

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.

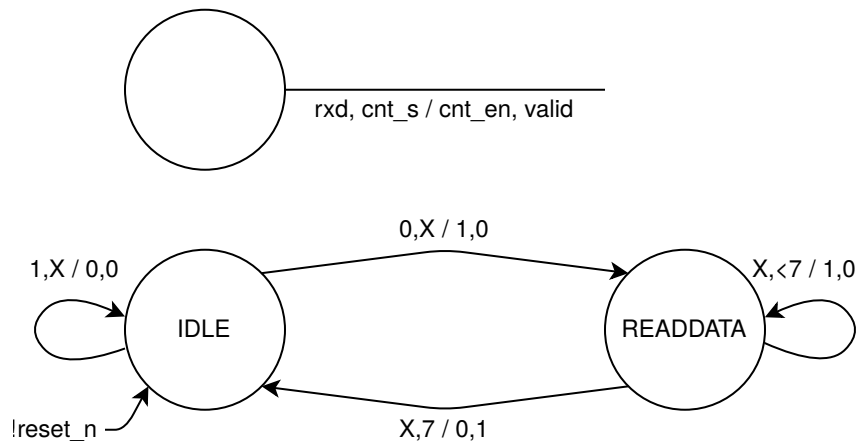


Figure 1: State transition graph for the **readserial** controller

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**.

```
23 sequence reset_sequence;
24     !reset_n;
25 endsequence;
26
27 property reset;
28     reset_sequence ==>
29
30     t ##0 (state_s == IDLE and !valid and !cnt_en and cnt_s == 0);
31 endproperty;
32
33 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 `rx_d = 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.

```
36 sequence idle_sequence;
37     state_s == IDLE;
38 endsequence;
39
40 property stay_in_idle;
41     t ##0 (idle_sequence and rx_d)
42
43     implies
44
45     t ##1 (idle_sequence and !valid);
46 endproperty;
47
48 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 `rx_d = 1'b0`. After this we store the values of `rx_d` 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.

```

48 sequence idle_sequence;
49     state_s == IDLE;
50 endsequence;
51
52 property read_byte;
53     logic [7:0] tmp;
54
55     t ##0 (idle_sequence and !rx) and
56     t ##8 set_freeze(tmp,
57         {
58             $past(rx, 7),
59             $past(rx, 6),
60             $past(rx, 5),
61             $past(rx, 4),
62             $past(rx, 3),
63             $past(rx, 2),
64             $past(rx, 1),
65             rx
66         })
67
68     implies
69
70     t ##1 (state_s == READDATA and !valid) [*8] and
71     t ##9 (idle_sequence and valid and data == tmp);
72 endproperty;
73
74 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_0` - 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`.

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

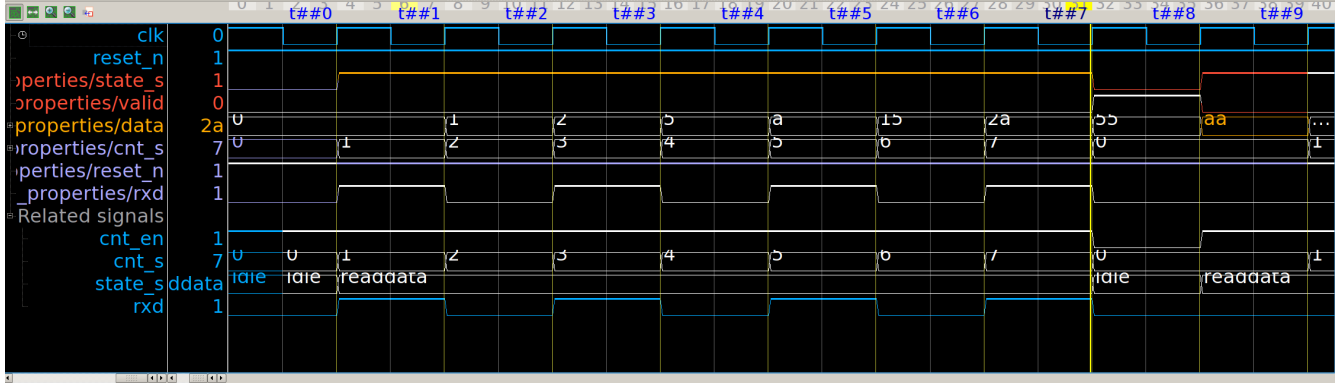


Figure 3: Counterexample before adding `in_idle_counter_not_enabled`.

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`

