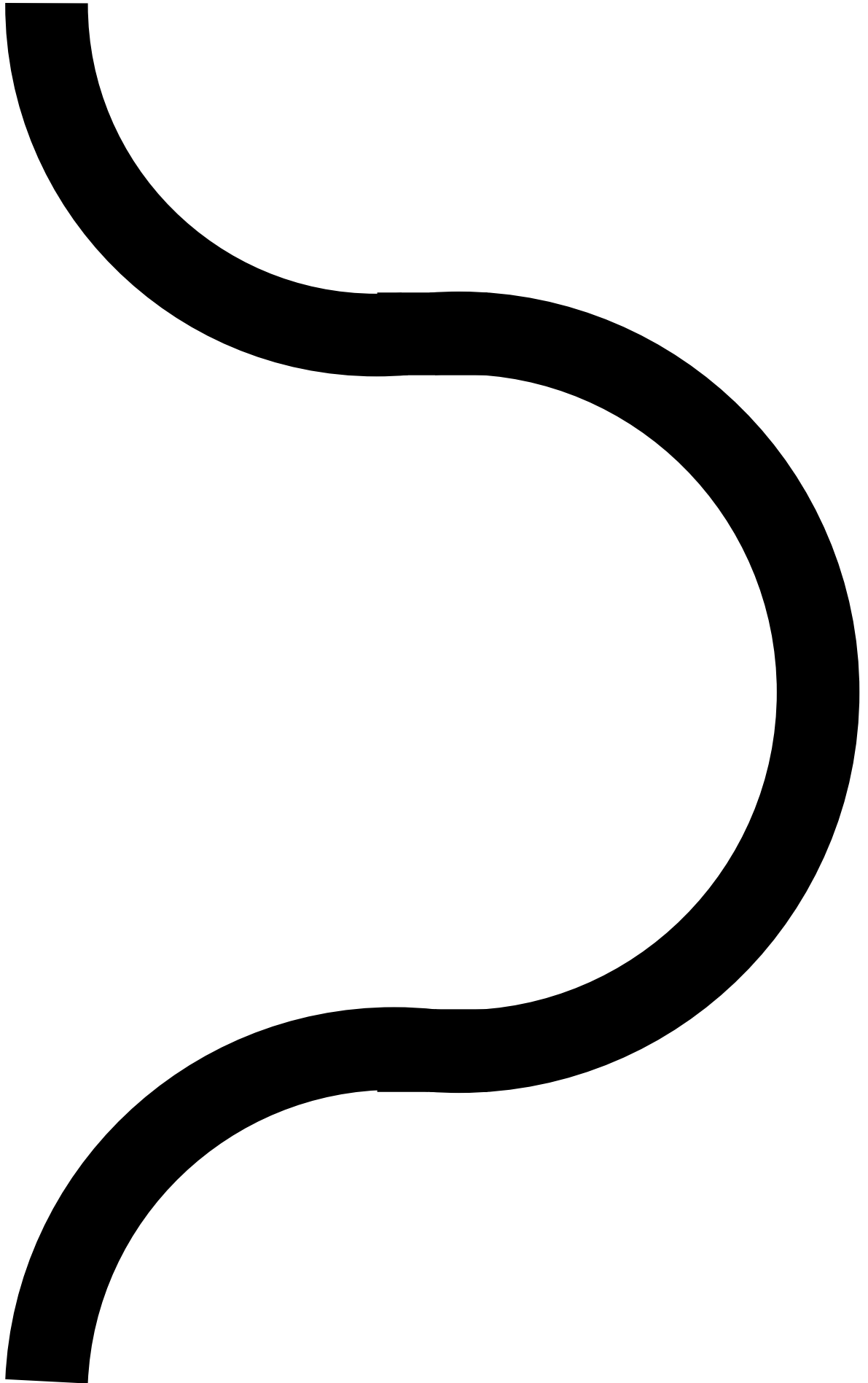
### Test lines



Line 1: straight line

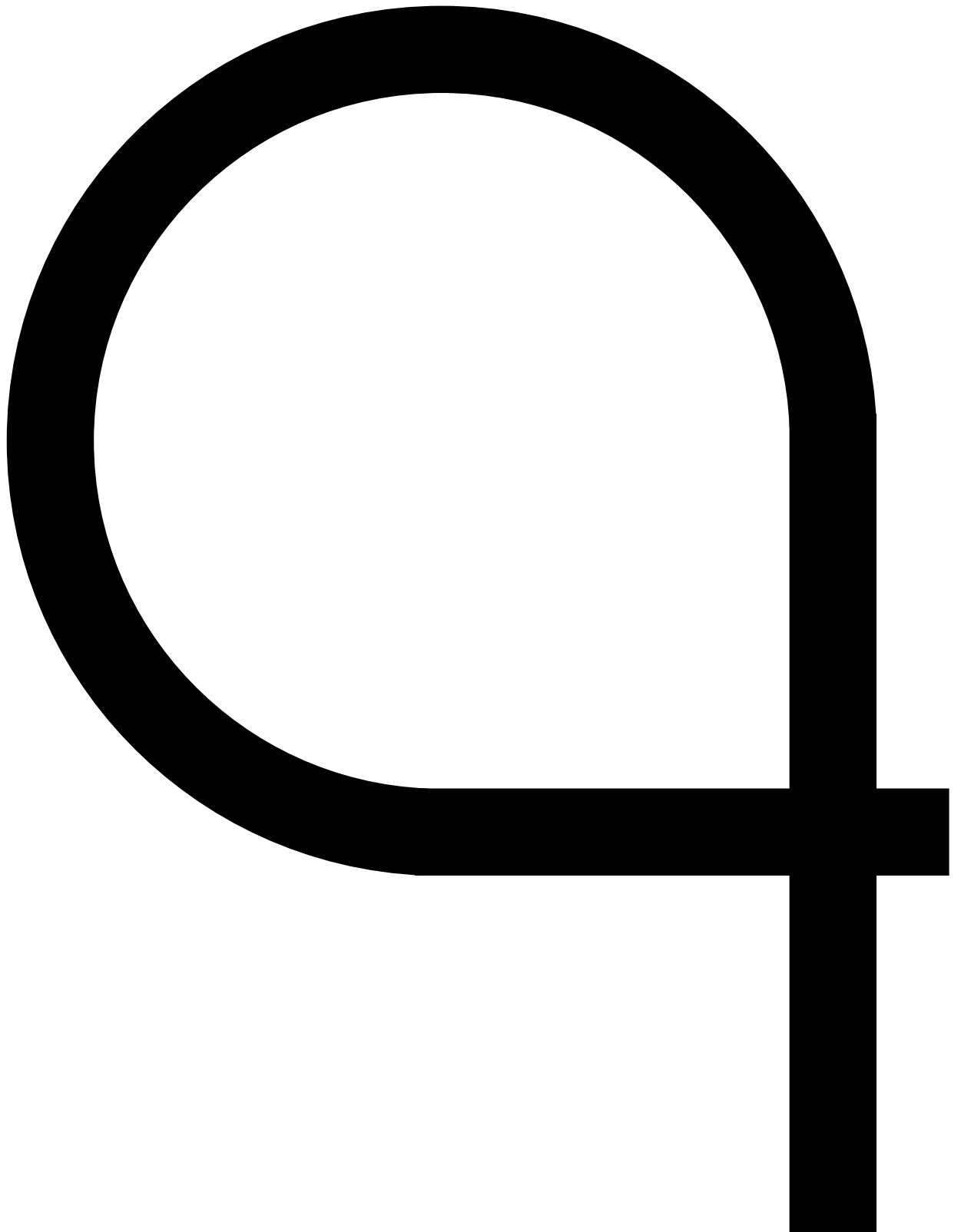


Line 2: slight bend



Line 3: sharp turn





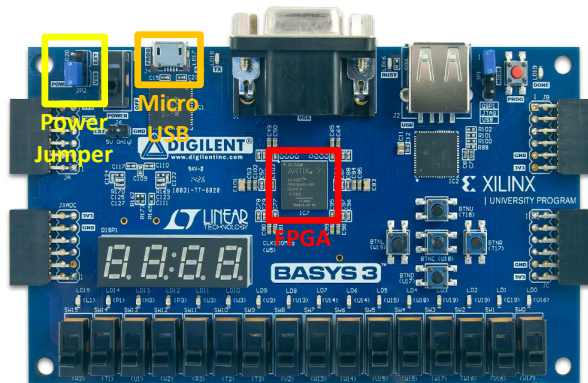
Line 4: 90° turn

# Appendix C

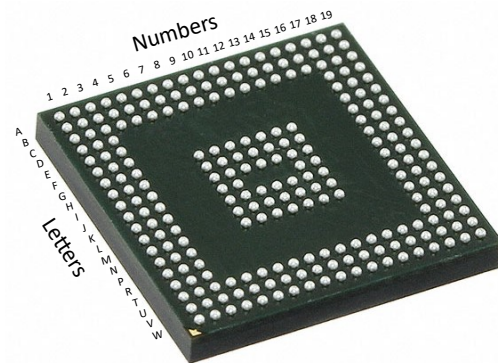
## Vivado 2023.2 Tutorial

### C.1 Introduction

In this tutorial you will make a circuit that makes an LED blink using an FPGA. Instead of soldering the circuit ourselves, we use a so-called “FPGA development board” (see Fig. C.1), specifically the Basys3 board. The FPGA, highlighted in orange, comes in a Ball Grid Array (BGA) package (see Fig. C.2). On a BGA package the input and output pins are metal balls arranged in a 2D array. Rows are labeled with letters and columns with numbers. E.g., pin D7 refers to the pin in the 4th column and the 7th row. Which pin is connected to which component on the board can be found in the Basys3 reference manual [1].



**Figure C.1:** Top view of the Basys3 development board. (adapted from [1])



**Figure C.2:** Underside of the ARTIX-7 FPGA that is on the Basys3 board. (adapted from [2])

The blinking LED circuit that you will make in this tutorial is shown in Fig. C.3. Normally, the LED blinks at a frequency of 1 Hz with a 50% duty cycle.

- The Basys3 board has a 100 MHz clock generation circuit that is connected to pin W5.
- Switch 0 / SW0 (connected to pin V17) resets the circuit when high.
- Switch 1 / SW1 (connected to pin V16) halves the duty cycle of the LED pulse to 25% when high.
- LED 0 (connected to pin U16) is used as the output.

The FPGA needs to be configured to behave like a pulse generator for the LED. To do this, the manufacturer of the FPGA (Xilinx) created a software tool called Vivado. The user can

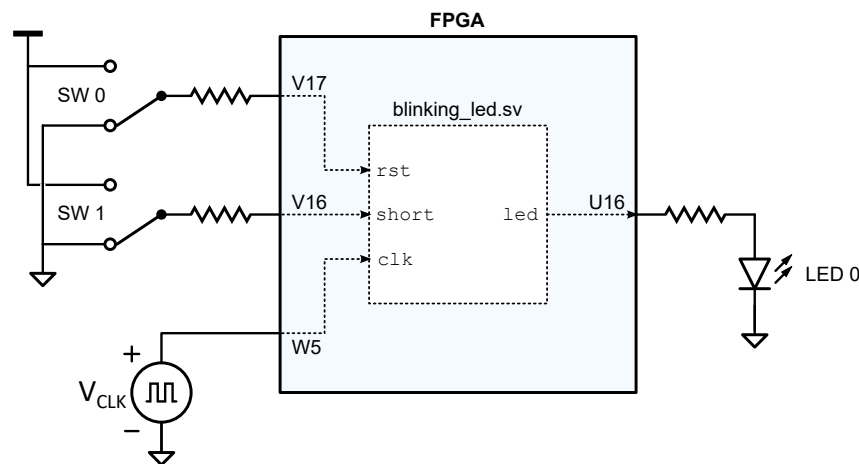


Figure C.3: Diagram of the blinking LED circuit.

supply Vivado with a circuit description in HDLs like SystemVerilog and VHDL, and Vivado will generate an FPGA configuration file to program the FPGA to behave like the supplied circuit description.

Before you can configure the FPGA, Vivado needs to go through a few steps:

