|  |  |
| --- | --- |
|  | |
|  | |
| Assignment 3 – 4 Bit Modulo Counter | |
| **Module code:** | **EG3205** |
| **Module name:** | **Programming Microelectronic and Multi-Core Systems** |
|  | |
| 1 March 2019 | |
|  | |
| **Author(s):** | **Nishat Tarannum** |
| **Student ID(s):** | **169018358** |
| **Degree:** | **MEng Software and Electronic Engineering** |
| **Tutor/Project supervisor:** | **Dr. Tim Pearce** |
|  | |
| **SUPERVISOR’S COPY/EXAMINER’S COPY [For 3rd/4th year projects only - delete as appropriate]** | |

By submitting this report for assessment I confirm that this assignment is my own work, is not copied from any other person's work (published or unpublished), and that it has not previously been submitted for assessment on any other module or course.

I am aware of the University of Leicester’s policy on plagiarism, and have taken the online tutorial on avoiding plagiarism.  I am aware that plagiarism in this project report may result in the application of severe penalties up to and including expulsion from the University without a degree.

Contents

[1 My Implementation 1](#_Toc2297780)

[a) Code for the test bench 1](#_Toc2297781)

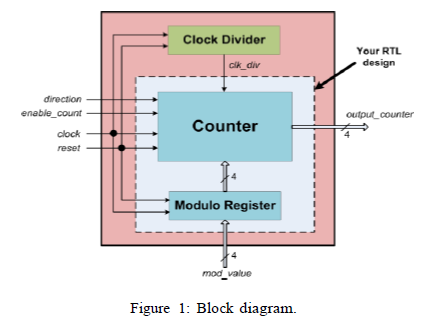
[b) Code for the board 3](#_Toc2297782)

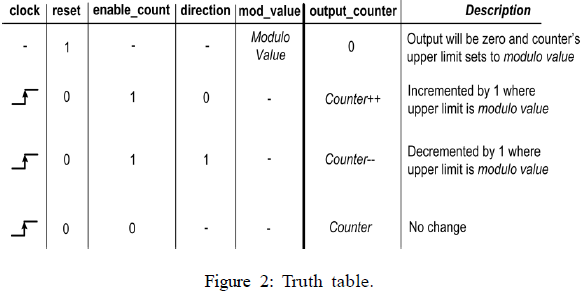
[2 Test Bench 4](#_Toc2297783)

[3 Conclusion 6](#_Toc2297784)

# My Implementation

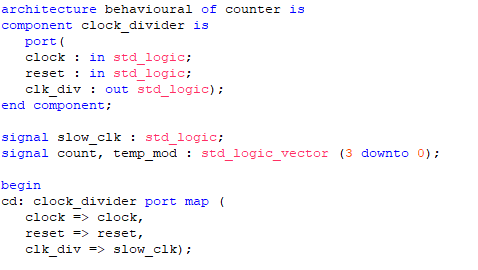
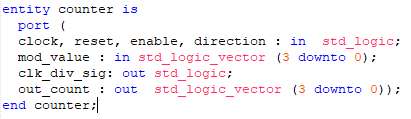
The assignment asked for a 4 bit up-down counter using a clock divider and modulo register. The clock divider was already implemented and the counter and modulo register had to be coded.





## Code for the test bench

For simulation slightly different code was written as this wasn’t being used with the board’s peripherals such as switches and push buttons. A synchronous reset was set to ensure that the output from the counter was not unpredictable, which may occur if an asynchronous reset is used and may lead to complications in design. The code is given below with annotations.



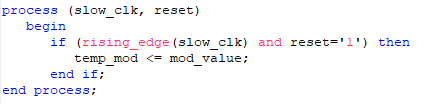
Three other signals are set up: the clock divider output is fed into the counter to which refers to slow\_clock , the count is used to hold the temporary value of the count e.g. if it is “1001” etc and, temp\_mod stores the set maximum value that the counter will count up to or down from.

