

# System Verilog Assertions

## LAB Material

**Ashok B. Mehta**

<http://www.defineview.com>

© 2006-2008

# Copyright Notice

---

## Copyright Notice

© 2006-2009

The material in this training guide is copyrighted by DefineView Consulting/Ashok B. Mehta of Los Gatos, California. All rights reserved. No material from this guide may be duplicated or transmitted by any means or in any form without the express written permission of Ashok B. Mehta

DefineView Consulting

<http://www.defineview.com>

Ashok B. Mehta

501 Pine Wood Lane

Los Gatos, CA 95032

(408) 309-1556

Email: [ashok@defineview.com](mailto:ashok@defineview.com)

Verilog is a registered trademark of Cadence Design Systems, San Jose, California.

---

Lab 1 ...

how to 'bind' a design module  
with it's properties ...

# LAB 1 : Basic Property

## LAB Overview

*This LAB is to help you understand how to 'bind' a design module to a property module (which contains assertions for the design module.)*

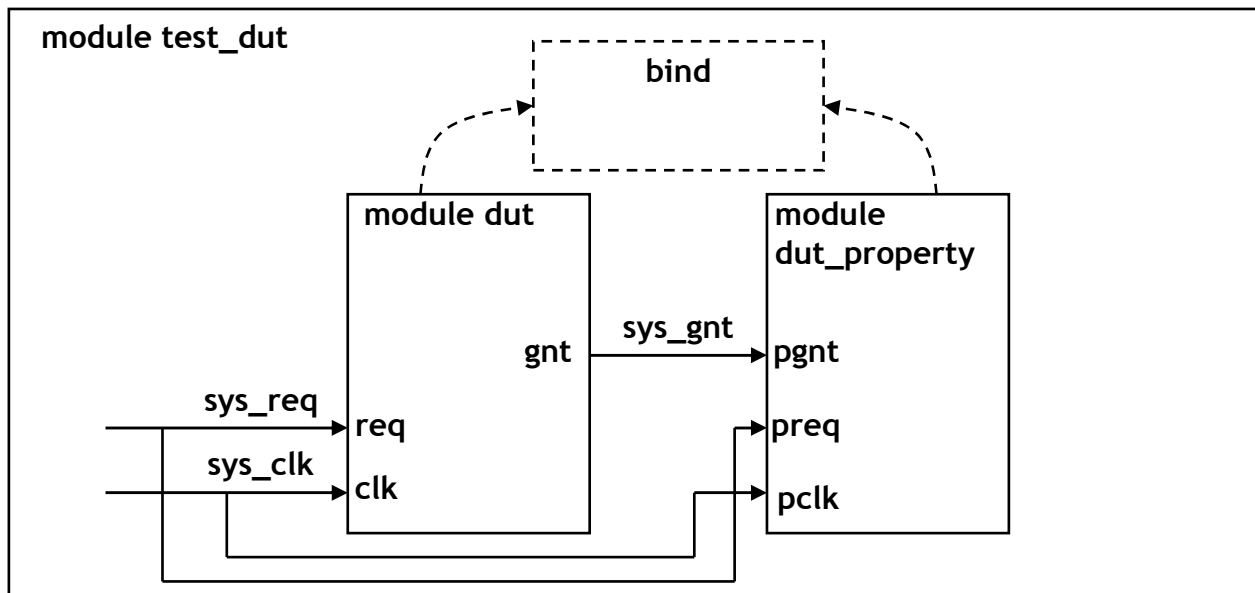
*It also highlights how property/sequence behaves with and without an implication and understand the vacuous pass phenomenon ;-)*

## LAB Objectives

1. 'bind' a design module to it's property module.
2. You will also see how the property can be 'cover'ed and how it can avoid a vacuous pass.

## LAB Design

*Here's the block diagram showing connection between 'dut' and 'dut\_property' modules. You need to bind these two modules.*



# LAB 1 : 'bind'ing design/property module

## LAB: Database

### FILES:

1. *dut.v :: Verilog module that drives a simple req/gnt protocol.*
2. *dut\_property.sv :: File that contains 'dut' properties/assertions.*
3. *test\_dut.sv :: Testbench for the 'dut'. This is the file in which you'll 'bind' the 'dut' with 'dut\_property'*
4. *LAB\_QUESTIONS :: This file has questions that you will answer.*

## LAB: How to compile/simulate - step by step instructions

1. `% cd <myDir>/SVA_LAB/LAB1`

2. `% vi test_dut.sv`  
*Search for "LAB EXERCISE - START"*

*Add code after the comments to  
'bind' the design 'dut' with it's property module 'dut\_property'*

*Save the file.*

*Then follow the compile/simulate steps described below.*

3. `% run_no_implication`

*This will create test\_dut\_no\_implication.log  
Compare ('diff') your log with the one stored in  
.solution/test\_dut\_no\_implication.log*

*If you did a correct bind of 'dut' and 'dut\_property', you will see a few PASS/FAIL  
in the log. If not, you won't see any pass or fail*

- ) *Study test\_dut\_no\_implication.log*
- ) *Answer the questions embedded in the file ./LAB\_QUESTIONS  
for +define+no\_implication*

# LAB 1 : 'bind'ing design/property module

---

## LAB: How to compile/simulate - step by step instructions

### 4. % run\_implication

*This will create test\_dut\_implication.log  
Compare ('diff') your log with the one stored in  
.solution/test\_dut\_implication.log*

*If you did a correct bind of 'dut' and 'dut\_property', you will see a few PASS/FAIL  
in the log. If not, you won't see any pass or fail*

- ) Study test\_dut\_implication.log
- ) Answer the questions embedded in the file ./LAB\_QUESTIONS  
for +define+implication

### 5. % run\_implication\_novac

*This will create test\_dut\_implication\_novac.log  
Compare ('diff') your log with the one stored in  
.solution/test\_dut\_implication\_novac.log*

*If you did a correct bind of 'dut' and 'dut\_property', you will see a few PASS/FAIL  
in the log. If not, you won't see any pass or fail*

- ) Study test\_dut\_implication\_novac.log
- ) Answer the questions embedded in the file ./LAB\_QUESTIONS  
for +define+implication\_novac

---

Lab 2 ...

overlapping and  
non-overlapping  
implication operator ...

# LAB 2 : Overlap and non-overlap operators

---

## LAB Overview

*This example is simply to high light how property behaves with an overlapping implication operator and with a non-overlapping implication operator. It also helps you understand how pipelined threads of a property work*

## LAB Objectives

1. *You will learn how an overlapping vs. non-overlapping implication operator works.*
2. *You will also learn how multiple pipelined threads work through a property.*

## LAB Design Under Test (DUT)

*There is no DUT as such in this example. Only a simple property coded with overlapping and non-overlapping operator.*



# LAB 2 : Overlap and non-overlap operators

---

## LAB: Database

### FILES:

1. *test\_overlap\_nonoverlap.sv* :: A simple example showing how to model a property with overlap and non-overlap implication operator.
2. *LAB\_QUESTIONS* :: This file has questions that you will answer as pointed out below in compile/simulate section.

## LAB: How to compile/simulate - step by step instructions

1. `% cd <myDir>/SVA_LAB/LAB2`
2. `% run_overlap`  
This will create *test\_overlap.log*
  - Study *test\_overlap.log*
  - Answer the questions embedded in the file *./LAB\_QUESTIONS* for +define+overlap
3. `% run_nonoverlap`  
This will create *test\_nonoverlap.log*
  - Study *test\_nonoverlap.log*
  - Answer the questions embedded in the file *./LAB\_QUESTIONS* for +define+nonoverlap

---

Lab 3 ...

good old fifo ...

