Design a pattern recognizer

Contents

1. Introduction

==========

For this assignment you hand-in in a one ZIP file with:

1. **A report (in pdf)**
2. **All VHDL files**
3. **Generated *.vho file generated by the synthesis tool (no other files generated by the synthesis tool and modelsim!)**
4. **Script files that I can use to analyse and simulate your design with ModelSim-Altera**
5. **The programming file for the DE1-SoC board (*.sof)**

Furthermore you have to show the correct operation on the Cyclone board.

==========

You have to design a circuit that counts the number of occurrences of a pattern. The requirements are informally given by our client, and you are not allowed to change it.

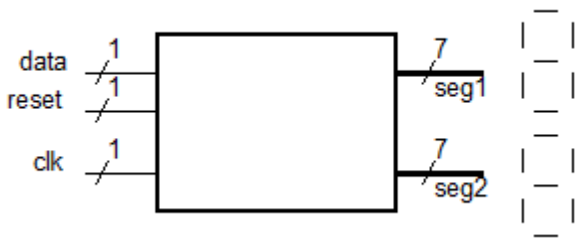
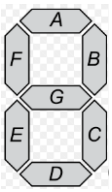


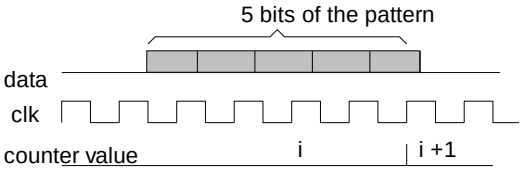
Figure 1: Input /Output



std_logic_vector(6 downto 0), with
A is index 0, B is index 1, ..

The informal description of this design is:

- It is a synchronous system, with an asynchronous reset
- The active edge of the clock is the rising edge of the clock.
- It recognises the pattern 11100 (from left to right)
- It counts the number of occurrences of this pattern
- In the clock cycle after the last bit of the pattern is detected the value on the display is incremented (Hence, before the next rising edge of the clock).



- It uses two seven-segment displays.
- After more than 99 occurrences the displays show '--' (overflow).
- After a reset the counter is zero again, if a part of the pattern was already recognised it is ignored.
- Before the first reset the circuit has an undefined behaviour. The display can show any pattern.

You have to design it in a top-down fashion (it is treated as if it is a really hard problem). On the next pages the design steps are give in more detail. The design steps are:

1. First you have to formalise the informal specification. So a behavioural description has to be written in VHDL
2. Next you have to test this behavioural description, for this you will write a test environment in VHDL.
3. Than the design is divided in sub systems. A structural description is to be written in VHDL
4. Behavioural descriptions of the sub systems
5. Test the new design
6. Synthesise the design
7. Perform a post simulation

Notes:
1. Two type of seven segment displays are available: common anode and common cathode. It depends on the development board what type is used (check the documentation/schematics of the board).
Similar the reset can be active low or active high (this also depends on the development board).
The documentation DE1-SoC development board is at:
https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=836

2. A behavioural description could also imply an implementation, sometimes it is also accepted by synthesis tools. The purpose of the behavioural description is that it should clearly describe the behaviour.

2. Formal description of the problem

Write a behavioural description for the previous informally given problem. The name of the entity is *pattern_recognizer* and the names of the inputs and outputs are also shown. Notice that *seg1* and *seg2* are of type std_logic_vector (6 downto 0).
It is **not** the intention that a behavioural description is focussed on implementation details. In stead it is important that it is readable (I mean understandable!) for others.

For the behavioural description of the *pattern_recognizer* only <u>one</u> concurrent statement is to be used. This concurrent statement is a process statement with the following structure:

```
process(reset, clk)
 <your declarations>
Begin
  -- no code here
  if reset='0' then -- reset is active low

     reset actions

  elsif rising_edge(clk) then;

     synchronous actions (sequential)

  end if;
  -- no code here
end process;
```

Note: In the behavioural description you have to count. You can use an integer with a range constraint. For debugging you can first replace the type of the output in an integer. If the design is correct you can replace the output in type std_logic_vector and use a conversion function:

function int2segm(Inp :integer) return **std_logic_vector** is -- NOT std_logic_vector (6 downto 0) !!!
…
begin
…
end int2segm;

Tip: put the function in a package. Than it can be used also in the other design steps.

3. Testing the behavioural description

3.1 pattern generator

Is the behavioural description correct? Is overflow correctly implemented? Is 'reset' working?

1. Write a test pattern generator in VHDL (entity testbench), that:

- Generate a periodic clk signal (clk period 20 ns)
- Generate simple test data
  - apply 110 times the pattern,
  - perform a reset,
  - apply 50 times the pattern

The previous pattern can be described with only one process description and an additional process for the clock generation. Furthermore add an assert statements to indicate what is test is performed e.g.

