|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Document no. | | | | | TFE4171 – DDS 2 |
| TFE4171-Ex01-01 | | | | |
| Revision | | Date | | |
| 1.0 | | 06.02.2015 | | |
| Revision By | |  | | |
| Hope / Grindheim | |  | | |
| ­ | | | | |
| TFE4171 | | | | |
| Exercise I | | | | |
| ­ | | | | | |
| Author(s):  S.Grindheim  B.Hope |  | |  |  | |
| Revision Notes | * 1.0: Initial release | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
| ***Bengt Erik Hope / Svein Grindheim © 2014 Bergen, NORWAY*** | | | | | |

**🗎Page intentionally left blank**

* Preface

#### Abstract

*This is a short and informal paper answering questions listed in*

*Exercise 1 in the NTNU course TFE4171* – Design of Digital Systems 2.

Contents

[Abstract i](#_Toc411007318)

[**Section 1** Introduction 3](#_Toc411007319)

[**Section 2** Lab 1 3](#_Toc411007320)

[**Section 3** Lab 2 7](#_Toc411007321)

[**Section 4** Lab 3 11](#_Toc411007322)

[**Section 5** Lab 4 16](#_Toc411007323)

[**Section 6** Bibliography 21](#_Toc411007324)

# Introduction

This paper documents the authors’ answers to Exercise 1 in the TFE4171 course.

Due to the (somewhat confusing) structure of the exercise text, we will list the problems as stated in the exercise text in writing, whereas our answers follows in ms words *answer font* type and designated by **A:** <answer comes here>.

# Lab 1

1. Perform task 2 in the instructions. Report how you solved it.

**A:** In order to bind the design module (*dut.v*) to the property module (*dut\_property.v*) the following line was added to the test module (*test\_dut.v*):

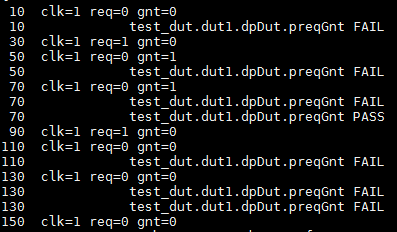


1. Perform task 3 (no\_implication) in the instructions. Answer the two questions for the no\_implication case:

The following code snippet represents the property (*pr1*) used in b):



Kernel output:



* Q: WHY IS THERE A FAIL -AND- A PASS AT TIME (70) ??

**A:** We notice that there is no implication operator in property *pr1* – meaning there is no antecedent to match before any checks begins!

We get vacuous FAILS because all we are ‘asking’ is that *req* be true at the positive clock edge and that two clks later *gnt* also must be true. We are not ‘asking’ to check the sequence only “if *req* is true at a positive edge of clk”. Hence, every clock that *req* is not true **the property FAILS**. This is also the case for the thread started at time 70.

Time 70 is a special case though, because in addition to the vacuous FAIL we get a PASS. **The property PASSES** at time 70 because two clock pulses earlier (@ time 30) *req* is HIGH so the property evaluation begins. *gnt* is found HIGH – as required by *pr1*,@ time 70 so the property evaluation PASSES.

* Q: WHY ARE THERE 2 FAILs AT TIME (130) ??

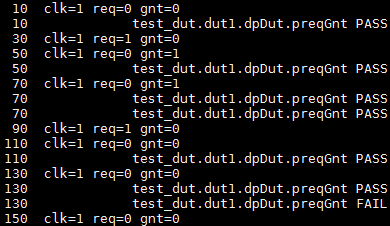
**A:** At time 130 we have two FAILS because:

1. *req* is LOW for the thread started @ time 130, so we get a vacuous FAIL (the reason for this was discussed above).
2. *req* was asserted HIGH which started a thread for property evaluation two clock cycles earlier (@ time 90). *gnt* is found to be LOW @ time 130 so the property evaluation fails.
3. Perform task 4 (implication) in the instructions. Answer the two questions for the implication case:

The following code snippet represents the property (*pr1*) used in c):



Kernel output:



* Q: WHY ARE THERE 2 PASSES AT TIME 70 ?

