



Delivery time: Friday, March 22, 16:00h.

Lab 1

[Working directory: [readserial](#)]

In this assignment we study *reachability constraints* and *invariants* for removing false counterexamples in Interval Property Checking.

Design

The design **readserial** is a serial receiver. It scans an input line (“serial bus”) named **rx****d** for serial transmissions of data bytes. A transmission begins with a start bit ‘0’ followed by 8 data bits. The most significant bit (MSB) is transmitted first. There is no parity bit and no stop bit. After the last data bit has been transferred a new transmission (beginning with a start bit, ‘0’) may immediately follow. If there is no new transmission the bus line goes high (‘1’, this is considered the “idle” bus signal). In this case the receiver waits until the next transmission begins.

The outputs of the design are an 8-bit parallel **data** signal and a **valid** signal. The **valid** signal goes high (‘1’) for one clock cycle after the last serial bit has been transmitted, indicating that a new data byte is ready.

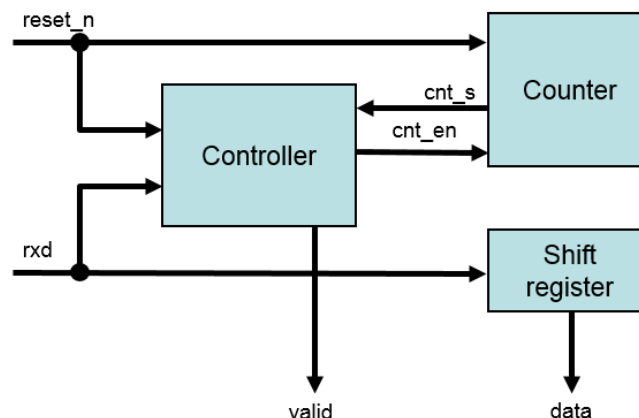


Figure 1: Structure of the serial receiver design

Fig. 1 shows the structure of the serial receiver. It includes a bit counter, a shift register and a main controller. Whenever the controller detects a start bit it enables the counter. After 8 data bits have been received the data output is valid.

Note: For the following tasks you need to use the **TiDAL** library.

Task 1.1

Analyze the design and draw a state transition graph for the main controller.

Task 1.2

Write a property called **reset** proving that right after applying the reset sequence the controller is in state **IDLE** and the counter is reset to 0. Verify the correct output behavior. The reset sequence can be formulated by defining an SVA sequence named **reset_sequence**:

```
sequence reset_sequence;  
    reset_n == 1'b0;  
endsequence;
```

Task 1.3

Write a property called **stay_in_idle** proving that the controller stays in state **IDLE** if no start bit arrives. Verify the correct output behavior for this case.

Task 1.4

Now we prove the more interesting case that a byte is transmitted. Write a property called **read_byte** that assumes that we are in state **IDLE** and that a start bit is detected. Besides checking the values of the outputs of the receiver, the property should also state that after the 8 data bits, the main controller returns to state **IDLE**.

Prove the property.

Task 1.5

The tool builds on a set of different solvers (including BDD-based methods) to minimize the probability of false counterexamples. In a new experiment, we now restrict the tool to use only the basic IPC solver as it is discussed in chapter “SAT-based property checking” in the lecture. In order to force the tool to use only this solver you need to run the script **pure_ipc.tcl** from the command line in the console window:

```
mv> source pure_ipc.tcl
```

The effects of the script can be undone by typing

```
mv> set_check_option -default
```


The following tasks should be carried out based on the restricted solver settings.

Note that the tool does not actually run a solver again if the validity of a property has already been proven. You can override this behavior by calling the **check** command in the shell window with the option **-force**, for example, like this:

```
mv> check -force a_read_byte
```



Alternatively, you can use the GUI's menu item "Check Selection..." which allows you to enable the "Force check" option.

Check the property **read_byte** again with the new settings. It should fail now.  Open the counterexample in the debugger. Is this a realistic counterexample? Why not?



1. If you look at the design you will find that the counter is 0 whenever the main controller is in state IDLE. (Convince yourself of this.) Write a *sequence* called **in_idle_counter_is_0** expressing this relationship.

Notice the new sequence corresponds to a so-called invariant (see lecture slides, chapter "SAT-based property checking") which describes a reachable set of states of the design (that is also closed under reachability). Which states have to be excluded from the invariant? Which have to be included?



Add the sequence **in_idle_counter_is_0** to the cause/assumption part in your property. Can you prove the property now?



2. Probably you cannot. Examine the counterexample: Is the counter enabled in state IDLE? Check with the design. Can this ever be the case? (No, it cannot. Convince yourself of this.) Write a sequence called **in_idle_counter_not_enabled** and add it to the cause/assumption part in your property. Can you prove the property now using the command **check_property**? (Now it should be possible!)



3. Extend the property to also verify the correctness of the output behavior.

Note:

After adding two constraints, expressed as sequences, we were able to prove the property. We obtained the constraints by analyzing the design and convincing ourselves that the constraints are valid if the design has been properly initialized at some time in the past. The constraints seem to be valid in the reachable state space of the design. We call them *reachability constraints*.

Task 1.6

We need to formally prove the validity of the reachability constraints; otherwise we would have made an invalid assumption in our property **read_byte** of the previous task. A common way of doing this is by proving that the constraint is an *invariant*, (i.e., it is closed under reachability), and that it contains the initial state. An inductive proof can be used for this purpose.

We begin with the reachability constraint **in_idle_counter_not_enabled**.

1. Write a property for the *induction step*, called **in_idle_counter_not_enabled__step**. The property shows that if the reachability constraint holds at an arbitrary time point **t**, it will also hold at **t+1**. Prove the property.