The clock divider provided is instantiated as a component in the counter architecture, which takes a clock signal and outputs a slowed down clock signal.

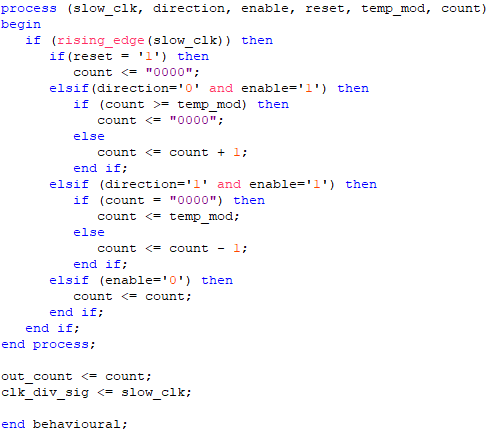
The port map of the clock divider is set up which assigns which of the inputs and outputs are connected to the inputs or outputs or signals of the counter.

First, the entity of the counter was set up with out\_count being the 4-bit output of the counter and clk\_div\_sig was used to see how the clock divider signal was working, it was used to help understand the test bench. Mod\_value is the value set to count up to or count down from.

The other signals are used to control the counter behaviour according to the truth table in Figure 2.



If there is a rising clock edge from the clock divider signal and the reset value is equal to 1 then the modulo register stores the modulo value to a signal called temp\_mod. The other part of the modulo register was implemented in the counter process where the output of the counter was set to “0000” when this happened. The behaviour can be seen from Figure 1. Although, nothing is defined in terms of the type of reset in the truth table in Figure 2, a synchronous reset was used in this code to ensure there were no unpredictable outputs from the counter.



If the reset is not equal to 1 then, the counter code first checks if the direction is 0 (for counting up) and count is enabled (set to 1).

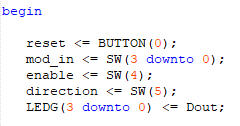
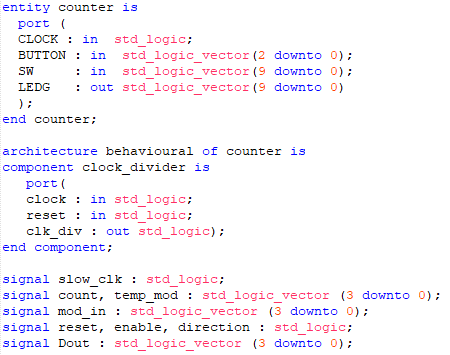
If the count value is incremented equal to or above the modulo value stored in temp\_mod then the count value is set to (“0000”) otherwise the count value continues to increment. The situation is vice versa for the next else if statement. When the direction is 1 (for counting down) and the count is enabled then if the count reaches “0000”, the value is set to the modulo value. Otherwise, the count value is decremented. Lastly, if count is disabled then count remains the same. The count value is output and also the clock divider signal, again purely for understanding the test bench.

### Code for the board

The code for the board was changed slightly, the overall operation is exactly the same in essence however, the synchronous reset was changed to an asynchronous one because when the reset button was being pressed (i.e. reset = ‘1’) the count was not being set to “0000”. The LEDs remained on (according to the last binary value) and would just behave like the enable switch. The LEDs would freeze and stop counting. However, after changing to an asynchronous reset, when the reset button was pressed, the LEDs would remain turned off and the modulo value was being able to be set without the LEDs turning on/off (i.e. without the counter still counting [which was the way it was working with a synchronous reset]). However, there may be implications due to using an asynchronous reset, if used inappropriately some signals may have metastable values due to propagation delays or may cause issues in circuits with feedback. Although so far there haven’t been any issues found in using this version on the board.