**A:** Now that we have an implication operator, the *antecedent (req)* must be true in order to evaluate the *consequent* (*gnt*). With this we have fixed the vacuous FAIL whenever *req* is LOW. However, we will now see a vacuous PASS when *req* is LOW instead. The reason for this is given by the LRM[[1]](#footnote-1) ‘‘*If there is no match of the antecedent sequence\_expr, then evaluation of the implication succeeds vacuously and returns true*’’.

Hence, whenever we see *req* LOW we get a vacuous PASS which triggers our PASS action block, and **we get the first PASS** displayed @ time 70 as a result.

**The second pass** is for the thread started @ time 30 because of the antecedent *req* being asserted HIGH. The property is evaluated to a PASS @ time 70 as *gnt* becomes HIGH as required.

* Q: WHY IS THERE A PASS -and- A FAIL AT TIME 130 ?

**A:**

1. There is a vacuous PASS @ time 130 because *req* is LOW. This is a result of using the implication operator without covering the PASS action block (ref. to previous answer).
2. There is a FAIL at time 130 because *gnt*=LOW two clock cycles after the antecedent *req* was asserted to HIGH (time 90), which started a thread for evaluating the property *pr1*.
3. Perform task 5 (implication\_novac) in the instructions. Do you get the expected result?

The following code snippets represents the properties (*pr1* and *pr2* respectively) used in d):

**A:** When running the simulator we get the expected result (no vacuous PASS/FAILS). This is because of the following lines in the property definitions:

**E:** Here it is used a ‘cover’ statement as well as an ‘else’ statement for the action block of the assertion. What this does is that it asks: ‘‘did you exercise this condition’’ or ‘‘did you cover this property’’.

So; if an assertion never fires, it can be because:

* 1. you don’t have any bug
  2. you never exercised the condition in the first place

Regarding the cover statement: If the condition is exercised but does not fail, we get that indication through the PASS action block we wrote with the cover statement. We also note that our simulator supports filtering vacuous pass for a 'cover' so the property *pr2* is not needed. We can simply use property *pr1* for 'cover' as well.

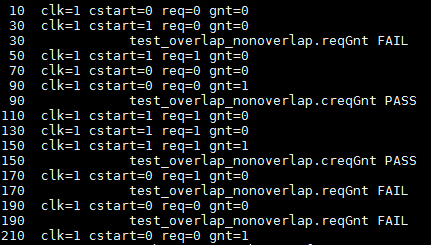
# Lab 2

1. Perform task 2 (overlap) in the instructions. Answer the following questions for the overlap case:

The code snippet below illustrates the property *pr1* with its sequence *sr1*. The property is utilized by the cover (*creqGnt*) and assert (*reqGnt*) statements in a) (for the same reasoning as mentioned in the last section: *pr1\_for\_cover* is not really needed in our case).



Kernel output:



* Q: WHY DOES THE PROPERTY FAIL at 30?

**A:** The assert property FAILS @ time 30 because the antecedent *cstart* is asserted HIGH but *req* remains LOW. As defined by the sequence *sr1* using a overlapping implication: *req* needs to go HIGH when *cstart* is asserted HIGH.

* Q: WHY DOES THE PROPERTY PASS at 90?

**A:** The property PASSES @ time 90 because the requirements set by *sr1* is fulfilled.

We have: Antecedent *cstart* and *req* is set HIGH two clock pulses ‘ago’ (@ time 50); evaluation completes @ 90 and PASSES since *gnt* is HIGH.

* Q: WHY DOES THE PROPERTY PASS at 150?

**A:** The property passes because the requirements in *sr1* is met: *gnt* goes HIGH two clock pulses after the antecedent *cstart* (time 110) triggers evaluation of the consequent. *req* was asserted HIGH as required by the overlapping implication (time 110) and *gnt*=HIGH two clock pulses thereafter (time 150).

