Hardware Verification with SystemVerilog

SV_lab_1

1. Verification of a Synchronous FIFO

1.1 Design under Test

The design under test (DUT) you will be working on is a synchronous FIFO written in VHDL. The symbol of the DUT and the functionality specification are given below.

Symbol:

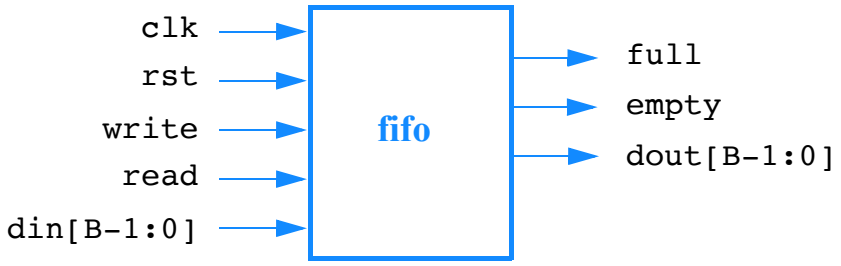

Fig. 1.1: Symbol of the DUT

Functionality Specification:

The synchronous FIFO has a single clock input `clk` for both data write (push or enqueue) and data read (pop or dequeue) operations. FIFO memory size is defined by the two parameters (generics) B and W, where 2^W is the number of words, and B is the number of bits per word. The data input and output are `din[B-1:0]` and `dout[B-1:0]`, respectively.

Data presented at data input `din` is written into the next available empty memory location on a rising clock edge when the write-enable input `write` is high. The memory full status flag `full` indicates that no more empty locations remain in the FIFO buffer. Data can be read out of the FIFO via the data output port `dout` in the same order in which it was written by asserting the read-enable input `read` prior to a rising clock edge. The memory-empty status flag `empty` indicates that no more data resides in the FIFO buffer.

Parallel read and write operations do not effect the FIFO status. In addition, the FIFO status cannot be corrupted by invalid requests. Requesting a read operation while the `empty` flag is active will not cause any change in the current state of the FIFO. Similarly, a write operation while the `full` flag is active will not cause any change in the current state of the FIFO.

The reset input `rst` is high-active. The activation of `rst` cleares the FIFO memory and initialize the full flag to `0` and the empty flag to `1`. The data output bits are set to `0`.

1.2 File Hierarchy and Overview

The file hierarchy of SV_lab_1 is organized as follows:

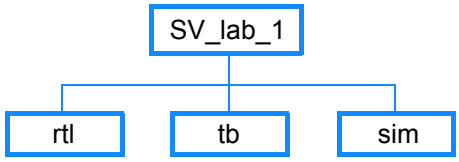

Fig. 1.2: File hierarchy of SV_lab_1

The VHDL design file (**sync_fifo.vhd**) is in the **rtl** directory. Do not edit this file.

The **tb** directory contains an incomplete testbench file (**fifo_tb.sv**), a program file (**fifo_prog.sv**), and and two header files (**fifo_env.svh** and **fifo_defines.svh**). The lab tasks describe how to complete the testbench file. The other files in this folder should remain unchanged.

In the **sim** directory, there is the compile file **compile_tb_fifo.f**, the simulation script file **run_lab_1.do** and the wave script file **wave_fifo.do**. The compile file must be completed, see compile informations of the OVL FIFO checker and the lab tasks.

Next, familiarize yourself with Questa first. To do this, study script commands and tools for checking simulation results. The next two chapters provide an overview.

1.3 Questa Script Files

Open a terminal, type `xilinx-`<tab>, and select the right choice. Start Questa in the welcome window with `vsim &`, where `&` brings the process into the background.

Compile and Simulation Scripts:

Compile and simulation scripts are files with extension `.do`. They include a series of commands for Questa and they are executed using the command `do` followed by the script name, e.g. for this lab it is

```
do run_lab_1.do
```

According to the directory structure shown above it must be started in the sim directory.

The following instructions aim to guide through basic script commands for Questa.

Terminate the Previous Simulation and Create a Work Library:

The script files should first terminate the previous simulation and provide for a cleaned up library. For simplicity, all files (design and testbench files) are compiled into the `work` library (default). If we assume that the library `work` already exists, it is better to delete it first and then create it again.

All this is achieved with the following commands:

```
quit -sim
if [file exists "work"] {vdel -all}
vlib work
```

Compile Commands:

VHDL models are compiled using the command `vcom`, e.g.

```
vcom +cover=bcs ../rtl/sync_fifo.vhd
```

Note that the scripts are called from the sim folder and therefore a path must be specified for the design and testbench files.

The option `+cover=bcs` of `vcom` and `vlog` specifies code coverage collection including branch (b), condition (c) and statement (s).