*Synthesis:* The HDL description of the circuit is interpreted and converted into a hardware model.

*Implementation:* The synthesized model is converted into a circuit using the blocks and wires on the FPGA.

*Bitstream generation:* A file is generated which, if programmed on the FPGA, will connect the logic blocks on the FPGA according to the circuit from the implementation step.

The step-by-step tutorial in the next section will guide you through these steps.



This tutorial only covers the basics. Vivado has thousands of pages worth of manuals, and this tutorial only scratches the surface. During your project you will inevitably encounter situations that are not covered in this tutorial. When you run into problems, knowing what you're doing makes it much easier to look up information online or in manuals. As such, try to understand as much as possible of what you're doing, and ask your TA if you don't understand something.

## C.2 Step-by-Step Tutorial

### Step 1: Before we begin

1. Create a folder on your computer for this tutorial called “tutorial\_vivado”.
2. Download the following files from Brightspace and move them into the folder “tutorial\_vivado”:
  - *blinking\_led.sv*: The SystemVerilog circuit description of the LED pulse generation circuit.
  - *blinking\_led\_tb.sv*: Its corresponding testbench.
3. Read the code of “blinking\_led.sv” to see how it works. Refer to section C.1 for a description of the inputs and outputs.
4. (optionally) Simulate the circuit using the testbench in QuestaSim for 2 seconds. Note that a simulation of such length could take a few minutes to complete.



The role of simulation in the FPGA workflow Finding bugs in a circuit on an FPGA is generally very difficult and time consuming. For this reason it is common practice to verify if the circuit behaves as intended by simulating its HDL description first. While Vivado has an inbuilt simulation tool, our main simulator during the project will be QuestaSim.

### Step 2: Create a Vivado Project

1. Open Vivado 2023.2. You should be greeted with the screen in Fig. C.4.
2. Click on *Create Project* >. A wizard will open. Press *Next* > to go to the first step.

