MP-30 C:

Praktikum
Programmierbare Elektronik

Versuchsanleitung

Wintersemester 2013/14

Justus-Liebig-Universität Gießen

Written by Thomas Geßler
Thomas.Gessler@exp2.physik.uni-giessen.de

Last change: April 15, 2014

This document was typeset using LaTeX and the memoir class. The text is set in Libertine and Biolinum. The source code is set in Bera Mono.

Contents

Nomenclature

BGA	Ball grid array
CLB	Configurable logic block
DCM	Digital clock manager
DDR	Double data rate
FG320	320-ball fine-pitch BGA
FIFO	First in, first out
FPGA	Field-programmable gate array
FSM	Finite-state machine
GND	Ground
HDL	Hardware description language
IBUF	Input buffer
IC	Integrated circuit
IOB	Input/output block
I/O	Input/output
JTAG	Joint Test Action Group
LIFO	Last in, first out
LSB	Least significant bit
LUT	Look-up table
Mux	Multiplexer
OBUF	Output buffer
PWM	pulse-width modulation
RAM	Random-access memory

RTL	Register-transfer level
RX	Receive
TTL	Transistor-transistor logic
TX	Transmit
UART	Universal asynchronous receiver/transmitter
UCF	User Constraints File
UUT	Unit under test
V_{cc}	Collector supply voltage (IC power-supply pin)
VGA	Video Graphics Array
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

Introduction

Scope of this Course

This course is an introduction to programmable logic, in particular FPGAs, and the hardware description language VHDL. There are no hard prerequisites for this course. You should have a basic understanding of digital signal processing: For example, you should know how to get the binary representation of an unsigned integer number. Apart from that, the required basics will be covered by the introductory lecture and the *Basics* section of each lab.

The following sections will give you a quick overview of the course's main subjects. If the section on digital electronics is completely new to you, you might want to read up on this topic before beginning with the labs.

Digital Electronics

Digital signals

Digital electronics is, simply speaking, concerned with information in the form of discrete signals. Usually, there are exactly two states that a digital signal can be in: 0 or 1. These *logic levels* can be labeled in many different ways, some of which are:

1	True	T	High	H	On	Yes	V_{cc}	$\cdots$
0	False	F	Low	L	Off	No	GND	$\cdots$

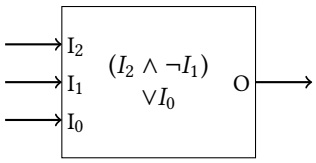
The analog level of a signal in an electronic circuit must be in a certain range in order to be a valid digital signal. The exact mapping depends on the technology or *logic family* used: The TTL family, for example, defines voltages between 0 V and 0.8 V as a logic 0, while voltages between 2 V and 5 V correspond to a logic 1. Voltages outside of these ranges are invalid as digital signals.

Logic functions and truth tables

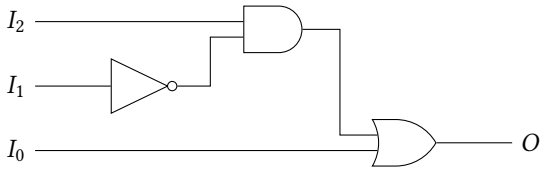
The information carried by a single logic signal is referred to as a *bit*. Logic functions are functions that map one or more input bits to one output bit. The exact mapping of input bits to output bit can be written down in a *truth table*. An example for a function with three input bits is shown in figure 1a. Since each of the three bits can

I_2	0	0	0	0	1	1	1	1
I_1	0	0	1	1	0	0	1	1
I_0	0	1	0	1	0	1	0	1
O	**0**	**1**	**0**	**1**	**1**	**1**	**0**	**1**

(a) Truth table



(b) Black-box diagram



(c) Circuit diagram

Figure 1: Representations of a logic function $O(I_2, I_1, I_0)$

be in two possible states, there are $2^3 = 8$ entries in the truth table. The function is defined by the output it produces for each possible input, i.e. by the "O-row" of its truth table. Since there are $2^8 = 256$ possible O-rows, there are 256 different logic functions with three input bits.

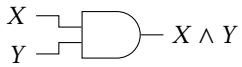
For any logic function with n input bits it is possible to create an electronic circuit with n input signals and one output signal that implements the function.[1] Such circuits are often represented with black-box diagrams like the one in figure 1b: The inputs, outputs, and description of the circuit are shown, but its actual implementation is hidden. We will use such diagrams for structural programming in hardware description languages.

Boolean algebra

The branch of mathematics that deals with logic functions is called *Boolean algebra*. The three basic operations in this theory are:

1. The *conjunction* AND of two signals, denoted as $\wedge$:

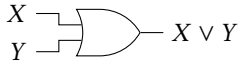
X	0	0	1	1
Y	0	1	0	1
$X \wedge Y$	**0**	**0**	**0**	**1**



   The result is 1 only if *both* X and Y are 1. Otherwise it is 0.

2. The *disjunction* OR of two signals, denoted as $\vee$:

X	0	0	1	1
Y	0	1	0	1
$X \vee Y$	**0**	**1**	**1**	**1**

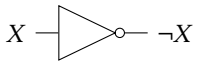


[1] Of course, any real circuit has a finite rise time: The output signal is delayed with respect to changes of the input signals, and when it changes, it passes through a continuous voltage range that includes invalid logic states. For now, we consider idealized circuits.

The result is 1 if *at least* one of X or Y is 1. Otherwise it is 0.

3. The *negation* NOT of a signal, denoted as $\neg$:

X	0	1
$\neg X$	**1**	**0**

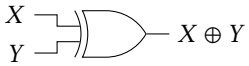


The result is the inverse of X.

Any logic function can be built from these operations—in fact, NOT and *one of* AND or OR are already sufficient. The example function given in figure 1 can be written as $O = (I_2 \wedge \neg I_1) \vee I_0$. A possible graphical representation is show in figure 1c.

Another important operation is the *exclusive disjunction* XOR of two signals, denoted as $\oplus$:

X	0	0	1	1
Y	0	1	0	1
$X \oplus Y$	**0**	**1**	**1**	**0**



The result is 1 if *exactly* one of X or Y is 1. Otherwise it is 0. The operation is associative, so we can write: $(X \oplus Y) \oplus Z = X \oplus (Y \oplus Z) =: X \oplus Y \oplus Z$. When applied to any number of inputs, it yields 1 if the number of 1s in the inputs is odd, and 0 if it is even (as can be seen for two inputs in the truth table). This is useful for many electronic circuits, for example parity generators and adders.

Programmable Logic

Hardware used in this course

The term *programmable logic* can refer to a range of different devices. In this course, we are only concerned with *field-programmable gate arrays* (FPGAs). For the labs we will use the Digilent Nexys2 FPGA board. Its reference manual [2] gives a complete description of the board. The FPGA used on the Nexys2 is a Xilinx Spartan-3E XC3S500E. Details about the FPGA can be found in its data sheet [3]. The structure and functionality of FPGAs will be explained using the Spartan-3E as example. Most concepts are, however, equally true or similar for other FPGA families.

The concept of FPGA programming

There is a fundamental difference between writing a computer program for a processor in a programming language like C, and programming an FPGA with the help of a hardware description language:

A program written in C is passed through a compiler, which (simply speaking) turns it into a set of instructions for a processor. The processor is an integrated circuit made up of several dedicated hardware blocks (arithmetic logic unit, cache, etc.) with distinct functions. The instructions in the compiled machine code tell the processor which operation to perform next: Get data from a memory address, add a certain number to it, use the result as the address from where to read the next instruction, ...

An FPGA, on the other hand, cannot perform any functions before it is programmed. It consists of many generic hardware elements that can be connected to each other arbitrarily. In this way, a multitude of different circuits can be realized

with the same hardware. Which elements are used, and how they are connected, is defined using a hardware description language. It is, for example, possible to use FPGA components to build a processor (which, in turn, can run software written in C). The available resources are the limiting factor in what is realizable with an FPGA.

Structure of an FPGA

FPGAs are integrated circuits: At their core is a small *die* (chip) of a semiconductor, usually silicon. The silicon is grown into large monocrystalline cylinders, which are then sawed into thin wafers. Electronic structures are produced on these wafers by doping (formation of transistors), metal deposition (formation of interconnect wires), and other techniques. Many separate dies can be produced on the same wafer. After the wafer has been processed and tested, it is diced (sawed, or scored and broken) into the individual dies.

Figure 2 shows a schematic overview of the Spartan3E-500 silicon chip. It consists of a regular arrangement of different fundamental device elements. The majority of the chip's area is taken by *configurable logic blocks* (CLBs): There are 46 rows and 34 columns of CLBs, with some of the area used up by other elements. The total number of CLBs is 1164.

The CLBs are the heart of the FPGA's programmability. Each CLB contains four *slices*. Each slice contains:

- two 4-input *look-up tables*, for implementation of arbitrary four-input logic functions,
- two *D flip-flops*, for storage of single bits, and
- additional logic, for larger logic functions and other purposes.

A wide variety of circuits can be realized using only CLBs. The usage of look-up tables and flip-flops will be explained in the first labs of this course.

Addressable memory and multipliers can be built from CLBs, but this is not very efficient, since such circuits are often needed and need many resources. Therefore they are included as dedicated circuits in many FPGAs. The Spartan3E-500 has 360 kbit of so-called *block RAM* and 20 dedicated multipliers. It also features 4 digital clock managers (DCMs) that serve many purposes related to clock signals (for example, double the frequency of a 50 MHz clock, or shift its phase by 180°).

IOBs and packaging

Signals inside the FPGA are connected to the outside through *input/output blocks* (IOBs). The IOBs convert between the voltage used inside the FPGA and a range of available logic families for external signals connected to the FPGA. They also provide additional functions, like inversion or delay of input signals, or storage of I/O signals in flip-flops.

The Spartan3E-500 has its IOBs arranged along all four edges of the silicon die. They can be seen as small, empty rectangles in figure 2. For each IOB there is a metal pad at the die edge, to which a wire can be connected. However, the FPGA die is usually not delivered as a "naked" silicon chip, but encased in an integrated circuit package that can be soldered on a printed circuit board.

Figure 3a shows the packaged Spartan-3E on the Nexys2 board. The package is an FG320 *ball grid array* (BGA) package: The die is encapsulated in a plastic case, and

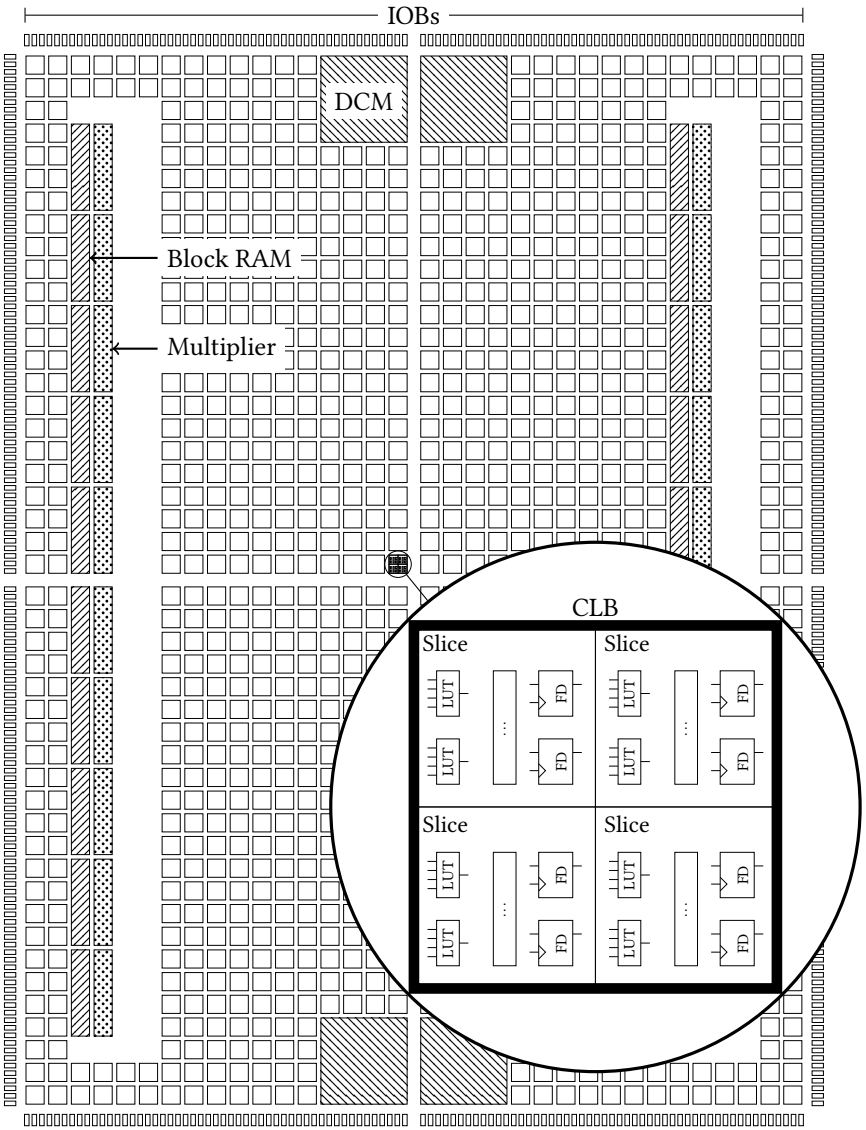
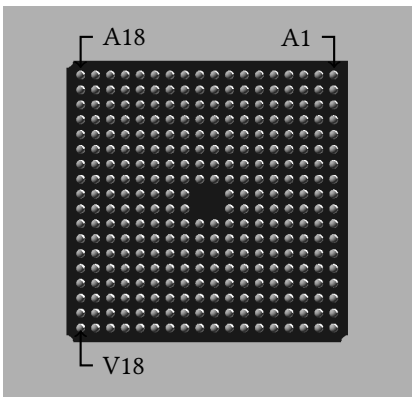


Figure 2: Components of the Spartan3E-500 integrated circuit die. The detailed view of one CLB shows four slices with look-up tables (LUT) and D flip-flops (FD); additional slice logic was omitted from this picture.

(a) FPGA on the Digilent Nexys2 board. The marking in the upper left-hand corner indicates the first pad (A1).



(b) Package bottom with 320 solder balls. The rows are labeled A through V, the columns are labeled 1 through 18.

Figure 3: Spartan3E-500 FPGA in an FG320 package

the IOB pads are connected, through wires inside the case, to a grid of solder balls on the package bottom (see figure 3b). There are 18×18 solder balls with a 2×2 gap in the center, adding up to 320 connections from FPGA to board. The solder pads are labeled A1 through V18, according to their position in the grid, and this nomenclature will be used during FPGA programming when referring to a certain I/O pin. Some of the pads used for special purposes, like voltage supplies or configuration signals. With the FG320 package, 232 of the IOBs from figure 2 are connected to a pad and can be used.

Hardware Description Languages

VHDL Basics

Hardware description languages make it possible to capture the behavior of an electronic circuit in the form of source code. They are also an important tool for FPGA programming. The two most important languages in this field are Verilog and VHDL (*VHSIC Hardware Description Language*; VHSIC stands for *Very High Speed Integrated Circuit*). In this course you will learn the basics of VHDL.

The example in listing 1 shows a VHDL file that describes a very simple electronic circuit: an AND gate. Some central VHDL keywords will be explained on this example:

- The **`library`** and **`use`** keywords are used to import functionality from external libraries, similar to the `#include` statement in C. These two lines will be at the top of every VHDL file written during this course. They define the `std_logic` data type, the basic data type for 1-bit electronic signals.

- The **`entity`** block (which has the arbitrary name `and_gate`) contains a **`port`** list. It gives a "black box" description of the hardware: The names and directions of all input and output ports are listed (the AND gate has the inputs `A` and `B` and the output `C`), but no information is given about what the circuit actually *does*.

Listing 1: VHDL model for an AND gate

```
library ieee;
use ieee.std_logic_1164.all;

entity and_gate is
    port(
        A : in  std_logic;
        B : in  std_logic;
        C : out std_logic
    );
end entity and_gate;

architecture behavioral of and_gate is
begin
    C <= (A and B) after 7 ns;
end architecture behavioral;
```

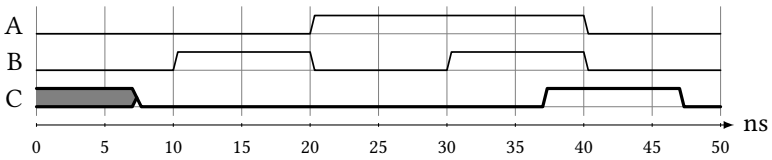
- The **architecture** block (which has the arbitrary name `behavioral` and belongs to the entity `and_gate`) determines the inner workings of the black box: It defines how the inputs and outputs are related to each other. We make a very simple statement here: Assign to the output `C` the result of the operation `A and B` with a delay of 7 ns.

