**Laboratory III.2.- “Formal Verification - Improving the Design Process”**

Authors: Diego Hernandez Ramirez  
**Date: 01/12/19**

# Objective

Experience yourself the beauty of Model Checking Tools by analyzing the Suffix Implication and vacuity problems.

# Material

1. PC
2. SEDA tool, FPV App, Coverage App

# Practical development

Although “Formal Verification” is mostly known as a verification tool only, that is an erroneous perspective. When using assertions, the solvers helps you to check that your properties hold under any circumstances. But this is just an small use of the wide spectrum of what solvers can do.

In this laboratory, we will design some modules and verify that our progress is made correctly, with the **coverage** app of SEDA.

**Part I: The Cover Statement**

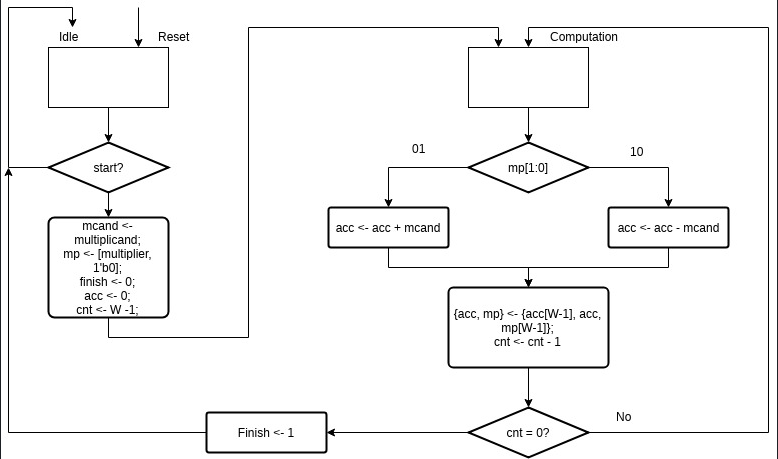
During this exercise, we will be using the **coverage** construct of SVA. To get an idea, please read [this blog post](https://fpgaparadox.com/index.php?title=Design_Exploration:_PCIe_History). It will help you to understand how to tackle the problems that will be presented here.

***Grade yourself:*** The source code of the PCIe is in the github repo.

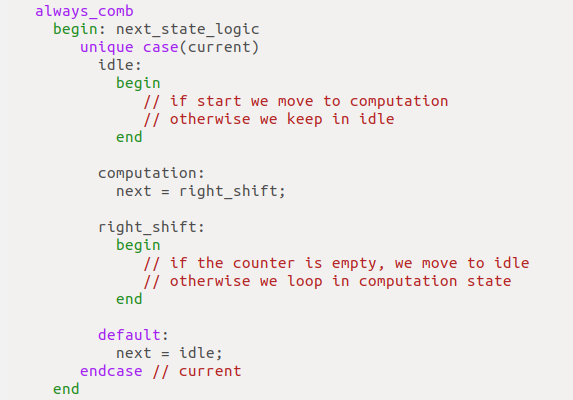
* Are you able to reproduce and fix the problems in the blog post?

**Part II: FSM Sanity**

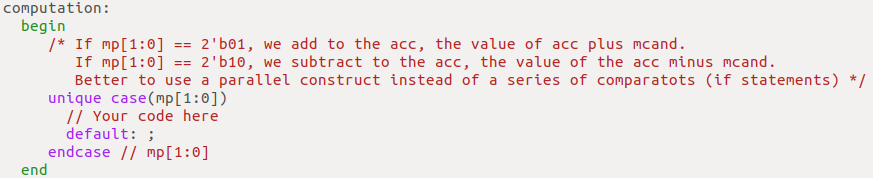
Retaking the *Booth Multiplier*, here is the flow chart of the algorithm. Assume you already started to translate this flow chart into a synthesizable model with SystemVerilog.

****

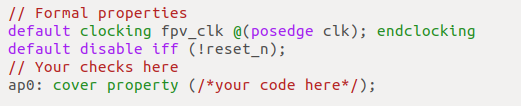
You still have to write some parts of the FSM:



And this part as well:



After completing the FSM, you next task is that show that the FSM is able to reach a point on which the **finish** port is asserted. In other words, that the FSM can move from idle to computation, to right\_shift, finish the shiftings and assert the **finish** signal. Use a **cover** property for that.



