|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Document no. | | | | | TFE4171 – DDS 2 |
| TFE4171-Ex02-01 | | | | |
| Revision | | Date | | |
| 1.0 | | 13.02.2015 | | |
| Revision By | |  | | |
| Hope / Grindheim | |  | | |
| ­ | | | | |
| TFE4171 | | | | |
| Exercise II | | | | |
| ­ | | | | | |
| 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 2 in the NTNU course TFE4171* – Design of Digital Systems 2.

Contents

[Abstract i](#_Toc411382695)

[**Section 1** Introduction 3](#_Toc411382696)

[**Section 2** Lab 5 3](#_Toc411382697)

[Task 2: 3](#_Toc411382698)

[Task 3: 3](#_Toc411382699)

[Task 4: 5](#_Toc411382700)

[Task 5: 6](#_Toc411382701)

[Task 6: 7](#_Toc411382702)

[**Section 3** Lab 6 8](#_Toc411382703)

[Task 2: 8](#_Toc411382704)

[Task 3: 9](#_Toc411382705)

[Task 4: 9](#_Toc411382706)

[Task 5: 10](#_Toc411382707)

[Task 6: 11](#_Toc411382708)

[**Section 4** Lab 7 12](#_Toc411382709)

[ Appendix A – ex-1.v source file 19](#_Toc411382710)

# Introduction

This paper documents the authors’ answers to Exercise 2 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 5

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

**A:**

##### Task 2:

We ran design without any bugs in it (using *run\_nobugs* file). By familiarize our self with the design we can summarize the requirements as follows:

* dValid must remain asserted for minimum of 2 clocks but no more than 4 clocks.
* 'data' must be known when 'dValid' is High.
* 'dack' going high signifies that target have accepted data and that master must deassert 'dValid' the clock after 'dack' goes high.

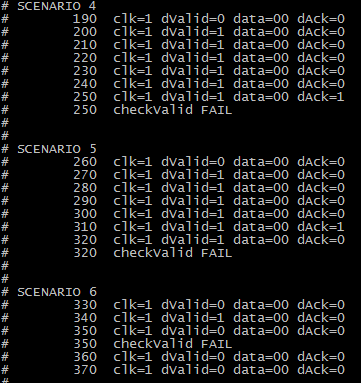
We note that the pulse diagram shown in SVALAB\_LINUX\_LAB5.pdf illustrates positive edge semantics, but the actual design file (*bus\_protocol.v*) operates sequentially on negative edge of clock. We take this into consideration when performing the tasks in this lab.

1. […] Perform task 3 (check1) and create the check1 assertion according to the instructions […]:

##### Task 3:

The following code snippet represents the property (*checkValid*) used in chek1:

Kernel output:



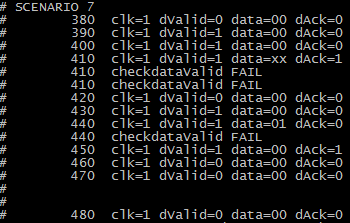
We see that the kernel output matches the solution provided. To make sure that *dValid* stays high consecutively for minimum two and maximum four clocks, we used the consecutive repetition operator ***\**** with range specification *[n:m]*. This way when the consequent is evaluated we detect unwanted behavior.

* In **Scenario 4** we detect that *dValid* stays HIGH @ 250 – whereas here it should have deasserted.
* In **Scenario 5** we detect that *dValid* stays HIGH @ 320 – whereas here it should have deasserted.
* In **Scenario 6** we detect that *dValid* is deasserted @ 350 – whereas here it should have stayed asserted for at least one more clock.

