2023

Exercise 1

TFE4171 - DESIGN OF DIGITAL SYSTEMS 2

SIMON DARGAHI & REZA NOROOZI

Table of Contents

Lab 1

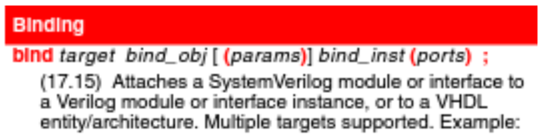
For the rest of the report. I will refer to the sub-parts of the Exercise 1 in Blackboard as “Task n”. These are not to be confused with “task m” which are in the “SVALAB_LINUX_LAB”. I will also discard, and not count any task that askes for the user to use the “cd” command to orient oneself into a folder.

Task 1

You can bind the property module with the DUT using this code:

```
bind dut1 dut_property bind_inst(
  .pclk(clk),
  .preq(req),
  .pgnt(gnt)
);
```

This works because of:


Binding
bind *target bind_obj* [(*params*)] *bind_inst* (*ports*) ;
(17.15) Attaches a SystemVerilog module or interface to a Verilog module or interface instance, or to a VHDL entity/architecture. Multiple targets supported. Example:

Task 2

The run_no_implication script sends a flag with the name “no_implication”. This means that this part of the script in dut_properties.sv is ran.

```
`ifdef no_implication
property pr1;
@(posedge pclk) preq ##2 pgnt;
endproperty
preqGnt: assert property (pr1) $display($stime,,,"\t\t %m PASS");
else $display($stime,,,"\t\t %m FAIL");
```

The output of this is the following:

```
#         10  clk=1 req=0 gnt=0
#         10         test_dut.dut1.bind_inst.preqGnt FAIL
#         30  clk=1 req=1 gnt=0
#         50  clk=1 req=0 gnt=1
#         50         test_dut.dut1.bind_inst.preqGnt FAIL
#         70  clk=1 req=0 gnt=1
#         70         test_dut.dut1.bind_inst.preqGnt FAIL
#         70         test_dut.dut1.bind_inst.preqGnt PASS
#         90  clk=1 req=1 gnt=0
#        110  clk=1 req=0 gnt=0
#        110         test_dut.dut1.bind_inst.preqGnt FAIL
#        130  clk=1 req=0 gnt=0
#        130         test_dut.dut1.bind_inst.preqGnt FAIL
#        130         test_dut.dut1.bind_inst.preqGnt FAIL
#        150  clk=1 req=0 gnt=0
```

Q: Would you get the same execution results when running no modules, you use signals sys clk, sys req, and sys implication if when binding the gnt instead of using clk, req, and gnt?

A: In theory, one could use the clk instead of sys_clk for smaller events. The only problem would be that for bigger events, there is a chance there would not be time to change states in delta delays.

Q: Why is there neither FAIL nor PASS at time 30?

A: Because of the first code. Line 3, there is this following code: preq ##2 pgnt; this means that once req is high, it will wait two clock cycles. At time 30, req is high, which means that at that point there is not known if “preq ##2 pgnt” is true or not, it will have to wait two clock cycles to know this. But at the same time, it knows that there is a possibility that it might be true, therefore it cannot show false.

Q: Why is there a FAIL at time 50?

A: because it knows that the statement “preq ##2 pgnt” is false. Because req is low at time 50. Because of this, it will show what is in the *else* statement in the assert. Witch is the FAIL message.

Q: Why is there a FAIL and a PASS at time 70?

A: The first FAIL comes from the assertion that spawned at time 70. At this time req is low. At the same time, it checks the assert spewed at time 30. Because it knew that at 30 req was high, it has now waited two clock cycles to check that gnt is high. And because of that the assert spawned at 30 is true, and it can output the PASS statement in the assert.

Q: Why are there 2 FAILS at time 130?

A: The first fail is the assertion that spewed at time 130. Because req is low, it knows that the assert that just spawned has to false. But at time 90, there was also an assert that spawned, this assert has waited two clock cycles and is now checking for gnt. And because these two clock cycles after 90, time 130 also is low, we get another FAIL.

Task 3

The run_implication script sends a flag with the name "implication". This means that this part of the script in dut_properties.sv is run.

```
`elsif implication
property pr1;
@(posedge pclk) preq |-> ##2 pgnt;
endproperty
preqGnt: assert property (pr1) $display($stime,,,"\t\t %m PASS");
else $display($stime,,,"\t\t %m FAIL");
```

The output of this is the following:

```
#          10  clk=1 req=0 gnt=0
#          10          test_dut.dut1.bind_inst.preqGnt PASS
#          30  clk=1 req=1 gnt=0
#          50  clk=1 req=0 gnt=1
#          50          test_dut.dut1.bind_inst.preqGnt PASS
#          70  clk=1 req=0 gnt=1
#          70          test_dut.dut1.bind_inst.preqGnt PASS
#          70          test_dut.dut1.bind_inst.preqGnt PASS
#          90  clk=1 req=1 gnt=0
#         110  clk=1 req=0 gnt=0
#         110          test_dut.dut1.bind_inst.preqGnt PASS
#         130  clk=1 req=0 gnt=0
#         130          test_dut.dut1.bind_inst.preqGnt PASS
#         130          test_dut.dut1.bind_inst.preqGnt FAIL
#         150  clk=1 req=0 gnt=0
```

Q: What is a vacuous pass?

A: vacuou irst variable before “|->” is false.

Q: Most of the passes are vacuous, so which passes are NOT vacuous? State the times in which the non-vacuous passes occur and explain why they occur in these specific clock cycles.

A: All of the passes, expect the second pass at time 70 are vacuous. At time 10, 50, first pass at 7, 110, and first pass at 130. They all check the following: “preq |-> ##2 pgnt” and they all se req as low. Because of that, we get vacuous passes.

Q: Why are there 2 PASSES at 70?

A: The first is a vacuous pass because req is low, the second one is a true pass because the assertian spawned at time 30. At time 30 it checked that req is high, and waited two clock cycles to check for gnt. And because gnt is high, we get a true pass.

Q: Why is there a PASS and a FAIL at time 130?

A: it first pass is a vacuous pass because of “130 clk=1 req=0 gnt=0” but beause “90 clk=1 req=1 gnt=0” it has waited two clock cycles to check for gnt. And because gnt is low, we get a fail.

Task 4

The run_implication_novac script sends a flag with the name “implication_novac”. This means that this part of the script in dut_properties.sv is ran.

```
1. `elsif implication_novac
2. property pr1;
3. @(posedge pclk) preq |-> ##2 pgnt;
4. endproperty
5. preqGnt: assert property (pr1) else $display($stime,,,"\t\t %m FAIL");
6. property pr2;
7. @(posedge pclk) preq ##2 pgnt;
8. endproperty
9. cpreqGnt: cover property (pr2) $display($stime,,,"\t\t %m PASS");
```

This outputs:

```
#         10  clk=1 req=0 gnt=0
#         30  clk=1 req=1 gnt=0
#         50  clk=1 req=0 gnt=1
#         70  clk=1 req=0 gnt=1
#         70         test_dut.dut1.dut_bind_inst.cpreqGnt PASS
#         90  clk=1 req=1 gnt=0
#        110  clk=1 req=0 gnt=0
#        130  clk=1 req=0 gnt=0
#        130         test_dut.dut1.dut_bind_inst.preqGnt FAIL
#        150  clk=1 req=0 gnt=0
```

Q: Why do you only get in total 1 PASS and 1 FAIL? Why don’t you get more PASSES and FAILS in the other clock cycles?

A: Because it does only show non- vacuous. If something is vacuous, it is overwritten by cover property pr2 between line 6 and 9

property pr1;

For every req=0 the cover property(pr2) will be spawned. But because it does not have a else output, it will output nothing.

