REPORT

SystemVerilog versus PSL with VHDL for mixed-signal design testbench validation

Name of the student : …Mladen Stanic………………………………

Name of supervisor: : …Alain Vachoux………………………………

Name of INPG Tutor : …Lorena Anghel………………………………

Company: …Microelectronic Systems Laboratory, EPFL……………………………………………………

Address : …ELD 338, Station 11, CH-1015, Lausanne……………………………………………………

Ecole Nationale Supérieure d'Electronique et de Radioélectricité de Grenoble

Ecole Nationale Supérieure d'Physique de Grenoble

INP – Minatec, 3 Parvis Louis Néel - B.P. 257
38016 GRENOBLE Cedex 1 – France
Tél : 33 (0)4 56 52 91 00 - Fax : 33 (0)4 56 52 91 03
The goal of this project was to master how the Semtech home-made VHDL analog package has to be used to write assertions in the PSL language and SystemVerilog native assertions. Moreover, how to write SystemVerilog testbenches using advanced verification methodology was studied. The evaluation has been done on the Semtech zooming A/D converter with a 12-bit processor and the I2C interface. Using Mentor QuestaSim 6.3 EDA tool, it has been shown that the various languages (PSL, VHDL, Verilog and SystemVerilog) are successfully and effectively mixed all together with the Semtech analog package. A future tool release should resolve the problems encountered during the project.

Le but du projet est d'étudier comment utiliser le package VHDL analogue fait maison de Semtech en rajoutant des assertions dans les languages PSL et dans SystemVerilog. De plus, il fallait étudier comment écrire des testbenches en SystemVerilog en utilisant une méthodologie avancée de verification. L'évaluation a été faite à partir d'un circuit convertisseur A/D de Semtech avec un processeur de 12 bits ainsi qu'un interface I2C. En utilisant l'outil CAD de Mentor Graphics Questasim 6.3, il a été montré que les différents languages (PSL, VHDL, Verilog et SystemVerilog) ont pu être mélangés avec succès en utilisant le package analogue de Semtech. Une version future de l'outil devrait résoudre les problèmes rencontrés durant le projet.

L'obiettivo del progetto è di analizzare come i VHDL-package in campo analogico realizzati alla SEMTECH debbano essere utilizzati al fine di scrivere assertions nei linguaggi PSL e SystemVerilog. Inoltre tecniche per scrivere testbench in SystemVerilog usando strategie di test avanzate vengono analizzate in questa tesi. Questo studio è stato condotto su un A/D converter equipaggiato di un processore a 12-bit e un interfaccia I2C. Utilizzando il tool EDA Mentor QuestaSim 6.3 è stato dimostrato che i vari linguaggi (PSL, VHDL, Verilog e SystemVerilog) posso essere utilizzati con successo con il package analogico prodotto dalla Semtech. Una version future del tool risolverà problemi incontrati durante il progetto.
# Table of Contents

**Introduction**

- 1.1 Motivation................................................................. 1
- 1.2 Report organization..................................................... 2

**Verification with PSL**

- 2.1 Introduction to PSL...................................................... 3
  - 2.1.1 Purpose and usage.................................................. 3
  - 2.1.2 The structure of PSL................................................ 3
  - 2.1.3 Simple properties.................................................... 4
  - 2.1.4 Temporal logic....................................................... 5
  - 2.1.5 Sequences............................................................. 5
  - 2.1.6 PSL built-in functions............................................. 6
- 2.2 QuestaSim Verification with PSL..................................... 6
  - 2.2.1 Event bounded window pattern............................... 7
  - 2.2.2 FSM state coverage pattern.................................... 8
  - 2.2.3 State sequence coverage pattern............................. 8
  - 2.2.4 Time bounded window pattern............................... 8
  - 2.2.5 Other properties and some remarks........................... 9

**The Semtech environment**

- 3.1 The Semtech Analog Package and PSL assertions.................. 10
- 3.2 The Semtech zooming A/D converter (SX8724)........................ 11

**PSL assertions for SX8724**

- 4.1 GriscSystem........................................................... 15
- 4.2 analog2std............................................................... 16
- 4.3 GriscSystem stability properties without virtual clock.......... 17
- 4.4 Grisc System interface protocol – protocol properties........ 19
- 4.5 ADC digital controller.............................................. 22
- 4.6 PGA................................................................. 23
- 4.7 ADC result............................................................ 24

**System Verilog assertions for SX8724**

- 5.1 Grisc System........................................................... 29
- 5.2 ADC digital controller.............................................. 29

**SystemVerilog testbench for SX8724**

- 6.1 Introduction to SV for Verification................................ 32
- 6.2 SV layered testbench.................................................. 34
6.2.1 Wrapper.............................................................................................................34
6.2.2 Top module........................................................................................................36
6.2.3 Test program......................................................................................................36
6.2.4 Environment......................................................................................................37
6.2.5 Transaction .......................................................................................................38
6.2.6 I2C Agent and I2C Checker..............................................................................38
6.2.7 I2C Driver and I2C Monitor.............................................................................40
6.2.8 Driver (ac[6])..................................................................................................41
6.2.9 Scoreboard........................................................................................................42

Conclusion.................................................................................................................44

Appendix 1..................................................................................................................47
Appendix 2..................................................................................................................49
Appendix 3..................................................................................................................56
Appendix 4..................................................................................................................60
Appendix 5..................................................................................................................80
Appendix 6..................................................................................................................82
### Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>PSL in Questa Debug Flow</td>
<td>7</td>
</tr>
<tr>
<td>3.1</td>
<td>SX8724 functional block diagram</td>
<td>11</td>
</tr>
<tr>
<td>3.2</td>
<td>chip block diagram</td>
<td>12</td>
</tr>
<tr>
<td>3.3</td>
<td>grisc core</td>
<td>13</td>
</tr>
<tr>
<td>3.4</td>
<td>Timing diagrams for GRISC data bus interface</td>
<td>13</td>
</tr>
<tr>
<td>4.1</td>
<td>rising_edge_dme and dme_stable</td>
<td>17</td>
</tr>
<tr>
<td>4.10</td>
<td>adc_result_1</td>
<td>24</td>
</tr>
<tr>
<td>4.2</td>
<td>setting_da and da_is_stable</td>
<td>18</td>
</tr>
<tr>
<td>4.3</td>
<td>status_register_reading</td>
<td>20</td>
</tr>
<tr>
<td>4.4</td>
<td>address_register_reading</td>
<td>20</td>
</tr>
<tr>
<td>4.5</td>
<td>data_register_reading_in_re</td>
<td>21</td>
</tr>
<tr>
<td>4.6</td>
<td>analog_input_multiplexer</td>
<td>22</td>
</tr>
<tr>
<td>4.7</td>
<td>adc_result_lsb</td>
<td>22</td>
</tr>
<tr>
<td>4.8</td>
<td>pga1_gain_10</td>
<td>23</td>
</tr>
<tr>
<td>5.1</td>
<td>dme_stable</td>
<td>29</td>
</tr>
<tr>
<td>6.1</td>
<td>Constrained-random test progres</td>
<td>32</td>
</tr>
<tr>
<td>6.2</td>
<td>Full testbench with all layers</td>
<td>34</td>
</tr>
</tbody>
</table>
Acronyms

ADC - Analog Digital Conversion/Converter
ADC DIG - ADC Digital controller
AFV - Advanced Functional Verification
AVM - Advanced Verification Methodology
DUT - Design Under Test
EDA - Electronic Design Automation
GRISC - SX8723 RISC controller
HDL - Hardware Description Language
HVL - Hardware Verification Language
I2C - multi-master serial bus (stands for Inter-Integrated Circuit)
LHS - Left Hand Side
LSB - Least Significant Bit
MSB - Most Significant Bit
PGA - Programmable Gain Amplifier
PSL - Property Specification Language
RHS - Right Hand Side
RTL - Register Transfer Level
VHDL - Hardware Description Language
SV - SystemVerilog
Gantt Chart

<table>
<thead>
<tr>
<th>Name</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSL syntax</td>
<td>12d</td>
</tr>
<tr>
<td>QuestaSim simulation with PSL</td>
<td>10d</td>
</tr>
<tr>
<td>Applying State of art assertions patterns</td>
<td>5d</td>
</tr>
<tr>
<td>Semtech Analog package and simple PSL assertions</td>
<td>5d</td>
</tr>
<tr>
<td>Learning SX8723 functionality</td>
<td>5d</td>
</tr>
<tr>
<td>Understanding GriscSystem functionality</td>
<td>5d</td>
</tr>
<tr>
<td>Assertions for GriscSystem</td>
<td>5d</td>
</tr>
<tr>
<td>Assertions for ana2std function</td>
<td>5d</td>
</tr>
<tr>
<td>Improving assertions for signal stability</td>
<td>5d</td>
</tr>
<tr>
<td>Assertions for Grisc interface protocol</td>
<td>5d</td>
</tr>
<tr>
<td>Assertions for ADC DIG</td>
<td>5d</td>
</tr>
<tr>
<td>Assertions for PGA</td>
<td>5d</td>
</tr>
<tr>
<td>Writing Unix Shell scripts for Compiling and Simulation</td>
<td>5d</td>
</tr>
<tr>
<td>SystemVerilog assertions for Grisc</td>
<td>5d</td>
</tr>
<tr>
<td>SystemVerilog assertions for ADC DIG</td>
<td>5d</td>
</tr>
<tr>
<td>Learning SystemVerilog for Verification</td>
<td>5d</td>
</tr>
<tr>
<td>Writing SystemVerilog testbench for SX8723</td>
<td>5d</td>
</tr>
<tr>
<td>Writing the report</td>
<td>12d</td>
</tr>
<tr>
<td>Making Ppt presentation</td>
<td>3d</td>
</tr>
<tr>
<td>Verification with the new testbench and final results</td>
<td>5d</td>
</tr>
<tr>
<td>EPFL presentation</td>
<td>1d</td>
</tr>
<tr>
<td>Semtech presentation</td>
<td>1d</td>
</tr>
<tr>
<td>ENSERG presentation</td>
<td>1d</td>
</tr>
</tbody>
</table>

Gantt Table

<table>
<thead>
<tr>
<th>WBS</th>
<th>Name</th>
<th>Start</th>
<th>Finish</th>
<th>Work</th>
<th>Slack</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PSL syntax</td>
<td>Mar 1</td>
<td>Mar 16</td>
<td>12d</td>
<td>124d</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>QuestaSim simulation with PSL</td>
<td>Mar 19</td>
<td>Mar 30</td>
<td>10d</td>
<td>114d</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Applying State of art assertions patterns</td>
<td>Mar 26</td>
<td>Mar 30</td>
<td>5d</td>
<td>114d</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>Semtech Analog package and simple PSL assertions</td>
<td>Apr 2</td>
<td>Apr 6</td>
<td>5d</td>
<td>109d</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>Learning SX8723 functionality</td>
<td>Apr 9</td>
<td>Apr 13</td>
<td>5d</td>
<td>104d</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>Understanding GriscSystem functionality</td>
<td>Apr 26</td>
<td>Apr 20</td>
<td>5d</td>
<td>99d</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>Assertions for GriscSystem</td>
<td>Apr 23</td>
<td>May 4</td>
<td>10d</td>
<td>88d</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>Assertions for ana2std function</td>
<td>May 7</td>
<td>May 11</td>
<td>5d</td>
<td>84d</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>Improving assertions for signal stability</td>
<td>May 14</td>
<td>May 18</td>
<td>5d</td>
<td>70d</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>Assertions for Grisc interface protocol</td>
<td>May 21</td>
<td>Jun 1</td>
<td>10d</td>
<td>60d</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>Assertions for ADC DIG</td>
<td>Jun 4</td>
<td>Jun 8</td>
<td>5d</td>
<td>64d</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>Assertions for PGA</td>
<td>Jun 11</td>
<td>Jun 15</td>
<td>5d</td>
<td>59d</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>Writing Unix Shell scripts for Compiling and Simulation</td>
<td>Jun 18</td>
<td>Jul 6</td>
<td>15d</td>
<td>44d</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>SystemVerilog assertions for Grisc</td>
<td>Jun 25</td>
<td>Jun 29</td>
<td>5d</td>
<td>45d</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>SystemVerilog assertions for ADC DIG</td>
<td>Jul 2</td>
<td>Jul 6</td>
<td>5d</td>
<td>44d</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>Learning SystemVerilog for Verification</td>
<td>Jul 9</td>
<td>Jul 13</td>
<td>5d</td>
<td>39d</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>Writing SystemVerilog testbench for SX8723</td>
<td>Jul 16</td>
<td>Aug 3</td>
<td>15d</td>
<td>24d</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>Writing the report</td>
<td>Aug 5</td>
<td>Aug 21</td>
<td>12d</td>
<td>12d</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>Making Ppt presentation</td>
<td>Aug 22</td>
<td>Aug 24</td>
<td>3d</td>
<td>9d</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>Verification with the new testbench and final results</td>
<td>Aug 27</td>
<td>Aug 31</td>
<td>5d</td>
<td>4d</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>EPFL presentation</td>
<td>Aug 28</td>
<td>Aug 28</td>
<td>1d</td>
<td>7d</td>
<td>0</td>
</tr>
<tr>
<td>22</td>
<td>Semtech presentation</td>
<td>Aug 30</td>
<td>Aug 30</td>
<td>1d</td>
<td>5d</td>
<td>0</td>
</tr>
<tr>
<td>23</td>
<td>ENSERG presentation</td>
<td>Sep 6</td>
<td>Sep 6</td>
<td>1d</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Discription of environment and estimations costs

This project was done in Semtech company (www.semtech.com), Neuchatel. Semtech Corporation is a supplier of analog and mixed-signal semiconductor products. The company is providing customers with wireless and sensing products. The Company's integrated circuits (ICs) are employed in communications, computer and computer-peripheral, automated test equipment, industrial and other commercial applications.

The EDA tool, used during the project is MentorGraphics' QuestaSim Advanced Functional Verification (6.2 and 6.3). Questa AFV features Advanced Verification Methodology (AVM), SystemVerilog, SystemC, Verilog, VHDL and PSL.

Questa AFV short term lease for 6 months with a floating license is 27 451 $. Estimated costs for Semtech are about 5 000 $.
Chapter 1
Introduction

1.1 Motivation

The Semtech company ([http://www.semtech.com/products/wireless&sensing/trans/semic.jsp](http://www.semtech.com/products/wireless&sensing/trans/semic.jsp)) is using VHDL for the validation of mixed-signal designs. Semtech developed a home-made VHDL package that provides VHDL types and routines for modeling analog components. The goal is not to describe the full details of the analog behaviors, but rather to verify that the interactions of the analog blocks with the rest of the (digital) system are correct. This is done after the system has been designed down to layout. VHDL netlists are generated from schematics and analog blocks are replaced by abstract VHDL models using the VHDL package mentioned earlier.

The growing complexity of the mixed-signal systems now requires a more efficient simulation technique (faster simulation times) and a more efficient verification technique (better formulation of design properties to be checked and better checking of these properties).

One way to improve this bottom-up verification methodology is to add formal assertions to the system VHDL model using PSL (Property Specification Language) that define properties the designed system must meet. PSL assertions are loaded when the VHDL netlist is loaded in the simulator (Modelsim) and checked during simulation. Testbenches are in this case still written in VHDL.

Another way is to use the SystemVerilog language for both testbenches and assertions. This approach would have the merit to be more integrated as a single language is used. Also, SystemVerilog natively provides advanced verification support such as randomization that should improve the verification coverage. In that case, testbenches would need to be rewritten in SystemVerilog, but the mixed-signal design to be verified would still be described using VHDL.

The objective of the project is therefore to evaluate how the two above approaches of using simulation/verification languages and associated tools may be put to work to improve the existing mixed-signal verification methodology used in Semtech. It amounts to set up appropriate testbenches using components described in different languages, to figure out how all these components may be used jointly in a consistent verification framework based on commercial EDA tools (Mentor Modelsim), and to define an associated verification methodology.

The evaluation will be done on a medium-complexity mixed-signal design, namely a zooming A/D converter with a small custom 12-bit processor and an I²C interface. Semtech will provide the verification rules to be applied. The project will not involve design tasks.
In summary, what will have to be mastered during the project is:
- How the Semtech analog VHDL package has to be used
- How to write assertions in the PSL language.
- How to write SystemVerilog testbenches using native SV assertions
- How to use the ModelSim tool, effectively mixing the various languages.

1.2 Report organization

This report will be organized as follows: Chapter 2 will firstly discuss how QuestaSim implements Verification with PSL. State of art assertions patterns are used. In Chapter 3 the Semtech VHDL analog package and A/D converter SX8723 are introduced and it shows how a simple assertion for analog signal can be written in the PSL language. Chapter 4 presents PSL assertions written for SX8724 modules. Properties for GriscSystem digital signals, PGA analog signals and ADC DIG mixed-signal are defined. Chapter 5 demonstrates how previously defined PSL assertions can be written in SystemVerilog. Questa Sim features for mixed-language simulation are discussed as well. Chapter 6 presents SystemVerilog layered testbench for SX8723 simulation. Chapter 7 will then briefly comment the project.
Chapter 2
Verification with PSL

2.1 Introduction to PSL

2.1.1 Purpose and usage

The goal of hardware verification is to ensure that a design's implementation satisfies its specification. Key to the design and verification process is the specification. The process of writing specification is creating a natural language description of design requirements. This form of specification is both ambiguous and unverifiable. Also, it is problematic to ensure that all functional aspects of the specification are verified and covered.

The Accellera Property Specification Language (PSL) was developed to resolve these problems. PSL specifies design properties using a concise syntax with clearly-defined formal semantics. It enables the verification engineer to validate that the implementation satisfies its specification through dynamic verification i.e through simulation.

PSL was specifically developed to fulfill the following general hardware functional specification requirements[3]:
— easy to learn, write, and read
— concise syntax
— rigorously well-defined formal semantics
— expressive power, permitting the specification for a large class of real world design properties
— known efficient underlying algorithms in simulation, as well as formal verification.

PSL as a language for the formal specification of hardware, is used to describe properties that are required to hold in the design under verification. With PSL we can write properties that are both easy to read and mathematically precise.

2.1.2 The structure of PSL

The PSL language is formally structured into four distinct layers: the boolean, temporal, verification and modelling layers. The verification and temporal layers have a native syntax of their own, whereas the modelling and boolean layers are coming from the syntax of the underlying HDL (VHDL or Verilog).

The boolean layer consists of boolean expressions containing variables and operators from the underlying language. Formally, the boolean layer consists of any expression that is allowed as the condition in an if statement in the underlying language:

\[(a & b) == 0\] // Verilog flavour
Expressions in the boolean layer are evaluated at a single point in time, i.e they are sampled on a
clock, since PSL is intended for designs with synchronous timing:

\[(a \text{ and } b) = 0 \quad -- \text{ VHDL flavour}\]

They can be sampled either on a rising edge, either on a falling edge (falling_edge(clk))
The verification layer consists of verification directives that are instructions to a tool to tell it
what to do with a property. The main verification directives are assert (the tool must prove the
property), assume (the tool may assume the given property is true) and cover (the tool should
measure how often the given property occurs during simulation).

\[\text{assert (always a and b) @rising_edge(clk)};\]

PSL properties, i.e verification directives can be embedded in the HDL code as comments or can
be written in a separate file (PSL file). In the latter case, they are grouped into verification units
(vunits), and a verification unit is explicitly bound to an HDL module or design unit.

\[
\text{vunit my_PSL_properties}(<\text{VHDL unit or Verilog module}>) \{ \\
\quad \text{assert (always a and b) @ rising_edge(clk)}; \\
\quad \text{assume (never b and c)@ rising_edge(clk)};
\}
\]

In the modelling layer we can include code from the underlying language (VHDL or Verilog) to
define properties, that couldn't be written using PSL alone. For example, the modelling layer
could be used to calculate the expected value of an output.

2.1.3 Simple properties

To avoid to repeat the explicit clock operator @ in every single assertion, it is also possible to
define a default clock:

\[
\text{default clock is rising_edge(clk)}; \\
\text{assert always CONDITION;}
\]

It means that every property, that is not explicitly clocked using the @ operator, will be clocked
with the default clock.

Very common still for property writing is to use the form of an implication, with a pre-condition
that must be satisfied before the main condition is checked.

\[\text{assert always PRECONDITION } \rightarrow \text{ CONDITION;}\]

This asserts that whenever PRECONDITION holds, CONDITION must hold in the same cycle.
The symbol \(\rightarrow\) denotes logical implication.

Also, very often within precondition and condition, we use temporal sequences enclosed in
braces.

\[\text{assert always } \{a;b\} \rightarrow \{c;d\};\]
The sequence \{a;b\} holds if a holds in a cycle and b holds in the following cycle. The implication \(|->\) means that if the first sequence holds, the second sequence must hold also, with the two sequences overlapping by a single cycle. (Alternative implication operator \(\Rightarrow\) requires that the two sequences do not overlap.)

A terminating condition \texttt{abort} which will cause the property to be abandoned through the matching of a temporal sequence. can be used as:

\[
\text{assert (always \{a;b\} \(\Rightarrow\) \{c;d\} abort reset))@ (posedge clk);}
\]

When the reset goes true, the obligation for the suffix implication to hold is removed.

PSL properties can be named, and later we assert them with a verification directive:

\[
\text{property my\_property is (always \{a;b\} \(\Rightarrow\) \{c;d\})@ rising\_edge(dme);}
\]

\[
\text{... assert my\_property;}
\]

2.1.4 Temporal logic

The temporal operators include \texttt{always}, \texttt{never}, \texttt{next}, \texttt{until} and \texttt{before}, amongst others. The meaning of these operators is quite intuitive. The always operator holds if its operand holds in every single cycle, whilst the never operator holds if its operand fails to hold in every single cycle. The next operator holds if its operand holds in the cycle that immediately follows. It means, the assertion

\[
\text{assert always a \(\Rightarrow\) next b;}
\]

means that whenever the HDL signal a is true, the HDL signal b must be true in the following cycle. The cycle is specified either by defining a default clock or by including the clocking operator @ within the property.

