TDT4171 exercise 1 report

Morten Paulsen
Magnus Ramsfjell

February 1, 2020

1 Lab 1

We set up the external server as described. We then completed the exercises in lab 1 as explained in the following sections.

1.1 Task a - Binding

To bind the design (dut) to its property module (dut_property) we used the `bind` keyword.

```
bind dut_1 dut_property dut_bind_inst (
    .pclk(clk),
    .preq(req),
    .pgnt(gnt)
);
```

1.2 Task b - run_no_implication

Running a `diff` to compare our binding with the provided solutions binding gave only differences in the comments, showing that out binding was successful. We then ran the run_no_implication script and answered the following questions:

- Why is there a fail -and- a pass at time (70)?

  The assertion checks the property both at the current and past clock cycles. This is why at time 70 it both passes the assertion from time 30 (two clock periods ago) and fails the current assertion because the `preq`-signal is 0.

- Why are there two fails at time (130)?

  The assertions fail at time 130 because both `pgnt` and `preq` is 0 when both are expected to be 1.

1.3 Task c - run_implication

Running the run_implication script and `diff`ing the output shows that once again the binding was correct. Similarly to the previous task we answered some questions:

- Why are there 2 passes at time 70?

  There are 2 passes at time 70 because the assertion at time 30 passes as well as the vacuous success at time 70 (the antecedent `preq` is 0).

- Why is there a pass -and- a fail at time 130?

  The pass at time 130 is due to the vacuous success from `preq` being 0. The fail is because `pgnt` is not set despite the request at time 90.

1.4 Task d - implication_novac

Yes, we got the expected result. The difference from the previous tasks is that the vacuous successes are not reported in the log.

2 Lab 2

Lab 2 familiarizes us with overlapping and non-overlapping implication operators.

2.1 Task a - overlap

Questions:

- Why does the property fail at 30?

  `cstart` is 1 when `req` is 0. This violates the `pr1` property.

- Why does the property pass at 90?

  The property that passes at time 90 is due to `cstart` and `req` being 1 at time 50 and `gnt` being set to 1 two clock cycles later at time 90.

- Why does the property pass at 150?

  The property passes at 150 for the same reason as it passes at 90: `cstart` and `req` being 1 at time 110 and `gnt` being set to 1 two clock cycles later.

- Why does the property fail at 170?

  The request from time 130 is not satisfied at time 170 (`gnt` is 0 at time 170).

- Why does the property fail at 190?

  The request from time 150 is not satisfied at time 190 (`gnt` is 0 at time 190).

2.2 Task b - non_overlap

The difference between the overlapping and non overlapping properties code-wise is that a normal implication is invoked with `|->` while a non overlapping implication is invoked with `|=>`.

Questions:

- Why does the property fail at 70?

  `cstart` is 1 at time 50 without `req` being set to 1 at time 70 is why the property fails at time 70.

- Why does the property pass at 90?

  The property passes at time 90 because the sequence that starts at time 30 with `cstart=1` is satisfied when the grant signal is high at time 90.

- Why does the property fail at 170?

  A sequence is started at time 110 when `cstart` is set to 1 that is not met because `gnt` is not set high at time 170.

- Why does the property fail at 190?

  A sequence is started at time 130 when `cstart` is set to 1 that is not met because `gnt` is not set high at time 190.

- Why does the property pass at 210?

  The sequence started at time 150 when `cstart` is 1 is fulfilled by `req` being high one clock cycle later as well as `gnt` being set to 1 at time 210, two clock cycles after that.

3 Lab 3

Lab 3 familiarizes us with a FIFO queue and how to test it.

3.1 Task a - run_nobugs

The FIFO worked as expected in the *nobugs* case.

3.2 Task b - run_check

3. The test gave the same results as the solutions log. The FAIL in the test was that `fifo_empty` was the value `x` (unknown) rather than 0 after a reset. The simulation fails at time 15 and 25. Code added:

```
@(posedge clk) !rst_ |->
‘rd_ptr==0 && ‘wr_ptr == 0 && ‘cnt == 0 && fifo_empty && !fifo_full;
```

   This asserts that the ‘`rd_ptr`, ‘`wr_ptr`, ‘`cnt`, `fifo_empty` and `fifo_full` had the indicated values when `rst_` was deasserted.