At 30 (clk=1 req=1 gnt=0) and 90 (clk=1 req=1 gnt=0) it will spawn both pr1 and pr2. And at 70 (clk=1 req=0 gnt=1) and 130 (clk=1 req=0 gnt=0), the 70 will output the cover:

“property (pr2) $display($stime,,,"\t\t %m PASS");”

because it is True to the statement “preq ##2 pgnt;“ and the 130 will output the else:
$display($stime,,,"\t\t %m FAIL");

This is because “preq |-> ##2 pgnt” is flase and the else statement will be triggerd.

Lab 2

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

```
reqGnt: assert property (pr1) else $display($stime,,,"\t\t %m FAIL");
creqGnt: cover property (pr1_for_cover) $display($stime,,,"\t\t %m PASS");
```

Task 1

The run_overlap script runes test_overlap_nooverlap.sv with the flag overlap. The flag makes it so that this piece of code run ran.

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

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

```
#         10  clk=1 cstart=0 req=0 gnt=0
#         30  clk=1 cstart=1 req=0 gnt=0
#         30          test_overlap_nonoverlap.reqGnt FAIL
#         50  clk=1 cstart=1 req=1 gnt=0
#         70  clk=1 cstart=0 req=0 gnt=0
#         90  clk=1 cstart=0 req=0 gnt=1
#         90          test_overlap_nonoverlap.creqGnt PASS
#        110  clk=1 cstart=1 req=1 gnt=0
#        130  clk=1 cstart=1 req=1 gnt=0
#        150  clk=1 cstart=1 req=1 gnt=1
#        150          test_overlap_nonoverlap.creqGnt PASS
#        170  clk=1 cstart=0 req=1 gnt=0
#        170          test_overlap_nonoverlap.reqGnt FAIL
#        190  clk=1 cstart=0 req=0 gnt=0
#        190          test_overlap_nonoverlap.reqGnt FAIL
#        210  clk=1 cstart=0 req=0 gnt=1
```

Q: Why does the property FAIL at 30?

A: It will check if sr1 is True when cstart is high. But because at time 30, cstart is high, it will then imedely check sr1 witch checks req, and because req is low it is an instant fail. And because “reqGnt: assert property (pr1) else $display($stime,,,"\t\t %m FAIL");” it will run the $display and output the ($stime,,,"\t\t %m FAIL");

Q: If both cstart and req are 1 at 50, why we don’t get a PASS at 50?

A: because of sr1, it will have to wait two clock cycles to check for gnt. It does therefor not know if sr1 is true or false at time 50. But it knows that it is not false because req is high.

Q: Why does the property PASS at 90?

A: For it to pass, you need to look at the code below

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

because at time 50, cstart is high, and pr1_for_cover will check that sr1 is true, to do this it will check “req ##2 gnt” it also sees that req at 50 is high, two clock cyklus later, gnt is high. This makes it so pr1_for_cover is true. It will therefor run the code “creqGnt: cover property (pr1_for_cover) $display($stime,,,"\t\t %m PASS");” and the “display($stime,,,"\t\t %m PASS");” command.

Q: Why does the property PASS at 150?

A: same reson it passed at 90. It runs the check to see if it is high. And it does not look for falling edge.

Q: Why does the property FAIL at 170?

A: beause of “130 clk=1 cstart=1 req=1 gnt=0” it will start the check in sr1 (req ##2 gnt;). And because gnt is low, it will fail the pr1 assertian and do the else statment witch outputs the FAIL message in the “else” assert.

Q: Why does the property FAIL at 190?

A: same reason it failes at 170 as mention above, but this time the sr1 assertian is triggerd by “150 clk=1 cstart=1 req=1 gnt=1” because cstart=req=1.

Task 2

The run_nooverlap script runes test_overlap_nooverlap.sv with the flag nooverlap. The flag makes it so that this piece of code run ran.

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

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

With the output:

```
#         10  clk=1 cstart=0 req=0 gnt=0
#         30  clk=1 cstart=1 req=0 gnt=0
#         50  clk=1 cstart=1 req=1 gnt=0
#         70  clk=1 cstart=0 req=0 gnt=0
#         70         test_overlap_nonoverlap.reqGnt FAIL
#         90  clk=1 cstart=0 req=0 gnt=1
#         90         test_overlap_nonoverlap.creqGnt PASS
#        110  clk=1 cstart=1 req=1 gnt=0
#        130  clk=1 cstart=1 req=1 gnt=0
#        150  clk=1 cstart=1 req=1 gnt=1
#        170  clk=1 cstart=0 req=1 gnt=0
#        170         test_overlap_nonoverlap.reqGnt FAIL
#        190  clk=1 cstart=0 req=0 gnt=0
#        190         test_overlap_nonoverlap.reqGnt FAIL
#        210  clk=1 cstart=0 req=0 gnt=1
#        210         test_overlap_nonoverlap.creqGnt PASS
```

Q: If cstart is 1 and req is 0 at 30, why we don't get a FAIL at 30?

A: It wont fail because |=> will check the next clock cykle, therefor sr1 and req will not be checked untill time 50.

Q: Why does the property FAIL at 70?

A: the code does the folloing

Time 50 starts triggers pr1

Because |=> it will check sr1 at the next clock cykle whitch is time 70

Becuase req 70 is low its an instant fail and the else statement witch outputs the fail print will be triggerd.

Q: Why does the property PASS at 90?

A: the code does the folloing

Time 30 starts triggers pr1_for_cover start

Because ##1 it will check sr1 at the next clock cykle whitch is time 50

Time 50 sr1 will do a check, it starts with req, and this is high.

Because of ##2 it will wait two clock cykle witch is time 90

At 90 gnt is high, witch makes the cover property true, and therefor print statemnt is executed.

Q: Why does the property FAIL at 170?

A: Time 110 starts triggers pr1 start

Because |=> it will check sr1 at the next clock cykle whitch is time 130

Time 130 sr1 will do a check, it starts with req, and this is high.

Because of ##2 it will wait two clock cykle witch is time 170

At 170 gnt is low, witch makes the asert property false, and therefor print statemnt behind the else declartion.

Q: Why does the property FAIL at 190?

A: Time 130 cstarts triggers pr1 start

Because |=> it will check sr1 at the next clock cykle whitch is time 150

Time 150 sr1 will do a check, it starts with req, and this is high.

Because of ##2 it will wait two clock cykle witch is time 190

At 190 gnt is low, witch makes the asert property false, and therefor print statemnt behind the else declartion.

Q: Why does the property PASS at 210?

A: Time 150 cstarts triggers pr1_for_cover start

Because ##1 it will check sr1 at the next clock cykle whitch is time 150

Time 170 sr1 will do a check, it starts with req, and this is high.

Because of ##2 it will wait two clock cykle witch is time 210

At 210 gnt is high, witch makes the cover property true, and therefor print statemnt is executed.

Lab 3

For taskes that ask for something in repeat, like the following: “Follow from task 3 until 9!” I will split each of the tasks from the question into new taskes in the lab report.

Therefor task 1 here will begin with task 3 in the “SVALAB_LINUX_LAB3”.

Each of the “run_check(k)” will run the fifo_properties.sv file with the flag check(k) where k is in rage of 1 to 7

Due to the lengt of the log’s, I will not include them here, but they can be found here: /home/courses/desdigsys2/2023s/dds223s14/ex-1/lab3/test_fifo_check*.log

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

Task 1

CHECK # 1. Check that on reset

rd_ptr=0; wr_ptr=0; cnt=0; fifo_empty=1 and fifo_full=0

Q: Which property did you use? Explain it.

A:

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

Check that on reset
rd_ptr=0; wr_ptr=0; cnt=0; fifo_empty=1 and fifo_full=0

It does this by checking if the opesit of rst_ with !rst_ is 0. so when rst_ == 0, you get 1. I could aslo have used "`rd_ptr==0",

It then uses |-> to at the same clock cycle check Check that on reset
`rd_ptr==0 && `wr_ptr == 0 && `cnt == 0 && fifo_empty.