2. Write a property for the *induction base*, called `in_idle_counter_not_enabled__base`. The property shows that after the reset sequence, (i.e., in the initial state), the reachability constraint holds.



Both proofs together show that the reachability constraint is fulfilled in all states reachable from the initial state. Hence it is a valid assumption that may be added to any property to strengthen its proof.



Task 1.7

Analogously to the previous task, write two properties called `in_idle_counter_is_0__step` and

`in_idle_counter_is_0__base` to establish an inductive proof for the reachability constraint `in_idle_counter_is_0`.

1. Can you prove the induction base? 
2. Can you prove the induction step? (You should not be able to prove it.) Analyze the counterexample. 

There are several ways to resolve this issue. You can identify more reachability constraints and add them to the proof of the induction step. Or, you can strengthen the inductive proof by considering more than one time frame. Here is the idea:

1. Induction step: Assume that the reachability constraint holds during n consecutive time frames. Prove that it will then also hold in the $(n+1)$ -th time frame.
2. Induction base: Show that after reset the reachability constraints holds during the first n consecutive time frames.

Write the corresponding properties and prove them. How large does n have to be in order to prove the properties?



Lab 2

[Working directory: [readserial](#)]

In the previous assignment we have verified the individual operations that the readserial design may execute. We have proven a property describing how the design can be set into a defined state after power-on (i.e., we verified the reset functionality) and we have proven a property describing how a bit sequence on the `rx_d` input is actually read and converted into a byte. In this assignment we will answer the question: Have we written enough properties for the design?

Checking completeness of a property set

Obviously, we have written enough properties if the property suite covers every possible behavior of the design. We will look at a methodology that allows checking completeness of a property suite using a software tool built into the OneSpin tool suite (the so-called *completeness checker*).

In this methodology, each property describes some operation of the design. The behavior of the design is viewed in terms of operations: Every possible simulation trace of the design can be expressed as a sequence of operations. The first operation in the trace is the reset operation. If we have written a property for every possible operation the design can execute then we have a complete set of properties.

The completeness checker in OneSpin can be used to find out whether there exists a property for every possible operation. (We cannot go into the details here but we can look at the basic ideas.) The completeness checker performs a number of tests on the set of properties written by the verification engineer:

1. The *case split test* checks that after some operation has been executed there exists a property for every follow-up operation possible in that situation.
2. The *determination test* checks that in every property the behavior of all outputs have been described according to the specification.
3. The *reset test* checks that there exists a property that moves the design into a defined state.
4. The *successor test* checks that every property only depends on inputs or on register values that have been calculated in the preceding operation (and proven in a corresponding predecessor property).

The completeness checker looks only at the properties themselves — the design (i.e., the RTL code of its implementation) is not taken into account! In order for the completeness checker to carry out the above tests, it needs to have information about

1. the sequencing of operations,
2. the signals that are to be uniquely determined by the operations,
3. the inputs that the operations depend on (remember: design information is not used).

This information must be provided to the tool in a special format. The following code snippet, contained in file **completeness.gfv** in the **readserial** project directory, captures a completeness information for the readserial design.

```
1: completeness readserial;
2: disable iff: (!reset_n);
3: inputs: reset_n, rxd;
4:
5: determination_requirements:
6:     determined(valid);
7:     if (valid) determined(data); endif;
8:
9: reset_property:
10:    sva/inst_readserial_properties/ops/a_reset;
11: property_graph:
12:    sva/inst_readserial_properties/ops/a_reset -> sva/inst_readserial_properties/ops/a_read_byte;
13:    sva/inst_readserial_properties/ops/a_read_byte -> sva/inst_readserial_properties/ops/a_read_byte;
14: end completeness;
```

In the following, we explain the example line by line.

Lines 1 and 14 encapsulate the information needed by the completeness checker.

Line 2: The completeness checks shall only be carried out for the normal operational mode, not when the reset is applied.

Line 3: The inputs to the design must be specified. (Remember that the completeness checker intentionally only analyzes the properties, not the design.) Input signals are assumed to be determined at all times.


Lines 4–7: The determination requirements specify which output signals should be uniquely determined in each operation. The valid signal (line 6) should always have a defined value. The data signal (line 7) only needs to have a defined value when the valid signal is asserted.

Lines 9 and 10: The reset operation is important because it brings the design into a defined state. The reset operation must be captured in a reset property (see the previous task description for that.)

Lines 11 to 13: The property graph specifies the correct sequencing of operations by listing all possible pairs of an operation property and a direct successor operation property.

Task 2.1

Write a completeness description for the properties you wrote for the readserial design.

- Use the file **completeness.gfv** as a starting point for your work. (The default file extension for completeness descriptions is “.gfv”, which stands for “gap-free verification”, OneSpin’s commercial name for the completeness checker.)
- Set the **onespin** tool into “MV” mode, read in your property file from the previous assignment, and then read in the completeness description. (When the file selector box appears make sure you select the file type “GFV Specification”, otherwise you might not see the file name in the selector window.)
- A new tab named “Gap Detection” appears. Click on that tab. You will see sub-tabs for each of the above-mentioned completeness checks (plus some more tabs that we ignore). Run each of the tests and debug any failing test and fix the problem. (Hint: Have we *really* covered every possible scenario in our property suite?) 

Notes:

- 1.) In case you encounter problems when the tools reads your completeness description, make sure the names of your properties are written correctly. You can find the names of the properties by clicking on the tab named “Assertion Checks” and folding out the “Properties” section.
- 2.) The reset property must be written in the following form, otherwise the completeness checker may not be able to parse it.

```
sequence reset_sequence;
    reset_n == 1'b0;
endsequence;

property reset;
    reset_sequence ==>
```

```
t ##0 state_s == IDLE and  
t ##0 valid == 1'b0;  
endproperty;
```