# LAB 3 : FIFO

---

## LAB Overview

*A simple synchronous FIFO design is presented. FIFOs are some of the most commonly used design elements which require close scrutiny. FIFO assertions deployed directly at the source of a FIFO can greatly reduce the time to debug since these assertions point to the exact instance of fifo where an assertion fires.*

## LAB Objectives

1. *You will learn how to model various FIFO assertions that will be applicable to most any FIFO.*
2. *You will learn use of boolean expressions and sampled value functions as part of this exercise.*

## LAB Design Under Test (DUT)

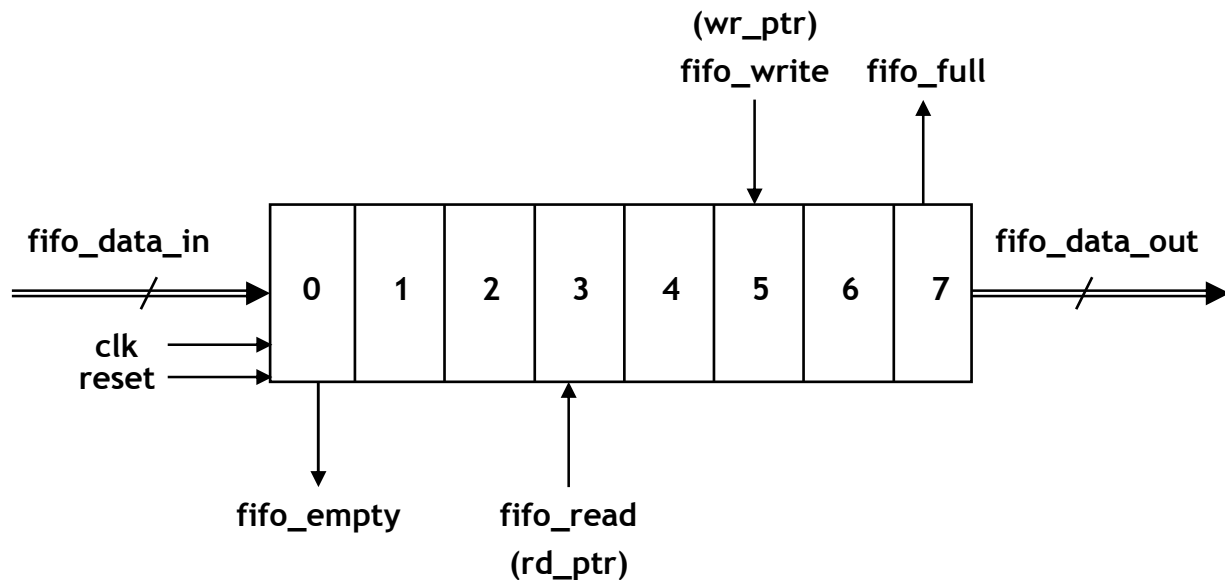
*A simple synchronous FIFO design is presented as the DUT.*

- *FIFO is 8 bit wide and 8 deep.*
- *FIFO INPUTS*  
*fifo\_write, fifo\_read, clk, rst\_ and fifo\_data\_in[7:0]*
- *FIFO OUTPUTS*  
*fifo\_full, fifo\_empty, fifo\_data\_out[7:0]*

# LAB 3 : FIFO

## FIFO Specs

- *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  $\geq 7$ , `fifo_full` is asserted*
  - *When fifo 'cnt' is 0, `fifo_empty` is asserted.*



# LAB 3 : FIFO

## LAB: Database

### FILES:

1. *fifo.v :: Verilog RTL for 'fifo'*
2. *fifo\_property.sv :: SVA file for fifo assertion.s*  
*This is the file in which you will add your assertions.*
3. *test\_fifo.sv :: Testbench for the fifo.*  
*Note the use of 'bind' in this testbench.*

## LAB: Assertions to Code

*Code assertions to check for the following conditions in the 'fifo' design.*

- CHECK # 1. Check that on reset  
rd\_ptr=0; wr\_ptr=0; cnt=0; fifo\_empty=1 and fifo\_full=0*
- CHECK # 2. Check that fifo\_empty is asserted when fifo 'cnt' is 0.  
Disable this property 'iff (!rst)'*
- CHECK # 3. Check that fifo\_full is asserted any time fifo 'cnt' is greater than 7.  
Disable this property 'iff (!rst)'*
- CHECK # 4. Check that if fifo is full and you attempt to write (but not read) that  
the wr\_ptr does not change.*
- CHECK # 5. Check that if fifo is empty and you attempt to read (but not write) that  
the rd\_ptr does not change.*
- CHECK # 6. Write a property to Warn on write to a full fifo*
- CHECK # 7. Write a property to Warn on read from an empty fifo*

# LAB 3 : FIFO

## *LAB: How to compile/simulate - step by step instructions*

Follow the steps below to add your assertion for each check.

Then compile/simulate with each of your assertions and see that your results match with those stored in the `./solution` directory

Here's step by step instructions...

1. `% cd <myDir>/SVA_LAB/LAB3`
2. First run the design without any bugs introduced in it.

`% run_nobugs`

- This will create the file `test_fifo_nobugs.log`
- Study this log to familiarize yourself with how the fifo works; when it reaches `fifo_full`, `fifo_empty` conditions, etc.

The remaining flow of the exercise is such that when you run any of the following steps, a specific bug is introduced in the design that your assertion should catch.

3. `% vi fifo_property.sv`
  - Look for ``ifdef check1`
  - Remove the 'DUMMY' property and code your property as specified in the comments
  - save the file and run the following simulation.

`% run_check1`

- If you have coded the property correct, you should see a failure for the CHECK #1 specified above.
- Simulation will create `test_fifo_check1.log`
- Compare `test_fifo_check1.log` with `./solution/test_fifo_check1.log` and see if your results match with the log in the `./solution` directory.
- If your results don't match revisit your property and repeat step 3.

**CONTINUED** ➔

# LAB 3 : FIFO

## *LAB: How to compile/simulate - step by step instructions - continued .*

### 4. % vi fifo\_property.sv

- Look for `ifdef check2
- Remove the 'DUMMY' property and code your property as specified in the comments
- save the file and run the following simulation.

% run\_check2

- If you have coded the property correct, you should see a failure for the CHECK #2 specified above.
- Simulation will create test\_fifo\_check2.log
- Compare test\_fifo\_check2.log with .solution/test\_fifo\_check2.log and see if your results match with the log in the .solution directory.
- If your results don't match revisit your property and repeat step 4.

### 5. % vi fifo\_property.sv

- Look for `ifdef check3
- Remove the 'DUMMY' property and code your property as specified in the comments
- save the file and run the following simulation.

% run\_check3

- If you have coded the property correct, you should see a failure for the CHECK #3 specified above.
- Simulation will create test\_fifo\_check3.log
- Compare test\_fifo\_check3.log with .solution/test\_fifo\_check3.log and see if your results match with the log in the .solution directory.
- If your results don't match revisit your property and repeat step 5

**CONTINUED** ➔

## LAB 3 : FIFO

### *LAB: How to compile/simulate - step by step instructions - continued .*

6. % vi fifo\_property.sv

- Look for `ifdef check4
- Remove the 'DUMMY' property and code your property as specified in the comments
- save the file and run the following simulation.

% run\_check4

- If you have coded the property correct, you should see a failure for the CHECK #4 specified above.
- Simulation will create test\_fifo\_check4.log
- Compare test\_fifo\_check4.log with .solution/test\_fifo\_check4.log and see if your results match with the log in the .solution directory.
- If your results don't match revisit your property and repeat step 6.

7. % vi fifo\_property.sv

- Look for `ifdef check5
- Remove the 'DUMMY' property and code your property as specified in the comments
- save the file and run the following simulation.

% run\_check5

- If you have coded the property correct, you should see a failure for the CHECK #5 specified above.
- Simulation will create test\_fifo\_check5.log
- Compare test\_fifo\_check5.log with .solution/test\_fifo\_check5.log and see if your results match with the log in the .solution directory.
- If your results don't match revisit your property and repeat step 7

**CONTINUED** ➔



## LAB 3 : FIFO

---

### *LAB: How to compile/simulate - step by step instructions - continued .*

8. % vi fifo\_property.sv

- Look for `ifdef check6
- Remove the 'DUMMY' property and code your property as specified in the comments
- save the file and run the following simulation.

% run\_check6

- If you have coded the property correct, you should see a failure for the CHECK #6 specified above.
- Simulation will create test\_fifo\_check6.log
- Compare test\_fifo\_check6.log with .solution/test\_fifo\_check6.log and see if your results match with the log in the .solution directory.
- If your results don't match revisit your property and repeat step 8.

9. % vi fifo\_property.sv

- Look for `ifdef check7
- Remove the 'DUMMY' property and code your property as specified in the comments
- write the file and run the following simulation.

% run\_check7

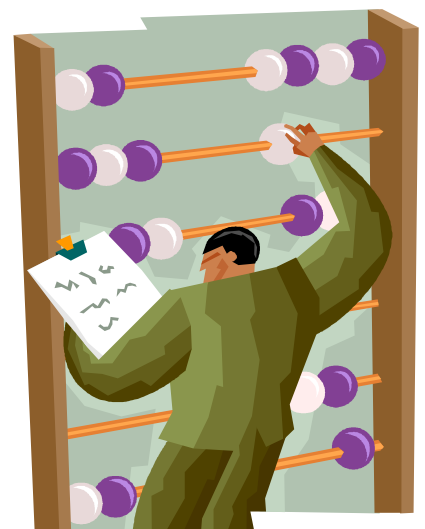
- If you have coded the property correct, you should see a failure for the CHECK #7 specified above.
- Simulation will create test\_fifo\_check7.log
- Compare test\_fifo\_check7.log with .solution/test\_fifo\_check7.log and see if your results match with the log in the .solution directory.
- If your results don't match revisit your property and repeat step 9

**DONE ... CONGRATULATIONS**

---

Lab 4 ...

a way to count ...