The && means or. This is to check multiple things at once, I could also have used ##0. ##0 can be better, because in can check each of the statements, one by one, therefor canceling once it hits a false statement. But due to readability I chose && logic. By testing this it shows a 0.01 slower compleition time comparid to ##0.

The "==" statement checks that both side are equal. It will return a booleon. I could also have used the "!" statement.

When I just use the variable name. It checks the value as a booleon, false if "fifo_emty < 1" true if "fifo_emty >= 1"

When all of the statements mentions are true, assert porpery will execute whats between "(check_reset)" and "else", because there is nothing there, nothing will be done. But if one of the statements afther the "|->" are false. It will execute the fail message behind the "else" statement at check_resetP assert property

Q:Explain the FAILS (or WARNINGS) you get in your log.

A:

```
#         15  rst_=0 clk=1 fifo_write=0 fifo_read=0 fifo_full=0 fifo_empty=x wr_ptr=0 rd_ptr=0 cnt=0
#         15        FAIL::check_reset
#
#         25  rst_=0 clk=1 fifo_write=0 fifo_read=0 fifo_full=0 fifo_empty=x wr_ptr=0 rd_ptr=0 cnt=0
#         25        FAIL::check_reset
```

But at both instances, "fifo_empty" does not have a value equal to, or greater than one. Therefor the statemenst fails and the assert prints whats behind the else statement.

Task 2

CHECK # 2. Check that fifo_empty is asserted when fifo 'cnt' is 0.

Disable this property 'iff (!rst)'

Q: Which property did you use? Explain it.

A:

```
`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");
```

The porperty is disable if rst_ is low, if not it chekcs if cnt is zero and at the same clock cycle check fifo_empty that it is high. I use |-> for this, because it checks them at the same clock cycle.

Q: Explain the FAILS (or WARNINGS) you get in your log.

A:

```
#        225  rst_=1 clk=1 fifo_write=0 fifo_read=1 fifo_full=0 fifo_empty=0 wr_ptr=8 rd_ptr=8 cnt=0
#        225          FAIL::fifo_empty condition
```

The fail at 225 comes beccause cnt is zero at the same time fifo_empty is zero.

Task 3

CHECK # 3. Check that fifo_full is asserted any time fifo 'cnt' is greater than 7.

Disable this property 'iff (!rst)'

Q: Which property did you use? Explain it.

A:

```
`ifdef check3
property fifofull;
  @(posedge clk) disable iff (!rst_) `cnt > 7 |-> fifo_full;
endproperty
fifofullP: assert property (fifofull) else $display($stime,"\t\t FAIL::fifo_full
condition\n");
```

When enabled(not reset) it will check if cnt is grater than 7, if this is the case, it will check that fifo_full is asserted, and because of |->, it will check at the same clock cykle.

Q: Explain the FAILS (or WARNINGS) you get in your log.

A:

```
#        125  rst_=1 clk=1 fifo_write=1 fifo_read=0 fifo_full=0 fifo_empty=0 wr_ptr=8 rd_ptr=0 cnt=8
#        125          FAIL::fifo_full condition
```

The failes accourse because reset is high(So it does not get disabled), cnt is 8, whitch is grater than 7 and fifo_full=0; if Fifo_full had been greater than, or equal to one. The error would not have came.

Task 4

CHECK # 4 Check that if fifo is full and you attempt to write (but not read) that the wr_ptr does not change.

Q: Which property did you use? Explain it.

A:

```
`ifdef check4
property fifo_full_write_stable_wrptr;
  @(posedge clk) 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");
```

we have three parameters to check for.

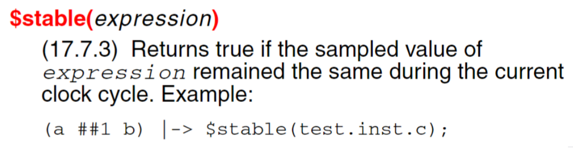
The first is that fifo is full, we do this by writing fifo_full, this will return true if it is high, letting us know its full.

The second is that we atemt to write, we do this the same way as fifo_full, but with fifo_read.

The third is (but not read), to do this, we find the opposite of the fifo_read value. We do this by using the "!" property. If fifo_read is high, it will return a false. If it is low, it will return a true.

We also have to check that wr_ptr is does not change. We can do this two ways, one way is to use the ##1, but we can also use |=> to check one clock cyckle later.

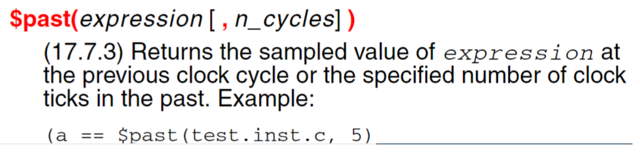
With $stable(vaklue) it will return a true if it is strable from the last clock cycle. We could also have used "`wr_ptr == $past(`wr_ptr)". But I personally prefer the word stable to descripe what we are doing, and therefor chose to stick with that due to readability.



$stable(*expression*)

(17.7.3) Returns true if the sampled value of `expression` remained the same during the current clock cycle. Example:

```
(a ##1 b) |-> $stable(test.inst.c);
```



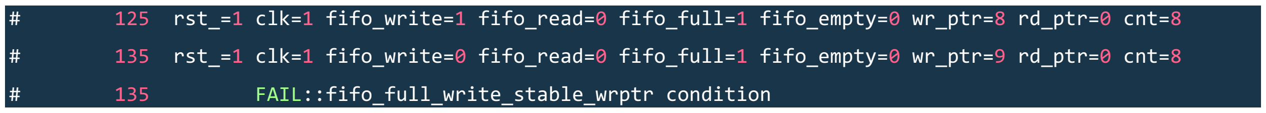
$past(*expression* [, *n_cycles*])

(17.7.3) Returns the sampled value of `expression` at the previous clock cycle or the specified number of clock ticks in the past. Example:

```
(a == $past(test.inst.c, 5)
```

Q: Explain the FAILS (or WARNINGS) you get in your log.

A:



```
#        125  rst_=1 clk=1 fifo_write=1 fifo_read=0 fifo_full=1 fifo_empty=0 wr_ptr=8 rd_ptr=0 cnt=8
#        135  rst_=1 clk=1 fifo_write=0 fifo_read=0 fifo_full=1 fifo_empty=0 wr_ptr=9 rd_ptr=0 cnt=8
#        135         FAIL::fifo_full_write_stable_wrptr condition
```

At time 125, the whole statement "fifo_full && fifo_write && !fifo_read" is true, therefor it will wait one clock cyckle to check the $stable(`wr_ptr). But because at time 135, wr_ptr have changed from 8 to 9, it therefor will be a false statement. Because of that the assert will fail, and output the message behind the "else" statement.

Task 5

CHECK # 5. Check that if fifo is empty and you attempt to read (but not write) that the rd_ptr does not change.

Q: Which property did you use? Explain it.

A:

```
`ifdef check5
property fifo_empty_read_stable_rdptr;
  @(posedge clk) fifo_empty && !fifo_write && fifo_read |=> $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");
```

Same principle as in task 4, just different variables.

Q: Explain the FAILS (or WARNINGS) you get in your log.

A:

```
#        225  rst_=1 clk=1 fifo_write=0 fifo_read=1 fifo_full=0 fifo_empty=1 wr_ptr=8 rd_ptr=8 cnt=0
#        235  rst_=1 clk=1 fifo_write=0 fifo_read=1 fifo_full=0 fifo_empty=1 wr_ptr=8 rd_ptr=9 cnt=0
#        235         FAIL::fifo_empty_read_stable_rdptr condition
#
```

Same principle as in task 4, just different variables.

Task 6

CHECK # 6. Write a property to Warn on write to a full fifo

Q: Which property did you use? Explain it.