4. Asserting that the correct signal is set when the fifo is empty. The simulation fails at time 225. Code added:

```
@(posedge clk) disable iff (!rst_) ‘cnt == 0 |-> fifo_empty;
```

   This asserts that `fifo_empty` is logic high on the same positive clock edge as ‘`cnt==0` when `rst_` is asserted.

5. Asserting that the correct signal is set when the fifo is empty. The simulation fails at time 125. Code added:

```
@(posedge clk) disable iff (!rst_) ‘cnt > 7 |-> fifo_full;
```

   This asserts that `fifo_full` is logic high on the same positive clock edge as ‘`cnt > 7` when `rst_` is asserted.

6. Attempts to write to a full fifo should not increment the write pointer. The simulation fails at time 135. Code added:

```
@(posedge clk) disable iff (!rst_) fifo_full && fifo_write && !fifo_read |=>
‘wr_ptr == $past(‘wr_ptr);
```

   This asserts that ‘`wr_ptr==$past(‘wr_ptr)` one clock cycle after `fifo_full` and `fifo_write` are asserted and `fifo_read` is deasserted.

7. Attempts to read from an empty fifo should not decrement the read pointer. The simulation fails at time 235. Code added:

```
@(posedge clk) disable iff (!rst_) fifo_empty && fifo_read && !fifo_write |=>
‘rd_ptr == $past(‘rd_ptr);
```

   This asserts that ‘`rd_ptr==$past(‘rd_ptr)` one clock cycle after `fifo_empty` and `fifo_read` are asserted and `fifo_write` is deasserted.

8. When writing to the fifo (without reading) the fifo should not be full. The simulation gives a warning at time 125. Code added:

```
@(posedge clk) disable iff (!rst_) fifo_write && !fifo_read |-> !fifo_full;
```

   This asserts that `fifo_full` is deasserted on the same clock edge as when `fifo_write` is asserted and `fifo_read` is deasserted

9. When reading from the fifo (without writing) the fifo should not be empty. The simulation gives a warning at time 225 and 235. Code added:

```
@(posedge clk) disable iff (!rst_) fifo_read && !fifo_write |-> !fifo_empty;
```

This asserts that `fifo_empty` is deasserted on the same clock edge as when `fifo_read` is asserted and `fifo_write` is deasserted.

4 Lab 4

4.1 Task a - run_nobugs

The counter worked as expected where it counted in both directions based on the input signals.

4.2 Task b - run_check

4. When resetting the counter should put `8'b0` on the `data_out` output bus. Code added:

```
@(posedge clock) (!rst_) |-> data_out == 8'b0;
```

This asserts that the `data_out` signal is reset on the same positive clock edge as `rst_`.

5. When `ld_cnt_` and `count_enb` is deasserted the counter should keep its value. Code added:

```
@(posedge clk) disable iff (!rst_) ld_cnt_ && !count_enb |=>
data_out == $past(data_out);
```

This asserts that `data_out == $past(data_out)` one clock cycle after both `ld_cnt_` and `count_enb` are deasserted.

6. When `ld_cnt_` is deasserted and `count_enb` is asserted the counter should count upwards (`updn_cnt == 1`) or downwards (`updn_cnt == 0`). Code added:

```
@(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));
```

This asserts than `data_out` is incremented one clock cycle after `ld_cnt_` is deasserted and `count_enb` and `updn_cnt` is asserted. It also asserts that `data_out` is decremented one clock cycle after `count_enb` is asserted and `ld_cnt_` and `updn_cnt` is deasserted.

5 Lab 5

5.1 Task a - run_nobugs

The bus works as expected when running the `run_nobugs` design.

5.2 Task b - run_check

3. `check_1` asserts that `dValid` behaves as expected. The simulation fails at times 250 and 320 when `dValid` remains high for too long and at time 350 where `dValid` is only high for one clock cycle instead of the required two. Code added:

```
@(posedge clk) disable iff (reset) $rose(dValid) |=>
 dValid [*2] ##[1:3] $fell(dValid);
```