This is a possible description for a real logic gate. It even takes switching delays into account: When one of the inputs changes, the output does not change to a new value immediately, but after a delay (induced by effects such as wire propagation times and transistor gate capacitances). A more thorough modeling of a real circuit might include different delay times for a rising and a falling output, or a short period after an input changes, during which the output is invalid (because it passes through an invalid analog band).

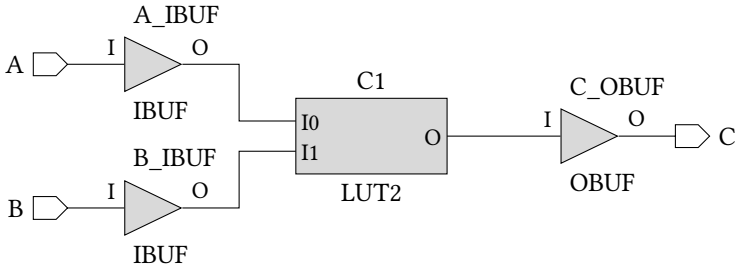
Synthesis for FPGAs

A VHDL model like the above can be used to simulate existing hardware with computer programs. The simulator uses a fixed pattern for the model's inputs (defined in a so-called *test bench*) and calculates the output values for each point in time. The result is a waveform like the one shown in figure 4a: The values for A and B at each time are specified by the test bench; the simulator determines the value of C using the VHDL model.

When working with FPGAs, however, we don't want to characterize an existing circuit. Our goal is to describe the behavior of a circuit with VHDL and make the FPGA behave in the described way. This requires a process called *synthesis*. FPGA vendors usually supply synthesis software for their products. The synthesis program uses HDL files like the above example and tries to create an equivalent circuit out of components available in the selected FPGA. This circuit is called *netlist*. The netlist for an AND gate, synthesized form the example VHDL code for the Spartan-3E with the Xilinx synthesis software, is shown in figure 4b.



(a) Simulation output waveform. The delay of 7 ns is simulated, which is why the output is undefined during the first 7 ns.



(b) Synthesis output netlist. The delay specification is ignored, and the **`and`** operation is synthesized as a 2-input look-up table.

Figure 4: Software outputs for the AND gate VHDL model

The netlist contains four FPGA components: An IOB for each input and output—they are configured, according to their specific function, as input buffers (IBUF) and output buffers (OBUF)—and one 2-input look-up table (labeled LUT2) that is configured to behave like an AND gate.[2]

Other programs use the netlist and additional information (such as I/O pin placement and timing relationships between external signals) and perform several processes that can be collectively referred to as *implementation*: Each component in the netlist is assigned an exact position in the FPGA. The configuration for the components is determined, and connections between them are established by configuring the FPGA's routing resources.

The result of the implementation processes is a programming file that can be loaded into the FPGA with special hardware (for example, a USB download cable from the FPGA vendor). In Xilinx terms this programming file is called *bitstream*. After the FPGA is loaded with the bitstream it should behave like the circuit described in the VHDL file.

Synthesizable HDL code

VHDL is a powerful language that can describe a wide variety of hardware. However, only a very specific subset of possible circuits can be realized with FPGA components. Synthesis tools can therefore not interpret all possible VHDL statements, but only a small part, referred to as "synthesizable VHDL".

The example given above contains a keyword that is *not* synthesizable: The expression **`after`** `7 ns` includes a delay specification: The output changes exactly

[2] A 2-input look-up table can emulate any logic function with two inputs. Note that the Spartan-3E actually has 4-input look-up tables as its basic components, but any two-input logic function can be written as a four-input function with two of its inputs ignored.

7 ns after one of the inputs changes. A VHDL simulation program will happily simulate this behavior (see figure 4a), but a synthesis program will run into problems: FPGAs do not usually have components that can delay a signal by an arbitrary amount of time. The FPGA circuit *will* have a certain delay, because of routing and component switching times, but the synthesis tool cannot guarantee a certain value for it.

Synthesis tools will ignore timing specifications like this one and interpret the assignment (`C <= A and B`) without them. Note that there are VHDL statements that are valid in simulations but will crash synthesis tools: `A <= not A after 10 ns;` produces an oscillating signal in a simulation, but a synthesis tool will abort when it encounters this line.

Lab Reports

A report is required for each finished lab. Begin with a short summary of everything you did and what the results were.

Describe, in reasonable detail, what you did in each part of the lab, what problems occurred, etc. Reasonable means: Don't write about basics or theory. It should be possible (without the lab instructions), for someone with the same equipment and preparation, to recreate your steps from your report. Include drawings and answers to any questions asked in the lab instructions. Where it makes sense, include screen shots, log outputs, and so on.

Append all VHDL and UCF files you created for the lab. The code has to be well-documented: Write comments with a short description for each entity, generic, port, signal, variable, process, function, etc.

Draw block diagrams for VHDL entities, showing all inputs and outputs. You can draw them as black-boxes, or include internal connections if it helps to understand the design better (for example, if mainly concurrent statements and instantiation of submodules are used). You don't have to draw flowcharts for processes and other sequential blocks, but you should comment or describe them sufficiently.

Lab 1

FPGA Components and VHDL Basics

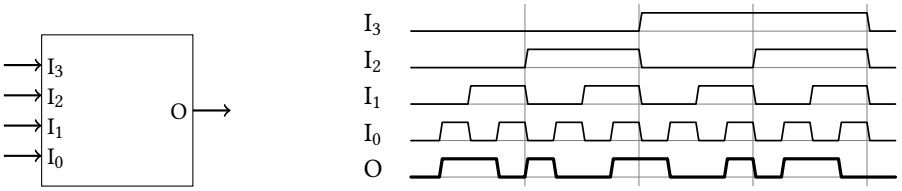
1.1 Learning Goals

- Digital electronics:
  - Look-up table
  - Full adder
  - Parity
- VHDL:
  - Basic concurrent assignments
  - Instantiation and inference
  - Structural and behavioral architecture
  - Location constraints
  - Signals

1.2 Basics — Digital Electronics

Look-up table

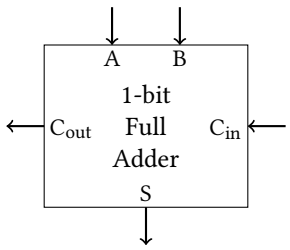
The Spartan3E-500 provides two 4-input look-up tables (LUTs) in each of its 4656 slices. Each LUT is a versatile logic gate that can be used to implement any logic function



(a) Black-box diagram

(b) Idealized timing behavior for a LUT, configured to implement the function $O = I_3 \oplus I_2 \oplus I_1 \oplus I_0$

Figure 1.1: A 4-input look-up table



(a) Black-box diagram

A	0	0	0	0	1	1	1	1
B	0	0	1	1	0	0	1	1
C_{in}	0	1	0	1	0	1	0	1
C_{out}	**0**	**0**	**0**	**1**	**0**	**1**	**1**	**1**
S	**0**	**1**	**1**	**0**	**1**	**0**	**0**	**1**

(b) Truth table

Figure 1.2: A 1-bit full adder

with up to four inputs. LUTs are therefore sometimes called *function generators*. Figure 1.1a shows a black-box diagram for a LUT with four inputs. The timing diagram in figure 1.1b shows the response of a LUT (configured with an example function) whose inputs go through all possible states.

To configure a LUT, the 16 bits for each possible input state (i.e., the O-row of the truth table) must be supplied. Fortunately, we do not have to determine this configuration ourselves; during FPGA programming, we usually *infer* LUTs: We specify the logic equation we want to implement in VHDL, and the synthesis and implementation tools determine how many LUT we need and how they are placed and configured.

Full adder

The addition of two binary numbers can be understood in terms of long addition, which works identically for decimal and binary numbers. Take the following example, which presents the same operation in decimal and binary radix:

	28			11100
+	14		+	01110
	1			1 1 1
=	42		=	101010

To add the two binary numbers, we first add the rightmost (lowest) digits: $0 + 0 = 0$. For the next digit, $0 + 1 = 1$. The result for the third digit is $1 + 1 = 10$, so 0 goes into the result and 1 is carried to the next digit (exactly like in the decimal example, where the 1 from $8 + 4 = 12$ is carried). For the fourth digit then, we have to take the carried bit into account, and get $1 + 1 + 1 = 11$. A 1 is written down, and a 1 is carried. The final operation is $1 + 0 + 1 = 10$. The carried 1 makes the resulting binary number longer by one bit than the two input numbers.

In digital electronics, binary addition of two numbers can be subdivided into identical circuits for each bit or digit, based on this principle. Each circuit (see figure 1.2a) takes as input the same digit from the two input numbers (A and B) and the carry bit from the previous digit (C_{in}). It has two outputs:

- The sum for the digit S. This is 1 if *exactly one* or *all three* of the inputs are 1. In other words, it is 1 if the number of 1s on the inputs is odd. This condition can be expressed with the XOR operation: $S = A \oplus B \oplus C_{\text{in}}$.

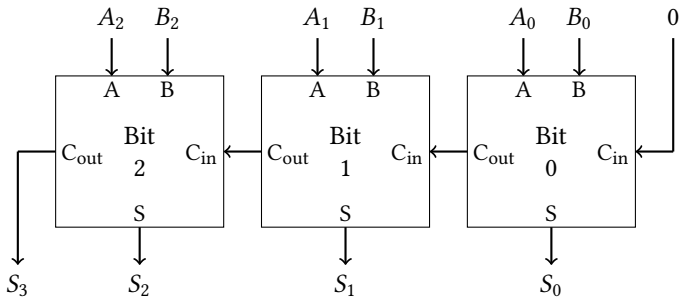


Figure 1.3: A 3-bit ripple-carry adder

- The carry bit for the next digit C_{out}. This in 1 if *at least* two of the inputs are 1: $C_{\text{out}} = (A \wedge B) \vee (A \wedge C_{\text{in}}) \vee (B \wedge C_{\text{in}})$.

This corresponds to the truth table shown in figure 1.2. Such a circuit is called a *full adder* (as opposed to a half adder, which has no carry input). For the addition of two n-bit numbers, we can build a *ripple-carry adder* by combining n full adders and connecting the C_{out} from one digit to the C_{in} of the next (see figure 1.3). The result can be interpreted as an $(n + 1)$-bit number whose highest bit is C_{out} from the last full adder in the carry chain.

Parity bits

A simple method to test the integrity of a signal with multiple bits—i.e., to check whether one of the bits has unexpectedly changed its value—is to add an additional *parity bit*. There are two types of parity bits:

1. An *even parity* bit is 1, if the number of signal bits with the value 1 is *odd*; otherwise, the parity bit is 0. The name "even parity" implies that the number of bits with the value 1 of the signal bit *plus* the even parity bit is even.

   Example: The even parity bit for the byte 00101010 is 1, because there are three bits (an odd number) with the value 1 in the byte.

2. Accordingly, *odd parity* bit is 1, if the number of signal bits with the value 1 is *even*; otherwise, the odd parity bit is 0.

   Example: The odd parity bit for the byte 00101010 is 0, because there are three bits (an odd number) with the value 1 in the byte.

If a multi-bit signal is transmitted from a sender to a receiver, and the sender adds a parity bit to the data, then the receiver can check whether the parity is still correct when the data arrives.

1.3 Basics — VHDL

Concurrent assignment and logic operators

The VHDL model in listing 1.1 describes a 2-input XOR gate (similar to the model of the AND gate from the introduction). It uses the operator `<=` to make a *concurrent assignment*. Concurrent assignments can appear in an **`architecture`** block between

Listing 1.1: An 2-input XOR gate

```
library ieee;
use ieee.std_logic_1164.all;

entity xor_2 is
    port (
        I : in  std_logic_vector(1 downto 0);
        O : out std_logic
    );
end entity xor_2;

architecture behavioral of xor_2 is
begin
    O <= I(0) xor I(1);
end architecture behavioral;
```

the **begin** and **end** keywords. They can assign a value to an output port. This value can be a constant, as in `X <= '1';`, or the result of a logical operation on one or more inputs, as in `Y <= A and B;`. Possible operations include **not**, **and**, **or**, **xor**, **nand**, **nor**, and **xnor**.

There is an important difference between concurrent assignments in VHDL and assignments in imperative programming languages like C: Assignments made with **<=** are not commands that are processed in order, like the statements in a C program. They are independent, parallel assignments, that have no particular order. During synthesis, they are translated into hard-wired connections and combinatorial logic.

A concurrent assignment to a port creates a *driver* for that port: a circuit that determines the logic level on this port at all times. For each port only one driver is allowed (just like you should not apply different voltage sources to the same wire). That is why the following assignment is forbidden:

```
X <= '1';
X <= A xor B; -- forbidden, since it creates a second driver
```

In an imperative language, the two lines would be interpreted in order: `X` is assigned the value `'1'`, and *then* `X` is assigned the result of the operation `A xor B`, making the first assignment obsolete. In VHDL, these statements produce an error during synthesis, since they are contradictory:

```
ERROR:Xst:528 - Multi-source in Unit <example_1> on signal <X>;
this signal is connected to multiple drivers.
```

Constants, vectors, and concatenation

Values in single quotes, like `'0'` or `'1'`, are 1-bit constants. Similarly, double quotes denote multi-bit constants: The number 42, written as an 8-bit binary constant in VHDL, would be `"00101010"`. The same constant can be written, more concisely, in hexadecimal notation with a preceding `x`: `x"2A"`.

The type `std_logic_vector` can be used to declare input and output ports as multi-bit values. `std_logic_vector` is an array of `std_logic`. Its range is declared with the keyword **`downto`**[1], as in:

```
X : out std_logic_vector(7 downto 0);
```

This declares an 8-bit output port, whose *leftmost* bit has the index 7, and whose *rightmost* bit has the index 0. Assigning a constant value to this port with the statement `X <= "11110000";` would set the bits with indices 0 to 3 to `'0'` and the bits with indices 4 to 7 to `'1'`;

Single elements of an array can be accessed with parentheses. The above assignment is equivalent to:

```
X(7) <= '1';
X(6) <= '1';
...
X(0) <= '0';
```

Several 1-bit and multi-bit values can be concatenated with the `&` character. Another equivalent notation for our 8-bit assignment would be: `X <= "111"& '1' & '0' & "00"& '0';`.

Instantiation and the port map

In listing 1.1 we described a 2-input XOR gate. Next we want to create a 4-input XOR gate, in other words: a circuit that implements the equation $O = (I_0 \oplus I_1) \oplus (I_2 \oplus I_3)$. In VHDL, it would be easy to describe this with a single line[2]:

```
O <= (I(0) xor I(1)) xor (I(2) xor I(3));
```

This is a possible *behavioral* description of a 4-input XOR gate. Now let us take a different approach and make a *structural* model of the same circuit.

In a structural description we use modules we already have as building blocks: To build a 4-input XOR gate, we can use three 2-input XOR gates: One combining I_0 and I_1, one combining I_2 and I_3, and one combining the results of the other two. (Of course, any other combination would produce the same result.) This concept is illustrated in figure 1.4.

In VHDL this can be realized by *instantiation* of a previously defined entity. In the case of our 4-input XOR gate, we would keep the file containing the definition of the entity `xor_2` in our design project. Next we create a *new* VHDL file for the model of the 4-input gate. This file will become the *top file* for our project. The top file of an FPGA design project contains the entity whose input and output ports correspond to physical inputs and outputs (i.e., IOBs) of the FPGA. All other VHDL files can be used as building blocks in a structural design.

Listing 1.2 shows what a structural model for a 4-input XOR gate might look like. The important part are the three blocks containing the **`port map`** keyword: Each of them instantiates[3] one copy of the entity `xor_2`, which is defined in another file.

[1] The range can also be declared with **`to`**, but we will not use this form during this course.

[2] We could have omitted the parentheses in both the equation and the VHDL assignment, since the XOR operation is associative.

[3] We use a method called direct instantiation. There is another possibility which requires the explicit declaration of a component, but we will not use it during this course.