The next operator can take a number of cycles as an argument, enclosed in square brackets, as in:

\[
\text{assert always a \(\Rightarrow\) next[2] (b);}
\]

This means that whenever a is true, b must be true two cycles later.

The meaning of the \texttt{until} operator is explained below:

\[
\text{assert always a \(\Rightarrow\) next (b until c);}
\]

This asserts that whenever a is true, b is true in the following cycle and b remains true until the first subsequent cycle in which c is true.

Finally, the \texttt{before} operator:

\[
\text{assert always a \(\Rightarrow\) next (b before c);}
\]

This asserts that whenever a is true, b must be true at least once in the period starting in the following cycle and ending in the last cycle before c is true.
2.1.5 Sequences

PSL permits longer sequences and families of related sequences to be written in a compact notation. For example:

\[ \{a[^*2]\} \] means \{a;a\}
\[ \{a[+]\} \] means \{a;a;...;a\} with a repeated one or more times
\[ \{a[^*]\} \] means \{a;a;...;a\} with a repeated zero of more times
\[ \{[^*]\} \] matches any sequence whatsoever
\[ \{a[=2]\} \] means \{[^*];a[^*];a[^*]\}, i.e. non-consecutive repetition
\[ \{a[^*1 to 3]\} \] means \{a\} or \{a;a\} or \{a;a;a\}

2.1.6 PSL built-in functions

The modeling layer includes several built-in functions: \texttt{prev()}, \texttt{next()}, \texttt{stable()}, \texttt{rose()} and \texttt{fell}().

\texttt{prev()} returns the value of an expression in the previous cycle and \texttt{next()} in the next cycle. \texttt{rose()} is true if an expression is true in the current cycle and it was false in the previous. \texttt{fell()} is true if an expression is false in the current cycle and it was true in the previous. Function \texttt{stable()} is true if an expression has the same value in the previous and in the current cycle.

2.2 QuestaSim Verification with PSL

EDA tool that has being used during the project is \texttt{MentorGraphics -QuestaSim 6.2g and 6.3a}. During the first phase of the project, the goal was to write simple assertions in the PSL language and to check how QuestaSim implements and support PSL assertions.

QuestaSim operates in either a \texttt{debug mode} (assertions are specified at compile time) or in the \texttt{performance flow} (assertions may be also specified at optimization time in an external PSL file). A debug mode which is used in this project is presented in Figure 2.1. In a debug mode assertions can be embedded in VHDL(Verilog) code or can be written in an external PSL file. If assertions are embedded \texttt{vcom/vlog} will compile them automatically. If the assertions are in a separate file, (this approach is used in this project), we use -\texttt{pslfile} argument for QuestaSim \texttt{vcom} command:

\texttt{vcom <vhdl file> -pslfile <psl file>}

In other words, a vhdl file is compiled together with an external psl file, which is bound to an architecture of the vhdl design unit. When we invoke \texttt{vsim} (command to invoke the simulator) on the top-level of the design, the simulator automatically handles all assertions that are present. While running simulation, one can analyze assertions failures and coverage information with various Questa GUI tools and reports (Wave window, Assertions pane, Cover directives pane....see [2]).
With the optional -assertdebug argument to vsim, we turn on Questa assertion debug tool, which helps us to analyze assertions failures with more details.

For this aim, VHDL design of 8-bit multiplier is used. This multiplier was designed in EPFL LSM during MasterNanotech winter semester EDA lab work. (Source codes of the design, testbench, PSL files and simulations wave diagrams are given in Appendix 2). The design interface includes the following ports: OPA, OPA(N-bit input operands), STB(strobe-control signal that starts the computation when asserted high), CLK(input clock signal), RST_B(input reset signal( active low)), MRES(2*N-bit output multiplication result), DONE(output signal indicating that the value on MRES is equal to OPA*OPB when asserted high). The design is realized with a finite state machine with five states: IDLE(in this state the multiplier waits for the strobe signal STB to start a new computation), INIT(the multiplier performs some initializations), CHECK( the multiplier checks whether it will have to perform an addition or a shift and to check for completion is done), ADD( the multiplier performs an add to accumulate partial sums) and SHIFT( the multiplier shifts operand values).

To validate some of multiplier's properties, already defined assertion patterns were used. (Assertion patterns are a system for documenting in a consistent form and describing commonly occurring assertions found in today's RTL designs and can be ideal as a quick reference for various classes of assertions. [1])

2.2.1 Event bounded window pattern

Sometimes it is necessary to ensure that some signals are bounded by a specified starting event and ending event. In our case, we wanted to check that 'stb' signal is always deasserted before signal 'done' is activated. In the external PSL file multn.psl (Appendix 2, multn.psl) (lines 30-31), it is demonstrated how this property is defined ('done before stb') and asserted:
property done_before_stb is (always stb -> next ((not stb) until done))
@falling_edge(clk);
assert done_before_stb;

(All assertions wave diagrams can be seen in QuestaSim's Simulation Wave Pane using
QuestaSim's 'assertdebug' tool - we can track assertions': Activations (green line), Passes (green
arrow), Failures (red arrow) and Deactivations (blue line)).)(See the A2 Wave Pane 1 picture in
the Appendix 2). It can be seen that this assertion (the assertion for the property 'done before stb')
is activated at a rising edge of the signal 'clk' when 'stb' is active. The assertion stays active until
a 'clk' rising edge when 'done'='1', when it finally passes (green arrow).

2.2.2 FSM state coverage pattern

Without visiting all states of an FSM during the course of verification, potential bugs may
go undetected. It is important during the course of verification to visit all legal states within a
given FSM.

In the external PSL file multn.psl (Appendix 1, multn.psl) (lines 6-10), it is demonstrated
how to cover all the states of the multiplier's FSM (IDLE, INIT, CHECK, ADD and SHIFT):.

-- functional coverage each state entered
cover {state=INIT};
cover {state=CHECK};
cover {state=ADD};
cover {state=SHIFT};
cover {state=IDLE};

2.2.3 State sequence coverage pattern

Without specifying expected state sequences, major functionality within the design could
go unverified. By specifying expected sequences, we can be assured that sufficient verification
has occurred. In multn.psl (lines 13-14) it is demonstrated how to report the occurrence (to
cover) of the sequence: IDLE, INIT, CHECK.

sequence multn_seq is 
{state = IDLE};
{state = INIT};
{state = CHECK});
c_multn_seq : cover {multn_seq};

2.2.4 Time bounded window pattern

Sometimes it is necessary to ensure that some signals are bounded by a specified starting
event and ending event within a specified number of cycles. In our case, we wanted to check that
signal 'done' must occur within 27 cycles after 'stb'.(Depending on number of bits '1' in operands
OPA and OPB, FSM needs a maximum of 27 cycles to finish a two operands multiplying.)
(multn.psl, lines 38-39).

property done_within_27clk is (always ((stb) => 
{[*0 to 27]};{done})))
@falling_edge(clk);
assert done_within_27clk;

To ensure that this property and others are defined properly, (using QuestaSim's
'assertdebug tool' which provides debugging assertions during simulation) we purposely generated certain bugs in our VHDL source code. If we take a look at the BUG1 we will see that due to this bug, FSM will be stucked at state SHIFT instead of going to the next state, the state CHECK. Due to this fact, signal 'done' will be never asserted, that means, that 27 cycles after 'stb', a property 'done_within_27clk' should fail! Indeed, during QuestaSim simulation, this property fails, what can be seen in QuestaSim Simulation Wave Pane using the 'assertdebug' tool. (Appendix 2, Wave Pane 2)

2.2.5 Other properties and some remarks

In the external file tb_multn.psl (which is bound to the testbench (tb_multn.vhd), but not to the design unit under test (! multn.vhd)), it is demonstrated that PSL assertions can be defined even with internal VHDL hierarchical signals, if we use a QuestaSim's function 'spy' and if we define a new VHDL signals in the modeling layer (tb_multn.psl, lines 6,7,12). Unfortunately, it is not possible with VHDL process variables!

type STATES is ( IDLE, INIT, CHECK, ADD, SHIFT);
signal tb_state : STATES;
init_signal_spy( "/tb_multn rtl/UUT/state", "tb_state" );

The property 'correct_result' is checking if a result of multiplication is really equal to the two operands multiplication. (The same functionality is checked in the testbench as well.) To ensure that this property is defined properly, we purposely generated BUG2. Due to this bug, multiplication result will not be correct and the assertion should fail. Indeed, during simulation (see Wave Pane 3) this assertion fails!
Chapter 3
The Semtech environment

3.1 The Semtech Analog Package and PSL assertions

The Semtech had developed a home-made VHDL package that provides VHDL types and routines for modeling analog components. This package allows system-level verification of mixed-signal designs and it keeps using (digital) VHDL as the modeling and simulation language. It provides fast mixed-signal top level simulation of large designs and reuses existing VHDL models of digital blocks.

This package models analog nodes with associated voltage and impedance. An analog node may have many contributors (drivers) which may be generators or loads and these contributors are modeled as a Thevenin or a Norton equivalent model. A resolution mechanism for an analog node is provided to compute the actual voltages and impedances. A current is also associated to each node and it represents the sum of all currents consumed by all load contributors. [Vachoux 2007 [7] ]

The package defines a type ‘analog’ as a record of elements (see Appendix3, analog_pack.vhd): V_value, I_value, Z_value, model, device, nb_gene and nb_load. First three elements are type real. V_value presents resolved node voltage, I_value is sum of all load currents at the node and Z_value is resolved node impedance. model and device present a kind of contribution and they are types: ‘model_type’ and ‘device_type’, already defined in the package. nb_gene and nb_load are numbers of generators and loads connected to node.

Also, the package defines associated procedures for value assignments to ‘analog’ record fields (v_gene, i_gene, high_z...) and conversion functions for conversion from and to std_logic. (See Appendix 3, analog_pack.vhd)

The next step of this work was an attempt to apply simple PSL assertions to signals of ‘analog’ type, and to obtain preliminary results about limits of PSL assertions usage with ‘analog’ signals. For this purpose, a simple VHDL model of a discrete-time model of a 1st-order -lowpass -filter was used. (See A3 LP_filter.vhd).

Using testbench that provides step function input voltage (A3 tb_LP_filter.vhd), we wanted to verify simple analog properties: 1. output voltage doesn't exceed input voltage amplitude, and 2. after time 5RC, filter output voltage reaches 99% of the input voltage.
amplitude. To define this analog properties in PSL, the main idea was to use v_value field of the signal Vout that is type 'analog'. Since this field is type 'real', in the PSL boolean layer it is easy to compare this value with any other real number. (See A3 LP_filter.psl) First property is simply defined that it must hold that output voltage (Vout.v_value) is always less than input voltage amplitude (AMPL)(line 13), and second property is defined that whenever time after Vin setting up is 5RC, then Vout must be bigger than 0.99*AMPL. (lines 9,10, 20):

\[
\begin{align*}
    a & \leq \text{time} > 5.0 \ast R \ast C; \\
    b & \leq \text{Vout.v_value} > \text{AMPL} \ast 0.99; \\
    \text{property Vout_after_5RC is always } (a \text{ and Vin.v_value=AMPL}) \rightarrow b;
\end{align*}
\]

During simulation (A2 Wave Pane 1) we can see that these properties really hold.

### 3.2. The Semtech zooming A/D converter (SX8724)

The main objective of this project is to evaluate how the simulation/verification languages (VHDL, SystemVerilog, PSL) and associated tools may be put to work to improve the existing mixed-signal verification methodology used in Semtech. The evaluation is being done on a medium-complexity mixed-signal design of Semtech A/D converter SX8724. This is a zooming A/D converter with a small custom 12-bit processor and an I2C interface. Verification with PSL, for this converter, is being done with already existing SEMTECH's VHDL testbenches, using external PSL files and applying PSL assertions to already implemented SX8724 design.

The SX8724 is a data acquisition system based on the ZoomingADC™. It directly connects most types of miniature sensors with a general purpose microcontroller. With 3 differential inputs, it can adapt to multiple sensor systems. Its digital outputs are used to bias or reset the sensing elements. (See a functional block diagram below, Figure 3.1) (Datasheet SX8724 [8])

It has: up to 16-bit differential data acquisition and programmable gain: (1/12 to 1000); sensor offset compensation up to 15 times full scale of input signal; 3 differential or 6 single ended signal inputs; Programmable Resolution versus Speed versus Supply current; 4 digital outputs to bias Sensors; Internal or external voltage reference; Internal time base Ultra low-power (250 uA for 16b @ 1 kS/s) and I2C interface.
Figure 3.1: SX8724 functional block diagram

A chip block diagram with main digital design units (blocks):
DIGCKGEN – generates the main digital clocks for the system (non-overlapping dual phase clocks: Digital Master system clock (ckmstr) and Digital Slave system clock (ckslv))
GRISC controller – the heart of the circuit; generates all the control signals required to access any data registers
I2C interface – gives access to the chip registers; it complies with the I2C protocol specifications, restricted to the slave side of the communication
FUSE DIG – is digital interface to the fuse2x8 block used to store I2C device address and the trimming values for the RC oscillator and the bandgap reference
TST – implements test mode selection for other peripherals
GPIO – is in charge of the configuration and control of the digital pads
RCLP – acknowledges a CPU requested for enabling/disabling the RC oscillations,
ADC DIG – digital controller for AD converter
is depicted in Figure 3.2.
The GriscSystem (GRISC, ROM, Address Decoder and I2C) is based on a GRISC micro-controller (grisc core) which block symbol is presented in Figure 3.3. (Block diagram schematics for GriscSystem is given in Appendix 4.)

GriscSystem data bus signals are: **dme** (data memory enable-indicates that the core wants to perform a peripheral data access), **rnw** (read/write signal – indicates whether the data access is a read access (rnw asserted) or a write access (rnw deasserted)), **da** (data memory address- signal bus that holds the address of a peripheral data memory the core wants to access-), **dt2st** (data to store – signal bus to write data into the peripheral data memory) and **dt2ld** (data to load f- signal bus to read from the peripheral data memory).

Timing diagrams for the GRISC data bus interface are presented in Figure 3.4:
Here, we can see that dme, rnw and da must be set up during ckslv high, and they must stay stable until the next rising edge of ckslv. Later in this work, it will be demonstrated how these timing properties can be defined and asserted in PSL.
Chapter 4
PSL assertions for SX8724

4.1 GriscSystem

Using PSL, first goal was to verify the GRISC system's RTL design implementation against its specification requirements of the peripheral interface. The aim was to check if GRISC data bus signals are set up and stable according to the peripheral timing diagrams. (VHDL toplevel source codes and graphics of the GRISC system, and PSL files are given in Appendix 4.)

The property that signal dme is set up during ckslv is high (property 'rising_edge_dme', grisc_system.psl, line 15)) is simply defined that whenever there is rising edge of dme at the same time ckslv must be high:

    property rising_edge_dme is always rose(dme) -> ckslv;

Here, it should be noticed that in order to detect rising edge of dme (using PSL built-in function rose()) it was necessary to use the default clock with higher frequency than ckslv and ckmstr. For this purpose we use virtual clock that is a new signal defined in PSL modeling layer (lines 6-8):

    signal clk: std_logic := '0';
    clk <= not clk after 1 ns;
    default clock is falling_edge(clk);

The property that dme is stable until next ckslv rising edge (property 'dme_stable', grisc_system.psl, line 16) is defined using PSL built-in function stable(). Once dme is set up, it must stay stable until next ckslv rising edge:

    property dme_stable is always rose(dme) -> next stable(dme)
    until rose(ckslv);

Set-up and stable properties for other GRISC's data bus signals are defined in similar way. All assertion wave diagrams are shown in Appendix 4, GriscSystem Postscript Wave Diagrams 1.

Second step was to verify all these, above defined, properties for the SX8724's gate level
model. Since it is synthesized in Verilog gate-level net-list, it was necessary to translate already defined VHDL flavored properties to Verilog flavored properties. These properties are demonstrated in grisc_system.v.psl. Of course, all properties are defined in the same way. It makes only difference to the syntax for the boolean expressions (boolean layer).

### 4.2 analog2std

As we already mentioned, in SEMTECH's analog package there is analog2std conversion function that converts signals of type 'analog' to signals of type 'std_logic'. During top level simulation it is important that input voltage of analog2std converter holds some analog properties.

For example, it is important to verify that input voltage doesn't exceed supply voltage. This property is defined in the same way as we did it for the LP_filter (A4 analog2std.psl, line 11):

```vhdl
property v_input_range is always (supply_pin.v_value >=
node_analog.v_value);
```

Also, it is important to check input voltage rising and falling transition time. Property 'rising_transition_time' (line 15) is defined that once an input voltage exceeds a half of supply voltage, then after a half of transition time, the input voltage must exceed 0.9 of supply voltage:

```vhdl
property rising_transition_time is always rose(node_analog.v_value >
supply_pin.v_value/2.0) -> {{*0 to t_trans/2}; (rose(node_analog.v_value >
supply_pin.v_value*0.9))};
```

Time measuring is provided using a virtual clock with 2ns period. It means that transition time is defined as t_trans number of cycles meaning t_trans*2 ns. To ensure that the property is defined properly we asserted it two times with 2 different time transition constants (t_trans and t_trans2). During simulation we obtained expected results (A4 Wave Pane 1): for the smaller transition time t_trans the assertion passes and for the bigger trans. time t_trans2 the assertion fails.

Property that verifies falling transition time, is defined in similar manner.

Properties 'monotonically_increasing_v_in' and 'monotonically_decreasing_v_in' verify if the input voltage rising and falling edges are monotonically increasing, i.e. monotonically decreasing. They are defined (lines 24, 25, 26) using PSL built-in function prev(), that gives the value of a signal in the previous cycle. In other words, once an input voltage exceeds a half of supply voltage, it must be higher in every next cycle until it reach 0.9 of the supply voltage:

```vhdl
property monotonically_increasing_v_in is always rose
(node_analog.v_value > supply_pin.v_value/2.0) -> (node_analog.v_value >
prev(node_analog.v_value)) until (node_analog.v_value >
supply_pin.v_value*0.9);
```

```vhdl
property monotonically_decreasing_v_in is always fell
(node_analog.v_value > supply_pin.v_value*0.9) -> (node_analog.v_value <
prev(node_analog.v_value)) until (node_analog.v_value <
```
supply_pin.v_value/2.0);

property monotonically_decreasing_v_in2 is always fell
(node_analog.v_value > supply_pin.v_value*0.9) \rightarrow (node_analog.v_value <
prev(node_analog.v_value)) until (node_analog.v_value <
supply_pin.v_value*0.1);

During simulation we obtained expected results according to Vininput and Vsupply values (A4
Wave Pane 1): the properties 'monotonically_increasing_v_in' and
' monotonically_decreasing_v_in' pass, and the property 'monotonically_increasing_v_in2' fails.

Until now we didn't mention, but it is important to understand that quality of property
defined with a virtual clock, strongly depends on virtual clock's frequency. With a higher
frequency we can be more sure that some property really holds! On the other hand, usage of
virtual clock with frequency higher than system clocks can dramatically slow down a simulation!

4.3 GriscSystem stability properties without virtual clock

As it has been explained in Section 4.1, using a virtual clock for defining properties for
the signal stability, will slow down a simulation. If we decrease the frequency of the virtual clock,
glitches, with a time period less than the virtual clock time period, can not be detected. The
problem can be resolved if a property is multi-clocked and/or if a property is clocked with a new
defined signal in the PSL modeling layer. These methods are demonstrated with the data memory
address bus(da) and data memory enable(dme), as it follows.

When clocks are nested, the inner clock causes alignment before the outer clock is
considered. For example, we want to write a property that means the following: if signal a is true
at rose(clk1) (rising edge of clk1), then at the next rising edge of clk2, signal b should be true. We
would write the property like this:

\[ \text{assert always } (a \rightarrow b \text{ @ rise(clk2)}) \text{ @ rise(clk1);} \]

and it would be equivalent to: \[ \text{assert always } (a @ clk1 \rightarrow b @ clk2). \]

If a signal c is defined as \( c <= \text{not } a \text{ or } b \), and if we write a property as:

\[ \text{assert } (\text{always } a \rightarrow b \text{ @ rise(c)}) \text{ @ rise(a);} \]

it means that the property is activated at a rising edge of a (LHS is clocked by a), and RHS is
evaluated at the next rising edge of c (RHS is clocked by c). It means RHS is evaluated at: the
next rising edge of b or the next falling edge of a. (See the definition of c.). At the former the
RHS (i.e the property) passes, and at the latter the property fails. It means, if signal a is stable
until the next rising edge of b, the assertions is true, and if a is not stable until the next rising edge
of b, the assertion is false. In other words, this assertion is checking stability of the signal a until
the rose(b). (Here it should be noticed that for any signal x, the property: \( x @ \text{rise(x)} \)
always holds (is always true) and the property \( x @ \text{fall(x)} \) is always false. Later it will
be shown if these are defined in System Verilog, the former is always false and the latter is always true!)

Therefore, using the same idea from the previous paragraph, the properties **rising_edge_dme** and **dme_stable**, introduced in Section 4.1, could be defined as:

```
property rising_edge_dme is (always ckslv and not pcc) @rising_edge(dme);

signal clk_dme: std_logic:= '0';
clk_dme <= (not dme or pcc);
property dme_stable is (always ckslv and not pcc -> pcc@rising_edge(clk_dme)) @rising_edge(dme);
```

The property **rising_edge_dme** is clocked by **dme**, instead of using a virtual clock. It means at a rising edge of **dme**, **ckslv** must be high and **pcc** must be low, i.e. **dme** must be set up during **ckslv** high and **pcc** low. (See Figure 4.X) The property **dme_stable** means: when **dme** is set up during **ckslv** high and **pcc** low (LHS clocked by **dme**) it must stay stable until rising edge of **pcc**. (RHS is clocked by **clk_dme**, which is defined as **clk_dme <= (not dme or pcc)**). During a simulation in QuestaSim wave diagram (Figure 4.1), the assertion is activated at rose(**dme**) and since **dme** is stable until rose(**pcc**), the assertion passes at rose(**pcc**), as it is expected. If **dme** is not stable, as it is expected, the assertion fails at fall(**dme**)  

![Figure 4.1: rising_edge_dme and dme_stable](image)

Precondition (LHS) in **dme_stable** is used to avoid the assertion activation every time when there is a rising edge of **dme**. In this way the assertion will be active only when **dme** is set up properly. Therefore, there will no be too often assertion evaluations and the influence on simulation time will be less.