**Note:** Your *witness* must show from idle state to asserting finish. If your waveform only shows a small trace before reaching the final state, that does not count. We want to witness the whole transition, not only the shortest step.

**Part III: An Avalon Slave Component**

For the next lab, you are asked to complete a simple Avalon-MM slave. It is just a simple PWM controller. In broad words, whenever master asserts **write** pin and set the **address** of the transaction, with the value in **writedata**, the slave in that address should be able to receive and process that information.

Here is the [Avalon Specification](https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf) if you want to read more. For this simple slave, we do not need to check anything more than the values of **dvsr** and **duty** are stored in **csr1** and **csr0** respectively.

**Writes:**

Asserting both **write** and **address == 1’b0**, will store the sampled value in **writedata** into **csr0.**

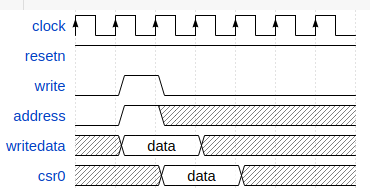
Asserting both **write** and **address == 1’b1**, will store the sampled value in **writedata** into **csr1.**

**Reads:**

Asserting both **read** and **address == 1’b0**, will send the sampled value in **csr0** (duty) to **readdata.**

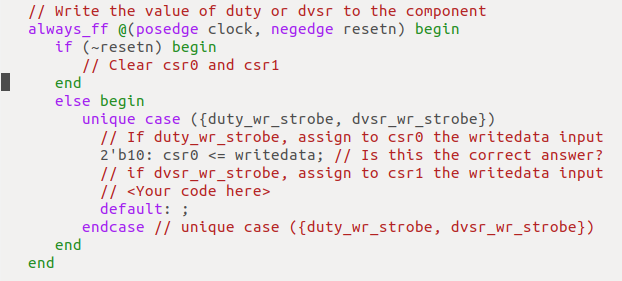
Asserting both **read** and **address == 1’b1**, will send the sampled value in **csr1** (divisor) to **readdata.**

Below is an example of the write process to **csr0**.

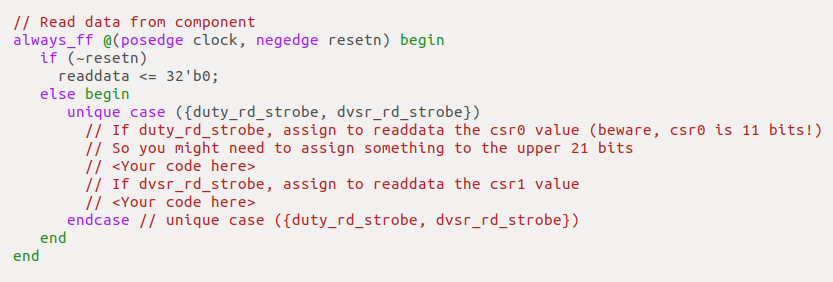
****

With that information, complete the Avalon Component:

**Complete the first part:**

****

**Also this another one:**

****

In a normal scenario, the good designer will use any Avalon transactional model to verify that the core is doing what is supposed to do, before releasing the project. The smart but lazy engineer may generate a small SoC with the NIOS-II core, write a trivial C/C++ program and review the core functionality. Both of those approaches are good, as long as you trust in the Avalon BFM, or your C/C++ skills.

Since we are learning Formal Methods, I have write for you two small properties at the end of the source code. **They must pass** when you finish to write the missing RTL constructs.

***Grade yourself:*** Explain both **write\_divisor\_value, write\_duty\_cycle** and the questions in the source code comments. Put your answers here.

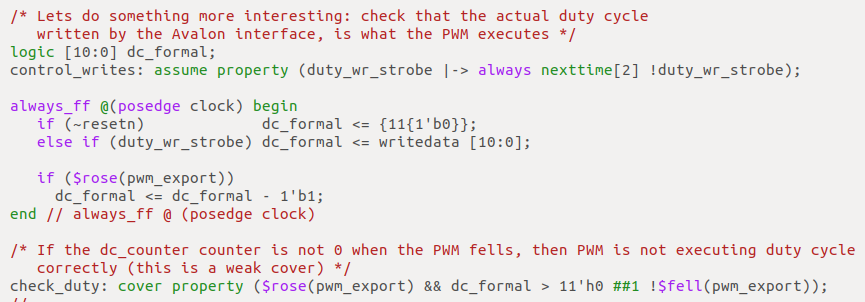
There is a new part, the **check\_duty** property. It clearly is different from any of the other properties we have used so far. This property uses **helper logic**.

