Insert here your thesis' task.

Czech Technical University in Prague
Faculty of Information Technology
Department of Digital design

Master's thesis

Design of a verification environment for a smart sensor

Bc. Ivo Háleček

Supervisor: Ing. Jakub Šťastný Ph.D.

29th April 2015

Acknowledgements

I wish to express my sincere thanks to all of the ASICentrum s.r.o. for their help and support. I am also grateful to my family, which supported me through this venture.

Declaration

In Prague on 29th April 2015 . . . .

Citation of this thesis

Háleček, Ivo. *Design of a verification environment for a smart sensor.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2015.

Abstrakt

Práce je zaměřena na návrh a implementaci verifikačního prostředí pro inteligentní senzor. Inteligentní senzor, podle obecně uznávaných průmyslových definic, spojuje měřící prvek, analogově číslicový převodník a sběrnici pro komunikaci.

Práce byla složena ze tří hlavních částí. První část představovala nastudovat literaturu za účelem seznámení se s moderními postupy pro verifikaci. Druhou částí bylo navrhnout a sestavit model inteligentního senzoru, který bude sloužit jako design-under-test pro verifikaci prostředí. Třetí částí bylo implementovat a zverifikovat testbench.

Hlavním výstupem práce je testbench vhodný pro verifikaci inteligentního senzoru, implementovaný pomocí Unified Verification Methodology (UVM) a SystemVerilogu. Testbench byl zverifikován pomocí simulátoru a bylo sledováno pokrytí pro sledování postupu verifikace.

Klíčová slova inteligentní senzor, SystemVerilog, UVM, verifikace, testbench, SPI, I2C

Abstract

This Master's thesis is focused on smart sensor verification environment implementation. A smart sensor, according to generally accepted industry definitions, combines a sensing element, an analog-to-digital converter (ADC), and a bus interface.

The work of this thesis was divided into three tasks. The first task was to study literature to get to know basics of modern verification approaches. The second task was to design and build a smart sensor model, which would serve as design under test for the testbench verification. The third task was to implement and verify the testbench.

The main output of this thesis is a testbench suitable for smart sensor verification, implemented using Unified Verification Methodology (UVM) and SystemVerilog. The testbench has been verified in simulator and coverage metrics have been collected during simulation to track the progress of verification.

Keywords smart sensor, SystemVerilog, UVM, verification, testbench, SPI, I2C

Contents

List of Figures

Introduction

The main topic of this thesis is implementation of verification environment for smart sensor.

As complexity of digital designs grows, system verification is more and more important. Bug revealed before tape-out is usually ten times cheaper to repair than after tape-out and even hundred times cheaper to repair than if revealed after product is delivered to customers. The more complex the design is, the more stimuli needs to be simulated. Nowadays verification process usually takes even more effort than the design alone. Therefore several approaches have been introduced to speed up verification process.

The Chapter 1 brings summary on verification methods commonly used and serves on theoretical basis for the thesis. First part of the chapter is dedicated to Coverage, followed by Verification approaches summary.

The Chapter 2 sets the goals of the thesis and briefly describes why or how to achieve them.

Analysis and design is documented in the Chapter 3. This chapter described what smart sensor is, and how to model smart sensor for verification purposes. Second part of this chapter is dedicated to verification framework selection. SystemVerilog and UVM have been selected.

System level design is described in the Chapter 4. Testbench follows classic UVM block-level testbench hierarchy. Steps needed to build smart sensor model are also described at the end of this chapter.

The Chapter 5 is dedicated to SPI UVM verification component. SPI agent supports typical smart sensor transactions, with configurable clock phase and clock polarity configuration, error injection, SPI slave timing checks, and configurable SPI master timing.

The Chapter 6 is dedicated to I2C UVM verification component. I2C agent supports typical smart sensor transactions, error injection, configurable timing and *sda* timing checks.

CHAPTER 1

State-of-the-art

This chapter gives a summary of commonly used techniques in digital design verification process.

Verification is a process of checking that design functions correspond to the specification. As average number of gates in designs grows, the verification process is more and more complex. Therefore verification process consumes often much more effort than design itself.

Unless specification is written in formal language, verification process cannot prove that design meets the specification. Specification documents are written in natural language, which can be interpreted in many different ways. If verification of design is done by another engineer than author of design, those misinterpretations can be revealed.

1.1 Coverage

1.1.1 Code coverage

Code coverage measures how many and what parts of design source code have been executed during verification. This metric should evolve towards 100%. While it's an effective metric for small design units, which have separate specification, it is not suitable when verifying designs composed of sub-designs that have been independently verified. The objective of that verification is to confirm that the sub-designs are interfaced and cooperate properly, not to verify individual features.

Coverage lower than 100% indicates that part of design source code has not been executed by testcases. It can indicate that there is need of new test case, or a part of design source code is unreachable and can be removed. In order to have reusable design parts, some parts of source code can be unused due to verified system configuration, which leads to lower coverage, but does not mean problem.

Results from coverage should be used only to help identify corner cases that were not executed by testcases. Low code coverage indicates that multiple design features of design has not been executed by verification environment and therefore verification is not complete. On the other hand, 100% coverage does not mean that design is verified. It only means all coverage points defined in simulation have been covered. Execution of design source code does not imply design behaves correctly [6].

Code coverage metrics are supported by verification tools and are collected automatically if enabled.

Statement coverage measures how much of the total lines of code have been executed by verification suite.

Path coverage measures all possible ways a sequence of statements can be executed. Every branching in the code adds new paths. 100 % statement coverage does not imply 100 % path coverage. All statements can be executed without executing every branch. It is always necessary to determine conditions which cause the path to be uncovered. Sometimes it can be even impossible to get 100 % coverage, because taking one branch can imply taking another one later in the code.

Number of possible paths grows exponentially with every code branching. Therefore it is good to keep sequential code lower than 100 lines.

Expression coverage measures various ways paths through the code are executed. One code branching can be affected by multiple conditions. As in the case of statement coverage, it is necessary to determine why control expression has not been executed and if such a condition can even occur.

FSM coverage measures all possible transitions in finite state machine (FSM). Any unvisited state should be already identifiable from statement coverage, because FSMs are usually coded using case statements. Some unreachable states can also be created by synthesis process. FSM having five states will probably be encoded to three bits, leaving three unused combinations. Verification engineer should also check what happens if design accidentally reaches one of those unreachable states, for example due to magnetic field effect.

1.1.2 Toggle coverage

Toggle coverage measures which signals changed its value from 0 to 1 and back. Signal is considered covered if it toggled back and forth at least once. This metric does not indicate that every value of register has been set. If a 4bit register changes its value from 0000 to 1111 then all bits have changed but only two combination have been set. However, this metric tells if all bits of vector has been toggled back and forth.

1.1.3 Functional coverage

Functional coverage measures which combinations of inputs and outputs have been verified and what internal states have been visited. By assigning weight to each functional coverage metric, it can be reduced to one single value measuring how much of functionality has been exercised. Giving higher weight to more important functional coverage, this value can indicate progress of the verification. This metric should raise rapidly toward 100 % at the beginning, then progress usually significantly slows down while only hard-to-reach functional coverage points remain.

1.2 Verification approaches

There are two basic approaches in verification process:

1.2.1 Formal verification

Formal verification uses formal methods of mathematics to prove correctness of circuits with respect to certain specification or property. More information on formal verification can be found in [7].

1.2.2 Functional verification

Functional verification is a process with goal of ensuring that a design implements desired functionality. As shown in Figure 1.1, functional verification compares design and its specification.

Functional verification is performed by simulation. It is process of stimulating design inputs with predefined input sequence and analyzing the output.

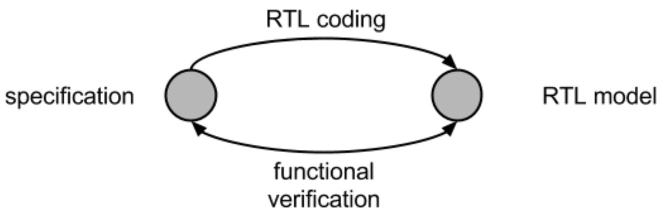

Figure 1.1: Functional verification

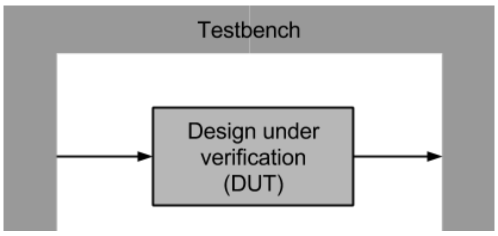

Figure 1.2: Testbench

Simulation code used to perform functional verification is usually referred as Testbench. The testbench is a verification environment, it is connected to design under verification ports (see Figure 1.2). Design inputs are stimulated and outputs are monitored according to sequences defined by testcases. Testcase is description for Testbench how to verify one set of related features.

It is however impossible for functional verification to prove that design is bug free. It can only prove that design has a bug. Number of input combinations of combinational circuit is 2^n, where n is number of inputs. For sequential circuits also number of internal states must be taken into account. Number of steps to cover all possible situations is 2^{m+n}, where n is number of inputs and m number of internal states. Therefore it is impossible to cover and verify all possible situations which can happen, in reasonable time.

Therefore verification is a process that can never be 100% finished. Verification is usually ended when the coverage goal has been reached and critical functions have been verified.