Stability of other Grisc data bus signals can be written in a similar way. (All stability properties and wave diagrams are given in Appendix 4.) To define stability property for bus signals, it is necessary to detect an event on a bus signal what later can trigger assertion's pass/fail. To this aim, VHDL signal attribute 'event' may be used. Therefore, the properties **setting_da** and **da_is_stable**, introduced in Section 4.1, could be written as:

```
signal da_event: std_logic := '0';
da_event <= '1', '0' after 1 ns when (da_event) else '0';
signal clk_da: std_logic:='0';
clk_da <= da_event or pcc;
```
property setting_da is (always (ckslv and not pcc) ->
(ckslv and not pcc)@rising_edge(clk_da))@rising_edge(dme);
property da_is_stable is (always (dme and ckslv and not pcc) ->
next pcc)@rising_edge(clk_da);

(The assertions wave diagrams are given in Figure 4.2.) Here, two new signals are
defined: da_event is going high when there is an event on da, and clk_da going high when there
is a rising edge of da_event or a rising edge of pcc. Therefore, the first property (setting_da) is
activated on a rising edge of dme (LHS is clocked by dme) when ckslv is high and pcc is low,
and RHS is evaluated (RHS is clocked by clk_da) on the next rising edge of clk_da. If the next
clk_da rising edge is triggered by rose(da_event) during ckslv high the assertion will pass. If the
next clk_da rising edge is triggered by rose(da_event) during ckslv low the assertion will fail. It
means that this assertion is checking if da is set up after dme is setup properly, and before ckslv
goes low. The second property (da_is_stable) is activated when da is set up properly (LHS is
equal to RHS of the previous property) and RHS will be checked at the next rose(clk_da). If rose
(clk_da) is triggered by rose(pcc) the assertion will pass. If rose(clk_da) is triggered by rose
(da_event) (meaning there is an event on da before rose(pcc)) the assertion will fail. It means
that this property is really checking if da is stable until the next rising edge of pcc.

![assert_setting_da](image)

Figure 4.2: setting_da and da_is_stable

4.4 Grisc System interface protocol – protocol properties

In this Section after a short description of the Grisc System data bus interface protocol, it
will be demonstrated, using this protocol, how PSL can be used to define protocol properties.

The Grisc Core (micro controller) with the data bus (dme, rnw, da[7:0], dt2st[7:0], dt2ld
[7:0]) is connected to blocks (peripheral data memories): I2C, FUSE DIG, RCLP, GPIO, ADC
DIG, VREF and TST, addressed from 1 to 8. The blocks are addresses by 4 MSB of da (block
address=da[7:4]), and the blocks' registers are addressed by 4 LSB of da (block's register
address=da[7:4]). dt2st is '8-bit std_logic_vector' and dt2ld is an array of 8 '8-bit std_logic_vectors'. (dt2ld[7] is bus for I2C, dt2ld[6] is bus for FUSE DIG, dt2ld[5] is bus for
When Grisc (Grisc Core) receives an event from I2C (event high), in the first cycle it reads the I2C status register. (dme=1, rnw=1, da[7:4]=1, da[3:0]=1), to check if I2C wants to perform peripheral data reading or peripheral data writing. (I2C, addressed with 1 (da[7:4]=1), has 3 registers: status register (addressed with 0 (da[3:0]=0), address register (addressed with 1 (da[3:0]=1) and data register (addressed with 2 (da[3:0]=2).) After the first cycle, i.e. after a reading I2C’s status register, in the next cycle (2nd) the core will read I2C’s address register (dme=1, rnw=1, da[7:4]=1, da[3:0]=1). The address register keeps the address of a peripheral data memory where I2C wants to perform reading or writing. If I2C wants to perform reading, in the 3rd cycle the core will read the addressed peripheral data memory, and in the 4th cycle the read data will be written in the I2C’s data register. If I2C wants to perform writing, in the 3rd cycle the core will read the I2C’s data register, and in the 4th cycle the read data will be written in the addressed peripheral data memory. In the final 5th cycle the core will write “x02” to the I2C’s status register, to acknowledge the end of the requested reading(writing) event. After this, I2C must deassert signal event.

Therefore, a property for the 1st cycle may be defined as:

```vhdl
property status_register_reading is (always rose(event_int) ->
  (not dme; dme='1' and rnw='1' and da_int=x"10") )
@rising_edge(pcc);
```

meaning when there is rose(event_int) (event_int is asserted - event_int is grisc’s internal signal name for event), in the next property clock cycle dme and rnw must be asserted (the core performs reading) and da_int (internal name for da) must be x”10” (the core performs reading from I2C’s status register. (Figure 4.3)

![Figure 4.3: status_register_reading](image)

To avoid writing unreadable properties, new signals (add_ind – address index, reg_ind – register index) may be defined in PSL modeling layer using type casting from std_logic_vector to integer:

```vhdl
signal add_ind: integer:= 1;
add_ind <= to_integer(unsigned(da_int(7 downto 4)));

signal reg_ind: integer:= 0;
reg_ind <= to_integer(unsigned(da_int(3 downto 0)));
```

A property for the 2nd cycle may be defined as:
property address_register_reading is (always reg_ind=0 and prev(reg_ind)=0 and rnw='1' and d2ld_bus_int(8 – add_ind)(1)='1' -> next rnw='1' and add_ind=prev(add_ind) and reg_ind=1)
@rising_edge(dme);

Figure 4.4: address_register_reading

LHS checks if the event is coming from a currently checking peripheral unit (in Grisc System events are always coming from I2C): \( d2ld\_bus\_int(8 – add\_ind)(1) = '1' \). Therefore, the assertion is activated (Figure 4.4) when the core is performing the 1\(^{st}\) cycle, and in the next cycle the property is checking if grisc core reads the address register \( – \) \( rnw \) must be high and \( reg\_ind \) must be 1. Since the cycles are synchronized with \( dme \), the property is clocked by \( dme \).

A property for the 3\(^{rd}\) cycle, when I2C wants to perform reading, may be defined as:

property data_register_reading_in_re is (always { (rnw='1' and reg_ind=0 and d2ld_bus_int(8 – add_ind)(0)='1'); rnw='1' and reg_ind=1 } =>
rnw='1' and da_int = prev(d2ld_bus_int(8 – (add_ind) ) ) )
@rising_edge(dme_psl);

LHS sequence means: if during the 1\(^{st}\) cycle I2C requested reading \( (rnw='1' \) and reg\_ind=0 and d2ld_bus_int(8 – add\_ind)(0)='1' \) – LSB of the I2C's status register defines reading/writing), after reading the address register \( (rnw='1' \) and reg\_ind=1 \) the assertion will be activated. (Figure 4.X) If LHS condition is satisfied in the next cycle the core must perform reading (RHS: \( rnw='1' \)) of the peripheral data memory addressed by I2C address register in the previous cycle \( (da\_int = prev(d2ld\_bus\_int \ (8–(add\_ind) ) ) \)). (In Figure 4.5 we can see that in 2\(^{nd}\) cycle the core reads x"51" from the I2C address register and in the next cycle the assertion passes since the core reading the peripheral data memory addressed with x"51".)
Here, we can see that protocol properties may be less readable and understandable. Also, due to using sequences in LSH (to ensure appropriate assertion activation after a number of cycles), assertions can be active for several cycles, what could cause simulation being slowed down.

4.5 ADC digital controller

ADC DIG, or ADC digital controller generates control signals for the AD conversion acquisition chain, and stores the analog-to-digital conversion (ADC) result. (See SX8724 Datasheet [8] for full ADC DIG description and Appendix 4 for Block diagram schematics) To ensure ADC DIG's main functionality, properties for the control signals are defined as it follows.

After the core performs data writing to ADC DIG's 8-bit registers, i.e. after ADC DIG's registers are configured, ADC DIG generates control signals. The generated control signals must correspond to the last written registers data. To this aim, a property for the control signal for the analog input multiplexer (amux) is defined as:

```vhdl
property analog_input_multiplexer is (always (cs='1' and address(2 downto 0)="111") -> next amux=prev(datain(5 downto 1)) and (next stable(amux) until (cs='1' and address(2 downto 0)="111") )@falling_edge(ckslv));
```

It means: when the data is written to the corresponding register (cs='1' and address(2 downto 0)="111") - LHS, in the next cycle the control signal amux must be equal to the last register written value (amux=prev(datain(5 downto 1))) and it must keep the value (stay stable) until the next configuration (next stable(amux) until(cs='1' and address(2 downto 0)="111") (address is internal da name, and the registers are addressed from 0 to 7. Register RegACCfg5, is addressed with 7(address(2 downto 0)
and bits with position from 5 downto 1 (RegACCfg5[5:1]) are defining amux's value. (See Appendix A)

Figure 4.6: analog_input_multiplexer

Properties for other control signals are written similarly. (See Appendix 4). It should be noticed that the assertion will be evaluated at every ckslv falling edge and it will stay active until the next register configuration.

MSB of ADC result are stored in RegACOutMSB \( \text{address(2 downto 0)="001"} \) and LSB of ADC result are stored in RegACOutLSB \( \text{address(2 downto 0)="000"} \). When the core is reading RegACOutMSB \( \text{cs='1' and readwrite='1' and address(2 downto 0)="001"} \) the register data must be equal to the 16 MSB of the ADC result \( \text{dataout=adc_out(15 downto 8)} \). Therefore property \( \text{adc_result_MSB} \) is written as:

```vhdl
property adc_result_MSB is (always cs='1' and readwrite='1' and address(2 downto 0)="001" -> dataout=adc_out(15 downto 8)) @falling_edge(ckslv);
```

In similar way, property \( \text{adc_result_LSB} \) is defined as:

```vhdl
property adc_result_LSB is (always cs='1' and readwrite='1' and address(2 downto 0)="000" -> dataout=adc_out(7 downto 0)) @falling_edge(ckslv);
```

In Figure 4.7 it is shown that the assertion \( \text{adc_result_LSB} \) really passes since \text{dataout} is equal to \( \text{adc_out[15:0]="01010100"} \).

Figure 4.7: adc_result_LSB

4.6 PGA
For the three differential programmable gain amplifiers (PGA), depicted in Figure 1, reference [8], gains for PGA 1, 2 and 3, and offsets for PGA 2 and 3 are programmed according to the ADC DIG registers’ configuration. During top level simulation it is important to ensure that gains and offsets values really match with expected programmed values. To define these properties, again we have to deal with analog signals from analog package, since the analog part of the zooming AD converter is modelized using Semtech analog package.

To check if the gain of PGA1 is 10, +/- 0.05, the property could be written as:

```vhdl
property p_pga1_gain_10 is (always gain_pga1='1' ->
abs(voutp_pga1.v_value-voutm_pga1.v_value) >
9.95*abs(vinp_pga.v_value-vinm_pga.v_value) and
abs(voutp_pga1.v_value-voutm_pga1.v_value) <
10.05*abs(vinp_pga.v_value-vinm_pga.v_value))@falling_edge(s1);
```

meaning: if gain_pga1 (controling signal for PGA1 gain), is '1', the real gain during simulation must be 10 (see [8], Table 1) with the 5% error. RHS condition is defined comparing the real output voltage difference (voutp_pga1.v_value-voutm_pga1.v_value) with the expected, concerning input voltage difference (vinp_pga.v_value-vinm_pga.v_value). abs is math function coming from VHDL flavor.

Figure 4.8 shows that the assertion is true, since the output voltage difference, |voutp_pga1-voutm_pga1|, is less than 10.05*|vinp_pga-vinm_pga| and bigger than 9.95*|vinp_pga-vinm_pga|.

Properties for other possible gains, errors and PGA must be defined as well (all possible cases must be covered) and they are presented in Appendix 4. Also properties for the input and reference multiplexers, checking the main functionality - multiplexing, are defined corresponding to vmux and amux values (ADC DIG control signals).

During simulation it was noticed that all gain property assertion fail when there is a very small input voltage difference. To avoid these false assertion failures, the properties could be improved. For example, in LHS a condition for no small input voltage difference may be added. In such way, assertions will not be activated when the input voltage difference is equal or almost
equal to zero.

4.7 ADC result

The ADC output code is a 16-bit word in two's complement format (see [8], Table 15). For input voltages outside the range, the output code is saturated to the closest full-scale value (i.e. 0x7FFF or 0x8000). For resolutions smaller than 16 bits, the non-significant bits are forced to the values shown in Table 16, Appendix A. The output code, expressed in LSBs, corresponds to:

$$\text{OUT}_{\text{adc}} = 2^{16} \times V_{\text{in}} \times (G_{\text{D tot}} - G_{\text{D off tot}} \times V_{\text{ref}} / V_{\text{in}}) \times (\text{OSR} + 1) / \text{OSR} / V_{\text{ref}}$$

Using this formula we wanted to define properties ensuring that ADC output code corresponds to the input voltage, i.e. the input signal is correctly converted into digital. They are defined for the Vref=5V, 16 bits resolution (OSR=100,101...111) and the total PGA offset=0 (in similar way they can be defined for other configurations.) Here, we demonstrate property \text{adc\_result\_1} that checks if the adc error is less than 1%. (Definitions for all other properties are given in Appendix 4)

property \text{adc\_result\_1} is ( always \text{adc\_out\_converted} > 0.99 * prev (\text{adc\_out\_expected}) and \text{adc\_out\_converted} < 1.01 * prev(\text{adc\_out\_expected}) )@rising_edge(spy\_busy);

A new conversion is starting with the rising edge of \text{spy\_busy} (see below for the \text{spy\_busy} definition). It means the previous ad conversion, (started with previous \text{spy\_busy} rising edge) is finished. Therefore, we are comparing \text{adc\_out\_converted} with \text{adc\_out\_expected} from the previous cycle. (Figure 4.10)

Figure 4.10: \text{adc\_result\_1}

For the output signals that we want use for new signals defining, and for the signals outside of ADC DIG, QuestaSim \text{spy} function must be used. \text{Spy} function for output signals must be use, since output signals can not be used in PSL modeling layer. \text{pga1\_gain}, \text{pga2\_gain} and \text{pga3\_gain} are output signals in ADC DIG (\text{adc\_dig\_dl.vhd}), which the PSL file \text{adc\_dig\_dl.psl} is bound to. Signal \text{busy} is coming from another VHDL unit \text{pga\_top.vhd}.

```vhdl
signal spy_busy: std_logic; -- := '0';
signal spy_vinp: real; -- := 0.0;
signal spy_vinn: real; -- := 0.0;
signal spy_gain1: std_logic; -- := '0';
signal spy_gain2: std_logic_vector(1 downto 0); -- := "00";
signal spy_gain3: std_logic_vector(6 downto 0); -- := "00000000";
signal spy_s1: std_logic; -- := '0';
```

25
Therefore new signals for oversampling ratio (\texttt{osr\_int}), PGA1 gain (\texttt{gain1}), PGA2 gain (\texttt{gain2}) and PGA3 gain (\texttt{gain3}) can be defined as:

```c
signal osr\_int: integer:=8;
osr\_int <= 8 * 2**(to\_integer(unsigned(osr\_i)));

signal gain1: integer:=1;
gain1 <= 1 when spy\_gain1='0' else
  10 when spy\_gain1='1';

signal gain2: integer:=1;
gain2 <= 1 when spy\_gain2='00' else
  2 when spy\_gain2='01' else
  5 when spy\_gain2='10' else
  10 when spy\_gain2='11';

signal gain3: real:=0.0;
gain3 <= real(to\_integer(unsigned(spy\_gain3)))/12.0;
```

Eventually, concerning the output code formula, \texttt{adc\_out\_converted} and \texttt{adc\_out\_expected} are defined as:

```c
signal adc\_out\_expected: real:=0.0;
adc\_out\_expected <= round( 2.0**16 * (spy\_vinp-spy\_vinm) * real(gain1 * gain2) * gain3 * real(osr\_int + 1) / real(osr\_int) / 5.0);

signal adc\_out\_converted: real:=0.0;
adc\_out\_converted <= real(to\_integer(signed(adc\_out)));
```
Chapter 5
System Verilog assertions for SX8724

At this point of the work, the goal was to write SV assertions for GriscSystem and ADC DIG, to compare them with PSL assertions and to make preliminary conclusions about an advantage and a disadvantage of SV assertions usage against PSL assertions.

After defining properties for SX8723 RTL design, we modified them for VHDL gate-level net-list. We wanted to avoid modifying them for Verilog gate-level net-list, i.e. changing PSL properties from VHDL flavor to Verilog flavor. Therefore already existing Verilog net-list was converted to VHDL net-list. Since after synthesis some bus signals are split, it was necessary to do redefinition for some bus signals in PSL modeling layer and net-list, and to slightly change property definitions. At that point, the advantage of using SystemVerilog assertions instead of PSL assertions, was more clear. Since System Verilog assertions can be written in a separate program or module, and this program(module) can be bound to VHDL entity(architecture) or Verilog module, it is not necessary to modify properties for doing gate-level simulation.

Therefore properties for GriscSystem and ADC DIG (presented in Chapter 4), has been defined again in SystemVerilog. Properties for GriscSystem are written in a separate file (System Verilog Module - sva_grisc_system_props.sv):

module sva_grisc_system_props (
  input ckslv, dme, pcc, rnw,
  input [7:0] cs_bus_int, da_int, d2st, d2ld);

  // (see Appendix 5 sva_grisc_system_props.sv);
endmodule

and properties for ADC DIG are written in the module:

module sva_ac_dig_dl_props (
  input ckslv, cs, vmux, pgal_gain, readwrite, cont, ac_irq,
  input [7:0] address, datain, dataout,
  input [4:0] amux,
  input [6:0] pga3_off, pga3_gain,
  input [1:0] fin, ib_amp_pga, ib_amp_adc, nelconv, pga2_gain,
  input [3:0] pga2_off, enable,
  input [15:0] adc_out,
  input [2:0] osr
);

27
The System Verilog assertions in these System Verilog modules apply to VHDL signals in VHDL entities. The binding for these modules to the VHDL entities `grisc_system.vhd` and `acDig_dl.vhd` is defined in a separate file (System Verilog Module - `sx8723_binds.sv`), using SystemVerilog feature `bind`:

```systemverilog
module sx8723_binds;

bind grisc_system sva_grisc_system_props sva_grisc_system_props_bind (.
    .ckslv(ckslv),
    .dme(dme),
    .pcc(pcc),
    .rnw(rnw),
    .da_int(da),
    .d2ld(d2ld),
    .d2st(d2st),
    .cs_bus_int({cs_bus_int_7, cs_bus})
);

bind acDig_dl sva_acDig_dl_props sva_acDig_dl_props_bind (.
    .ckslv(ckslv),
    .cs(cs),
    .amux(amux),
    .address({address_2y, address_1y, address_0y}),
    .datain(datain),
    .vmux(vmux),
    .pga3_off(pga3_off),
    .pga1_gain(pga1_gain),
    .pga3_gain(pga3_gain),
    .fin(ac_freq),
    .adc_out(adc_out),
    .dataout(dataout),
    .ib_amp_adc(ib_amp_adc),
    .ib_amp_pga(ib_amp_pga),
    .nelconv(nelconv),
    .osr(osr),
    .pga2_gain(pga2_gain),
    .pga2_off(pga2_off),
    .readnwrite(readnwrite),
    .cont(cont),
    .enable(enable),
    .ac_irq(ac_irq)
);
endmodule
```

Here, it should be noticed that binding of System Verilog modules to VHDL entities, is a QuestaSim feature for Mixed-Language Simulation! The allowed VHDL types for port mapping to SystemVerilog port vectors are: `bit, bit_vector, std_logic, std_logic_vector, vl_logic` and `vl_logic_vector`.

Unfortunately there is no mapping for signals of type real. (This mixed language feature is planned for a future release.) Therefore, it was not possible to write the analog signal properties
(presented in Sections 4.2 and 4.6) in System Verilog. One possible solution could be converting from **real** to **integer** on the VHDL side and from **integer** to **real** on the SystemVerilog size. (This method is presented in Chapter 6, where the VHDL entity is directly instantiated in a System Verilog module. Instantiation VHDL entities in SystemVerilog modules and vice versa, is also a QuestaSim feature for Mixed-Language Simulation) Another solution, the better one, is by using SV standard System tasks/functions $\text{bitstoreal} / $\text{realtobits} as well as $\text{bitstoshortreal} / $\text{shortrealtobits} on the SystemVerilog side and a user generated FLI function that does the equivalent. Then at the language boundary we pass bits/ vectors across. (Unfortunately VHDL does not yet have a standard procedure / function for converting from bits to real type and vice versa).

We can see that 3 LSB of System Verilog signal **logic** [7:0] **address** is directly bound to 3 VHDL signals **std_logic** **address_2y**, **address_1y**, **address_0y**. It means that binding between vectors(arrays) and scalars are possible using SystemVerilog concatenation {}.

Finally, our testbench (**sx8723_tb.vhd**) is simulated invoking **vsim** command in this way:

```
vsim sx8723_tb sx8723_binds
```

where **sx8723_tb** is our testbench’s entity name. (In Appendix 7 the Unix shell script for sx8723 testbench simulation is presented. The script provides simulation with PSL and SystemVerilog assertions and with QuestaSim's Assertdebug tool.)