Listing 1.2: A 4-input XOR gate with, built out of three 2-input XOR gates

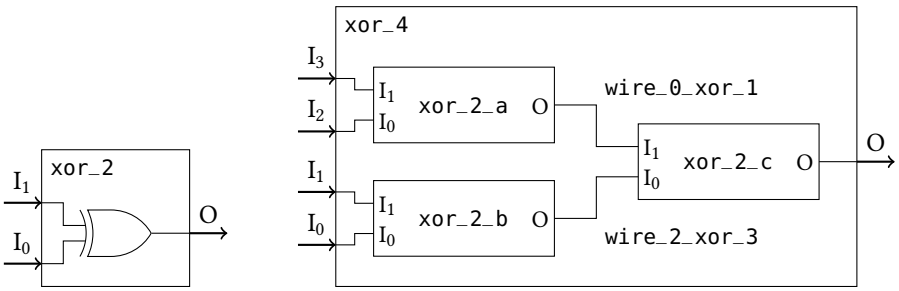
```
library ieee;
use ieee.std_logic_1164.all;

entity xor_4 is
    port(
        I : in  std_logic_vector(3 downto 0);
        O : out std_logic
    );
end entity xor_4;

architecture structural of xor_4 is
    signal wire_0_xor_1, wire_2_xor_3 : std_logic;
begin
    xor_2_a : entity xor_2
    port map (
        I(0) => I(0),
        I(1) => I(1),
        O    => wire_0_xor_1
    );

    xor_2_b : entity xor_2
    port map (
        I(0) => I(2),
        I(1) => I(3),
        O    => wire_2_xor_3
    );

    xor_2_c : entity xor_2
    port map (
        I(0) => wire_0_xor_1,
        I(1) => wire_2_xor_3,
        O    => O
    );
end architecture structural;
```



(a) A single XOR gate with two inputs

(b) A XOR gate with four inputs, built using three 2-input XOR gates as black boxes

Figure 1.4: Structural description of a 4-input XOR gate

Each instance of `xor_2` we declare must be given a unique name, preceding the colon. The entries in the **`port map`** determine how the input and output ports of the instance are connected to the ports of our top module (or to constant values or signals, see below). The port of the instantiated entity is always on the left side of the `=>` operator; whatever connects to this port is on the right side of the operator. For example, the line `I(0)=> I(2)` in the port map of instance `xor_2_b` connects the input `I(2)` of our 4-input XOR gate to the input `I(0)` of the instance `xor_2_b`.

All input ports of an instantiated entity must be connected to something[4], but connections to constant values like `I(0)=> '0'` are also valid. Unused output ports can be omitted from the port map, or explicitly declared as unconnected with `O => open`.

Signals

We use *signals* to establish the internal connections between the instantiated XOR gates. In the example, these signals are named `wire_0_xor_1` and `wire_2_xor_3` (compare figure 1.4b).

For now, you can think of signals as wires. Signals must be declared before they can be used. Signal declaration must be made in the **`architecture`** block *before* the **`begin`** keyword. Like ports, signals have a type (for example, `std_logic`), but unlike ports, they do not have a direction.

Location constraints

The implementation software determines where exactly in the FPGA each component (LUT, IOB, etc.) is placed. Depending on the software settings, the placement is done under certain criteria, for example, to use as little of the FPGA area as possible, or to minimize the distance between the output of one component and the input of the next. The user can influence this process by setting so-called *constraints*. Constraints are usually declared in a separate file, the *User Constraints File* (UCF).

For now, we are concerned only with *location constraints*. We can set a location constraint for each input and output port of our top entity. The constraint fixes the placement of the according IOB, so that it connects to an FPGA pad of our choice. (The pad name according to the grid position is used, for example J14; see figure 3b.)

[4]Unless they have been declared with a default value.

This is useful, because we want to use the buttons, switches, LEDs, etc. on the Nexys2 as our inputs and outputs, and each of them is connected to a certain FPGA pad. A label is printed next to each component on the Nexys2, telling us which pad it is connected to. The pad J14, for example, belongs to the rightmost LED.

Each constraint in the UCF file is a single line of the following form:

```
NET "I"      LOC=G18; # Use the rightmost switch as input I
NET "LED<0>" LOC=J14; # Use the rightmost LED as output LED(0)
```

For multi-bit ports, angle brackets must be used instead of parentheses.

1.4 Lab Instructions

Please remember to keep the created VHDL and UCF files after each exercise, since you must include them in your lab report. If an exercise builds upon a previous one, copy-and-paste the reused code into a new file.

Exercise 1: Basic workflow

1. Follow the instructions in the software tutorial (appendix A) to create a new project with PlanAhead and add a VHDL and a UCF file.
2. Create a VHDL entity with the following ports:
   - an 8-bit input vector `SW`; and
   - an 8-bit output vector `LED`.
3. Write an architecture block that makes a direct connection from the input vector to the output vector (i.e., the switches will control the LEDs).
4. Add location constraints that connect the switches on the Nexys2 to `SW` and the LEDs to `LED`.
5. Synthesize and implement the design. Generate a bitstream and test it on the FPGA.

Exercise 2: Simple logic function

1. Create a VHDL entity with the following ports:
   - an 4-bit input vector `I`; and
   - an 1-bit output port `O`.

   Remember to make this the top entity, and deactivate any existing UCF files! Do not write a new UCF file yet.
2. Write an architecture block that implements a 4-input logic function of your choice, using `I` as inputs and `O` as output.
3. Synthesize the design and inspect the netlist. How was the function realized? Save the netlist schematic.
4. Implement the design and inspect the device view. How was the placement optimized? Make screenshots.

5. Create a UCF file and assign `I` to four switches and `O` to an LED. Rerun the implementation. How did the placement change? Make screenshots.

6. Test the design on the FPGA.

Exercise 3: Parity generator

1. Create a VHDL entity with the following ports:

   - an 8-bit input vector `I`;
   - an 1-bit output port `P_EVEN`; and
   - an 1-bit output port `P_ODD`.

2. Write an architecture block that calculates the even parity bit (`P_EVEN`) and odd parity bit (`P_ODD`) for the input `I`.

3. Synthesize the design and document the netlist. Which FPGA components were used?

4. Create a UCF file and assign `I` to the eight switches and `P_EVEN` and `P_ODD` to an LED each. Generate the bitstream and test the design on the FPGA.

Exercise 4: Full adder

1. Create a VHDL entity with inputs and outputs according to figure 1.2.

2. Write the architecture for a full adder. Synthesize it and document the netlist.

3. Write a UCF file, implement the design, and test it on the FPGA.

Exercise 5: Ripple-carry adder

1. Create a VHDL entity with the following ports:

   - a 4-bit input vector `A`;
   - a 4-bit input vector `B`; and
   - a 5-bit output vector `S`.

2. Write a structural architecture for a 4-bit ripple-carry adder by instantiating full adders. Use `A` and `B` as inputs and `S` as the sum. (Use the carry output from the last adder as the highest bit for `S`.)

3. Synthesize the design and document the netlist. Which FPGA components were used?

4. Write a UCF file, implement the design, and test it on the FPGA.

Exercise 6 (∗): Inferring an adder from VHDL

VHDL has data types that allow simple arithmetic operations. They require the `numeric_std` package to be used:

```
use ieee.numeric_std.all;
```

With this package, an adder for two n-bit numbers `A` and `B` with the $n + 1$-bit sum `S` can be described with a single line:

```
S <= std_logic_vector(unsigned('0' & A) + unsigned('0' & B));
```

1. Create an adder with the same inputs and outputs as in the last exercise, using the above line.
2. Compare the netlist to the one from your structural architecture. Are there any differences?

Lab 2

Conditional Assignments

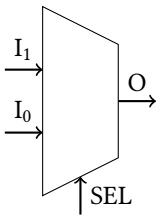
2.1 Learning Goals

- Digital electronics:
  - Multiplexer
  - Encoder
  - Seven-segment digit
- VHDL:
  - Conditional assignment
  - Selected assignment

2.2 Basics — Digital Electronics

Multiplexer

A 2^n-to-1 *multiplexer* (or *mux* for short) is a circuit with a 2^n-bit *data input* and a 1-bit output. An additional n-bit *select input* (SEL) is used to select the output bit out of the different data input bits: If SEL has the value 0, then the output takes the value of bit number 0 of the data input, and so on. This is illustrated in figure 2.1 for a 2-to-1 mux.

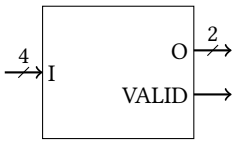


(a) Typical mux representation

SEL	0	0	0	0	1	1	1	1
I_1	0	0	1	1	0	0	1	1
I_0	0	1	0	1	0	1	0	1
O	**0**	**1**	**0**	**1**	**0**	**0**	**1**	**1**

(b) Truth table

Figure 2.1: A 2-to-1 multiplexer

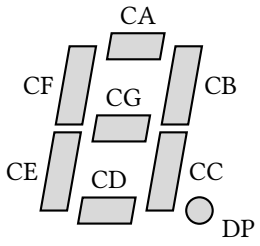


(a) Black-box diagram. The small numbers indicate the signal width.

I_3	I_2	I_1	I_0	O_1	O_0	VALID
0	0	0	1	**0**	**0**	**1**
0	0	1	0	**0**	**1**	**1**
0	1	0	0	**1**	**0**	**1**
1	0	0	0	**1**	**1**	**1**
others				**X**	**X**	**0**

(b) Truth table. The "don't care" value X can be set to 0 or 1 arbitrarily.

Figure 2.2: A 4-to-2 encoder



(a) One digit with cathode signal names

CA	→	C(0)			
CB	→	C(1)			
CC	→	C(2)			
CD	→	C(3)			
CE	→	C(4)	A0	→	A(0)
CF	→	C(5)	A1	→	A(1)
CG	→	C(6)	A2	→	A(2)
DP	→	C(7)	A3	→	A(3)

(b) Suggested naming scheme for the signals

Figure 2.3: Seven-segment display on the Nexys2

Encoder

A multi-bit signal is called *one-hot* if it is guaranteed that exactly one of its bits has the value 1 at all times: `"00000100"` is a valid one-hot signal, `"00101010"` is not.

A 2^n-to-n *encoder* is a circuit with a 2^n-bit input and an n-bit output. The input is a one-hot signal. The circuit returns the binary representation of the position of the active bit: In the case of `"00000100"`, bit number two (counted from the right, and stating from 0) is active, so the 8-to-3 encoder would return `"010"`.

If the input vector cannot be guaranteed to be one-hot, the encoder can have an additional 1-bit output that returns 1 if the input is a valid one-hot signal, and 0 if it is not. See figure 2.2 for an example of an 8-to-3 encoder.

Seven-segment digit

The Nexys2 has a seven-segment display with four digits. Each digit consists of eight LEDs (see figure 2.3a): The usual seven elements in a figure-eight pattern, and an additional decimal point.

The 32 LEDs of the seven-segment display are connected to the FPGA with twelve wires. The wires going to the cathodes of the LEDs are called CA through CG. CA is connected to the cathode of the top LED of *all digits*, and so on. The wires going to the anodes of the LEDs are called A0 through A3. A0 is connected to the anodes of *all LEDs* of the rightmost digit, and so on. The signals use inverted logic, so in order

Listing 2.1: VHDL model for a 4-to-1 multiplexer

```
library ieee;
use ieee.std_logic_1164.all;

entity mux is
    port(
        I   : in  std_logic_vector(3 downto 0);
        SEL : in  std_logic_vector(1 downto 0);
        O   : out std_logic
    );
end mux;

architecture behavioral of mux is
begin
    with SEL select O <=
        I(0) when "00",
        I(1) when "01",
        I(2) when "10",
        I(3) when "11",
        '0'  when others;
end behavioral;
```

to light up only the center LED of the second digit, CG and A1 must be driven to 0, all other signals to 1.

This implies that it is not possible to drive the different digits with different patterns at the same time. If the cathode wires are set to a fixed value and the anodes of two digits are active, then the same pattern will appear on both digits. In order to display, say, a decimal number with four digits, the individual digits must be driven alternatingly. The switching must be done so fast that it becomes indiscernible to the eye.

In this lab, only a single digit will be used. A driver for the whole display will be programmed later during this course. In order to make the signals easier to handle in VHDL, it is suggested to combine the cathode and anode signals into two vectors, as shown in figure 2.3b. For a suggestion on how to represent numbers and characters, refer to appendix B.

2.3 Basics — VHDL

Selected assignment

Complicated combinatorial circuits, like multiplexers and encoders, are often difficult to write in terms of basic logic functions. For a multiplexer, it is much more intuitive to express the output bit using a conditional expression: Based on the value of SEL, assign to the output one of several input bits. In VHDL such an expression is possible with *selected assignments*.

This is illustrated in listing 2.1, where a 4-to-1 multiplexer is described. Following the **`with`** keyword, an input port or signal (in this case `SEL`) is checked for different

Listing 2.2: VHDL model for a 8-to-3 encoder

```
library ieee;
use ieee.std_logic_1164.all;

entity encoder is
    port (
        I     : in  std_logic_vector(7 downto 0);
        O     : out std_logic_vector(2 downto 0);
        VALID : out std_logic
    );
end encoder;

architecture behavioral of encoder is
begin
    O <= "000" when I = "00000001" else
         "001" when I = "00000010" else
         "010" when I = "00000100" else
         "011" when I = "00001000" else
         "100" when I = "00010000" else
         "101" when I = "00100000" else
         "110" when I = "01000000" else
         "111" when I = "10000000" else
         "000";

    with I select VALID <=
        '1' when "00000001" | "00000010" | "00000100" | "00001000"
               | "00010000" | "00100000" | "01000000" | "10000000",
        '0' when others;
end behavioral;
```

values. Based on the value of the selecting signal, an assignment is made to an output port or signal that follows the **`select`** keyword (in this case `O`). After the `<=` operator follows the list that states which value is assigned to `O` for each value of `SEL`, using the **`when`** keyword.

A final **`when others`** clause covers all values of the selecting signal not explicitly mentioned before. It is mandatory that either *all possible values* of the selecting signal are covered or a **`when others`** option is used. For synthesis, you should *always* use **`when others`** after the explicit assignments[1].

Conditional assignment

Selected assignments are mostly useful for describing multiplexer-like circuits: cases where an output depends directly on the value of one input. More complicated conditions can be modeled with *conditional assignments*.

[1]The reason is that signals and ports with the type `std_logic` can have other values than `'0'` and `'1'`. These are mostly used for simulations.

In listing 2.2 a conditional assignment is used to describe the output of a 8-to-3 encoder. (The `VALID` bit is modeled with an additional selected assignment, showing how several cases can be combined with the `|` operator.) A conditional assignment begins with a simple concurrent assignment with the `<=` operator, followed by the **`when`** keyword and a condition. The assignment is only made if the condition is evaluated as true. If the condition is followed by the **`else`** keyword, another assignment can follow, which may depend on another condition, and so on.

After a list of assignments with different condition, a final **`else`** with an unconditional assignment should follow. This "default case" is used if all previous conditions were evaluated as false, much like the **`when others`** clause in selected assignments. Omitting the default case may lead to unexpected synthesis results[2].

Note that conditions can be more complex than the ones in the example. They can depend on multiple inputs and contain logic equations: The assignment

```
C <= '1' when A = '1' and B = '1' else '0';
```

creates an AND gate (which, of course, could be described much shorter).

2.4 Lab Instructions

Exercise 1: Seven-segment digit: individual segments

1. Create an entity with the following ports:
   - an 8-bit input vector `SW`;
   - an 8-bit output vector `C`; and
   - a 4-bit output vector `A`.
2. Create constraints that assign `SW` to the switches and `C` and `A` to the cathode and anode wires, respectively, of the seven-segment display.
3. Connect `C` to `SW` directly and assign a fixed value to `A`, so that only the rightmost digit is driven.
4. Generate a bitstream and test the design on the FPGA.

Exercise 2: Seven-segment digit: hexadecimal number

1. Create an entity with the following ports:
   - an 4-bit input vector `I`; and
   - an 8-bit output vector `C`.
2. Write an architecture, so that `C` returns the seven-segment cathode signals for the hexadecimal representation (0 – F) of the 4-bit number `I`.
3. To test the new entity, copy the VHDL file from the previous exercise and reuse the constraints. Remove the direct connection from `SW` to `C`. Instantiate the new entity and connect its input to the lower four bits of `SW` and its output to `C`.
4. Generate a bitstream and test the design on the FPGA.

[2] If there is a case that is not covered by any previous condition, the assigned-to signal implicitly keeps its value. This can lead to the synthesis of a latch—the level-sensitive counterpart to a flip-flop—which is usually unintended.

Exercise 3: Seven-segment digit: ASCII character

1. Create an entity with the following ports:
   - an 8-bit input vector `I`; and
   - an 8-bit output vector `C`.
2. Repeat the remaining steps from the previous exercise, except that the output `C` of the new entity should now represent the ASCII character with the 8-bit value `I`. In the top entity, connect all eight bits of `SW` to `I`.