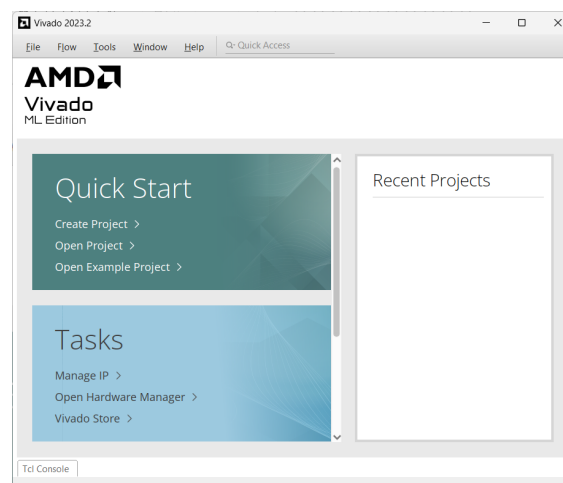
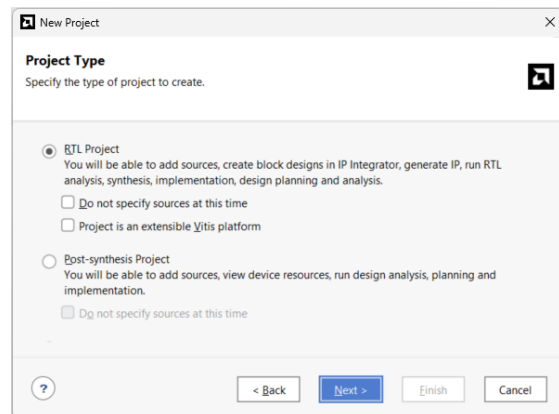
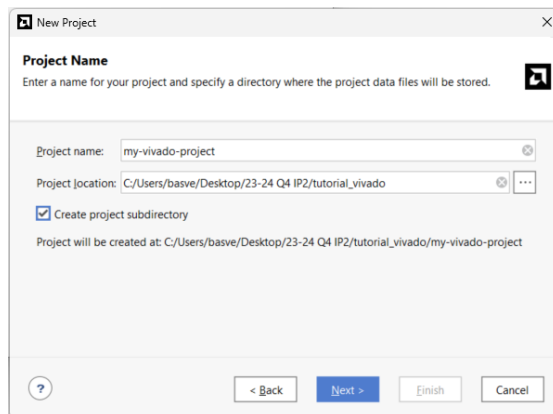


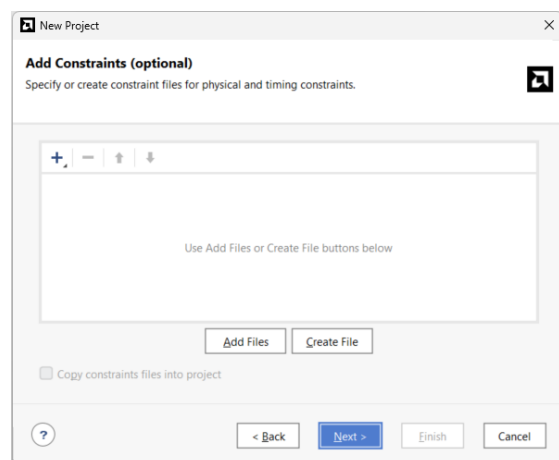
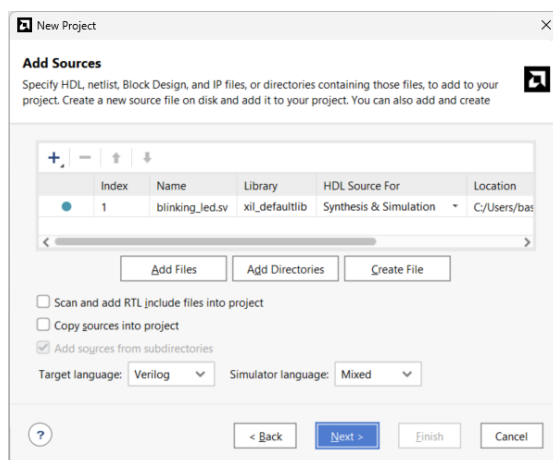
Figure C.4: Vivado welcome screen



3. Name your project “my-vivado-project”.
4. Select “tutorial\_vivado” as the project location.
5. Check *Create project subdirectory* to let Vivado create a folder called “my-vivado-project” in “tutorial\_vivado”. All files Vivado will generate will be saved in the subfolder “my-vivado-project”.
6. Press *Next >*.

7. Select *RTL Project* as the project type. RTL stands for “Register Transfer Level”: A commonly used abstraction of synchronous digital circuits.

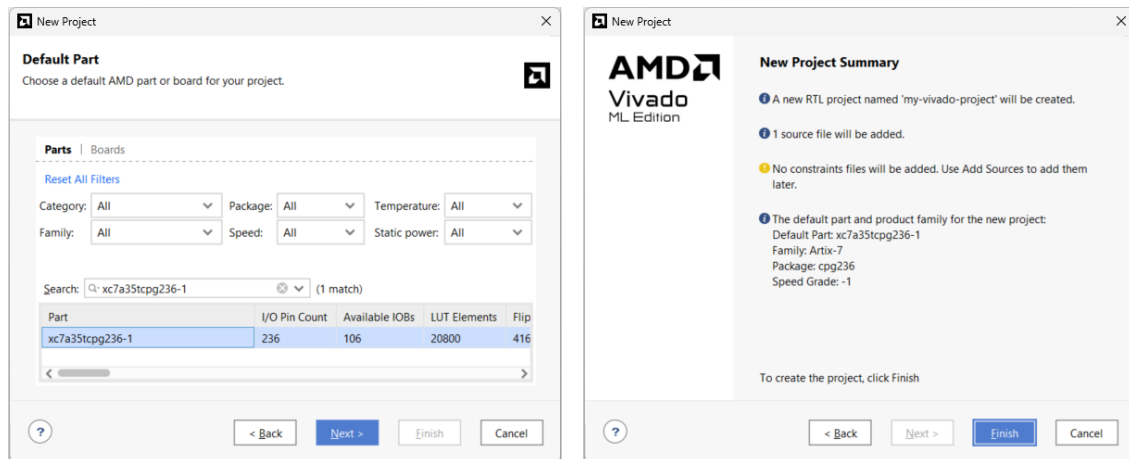
8. Press *Next >*.



9. Click the + sign to add the file “blinking\_led.sv” to the Vivado project.
10. Press *Next >*.

11. Skip this step for now. We will come back to this later.

12. Press *Next >*.

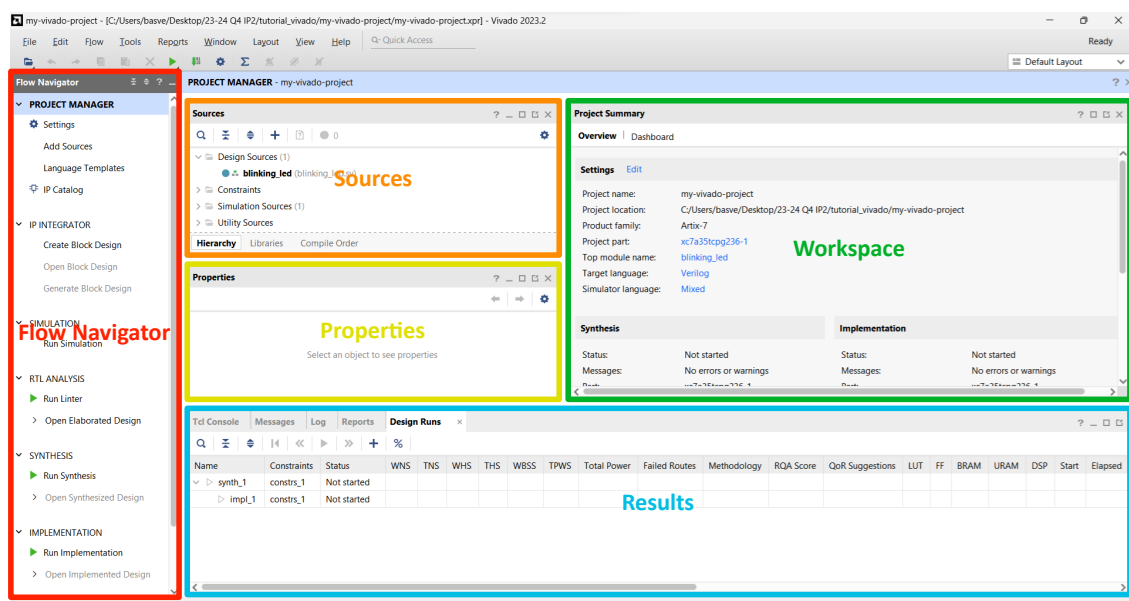


13. Select the FPGA part number of the FPGA on the Basys3 board in the catalog. This part number is xc7a35tcpg236-1, and can be found in the Basys3 reference manual [1]. It is also partially written on the FPGA chip itself.
14. Press *Next >*.
15. Press *Finish* to close the wizard and create the project.

