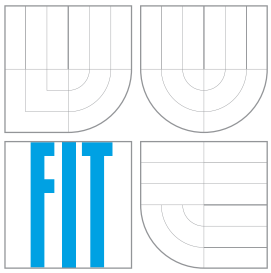
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

TEST DRIVEN DEVELOPMENT FOR FPGA DESIGN

TEST DRIVEN DEVELOPMENT FOR FPGA DESIGN

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

DÁVID HALÁSZ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. VÁCLAV ŠIMEK

BRNO 2013

Abstrakt

Tato bakalářská práce popisuje, jak může být princip TDD uplatněn u hardware, převážně pro vývoj FPGA. Je popsána důležitá teorie pro pochopení kontextu. Na referenčním návrhu jsou představeny některé dostupné a užitečné verifikační nástroje. Jeden z těchto nástrojů byl vybrán a pomocí TDD byl vytvořen a úspěšně otestován návrh komunikačního modulu SPI.

Abstract

This bachelor's thesis describes, how test-driven development can be used in hardware, especially for FPGA development. The essential theory for understanding the context is described. Some available tools for assertion-based hardware verification and unit-testing are presented and demonstrated on a reference design. One of the introduced tools was selected and with that a test-driven developed SPI interface was created and successfully verified.

Klíčová slova

aserce, verifikace, HDL, FPGA, navrh hardware, simulace, TDD, testovaci soubor, testovani aplikacnich jednotek

Keywords

assertion, verification, HDL, FPGA, hardware design, testbench, simulation, test-driven development, TDD, unit-tests

Citace

Dávid Halász: Test Driven Development for FPGA Design, bakalářská práce, Brno, FIT VUT v Brně, 2013

Test Driven Development for FPGA Design

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně na studijním pobytě na KHBO v Belgii pod vedením pana dr. ing. Jeroena Boydense a ing. Robbie Vinckeho, a na VUT v Brně pod vedením pana Ing. Václava Šimka.

.....
Dávid Halász
May 10, 2013

Poděkování

Chtěl bych poděkovat lidem v komisi výběrového řízení ERASMUS, kteří mi umožnili, abych mohl studovat na KHBO. Dále panu Ing. Šimkovi za vedení této práce a zaměstnancům výzkumné skupiny EP na KHBO za užitečné rady a pomoc.

Contents

Chapter 1

Introduction

Test-driven development (TDD) is a methodology, commonly used in software development. The concept of TDD is to write a test before the actual implementation of the code. The tests contain assertions and compare the expected result with the actual result. Gradually adding tests and expanding functionality, while keeping previous tests passing and avoiding of unnoticed bugs.

In FPGA development functional verification is usually done with testbenches and simulations. However complex designs are hard and time-consuming to verify. Therefore new verification tools and techniques are needed. Assertion-based verification, in combination with test-driven development, can benefit for hardware design. Today, assertion-based verifications frameworks exist for FPGA development. Combining the test-driven development methodology and assertion-based verification can provide major advantages in hardware verification where verification cycles take most of development time.

The main goal of this thesis is to demonstrate how the TDD methodology can be applied in the hardware context, so in the further chapters instead of using FPGA design or FPGA development terms, hardware design and development will be used. The next chapter describes the essential theory, the third chapter introduces demonstrates and compares some assertion-based and test-driven techniques on a reference design. A demonstrational hardware was made using the introduced TDD techniques and the development report can be found in the fourth chapter. Finally, the stated facts and acquired results are summarized and evaluated in the final chapter.

This thesis is shipped with a CD, which contains all the examples and the demonstrational hardware design.

Chapter 2

Theory basis

2.1 Test-driven development

2.1.1 Assertions

An assertion is a (true-false) statement in a program code defined by the developer, that (based on the developers knowledge) is always true.[5] They can be defined, e.g. in comments to help the programmers see through the code and use them to speed up the development process. Many programming languages support checked assertions, depending on the language, they can be defined with a built-in keyword or function. They are evaluated during runtime and when some of them fail, the program exits with an assertion report. From the report, the developer can determinate which assertion failed, conclude what causes the error and fix it. These assertions should be removed from the final code, some compilers have a switch for omitting them.

2.1.2 Unit-tests

A more sophisticated way of testing with assertions is using them in unit-tests. The source code should be divided into individual units, they can contain one or more functions, procedures, modules or objects.[6] Each unit should be tested individually, isolated from other test cases. In the test cases, assertions or similar techniques should be used to generate a report after running the test.

The test cases can always be written from scratch, but some patterns will be repeated in all tests, therefore it is obvious that one can take advantage of the reusability from the object-oriented programming. The first unit-test framework on this basis was developed in Smalltalk by Keny Beck, whereof the *xUnit* architecture was specified. The implementation of the architecture varies for every programming language, but the four base classes are the same:

- **test case** - the smallest unit of testing, every test should be inherited from this class
- **test fixture** - it creates the test context and does a cleanup after running
- **test suite** - a collection of test cases with the same fixture
- **test runner** - it runs the test cases or suites and generates a report

2.1.3 TDD

TDD is an agile software development methodology developed by the aforementioned software engineer, Kent Beck. The main idea of the technique is to write (automated) tests **before** implementing the functionality and step forward only when all tests have succeeded. Only that many functionality is implemented as many successfully pass the tests.

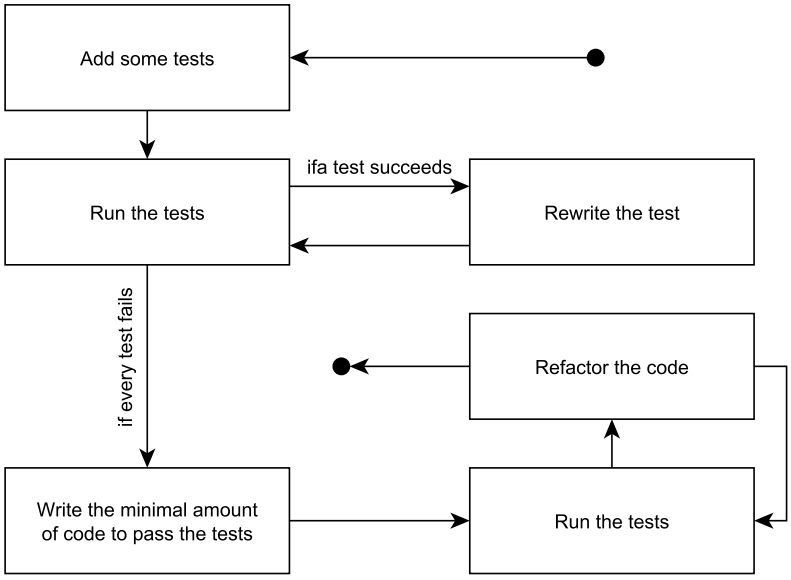


Figure 2.1: TDD's development cycle based on Beck's definition[3]

Without adding tests, it is not possible to implement new functionality. When a test succeeds initially, it is meaningless and should be rewritten to fail and be in accordance with the new functionality. In order to do this successfully, the developer should know the precise specifications and requirements of the new feature.

When every added test fails, the minimum amount of code has to be written to pass the tests. The added code usually does nothing more than pass the tests and in many cases it is not very elegant. This is the part of the methodology, it will be fixed in the next step. When some of the tests are unsuccessful, the code should be corrected until it passes all the tests.

Even if all the tests are successful, the functionality is still not implemented, the code must be refactored. This means that the internal behavior of the code should be changed to eliminate duplication. The external behavior must remain the same, so all the tests should successfully pass after refactoring.

When the refactoring is done and every test succeeds, the task of adding new functionality is completed. The tests are the guarantee that the feature works correctly, so the quality of the tests has an influence on the quality of the definitive code. To create new functionalities, these steps should be repeated as long as the software is not complete.

The size of the steps can be chosen by the developer and can be changed any time during the development. However, this should be carried out with caution since making very small steps may result in easy coding, yet the whole product will be finished later due to the high number of iterations.