##### Task 4:

The following code snippet represents the property (*checkdataValid*) used in check 2:

Kernel output:



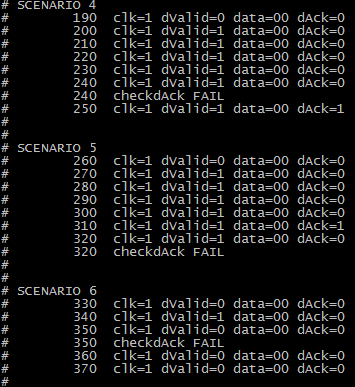
We see that the kernel output matches the solution provided. Here we utilize the *$isunknown(..)* method to check that *data* is not unknown AND remains stable after *dValid* goes high until the qualifying event $rose(*dAck)*. This triggers the following assertions in **Scenario 7**:

* Two FAILS @ 410, where the first is because data is not stable and the second happens because *data* evaluates to an unknown state.
* One FAIL @ 440 because *data* does not stay stable after the antecedent *dValid* triggers evaluation.

##### Task 5:

The following code snippet represents the property (*checkdataValid*) used in check 3:

Kernel output:



We see that the kernel output matches the solution provided. Here we first check if *dValid* has risen, we then evaluate the consequent which states that *dAck* must be deasserted AND *dValid* must be asserted for the 3 first clocks after the non overlapping implication. One clock later *dAck* must have risen and one clock after that *dValid* must have been deasserted. This triggers the following assertions:

* **Scenario 4:** FAILS @ 240, *dAck* is not asserted within the timing restrictions.
* **Scenario 5**: FAILs @ 320, *dValid* is not deasserted one clock after *dAck* has been asserted.
* **Scenario 6**: FAILS @ 350, *dValid* is deasserted before the minimum of 2 clocks after it rose.

1. Perform task 6 (checkall) in the instructions. Report how it works wrt. several bugs exposed by several assertions.

##### Task 6:

We ran the *run\_checkall* file. We cross-referenced it with check1 through check3 in order to confirm that all bugs were detected – and they were!

This lab has exposed the advantages in splitting up bug detection into smaller assertions, where each property could represent a specific sequence triggering a selected signal in the design.

# Lab 6

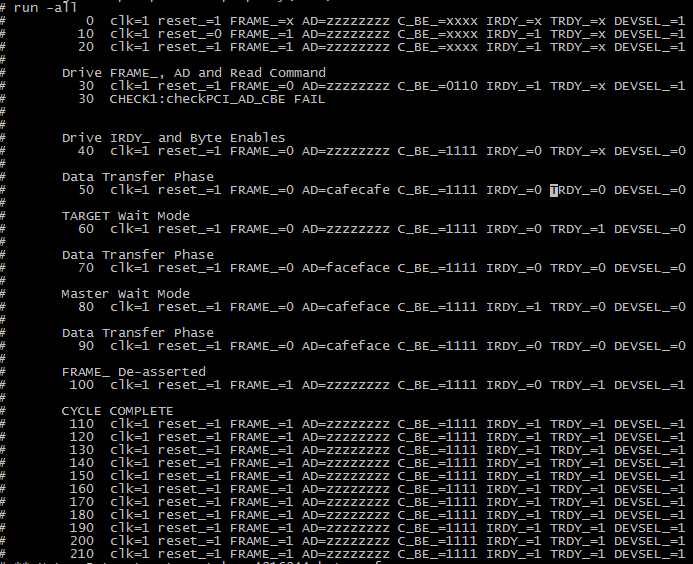
1. […]Perform task 2 (check1)[…]

**A:**

##### Task 2:

The following code snippet represents the property used in check 1:

Kernel output:



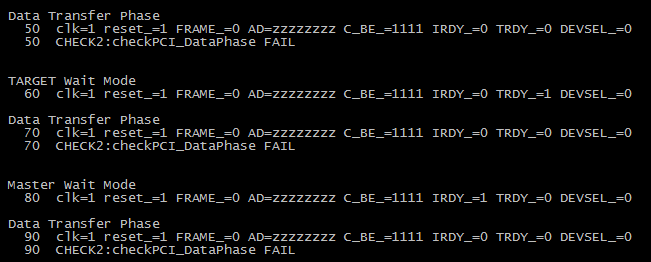
We see that the kernel output matches the solution provided. Here we utilize the *$isunknown(..)* method to check that AD or C\_BE\_ cannot be unknown when FRAME\_ transitions from 1->0. This triggers the following assertions:

* One FAIL @ 30 because *AD*  is unknown when FRAME\_ transitions from 1->0.