### 5.1 Grisc System

Here we demonstrate how the property **dme_stable**, introduced in Chapter 4, can be defined in System Verilog. Other properties for Gris System in ADC DIG in System Verilog are given in Appendix 5.

If we would follow the same methodology presented in Section 4.3, the property in SystemVerilog would be written:

```vhdl
property dme_stable;
  @(dme or posedge pcc) (ckslv && !pcc) |=> dme;
endproperty
```

the System Verilog assertions can be clocked by more than one signal. Therefore if we write @ (dme or posedge pcc), it means that assertions will be clocked by **dme** rising or falling edge, or **pcc** rising (**posedge**) or falling (**negedge**) edge. It means that here it is not necessary to define a new signal **clk_dme**, as it was necessary in PSL.

On the other hand, in SystemVerilog, due to new division of time slot, property **x@ (posedge x)** is always false, and property **x@(negedge x)** is always true. Therefore to define the property correctly, a possible solution could be to define new signals (**dme** and **pcc** delayed):
logic dme_c, pcc_c;
always begin
  dme_c = #1ns dme;
  pcc_c = #1ns pcc;
end

Now the property can be defined properly as:

property dme_stable;
  @(dme_c or posedge pcc_c) (ckslv && !pcc) |=> dme;
endproperty

Indeed (Figure 5.1), during simulation it is shown that the property is really checking if the signal dme is stable until pcc rising edge.

![Figure 5.1: dme_stable](image)

Also, if the System Verilog assertion is clocked by vector, it will be clocked by any event on this vector. Therefore the properties setting_da and da_is_stable (Chapter 4) can be define without previously defining a new signal using VHDL attribute 'event:

property setting_(da_int);
  @(posedge dme) ckslv && !pcc |=> @(da_int or posedge pcc) ckslv && !pcc;
endproperty

property _is_stable(da_int);
  @(da_int or posedge pcc_c) dme && ckslv && !pcc |=> pcc;
endproperty

RHS of the first property is clocked by @(da_int or posedge pcc), meaning it is clocked whenever there is a rising edge of pcc or any event on da_int. (Note, that in the previously presented binding, VHDL 8-bit std_logic_vector da is mapped to SystemVerilog vector logic [7:0] da_int.)

5.2 ADC digital controller

Here we demonstrate, using ADC DIG properties as example (Section 4.5), how SystemVerilog allows us to define a named property declaration with arguments, which facilitates property reuse.
Properties for the register RegACCfg1 controlling signals `ib_amp_adc`, `ib_amp_pga` and `enable` (correspond to the register fields, See [8]), have to be written in PSL separately:

```verilog
property p_ib_amp_adc is (always (cs='1' and address(2 downto 0)="011") ->
next ib_amp_adc=prev(datain(7 downto 6)) and
(next stable(ib_amp_adc) until (cs='1' and address(2 downto 0)="011") ) )
@falling_edge(ckslv);
assert p_ib_amp_adc;

property p_ib_amp_pga is (always (cs='1' and address(2 downto 0)="011") ->
next ib_amp_pga=prev(datain(5 downto 4)) and
(next stable(ib_amp_pga) until (cs='1' and address(2 downto 0)="011") ) )
@falling_edge(ckslv);
assert p_ib_amp_pga;

property p_enable is (always (cs='1' and address(2 downto 0)="011") ->
next enable=prev(datain(3 downto 0)) and
(next stable(enable) until (cs='1' and address(2 downto 0)="011") ) )
@falling_edge(ckslv);
assert p_enable;
```

In System Verilog, only one named property with arguments (`fields`, `hi` and `lo`) can be defined:

```verilog
property p_RegACCfg1(field,hi,lo);
@ (negedge ckslv) cs && address[2:0]=3'b011 |=> field=$past(datain [hi:lo]) and (1 ##1 $stable(field) throughout (cs && address[2:0] ==3'b011)[-1]);
endproperty
```

where `field` is a controlling signal ( `ib_amp_adc`, `ib_amp_pga` and `enable`), `hi` and `lo` are MSB and LSB bit position in RegACCfg1. (PSL expression `a until b` is equivalent to the SystemVerilog expression `a throughout b[->1] !!!`)  

Now, this parametrized property can be asserted (instantiated) 3 times with different argument values:

```verilog
assert property (p_RegACCfg1(ib_amp_adc,7,6));
assert property (p_RegACCfg1(ib_amp_pga,5,4));
assert property (p_RegACCfg1(enable,3,0));
```

To define property `adc_result` (Section 4.7) signals `std_logic busy`, and `real vinm, vimp` from `pga_top.vhd` must be “spied” (QuestSim `spy` function). It is possible to “spy” `std_logic` signals, but unfortunately it is not possible to “spy” `real` signals. This is a consequence of the fact that in QuestaSim there is no mapping for `real` signals. This mixed language feature is planned for a future release.

Therefore, at this place, we can not define `adc_result`, as it was done in Section 4.7. (In Chapter 6 it is demonstrated how this main chip functionality can be checked in the Scoreboard of the SystemVerilog layered testbench.)
Chapter 6
SystemVerilog testbench for SX8724

In this Chapter we very briefly explain how SV testbench for our SX8723 can be written, using below explained methodology. For the complete testbench understanding a reader must be knowledgeable about SystemVerilog and about SX8723 chip functionality!

6.1 Introduction to SV for Verification

SystemVerilog (IEEE Std. 1800-2005) is set of extensions to the Verilog Hardware Description Language (IEEE Std. 1364-2005). SystemVerilog is used to simulate and verify the functionality of digital circuits at levels of abstraction ranging from system level, stochastic and pure behavior down to gate and switch level, and is also used to synthesize gate level description from more abstract RTL description. [9]

Some of the typical features of SystemVerilog, as a Hardware Verification Language (HVL), that distinguish it from a Hardware Description Language such as Verilog or VHDL are:

- Constrained-random stimulus generation
- Functional coverage
- Higher-level structures, especially Object Oriented Programming
- Multi-threading and interprocess communication
- Support for HDL types such as Verilog's 4-state values
- Tight integration with event-simulator for control of the design

These features allow us to create test-benches at a higher level of abstraction than it is possible to achieve with an HDL or programming language such as C [10]. The purposes of a testbench is to determine the correctness of the design under test(DUT). This is accomplished by the following steps: generate stimulus, apply stimulus to the DUT, capture the response, check for correctness and measure progress against the overall verification goals.

Traditionally, directed tests are used, where we write a verification plan with a list of tests, each of which concentrated on a set of related hardware specification features. According to this plan, we write stimulus vectors and simulate the DUT with these vectors and manually
review the resulting log files and waveforms to make sure the design does what we expect. Once the test works correctly we move to the next one in the verification plan.[10]

Figure 6.1 shows how directed tests incrementally cover the features in the verification plan. While we are making forward progress, the slope remains the same. If the design complexity doubles, it takes twice to complete the verification. Therefore, we need a new methodology that finds bugs faster in order to reach the goal of 100% coverage. [10]

![Figure 6.1: Constrained-random test progress](image)

Instead of using directed testing, with SystemVerilog features previously explained, we can make random tests. With constrained-random generation we generate constrained-random stimulus. With Object-oriented programming and multi-threading we build layered testbench with transactors.

A directed test finds the bugs we expect but a random test can find bugs we never expect. When using random stimulus, we need functional coverage to measure verification progress. A layered testbench helps us control the complexity by breaking the problem into manageable pieces. These pieces are Transactors. With an appropriate planning, we can build a testbench infrastructure that can be shared by all tests without having to be modified frequently. Code specific to a single test must be kept separate from the testbench so it does not complicate the infrastructure.[10]

Building a layered testbench takes longer than a traditional directed testbench. On the other hand, every test shares this common layered testbench, as opposed to directed tests where each is written from scratch. Each random test contains multiple random test. Each random test contains lines of code to constrain the stimulus in a certain direction and cause any desired exception. The result is that the a single constrained-random testbenck is finding bugs faster than directed one.

Figure 6.2 shows the layered testbench. At the bottom is the signal layer that contains the design under test and the signals that connect it to the testbench. The next level up is the command layer. The DUT's inputs are driven by the driver that runs single commands. The DUT's output drives the monitor that takes signal transitions and groups them together into
commands. Assertions cross the command/signal layer. The functional layer feed the command layer. The agent block receives higher-level transactions and breaks them into individual commands. These commands are also sent to the scoreboard that predicts the result of the transaction. The checker compares the commands from the monitor with those in the scoreboard. The functional layer is driven by the generator in the scenario layer. The test contains the constraints to create the stimulus. Functional coverage measures the progress of all tests in fulfilling the requirements in the verification plan. [10]

The number of layers depends on our DUT. For simple designs we can merge some layers. Also, connections between blocks are not strict. Our testbench can have a different set of them. Depending on our needs, we can build testbenches with different number of layers and with different connections.

![Figure 6.2: Full testbench with all layers](image)

The Generator, Agent, Driver, Monitor, Checker, and Scoreboard are all *classes*, modeled as transactors (described below). They are instantiated inside the Environment class. The test is at the top of the hierarchy, as is the *program* that instantiates the Environment class.

### 6.2 SV layered testbench

In this Chapter we very briefly explain how SV testbench for our SX8723 can be written, using previously explained methodology. For the complete testbench understanding a reader must be knowledgeable about SystemVerilog and about SX8723 chip functionality!
6.2.1 Wrapper

QuestaSim multi-language simulation feature allows us to instantiate VHDL design under test directly inside SV testbench. Unfortunately, since there is no mapping for real signals (see Chapter 5), it is necessary to build wrapper for our SX8723 VHDL design instantiation in SV. One possible solution could be as it follows. (All pins are modelized as analog signals.)

For the digital pins (D0, D1, D2, D3, SCL, SDA and READY), using one directional and bidirectional converters and conversion functions from the analog package, analog signals are converted to std_logic:

    analog-to-digital converters
    i_ready_conv : entity lib_xe_project.analog2std(bhv)
    port map(node_analog => pad_ready, node_logic => pad_ready_dig, supply_pin => pad_vdd);
    i_d0_conv : entity lib_xe_project.bidir_conv_supply(bhv)
    port map(node_analog => pad_d0, node_logic => pad_d_dig(0), supply_pin => pad_vdd);
    i_d1_conv : entity lib_xe_project.bidir_conv_supply(bhv)
    port map(node_analog => pad_d1, node_logic => pad_d_dig(1), supply_pin => pad_vdd);
    i_d2_conv : entity lib_xe_project.bidir_conv_supply(bhv)
    port map(node_analog => pad_d2, node_logic => pad_d_dig(2), supply_pin => pad_vdd);
    i_d3_conv : entity lib_xe_project.bidir_conv_supply(bhv)
    port map(node_analog => pad_d3, node_logic => pad_d_dig(3), supply_pin => pad_vdd);
    pad_scl_dig_out <= ana2std(pad_scl, pad_vdd.v_value);
    pad_sda_dig_out <= ana2std(pad_sda, pad_vdd.v_value);

Now, these std_logics can be mapped to logic on SystemVerilog (testbench) side.

The analog pins (AC2, AC3, AC4, AC5, AC6, AC7, VSS, VPUMP and VBATT), are driven by v_gen procedure from analog package:

    v_gene(pad_vdd, vdd_v, vdd_z);
    v_gene(pad_vpump, vpump_v, vpump_z);
    v_gene(pad_vss, vss_v, vss_v);
    v_gene(pad_ac2, ac2_v, ac2_z);
    v_gene(pad_ac3, ac3_v, ac3_z);
    v_gene(pad_ac4, ac4_v, ac4_z);
    v_gene(pad_ac5, ac5_v, ac5_z);
    v_gene(pad_ac6, ac6_v, ac6_z);
    v_gene(pad_ac7, ac7_v, ac7_z);

The voltages (vdd_v, vss_v, ac2_v...) and impedances (vdd_z, vss_z, ac2_z...) are reals and driven by SV testbench. Therefore, these reals must be converted from real to integer on SV side, and after from integer to real on VHDL side:

    vdd_v <= real(vddv) /10.0e7;
    vdd_z <= real(vddz) /10.0e7;
    vpump_v <= real(vpumpv)/10.0e7;
    vpump_z <= real(vpumpz)/10.0e7;
    vss_v <= real(vssv) /10.0e7;
    ac6_v <= real(ac6v) /10.0e7;
    ac6_z <= real(ac6z) /10.0e7;


35
c7_v  <= real(ac7v) /10.0e7;
ac7_z  <= real(ac7z) /10.0e7;

(For the conversion factor 10.0e7 the maximum voltage/impedance is 214V/214Ohm.)

6.2.2 Top module

In module top, VHDL unit “sx8723_for_sv_tb” (i.e. wrapper) and program test are instantiated. Signals, connecting them, are defined as well. At this place real signals are converted to integer.

module top;

// ... (see Appendix 6, top_sv_vhdl.sv)
sx8723_for_sv_tb sx(.*);
test    te(.*);

assign ac2v = acv[0]*10.0e7; // ( conversion from integer to real )
assign ac3v = acv[1]*10.0e7;
assign ac4v = acv[2]*10.0e7;

// ... (see Appendix 6, top_sv_vhdl.sv)
endmodule

6.2.3 Test program

The SV program block is a construct that can be instanced in the same way as a module and is usually used for modeling a test environment [9]. It is closer to a program in C, with one (or more) entry points[10]. In program, we can put tasks, functions, classes and initial blocks. We go through initialization, drive and respond to design activity, and then program completes. When the last initial block completes, simulation implicitly ends or explicitly calling system function $finish or $exit.

Arguments for our program test are input and output signals for SX8723 wrapper: acv[6] -voltage for voltage generators driving SX8723 analog pins, acz[6]-impedance for voltage generators driving SX8723 analog pins, pad_readyDig – pin ready, scl_in and scl_out – input and output for pin scl, sda_in and sda_out - input and output for pin sda. Inside test, object env of type class Environment from package EnvironmentPackage is declared first. (See the next subsection for the Environment definition). In initial block, object env is constructed with function new. Once the Environment is constructed, using Environments methods drvCfg and build, we do drivers configuration and we build the environment. With method run, (env.run), first directed test is run with 50 transactions (transactions randomization is turned off – env.agt.tr[i].rand_mode(0)). After directed test, randomization is turned on, and we run constrained-random test with 100 transactions. Finally, we do Environment wrapup and finish simulation.

36
program test(output real acv[6], acz[6],
    input logic pad_ready_dig,
    output logic scl_out=1, sda_out=1,
    input logic scl_in, sda_in);
import EnvironmentPackage::*;
Environment env;
initial begin
    env = new;
    env.drv_cfg;
    env.build;
    foreach ( env.agt.tr[i]) begin
        env.agt.tr[i].rand_mode(0);
        env.agt.tr[i].c_tr.constraint_mode(0);
    end
    env.run(sda_out, sda_in, scl_out, scl_in, acv, acz, pad_ready_dig, 50);
    foreach ( env.agt.tr[i]) begin
        if (i<20) env.agt.tr[i].rand_mode(1);
    end
    env.run(sda_out, sda_in, scl_out, scl_in, acv, acz, pad_ready_dig, 100);
    env.wrapup;
    $finish;
end
endprogram

6.2.4 Environment

Our environment has all transactors described in the first section. For communication between transactors, mailboxes (gen2agt, agt2drv, agt2drv_i2c, drv_i2c2agt) are used. (Transactors (gen, agt, drv[6], drv_i2c) are sending and receiving Transactions to/from other Transactors via mailboxes.) Function new constructs and initializes all objects (Transactors, Scoreboard, Config and mailboxes) previously declared, passing Mailboxes to Transactors as arguments. (See subsections below for Transactors definitions.) Drivers (drv) for SX8723 analog pins are constructed according to Config variable in_use, which can be randomized. 2, 4 or 6 drivers for 3 differential inputs can be constructed. drv_i2c is driver for I2C pins scl and sda. Methods (routines) build, run and wrapup build, run and wrapup the environment respectively. (See Appendix 6, environment_classes.sv for methods definitions.) Task run spawns threads, i.e. execute Generator, Agent and drivers to run in parallel, using fork...join statement. Class Generator (for I2C) is doing nothing and its functionality can be implemented in the future. For SX8723 analog input pins drivers (drv[6]) are implemented; agent and generator are not defined.

class Environment;
Scoreboard sbd;
Generator gen;
Agent agt;
Driver drv[6];
//Driver_i2c drv_i2c;
Driver_i2c_good drv_i2c;
Config cfg;
mailbox #(Transaction) gen2agt, agt2drv, agt2drv_i2c, drv_i2c2agt;
extern function new;
extern function void drv_cfg;
extern function void build;
extern task run( ref sda_out, sda_in, scl_out, scl_in, ref real acv[6], acz[6],
  ref pad_ready_dig, input int run_for_n_trans=10);
extern task wrapup;
endclass

function Environment::new;
$display("Environment.new @%0d", $time);
//Initialize a configuration and a scoreboard
  cfg = new;
  sbd = new;
//Initialize mailboxes
  gen2agt = new;
  agt2drv = new;
  agt2drv_i2c = new;
  drv_i2c2agt = new;
//Initialize transactors
  gen = new(gen2agt);
  agt = new(gen2agt, agt2drv_i2c, drv_i2c2agt, sbd);
  foreach (drv[i]) if (cfg.in_use[i])
    drv[i] = new(i, agt2drv, sbd);
  drv_i2c = new(agt2drv_i2c, drv_i2c2agt);
endfunction
endtask
endpackage

6.2.5 Transaction

Class Transaction is defined for transactions between I2C agent (agt) and I2C driver (drv_i2c). The class has 4 variables for slave address (slvadd), memory address (memadd), memory data (data) and write/read (nwr). Constraint c_tr is defined to constrain slvadd, memadd and data for reading and writing to all SX8723 registers all possible configurations. Constraint c_tr_adc is defined for the 16 bits continues conversion with offset 0.

class Transaction;
rand bit [6:0] slvadd;
rand bit [7:0] memadd, data;
rand bit nwr;

constraint c_tr    ( See Appendix 6, environment_classes.sv )
constraint c_tr_adc ( / See Appendix 6, environment_classes.sv )
endclass

6.2.6 I2C Agent and I2C Checker

Class Agent is defined for SX8723 I2C interface pin. Class Agent has 3 mailboxes: gen2agt (for transactions from from I2C generator to agt), agt2drv_i2c (for transactions from I2C agent to I2C driver) and drv_i2c2agt for vice versa. Transactions from Transaction array tr
[] are put to agt2drv_i2c, and Transaction tr_get is getting Transactions from drv_i2c2agt. New type Struct i2c_instruction_s is defined for SX8723 I2C interface’ instruction according to [8]. (The same format is used as in Semtech VHDL testbench.) The structure (instruction) has 4 variables: nwr (I2C master performing writing or reading) , slvadd (I2C slave address), memadd (SX8723 memory address) and data (data to write/read to/from SX8723 memory). instr[$], i2c_instruction_s queue, includes instructions for directed test (Directed test is configuring SX8723 registers for the continues AD conversion.) Predefined instructions read_adc_msb, read_adc_lsb, stop_adc, start_adc are defined to read ADC result, to start continues ADC and to stop continues ADC respectively. (In a future improved testbench, these predefined instructions could come from the upper level, i.e. Generator.)

class Agent;

mailbox #(Transaction) gen2agt, agt2drv_i2c, drv_i2c2agt;
Transaction tr[];
protected Transaction tr_get;
Scoreboard sbd;

typedef enum bit (write, read) nwr_t;
typedef struct {nwr_t nwr;
    bit [6:0] slvadd;
    bit [7:0] memadd;
    bit [7:0] data; } i2c_instruction_s;

i2c_instruction_s read_adc_msb='{read, 72, 8'h51, 8'hxx};
i2c_instruction_s read_adc_lsb='{read, 72, 8'h50, 8'hxx};
i2c_instruction_s stop_adc='{write, 72, 8'h52, 8'h00};
i2c_instruction_s start_adc='{write, 72, 8'h52, 8'bxxxxxx1x};
i2c_instruction_s read_stop_adc='{read, 72, 8'h52, 8'hxx};

i2c_instruction_s instr[$] ={ '{write, 72, 8'h80, 8'ha0},
    '{read, 72, 8'h80, 8'ha0},
    '{write, 72, 8'h80, 8'h40},
    '{write, 72, 8'h52, 8'h52} }

function new(mailbox #(Transaction) gen2agt, agt2drv_i2c, drv_i2c2agt,
    Scoreboard sbd);
    // See Appendix 6, environment_classes.sv
    endfunction

function void build(int n_truns=10);
    repeat (n_truns) begin
        instr.push_back(read_adc_msb);
        instr.push_back(read_adc_lsb);
    end
    instr.push_back(stop_adc);
    instr.push_back(read_stop_adc);
    tr = new[$size(instr)];
    foreach (tr[i]) begin
        tr[i] = new;
        tr[i].nwr = instr[i].nwr;
        tr[i].slvadd = instr[i].slvadd;
        tr[i].memadd = instr[i].memadd;
        tr[i].data = instr[i].data;
    end
endfunction

task run(ref pad_readyDig);
    foreach(tr[i]) begin
        assert (tr[i].randomize) $display("after randomization:");
    end

39
else $display("randomization failed: ");
if (tr[i].memadd!=8'h50) wait (pad_ready_dig);
agt2drv_i2c.put(tr[i]);
tr_get=new;
drv_i2c2agt.get(tr_get);
sbd.write(tr_get);
end
tr_get=null; agt2drv_i2c.put(tr_get);
endtask
endclass

Function build is building Transactions for sending them to the driver. First, taking the instructions from i2c_instruction_s, Transactions for the register configuring are built. After this, Transactions for ADC result reading are built.