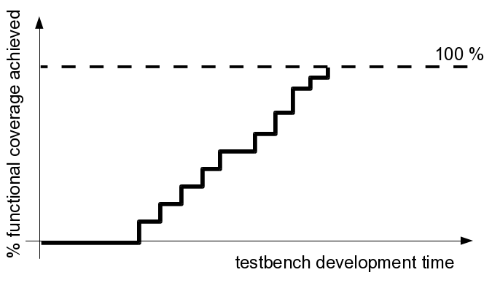

Figure 1.3: Directed testbench

1.2.3 Directed approach

With directed testbenches, every feature is checked using individual simulation code. All the stimulus are generated manually. While directed testbench is suitable for aiming to specific feature verification, it lacks scalability and verification of large design will consume too much effort if done only by directed testbench.

As seen in Figure 1.3, coverage does not show increase in the beginning, because verification environment is being developed. Then progress follows as more and more testcases are covered by testbenches.

Completion of coverage of initial testcases does not necessarily mean that verification is over. Other metrics can indicate that original testcases are not as deep as imagined and additional testcases needs to be created.

1.2.4 Randomized constrained testbench

In order to increase scalability of verification process and to decrease effort needed, constrained testbench can be used. Constrained testbenches use randomization. That does not mean all inputs are stimulated with completely random bit combinations. Random generators are constrained by constraints to avoid generating lots of invalid combinations (eg. different data packets to be sent through bus while no transaction will be performed).

As seen in Figure 1.4, initial effort is spent on verification environment development and constraints definitions, but then coverage raises rapidly compared to directed testbench. After some time coverage increase is not as steep, so constraints are updated to verify uncovered features. In the end, only few more coverage points are not covered, which can be finished by directed testcases.

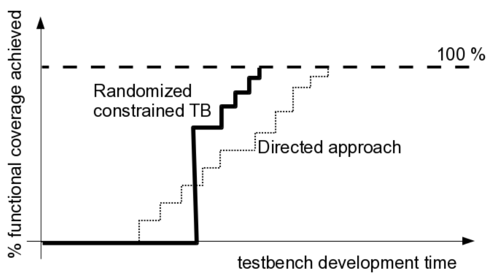

Figure 1.4: Constrained testbench

1.3 Verification plan

Verification plan is a document, which specifies what and how will be verified, who is responsible for individual parts verification and when verification process will be finished. Plan contains detailed goals using measurable metrics, along with optimal resource usage and realistic schedule estimates. Purpose of the document is to review and discuss verification strategy before verification itself is started and to serve on basis for planning.

1.4 Verification tools

1.4.1 Simulator

Functional verification is done in simulator. Simulation is always performed on a simplified model of verified system. Proper level of abstraction has to be selected with respect to the verified feature. If digital part of system is verified, we can abstract from electrical properties of logical parts. We do not need to care of exact voltage on a wire, we just need to know if it is in logical state high or low. This will save time needed for simulation for the price of not simulating features at lower abstraction levels, which can however be verified at lower level of abstraction.

System level simulation verifies that system blocks communicates correctly at high abstraction level. Communication is usually represented by transactions.

Register transfer layer (RTL) simulation verifies synchronous system in terms of flow between hardware registers and logical functions performed on signals [8].

Gate level simulation (GLS) is performed at lower level of abstraction (with standard library cells as basic units), after synthesis with respect to gate-level issues. GLS have higher memory and performance requirements, therefore it is necessary not to waste simulation cycles on features, that can be verified at higher abstraction level [9].

Transistor level simulation is performed at low level of abstraction with transistors being basic unit.

1.4.2 Verification languages

Functional verification is usually done using either hardware description language (HDL - VHDL, Verilog, . . .) or hardware verification language (HVL - OpenVera, e, SystemC, ...). When performing verification using directed testbench, HDL can be used with the advantage of having the same language for design and verification. When using constrained testbench classical HDL lacks support for randomization. Note that randomization requirements for constrained verification are not just to have a random generator, but to have advanced constrained generators, which are hard to implement. Hardware verification languages traditionally do have constrained randomization support, but it is new language to learn.

One of the modern verification languages is **SystemVerilog**, sometimes referred as HDVL, because it is based on HDL Verilog, but it implements support for functional verification and high level design. Learning SystemVerilog is easy especially for engineers already familiar with Verilog. Using the same language for design and verification have also advantage of better access to design and no need of special interfaces [10].

There are also frameworks based on SystemVerilog making verification even more simple (UVM, OVM, VMM), which are discussed in the Section 1.4.3.

Apart from standard verification languages, also **property specification language** (PSL) can be used. PSL is language describing boolean (those which can be true or false) properties of design. PSL can be used with VHDL or Verilog design. Properties are used to create assertions, which are checked by simulator. PSL can be for example used by monitors to check state of design during simulation, for functional coverage or to define legal sequences of input vectors for simulation [11].

1.4.3 Frameworks for verification

Complex designs require lots of effort to be spent on verification, therefore some frameworks have been introduced to help engineers develop powerful test environment faster. The most known ones are VMM, OVM and UVM.

Verification Methodology Manual (VMM) was the first successful set of practices for creation reusable verification environments, based on SystemVerilog, created by Synopsys. VMM uses advantages of object-oriented programming, constrained randomization and functional coverage. VMM contributed as source of inspiration when UVM was created [12].

Open Verification Methodology (OVM) is the library of objects and functions for creation of constrained testbenches, collecting functional coverage and transaction based stimulating of inputs. This was the first library available for SystemVerilog on multiple simulators. OVM contributed significantly to development of it's successor, Universal Verification Methodology.

Unified Verification Methodology (UVM) is modern open source verification library based on OVM and VMM, available for SystemVerilog. It's aimed for creation of flexible, reusable verification components and assembly of test environments with functionality for constrained randomization stimulus and functional coverage. UVM is a combined effort of designers and tool vendors, based on the successful OVM and VMM methodologies [13].

Chapter 2

Goals of thesis

The goal of the thesis is to implement a verification environment for a smart sensor. This includes initial analysis of possible solutions, model of a smart sensor, implementation itself, and verification. Expected output of the thesis is:

- *Verification approaches analysis.* Several books have been written to the topic of verification. Analysis of possible solutions will serve as basis for the testbench design.
- *Smart sensor model.* Verification of testbench will be done against smart sensor model used as design under verification.
- *Testbench for a smart sensor.*
- *Verification plan.* The testbench itself must be verified. Verification plan defines will serve as basis for the testbench verification.
- *Testbench verification report.* Verification report includes functional coverage report and conclusion of verification.

Chapter 3

Analysis and design

This chapter covers my effort on analysis and design of smart sensor verification environment, which had to be done before implementation. An example of smart sensor is given in Section 3.1, followed by specification of smart sensor model needed to verify verification environment.

Verification tools and framework selection are discussed in Section 1.4.

3.1 Smart sensor

A smart sensor, according to generally accepted industry definitions, combines a sensing element, analog interface circuit, an analog-to-digital converter (ADC), and a bus interface, all in one housing. Making the grade against the newest generation of smart sensors, however, means that additional functionality must be included, such as self-testing, self-identification, self-validation, or self-adaptation. Of particular interest and importance to designers are such smart sensor capabilities as self-calibration and self-diagnosis, the ability to use signal processing, and multi-sensing capabilities [14].

An example of sensor (ADIS 16203) can be seen in Figure 3.1. It contains some sensing elements with analog-digital converters, data processing units, self-test, register map, and SPI Slave. Smart sensor model will be used as DUT in verification environment to verify environment itself and shall be capable of reading value from ADC converter, process received data, and send it through one of the buses connected.

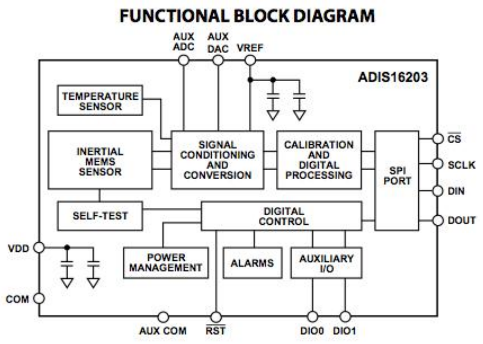

Figure 3.1: ADIS 16203 smart sensor

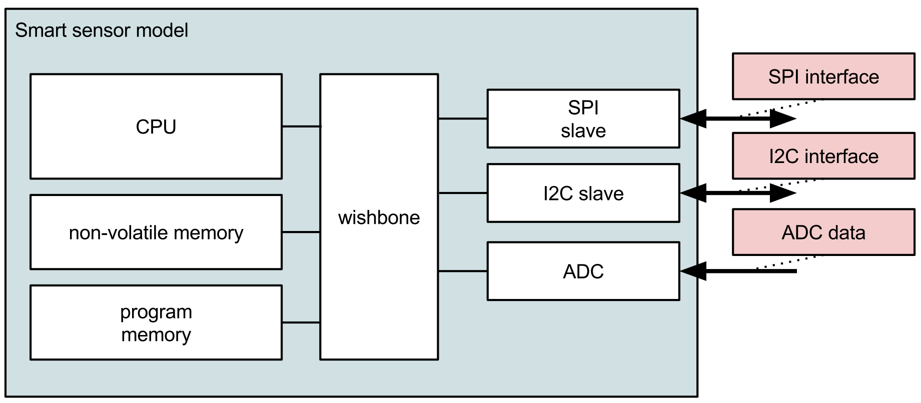

Figure 3.2: Sensor block model

As smart sensor design is not the goal of this thesis, model should use as much available cores as possible, and more effort should be spent on verification environment itself. A processor can be used as data processing unit with advantage of possibility to define behavior in C instead of HDL, which will result in more flexible model and simplified implementation. Behavior, written in C can be compiled and loaded as boot image. A good resource of open-source cores is OpenCores website [15].