* Q: WHY DOES THE PROPERTY FAIL at 170?

**A:** The property FAILS at time 170 because two clock cycles ago (time 130) the antecedent *cstart* matches – which starts evaluation of the consequent, and *req* was asserted HIGH (as required by the overlapping operator). The evaluation continues to time 170 where *gnt* is found to be LOW, which breaks the property requirement and the action block reports a FAIL.

**E:** Because the antecedent is held high over several clock pulses (time 110 through 150) the consequent will always be reevaluated at each clock pulse; we must ask our self if this is something we really want? Maybe it would be more sensible to evaluate the antecedent with a $rose(…) or $fell(…) statement.

* Q: WHY DOES THE PROPERTY FAIL at 190?

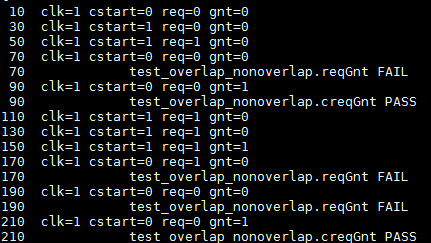
**A:** The property FAILS because *gnt* remains unchanged even though the antecedent and *req* was HIGH two clock pulses ago (time 150). This issue was also discussed above…

1. Perform task 3 (non\_overlap) in the instructions. Answer the following questions for the non\_overlap case:

The following code snippet illustrates the property (*pr1*) used in b):



Kernel output:



* Q: WHY DOES THE PROPERTY FAIL at 70?

**A:** The antecedent *cstart* is asserted HIGH at time 50 and the consequent is evaluated in a new thread. *req* is false at the subsequent rising clock edge (time 70) and the assertion is fired.

* Q: WHY DOES THE PROPERTY PASS at 90?

**A:** *cstart* is asserted HIGH at 30 and the consequent is evaluated. The cover property passes at time 90 as *req* and *gnt* are true at 50 and 90 respectively and all requirements specified in property *pr1* are fulfilled.

* Q: WHY DOES THE PROPERTY FAIL at 170?

**A:** *cstart* is asserted HIGH at time 110 and the consequent is evaluated. *req=*HIGH at 130, as required by the non-overlapping implication. However, the assertion property FAILS at 170 since *gnt*=LOW.

* Q: WHY DOES THE PROPERTY FAIL at 190?

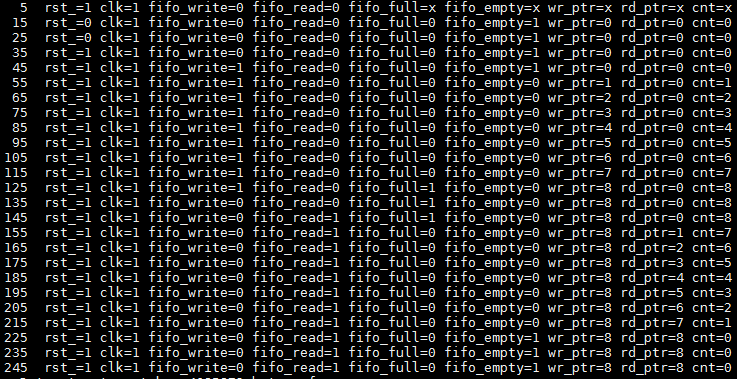
**A:** *cstart* is asserted HIGH at 130 and the consequent is evaluated in a new thread. *req* is true at 150 – as required by the non-overlapping implication, however the asserted property fails at 190 as *gnt* is false at 190.

* Q: WHY DOES THE PROPERTY PASS at 210?

**A:**The antecedent *cstart* is asserted HIGH at 150 and the consequent is evaluated. At the subsequent clock (time 170) req=HIGH as required by the non-overlapping implication. The cover property PASSES at 210 as *gnt*=HIGH as specified in the property *pr1*.

