lea.schoenberger [☺] tu-dortmund.de
mikail.yayla [☺] tu-dortmund.de

Exercises for the lecture
Cyber-Physical Systems Fundamentals
Summer Term 2018

Exercise Sheet 3

(10 Points)

Lab exercises starting from Monday, 11th June 2018

The lab exercises take place at room OH16/U08. The exercise sheets will be solved during the exercise sessions.

For this exercise sheet, we use the virtual machine **ES-FPGA**.

Preparation

Please note: Unfortunately, the virtual machine's desktop resolution is not always configured properly. In case your desktop appears as a small, centered rectangle, you can set the resolution to 1920x1080 manually via *Applications* → *Settings Manager* → *Display* → *Resolution*.

After loggin on, you can access the archives containing the assignments as well as the neccessary data under *File System* → *cpsf*. The development environment ISE by Xilinx can be found via the system search (*Applications* → "RUN ISE").

1. Navigate to the folder `blatt03` on the server and extract the archive (click right,"Extract to...").
2. Start Xilinx ISE and open the project file `blatt3_1.xise` (*File* → *Open Project*).

The figure below portrays an open project in the ISE project navigator window. You can switch between synthesis (implementation) and simulation in the left column. Below, an overview about the project's module hierarchy is available. With a double click on this hierarchy, the respective source code is displayed in the editor window on the right side.

The available commands listed in the box under the hierarchy tree depends on which hierarchy entry is selected. In many cases, less frequently used commands can be shown by clicking on the "+" symbol on the left side of the command list. Status information can be found in the console window at the bottom and may be filtered with respect to error messages by clicking on the "Errors" tab.

Background

VHDL (VHSIC (Very High Speed Integrated Circuit) Hardware Description Language) has been developed to describe digital circuits for simulation purposes. Hence, modeling of timing behavior is one of the language's main features. However, concerning hardware synthesis, this timing behavior is completely ignored. An incomplete VHDL overview is given on the last page of this exercise sheet.

This exercise sheet focuses on both application areas of VHDL, simulation and hardware synthesis. We use a board with an FPGA (Field Programmable Gate Array) as target platform for the sythesis. This chip category has been designed for implementing arbitrary logical functions without having to modify the hardware. The synthesis software computes how to realize the respective logical functions by means of the particular elements forming the FPGA, i.e., how to wire them, and produces a so-called *Bitstream* file that configures the chip accordingly. Details regarding this topic can be found, e.g., in the slides of the lecture *Synthesis of Embedded Systems* that has been offered in WS 2012/13.

3.1 VHDL as simulation language (7 Points)

In the lecture, a VHDL description of a full adder has been presented which is given below in a slightly modified version and may be used in order to solve the second assignment:

```
1  entity full_adder is
2    port (
3      a, b, carry_in : in  std_logic;
4      sum,  carry_out: out std_logic
5    );
6  end full_adder;
7
8  architecture behavior of full_adder is
9  begin
10   sum       <= (a xor b) xor carry_in after 10 ns;
11   carry_out <= (a and b)        or
12                (a and carry_in) or
13                (b and carry_in)       after 10 ns;
14 end behavior;
```

This adder is provided in the file `full_adder.vhd` and will be a component of this assignment's solution. Furthermore, a so-called *Testbench* can be found in the given data, which supplies an instance of your 4-Bit-Adder with input data and verifies if the retrieved result is correct.

In-line tutorial for the simulator

First, the simulator will be demonstrated by means of an incomplete implementation:

3. If necessary, switch to the simulation view (*View: Simulation*) using the buttons above the hierarchy box.
4. Choose the hierarchy entry `adder_test` to display the possible commands for this entry (you can open the file by double clicking, but this is not necessary here). Wählen Sie in der Hierarchie den Eintrag `adder_test`
5. Double click on *Simulate Behavioral Model* in the command list (you may need to expand the *ISim Simulator* by clicking on the plus symbol).