Block diagram of the smart sensor model can be seen in the Figure 3.2. The smart sensor is composed of processor, SPI and I2C slave cores, non-volatile memory and an interface to controll ADC. The sensor interface is described in the Table 3.1.

Table 3.1: Smart sensor interface

Interface	Signal	Type	Description
CLK	clk	logic	Clock
	reset	logic	Reset (active high)
SPI	csb	logic	Chip select bar (active low)
	sck	logic	Serial clock
	miso	logic	Master in, slave out
	mosi	logic	Master out, slave in
I2C	scl	logic	Serial clock
	sda	logic	Serial data
ADC	clk	logic	Clock
	start	logic	Start measurement
	data	logic[7:0]	Measured value
	ready	logic	Measurement complete

3.1.1 Processor

I decided to use OpenRISC 1000 processor, because of consistent documentation and because it's developed and used by big community, so potential problems should be simpler to solve.

I could however choose from a few implementations of OpenRISC 1000 processor.

The OR1200 is a 32-bit scalar RISC with Harvard micro-architecture, 5 stage integer pipeline, virtual memory support (MMU) and basic DSP capabilities.[16]

The OR10 core implements the complete OpenRISC ORBIS32 instruction set. It has a very simple design, one could say suboptimal, or even naive. It does not implement the jump delay slot. It has a single Wishbone bus for both instruction fetches and data access.

There are some other implementations of OpenRisc 1000 family, but they does not seem to be finished or does not differ from OR1200 in a way which is relevant to this project.

I prefer to use OR10 processor as it is simple, source is well documented and although it is not as powerful as OR1200, performance is not important for this purpose.

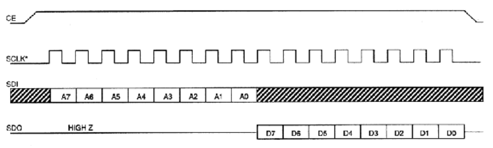

Figure 3.3: DS1722 read transaction [1]

3.1.2 SPI slave core

The OpenCores website [15] offers in time of writing this thesis only one SPI slave core [17], which suits license requirements and implementation is done. However this core is not wishbone compliant, so interface unit will have to be developed.

As smart sensors usually contain multiple registers, also register address has to be transferred. DS1722 datasheet [1] describes how register address and data are transferred through SPI in this sensor.

This sensor supports single and multiple read and write operations in half-duplex mode. Timing diagram for single byte read can be seen in Figure 3.3. Other operations run similarly. Multi-byte transfer starts with starting register address and data, for every following data byte target/source register address is increased by one.

3.1.3 I2C slave core

Three stable I2C cores, supporting slave mode, can be found on the OpenCores website [18] [19] [20]. I selected [20], because it is the only one which is wishbone compliant.

Digital thermometer sensor - LM75 [2] features I2C interface for register map access. As seen in Figure 3.4, this sensor uses three least significant slave address bits as register map address. This register map addressing cuts down number of other connectible slave devices, so I decided to use similar addressing as in case of SPI.

3.1.4 ADC design

Analog-to-digital converter (ADC) is a device that converts a continuous physical quantity (usually voltage) to a digital number that represents the quantity's amplitude [21].

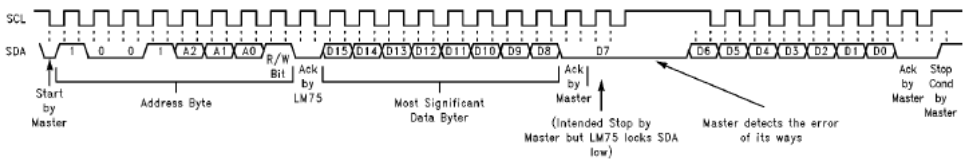

Figure 3.4: LM75 i2c transaction [2]

Table 3.2: ADC registers

Address	Type	Name	Description
0x0	RO	adc_meas	Last measured value
0x4	RW	adc_cnt	Control register
0x8	RO	adc_st	Status register

Table 3.3: adc_cnt register

7	6	5	4	3	2	1	0
0	0	0	0	irq_en	auto_start	start	0

I did not find any ADC wishbone compliant model, so I had to design my own. Ordinary ADC interface used in designs includes control signal to start adc measurement, signal telling that ADC measurement has finished, data bus, clock, and analog input.

As the ADC will be connected to processor via wishbone, behavior will be controlled by ADC 8 bit registers (described in the Table 3.2) accessible via wishbone. End of measurement will be signaled by processor interrupt.

- *irq_en* - Interrupt request enable. When set to 1, interrupt request is generated when adc measurement is finished.
- *auto_start* - ADC measurement auto start. When set to 1, ADC runs in continuous mode and new measurement is started right after previous one is finished.
- *start* - ADC measurement start. If this bit is set to 1, ADC performs measurement. This bit is set to the value of *auto_start* at the end of measurement.

Table 3.4: adc_st register

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	data_ready

- *data_ready* - ADC data ready. This bit is set 1 when ADC measurement is finished. If *irq_en* is 1, interrupt request is generated. When register *adc_meas* is read, *data_ready* and interrupt request is cleared.

3.1.5 Non-volatile memory

Non-volatile memory will introduce eight sensor registers accessable via SPI and I2C. Data will be persistent, will be loaded from memory file at reset and writen to file when after every write to the memory. As I could not find any suitable NVM model core available, I will implement it by myself.

Sensor registers will store information about last measured value, conversion constants and spi setup. Registers are described in the Table 3.5. Registers **sensor_info** and **free1** store user defined values and sensor does not use them. Register **cnt_reg** stores ADC control values, described in Section 3.1.4. **Status** register stores one-bit information whether an ADC value has been measured since reset (see the Table 3.6. Register **meas_val** contains measured ADC data converted according to constants stored in **meas_const** register. Two most significant bits of **meas_const** register defines number of bits ADC measured value will be shifted right. Six remaining bits defines additive constant which will be added to shifted ADC measured value. For details, please see the Table 3.7. Register **bus_setup** contains configuration related to bus peripherals. Detailed information about single bits can be found in the Table 3.8.

Table 3.5: Sensor reg map

Address	Type	Name	Description
0x00	RW	sensor_info	Info register
0x01	RW	cnt_reg	Control register
0x02	RO	adc_data	Raw ADC measured value
0x03	RO	status	Status Reg
0x04	RO	meas_val	Converted ADC value
0x05	RW	meas_const	Conversion constants
0x06	RW	free1	Free register
0x07	RW	bus_setup	SPI and I2C setup

Table 3.6: status register

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	measured

Table 3.7: measurements constants register

7	6	5	4	3	2	1	0
sh1	sh2	add5	add4	add3	add2	add1	add0

Table 3.8: bus setup register

7	6	5	4	3	2	1	0
0	0	0	0	0	0	spi_cphase	spi_cpol

3.1.6 Sensor behavior

Sensor behavior is implemented in C, which is compiled to boot image for OpenRISC, saved in memory model.

When a peripheral core (SPI slave, I2C slave, or ADC) has received data or needs new data to transmit, it generates interrupt request to OpenRISC processor. Bus interrupts (SPI and I2C), changes specific bus register address pointer and handles data transfer. The ADC interrupt handler is shown in Figure 3.5.

When ADC interrupts come, ADC measured data are converted using constants read from register *meas_const*. Detailed information about conversion was already provided in the Section 3.1.5. This simple conversion is performed to simulate real smart sensor behavior, like conversion of ADC raw data to temperature. Raw data are written to register *adc_data*, converted data to register *meas_val*. The least significant bit of register *status* is set 1. As this bit is cleared after reset, this bit can be used to signalize if at least one measurement has been performed since reset.

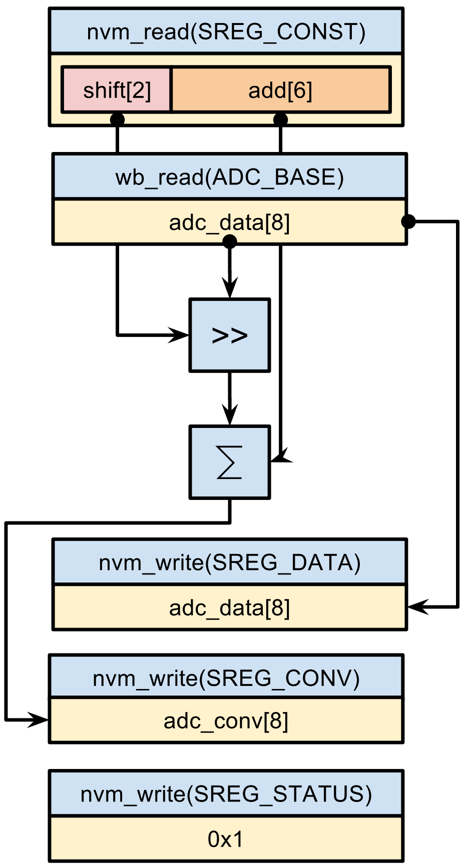

Figure 3.5: Sensor behavior loop

3.2 Framework selection

As mentioned in the Chapter 1.4, there are multiple hardware verification languages available. SystemVerilog provides functions for constrained randomization and together with one of verification methodologies introduced (VMM, OVM, and UVM) offers framework for reusable testbenches.

UVM is successor of OVM and VMM and therefore I selected it as most suitable methodology.

Chapter 4

System level design

This chapter describes the implementation of smart sensor verification environment.

4.1 Testbench components

UVM testbench hierarchy is described well in UVM Cookbook [3]. An example showing advised structure can be seen in Figure 4.1.