# Lab 3

1. Perform task 2 by running the nobugs case. Study carefully the log to understand the function of the FIFO. Remember that no assertions are active yet.



**A:** We analyze the functionality and summarize it as:

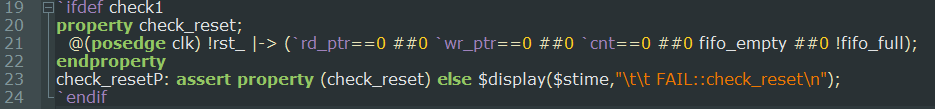
* fifo maintains a wr\_ptr and a rd\_ptr
* wr\_ptr increments by 1 everytime a write is posted to the fifo on a fifo\_write request
* rd\_ptr increments by 1 everytime a read is posted to the fifo on a fifo\_read request
* fifo maintains a 'cnt' that increments on a write and decrements on a read. It is used to signal fifo\_full and fifo\_empty conditions as follows
* When fifo 'cnt' is >= 7, fifo\_full is asserted
* When fifo 'cnt' is 0, fifo\_empty is asserted

1. Perform tasks 3 through 9 by adding one by one the given properties (that are already asserted) by removing the DUMMY code and adding your property code as specified. Run the check for each property and compare the log with the solution log. Report the result, change your property code if necessary, and report your solution.

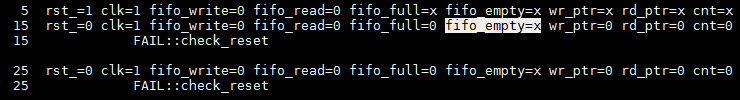
**A:**

**Task 3:**

We added the following property to the *fifo\_property.sv* file:



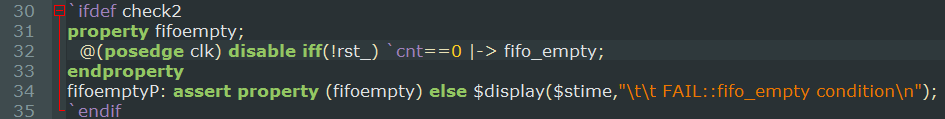
This gave us the following simulator output:



From the output we see that the property we introduced catches an RTL error where *fifo\_empty*  is left unspecified on reset (active low).

**Task 4:**

We added the following property to the *fifo\_property.sv* file:



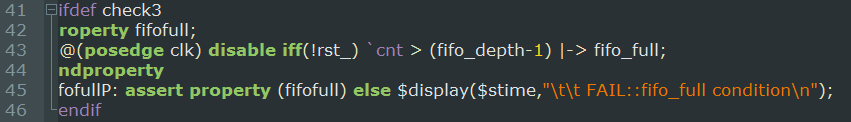
This gave us the following simulator output:



From the output we see that the property we introduced catches an RTL error where *fifo\_empty=0* when cnt=0. We would have wanted to see *fifo\_empty=1.*

**Task 5:**

We added the following property to the *fifo\_property.sv* file:



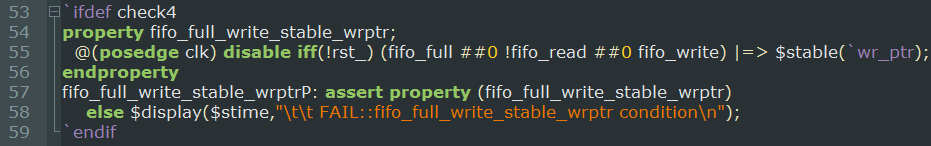
This gave us the following simulator output:



From the output we see that the property we introduced catches an RTL error where *fifo\_full=0* when cnt = 8. We would have wanted to see *fifo\_empty=1, because the fifo is full.*

**Task 6:**

We added the following property to the *fifo\_property.sv* file:



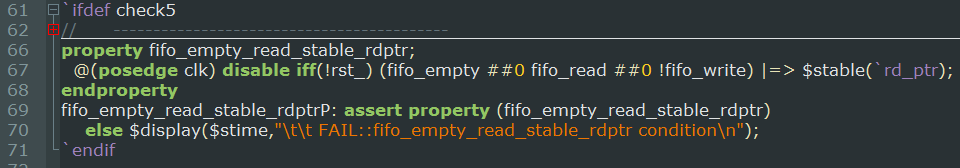
This gave us the following simulator output:



From the output we see that the property we introduced catches an RTL error where *wr\_ptr* increments when *fifo\_full*=1, fifo\_write=1 and *fifo\_read*=0. We would have wanted to see *wr*\_*ptr* stable*.*

**Task 7:**

We added the following property to the *fifo\_property.sv* file:



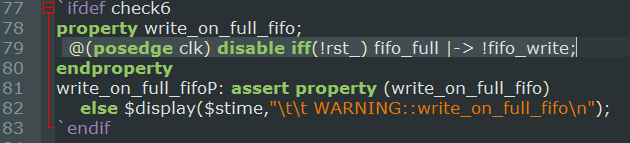
This gave us the following simulator output:



From the output we see that the property we introduced catches an RTL error where *rd\_ptr* increments when *fifo\_empty*=1, *fifo*\_*write*=0 and *fifo\_read*=1. We would have wanted to see *rd*\_*ptr* stable*.*

**Task 8:**

We added the following property to the *fifo\_property.sv* file:



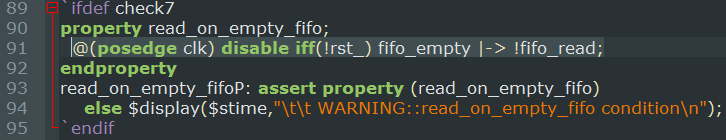
This gave us the following simulator output:



From the output we see that the property we introduced warns us when *fifo\_full=1* and *fifo\_write=1* on the same rising clock edge. We should not write to the fifo when it is full.

**Task 9:**

We added the following property to the *fifo\_property.sv* file:



This gave us the following simulator output:



From the output we see that the property we introduced warns us when *fifo\_empty=1* and *fifo\_read=1* on the same rising clock edge. We should not read from the fifo when it is empty.

# Lab 4

1. Perform task 2 by running the nobugs case. Study carefully the log to understand the function of the Counter. Remember that no assertions are active yet.



**A:** We analyze the functionality and summarize it as:

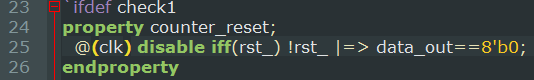
* The counter has 8 bit data input and 8 bit data output
* When ld\_cnt\_ is asserted (active Low), data\_in is loaded and output to data\_out
* When count\_enb (active High) is enabled (high) and
* updn\_cnt is high, data\_out = data\_out+1;
* updn\_cnt is log, data\_out = data\_out-1;
* When count\_enb is LOW, data\_out = data\_out:

1. Perform tasks 3 through 5 by adding one by one the given properties (that are already asserted) by removing the DUMMY code and adding your property code as specified. Run the check for each property and compare the log with the solution log. Report the result, change your property code if necessary, and report how you solved it.

**A:**

**Task 3:**

We added the following property to the *counter\_property.sv* file:



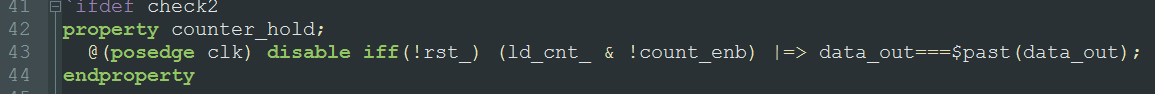
This gave us the following simulator output:



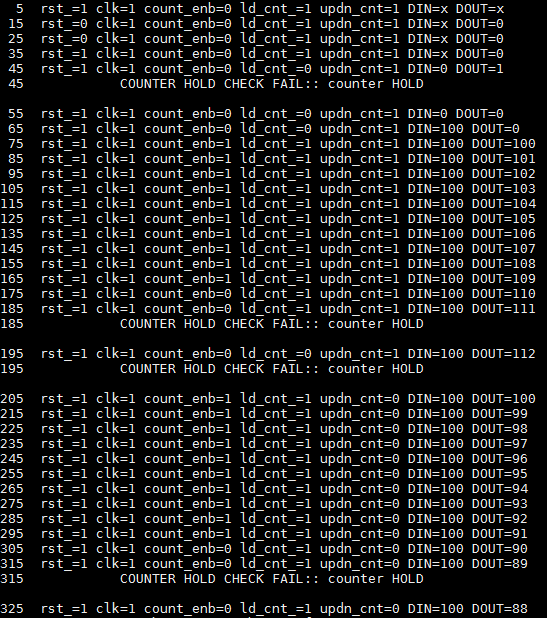
From the output we see that the property we introduced FAILS when *DOUT* is undefined during a reset (*rst\_=0*).

**Task 4:**

We added the following property to the *counter\_property.sv* file:



This gave us the following simulator output:

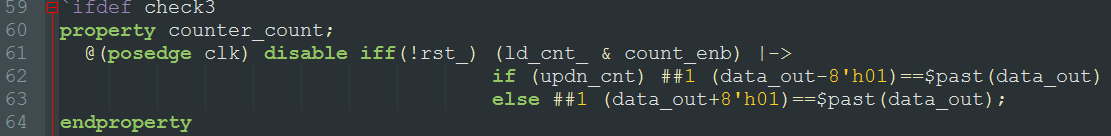


From the output we see that the property we introduced matches the solution log.

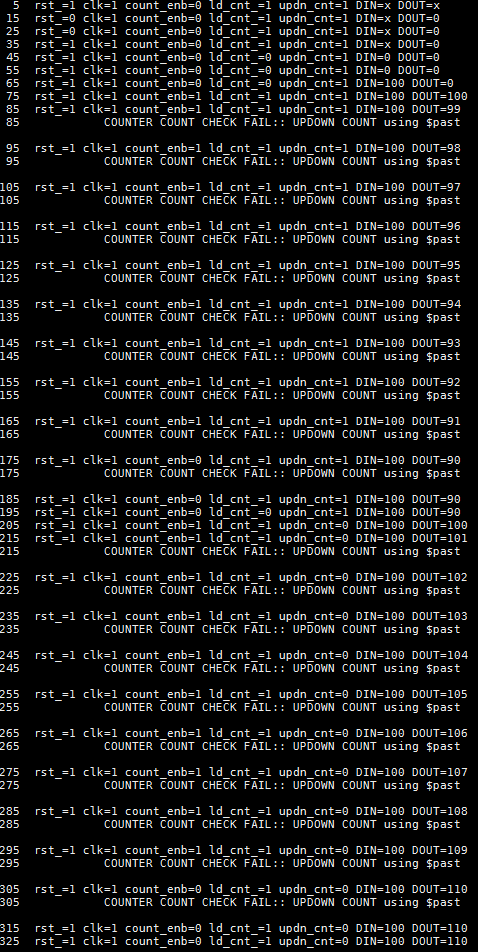
Namely; If ld\_cnt\_ is deasserted (==1) and count\_enb is not enabled (==0), data\_out will HOLD it's previous value. Property is disabled 'iff (!rst)'.

**Task 5:**

We added the following property to the *counter\_property.sv* file:



See next page for simulation output.



From the output we see that the property we wrote introduces assertion as specified. Namely; if ld\_cnt\_ is deasserted (==1) and count\_enb is enabled (==1) that if updn\_cnt==1 the count goes UP and if updn\_cnt==0 the count goes DOWN.

Property is disabled 'iff (!rst)'.

# Bibliography

Mehta, A. B. *System Verilog Assertions and Functional Coverage.* Springer.

1. **L**anguage **R**eference **M**anual [↑](#footnote-ref-1)