### Overview of Vivado

After closing the wizard, the window shown in Fig. C.5 should appear. This window consists of a number of sub-windows, commonly called “panes” (Dutch: “ruit”).

Of these panes, the *Flow Navigator* (flow as in workflow) is the most important. The *Flow Navigator* is the pane from which you open different menus. For this tutorial the relevant menus are *Project Management*, *Synthesis*, *Implementation* and *Program and Debug*.



**Figure C.5:** Main window of Vivado. All items and panes related to “Project Manager” in the Flow Navigator pane are inside the red selection.

For every *Flow Navigator* menu, the panes on the right might look slightly different. However, they are generally as follows:

**Sources** lists all user-created files that Vivado needs to do things. Vivado sees them as the “source” of the files it creates.

**Properties** shows the properties of the file that is selected in the *Sources* pane. Click on a file to select it.

**Workspace** shows information that is relevant to the menu selected in the *Flow Navigator*.

**Results** has a number of tabs with tools that help with debugging:

**Tcl Console** is a terminal that can be used to run interface with Vivado without using the GUI.

**Messages** is where Vivado shows any errors or warnings it encounters. If, for example, there is something wrong with your code, it will show up here.

**Log** tracks every important task that Vivado performs while it is running a routine.

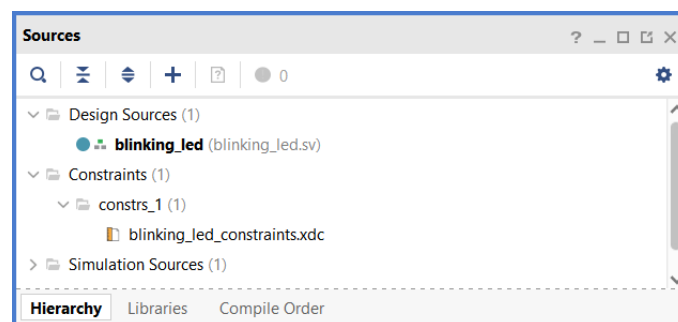
**Reports** is a menu from which you can open summaries with information about the execution of a routine.

**Design Runs** shows the status and properties of a routine.

### Step 3: Add Constraints

During synthesis Vivado will interpret your HDL source files and generate a circuit model from it. With the files that Vivado currently has, it cannot generate a complete model. It still misses knowledge of which port of the SystemVerilog module in “blinking\_led.sv” should be connected to which FPGA pin, and how each pin should be configured. We do this by creating a Xilinx Design Constraint (XDC) file.

1. Click the “+” icon in the *Sources* pane. A wizard should pop up. Select *Add or create constraints* and click *Next* >.
2. In this next section of the wizard, click “Create File”. A pop-up should appear. Name your constraint file “blinking\_led\_constraints” and press OK.
3. Finally, press *Finish* to close the wizard. Your *Sources* pane should now look like Fig. C.6.



**Figure C.6:** The “Sources” pane with the newly added constraint file (Flow Navigator > Project Manager).

4. The following XDC code associates the port called `clk` in “blinking\_led.sv” with FPGA pin W5, and configures it to use Low Voltage CMOS (LVCMOS) circuitry where binary 1’s are represented by 3.3 V.

```
set_property PACKAGE_PIN W5 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
```

The following XDC code lets Vivado know that the port `clk` is a clock. When Vivado knows this, it will allow the dedicated clock circuitry on the FPGA.

```
create_clock -period 10.0 -name sys_clk_pin -waveform {0.0 5.0} [get_ports clk]
```

5. Next, add modified copies of the 2 `set_property` lines above to connect port `rst` to pin V17, port `short` to pin V16 and port `led` to pin U16 as shown in Fig. C.3.
6. The following XDC code sets two parameters that Vivado wants to know to configure the FPGA. Technically, you could omit these lines, but Vivado will show a warning in the *Messages* tab in step 5 if you do.

```
set_property CONFIG_VOLTAGE 3.3 [current_design]
set_property CFGBVS VCC0 [current_design]
```

7. Save your changes to the file.

#### Step 4: Synthesis

In this step we're going to let Vivado synthesize a circuit model from the provided HDL code and constraints file.

1. In the Flow Navigator, click "Run Synthesis." A window may pop up which you can simply close by clicking OK.
2. While Vivado is running the synthesis, go to the *Design Runs* tab in the bottom right. In this tab you can see when the synthesis has finished.
3. Once the synthesis is completed a confirmation pop-up appears saying "Synthesis successfully completed." On the pop-up, select *Open Synthesized Design* and press OK. Alternatively, you can close the pop-up, and click *Open Synthesized Design* in the Flow Navigator.

Once Vivado has finished loading you can see from the Flow Navigator that *Synthesis* is now selected instead of *Project Manager*.

4. To view the synthesized model, click on *Schematic* under *Open Synthesized Design* in the Flow Navigator. The schematic shown in Fig. C.7 should appear. Feel free to explore the schematic by zooming in (ctrl + scroll) on it and by double clicking the blocks.

#### Step 5: Implementation

In this step we're going to let Vivado create a circuit using the logic blocks on the FPGA.

1. Click on *Run Implementation* in the Flow Navigator. If a pop-up appears right after pressing *Run Implementation*, click OK.
2. By default a tab called *Device* opens with a schematic of the internal circuitry of the FPGA (See Fig. C.8). Feel free to explore the schematic by zooming in (ctrl + scroll) on it and by clicking on blocks.



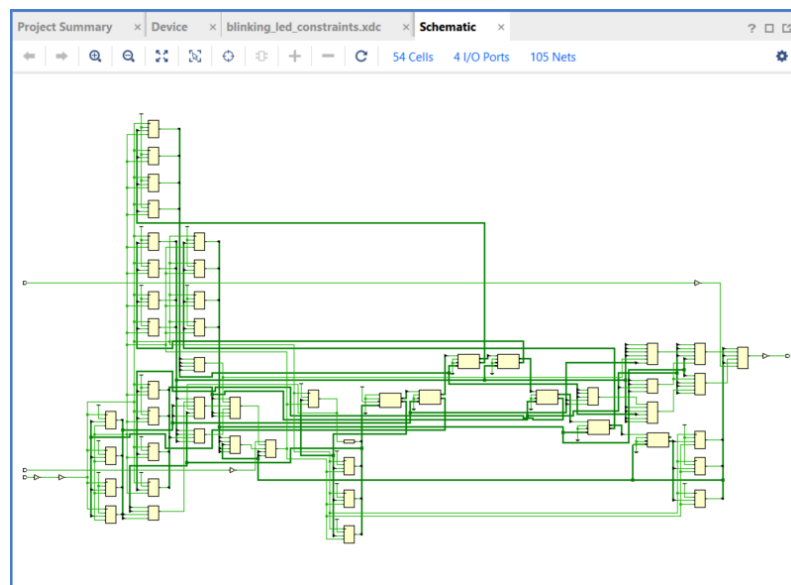
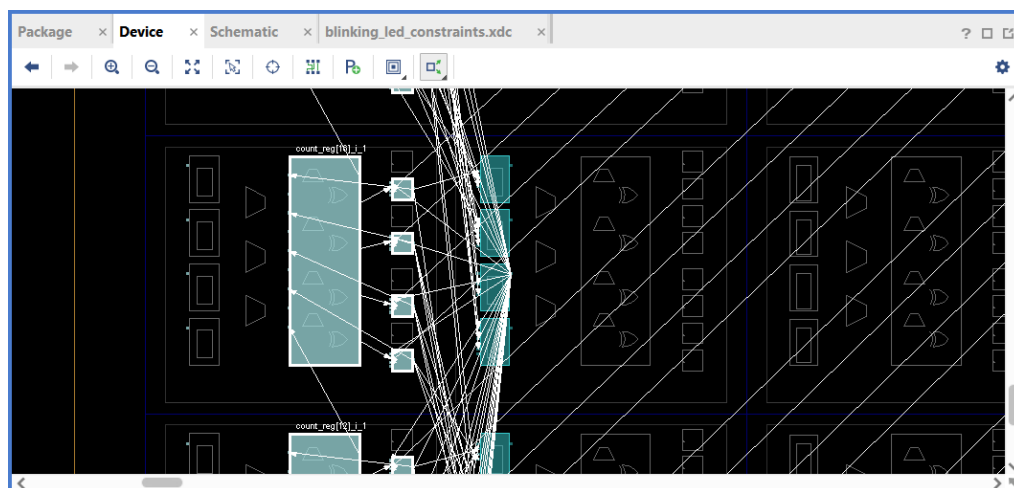