ASSERT false REPORT "the input pattern is generated 50 times" SEVERITY note;

1. Analyse and Simulate the testbench.

3.2 structural description of the test

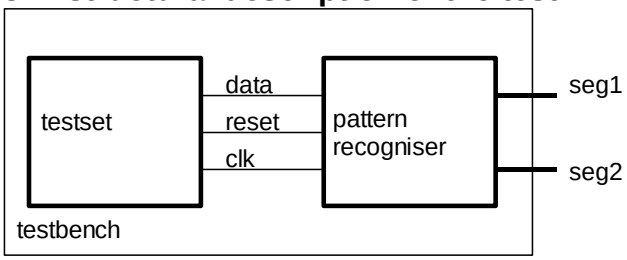


figure 2: the test environment

1. Make a structural description in VHDL of the above description.
2. Analyse the test environment.

4. The three subsystem

4.1 The structural description

Design the pattern recognizer with the same external behaviour with sub systems. Each sub system is modelled with an entity and architecture:

- list detection
- counter
- display drivers

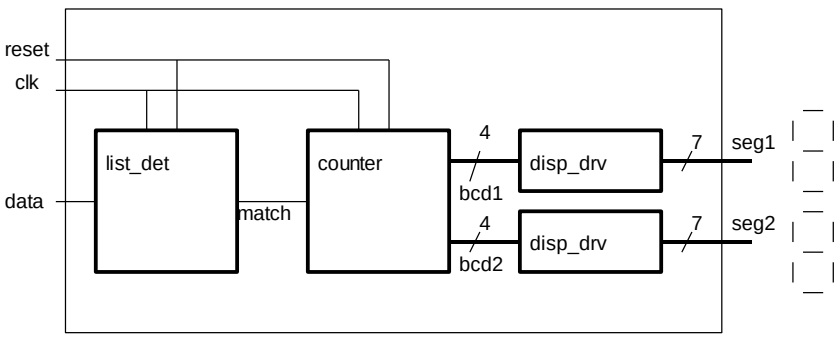


figure 3: division of the system in four sub systems

The system *list_det* detects the pattern and generates a '1' at the output *match* if the pattern is detected. This *match* signal is used to increment the counter value. The output of the counter are two bcd codes, type integer, .. (You can choose). Design *disp_drv* converts the input to the seven segment code.

1. Make a synthesisable VHDL descriptions of the sub systems *list_det*, *counter* and *disp_drv*. The active clock edge of all flipflips in your design is the rising edge of the clock.
2. Make a structural description in which these components are used. You already have the top-level entity (don't copy that entity description). The name of the architecture must be: *subsystems*

Notes:

1. Quartus supports an integer value divided by a constant value 10. Many synthesis tools only support division if the right operand is a constant that is a power of 2. Consider using two integers (both from 0 to 9) for *list_det*.
2. For a large system it is advised to write a behavioural description of the subsystems in stead of a implementation directly, since:

- Behavioural descriptions are generally less error prone.
- Simulation of the whole system (testing your subdivision) is easier.
- After a sub system is designed it can be tested with still a behavioural description for the other sub systems at a behaviour. This is a simple design, so you can give descriptions that are synthesisable (area is important; the clock frequency of the systems is 50 MHz).

4.2 Test the structural description

Analyse and test your design with four sub systems.

4.3 Test environment

In the previous task you have test the correctness of the design by simulation. You examined the output. However, it is better to do this automatically by comparing it with the behavioural description.

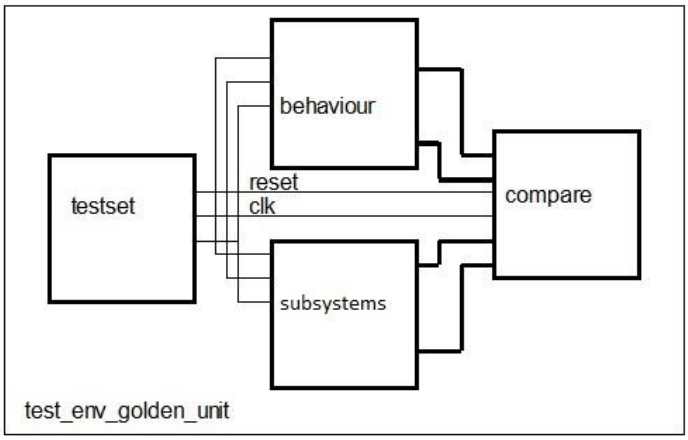


figure 4: testing against a golden unit

Note:
The architecture *behaviour* and architecture *subsystems* share the same entity. But how to distinghuish between the two architectures in design *test_env_golden_unit*? A **configuration** (this is also a design unit) for *test_env_golden_unit* is to be used.

