UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ESIGELEC

Assignments

VHDL and logic synthesis course

(spring 2023 edition)

Lab assignment s6

Design of a `simple_counter` counter

Francisco Rodríguez Ballester
prodrig@disca.upv.es

Computing Engineering Department
Universitat Politècnica de València

Spring 2023

Assignments for the VHDL and logic synthesis course (spring 2023 edition)

Lab assignment s6

Design of a `simple_counter` counter

Contents

Pay attention

During the course you will need to develop different design units or alternative versions of a given design unit.
You are advised to take precautions in order to avoid losing your work when developing one of these designs or variations. To clearly distinguish files from separate assignments you should solve **each assignment in a separate, empty folder**, making copies of those files you need from previous assignments instead of reusing the source code files from a previous assignment to develop a new one.
The same advice is applicable to synthesis projects: it is wiser to create a new project per assignment than to reuse a project from an already finished assignment unless you are explicitly asked to do so.
If you work this way you will always have the *opportunity to review, modify, compile, simulate, etc. any finished and already submitted assignment* if the need arises. For example you **may receive a comment from your instructor** about an already submitted assignment **asking you to do some modifications** to your code or changes to the project in order to resubmit the assignment and obtain a better mark.

1 Introduction

In this lab assignment you have to create a design unit of a flexible counter. This counter is flexible in the sense that a generic parameter allows to set the maximum count stored into the counter. It has a couple of input ports (called `rst` and `clk`) both of them `std_logic`, and a couple of output ports (called `count` and `mcount`), one of them integer and the other `std_logic`; it also has a generic (called `MAX_COUNT`) to set the maximum value that can be stored into the counter.

The counter must maintain the count (an integer value ranging from 0 to `MAX_COUNT`) internally in order to set the `count` ouput and determine if the value of the `mcount` output is either '1' or '0'. There must be an internal data object maintaining the counter value as this value needs to be read and written, something impossible for an output port like `count`.

This is a **simulation-only** assignment in which you must develop and test a simple yet flexlible counter.

This design unit is composed of two separate blocks:

- An edge-triggered counter that increases its internal count on each active edge of the clock, from 0 to a given maximum value ciclycally. This internal count value is then copied onto the `count` output port. The use of a `process` with the correct **sensitivity list** to describe its behaviour is **compulsory**.

- A combinational block that determines, from the internal count and the generic `MAX_COUNT` values, if the output `mcount`has to be '0' or '1'. It is **compulsory** to model this combinational block using a **parallel** `when` **signal assignment**.

In this lab assignment you are required to develop both the design unit and the testbench code from scratch.

1.1 Important notes about work authoring

In this 2023 edition of the course you may work on your own or with a team mate. You can comment your work with other classmates (other than your team mate, if you have one); you're in fact encouraged to do so. However **sharing your code is well off limits!** You can comment, reason, criticize, etc. the work of your other classmates but it is absolutely required you add personal/team work to the files you submit in order to get credit.

If you work with a team mate, only one of the team members must submit the assignment. It is also **essential** that you **include your name(s)** as a comment at the very start of each text file in order to get credit. This applies also to files downloaded from the course site, *if you have to edit them.*

1.2 VHDL background before you start

Before you start working in this assignment there are some VHDL concepts you must be familiar with, added to the VHDL concepts from all previous assignments. If you're not comfortable with any of them, please resort to the referenced documentation:

1. Using an `process` to model an edge-triggered memory block with asynchronous inputs.
   - All *Rx* templates (from *R1* to *R8*) from the *VHDL model templates* PDF document you may find in the moodle course page.
   - Unit 5, slides 20-24, 27-29, 32-34.
2. Using parallel assignments to model a combinational block through a dataflow design.
   - Unit 4, slides 5-12

2 Project setup

Repeat the steps of the first assignment (lab assignment s1 - dec3to8i) to create a Modelsim project.

The project name must be `lab_s6_<YourSurname>` to clearly identify your work once downloaded by the instructor, and it must be placed in a folder of the same name.

Replace <*YourSurname*> with your last name with no spaces (and without '<' and '>'). Avoid using characters not belonging to the English alphabet like accented letters; for example if your name is Paco Rodríguez (note the accented í) the project name should be `lab_s6_Rodriguez`.

Activity 1 — Create the project

Create a new Modelsim project called `lab_s6_<YourSurname>` as described in this section.