Compared to SVA and simulation, where you can write any kind of complex assertions and sequences, in Formal, we try to keep everything as simple as possible.

For instance, instead of writing large temporal delays such as **##10000**, it is better to use the unbounded temporal delay version: **##[1:$]**, even with ***liveness properties*** the unbounded delay works fine.

In the case of **check\_duty**, it is better to set up some conditions and use registers to keep track of interesting values, instead of using complex sequence constructs.

Here is such property with the helper logic. Try to understand it. This is very useful when dealing with complex events.



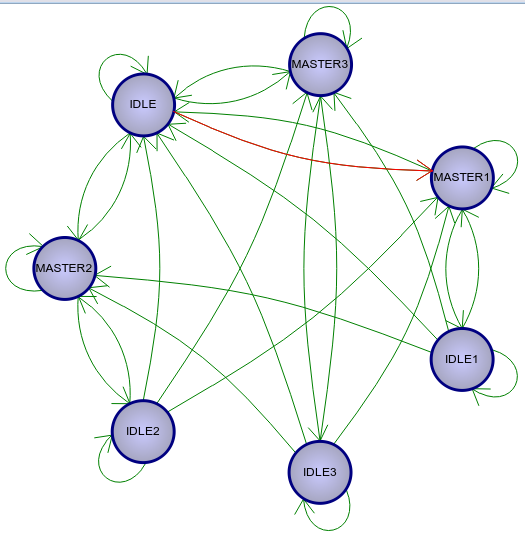
***Grade yourself:*** Until now, you have been developing the missing part of the RTL, and using the **write** properties to double check your code. It is time for you to check the **read** properties.

**Part III: Fair Bus Arbiter**

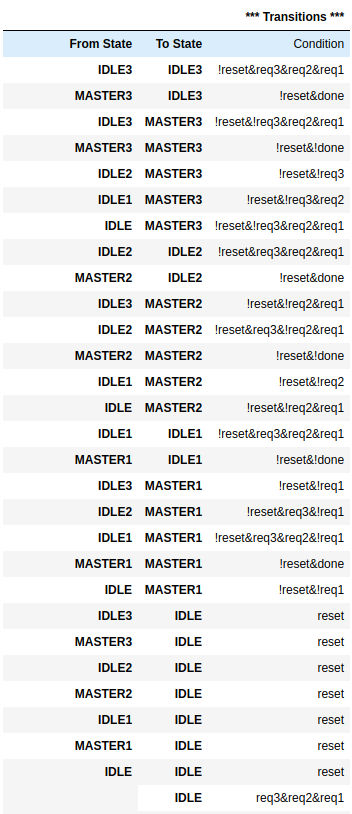
A bus arbiter is used in multi master systems, to arbiter the access to the bus (i.e., avoiding multiple masters accessing the resources at the same time, or that some master with access never leaves the bus for others that are waiting, or that some master is requesting the bus but never gets it).

Bus arbiters are good examples for both ***safety*** and ***liveness*** properties.

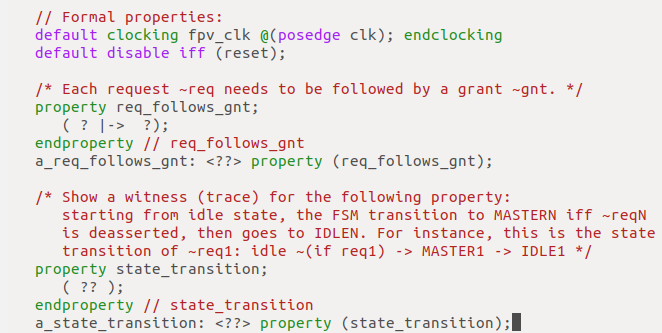
In the lab folder for session III, there is a bus arbiter that you can use during this lab. The FSM transitions is as follows:



And the transitions:



Your task is to prove the following properties:



***Grade yourself:*** Can you prove the fairness of the bus arbiter? (i.e., all requests are granted in equally distribution).