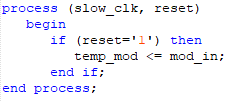
Here is the code highlighting the differences in implementation:

In this entity, clock is used as the input, a button is used for implementing the reset function, switches are used to input the modulo value, to change the enable and the direction of the counting and finally 4 LEDs are used to output the counter value.

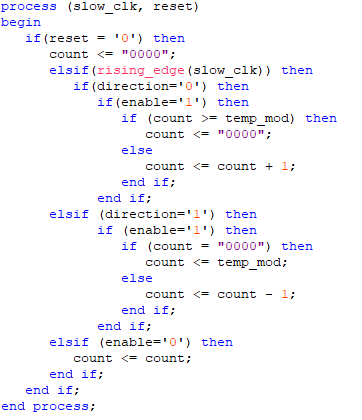


Specific bits of peripherals are assigned to the signals. Here one button is set up for the reset, 4 switches for inputting the modulo value, a switch for enabling the count, another switch for setting the direction and 4 LEDs for outputting the counter value.

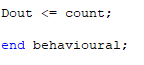
As before the clock divider is set up as a component, and other signals which will be needed are also declared.



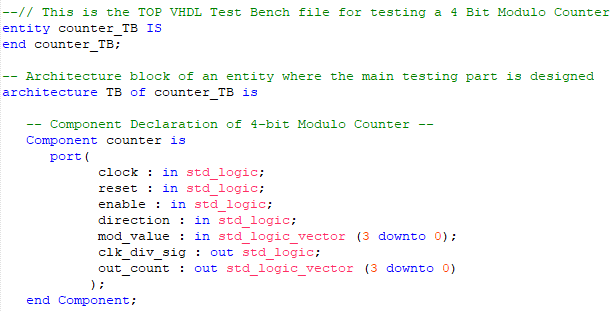
For the modulo register, an asynchronous reset has been used instead of the synchronous one in the code for the simulation. Both the asynchronous and synchronous functions essentially do the same thing, except this doesn’t require a clock signal to set the modulo value.



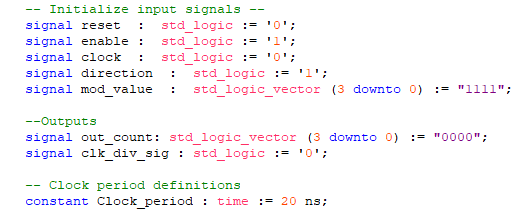
Again, this code is very similar to before except now an active low asynchronous reset has been used to implement the first row of the truth table in Figure 2. This ensures that the count is set to “0000” i.e. all the LEDs turn off if the reset button is pressed. The clock signal is checked for in the next elsif statement. The rest of the code is no different. The counter value is lastly output on the LEDs.



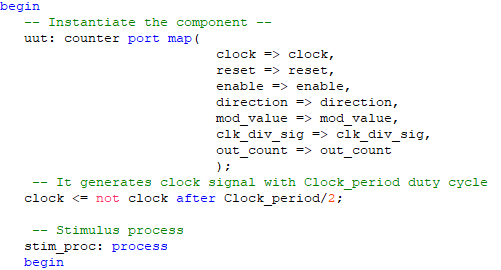
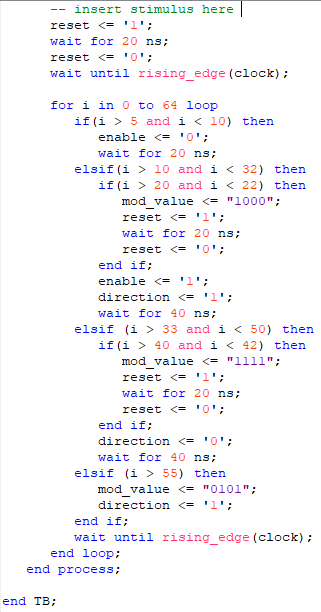
# Test Bench



In the test bench, the entity is set up and the counter component is set up in the architecture. A signal called clk\_div\_sig is also set as an output to see how the clock divider signal is functioning.