Increasing the size of the steps results in a lower number of iterations, but the coding will be more complex. For every project there is an optimum, but it is still variable depending on the skills of the developer. This optimum should be determined as soon as possible and be used in the whole process. It has to be changed when it is necessary.

The methodology teaches how to write and interpret specifications. The errors are discovered earlier and faster, so debugging becomes quick and easy. It forces to write the simplest code, so the final code will be clean and easy to understand. On the other hand, it is hard to learn and not everybody can write good tests. It is not suitable for developing a program which does not have an exact specification.

2.2 Hardware verification

According to the PMBOK Guide[1]:

- „**Validation**. The assurance that a product, service, or system meets the needs of the customer and other identified stakeholders. It often involves acceptance and suitability with external customers. Contrast with *verification.*“
- „**Verification**. The evaluation of whether or not a product, service, or system complies with a regulation, requirement, specification, or imposed condition. It is often an internal process. Contrast with *validation.*“

In the hardware field, verification means a process that proves a homomorphic relationship between a register-transfer level (RTL) model of the developed hardware and his specifications.

2.2.1 Simulation and testing

The method is often called *bug hunting*, because its main objective is to find bugs in the design. With simulation, it is only possible to detect the errors, not to fix them.[11] A special software environment is necessary which compiles the design and runs it. The output is often a waveform where the logic levels of inputs, outputs and inner signals are shown in the function of time. Since the unit under test needs some input variables, a testbench is needed to generate these. All inputs and outputs of the unit are connected to this testbench.

The main disadvantage of this method is that it is time-consuming. Simulating a complex design can take days or more to run, and with the examination of the output waveform, the problem is similar. Eventually it is very hard or impossible to write a testbench which generates all possible input combinations. In many cases simulation is used only for small parts of a larger module.

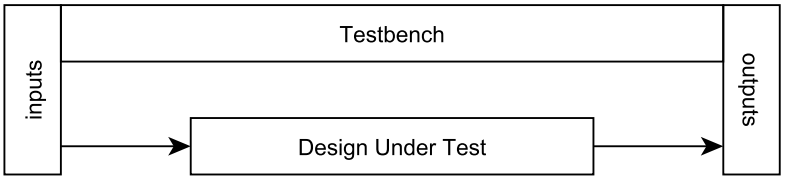


Figure 2.2: Typical usage of a testbench in simulation[4]

2.2.2 Formal verification

Formal verification uses formal methods and mathematics to verify that the specifications are preserved in the implementation.[10] It can completely prove the correctness of the design, not just draw some conclusions from test results. That is because testing can be successfull even then when it does not cover the whole system and the formal methods should always be working with the complete model. It is not always sure that a formal verification process will be finite and will be terminated at some point, but it can be still be helpful to find some errors in the design.

When a system can be represented by a finite-state machine or equivalent, it is possible to check algorithmically (e.g. by state space search) if the system succesfully satisfies a given specification. This method is called **model checking**.

An alternative can be **theorem proving** which is a deductive verification process. The specifications and the model can be described mathematically, equivalence can be proved between them. This method is semi-automated, it often requires a significant manual effort of users.[11]

Static analysis is linked with automated analysis of the source code. It can be used not only for verification but for optimalization and code generation too; his main attribute is that it does not the model of the system and it avoids the execution of the code.

2.2.3 Functional verification

Functional verification is more practical than the formal one. It verifies the system by examining the inputs and outputs of various simulations. To facilitate the whole process, it uses more sophisticated techniques like constrained-random stimulus generation, self checking mechanisms, assertions and coverage-driven verification.

When verifying large systems, it is very difficult to test the complete set of input combinations. A suitable alternative is to generate random inputs which are circumscribed by constraints to be valid for testing. This is called **constrained-random stimulus generation**. The constraints can be targeted, e.g. to cause corner cases or given states of the system.

With **coverage-driven verification** it can be measured which parts of the system were correctly verified. The types of observable coverages are:

- Code coverage
- Functional coverage
- Path coverage
- FSM coverage
- others...

Assertion-based verification uses assertions (see 2.1.1), which can be helpful to formally express properties of the system and to verify awaited (partial) results. Failing an assertion terminates the verification and makes easy to find the source of the problem.

Self checking mechanisms are based on calculating the outputs independently from the implementation and comparing them with the outputs received from the simulation. It can be used for detecting data loss and the correct order of the output data. This process can be fully automatized.

2.3 Test-driven development in hardware verification

It is necessary to redefine the hardware development cycle and the verification process so that they should fit into the TDD paradigm.

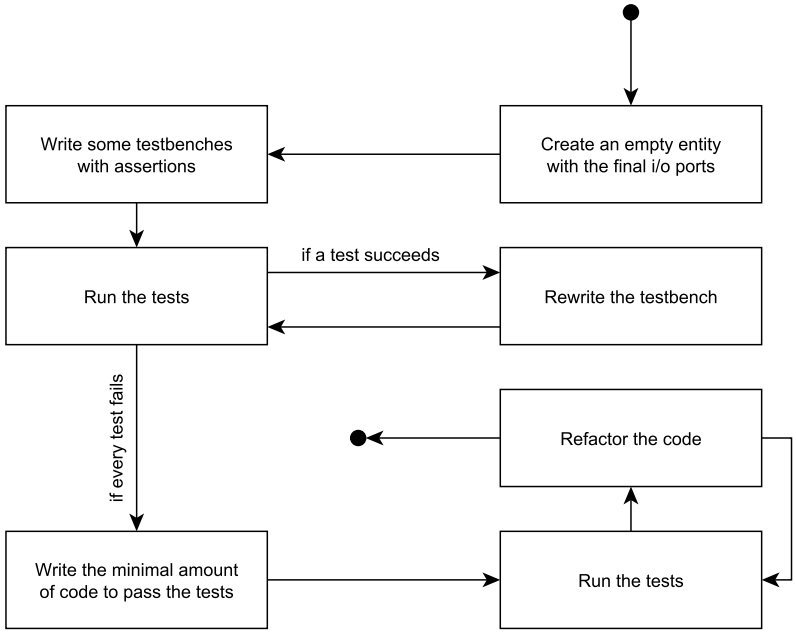


Figure 2.3: TDD's development cycle modified for hardware development

Creating a testbench before implementing any module is not possible, because the testbench needs to instantiate the entity to test (see 2.2). So before creating the first group of tests, the final input/output signals should be implemented into an empty entity.

Implementing and running the testbenches can be done in the classical way, but using some macros or additional tools can facilitate the development process. Many hardware verification environments have support for assertions, but it is also possible to extend the environment with unit-testing functionalities. When the testbenches are created, they should fail at the initial run.

According to Beck's second step the minimal amount of code has to be implemented, which successfully passes all the tests. This newly added code is sometimes not synthesizable and it is often just a workaround which fakes the tests. To make the code complete and synthesizable, it has to be refactored in the similar way, which was mentioned in 2.1.3, combined by running the hardware synthesis.

After the refactoring a synthesizable code should be produced. This can be reached by running a synthesis tool together with the simulation during the refactoring process. When it is done, the new functionality is successfully implemented and the whole process can be started over.

Chapter 3

Available tools

3.1 Reference design

To demonstrate the available techniques which can facilitate the TDD process, a reference design was made based on the 16550 UART module. The UART is a simple bi-directional serial communication device, it converts data bytes into individual bits and sends them sequentially and vice-versa.

According to the specification when no data is received, on the receiver port is logic 1. The data transfer starts with a start bit (which is logic 0 for one baud-cycle), from the falling edge of this bit the receiver should be able to generate a synchronized baud rate using the clock signal. After the start bit the data bits are received, starting with the first bit, one in every baud-cycle. The following received bits can be parity and/or stop bits, depending on the chosen configuration. The received data should be forwarded to the parallel port. The occuring parity and other errors should be signalized, such as the usage of the module. The transmitter should work on the same principle with the same parallel port, only in the opposite direction. To settle the difference between the baud rate and the clock signal, additional FIFO modules should be connected to the transmitter and receiver. Another signals are needed to select the transmission configuration, indicate errors, usage, availability and decide the flow on the parallel port. The specification also mentions some signals for communicating with a serial modem.