A:

```
property write_on_full_fifo;
  @(posedge clk) fifo_write |-> !fifo_full;
endproperty
write_on_full_fifoP: assert property (write_on_full_fifo)
  else $display($stime,"\t\t WARNING::write_on_full_fifo\n");
```

first it checks if fifo_write is equal to or greater than 1. If this is true, it will do the same check for fifo_full at the same clock cyckle, but it will return the opposite for the second pard. This is because of "!".

Q:Explain the FAILS (or WARNINGS) you get in your log.

A:

```
#        125  rst_=1 clk=1 fifo_write=1 fifo_read=0 fifo_full=1 fifo_empty=0 wr_ptr=8 rd_ptr=0 cnt=8
#        125          WARNING::write_on_full_fifo
```

Because fifo_write is equal to one. It will then check if fifo_full is less than one. But because fifo_full is equal to one, it will fail and give out the warning message. This is a warning instead of an fail because it is stated so in the $display function.

Task 7

CHECK # 7. Write a property to Warn on read from an empty fifo

Q: Which property did you use? Explain it.

A:

```
`ifdef check7
property read_on_empty_fifo;
  @(posedge clk) 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");
```

Same principle as in task 6, just different variables.

Q: Explain the FAILS (or WARNINGS) you get in your log.

A:

```
#        225  rst_=1 clk=1 fifo_write=0 fifo_read=1 fifo_full=0 fifo_empty=1 wr_ptr=8 rd_ptr=8 cnt=0
#        225          WARNING::read_on_empty_fifo condition
#
#        235  rst_=1 clk=1 fifo_write=0 fifo_read=1 fifo_full=0 fifo_empty=1 wr_ptr=8 rd_ptr=8 cnt=0
#        235          WARNING::read_on_empty_fifo condition
```

Same principle as in task 6, just different variables. But this time, there is two instances where whats before |-> is true, and whats behind |-> is false.

Lab 4

Task 1

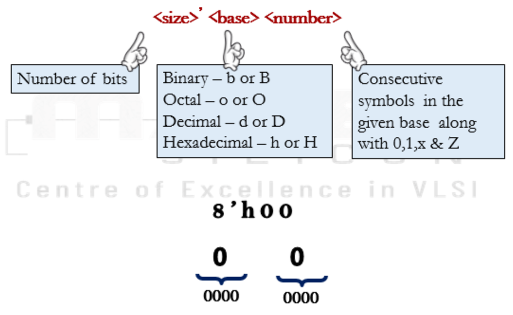
CHECK # 1. Check that when 'rst_' is asserted (==0) that data_out == 8'b0

Q: Which property did you use? Explain it.

A:

```
`ifdef check1
property counter_reset;
  @(posedge clk) !rst_ |-> data_out == 8'b0; //DUMMY - REMOVE this line and code
correct assertion
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
```

We use !rst_ to check if rst_ is asserted, if it is, the statement will return true and and the same clock cycle check that data_out is equal to 8'b0. 8'b0 = 8 bit, binary with value 0.



Source: https://www.quora.com/What-is-the-meaning-of-8h00-in-Verilog

Q: Explain the FAILS you get in your log.

A:

```
#          15   rst_=0 clk=1 count_enb=0 ld_cnt_=1 updn_cnt=1 DIN=x DOUT=x
#          15         COUNTER RESET CHECK FAIL:: rst_=0 data_out=x
#
```

```
#          25  rst_=0 clk=1 count_enb=0 ld_cnt_=1 updn_cnt=1 DIN=x DOUT=x
#          25         COUNTER RESET CHECK FAIL:: rst_=0 data_out=x
```

At time 15 and 25 both have rst_ asserted. This spawnes property counter_reset. Counter_reset checks if data_out /DOUT is equal to 8 bit, binary with value 0. Witch it is not, failing the assert, and outputting the message behind the "else" statement.

Task 2

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.

Q: Which property did you use? Explain it.

A:

```
`ifdef check2
property counter_hold;
  @(posedge clk) disable iff (!rst_) ld_cnt_ && !count_enb |=> $stable(data_out);
//DUMMY - REMOVE  this line and code correct assertion
endproperty
counter_hold_check: assert property(counter_hold)
  else $display($stime,,,"\t\tCOUNTER HOLD CHECK FAIL:: counter HOLD \n");
`endif
```

To check if ld_cnt_ is deasserted, we just use the name, this will return a boolean true if it is deasserted, we also check that count_enb is asserted with "!count_enb" this will return true of asserted.

Out group also had some difficulties understanding if the question. If the question wants us to see if data_out "holds its previous value" one would have to check the value of data_out previous to the assert being triggerd. To do this one might be able to use "|-> data_out == $past(data_out)" but in the ".solution" folder. It look like the question should be "Holds its value" seeing how the ".solution" folder checks if it holds its value after the assertion has been triggers in the next clock cycle.

This made for some confusion, seeing how the ".solution" also have some errors other places. In /home/courses/desdigsys2/2023s/dds223s14/ex-1/lab4/.solution/test_counter_check1.log there is also an error, at line 21 it prints out that the error comes at the time 20 but the assertian spawed in at time 15. The error should also come at time 15.

Q: Explain the FAILS you get in your log.

A:

```
#          35  rst_=1 clk=1 count_enb=0 ld_cnt_=1 updn_cnt=1 DIN=x DOUT=0
#         45  rst_=1 clk=1 count_enb=0 ld_cnt_=0 updn_cnt=1 DIN=0 DOUT=1
#         45        COUNTER HOLD CHECK FAIL:: counter HOLD
........................................................................................
#        175  rst_=1 clk=1 count_enb=0 ld_cnt_=1 updn_cnt=1 DIN=100 DOUT=110
#        185  rst_=1 clk=1 count_enb=0 ld_cnt_=1 updn_cnt=1 DIN=100 DOUT=111
#        185        COUNTER HOLD CHECK FAIL:: counter HOLD
#
#        195  rst_=1 clk=1 count_enb=0 ld_cnt_=0 updn_cnt=1 DIN=100 DOUT=112
#        195        COUNTER HOLD CHECK FAIL:: counter HOLD
```

At 35, 175 and 185, rst_ is deaserted, so it will check the first statement. ld_cnt_ && !count_enb. And because they both are true, it will then wait one clock cycle and run $stable(data_out). And because there is a change between the 35-45, 175-185 and 185-195, we get three fails.

Task 3

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

Q: Which property did you use? Explain it.

A:

```
property counter_count;
  @(posedge clk)  disable iff (!rst_) ld_cnt_ && count_enb |-> updn_cnt ##1
data_out == ($past(data_out) + 1) or !updn_cnt ##1 data_out == ($past(data_out) -
1); //DUMMY - REMOVE  this line and code correct assertion
endproperty
```

disable iff (!rst_) checks if it reset is deasserted, that it will not run. Then it will check if ld_cnt_ and count_enb as deasserted. If this is true, then it will do two things at the same clock cyckle:

1. check if updn_cnt is 1, wait 1 clock cycle, then check if data_out at that time, is the same as what it was in the previous clock cycle, plus one. Or in other words, if it increment (++).
2. check if updn_cnt is 0, wait 1 clock cycle, then check if data_out at that time, is the same as what it was in the previous clock cycle, minus one. Or in other words, if it decremented (--).

If one of the following statements is true, the statement behind the |-> will be true.

Q: Explain the FAILS you get in your log.