Finally, task run is synchronously sending/receiving non randomized or randomized Transactions to/from the driver. After the agent has sent Transaction to the driver, it will wait for a completion message from the driver. If the Agent is performing writing, it will wait for the same Transaction, meaning the driver has finished writing successfully. If the Agent is performing reading, it will wait for the reading result, i.e. data the driver read from the SX8723. After this, writing/reading transaction is recorded in the Scoreboard, calling Scoreboard's function write. Here, it should be noticed that since SX8723 I2C pins (SDA and SCL) are bidirectional pins, class Agent is realized to perform both writing and reading, i.e. to make stimulus and to read response. It means, class Agent is agent and checker at the same time. (In this class, the usage of two mailboxes has two functions: to synchronize transactions and to implement checker functionality.)

The last message sent to the driver is a deallocated handle tr_get (tr_get=null), to indicate to the driver the last transaction is sent.

6.2.7 I2C Driver and I2C Monitor

Class Driver is implemented as a driver and a monitor at the same time. (For the same reason it has been explained in the previous subsection.) The task run is receiving the transactions from the agent, is copying transactions variables (slvadd, memadd, data and nwr) and sending them to the I2C interface. To send them to the I2C interface, local task mstr2svl is defined. mstr2svl is written according to the I2C-bus Specification [11] for the master to slave communication. First memadd is sent, and after data is sent or read to/from memadd, depending on nwr ([8] I2C communication format). Finally, after the second mstr2svl is finished reading/writing, data is copied to the transaction and the transaction is sent back to the agent. If the driver receives null, we leave the loop, and the task is finished.

class Driver_i2c;

mailbox #(Transaction) agt2drv_i2c, drv_i2c2agt;
protected Transaction tr;

protected bit w=1'b0, r=1'b1, a=1'b1;
6.2.8 Driver (ac[6])

Class Driver is defined for driving the SX8723 differential analog inputs. Function build define frequency (f), dc (dc), ac (ac) and impedance (z) for the voltage generator. In task run , sinusoidal driving voltage acv is spawned forever with fork...join_none statement. The generator voltage acv is recorded in Scoreboard calling Scoreboard's function write_acv. When all I2C transactions are finished, task run will be disabled from task wrapup. (See program test and class Environment.)

class Driver;

mailbox #(Transaction) agt2drv;
Transaction tr;
Scoreboard sbd;
int id;
real f, dc, ac;
real z;

function new(int id, mailbox #(Transaction) agt2drv, Scoreboard sbd);
    // ...
endfunction

function void build( input real f=0.01, dc=1.3, ac=0.001,z=1.0);
    this.f=f;
    this.dc=dc;
    if (id inside {1,3,5}) this.ac=ac; else this.ac=-ac;
    this.z=z;
endfunction

task run (ref real acv[6], acz[6]);
class Scoreboard;

// ...
function new;
// ...
    diagnosticsfile = $fopen("diagnostics.txt");
endfunction

function automatic shortint expadc(real vin);
    expadc=(2.0**16*vin/vref*(gdtot-gdofftot*vref/vin)*(osr+1)/osr);
endfunction

function void write_acv(input int id, input real ac);
    acv[id]=ac;
endfunction

function automatic real vin;
    casex (exp_data[57][5:1])
        5'b00x00 : vin=acv[1]-acv[0];
        5'b00x01 : vin=acv[3]-acv[2];
    endcase
endfunction

6.2.9 Scoreboard

Class Scoreboard is recording written values to the SX8723 registers and comparing them
with the read values. Also, the Scoreboard is recording the single analog inputs, predicts ADC
results, compares ADC results with predicted and calculates the ADC error.

Function new with SV system function $fopen opens the diagnostics file
diagnostics.txt where all transactions, predicted and obtained ADC results and other
messages are written. Function expadc, returning shortint, calculates an expected ADC result. A
result is calculated according to [8] Equation 11, with the function argument vin (differential
analog input) Void function write_acv is recording the single analog inputs. This function is
called from the class Driver. Function vin, returning real, calculates differential analog input,
according to [8] (Table 2. - Analog input selection). Void function write_exp_adc is recording
expected ADC result. Function adc_error, returning string, calculates the ADC error and returns
a message about it.
Function write records the values written to the registers and compare them with the read values. When the test is reading ADC result registers, function write is comparing their values to the predicted ADC result.

At the end of the simulation, when function wrapup will be called, the diagnostics file will be closed.

A diagnostic file, obtained during simulation, is given in Appendix 6. (diagnostics.txt) Also, a part of the transcript file is given.(transcript_ms_menthe3)
Chapter 7
Conclusion

This project has shown that PSL assertions can be used for the validation of mixed-signal designs. It is possible with Property Specification Language to describe with details analog behaviors modelized with Semtech analog package. There is no constrictions to define PSL properties using analog signals from Semtech analog package.

To ensure that some analog properties really hold, assertions must be evaluated very frequently, i.e. we must try to approach from digital to analog domain as close as possible. It means, assertions must be clocked with very high frequency clocks. On the other hand, when PSL assertions are clocked with clock frequency higher than system clocks, simulation will be slowed down. A compromise between these two must be done.

During the project, it was demonstrated that PSL can be effectively used to check digital signals stability.

Interface protocols properties can be defined and asserted as well. Defining properties is becoming complex, less understandable and not reusable. Non reusable properties are consequence of the fact that PSL assertions do not have arguments.

Using QuestaSim “spy” function we successfully defined properties with analog and digital signals coming from different VHDL units, Verilog and SystemVerilog modules.

SystemVerilog assertions have certain number of advantages. SV assertions have variables and SV assertions can be clocked with more than one signals and are triggered on any clock event. In order to apply PSL assertions, written in VHDL flavor, to Verilog module, they must be translated to Verilog flavor. Once defined SV assertions, can be applied to both VHDL and Verilog units. Also, it is more natural to do verification with SV testbench and SV assertions, than with SV testbench and PSL assertions.

QuestaSim successfully supports mixed-languages simulation. Except, there is no signal mapping for real signals. Since the Semtech analog signal is the record of reals, it has made some problems presented during the project. Fortunately, a future tool release will have the mapping for
real signals.

Finally, a simple SV layered testbench for SX8723 has been made. It has shown how VHDL design is directly instantiated in SV and successfully demonstrated basics of SV advanced verification methods.
Copyright (C) Semtech SA. All rights reserved.
All PSL and SV properties, SystemVerilog and VHDL design units are developed at Semtech SA, Neuchatel, Switzerland.
All rights reserved. Reproduction in whole or part is prohibited without the written permission of the copyright owner.
# A6 simulate.sh

```bash
#!/bin/sh

# simulate sxs723 with the SV testbench (with PSL and SV assertions)

print_usage ()
{
    echo -e "\t-b for batch mode\n"    echo -e "\t-g for gate-level simulations with SDF backannotation\n"    echo -e "\t-n for simulations without PSL assertions\n"    echo -e "\t-a for simulations with assertdebug tool for debugging assertions\n"    echo -e "\t-f to enable assertion failure logging\n"    echo -e "\t-m to record assertion messages in assertion_messages.log file\n\totherwise they are sent to standard output\n"    echo -e "\t-r to quit simulation after runtime or testbench is ended\n"    echo -e "\t-r to specifie the number of simulation time unit (ms)\n\totherwise simulation runs forever (run -all)"
}

batchmode=0
gatemode=0
nopsl=0
assertdebug=0
faillog=0
passlog=0
recommands=0
quitsim=0
runtime="-all"

while getopts "b:g:n:a:f:p:m:q:r:" options; do case $options in
    b) batchmode=1
       ;;
    g) gatemode=1
       ;;
    n) nopsl=1
       ;;
    a) assertdebug=1
       ;;
    f) faillog=1
       ;;
    p) passlog=1
       ;;
    m) recommands=1
       ;;
    q) quitsim=1
       ;;
    r) runtime="$OPTARG ms"
       ;;
    *) print_usage
       exit -1
       ;;
esac
done

host=`hostname`
mode=`expr substr $host 1 7` #$_$testbench_Sbatchmode$nopsl$assertdebug$faillog

```

Appendix 1
if [ -e ./assertion_report_$mode.txt ]; then
    /bin/rm ./assertion_report_$mode.txt
fi

if [ -e ./assertion_messages_$mode.log ]; then
    /bin/rm ./assertion_messages_$mode.log
fi

if [ -e /export/home/tmp/stand/vsim_$mode.wlf ]; then
    /bin/rm /export/home/tmp/stand/vsim_$mode.wlf
fi

if [ -e ./transcript_ma_$mode ]; then
    /bin/rm ./transcript_ma_$mode
fi

if [ -e ./vsim_$mode.wlf ]; then
    /bin/rm ./vsim_$mode.wlf
fi

if [ -e /lib_xe_project/sx8723_binds ] && [ $nopsl = = 0 ]; then
    svabinds="lib_xe_project.sx8723_binds"
fi

if [ $gatemode = = 1 ]; then
    vsim="modelsim -6.3 vsim -l csl_41 -lwf ./vsim_$mode.wlf -lwfdeleteonquit -1 ./transcript_ma_$mode +no_glitch_msg -sdftyp sx8723_ttb/l_sx8723/ai_sx8723tc/l_sx8723dcos ./sdf/sx8723dc_patch.sdf -sdftyp sx8723_ttb/l_sx8723/ai_sx8723tc/l_sx8723dcos ./digtgen ../sdf/dgkgen_patch.sdf" # wlflim 1000ns
else
    vsim="modelsim -6.3 vsim -l csl_41 -lwf ./vsim_$mode.wlf -lwfdeleteonquit -1 ./transcript_ma_$mode" # -gui -wlflim 1000 ns
fi

if [ $nopsl = = 1 ]; then
    vsim="$vsim -nopsl"
fi

if [ $assertdebug = = 1 ]; then
    vsim="$vsim -assertdebug"
fi

if [ $faillog = = 1 ]; then
    do="set assertion fail -enable -r * ; assertion pass -enable -r * ; assertion fail -log on -r *"
else
    do="set assertion fail -enable -r * ; assertion pass -enable -r * ; assertion fail -log off -r *"
fi

if [ $passlog = = 1 ]; then
    do="$do; assertion pass -log on -r *"
else
    do="$do; assertion pass -log off -r *"
fi

if [ $recmessages = 1 ]; then
    vsim="$vsim -assertfile ./assertion_messages_$mode.log"
fi

do="$do; run $runtime; assertion report -file assertion_report_$mode.txt /top -r *" # run $runtime
if [ $quitvsim = = 1 ]; then
    do="$do; quit -f"
fi

wave="wave.do"

# Interactive mode
if [ $batchmode = = 0 ]; then
    # check wave file

if [ -e ./wave ]; then
    do="do $wave; $do"
fi

if [ "$quitvsim" = = "1" ] && [ "$runtime" = = "-all" ]; then
    $vsim -gui lib_xe_project.top $svabinds -do "$do" & &
    expect < <!
    spawn tail -f --retry ./transcript_ma_$mode
    expect {"re" "end of testbench" (exec killall vsimk)
    timeout (exp_continue)
    }!
else
    $vsim -gui lib_xe_project.sx8723_ttb $svabinds -do "$do"
    $vsim -gui lib_xe_project.top $svabinds -do "$do" $lib_xe_project.bind_tb_sx
fi
else
  # !!! Batch mode only !!!
  if [ "$quit_sim" == "1" ] && [ "$runtime" == "-all" ]; then
    expect <
      spawn $vsim -c lib_xe_project.top $svabinds -do "$do"
      expect :
        " end of testbench" { send "break\r"}
      timeout [exp_continue]
    }
    expect {
      "VSI_M" { send "assertion report -file assertion_report.txt /top = *" }
      timeout [exp_continue]
    }
  else
    $vsim -c lib_xe_project.top $svabinds -do "$do"
  fi
fi

---
\[ Multn \]

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.ALL;

Entity multn is
generic ( BUG1 : Boolean := FALSE;
  BUG2 : Boolean := FALSE;
  N : natural := 8 );
Port ( OPA : in unsigned(N-1 downto 0);
  OPB : in unsigned(N-1 downto 0);
  STB : in std_logic;
  clk : in std_logic;
  RST_B : in std_logic;
  MRES : out unsigned(2*N-1 downto 0);
  DONE : out std_logic );
end entity multn;

architecture rtl of multn is
type STATES is ( IDLE, INIT, CHECK, ADD, SHIFT);
signal state : STATES;
begin
  proc_multn:
  process(RST_B, clk)
    -- variable state : STATES;
    variable OPA_REG : unsigned(N-1 downto 0);
    variable OPB_REG, ACC : unsigned(2*N-1 downto 0);
  begin
    if RST_B = '0' then
      state <= IDLE;
    elsif clk'event and clk = '1' then
      case state is
      when IDLE =>
        if (STB='1') then
          state <= INIT;
        end if;
      when INIT =>
        OPA_REG:=OPA;
        OPB_REG:=to_unsigned(0,2*N);
        OPB_REG(N-1 downto 0):=OPB;
        ACC:=to_unsigned(0,2*N);
        DONE<="0";
        state <= CHECK;
      when CHECK =>
        if (OPA_REG=0) then
          MRES<=ACC;
          DONE<="1";
          state <= IDLE;
        elsif OPA_REG(0)="1" then
          state <= ADD;
```
else
    state <= SHIFT;
end if;
when ADD =>
-- BUG2
  if BUG2 then
    ACC:=ACC + OP_A_REG;
  else
    ACC:=ACC + OP_B_REG;
    end if;
    state <= SHIFT;
when SHIFT =>
    OP_A_REG:= shift_right(OP_A_REG,1);
    OP_B_REG:= shift_left(OP_B_REG,1);
--BUG1
  if not BUG1 then
    state <= CHECK;
  end if;
end case;
end if;
end process;
end architecture rtl;

A2 tb_multn_rtl.vhd

-- testbench for RTL model of the N-bit generic multiplier

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity tb_multn_rtl is end;

architecture bench of tb_multn_rtl is
  constant NBITS : positive := 8;
  constant MAX     : unsigned(NBITS-1 downto 0) := (others => '1');
  constant CLK_PER: time := 15 ns;
signal opa, opb: unsigned(NBITS-1 downto 0);
signal mres: unsigned(2*NBITS-1 downto 0);
signal stb, rst_b, done: std_logic;
signal clk: std_logic := '0';
signal finished: boolean := FALSE;
begun
  UUT: entity work.multn(rtl)
  generic map (FALSE, TRUE, NBITS)
  port map (
op_a => opa,
op_b => opb,
  stb => stb, 
  clk => clk,
rst_b => rst_b,
mres => mres,
done => done);
-- clock generator
  process (clk, finished)
  begin
  -- stopping the clock once all possible input values have been scanned
  -- allows for executing a "run -all" command
  if not finished then
    clk <= not clk after CLK_PER/2;
  end if;
end process;
-- initial asynchronous reset
  rst_b <= '1', '0' after 3*CLK_PER/4, '1' after 5*CLK_PER/4;
-- stimulus generator and output verification
  proc_stim:
  process
  variable va, vb: unsigned(NBITS-1 downto 0);
  variable exp_mres: unsigned(2*NBITS-1 downto 0);
begun
  stb <= '0';
  -- keep the first clock cycle for the initial reset
  wait until clk = '0';
  -- all possible NBITS-bit input operands are applied
  for i in 0 to to_integer(MAX) loop
    for j in 0 to to_integer(MAX) loop
      -- assign new input values at next falling clock edge
      va := to_unsigned(i, opa'length); opa <= va;
      vb := to_unsigned(j, opb'length); opb <= vb;
      -- start computation at next rising clock edge
      stb <= '1';
      wait until clk = '0';
      stb <= '0';
      -- wait until computation is finished
      wait until done = '1';
      -- check multiplier result against true result
      -- wait until outputs stabilize
end process;
end process;
wait until clk = '0';
exp_mres := va*vb;
assert mres = exp_mres;
report "Incorrect result -- " &
"expected: " & integer'image(to_integer(exp_mres)) & ", " &
"got: " & integer'image(to_integer(mres))
severity ERROR;
end loop;
end loop;
finished <= TRUE;
wait;
end process;
end architecture bench;

A2 multn.psl

1 vunit check_multn(mult(RTL))
2 |
3 default clock is (rising_edge(clk));
4 -- (FSM state coverage pattern)
5 -- functional coverage each state entered
6 cover (state=INIT);
7 cover (state=CHECK);
8 cover (state=ADD);
9 cover (state=SHIFT);
10 cover (state=IDLE);
11
12 --(state sequence coverage pattern)
13 sequence multn_seq is ((state = IDLE);(state = INIT);(state = CHECK));
14 c_multn_seq : cover (mult_seq);
15
16 -- (valid opcode pattern)
17 --Check if the state is legal state
18 property legal_states is always ((state = INIT) or
19 (state = CHECK) or
20 (state = ADD) or
21 (state = SHIFT) or
22 (state = IDLE));
23 assert legal_states;
24
25 property trans_from_CHECK is always ((state=CHECK) => (state=IDLE or state=ADD or state=SHIFT));
26 assert trans_from_CHECK;
27
28 -- (eventbounded window pattern)
29 a new 'stb' cannot start before the first one completes
30 property done_before_stb is (always stb -> next ((not stb) until done))@falling_edge(clk);
31 assert done_before_stb;
32
33 property done_before_stb2 is (always stb -> next ((not stb) until done));
34 assert done_before_stb2;
35
36 -- (timebounded window pattern)
37 -- assert that 'done' must occur within 27 cycles after a 'stb'
38 property done_within_27clk is (always ((stb) => ((*0 to 27);{done}))))@falling_edge(clk);
39 assert done_within_27clk;
40
41 -- (conditional expression pattern)
42 -- if 'done' signal is active current state must be 'IDLE' or 'INIT'
43 assert always done -> state=IDLE or state=INIT;
44

A2 tb_multn.psl

1library modelsim_lib;
2
3vunit check_tb_multn(tb_multn_rtl(bench))
4|
5 use modelsim_lib.util.all;
6 type STATES is ( IDLE, INIT, CHECK, ADD, SHIFT);
7 signal tb_state : STATES;
8 signal tb_mres : unsigned(2*NBITES-1 downto 0);
9 signal tb_ACC : unsigned(2*NBITES-1 downto 0);
10 signal exp_mres : unsigned(2*NBITES-1 downto 0);
11
12 init_signal_spy ("/tb_multn_rtl/UUT/state", "tb_state");
13--init_signal_spy ("/tb_multn_rtl/UUT/proc_multn/ACC", "tb_ACC");
14--init_signal_spy ("/tb_multn_rtl/proc_ssim/exp_mres", "exp_mres");
15
16 default clock is (falling_edge(clk));
17
18 -- (valid opcode pattern)
19 --Check if the state is legal state
20 property legal_states is always ((tb_state = INIT) or
21 (tb_state = CHECK) or
22}
(tb_state = ADD) or
(tb_state = SHIFT) or
(tb_state = IDLE));
assert legal_states;

-- check if 'ACC' is '0' after INITialisation

-- property ACC_after_INIT is always (tb_state=INIT -> tb_ACC=0) abort not rst_b;

-- assert ACC_after_INIT;

-- check multiplier result against true result

property correct_result is always (rose(done) -> mres = opa * opb);
assert correct_result;

-- check multiplier result against true result

-- property correct_result2 is always (rose(done) -> mres = exp_mres);
-- assert correct_result2;
Appendix 3

A3 analog_pack.vhd

-- Copyright (C) Xemics SA. All rights reserved.
-- Developed at Xemics SA, Neuchatel, Switzerland
-- All rights reserved. Reproduction in whole or part is prohibited without
-- the written permission of the copyright owner.
-- A. FUNCTION;
-- This package allows to modelize precisely the impedance of each device.
-- As a result the voltage result is no more an estimation but a significant
-- value. This model detects also the "multi drive" configurations.
-- The current returned in the corresponding field represents the current
-- consumed by the devices set as "loads".
-- Two fields return the number of gene and loads connected to each node.
--
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

PACKAGE analog IS

-- enumerated types used
TYPE model_type IS (undefined,voltage,current);
TYPE device_type IS (hiz,gene,load);
CONSTANT DEFAULT_Z : real := 1.0e22;
CONSTANT NULL_Z : real := 1.0e-12;

-- debug option
signal debug_analog : boolean := false; -- set to true to enable debug error

SUBTYPE real32 is real range -1.0e38 to 1.0e38;
signal debug_analog : boolean := false;

-- definition of the unresolved analog_device type
TYPE analog_device IS record
  V_value : real32; -- will work on any system (32 or 64 bit real implementation. e.g Modelsim 5.2 vs 5.3) DSI/02/11/99
  I_value : real32;
  Z_value : real32;
  model : model_type;
  device : device_type;
  nb_gene : natural; -- added for check purpose
  nb_load : natural; -- added for check purpose
END RECORD;

-- definition of vectors and resolved subtype analog_d
TYPE analog_d_vector IS ARRAY (natural range <>) OF analog_device;
FUNCTION resolve_d (inputs : analog_d_vector) RETURN analog_device;

SUBTYPE analog IS resolve_d analog_device;
TYPE analog_vector IS ARRAY (natural RANGE <>) OF analog;

-- procedure and function
PROCEDURE v_gene(SIGNAL a : INOUT analog;V_value:real;Z_value:real);
PROCEDURE i_gene(SIGNAL a : INOUT analog;I_value:real;Z_value:real);
PROCEDURE v_load(SIGNAL a : INOUT analog;V_value:real;Z_value:real);
PROCEDURE i_load(SIGNAL a : INOUT analog;I_value:real;Z_value:real);
PROCEDURE r_load(SIGNAL a : INOUT analog;Z_value:real);
PROCEDURE high_z(SIGNAL a : INOUT analog);

-- Overloaded procedures
--
PROCEDURE v_gene(SIGNAL a : INOUT analog;V_value:real;Z_value:real;SIGNAL I_deliver: out real);
PROCEDURE i_gene(SIGNAL a : INOUT analog;I_value:real;Z_value:real;SIGNAL I_deliver: out real);
PROCEDURE v_load(SIGNAL a : INOUT analog;V_value:real;Z_value:real;SIGNAL I_deliver: out real);
PROCEDURE i_load(SIGNAL a : INOUT analog;I_value:real;Z_value:real;SIGNAL I_deliver: out real);
PROCEDURE r_load(SIGNAL a : INOUT analog;Z_value:real;SIGNAL I_deliver: out real);

FUNCTION check_z(z : real) RETURN real;
FUNCTION to_real32(r : real) RETURN real32;
PROCEDURE uni_dir_sw(SIGNAL a:INOUT analog;in_sgn:analog;command:std_logic);

SIGNAL enable_sw : boolean:=true;
Utility functions for conversion from and to std_logic. Functions are overloaded for vectors.