2.1 Add files to the project

Once the project is created perform the following steps to prepare the project files for this assignment:

1. Create a new file and add it to the project. This first file, `simple_counter.vhd`, is a *VHDL Module* that will contain the description of the counter.

2. Create a new second file, `simple_counter_tb.vhd`, and add it to the project. This second file is a *VHDL Testbench* that will contain the VHDL code used to test the counter.

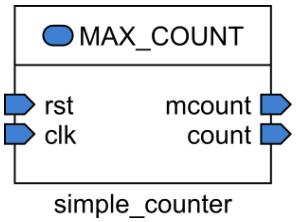


Figure 1: Simple counter interface.

3 Develop the design unit interface

Remember you must import all declarations from the `std_logic_1164` package (compiled into the `ieee` library) in order to be able to use the data types `std_logic` or `std_logic_vector` when writing the interface for the `simple_counter` module.

The information required to develop the interface of the counter design unit is shown in Table 1 and depicted in Figure 1.

Please note: The `rst` input is low-level active and the `clk` input is active on the rising edge. This information is not part of the module interface (in both cases the data type is `std_logic`) but part of the description of the module's behaviour; ie. this information will be used to code the module's `architecture`.

Table 1: Simple counter interface

Filename		`simple_counter.vhd`	
Entity name		`simple_counter`	
	Generics	**Generic name**	**Generic type and default value**
		`MAX_COUNT`	Maximum counter value. Type `integer`, default value 5
	Inputs	**Port name**	**Port description**
		`rst`	Reset input. Low-level active. Type `std_logic`, default value '1'
		`clk`	Clock input. Triggered on the rising edge. Type `std_logic`
	Outputs	**Port name**	**Port description**
		`mcount`	Maximum count output indication. Type `std_logic`
		`count`	Counter value. Type `integer range 0 to MAX_COUNT`

Imagine that, in the future, we want to use (instantiate) a counter with the same behaviour as this one except that we do not want/need the reset input; that is, we want to use a `simple_counter` counter *as if it hand't a `rst` input*.

We could, of course, create another module with almost the same behaviour but without a `rst` input. Instead creating another module, however, we can use the power of VHDL to allow the use of this counter (that declares a `rst` input) *as if it hand't a `rst` input*.

That is, with a single `entity` we can create two counters: one with the `rst` input and reset functionality and another one without the `rst` input and, obviously, without the possibility to enter the reset condition. And the difference between the two will be just including (in the first case) or excluding (in the second one) the `rst` input from the port mapping section at the instantiation, and the use of the correct default value.

In order to determine the correct default value for the `rst` port we need to know that `rst` is a low-level active value (so the counter is reset when this input is `0`). With this in mind the reasoning is

1. If we set a default value of `0` for the `rst` input, when this input port is not mapped in an instantiation the `rst` port will have a constant value of `0` and hence the counter will be permanently in the reset condition.

2. If we set a default value of `1` for the `rst` input, when this input port is not mapped in an instantiation the `rst` port will have a constant value of `1` and hence the counter won't be able to enter the reset condition.

A counter that can't count because it can't leave the reset codition is useless, so from the above two options, the correct default value for `rst` is `1`. If this default value is used

is because the `rst` port has been excluded from the port mapping in the instantiation, so the counter won't be able to enter the reset condition ...
But that is exactly what happens with a counter without reset input, right?

4 Develop the design unit implementation

The behaviour of this counter is quite simple once you realize it **consists of two independent blocks**:

- There must be a block to maintain the counter value; this the *block1* in Figure 2. This must be modelled after a flip-flop / register template as the counter counts on the rising edge of the clock input. The calculated value must be maintained into an internal data object (without direction restrictions, so can be read and written) and copied onto the `count` output port with a parallel assignment; this is because the `count` output port has direction restrictions and can only be written into and to count we need to read and write (the VHDL code for counting is something along the lines of `count_value <= count_value + 1`).

  1. If the reset input is active (low-level, '0') the counter value must be reset to 0 immediately (asynchronously).
  2. If the reset input is not active then the VHDL code must test if a rising edge is detected on the clock input. If this is the case then the counter value must be incremented in one unit unless the value equals `MAX_COUNT` because the next counter value must be zero.