A:

```
#          75  rst_=1 clk=1 count_enb=1 ld_cnt_=1 updn_cnt=1 DIN=100 DOUT=100
#          85  rst_=1 clk=1 count_enb=1 ld_cnt_=1 updn_cnt=1 DIN=100 DOUT=99
#          85        COUNTER COUNT CHECK FAIL:: UPDOWN COUNT using $past
#
#          95  rst_=1 clk=1 count_enb=1 ld_cnt_=1 updn_cnt=1 DIN=100 DOUT=98
#          95        COUNTER COUNT CHECK FAIL:: UPDOWN COUNT using $past
#
#         105  rst_=1 clk=1 count_enb=1 ld_cnt_=1 updn_cnt=1 DIN=100 DOUT=97
#         105        COUNTER COUNT CHECK FAIL:: UPDOWN COUNT using $past
#
#         115  rst_=1 clk=1 count_enb=1 ld_cnt_=1 updn_cnt=1 DIN=100 DOUT=96
#         115        COUNTER COUNT CHECK FAIL:: UPDOWN COUNT using $past
#
#         125  rst_=1 clk=1 count_enb=1 ld_cnt_=1 updn_cnt=1 DIN=100 DOUT=95
#         125        COUNTER COUNT CHECK FAIL:: UPDOWN COUNT using $past
#
#         135  rst_=1 clk=1 count_enb=1 ld_cnt_=1 updn_cnt=1 DIN=100 DOUT=94
#         135        COUNTER COUNT CHECK FAIL:: UPDOWN COUNT using $past
#
#         145  rst_=1 clk=1 count_enb=1 ld_cnt_=1 updn_cnt=1 DIN=100 DOUT=93
#         145        COUNTER COUNT CHECK FAIL:: UPDOWN COUNT using $past
#
#         155  rst_=1 clk=1 count_enb=1 ld_cnt_=1 updn_cnt=1 DIN=100 DOUT=92
#         155        COUNTER COUNT CHECK FAIL:: UPDOWN COUNT using $past
#
#         165  rst_=1 clk=1 count_enb=1 ld_cnt_=1 updn_cnt=1 DIN=100 DOUT=91
#         165        COUNTER COUNT CHECK FAIL:: UPDOWN COUNT using $past
#
#         175  rst_=1 clk=1 count_enb=0 ld_cnt_=1 updn_cnt=1 DIN=100 DOUT=90
#         175        COUNTER COUNT CHECK FAIL:: UPDOWN COUNT using $past
#
#         185  rst_=1 clk=1 count_enb=0 ld_cnt_=1 updn_cnt=1 DIN=100 DOUT=90
#         195  rst_=1 clk=1 count_enb=0 ld_cnt_=0 updn_cnt=1 DIN=100 DOUT=90
#         205  rst_=1 clk=1 count_enb=1 ld_cnt_=1 updn_cnt=0 DIN=100 DOUT=100
#         215  rst_=1 clk=1 count_enb=1 ld_cnt_=1 updn_cnt=0 DIN=100 DOUT=101
#         215        COUNTER COUNT CHECK FAIL:: UPDOWN COUNT using $past
#
#         225  rst_=1 clk=1 count_enb=1 ld_cnt_=1 updn_cnt=0 DIN=100 DOUT=102
#         225        COUNTER COUNT CHECK FAIL:: UPDOWN COUNT using $past
#
#         235  rst_=1 clk=1 count_enb=1 ld_cnt_=1 updn_cnt=0 DIN=100 DOUT=103
#         235        COUNTER COUNT CHECK FAIL:: UPDOWN COUNT using $past
```

```
#
#        245  rst_=1 clk=1 count_enb=1 ld_cnt_=1 updn_cnt=0 DIN=100 DOUT=104
#        245        COUNTER COUNT CHECK FAIL:: UPDOWN COUNT using $past
#
#        255  rst_=1 clk=1 count_enb=1 ld_cnt_=1 updn_cnt=0 DIN=100 DOUT=105
#        255        COUNTER COUNT CHECK FAIL:: UPDOWN COUNT using $past
#
#        265  rst_=1 clk=1 count_enb=1 ld_cnt_=1 updn_cnt=0 DIN=100 DOUT=106
#        265        COUNTER COUNT CHECK FAIL:: UPDOWN COUNT using $past
#
#        275  rst_=1 clk=1 count_enb=1 ld_cnt_=1 updn_cnt=0 DIN=100 DOUT=107
#        275        COUNTER COUNT CHECK FAIL:: UPDOWN COUNT using $past
#
#        285  rst_=1 clk=1 count_enb=1 ld_cnt_=1 updn_cnt=0 DIN=100 DOUT=108
#        285        COUNTER COUNT CHECK FAIL:: UPDOWN COUNT using $past
#
#        295  rst_=1 clk=1 count_enb=1 ld_cnt_=1 updn_cnt=0 DIN=100 DOUT=109
#        295        COUNTER COUNT CHECK FAIL:: UPDOWN COUNT using $past
#
#        305  rst_=1 clk=1 count_enb=0 ld_cnt_=1 updn_cnt=0 DIN=100 DOUT=110
#        305        COUNTER COUNT CHECK FAIL:: UPDOWN COUNT using $past
```

Every fail comes because the following:

One clock cycle before the fail, ld_cnt_ is deasserted (==1) and count_enb is enabled

(==1).

For the fail [85, 95, 105, 115, 125, 135, 145, 155, 165, 175] it was looking a increment in DOUT compared to the previous clock cycle, this is because updn_count deasserted. And because the DOUT did not increment, it printed out the FAIL

For the fail [215, 225, 235, 245, 255, 265, 275, 285, 295, 305] it was looking a decrement in DOUT compared to the previous clock cycle, this is because updn_count asserted. And because the DOUT did not idecrement, it printed out the FAIL

Lab 5

Task 1

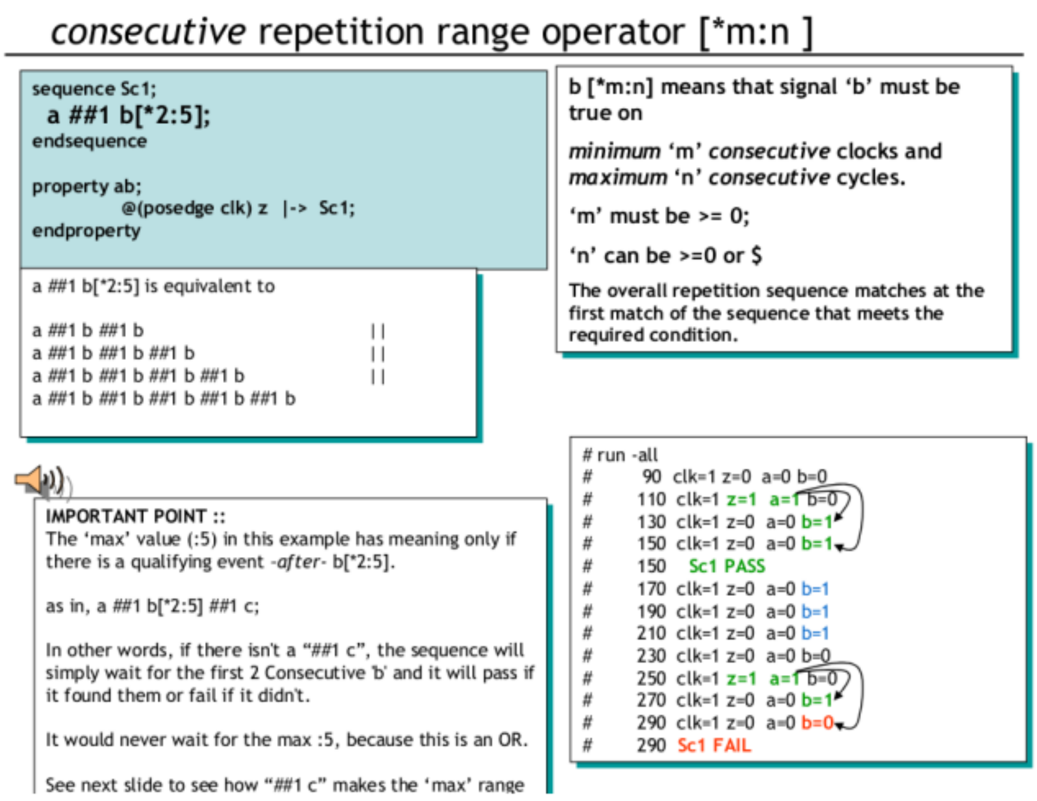
CHECK # 1. Check that once dValid goes high that it is consecutively asserted (high) for minimum 2 and maximum 4 clocks

Q: Which property did you use? Explain it.