UVM verification environment for smart sensor based on hierarchy de-

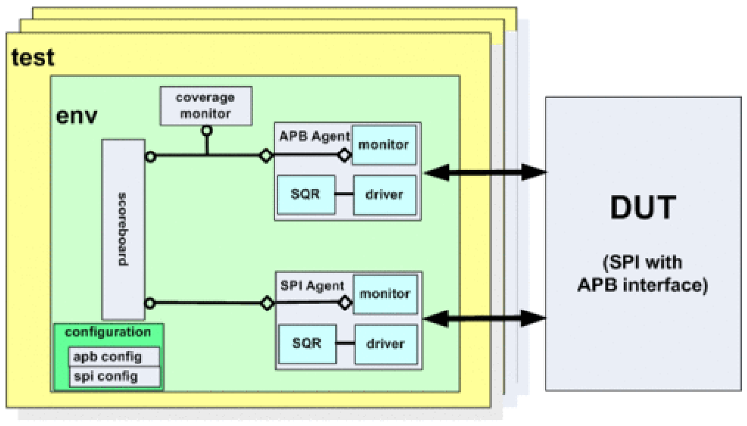

Figure 4.1: UVM Testbench block-level hierarchy example [3]

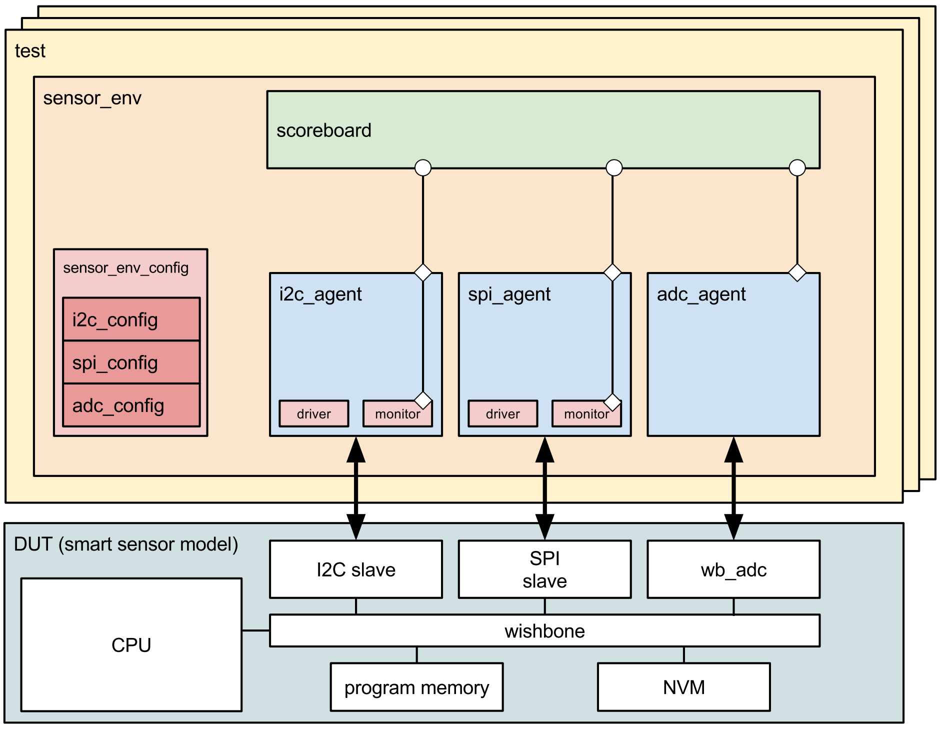

Figure 4.2: Smart sensor testbench

scribed in Figure 4.1 can be seen in Figure 4.2.

Testbench top encapsulates individual tests, interfaces and DUT. Tests contain verification environment, each test with individual environment configuration. Test may contain various agents depending on build configuration of the environment, virtual sequencer, and also a scoreboard.

4.2 Register transaction

As described in Section 3.1.2, sensor registers are accessed by register address, write bit and data (in case of write transaction). Register transaction represents read or write of one or more bytes to consecutive registers. Register transaction is represented by class *bus_seq_item*. Structure is represented in the Table 4.1.

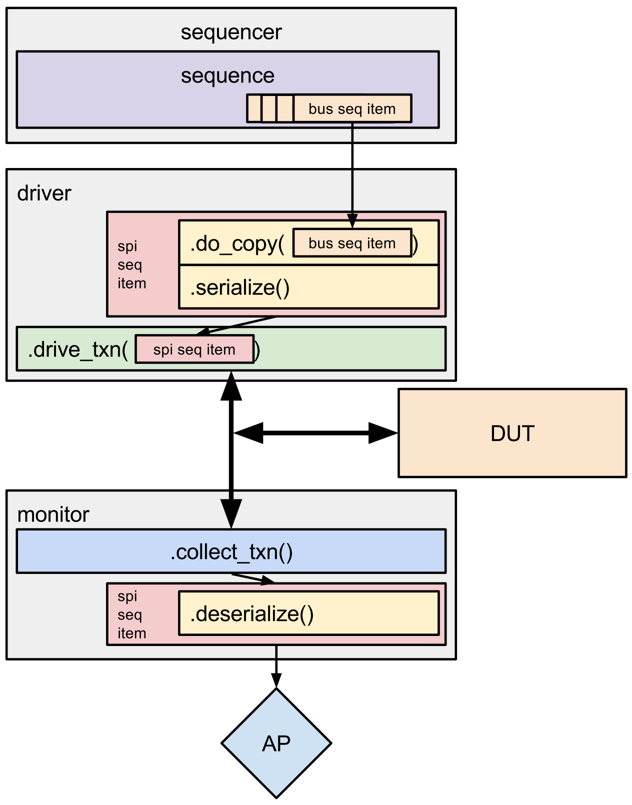

Figure 4.3: Register transaction usage diagram

4.2.1 Usage

Register transaction usage is shown in the Figure 4.3. Register transaction can be generated without knowing, which interface will be used. Once decision is made, it is sent to protocol-specific sequencer. Driver shall convert *bus_seq_item* to protocol-specific sequence item by calling its method *do_copy(bus_seq_item)*. Once converted, other protocol-specific properties can be set (eg. slave address for I2C). Driver shall call function *serialize*, which converts transaction properties to streams of bits to be send.

Monitor shall call function *deserialize* once transaction is collected, and send it to agent *analysis point*. Functions *serialize* and *deserialize* are empty in *bus_seq_item* and must be implemented by protocol-specific subclasses.

To convert this abstract transaction to protocol-specific transaction, functions *serialize* and *deserialize* are introduced. Every protocol-specific subclass must implement these functions.

Table 4.1: Register transaction

Name	Type	Randomized	Description
addr	logic[‘ADDR_LEN]	yes	Register address
data	logic[‘MAX_DATA_BITS]	yes	Data
data_length	integer	yes	Data length in bits
write	bit	yes	Transaction direction. 1 = master writes to slave.

Table 4.2: Register transaction functions

Function name	Param	Type	Description
to_string	return	string	Get printable form of transaction (addr, write bit and data).
reverse	return	void	Convert between LSB first and MSB first representations.
push_byte	return	void	Push data byte at the end of byte array.
	txn_byte	logic[7:0]	Data byte.
	bits_num	int	Number of valid bits in *txn_byte*.

- *serialize* - converts abstract transaction data members to protocol-specific bitstream. Driver should call this function before driving the transaction to corresponding bus.

- *deserialize* - converts protocol-specific bitstream to abstract transaction data members. Monitor should call this function after collecting the transaction from corresponding bus.

Error transaction can be generated by setting *data_length* not dividable by 8. Transaction process is interrupted after all valid bits are transferred.

4.2.2 Functions

List of functions implemented in bus_seq_item can be seen in the Table 4.2.

Table 4.3: Register transaction functions

Function name	Param	Type	Description
register_write	return	void	Write to register.
	addr	logic[]	Register address.
	data	logic[]	Data to be written.
	frc	bit	If set 1, read-only flag is ignored.
register_get	return	logic[]	Get register value.
	addr	logic[]	Register address.
register_check	return	logic[]	Check if data provided equals to register value.
	addr	logic[]	Register address.
	data	logic[]	Data to be check.

4.3 Scoreboard

The scoreboard is used to predict DUT's register content and check if transaction collected corresponds to expected register value. Registers mirroring DUT's readonly registers are writable only by ADC agent. Write operation from SPI or I2C analysis port is ignored. The scoreboard offers following register access functions described in the Table 4.3.

4.4 ADC UVC

The ADC UVM verification component is responsible of random ADC values generation and response to sensor ADC interface, described in the Section 3.1.4. ADC agent structure can be seen in the Figure 4.4. ADC measurement is started syncronously if no measurement is performed and *start* is high. After number of *clk* edges specified in Section 3.1.4, random value is generated to *data* and *ready* is set to 0.

4.4.1 ADC interface

ADC agent interface can be seen in the Table 4.4.