# LAB 4 : COUNTER

---

## LAB Overview

*A simple UP/DOWN COUNTER design is presented. Counter assertions deployed directly at the source can greatly reduce the time to debug since these assertions will point to the exact cause of a Counter error without the need for extensive back-tracing debug when design fails.*

## LAB Objectives

1. *You will learn use of sampled value functions.*
2. *Alternate ways of modeling an assertion.*

## LAB Design Under Test (DUT)

*A simple UP/DOWN COUNTER design is presented as the DUT.*

- \*) 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;*

# LAB 4 : COUNTER

---

## LAB: Database

### FILES:

1. *counter.v :: Verilog RTL for a simple counter.*
2. *counter\_property.sv :: SVA file for counter properties  
This is the file in which you will add your assertions.*
3. *test\_counter.sv :: Testbench for the counter.  
Note the use of 'bind' in this testbench.*

## LAB: Assertions to Code

*Code assertions to check for the following conditions in the 'counter' design.*

*CHECK # 1. Check that when 'rst\_' is asserted (==0) that data\_out == 8'b0*

*CHECK # 2. Check that if ld\_cnt\_ is deasserted (==1) and count\_enb is not enabled (==0) that data\_out HOLDS it's previous value.*

*Disable this property if rst is low.*

*CHECK # 3. Check that 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.*

*Disable this property if rst is low.*

# LAB 4 : COUNTER

## *LAB: How to compile/simulate - step by step instructions*

Follow the steps below to add your assertion for each check.

Then compile/simulate with each of your assertions and see that your results match with those stored in the ./solution directory

Here's step by step instructions...

1. % cd <myDir>/SVA\_LAB/LAB4
2. First run the design without any bugs introduced in it.

% run\_nobugs

- This will create the file test\_counter\_nobugs.log
- Study this log to familiarize yourself with how the counter works.

The remaining flow of the exercise is such that when you run any of the following steps, a specific bug is introduced in the design that your assertion should catch.

3. % vi counter\_property.sv

- Look for `ifdef check1
- Remove the 'DUMMY' property and code your property as specified above for CHECK #1
- Save the file and run the following simulation.

% run\_check1

- If you have coded the property correct, you should see a failure for the CHECK #1 specified above.
- Simulation will create test\_counter\_check1.log
- Compare test\_counter\_check1.log with ./solution/test\_counter\_check1.log and see if your results match with the log in the .solution directory.
- If your results don't match revisit your property and repeat step 3.

**CONTINUED** ➔

# LAB 4 : COUNTER

---

## *LAB: How to compile/simulate - step by step instructions*

4. % vi counter\_property.sv
- Look for `ifdef check2
  - Remove the 'DUMMY' property and code your property as specified above for CHECK #2
  - Save the file and run the following simulation.

% run\_check2

- If you have coded the property correct, you should see a failure for the CHECK #2 specified above.
  - Simulation will create test\_counter\_check2.log
  - Compare test\_counter\_check2.log with .solution/test\_counter\_check2.log and see if your results match with the log in the .solution directory.
  - If your results don't match revisit your property and repeat step 4.

5. % vi counter\_property.sv
- Look for `ifdef check3
  - Remove the 'DUMMY' property and code your property as specified above for CHECK #3
  - Save the file and run the following simulation.

% run\_check3

- If you have coded the property correct, you should see a failure for the CHECK #3 specified above.
  - Simulation will create test\_counter\_check3.log
  - Compare test\_counter\_check3.log with .solution/test\_counter\_check3.log and see if your results match with the log in the .solution directory.
  - If your results don't match revisit your property and repeat step 4.

*DONE... CONGRATULATIONS*

---

Lab 5 ...

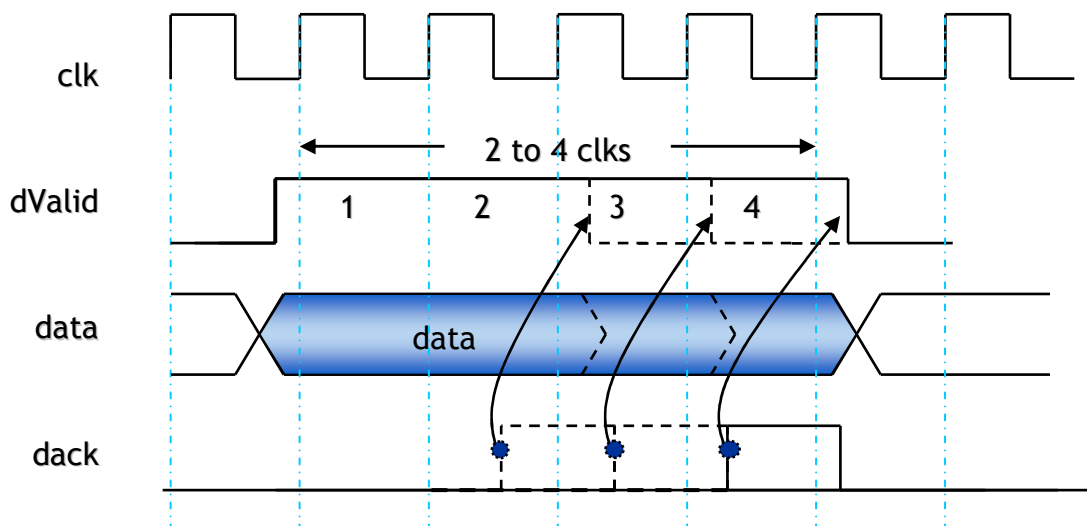
follow the protocol ...

# LAB 5 : BUS PROTOCOL

## LAB Overview

### Specification for a simple data transfer protocol.

- 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 de-assert 'dValid' the clock after 'dack' goes high.
- Note that since data must be valid for minimum 2 cycles, that 'dack' cannot go High for at least 1 clock after the transfer starts (i.e. after the rising edge of 'dValid') and that it must not remain low for more than 3 clocks (because data must transfer in max 4 clocks).*





# LAB 5 : BUS PROTOCOL

---

## LAB Objectives

*Bus interfaces are common to any design and this lab will show you how to model assertions for common bus protocol specification.*

*You will learn*

- 1. Modeling temporal domain assertions for bus interface type logic.*
- 2. Reinforce understanding of Edge sensitive and sampled value functions, consecutive repetition, boolean expressions, etc.*

## LAB: Database

**FILES:**

- 1. bus\_protocol.v :: bus\_protocol module that drive a simple bus protocol*
- 2. bus\_protocol\_property.sv :: SVA file for bus\_protocol assertions.  
Note that this file is only an empty module shell.  
You will add properties that meet the specification described above.*
- 3. test\_bus\_protocol.sv :: Testbench for the bus\_protocol module.  
Note the use of 'bind' in this testbench.*

# LAB 5 : BUS PROTOCOL

---

## LAB: Assertions to Code

*Code assertions to check for the following conditions in the 'bus protocol' design.*

*CHECK # 1. Check that once dValid goes high that it is consecutively asserted (high) for minimum 2 and maximum 4 clocks*

*CHECK # 2. Check that data is not unknown and remains stable after dValid goes high and until dAck goes high.*

*CHECK # 3. Check that 'dAck' and 'dValid' relationship is maintained to complete the data transfer.*

*In other words,*

*'dack' going high signifies that target have accepted data and that master must de-assert 'dValid' the clock after 'dack' goes high.*

*Note that since data must be valid for minimum 2 cycles, that 'dack' cannot go High for at least 1 clock after the transfer starts (i.e. after the rising edge of 'dValid') and that it must not remain low for more than 3 clocks (because data must transfer in max 4 clocks).*

# LAB 5 : BUS PROTOCOL

## *LAB: How to compile/simulate - step by step instructions*

Follow the steps below to add your assertion for each check.

Then compile/simulate with each of your assertions and see that your results match with those stored in the `./solution` directory

Here's step by step instructions...

1. `% cd <myDir>/SVA_LAB/LAB5`

2. First run the design without any bugs introduced in it.

`% run_nobugs`

- This will create the file `test_bus_protocol_nobugs.log`
  - Study this log to familiarize yourself with how the bus\_protocol works.

The remaining flow of the exercise is such that when you run any of the following steps, a specific bug is introduced in the design that your assertion should catch.

3. `% vi bus_protocol_property.sv`

- Look for ``ifdef check1`
  - Remove the 'DUMMY' property and code your property as specified above for CHECK #1
  - Save the file and run the following simulation.

`% run_check1`

- If you have coded the property correct, you should see a failure for the CHECK #1 specified above.
  - Simulation will create `test_bus_protocol_check1.log`
  - Compare `test_bus_protocol_check1.log` with `./solution/test_bus_protocol_check1.log` and see if your results match with the log in the `./solution` directory.
  - If your results don't match, revisit your property and repeat step 3.