Exercise 4: Priority Encoder

A priority encoder works similar the simple encoder introduced in the *Basics* section of this lab, with one exception: There is only one invalid input: all bits are `0`. For all other inputs, the binary representation of the position of the *most significant* (i.e., leftmost) bit with the value `1` is returned. For example, an 8-to-3 priority encoder with the input `"01010000"` would return `"110"`, since the highest active bit has the index 6.

1. Create an entity with the following ports:
   - an 8-bit input vector `I`;
   - a 3-bit output vector `O`; and
   - a 1-bit output `VALID`.
2. Create a priority encoder.
3. Test the priority encoder on the FPGA by connecting `I` to the switches and `O` and `VALID` to four LEDs.

Exercise 5: Seven-segment digit: mode selection

1. Create an entity with the following ports:
   - an 8-bit input vector `SW`;
   - an 8-bit output vector `C`;
   - a 4-bit output vector `A`;
   - a 1-bit output `LED`; and
   - a 4-bit input vector `BTN`.
2. Create location constraints for the old ports as before, and assign `BTN` to the four push buttons on the Nexys2.
3. Instantiate your hex-display module, ASCII display module and priority encoder. Use the right two push buttons as “mode inputs”. Implement the following modes:

   `"00"` The segments of one of the seven-segment digits can be controlled with the switches directly.

"01" The binary number selected with the switches is displayed on one of the seven-segment digits in hexadecimal representation. If the number is greater than 15, a dash is displayed instead, and an error LED lights up.

"10" The ASCII character with the code selected with the switches is displayed on one of the seven-segment digits.

"11" The number of the highest active switch is displayed on one of the seven-segment digits in hexadecimal representation. (The switches are numbered 0 – 7, from right to left.) If all switches are off, a dash is displayed instead, and an error LED lights up.

The left two push buttons, interpreted as a two-bit binary number, select which of the four seven-segment digits is used: The righmost digit is number 0, the leftmost one number 3.

4. Generate a bitstream and test the design on the FPGA.

Lab 3

Sequential Programming

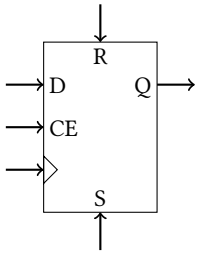
3.1 Learning Goals

- Digital electronics:
  - D flip-flop
  - Synchronous logic
- VHDL:
  - Clocked processes
  - The `if` statement
  - Generics
  - Numeric data types

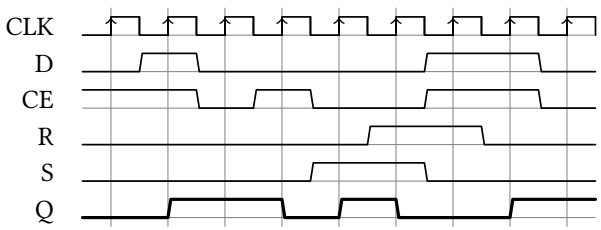
3.2 Basics — Digital Electronics

D flip-flop

For each LUT in the slices of the Spartan-3E there is one D flip-flop. Flip-flops can be used as storage elements that can hold 1 bit of information. Unlike logic gates and



(a) Black-box diagram, with an angle symbol for the clock (CLK) input



(b) Idealized timing behavior for a D flip-flop with example inputs. For easier readability, CLK was chosen as periodic, and rising edges of CLK are marked with vertical lines.

Figure 3.1: A D flip-flop

LUTs (so-called combinatorial circuits), the output of a flip-flop does not only depend on its current inputs, but also on an internal state that has two allowed values: 0 and 1. This internal state corresponds to the stored bit.

There are several types of flip-flops that differ mainly in the mechanism for the update of the stored value. The Spartan-3E contains *D flip-flops* with some additional optional inputs. There are two inputs and one output that are defining for a D flip-flop:

- The output Q always has the value of the currently stored bit.
- The input D is the *data input*: The stored bit will change to the value of D, but only under a condition:
- The input CLK (commonly marked with an angle symbol like in figure 3.1a) is the *clock input*. The value of D is captured only on a *rising edge* of this input; in other words, when CLK changes from a 0 to a 1, the stored bit will change to the value of D at that time.

There are three more optional inputs:

- The input CE is the *clock-enable input*: CE must be 1 during a rising edge of CLK in order for D to be captured. If CE is 0, the previous value is kept. If it is unused, CE must be fixed to 1.
- The R input performs a *synchronous reset*: When R is 1 during a rising edge of CLK, the stored bit changes to 0, no matter what the current value of D or CE is. If it is unused, R must be fixed to 0.
- The S input performs a *synchronous set*: When S is 1 during a rising edge of CLK, the stored bit changes to 1, no matter what the current value of D or CE is. R has a higher priority than S: If both are 1 during a clock edge, the stored bit is reset to 0. If it is unused, S must be fixed to 0.

Please look at the timing diagram in figure 3.1b and try to understand how the value of Q changes, at each rising clock edge, depending on these inputs.

Synchronous logic

Complex circuits like microprocessors can be thought of, on the so-called *register-transfer level* (RTL), as networks built out of registers and combinatorial logic. Registers, in this context, are storage elements for one or more bits that are controlled by a clock signal. They can be built out of one or more flip-flops. Combinatorial logic in FPGAs can be realized with LUTs.

The larger a circuit is, the more complicated it becomes to synchronize the data flow through the circuit. Take, for example, a data word consisting of 64 bits arriving at the input of a microprocessor. In order to store all bits in a register, all of the 64 flip-flops the register consists of should (ideally) store their respective bit at exactly the same time. If one of the flip-flops switches a bit too late, the word may have changed to a new value in the meantime, and now the flip-flops in the same register contain bits from different words! The same may happen between internal registers of a circuit: The output of one register passes through a block of combinatorial logic and to the input of another register. The combinatorial path for different bits can take different amounts of time. Still we must guarantee that the second register does not capture its input until all bits have settled to a new valid value.

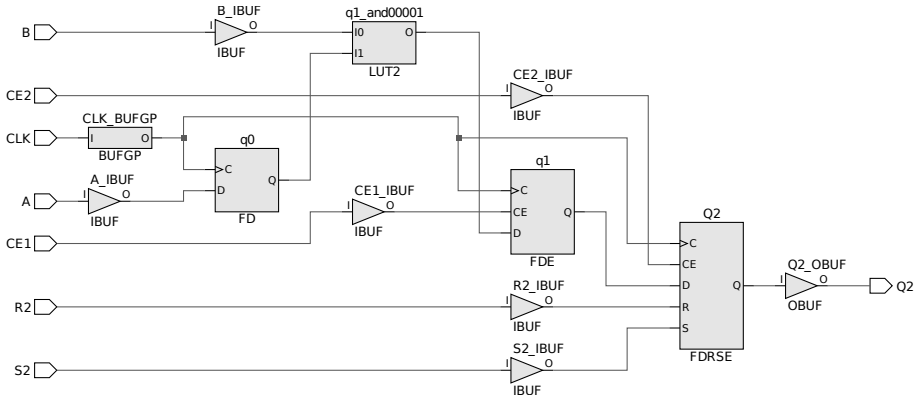


Figure 3.2: Netlist synthesized from the model in listing 3.1

A *clock signal* is used to perform this synchronization. It is distributed to all flip-flops in a synchronous circuit and optimized to keep the skew (i.e., the difference between arrival times at the different flip-flops) as small as possible. Clocks are usually periodic signals with a duty cycle of 50 %: They oscillate between equally long intervals of 0 and 1 (see, for example, the clock signal in figure 3.1b). Registers typically capture data at the rising clock edge (the transition from 0 to 1)[1].

The higher the oscillation frequency of the clock driving a circuit is, the faster the circuit operates (since data is transferred from one register to the next more frequently), but the shorter also the clock period is. Every circuit has a minimum clock period (or maximum frequency) that is dictated mostly by the longest combinatorial path between two registers: The output of the first register changes after one clock edge, and must pass through the subsequent logic and arrive at the input of the second register before the next clock edge occurs, one period later. If the clock runs too fast, invalid data is captured by the second register, and the circuit fails to operate.

3.3 Basics — VHDL

Clocked processes

The VHDL language elements introduced so far are only useful to describe combinatorial logic and infer (mainly) LUTs. To describe synchronous circuits we must be able to model edge-sensitive hardware elements. The VHDL model shown in listing 3.1 describes three D flip-flops. Figure 3.2 shows the netlist synthesized from this code.

A process in VHDL is a code block that, like the concurrent assignments introduced previously, can appear between the **`begin`** and **`end`** keywords of an architecture block. It begins with the **`process`** keyword, followed by a list of signal or port names in parentheses. This *sensitivity list* tells the synthesis and simulation tools on which signals the process triggers: The code in the process body is evaluated whenever one of the signals in the sensitivity list changes its value.

For now, we only want to use processes that infer synchronous logic. Synchronous circuit elements only change their output at a rising clock edge; therefore the sensitivity list of such processes should only contain the clock signal.

[1] Some circuits operate at the falling clock edge. *Double data rate* (DDR) circuits, in particular, capture data at both edges.

Listing 3.1: VHDL model of three flip-flops with different inputs

```
library ieee;
use ieee.std_logic_1164.all;

entity flip_flops is
    port(
        CLK : in  std_logic;
        A   : in  std_logic;
        B   : in  std_logic;
        CE1 : in  std_logic;
        CE2 : in  std_logic;
        R2  : in  std_logic;
        S2  : in  std_logic;
        Q2  : out std_logic
    );
end entity flip_flops;

architecture behavioral of flip_flops is
    signal q0, q1 : std_logic;
begin
    process(CLK)
    begin
        if rising_edge(CLK) then
            q0 <= A;

            if CE1 = '1' then
                q1 <= q0 and B;
            end if;
            -- Implicit: else q1 <= q1

            if R2 = '1' then
                Q2 <= '0';
            elsif S2 = '1' then
                Q2 <= '1';
            elsif CE = '1' then
                Q2 <= q1;
            end if;
            -- Implicit: else Q2 <= Q2
        end if;
    end process;
end architecture behavioral;
```

The process as a whole counts as a concurrent statement in an architecture block. The process body, in contrast, contains a list of *sequential statements*. These statements are processed *in order*, just like the statements in an imperative programming language. The two lines

```
X <= '1'; -- ignored, since the next line overrides the assignment
X <= A xor B;
```

are forbidden in an architecture block, where they count as contradictory concurrent statements. Inside of a process body, however, they count as sequential statements, so the second assignment overrides the first one.

The `if` statement

The **`if`** statement can be used to introduce a condition for the execution of a block of sequential statements. It is only available inside of sequential blocks, like process bodies.

Processes that describe synchronous logic always have the following form:

```
process(CLK)
begin
    if rising_edge(CLK) then
        ...
    end if;
end process;
```

The condition **`if`** `rising_edge(CLK)`**`then`** is necessary, in addition to the `CLK` in the sensitivity list, to tell the synthesis and simulation tools that the following block of statements describes edge-sensitive logic[2].

The first line inside of this edge-sensitive block in the example is the assignment `q0 <= A;` of the input port `A` to the signal `q0`. This line infers a simple flip-flop: On every rising clock edge, `q0` takes the value of `A`, so `q0` describes the output of a flip-flop whose clock input is connected to `CLK` and whose data input is connected to `A`.

The assignment for the second inferred flip-flop is linked to an additional condition: The assignment appears inside of a second **`if`** block, which is only executed if `CE1` is 1 (at the time of the rising clock edge). This condition corresponds to a clock-enable input, so a flip-flop with a CE input (an FDE) is inferred.

In addition, the value assigned to this flip-flop is not a simple input, but the result of a logic equation: `q1 <= q0` **`and`** `B;`. One of the inputs of this equation is the signal `q0`, to which an input was assigned earlier in the same process. This has two noteworthy consequences that you can confirm by looking at the inferred netlist in figure 3.2:

1. If the value assigned to a signal in a clocked process contains a logic equation, a flip-flop *and* the combinatorial logic (LUTs) connected to the data input of the flip-flop are inferred; and

2. Signals do *not* work like variables: Although the assignment `q0 <= A;` was made in the process before `q0` is used in `q1 <= q0` **`and`** `B;`, the second assign-

[2] Another common form that is often found instead of **`if`** `rising_edge(CLK)` is **`if`** `CLK'event` **`and`** `CLK = '1'`.

ment does *not* use the "new" value of `q0`: The value going into the input of the inferred LUT is not `A`, but the output of the flip-flop inferred for `q0`.

Signals used on the right side of an assignment inside of a process (or in an **`if`** condition, etc.) always represent the value they had *before* the execution of the process.

The third inferred flip-flop has additional conditions: Depending on the state of three inputs, it is set to 0, set to 1, or set to the output of the second flip-flop. This corresponds directly to the behavior of a reset, set, and clock-enable input, so a flip-flop with all available inputs (FDRSE) in inferred. Note that the order of the different **`if`** conditions is important: The input `R2` is checked first; only if it is 0, `S2` is checked, and so on. Therefore the reset input has a higher priority than the set input, which, in turn, has a higher priority than the clock-enable input.

The **`if`** conditions in the example have no explicit **`else`** case. If, for example, `CE1` is 0 at the rising clock edge, the process does not contain any assignment to `q1`. In such a case, it is implicitly assumed that `q1` keeps its previous value, just like a flip-flop with inactive clock-enable input should.

Generics

The VHDL model in listing 3.2 infers a counter with variable width: The output `COUNT` is a vector whose width is defined by a *generic* with the name `WIDTH`.

Generics, like ports, are defined in a special block at the definition of an entity. They define options for an entity, which can be selected individually for each instance. During instantiation, the value for an entity's generics can be defined with a **`generic map`**:

```
counter_1: entity counter
generic map (
    WIDTH => 16
)
port map (
    CLK      => CLK,
    INC      => INC,
    COUNT    => COUNT,
    OVERFLOW => OVERFLOW
);
```

The value given in the generic map is constant for this particular instance of the entity, but can be different for other instances. If a default value is supplied (like the 4 in the example), the generic map can be omitted. The default value is also used if the entity is used as the design's top module, instead of being instantiated.

Numeric vector data types

By using the package

```
use ieee.numeric_std.all;
```

you can create signals with the type `unsigned`. Like a `std_logic_vector`, `unsigned` is an array type and must be given a range using **`downto`**.

Listing 3.2: VHDL model for a counter with variable width

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity counter is
    generic (
        WIDTH : integer := 4
    );
    port (
        CLK      : in  std_logic;
        INC      : in  std_logic;
        COUNT    : out std_logic_vector(WIDTH-1 downto 0);
        OVERFLOW : out std_logic
    );
end entity counter;

architecture behavioral of counter is
    signal count_i : unsigned(WIDTH-1 downto 0);
begin
    process(CLK)
    begin
        if rising_edge(CLK) then
            OVERFLOW <= '0';
            if INC = '1' then
                count_i <= count_i + 1;
                if count_i = (2**WIDTH)-1 then
                    OVERFLOW <= '1';
                end if;
            end if;
        end if;
    end process;

    COUNT <= std_logic_vector(count_i);
end architecture behavioral;
```

For signals and ports that are declared as `unsigned`, arithmetic operations are defined. This is used in the example to infer a synchronous counter: In a clocked process, the value of the signal `count_i` of type `unsigned` in increased by 1 if the enable input `INC` is active.

Counters inferred this way flow over implicitly: If 1 is added to a 4-bit counter of type `unsigned` that is already at its maximum value `"1111"` (15), the counter flows over, and the next value is `"0000"` (0). The counter in the example includes an additional mechanism to detect such an overflow: The output port `OVERFLOW` is set to 0 at the beginning of the process (i.e., after every clock edge), but this assignment is overridden if the counter is increased while it is already at its maximum value. This condition is tested by comparing the count to the integer value `(2**WIDTH)-1`, where `**` is the exponentiation operator. If that is the case, `OVERFLOW` is set to 1, and will keep this value until the next clock edge, at which point it is reset back to 0.

For input and output ports, only the types `std_logic` and `std_logic_vector` should ever be used. In order to forward a signal of type `unsigned` to an output port, a type cast must be used like in the example.

Sometimes the width of the counter is initially unknown, but the maximum number that the counter should be able to hold is given, for example, by the generic `MAX`. In such a case, you can calculate the width $\lceil \log_2(\texttt{MAX}) \rceil$ by including

```
use ieee.math_real.all;
```

and declaring a constant in the architecture header[3]:

```
constant WIDTH : integer := integer(ceil(log2(real(MAX))));
```

In order to set the counter to a fixed integer value (for example, to reset it to 0 explicitly), a conversion function from `integer` to `unsigned` must be used. This requires the width of the `unsigned vector as a second argument`:

```
count <= to_unsigned(0, WIDTH);
```

3.4 Lab Instructions

Exercise 1: Timer

1. Create a VHDL entity with the following ports:

   - a 1-bit input `CLK`; and
   - a 1-bit output `T`.

   Give the entity two generics:

   - an integer `FREQ_IN`; and
   - an integer `FREQ_OUT`.

2. Write a timer module that uses the clock input `CLK` with the frequency `FREQ_IN` to produce an output `T`. `T` oscillates with the frequency `FREQ_OUT`, but stays

[3] This construct uses the `ceil` and `log2` functions defined for the data type `real`, which is why some type casts are necessary.

high for only *one period* of the input clock. Both frequencies are specified in Hertz.

Example: The input clock runs at 1 MHz, so its period is 1 µs. `FREQ_OUT` is specified to be 1 kHz; the according period would be 1 ms. In that case, `T` should go to 1 once per millisecond, go back to 0 after 1 µs, and stay 0 for the remaining 999 µs before going high again.

3. Create a VHDL entity that will be used to test your timer module. Give it the following ports:
   - a 1-bit input `CLK`; and
   - a 1-bit output `LED`.
4. In the new module, instantiate your timer and connect `CLK` of the top module to `CLK` of the timer. Set `FREQ_IN` to 50 MHz (the frequency of the crystal oscillator on the Nexys2) and `FREQ_OUT` to 2 Hz.
5. Create a clocked process with a signal that is assigned its own inverse at the rising edge of `CLK` if the timer output `T` is 1. (In other words: Infer a toggling flip-flop that uses `T` as its clock-enable input.)
6. Connect `LED` to the toggling signal.
7. Create a UCF file that assigns `LED` to an LED and `CLK` to the crystal oscillator on the Nexys2.
8. Create a bitstream and test the design on the FPGA.

Exercise 2: Instantiating a netlist file

For this exercise you will get the two files `display.ngc` and `display_wrapper.vhd` from the teaching assistant. The NGC file contains an already synthesized netlist. It converts a 14-bit binary number into the signals necessary to display it on a seven-segment display as a 4-digit decimal number.

1. Create an entity with the following ports:
   - a 1-bit input `CLK`;
   - an 8-bit input vector `SW`;
   - an 8-bit output vector `C`; and
   - a 4-bit output vector `A`.
2. Add `display.ngc` and `display_wrapper.vhd` to your project as sources.
3. Instantiate the entity in `display_wrapper.vhd` and connect its ports to the ports of your entity in the following way:

```
CLK                   →   CLK
BINARY(13 downto 8)   →   "000000"
BINARY( 7 downto 0)   →   SW
SEG                   →   C(6 downto 0)
DP                    →   C(7)
AN                    →   A
```

4. Create constraints that assign `CLK` to the 50 MHz oscillator, `SW` to the switches, and `C` and `A` to the seven-segment display's cathode and anode wires, according to figure 2.3b.

5. Create a bitstream and test the design on the FPGA.

Exercise 3: Time counter

1. Copy the VHDL file from the previous exercise and rename the entity. Make this you new top entity. Remove the input `SW` from the entity and the constraints.

2. Create a 14-bit counter with an enable signal and display its current count on the seven-segment display. Instantiate your timer and use it to make the counter run at 100 Hz.

3. Create a bitstream and test the design on the FPGA.

Exercise 4: Debouncing

1. Copy the VHDL file from the previous exercise and rename the entity. Make this you new top entity. Remove the timer. Add a 1-bit input `SW` and assign it to one of the push buttons.

2. Create a new entity and instantiate it in your top entity. Give it the following ports and connections:

   - a 1-bit input `CLK` (connect this to `CLK`);
   - a 1-bit input `I` (connect this to `SW`); and
   - a 1-bit output `EDGE` (connect this to a signal and use it as the enable input for the counter).

3. Write the architecture for the new submodule. It acts as an edge detector for the input `I` (i.e., the push button): If `I` was 0 in the previous clock cycle and is 1 now (this corresponds to the button being pushed down), `EDGE` goes to 1 for *one clock cycle*. Otherwise, `EDGE` is 0.

   Now, in theory, each time the push button is pressed, `EDGE` should go to 1 for one clock cycle, which would increase the counter by 1.

4. Create a bitstream and test the design on the FPGA. Press the button 20 times. How many counts are displayed?

5. Mechanical buttons have a tendency to bounce: When pressed or released, they oscillate between the on and off states for up to a few milliseconds. When sampled with 50 MHz, this results in the detection of multiple rising edges.

   Think of a way to reduce the effects of the bouncing, and change your module accordingly. Test your design by pressing the button 50 times. Exactly 50 counts should be displayed.

Lab 4

Driving a Seven-Segment Display

4.1 Learning Goals

- Digital electronics:
  - Binary-coded decimal
  - Double dabble algorithm
- VHDL:
  - Combinatorial processes
  - Variables
  - For loops

4.2 Basics — Digital Electronics

Binary-coded decimal

A single decimal digit (0 – 9) can be represented with a 4-bit binary number. For a decimal number with more than one digit, the *binary-coded decimal* (BCD) representation of the number can be obtained by concatenating the 4-bit representations of all digits. This is *not* the same as the decimal number's binary representation. See table 4.1 for some examples.

Double dabble

In order to display a number on a seven-segment display in decimal form, it must be converted from binary to BCD first, so that each digit can be displayed on one

Decimal	Binary	BCD
42	10 1010	0100 0010
100	110 0100	0001 0000 0000
1234	100 1101 0010	0001 0010 0011 0100

Table 4.1: Examples for the BCD representation of decimal numbers

BCD	Binary	Step	Operation
0000 0000	10 1010	1	Initialize
0000 0001	01 0100	2[1]b	Shift left
0000 0010	10 1000	2[2]b	Shift left
0000 0101	01 0000	2[3]b	Shift left
0000 1000	01 0000	2[4]a	Right BCD digit + 3
0001 0000	10 0000	2[4]b	Shift left
0010 0001	00 0000	2[5]b	Shift left
0100 0010	00 0000	2[6]b	Shift left

Table 4.2: Double dabble conversion of the number 42. The number has two digits ($m = 2$), and its binary representation has 6 bits ($n = 6$). The vector has $4 \times 2 + 6 = 14$ bits. Empty operations were omitted.

cell of the display. One possible way of doing this conversion is the "double dabble" algorithm.

The double dabble algorithm converts an n-bit binary number into a BCD number with m digits (i.e., $4m$ bits). It consists of the following steps:

1. Create a vector with $4m + n$ bits. Initialize the n bits on the right with the number you want to convert and the $4m$ bits on the left with zeros.
2. Perform the following steps n times:
   a) For each of the m BCD digits on the left of the vector, check if is is greater than 4. If it is, increment it by 3.
   b) Shift the content of the entire vector 1 bit to the left.

After the algorithm finishes, the $4m$ bits on the left will hold the BCD digits of the converted number. Table 4.2 shows an example of this algorithm in action.

4.3 Basics — VHDL

Combinatorial processes

In lab 3 you learned about processes, and how they can be used to model synchronous circuits. This was achieved by using the clock signal in the sensitivity list and testing for `rising_edge(CLK)`. Processes can, however, also be used to infer combinatorial logic. In that case, the check for the rising edge is omitted, and *all inputs* to the process—this includes all ports and signals that are on the right side of assignments and that are tested in **`if`** conditions—must appear in the sensitivity list.

Many applications that are complicated to model with logic equations and conditional assignments can be realized with combinatorial processes easily. Listing 4.1 shows an example for this: a circuit that counts the number of bits with the value 1 in an 8-bit vector.

For loops

For loops perform a block of sequential statements for a fixed number of iterations. A loop variable (`bit` in the example) is implicitly defined. Note that, when used in a

Listing 4.1: VHDL model for a circuit counting the active bits in an 8-bit vector

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity count_ones is
    port (
        I    : in  std_logic_vector(7 downto 0);
        ONES : out std_logic_vector(3 downto 0)
    );
end entity count_ones;

architecture behavioral of count_ones is
begin
    process (I) is
        variable one_cnt : unsigned(3 downto 0);
    begin
        one_cnt := "0000";
        for nbit in 7 downto 0 loop
            if I(nbit) = '1' then
                one_cnt := one_cnt + 1;
            end if;
        end loop;
        ONES <= std_logic_vector(one_cnt);
    end process;
end architecture behavioral;
```

combinatorial process, even for loops are synthesized into combinatorial logic. The netlist generated by the example code is the same that would be obtained with 8 single assignments for each bit.

Variables

Signals that are used as inputs in a process always return the value that they had *before* the execution of the process, even if their value was changed inside of the process before it is referred to. *Variables*, in contrast, show the opposite behavior.

Variables are declared locally in a process, before the **`begin`** keyword. They are assigned to with the `:=` operator. A variable that is referred to after being assigned a value will return the *new* value. Take, for example, the variable `v` declared as `unsigned(2 downto 0)`. After the statements

```
v := "000";
v := v + 1;
v := v + 1;
```

`v` will have the value `"010"` (2). This even works between iterations of a for loop. In the example this is used by looping over the bits of the input vector and, if a bit has the value 1, incrementing the variable `one_cnt`.

Since variables are only available in the process inside of which they were declared, their value must be passed to a signal or output port if it is to be used outside of the process. Variables retain their value between different executions of the same process.

4.4 Lab Instructions

Exercise 1: Two-byte hex display

1. Create an entity with the following ports:
   - a 1-bit input `CLK`;
   - an 8-bit input vector `SW`;
   - an 8-bit output vector `C`; and
   - a 4-bit output vector `A`.
2. Create constraints that assign these ports to the oscillator, switches, and seven-segment cathode and anode signals, respectively, of the Nexys2.
3. Instantiate your 4-bit hex-display module from lab 2 four times. (Each one will format the hex digit for one cell of the seven-segment display.) For two of the modules, connect the inputs `I` to four of the switches each. For the other two modules, connect `I` to constant values of your choice. The outputs of the modules will be connected in the next step.
4. Create a new entity and instantiate it in your top entity. Give it the following ports and connections:
   - a 1-bit input `CLK` (connect this to `CLK`);
   - an 8-bit input vector `C0` (connect this to the output `C` of the first hex-display module);
   - three more 8-bit input vectors `C1`, `C2`, and `C3` (connect these accordingly);
   - an 8-bit output vector `C` (connect this to `C` of the top module); and
   - a 4-bit output vector `A` (connect this to `A` of the top module).
5. Write the architecture for the new submodule. It acts as a timing multiplexer. It uses the clock to cycle through different values for `A`, so that at each time only one seven-segment cell is active (`"1110"`, `"1101"`, ...). At the same time, the cathode input (`C0`–`C3`) belonging to the currently activated digit is forwarded to `C`.
6. Create a bitstream and test the design on the FPGA. What do you notice?
7. Use your timer (or another method) to reduce the cycling frequency until you get a stable display. Which frequency gives a good result?

Exercise 2: Decimal display for a 14-bit number

1. Copy the top entity from the previous exercise and rename it. Remove the assignments of the switches and constant values to the inputs of the hex-display modules.

2. Create a new entity and instantiate it in your top entity. Give it the following ports and connections:
   - a 14-bit input vector `BIN` (connect the lower 8 bits to `SW` and the remaining bits to zeros);
   - a 16-bit output vector `BCD` (connect the four 4-bit slices of this to the inputs of the hex-display modules); and
   - a 1-bit output `VALID` (leave this unconnected for now).
3. Write the architecture for the new submodule. It acts as a binary-to-BCD converter.

   A 4-digit decimal number can have up to 14 bits. Its BCD representation has 16 bits. Create a combinatorial process that converts `BIN` to `BCD` using the double dabble algorithm (or come up with a method of your own). The output `VALID` is 0 if the input number is greater than 9999 (in which case four decimal digits cannot give a correct representation). Otherwise it is 1.
4. Create a bitstream and test the design on the FPGA.
5. Add an "error message": If the converter output is invalid, display a message of your choice instead of the output from the timing multiplexer.

Exercise 3: Four-byte ASCII display

1. Similar to the first exercise, instantiate your 8-bit ASCII-display module from lab 2 four times. Connect the outputs to the timing multiplexer and the inputs to switches or constant values.
2. Create a bitstream and test the design on the FPGA.

Exercise 4 (∗): Multi-purpose display

1. Create a multi-purpose seven-segment driver that can be instantiated in later projects. Give it the following ports:
   - a 1-bit input `CLK`;
   - a 32-bit input vector `I`;
   - a 2-bit input vector `MODE`;
   - an 8-bit output vector `C`; and
   - a 4-bit output vector `A`.
2. Depending on the `MODE` input, the module does the following:

   `"00"` The 32 bits of `I` are interpreted as four 8-bit seven-segment cathode signals, and directly passed to the timing multiplexer.

   `"01"` The lowest 16 bits of `I` are displayed as four 4-bit hex digits.

   `"10"` The lowest 14 bits of `I` are displayed as four decimal digits, including an error message when the number is greater than 9999.

   `"11"` The 32 bits of `I` are displayed as four 8-bit ASCII characters.

Lab 5

Finite-State Machines

5.1 Learning Goals

- Digital electronics:
  - Finite-state machines
- VHDL:
  - Enumerated types
  - The **`case`** statement
  - Numeric data types
  - One-process state machine model

5.2 Basics — Digital Electronics

Finite-state machines

A finite-state machine (FSM) is characterized by a set of states, inputs, and outputs. The FSM starts out in one of the possible states. The next state is determined based on the current state and the inputs.

Models of FSMs are the subject of theoretical computer science. They can be realized, for example, as computer programs or sequential logic. If implemented as a synchronous electronic circuit, the transition from one state to the next usually happens at every clock edge.

The way the outputs are determined depends on the FSM type. The FSM you will program in this lab can be most closely described as a *Mealy machine*. The outputs of a Mealy machine depend on its current state and the inputs.

5.3 Basics — VHDL

Enumerated types

The VHDL model in listing 5.1 describes a simple FSM. It goes through three states, depending on the input from two buttons and a time counter, and lights one of three LEDs in each state.

Listing 5.1: VHDL model of an FSM

```
library ieee;
use ieee.std_logic_1164.all;

entity fsm is
    port(
        CLK   : in  std_logic;
        RESET : in  std_logic;
        BTN   : in  std_logic_vector(1 downto 0);
        LED   : out std_logic_vector(2 downto 0)
    );
end entity fsm;

architecture behavioral of fsm is
    type   state_type is (s1, s2, s3);
    signal state : state_type := s1;
    signal count : integer range 0 to 50e6-1 := 0;
begin
    process (CLK) is
    begin
        if rising_edge(CLK) then
            if RESET = '1' then
                LED   <= "000";
                count <= 0;
                state <= s1;
            else
                case state is
                    when s1 =>
                        LED <= "001";
                        if BTN(0) = '1' then
                            state <= s2;
                        end if;
                    when s2 =>
                        LED   <= "010";
                        count <= count + 1;
                        if count = 50e6-1 then
                            count <= 0;
                            state <= s3;
                        end if;
                    when s3 =>
                        LED <= "100";
                        if BTN(1) = '1' then
                            state <= s1;
                        end if;
                end case;
            end if;
        end if;
    end process;
end architecture behavioral;
```

The **type** keyword can be used to define custom types for signals and variables. For keeping track of the current state of a state machine, *enumerated types* are useful. A variable of an enumerated type can only take values from a defined set. In the example, the possible states are `s1`, `s2`, and `s3`. During synthesis, this will be translated into a state register with a certain number of bits, depending on the encoding. For an FSM with n states, the commonly used one-hot encoding creates an n-bit register for which only the bit belonging to the current state is 1.

The case statement

With the **case** statement, a signal in a process can be checked for different possible values. Depending on the actual value of a signal, one of several different blocks of statements is executed. In a **case** statement, *all possible values* of a signal must be accounted for (similar to the selected concurrent assignments). If there are any omitted possibilities, a final **when others** case must be present.

In an FSM, the **case** statement can be used to check which is the current state. Based on the state, the output and the next state are set. The example shown in listing 5.1 uses a single process to model the FSM. Other methods, using two or more processes, exist. They separate, for example, the output function from the next-state function, and combinatorial from registered assignments. Which method is used largely depends on preference and the application at hand.

The FSM in the example has three possible states:

- Initially, and after a reset, it goes into state `s1`, and the first LED is lit up. If the first button is pressed in this state, the FSM changes to state `s2` in the next clock cycle. Otherwise it stays in state `s1`.
- In state `s2`, the second LED is lit up. For each clock cycle the FSM remains in this state, the counter (initially 0) is incremented by 1, until it reaches 50 million. Then the FSM changes to state `s3`. For a 50 MHz clock, this means that the machine stays in `s2` for exactly one second.
- In state `s3`, the third LED is lit up. The FSM remains in this state `s3` until the second button is pressed. Then it goes back to state `s1`.