A:

```
`ifdef check1
        property checkValid;
          @(posedge clk) $rose(dValid) |=> dValid[*2:4] ##1 !dValid;
        endproperty
        assert property (checkValid) else $display($stime,,,"checkValid FAIL");
`endif
```

When dValid has a rising edge, it will start. It will then wait one clock cycle, then it will check that dValid remains high for between two to four clock cycles, and when it hits its limit of four, or falls, it will check that dValid remains low one clock cycle later.



consecutive repetition range operator [*m:n]

```
sequence Sc1;
  a ##1 b[*2:5];
endsequence

property ab;
          @(posedge clk) z  |-> Sc1;
endproperty
```

a ##1 b[*2:5] is equivalent to

```
a ##1 b ##1 b                          ||
a ##1 b ##1 b ##1 b                    ||
a ##1 b ##1 b ##1 b ##1 b              ||
a ##1 b ##1 b ##1 b ##1 b ##1 b
```

b [*m:n] means that signal 'b' must be true on

minimum 'm' *consecutive* clocks and *maximum* 'n' *consecutive* cycles.

'm' must be >= 0;

'n' can be >=0 or $

The overall repetition sequence matches at the first match of the sequence that meets the required condition.

IMPORTANT POINT ::
The 'max' value (:5) in this example has meaning only if there is a qualifying event -*after*- b[*2:5].

as in, a ##1 b[*2:5] ##1 c;

In other words, if there isn't a "##1 c", the sequence will simply wait for the first 2 Consecutive 'b' and it will pass if it found them or fail if it didn't.

It would never wait for the max :5, because this is an OR.

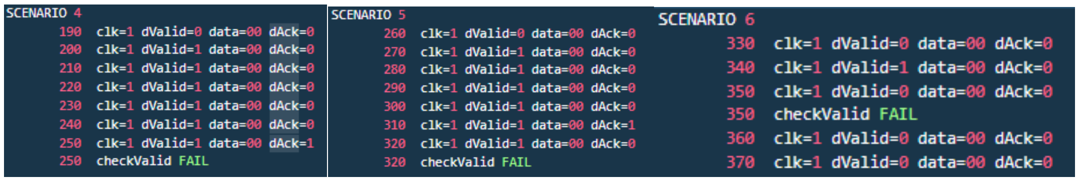
See next slide to see how "##1 c" makes the 'max' range

```
# run -all
#        90  clk=1 z=0  a=0 b=0
#       110  clk=1 z=1  a=1 b=0
#       130  clk=1 z=0  a=0 b=1
#       150  clk=1 z=0  a=0 b=1
#       150    Sc1 PASS
#       170  clk=1 z=0  a=0 b=1
#       190  clk=1 z=0  a=0 b=1
#       210  clk=1 z=0  a=0 b=1
#       230  clk=1 z=0  a=0 b=0
#       250  clk=1 z=1  a=1 b=0
#       270  clk=1 z=0  a=0 b=1
#       290  clk=1 z=0  a=0 b=0
#       290  Sc1 FAIL
```

Source: Blackboard: 2023.02.02; SystemVerilog Assertions

Q: Explain the FAILS you get in your log

A:



```
SCENARIO 4
       190  clk=1 dValid=0 data=00 dAck=0
       200  clk=1 dValid=1 data=00 dAck=0
       210  clk=1 dValid=1 data=00 dAck=0
       220  clk=1 dValid=1 data=00 dAck=0
       230  clk=1 dValid=1 data=00 dAck=0
       240  clk=1 dValid=1 data=00 dAck=0
       250  clk=1 dValid=1 data=00 dAck=1
       250  checkValid FAIL
```

```
SCENARIO 5
       260  clk=1 dValid=0 data=00 dAck=0
       270  clk=1 dValid=1 data=00 dAck=0
       280  clk=1 dValid=1 data=00 dAck=0
       290  clk=1 dValid=1 data=00 dAck=0
       300  clk=1 dValid=1 data=00 dAck=0
       310  clk=1 dValid=1 data=00 dAck=1
       320  clk=1 dValid=1 data=00 dAck=0
       320  checkValid FAIL
```

```
SCENARIO 6
       330  clk=1 dValid=0 data=00 dAck=0
       340  clk=1 dValid=1 data=00 dAck=0
       350  clk=1 dValid=0 data=00 dAck=0
       350  checkValid FAIL
       360  clk=1 dValid=0 data=00 dAck=0
       370  clk=1 dValid=0 data=00 dAck=0
```

Scenario 4

dValid high for longer than four clock cycle

scenario 5

dValid high for longer than four clock cycle

scenario 6

dValid high under the minimal requirements of 2 clock cycles. It is only high for 1 clock cycle

Task 2

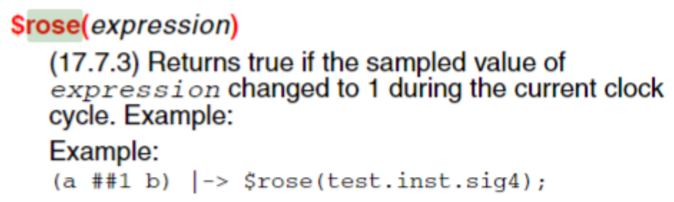
CHECK # 2. Check that data is not unknow and remains stable after dValid goes high and until dAck goes high.

Q: Which property did you use? Explain it.

A:

```
`ifdef check2
        property checkdataValid;
         @(posedge clk) disable iff (reset)
          @(posedge clk)  $rose(dValid) |-> ($stable(data) and !$isunknown(data))
until_with dAck;
        endproperty
        assert property (checkdataValid) else $display($stime,,,"checkdataValid
FAIL");
`endif
```

First it checks rising edge. Then it at the same clock cycle checks if data is stable, meaning it has not changed AND Also it checks if it is unknown.

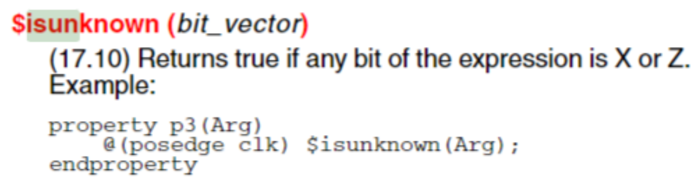


$rose(*expression*)

(17.7.3) Returns true if the sampled value of *expression* changed to 1 during the current clock cycle. Example:

Example:

```
(a ##1 b) |-> $rose(test.inst.sig4);
```



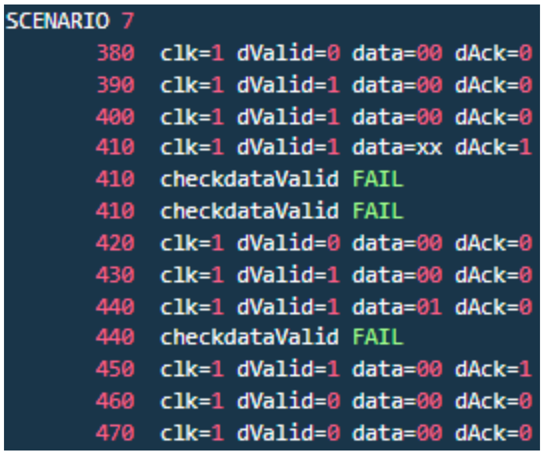
$isunknown (*bit_vector*)

(17.10) Returns true if any bit of the expression is X or Z.
Example:

```
property p3(Arg)
    @(posedge clk) $isunknown(Arg);
endproperty
```

Q: Explain the FAILS you get in your log

A:



```
SCENARIO 7
        380  clk=1 dValid=0 data=00 dAck=0
        390  clk=1 dValid=1 data=00 dAck=0
        400  clk=1 dValid=1 data=00 dAck=0
        410  clk=1 dValid=1 data=xx dAck=1
        410  checkdataValid FAIL
        410  checkdataValid FAIL
        420  clk=1 dValid=0 data=00 dAck=0
        430  clk=1 dValid=1 data=00 dAck=0
        440  clk=1 dValid=1 data=01 dAck=0
        440  checkdataValid FAIL
        450  clk=1 dValid=1 data=00 dAck=1
        460  clk=1 dValid=0 data=00 dAck=0
        470  clk=1 dValid=0 data=00 dAck=0
```

410: one fail for invalid data, one fail because data has changed

440: one fail because data changed.