```vhd
function stdu2ana ( std : std_logic; vdd : real ) return analog;
function stdu2ana ( std : std_logic_vector; vdd : real ) return analog_vector;
function ana2stdu ( ana : analog; vdd : real ) return std_logic;
function ana2stdu ( ana : analog_vector; vdd : real ) return std_logic_vector;
function std2ana ( std : std_logic; vdd : real ) return analog;
function std2ana ( std : std_logic_vector; vdd : real ) return analog_vector;
function ana2std ( ana : analog; vdd : real ) return std_logic;
function ana2std ( ana : analog_vector; vdd : real ) return std_logic_vector;
```

A3 LP_filter.vhd

```vhd
library semtech;
use semtech.analog.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity LP_filter is
generic ( dt : real := 10.0; R : real := 1.0e6; C : real := 10.0e-12 );
port ( Vin : inout analog; Vout : inout analog; clk : in std_logic );
```
end entity LP_filter;

architecture dtm of LP_filter is
begin
  process (clk)
    variable vv: real := 0.0;
  begin
    -- if rising_edge(clk) then
      if clk'event and clk = '1' then
        vv := (dt*Vin.v_value + R*C*vv) / (dt + R*C);
        v_gene(Vout, vv, 1.0);
      end if;
    end process;
  end process;
end architecture dtm;

A3 tb_LP_filter.vhd

library ieee, semtech;
use ieee.math_real.all;
use semtech.analog.all;
library ieee;
--use ieee.numeric_std.all;
use ieee.std_logic_1164.all;

entity tb_LP_filter is end;

architecture bench of tb_LP_filter is
constant AMP: real := 1.0;
constant dt: real := 100*1.0e-9; -- ns
constant CLK_PER: time := integer(dt*1.0e9)*ns;
constant R: real := 10.0e3;
constant C: real := 100.0e-12;
signal vsr, svload: real := 0.0;
signal clk: std_logic:=0';
signal Vin, Vout: analog;
signal time: real; -- used for the PSL file only
begin
  UUT: entity work.LP_filter(dtm)
    generic map (dt, R, C )
    port map (Vin, Vout, clk);
  clk <= not clk after CLK_PER/2;
  process
    variable vsr, vtime: real := 0.0;
  begin
    wait until clk='1';
    vsr := AMP;
    report "--- vsr = " & real'image(vsr);
    vsr <= vsr;
    v_gene(Vin, vsr, 1.0);
    vtime := real(now/ns)*1.0e-9;
    time <= vtime;
    if vtime > 10000.0*1.0e-9 then
      v_gene(Vin, 0.0, 1.0);
    end if;
  end process;
  r_load(Vout, 1.0e6);
  svload <= Vout.v_value;
end architecture bench;

A3 LP_filter.psl

1 library ieee;
2 use ieee.math_real.all;
3
4 vunit check_tb_LP_filter(tb_LP_filter(Bench))
5|
6 signal a, b: boolean;
7 default clock is rose(Clk);
8 |
9 a <= time > 5.0*R*C;
10 b <= Vout.v_value > AMPL*0.99;
11 |
12 -- check that Vout < AMPL
13 property Vout_range is always (Vout.v_value < AMPL);
14 assert Vout_range;
15 |
16 -- check that Vout = 0.99Vin after 5RC
17 property tSRC is (now/ns)*1.0e-9 > 5.0*R*C)--a
18 property Vout_reaching_AMPL is Vout.v_value > AMPL*0.99;--b
19 --property Vout_after_5RC is always t_SRC -> Vout_reaching_AMPL;
20 property Vout_after_5RC is always (a and Vin.v_value=AMPL)--> b;
21 assert Vout_after_5RC;
A4 grisc_system.vhd

```vhdl
library ieee;
use ieee.std_logic_1164.all;
library lib_xe_project;
use lib_xe_project.im_pkg.all;
use lib_xe_project.grisc_pkg.all;
use lib_xe_project.addec_pkg.all;
use lib_xe_project.i2c_pkg.all;

entity grisc_system is
    port(
        -- general
        nreset : in std_logic;
        ckmsr : in std_logic;
        sclk_phi1 : in std_logic;
        sclk_phi2 : in std_logic;
        pcl : in std_logic;
        pcc : in std_logic;
        -- scan
        sm : in std_logic;
        sw : in std_logic;
        si : in std_logic;
        so : out std_logic;
        -- Data Bus
d2ld_bus : in d2ld_bus_t(indexLast-1 downto 0);
cs_bus : out std_logic_vector(indexLast-1 downto 0);
        rnw : out std_logic;
        da : out std_logic_vector(7 downto 0);
d2st : out std_logic_vector(7 downto 0);
        -- State Machine
        events : in std_logic;
        nfreeze : out std_logic;
        -- I2C
        i2c_address : in std_logic_vector(6 downto 0);
i2c_idle : out std_logic;
i2c_gcr : out std_logic;
        -- low-power mode
        wakeup : out std_logic;
        -- I2C pads
        sclk_in : in std_logic;
sclk_out : out std_logic;
sclk_oen : out std_logic;
sda_in : in std_logic;
sda_out : out std_logic;
sda_oen : out std_logic;);
    end grisc_system;