Figure C.7: Schematic pane in Vivado

Figure C.8: Device pane in Vivado after enabling *Show Cell Connections* (rightmost button in the *Device* pane) and clicking a logic block that is used in the circuit.

### Step 6: Programming

In this step we're going to let Vivado program the FPGA on the Basys3 board.

1. Get a robot.
2. Connect the Basys3 board to your PC with an USB-A to micro-USB cable. Connect to the highlighted USB port in Fig. C.1.
3. Set the *POWER* jumper highlighted in Fig. C.1) to *USB*. Now the board gets its power through the USB cable.
4. Switch the board on using the switch to the right of the *POWER* jumper.
5. In the Flow Navigator in Vivado under *Program and Debug*, click *Generate Bitstream*. Answer all pop-ups that may appear with *OK*.

6. After Vivado completes the bitstream generation, click *Open Hardware Manager* under *Program and Debug* in the Flow Navigator.
7. Click *Open target* in the green bar on top of the workspace pane (See Fig. C.9) and choose *Auto Connect*. This will establish a connection between Vivado and the FPGA on the Basys3 board.
8. Click *Program device* in the green bar to configure the FPGA (See Fig. C.9).

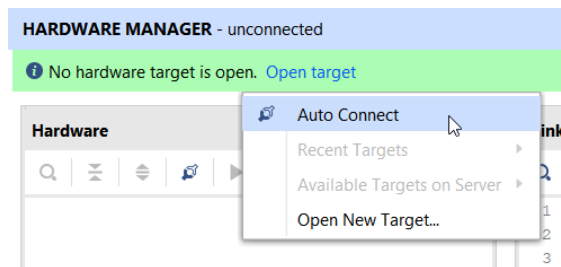


Figure C.9: Open target button

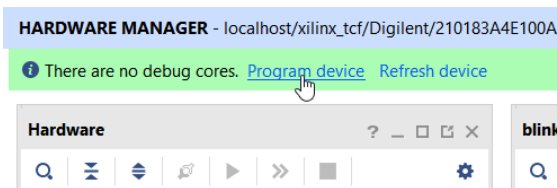


Figure C.10: Program device button

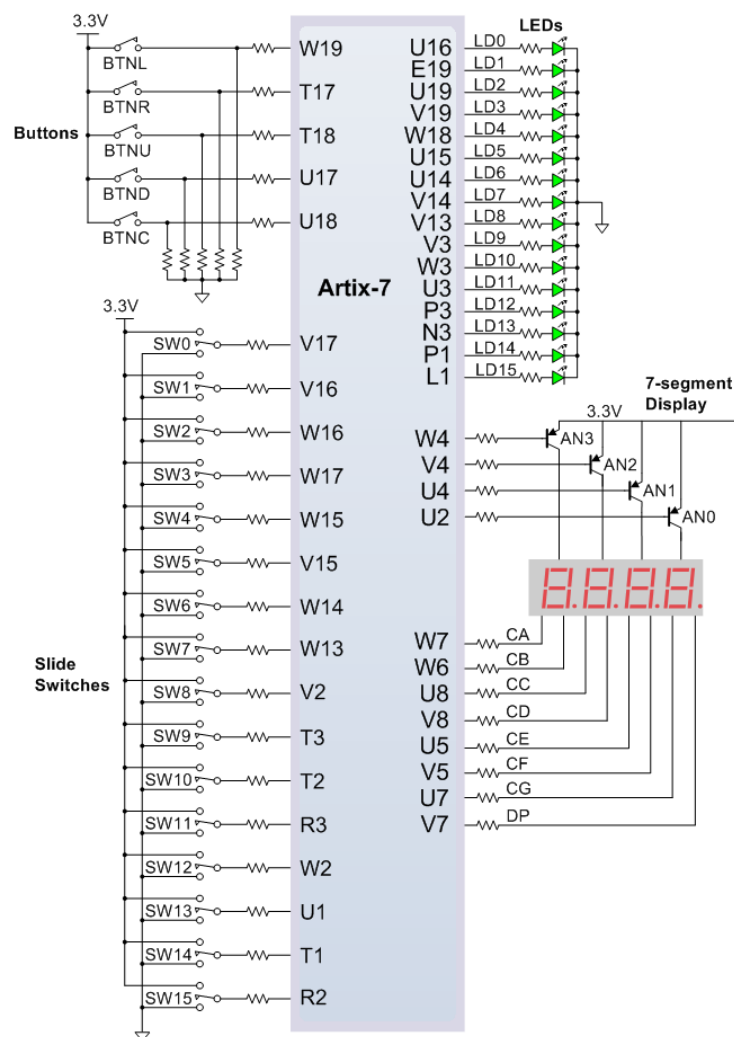
9. Set both switch 0 and 1 on the Basys3 board low. LED 0 should now blink.
10. Set switch 1 high. LED 0 should have a shorter pulse duration.
11. Finally, turn the FPGA off and set the *POWER* jumper back to *EXT* so that the next user can run the robot from a battery pack.

## Appendix D

# Overview of the FPGA pins

### D.1 Selected IO connections

The following picture shows how pins of the Artix 7 FPGA are directly connected to switches, buttons and LEDs on the Basys 3 board.