Task 3

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

Q: Which property did you use? Explain it.

A:

```
`ifdef check3
        property checkdAck;
          @(posedge clk) $rose(dValid) |-> dValid[*2:4] ##1 dAck ##1
$fell(dValid); //DUMMY - REMOVE this line and code correct assertion
        endproperty
        assert property (checkdAck) else $display($stime,,,"checkdAck FAIL");
`endif
```

Checks for rising edge.

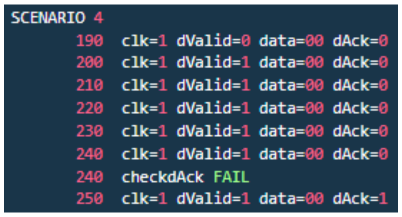
Same clock cycle checks that dValid is high between the interval of two and four clock cycle.

Waits one clock cyckle and then checks if dAck is high.

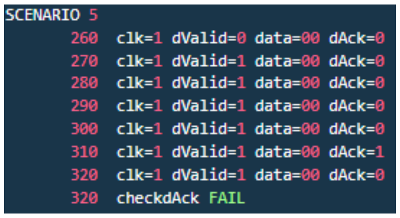
Checks that dValid has a falling edge.

Q: Explain the FAILS you get in your log

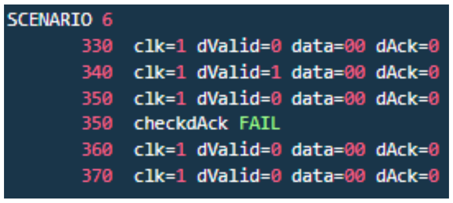
A:



```
SCENARIO 4
        190  clk=1 dValid=0 data=00 dAck=0
        200  clk=1 dValid=1 data=00 dAck=0
        210  clk=1 dValid=1 data=00 dAck=0
        220  clk=1 dValid=1 data=00 dAck=0
        230  clk=1 dValid=1 data=00 dAck=0
        240  clk=1 dValid=1 data=00 dAck=0
        240  checkdAck FAIL
        250  clk=1 dValid=1 data=00 dAck=1
```



```
SCENARIO 5
        260  clk=1 dValid=0 data=00 dAck=0
        270  clk=1 dValid=1 data=00 dAck=0
        280  clk=1 dValid=1 data=00 dAck=0
        290  clk=1 dValid=1 data=00 dAck=0
        300  clk=1 dValid=1 data=00 dAck=0
        310  clk=1 dValid=1 data=00 dAck=1
        320  clk=1 dValid=1 data=00 dAck=0
        320  checkdAck FAIL
```



```
SCENARIO 6
        330  clk=1 dValid=0 data=00 dAck=0
        340  clk=1 dValid=1 data=00 dAck=0
        350  clk=1 dValid=0 data=00 dAck=0
        350  checkdAck FAIL
        360  clk=1 dValid=0 data=00 dAck=0
        370  clk=1 dValid=0 data=00 dAck=0
```

Scenario 4: dValid higher hits the maximum of four clock cycle and then checks one clock cycle later for dAck high. dAck is low at 240. It will then fail.

Scenario 5: dValid higher hits the maximum of four clock cycle and then checks one clock cycle later for dAck high. dAck is high at 310. But it fails when it does not see a falling edge at 320.

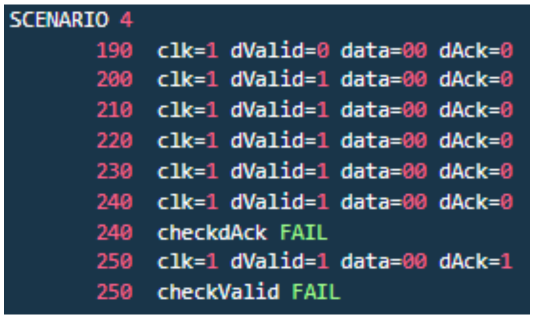
Scenario 6: dValid low afther the initiation of the rising edge.

Task 4

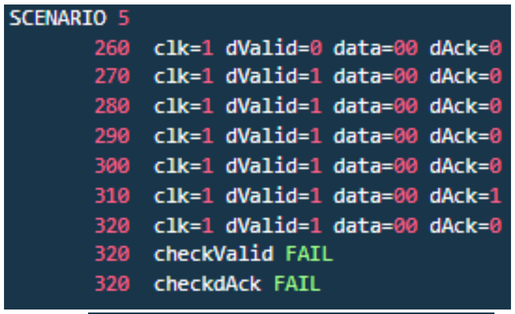
CHECK # ALL

Q: Explain the FAILS you get in your log.

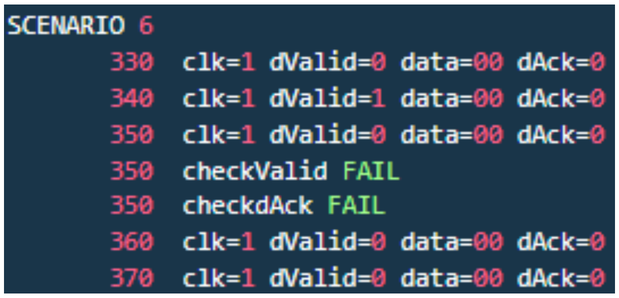
A:



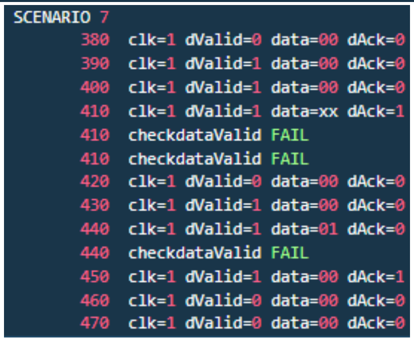
```
SCENARIO 4
        190   clk=1 dValid=0 data=00 dAck=0
        200   clk=1 dValid=1 data=00 dAck=0
        210   clk=1 dValid=1 data=00 dAck=0
        220   clk=1 dValid=1 data=00 dAck=0
        230   clk=1 dValid=1 data=00 dAck=0
        240   clk=1 dValid=1 data=00 dAck=0
        240   checkdAck FAIL
        250   clk=1 dValid=1 data=00 dAck=1
        250   checkValid FAIL
```



```
SCENARIO 5
        260   clk=1 dValid=0 data=00 dAck=0
        270   clk=1 dValid=1 data=00 dAck=0
        280   clk=1 dValid=1 data=00 dAck=0
        290   clk=1 dValid=1 data=00 dAck=0
        300   clk=1 dValid=1 data=00 dAck=0
        310   clk=1 dValid=1 data=00 dAck=1
        320   clk=1 dValid=1 data=00 dAck=0
        320   checkValid FAIL
        320   checkdAck FAIL
```



```
SCENARIO 6
        330   clk=1 dValid=0 data=00 dAck=0
        340   clk=1 dValid=1 data=00 dAck=0
        350   clk=1 dValid=0 data=00 dAck=0
        350   checkValid FAIL
        350   checkdAck FAIL
        360   clk=1 dValid=0 data=00 dAck=0
        370   clk=1 dValid=0 data=00 dAck=0
```



```
SCENARIO 7
        380   clk=1 dValid=0 data=00 dAck=0
        390   clk=1 dValid=1 data=00 dAck=0
        400   clk=1 dValid=1 data=00 dAck=0
        410   clk=1 dValid=1 data=xx dAck=1
        410   checkdataValid FAIL
        410   checkdataValid FAIL
        420   clk=1 dValid=0 data=00 dAck=0
        430   clk=1 dValid=1 data=00 dAck=0
        440   clk=1 dValid=1 data=01 dAck=0
        440   checkdataValid FAIL
        450   clk=1 dValid=1 data=00 dAck=1
        460   clk=1 dValid=0 data=00 dAck=0
        470   clk=1 dValid=0 data=00 dAck=0
```

Scenario 4:

CheckAdc: 340: rising edge initate checkAdc, but it does not see a high value for dValid for the event at time 350

CheckValid : 340: rising edge initate checkAdc, but it does not see a high value for dValid for the event at time 350

Scenario 5:

Both CheckValid and CheckAdc start with a rising edge. The next clock cycle it checks that it will be asserted between two and four clock cycles. But it stays on for <u>longer</u>, it then fails both.