What is a configuration?
You find more information on the design unit *configuration* in the presentation *VHDL in more detail* on slide 64 (it could be that the slide number is changed due to updating of the presentation). On this slide a configuration is given for a *test_bench1* that compares two instantiations of an *sr_latch*. An alternative is (and many more similar explanations on the web):

https://www.vhdl-online.de/courses/system_design/vhdl_language_and_syntax/vhdl_structural_elements/configuration (accessed 26 January 2021)

The entity *test_env_golden_unit*, which has no inputs and outputs is used for this. In this figure the component *compare* is new. It compares the generated outputs (*seg1* and *seg2*) of both pattern recognizers. **However, the output patterns need not to be same all time:**

- In a synchronous system only just before the next rising edge of the clock need to be the same.
- Furthermore the outputs need not be the same before the first 'reset'.

If the component *compare* detects an error an warning, a warning must be given. Use an assert statement for this.

1. Write an entity with behavioural description of compare, and analyse it.
2. Write the structural description of the test environment, and analyse it.
3. Perform a simulation

5. Synthesize

Notes:
1. It is very important that you copy the .qsf file and the .sdc file in the project directory BEFORE you start Quartus II. If you violate this a random pin mapping is used in stead of the pin mapping that is in the .qsf file. As a result your design is not successfully realized on the DE1-SoC (chapter 6).
2. In this design the top-level entity description only has types *std_logic* and *std_logic_vector*. The setting of the synthesis tool is such that also a top-level entity is generated, but that is not necessary because you already have it. Therefore **instruct Quartus not to generate a top-level entity**. After creating the project (and before compilation) do:
- "assignments" →"settings"→select "simulation"→select "More EDA Netlist Write Settings …"
- Change the setting "Do not write top level VHDL entity" in " On"
- finalize with " OK" and " OK"

After compilation check that you *.vho does not contain the entity description.

For timing analysis Quartus requires an SDC file that includes the required clk (the clock frequency is 50 MHz) and I/O constraints.
File **<name of top level entity>.sdc** with:

```
create_clock -period 20.000 -name clk [get_ports clk]
derive_clock_uncertainty
set_input_delay -clock clk -max 0 [get_ports reset]
set_input_delay -clock clk -min 0 [get_ports reset]
set_input_delay -clock clk -max 0 [get_ports data]
set_input_delay -clock clk -min 0 [get_ports data]
set_output_delay -clock clk -max 0 [get_ports seg1[*]]
set_output_delay -clock clk -min 0 [get_ports seg1[*]]
set_output_delay -clock clk -max 0 [get_ports seg2[*]]
set_output_delay -clock clk -min 0 [get_ports seg2[*]]
```

Synthesize the design (with the sub systems) for the Altera device: Cyclone V (device 5CSEMA5F31C6). Generate also a post simulation file (language VHDL). The result is in directory: ..\simulation\modelsim\

1. Add the generated schematic (RTL view) in your report.
2. Explain the number of flipflops in your design.
3. The Timing Analyzer (part of Quartus) can generate a datasheet with setup times, hold times, clock to output times and minimum Clock to Output Times. Add a screenshot of these reports in the report (mention explicitly the "set operation Condition"(e.g. fast 1100mV 0C Model).
   See chapter 4.3 in document "The Intel Quartus Prime Time Analyzer" (on Canvas)
   Note: start the Timing Analyser tool via Tools→Timing Analyzer

6. Post simulation

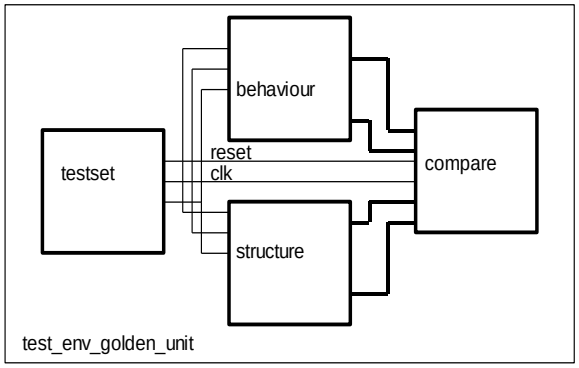


figure 5: test generated post simulation model against a golden unit

For component "structure" in figure 5 use the generated post simulation model.

1. Simulate the design in figure 5
2. Add a screenshot of a selection in your report (zoom in for ~3 clock periods, that includes an increment of the seven segment displays. Show the internal lines of figure 5 in the wave window).

7. Program the FPGA

Program the FPGA.
Note 1: Did you used the correct qsf file?
Note 2: Key0 is the reset button and Key1 is connected to the data input. Pressing and releasing this key generates at least 3 ones and 2 zero's (clock is 50 MHz ☺).

Note: due to Covid-19 we have to do the lab at home. Therefore I will program the FPGA with your design and demonstrate it. **It is required that you change the name of the *.sof file in:**
<surname student1>_<std number student1>_<surname student2>_<std number student2>.sof, e.g.
Biden_s1234567_Harris_s7654321.sof

Procedure: during the first three lab sessions on Thursday morning you can email me the sof file (e.molenkamp@utwente.nl). **don't send sof files at other times because I won't do anything with it. The number of times you can send an email is limited to 3.**