##### Task 3:

The following code snippet represents the property used in check 2:

Kernel output:



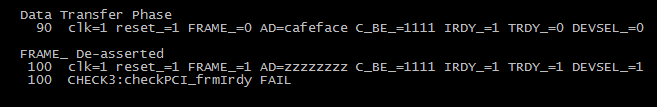
We see that the kernel output matches the solution provided. Here we also utilize the *$isunknown(..)* method to check that AD or C\_BE\_ cannot be unknown when *IRDY\_* AND *TRDY\_*  are asserted LOW. This triggers the following assertions:

* FAIL @ 50 and 90 because *AD*  is unknown when when *IRDY\_* AND *TRDY\_* =LOW.

##### Task 4:

The following code snippet represents the property used in check 3:

Kernel output:



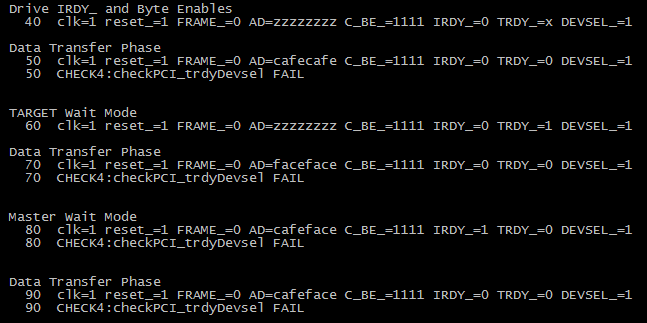
We see that the kernel output matches the solution provided. FRAME\_ can go High only if IRDY\_ is asserted. This triggers the following assertions:

* FAIL @ 100 because IRDY\_ is deasserted when FRAME\_ transition from 0->1.

##### Task 5:

The following code snippet represents the property used in check 4:

Kernel output:



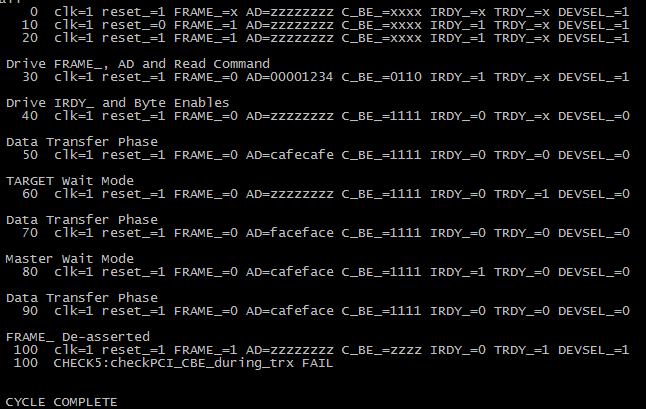
We see that the kernel output matches the solution provided. TRDY\_ can only be asserted (low) if DEVSEL\_ is asserted (low). This triggers the following assertions:

* FAIL @ 50, 70, 80 and 90 because TRDY\_ is asserted (low) when DEVSEL\_ is deasserted (high).

##### Task 6:

The following code snippet represents the property used in check 5:

Kernel output:



We see that the kernel output matches the solution provided. Once the cycle starts (at FRAME\_ assertion) C\_BE\_ should not float until FRAME\_ is deasserted. This triggers the following assertions:

* FAIL @ 100 because as FRAME\_ is deasserted (high), C\_BE\_ should not floats (tristate/high impedance).

# Lab 7

**A:**

We ran design without any bugs in it (using *run\_check* file) to familiarize us with the design.

