Oblig 6
VHDL Programming using modules - Control of seven segments

IN3160/4160

Version 3/2023-02

Procedure for VHDL coding

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

- Try to form a correct idea of what type of solution is required.
  - 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 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. Comments in the code can be used to help structuring your solution while working with it.

Another rule of thumb is to keep code at the same level within each module.

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 guidelines 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 guidelines for VHDL source files

File content	File name
Entity and architecture	<design_unit_name>.vhd
Standalone entity	<design_unit_name>_ent.vhd
Standalone architecture[1]	<design_unit_name>_arch.vhd
Package	<packagename>_pkg.vhd
Behavioral architecture (simulation model)	<design_unit_name>_beh.vhd
Test bench (*not VHDL when using python*)	tb_<design_unit_name>.py

<design_unit> should be names that identify the function/content of the file (for example: seg7ctrl_ent.vhd, seg7ctrl_arch.vhd, tb_seg7ctrl.py).

In many cases, it may be beneficial to keep both entity and architecture in a single file to avoid having too many files, but it is also possible to separate each compilation unit to avoid spending time on compiling code that has not been changed.

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).

Test benches

The key question when designing a test bench is what test vectors must be generated for a complete simulation of the chip's behavior. 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.

Self-test vs test bench

Normally we talk of running test benches as a method for verification, while a self-test is one way of testing[2]. While a test bench verifies functionality by creating a simulation environment providing input and checking output values before implementation, a self-test does physically test a system, after implementation on physical hardware. In many complex systems in use today, it is usual to have self-tests implemented. Examples of self-test modules can be built in self-tests for RAM in PCs, simple test such as all the dashboard lights being lit when turning on a vehicle, or other diagnostic tests that can be accessed through menus when needed. Advanced self-tests are often hidden from the end-user, but available to service personnel when it is needed to perform necessary diagnostics.

---

[1] In designs having multiple architectures for one entity, architecture names should be `<entityname_architecturename>_arch.vhd`

[2] The word "testing" normally implies it is performed on physical hardware, while "verification" normally describes what is done to verify functionality before implementation.

Test bench for VHDL simulation

Seven segment displays used in this exercise

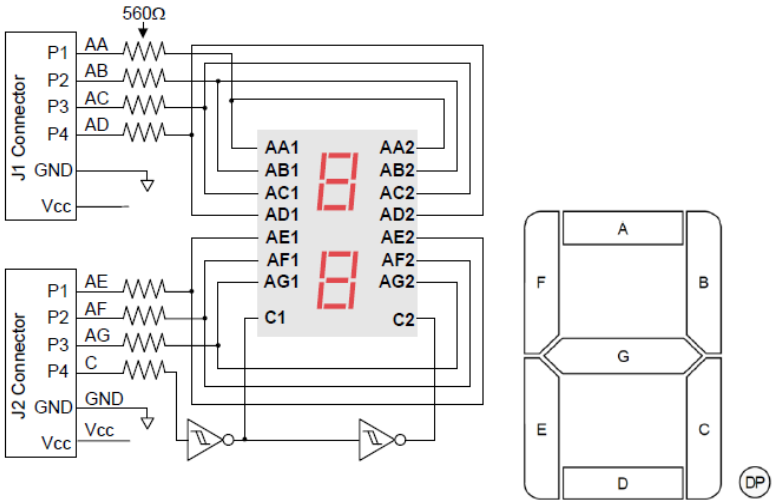

Figure 1. Seven-segment display connection diagram[3]

Table 1 Truth table for a seven-segment display

Di(3:0) i=1,0	abcdefg	Character
0000	1111110	0
0001	0110000	1
0010	1101101	2
0011	1111001	3
0100	0110011	4
0101	1011011	5
0110	1011111	6
0111	1110000	7
1000	1111111	8
1001	1111011	9
1010	1110111	A
1011	0011111	B
1100	1001110	C
1101	0111101	D
1110	1001111	E
1111	1000111	F

[3] Source: Pmod seven-segment display reference manual: pmodssd_rm.pdf

Report and deliverables

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 this exercise, you must hand in:

- All VHDL source file(s)
- Python test bench for b) and c)
- *Makefile the testbenches (comment out changes/replacements)*
- *Waveforms (.ghw)*
- Utilization report and Timing summary report for d)
- The .bit file used for programming the board.

All the submitted VHDL files shall follow the naming guidelines for VHDL files and use indentation consistently to ensure good readability.

a) *bin2ssd*

```
entity bin2ssd_test is
  port
  (
    di         : in std_logic_vector(3 downto 0);
    abcdefg    : out std_logic_vector(6 downto 0);
  );
end entity bin2ssd_test;
```

Create a VHDL function `bin2ssd` that implements the translation from binary number to seven segment code according to Table 1. Use the function in an architecture with the entity given above and test your function using the provided testbench tb_bin2ssd.py.