This will start the simulator ISim, execute the simulation for a suffient amount of time and show the results. ISim always shifts the waveforms to the right so that the initial simulation result is not visible, but, however, this problem can be solved by making use of the scroll bar below the waveform window.

Figure 1 shows the application window of ISim. By means of the provided material's settings, the relevant signals will be shown automatically and the simulation will be executed suffiently long. Regarding the signals *b*, *sum* und *sum_want*, the number of signal transitions exceeds the width of the waveform window. This can be adjusted via the zoom icons in the tool bar or pressing F7 or F8, respectively.

The signal `match` indicates if the output value of the 4-bit adder complies with the expected value. At the moment, this signal's value is 0 at any point, due to the fact that the 4-bit adder is not yet implemented and thus the expected value cannot be obtained. More precisely, both output signals of the adder (`sum` and `carry_out`) exhibit undefined values [1] (cf. line in the middle of the row) – this is visible for `sum`, if the display detail is sufficiently enlarged.

Pleas note: Please close ISim before modifying the VHDL files.

6. Close ISim (*File* → *Exit*).
7. Expand the entries in the hierarchy browser until you find the component `adder_4bit`. Open this one with a double click.

[1] in VHDL, `U` is the default value for signals that have not been initialized

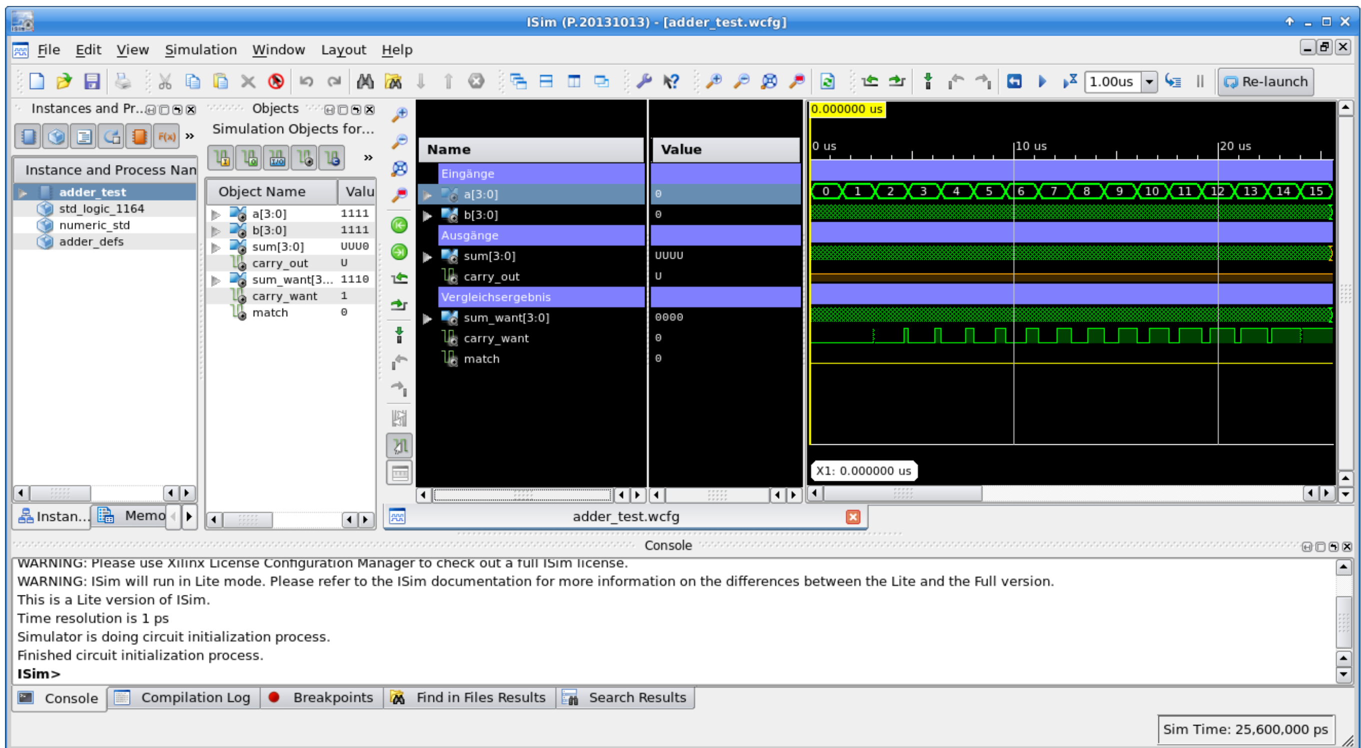

Figure 1: ISim

Assignment

Implement a 4-bit adder in `adder_4bit.vhd` with the help of the provided full adder (`full_adder.vhd`). The implementation for the lowest bit as well as the definitions of all necessary internal signals are included in the given material.

