Lab Exercise 2
VHDL Programming
Functions and Procedures/Libraries
Control of seven segments

INF3430/4431 Autumn 2017

Version 1.6/24.09.2017

Introduction

The goals of this lab exercise are:

- Learning about subprograms and libraries in VHDL
- Learning how to create *hardware* descriptions in VHDL, and to create test benches to simulate them.

This lab exercise is divided into a number of partial exercises. The first part of the exercise is to create subprograms and combine them in a library. The next parts of the exercise concern designing code aimed at the test board. They are organized with increasing levels of difficulty. For each part of the exercise you must follow the same design flow as in lab exercise 1.

Exercise 1

In this part of the exercise we will look at subprograms (`function` and `procedure`).

a)

Simulate the attached code in `pargen.vhd` and `tb_pargen.vhd`, where two 16-bit vectors are read in and a parity signal is generated. Both the method that is used in the textbook (see Zwo, Chapter 4) and the more common `XOR`-based method are used in the code. Compare these two methods by synthesizing `pargen.vhd` and looking at the results with RTL and the Technology views in Vivado. Is there any difference in the consumption of LUTs? Which method has the shortest delay?

b)

Modify the code in `pargen.vhd` now to use two functions with the same name (use overloading of functions in VHDL) to generate parity for each of the two input data signals.

c)

Modify the code in `pargen.vhd` from part a of the exercise now to use a procedure with the same name as the name selected in part b of the exercise (use overloading of functions in VHDL) to generate parity in a procedure.

d)

Move the functions in part b of the exercise to a *`subprog_pck`* package. Modify `pargen.vhd` from part b of the exercise to use the functions from the *`subprog_pck`* package.

e)

Also move the procedure in part c of the exercise to *`subprog_pck`* package. Modify `pargen.vhd` from part c of the exercise to use the procedure from the *`subprog_pck`* package.

Approval:

VHDL source file for the individual questions.

Procedure for VHDL coding

The creation of a *hardware* description in VHDL can be divided into four main elements:

- Try to form a correct picture of the problem.
- Read the exercise text carefully.
- Study the documentation for the test board.
- Create a block diagram for the inputs and outputs to be created for each module.

Structuring of a solution

The construction you will be implementing in the FPGA has an external interface to the other components on the test board and an internal structure. The external interface, in the form of input and output signals, corresponds to the `entity` declaration in a VHDL description. The internal structure corresponds to the `architecture` part of the VHDL description. It is often appropriate to divide the internal structure into a *data path* structure and a *control* structure.

The data path structure contains elements such as registers, adders, multiplexers that are connected to data buses. This structure is well-suited for description by a block diagram. If the structure is complex, it is a good idea to divide it up into smaller blocks.

The control logic's input and output signals can be described by a block diagram together with the data path structure. The internal structure is described best in the form of: truth value tables, Boolean equations and state diagrams. It is a good idea to perform parts of the work to structure the construction by making use of the option to enter comments in the VHDL source file.

Code the structured solution in VHDL

With a well-documented structure as the starting point, it is normally an easy job to create functioning code.

Naming rules for VHDL design files

To identify VHDL source files, it is a good idea to have certain naming conventions and rules for what the various files should contain. In all of the designs you create starting with part 2 of the exercise, we will save the entity and architecture in different files and follow the naming rules given in the following table:

Table 1. Naming rules for VHDL source files

File content	File name
Entity	`<design_unit>_ent.vhd`
RTL architecture	`<design_unit>_rtl.vhd`
STR architecture	`<design_unit>_str.vhd`
Behavioural architecture (simulation model not intended for synthesis)	`<design_unit>_beh.vhd`
Configuration	`<design_unit>_cfg.vhd`
Package def	`<library_unit>_pkg.vhd`
Package body	`<library_unit>_bdy.vhd`
Test bench	`tb_<design_unit>.vhd`
Test bench config	`tb_<design_unit>_cfg.vhd`

`<design_unit>` and `<library_unit>` should be names that identify the function/content of the file (for example: `seg7ctrl_ent.vhd`, `seg7ctrl_rtl.vhd`, `tb_seg7ctrl.vhd`). It is common to have a small number of design packages, often just one, that is synthesizable (for example: `mydesign_pck.vhd` and `mydesign_bdy.vhd`). In addition, there are one or more test bench packages (for example: `tb_mydesign_pck.vhd` and `tb_mydesign_bdy.vhd`).

Test bench for VHDL simulation

The key question when designing a test bench is what test vectors must be generated for a complete simulation of the chip's behaviour. Depending on the complexity of the chip, this may be a very easy or a very difficult task. In many cases it is a good idea to start with a table with all the relevant input signal combinations and expected output signal values.