**CONTINUED ➔**

# LAB 5 : BUS PROTOCOL

## *LAB: How to compile/simulate - step by step instructions - continued..*

### 4. % vi bus\_protocol\_property.sv

- Look for `ifdef check2
- Remove the 'DUMMY' property and code your property as specified for CHECK #2
- Save the file and run the following simulation.

% run\_check2

- If you have coded the property correct, you should see a failure for the CHECK#2.
  - Simulation will create test\_bus\_protocol\_check2.log
  - Compare test\_bus\_protocol\_check2.log with .solution/test\_bus\_protocol\_check2.log and see if your results match with the log in the .solution directory.
  - If your results don't match, revisit your property and repeat step 4.

### 5. % vi bus\_protocol\_property.sv

- Look for `ifdef check3
  - Remove the 'DUMMY' property and code your property as specified for CHECK #3
  - Save the file and run the following simulation.

% run\_check3

- If you have coded the property correct, you should see a failure for the CHECK #3.
  - Simulation will create test\_bus\_protocol\_check3.log
- Compare test\_bus\_protocol\_check3.log with .solution/test\_bus\_protocol\_check3.log and see if your results match with the log in the .solution directory.
- If your results don't match, revisit your property and repeat step 5.

**CONTINUED** ➔

# LAB 5 : BUS PROTOCOL

---

## *LAB: How to compile/simulate - step by step instructions - continued..*

6. This step is to simply run the design with ALL the bugs introduced and all the assertions fired.

`% run_checkall`

- If you have coded all your properties correct, you should see all of the failures specifically to learn how more than one design bug could be present at a given time.
  - Simulation will create test\_bus\_protocol\_checkall.log
  - Compare test\_bus\_protocol\_checkall.log with .solution/test\_bus\_protocol\_checkall.log and see if your results match with the log in the .solution directory.
  - If your results don't match, one of your steps 3 or 4 or 5 did not complete correct.

**DONE... CONGRATULATIONS**

---

Lab 6 ...

PCI Read protocol ...

# LAB 6: PCI Read Protocol

---

## LAB Overview

A simple system with a PCI Master and PCI Target modules designed to do a simple basic PCI Read operation.

The LAB shows how to derive and write simple but effective assertions for a PCI type bus.

## LAB Objectives

- 1) Learn how to model temporal domain assertions for bus interface type logic.
- 2) Reinforce understanding of Edge sensitive sampled value functions, consecutive repetition, boolean expressions, etc.

## LAB: Database

### FILES:

*pci\_master.v :: A (very) simple PCI Master module driving only a simple Read cycle.*

*pci\_target.v :: A (very) simple PCI Target module responding to a simple Read Cycle.*

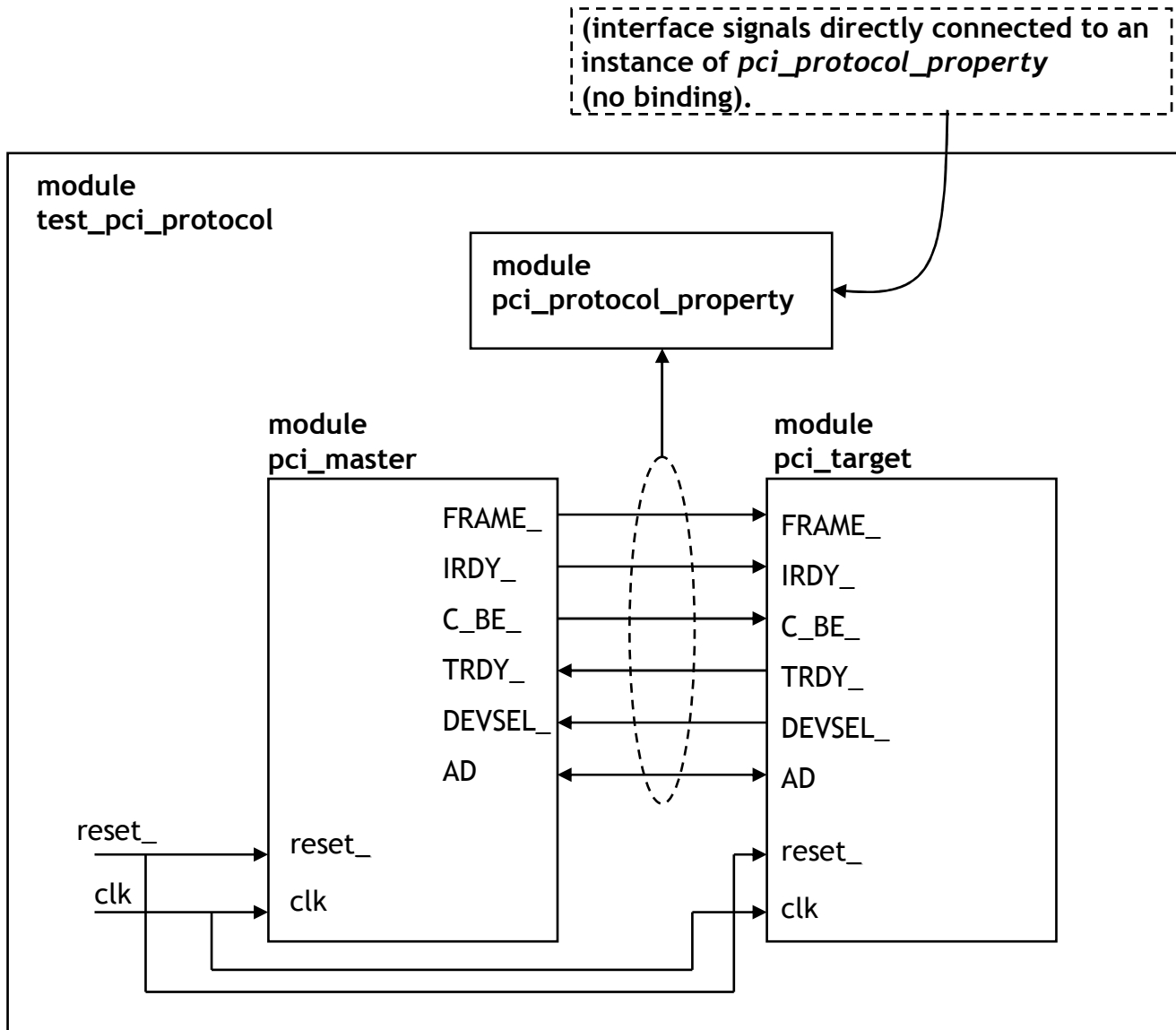
*pci\_protocol\_property.v :: SVA file for PCI Read cycle assertions.*

*Note that this file is only an empty module shell.*

*You will add properties that meet the specification described below.*

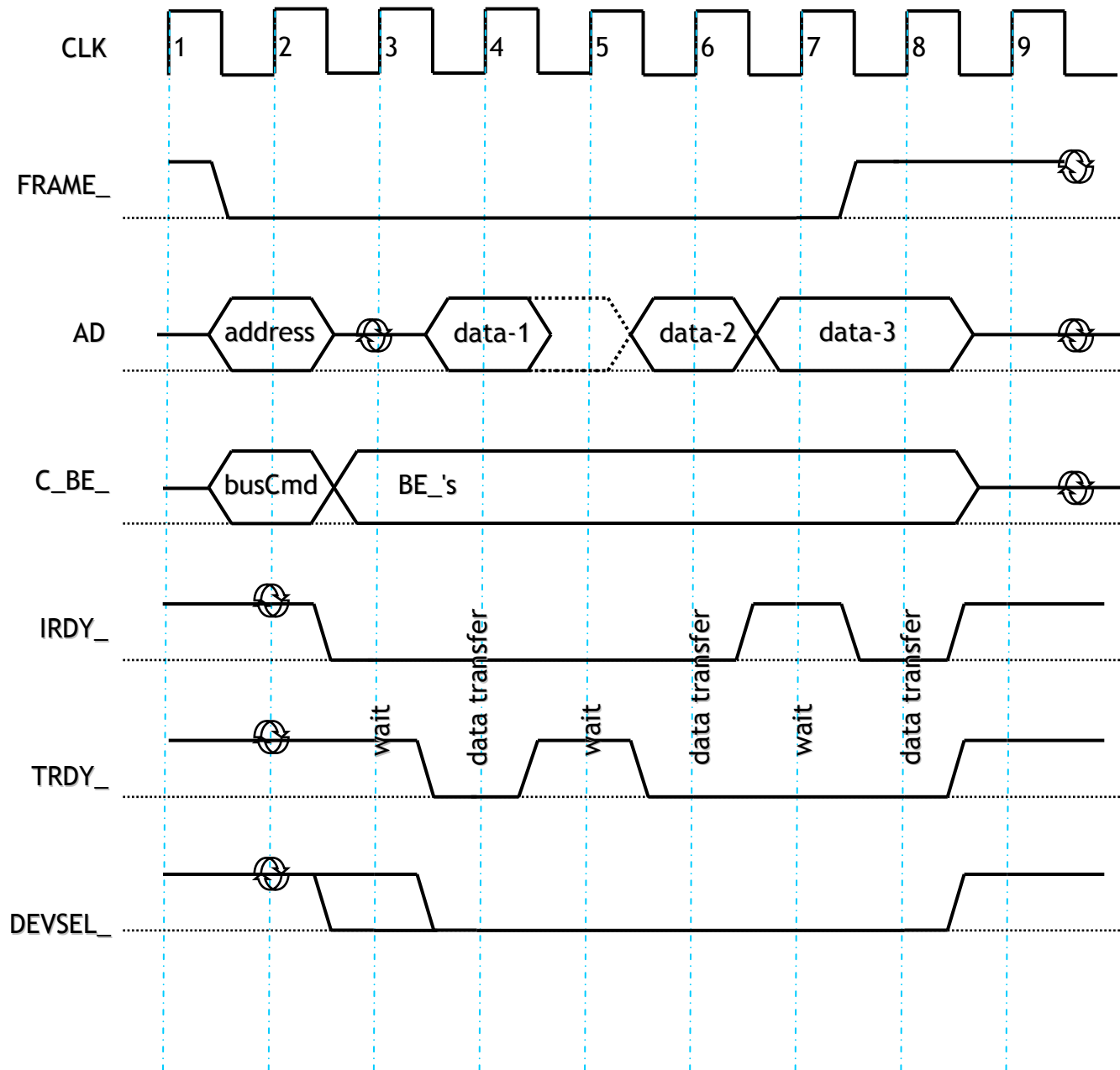
*test\_pci\_protocol.sv :: Testbench for the pci\_protocol module.*