The output was consistent with the functionality description provided in the exercise text referenced below:

*[…]*

*When validi=1 in three consecutive clk cycles, compute data\_out=a\*b+c where a is data\_in two clk cycles ago, b is data\_in one clk cycle ago, and c is the current data\_in. Also set valido=1 when this occurs to flag valid data\_out. Else valido=0 which means data\_out is not valid.*

*[…]*

We then added the following code snippet to the property and ran check 1:

Kernel output:



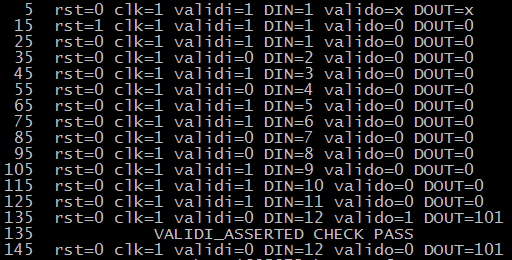
Here we only needed a simple check where *rst* was defined as the antecedent, which would trigger evaluation of *data\_out*=0x00000000. This triggers the following assertion:

* Reset check PASS @ 15; rst asserted HIGH, DOUT emptied;

The following code snippet represents the property used in check 2:



Kernel output:



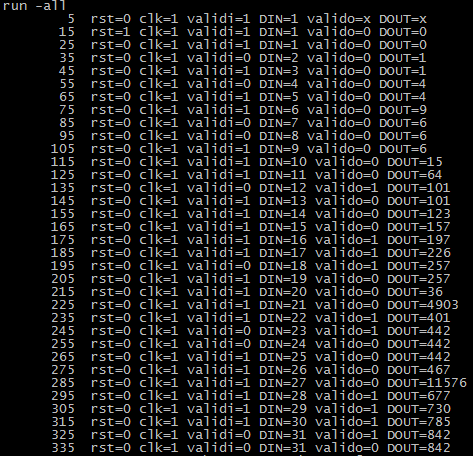
We first check if *validi* is asserted for three consecutive clocks as the antecedent[[1]](#footnote-1) then we check the consequent, namely if *valido* is asserted the following clock . This triggers the following assertion:

* Validi check PASS @ 135; *valido* asserted HIGH after three consecutive asserted *validi*;

The following code snippet represents the property used in check 3:



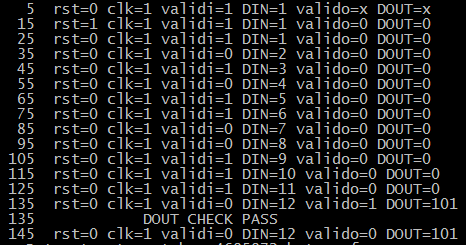
Kernel output:



Here we use $past(*sig, numClock*) to verify that *valido* is not asserted high unless *validi* has been high for three consecutive clocks. As we can see from the kernel output above, this assertion check will not trigger any action blocks as there are no bugs to detect. Note that we have not included a PASS printout as this would only clutter our output.

The following code snippet represents the property used in check 4:

Kernel output:

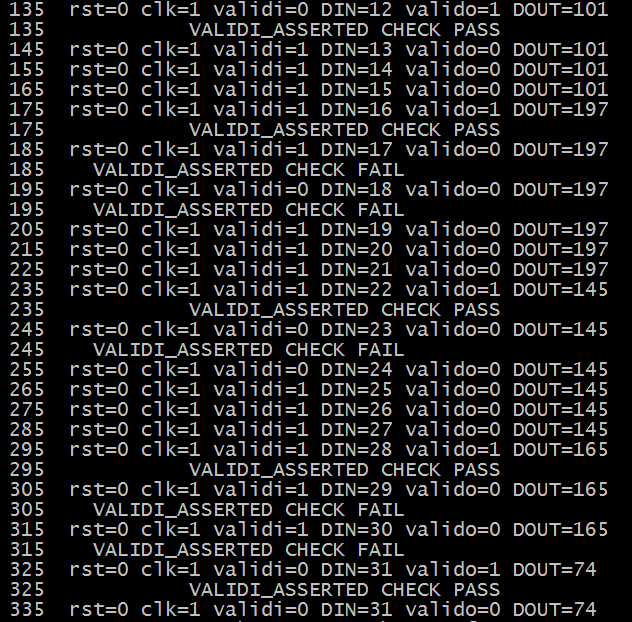


Here we again use $past(*sig, numClock*) to get previous values of DIN and compare them with the contents of DOUT when valido is asserted high. This triggers the following assertion:

* DOUT check PASS @ 135; *valido* asserted HIGH, contents of DOUT compared to sampled values of DIN;

We added the commented stimulation to the *test-ex2-1.sv* file. We now have new stimulation from time 155 and out.