## D.2 Full list of connections

```

1  ## This file is the constraints file for the Robot.
2  ## It is based on the general .xdc for
3  ## the Basys3 rev B board by Digilent.
4  ## To use it in a project:
5  ## - uncomment the lines corresponding to used pins
6  ## - rename the used ports (in each line, after get_ports) according to the top level signal names in
   the project
7
8  ## Clock signal
9  set_property PACKAGE_PIN W5 [get_ports clk]
10 set_property IOSTANDARD LVCMOS33 [get_ports clk]
11 create_clock -period 10.000 -name sys_clk_pin -waveform {0.000 5.000} -add [get_ports clk]
12
13
14 ###
15 ### The signals below are specific for the Mars Rover
16 ###
17
18 ### Header JA, servo connections
19 set_property PACKAGE_PIN J1 [get_ports motor_r_pwm]
20 set_property IOSTANDARD LVCMOS33 [get_ports motor_r_pwm]
21 set_property PACKAGE_PIN L2 [get_ports motor_l_pwm]
22 set_property IOSTANDARD LVCMOS33 [get_ports motor_l_pwm]
23
24 ### Header JXADC, sensor connections
25 set_property PACKAGE_PIN J3 [get_ports sensor_r_in]
26 set_property IOSTANDARD LVCMOS33 [get_ports sensor_r_in]
27 set_property PACKAGE_PIN L3 [get_ports sensor_m_in]
28 set_property IOSTANDARD LVCMOS33 [get_ports sensor_m_in]
29 set_property PACKAGE_PIN M2 [get_ports sensor_l_in]
30 set_property IOSTANDARD LVCMOS33 [get_ports sensor_l_in]
31
32
33 ## Switches
34 #set_property PACKAGE_PIN V17 [get_ports {sw[0]}]
35 #set_property IOSTANDARD LVCMOS33 [get_ports {sw[0]}]
36 #set_property PACKAGE_PIN V16 [get_ports {sw[1]}]
37 #set_property IOSTANDARD LVCMOS33 [get_ports {sw[1]}]
38 #set_property PACKAGE_PIN W16 [get_ports {sw[2]}]
39 #set_property IOSTANDARD LVCMOS33 [get_ports {sw[2]}]
40 #set_property PACKAGE_PIN W17 [get_ports {sw[3]}]
41 #set_property IOSTANDARD LVCMOS33 [get_ports {sw[3]}]
42 #set_property PACKAGE_PIN W15 [get_ports {sw[4]}]
43 #set_property IOSTANDARD LVCMOS33 [get_ports {sw[4]}]
44 #set_property PACKAGE_PIN V15 [get_ports {sw[5]}]
45 #set_property IOSTANDARD LVCMOS33 [get_ports {sw[5]}]
46 #set_property PACKAGE_PIN W14 [get_ports {sw[6]}]
47 #set_property IOSTANDARD LVCMOS33 [get_ports {sw[6]}]
48 #set_property PACKAGE_PIN W13 [get_ports {sw[7]}]
49 #set_property IOSTANDARD LVCMOS33 [get_ports {sw[7]}]
50 #set_property PACKAGE_PIN V2 [get_ports {sw[8]}]
51 #set_property IOSTANDARD LVCMOS33 [get_ports {sw[8]}]
52 #set_property PACKAGE_PIN T3 [get_ports {sw[9]}]
53 #set_property IOSTANDARD LVCMOS33 [get_ports {sw[9]}]
54 #set_property PACKAGE_PIN T2 [get_ports {sw[10]}]
55 #set_property IOSTANDARD LVCMOS33 [get_ports {sw[10]}]
56 #set_property PACKAGE_PIN R3 [get_ports {sw[11]}]
57 #set_property IOSTANDARD LVCMOS33 [get_ports {sw[11]}]
58 #set_property PACKAGE_PIN W2 [get_ports {sw[12]}]
59 #set_property IOSTANDARD LVCMOS33 [get_ports {sw[12]}]
60 #set_property PACKAGE_PIN U1 [get_ports {sw[13]}]
61 #set_property IOSTANDARD LVCMOS33 [get_ports {sw[13]}]
62 #set_property PACKAGE_PIN T1 [get_ports {sw[14]}]
63 #set_property IOSTANDARD LVCMOS33 [get_ports {sw[14]}]
64 #set_property PACKAGE_PIN R2 [get_ports {sw[15]}]
65 #set_property IOSTANDARD LVCMOS33 [get_ports {sw[15]}]
66
67
68 ## LEDs
69 #set_property PACKAGE_PIN U16 [get_ports {leds[0]}]
70 # set_property IOSTANDARD LVCMOS33 [get_ports {leds[0]}]
71 #set_property PACKAGE_PIN E19 [get_ports {leds[1]}]
72 # set_property IOSTANDARD LVCMOS33 [get_ports {leds[1]}]
73 #set_property PACKAGE_PIN U19 [get_ports {leds[2]}]
74 # set_property IOSTANDARD LVCMOS33 [get_ports {leds[2]}]
75 #set_property PACKAGE_PIN V19 [get_ports {leds[3]}]
76 # set_property IOSTANDARD LVCMOS33 [get_ports {leds[3]}]

```

```

77 #set_property PACKAGE_PIN W18 [get_ports {leds[4]]}
78 #   set_property IOSTANDARD LVCMOS33 [get_ports {leds[4]]}
79 #set_property PACKAGE_PIN U15 [get_ports {leds[5]]}
80 #   set_property IOSTANDARD LVCMOS33 [get_ports {leds[5]]}
81 #set_property PACKAGE_PIN U14 [get_ports {leds[6]]}
82 #   set_property IOSTANDARD LVCMOS33 [get_ports {leds[6]]}
83 #set_property PACKAGE_PIN V14 [get_ports {leds[7]]}
84 #   set_property IOSTANDARD LVCMOS33 [get_ports {leds[7]]}
85 #set_property PACKAGE_PIN V13 [get_ports {leds[8]]}
86 #   set_property IOSTANDARD LVCMOS33 [get_ports {leds[8]]}
87 #set_property PACKAGE_PIN V3 [get_ports {leds[9]]}
88 #   set_property IOSTANDARD LVCMOS33 [get_ports {leds[9]]}
89 #set_property PACKAGE_PIN W3 [get_ports {leds[10]]}
90 #   set_property IOSTANDARD LVCMOS33 [get_ports {leds[10]]}
91 #set_property PACKAGE_PIN U3 [get_ports {leds[11]]}
92 #   set_property IOSTANDARD LVCMOS33 [get_ports {leds[11]]}
93 #set_property PACKAGE_PIN P3 [get_ports {leds[12]]}
94 #   set_property IOSTANDARD LVCMOS33 [get_ports {leds[12]]}
95 #set_property PACKAGE_PIN N3 [get_ports {leds[13]]}
96 #   set_property IOSTANDARD LVCMOS33 [get_ports {leds[13]]}
97 #set_property PACKAGE_PIN P1 [get_ports {leds[14]]}
98 #   set_property IOSTANDARD LVCMOS33 [get_ports {leds[14]]}
99 #set_property PACKAGE_PIN L1 [get_ports {leds[15]]}
100 #   set_property IOSTANDARD LVCMOS33 [get_ports {leds[15]]}
101
102
103 ##7 segment display
104 #set_property PACKAGE_PIN W7 [get_ports {seg[0]]}
105 #set_property IOSTANDARD LVCMOS33 [get_ports {seg[0]]}
106 #set_property PACKAGE_PIN W6 [get_ports {seg[1]]}
107 #set_property IOSTANDARD LVCMOS33 [get_ports {seg[1]]}
108 #set_property PACKAGE_PIN U8 [get_ports {seg[2]]}
109 #set_property IOSTANDARD LVCMOS33 [get_ports {seg[2]]}
110 #set_property PACKAGE_PIN V8 [get_ports {seg[3]]}
111 #set_property IOSTANDARD LVCMOS33 [get_ports {seg[3]]}
112 #set_property PACKAGE_PIN U5 [get_ports {seg[4]]}
113 #set_property IOSTANDARD LVCMOS33 [get_ports {seg[4]]}
114 #set_property PACKAGE_PIN V5 [get_ports {seg[5]]}
115 #set_property IOSTANDARD LVCMOS33 [get_ports {seg[5]]}
116 #set_property PACKAGE_PIN U7 [get_ports {seg[6]]}
117 #set_property IOSTANDARD LVCMOS33 [get_ports {seg[6]]}
118
119 #set_property PACKAGE_PIN V7 [get_ports dp]
120 #set_property IOSTANDARD LVCMOS33 [get_ports dp]
121
122 #set_property PACKAGE_PIN U2 [get_ports {an[0]]}
123 #set_property IOSTANDARD LVCMOS33 [get_ports {an[0]]}
124 #set_property PACKAGE_PIN U4 [get_ports {an[1]]}
125 #set_property IOSTANDARD LVCMOS33 [get_ports {an[1]]}
126 #set_property PACKAGE_PIN V4 [get_ports {an[2]]}
127 #set_property IOSTANDARD LVCMOS33 [get_ports {an[2]]}
128 #set_property PACKAGE_PIN W4 [get_ports {an[3]]}
129 #set_property IOSTANDARD LVCMOS33 [get_ports {an[3]]}
130
131
132 ##Buttons
133 #set_property PACKAGE_PIN T18 [get_ports btnU]
134 #set_property IOSTANDARD LVCMOS33 [get_ports btnU]
135 #set_property PACKAGE_PIN W19 [get_ports btnL]
136 #set_property IOSTANDARD LVCMOS33 [get_ports btnL]
137 #set_property PACKAGE_PIN T17 [get_ports btnR]
138 #set_property IOSTANDARD LVCMOS33 [get_ports btnR]
139 #set_property PACKAGE_PIN U17 [get_ports btnD]
140 #set_property IOSTANDARD LVCMOS33 [get_ports btnD]
141
142 ##Pmod Header JA
143 ##Sch name = JA1
144 #set_property PACKAGE_PIN J1 [get_ports {JA[0]]}
145 #set_property IOSTANDARD LVCMOS33 [get_ports {JA[0]]}
146 ##Sch name = JA2
147 #set_property PACKAGE_PIN L2 [get_ports {JA[1]]}
148 #set_property IOSTANDARD LVCMOS33 [get_ports {JA[1]]}
149 ##Sch name = JA3
150 #set_property PACKAGE_PIN J2 [get_ports {JA[2]]}
151 #set_property IOSTANDARD LVCMOS33 [get_ports {JA[2]]}
152 ##Sch name = JA4
153 #set_property PACKAGE_PIN G2 [get_ports {JA[3]]}
154 #set_property IOSTANDARD LVCMOS33 [get_ports {JA[3]]}
155 ##Sch name = JA7
156 #set_property PACKAGE_PIN H1 [get_ports {JA[4]]}