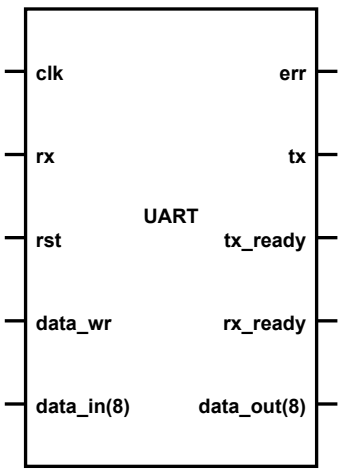


Figure 3.1: Schematic of the simplified UART module

In the reference design, the above specification was significantly simplified. The data-flow configuration was fixated to 8-N-1 (8 data bits, no parity, 1 stop bit). The baud rate selection was eliminated too, the signal is generated directly from the clock signal, dividing it by 16. The original two-way 8-bit data port has been separated into independent input and output ports, the FIFO on the receiver side and the modem support were eliminated. Thanks to these simplifications, some configuration and indicational signals were necessary.

The module was implemented using conventional hardware development techniques in VHDL. The final design was successfully verified by using simulations.

3.2 Assertions

To demonstrate the power of assertions on the reference design, some statements from the sub-modules' specifications were selected:

- Receiver
  - when *rx_ready* is in logic 1, the received data should match with the sent
  - when a stop bit is not received at the end of the communication, the *err* port must be in logic 1 for one clock cycle
- Transmitter
  - the communication should be started with a start-bit which is logic 0
  - each data bit should match with the *tx* port for 16 clock cycles
  - the communication should be ended with a stop bit which is logic 1
- FIFO
  - after a reset, on the *empty* port should be logic 1
  - after a write, on the *empty* port should be logic 0
  - when one byte is written and read, they must match
  - when multiple bytes are written and read, they must match and the same order must be preserved (FIFO)
  - after writing 254 bytes, on the *full* port should be logic 1
  - after reading out all values, the last one should be preserved on the *data* port

For each sub-module a new testbench was designed, containing assertions with the statements listed above. The testbenches were implemented first in VHDL using standard assertions, then they were converted into PSL and finally all three testbenches were rewritten into SystemVerilog.

The testbenches were successfully compiled and run using the Mentor Graphics's QuestaSim environment. All of the source codes (including some scripts for automated running) are available on the CD attached to the present paper.

3.2.1 VHDL

The assertions in VHDL can be defined in linear structures (processes, functions, procedures) and in concurrent descriptions (entities, architectures) too. The construction is not synthesizable, so they can be used only in simulations.

```
assert condition report string severity level;
```

Listing 3.1: VHDL assert syntax

When the *condition* is false, the *string* is written to the output console with the given *level*. This level can be *note*, *warning*, *error* or *failure*. If the severity-level pair is omitted, the default level will be used, which is *error*.

There are two modes of using these assertions in the practice. The first is to add them directly into existing stimulus processes between other lines of code. This makes easier to check the value of a signal in a given time moment, because it is not necessary to search for the expected results in the simulation waveform. The second solution is to use concurrent assertions which are evaluated in every time moment (e.g. two signals cannot be equal) or create a process which waits for a triggering event and evaluates assertions (e.g. two nanoseconds after setting a signal to high, an other has to be low). It is necessary to define the triggering events in the stimulus processes which can be very complicated when having several tests.

```
test_empty_after_reset : process begin
        wait until rst='1';
        wait until rst='0';

        assert empty='1'
        report "Test: empty_after_reset FAILED!"
        severity error;

        assert empty='0'
        report "Test: empty_after_reset PASSED!"
        severity note;
end process test_empty_after_reset;
```

Listing 3.2: VHDL assertion in a separate process

To prevent code repetition in the stimulus processes, assertions can be written into procedures/functions and they could be called when necessary. Unfortunately VHDL does not allow wait statements in procedures/functions, so complex (e.g. timed) assertions should remain inside stimulus processes.

Concurrent assertions are evaluated in every discrete moment of the simulation process, so they are only useful for checking simple statements which are independent of time. For example, they can be used for checking that two signals are never in logic 1 at the same time, but it is not possible to validate what happens in the next clock-cycle after triggering an event.

```
assert write='1' and read='1'
report "Read and write could not be in logic 1 at the same time."
severity failure;
```

Listing 3.3: Concurrent VHDL assertion

It is a great advantage that assertions can be combined with almost any VHDL control structures. For example, the correctness of the transmitter module was verified using a for loop which compared the output during time with the appropriate bit from the parallel input.

```
for i in 0 to 7 loop
        wait for clk_period*16;

        assert tx=data(i)
        report "Test: bit #" & integer'image(i) & " FAILED"
        severity error;

        assert tx/=data(i)
        report "Test: bit #" & integer'image(i) & " PASSED"
        severity note;
end loop;
```

Listing 3.4: Assertion combined with a for loop

To generate an advanced text-based output with assertion reports, it is necessary to define two assertions for each property (see the example above). One has to fail when the property is not true and the other should report when the negation of the property fails. That is the only way to generate a report not just when a test fails, but when it succeeds too. It makes the report more transparent when different severity levels are used for the two assertions. When running the simulator, a special option is needed to generate a text file containing the reports.

When the amount of input/output signals is large or the simulation time is very long, the assertion report can become very difficult to follow. Whenever a test fails, checking the waveform is inevitable. Some simulation environments provide an option to show the assertions directly in the waveform.

The language is suitable for test-driven development, but only for smaller projects with a moderate number of tests. The simplicity of the concurrent assertions makes them almost entirely unnecessary, because a non-concurrent assertion in a process can do the same and using only one type of assertions makes the code cleaner.

3.2.2 PSL

The Property Specification Language (PSL) can be used to formally describe the properties of hardware designs. It is independent from hardware description languages, usually it is embedded into comments or written into separate files. His purpose is to define assertions, it provides very sophisticated and advanced solutions. The language can be divided into four layers.[2]

Boolean layer

In the boolean layer logic expressions can be specified on the lowest (true/false) level using standard HDL syntax. It is extended with utility functions to detect e.g. one-hot-encoding, changes in the value of the expression (together with determining his previous value) and with some logical operators.

Temporal layer

The temporal layer specifies when the expressions from the previous layer should be valid. The time window of the validity can be specified as Foundation Language (FL) temporal operators or by using Sequential Extended Regular Expressions (SERE). Combining these two methods, almost any property can be specified with PSL. The temporal units can be grouped into named sequences and so they are reusable.

Verification layer

In the verification layer restrictions, assertions, assumptions and functional coverage can be specified using the previous layers. This layer also offers the division of the verification into verification units (*vunit*) which can be bound/unbound to modules and can be inherited.

Modeling layer

The modeling layer should contain the auxiliary HDL code which is not part of the hardware design, but it helps to describe combinational signals and/or complex state machines.

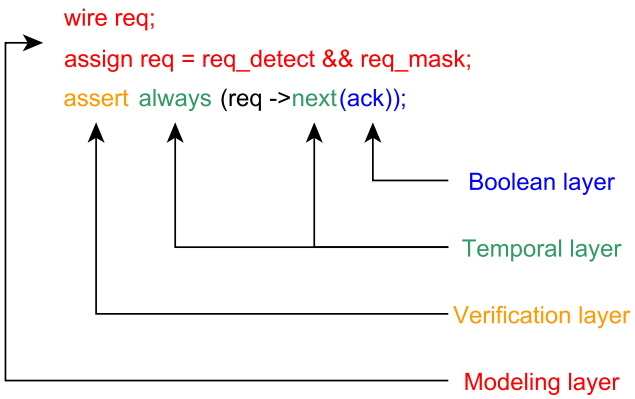


Figure 3.2: PSL Layers [2]

The PSL assertions could be directly integrated into any hardware description language (usually into testbenches) using comments and a special *psl* keyword, but they can be written into a separate file as well. Depending on the simulation environment, some additional switches may be needed to interpret these commented lines and show their output together with the simulation results. Similarly to VHDL assertions in separate processes, triggering events should be added in the stimulus processes.

```
-- psl property name is always {(cond_1)} |-> {(cond_2)} @clk_ev;
-- psl assert (name) report "string";
```

Listing 3.5: PSL property definition and assertion syntax example in VHDL

Because the PSL is logically separated from the testbench, it is not possible to write assertions inside stimulus processes. Instead of them, the conditions prior to an assertion could be described using the temporal layer. These conditions can be grouped into sequences and they can be used in multiple assertions.