Optional: put the `bin2ssd` function in a separate package `seg7_pkg` that can be used later in this and coming assignments.

b) *seg7ctrl*

In this exercise, you will implement control of the seven segments, so that all of them seem active *simultaneously*. It is possible to achieve this by creating a construction in which both displays are activated in sequence, using a high enough frequency. The human eye will normally be unable to detect flicker from light being strobed at frequencies of 40-100Hz or above, depending on the duty cycle (duty cycle = on/off ratio). This can be achieved by using a counter to keep track of time.

The output *c* indicates which of the two displays are active, and the value of *abcdefg* must be according to d0/d1 and the combinational function, depending on which of the displays are supposed to be active.

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 high
    d0         : in std_logic_vector(3 downto 0);
    d1         : in std_logic_vector(3 downto 0);
    abcdefg    : out std_logic_vector(6 downto 0);
    c          : out std_logic
  );
end entity seg7ctrl;
```

This module shall be synthesized, but not tested on the test board in this part of the exercise. Create a testbench that uses a python dictionary such as in the provided testbench to verify that *abcdefg* are the are the decoded value of *d0* when *C=0* and *d1* when C=*1*.
(You may choose to build upon the testbench provided or make your own.)

Simulate the test bench with the RTL code.

For arithmetic operations, such as +, -, * and /, use the numeric.std package (use ieee.numeric_std.all).

How many bits must the counter have, and which bit is used to display clear characters on each of the seven segments? *(If you are clearing your counter using a predefined value, how many bits do you need for this value.*)

Guidance:
Create a block diagram that shows the data paths from d0 and d1 to the seven-segment displays. How should the counter be used? The counter and the decoder from oblig 1 and 2 may be useful here.

Note! Each display must be active for a number of clock cycles in order for characters to be displayed correctly. If c pulses are too short, characters from the other segment may flow together.

Tips: While entity ports normally should be `std_logic`, it may improve readability to declare signals (such as counter registers) as `unsigned`, to avoid excessive use of type conversions.

c) self-test unit

To be able to test the seven-segment module when it is synthetized and implemented on the FPGA (i.e. not in a testbench/simulation), you shall build a self-test module that displays a pre-defined pattern of characters and numbers on the seven-segment display.

The self-test unit shall feed a modified the *seg7ctrl* module a sequence of characters.

- Create an alternative architecture for *seg7ctrl*, named *seg7ctrl_arch.vhd*. In this file, copy the architecture from b) and replace the values for the seven segment output with the values given in Table 2 below. This alternative encoding paired with the values for D1 and D0 from Table 3 below, that you will feed into your seg7ctrl, will display a "secret" message on the seven segment display when done correctly!

Di	abcdefg		Di	abcdefg
0000	0000000		1000	0111011
0001	0011110		1001	0111110
0010	0111100		1010	1110111
0011	1001111		1011	0000101
0100	0001110		1100	1111011
0101	0111101		1101	0011100
0110	0011101		1110	0001101
0111	0010101		1111	1111111

Table 2 new seven-segment code table

During compilation, make sure the new architecture is compiled after seg7ctrl.vhd, in order to replace the architecture.

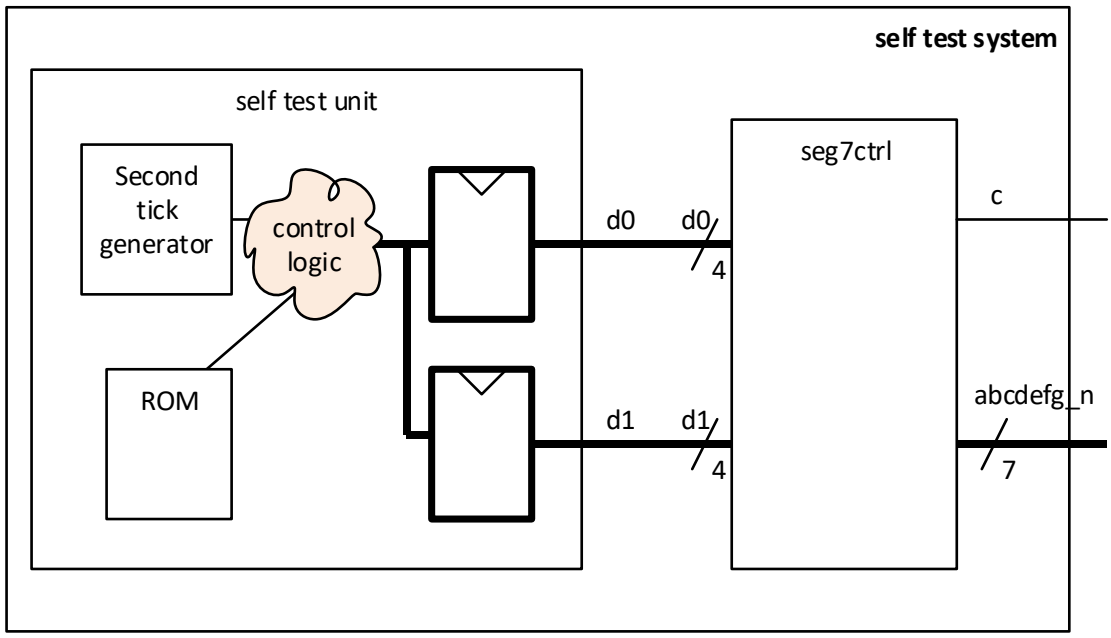

Figure 2: The self-test unit.