# LAB 6: PCI System





# LAB6: PCI Read Protocol



# PCI: Basic Read Protocol Checkers

---

## *LAB: Assertions to Code*

Property Name	Description
checkPCI_AD_CBE (check1)	On falling edge of FRAME_, AD or C_BE_ bus cannot be unknown
checkPCI_DataPhase (check2)	When both IRDY_ and TRDY_ are asserted, AD or C_BE_ bus cannot be unknown
checkPCI_Frame_Irdy (check3)	FRAME can be de-asserted only if IRDY_ is asserted
checkPCI_trdyDevsel (check4)	TRDY_ can be asserted only if DEVSEL_ is asserted
checkPCI_CBE_during_trx (check5)	Once the cycle starts (i.e. at FRAME_ assertion) C_BE_ cannot float until FRAME_ is de-asserted.

# LAB 6 : PCI Read Protocol

## *LAB: How to compile/simulate - step by step instructions*

Follow the steps below to add your assertion for each check.

Then compile/simulate with each of your assertions and see that your results match with those stored in the ./solution directory

Here's step by step instructions...

1. % cd <myDir>/SVA\_LAB/LAB6  
% vi pci\_protocol\_property.sv

Edit this file to add your properties.

Note that DUMMY properties are coded in pci\_protocol\_property.sv to simply allow the module to compile.

You must remove the DUMMY properties and code correct properties as required above.

2. % vi pci\_protocol\_property.sv
  - Look for `ifdef check1
  - Remove the 'DUMMY' property and code your property as specified above for CHECK #1
  - Save the file and run the following simulation.

% run\_check1

- If you have coded the property correct, you should see a failure for the CHECK #1 specified above.
- Simulation will create test\_pci\_protocol\_check1.log
- Compare test\_pci\_protocol\_check1.log with ./solution/test\_pci\_protocol\_check1.log and see if your results match with the log in the .solution directory.
- If your results don't match, revisit your property and repeat this step.

**CONTINUED** ➔

# LAB 6 : PCI Read Protocol

## *LAB: How to compile/simulate - step by step instructions*

3.   % vi pci\_protocol\_property.sv
  - Look for `ifdef check2
  - Remove the 'DUMMY' property and code your property as specified above for CHECK #2
  - Save the file and run the following simulation.

% run\_check2

- If you have coded the property correct, you should see a failure for the CHECK #2 specified above.
- Simulation will create test\_pci\_protocol\_check2.log
- Compare test\_pci\_protocol\_check2.log with .solution/test\_pci\_protocol\_check2.log and see if your results match with the log in the .solution directory.
- If your results don't match, revisit your property and repeat this step.

4.   % vi pci\_protocol\_property.sv
  - Look for `ifdef check3
  - Remove the 'DUMMY' property and code your property as specified above for CHECK #3
  - Save the file and run the following simulation.

% run\_check3

- If you have coded the property correct, you should see a failure for the CHECK #3 specified above.
- Simulation will create test\_pci\_protocol\_check3.log
- Compare test\_pci\_protocol\_check3.log with .solution/test\_pci\_protocol\_check3.log and see if your results match with the log in the .solution directory.
- If your results don't match, revisit your property and repeat this step.

**CONTINUED ➔**

# LAB 6 : PCI Read Protocol

## *LAB: How to compile/simulate - step by step instructions*

5. % vi pci\_protocol\_property.sv
- Look for `ifdef check4
  - Remove the 'DUMMY' property and code your property as specified above for CHECK #4
  - Save the file and run the following simulation.

% run\_check4

- If you have coded the property correct, you should see a failure for the CHECK #4 specified above.
- Simulation will create test\_pci\_protocol\_check4.log
- Compare test\_pci\_protocol\_check4.log with .solution/test\_pci\_protocol\_check4.log and see if your results match with the log in the .solution directory.
- If your results don't match, revisit your property and repeat this step.

6. % vi pci\_protocol\_property.sv
- Look for `ifdef check5
  - Remove the 'DUMMY' property and code your property as specified above for CHECK #5
  - Save the file and run the following simulation.

% run\_check5

- If you have coded the property correct, you should see a failure for the CHECK #5 specified above.
- Simulation will create test\_pci\_protocol\_check5.log
- Compare test\_pci\_protocol\_check5.log with .solution/test\_pci\_protocol\_check5.log and see if your results match with the log in the .solution directory.
- If your results don't match, revisit your property and repeat this step.

**DONE... CONGRATULATIONS**

---

Lab

solutions ...

---

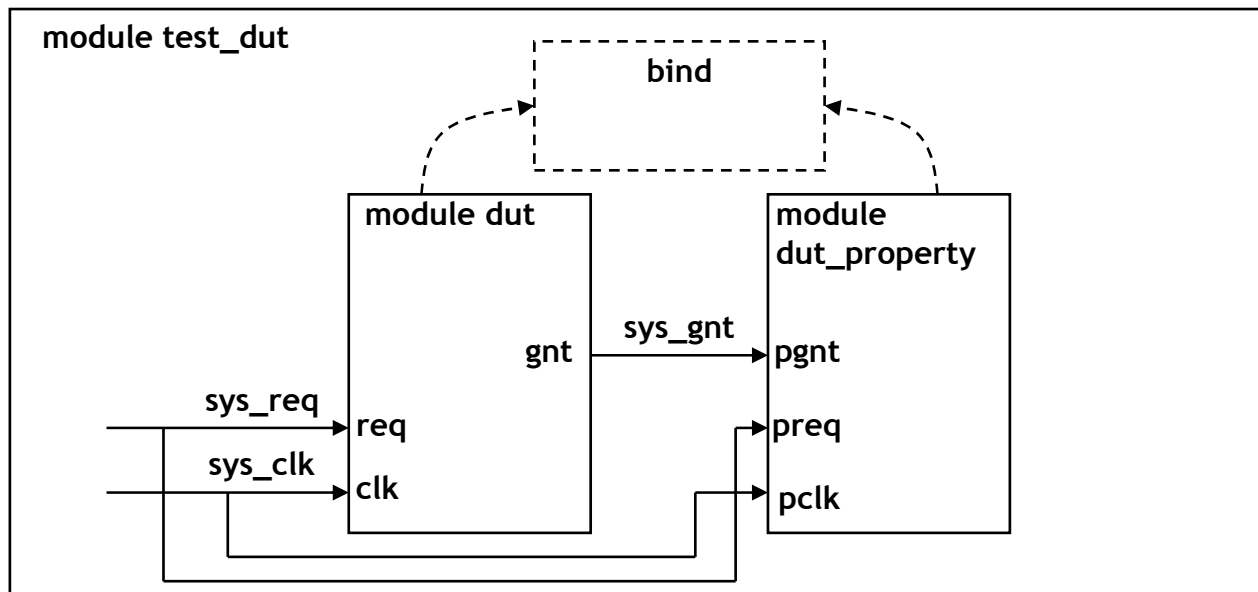
Lab 1 ...

how to 'bind' a design module  
with it's properties ...