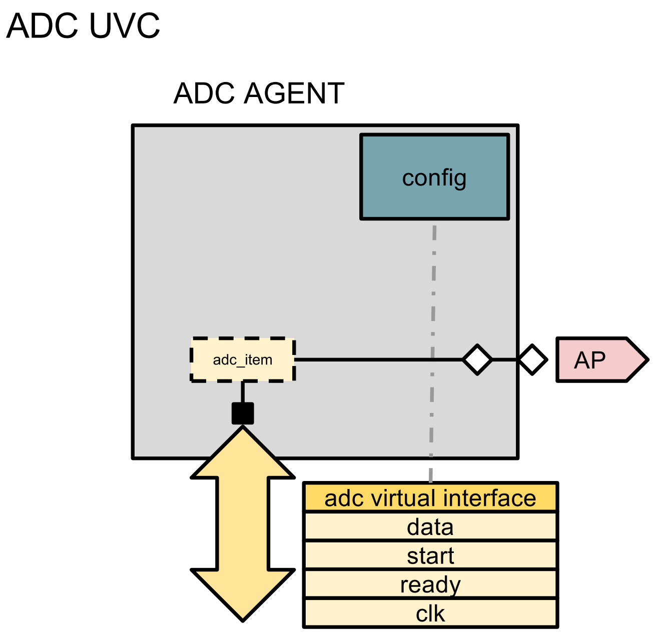

Figure 4.4: ADC UVC structure

Table 4.4: ADC Interface

Signal	Type	Description
clk	logic	Clock
start	logic	Start measumerent
data	logic[7:0]	Measured value
ready	logic	Measurement complete

Table 4.5: Run phase configuration

Parameter	Type	Default value	Description
meas_delay	int	50000	Measurement duration (from *start* rising edge to *ready* falling edge), specified in *clk* pulses.

4.4.2 Configuration

ADC agent configuration is stored in *adc_agent_config* class. Configuration (see Table 4.5) takes effect right after it is changed.

4.5 Design under test

In order to verify the testbench, design under test had to be build. Smart sensor model, designed in the Chapter 3, will be used as a design under test. Some changes and fixes had to be done to DUT components.

The OR10 implementation of OpenRISC processor already came up with wishbone interconnect module as well as with memory model capable of loading program from compilled C file. Also Ethernet module and JTAG module was included, but I did not need those, so I removed them.

I experienced some problems with or1k-elf-* toolchain as most of OpenRISC documentation reffered to older or32-* toolchain, which is not recommended to use anymore. As memory model attached to OR10 supported loading firmware as hexadecimal words written text format (one four-byte instruction per line), I had to write compile script which does following operations:

- compile C source without jump-delay slot support
- convert to binary format with or1k-elf-objcopy
- convert binary object to hexadecimal words written in text
- update firmware size in SystemVerilog compiler arguments file (dot f file)

Jump delay slot is an instruction slot executed after jump instruction regardless of preceding instruction. The common form of jump delay slot instruction is a single arbitrary instruction placed immediately after branch instruction, which gets executed even if preceding branch is taken. As OR10 does not support jump delay slots, firmware has to be compiled without jump delays, which means that nop instruction is inserted after every jump instruction.

4.5.1 Changes made to SPI core

The selected SPI core (selection discussed in the Section 3.1.2) was not wishbone compilant. I had to implement interface between wishbone and SPI core processor interface to be able to connect core to processor.

As SPI core only generates interrupt request once eight bits are transferred, I had to implement counter, which keeps track on number of bytes already transferred during transaction. Register address is always transferred in first byte of transaction, while data bytes are the other ones. This counter is reset by *csb* going high.

4.5.2 Changes made to I2C core

The selected I2C core (selection discussed in the Section 3.1.3) should have supported master and slave mode, but it did not operate correctly during read transaction in slave mode. During write transaction, I2C master tells the slave whether another byte is requested to be sent by setting acknowledge bit high or low. I2C slave always wanted to send another byte regardless of the acknowledge bit value. I had to fix this issue to make core work properly.

As one register access is performed by two I2C transactions (for details see the Section 3.1.3) and I2C core only generates interrupt request once nine bits (eight data bits and acknowledge bit, resp. seven slave address bits, write bit and acknowledge bit) are transferred, processor would be unaware if new data byte is transferred (or requested), or new register access has been started. Therefore I had to implement simple state machine in I2C core which keeps track on which byte (register address change, or data to be written / read) has been transferred.

4.5.3 Non-volatile memory

As mentioned in the Section 3.1.5, I implemented the NVM core. I selected VHDL as description language, as I have more experience with this HDL. This showed up to not be good decision as VHDL has poor functions to access files in filesystem. I had problem changing one single line in memory file after NVM write operation, so whole register map is saved to file after every write.

This is not such a big problem for this smart sensor model as it has only eight registers, but could significantly slow down simulation if used with much larger number of registers.

The address and data with are defined through generics as well as memory file. The memory file needs to exist and be accessible and has number of rows greater or equal to number of registers (2^w, where w is the address width). Each register content is stored in one line, written as sequence of text bit values. The first line contains content for address 0 and first character of every line reffers to most significant bit. For example 4-bit register with address 0x3 and value 0xA would be stored at 4th row as 1010 string.

Cadence offers tools for UVM regmap generation (iregGen). Register map is described in XML structure (IP-XACT), including read-only bits masks. IregGen can use this file to generate NVM model for DUT and also for Scoreboard. Unfortunatelly I could not implement this due to lack of time.

Chapter 5

SPI UVC

SPI UVM verification component (see Figure 5.1) verifies SPI interface. Sequence of transactions is received through sequence port, sequencer passes transactions to driver, driver generates SPI transaction. Monitor reads DUT responce from SPI interface, reconstructs transaction. Monitor transaction is sent through analysis port to scoreboard.

SPI UVC shall be capable of following features:

- SPI clock phase and polarity
- MSb or LSb first
- multibyte read
- error injection
  - unexpected end of transaction
  - 'x' driven to *mosi* out of setup and hold time
- check: *miso* stable during *csb* high
- *miso* timing checks

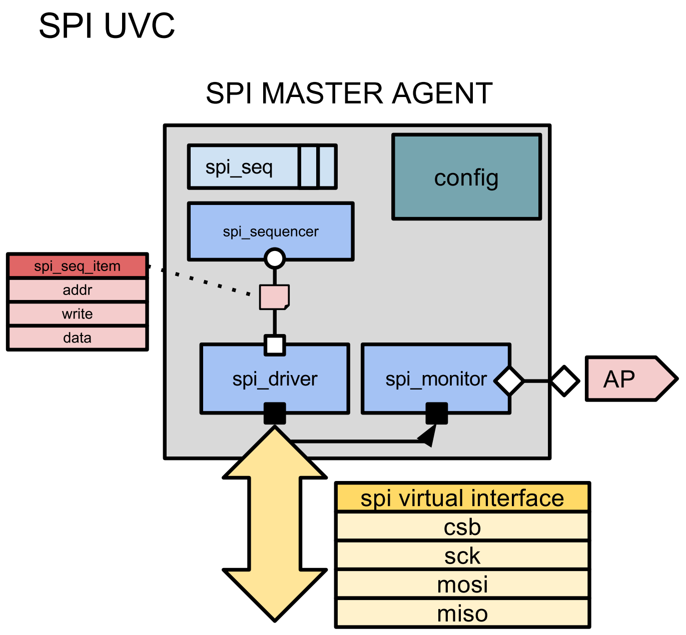

Figure 5.1: SPI UVC structure

5.1 SPI interface

SPI (Serial peripheral interface) is synchronous, master-slave serial communication interface developed by Motorola [22]. It has four wires, which are described in Table 5.1. Detailed transmission protocol can be seen in Figure 5.2. When *csb* goes low, slave is activated and transmission started. Data is sent through *miso* and *mosi* wires syncronously with *sck*, depending on *cphase* and *cpol* configuration.

- *CPOL* stands for clock polarity configuration. It selects which sck edge is deemed as active edge. If $cpol = 1$, active edge is low and in idle state sck is high. If $cpol = 0$, active edge is high and in idle state sck is low [23].
- *CPHA* (Clock phase) determines SPI clock format. If $cphase = 1$, sampling of data occurs at even $(2, 4, \dots)$ edges. If $cphase = 0$, sampling of data occurs at odd $(1, 3, \dots)$ edges [23].

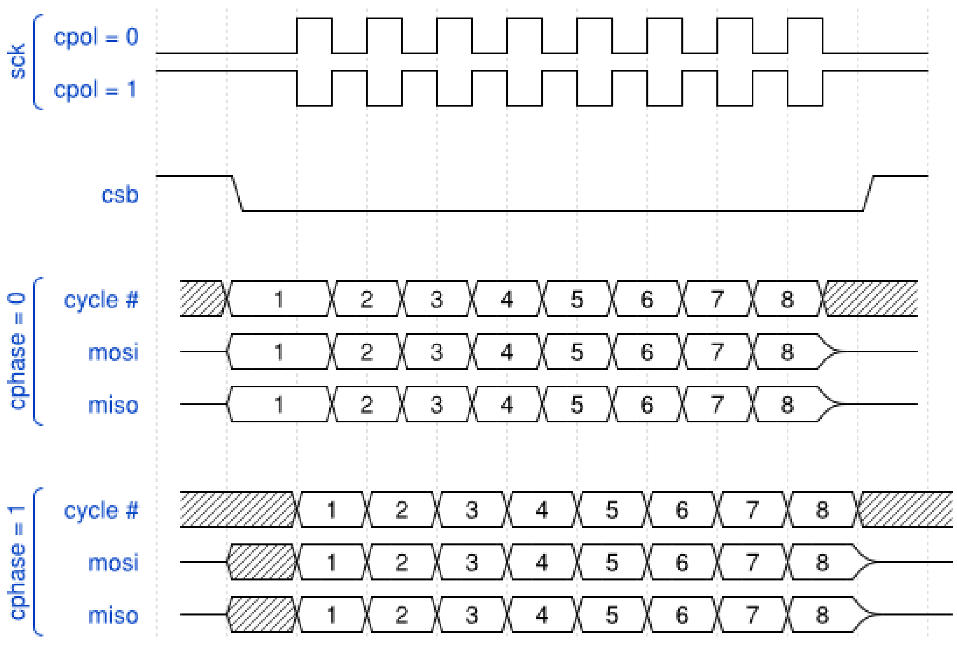

Figure 5.2: SPI timing diagram

Table 5.1: SPI Interface

Signal	Type	Description
csb	logic	Chip select bar (active low)
sck	logic	Serial clock
miso	logic	Master in, slave out
mosi	logic	Master out, slave in

5.2 Configuration

SPI agent configuration is stored in *spi_agent_config* class. Parameters are divided to build phase configuration and run phase configuration. These configuration groups reffer to corresponding UVM phases. Build phase configuration (see Table 5.2) is taken into effect only during build phase, before test is run. Run phase configuration (see Table 5.3) can be changed during test and it is immediately taken into effect.