- There must be a combinational block (*block2* in Figure 2) that compares the current counter value against `MAX_COUNT` and determines the value of the `mcount` port to be '1' if those values are equal and '0' otherwise.

As a specific requirement of this assignment it is **compulsory to use a `process` with sensitivity list** to model the behaviour of the first block. You are adviced to look for the template that best suits the above description (register with an asynchronous reset input) in the *VHDL Process templates* PDF document you may find in the moodle web page.

It is also **compulsory to use a parallel `when` signal assignment** to model the second block.

4.1 Implementation tips

In order to express what you want to get from the circuit (the circuit behaviour) you need to have a clear idea of such behaviour. This section tries to guide you during the coding process, starting from the reasoning that must be carried out **before** code typing.

First of all you must depict a block diagram of the blocks/subsystems your circuit is composed of in order to

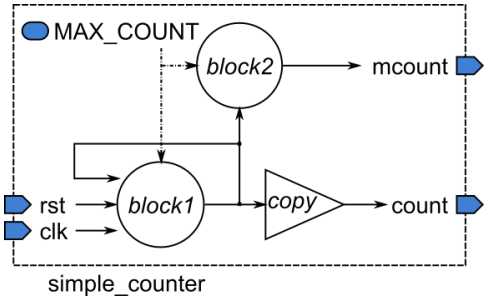


Figure 2: Simple counter internal blocks (initial version).

- Isolate each subsystem to ease the analysis of what kind of circuit is required: either combinational, latch or register.
- Determine the inputs of each subsystem.

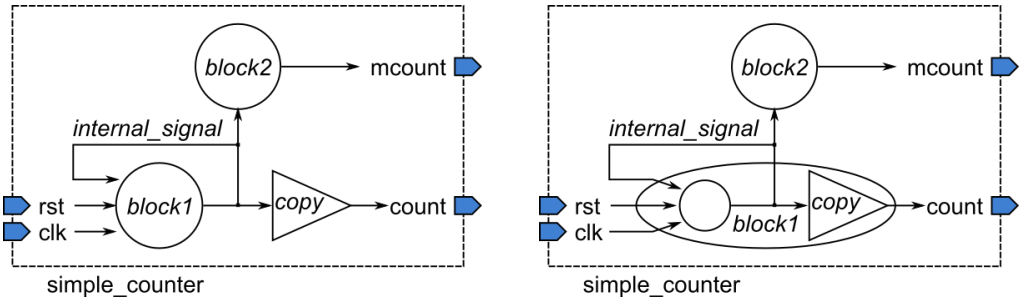
This divide & conquer strategy in which you devise the subsystems of the circuit you want to design and their connections does not preclude the use of a single design unit to develop your description, but it helps to better understand the problem and to select the best description strategy (dataflow, structural, or behavioral) for each subsystem.

In this particular case there are two subsystems or blocks that can be described separately, each devoted to determine the value of one of the circuit outputs as depicted in Figure 2.

Let's analyze this block diagram and try to extract as much information as we can

1. The block1 must use the `process` template of a register with an asynchronous reset input.
2. The block1 must read the current value of the counter to calculate the next one; this is the reason the calculated value is used also as a block input.
3. Due to the fact it is a requirement of the assignment to separate the description of block2 as a parallel `when` signal assignment, a signal must connect both blocks; it has to be a signal? yes, can't use a variable (in this particular case) because a variable can't be used outside the `process` where it is declared.
4. As output ports can't be read, the internal counter value can't be held into the `count` port; doing so is impossible because it is an output port but the counter value must be read from blocks block1 and block2.

From the above analysis it should be evident the first system block diagram version has to be refined to avoid reading from `count`. The solution is quite simple in fact: declare and use an internal signal for all the calculations and use a third block to copy the calculated value onto `count`. This block is also depicted in Figure 2 as *copy*; the name



(a) Copy block is an independent block.

(b) Copy block is part of block1.

Figure 3: Simple counter internal blocks (final version).

has been selected to emphasize the idea this block is a mere parallel signal assignment from the internal signal (whatever name you choose) onto the `count` output port.

There are two options to describe the copy block, as depicted in Figure 3[1]:

- An independent block as depicted in Figure 3a. In this case the signal assignment is a parallel one.
- Part of the block1 (see Figure 3b). In this case the signal assignment is a sequential one as it is inside the `process` that describes block1.

