TFE4171 Exercise 4 Report

Morten Paulsen
Magnus Ramsfjell

March 29, 2020

1 Lab 1 - Formal verification of readserial

1.1 State Transition Graph

The **readserial** controller has two states: **IDLE** and **READDATA**. It transitions according to the graph in Figure 1.

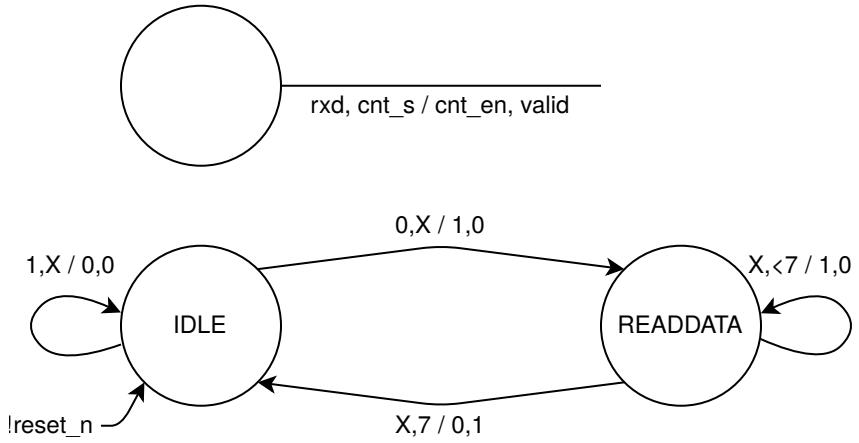

Figure 1: State transition graph for the **readserial** controller

1.2 reset_sequence

For the `reset` property we need to assert that all the control signals are in the correct state, so the signals we have to check are `state_s`, `valid`, `cnt_en`, and `cnt_s`.

```
24  sequence reset_sequence;
25          !reset_n;
26  endsequence;
...
41  property reset;
42      reset_sequence |=>
43
44      t ##0 (state_s == IDLE and !valid and !cnt_en and cnt_s == 0);
45  endproperty;
...
116 a_reset: assert property (@(posedge clk) reset);
```

Listing 1: Implementation of the **reset** property

1.3 stay_in_idle

For the `stay_in_idle` property we need to check that at the beginning we are in idle by checking the status signal `state_s` and that we receive `rxd = 1'b1` to stay in idle. After that we need to assert that we still are in idle by checking that the status signal is the same and that the `valid` signal is deasserted.

```
 28  sequence idle_sequence;
 29      state_s == IDLE;
 30  endsequence;
...
 47  property stay_in_idle;
 48      t ##0 (idle_sequence and rxd)
 49
 50      implies
 51
 52      t ##1 (idle_sequence and !valid);
 53  endproperty;
...
116  a_stay_in_idle: assert property (@(posedge clk) stay_in_idle);
```

Listing 2: Implementation of the **stay_in_idle** property

1.4 read_byte

For the `read_byte` property we need to check the complete sequence from initiation to end. This is done by checking that we start in `state_s = IDLE` and that we receive `rxd = 1'b0`. After this we store the values of `rxd` throughout the reading (8 cycles). In the implication part we check that the `valid` flag is deasserted and that `state_s = READDATA` during the reading. In the last cycle we check that we have returned to `state_s = IDLE` and that the read data is correct (`data == tmp`) and lastly we check that `valid` is asserted.

```
 28  sequence idle_sequence;
 29          state_s == IDLE;
 30  endsequence;
...
 55  property read_byte;
 56          logic [7:0] tmp;
 57
 58          t ##0 (idle_sequence and !rxd) and
 59          t ##8 set_freeze(tmp,
 60                  {
 61                          $past(rxd, 7),
 62                          $past(rxd, 6),
 63                          $past(rxd, 5),
 64                          $past(rxd, 4),
 65                          $past(rxd, 3),
 66                          $past(rxd, 2),
 67                          $past(rxd, 1),
 68                           rxd
 69                  })
 70
 71          implies
 72
 73          t ##1 (state_s == READDATA and !valid) [*8] and
 74          t ##9 (idle_sequence and valid and data == tmp);
 75  endproperty;
...
117  a_read_byte: assert property (@(posedge clk) disable iff (!reset_n) read_byte);
```