- Create a VHDL file for the *self test unit*. The *self test unit* shall contain a counter that will create a signal "*second_tick"* that is active ('1') exactly one clock cycle every second.

For each time the "*second_tick*" signal 'ticks', the *self test unit* shall change the value of D1 and D0 into seg7ctrl in accordance with Table 3 below in order to display the "secret" message.

Store the values that shall be sent into D1 and D0 (Table 3) in a ROM (file IO is optional[4]), and access each value sequentially (two at a time, one value for each signal) every time a second passes, in order to display the message. The message shall start over once finished.

- Create a testbench that gives stimuli to the *self test unit* (only a reset and a clock). It is sufficient to check that d0 and d1 is what you expect and changing with every *second_tick, b*ut feel free to make it self-checking[5].

D1 (hex)	D0 (hex)
1	2
3	4
4	0
0	0
5	6
7	3
0	0
8	6
9	0
0	0
A	B
3	0
0	0
C	6
6	5
0	0

Table 3 «Secret» message ROM table

[4] Using File IO and some creativity, it is easy to create arbitrary long messages in a ROM. Using both the displays you can create and decode almost any letter.

[5] When simulating it is ok to make second_tick faster by adjusting the counter limit. Do remember to change this back when moving to implementation!

d) Implementation

Implement the design on the zedboard, and check that it runs[6] and does what it is supposed to do.
- To do this you shall wrap your *self test unit* and *seg7ctrl* modules in a new module named self test system. This module does not have any inputs (only a reset and clock), as all the data is handled made by the *self test unit* itself, but it does have two outputs. One std_logic for C and one std_logic_vector[6:0] for abcdefg. See Figure 2 above.

- Create a testbench for the *self test system*. This can be very similar to the one you made in task c) for *self test unit*. Verify that the outputs c and *abcdefg* is as you expect

- Create a project in Vivado with all your needed synthesizable design files and go through the Vivado workflow to assign pins to the ports of your module (See Table 3 below) and generate a bitstream. Be sure that there are no critical warnings and upload the bitstream.
(Keep in mind that this time around we will NOT map our clock to a button, but to a crystal oscillator located on the Zedboard.)

The zedboard has a crystal oscillator that generates a 100MHz clock signal which is connected to the Zynq-7000 Package Pin Y9 (Se chapter 2.5 in the Zedboard user guide). Pin numbering on the chip can be found in the Pmod Connections in the Zedboard documentation. The reset signal shall be connected to the BTNR in the zedboard. Use the Zedboard User Guide to find the package pin correct pin.

All input and output for the project is 3.3V.

Table 3. Seven segment board hardware pins

PmodSSD	ZedBoard	Xilinx Zynq chip See Zedboard user guide
Segment A	JC1_P	AB7
Segment B	JC1_N	AB6
Segment C	JC2_P	Y4
Segment D	JC2_N	AA4
Segment E	JD1_P	V7
Segment F	JD1_N	W7
Segment G	JD2_P	V5
C (CAT)	JD2_N	V4

Good luck!

[6] When working at home, the Zedboards in the lab can be accessed and viewed as described in the wiki: https://robin.wiki.ifi.uio.no/Remote_access

e) Optional (not needed for approval):

Create a system that counts the each time the button BTNL is pressed and displays the count as a base10 number on the display. The system should be reset when pushing the BTNR button.

Hint: Use separate counters for each digit, rather than translating 8bit binary to two decimal numbers.

How well does the number readout correspond to the times you push the button?
Why is it like this?

How can we achieve a better readout if we only want to count the actual number of times we physically press the button?