Hardware Verification Assignment 2 – Network Verification with Cadence IFV

j.schmaltz@tue.nl
www.win.tue.nl/∼jschmalt/

This second assignment is about verifying hardware designs of small communication fabrics by writing System Verilog Assertions and proving them using Cadence Incisive Formal Verifier.

1 Getting started

In your home directory on the server, you will find a file named `allprim.tar.gz` containing the design and a first version of the IFV verification script. Simply type `tar -xvf allprim.tar.tgz` to untar the archive. You should then be ready to start!

The deadline for solutions is Monday October 3rd 2016 at noon. Please email a .tar.gz archive of your top directory (allprim/) together with a short report (plain text) with what you did.

2 Getting around with IFV

How to login: ssh -Y yourloginname@cadence01.win.tue.nl (The -Y is to export display).

Every time you log in you need to "source" the script to load the Cadence environment variables and get a license. At your home directory, type:

```
. .cadence_project.sh
```

To check that you have the tool running just type:

```
which ifv
```

and you should get the place where ifv is installed.

The installation direction is the following

```
/eda/cadence/2014-15/RHELx86/INCISIVE_14.10.004/kits/VerificationKit/
```

The user manual for IFV is at

```
/eda/cadence/2014-15/RHELx86/INCISIVE_14.10.004/kits/VerificationKit/doc/ifvref/
```

For writing assertions, documentation can be found at:

```
/eda/cadence/2014-15/RHELx86/INCISIVE_14.10.004/kits/VerificationKit/doc/abvwrite/
```

3 A simple fabric

Figure 1 shows an abstract view of the first design, called `allprim.v`. It is a small fabric where two sources – named `reqrsp` and `req` – inject messages that are consumed by a sink, named `snk` on the right. There are two kinds of messages: requests and responses. The top source – `reqrsp` – injects both requests and responses. The lower source – `req` – only injects requests. The top source is connected to a fork that duplicates a message and send the two copies to the two queues, named

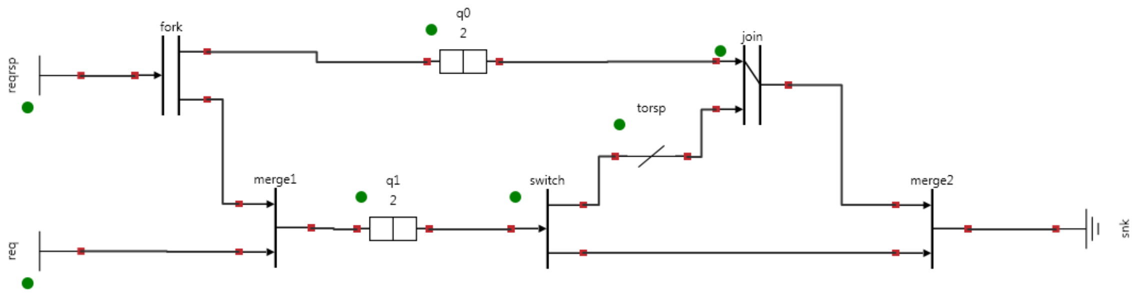

Figure 1: A simple communication fabric.

q_0 and q_1. The switch then forwards responses to its top output and requests to its lower output. Messages at the top output of the switch are transformed to responses by a function named `torsp`. The output of this function is joined with whatever messages are in the top queue, q_0. Finally, all messages are consumed at the sink.

In this picture, the lines actually are channels composed of three signals.

1. A data line carrying the actual data exchanged between the components. In this example, data are the type of the messages, that is, a 0 for a request and a 1 for a response.

2. A `trdy` signal that is high when the target – that is, the end point of a channel – is ready to receive a message.

3. A `irdy` signal that is high when the initiator – the start point of channel – is ready to send a message.

In the next sections, you are going to write several assertions to verify this design. This design actually has errors and in particular it is not "live" as messages can get stuck forever in queues. Your objective is to write properties to thoroughly inspect and analyse the design. You should leave the design untouched.

Note: to search for modules in `allprim.v`, here is the naming convention used to name modules

- All source modules have a name starting with `SRC`.
- All sink modules have a name starting with `SNK`.
- All queue modules have a name starting with `Q`.

- All merge modules have a name starting with `MRG`.
- All switch modules have a name starting with `SW`.
- All function modules have a name starting with `FUN`.
- All join modules have a name starting with `CJN`.
- All fork modules have a name starting with `FRK`.

3.1 Persistency

Persistency is a channel property stating that once the `irdy` signal is high, it remains high until a `trdy` is high.

Exercise 1. Write assertions checking that all channels are persistent. Prove with IFV that these assertions hold. Note: if every in and out interfaces of each component are persistent, then all channels are persistent.

3.2 Fork and Join

A fork duplicates its input to its two outputs. It does this only when the two outputs are ready to receive a message. A join is the dual of a fork. It "merges" its two inputs to its output. It does this only when the two inputs have a message.

Exercise 2. For a fork, write assertions stating that there is a transfer on one of the output channels if and only if there is a transfer on the other output channel. Write more assertions stating that there is a transfer on one of the outputs if and only if there is a transfer on the input channel.

Exercise 3. For a join, write assertions stating that if there is a transfer on one of the input channels, then there is transfer on the other input channel at the same cycle and there is a transfer at the output channel. Write similar assertions for the other input. Add assertions checking that transfers occur at the input if and only if there is a transfer at the output.

3.3 Fairness of arbitration points

The design has two arbiters, called "merges" and named `merge1` and `merge1`. An important property for arbiters is fairness, that is, arbiters must give the turn to all its inputs. More formally, all requests must be eventually granted.

Exercise 4. Write assertions expressing fairness of the two merges and prove them using IFV. What is the tightest bound on the number of cycles that an input needs to wait to get the turn?

Note: You might see that the trigger of some assertions always fail. The issue is that the Verilog implementation of the top source generates only requests. Search for the line

```
assign o0$data$type = newgen?0 :  prevo0$data$type;
```

of the source module. You can write `newgen ?  1` instead to have the source generate response only. The trigger for the merge fairness assertion should now pass. If the top level source never sends a response, the top input of `merge2` is never active.

3.4 Liveness

Liveness is crucial to communication fabrics. Intuitively, liveness is achieved if messages never get stuck on their way, that is, it is never the case that a message stays in a queue forever. In this design, a message can get stuck at the head of each queue or at a source, that is, a source might try to send a message but will never be able to do so. Note that because a source is persistent, it will never de-assert an `irdy` signal before a high `trdy` signal is received.

Exercise 5. Write assertions expressing the fact that each source is "alive". This means that every time a source is ready to send a message, it will eventually be able to do so. Write similar assertions for the two queues.

4 Latency

Latency properties are also crucial for networks. In general, they are very hard to prove. For the simple design of this assignment, it might still be possible to prove some latency properties. For packets injected at the top level source, the deadlock prevents packets to reach their destination. So, here latency is infinite. However, the lower source does not suffer from this issue.

Exercise 6. Write an assertion stating the number of cycles needed by a request injected in the lower source to reach the sink.