Table 5.2: Build phase configuration

Parameter	Type	Default value	Description
active	uvm_active_pasive_enum	UVM_ACTIVE	Active or passive agent selection
has_coverage	bit	0	1 Enable coverage collection

Table 5.3: Run phase configuration

Parameter	Type	Default value	Description
checks_enable	bit	1	1 Enable functional checks (miso static during csb high)
timing_checks_enable	bit	0	1 Enable timing checks (miso setup and hold time)
cpol	bit	0	Clock polarity
cpha	bit	0	Clock phase
MSB	bit	1	1 Most significant bit is sent first
sck_period	time	30 μs	sck period
TSetup_csb	time	5 ns	time between the last edge of sck and the rising edge of the csb
THold_csb	time	5 ns	time between the falling edge of csb and first edge of sck
TSetup_mosi	time	20 ns	mosi setup time
THold_mosi	time	20 ns	mosi hold time

5.3 SPI transaction

SPI transaction is represented by *spi_seq_item*, which is derived from *bus_seq_item* 4.2. SPI transaction implements functions serialize and deserialize. Function *serialize* converts super class *bus_seq_item* data members to *mosi* bitstream. Bitstreams represent streams of bits ready to be driven to bus (resp. bits monitored from bus), including register address and write bit. Serialized write transaction (resp. read transaction) is shown in the Figure 5.3 (resp. 5.4). Function *deserialize* converts *mosi* and *miso* bitstreams to super class data members. Transaction protocol-specific data members can be found in table 5.4. Data members listed already in super class are not listed and can be

Table 5.4: SPI transaction members

Member	Type	Description
miso	logic[‘MAX_BITSTREAM_LEN:0]	Slave output bit stream
mosi	logic[‘MAX_BITSTREAM_LEN:0]	Master output bit stream
item_len	integer	Length of bit stream (same for mosi and miso) in bits

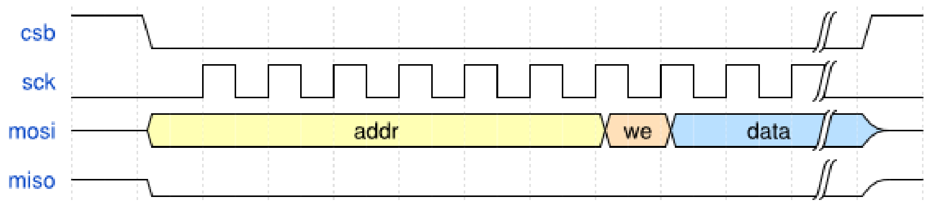

Figure 5.3: SPI write transaction

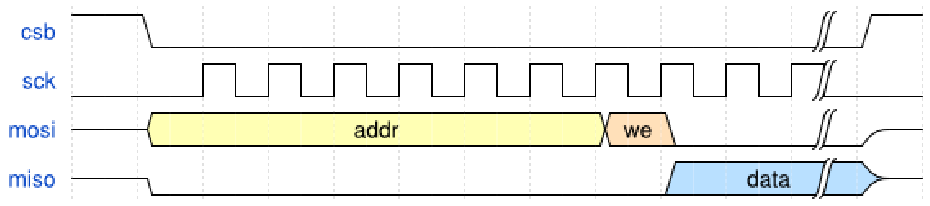

Figure 5.4: SPI read transaction

found in Section 4.2.

If error transaction is produced (*data_length* not divisible by 8), *csb* goes high after number of *sck* pulses equal to *data_length*. No other data bits are driven to *mosi*.

5.3.1 Functions

List of functions implemented in spi_seq_item can be seen in the Table 5.5. Note that functions already implemented in super class bus_seq_item are not listed. They can be seen in the Section 4.2 instead.

5.3.2 Serialize

Function serialize is called by SPI Driver before transaction is driven to SPI bus. Register address is serialized to seven most significant bits of bitstream

Table 5.5: SPI transaction functions

Function name	Param	Type	Description
get_byte_mosi	return	logic[7:0]	Get serialized bitstream mosi byte.
	byte_num	int	Requested byte number
get_byte_miso	return	logic[7:0]	Get serialized bitstream miso byte.
	byte_num	int	Requested byte number
get_bit_mosi	return	logic	Get serialized bitstream mosi bit.
	index	int	Requested bit number
get_bit_miso	return	logic	Get serialized bitstream miso bit.
	index	int	Requested bit number
push_bit	return	void	Push monitored mosi and miso bit to bitstreams.
	mosi	logic	Monitored mosi bit.
	miso	logic	Monitored miso bit.

mosi, followed by write bit and data bytes to be sent (first byte to be sent is placed right after the write bit). If transaction direction is *READ* (Master reads from Slave), *mosi* contains only register address and write bit, rest is filled with zeros. Bitstream *miso* is always set to zeroes during serialization. *Item_len* is set to $data\_length + 8$.

5.3.3 Deserialize

Function deserialize is called by SPI Monitor after transaction is collected from SPI bus. Highest seven *mosi* bitstream bits are deserialized as register address, following bit as write bit. First 8 bits of *miso* are ignored. Depending on write bit, *data* is set to remaining *mosi* bits if write bit is 1, resp. to remaining *miso* bits if write bit is 0. *Data_length* is set to $item\_length - 1$.

5.4 Driver

Driver is responsible for driving transactions passed from sequencer to SPI interface. Transaction is converted to bits by calling function serialize. If configuration bit MSB is set 0, all data bytes from transaction are reversed, only address bits are reversed in byte containing register address and write bit (write bit is kept in the place).

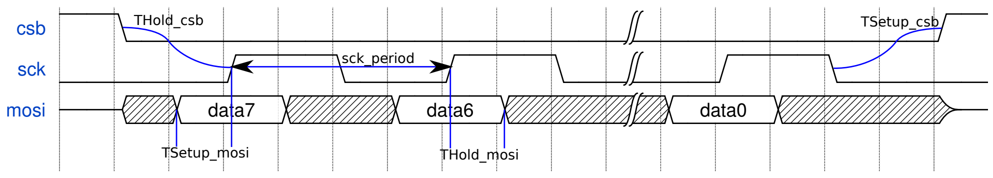

Figure 5.5: *Mosi* timing diagram

Mosi data are only valid setup time before and hold time after sck sampling edge, 'x' is put on mosi when data are invalid. These timing values can be configured in agent configuration described in the Table 5.3. If setup and hold time is set to 0, mosi is not set 'x' during transaction. The timing diagram of *mosi* can be seen in the Figure 5.5.

5.5 Monitor

Monitor observes SPI bus and stores *mosi* and *miso* values to *spi_seq_item* at every sampling edge. Once csb goes high, transaction is finished and monitor calls function *deserialize* to convert *mosi* and *miso* bitstreams to register transaction data members (register address, write bit and data). Transaction collected is sent to the analysis port.

Checks are also done in monitor, if enable_checks is set to 1, timing checks are performed if timing_checks is set 1.

5.5.1 Functional checks

Slave can change miso value only if csb is low. Otherwise monitor reports an error if checks are enabled.

5.5.2 Timing checks

Slave must keep miso stable for setup time before drive clock edge and for hold time after drive clock edge. Otherwise monitor reports error if timing checks are enabled. Miso setup and hold time is configured in spi agent config.

Table 5.6: SPI functional coverage

Watched property	Coverage point	Values
SPI clock phase	spi_phase	0, 1
SPI clock polarity	spi_polarity	0, 1
Cross phase, pol.	cross_polarity_phase	00, 01, 10, 11
Bit order	spi_MSB	MSB first, LSB first
Transaction len.	spi_txn_len txn_multibyte	standard, error[1] singlebyte, multibyte

5.6 Verification of SPI UVC

SPI UVC is verified against SPI core in smart sensor model. Coverage collection was designed to cover features specified at the beginning of the chapter. The goal of SPI UVC functional coverage is set to 100 %. The Table 5.6 provides summary on coverage points collected by testbench.

5.6.1 Verification Plan

This section provides basic planning for SPI UVC verification. Verification plan can be seen in the Table 5.7. All the tests results except *mosi_timing* are expected to be checked from text report generated by testbench. The *mosi_timing* test, which purpose is to verify that driver drives 'x' to *mosi* when out of hold and setup time, is meant to be checked from waveform. The *mosi_timing* test result can be reported by waveform screenshot.

All the tests except for *spi_monitor_checks* test are expected to be finished without error and are considered failed if errors reported. The *spi_monitor_checks* test is expected to produce *mosi not stable during csb high* and hold and setup timing checks violation. The *spi_monitor_checks* test is considered failed if any of mentioned errors is never reported.

5.6.2 Coverage report

As seen in the Table 5.8, 100 % coverage has been reached, which was the coverage goal.

[1]By error transaction is meant transaction with number of data bits not dividable by 8.