architecture rtl of grisc_system is
    signal ia : std_logic_vector(7 downto 0);
    signal iw : std_logic_vector(11 downto 0);
    signal s : std_logic_vector(2 downto 0);
    signal dme : std_logic;
    signal d2ld : std_logic_vector(7 downto 0);
    signal i2c_event : std_logic;
    signal rnw_int : std_logic;
    signal da_int : std_logic_vector(7 downto 0);
    signal cs_bus_int : std_logic_vector(indexLast downto 0);
    signal event_int : std_logic;
    signal d2ld_bus_int : d2ld_bus_t(indexLast downto 0);
    begin
        s(0) <= s1;
s0 <= s(2);
cs_bus <= cs_bus_int(indexLast-1 downto 0);
d2st <= d2st_int;
rnw <= rnw_int;
da <= da_int;
event_int <= events or i2c_event;
d2ld_bus_int(indexLast-1 downto 0) <= d2ld_bus;
i_im : im port map(
    ia => ia,
```
iw => iw,  
pc1 => pci,  
ccc => pcc  
);
i_grisc_core : grisc_core  
port map(  
nreset => nreset,  
cpuckmstr => ckmstr,  
cpuckslv => ckslv,  
scannode => sm,  
shiften => se,  
shiftin => s(0),  
shiftout => s(1),  
dt2ld => d2ld,  
dme => dme,  
rnw => rnw_int,  
da => da_int,  
dt2st => dt2st_int,  
iw => ia,  
la => la,  
events => event_int,  
nfreeze => nfreeze  
);
i_addec : addec  
port map(  
d2ld_bus => d2ld_bus_int,  
d2ld => d2ld,  
address => da_int,  
dme => dme,  
bus => cs_bus_int  
);
i_2c : i2c  
port map(  
nreset => nreset,  
ckmstr => ckmstr,  
ckslv => ckslv,  
scl_phi1 => scl_phi1,  
scl_phi2 => scl_phi2,  
si => s(1),  
so => s(2),  
se => se,  
sm => sm,  
bus => cs_bus_int(indexLast),  
rnw => rnw_int,  
da => da_int,  
dt2st => dt2st_int,  
d2ld => d2ld_bus_int(indexLast),  
i2c_address => i2c_address,  
i2c_gcr => i2c_gcr,  
i2c_event => i2c_event,  
i2c_idle => i2c_idle,  
wait => wait,  
scl_in => scl_in,  
scl_out => scl_out,  
scl_oen => scl_oen,  
sda_in => sda_in,  
sda_out => sda_out,  
sda_oen => sda_oen  
);
end rtl;
A4: GriscSystem block diagram schematics
A4 grisc_system.psl 1

1 vunit check_grisc_system_interface(grisc_system(rlt))
2 |
3 signal cs_bus_int_bv: bit_vector(indexLast downto 0);
4 cs_bus_int_bv <= to_bitvector(cs_bus_int);
5 |
6 signal clk: std_logic := '0';
7 clk <= not clk after 1 ns;
8 default clock is falling_edge(clk);
9 |
10 property onehotCS is (always dme -> onehot(cs_bus_int_bv))@falling_edge(ckslv);
11 assert onehotCS;
12 |
13 assert (always onehot(cs_bus_int_bv))@falling_edge(ckslv);
14 |
15 property rising_edge_dme is always rose(dme) -> ckslv;
16 property dme_stable is always rose(dme) -> next stable(dme) until rose(ckslv);
17 assert rising_edge_dme;
18 assert dme_stable;
19 |
20 property setting_rnw is always (not stable(rnw)) and dme='1' -> ckslv;
21 property rnw_is_stable is always fell(ckslv) and dme='1' -> next stable(rnw) until rose(ckslv);
22 assert setting_rnw;
23 assert rnw_is_stable;
24 |
25 property setting_cs is always rose(dme) -> not stable(cs_bus_int) before fell(ckslv);
26 property cs_is_stable is always (not stable(cs_bus_int)) and dme='1' -> next stable(cs_bus_int) until rose(ckslv);
27 assert setting_cs;
28 assert cs_is_stable;
29 |
30 property setting_da is always rose(dme) -> not stable(da) before fell(ckslv);
31 property da_is_stable is always (not stable(da)) and dme='1' -> next stable(da) until rose(ckslv);
32 assert setting_da;
33 assert da_is_stable;
34 |
35 property datain_is_latched is always rnw='1' and fell(ckmstr) and dme='1' -> next stable (d2ld_bus_int) until rose(ckslv);
36 assert datain_is_latched;
37 |
38 property dataout_is_latched is always rnw='0' and fell(ckslv) and dme='1' -> next stable(d2st) until rose(ckslv);
39 assert dataout_is_latched;

---

40 constant phase1: integer:= 60; -- ns
41 constant phase2: integer:= 2; -- ns
42 |
43 property nonoverlappingslaveclock is always rose(ckmstr) -> ckslv='0' until _fell(ckmstr);
44 property slaveaftermaster is always fell(ckmstr) -> rose(ckslv) before rose(ckmstr);
45 property nonoverlappingmasterclock is always rose(ckslv) -> ckmstr='0' until _fell(ckmstr);
46 property min_ph_dif1 is always fell(ckmstr) -> next_a[1 to phase1] [ckslv='0'];
47 property min_ph_dif2 is always fell(ckmstr) -> next_a[1 to phase2] [ckslv='0'];
48 |
49 assert nonoverlappingslaveclock;
50 assert slaveaftermaster;
51 assert nonoverlappingmasterclock;
52 assert min_ph_dif1;
53 assert min_ph_dif2;
54
55}
A4: GriscSystem – Postscript Wave Diagrams 1

A4 grisc_system_v.psl

vunit check_grisc_system_interface(grisc_system)
|
// signal cs_bus_int_bv: bit_vector(indexLast downto 0);
// cs_bus_int_bv <= to_bitvector(cs_bus_int);

reg clk;
initial
begin
  clk = 0;
  forever $1000 clk = ~clk;
end

wire [7:0] cs_bus_int = {cs_bus_int_7, cs_bus};
default clock = (negedge clk);

property onehotCS = (always dme -> (onehot(cs_bus) ^ cs_bus_int_7))#(negedge ckslv);
assert onehotCS;
assert (always onehot (cs_bus_int)) abort (!dme) @ (negedge ckslv);

//assert always onehot(cs_bus_int); // added for simulation time testing reason // since this
assertion is synchronised with the virtual clock, it generates an enormous assertion log file
assert (always onehot(cs_bus_int)) @ (negedge ckslv); // added for simulation time testing reason
assert (always dme); //@ (negedge ckslv); // added for simulation time testing reason

property rising_edge_dme = always rose (dme) -> ckslv;
property dme_stable = always rose(dme) -> next stable(dme) until rose(ckslv);
assert rising_edge_dme;
assert dme_stable;

property setting_rnw = always !stable(rnw) && dme -> ckslv;
property rnw_is_stable = always fell(ckslv) && dme=1'b1 -> next stable(rnw) until rose(ckslv);
assert setting_rnw;
assert rnw_is_stable;

property setting_cs = always rose(dme) -> (!stable(cs_bus) || !stable(cs_bus_int_7)) before fell (ckslv);
property cs_is_stable = always ((stable(cs_bus) || !stable(cs_bus_int_7)) && dme=1'b1 -> next
(stable(cs_bus) && stable(cs_bus_int_7)) until! rose(ckslv));
assert setting_cs;
assert cs_is_stable;

property setting_da = always rose(dme) -> !stable(da) before fell(ckslv);
property da_is_stable = always !stable(da) && dme=1'b1 -> next stable(da) until rose(ckslv);
assert setting_da;
assert da_is_stable;

property datain_is_latched = always rnw==1'b1 && fell(ckslv) && dme==1'b1 -> next stable(d2ld_bus)
until rose(ckslv);
assert datain_is_latched;

property dataout_is_latched = always rnw==1'b0 && fell(ckslv) && dme==1'b1 -> next stable(d2st) until
rose(ckslv);
assert dataout_is_latched;

A4 analog2std.psl

1 uinit check_analog2std (analog2std(bhv))
2 |
3 constant t_trans: integer:= 30; -- ns
4 constant t_trans2: integer:= 100; -- ns
5 |
6 signal clk: std_logic:= '0';
7 clk <= not clk after 1 ns;
8 default clock is rising_edge(clk);
9 |
10 -- input voltage is always less or equal to supply voltage
11 property v_input_range is always (supply_pin.v_value >= node_analog.v_value);
12 assert v_input_range;
13 |
14 -- input voltage transition time is less than t_trans
15 property rising_transition_time is always rose(node_analog.v_value > supply_pin.v_value/2.0) ->
16 (*0 to t_trans/2); (rose(node_analog.v_value > supply_pin.v_value*0.9));
17 property rising_transition_time2 is always rose(node_analog.v_value > supply_pin.v_value/2.0) ->
18 (*0 to t_trans2/2); (rose(node_analog.v_value > supply_pin.v_value*0.9));
19 |
20 property falling_transition_time is always fell(node_analog.v_value > supply_pin.v_value*0.9) ->
21 (*0 to t_trans/2); (fell(node_analog.v_value > supply_pin.v_value*0.9));
22 property falling_transition_time2 is always fell(node_analog.v_value > supply_pin.v_value*0.9) ->
23 (*0 to t_trans2/2); (fell(node_analog.v_value > supply_pin.v_value*0.9));
24 assert falling_transition_time;
25 assert falling_transition_time2;
26 |
27 property monotonically_increasing_v_in is always rose(node_analog.v_value > supply_pin.v_value/2.0)
28 -> (node_analog.v_value > prev(node_analog.v_value)) until (node_analog.v_value >
29 supply_pin.v_value*0.9);)
30 property monotonically_decreasing_v_in is always fell(node_analog.v_value > supply_pin.v_value/2.0)
31 -> (node_analog.v_value < prev(node_analog.v_value)) until (node_analog.v_value <
32 supply_pin.v_value*0.9);)
33 assert monotonically_increasing_v_in;
34 assert monotonically_decreasing_v_in;
35
A4 grisc_system.psl 2

Semtech SX8724's GRISC System --

-- this PSL file verifies setting and stability of the GRISC system interface signals
-- this PSL file verifies non-overlapping and phase difference of the dual-phase system clocks
-- !!! in this PSL file, there is no the virtual clock; the properties are defined in a less understandable way !!!
-- !!! simulation will not be slowed down !!!

unit check_grisc_system_interface_2(grisc_system(rtl))

-- dme must be asserted during ckslv high and pcc='0'
-- and it must stay stable until next rising edge of ckslv
property rising_edge_dme is (always ckslv and not pcc)@rising_edge(dme);

signal clk_dme: std_logic := '0';
clk_dme <= (not dme or pcc);
property dme_stable is (always ckslv and not pcc -> pcc@rising_edge(clk_dme))@rising_edge(dme);

assert rising_edge_dme;
assert dme_stable;

-- if rnw is asserted/deasserted it must be during ckslv='1' (phase2)
-- and it must stay stable until next rising edge of ckslv

signal rnw_event: std_logic := '0';
rnw_event <= '1' , '0' after 1 ns when (rnw_int'event) else '0';
signal clk_rnw: std_logic := '0';
clk_rnw <= rnw_event or pcc;

property setting_rnw is (always (dme and not pcc) -> ckslv)@rising_edge(rnw_event);
property rnw_is_stable is (always (dme and ckslv and not pcc) -> next pcc)@rising_edge(clk_rnw);

assert setting_rnw;
assert rnw_is_stable;

-- if dme is asserted, signal cs_bus_int must be asserted before next falling edge of ckslv (phase2)
-- and it must stay stable until next rising edge of ckslv

signal cs_event: std_logic := '0';
cs_event <= '1' , '0' after 1 ns when (cs_bus_int'event) else '0';
signal clk_cs: std_logic := '0';
clk_cs <= cs_event or pcc;

property setting_cs is (always (ckslv and not pcc) -> (ckslv and not pcc)@rising_edge(clk_cs))@rising_edge(dme);
property cs_is_stable is (always (dme and ckslv and not pcc) -> next pcc)@rising_edge(clk_cs);

assert setting_cs;
assert cs_is_stable;

-- if dme is asserted, signal da must be asserted before next falling edge of ckslv (phase2)
-- and it must stay stable until next rising edge of ckslv

signal da_event: std_logic := '0';
da_event <= '1' , '0' after 1 ns when (da_int'event) else '0';
signal clk_da: std_logic := '0';
clk_da <= da_event or pcc;

property setting_da is (always (ckslv and not pcc) -> (ckslv and not pcc)@rising_edge(clk_da))@rising_edge(dme);
property da_is_stable is (always (dme and ckslv and not pcc) -> next pcc)@rising_edge(clk_da);

assert setting_da;
assert da_is_stable;

-- datain must be asserted during ckslv high (phase2)

signal d2ld_event: std_logic := '0';
d2ld_event <= '1' , '0' after 1 ns when (d2ld_event) else '0';
property datain_is_latched is (always (dme and not pcc and rnw) -> ckslv)@rising_edge(d2ld_event);

assert datain_is_latched;

-- dataout must be asserted during ckslv high (phase2)

signal d2st_event: std_logic := '0';
d2st_event <= '1' , '0' after 1 ns when (d2st_int'event) else '0';
property dataout_is_latched is (always (dme and not pcc and not rnw) -> ckslv)@rising_edge
(d2st_event);

assert dataout_is_latched;

constant phase1: time := 120 ns; -- ns
constant phase2: time := 4 ns; -- ns

signal ckmstr_d1: std_logic := '0';
ckmstr_d1 <= transport ckmstr after phase1;
signal ckmstr_d2: std_logic := '0';
ckmstr_d2 <= transport ckmstr after phase2;

property nonoverlappingslaveclock is (always ckmstr='0' @rising_edge(ckslv));
property nonoverlappingmasterclock is (always ckslv='0' @rising_edge(ckmstr));

property min_ph_diff1a is (always ckmstr_d1='0' @rising_edge(ckslv));
property min_ph_diff1b is (always ckslv='0' @rising_edge(ckmstr_d1));
property min_ph_diff2b is (always ckslv='0' @rising_edge(ckmstr_d2));

assert nonoverlappingslaveclock;
assert nonoverlappingmasterclock;
aassert min_ph_diff1a;
aassert min_ph_diff1b;
aassert min_ph_diff2b;

}
assert valid_cs;
-- assert valid_cs2;
assert valid_cs_1;
assert valid_cs_2;
assert valid_cs_3;
assert valid_cs_4;
assert valid_cs_5;
assert valid_cs_6;
assert valid_cs_7;
assert valid_cs_8;
----------------------------------------------------------------------------------------------- -- data bus protocol

property dme_only_event is (always event_int)@rising_edge(dme);

-- when there is an event, in the next cycle the griscore will read the first status register (the status register of the first peripheral unit),
-- so status_register_reading is always (not event_int) -> (dme='1' and reg='1' and d2l_bus_int(8 - add_ind) (1)='0' -->
-- next rw='1' and add=prev(add_ind)+1 and reg=0)
@rising_edge(dme_psl);
assert next_status_register_reading;

-- if the event is not from a peripheral unit being checked currently, the griscore will check the status register of the next one
property next_status_register_reading is (always reg_ind=0 and prev(reg_ind)=0 and rw='1' and d2l_bus_int(8 - add_ind) (1)='1' -->
-- next rw='1' and add=prev(add_ind)+1 and reg=0)
@rising_edge(dme_psl);
assert next_status_register_reading;

-- if the event is coming from a currently checking peripheral unit, in the next cycle the griscore
-- will read the address register
property address_register_reading is (always reg_ind=0 and prev(reg_ind)='1' and rw='1' and
d2l_bus_int(8 - add_ind) (1)='1' --> next rw='1' and add=prev(add_ind) and reg=1 )@rising_edge(dme_psl);
assert address_register_reading;

-- after reading an address register, and if the peripheral unit requested a writing, the griscore will
-- read data register of the same peripheral unit in the next cycle,
property data_register_reading_in_we is (always ((rw='1' and reg_ind=0 and d2l_bus_int(8 - add_ind) (0)='1') (rw='1' and reg=1)) --> rw='1' and da_int = prev(d2l_bus_int(8 - (add_ind))))
@rising_edge(dme_psl);

-- after reading an address register, and if the peripheral unit requested a reading other peripheral unit, the
-- griscore will read data register of that peripheral unit, in the next cycle,
property data_register_reading_in_re is (always ((rw='1' and reg_ind=0 and d2l_bus_int(8 - add_ind) (0)='1') (rw='1' and reg=1)) --> rw='1' and da_int = prev(d2l_bus_int(8 - (add_ind))))
@rising_edge(dme_psl);

assert data_register_reading_in_re;
assert data_register_reading_in_we;

-- after reading an address register of the selected peripheral unit, in the next cycle the griscore
-- will write the read data to the data register of the peripheral unit that requested the reading
property data_register_writing_in_re is (always ((rw='1' and reg_ind=0 and d2l_bus_int(8 - add_ind) (0)='1') (rw='1' and reg=1)) --> rw='0' and add=prev(add_ind)+2 and reg=2 and
d2st_int = prev(d2l_bus_int(8+add_ind)) )@rising_edge(dme_psl);

-- after a reading an address register of the peripheral unit that requested a writing, in the next cycle
-- the griscore will write the read data to the data register of the (selected) addressed peripheral unit 2 cycles before
property data_register_writing_in_we is (always ((rw='1' and reg_ind=0 and d2l_bus_int(8 - add_ind) (0)='0') (rw='1' and reg=1)) --> rw='0' and da_int = prev(d2l_bus_int(8 - (add_ind))+2) and
d2st_int = prev(d2l_bus_int(8+add_ind)) )@rising_edge(dme_psl);

assert data_register_writing_in_re;
assert data_register_writing_in_we;

-- after a data register writing cycle, in the next and the last cycle, the griscore will acknowledge
-- the end of the requested reading-writing event, with writing x*02* to the status register
property status_register_writing is (always (rw='1' and rw='0') --> rw='0' and add=prev(add_ind, 3) and reg=0 and d2st_int=x*02* )@rising_edge(dme_psl);
assert status_register_writing;

---

property valid_cs is (always cs_bus_int (8 - add_ind) = '1')@rising_edge(dme_psl);
-- property valid_cs2 is (always cs_bus_int = (B'100010000' sll 2))@rising_edge(dme); --to_integer
(unsigned(da_int(7 downto 4))=1 srl 2
property valid_cs_1 is (always add_ind = 1 -> cs_bus_int='10000000')@rising_edge(dme_psl);
property valid_cs_2 is (always add_ind = 2 -> cs_bus_int='10000000')@rising_edge(dme_psl);
property valid_cs_3 is (always add_ind = 3 -> cs_bus_int='00010000')@rising_edge(dme_psl);
property valid_cs_4 is (always add_ind = 4 -> cs_bus_int='00001000')@rising_edge(dme_psl);
property valid_cs_5 is (always add_ind = 5 -> cs_bus_int='00000100')@rising_edge(dme_psl);
property valid_cs_6 is (always add_ind = 6 -> cs_bus_int='00000010')@rising_edge(dme_psl);
property valid_cs_7 is (always add_ind = 7 -> cs_bus_int='00000001')@rising_edge(dme_psl);
property valid_cs_8 is (always add_ind = 8 -> cs_bus_int='00000000')@rising_edge(dme_psl);

assert valid_cs;
-- assert valid_cs2;
assert valid_cs_1;
assert valid_cs_2;
assert valid_cs_3;
assert valid_cs_4;
assert valid_cs_5;
assert valid_cs_6;
assert valid_cs_7;
assert valid_cs_8;

-----------------------------------------------------------------------------------
-- if a data bus protocol cycle is finished, signal event must be deasserted
property event_deassertion is (always dme='1' and rmw='0' and reg_ind=0 and d2st_int=x'02' -> next
event_int='0')@falling_edge(cks1v);
assert event_deassertion;

-- an acknowledge can not be received before specified minimum limit
property ack_atleast_5_cycles_after_event is (always rose(event_int) -> next_a [1 to 5] {event_int})
@rising_edge(pcc);
assert ack_atleast_5_cycles_after_event;

-- an acknowledge must be received in the specified maximum limit
property ack_within_15_cycles_after_event is (always rose(event_int) -> {[*5 to 15];fell{event_int}})
@rising_edge(pcc);
assert ack_within_15_cycles_after_event;

A4: Grisc System - Assertions Wave Diagrams 2
A4: Grisc protocol (reading) - assertions wave diagrams
A4 ADC digital controller – assertions wave diagrams
<table>
<thead>
<tr>
<th>assert_p_pga2_offset</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>0012</td>
</tr>
<tr>
<td>pga2_offset</td>
<td>0</td>
</tr>
<tr>
<td>datain</td>
<td>02</td>
</tr>
<tr>
<td>ActiveCount</td>
<td>0</td>
</tr>
<tr>
<td>assert_p_pga2_gain</td>
<td></td>
</tr>
<tr>
<td>address</td>
<td>0012</td>
</tr>
<tr>
<td>pga2_gain</td>
<td>9</td>
</tr>
<tr>
<td>datain</td>
<td>02</td>
</tr>
<tr>
<td>ActiveCount</td>
<td>0</td>
</tr>
<tr>
<td>assert_p_fin</td>
<td></td>
</tr>
<tr>
<td>address</td>
<td>0012</td>
</tr>
<tr>
<td>fin</td>
<td>9</td>
</tr>
<tr>
<td>datain</td>
<td>02</td>
</tr>
<tr>
<td>ActiveCount</td>
<td>0</td>
</tr>
</tbody>
</table>
A4 pga_top – block diagram schematics
library ieee;
use ieee.numeric_std.all;
-- use arithmetic.std_logic_arith; unsigned;

unit check_pga_top (pga_top(bhv)) is
  property p_amux_32 is (always amux"00001" -> vinp.v_value=ac.a(3).v_value and vinm.v_value=ac.a(2).v_value )@falling_edge(s1);
  assert p_amux_32;

  property p_amux_10 is (always amux"00000" -> vinp.v_value=ac.a(1).v_value and vinm.v_value=ac.a(0).v_value )@falling_edge(s1);
  assert p_amux_10;

  -- .. and so on for other AMUX values...
  property p_vmux_0 is (always vmux"0" -> vrefp.v_value=ac.vrefp0.v_value and vrefm.v_value=ac.vrefm0.v_value )@falling_edge(s1);
  assert p_vmux_0;

  property p_vmux_1 is (always vmux"1" -> vrefp.v_value=ac.vrefp1.v_value and vrefm.v_value=ac.vrefm1.v_value )@falling_edge(s1);
  assert p_vmux_1;

  property p_pga1_gain_1 is (always gain_pga"0" -> abs(voutp_pga1.v_value-voutm_pga1.v_value) > 0.995*abs(vinp_pga1.v_value-vinm_pga1.v_value) and abs(voutp_pga1.v_value-voutm_pga1.v_value)<1.005*abs(vinp_pga1.v_value-vinm_pga1.v_value) )@falling_edge(s1);
  assert p_pga1_gain_1;

  property p_pga1_gain_10 is (always gain_pga"1" -> abs(voutp_pga1.v_value-voutm_pga1.v_value) > 9.5*abs(vinp_pga1.v_value-vinm_pga1.v_value) and abs(voutp_pga1.v_value-voutm_pga1.v_value)<10.0*abs(vinp_pga1.v_value-vinm_pga1.v_value) )@falling_edge(s1);
  assert p_pga1_gain_10;

  property p_pga1_gain_10b is (always gain_pga"1" -> abs(voutp_pga1.v_value-voutm_pga1.v_value) > 9.5*abs(vinp_pga1.v_value-vinm_pga1.v_value) and abs(voutp_pga1.v_value-voutm_pga1.v_value)<10.0*abs(vinp_pga1.v_value-vinm_pga1.v_value) )@falling_edge(s1);
  assert p_pga1_gain_10b;

  property p_pga1_gain_10_10 is (always gain_pga"10" -> abs(voutp_pga1.v_value-voutm_pga1.v_value) > 9.5*abs(vinp_pga1.v_value-vinm_pga1.v_value) and abs(voutp_pga1.v_value-voutm_pga1.v_value)<10.0*abs(vinp_pga1.v_value-vinm_pga1.v_value) )@falling_edge(s1);
  assert p_pga1_gain_10_10;

  property p_pga1_gain_10_20 is (always gain_pga"10" -> abs(voutp_pga1.v_value-voutm_pga1.v_value) > 9.5*abs(vinp_pga1.v_value-vinm_pga1.v_value) and abs(voutp_pga1.v_value-voutm_pga1.v_value)<10.0*abs(vinp_pga1.v_value-vinm_pga1.v_value) )@falling_edge(s1);
  assert p_pga1_gain_10_20;

  signal gdf0f2: real := 0.0;
gdf0f2 <= (real(to_integer(ieee.numeric_std.unsigned(off_pga2[2 downto 0])))<0.2 when off_pga2(3)="0" else
                  -real(to_integer(ieee.numeric_std.unsigned(off_pga2[2 downto 0])))<0.2 when off_pga2(3)="1";
signal gdf0f3: real := 0.0;
gdf0f3 <= (real(to_integer(ieee.numeric_std.unsigned(off_pga3[5 downto 0])))<12.0 when off_pga3(6)="0" else
                  -real(to_integer(ieee.numeric_std.unsigned(off_pga3[5 downto 0])))<12.0 when off_pga3(6)="1";

  property p_pga2_gain_1 is (always gain_pga"0" -> ( abs(voutp_pga2.v_value-voutm_pga2.v_value) > 0.995*abs(vinp_pga2.v_value-vinm_pga2.v_value) ) )
                                and ( abs(voutp_pga2.v_value-voutm_pga2.v_value) < 1.005*abs(vinp_pga2.v_value-vinm_pga2.v_value) )
                    )@falling_edge(s1);
  assert p_pga2_gain_1;

  property p_pga2_gain_2 is (always gain_pga"01" -> abs(voutp_pga2.v_value-voutm_pga2.v_value)>0.995*(2.0*abs(vinp_pga2.v_value-vinm_pga2.v_value)-gdf0f2*abs(voutp_pga2.v_value-voutm_pga2.v_value))
                                and ( abs(voutp_pga2.v_value-voutm_pga2.v_value) < 1.005*(2.0*abs(vinp_pga2.v_value-vinm_pga2.v_value)-gdf0f2*abs(voutp_pga2.v_value-voutm_pga2.v_value))
                    )@falling_edge(s1);
  assert p_pga2_gain_2;

  property p_pga2_gain_5 is (always gain_pga"10" -> abs(voutp_pga2.v_value-voutm_pga2.v_value)>0.995*(5.0*abs(vinp_pga2.v_value-vinm_pga2.v_value)-gdf0f2*abs(voutp_pga2.v_value-voutm_pga2.v_value))
                                and ( abs(voutp_pga2.v_value-voutm_pga2.v_value) < 1.005*(5.0*abs(vinp_pga2.v_value-vinm_pga2.v_value)-gdf0f2*abs(voutp_pga2.v_value-voutm_pga2.v_value))
                    )@falling_edge(s1);
  assert p_pga2_gain_2;

  property p_pga2_gain_10 is (always gain_pga"11" -> abs(voutp_pga2.v_value-voutm_pga2.v_value)>0.995*(10.0*abs(vinp_pga2.v_value-vinm_pga2.v_value)-gdf0f2*abs(voutp_pga2.v_value-voutm_pga2.v_value))
                                and ( abs(voutp_pga2.v_value-voutm_pga2.v_value) < 1.005*(10.0*abs(vinp_pga2.v_value-vinm_pga2.v_value)-gdf0f2*abs(voutp_pga2.v_value-voutm_pga2.v_value))
                    )@falling_edge(s1);
  assert p_pga2_gain_10;

  property p_pga3Gain_10 is (always gain_pga3="111100" -> abs(voutp_pga3.v_value-voutm_pga3.v_value)
and it must stay stable until next rising edge of cs1v
if dmé is asserted, signal cs1v must be asserted before next falling edge of cs1v (phase2)

assert property (settop_c);  
endproperty

property _is_stable(cs_bus_int);  
@ (cs_bus_int or posedge pcc_c) dme & & cs1v & & !pcc | - > pcc;  
endproperty

assert property (_is_stable(cs_bus_int));

// and it must stay stable until next rising edge of cs1v
// dme must be asserted during cs1v high and pcc='0'
// and it must stay stable until next rising edge of cs1v
property rising_edge_dmé;  
@ (posedge dme) cs1v & & !pcc;  
endproperty

assert property (rising_edge_dmé) else $error("dme rising edge not during cs1v=1 and pcc=0.");

property dme_stable;  
@ (dme_c or posedge pcc_c) (cs1v & & !pcc) | - > dme;  
endproperty

assert property (dme_stable) else $error("dme is not stable.");

// if dmé is asserted, signal pcc must be asserted before next falling edge of cs1v (phase2)
// and it must stay stable until next rising edge of cs1v
// if dmé is asserted, deasserted it must be during cs1v='1' (phase2)
// and it must stay stable until next rising edge of cs1v
property setting_dmé;  
@ (posedge dmé) !pcc | = > cs1v;  
endproperty

assert property (setting_dmé) else $error("dmé is not stable.");

@ (negedge cs1v) dme | - > $onehot(cs_bus_int);  
endproperty