```

93 property in_idle_counter_not_enabled__step;
94     t ##0 (in_idle_counter_not_enabled and rxd)
95
96     implies
97
98     t ##1 in_idle_counter_not_enabled;
99 endproperty;
100
101 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`

```

101 property in_idle_counter_not_enabled__base;
102     t ##0 reset_sequence
103
104     implies
105
106     t ##1 in_idle_counter_not_enabled;
107 endproperty;
108
109 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

```

109 property in_idle_counter_is_0__step;
110     t ##0 (in_idle_counter_is_0 and rxd)
111
112     implies
113
114     t ##1 in_idle_counter_is_0;
115 endproperty;
116
117 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`.

```

117 property in_idle_counter_is_0__base;
118     t ##0 reset_sequence
119
120     implies
121
122     t ##1 in_idle_counter_is_0;
123 endproperty;
124
125 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.

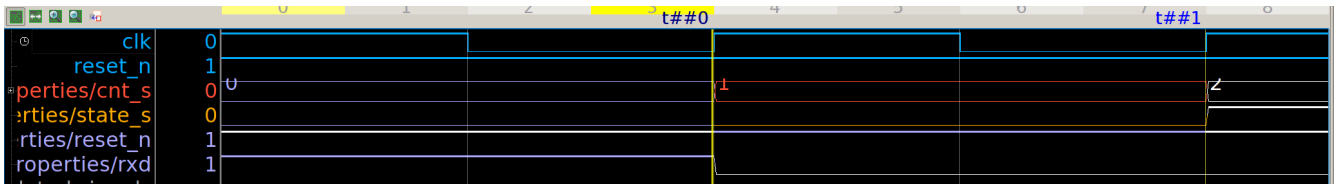


Figure 4: Counterexample for the inductive proof for `in_idle_counter_is_0__step`.

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

```

109 property in_idle_counter_is_0_step;
110     t ##0 (in_idle_counter_is_0 and rxd) and
111     t ##1 (in_idle_counter_is_0 and rxd)
112
113     implies
114
115     t ##2 in_idle_counter_is_0;
116 endproperty;
117
118 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`.

```

118 property in_idle_counter_is_0_base;
119     t ##0 reset_sequence and
120     t ##1 rxd
121
122     implies
123
124     t ##1 in_idle_counter_is_0 and
125     t ##2 in_idle_counter_is_0;
126 endproperty;
127
128 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.

```

1  // @lang=vli @ts=2
2
3  completeness readserial;
4  disable iff: (!reset_n);
5  inputs: reset_n, rxd;
6
7  determination_requirements:
8      determined(valid);
9      if (valid) determined(data); endif;
10
11 reset_property:
12     sva/inst_readserial_properties/ops/a_reset;
13 property_graph:
14     sva/inst_readserial_properties/ops/a_reset -> sva/inst_readserial_properties/ops/a_read_byte;
15     sva/inst_readserial_properties/ops/a_reset -> sva/inst_readserial_properties/ops/a_stay_in_idle;
16     sva/inst_readserial_properties/ops/a_stay_in_idle -> sva/inst_readserial_properties/ops/a_read_byte;
17     sva/inst_readserial_properties/ops/a_stay_in_idle -> sva/inst_readserial_properties/ops/a_stay_in_idle;
18     sva/inst_readserial_properties/ops/a_read_byte -> sva/inst_readserial_properties/ops/a_read_byte;
19     sva/inst_readserial_properties/ops/a_read_byte -> sva/inst_readserial_properties/ops/a_stay_in_idle;
20 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.

```

1 sequence idle_sequence;
2     state_s == IDLE and !cnt_en and cnt_s == 0;
3 endsequence;

```

Listing 12: The modifications done for the completeness checker.