The following example demonstrates a very long assertion which was used to test the functionality of the transmitter module. The same test was described much simpler in VHDL using a for loop (see listing 3.4).

```
-- psl sequence data_bits is
-- psl {tx=data(0)[*16];tx=data(1)[*16];
-- psl tx=data(2)[*16];tx=data(3)[*16];
-- psl tx=data(4)[*16];tx=data(5)[*16];
-- psl tx=data(6)[*16];tx=data(7)[*16]};
-- psl property data_test is always
-- psl {tx_ready;not(tx_ready);tx[+];
-- psl not(tx)[*16]} |=> data_bits;
-- psl assert data_test;
```

Listing 3.6: Matching the tx output with the data input using PSL

Because the PSL is evaluated independently by the simulator, it is possible to combine it with VHDL (or other HDL) assertions. The developer can always select that one for each assertion, which can describe its properties easier.

Unlike VHDL, it does not supports custom text-based output generation, so a final report generation has to be implemented in the simulation environment, or by using e.g. TCL or other scripts. However, the assertion results can be summarized in the simulation waveform window, from which the developer can easily determine which assertion failed or passed in a given time moment.

In larger projects, PSL is not an optimal solution, because it needs to be combined with another language which generates the stimulus processes. Using multiple languages in larger projects and keeping them in accordance is a difficult task for any developer. The sequence-based property specification, however, is a very useful technique, it should be implemented in other verification languages.

3.2.3 SystemVerilog

SystemVerilog is a hardware definition and verification language developed from Verilog-2005. Syntactically the two languages are the same, the SystemVerilog is just extended with some features for verification such as object-oriented design and complex PSL-like assertions.

Non-concurrent assertions are supported and they are very similar to the VHDL ones. By using them, it is possible to generate reports with advanced text-based output. Unlike in VHDL, it does not need two assertions to generate output for both test failing and passing.

```
label:
        assert (condition)
        $display("message␣if␣the␣assertion␣passes");
        else $error("message␣if␣the␣assertion␣fails");
```

Listing 3.7: Non-concurrent assertion syntax in SystemVerilog

It supports concurrent assertions too, but in much advanced level than VHDL. It was inspired by the PSL, so only few differences are present, e.g. the logical and timing operators should be described using SystemVerilog syntax. Boolean values could be defined in sequences, they could be specified in properties and finally they could be evaluated using assertions.

It also provides local variables inside sequences and properties which could be very helpful when testing reactions to external inputs. However, the the syntax of defining a value for a variable can be very complicated, e.g it is not possible to pass a value in a discrete time moment without specifying at least a boolean value.

```
property read;
        reg[0:7] input;
        en ##1 (1, input=rx) |=> data_read==input;
endproperty;
always @(posedge clk) assert property(read);
```

Listing 3.8: PSL-like assertion example with a local variable

When it is complicated to describe an advanced sequence with concurrent assertions, it is possible to combine standard SystemVerilog control and timing structures with non-concurrent assertions.

```
always @(posedge data_req) begin
        #18 start_bit:
        assert (tx == 0)
                $display("Test:␣start_bit␣PASSED!");
                else $error("Test:␣start_bit␣FAILED!");

        for (integer i=0;i<8;i++) begin;
                #32 data_bit:
                        assert (tx==data[i])
                        $display("Test:␣data_bit␣#%d␣PASSED!", i);
                        else $error("Test:␣data_bit␣#%d␣FAILED!", i);
        end;
        #32 stop_bit:
        assert (tx==1)
                $display("Test:␣stop_bit␣PASSED!");
                else $error("Test:␣stop_bit␣FAILED!");
end
```

Listing 3.9: Assertions combined with standard SystemVerilog control structures

The text-based report generation of the two types of assertions could not be joined into one file. The concurrent assertions can use only the simulator's built-in report-generation system which shows only the failure/pass count of each assertion. Between assertion-based verification tools, SystemVerilog supports the most features, it can be used object-oriented, so an advanced unit-test framework can be created using just this language.

As a demonstration, the reference VHDL design was instantiated in a SystemVerilog testbench, there was no need to reimplement everything in SystemVerilog. PSL-like and non-concurrent assertions were used together, for each always that one which can describe the given properties simpler.

3.2.4 Conclusion

All of the previously mentioned tools are usable together with the TDD methodology. The automatization of running the tests does not depend on the language of implementation, but on the used verification environment. It is an important fact, that checking the waveform window can not be avoided with automated test-report generation due to high number of possible input/output signal combinations.

The VHDL assertions are not suitable for production use, just like PSL, they should be used together. However, for a developer it is easier to use only one language and SystemVerilog does not need any other language to create advanced tests. The missing repetition operators could be substituted by non-concurrent assertions combined with standard control structures or by combining with PSL when it is really necessary.

	VHDL	**PSL**	**VHDL+PSL**	**SystemVerilog**	**SV+PSL**
Concurrent assertions	*only simple*	*yes*	*yes*	*yes*	*yes*
Non-concurrent assertions	*yes*	*no*	*yes*	*yes*	*yes*
Assertions combined with control structures	*yes*	*no*	*yes*	*yes*	*yes*
Full scale of repetition operators	*no*	*yes*	*yes*	*no*	*yes*
Advanced timing options	*no*	*no*	*no*	*yes*	*yes*
Can be combined with other languages	*no*	*yes*	*N/A*	*no*	N/A
Assertions groupable into units	*no*	*yes*	*yes*	*yes*	*yes*
Number of languages to know	*1*	*1*	*2*	*1*	*2*
Suitable for production use	*no*	*no*	*yes*	*yes*	*yes*

Table 3.1: Comparison of the three languages and they combinations

Depending on the size and complexity of a hardware design, the table above can serve as a starting point to select the best language (or language combination) for assertion-based test-driven development. For more complex projects, more advanced tools, such as unit-test frameworks or test automation scripts should be used.

3.3 Unit-test frameworks

As it has been mentioned before, unit-test frameworks provide an environment for automated testing which can be very useful for TDD. For implementing e.g. the xUnit architecture, object-oriented programming should be supported by the HDL at least on the verification side.

3.3.1 VhdlUnit

Unfortunately, VHDL does not support object-oriented programming, so it is not possible to implement the xUnit architecture in it. However, it is possible to define macros for regularly used testing structures and implement external scripts for test automation and better report generation.

The VhdlUnit was made on this principle: procedures are helping the test creation and a TCL script generates a logically arranged report. Unfortunately, it is documented only in Polish and it has not been not under development for the past nine years.

After examining the source code, it is sure that the tool cannot be combined with PSL assertions, so except the html test report generation and the unified testing it does not give anything more. It is not suitable for production use, because it is easier for a developer to write his own scripts and macros than using a non-documented framework.

3.3.2 SVUnit

The SVUnit framework can be divided into an object-oriented model of a unit-test framework for SystemVerilog and helper scripts for code generation implemented in Perl. It is a very young tool still under development, in this document is introduced using [9].