Note that there is no explicit assignment to `state` and `counter` for every case: In the states `s1` and `s3` the counter is not assigned a value, and in all states, `state` is only assigned under a condition: if a button is pushed, or the counter has a certain value. In such a case, the signals implicitly keep their current value: The FSM remains in the current state, and the counter stays at its current count.

Numeric data types

You have encountered numeric data types before: Values of the type `integer` appear in the range declaration of vectors, and you used them in generics and in comparisons with `unsigned` vectors, as in **if** `count_i > 9999` **then**.

The example in listing 5.1 shows a way to use a signal of type `integer`, declared with a fixed range, to infer a counter. In that case, the synthesis tool automatically determines the required signal width. For `integer` signals, assigning a fixed value—for example, to reset a counter to 0—is easier than for `unsigned` signals, since a type cast is not necessary. On the other hand, `unsigned` signals allow a more direct control over how the counter is synthesized.

Passing an `integer` signal to an output port of type `std_logic_vector` is a bit complicated, since this requires several type casts:

```
use ieee.numeric_std.all;
 ...
O : out std_logic_vector(7 downto 0);
 ...
signal count : integer range 0 to 255;
 ...
O <= std_logic_vector(to_unsigned(count, 8));
```

5.4 Lab Instructions

Exercise 1: Three-stage code lock

Create a finite-state machine for a code lock. Implement the following features step by step, and test the design on the FPGA in between. You will need your multi-purpose display. If you do not have a working implementation, you will get a netlist from the teaching assistant.

- Three 8-bit numbers must be entered to open the lock. An initial value is set for the code.
- Numbers are entered by selecting them with the switches and then pressing one of the buttons (`ENTER`). Use a debouncer, or you will advance several states per button push!
- The FSM starts in a state where the first number can be entered. The number currently selected by the switches is displayed on the seven-segment display, and the first LED is lit up. After pressing `ENTER`, the second number can be selected and the second LED is lit up, etc. The entered numbers are stored but *not* checked before the third number has been entered.
- At any time before the third number has been entered, a second button (`ABORT`) returns the user to the input for the first number.
- If, after the third number has been entered, the input code is wrong, an error message is displayed for a fixed time. This message contains the number of wrong inputs so far. Afterwards, the lock returns to the input for the first number.
- After the third wrong input, the lock goes into a permanent lock-down state. This state is indicated by a message on the display.
- After a correct input, the error count is reset, and the lock opens. This state is also indicated by an according message.
- From the open state, the lock can be closed by pressing `ABORT`. This causes a message to be displayed for a fixed time, before returning to the input for the first number.
- A third button (`RESET`) returns the lock to its initial state from any other state. (This should normally, of course, not be accessible by the user.)

(∗) Another option from the open state is to press a fourth button (`REPROGRAM`). This lets the user enter a new code. This works similar to the previous code input: The display shows the selected number, an LED shows the current position, and `ENTER` advances to the next position. An additional LED shows that the reprogramming mode is active. After the new code has been input, an affirmative message is shown for a fixed time; then the lock returns to the open state. `ABORT` also returns to the open state.

Lab 6

Pulse-Width Modulation

6.1 Learning Goals

- Digital electronics:
  - Pulse-width modulation
- VHDL:
  - Simulation test bench
  - The **`after`** statement
  - The **`wait for`** statement

6.2 Basics — Digital Electronics

Pulse-width modulation

Signals in a digital circuit can usually take only one of two defined values: 1 or 0. If the output of such a circuit is connected to, for instance, the anode of an LED, then current will either flow through the LED or not, and the LED will either light up or not[1].

Direct control over the LED's brightness level is possible by varying the anode voltage between different analog values. This can be achieved with a digital-to-analog converter; however, in many cases, the effect of different analog levels can be mimicked by varying the *average voltage* supplied to a load. This makes it possible to change, for example, the brightness of an LED or the loudness of a buzzer.

A simple form of *pulse-width modulation* (PWM) can be used to control the average output voltage of a digital signal. In this technique, a periodic signal (similar to a clock signal) is produced, and its *duty cycle*—i.e., the fraction of time, during one period, in which the signal is 1—is varied. Figure 6.1 shows an example of PWM signals with different duty cycles.

[1] Provided its cathode is connected, through an appropriate resistor, to the voltage level corresponding to a logic 0.

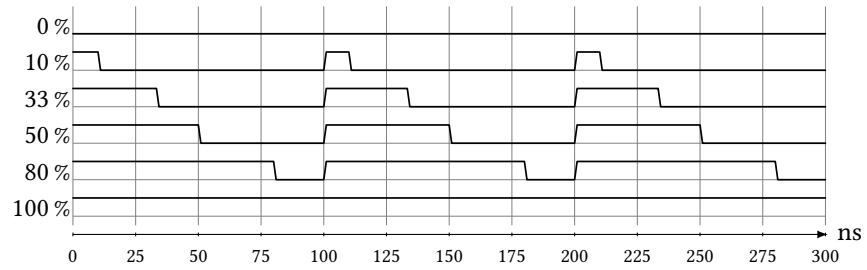


Figure 6.1: PWM signal with period 100 ns and varying duty cycles

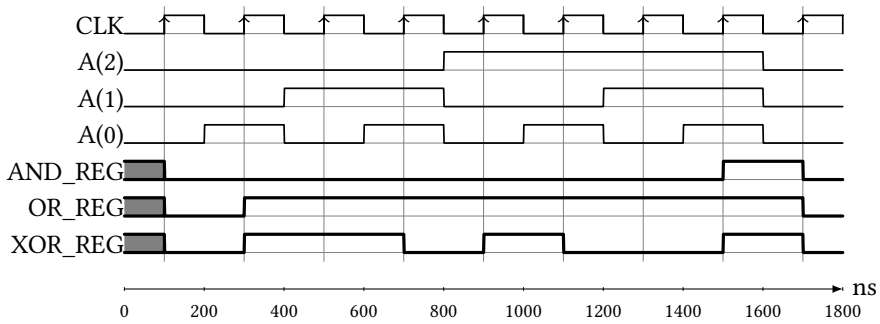


Figure 6.2: Simulated waveform for the example test bench

6.3 Basics — VHDL

Simulation test bench

The functionality of an FPGA design can be verified with a simulation of the HDL source code. A special computer program—the *simulator*—determines how the design behaves and which outputs it produces for predefined inputs. A simulation of a VHDL entity requires a *test bench* that specifies the *stimulus* (i.e., the way the inputs of the entity change over time) for the simulation.

The VHDL entity whose behavior is simulated is called the *unit under test* (UUT). The test bench is itself a VHDL entity and usually does not have any ports or generics. The UUT is instantiated inside of the test bench. In the test bench architecture, signals are declared for all ports of the UUT and connected accordingly in the port map. The signals assigned to the UUT's input ports can then be set to any desired values. Since the test bench is only used during simulation and never synthesized, non-synthesizable VHDL code may be used.

Listing 6.1 shows a VHDL entity that will be used as an example UUT. At each clock cycle, it outputs the results of the AND, OR, and XOR operations for three inputs. The example test bench in listing 6.2 instantiates this entity and creates a stimulus that goes through different input values. The example shows several techniques—all using non-synthesizable VHDL—for creating the stimulus for the inputs. The new keywords used will be explained below. Note that for a simulation, the VHDL library of the instantiated module must be supplied. This is `work`, unless explicitly changed to something else.

Listing 6.1: Example UUT

```
library ieee;
use ieee.std_logic_1164.all;

entity example_uut is
    port(
        A       : in  std_logic_vector(2 downto 0);
        CLK     : in  std_logic;
        AND_REG : out std_logic;
        OR_REG  : out std_logic;
        XOR_REG : out std_logic
    );
end entity example_uut;

architecture behavioral of example_uut is
begin
    process (CLK) is
    begin
        if rising_edge(CLK) then
            AND_REG <= A(0) and A(1) and A(2);
            OR_REG  <= A(0) or  A(1) or  A(2);
            XOR_REG <= A(0) xor A(1) xor A(2);
        end if;
    end process;
end architecture behavioral;
```

Figure 6.2 shows the waveform produced by the simulation of this example[2]. Note that, since the outputs are assigned in a clocked process, they change their value only the rising edges of `CLK` (corresponding to the behavior of a flip-flop, which the synthesis tool would produce). Before the first clock edge, the outputs are undefined. In order to avoid ambiguity, the test bench deliberately changes the inputs `A` only in-between clock edges.

This is a very basic example of how a test bench can be written. Using more advanced techniques, the outputs from the UUT can be tested against expected values, files can be used for input and output, information can be printed to the console, and much more.

Such a *behavioral simulation* is completely independent from the FPGA synthesis and implementation processes. The VHDL code itself, not the netlist or the finished design, is simulated. This can lead to discrepancies between the simulation result and the behavior of the actual hardware. For instance, a simple hardware description like in the example does not include component switching and signal propagation times. All assignments are made without a delay, so clocks and inputs could, in principle, change with any desired speed. With the 50 MHz clock in the example, the design will work in both simulation and hardware. If the clock were changed to 1 GHz, however, the simulation would not complain, but the hardware would stop working.

[2]Of course, it is possible to examine the simulation result in much detail with the simulator. For example, the value of all internal signals of the UUT and any of its submodules, at any time, can be seen.

Listing 6.2: Example test bench

```
library ieee;
use ieee.std_logic_1164.all;

entity example_tb is
end example_tb;

architecture tb of example_tb is
    signal A       : std_logic_vector(2 downto 0) := "000";
    signal CLK     : std_logic := '0';
    signal AND_REG : std_logic;
    signal OR_REG  : std_logic;
    signal XOR_REG : std_logic;

    constant CLK_PERIOD : time := 200 ns;
begin
    uut : entity work.example_uut
    port map(
        A       => A,
        CLK     => CLK,
        AND_REG => AND_REG,
        OR_REG  => OR_REG,
        XOR_REG => XOR_REG
    );

    CLK_PROC : process is
    begin
        CLK <= '0';
        wait for CLK_PERIOD / 2;
        CLK <= '1';
        wait for CLK_PERIOD / 2;
    end process CLK_PROC;

    A(0) <= not A(0) after 200 ns;

    A(1) <= '0',
            '1' after 400 ns,
            '0' after 800 ns,
            '1' after 1200 ns,
            '0' after 1600 ns;

    A2_PROC : process is
    begin
        wait for 800 ns;
        A(2) <= not A(2);
        wait for 800 ns;
        A(2) <= not A(2);
        wait;
    end process A2_PROC;
end architecture tb;
```

Another difference between simulation and hardware can arise from the implicit initialization of memory elements by the synthesis tool: A flip-flop in the FPGA must have an initial value that it takes immediately after the FPGA is programmed. This value is usually derived from the default value of the signal used to infer the flip-flop. If no default value is given, it is assumed to be `'0'`. The simulator makes no such assumption, and regards all signals as undefined until they are explicitly assigned a value.

Process sensitivity list can be an additional snag. The simulator only ever executes a process if one of the signals in the sensitivity list changes. Synthesis tools, on the other hand, usually produce the expected netlist even with an incorrect sensitivity list, and might only throw a warning. This can lead to a completely different behavior of simulation and netlist.

If a more precise simulation of the hardware behavior is desired, a *timing simulation* can be made, using the completely routed FPGA design. After the implementation step, the required FPGA components and their exact configuration, placement, and interconnection are known. The simulation is therefore not based on VHDL code, but on a structural description of the FPGA design, using models for the components, signal propagation times, etc. While a timing simulation is useful to get a realistic picture of the behavior of an FPGA design and analyze timing problems, it requires a finished design and takes a much longer time than a behavioral simulation. Furthermore, HDL signal names and hierarchical features can be lost during the implementation process, making the analysis of the design's inner workings much harder.

The `after` statement

In the example test bench, the stimuli for the UUT inputs `A(0)` and `A(1)` are created with concurrent assignments using the **`after`** statement. The **`after`** statement delays a concurrent assignment by a fixed amount of time. In case of `A(0)`, an oscillating signal is produced, because every time `A(0)` changes, it is scheduled to change again to the opposite value 200 ns later. For `A(1)`, assignments to fixed values are made at predefined times.

The `wait for` statement

The stimuli for the UUT inputs `CLK` and `A(2)` are produced using processes. (Note that these processes were given labels, that can help to identify signal drivers during simulation.) Both processes do not have a sensitivity list that tells the simulator when to execute them. Instead, the **`wait for`** statement is used: The processes are executed at the beginning of the simulation, and the simulator pauses them for a fixed amount of time when a **`wait for`** statement is encountered. If the end of the process is reached, it immediately starts again from the beginning, which is why `CLK_PROC` produces an oscillating signal. If a **`wait`** statement without time specification is encountered, the process stops indefinitely.

6.4 Lab Instructions

Exercise 1: LED dimmer

1. Create an entity with the following ports:

   - a 1-bit input `CLK`;
   - an 8-bit input vector `DC`; and
   - a 1-bit output `PWM`.

2. Write a PWM module with the following properties:

   - Using the clock input `CLK`, the module creates a pulse-width modulated signal `PWM`.
   - The PWM signal has a period of 255 (*not* 256) clock cycles.
   - The PWM signal's Its duty cycle is determined by `DC`. It can be regulated from 0 % to 100 %.

3. Create constraints that assign `CLK` to the 50 MHz oscillator, `DC` to the switches, and `PWM` to an LED on the Nexys2.

4. Create a bitstream and test the design on the FPGA.

5. Write a test bench that creates the following stimulus:

   - `CLK` is a 50 MHz clock; and
   - `DC` increases from 0 to 255, and stays at every value for two PWM periods.

6. Follow the instructions in appendix A.3 and run a behavioral simulation. Check that the PWM signal has the correct duty cycle, especially for 0 % and 100 %. How long does the simulator take (approximately) to simulate 1 s of real time for this design?

Exercise 2: Frequency generator

1. Create an entity with the following ports:

   - a 1-bit input `CLK`;
   - an 8-bit input vector `SW`; and
   - a 1-bit output `SPK`.

2. Create constraints that assign these ports to the oscillator, switches, and an extension header pin of the Nexys2.

3. Create a new entity and instantiate it in your top entity. Give it the following ports and connections:

   - a 1-bit input `CLK` (connect this to `CLK`);
   - a 16-bit input vector `FREQ` (connect the lower 8 bits to `SW` and the remaining bits to zeros); and
   - a 1-bit output `SND` (connect this to `SPK`).

4. Write a frequency generator module. Using the 50 MHz clock input `CLK`, the module generates a square wave `SND` with a duty cycle of 50 % and variable frequency `FREQ`.

5. Create a bitstream and test the design on the FPGA. Connect a speaker between the pin assigned to `SPK` and a ground level pin. What do you hear for the lowest frequencies?

6. Next, connect the upper 8 bits of `SW` to `FREQ` and the rest to ones. Test the design on the FPGA. What is the highest frequency you can hear?

7. Write a test bench that creates the following stimulus:

   - `CLK` is a 50 MHz clock; and
   - `FREQ` is 100 for 20 ms, 1000 for 2 ms, and 10000 for 200 µs.

8. Run a behavioral simulation. Which periods do you measure for the three selected frequencies?

Exercise 3: Melody

Instantiate your frequency generator and write a module that plays a short melody. Use a state machine or similar construct to go from one note to the next. Come up with your own melody, or use this one:

- 784 Hz for 130 ms;
- 740 Hz for 130 ms;
- 622 Hz for 130 ms;
- 440 Hz for 130 ms;
- 415 Hz for 130 ms;
- 659 Hz for 130 ms;
- 831 Hz for 130 ms;
- 1047 Hz for 390 ms.

Exercise 4: Loudness regulation for a fixed frequency

1. Copy your PWM module into a new file and rename the entity. Assign `PWM` to the speaker pin instead of an LED.

2. Change the PWM signal, so that it has a frequency of 440 Hz. The duty cycle should still be selectable from 0 % to 100 %.

3. Create a bitstream and test the design on the FPGA. Which duty cycle gives the loudest sound? What else do you notice when switching between duty cycles?

Exercise 5 (∗): Frequency generator with loudness regulation

1. Create an entity with the following ports:
   - a 1-bit input `CLK`;
   - an 8-bit input vector `DC`;
   - a 16-bit input vector `FREQ`; and
   - a 1-bit output `SPK`.
2. Create a module that allows you to select the frequency *and* duty cycle of the output signal.
3. Create a suitable top module and constraints. Create a bitstream and test the design on the FPGA.