Please note: Before simulating your solution, make sure you chose the entry `adder_test` in the hierarchy box – otherwise the wrong file will be simulated.

Question: Does the test bench recognize a possible correct behaviour of your adder at any point of time? If it does, why? Otherwise, why not?

3.2 VHDL as hardware description language (3 Points)

In this exercise, you will synthesize your 4-bit adder for the execution on a FPGA. Figure 2 shows the FPGA board we will use. The on/off switch is located on the upper left side. Close to the lower edge, several switches as well as LEDs can be found. Four seven-segment displays are situated over a number of buttions on the right side.

8. In ISE, switch to the synthesis view *Implementation* using the buttons over the hierarchy box.
9. Open the file `adder_nexys` with a double click.
10. Click on *Configure Target Device* in the command list and wait.

After a double click, the development environment automatically executes all necessary steps from the synthesis until the actual programming of the FPGA board. During the most steps, an animation is shown in front of the respective entry in the command list, but, however, not during *Configure Target Device*. Warnings (yellow triangles containing an exclamation mark) are not problematic[2], red error marks should not yet occur (otherwise they can be found in the console in the lower window part).

[2]it is not easy to create projects in Xilinx ISE that do not generate warnings

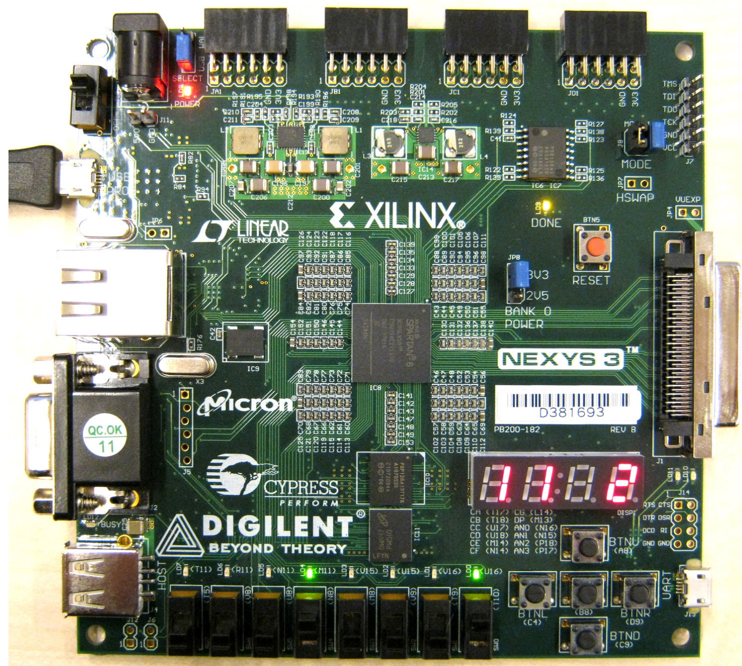

Figure 2: Digilent Nexys 3

As described in the file `adder_nexys.vhd`, an instance of your 4-bit adder is generated, its inputs are connected to the switches at the lower edge of the board (the left four switches are connected to input A, the right four switches to B, switch up = 1), and the resulut is displayed as a hexadecimal number on the seven-segment display.