Now that you have a clear vision of the different blocks it is easier to describe each one in the description/implementation area of the `architecture` of the simple counter module. Just remember to declare the internal signal in the declarative area of the `architecture`.

5 Develop the testbench

For this assignment you have to create the testbench from scratch. This is all the information needed to create it:

- The testbench entity has no ports, all data objects are internal.
- The testbench architecture must contain, in the declarative area ...
  - The component declaration of the module you want to test, the `simple_counter` counter. This declaration is the same as the `simple_counter entity` declaration changing the keyword to `component`.
  - A constant `TB_MAX_COUNT`, type `integer`, value 5. The counter's `MAX_COUNT` generic will be mapped to this constant, so changing the testbench constant value will change the maximum count of the tested counter.

[1] From both Figure 3a and Figure 3b the reference to the generic `MAX_COUNT` has been removed in order to improve the readability of the images.

  - The internal signals needed to map all counter inputs and outputs (`rst`, `clk`, `count`, and `mcount`). The type of these internal signals is the same as the ports of the counter, except the range of `count` is `0 to TB_MAX_COUNT`[2].

- The testbench architecture must contain, in the implementation area ...

  - An instance of the `simple_counter` counter with the instance name `dut`. In this instance the counter `MAX_COUNT` generic is mapped with the testbench constant `TB_MAX_COUNT`, and all counter ports are mapped with the testbench internal signals of the same name.
  - A simple `process` to create the clock as a fixed-frequency digital signal of, say, 20 ns period (10 ns high and 10 ns low).
  - Another `process` to create the reset input to the `dut`. Call this process *stimuli*; it has no sensitivity list in order to be able to include `wait` statements inside. The functionality of this process is detailed in subsection 5.1

5.1 The stimuli

When creating the stimuli or input vectors for a given dut you must try to cover all possible cases; if there is a huge number of input cases this becomes impractical and you must select enough cases to cover all the dut functionality.

The functionality to be tested is summarized below:

1. The counter resets asynchronously as soon as the `rst` input is asserted (low).
   - To test the counter reacts asynchronously to the reset this input must be asserted when there is no clock edge.
   - To test the `rst` input has higher priority than the `clk` input, the reset must be asserted during enough time to clash with clock edges.
2. The counter can be used without mapping the reset input or mapping the reset input with the `open` keyword. That is, we must verify that the counter can be used as if it hadn't a reset input.
3. The counter counts cyclically from `0` to the value mapped to the `MAX_COUNT` generic.

The VHDL code of the stimuli process can test only the first of the above items. The other two require modifications to the testbench code in order to create a new testbench version (with or without reset input, change the `TB_MAX_COUNT` constant value),

[2] As the constant `TB_MAX_COUNT` of the testbench will be mapped with the generic `MAX_COUNT` of the counter, the range of the testbench internal signal `count(0 to TB_MAX_COUNT)` will be the same as the the range of the counter port `count(0 to MAX_COUNT)`.
Can't use `MAX_COUNT` in the declaration of the testbench constant because `MAX_COUNT` is the generic of the `simple_counter` counter and hence accesible only inside the counter, but not accesible from the testbench.

compiling the modified testbench code and simulating the new version; all these steps are detailed in the tests section (subsection 5.3).

To test that the counter reacts asynchronously to the reset input, the code of the stimuli process must create the following waveform for the `rst` input:

1. Low during 3 clock cycles. This ensures the counter starts with the value 0, and that several clock edges arrive while the reset is asserted.

   This part of the process code is

   ```
   rst <= '0';
   for i in 1 to 3 loop
       wait until rising_edge(clk);
   end loop;
   wait for 2 ns;
   ```

   The code above uses a `wait` statement until the `clk` signal has a rising edge condition; this kind of waiting code help us to develop the stimuli process irrespective of the amount of time assigned to the clock period (20 ns or 100 ns is irrelevant, we don't have to change the code of the stimuli process).

   The final `wait` statement with a time period of 2 ns guarantees the reset signal is deasserted when there is no clock edge, so we can test the counter reaction in isolation.

2. Then high during `TB_MAX_COUNT+1` clock cycles. This ensures the counter has enough time to count from 0 to the maximum count and then 0 again. The way to code waiting for some clock cycles is similar to the code snippet of the first item.

3. Then low again during 3 clock cycles. This guarantees we can see the reaction of the counter to the reset input with no clock edge at the same time. The way to code waiting for some clock cycles is similar to the code snippet of the first item.