A further coverage option is (f) for Finite State Machines. Type `vcom -help` for complete syntax.

SystemVerilog files are compiled using the command `vlog`, e.g.

```
vlog -sv ../tb/fifo_tb.sv
```

where the option `-sv` specifies that the source files contains SystemVerilog code.

Another command line option for `vcom` and `vlog` is `-f <file_name.f>`, e.g.

```
vlog -f compile_tb_fifo.f
```

where `compile_tb_fifo.f` is the file including all compile commands for the testbench files.

Optimize Command:

Optimization (optional) can be used to accelerate the simulation. Objects that are needed for debugging should not be optimized away.

The output file `tb_opt` of the command

```
vopt +acc -o tb_opt fifo_tb
```

is an optimized design, where the option `+acc` enables debug.

The name after the option `-o` specifies the name of the optimized design. In this example, it is the optimized design for the fifo testbench `fifo_tb`. Type `vopt -help` for complete syntax.

Simulate Commands:

To simulate, first load the (optimized) design into the simulator using the command

```
vsim -assertdebug -coverage -onfinish stop work.tb_opt
```

where the command `vsim` brings up the Questa GUI.

The `-assertdebug` option enables SVA debugging and optimizes the Questa GUI to analyze assertion results.

The `-coverage` option enables the collection of coverage data and sets the GUI layout at tool start-up to be “Coverage“.

Additionally, `-wlfdeleteonquit` can be used to delete unused wlf files (wave files).

Type `vsim -help` for complete syntax.

Simulation is exectuted using the command `run`, e.g. `run -all`. In this case, the simulation runs until the execution flow reaches a breakpoint or the system task **$finish**.

The option `-onfinish stop` provokes the simulator to pause.

Wave Viewer:

The wave viewer can be opened using the command

```
view wave
```

Testbench objects can be added to the wave viewer using the command `add wave` in the script, e.g. using a wildcard for all testbench objects

```
add wave /fifo_tb/*
```

Alternatively you can configure the wave manually, and save (File > Save Format) your configured wave in a script file (e.g. `wave_fifo.do`). Then call this wave script in the compile and simulation script, e.g. for this lab with

```
do wave_fifo.do
```

where `wave_fifo.do` must be in the **sim** directory.

Assertions can be added to the wave viewer in a similiar way by using the path and the name of the assertion, e.g.

```
add wave /fifo_tb/my_ovl_fifo/ovl_assert/A_OVL_FIFO_FULL_P
```

to add the assertion `A_OVL_FIFO_FULL_P` of the OVL FIFO checker.

The command

```
wave zoomfull
```

can be used to display the wave completely after the simulation is finished.

To short the displayed name of the objects in the wave viewer, you can add the command

```
configure wave -signalnamewidth 1
```

Assertion Thread Viewer:

To enable the assertion thread viewer (ATV) for analyzing concurrent assertions, you can use an `atv` command in the script file, e.g.

```
atv log -asserts -enable /fifo_tb/my_ovl_fifo/ovl_assert/A*
```

for all concurrent assertions with label `A*` (or replace wildcard `*` with individual name).
Note that at the beginning, the assertion module does not contain any assertions yet.

1.4 Tools to check Simulation Results

After a simulation, the results must be checked. This includes the analysis of code coverage, assertion coverage and functional coverage (covergroups and cover directives). There are some tools available in Questa for this purpose.

For code coverage analysis, Questa indicates in the design file(s) which lines have been executed with a green tick (if code coverage is activated, see `vcom` command). On the other hand, unexecuted lines would be marked with a red cross.

Same results can seen using

```
View -> Coverage -> Code Coverage Analysis
```

for the selected object.

For assertion analysis, Questa shows pass and fail cases in the wave viewer using green and red triangles. The pass and fail cases can be analyzed in more detail in the assertion pane and with the ATV tool (see above). If the assertion pane is not active, this can be made up with

```
View -> Coverage -> Assertions
```

In the assertion pane all assertions are listed with type, failure count, pass count, and some more informations.

Note that the assertion pane is linked to the source viewer (cross referencing). Double click on one of the assertions in the assertion pane shows the source file and the line where the assertion is stated.

A right click in an empty area in the assertion pane opens a context menu for the assertions.

This applies in the same way to the Covergroups pane and the Cover Directives pane.

You can also create coverage reports as text files using

```
Tools -> Coverage report -> Text
```

or in HTML format shown in ab browser using

```
Tools -> Coverage report -> HTML
```

1.5 OVL FIFO checker

In this lab exercise, the FIFO checker of the OVL library should be used. It can check FIFO designs with different behavior.