Table 5.7: SPI verification plan

test	cphase	cpol	MSB	func. checks	timing checks	multi-byte read	whole bytes only	mosi timing checks unsatisfable	errors expected
spi_cfg_00	0	0	1	0	1		√		
spi_cfg_01	0	1	1	0	1		√		
spi_cfg_10	1	0	1	0	1		√		
spi_cfg_11	1	1	1	0	1		√		
spi_err	0	0	1	0	1				
spi_lsb_first	0	0	0	0	1		√		
spi_multi_read	0	0	1	0	1	√	√		
spi_monitor_checks	0	0	1	1	1		√	√	√
mosi_timing	0	0	1	0	1		√		

Table 5.8: SPI Coverage report

Coverage group	Coverage point	Coverage
spi_cov	spi_phase	100 %
	spi_polarity	100 %
	cross_polarity_phase	100 %
	spi_MSB	100 %
	txn_len	100 %
	multibyte	100 %
	total	**100 %**

Table 5.9: SPI UVC Verification report

Test	Errors	Result	Note
spi_cfg_00	0	PASSED	
spi_cfg_01	0	PASSED	
spi_cfg_10	0	PASSED	
spi_cfg_11	0	PASSED	
spi_err	0	PASSED	
spi_lsb_first	0	PASSED	
spi_multi_read	0	PASSED	
spi_monitor_checks	118	PASSED	Only *miso* timing and *miso not stable during csn high* errors observed.

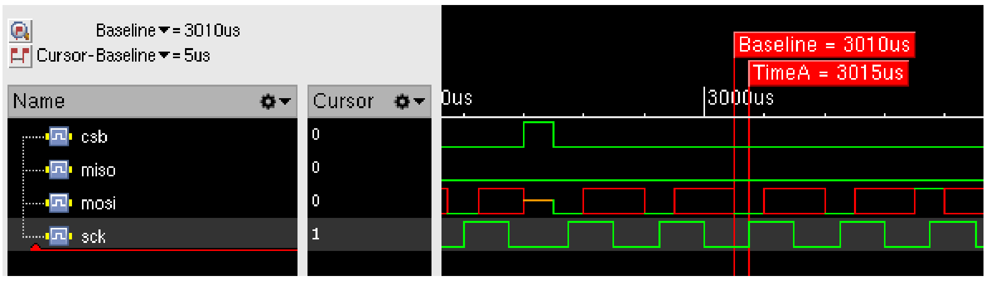

Figure 5.6: *Mosi* timing check

5.6.3 Verification report

Verification report for tests reported by text can be found in the Table 5.9. Verification report for tests reported by waveform screenshot can be found below. As all tests have passed and coverage goal has been reached, SPI UVC verification is considered successfully finished.

spi_monitor_checks_test

Mosi hold and setup times have been verified from waveform. As seen in the Figure 5.6, measured *mosi* setup time is $5us$ (note the Cursor - Baseline delta time), which is same as *mosi* hold time and corresponds to hold and setup times configured in the test.

Miso setup time has been verified by setting *TSetup_miso* to value higher than the SPI slave core design setup time ($5ns$) and observing *miso* setup time violation errors.

Chapter 6

I2C UVC

I2C UVM verification component (see Figure 6.1) verifies I2C interface. Sequence of transactions is received through sequence port, sequencer passes transactions to driver, driver generates I2C transaction. Monitor reads DUT responce from I2C interface, reconstructs transaction. Monitor transaction is sent through analysis port to scoreboard.

I2C UVC shall be capable of following features:

- MSb or LSb first
- multibyte read
- error injection
  - unexpected end of transaction
  - 'x' driven to *mosi* out of setup and hold time
- *sda* timing checks

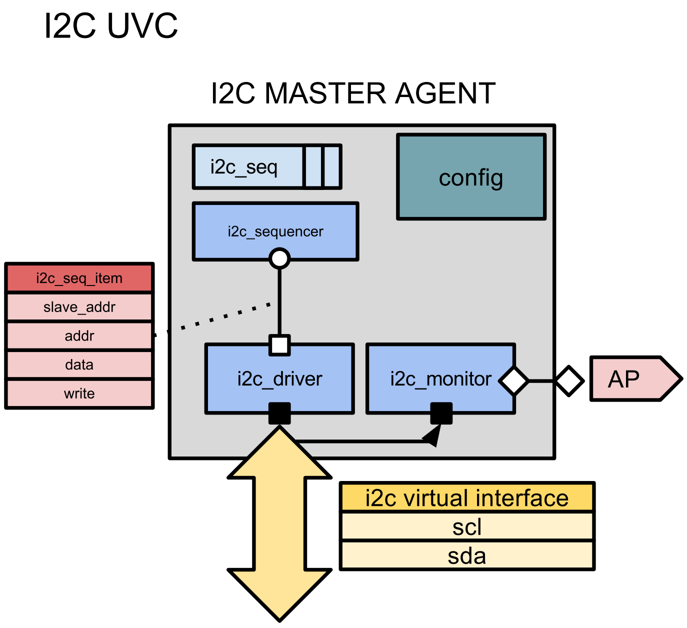

Figure 6.1: I2C UVC structure

6.1 I2C interface

I2C is a serial protocol for two-wire interface (see th Table 6.1) to connect low-speed devices like microcontrollers, EEPROMs, A/D and D/A converters, I/O interfaces and other similar peripherals in embedded systems, invented by Philips. Each I2C slave device needs an address – they must still be obtained from NXP (formerly Philips semiconductors) [4].

Transaction timing diagram can be seen in the Figure 6.2.

Table 6.1: I2C Interface

Signal	Type	Description
scl	logic	Serial clock
sda	logic	Serial data

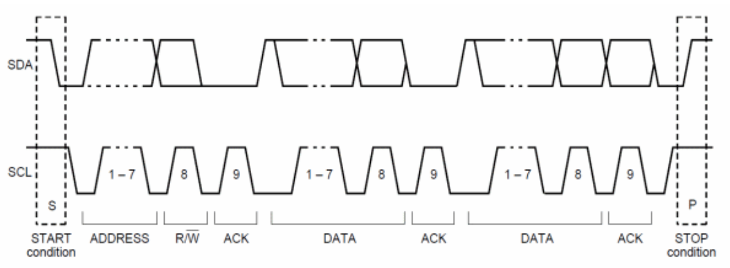

Figure 6.2: I2C timing diagram [4]

6.2 Configuration

I2C agent configuration is stored in *i2c_agent_config* class. Parameters are divided to build phase configuration and run phase configuration. These configuration groups reffer to corresponding UVM phases. Build phase configuration (see table 6.2) is taken into effect only during build phase, before test is run. Run phase configuration (see table 6.3) can be changed during test and it is immidiately taken into effect. Note that *scl* period is set via *Tscl_low* and *Tscl_high* parameter while $Tscl = Tscl\_low + Tscl\_high$.

Table 6.2: Build phase configuration

Parameter	Type	Default value	Description
active	uvm_active_pasive_enum	UVM_ACTIVE	Active or passive agent selection
has_coverage	bit	0	1 Enable coverage collection

Table 6.3: Run phase configuration

Parameter	Type	Default value	Description
checks_enable	bit	1	1 Enable functional checks (miso static during csb high)
timing_checks_enable	bit	0	1 Enable timing checks (miso setup and hold time)
TSetup_stop	time	4.00 us	Stop condition setup time
THold_start	time	4.00 us	Start condition hold time
TSetup_start	time	4.70 us	Repeated start condition setup time
THold_sda	time	0.25 us	Data hold time
TSetup_sda	time	0.10 us	Data setup time
Tbuf	time	47.00 us	Bus free time between stop and start condition
Tscl_low	time	5.00 us	Scl low duration
Tscl_high	time	15.0 ns	Scl high duration

6.3 I2C transaction

I2C transaction is represented by *i2c_seq_item*, which is derived from *bus_seq_item* 4.2. I2C transaction implements functions serialize and deserialize. Function *serialize* converts super class *bus_seq_item* data members to *bitstream. Bitstream* represents stream of bits ready to be driven to *sda* wire (resp. bits monitored from *sda*), including register address and write bit. Function *deserialize* converts *bitstream* to super class data members. Transaction protocol-specific data members can be found in table 6.4. Data members listed already in super class are not listed and can be found in Section 4.2.

Note that although *slave_addr* has 10 bit length, I2C currently supports only 7 bit slave address mode. Slave address is stored in bits 0 to 6. Remaining 3 bits are reserved for future 10 bit slave address mode extension.

If error transaction is produced (*data_length* not divisible by 8), stop condition is generated after number of *scl* pulses equal to *data_length.* No other data bits are driven to *sda.*

Table 6.4: I2C transaction members

Member	Type	Description
bitstream	logic[‘MAX_BITSTREAM_LEN:0]	*sda* bit stream
item_length	integer	Length of *sda* bit stream in bits
slave_addr	logic[9:0]	Transaction slave address
slave_addr_length	int	Slave address length

6.4 Driver

Driver is responsible for driving transactions passed from sequencer to I2C interface. Transaction is converted to bits by calling function serialize. If configuration bit MSB is set 0, all data bytes from transaction are reversed, only address bits are reversed in byte containing register address and write bit (write bit is kept in the place).

During i2c write operation, *sda* data are only valid setup time before rising edge of *scl* and hold time after *scl* falling edge, 'x' is put on *sda* when data are invalid. These timing values can be configured in agent configuration described in the Table 6.3. If setup and hold time is set to 0, *sda* is not set 'x' during transaction.

Due to half-duplex characteristic of I2C interface, register read transaction cannot be performed in one i2c operation. Transaction must be divided to two operations: write operation where register address is transfered to slave, and read operation where slave transfers register data to master. To keep unified driving process, register write transaction is divided to two I2C write operations: first transfers register address and second one transfers data to be written. Using two I2C transactions to perform register write operation is not power effective, but low power approach is not goal of this thesis.