Objects

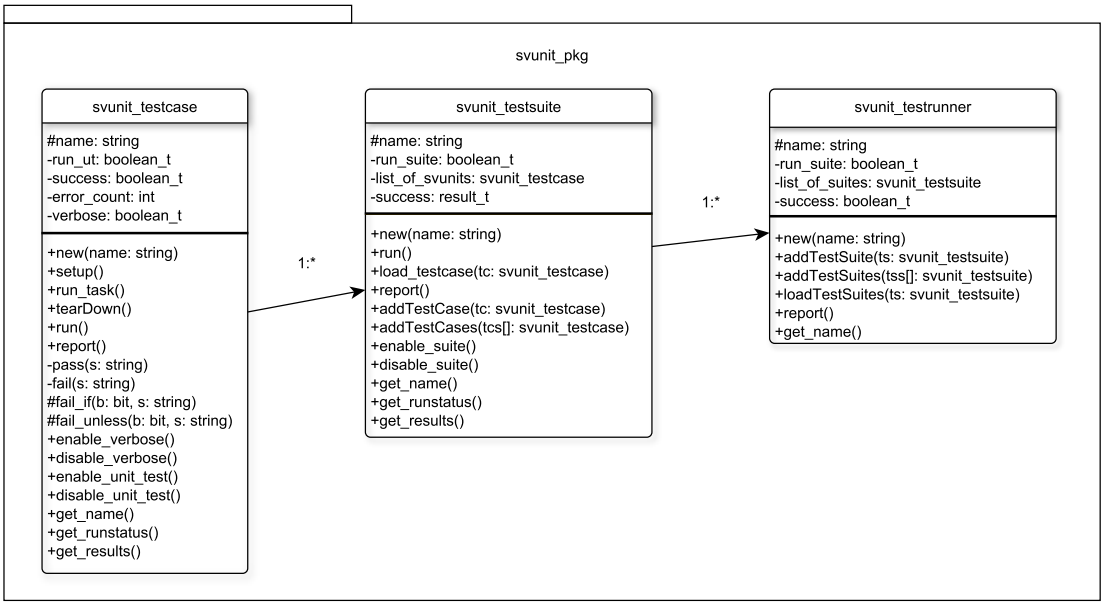


Figure 3.3: SVUnit class diagram

Each unit-test should be inherited from the *svunit_testcase* class, it is a file naming convention in the framework which requires that the unit-test filename should be ended with *_unit_test.sv*. In the unit-test three functions should be defined:

- *setup():* - here should be initialized the test preconditions
- *run_test():* - this runs the tests
- *teardown():* - here should be implemented the cleanup

The unit tests can be grouped into test suites using *add_testcase()* and *add_testcases()* functions. On smaller hardware modules, sometimes one test suite should be enough for the whole module. For larger modules, it is recommended to aggregate only those tests which can tolerate each other's context.

All test suites should be collected into a test runner inheriting the *svunit_testrunner* class and using the *add_testsuite()* and *add_testsuites()* functions. This test runner will iterate all the test suites and they will iterate each unit-test inside. The unit-tests and the test suites provide a flag to enable/disable their run.

Finally, in the highest level the user's test runner class should be instantiated and his *run* method should be called to start the testing. When everything is set correctly, all test suites and tests will be iterated and finally a report is made containing the results.

Scripts

To make the test-creation easier, a set of Perl scripts are available. These scripts can generate unit-tests, test suites and test runner. They also provide adding unit-tests to existing test suites or adding a new test suite to the existing test runner. These scripts are using some global variables, which can be set in the *svunit_test_globals.pl* file, such as the format of the header text.

- **create_unit_test.pl**
  The script takes an existing class or header file and generates for him a unit-test template using the file naming convention mentioned before. It takes all the functions from the original class and generates for each an empty test function with the prefix *test_*. Each generated function is added into the *run_test()* function.

  For example, for the *transmitter.sv* file with a class named *transmitter*, it will create a file *transmitter_unit_test.sv* with a class named *transmitter_unit_test* which is inherited from the *svunit_testcase* class. An empty *test_baud_gen()* function is generated for testing the original *baud_gen()* function. Finally the *test_baud_gen()* and all other test functions are added into the *run_test()*.

  The tests should be implemented by the user using the *'FAIL_IF* and *'FAIL_UNLESS* macros or using any other methods which increment the *error_count* variable when an error occurs.

```
# create_unit_test.pl [ -help | -out <output_file> | -i |
                        -author "name" | -overwrite | <filename> ]
```

Listing 3.10: create_unit_test.pl script syntax

  The *„-i“* argument enables the interactive mode which allows the user to select the functions to test, so some functions can be omitted from testing. The description of any other argument is trivial.

- **create_testsuite.pl**
  The script creates a test suite for the unit-tests within the current directory (or in all subdirectories when the *„-r“* argument is set), using a search for files that are ending to *„_unit_test.sv“*. It creates a template which is inherited from the *svunit_testsuite* class and appends each test into the suite using the *add_testcase()* function. Interactive mode is supported just like in the script before and with the *„-add“* argument a single test can be added into an existing suite.

```
# create_testsuite.pl [ -help | -out <output_file> | -i | -r |
                        -author "name" | -overwrite | -add <testname> ]
```

Listing 3.11: create_testsuite.pl script syntax

  As an output, a package file is generated that imports the *svunit_pkg* and includes all required files for running the unit-tests. For larger projects, this package file should be modified by including additional required files.

- **create_testrunner.pl**
  The script works similar to the *create_testsuite.pl*, it aggregates test suites not unit-tests, the syntax is the same.

- **create_svunit.pl**
  This script includes the previous three scripts and should be used on existing environments. It iterates through the current directory (or in all subdirectories when the *„-r“* argument is set) and searches for all SystemVerilog files. For each declared class generates a unit-test, it aggregates the unit-tests into test suites based on the subdirectory structure. It also creates a test runner including all generated test suites. When the *„top“* argument is set, an *svunit_top.sv* file is generated which instantiates the test runner and calls the *run()* function. A package file for including all necessary files is generated as well.

```
# create_testsuite.pl [ -help | -out <output_file> | -i | -r | -no_ut |
                        -author "name" | -overwrite | -top ]
```

Listing 3.12: create_svunit.pl script syntax

Only two simple assertions (‘FAIL_IF and ‘FAIL_UNLESS) are supported, concurrent PSL-like assertions are not. They can be added into the code and the simulator will evaluate them, but the framework's test report will not contain them. However, their combination would not make any sense, it just makes test creation diffucult.

An advanced simulation environment, such as the QuestaSim and a Unix-based operating system is necessary to use this framework. Unfortunately, the QuestaSim used in previous sections was licensed only for Windows, so it was not possible to try and demonstrate this framework.

3.3.3 MyHDL

The MyHDL is not a unit-test framework, but a Python module which provides hardware description and verification using Python language. So all the advantages of Python, such as easy learning, simplicity and elegancy, can be used in hardware development. It supports concurrent hardware modeling similar like VHDL processes and signal-like classes for connecting endpoints. It has a built-in simulator, but it supports Verilog co-simulation, too.

It does not support hardware synthesis, but it can convert the Python descriptions to synthesizable VHDL/Verilog code. High-level constructs are usable for hardware descriptions, such as objects and exception handling. When running the code, it is hard to find the errors, because sometimes the interpreter just skips them without displaying any error or warning. PSL-like assertions are not supported, testbenches have to be written using Python structures.

The MyHDL alone does not support unit-testing, but it is possible to use Python assertions or a unit-test framework such as xUnit or pytest. From the TDD's point of view, it gives nothing more than VHDL combined with assertions, but it is possible that in the future it will support advanced timing from the verification side.

Some parts of the reference design were tried to reimplement in MyHDL and testbenches were added, too, by using assertions. During the simulation with the built-in simulator, it skipped running of multiple processes without giving any notifications or error messages. When the order of the processes in the code was changed, some errors were printed out, but from the output it was not possible to detect where the errors are occured. Conversion to VHDL/Verilog code returned with similar errors. From this it can be stated that MyHDL is not suitable for production use.

3.3.4 Conclusion

Unfortunately, there was a problem with every unit-test framework, so they were not demonstrated. None of them is ready for production use, but in the future they could be the essential tools of hardware development.

Because unit-testing with the xUnit architecture is defined as object-oriented, a fully object-oriented hardware description and verification language with PSL-like concurrent assertions could be an optimal solution. SystemVerilog is very close to that description, because the verification part can be fully object-oriented. But the potential is inside MyHDL too, because Python is a very powerful language from the semantical side.

Chapter 4

Demonstration

In the final part of this thesis, a complete hardware-design was made using test-driven development. The choice fell on an N-bit parallel-to-SPI interface which can communicate with M number of slaves.