# LAB 1 :: How to 'bind'

*LAB 1 : Code snippet from test\_dut.sv showing 'bind' between 'dut' and 'dut\_property'*

```
bind dut dut_property dut_bind_inst (  
    .pclk(clk),  
    .preq(req),  
    .pgnt(gnt)  
);
```





## LAB 1 : Q & A

---

### *LAB 1 : Code snippet for "no\_implication"*

```
property pr1;
  @(posedge clk) req ##2 gnt;
endproperty
reqGnt: assert property (pr1) $display($stime,, "\t\t %m PASS");
      else $display($stime,, "\t\t %m FAIL");
```

### *LAB 1 : Q&A on "no\_implication"*

```
/* +define+no_implication
```

```
run -all
KERNEL:    10  clk=1 req=0 gnt=0
KERNEL:    10           test_implication FAIL
KERNEL:    30  clk=1 req=1 gnt=0
KERNEL:    50  clk=1 req=0 gnt=0
KERNEL:    50           test_implication FAIL
KERNEL:    70  clk=1 req=0 gnt=1
KERNEL:    70           test_implication FAIL
KERNEL:    70           test_implication PASS
```

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

A: The FAIL at 70 is for the thread starting at time 70.

At 70, req==0 and since there is no implication, the property fails because without an implication there is no antecedent to match before the check begins. Whenever at posedge clk, 'req' is detected low, the property will fail.

The PASS at 70 is for the thread that starts at 30.

At 30, req==1, so property eval proceeds.

At 70 (i.e. 2 clocks later) gnt==1 as required by the property and the property PASSES.

## LAB 1 : Q & A

---

### LAB 1 : Code snippet for "no\_implication"

```
property pr1;  
  @(posedge clk) req ##2 gnt;  
endproperty  
reqGnt: assert property (pr1) $display($stime,, "\t\t %m PASS");  
      else $display($stime,, "\t\t %m FAIL");
```

### LAB 1 : Q&A on "no\_implication"

```
KERNEL:    90  clk=1 req=1 gnt=0  
KERNEL:   110  clk=1 req=0 gnt=0  
KERNEL:   110          test_implication FAIL  
KERNEL:   130  clk=1 req=0 gnt=0  
KERNEL:   130          test_implication FAIL  
KERNEL:   130          test_implication FAIL
```

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

A: The first failures is for the thread starting at time 90

At 90, req==1, so property eval proceeds.

At 130 (i.e. 2 clocks later) gnt==0 which violates the property and the property FAILs.

The second failure is for the thread starting at time 130.

At 130, req==0 and since there is no implication, the property fails because without an implication there is no antecedent to match before the check begins. Whenever at posedge clk, 'req' detected low, the property will fail.

# LAB 1 : Q & A

---

## LAB 1 : Code snippet for "implication"

```
property pr1;
  @(posedge clk) req |-> ##2 gnt;
endproperty

reqGnt: assert property (pr1) $display($stime,, "\t\t %m PASS");
      else $display($stime,, "\t\t %m FAIL");
```

## LAB 1 : Q&A on "implication"

```
run -all
KERNEL:    10  clk=1 req=0 gnt=0
KERNEL:    10           test_implication PASS
KERNEL:    30  clk=1 req=1 gnt=0
KERNEL:    50  clk=1 req=0 gnt=0
KERNEL:    50           test_implication PASS
KERNEL:    70  clk=1 req=0 gnt=1
KERNEL:    70           test_implication PASS
KERNEL:    70           test_implication PASS
```

Q: WHY ARE THERE 2 PASSes AT TIME 70 ??

A: The first pass is for the thread starting at time 30.

At 30, req==1, so property eval proceeds.

At 70 (i.e. 2 clocks later) gnt==1 as required by the property and the property PASSes.

The second pass is for the thread starting at time 70.

At 70, req==0 and since there is implication, the consequent eval won't start. However, there is a PASS action\_block associated with the property which triggers because of the vacuous pass phenomenon. In other words, whenever 'req' is low, the antecedent won't match and the property will pass vacuously.

## LAB 1 : Q & A

---

### LAB 1 : Code snippet for "implication"

```
property pr1;  
  @(posedge clk) req |-> ##2 gnt;  
endproperty  
  
reqGnt: assert property (pr1) $display($stime,, "\t\t %m PASS");  
        else $display($stime,, "\t\t %m FAIL");
```

### LAB 1 : Q&A on "implication"

```
KERNEL:    90  clk=1 req=1 gnt=0  
KERNEL:   110  clk=1 req=0 gnt=0  
KERNEL:   110          test_implication PASS  
KERNEL:   130  clk=1 req=0 gnt=0  
KERNEL:   130          test_implication FAIL  
KERNEL:   130          test_implication PASS
```

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

A: The failure is for the thread starting at time 90.

At 90, req==1, so property eval proceeds.

At 130 (i.e. 2 clocks later) gnt==0 which violates the property and the property FAILs.

The pass is for the property stating at 130.

At 130, req==0 and since there is implication, the consequent eval won't start. However, there is a PASS action\_block associated with the property which triggers because of the vacuous pass phenomenon. In other words, whenever 'req' is low, the antecedent won't match and the property will pass vacuously.

---

## Lab 2 ... Q&A

basic property with  
overlapping and non-  
overlapping implication  
operator ...

## LAB 2 : Q & A

### LAB 2 : Code snippet with "overlap" operator

```
sequence sr1;
  req ##2 gnt;
endsequence

property pr1;
  @(posedge clk) cstart |-> sr1;
endproperty

property pr1_for_cover;
  @(posedge clk) cstart ##0 sr1;
endproperty
```

### LAB 2 : Q&A on "overlap" operator

```
run -all
KERNEL:      10  clk=1 cstart=0 req=0 gnt=0
KERNEL:      30  clk=1 cstart=1 req=0 gnt=0
KERNEL:      30          test_overlap_nonoverlap FAIL
```

Q: WHY DOES THE PROPERTY FAIL at 30?

A: At time 30, cstart=1; so antecedent matches and consequent eval starts  
At time 30, req is NOT equal to 1 as required by overlapping implication  
and the consequent fails right away and the property FAILs.

```
KERNEL:      50  clk=1 cstart=1 req=1 gnt=0
KERNEL:      70  clk=1 cstart=0 req=0 gnt=0
KERNEL:      90  clk=1 cstart=0 req=0 gnt=1
KERNEL:      90          test_overlap_nonoverlap PASS
```

Q: WHY DOES THE PROPERTY PASS at 90?

A: At time 50, cstart=1; so antecedent matches and consequent eval starts  
At time 50 (i.e, the same clock as required by overlapping implication),  
req=1; so consequent eval continues  
At time 70, gnt=1 as required by the property and the consequent  
matches and the property PASSes.

## LAB 2 : Q & A

### LAB 2 : Q&A on "overlap" operator

```
KERNEL:    110 clk=1 cstart=1 req=1 gnt=0
KERNEL:    130 clk=1 cstart=1 req=1 gnt=0
KERNEL:    150 clk=1 cstart=1 req=1 gnt=1
KERNEL:    150          test_overlap_nonoverlap PASS
```

Q: WHY DOES THE PROPERTY PASS at 150?

A: At time 110, cstart=1; antecedent matches and consequent eval starts.  
At time 110 (i.e, the same clock as required by overlapping implication), req=1; so consequent eval continues  
At time 150 (i.e 2 clocks after 110), gnt=1 as required by the property so the consequent matches and the property PASSES

```
KERNEL:    170 clk=1 cstart=0 req=1 gnt=0
KERNEL:    170          test_overlap_nonoverlap FAIL
```

Q: WHY DOES THE PROPERTY FAIL at 170?

A: At time 130, cstart=1; antecedent matches and consequent eval starts  
At time 130 (i.e, the same clock as required by overlapping implication), req=1; so consequent eval continues  
At time 170 (i.e 2 clocks after 130), gnt is NOT equal to 0 as required by the property so the consequent does not match and the property FAILS

```
KERNEL:    190 clk=1 cstart=0 req=0 gnt=0
KERNEL:    190          test_overlap_nonoverlap FAIL
```

Q: WHY DOES THE PROPERTY FAIL at 190?

A: At time 150, cstart=1; antecedent matches and consequent eval starts  
At time 150 (i.e, the same clock as required by overlapping implication), req=1; so consequent eval continues  
At time 190 (i.e 2 clocks after 150), gnt is NOT equal 0 as required by the property so the consequent does not match and the property FAILS

## LAB 2 : Q & A

### LAB 2 : Code snippet with "non-overlap" operator

```
sequence sr1;
  req ##2 gnt;
endsequence

property pr1;
  @(posedge clk) cstart | => sr1;
endproperty

property pr1_for_cover;
  @(posedge clk) cstart ##1 sr1;
endproperty
```

### LAB 2 : Q&A on "non-overlap" operator

```
KERNEL:      10  clk=1 cstart=0 req=0 gnt=0
KERNEL:      30  clk=1 cstart=1 req=0 gnt=0
KERNEL:      50  clk=1 cstart=1 req=1 gnt=0
KERNEL:      70  clk=1 cstart=0 req=0 gnt=0
KERNEL:      70          test_overlap_nonoverlap FAIL
```

Q: WHY DOES THE PROPERTY FAIL at 70?