Kernel output:

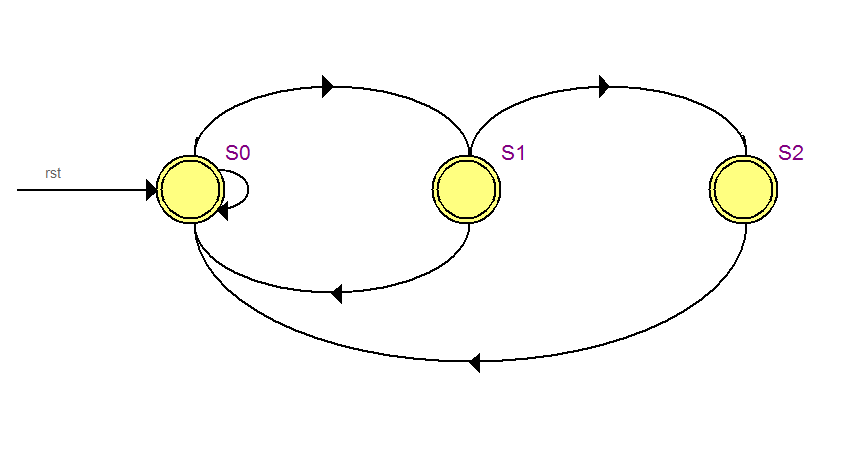


The new stimulus outputs triggers several assertions:

* PASS @ 175, 235, 295 and 325: *validi*  stays asserted for 3 consecutive clocks; next clock *valido*=HIGH as design requires.
* **More interestingly:**  FAIL @ 185, 195, 245, 305 and 315 because of overlapping sequences in our property *validi\_asserted*. The sequence is re-evaluated at each time *validi* = high. However *valido*  will of course not be high in these cases (except for an design glitch, which is not the case in our situation)

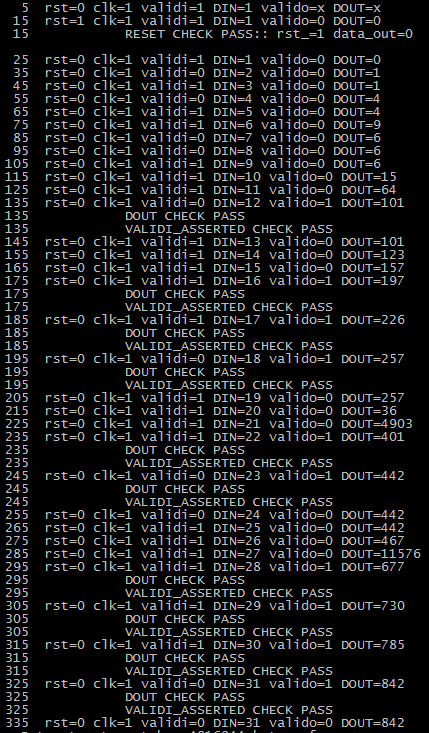
We will now fix the functional fault that was exposed in the previous task – that is the issue regarding overlapping sequences when checking the validity of *valido*=high. The design module found in the *ex2-1.v* was updated….