Scenario 6:

Both CheckValid and CheckAdc start with a rising edge. The next clock cycle it checks that it will be asserted between two and four clock cycles. But it stays on for <u>shorter</u>, it then fails both.

Scenario 7:

410: one fail for invalid data, one fail because data has changed

440: one fail because data changed.

Lab 6

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.

Task 1

Q: Which property did you use? Explain it.

A:

```
`ifdef check1
        property checkPCI_AD_CBE;
          @(posedge clk)
                $fell(FRAME_) |-> !($isunknown(AD) || $isunknown(C_BE_));
        endproperty
        assert property (checkPCI_AD_CBE) else
$display($stime,,,"CHECK1:checkPCI_AD_CBE FAIL\n");
`endif
```

It checks for a falling edge in FRAME_

If true, it will at the same time check if AD or C_BE_ is unknown, and return the opposite.

If one of them is unknown the statement within "()" will be 1. And the opposite is 0.

If both of them is unknown, the opposite will be 0.

If none of them is unknown, the opposite will be 1.

Q: Explain the results you get in your log.
A:

```
#         30  clk=1 reset_=1 FRAME_=0 AD=zzzzzzzz C_BE_=0110 IRDY_=1 TRDY_=x DEVSEL_=1
#         30  CHECK1:checkPCI_AD_CBE FAIL
```

AD is unknown, making the “!()” statement false, failing the assert.

Task 2

Q: Which property did you use? Explain it.

A:

```
`ifdef check2
        property checkPCI_DataPhase;
          @(posedge clk)
                !IRDY_ && !TRDY_ |-> !($isunknown(AD) || $isunknown(C_BE_));
        endproperty
        assert property (checkPCI_DataPhase) else
$display($stime,,,"CHECK2:checkPCI_DataPhase FAIL\n");
`endif
```

Checks if IRDY and TRDY is both low.

At the same clock cycle, checks if AD and C_BE is not unknown. It does this the same way as in task 1.

Q: Explain the results you get in your log.
A:

```
#   Data Transfer Phase
#         50  clk=1 reset_=1 FRAME_=0 AD=zzzzzzzz C_BE_=1111 IRDY_=0 TRDY_=0 DEVSEL_=0
#         50  CHECK2:checkPCI_DataPhase FAIL
#   Data Transfer Phase
#         70  clk=1 reset_=1 FRAME_=0 AD=zzzzzzzz C_BE_=1111 IRDY_=0 TRDY_=0 DEVSEL_=0
#         70  CHECK2:checkPCI_DataPhase FAIL
#   Data Transfer Phase
#         90  clk=1 reset_=1 FRAME_=0 AD=zzzzzzzz C_BE_=1111 IRDY_=0 TRDY_=0 DEVSEL_=0
#         90  CHECK2:checkPCI_DataPhase FAIL
```

All the fails occurs because AD is unknown.

Task 3

Q: Which property did you use? Explain it.

A:

```
`ifdef check3
        property checkPCI_Frame_Irdy;
          @(posedge clk)
                $rose(FRAME_) |-> !IRDY_;
        endproperty
        assert property (checkPCI_Frame_Irdy) else
$display($stime,,,"CHECK3:checkPCI_frmIrdy FAIL\n");
`endif
```

It checks for a rising edge on FRAME and at the same clock cycle checks that IRDY is low.

Q: Explain the results you get in your log.
A:

```
#   FRAME_ De-asserted
#        100  clk=1 reset_=1 FRAME_=1 AD=zzzzzzzz C_BE_=1111 IRDY_=1 TRDY_=1 DEVSEL_=1
#        100  CHECK3:checkPCI_frmIrdy FAIL
```

Fail because IRDY is not low.

Task 4

Q: Which property did you use? Explain it.

A:

```
`ifdef check4
        property checkPCI_trdyDevsel;
          @(posedge clk)
                !TRDY_ |-> !DEVSEL_;
        endproperty
        assert property (checkPCI_trdyDevsel) else
$display($stime,,,"CHECK4:checkPCI_trdyDevsel FAIL\n");
`endif
```

Checks that TRDY is low, and at the same clock cycle checks tat DEVSEL is low.

Q: Explain the results you get in your log.
A:

```
#   Data Transfer Phase
#         50  clk=1 reset_=1 FRAME_=0 AD=cafecafe C_BE_=1111 IRDY_=0 TRDY_=0 DEVSEL_=1
#         50  CHECK4:checkPCI_trdyDevsel FAIL
#
```

```
#   Data Transfer Phase
#         70  clk=1 reset_=1 FRAME_=0 AD=faceface C_BE_=1111 IRDY_=0 TRDY_=0 DEVSEL_=1
#         70  CHECK4:checkPCI_trdyDevsel FAIL
#
#
#   Master Wait Mode
#         80  clk=1 reset_=1 FRAME_=0 AD=cafeface C_BE_=1111 IRDY_=1 TRDY_=0 DEVSEL_=1
#         80  CHECK4:checkPCI_trdyDevsel FAIL
#
#
#   Data Transfer Phase
#         90  clk=1 reset_=1 FRAME_=0 AD=cafeface C_BE_=1111 IRDY_=0 TRDY_=0 DEVSEL_=1
#         90  CHECK4:checkPCI_trdyDevsel FAIL
```
All of the fails occurs because TRDY is low, spawning the assert, but all of them have DEVSEL high, thus failing the assert.

Task 5

Q: Which property did you use? Explain it.

A:
```
`ifdef check5
        property checkPCI_CBE_during_trx;
          @(posedge clk) disable iff (!reset_)
                $fell(FRAME_) |-> !$isunknown(C_BE_) until_with FRAME_;
        endproperty
        assert property (checkPCI_CBE_during_trx) else
$display($stime,,,"CHECK5:checkPCI_CBE_during_trx FAIL\n");
`endif
```
Checks that FRAME has a falling edge, then at the same clock cycle checks that C_BE is known until the next clock cycle afther Frame goes high.

Weak & Non-Overlapping	until	property_expr1 until property_expr2
Strong & Non-Overlapping	s_until	property_expr1 s_until property_expr2
Weak & Overlapping	until_with	property_expr1 until_with property_expr2
Strong & Overlapping	s_until_with	property_expr1 s_until_with property_expr2

Source: https://vlsi.pro/sva-properties-iv-until-property/

Q: Explain the results you get in your log.
A:

```
#   Data Transfer Phase
#         90  clk=1 reset_=1 FRAME_=0 AD=cafeface C_BE_=1111 IRDY_=0 TRDY_=0 DEVSEL_=0
#   FRAME_ De-asserted
#        100  clk=1 reset_=1 FRAME_=1 AD=zzzzzzzz C_BE_=zzzz IRDY_=0 TRDY_=1 DEVSEL_=1
#        100  CHECK5:checkPCI_CBE_during_trx FAIL
```

Fail because FRAME just turned high, not giving time for C_BE to be allowed unknow status. Thus failing.