imec

The **Microelectronics Training Center**

The MTC is an initiative within the INVOMEC division

Industrialization & Training in Microelectronics

MTC

Lab-exercise

Lab 4:

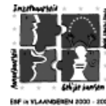
VHDL basics: counters

Cluster: Cluster1
Module: Module1c

Target group: Students

Version: 1.1
Date: 31/03/06
Author: Osman Allam
Modified by: Geert Vanwijnsberghe
History : Testbenches added

This material was developed with support of the European Social Fund.
ESF: Prevent and combat unemployment by promoting employability, entrepreneurship, adaptability and equal opportunities between women and men, and by investment in people.
http://www.esf-agentschap.be



For Academic Use Only

©IMEC2005 www.mtc-online.be

Introduction

Micro6 contains 4 counters; the program counter and the memory-stack pointers.

The program counter (PC) in the microprocessor is implemented as an up-counter. The count increment is constant because all instructions occupy the same number of memory locations.

The 3 memory-stack pointers inside the register file are updown counters.

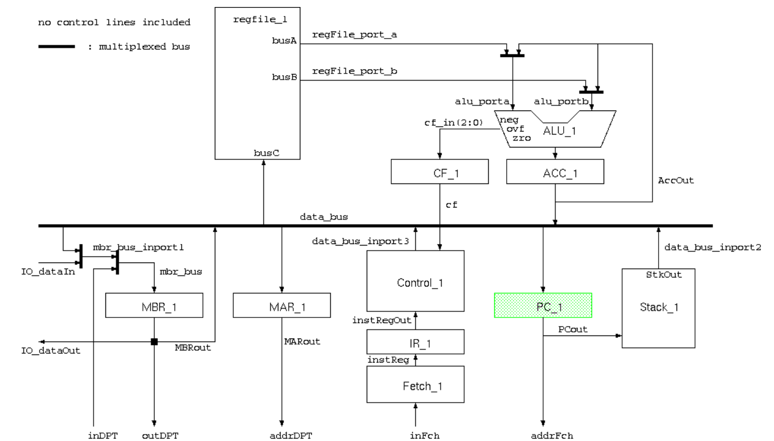


Figure 1: CPU architecture

Objectives

After completing this module you will be able to design elementary sequential circuits

Knowledge background

Basic VHDL knowledge

Classification

- Level: 1
- Duration: 30 minutes

Input

- VHDL template of the up counter (`counter.vhd`).
- VHDL template of the updown counter (`counter_updown.vhd`).

The lab

Up counter

In this exercise, you design an up-counter. The contents of the counter are incremented by one on the rising edge of the clock if the *increment* input is asserted. Provide a means to *load* new data into the counter. In case both the *increment* and the *load* control lines are asserted simultaneously, one of them has to be assigned priority over the other to determine the behavior of the counter.

There are two ways for assigning priorities in VHDL

1. Priority
This is modeled by if-elsif constructs.

```
If (high_prio_control = '1') then
  -- its action
elsif (2_high_prio_control = '1') then
  -- its action
…
elsif (low_prio_control = '1') then
  -- its action
end if;
```

(Arrow down the side of the code: Increased area and longer delays)

It is implicit in the sense that in if-statements conditions are checked sequentially, if a condition is true, the remaining conditions are not checked.

2. No priority

This is achieved by explicitly specifying an action for each possible combination of the control inputs. This is modeled in VHDL using a case statement. This technique may generate more complex logic in synthesis.

Example: 2 control inputs:

```
Signal ctrl : std_logic_vector; -- declare a signal
…
Case std_logic_vector'(c1 & c2) is
  When "00" => -- action when nothing is asserted
  When "01" => -- action when c2 is asserted
  When "10" => -- action when c1 is asserted
  When "11" => -- action when both are asserted
  When others => -- action in case of meta values
End case;
```

Note: c1 and c2 are of type std_logic.

You can think of other ways to resolve conflicting control actions!

There are two ways to write a VHDL model for counters that can be re-used in different sizes.

1. Using generics: the counter size is determined by the value of the generic, which can be assigned at instantiation. You can give generics default values.


```
Entity counter is
  Generic (size : positive := 8); – default size is 8
  Port (
    rst : in std_logic;
    Clk : in std_logic;
    ld  : in std_logic;
    inc : in std_logic;
    D   : in std_logic_vector (size – 1 downto 0);
    Q   : out std_logic_Vector (size – 1 downto 0));
End entity;
```

2. Using unconstrained ports: the counter size is automatically determined at instantiation by the width of the input and output signals connected to it.

```
Entity counter_en is
  Port (
    rst : in std_logic;
    Clk : in std_logic;
    ld  : in std_logic;
    inc : in std_logic;
    D   : in std_logic_vector;  -- unconstrained vector
    Q   : out std_logic_Vector);-- unconstrained vector
End entity;
```

However, the width of any internal signals should match the width of the external interface signals of the unit. This can be achieved by specifying the width of the internal signals in terms of the width of the external interface signals by using the attribute 'width or 'range. For example:

```
Signal count_int : std_logic_vector (D'range);
```

Or

```
Signal count_int : std_logic_vector (D'width – 1 downto 0);
```

Exercise: Up counter

Design an up-counter with unconstrained width. The contents of the counter are incremented by 1 every clock cycle when the increment (inc) input is asserted. The counter is loaded with the data on its inputs every clock cycle when the load (ld) input is asserted. The counter is reset asynchronously by the signal (rst).
The waveforms below show the priorities of different control inputs. Use the implicit priority method to describe the behavior of your counter.

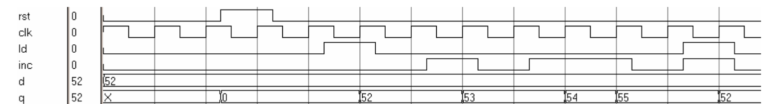


Figure 2: Waveform of up counter

Use the template provided in the `counter.vhd` file.

Investigate the testbench in the tb_counter.vhd file. You can see that an 8 bits wide counter is instantiated and that the clock is generated as a concurrent procedure. The expected output is stored in an array and verified in an extra process.
Compile and simulate the testbench. No errors should be logged.

Exercise: Updown counter

Design an updown counter. The contents are incremented by 1 every clock cycle when the increment (inc) input is asserted and they are decremented by 1 every clock cycle when the decrement (dec) input is asserted. The counter is loaded with the data on its inputs every clock cycle when the load (ld) input is asserted. The counter is reset asynchronously by the signal (rst).

The waveforms below show the priorities of different control inputs. Use the implicit priority method to describe the behavior of your counter.

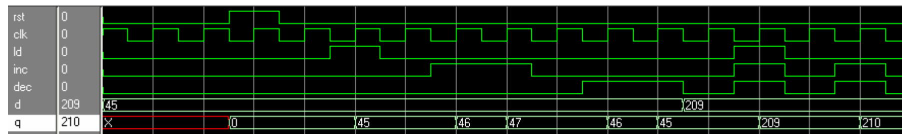


Figure 3: Waveform of updown counter

Use the template provided in the file `counter_updown.vhd. Use the template tb_counter_updown.vhd to create a testbench for a 9 bits updown counter. Compile and simulate.` No errors should be logged.