General information: An overview about the exercise sessions as well as further information can be found on `https://ls12-www.cs.tu-dortmund.de/daes/en/lehre/courses/sommersemester-2018/cyber-physical-system-fundamentals-ss-2018.html`. The exercise sheets will usually be published on the course website on Mondays and will be solved during the respective exercise sessions. The exercises are divided into two parts, in each of which at least 50% of the points must be achieved in order to receive the exam admission.

VHDL overview

```
1  -- comments are introduced by two minus signs and end at the line end
2
3  -- an entity describes how the interface of the current mode looks like
4  -- (more commonly: a file in VHDL is a module)
5  entity Example_Entity is   -- VHDL is not case sensitive
6    port (
7      Foo : in  std_logic;                      -- input called "Foo", single bit
8      Bar : in  std_logic_vector(7 downto 0); -- input called "Bar", 8 bit array
9      Baz : out std_logic;                      -- output called "Baz", single bit
10     Quux: out std_logic_vector(7 downto 0)  -- output called "Quux", 8 bit array
11   );
12 end Example_Entity;  -- end of entity block
13
14 -- an architecture describes what the current module actually does
15 architecture Behavioral of Example_Entity is -- "Behavioral" can be chosen arbitrarily
16   -- (traditionally, the architecture is named "Behavioral")
17
18   -- at this point, internal signals are declared , comparable to
19   -- local variables in Java functions
20   signal int_sig_1: std_logic;
21   signal int_sig_2: std_logic_vector(7 downto 0);
22
23   -- signals can be initialized at the point of their declaration
24   signal demo_bit   : std_logic := '0';                        -- use single bits ''
25   signal demo_array: std_logic_vector(3 downto 0) := "0101"; -- and use arrays ""
26
27 begin
28   -- the actual description of the behavior begins after "begin"
29   -- everything written here will be "executed" at the same time
30
31   -- it is, e.g., possible to create instances of other modules:
32   Example_Entity: SignalGen port map ( -- creates an instance of "SignalGen" and calls
33                                        -- it "Example_Entity". The name is usually
34                                        -- not important, but each instance must have
35                                        -- a unique name.
36     -- the port map describes how inputs and outputs of the module are "wired":
37     SigGenInput1  => foo,         -- connect "SigGenInput1" with "foo"
38     SigGenInput2  => bar,         -- etc.
39     SigGenInput3  => '0',         -- constants can also be used, single bits use ''
40     SigGenInput4  => "10101010", -- ...and arrays use ""
41     SigGenOutput1 => int_sig_1,  -- (the arrows' direction is the same for inputs and outputs!)
42     SigGenOutput2 => int_sig_2   -- no comma after the last entry!
43   );
44   -- The names of the inputs and outputs of the module that is instatiated here can be looked up in
45   -- its entity. It holds: Each input must receive a value.
46
47   -- signal assignments start and end at everything that is given in the entity or that was
48   -- declared as local signal:
49   Baz <= not int_sig_1; -- Baz receives the inverse value of int_sig_1
50   Quux(3 downto 0) <= int_sig_2(7 downto 4); -- access parts of an array by means of braces
51   Quux(4) <= int_sig_2(3) and int_sig_1;      -- single elements of an array, and-operator
52   Quux(5) <= int_sig_2(2) or  int_sig_2(1);   -- or-operator
53   Quux(6) <= int_sig_2(1) xor '1';            -- xor-operator
54   demo_array <= "11" & "0" & demo_bit;        -- & concatenates bits/arrays: the result here
55                                                -- is "110_", with the result of demo_bit as _
56
57   -- conditional signal assignments are also possible, here a rather complicated example:
58   -- (complicated, because "Quux(7) <= not int_sig_1" would lead to the same result)
59   demo_bit <= '1' when int_sig_1 = '0' else '0';
60
61   -- (processes could also be listet here, but those are not necessary for this exercise sheet)
62 end Behavioral; -- also the architecture has to be concluded with an "end"
```