The implementation was planned to made using the SVUnit framework, but it had platform collision problems, so finally it was made in VHDL combined with SystemVerilog. The entity was made in VHDL and it was instantiated in a SystemVerilog testbench. To make test running and report generation easier, a TCL script was created.

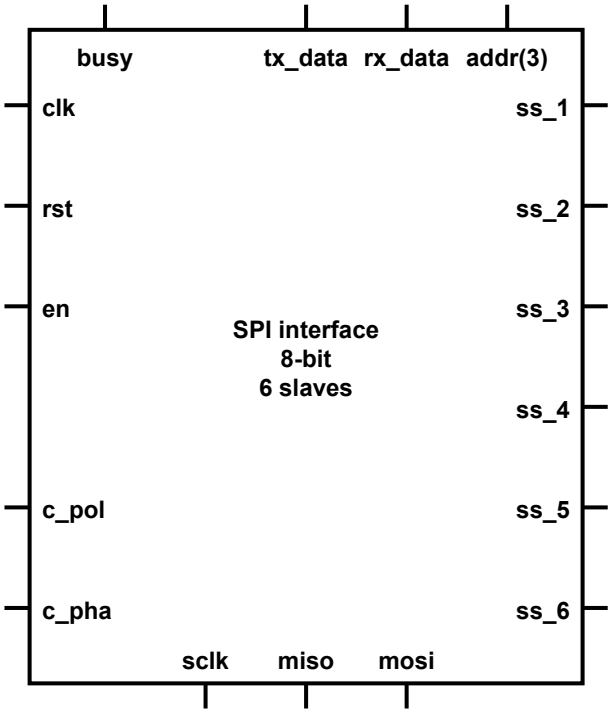


Figure 4.1: Schematic of a 8-bit parallel-to-SPI interface which supports 6 slaves

The SPI is a serial, synchronous and bi-directional communication interface between two endpoints. One endpoint operates in master, the second in slave mode. The master generates the clock signal and initiates communication, the slave is only active when the master allows it. It is possible to connect many different slave devices to one master, but only one slave can be active at a time.

4.1 Specification

As mentioned in 2.3, a very accurate and clear specification is necessary before writing the tests. For the demonstration module it was made using [8] and [7].

Port descriptions

Name	Data width	Mode	Description
clk	1	in	Clock signal
rst	1	in	Asynchronous reset
en	1	in	Initiates a transaction when in high
c_pol	1	in	Clock polarity selector
c_pha	1	in	Clock phase selector
addr	$\lceil \log_2 M \rceil$	in	Slave address selector
ss_1	1	out	Selects the first slave when in low
ss_2	1	out	Selects the second slave when in low
ss_3	1	out	Selects the third slave when in low
...	...	...	...
ss_M	1	out	Selects the M-th slave when in low
sclk	1	out	SPI clock signal
miso	1	in	Master in slave out
mosi	1	out	Master out slave in
tx_data	N	in	Data to transmit
rx_data	N	out	Received data
busy	1	out	Busy indicator

Input/output

Communication through SPI requires four data wires. The master should generate a *sclk* signal and pull one from the *ss* signals to low to activate a slave. The bi-directional communication is then realized through the master out, slave in (*mosi*) and the master in, slave out (*miso*) wires.

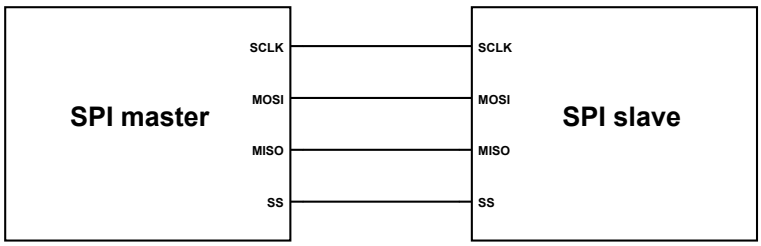


Figure 4.2: SPI master and slave connected together

The information which sent to the *tx_data* bus on the parallel side is transmitted sequentially through the *mosi* wire, one-by-one. The data receiving works on the same principle, only in the opposite direction through the *rx_data* bus and the *miso* wire.

Timing

From the clock signal's point of view, the SPI interface has four operational modes. The phase and the polarity of the clock signal both have two varieties which can be combined. In order to set these values on the master, two input wires are needed, the *c_pol* for the polarity and *c_pha* for the phase selection. The *c_pol* sets the initial value of the SPI clock signal, so when it is in logic 0, the *sclk* starts from logic 0 or logic 1 when it is in logic 1. The edge of the clock signal which the SPI interface should react to can be set with the *c_pha* signal.

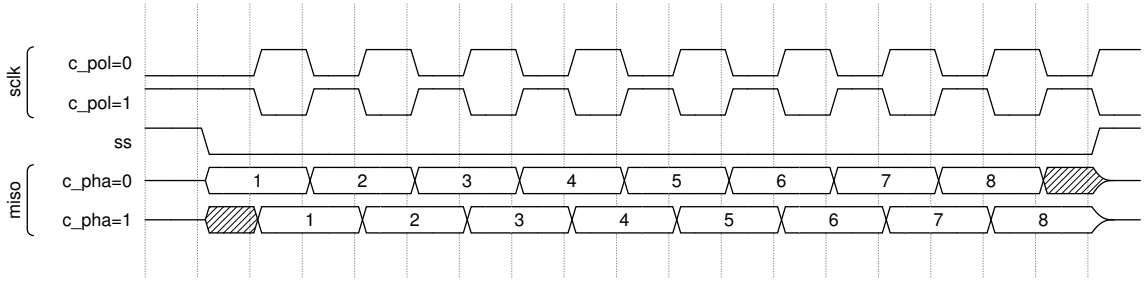


Figure 4.3: SPI operation modes demonstrated on waveform [8]

The clock generation was completely removed from the design, it is necessary to provide an external clock signal which has to be twice as fast than the desired SPI clock frequency.

Multiple slaves

There are several techniques to connect a master with many slaves, but the most well known is to use common *mosi*, *miso* and *sclk* signals. For each slave device, the master has to possess one dedicated slave selection wire which are used to activate the appropriate slave device. This requires an *addr* bus which should be minimally code M number of addresses to select the proper slave device and M number of *ss_x* signals (where $x \in \langle 1; M-1 \rangle$).

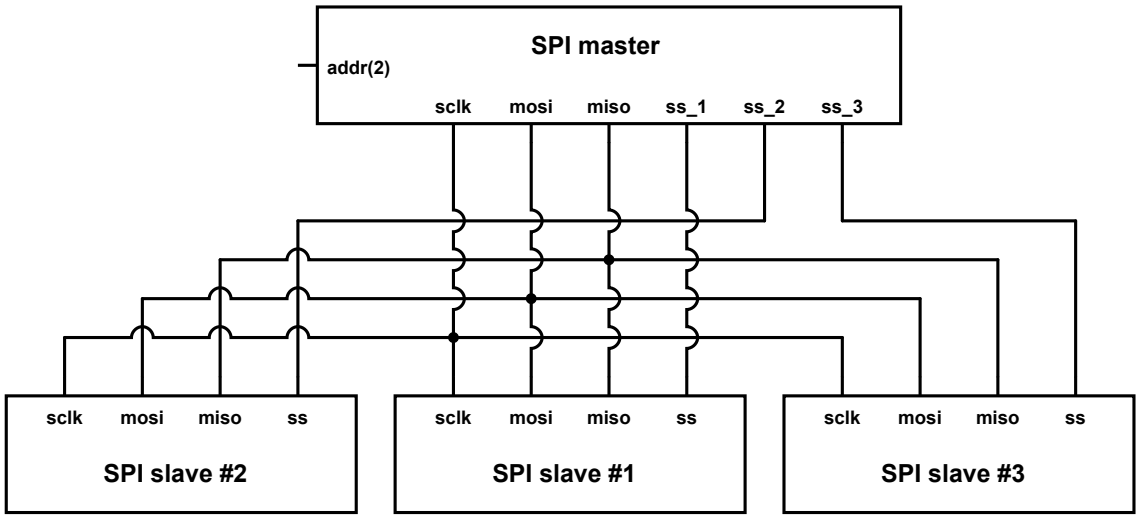


Figure 4.4: Multiple slaves connected to one master

Transaction description

When the *busy* signal is in logic 0, the module is idle and ready to begin a transaction, otherwise it will ignore anything except reset (see below).