State machine diagram of the final implementation:



The first thing we needed was a way to store previous values of *data\_in* when overlapping sequences occurs. We introduced two more variables to store values from *data\_in*. Since the calculation of the output comprises of three operations, one for each state, we only need three variables(*a1, a2 and a3*) in total to handle storing of previous values. The next challenge was to assert *valido* at the correct time. This was solved by introducing a simple *counter* in the module. The counter keeps track of how many consecutive times *validi* is asserted. Every state is “linked” to every third number from their base number, i.e. S0: 0, 3, 6, 9…, S1: 1, 4, 7, 10…, S3: 2, 5, 8, 11… We then use the modulus operator to determine when a given calculation of a sequence is done and assert *valido*. We also check for *counter* >2 in the first two states to make sure that *valido* is not asserted in cases when *validi* is asserted two or less consecutive times in a row. Refer to Appendix A for source code.

Kernel output after running all assertion checks (*cmd: run\_check 5*):



* Appendix A – ex-1.v source file

GNU nano 2.0.9 File**:** ex2**-**1.v

/\*

\* ex2\_1

\*

\* Purpose:

\* - Reset on rst=1

\* - When validi=1 three clk's in a row, compute data\_out=a\*b+c

\* where a is data\_in on the first clk, b on the second and c

\* on the third. Also set valido=1. Else valido=0 which means

\* data\_out is not valid.

\*/

**module** ex2\_1 **(**

**input** clk**,** rst**,** validi**,**

**input** **[**31**:**0**]** data\_in**,**

**output** **logic** valido**,**

**output** **logic** **[**31**:**0**]** data\_out

**);**

**enum** **{**S0**,** S1**,** S2**}** state**,** next**;**

**logic** **[**31**:**0**]** a1**,** a2**,** a3**;**

**logic** **[**31**:**0**]** counter**;**

**initial** **begin**

state **=** S0**;**

**end**

**always\_ff** **@(posedge** clk **or** **posedge** rst**)** **begin**

**if** **(**rst**)** **begin**

data\_out **<=** 32'b0**;**

valido **<=** 1'b0**;**

state **=** S0**;**

counter **=** 0**;**

a1 **=** 0**;**

a2 **=** 0**;**

a3 **=** 0**;**

**end**

**else** **begin**

**case** **(**state**)**

// S0

S0**:** **begin**

//valido <= 1'b0;

**if** **(**validi**)**

**begin**

a1 **=** data\_in**;**

a2 **+=** data\_in**;**

a3 **\*=** data\_in**;**

data\_out **<=** a2**;**

**if(**counter **>** 2 **&&** **(**counter **%** 3 **==** 0**))**

**begin**

valido **<=** 1'b1**;**

**end**

next **=** S1**;**

counter **+=** 1**;**

**end**

**else**

**begin**

next **=** S0**;**

counter **=** 0**;**

valido **<=** 1'b0**;**

**end**

**end**

// S1

S1**:** **begin**

**if** **(**validi**)**

**begin**

a1 **\*=** data\_in**;**

a2 **=** data\_in**;**

a3 **+=** data\_in**;**

data\_out **<=** a3**;**

**if(**counter **>** 2 **&&** **(**counter **%** 3 **==** 1**))**

**begin**

valido **<=** 1'b1**;**

**end**

next **=** S2**;**

counter **+=** 1**;**

**end**

**else**

**begin**

next **=** S0**;**

counter **=** 0**;**

valido **<=** 1'b0**;**

**end**

**end**

// S2

S2**:** **begin**

**if** **(**validi**)**

**begin**

a1 **+=** data\_in**;**

a2 **\*=** data\_in**;**

a3 **=** data\_in**;**

data\_out **<=** a1**;**

**if(**counter **%** 3 **==** 2**)**

**begin**

valido **<=** 1'b1**;**

**end**

next **=** S0**;**

counter **+=** 1**;**

**end**

**else**

**begin**

next **=** S0**;**

counter **=** 0**;**

valido **<=** 1'b0**;**

**end**

**end**

**endcase**

state **=** next**;**

**end**

**end**

**endmodule**

1. Here we should note that since we are not using $rose(*validi*) simulation performance can be effected due to a large number of threads can potentially be spawned [↑](#footnote-ref-1)