4. Then high again during 3 clock cycles. The way to code waiting for some clock cycles is similar to the code snippet of the first item.

5. The simulation is finally automatically stopped with a `assert` statement of `failure` level, as described in a previous assignment.

5.2 Expected results

Once we perform the testbench simulation and while reviewing the outcome the question that must be answered is: *Does the `simple_counter` counter behaves correctly?*

To answer this question we need to know what are the expected results; these change depending on the section of the stimuli process:

1. Low during 3 clock cycles. The counter must be held in reset during this simulation section, so `count` must be 0 and `mcount` must be '0'. When the `rst` deasserts 2 ns after the clock edge the counter must not change its value as no reset and no clock edge means "*Maintain the stored value*".

2. Then high during `TB_MAX_COUNT+1` clock cycles. The counter `count` output must change from 0 to 1 on the first clock edge of this simulation section, from 1 to 2 on the second, and so on, till it reaches the value `TB_MAX_COUNT` and then the next value must be 0 again. During this simulation section the `mcount` output must be '0' except while `count` has the value `TB_MAX_COUNT`.

3. Then low again during 3 clock cycles. The counter must be held in reset again during this part, so `count` must be 0 and `mcount` must be '0'. Expect same results as from item 1.

4. Then high again during 3 clock cycles. The counter must count again during this part, so expect the same results as obtained from item 2.

5.3 The tests

There are some functionality aspects that can only be tested modifying the testbench code. In particular, the use of the counter as if it hadn't a reset input and the use of the counter with different values of its `MAX_COUNT` generic.

We can't test all possible values of the `MAX_COUNT` generic, but testing some values will be enough; after all, what we are testing is if the code you developed to create the `simple_counter` contains some hardcode value (wrong) as maximum count or if it actually depends on the generic value (right).

The tests to be performed are described below. For each test you must recompile the testbench code, perform a simulation, make a screenshot of the simulation results that cover all sections of the stimuli process code, rename the image file as `simulation_1.png` in case of the test 1, `simulation_2.png` in case of the test 2, and so on, and save the image file into the project folder.

Test 1 Map the `rst` port of the counter with the `rst` internal signal of the testbench, and set the `TB_MAX_COUNT` constant of the testbench to the value 2.

The name of the screenshot image file is `simulation_1.png`; remember to store it into the project folder.

Test 2 Map the `rst` port of the counter with the `rst` internal signal of the testbench, and set the `TB_MAX_COUNT` constant of the testbench to the value 3.

The name of the screenshot image file is `simulation_2.png`; remember to store it into the project folder.

Test 3 Map the `rst` port of the counter with the `rst` internal signal of the testbench, and set the `TB_MAX_COUNT` constant of the testbench to the value 5.

The name of the screenshot image file is `simulation_3.png`; remember to store it into the project folder.

Test 4 Map the `rst` port of the counter with the `open` keyword, and set the `TB_MAX_COUNT` constant of the testbench to the value 10. In this test the counter must be counting at all rising edges of `clk`, irrespective of the `rst` value.

The name of the screenshot image file is `simulation_4.png`; remember to store it into the project folder.

6 Checklist

Once you are satisfied with the simulation results it is time to check the following list before uploading your work

1. Your VHDL file is called `simple_counter.vhd`.
   Please remember the names of all team members must appear at the start of this file in a VHDL comment in order to get credit for this assignment.

2. You have respected the `simple_counter` interface requirements in full: entity name, names and types of ports and generics.

3. You have respected the `simple_counter` implementation requirements in full: two separate blocks, one to describe the `count` output using a `process` with sensitivity list and a second block using a parallel `when` signal assignment to describe the `mcount` output.

4. You have create a VHDL testbench file to simulate the m design and help verify the functionality is correct.
   Please remember the names of all team members must appear at the start of this file in a VHDL comment in order to get credit for this assignment.

5. You have one screenshot per simulation (four total image files) where the input and output values are clearly visible.
   You have **neither edit nor crop** the screenshot image(s). If you need to zoom in in order for the values to be visible, grab several screenshots.

7 Submission

Once you have verified the checklist of the previous section you can proceed to submit your work onto the corresponding assignment page of the course website.

What to submit: A ZIP archive with the whole project folder and screenshot image files.