Lab 7

UART

7.1 Learning Goals

- Digital electronics:
  - UART
  - RS-232
  - Oversampling
  - Shift register

7.2 Basics — Digital Electronics

UART

Data can be transmitted from one device to another using a *universal asynchronous receiver/transmitter* (UART). This type of transmission is *serial* (i.e., bitwise, on a single wire) and *asynchronous* (i.e., receiver and transmitted have separate clock sources).

For a two-way communication between two hosts (A and B), two data lines are necessary: On the A-B-line, host A sends out data on its TX (transmit) port and host B receives data on its RX (receive) port. On the B-A-line, host B sends out data on its TX port and host A receives data on its RX port. The electrical characteristics of the lines depend on the physical standard (usually RS-232).

Sender and receiver must use the same baud rate (i.e., bit rate), parity mode, and number of data bits and STOP bits. The most common baud rates are 9600 and 115200.

When idle, a UART line is in the logic high state. A single low bit (START bit) indicates the start of a transmission. Then 7 or 8 payload data bits are sent, with

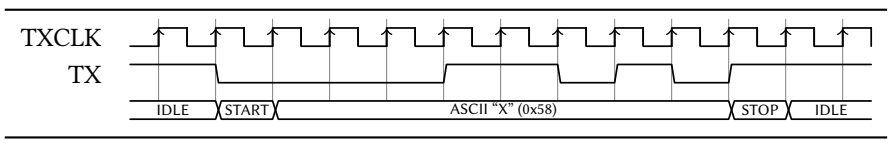


Figure 7.1: Timing diagram of a UART transmission of the ASCII letter X, without parity bit and with a single STOP bit

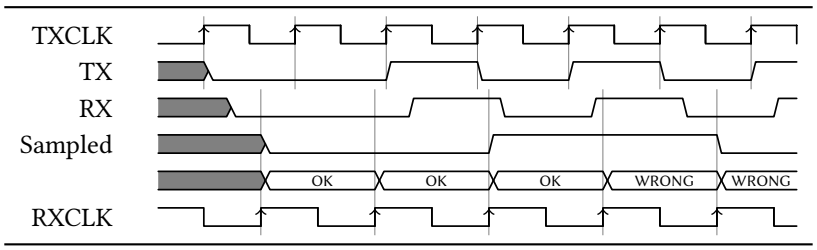


Figure 7.2: Effect of an asynchronous data transmission: The receiver clock runs 25 % slower than the sender clock. 0010101 is sent, but 00110… is received.

the LSB (least significant bit) sent first. Depending on the configuration, an even or odd parity bit can follow the payload data. A 1-bit or 2-bit logic high STOP signal terminates the transmission. Figure 7.1 shows an example UART transmission.

RS-232

RS-232 defines physical parameters for a serial communication. For a UART transmission using RS-232, at least three lines are necessary: In addition to the A-B-line and B-A-line, the ground of receiver and transmitter must be connected in order to obtain a common reference voltage.

In RS-232, the logic *high* state corresponds to a *negative* voltage (between −3 V and −15 V) and the logic *low* state corresponds to a *positive* voltage (between 3 V and 15 V). The conversion between these voltages and the FPGA I/O levels is handled by an additional chip on the board. Therefore, from the point of view of the FPGA, 1 and 0 can be used for logic high and low as usual.

When monitoring a UART transmission with a logic analyzer, it is important to probe the signals *before* the converter chip (i.e., *not* on the serial cable), since the RS-232 voltages can destroy the logic analyzer.

Oversampling

Asynchronous data transmissions lead to certain problems, because sender and receiver do not use a common clock source. As a result, the clocks are usually out of phase and can have a slightly different frequency. In addition, there is a delay of the data signal due to wire propagation times. There is a chance that the receiver samples the incoming signal at the exact moment of a rising or falling edge, which would lead to a metastable state of the sampling flip-flop.

The clock frequency difference additionally leads to a drift of the phase difference: In figure 7.2, the receiver clock is slightly slower than the transmitter clock. The receiver sees the START bit at *some* point during its low state. Then it samples the first two data bits correctly. However, because of the slower receiver clock, it misses the third data bit.

This effect can be eliminated by a technique called *oversampling*: Instead of sampling the incoming data with a frequency that is equal to the baud rate, a multiple of the baud rate is used. A common oversampling factor is 16: The receiver clock has 16 times the frequency of the transmitter clock.

If a receiver, oversampling with a factor of 16, detects a START condition—a logic low after an idle state—this means that it is at a point somewhere within the first 16th

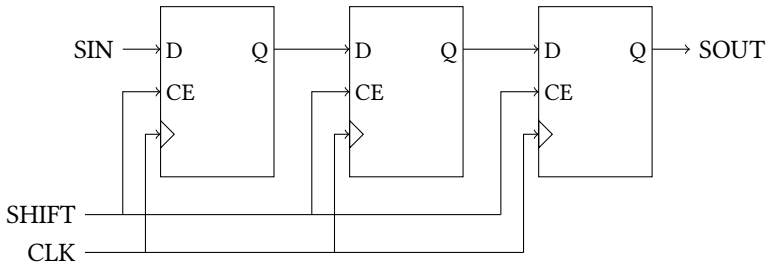


Figure 7.3: A 3-bit shift register

of the START bit. Waiting for 8 clock cycles then puts it approximately in the center of the START bit.

From there it is possible to advance to the approximate center of the next bit in steps of 16 clock cycles. If the frequency difference is not too big, it should be safe to sample the bit at this point. By repeatedly advancing 16 clock cycles, the remaining bits of the incoming byte can be sampled.

Shift register

An *n*-bit *shift register* can be built out of *n* D flip-flops. All flip-flops use the same clock, and the output Q of each flip-flop is connected to the data input D of the next. This is shown for three bits in figure 7.3.

At each clock edge, an incoming bit SIN is captured by the first flip-flop. At the same time, the saved bit of each flip-flop is shifted to the next. In addition, the clock-enable inputs of all flip-flops can be connected to a common SHIFT signal that controls in which clock cycles a shift occurs.

Shift registers can be used to convert data received on a serial input into parallel form: Each time a bit is received, it is shifted into an 8-bit shift register; after 8 received bits, the complete byte can be read out on the outputs of the flip-flops.

7.3 Lab Instructions

Exercise 1: Direct loop-back

1. Forward the serial RX input on the FPGA directly to the TX output. In addition, connect it to one of the extension I/O pins, so that you can monitor the signal with a logic analyzer.

2. Connect the serial port of the Nexys2 to the serial interface of a PC. Start a terminal emulator on the PC. On Linux, you can use:

   ```
   picocom -b 115200 /dev/ttyUSB0
   ```

   This sets the baud rate to 115200. The device name depends upon the port used, and might also be `/dev/ttyS0`, `/dev/ttyUSB1`, ...

3. Connect a logic analyzer to the chosen output pin and set it to trigger on a high-to-low transition (START bit). Send a few bytes from the PC to the Nexys2. Confirm that they are reuturned and check the output from the logic analyzer.

Exercise 2: UART sender

1. Create an entity with the following ports:
   - a 1-bit input `CLK`;
   - an 8-bit input vector `SW`;
   - a 2-bit input vector `BTN`;
   - a 1-bit output `LED`; and
   - a 1-bit output `TX`.
2. Create constraints that assign these ports to the oscillator, switches, two buttons, an LED, and the serial TX port of the Nexys2.
3. Create a new entity and instantiate it in your top entity. Give it the following ports and connections:
   - a 1-bit input `CLK` (connect this to `CLK`);
   - a 1-bit input `RST` (connect this to `BTN(1)`);
   - an 8-bit input vector `DIN` (connect this to `SW`);
   - a 1-bit input `SEND` (connect this to a debouncer sourced by `BTN(0)`);
   - a 1-bit output `IDLE` (connect this to `LED`); and
   - a 1-bit output `TX` (connect this to `TX`).
4. Write a UART sender module in the new entity. Instantiate inside of it a timer module and set the output frequency to 115 200 Hz. Use the timer signal in a state machine.

   The state machine starts out in an idle state (indicate this by driving `IDLE` to 1) and captures the byte on the data input `DIN` when `SEND` is 1 during a rising clock edge. It then sends out the captured byte on `TX` (8-bit data, no parity, 1-bit STOP). The state machine is reset if `RST` goes to 1.
5. Connect the board to a PC and run a terminal emulator that displays the received bytes.
6. Create a bitstream and test the design on the FPGA.

   What is the actual frequency produced by the timer module? Is the frequency difference a problem for the receiver? Check that the PC displays the correct bytes.

Exercise 3: Simple UART receiver

1. Create an entity with the following ports:
   - a 1-bit input `CLK`;
   - a 1-bit input `BTN`;
   - an 8-bit output vector `LED`;
   - an 8-bit output vector `C`;
   - a 4-bit output vector `A`; and

- a 1-bit input `RX`.

2. Create constraints that assign these ports to the oscillator, a button, the LEDs, the seven-segment cathode and anode signals, and the serial RX port of the Nexys2.

3. Create a new entity and instantiate it in your top entity. Give it the following ports and connections:

   - a 1-bit input `CLK` (connect this to `CLK`);
   - a 1-bit input `RST` (connect this to `BTN`);
   - an 8-bit output vector `DOUT` (connect this to a signal);
   - a 1-bit output `VALID` (connect this to a signal);
   - a 1-bit output `FERR` (connect this to a signal); and
   - a 1-bit input `RX` (connect this to `RX`).

4. Write a UART receiver module in the new entity. Instantiate inside of it a timer module and set the output frequency to 115 200 Hz. Use the timer signal in a state machine.

   The state machine checks the bits on the `RX` input when the timer signal is 1. When it sees a 0 (a START bit), it shifts the next 8 bits into a shift register[1].

   After the whole byte was received, the content of the shift register is output on `DOUT`. Check the STOP bit; if it is 0 instead of 1, this is a framing error. Indicate this by setting `FERR` to 1. Then indicate the end of the reception by setting `VALID` to 1 for *one* clock cycle.

   The state machine is reset if `RST` goes to 1.

5. Write a clocked process in the top entity. Inside the process, if `VALID` is 1, assign `DOUT` to `LED` and increment a framing error counter if `FERR` is 1. Display the number of framing errors on the seven-segment display.

6. Connect the board to a PC and run a terminal emulator.

7. Create a bitstream and test the design on the FPGA. Send 100 identical characters to the board. How often do you see the correct character on the LEDs, how often a wrong one? How many framing errros are counted?

Exercise 4: Oversampling UART receiver

1. Copy the entities from the previous exercise into new files and rename them.

2. Change the output frequency of the timer to 16×115 200 Hz. Change the state machine to oversample the received data.

3. Repeat the receiver test. How many incorrect bytes and how many framing errors do you count?

[1] In VHDL, a shift register can be realized with a simple `std_logic_vector` and assignments of the form `SREG(7 downto 0)<= SREG(6 downto 0)& SIN;`.

Exercise 5 (∗): RX display

Instead of the framing error count, show the last four received bytes on the seven-segment display in their ASCII representation.

Exercise 6 (∗): Additional UART modes

Add features to the UART transmitter and receiver, and test them in a suitable way.

- Add an input that chooses between 1-bit and 2-bit STOP. A transmitter with 2-bit STOP does not accept a new byte to send before it has sent two high STOP bits. A receiver with 2-bit STOP throws a framing error if any of the two STOP bits is low.
- Add an input that activates parity, and another one that chooses between even and odd parity. A transmitter with parity inserts a parity bit of the chosen type between the data and STOP bits. A receiver with parity throws a parity error (add an additional output for this) if the received parity bit has the wrong logic level.

Lab 8

Memory

8.1 Learning Goals

- VHDL:
  - Inferring memory
  - Arrays

8.2 Basics — VHDL

Inferring memory

The VHDL model in listing 8.1 describes a memory with the following properties:

- It is a 256×8 bit RAM: The *depth* is 256, i.e., the memory can store 256 elements. The *width* is 8, i.e., the stored elements and the data ports have a size of 8 bits. All stored elements can be addressed individually, so the address ports need $\log_2(256) = 8$ bits.
- It has independent write and read ports with a common clock.
- If `WEN` (write enable) is 1 during a clock edge, the byte on input `DIN` is written to address `WADD`.
- If `REN` (read enable) is 1 during a clock edge, the byte stored in address `RADD` is output to `DOUT`.
- The memory is in *read first mode*: If an address is written to and read from at the same time, the "old" value will be output on `DOUT`.

The example code is synthesized as Block RAM in the Spartan-3E.

Arrays

In the example, a new data type is declared for the signal that holds the memory content. The data type is an *array* of 256 8-bit vectors. Reading and writing individual elements of an array works with parentheses, just like addressing single bits in a `std_logic_vector`—in fact, `std_logic_vector` is an array of `std_logic`.

Listing 8.1: VHDL model for a 256×8 bit RAM

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ram is
    port (
        CLK  : in  std_logic;

        WEN  : in  std_logic;
        WADD : in  std_logic_vector(7 downto 0);
        DIN  : in  std_logic_vector(7 downto 0);

        REN  : in  std_logic;
        RADD : in  std_logic_vector(7 downto 0);
        DOUT : out std_logic_vector(7 downto 0)
    );
end entity ram;

architecture behavioral of ram is
    type ram_type is array (0 to 255) of std_logic_vector (7 downto 0);
    signal ram : ram_type;
begin
    process (CLK) is
    begin
        if rising_edge(CLK) then
            if WEN = '1' then
                ram(to_integer(unsigned(WADD))) <= DIN;
            end if;

            if REN = '1' then
                DOUT <= ram(to_integer(unsigned(RADD)));
            end if;
        end if;
    end process;
end architecture behavioral;
```

8.3 Lab Instructions

Exercise 1: Memory module

1. Create an entity with the following ports:
   - a 1-bit input `CLK`;
   - an 8-bit input vector `SW`;
   - a 1-bit input `BTN`;
   - an 8-bit output vector `C`;
   - a 4-bit output vector `A`; and
   - a 1-bit input `RX`.
2. Create constraints that assign these ports to the oscillator, the switches, a button, the seven-segment cathode and anode signals, and the serial RX port of the Nexys2.
3. Create a new entity and instantiate it in your top entity. Give it the following generics:
   - an integer `AWIDTH` (set this to 8); and
   - an integer `DWIDTH` (set this to 8).

   Give the new entity the following ports and connections:
   - a 1-bit input `CLK` (connect this to `CLK`);
   - a 1-bit input `WEN` (connect this to a signal);
   - an `AWIDTH`-bit input `WADD` (connect this to a signal);
   - a `DWIDTH`-bit input `DIN` (connect this to a signal);
   - a 1-bit input `REN` (connect this to a signal);
   - an `AWIDTH`-bit input `RADD` (connect this to a signal); and
   - a `DWIDTH`-bit output `DOUT` (connect this to a signal).
4. Write a memory module in the new entity. It has the same basic properties as the one in listing 8.1, but the data and address widths are configurable. The memory depth is as high as possible for the selected address width.
5. In the top entity, instantiate an oversampling UART receiver and a multi-purpose seven-segment display. Connect them to the memory module in the following way:
   - The first byte received over UART is stored at memory address 0, the next byte at address 1, and so on. After 256 bytes, start over at 0.
   - If the button is pushed, the byte from the address selected with the switches is read back from the memory and displayed on the seven-segment display.
6. Create a bitstream and test the design on the FPGA.

Exercise 2: LIFO

1. Create an entity with the following generics:

   - an integer `DEPTH`; and
   - an integer `DWIDTH`.

   Give the new entity the following ports:

   - a 1-bit input `CLK`;
   - a 1-bit input `WEN`;
   - a `DWIDTH`-bit input `DIN`;
   - a 1-bit output `DACK`;
   - a 1-bit output `FULL`;
   - a 1-bit output `OVERFLOW`;
   - a 1-bit input `REN`;
   - a `DWIDTH`-bit output `DOUT`;
   - a 1-bit output `DV`;
   - a 1-bit output `EMPTY`; and
   - a 1-bit output `UNDERFLOW`.

2. Write a *last in, first out* (LIFO) memory module in the new entity. This memory type is also known as a stack: The last element that was written to the memory is put on top of the stack, and it will be the first element that is returned on a read operation.

   Give your LIFO the following features:

   - `DEPTH` sets the maximum number of stored elements and `DWIDTH` the data width.
   - If `WEN` is 1 during a clock edge, the data on the `DIN` port is added to the stack and `DACK` goes to 1 for one clock cycle, but only if the stack is not full (indicated by the `FULL` flag). Writing to a full stack leads to the assertion of the `OVERFLOW` flag for one clock cycle.
   - If `REN` is 1 during a clock edge, the data on the top of the stack is output on the `DOUT` port and `DV` goes to 1 for one clock cycle, but only if the stack is not empty (indicated by the `EMPTY` flag). Reading from an empty stack leads to the assertion of the `UNDERFLOW` flag for one clock cycle.

   For the calculation of the address counter width, refer to the last part of section 3.3.

3. Verify the LIFO's function by creating a test bench and running a behavioral simulation. Include the following scenarios in the test bench:

   - reading from an empty stack;
   - writing to a full stack; and
   - simultaneous reading and writing, including cases where the stack is full or empty.

4. Copy the top module from the previous exercise, rename the entity, and replace the memory module with a LIFO: Bytes received over UART are put on top of the stack, and a button push gets the top element from the stack and displays it on the seven-segment display.

Exercise 3 (∗): FIFO

Copy the files from the previous exercise, rename the entities, and replace the LIFO with a *first in, first out* (FIFO) memory. This memory type is also known as a queue: The last element that was written to the memory is put at the end of the queue, and the element at the front of the queue will be the first element that is returned on a read operation.

A FIFO is a bit trickier than a LIFO: You will need to keep track of the position of the front and the end of the queue, and account for an overflow of the address counters in the correct way. Make the same tests (behavioral simulation and UART buffer) as for the LIFO.

Lab 9

Driving a VGA Monitor

9.1 Learning Goals

- Digital electronics:
  - VGA
- VHDL:
  - Memory initialization

9.2 Basics — Digital Electronics

VGA

The *Video Graphics Array* (VGA) connector on the Nexys2 allows the control of a monitor. VGA needs at least five analog signal wires, plus ground:

- The horizontal synchronization signal (HS) controls the propagation through the pixels in one line of the display. It goes through four phases:
  1. active (HS = 1);
  2. front porch (HS = 1);
  3. sync pulse (HS = 0); and
  4. back porch (HS = 1).

  The length of each phase depends on the video mode: the number of horizontal pixels, vertical pixels, the refresh rate, etc. For example, the active phase has a length of 640 clock cycles for a display mode with 640 horizontal pixels. One complete period of the HS signal corresponds to one displayed line.
- The vertical synchronization signal (VS) controls the propagation through the display lines. It goes through the same phases as HS, but it only advances for every finished period of the HS signal, so the phases are much longer. One complete period of the VS signal corresponds to one displayed frame.
- Three color signals (RED, GREEN, and BLUE) control the intensity of each color for the pixels. These are analog signals, but they can be varied between a few

fixed values with voltage dividers on the Nexys2. The color signals are valid when both HS and VS are in their active phases. They should be 0 if one of HS or VS is not active.

A nice description of how this works, and about VGA signaling in general, is given in the Nexys2 Board Reference Manual [2] in the section *VGA Port*. Please read this section as preparation for this lab.

9.3 Basics — VHDL

Memory initialization

The example in listing 9.1 shows how to initialize the contents of an inferred memory block from a file. It uses some advanced VHDL concepts, like the `std.textio` package and a function; you can simply copy this part and use it to initialize your memory module.

The file with the initialization data, called `init_file.dat` in the example, has the following format:

```
00000001
00000010
00000100
00001000
00010000
00100000
01000000
10000000
```

It must contain a line for *every* element in the memory.

9.4 Lab Instructions

Exercise 1: Solid color

1. Create an entity with the following ports:
   - a 1-bit input `CLK`;
   - an 8-bit input vector `SW`;
   - a 1-bit output `HS`;
   - a 1-bit output `VS`;
   - a 3-bit output vector `RED`;
   - a 3-bit output vector `GREEN`; and
   - a 2-bit output vector `BLUE`.
2. Create constraints that assign these ports to the oscillator, the switches, and the VGA signals of the Nexys2. (See the the Nexys2 Board Reference Manual [2] for the VGA pin associations.)

Listing 9.1: VHDL model for a 8×8 bit RAM initialized from a file

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;

entity ram_init is
    port (
        CLK  : in  std_logic;
        WEN  : in  std_logic;
        REN  : in  std_logic;
        WADD : in  std_logic_vector(2 downto 0);
        RADD : in  std_logic_vector(2 downto 0);
        DIN  : in  std_logic_vector(7 downto 0);
        DOUT : out std_logic_vector(7 downto 0)
    );
end ram_init;

architecture behavioral of ram_init is
    type   ram_type is array (0 to 7) of std_logic_vector (7 downto 0);

    impure function init_ram (data_file_name : in string) return
        ram_type is
        file     data_file : text is in data_file_name;
        variable data_line : line;
        variable ram_i     : ram_type;
        variable bit_word  : bit_vector(ram_i(0)'range);
    begin
        for nline in ram_type'range loop
            readline(data_file, data_line);
            read(data_line, bit_word);
            ram_i(nline) := to_stdlogicvector(bit_word);
        end loop;
        return ram_i;
    end function;

    signal ram : ram_type := init_ram("init_file.dat");
begin
    process (CLK)
    begin
        if (CLK'event and CLK = '1') then
            if WEN = '1' then
                ram(to_integer(unsigned(WADD))) <= DIN;
            end if;
            if REN = '1' then
                DOUT <= ram(to_integer(unsigned(RADD)));
            end if;
        end if;
    end process;
end behavioral;
```

3. Create a timer for generating the horizontal sync signal. Increment the timer at every other clock edge. Use the timings given in the Nexys2 for a resolution of 640×480 and a pixel clock of 25 MHz. Start with the active phase, followed by front porch, sync pulse, and back porch.
4. Create a similar timer for the vertical sync signal. Increment the timer at the onset of the horizontal sync pulse.
5. Outside of the active phases, set the colors to zero. During the display time, they are controlled by the board's switches.
6. Create a bitstream and test the design on the FPGA. If you run into problems, simulate the output from your entity.

Exercise 2: Patterns

Try to create the following patterns on the VGA display:

1. A stripes pattern. (*Hint*: Use the `mod` operator.)
2. A checkerboard pattern. (*Hint*: Use the `xor` operator.)
3. Circles around the display's center in alternating colors.
4. An "animated" pattern (e.g., a square or line moving through the display).
5. A pattern with a color rotation.

Exercise 3 (∗): Video memory

1. Create a submodule for a video memory. The memory has 480×640 bits that are initialized from a file. Each memory bit corresponds to one pixel, so the picture will be monochrome.
2. Get an initialization file with picture data from the teaching assistant. Instantiate the video memory and display its contents on the VGA display.

Exercise 4 (∗): Video data from UART

Fill the video memory with data received over UART. Send a picture from a PC to the FPGA.

Appendix A

Tutorial: Design Workflow with Xilinx Tools

A.1 Design Implementation with PlanAhead

This part of the tutorial will guide you through the steps necessary to create a new FPGA design for the Nexys2 using the Xilinx PlanAhead software and inspect the synthesis and implementation results. You need access to an installation of the Xilinx ISE Design Suite.

Creating a new design

1. Open a terminal. Type

   ```
   planAhead
   ```

   and press the enter key. A new PlanAhead window should open.

2. Select **File > New Project...**.

3. Click **Next**.

4. Select a name and location for the project directory. Make sure that **Create project subdirectory** is checked. Click **Next**.

5. Select **RTL project**. Click **Next**.

6. Set **Target language** to **VHDL**. Click **Next**.

7. Click **Next**.

8. Click **Next**.

9. Select the target FPGA from the part list. Use the available filters:

   - **Family: Spartan-3E**
   - **Package: fg320**
   - **Speed grade: -4**

The correct FPGA is the one with 9312 LUTs and flip-flops. (The exact device name is **xc3s500efg320-4**). Click **Next**.

10. Click **Finish**.

Adding design sources

1. Select **File > Add Sources...**.
2. Select the type of file you want to create:
   - For VHDL files, select **Add or Create Design Sources**.
   - For UCF files, select **Add or Create Constraints**.

   Click **Next**.
3. Select **Create File...**.
4. In the new window, if you are creating a VHDL file, set **File type** to **VHDL**. Choose a file name (without the extension). Set **File location** to **<Local to Project>**. Click **OK**.
5. Click **Finish**.
6. If you are creating a VHDL file, a wizard will appear. The wizard automatically creates entity and architecture blocks based on I/O port definitions. Since we will write these blocks ourselves, simply click **Cancel** and **Yes** here.

   The new files will appear in the **Project Manager** window under the **Sources** pane. VHDL files are in the **Design Sources** tree, UCF files are in the **Constraints** tree. You can open an editor by double-clicking on the file name.
7. Add your VHDL code and design constraints to the created files.

Synthesis and netlist schematic

1. Make sure that the correct VHDL file is selected as the top file for the project. Right-click on the correct file (in the **Sources** pane of the **Project Manager** window) and select **Set as Top**.
2. Make sure that the correct constraints are activated. By default, PlanAhead uses constraints from *all* available UCF files. Right-click on any UCF file you would not like to use and select **Disable File**. Select **Enable File** to reactivate a file.
3. Select **Flow > Run Synthesis**.
4. When the synthesis process is finished, a new window with the text "Synthesis successfully completed" should pop up. Select **Open Synthesized Design** and click **OK**. (The **Device** window with an overview of the FPGA die will open, but it does not contain the whole design yet, since the implementation processes have not been run.)
5. Click on the top element (marked with a capital N icon) in the **Netlist** pane of the **Synthesized Design** window, then select **Tools > Schematic**.

6. In the **Schematic** window you can explore the netlist of your design. For example, select a LUT by clicking on it in the **Schematic** window or in the **Primitives** tree of the **Netlist** pane, then open the **Truth Table** or **ROM Values** tab in the **Instance Properties** pane to view the LUT's truth table. You can save the netlist view by right-clicking somewhere in the **Schematic** window and selecting **Save as PDF File...**.

Implementation and bitstream generation

1. Make sure that the synthesis is complete and the correct constraints for your design are enabled.
2. Select **Flow > Run Implementation**.
3. When the implementation processes are finished, a new window with the text "Implementation successfully completed" should pop up. Select **Open Implemented Design** and click **OK**. (When prompted whether **Synthesized Design** should be closed, click **Yes**).
4. In the **Device** window, you can now explore how the elements from the netlist were placed in the FPGA. For example, open the **Nets** and **Primitives** subtrees in the **Netlist** pane by clicking on the circle symbol to their left; then press **Ctrl+A** to select all. You should get an impression from the **Device** window of how the signals propagate through the FPGA. (You may have to zoom in quite a lot.)
5. Select **Flow > Generate Bitstream**.

   (When you change something in a source file and want to create a new bitstream, you don't have to explicitly run synthesis and implementation first. You can simply run **Generate Bitstream**; PlanAhead will notice that the synthesis results are out-of-date and offer to automatically rerun synthesis and implementation before creating the bitstream.)

A.2 Bitstream Download with iMPACT

In this part of the tutorial you will learn how to set up the connection between the Nexys2 and you PC and download the generated bitstream file to the FPGA. It is assumed that you are using a Xilinx Platform Cable USB or Platform Cable USB II.

Hardware set-up

1. Connect the Platform Cable to your PC with a USB cord. The LED should light up amber.
2. Use the "squid cable" adapter to connect the Platform Cable to the JTAG[1] header on the Nexys2 board. See table A.1 for the correct mapping.
3. Connect the Nexys2 to your PC with a Mini-USB cable. (We use this connection only as a power supply.) Make sure the **POWER SELECT** jumper is set to **USB** and the **MODE** jumper is set to **JTAG**. Set the **POWER** switch to **ON**. The

[1] The *Joint Test Action Group* standard can be used to program, monitor, and debug integrated circuits.

Header pin	JTAG signal	Adapter wire color
1	TMS	green
2	TDI	white
3	TDO	purple
4	TCK	yellow
5	GND	black
6	VREF	red

Table A.1: Nexys2 JTAG header pinout

POWER LED on the Nexys2 should light up red, and the LED on the Platform Cable should turn green.

Bitstream download

1. In PlanAhead, make sure that the bitstream generation is complete.
2. Select **Flow > Launch iMPACT...**.
3. Choose a location and name for the iMPACT project file. Click **Save**.
4. Click **OK**.
5. If everything went right, you should see a *JTAG chain* in the **Boundary Scan** tab: Two icons with a Xilinx logo, representing devices on the Nexys2 that were detected by the Platform Cable.

   One of these devices is the FPGA. To download the bitstream to the FPGA, right-click on the symbol, choose **Program**, and click **OK**.
6. When the FPGA is programmed, the **DONE** LED on the Nexys2 will light up amber, and you can test your design. You can leave iMPACT open while making changes in PlanAhead, and simply reprogram the FPGA after generating a new bitstream. You only have to restart iMPACT after changing the top module of your project.

A.3 Simulation with ISim

This part of the tutorial will explain how to create a test bench for a behavioral simulation in PlanAhead, launch the ISim simulator, and examine the resulting waveform.

Creating a test bench file

1. In PlanAhead, select **File > Add Sources...**.
2. Select **Add or Create Simulation Sources**. Click **Next**.
3. Select **Create File...**.
4. In the new window, set **File type** to **VHDL**. Choose a file name (without the extension). Set **File location** to **<Local to Project>**. Click **OK**.

5. Click **Finish**.
6. To cancel the **Define Module** wizard, click **Cancel**, then **Yes**.
7. The new file will appear in the **Project Manager** window under the **Sources** pane in the **Simulation Sources** tree. Right-click on the new file and select **Set as Top**.
8. Add your VHDL test bench code to the created file.

Launching a behavioral simulation

1. In PlanAhead, select **Tools > Simulation > Run Behavioral Simulation ...**.
2. In the new window, make sure that **Simulation top module name** is set to the entity name of your test bench.
3. Click **Launch**.
4. If everything worked, a new window with the simulator ISim should open.

Examining simulation results in ISim

1. In ISim, you can find a hierarchical tree of your design in the **Instances and Processes** window. By selecting an instance, its ports and signals become available in the **Objects** window. You can drag any signal you would like to examine into the waveform window.
2. If not configured otherwise, 1 µs of real time is simulated when ISim is started. In the menu bar, you can set an amount of time for the simulation and click on the **Run** button to continue the simulation for that interval. By clicking **Run All**, the simulation is continued until it is interrupted by clicking on **Break**. **Restart** lets you start the simulation from the beginning, for example if a new signal was added to the waveform window.
3. In the waveform window, you can check the simulation results. The time interval between two events can be measured by placing the two cursors between them.

Appendix **B**

Seven-Segment Characters

Table B.1 lists several printable characters (alphanumeric and some punctuation) along with their ASCII values for upper and lower case. In addition, the d7seg font from the capbas font collection [1] is given as an example representation of these characters on a seven-segment display. The values for the C*X* and DP pins needed to display these characters on the seven-segment display of the Nexys2 are also listed. Note that, owing to the small number of elements in seven-segment displays, there is no distinction between upper case and lower case glyphs in the d7seg font; some of the glyphs resemble upper case letters, some lower case letters, and a few do not look very much like either one.

Table B.1: Printable characters with ASCII hex values and an example seven-segment representation

Character	ASCII hex value		d7seg font		
	Upper case	Lower case	Glyph	CG…CA	DP
!	21			1111101	0
,	2C			1111111	0
.	2E			1111111	0
0	30			1000000	1
1	31			1111001	1
2	32			0100100	1
3	33			0110000	1
4	34			0011001	1
5	35			0010010	1
6	36			0000010	1
7	37			1111000	1
8	38			0000000	1
9	39			0010000	1
?	3F			1111100	0
A	41	61		0001000	1
B	42	62		0000011	1
C	43	63		0100111	1

Continued on next page

Table B.1 — *Continued from previous page*

Character	ASCII hex value		d7seg font		
	Upper case	Lower case	Glyph	CG…CA	DP
D	44	64		0100001	1
E	45	65		0000110	1
F	46	66		0001110	1
G	47	67		1000010	1
H	48	68		0001001	1
I	49	69		1001111	1
J	4A	6A		1110001	1
K	4B	6B		0000101	1
L	4C	6C		1000111	1
M	4D	6D		0001010	1
N	4E	6E		1001000	1
O	4F	6F		0100011	1
P	50	70		0001100	1
Q	51	71		0011000	1
R	52	72		0101111	1
S	53	73		0010010	1
T	54	74		0000111	1
U	55	75		1000001	1
V	56	76		0001101	1
W	57	77		0000001	1
X	58	78		0101010	1
Y	59	79		1010001	1
Z	5A	7A		0100100	1

Bibliography

[1] Phons Bloemen. capbas — capital baseball “matrix printer” font collection. http://www.ctan.org/tex-archive/fonts/capbas.

[2] Digilent, Inc. Digilent Nexys2 board reference manual. https://digilentinc.com/Data/Products/NEXYS2/Nexys2_rm.pdf.

[3] Xilinx, Inc. Spartan-3E FPGA family data sheet. http://www.xilinx.com/support/documentation/data_sheets/ds312.pdf.