Listing 3: Implementation of the **read_byte** property

1.5 read_byte, basic IPC solver

By using the basic IPC solver we need to include some additional sequences - `in_idle_counter_is_0` and `in_idle_counter_not_` - this is to assert that the internal counter signals are correct. This is done by asserting that in `state_s = IDLE` we have `cnt_en = 1'b0` and `cnt_s = 3'b0`.

```
28  sequence idle_sequence;
29          state_s == IDLE;
30  endsequence;
31
32  sequence in_idle_counter_is_0;
33          state_s == IDLE and cnt_s == 0;
34  endsequence;
35
36  sequence in_idle_counter_not_enabled;
37          state_s == IDLE and !cnt_en;
38  endsequence;
...
55  property read_byte;
56          logic [7:0] tmp;
57
58          t ##0 (idle_sequence and !rxd) and
59          t ##0 in_idle_counter_is_0 and
60          t ##0 in_idle_counter_not_enabled and
61          t ##8 set_freeze(tmp,
62                  {
63                          $past(rxd, 7),
64                          $past(rxd, 6),
65                          $past(rxd, 5),
66                          $past(rxd, 4),
67                          $past(rxd, 3),
68                          $past(rxd, 2),
69                          $past(rxd, 1),
70                           rxd
71                  })
72
73          implies
74
75          t ##1 (state_s == READDATA and !valid) [*8] and
76          t ##9 (idle_sequence and valid and data == tmp);
77  endproperty;
...
117 a_read_byte: assert property (@(posedge clk) disable iff (!reset_n) read_byte);
```

Listing 4: Implementation of the updated **read_byte** property

The sequence `in_idle_counter_is_0` is added to prevent OneSpin from assuming that `cnt_s` can be any value, and not only 0. The counterexample is shown in Fig. 2

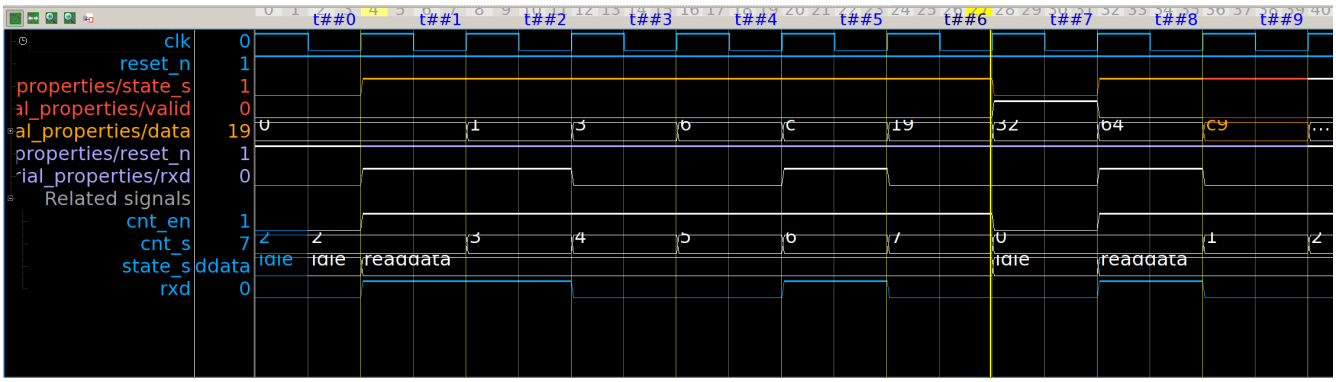

Figure 2: Counterexample before adding in_idle_counter_is_0.

The sequence `in_idle_counter_not_enabled` is added to prevent OneSpin from assuming that `cnt_en` can be

enabled when `state_s` is `IDLE`. The counterexample is shown in Fig. 3.

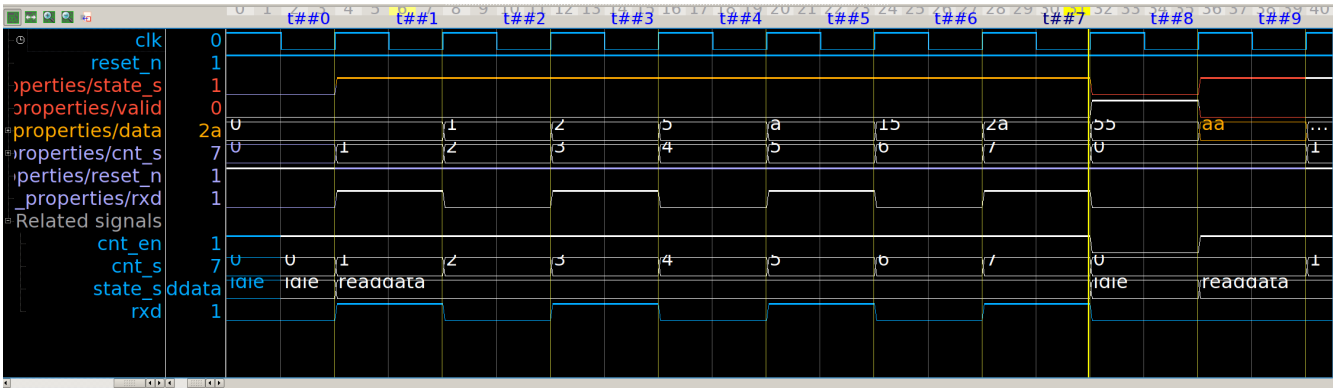

Figure 3: Counterexample before adding in_idle_counter_not_enabled.