6.5 Monitor

Monitor observes I2C bus and stores *sda* values to *i2c_seq_item* at every sampling edge. Once stop condition is observed, transaction is finished and monitor calls function *deserialize* to convert *sda* bitstream to register transaction data members (register address, write bit and data). Transaction collected is sent to the analysis port.

Checks are also done in monitor, if *enable_checks* is set to 1, timing checks are performed if *timing_checks* is set 1.

6.5.1 Timing checks

Slave must keep *sda* stable for setup time before rising edge of *scl* and for hold time after *scl* falling edge. Otherwise monitor reports error if timing checks are enabled. *Sda* setup and hold time is configured in *i2c_agent_config*.

6.6 Verification of I2C UVC

I2C UVC is verified against I2C core in smart sensor model. Coverage collection was designed to cover features specified at the beginning of the chapter. The goal of I2C UVC functional coverage is set to 100 %. The Table 6.5 provides summary on coverage points collected by testbench.

6.6.1 Verification Plan

This section provides basic planning for I2C UVC verification. Verification plan can be seen in the Table 6.6. All the tests results except *sda_timing* are expected to be checked from text report generated by testbench. The *sda_timing* test, which purpose is to verify that driver drives 'x' to *sda* when out of hold and setup time, is meant to be checked from waveform. The *sda_timing* test result can be reported by waveform screenshot.

All the tests except for *i2c_monitor_checks* test are expected to be finished without error and are considered failed if errors reported. The *i2c_monitor_checks* test is expected to produce hold and setup timing checks violation. The *i2c_monitor_checks* test is considered failed if any of mentioned errors is never reported.

[2]By error transaction is meant transaction with number of data bits not dividable by 8.

Table 6.5: I2C functional coverage

Watched property	Coverage point	Values
Bit order	i2c_MSB	MSB first, LSB first
Transaction len.	i2c_txn_len	standard, error[2]
	txn_multibyte	singlebyte, multibyte

Table 6.6: I2C verification plan

test	MSB	timing checks	multi-byte read	whole bytes only	sda timing checks unsatisfable	errors expected
i2c_base	1	1		√		
i2c_err	1	1				
i2c_lsb_first	0	1		√		
i2c_multi_read	1	1	√	√		
i2c_monitor_checks	1	1		√	√	√
sda_timing	1	1		√		

Table 6.7: I2C Coverage report

Coverage group	Coverage point	Coverage
i2c_cov	i2c_MSB	100 %
	txn_len	100 %
	multibyte	100 %
	total	100 %

Coverage report

As seen in the Table 6.7, 100 % coverage has been reached, which was the coverage goal.

6.6.2 Verification report

Verification report for tests reported by text can be found in the Table 6.8. Verification report for tests reported by waveform screenshot can be found below. As all tests have passed and coverage goal has been reached, I2C UVC verification is considered successfully finished.

Table 6.8: I2C verification report

Test	Errors	Result	Note
i2c_test	0	PASSED	
i2c_err	0	PASSED	
i2c_lsb_first	0	PASSED	
i2c_multi_read	0	PASSED	
i2c_monitor_checks	1284	PASSED	Only *sda* timing errors observed.

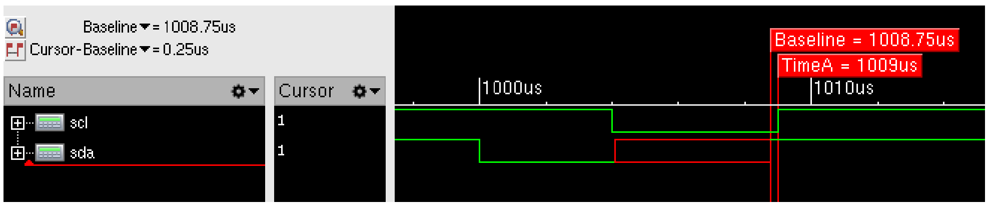

Figure 6.3: *Sda* timing check

i2c_monitor_checks_test

Sda hold and setup times have been verified from waveform. As seen in the Figure 6.3, measured *sda* setup time is 0.25 us (note the Cursor - Baseline delta time), which is same as *sda* hold time and corresponds to hold and setup times configured in the test.

Conclusion

The main task of this thesis was to build smart sensor model from free available cores, implement testbench for smart sensors and verify the testbench against the model.

The sensor model has been designed to receive measured data from ADC, perform conversion according to constants stored in register and store converted value to register. Register map is accessable through SPI and I2C buses.

The sensor model has been built from cores available at OpenCores. OpenRISC, I2C slave core and SPI slave core have been used, Wishbone has been used as interconnection bus. Some bug fixes must have been made to cores. Wishbone-to-ADC interface had to be implemented, as none was available during writing of this thesis. Sensor behavior has been written in C and compiled to memory model.

SystemVerilog and UVM have been selected as verification framework. The testbench follows classic UVM block-level hierarchy and includes standalone SPI agent, I2C agent, and ADC agent. SPI and I2C agents are capable of error injection, coverage collection and timing checks.

The smart sensor testbench has been verified against smart sensor model, coverage goals have been reached and testcases passed. The potential future works could make some improvements to the testbench. Non-volatile memory model could be reworked to use Cadence tools for register map generation from IP-XACT file. Bus transaction hierarchy could be changed to let bus agents send any transaction (without register address and write bit), which would increase reusability.

In conclusion, smart sensor testbench has been implemented and verified against smart sensor model, satisfying the requirements given.

Bibliography

[1] Dallas semiconductor. *DS1722 - Digital thermometer with SPI.*

[2] National semiconductor corporation. *LM75 - I2C Digital Temperature Sensor and Thermal Watchdog.*

[3] Verification Academy. *UVM Cookbook [online].* [cit. 2015-02-26], account required. Available from: `https://verificationacademy.com/cookbook/`

[4] I2C Info – I2C Bus, Interface and Protocol. [cit. 2015-04-09]. Available from: `http://http://i2c.info/`

[5] SPI Core UVC specification. 2010, [internal document].

[6] Bergeron, J. *Writing testbenches.* Kluwer Academic Publishers, second edition, 2003, ISBN 1-4020-7401-8.

[7] Sanghavi, A. What is formal verification? 2010, [cit. 2015-04-05]. Available from: `http://www.eetasia.com/STATIC/PDF/201005/EEOL_2010MAY21_EDA_TA_01.pdf`

[8] Register-transfer level. [cit. 2015-03-24]. Available from: `http://en.wikipedia.org/wiki/Register-transfer_level`

[9] Cadence. *Gate-Level Simulation Methodology [online].* [cit. 2015-03-24]. Available from: `http://www.cadence.com/rl/resources/white_papers/gate_level_simulation_wp.pdf`

[10] Spear, C. *Systemverilog for Verification.* Springer, second edition, 2008, ISBN 978-0-387-76529-7.

[11] The Designer's Guide To PSL. [cit. 2015-03-02]. Available from: `https://www.doulos.com/knowhow/psl/`

[12] UVM, OVM and VMM. [cit. 2015-03-02]. Available from: `https://www.aldec.com/en/solutions/functional_verification/uvm_ovm_vmm`

[13] A Meade, K.; Rosenberg, S. *A practical Guide to Adopting the Universal Verification Methodology (UVM)*. Cadence design systems, second edition, 2013, ISBN 978-1-300-53593-5.

[14] Smart Sensors – Not Only Intelligent, but Adaptable. 2011, [cit. 2015-02-23]. Available from: `http://www.digikey.com/en/articles/techzone/2011/sep/smart-sensors---not-only-intelligent-but-adaptable`

[15] OpenCores website. [cit. 23. 2. 2015]. Available from: `http://opencores.org`

[16] OpenRISC 1200 project page. [cit. 23. 2. 2015]. Available from: `http://opencores.org/or1k/OR1200_OpenRISC_Processor`

[17] Spi master/slave core. [cit. 24. 3. 2015]. Available from: `http://opencores.org/project,spi_master_slave`

[18] I2C Slave. [cit. 2015-04-05]. Available from: `http://opencores.org/project,i2cslave`

[19] I2C Master slave core. [cit. 2015-04-05]. Available from: `http://opencores.org/project,i2c_master_slave`

[20] I2C master/slave core. [cit. 2015-04-05]. Available from: `http://opencores.org/project,i2c_master_slave_core`

[21] Analog-to-digital converter. [cit. 2015-04-05]. Available from: `http://en.wikipedia.org/wiki/Analog-to-digital_converter`

[22] Serial Peripheral Interface Bus. [cit. 2015-03-24]. Available from: `http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus`

[23] Motorola. *SPI block guide [online]*. [cit. 2015-03-24]. Available from: `http://www.ee.nmt.edu/~teare/ee308l/datasheets/S12SPIV3.pdf`

Appendix A

Acronyms

DUT	Design under test
UVM	Unified verification methodology
OVM	Open verification methodology
VMM	Verification methodology manual
I2C	Inter-integrated circuit
SPI	Serial peripheral interface
ADC	Analog/Digital converter
GPIO	General purpose Input/Output
LSB	Least significant bit
MSB	Most significant bit
UVC	UVM verification component
NVM	Non-volatile memory

Appendix B

Contents of enclosed CD

```
readme.txt ...................... the file with CD contents description
src ..................................... the directory of source codes
    thesis .............. the directory of LaTeX source codes of the thesis
    DP_Halecek_Ivo_2015.pdf ............. the thesis text in PDF format
    DP_Halecek_Ivo_2015.ps ................ the thesis text in PS format
```