Lab Workbook

SS 2024

Industrial Hardware Verification

Jakob Lechner
Markus Ferringer

Part 3: PSL

Contents

Lab Mode

- You have to elaborate these exercises on your own. This lab part is **not** a team effort.
- We provide stubs containing basic templates for you which are to be extended with your implementations
- Questions are to be ansered in the respective source-files of the corresponding exercise. Just add appropriate comments at the end
- I strongly recommend to read through an entire task before starting with the implementation (including the questions section). You will sometimes find hints later that might simplify things
- Once done with all exercises, please execute `./make.sh clean` in each folder, zip **everything** (including makefiles, scripts, and frameworks) and submit it via TUWEL

Remarks

The syntax presented in the lextures was mostly based on Verilog flavor (because it's shorter and thus better fits on slides). Keep this in mind esp. for locial operators like !, &&, ||: Those have to be replaces by `not, and, or`. Most of the time, the equality operator (or assignment operator) `=` is to be replaced with `is`. I found that esp. LTL style is not supported according to the standard. For example, I could not get the `next` command running in combination with specifying a range, like `next[2 to 3]` or `next[2:3]`.

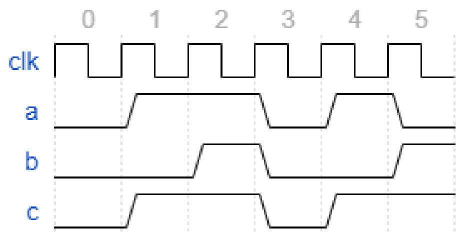
Just try different combinations / styles, and be sure not to leave out any braces. PSL is VERY picky about braces.

Chapter 1

PSL

1.1 Exercise 1: PSL from Waveform

- For the given waveforms, write PSL assertions that check the waveforms as depicted.
- In this exercise, the signals do not depend on each other, so you just have to describe the depicted, static waveform in PSL



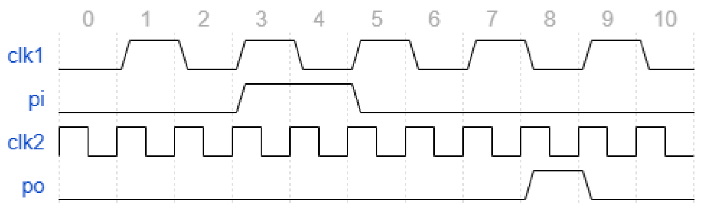
1.1.1 Task 1 [1 Point]

- Describe the given trace using assertions in LTL style

1.1.2 Task 2 [1 Point]

- Describe the given trace using assertions in SERE style

1.2 Exercise 2: Pulse Clock Crosser



The unit under test is a Pulse Clock Crosser. As you can see in the waveform, it crosses a single-cycle pulse from clock domain 1 to a single-cycle pulse in clock domain 2. Those clock domains can be totally unrelated.

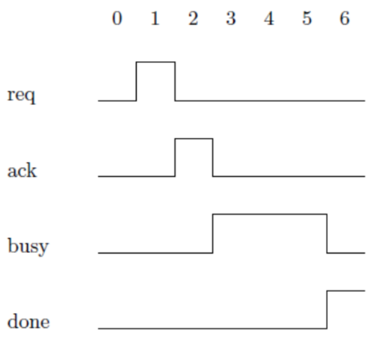
1.2.1 Task 1 [2 Points]

Write PSL expressions, either LTL or SERE style (or both), to cover the following requirements:

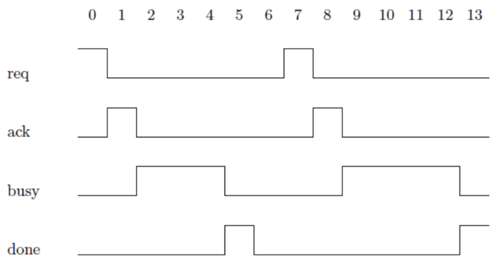
- The input `pi` is high for exactly one cycle at a time. It belongs to clock `clk1`
- The output `po` is high for exactly one cycle at a time. It belongs to clock `clk2`
- After input `pi` has been asserted, it must not be asserted again before the corresponding `po` has been asserted first
- After input `pi` has been asserted, the output `po` must be asserted eventually
- After input `pi` has been asserted (at `clk1`), the output `po` must not be asserted for at least 2 clock cylces (of `clk2`). There are internal clock crossers, and they need some time. Notice: This does not mean that `po` must be asserted in the 3rd cycle - it could take longer!