1.6 read_byte, inductive proof for in_idle_counter_not_enabled

To be able to use the sequence `in_idle_counter_not_enabled` we have to prove that it actually holds for all time steps. This can be done by using an inductive proof, show that it holds after a reset and that it holds in any arbitrary time step too. These two properties are shown in `in_idle_counter_not_enabled__step` and `in_idle_counter_not_enabled__base`.

1.6.1 in_idle_counter_not_enabled__step

```
79  property in_idle_counter_not_enabled__step;
80          t ##0 (in_idle_counter_not_enabled and rxd)
81
82          implies
83
84          t ##1 in_idle_counter_not_enabled;
85  endproperty;
...
118 a_in_idle_counter_not_enabled__step: assert property (@(posedge clk) disable iff (!reset_n) in_idle_counter_not_enabled__step);
```

Listing 5: Step proof for in_idle_counter_not_enabled.

1.6.2 in_idle_counter_not_enabled__base

```
87  property in_idle_counter_not_enabled__base;
88          t ##0 reset_sequence
89
90          implies
91
92          t ##1 in_idle_counter_not_enabled;
93  endproperty;
...
119 a_in_idle_counter_not_enabled__base: assert property (@(posedge clk) in_idle_counter_not_enabled__base);
```

Listing 6: Base proof for in_idle_counter_not_enabled.

1.7 read_byte, inductive proof for in_idle_counter_is_0

We also have to do the same for `in_idle_counter_is_not_0` as we did for `in_idle_counter_not_enabled`. Our first attempt is shown below in section 1.7.1. Here we only use a single time frame to try to prove the validity of the sequence, but it fails as described in the section. Our second attempt uses multiple time frames (2) and it succeeds, as explained in section 1.7.2.

1.7.1 in_idle_counter_is_0, first attempt

```
95  property in_idle_counter_is_0__step;
96          t ##0 (in_idle_counter_is_0 and rxd)
97
98          implies
99
100         t ##1 in_idle_counter_is_0;
101 endproperty;
...
120 a_in_idle_counter_is_0__step: assert property (@(posedge clk) disable iff (!reset_n) in_idle_counter_is_0__step);
```

Listing 7: Step proof for in_idle_counter_is_0.

```
104 property in_idle_counter_is_0__base;
105         t ##0 reset_sequence
106
107         implies
108
109         t ##1 in_idle_counter_is_0;
110 endproperty;
...
121 a_in_idle_counter_is_0__base: assert property (@(posedge clk) in_idle_counter_is_0__base);
```

Listing 8: Base proof for in_idle_counter_is_0.

The counterexample given by OneSpin when we used the single-step inductive proof is shown in Fig. 4.

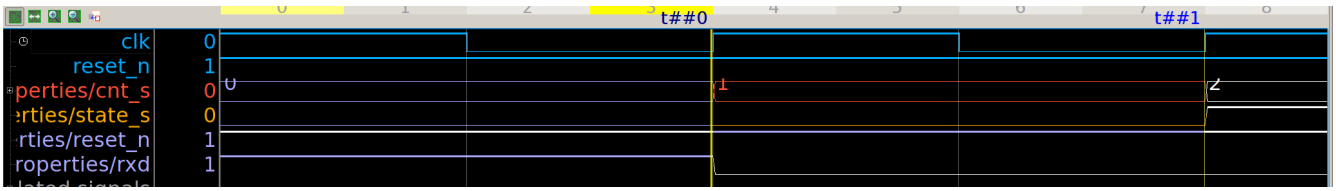

Figure 4: Counterexample for the inductive proof for in_idle_counter_is_0__step.

The counterexample shows an example where `cnt_s` is incremented when `state_s == IDLE`. This happens since the assumption is only based on one clock cycle and there is not made any assumption that `cnt_s` is to be stable between cycles. This can be prevented by introducing an extra step in the assumption, if we assume that the sequence holds in two cycles (`##0` and `##1`) it must also hold in cycle `##2`, as the only way to find a counterexample to the implication is to break the assumption.