The transaction can be initiated by setting the *en* signal to high. On the first rising edge of the *clk*, the module latches the settings and the *tx_data*. On the next rising edge, the busy flag is set and the sending of data has commenced. After all bytes were sent, the busy signal goes back to logic 0, which indicates that the received data is available on the *rx_data*.

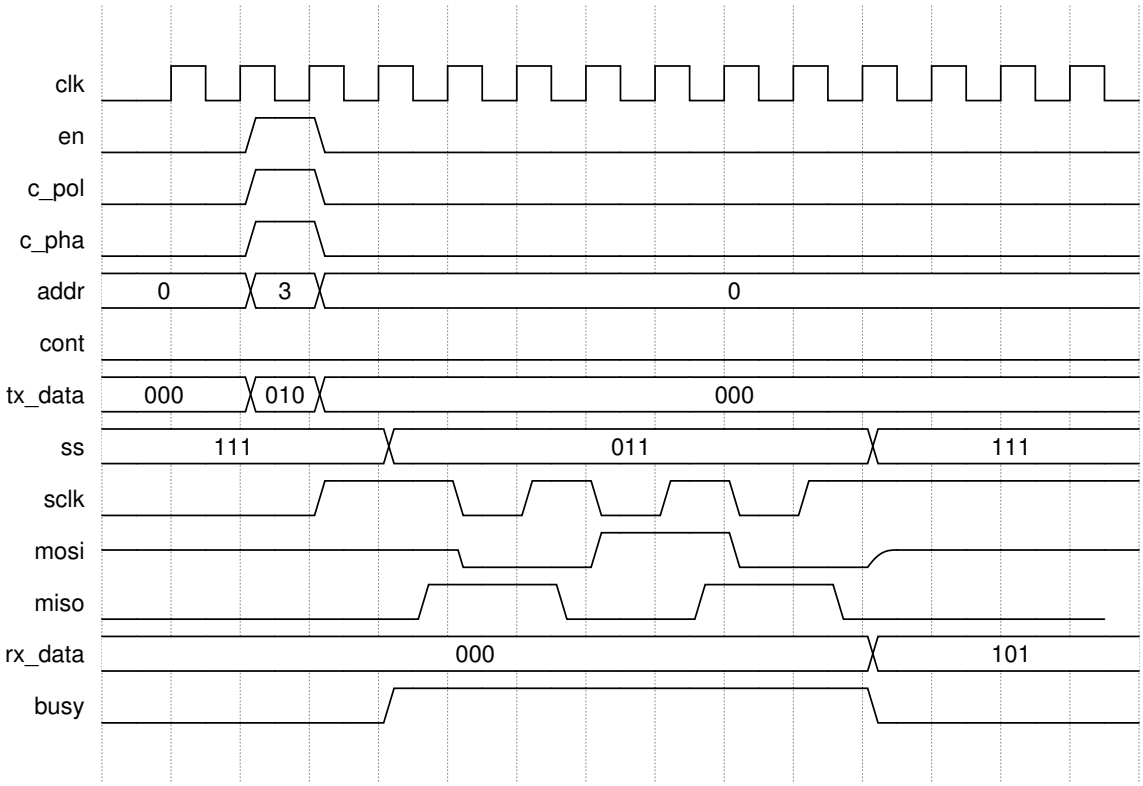


Figure 4.5: SPI transaction waveform

The waveform above shows an example transaction using a module which supports 3 slaves and uses 3 bit data-width. It transfers the *010* sequence to the third slave, while simultaneously receiving the *101* sequence. The clock polarity and phase settings are both in logic 1.

Reset and initial state

With the *rst* signal, it is possible to asynchronously set the module into its initial state. It can be triggered at any time and it causes the module to stop the current operation immediately and set the *busy* signal to logic 1 until the *rst* goes back to logic 0.

In the initial state all *ss* outputs are set back to logic 1, the *mosi* output is set into high impedance and the *rx_data* is cleared.

4.2 Implementation

Before writing a test, it is necessary to create an empty top level entity for the module using the port descriptions and a testbench with only clock generation. To support the N-bit data-width and M number of slaves, VHDL generics were used together with a custom base 2 logarithm function to determine the width of the *addr* bus. During the tests, their default values (N=8, M=3) were used to avoid complications when instantiating a VHDL module in SystemVerilog.

Reset and initial state

- When the reset signal is active, the busy should be active too.
- After the reset goes back to logic 0, the initial settings of the module should be set.

These two simple facts were transformed into assertions in the newly created testbench. For further operations it is necessary to reset the module at start, so the first triggering of the *rst* was added into the initialization part of the testbench. In the following sections, these extensions of the initialization part will not be mentioned. It should be understood that every possible combination, that the suitable test needs has been implicitly added.

```
property initial_state;
       rst [*1:$] ##1 !rst |=> accept_on(rst)
       (ss==3'b111) && (rx_data==8'h00) && (mosi===1'bz) && !busy;
endproperty;
always @(posedge clk) assert property(initial_state);
```

Listing 4.1: Assertion for testing the initial state

Because the reset signal is not handled, running these tests will fail. Hereby allowing to continue with the next step of the TDD methodology, by implementing the code for passing the tests.

```
init : process (rst) begin
       if (rst = '1') then
              busy <= '1';
              ss <= (others => '1');
              mosi <= 'Z';
              rx_data <= (others => '0');
       else
              busy <= '0';
       end if;
end process;
```

Listing 4.2: The first lines of code after the failing tests

When the tests successfully pass, the implementation is ready and new tests can be added. This is done only in the first TDD iteration, because the added code (from the perspective of the two tests) is clean and does not contain any duplications. Refactoring will be made later, after adding more tests.

Clock generation

- ~~When the reset signal is active, the busy should be active too.~~
- ~~After the reset goes back to logic 0, the initial settings of the module should be set.~~
- After an enable sequence, the SPI clock has to be initialized with the correct polarity.
- The SPI clock should oscillate for N ticks, twice as slow than the *clk*.

To pass the third test, only one line has to be added (sclk <= c_pol), but refactoring will be needed. Passing the fourth test needs a finite-state machine with four states. The starting *idle* state latches the *c_pol* setting and waits for the enable signal. The enable signal activates the *start* state, where the *sclk* signal is set to the aprorpiate clock polarity. The last two states are responsible for oscillating the clock signal. When the number of clock ticks equals to the data-width of the module, the FSM returns to the *idle* state.

```
property sclk_generation;
       reg polarity;
       (en, polarity=c_pol) ##1 !en |=> accept_on(rst)
       (sclk == polarity ##1 sclk != polarity) [*8] ##1 sclk == polarity;
endproperty;
always @(posedge clk) assert property(sclk_generation);
```

Listing 4.3: The test of the spi clock's oscillation

This FSM is not the representation of the minimal amount of code, but is the product of the refactoring.

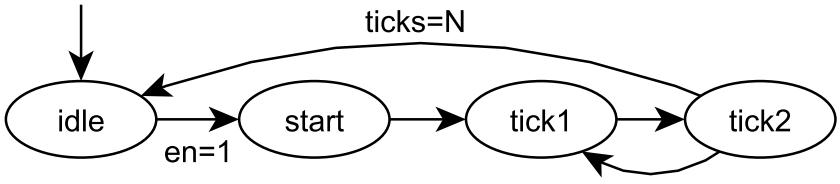


Figure 4.6: SPI clock generation FSM

Data transfer

- ...
- The busy signal has to be in logic 1 for N ticks after the enable sequence.
- The correct slave has to be selected, but only when the transaction was initiated and the busy signal is high.
- The bits of *tx_data* should appear on the *mosi*, one-by-one, then it has to return to high-impedance state.
- When the busy signal goes low after the transaction, the *rx_data* should contain the received bits from *miso*.