```

```

157 #set_property IOSTANDARD LVCMOS33 [get_ports {JA[4]}]
158 ##Sch name = JA8
159 #set_property PACKAGE_PIN K2 [get_ports {JA[5]}]
160 #set_property IOSTANDARD LVCMOS33 [get_ports {JA[5]}]
161 ##Sch name = JA9
162 #set_property PACKAGE_PIN H2 [get_ports {JA[6]}]
163 #set_property IOSTANDARD LVCMOS33 [get_ports {JA[6]}]
164 ##Sch name = JA10
165 #set_property PACKAGE_PIN G3 [get_ports {JA[7]}]
166 #set_property IOSTANDARD LVCMOS33 [get_ports {JA[7]}]
167
168
169
170 ##Pmod Header JB
171 ##Sch name = JB1
172 #set_property PACKAGE_PIN A14 [get_ports {JB[0]}]
173 #set_property IOSTANDARD LVCMOS33 [get_ports {JB[0]}]
174 ##Sch name = JB2
175 #set_property PACKAGE_PIN A16 [get_ports {JB[1]}]
176 #set_property IOSTANDARD LVCMOS33 [get_ports {JB[1]}]
177 ##Sch name = JB3
178 #set_property PACKAGE_PIN B15 [get_ports {JB[2]}]
179 #set_property IOSTANDARD LVCMOS33 [get_ports {JB[2]}]
180 ##Sch name = JB4
181 #set_property PACKAGE_PIN B16 [get_ports {JB[3]}]
182 #set_property IOSTANDARD LVCMOS33 [get_ports {JB[3]}]
183 ##Sch name = JB7
184 #set_property PACKAGE_PIN A15 [get_ports {JB[4]}]
185 #set_property IOSTANDARD LVCMOS33 [get_ports {JB[4]}]
186 ##Sch name = JB8
187 #set_property PACKAGE_PIN A17 [get_ports {JB[5]}]
188 #set_property IOSTANDARD LVCMOS33 [get_ports {JB[5]}]
189 ##Sch name = JB9
190 #set_property PACKAGE_PIN C15 [get_ports {JB[6]}]
191 #set_property IOSTANDARD LVCMOS33 [get_ports {JB[6]}]
192 ##Sch name = JB10
193 #set_property PACKAGE_PIN C16 [get_ports {JB[7]}]
194 #set_property IOSTANDARD LVCMOS33 [get_ports {JB[7]}]
195
196
197
198 ##Pmod Header JC
199 ##Sch name = JC1
200 #set_property PACKAGE_PIN K17 [get_ports {JC[0]}]
201 #set_property IOSTANDARD LVCMOS33 [get_ports {JC[0]}]
202 ##Sch name = JC2
203 #set_property PACKAGE_PIN M18 [get_ports {JC[1]}]
204 #set_property IOSTANDARD LVCMOS33 [get_ports {JC[1]}]
205 ##Sch name = JC3
206 #set_property PACKAGE_PIN N17 [get_ports {JC[2]}]
207 #set_property IOSTANDARD LVCMOS33 [get_ports {JC[2]}]
208 ##Sch name = JC4
209 #set_property PACKAGE_PIN P18 [get_ports {JC[3]}]
210 #set_property IOSTANDARD LVCMOS33 [get_ports {JC[3]}]
211 ##Sch name = JC7
212 #set_property PACKAGE_PIN L17 [get_ports {JC[4]}]
213 #set_property IOSTANDARD LVCMOS33 [get_ports {JC[4]}]
214 ##Sch name = JC8
215 #set_property PACKAGE_PIN M19 [get_ports {JC[5]}]
216 #set_property IOSTANDARD LVCMOS33 [get_ports {JC[5]}]
217 ##Sch name = JC9
218 #set_property PACKAGE_PIN P17 [get_ports {JC[6]}]
219 #set_property IOSTANDARD LVCMOS33 [get_ports {JC[6]}]
220 ##Sch name = JC10
221 #set_property PACKAGE_PIN R18 [get_ports {JC[7]}]
222 #set_property IOSTANDARD LVCMOS33 [get_ports {JC[7]}]
223
224
225 ##Pmod Header JXADC
226 ##Sch name = XA1_P
227 #set_property PACKAGE_PIN J3 [get_ports {JXADC[0]}]
228 #set_property IOSTANDARD LVCMOS33 [get_ports {JXADC[0]}]
229 ##Sch name = XA2_P
230 #set_property PACKAGE_PIN L3 [get_ports {JXADC[1]}]
231 #set_property IOSTANDARD LVCMOS33 [get_ports {JXADC[1]}]
232 ##Sch name = XA3_P
233 #set_property PACKAGE_PIN M2 [get_ports {JXADC[2]}]
234 #set_property IOSTANDARD LVCMOS33 [get_ports {JXADC[2]}]
235 ##Sch name = XA4_P
236 #set_property PACKAGE_PIN N2 [get_ports {JXADC[3]}]