For this counterexample specifically, the `in_idle_counter_is_0` sequence holds for the first cycle (`t ##0`), but fails at the next cycle because `cnt_s` has increased. This counterexample is found by having the internal signal `cnt_en` set high—which has not been proved to be unreachable in this property—which started the counter. The counter then increments to 1 at `t ##1`, causing the property to fail. Solving this requires either appending `and !cnt_en` to the `in_idle_counter_is_0` sequence (which somewhat defeats the purpose of the `in_idle_counter_not_enabled`), adding the
`in_idle_counter_not_enabled` sequence to the `in_idle_counter_is_0__step` property assumption, or by assuming that the `in_idle_counter_is_0` sequence holds for two consecutive cycles. The former could be regarded

as the "proper" solution, as it implicitly captures the fact that the count signal is not enabled (which it never should have been in this case) by using two cycles where `cnt_s` did not change.

1.7.2 in_idle_counter_is_0, second attempt

```
 95  property in_idle_counter_is_0__step;
 96          t ##0 (in_idle_counter_is_0 and rxd) and
 97          t ##1 (in_idle_counter_is_0 and rxd)
 98
 99          implies
100
101          t ##2 in_idle_counter_is_0;
102  endproperty;
...
120  a_in_idle_counter_is_0__step: assert property (@(posedge clk) disable iff (!reset_n) in_idle_counter_is_0__step);
```

Listing 9: Step proof for in_idle_counter_is_0.

```
104  property in_idle_counter_is_0__base;
105          t ##0 reset_sequence and
106          t ##1 rxd
107
108          implies
109
110          t ##1 in_idle_counter_is_0 and
111          t ##2 in_idle_counter_is_0;
112  endproperty;
...
121  a_in_idle_counter_is_0__base: assert property (@(posedge clk) in_idle_counter_is_0__base);
```

Listing 10: Base proof for in_idle_counter_is_0.

To prove the properties we need to have `n=2` as this ensures that `cnt_s` is kept at 0. `cnt_s` has no change between `t ##0` and `t ##1` nor `t ##1` and `t ##2`. This was not taken consideration in the first attempt.

2 Lab 2 - Completeness checking of readserial

In Lab 2 we are to use a completeness checker to assert that we actually have included a property for every possible operation in the module. The completeness checker does not use the RTL code at all, but only uses the properties to conclude.

```
// @lang=vli @ts=2

completeness readserial;
disable iff: (!reset_n);
inputs: reset_n, rxd;

determination_requirements:
  determined(valid);
  if (valid) determined(data); endif;

reset_property:
  sva/inst_readserial_properties/ops/a_reset;
property_graph:
  sva/inst_readserial_properties/ops/a_reset -> sva/inst_readserial_properties/ops/a_read_byte;
  sva/inst_readserial_properties/ops/a_reset -> sva/inst_readserial_properties/ops/a_stay_in_idle;
  sva/inst_readserial_properties/ops/a_stay_in_idle -> sva/inst_readserial_properties/ops/a_read_byte;
  sva/inst_readserial_properties/ops/a_stay_in_idle -> sva/inst_readserial_properties/ops/a_stay_in_idle;
  sva/inst_readserial_properties/ops/a_read_byte -> sva/inst_readserial_properties/ops/a_read_byte;
  sva/inst_readserial_properties/ops/a_read_byte -> sva/inst_readserial_properties/ops/a_stay_in_idle;
end completeness;
```

Listing 11: The modified completeness.gfv

In Listing 11 we added the following pairs of operation property and direct successor operation property:

```
sva/inst_readserial_properties/ops/a_reset -> sva/inst_readserial_properties/ops/a_stay_in_idle;
sva/inst_readserial_properties/ops/a_stay_in_idle -> sva/inst_readserial_properties/ops/a_read_byte;
sva/inst_readserial_properties/ops/a_stay_in_idle -> sva/inst_readserial_properties/ops/a_stay_in_idle;
sva/inst_readserial_properties/ops/a_read_byte -> sva/inst_readserial_properties/ops/a_stay_in_idle;
```

These pairs makes the set of possible property transitions complete.

We also had to make some modifications to the properties themselves for the end point of a operation property to align with the start point of the direct successor operation property. The modifications was to make sure that the signals `state_s`, `cnt_en` and `cnt_s` was explicitly set both at the beginning and at the end of the properties `reset`, `stay_in_idle` and `read_byte`. This was achieved by simply modifying `idle_sequence` to include `cnt_en` and `cnt_s`. The modification is shown in Listing 12.

```
sequence idle_sequence;
        state_s == IDLE and !cnt_en and cnt_s == 0;
endsequence;
```

Listing 12: The modifications done for the completeness checker.