A: This failure is for the thread that started at time 50 (and not 30).

At time 50, cstart=1; so antecedent matches and consequent eval starts  
At time 70 (i.e, one clock later as required by nonoverlapping  
implication), req is NOT EQUAL to 1; so consequent does not match and  
the property FAILs

```
KERNEL:      90  clk=1 cstart=0 req=0 gnt=1
KERNEL:      90          test_overlap_nonoverlap PASS
```

Q: WHY DOES THE PROPERTY PASS at 90?

A: This pass is for the thread that started at time 30 (and not 50).

At time 30, cstart=1; so antecedent matches and consequent eval starts  
At time 50 (i.e, one clock later as required by nonoverlapping  
implication), req == 1; so consequent eval continues  
At time 90 (i.e, two clocks later as required by the property),  
gnt == 1; so consequent matches and the property PASSES.



## LAB 2 : Q & A

### LAB 2 : Q&A on "non-overlap" operator

```
KERNEL: 110 clk=1 cstart=1 req=1 gnt=0
KERNEL: 130 clk=1 cstart=1 req=1 gnt=0
KERNEL: 150 clk=1 cstart=1 req=1 gnt=1
KERNEL: 170 clk=1 cstart=0 req=1 gnt=0
KERNEL: 170          test_overlap_nonoverlap FAIL
```

Q: WHY DOES THE PROPERTY FAIL at 170?

A: This failure is for the thread that started at time 110

At time 110, cstart=1; antecedent matches and consequent eval starts

At time 130 (i.e, one clock later as required by nonoverlapping implication), req is EQUAL to 1; so consequent eval continues.

At time 170 (i.e, two clocks later as required by the property), gnt is NOT EQUAL to 1; so consequent does not match and the property FAILs.

```
KERNEL: 190 clk=1 cstart=0 req=0 gnt=0
KERNEL: 190          test_overlap_nonoverlap FAIL
```

Q: WHY DOES THE PROPERTY FAIL at 190?

A: This failure is for the thread that started at time 130

At time 110, cstart=1; so antecedent matches and consequent eval starts

At time 150 (i.e, one clock later as required by nonoverlapping implication), req is EQUAL to 1; so consequent eval continues.

At time 190 (i.e, two clocks later as required by the property), gnt is NOT EQUAL to 1; so consequent does NOT match and the property FAILs.

```
KERNEL: 210 clk=1 cstart=0 req=0 gnt=1
KERNEL: 210          test_overlap_nonoverlap PASS
```

Q: WHY DOES THE PROPERTY PASS at 210?

A: This pass is for the thread that started at time 150

At time 150, cstart=1; antecedent matches and consequent eval starts

At time 170 (i.e, one clock later as required by nonoverlapping implication), req is EQUAL to 1; so consequent eval continues.

At time 210 (i.e, two clocks later as required by the property), gnt is EQUAL to 1; so consequent matches and the property PASSES.

---

## Lab 3 : Solution ...

good old fifo ...

# Lab 3 : FIFO - Solution

## LAB 3 : fifo\_property.sv

```
// -----
// 1. Check that on reset,
//      rd_ptr=0; wr_ptr=0; cnt=0; fifo_empty=1 and fifo_full=0
// -----
`ifdef check1
property check_reset;
  @(posedge clk)
    (!rst_ |-> (`rd_ptr==0 && `wr_ptr==0 && fifo_empty==1 && fifo_full==0));
endproperty
check_resetP: assert property (check_reset) else $display($stime,"\t\t
FAIL::check_reset\n");
`endif

// -----
// 2. Check that fifo_empty is asserted the same clock that fifo 'cnt' is 0.
//      Disable this property 'iff (!rst_)'
// -----
`ifdef check2
property fifoempty;
  @(posedge clk) disable iff (!rst_)
    (`cnt==0 |-> fifo_empty);
endproperty
fifoemptyP: assert property (fifoempty) else $display($stime,"\t\t FAIL::fifo_empty
condition\n");
`endif

// -----
// 3. Check that fifo_full is asserted any time fifo 'cnt' is greater than 7.
//      Disable this property 'iff (!rst_)'
// -----
`ifdef check3
property fifofull;
  @(posedge clk) disable iff (!rst_)
    (`cnt>(fifo_depth-1) |-> fifo_full);
endproperty
fifofullP: assert property (fifofull) else $display($stime,"\t\t FAIL::fifo_full condition\n");
`endif
```

## Lab 3 : FIFO - Solution

### LAB 3 : *fifo\_property.sv*

```
// -----
//      4. Check that if fifo is full and you attempt to write (but not read) that
//      the wr_ptr does not change.
// -----
`ifdef check4
property fifo_full_write_stable_wrptr;
  @(posedge clk) disable iff (!rst_)
    (fifo_full && fifo_write && !fifo_read | => $stable(`wr_ptr));
endproperty
fifo_full_write_stable_wrptrP: assert property (fifo_full_write_stable_wrptr)
    else $display($stime,"\t\t FAIL::fifo_full_write_stable_wrptr condition\n");
`endif

`ifdef check5
// -----
//      5. Check that if fifo is empty and you attempt to read (but not write) that
//      the rd_ptr does not change.
// -----
property fifo_empty_read_stable_rdptr;
  @(posedge clk) disable iff (!rst_)
    (fifo_empty && fifo_read && !fifo_write | => $stable(`rd_ptr));
endproperty
fifo_empty_read_stable_rdptrP: assert property (fifo_empty_read_stable_rdptr)
    else $display($stime,"\t\t FAIL::fifo_empty_read_stable_rdptr
condition\n");
`endif

// -----
//      6. Write a property to Warn on write to a full fifo
//      This property will give Warning with all simulations
// -----
`ifdef check6
property write_on_full_fifo;
  @(posedge clk) disable iff (!rst_)
    fifo_full |-> !fifo_write;
endproperty
write_on_full_fifoP: assert property (write_on_full_fifo)
    else $display($stime,"\t\t WARNING::write_on_full_fifo\n");
`endif
```

# Lab 3 : FIFO - Solution

---

## LAB 3 : *fifo\_property.sv*

```
// -----  
// 7. Write a property to Warn on read from an empty fifo  
// This property will give Warning with all simulations  
// -----  
`ifdef check7  
property read_on_empty_fifo;  
  @(posedge clk) disable iff (!rst_)  
    fifo_empty |-> !fifo_read;  
endproperty  
read_on_empty_fifoP: assert property (read_on_empty_fifo)  
    else $display($stime,"\\t\\t WARNING::read_on_empty_fifo condition\\n");  
`endif
```

---

Lab 4 : Solution ...

counter example ...

## Lab 4 : COUNTER - Solution

### LAB 4 : counter\_property.sv

```
//-----  
//      CHECK # 1. Check that when 'rst_' is asserted (==0) that data_out == 8'b0  
//-----  
`ifdef check1  
property counter_reset;  
  @(clk) disable iff (rst_) !rst_ | => (data_out == 8'b0);  
endproperty  
  
counter_reset_check: assert property(counter_reset)  
  else $display($stime,,, "\t\tCOUNTER RESET CHECK FAIL:: rst_=%b data_out=%0d \n",  
                                                         rst_,data_out);  
`endif  
  
//-----  
//      CHECK # 2. Check that if ld_cnt_ is deasserted (==1) and count_enb is not enabled  
//      (==0) that data_out HOLDS it's previous value.  
//      Disable this property 'iff (!rst_)'  
//-----  
`ifdef check2  
property counter_hold;  
  @(posedge clk) disable iff (!rst_) (ld_cnt_ & !count_enb) | => data_out ==  
  $past(data_out);  
endproperty  
  
counter_hold_check: assert property(counter_hold)  
  else $display($stime,,, "\t\tCOUNTER HOLD CHECK FAIL:: counter HOLD \n");  
`endif
```

## Lab 4 : COUNTER - Solution

---

### LAB 4 : counter\_property.sv

```
`ifdef check3
property counter_count;
  @(posedge clk) disable iff (!rst_) (ld_cnt_ & count_enb) |->
    if (updn_cnt) ##1 (data_out-8'h01) == $past(data_out)
    else      ##1 (data_out+8'h01) == $past(data_out);
endproperty

counter_count_check: assert property(counter_count)
  else $display($stime,,, "\t\tCOUNTER COUNT CHECK FAIL:: UPDOWN COUNT using $past \n");
`endif

//-----
// Alternate way of writing assertion for CHECK # 3
// Check for count using local variable
//-----
/*
`ifdef check3
property counter_count_local;
logic[7:0] local_data;
  @(posedge clk) disable iff (!rst_) (ld_cnt_ & count_enb, local_data = data_out) |->
    if (updn_cnt) ##1 (data_out == (local_data+8'h01))
    else      ##1 (data_out == (local_data-8'h01));
endproperty

counter_count_check: assert property(counter_count)
  else $display($stime,,, "\t\tCOUNTER COUNT CHECK FAIL:: UPDOWN COUNT using $past \n");

`endif
*/
```



---

Lab 5 : Solution ...

bus protocol example ...