```

```

237 #set_property IOSTANDARD LVCMOS33 [get_ports {JXADC[3]}]
238 ##Sch name = XA1_N
239 #set_property PACKAGE_PIN K3 [get_ports {JXADC[4]}]
240 #set_property IOSTANDARD LVCMOS33 [get_ports {JXADC[4]}]
241 ##Sch name = XA2_N
242 #set_property PACKAGE_PIN M3 [get_ports {JXADC[5]}]
243 #set_property IOSTANDARD LVCMOS33 [get_ports {JXADC[5]}]
244 ##Sch name = XA3_N
245 #set_property PACKAGE_PIN M1 [get_ports {JXADC[6]}]
246 #set_property IOSTANDARD LVCMOS33 [get_ports {JXADC[6]}]
247 ##Sch name = XA4_N
248 #set_property PACKAGE_PIN N1 [get_ports {JXADC[7]}]
249 #set_property IOSTANDARD LVCMOS33 [get_ports {JXADC[7]}]
250
251
252
253 ##VGA Connector
254 #set_property PACKAGE_PIN G19 [get_ports {vgaRed[0]}]
255 #set_property IOSTANDARD LVCMOS33 [get_ports {vgaRed[0]}]
256 #set_property PACKAGE_PIN H19 [get_ports {vgaRed[1]}]
257 #set_property IOSTANDARD LVCMOS33 [get_ports {vgaRed[1]}]
258 #set_property PACKAGE_PIN J19 [get_ports {vgaRed[2]}]
259 #set_property IOSTANDARD LVCMOS33 [get_ports {vgaRed[2]}]
260 #set_property PACKAGE_PIN N19 [get_ports {vgaRed[3]}]
261 #set_property IOSTANDARD LVCMOS33 [get_ports {vgaRed[3]}]
262 #set_property PACKAGE_PIN N18 [get_ports {vgaBlue[0]}]
263 #set_property IOSTANDARD LVCMOS33 [get_ports {vgaBlue[0]}]
264 #set_property PACKAGE_PIN L18 [get_ports {vgaBlue[1]}]
265 #set_property IOSTANDARD LVCMOS33 [get_ports {vgaBlue[1]}]
266 #set_property PACKAGE_PIN K18 [get_ports {vgaBlue[2]}]
267 #set_property IOSTANDARD LVCMOS33 [get_ports {vgaBlue[2]}]
268 #set_property PACKAGE_PIN J18 [get_ports {vgaBlue[3]}]
269 #set_property IOSTANDARD LVCMOS33 [get_ports {vgaBlue[3]}]
270 #set_property PACKAGE_PIN J17 [get_ports {vgaGreen[0]}]
271 #set_property IOSTANDARD LVCMOS33 [get_ports {vgaGreen[0]}]
272 #set_property PACKAGE_PIN H17 [get_ports {vgaGreen[1]}]
273 #set_property IOSTANDARD LVCMOS33 [get_ports {vgaGreen[1]}]
274 #set_property PACKAGE_PIN G17 [get_ports {vgaGreen[2]}]
275 #set_property IOSTANDARD LVCMOS33 [get_ports {vgaGreen[2]}]
276 #set_property PACKAGE_PIN D17 [get_ports {vgaGreen[3]}]
277 #set_property IOSTANDARD LVCMOS33 [get_ports {vgaGreen[3]}]
278 #set_property PACKAGE_PIN P19 [get_ports Hsync]
279 #set_property IOSTANDARD LVCMOS33 [get_ports Hsync]
280 #set_property PACKAGE_PIN R19 [get_ports Vsync]
281 #set_property IOSTANDARD LVCMOS33 [get_ports Vsync]
282
283
284 ##USB-RS232 Interface
285 #set_property PACKAGE_PIN B18 [get_ports RsRx]
286 #set_property IOSTANDARD LVCMOS33 [get_ports RsRx]
287 #set_property PACKAGE_PIN A18 [get_ports RsTx]
288 #set_property IOSTANDARD LVCMOS33 [get_ports RsTx]
289
290
291 ##USB HID (PS/2)
292 #set_property PACKAGE_PIN C17 [get_ports PS2Clk]
293 #set_property IOSTANDARD LVCMOS33 [get_ports PS2Clk]
294 #set_property PULLUP true [get_ports PS2Clk]
295 #set_property PACKAGE_PIN B17 [get_ports PS2Data]
296 #set_property IOSTANDARD LVCMOS33 [get_ports PS2Data]
297 #set_property PULLUP true [get_ports PS2Data]
298
299
300 ##Quad SPI Flash
301 ##Note that CCLK_o cannot be placed in 7 series devices. You can access it using the
302 ##STARTUPE2 primitive.
303 #set_property PACKAGE_PIN D18 [get_ports {QspiDB[0]}]
304 #set_property IOSTANDARD LVCMOS33 [get_ports {QspiDB[0]}]
305 #set_property PACKAGE_PIN D19 [get_ports {QspiDB[1]}]
306 #set_property IOSTANDARD LVCMOS33 [get_ports {QspiDB[1]}]
307 #set_property PACKAGE_PIN G18 [get_ports {QspiDB[2]}]
308 #set_property IOSTANDARD LVCMOS33 [get_ports {QspiDB[2]}]
309 #set_property PACKAGE_PIN F18 [get_ports {QspiDB[3]}]
310 #set_property IOSTANDARD LVCMOS33 [get_ports {QspiDB[3]}]
311 #set_property PACKAGE_PIN K19 [get_ports QspiCSn]
312 #set_property IOSTANDARD LVCMOS33 [get_ports QspiCSn]
313
314
315 set_property BITSTREAM.CONFIG.CONFIGRATE 33 [current_design]
316 set_property CONFIG_MODE SPIx4 [current_design]

```

```
317  
318 set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]
```



## Appendix E

# Working with an oscilloscope

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 2022B 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 2022B is depicted in Figure E.1. The buttons on the devices are grouped, based on functionality.

### 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

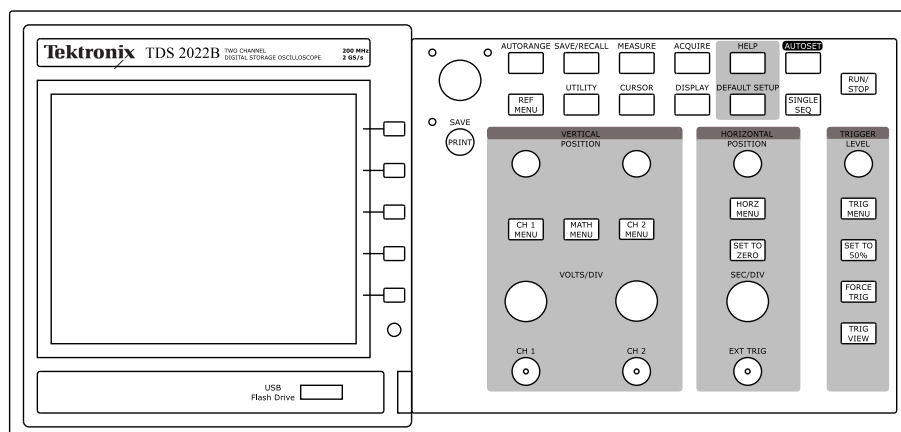


Figure E.1: The Tektronix TDS 2022B Digital Storage Oscilloscope

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×, 10×, 100×, 1000×)  
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 Sequence

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 autosest 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<sup>1</sup> 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;

---

<sup>1</sup>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.

**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.

### 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.

# Bibliography

- [1] "Basys 3 Reference Manual - Digilent Reference," diligent.com.  
<https://diligent.com/reference/programmable-logic/basys-3/reference-manual>
- [2] "XC7A35T-1CPG236C," <https://www.digikey.nl/en/products/detail/amd/XC7A35T-1CPG236C/5039071>