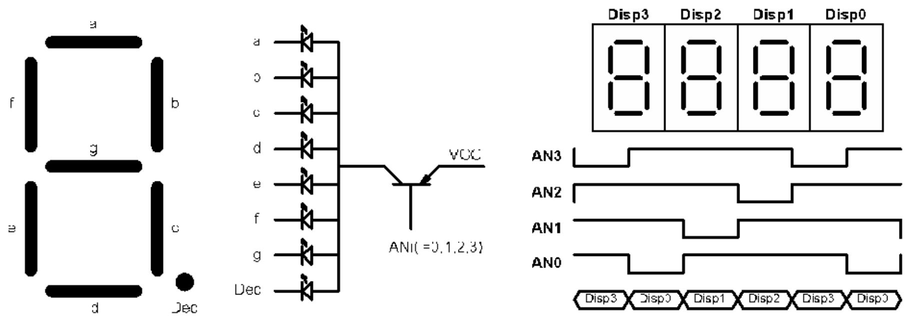

Figure 1. Multiplexing seven segment displays

Table 2. Truth table for a seven segment display

Di(3:0) i=3,2,1,0	Dec(i) i=3,2,1,0	abcdefgdec	Character
0000	0	00000011	0
0001	0	10011111	1
0010	0	00100101	2
0011	0	00001101	3
0100	0	10011001	4
0101	0	01001001	5
0110	0	01000001	6
0111	0	00011111	7
1000	0	00000001	8
1001	0	00011001	9
1010	0	00010001	A
1011	0	11000001	b
1100	0	01100011	C
1101	0	10000101	D
1110	0	01100001	E
1111	0	01110001	F
0000	1	00000010	0.
0001	1	10011110	1.
0010	1	00100100	2.
0011	1	00001100	3.
0100	1	10011000	4.
0101	1	01001000	5.
0110	1	01000000	6.
0111	1	00011110	7.
1000	1	00000000	8.
1001	1	00011000	9.
1010	1	00010000	A.
1011	1	11000000	b.
1100	1	01100010	C.
1101	1	10000100	D.
1110	1	01100000	E.
1111	1	01110000	F.

Report

A brief report that sums up what has been done and includes problems/challenges. It may be a good idea to draw figures with block diagrams, for example.

For each part of the exercise, you must hand in:

- VHDL source file(s)
- VHDL test bench
- *Do* file(s) for simulation in Modelsim
- Utilization report and Timing summary report

Exercise 4 and 5 must be demonstrated to the lab supervisor.

All the submitted VHDL files must follow the naming rules for VHDL files and indenting guidelines as described in the cookbook.

Exercise 2

Create a VHDL function *hex2seg7* that implements a combinatorial function in accordance with Table 2. Place the hex2seg7 function in the *subprog_pck* from Exercise 1

Create a test bench, *tb_hex2seg7.vhd*, to test the *hex2seg7* function. You should use the attached simulation model for the seven segments (seg7model_beh.vhd). By selecting Radix Hexadecimal for the DISPi outputs (i=3,2,1,0) of the model, you should obtain numbers in the waveform viewer that will be displayed in the seven segments. This will also be clear when a display is not active.

Exercise 3

In this exercise, you will implement control of the seven segments, so that all of them are active *simultaneously*. It is possible to achieve this by creating a construction in which the four displays are activated in sequence, in which the frequency is high enough to light the four displays simultaneously. Such a construction can be implemented by means of a counter. Note! ANi (i=3,2,1,0) must be active for a number of 100MHz periods in order for characters to be displayed correctly. If the AN pulses are too short, characters from the neighbouring segments will flow together.

This module should have the following entity:

```
entity seg7ctrl is
  port
  (
    mclk        : in std_logic; --100MHz, positive flank
    reset       : in std_logic; --Asynchronous reset, active h
    d0          : in std_logic_vector(3 downto 0);
    d1          : in std_logic_vector(3 downto 0);
    d2          : in std_logic_vector(3 downto 0);
    d3          : in std_logic_vector(3 downto 0);
    dec         : in std_logic_vector(3 downto 0);
    abcdefgdec_n : out std_logic_vector(7 downto 0);
    a_n         : out std_logic_vector(3 downto 0)
  );
end entity seg7ctrl;
```

This module is to be synthesized, but not tested on the test board in this part of the exercise. A test bench is to be created that includes the simulation model for the seven segments, and RTL code is to be simulated. For arithmetic operations, such as `+`, `-`, `*` and `/`, only the `ieee.numeric_std.all` package is to be used.

Guidance:

Create a block diagram that shows the data paths from `d0`, `d1`, `d2` and `d3` to the seven-segment displays. How should the counter be used; how many bits counter do you need? The counter and the decoder from lab exercise 1 may be useful here.