OVL checkers are composed of one or more properties, a message, a severity and coverage constructs. Properties are design attributes that are being verified by an assertion. A property can be classified as a combinational or a temporal property. A combinational property defines relations between signals during the same clock cycle while a temporal property describes the relation between the signals over several (possibly infinitely many) cycles. Message is a string that is displayed in the case of an assertion failure. Severity indicates whether the error captured by the assertion library is a major or minor problem. Coverage indicates whether or not specific corner-case events occur and counts the occurrences of specific events.

All OVL checkers are parameterizable. To apply a OVL checker, it must be instantiated, the parameters must be specified, and the checker ports must be connected to the DUT input and output ports. Depending on the specified macros a OVL checker checks the design behavior on different levels.

The symbol, the parameters and the module instantiation syntax of the OVL FIFO checker are given below.

Symbol and Parameters:

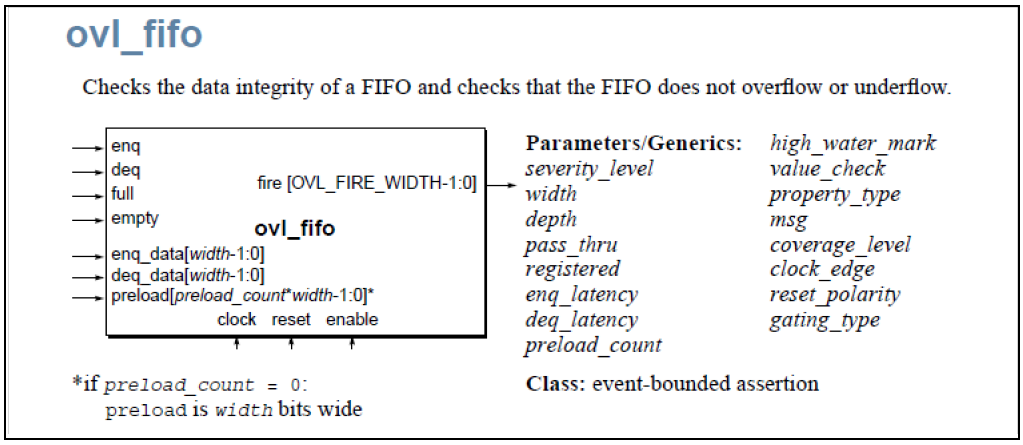

Fig. 1.3: Symbol and parameters of ovl_fifo

Syntax for Instances:

```
ovl_fifo [#( severity_level, depth, width, high_water_mark,
             enq_latency, deq_latency, value_check, pass_thru,
             registered, preload_count, property_type, msg,
             coverage_level, clock_edge, reset_polarity,
             gating_type)]
  instance_name (clock, reset, enable, enq, enq_data, deq,
                 deq_data, full, empty, preload, fire);
```

Detailed informations about all parameters and ports of the checker are given in the OVL language reference manual, see ovl_lrm.pdf.

Compile Informations:

The OVL library is stored in the directory **std_ovl**. The OVL library is already installed. The variable **$STD_OVL_DIR** points to the OVL directory.

The OVL directory contains the modules for all checkers. It also contains the include file **std_ovl_defines.h** for all OVL macros.

To compile the **ovl_fifo** checker you need to tell the compiler four compile informations:

1. Where to find the OVL library (OVL directory).
2. The file extensions for the OVL library (**.v** and .**vlib**).
3. Where to find the OVL include file **std_ovl_defines.h** (OVL directory).
4. Which macros of the OVL library should be used (see below and OVL LRM).

Essential OVL macros are **OVL_SVA**, **OVL_ASSERT_ON** and **OVL_COVER_ON**.

Detailed informations about all checker macros are described in the OVL LRM.

1.6 Lab Tasks

This lab includes the following tasks:

- Load the lab files.
- Instantiate the program (**fifo_prog**), the DUT (**sync_fifo**) and the OVL FIFO checker (**ovl_fifo**) in the testbench (**fifo_tb**).

  The program is already complete and it generates stimuli using the classes defined in the header file **fifo_env.svh**. The header file **fifo_defines.svh** contains the FIFO parameters. Do not edit the program file and the header files.

  Do not forget to change the generics of the DUT and the parameters of the OVL FIFO checker in the instances. Pay special attention to the OVL FIFO checker parameters, in particular **pass_thru, registered**, and **value_check**.

  Set the **coverage_level** to **OVL_COVER_ALL**.

  Note that some signals of the DUT, the program and the checker may may be defined with different polarities.

- Write all compile informations for the OVL checker in the file **compile_tb_fifo.f** (see compile informations above).
- Simulate the design and the OVL FIFO checker with the given script **run_lab_1.do**, your completed compile file and the final testbench file.
- Observe the results of the assertions, the cover directives, and the covergroup. Note the messages in the transcript window.