4. `check_2` asserts that the data is not unknown (`'X'`) and remains stable after `dValid` goes high until an acknowledge signal is received (even if `dAck` is set high after `dValid` has gone back low). The simulation fails at times 410 (two fails) and 440. At time 410 data goes from a stable known signal value to unknown. This assertion is triggered twice by `dValid` going high at times 340 and 390. At time 440 the data changes from 0 to 1 when it should remain stable. Code added:

```
@(posedge clk) disable iff (reset) $rose(dValid) |=>
 (!$isunknown(data) && $stable(data) throughout !dAck ##[1:$] $rose(dAck));
```

5. `check_3` asserts that once the transfer is acknowledged by `dAck`, `dValid` goes low one clock cycle later. It also checks that `dValid` is high until an acknowledge signal is received one to three cycles after the initialization of the transaction. The simulation fails at times 240, 320 and 350. At time 240 `dAck` goes high on the fifth cycle after `dValid` rose, when it has to acknowledge within four cycles. At time 320 `dValid` remains erroneously high the cycle after `dAck` is received. Code added:

```
@(posedge clk) disable iff (reset) $rose(dValid) |=>
 (dValid && !dAck) [*1:3] ##1 $rose(dAck) ##1 $fell(dValid);
```

5.3 Task c - run_checkall

Running `checkall` exposes several bugs. Many of these bugs are exposed by different assertions within the same time frame. The simulation fails at times 240 (`checkdAck`), 250 (`checkValid`), 320 (`checkdAck` and `checkValid`), 350 (`checkdAck` and `checkValid`), 410 (`checkdataValid` twice) and 440 (`checkdataValid`).

6 Lab 6

6.1 Task a - run_check

2. `check1` asserts that `AD` and `C_BE_` is not unknown when `FRAME_` falls. The simulation fails at time 30 when `AD` is high impedance (`'Z'`). Code added:

```
@(posedge clk) disable iff (!reset_) $fell(FRAME_) |->
    !$isunknown(AD) && !$isunknown(C_BE_);
```

3. `check2` asserts that `AD` and `C_BE_` is not unknown when both `IRDY_` and `TRDY_` is asserted (logic low).

```
@(posedge clk) disable iff (!reset_) !IRDY_ && !TRDY_ |->
    !$isunknown(AD) && !$isunknown(C_BE_);
```

4. `check3` asserts that `FRAME_` can only get de-asserted (go high) if `IRDY_` is asserted (goes low). This particular task was difficult because the lab manual says '`FRAME`' (without the underscore), which would suggest the opposite behavior from `FRAME_`. We assumed this was an error in the manual. The simulation fails at time 100 due to `FRAME_` going high while `IRDY_` is high. Code added:

```
@(posedge clk) disable iff (!reset_) $rose(FRAME_) |-> !IRDY_;
```

5. `check4` asserts that `TRDY_` can only be asserted (go low) when `DEVSEL_` is asserted (is low). The simulation fails at times 50, 70, 80 and 90 when this property is not satisfied. Code added:

```
@(posedge clk) disable iff (!reset_) !TRDY_ |-> !DEVSEL_;
```

6. `check5` asserts that during a transaction (while `FRAME_` is asserted) `C_BE_` cannot float. The simulation fails at time 100 when `C_BE_` is high impedance (`'Z'`) while `FRAME_` is asserted. Code added:

```
@(posedge clk) disable iff (!reset_) $fell(FRAME_) |->
!$isunknown(C_BE_) [*0:$] ##0 $rose(FRAME_);
```

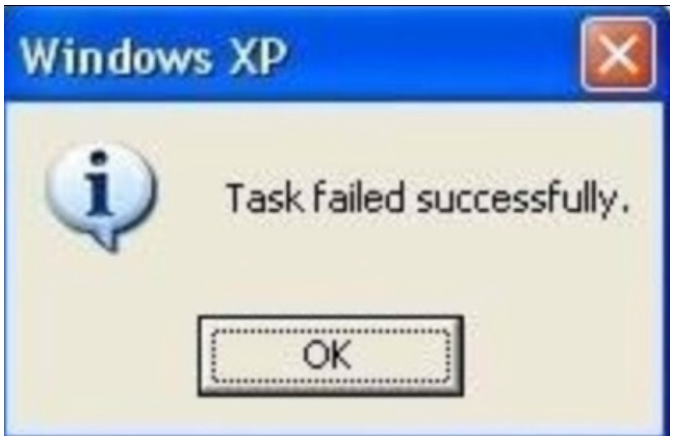

Figure 1: DONE... CONGRATULATIONS