The tests described above were first made only for $c\_pha = 0$. The *busy* indicator was added into the appropriate states of the FSM, together with the slave selection which was desribed with a $\log_2 M$ to M decoder.

```
property tx_transaction;
       reg[0:7] tx;
       int unsigned i;
       (en, tx=tx_data, i=0) ##1 !en |=> accept_on(rst)
       (tx[i]==mosi ##1 tx[i++]==mosi) [*8] ##1 mosi===1'bz;
endproperty;
always @(posedge clk) assert property(tx_transaction);
```

Listing 4.4: Validation of the transmitted data

The data sending and receiving were integrated into the *tick1* and *tick2* states. There was already a counter for counting the number of *sclk* ticks, so it was used to select the appropriate byte to send.

Phase selection

To support the $c\_pha = 1$ mode, the original transaction tests were duplicated and reclocked, so finally four tests cover the whole transaction system of the module. The implementation of these functionalities was made by adding some extra states to the existing FSM which was very complicated, but it passed all the tests.

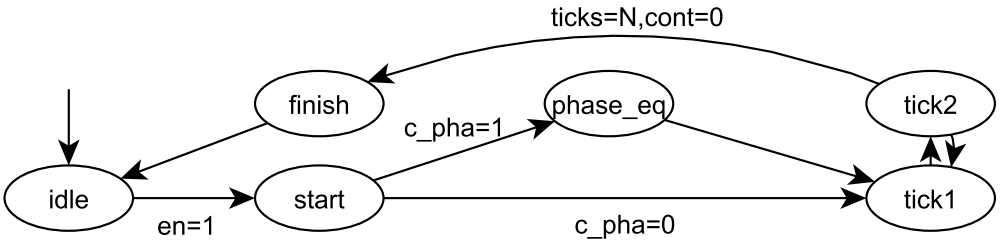


Figure 4.7: FSM which supports both clock phases

Final refactoring

The *idle* and *start* states are necessary, they cannot be eliminated, but the other five states could be joined into one *work* state by extending the range and the purpose of the *cnt*.

In this newly added state, the counter counts from zero and it generates the *sclk* signal. When the value of the *clk* variable equals zero, the appropriate slave is selected, the *busy* signal is set to logic 1 and when the clock's polarity is set to zero, the first bit is sent out through the *mosi* wire. Further writing and reading is solved by checking the counter's value. Whether or not a reaction is needed, depends on the clock polarity selection too. When the counter reaches the final value, the machine sends the received N-bytes to the *rx_data* and returns to the *idle* state.

```
if cnt=0 then
        if phase='0' then
                mosi <= tx(N-1);
        end if;
elsif cnt mod 2 = 0 then
        if phase='0' then
                mosi <= tx(N-1-cnt/2);
        else
                rx_data(N-cnt/2) <= miso;
        end if;
else
        if phase='0' then
                rx_data(N-1-cnt/2) <= miso;
        else
                mosi <= tx(N-1-cnt/2);
        end if;
end if;
```

Listing 4.5: Transaction handling with even-odd-zero checking

4.3 Evaluation

After the final refactoring of the code was synthesizable and it passed all the tests. Compared with the reference design, they were both communication interfaces with easily available and well documented specifications. However, the development of the SPI interface was much faster, because writing the tests before implementation causes that the developer pays more attention to the specifications and memorizes them better.

It can be stated that the use of SystemVerilog assertions is the best available solution yet for test-driven hardware development. Using them for very large projects with many tests but can be very difficult.

Running a simulation can sometimes take hours or more and in TDD, it is a very frequent operation. When the simulation of a module takes too much time, it has to be decomposed and the sub-modules should simulated separately. It does not decreases the simulation time, because the design remains the same, but it reduces the time of waiting between the steps of TDD.

The most difficult part of writing the testbenches is the timing, especially when using concurrent assertions or assertions in separate processes. For having good tests, it is necessary to add every possible combination of input signals into the tectbench's stimulus process and keep them in accordance with the assertions. Seeing through two types of code-structure which are placed in two different places is not an easy task for any developer.

Chapter 5

Conclusion

> A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away.
>
> (*Antoine de Saint-Exupéry*)

It is proved that the test-driven method highly facilitates the software development process. The red-green-refactor model always proves to be the most simplest and cleanest code which is an important quality factor of a software product. Saint-Exupéry's quote expresses the same idea: if one cannot simplify the code any further so that its functionality remains intact, then perfection has been achieved and the product is finished.

From the previous chapters is evident that test-driven development methodology has a place in the toolbox of a hardware developer as well. It helps the developer to understand the specifications deeper and forces him to write high quality tests. It does not matter when the tests are written, it takes the same amount of time to create them, so the metodology at the worst case is neither slower, than the classical hardware development.

It is very important, that the methodology is not applicable one-to-one, because of the differences between software testing and hardware simulation. Creating a test to verify a function which calculates e.g. the square root of a number is much easier, than simulating an entity what does the same. However, the basic idea is the same, just the tests are grouped into testbenches and instead of testing the operation it is called simulation.

To use TDD in larger projects, a complete and generally available unit-test, or at least, a test-automation framework is necessary. The continuation of the present paper in the future could be an object-oriented, xUnit based hardware verification and unit-test framework. Since the now available similar tools does not support PSL-like assertions and they were very useful in the demonstrational project, this future framework should support them as well. To make refactoring faster, the simulation environment should be extended with automated running of synthesis.

Bibliography

[1] *A Guide to the Project Management Body of Knowledge*. Project Management Institute, 4th edition, November 2009. ISBN 978-1933890517.

[2] Jasper Design Automation. Property specification language. http://oskitech.com/papers/psl-ucb0405.pdf, 2005. [Online], [cit. 2013-04-25].

[3] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, November 2002. ISBN 978-0321146533.

[4] J. Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Springer, second edition, 2003. ISBN 978-1-4020-7401-1.

[5] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[6] Dorota Huizinga and Adam Kolawa. *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society Press, October 2007. ISBN 978-0470042120.

[7] Motorola Inc. Spi block guide v03.06. http://www.ee.nmt.edu/~teare/ee308l/datasheets/S12SPIV3.pdf, Jan 2000. [Online], [rev. 2003-02-04], [cit. 2013-04-25].

[8] Scott Larson. Serial peripheral interface (spi) master (vhdl). http://www.eewiki.net/display/LOGIC/Serial+Peripheral+Interface+(SPI)+Master+(VHDL). [Online], [rev. 2013-04-11], [cit. 2013-04-25].

[9] Bryan Morris and Rob Saxe. svunit: Bringing agile methods into functional verification. Technical report, XtremeEDA, Ottawa, Canada, 2009.

[10] Alok Shanghavi. What is formal verification? *EE Times-Asia*.

[11] Marcela Šimková. Hardware accelerated functional verification. diplomová práce, FIT VUT v Brně, Brno, 2011.

Appendix A

Contents of CD

- **/demonstration** - directory containing the source files of the demonstration module
- **/latex** - directory containing the LaTeX and other source files for creating the thesis
- **/pdf** - directory containing the PDF version of the thesis
- **/examples** - directory containing the source files of the assertion and unit-test examples
  - **/examples/reference** - directory containing the source files of the reference design
  - **/examples/vhdl** - directory containing the VHDL assertion examples
  - **/examples/systemverilog** - directory containing SystemVerilog assertion examples
  - **/examples/psl** - directory containing the the PSL assertion examples

Appendix B

Manual

To run the simulations, ModelSim/QuestaSim simulation environment is necessary. Unfortunately ModelSim does not support SystemVerilog assertions, so the demonstrational module and the SystemVerilog assertion examples could not be run by using it.

In each directory are files with *.fdo* extension which have to be started from the simulator's command line. Every testbench generates an assertion report into a *.log* file in his own directory.

The reference and the demonstrational hardware designs both can be synthesized in Xilinx ISE, by importing all the *.vhd* files to an empty project.