Table 3. Seven segment board hardware pins

Function	ZedBoard Pin
Segment A	JB7
Segment B	JB1
Segment C	JB8
Segment D	JB2
Segment E	JB9
Segment F	JB3
Segment G	JB10
Decimal point	JB4
Digit 0	JA10
Digit 1	JA9
Digit 2	JA8
Digit 3	JA7

Exercise 4

Use the entity in part 3 of the exercise together with the switches SWi (i=7,6,4...0) and push buttons `BTNR` and `BTNL` to implement a function that works as follows:

The value from the switches `SW(3 downto 0)` is to be stored in the four 4-bit registers `Di(3 downto 0)`, i = 3,2,1,0. The output from `Di` is to be displayed on DISPi, i=3,2,1,0. The switches `SW(7 downto 6)` together with a press of the button `BTN0` writes the value of `SW(3 downto 0)`, called `INP` in Table 2, to `Di`, where `i` corresponds to the binary number formed by `SW7` and `SW6`. The reading of new values to the registers is to be synchronized with the positive flank of the 100MHz clock on the board. Activating `RESET` sets all the registers to zero. How many bits must the counter have in part 3 of the exercise, and which bit is used to display clear characters on each of the seven segments?

Create a `regs` component as illustrated in the figure below and connect the `seg7ctrl` and `regctrl` components in a structural (`str`) top-level architecture. All the registers must be in reset when the reset signal is active `'1'`.

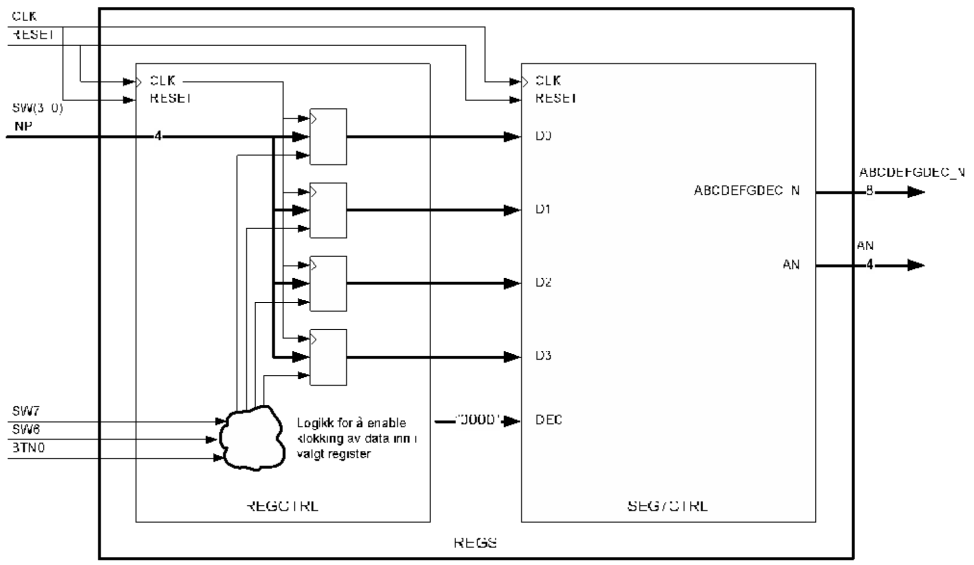

Figure 2. `regs` *component*

Table 4.

CLK	**BTNR** (reset)	**BTNL** (load)	**SW7 SW6** (sel)	**D3**	**D2**	**D1**	**D0**
-	1	-	-	00	00	00	00
-	0	0	-	D3	D2	D1	D0
↑	0	1	00	D3	D2	D1	INP
↑	0	1	01	D3	D2	INP	D0
↑	0	1	10	D3	INP	D1	D0
↑	0	1	11	INP	D2	D1	D0

Exercise 5

Change the design from part 4 of the exercise so that it becomes a digital stopwatch. Now the `load` and `sel` signals become unnecessary. They are replaced by a start signal in `BTNL` and a stop signal in `BTNC`. Reset in `BTNR` is maintained. After stop has been pressed, pressing the start button again will continue the counting. Reset sets everything to zero and waits for the start signal. After the display shows 9999, the counting goes to 0000 and starts over again.

Hint 1: Replace the `regctrl` component with a `clock` component that has `mclk`, `reset`, `start` and `stop` as its inputs.

Hint 2: A clock frequency of 100 MHz gives 10 ns for each clock period. A counter should be created in the clock module that counts up to 1 second, and then the display value increases by 1 for each second.

Exercise 6 (mandatory for inf4431 and optional for inf3430**)**

Change the design from exercise 5 to include lap time. When `BTND` is pressed, the display should show and hold the current counter value. The counter shall still run in the background, but not be displayed. When start is pressed, the display should return to showing the running counter.

Good luck!