Designing a pattern recogniser

In this exercise you have to desing a circuit that counts the number of occurrences of an input pattern. The requirements are informally given by our client, and we are not allowed to change it.

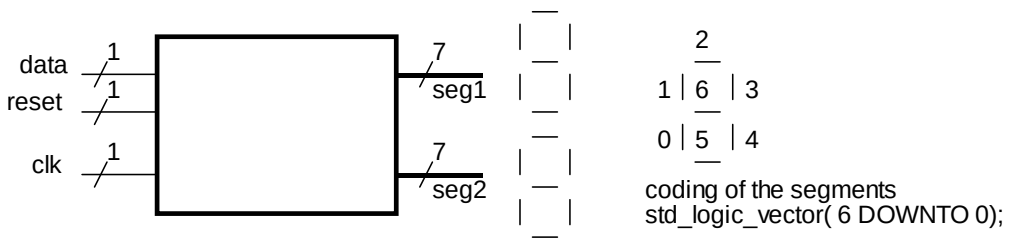
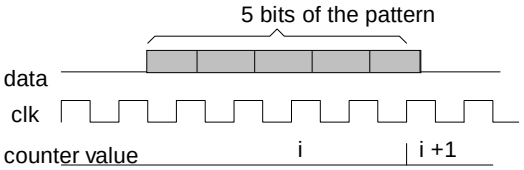


figure 1: Input /Output

The informal description of this design is:

- It is synchronous system (including the reset)
- It recognises the pattern 11100
- It counts the number of occurrences of this pattern
- If a pattern is received, then the counter is updated (see figure below).



- It uses two seven-segment displays to show the number.
- After more than 99 occurrence the display shows '--' (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.

This problem will be designed in a top-down fashion, and it will be treated as it is a real hard problem. On the next pages the problems are discussed

1. First of all we have to formalise the informal specification. So a behavioural description has to be written.
2. Next we have to test this description, for this we will write also a test environment in VHDL.
3. Than a first division of the problem in three subsystems is given. A structural description description is written in VHDL
4. Give a behavioural description of the three subsystems
5. Test this description
6. Synthesise these implementations

1. Executable specification of the problem

Write a behavioural description for the previous informally given problem. The name of the entity is *pattern_recogniser* 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 very important that it is readable (I mean understandable!) for others!

Note:
A behavioural description could also imply an implementation, sometimes it is also accepted by synthesis tools. But the essence is that it gives a specification of the system to design.

For the behavioural description of the *pattern_recogniser* only one concurrent statement is to be used. This concurrent statement is a process with the following structure:

```
process
 <your declarations>
begin
  wait until rising_edge(clk);
  if reset='1' then

     reset actions

  else

     synchronous actions (sequential)

  end if;
end process;
```

Analyse this description.

Note 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. Therefore use two integers (both from 0 to 9).

Note 2: In the behavioural description you have to count. Therefore you probably use an integer (with range constraint). As first approach replace the two 7-segment display outputs with an integer. This easier for debugging. The conversion from integer to a 7-segment display use a function.

Function int2segm(Inp :integer) return **std_logic_vector** is -- NOT std_logic_vector (6 downto 0) !!!
…
Begin
…
End int2segm;

2. Testing the behavioural description

2.1 pattern generator

Is the behavioural description correct? Is it counting the specified pattern only?, Is overflow correctly implemented?, Is 'reset' working?

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

- generates periodically the clk signal (period time 10 ns)
- apply a simple simple test data
  - apply 110 times the pattern,

- perform a reset,
- apply 50 times the pattern

The previous pattern can be described easy with only one process description, and an additional process for the clock generation. Furthermore add assert statements to indicate what is tested at that moment, e.g.

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

Analyse and Simulate the testbench.

2.2 structural description of the test

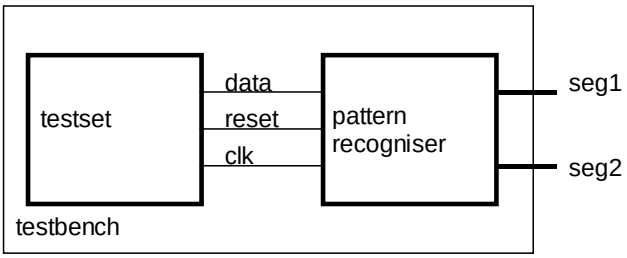


figure 2: the test environment

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

3. The three subsystem

3.1 The structural description

Examine the problem, it is possible to recognise three subsystems:

- list detection
- counter
- display drivers

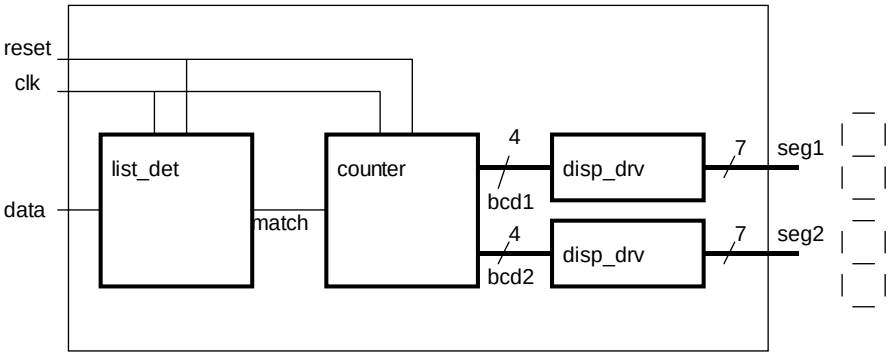


figure 3: division of the system in four subsystems

The *list_det* detect the pattern and generates a '1' at the output *match* if the pattern is recognised. This *match* signal is used to increments the *counter*. The output of the counter are two bcd codes, that are used to drive the *disp_drv*, which converts the bcd code in to the seven segment code.

1. Make a structural description in which these components are used (the entity declaration is not to be written again!).

3.2 Behavioural description of the three entities

1. Make the behavioural description of the three entities 'list_det', 'counter' and 'disp_drv'.

Note:
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 easy.
- After a subsystem is designed it can be tested with still a behavioural description for the other subsystems at a behaviour.

3.3 Test the structural description

3.4 Test environment

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

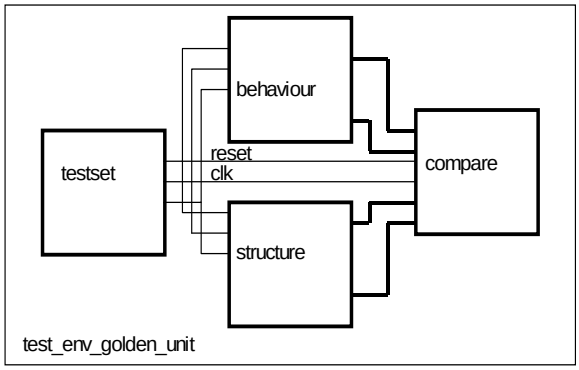


figure 4: testing agianst a golden unit

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 recogniser's'. **However, the output patterns need not to be same all the time.**

- In a synchronous system only the moment 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* finds an error the following message "behaviour differs from design step" is printed; using the assert statement.

***assert** false **report** "behaviour differs from design step" **report** warning;*

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
   (Ans is your design step correct?)

4. Synthesize

Synthesize the three subdesigns (or perhaps your behavioural description is already synthesisable).
If synthesisable can you explain the number of flipflops in the design?