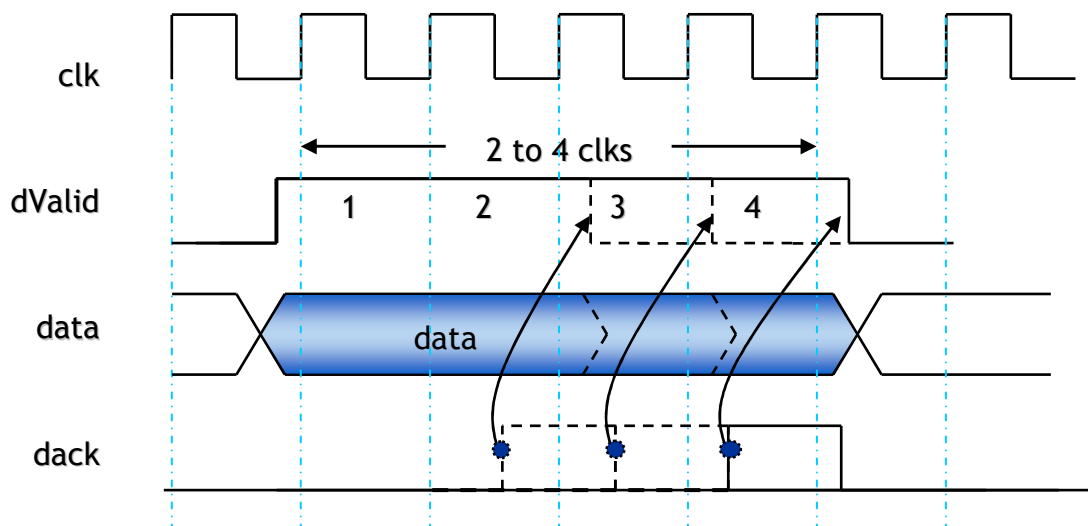
# LAB 5 : BUS PROTOCOL

## LAB Overview

### Specification for a simple data transfer protocol.

- 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 de-assert 'dValid' the clock after 'dack' goes high.

*Note that since data must be valid for minimum 2 cycles, that 'dack' cannot go High for at least 1 clock after the transfer starts (i.e. after the rising edge of 'dValid') and that it must not remain low for more than 3 clocks (because data must transfer in max 4 clocks).*



# Lab 5 : Bus Protocol - Solution

## LAB 5 : *bus\_protocol\_property.sv*

```
/*-----  
CHECK # 1. Check that once dValid goes high that it is consecutively  
asserted (high) for minimum 2 and maximum 4 clocks.  
Check also that once dValid is asserted (high) for 2 to 4 clocks that  
it does de-assert (low) the very next clock.  
-----*/  
`ifdef check1  
property checkValid;  
@(posedge clk) disable iff (reset) $rose(dValid) | => (dValid) [*2:4] ##1 $fell(dValid);  
endproperty  
assert property (checkValid) else $display($stime,, "checkValid FAIL");  
`endif  
  
/*-----  
CHECK # 2. Check that data is not unknown and remains stable after dValid goes  
high and until dAck goes high.  
-----*/  
`ifdef check2  
property checkdataValid;  
@(posedge clk) disable iff (reset)  
$rose(dValid) | => (!$isunknown(data) && $stable(data)) [*1:$] ##0 $rose(dAck);  
endproperty  
assert property (checkdataValid) else $display($stime,, "checkdataValid FAIL");  
`endif
```

# Lab 5 : Bus Protocol - Solution

## LAB 5 : bus\_protocol\_property.sv

```
/*-----  
    CHECK # 3. Check that 'dAck' and 'dValid' relationship is maintained to complete  
    the data transfer. In other words,  
  
    'dack' going high signifies that target have accepted data and that master must de-  
    assert 'dValid' the clock after 'dack' goes high.  
  
    Note that since data must be valid for minimum 2 cycles, that 'dack' cannot go High  
    for at least 1 clock after the transfer starts (i.e. after the rising edge of 'dValid') and  
    that it must not remain low for more than 3 clocks (because data must transfer in max 4  
    clocks).  
    -----*/  
`ifdef check3  
    property checkdAck;  
        @(posedge clk) disable iff (reset)  
            $rose(dValid) | => (dValid && !dAck) [*1:3] ##1 $rose (dAck) ##1 $fell  
(dValid);  
    endproperty  
    assert property (checkdAck) else $display($stime,,,"checkdAck FAIL");  
`endif
```

---

Lab 6 : Solution ...

PCI Read Protocol ...

# Lab 6 : PCI Read Protocol - Solution

## LAB 6 : *pci\_protocol\_property.sv*

```
/*-----
CHECK # 1. On falling edge of FRAME_, AD or C_BE_ cannot be unknown.
-----*/
`ifdef check1
    property checkPCI_AD_CBE;
        @(posedge clk) disable iff (!reset_) $fell(FRAME_) |->
            !($isunknown(AD) || $isunknown(C_BE_)) ;
    endproperty
    assert property (checkPCI_AD_CBE) else
$display($stime,,,"CHECK1:checkPCI_AD_CBE FAIL\n");
`endif

/*-----
CHECK # 2. When IRDY_ and TRDY_ are asserted (low) AD or C_BE_ cannot be
unknown.
-----*/
`ifdef check2
    property checkPCI_DataPhase;
        @(posedge clk) disable iff (!reset_) (!IRDY_ && !TRDY_) |->
            !($isunknown(AD) || $isunknown(C_BE_)) ;
    endproperty
    assert property (checkPCI_DataPhase) else
$display($stime,,,"CHECK2:checkPCI_DataPhase FAIL\n");
`endif

/*-----
CHECK # 3. FRAME_ can go High only if IRDY_ is asserted.
        In other words, master can signify end of cycle only if IRDY_ is asserted.
-----*/
`ifdef check3
    property checkPCI_Frame_Irddy;
        @(posedge clk) disable iff (!reset_) $rose(FRAME_) |-> !IRDY_;
    endproperty
    assert property (checkPCI_Frame_Irddy) else
$display($stime,,,"CHECK3:checkPCI_frmlrddy FAIL\n");
`endif
```

# Lab 6 : PCI Read Protocol - Solution

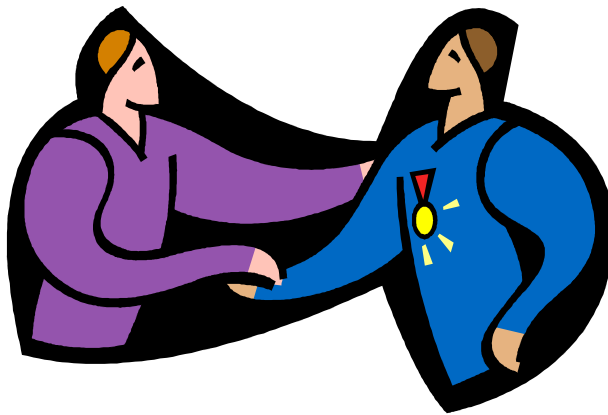
## LAB 6 : pci\_protocol\_property.sv

```
/*-----  
CHECK # 4. TRDY_ can be asserted (low) only if DEVSEL_ is asserted (low)  
-----*/  
`ifdef check4  
    property checkPCI_trdyDevsel;  
        @(posedge clk) disable iff (!reset_) !TRDY_ |-> !DEVSEL_;  
    endproperty  
    assert property (checkPCI_trdyDevsel) else  
$display($stime,,,"CHECK4:checkPCI_trdyDevsel FAIL\n");  
`endif  
  
/*-----  
CHECK # 5. Once the cycle starts (i.e. at FRAME_ assertion)  
        C_BE_ should not float until FRAME_ is de-asserted  
-----*/  
`ifdef check5  
    property checkPCI_CBE_during_trx;  
        @(posedge clk) disable iff (!reset_)  
            $fell(FRAME_) |-> !($isunknown(C_BE_)) [*0:$] ##0 $rose(FRAME_);  
    endproperty  
    assert property (checkPCI_CBE_during_trx) else  
$display($stime,,,"CHECK5:checkPCI_CBE_during_trx FAIL\n");  
`endif
```

---

‘that’s all folks...’

happy asserting...



*Questions/Comments? Please contact ...*

*DefineView Consulting*  
[www.defineview.com](http://www.defineview.com)

(email) [ashok@defineview.com](mailto:ashok@defineview.com)  
(phone) 408.309.1556