1.3 Exercise 3: Check Waveform

1.3.1 Task 1 [1 Point]



(a)

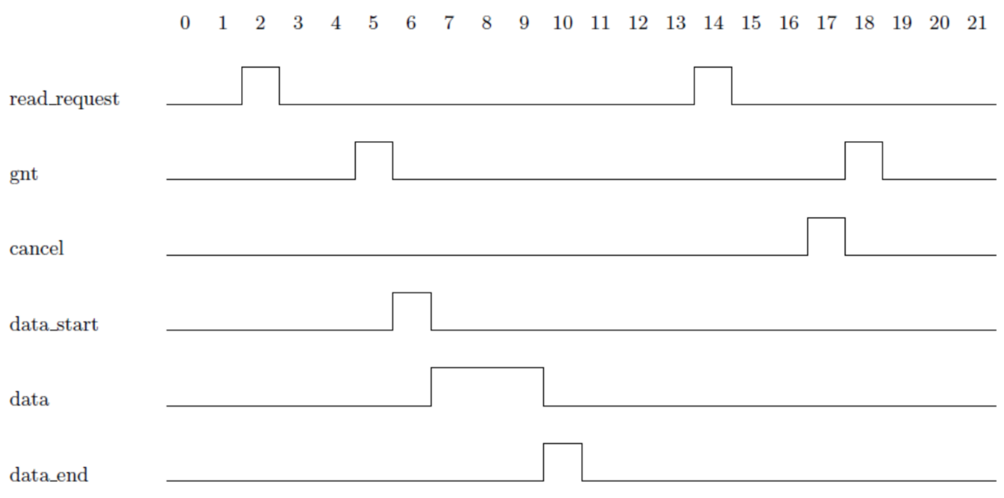


(b)

There are waveforms (a) and (b) and PSL expressions p1, p2, p3. Check if the PSL expressions hold for the given waveforms. If not, state in which cycle they fail, and why.

```
p1: assert always {req} |=> {ack;busy;busy;busy;done};
p2: assert always {req} |=> {ack ; busy[*3] ; done};
p3: assert always {req} |=> {ack ; busy[*3:5] ; done};
```

1.3.2 Task 2 [1 Point]

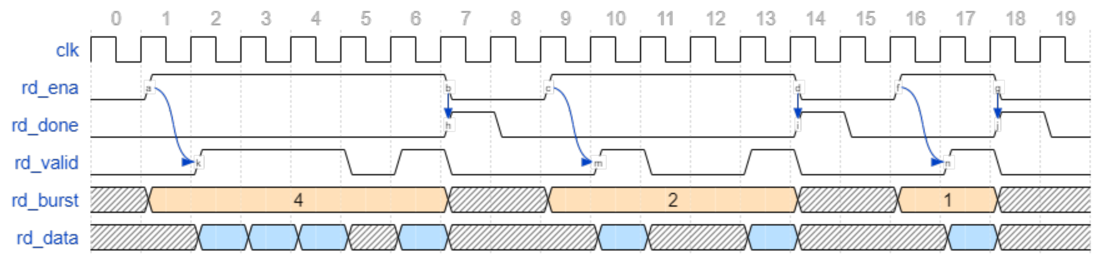


(a)

There is waveform (a) and PSL expressions p1, p2. Check if the PSL expressions hold for the given waveform. If not, state in which cycle they fail, and why.

```
p1: assert always
  {{read_req ; [*0:4] ; gnt} && {cancel[=0]}} |=>
  {data_start ; data[*] ; data_end};
p2: assert always
  {{read_req ; [*0:4] ; gnt} & {cancel[=0]}} |=>
  {data_start ; data[*] ; data_end};
```

1.4 Exercise 4: Burst Reads



The image shows a typical waveform of a burst-read interface:

- With `rd_ena`, also `rd_burst` is asserted.
- `rd_burst` is in range 1-7 and defines the number of data values to be read
- Starting with the next cycle, data is applied by asserting `rd_valid` (and setting `rd_data`)
- After the last transmitted word, `rd_done` is asserted
- `rd_valid` is not necessarily high for the entire burst. Pauses are allowed
- However, the first data value directly after `rd_ena` is asserted, is always valid
- `rd_ena` and `rd_done` are never active at the same time

1.4.1 Task 1 [4 Points]

For the above waveform with the given specification, define the following PSL statements. Please take a note at the hints for this exercise.

- Check that whenever `rd_ena` becomes 1 (i.e., changes from 0 to 1), the next cycle always contains valid data (with `rd_ena` still asserted and `rd_done` being deasserted).
- Check that when `rd_ena` becomes 1, it is followed by 1 to 7 assertions of `rd_valid` (with `rd_done`=0), followed by a single assertion of `rd_done`. Notice that `rd_valid` can have interruptions.
- Check that `rd_ena` stays asserted until `rd_done` is asserted
- Check that when `rd_done` is asserted, the correct number of `rd_valid` cycles (according to `rd_burst`) have occured

Hints: There are probably various ways how the above requirements can be checked. I will provide hints for one possible solution:

- In the PSL file, you can define VHDL signals. Define `burst` of type `integer`. You can just write "inline VHDL" like every other PSL expression.
- Make a concurrent (VHDL-)assignment to `burst` such that it assigns the current value of `rd_burst` at the rising edge of `rd_ena`, and that it decrements itself when both `rd_ena` and `rd_valid` are asserted (otherwise, leave unaffected).
- You can then use the signal `burst` inside a PSL expression and check if it is indeed 0 when `rd_done` is asserted
- You can use `rising_edge()` inside PSL expressions