Hardware Verification Assignment 2 – Network Verification with Jasper Gold

Dragoş Perju, Mart P.

Exercise 1 (newgen?0 => switch o0, join, torsp blocked by blocked q0 or newgen?1 at SRC$reqrsp)

To prove the persistency of channels, we want ***irdy***'s from the outputs of all modules to hold high, from the moment when they become high until they overlap once with high ***trdy***'s received by the same outputs. There is no reason to check the inputs of all modules as well, since we would be testing the same wires. Page 22 of the slides has shown a useful expression paired with a representative graph, ensuring the solution for this exercise: `irdy |-> irdy until_with trdy;`.

However, an alternative expression that is believed to be equivalent has been given for a detailed understanding: `irdy |-> irdy [*1:$] ##[0:1] (irdy & trdy);`. We imply that ***irdy*** has to be high once before the assertion is even to be tested (`irdy |->`) and we want it to hold its high (`|-> irdy`) at least one clock (`[*1:$]`) until, on a following clock (`##1`), ***trdy*** becomes high as well while ***irdy*** is high. There are instances where ***irdy*** and ***trdy*** can be immediately high the first time the *top* module is ever "powered on". Hence, we include 0 in `##[0:1]` to allow the situation where ***irdy*** and ***trdy*** are both high at clock time 0 and no ***irdy*** alone should be high before.

Exercise 2 (newgen?0 or newgen?1 at SRC$reqrsp)

In order to prove that we have a transfer over a fork, we are interested in when one of the output channels of the fork is available, the other has to be available too. Furthermore we also want to know that once the input is done, either one of the outputs will make something available on the channel; we then use the previously defined property to ensure that both outputs are available. Therefore, we check `irdy_0 |-> irdy_1 && irdy_1 |-> irdy_0` which means that we could start a transfer from both channels at the same time. For the second part – where we check that an input will be "forked" – we use the same procedure, but instead of 2 inputs into a channel, we take a single input and a single output. Since we do not need any guarantees about when the transfers happen or the duration we can leave it with this check.

Exercise 3 (newgen?0 => join blocked or newgen?1 at SRC$reqrsp)

We now need to prove the exact reverse of what we attempted to prove at Exercise 2, instead of both outputs to be active simultaneously, we want both inputs to be active simultaneously, and for each input we need the output to be active. The same procedure is used for checking, but now both inputs have to be active, and when one input is active the output has to be active as well.

Exercise 4 (newgen?1 at SRC$reqrsp only)

Fairness is when a value for a variable is assigned to it *infinitely often* while time passes. The key for fairness however is that every possible value of the variable should be tested, especially when there's more than one possible value (values represented on 2 or more bits = 4 or more possibilities). Therefore, in this exercise both the high and low values of the ***sel*** variable of the arbiter module from each of the merge modules are tested for fairness (4 fairness tests). In this way we adopt the formula of *GFp* from LTL for fairness, for every possible *p*.

The tight lower bound for all cases is 1 clock, except for when ***sel*** is to be equal to 1 in merge2, where in this case the tight lower bound is 2. This is because sel=1 is not the default initialization and changing to sel=1 in this situation proves to have at least one example case that is slower than usual (at least: since we're testing with cover). The tight upper bound is disputable, as sel=0 is the default initialization (at reset) for both merges, therefore should have 0 as its tight upper bound since the selection is instant for input 0, but for choosing to transmit based on ***sel***, the ***irdy*** inputs are regarded as well and latency takes place. The tight upper bound for sel=1 should also be 1. Both would be tested with an `initial assert` structure, but it is not a property assertion and not supported by Cadence.

Exercise 5 (newgen?0 => q0 blocked or newgen?1 at SRC$reqrsp)

In order to show that each channel is alive, we need to ensure that after every active ***irdy*** we – eventually – get an active ***trdy***. So for every ***irdy-trdy*** pair from every source and queue in the network, we check whether when ***irdy*** becomes active, that eventually ***trdy*** will become active too; and this should happen globally/always in the module's lifetime. Ore more simply put: `always(irdy |-> s_eventually trdy)`. We see q0 being blocked in this case.

Exercise 6 (newgen?0 at SRC$reqrsp only)

For checking latency from the request source to the sink, we have found that monitoring the changing of datatype from both modules to be a good guidance in measuring latency. Therefore, similar to exercise 4, we search the "pattern" of ***data$type*** from *req* source to be 0, on the same clock cycle the ***data$type*** from *sink* to be different from 0 and in the following clock cycle the ***data$type*** from *sink* to be 0 mirroring the data sent from *req* source.

This assertion passes, therefore we can say that the latency is of 1 clock from start to finish. The assertions that follow, that check for 2 clocks or 3 clocks or more, do not pass – enforcing the fact that the latency for the whole design is 1 clock. In this exercise, we do not check all values of ***data$type*** since we're not checking fairness or have the need of checking all possible values.