Input and output signals are set up to ensure that there are no undefined values at the beginning of the simulation. Furthermore, if the inputs are defined then this would mean that even if some values are not changed in the main testing, the test bench may still compile. Although that would be bad practice when testing, as values should be changed to see the behaviour. The counter port map is instantiated according to which signal is connected to what port. The clock is set up with a period of 20 ns, and a clock signal is generated.



First, the reset is changed from 1 to 0 and this is done to set the modulo value to “1111”, which is declared in the inputs. It waits for a rising clock edge as it is a synchronous reset. As shown in Figure 3 on the next page. Counting began from the next reset. Furthermore, the clock divider signal also goes to zero when reset is et to 1, which means the component also is working correctly.

All the other testing is done in the main for loop. Firstly, i is each clock pulse, so between the values of i = 6 and i = 9, enable is set to zero which should mean that counting stops and the value remains constant, shown in Figure 4.

The next thing tested is if the modulo value can be changed during execution. Again, a reset signal has been incorporated because otherwise this modulo value will only be set after the next reset which is at the beginning of the process. The modulo value is changed to “1000” on the 22nd clock pulse, counting is enabled and remains so for the rest of the simulation and the counter counts down between the 11th and the 33rd clock pulses. It can be seen that the clock divider returns to zero and so does the counter value when the modulo value is set, shown in Figure 5.

Again, the modulo value is changed during execution to “1111” (which is the maximum value) on the 42nd clock pulse and the counter counts up between the 34th and the 51st clock pulse. It can be seen that the clock divider returns to zero and so does the counter value when the modulo value is set, shown in Figure 6.

Lastly, the modulo value is changed to “0101” (which is the maximum value) on the 56th clock pulse and the counter counts down after this point. However, this time there is no reset so the modulo value is set on the next reset which is at the beginning of the process, shown in Figure 7.

# 

Figure 3 - Testing for setting of modulo value

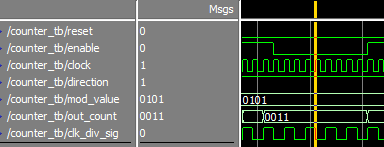


Figure 4 - Testing if output remains constant when enable is zero

# 

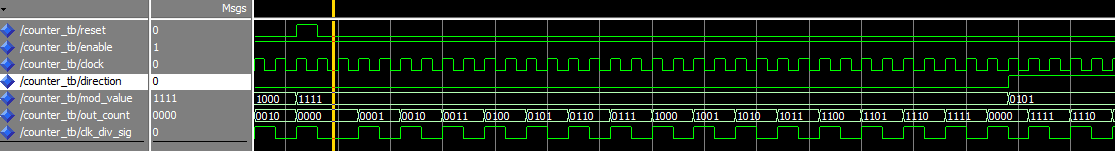
Figure 5 - Testing if modulo value is can be set, whether counter returns to 0000 and if the counter counts down

Figure 6 - Testing if the modulo value can be set, whether counter returns to 0000 and if the counter counts up

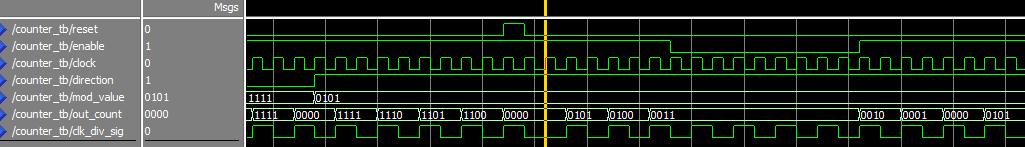


Figure 7 - Testing if the modulo value remains stored and is changed on the next reset, also change in the direction back to counting down

# Conclusion

These are the various tests run which show that the design with the synchronous reset is working correctly on the test bench. The code works correctly on the board however with an asynchronous reset.