assert property (onehot_c); else $error("More than one bit of the cs bus is active.");

@ (negedge cs1v) dme | - > $onehot(cs_bus_int);  
endproperty

assert property (onehot_c) else $error("More than one bit of the cs bus is active.");

// if dmé is asserted, signal pcc must be asserted before next falling edge of cs1v (phase2)
// and it must stay stable until next rising edge of cs1v
// if dmé is asserted, deasserted it must be during cs1v='1' (phase2)
// and it must stay stable until next rising edge of cs1v
property setting_dmé;  
@ (posedge dmé) !pcc | = > @ (negedge cs1v) dme & & !pcc;  
endproperty

assert property (setting_dmé) else $error("dme is not stable.");

// if dmé is asserted, signal cs1v must be asserted before next falling edge of cs1v (phase2)
// and it must stay stable until next rising edge of cs1v
property setting_cs_bus_int;  
@ (posedge dme) cs1v & & !pcc | = > @ (negedge cs1v) dme & & !pcc;  
endproperty

assert property (setting_cs_bus_int) else $error("cs1v is not stable.");
A5 sva_acDig_dl.props

module sva_acDig_dl_props (  
   input ckslv, cs, vmux, pgal_gain, readwrite, cont, ac_irq,  
   input [7:0] address, datain, dataout,  
   input [4:0] amux,  
   input [6:0] pgal_off, pgal3_gain,  
   input [1:0] fin, lb_amp_pga, lb_amp_adc, nelenv, pgal2_gain,  
   input [3:0] pgal2_off, enable,  
   input [15:0] adc_out,  
   input [2:0] osr  
);  
logic spy_busy, spy_s1;  
real spy_vinp;  
initial begin  
$init_signal_spy("/sx8723_tb/l_sx8723/xi_sx8723ta/xi_ac_analay/h_busy", "spy_busy" );  
$init_signal_spy("/sx8723_tb/l_sx8723/xi_sx8723ta/xi_ac_analay/s1", "spy_s1" );  
$init_signal_spy("/sx8723_tb/l_sx8723/xi_sx8723ta/xi_ac_analay/pgatop/vinp.v_value", "spy_vinp" );  
$init_signal_spy("/sx8723_tb/l_sx8723/xi_sx8723ta/xi_ac_analay/pgatop/vinm.v_value", "spy_vinm" );  
$init_signal_spy("/sx8723_tb/l_sx8723/xi_sx8723ta/xi_ac_analay/pgatop/pgal_gain1", *spy_gain1* );  
$init_signal_spy("/sx8723_tb/l_sx8723/xi_sx8723ta/xi_ac_analay/pgatop/pgal_gain2", *spy_gain2* );  
$init_signal_spy("/sx8723_tb/l_sx8723/xi_sx8723ta/xi_ac_analay/pgatop/pgal3_gain1", *spy_gain3* );  
end  
property new_busy_not_before_irq;  
@ (#edge spy_s1) $rose(spy_busy) |-> !$rose(spy_busy) throughout ac_irq[1:];  
endproperty;  
assert property(new_busy_not_before_irq);  
property busy_irq_busy;  
@ (#edge spy_s1) $tell(spy_busy) |-> ac_irq #1 spy_busy & !ac_irq;  
endproperty;  
assert property(busy_irq_busy);  

$init_signal_spy("/sx8723_tb/l_sx8723/xi_sx8723ta/xi_ac_analay/l_data", "data_in" );  
$init_signal_spy("/sx8723_tb/l_sx8723/xi_sx8723ta/xi_ac_analay/l_addr", "data_addr" );  
$init_signal_spy("/sx8723_tb/l_sx8723/xi_sx8723ta/xi_ac_analay/l_fsm", "data_fsm" );  
$init_signal_spy("/sx8723_tb/l_sx8723/xi_sx8723ta/xi_ac_analay/l_clock", "data_clock" );  
$init_signal_spy("/sx8723_tb/l_sx8723/xi_sx8723ta/xi_ac_analay/l_test", "data_test" );  
$init_signal_spy("/sx8723_tb/l_sx8723/xi_sx8723ta/xi_ac_analay/l_reset", "data_reset" );  
$init_signal_spy("/sx8723_tb/l_sx8723/xi_sx8723ta/xi_ac_analay/l_power", "data_power" );  
$init_signal_spy("/sx8723_tb/l_sx8723/xi_sx8723ta/xi_ac_analay/l_i2c_recv", "data_i2c_recv" );  
$init_signal_spy("/sx8723_tb/l_sx8723/xi_sx8723ta/xi_ac_analay/l_i2c_send", "data_i2c_send" );  
$init_signal_spy("/sx8723_tb/l_sx8723/xi_sx8723ta/xi_ac_analay/l_i2c_addr", "data_i2c_addr" );  
$init_signal_spy("/sx8723_tb/l_sx8723/xi_sx8723ta/xi_ac_analay/l_i2c_cmd", "data_i2c_cmd" );  
$init_signal_spy("/sx8723_tb/l_sx8723/xi_sx8723ta/xi_ac_analay/l_i2c_data", "data_i2c_data" );  
$init_signal_spy("/sx8723_tb/l_sx8723/xi_sx8723ta/xi_ac_analay/l_i2c_writedata", "data_i2c_writedata" );  
$init_signal_spy("/sx8723_tb/l_sx8723/xi_sx8723ta/xi_ac_analay/l_i2c_readdata", "data_i2c_readdata" );  
$init_signal_spy("/sx8723_tb/l_sx8723/xi_sx8723ta/xi_ac_analay/l_i2c_readbytes", "data_i2c_readbytes" );  
$init_signal_spy("/sx8723_tb/l_sx8723/xi_sx8723ta/xi_ac_analay/l_i2c_counter", "data_i2c_counter" );
assert property (reference_input_multiplexer);

/*****************************************************************************
  * a control signal for the PGA3 offset (pga3_off) is equal to the last RegACCfg4 read value
  *****************************************************************************/

property ppga3_offset;
@ (negedge ckslv) cs && address[2:0]==3'b101 => pga3_off==$past(datain[6:0]) and (1 && $stable (pga3_off) throughout (cs && address[2:0]==3'b101)[-1]);
endproperty

assert property (ppga3_offset);

/*****************************************************************************
  * a control signal for the PGA1 gain (pga1_gain) is equal to the last RegACCfg3 read value
  *****************************************************************************/

property p_pga1_gain;
@ (negedge ckslv) cs && address[2:0]==3'b101 => pga1_gain==$past(datain[7]) and (1 && $stable (pga1_gain) throughout (cs && address[2:0]==3'b101)[-1]);
endproperty

assert property (p_pga1_gain);

/*****************************************************************************
  * a control signal for the PGA2 gain (pga2_gain) is equal to the last RegACCfg2 read value
  *****************************************************************************/

property p_pga2_gain;
@ (negedge ckslv) cs && address[2:0]==3'b100 => pga2_gain==$past(datain[5:4]) and (1 && $stable (pga2_gain) throughout (cs && address[2:0]==3'b100)[-1]);
endproperty

assert property (p_pga2_gain);

/*****************************************************************************
  * a control signal for the PGA2 offset (pga2_off) is equal to the last RegACCfg2 read value
  *****************************************************************************/

property p_pga2_offset;
@ (negedge ckslv) cs && address[2:0]==3'b100 => pga2_off==$past(datain[3:0]) and (1 && $stable (pga2_off) throughout (cs && address[2:0]==3'b100)[-1]);
endproperty

assert property (p_pga2_offset);

/*****************************************************************************
  * a control signal for the bias current in the ADC (ib_amp_adc) is equal to the last RegACCfg1 read value
  * a control signal for the bias current in the PGA1 (ib_amp_pga) is equal to the last RegACCfg1 read value
  * a control signal ENABLE is equal to the last RegACCfg1 read value
  *****************************************************************************/

property p_RegACCfg1(field,hi,lo);
@ (negedge ckslv) cs && address[2:0]==3'b011 => field==$past(datain[hi:lo]) and (1 && $stable(field) throughout (cs && address[2:0]==3'b011)[-1]);
endproperty

assert property (p_RegACCfg1(ib_amp_adc,7,6));
assert property (p_RegACCfg1(ib_amp_pga,5,4));
assert property (p_RegACCfg1(enable,3,0));
a control signal for the number of elementary conversions (nelconv) is equal to the last RegACCfg0 read value
a control signal for the over-sampling ratio (osr) is equal to the last RegACCfg0 read value
a control signal for the continuous conversion (cont) is equal to the last RegACCfg0 read value

property p_RegACCfg0(field,hi,lo);
@ (negedge ckslv) cs & & address[2:0]==3'b010 => field==$past{databin[hi:lo]} and (1 #1 $stable{field}) throughout (cs & & address[2:0]==3'b010)[-1];
endproperty
assert property (p_RegACCfg0(nelconv,6,5));
assert property (p_RegACCfg0(osr,4,2));
assert property (p_RegACCfg0(cont,1,1));

property adc_result_MSB;
@ (negedge ckslv) cs & & readwrite & & address[2:0]==3'b001 |-> dataout==adc_out[15:8];
endproperty
property adc_result_LSB;
@ (negedge ckslv) cs & & readwrite & & address[2:0]==3'b000 |-> dataout==adc_out[7:0];
endproperty
assert property (adc_result_MSB);
assert property (adc_result_LSB);
endmodule

Appendix 6

A6 top_sv_vhdl.sv

program test(output int    vddv, vddz , vpumpv, vpumps , vsss, vssz ,
output real acv[6], acz[6],
wire [3:0] pad_d_dig ,
input logic pad_ready_dig ,
output logic scl_out=1, sda_out=1,
input logic scl_in, sda_in);
import EnvironmentPackage::*;
Environment env;
initial begin
env = new;
env.drv_cfg;
env.build;
foreach ( env.agt.tr[i] ) begin
  env.agt.tr[i].rand_mode(0);
  env.agt.tr[i].t_tr_constraint_mode(0);
end
env.run(sda_out, sda_in, scl_out, scl_in, acv, acz, pad_ready_dig, 50);
foreach ( env.agt.tr[i] ) begin
  if ( i<20 ) env.agt.tr[i].rand_mode(1);
end
env.run(sda_out, sda_in, scl_out, scl_in, acv, acz, pad_ready_dig, 100);
env.wrapup;
$display("the end @%0d", $time);
$finish;
end

module top;

int          vddv, vddz, vpmvp, vpmzp, vsst, vsstz, ac2v, ac3v, ac4v, ac6v, ac7v, ac2z, ac3z, ac4z, ac5z, ac6z, ac7z;
wire [3:0]   pad_d_dig;
logic        pad_ready_dig;
logic        pad_scl_dig, pad_sda_dig;
real         acv[6], acz[6];
logic        sda_out, sda_in, scl_out, scl_in;
sx8723_for_sv_tb sx(vddv, vddz, vpmvp, vpmzp, vsst, vsstz, ac2v, ac3v, ac4v, ac6v, ac7v, ac2z, ac3z, ac4z, ac5z, ac6z, ac7z, pad_d_dig, pad_ready_dig, pad_scl_dig, pad_sda_dig, pad_scl_dig_out, pad_sda_dig_out);
test    te(.*);
assign ac2v = acv[0]*10.0e7;
assign ac3v = acv[1]*10.0e7;
assign ac4v = acv[2]*10.0e7;
assign ac5v = acv[3]*10.0e7;
assign ac6v = acv[4]*10.0e7;
assign ac7v = acv[5]*10.0e7;
assign ac2z = acz[0]*10.0e7;
assign ac3z = acz[1]*10.0e7;
assign ac4z = acz[2]*10.0e7;
assign ac5z = acz[3]*10.0e7;
assign ac6z = acz[4]*10.0e7;
assign ac7z = acz[5]*10.0e7;
assign pad_scl_dig = scl_out;
assign pad_sda_dig = sda_out;
assign scl_in = pad_scl_dig_out;
assign sda_in = pad_sda_dig_out;
endmodule

A6 environment_classes.sv

-package EnvironmentPackage;
-class Transaction;
-rand bit [6:0] slvadd;
-rand bit [7:0] memadd, data;
-rand bit nwr;
-constraint c_tr   { slvadd == 72;
  memadd[7:4] inside [[1:8]];
  (memadd[7:4]==1)  -> (memadd[3:0] inside [[0:2]]);
  (memadd[7:4]==2)  -> (memadd[3:0] inside [[0:3]]);
  (memadd[7:4]==3)  -> (memadd[3:0] == 0);
  (memadd[7:4]==4)  -> (memadd[3:0] inside [[0:2]]);
  (memadd[7:4]==5)  -> (memadd[3:0] inside [[0:7]]);
  (memadd[7:4]==6)  -> (memadd[3:0] inside [[0:2]]);
  (memadd[7:4]==7)  -> (memadd[3:0] inside [[0:2]]);
  (memadd[7:4]==8)  -> (memadd[3:0] == 0);
}

// constraints for the 16 bits continues conversion, offset=0
-constraint c_tr_adc   ///nwr dist {0:1, 1:99};
  nwr==0;
  slvadd == 72;
memadd[7:4] == 5;
  // (nwr=1) -> (memadd[3:0] == 1);
  // (nwr=0) -> (memadd[3:0] inside [2:7]);
  (memadd[3:0] inside [2:7]);

  // (nwr=0) -> (memadd[3:0] == 1);
  // (nwr=1) -> (memadd[3:0] == 2);

  (memadd[3:0] == 2) -> (data[7] == 0 & & data[4:3] == 3 & & data[1:0] == 2);
  (memadd[3:0] == 3) -> (data == 8'hff);
  (memadd[3:0] == 4) -> (data[7:6] == 2'b11 & & data[3:0] == 0);
  (memadd[3:0] == 6) -> (data == 8'hd0);
  (memadd[3:0] == 7) -> (data[7:6] == 2'b00 & & data[0] == 0);

}
5'b00x10 : vin=acv[5]-acv[4];
5'b00x11 : vin=acv[7]-acv[6];
5'b10x00 : vin=acv[0]-acv[1];
5'b10x01 : vin=acv[2]-acv[3];
5'b10x10 : vin=acv[4]-acv[5];
5'b10x11 : vin=acv[6]-acv[7];

endcase
endfunction

function automatic void write_exp_adc;
    exp_adc=expadc(vin);
    $display("exp_adc= %h @ %0d", exp_adc, $time);
    exp_data[8'h51]=exp_adc[15:8];
    exp_data[8'h50]=exp_adc[7:0];
endfunction

function string adc_error;
    real err=real(' (act_adc-exp_adc)/real('exp_adc);
    if (-0.001<err && err<0.001) adc_error="adc error less than 0.1%";
    else if (-0.01<err && err<0.01) adc_error="adc error less than 1%";
    else if (-0.05<err && err<0.05) adc_error="adc error less than 5%";
    else adc_error="adc error greater than 5%";
endfunction

function void write (input Transaction tr);
    string report_str;
    if(tr.nwr == 0) begin
        // record addresses & data written to memory
        exp_data[tr.memadd] = tr.data;
        // mark address as written
        track_addr[tr.memadd] = 1;
        $display(diagnostic_file, "data written to memory %h: %h", tr.memadd, exp_data[tr.memadd]);
    end else if (tr.nwr == 1) begin
        // verify data read from memory has not been corrupted since the write
        $display(diagnostic_file, "data read from memory %h: %h", tr.memadd, tr.data);
        assert(tr.data == exp_data[tr.memadd]) else begin
            $format(report_str, "ACTUAL %h not equal EXPECTED %h", tr.data, exp_data[tr.memadd]);
            $display(diagnostic_file, "Scoreboard READ ERROR: ", report_str);
            if (tr.memadd==8'h51) act_adc[15:8]=tr.data;
            if (tr.memadd==8'h50) begin
                act_adc[7:0]=tr.data;
                $display(diagnostic_file, adc_error);
                write_exp_adc;
                ++e_cnt;
            end
        end
        // remove address after a read from address array
        track_addr[tr.memadd] = 0;
    end else begin
        // generate messages for unknown transaction type
        $format(report_str, "%s", tr.nwr);
        $display(diagnostic_file,"Unknown Transaction Type in Scoreboard", report_str);
    end
endfunction

function void wrapup;
    report;
    $fclose(diagnostic_file);
endfunction
endclass

class Config;
    // rand bit [5:0] in_use=6'b11_11_11;
    bit [5:0] in_use=6'b11_11_11;
    constraint c { in_use[0]==in_use[1];
                  in_use[2]==in_use[3];
                  in_use[4]==in_use[5];}
endclass

class Generator;
    mailbox #(Transaction) gen2agt;
    Transaction blueprint;

    function new(mailbox #(Transaction) gen2agt);
        this.gen2agt = gen2agt;
    endfunction

    function void build;
        endfunction

    task run(run_for_n_trans);
        Transaction tr;
        // forever begin
        // assert(blueprint.randomize);
        // tr = blueprint.copy;
        // gen2agt.put(tr);
        // end
    endtask

    task wrapup;
        endtask
endclass

class Agent;
    mailbox #(Transaction) gen2agt, agt2drv_i2c, drv_i2c2agt;
    Transaction tr[];
    protected Transaction tr_get;
    Scoreboard sbd;

typedef enum bit {write, read} nwr_t;
typedef struct (nwr_t nwr;
                bit [6:0] slvadd;
                bit [7:0] memadd;
                bit [7:0] data;) i2c_instruction_s;
i2c_instruction_s read_adc_msb='(read, 72, 8'h51, 8'hxx);
i2c_instruction_s read_adc_lsb='(read, 72, 8'h50, 8'hxx);
i2c_instruction_s stop_adc='(write, 72, 8'h52, 8'h00);
i2c_instruction_s start_adc='(write, 72, 8'h52, 8'bxxxxx1x);
i2c_instruction_s read_stop_adc='(read, 72, 8'h52, 8'hxx);
i2c_instruction_s instr[] =
    ' [',
      '(write, 72, 8'h80, 8'ha0),
      '(read, 72, 8'h80, 8'ha0),
      '(write, 72, 8'h80, 8'h40),
      '(write, 72, 8'h80, 8'h48),
      '(read, 72, 8'h80, 8'h40),
      '(read, 72, 8'h80, 8'h40),
      '(write, 72, 8'h80, 8'h48),
      '(write, 72, 8'h80, 8'h48),
      '(read, 72, 8'h30, 8'h40),
      '(read, 72, 8'h30, 8'h40),
      '(write, 72, 8'h70, 8'h00),
      '(write, 72, 8'h70, 8'h00),
      '(write, 72, 8'h57, 8'h02),
      '(read, 72, 8'h57, 8'h02),
      '(write, 72, 8'h56, 8'h00),
      '(write, 72, 8'h56, 8'h00),
      '(write, 72, 8'h55, 8'h00),
      '(write, 72, 8'h55, 8'h40),
      '];
function new(mailbox # (Transaction) gen2agt, agt2drv_i2c, drv_i2c2agt, scoreboard sbd);
  this.gen2agt = gen2agt;
  this.agt2drv_i2c = agt2drv_i2c;
  this.drv_i2c2agt = drv_i2c2agt;
  this.sbd = sbd;
endfunction

function void build(int n_truns=10);
  repeat (n_truns) begin
    instr.push_back(read_adc_msb);
    instr.push_back(read_adc_lsb);
  end
  instr.push_back(stop_adc);
  instr.push_back(read_stop_adc);
  tr = new[size(instr)];
  foreach (tr[i]) begin
    tr[i] = new;
    tr[i].nwr = instr[i].nwr;
    tr[i].slvadd = instr[i].slvadd;
    tr[i].memadd = instr[i].memadd;
    tr[i].data = instr[i].data;
  end
endfunction

task run(ref pad_ready_dig);
  foreach(tr[i]) begin
    assert (tr[i].randomize) $display("after randomization:");
    else $display("randomization failed:");
    $display("tr[%0d].nwr = %b	ntr[%0d].slvadd = %d
tr[%0d].memadd = %h
tr[%0d].data = %h", i, tr[i].nwr, i, tr[i].slvadd, i, tr[i].memadd, i, tr[i].data);
    if (tr[i].memadd==8'h50) wait (pad_ready_dig);
    $display("put tr[%0d] to agt2drv @%0d", i, $time);
    agt2drv_i2c.put(tr[i]);
    tr_get=new;
    $display("get tr[%0d] from drv_i2c2agt @%0d", i, $time);
    drv_i2c2agt.get(tr_get);
    $display("got tr[%0d] from drv_i2c2agt @%0d", i, $time);
    sbd.write(tr_get);
  end
  tr_get=null; agt2drv_i2c.put(tr_get);
endtask

task wrapup;
endtask

class
class Driver;

mailbox #(Transaction) agt2drv;
Transaction tr;
Scoreboard sbd;
int id;
real f, dc, ac;
real z;

function new(int id, mailbox #(Transaction) agt2drv, Scoreboard sbd);
  this.id=id;
  this.agt2drv = agt2drv;
  this.sbd = sbd;
endfunction

function void build(input real f=0.01, dc=1.3, ac=0.001, z=1.0);
  $display("Driver.build @%0d", id, $time);
  this.f=f;
  this.dc=dc;
if (id inside {1,3,5}) this.ac=ac; else this.ac=-ac;
endfunction

task run (ref real acv[6], acz[6]);
$display("Driver.run @%0d, %0d", id, $time);
//const time sample=(1.0/(f*50.0))*10.0e9;
acv[id]=5.0;
for forever begin
  #100us acv[id] = (dc + ac*$cos(2*3.14*f*$time*10e-12));
  //#100us acv[id] = (dc + ac);
  sbd.write_acv(id, acv[id]);
end
join none
display("Driver.run @%0d", $time);
endtask

task wrapup;
$display("Driver.wrapup @%0d", $time);
disable run;
endtask

endclass

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

class Driver_i2c;
mailbox #(Transaction) agt2drv_i2c, drv_i2c2agt;
protected Transaction tr;

protected bit w=1'b0, r=1'b1, a=1'b1;
protected const bit start=1'b0, stop=1'b1;
protected bit [6:0] slvadd=7'h48;
protected bit [7:0] memadd=8'h10, data=8'h aa, slvdata;
local bit [38:0] aq={start, slvadd, w, a, memadd, a, start, slvadd, nwr, a, data, a, stop};
local bit q[0];

function mailbox #(Transaction) agt2drv_i2c, drv_i2c2agt);
$display("Driver_i2c.new @%0d", $time);
this.agt2drv_i2c = agt2drv_i2c;
this.drv_i2c2agt = drv_i2c2agt;
endfunction

function void build( );
$display("Driver_i2c.build @%0d", $time);
endfunction

task automatic run (ref sda, sda_in, scl, scl_in, clk1, clk2);
int i=0;
forever begin
  i++;
  tr=new;
  $display("get tr[%0d] from agt2drv_i2c", i);
  if (agt2drv_i2c.try_get(tr)) $display("got tr[%0d] from agt2drv_i2c", i);
  else break;
  if (tr == null) $display("tr[1] is null!");
  $display("tr.memadd = %h, tr.data = %h", tr.memadd, tr.data);
  slvadd=tr.slvadd;
  memadd=tr.memadd;
  data=tr.data;
  $display("memadd= %0d", slvadd);
  $display("memadd= %h", memadd);
  $display("data= %h", data);
  aq={start, slvadd, w, a, memadd, a, start, slvadd, nwr, a, data, a, stop};
  $display("Driver_i2c.run @%0d", $time);
  $display("%b", aq);
  foreach (aq[i]) begin
    q.push_back(aq[i]);
  end
  // start
  @posedge clk2) sda = q.pop_front;
  // slvadd
  repeat (9) @posedge clk1) sda=q.pop_front;
  // wait for ack
  // memadd
  repeat (9) @posedge clk1) sda=q.pop_front;
// wait for ack
@ (posedge clk1) sda = 0;
@ (posedge scl) #0.1us sda = 1;
@ (posedge clk2) sda = q.pop_front;
repeat (9) @ (posedge clk1) sda = q.pop_front;
// wait for ack
repeat (9) @ (posedge clk1) sda = q.pop_front;
@ (posedge clk2) sda = q.pop_front;
end
endtask
task wrapup;
$display("Driver_i2c.wrapup @%0d", $time);
endtask

UPLOAD

class Driver_i2c_good extends Driver_i2c;
local const bit W = 0;
local const bit R = 1;
local logic scl = 0;
local logic clk1 = 0;
local logic clk2 = 0;

function new(mailbox # (Transaction) agt2drv_i2c, drv_i2c2agt);
super.new(agt2drv_i2c, drv_i2c2agt);
endfunction

local task automatic clocking_scl (ref scl_in, scl_out);
time T_SCL_LOW = 1.3us;
time T_SCL_HIGH = 0.6us;
time t_scl = (T_SCL_LOW + T_SCL_HIGH)/2;
fork
forever begin
 wait(scl_in); scl = 1;
 # T_SCL_HIGH scl_out = 0; scl = 0;
 # T_SCL_LOW scl_out =1;
end
forever begin
 @ (negedge scl) #(T_SCL_LOW/2) clk1 = 1;
 @ (posedge scl) clk1 = 0;
end
forever begin
 @ (posedge scl) #(T_SCL_HIGH/2) clk2 = 1;
 @ (negedge scl) clk2 = 0;
end
join_none
endtask

local task automatic mstr2slv(ref bit [7:0] slvdata, ref logic sda_out, sda_in, scl_out, scl_in, input bit WR, $);
$display("start a slave data sending session");
clocking_scl(scl_in, scl_out);
// there is only one slave, therefore if the slave doesn't send ack,
// we are trying forever to send data to the same slave
forever begin
@ (posedge clk2) sda_out = start;
$display("sending the slave address to the slave");
for (logic [2:0] i=6; i>7; --) @ (posedge clk1) begin sda_out = slvadd[i];$display("slvadd[%0d]=%b", i, slvadd[i]); end
$display("send WR bit");
@ (posedge clk1) sda_out=WR;
$display("check for ack");
// the master releases sda
// if there is no ack from the slave, start the session from the beginning
@ (posedge clk1) sda_out=1;
@ (posedge clk2) if (sda_in != 0) continue;
$display("%h", (WR) ? "reading the slave data from the slave: h" : "sending the slave data to the slave: h", slvdata);
for (logic [3:0] i=7; i>8; --) @ (posedge clk1)
if (WR==0) begin slvdata[i]=sldvdata[i];$display("sldvdata[%0d]=%b", i, slvdata[i]); end
else begin slvdata[i]=sda_in;$display("sldvdata[%0d]=%b", i, slvdata[i]); end
$display("check for the 2nd ack");
// if there is no ack, start the session from the beginning
@ (posedge clk1) sda_out=1;
@ (posedge clk2) if (WR==0 && sda_in!=0) continue; else break;
end

if (S==stop)
begin
@ (posedge clk1) sda_out=0;
@ (posedge clk2) sda_out=1; disable clocking scl;
end
endtask
task automatic run (ref sda_out, sda_in, scl_out, scl_in);
int i=0;
$display("Driver_i2c.run @%0d", $time);
forever begin
tr=new;
$display("get tr@%0d from agt2drv_i2c @%0d", i, $time);
agt2drv_i2c.get(tr);
$display("got tr@%0d from agt2drv_i2c @%0d", i, $time);
if (tr == null) begin $display("tr@%0d is null!", i); break; end
slvadd=tr.slvadd;
memadd=tr.memadd;
data=tr.data;
nwr=tr.nwr;
$display("slvadd= %0d", slvadd);
$display("memadd= %h", memadd);
$display("data= %h", data);
$display("nwr= %b", nwr);
mstr2slv(memadd, sda_out, sda_in, scl_out, scl_in, W, start);
mstr2slv(data, sda_out, sda_in, scl_out, scl_in, nwr, stop);
if (tr.data==)
$display("put tr@%0d to drv_i2c2agt @%0d", i, $time);
drv_i2c2agt.put(tr);
$display("tr@%0d is put to drv_i2c2agt @%0d", i, $time);
i++;
end
endtask
class Environment;
	Scoreboard sbd;
	Generator gen;
	Agent agt;
	Driver drv[6];
	//Driver_i2c drv_i2c;
	Driver_i2c_good drv_i2c;
	Config cfg;
	mailbox #(Transaction) gen2agt, agt2drv, agt2drv_i2c, drv_i2c2agt;
extern function new;
extern function void drv_cfg;
extern function void build;
extern task run( ref sda_out, sda_in, scl_out, scl_in, ref real acv[6], acz[6], ref pad_readyDig,
	input int run_for_n_trans=10);
extern task wrapup;
endclass
function Environment::new;
$display("Environment.new @%0d", $time);
//Initialize a configuration and a scoreboard
cfg = new;
sbd = new;
//Initialize mailboxes
gen2agt = new;
agt2drv = new;
agt2drv_i2c = new;
drv_i2c2agt = new;
//Initialize transactors
gen = new(gen2agt);
agt = new(gen2agt, agt2drv_i2c, drv_i2c2agt, sbd);
foreach (drv[]) if (cfg.in_use[]) 

drv[]=new(1, agt2drv, sbd);

drv_i2c = new(agt2drv_i2c, drv_i2c2agt);
endfunction
function void Environment::drvCfg;
    $display("Environment.drv_cfg @%0d", $time);
    assert(cfg.randomize);
    $display("in_use = %b", cfg.in_use);
endfunction

function void Environment::build;
    $display("Environment.build @%0d", $time);
gen.build;
agent.build;
foreach (drv[i]) if (cfg.in_use[i])
    drv[i].build;
endfunction

task automatic Environment::run (ref sda_out, sda_in, scl_out, scl_in, ref real acv[6], acz[6], ref pad_ready_dig, input int run_for_n_trans);
    $display("Environment.run @%0d", $time);
    fork
        gen.run(run_for_n_trans);
        agent.run(pad_ready_dig);
        foreach (drv[i])
            if (drv[i] != null) begin
                $display("drv[%0d].id = %0d", i, drv[i].id);
                drv[i].run(acv, acz);
            end
        drv_i2c.run( sda_out, sda_in, scl_out, scl_in);
    join
endtask

task Environment::wrapup;
    $display("Environment.wrapup @%0d", $time);
    fork
        gen.wrapup;
        agent.wrapup;
        foreach (drv[i]) if (cfg.in_use[i])
            drv[i].wrapup;
        drv_i2c.wrapup;
        sbd.wrapup;
    join
endtask

depackage

A6 transcript_ms_menthe3 (a part of the transcript of SX723 simulation with SV testbench)

# Reading /net/arbois/export/home3/modelsim/6.3a/testasim/tcl/vsim.pref.tcl
# // QuestaSim 6.3a Jun 25 2007 Linux 2.6.9-42.0.2.ELAMP
# // Copyright 1991-2007 Mentor Graphics Corporation
# // All Rights Reserved.
# //
# // THIS WORK CONTAINS TRADE SECRET AND
# // PROPRIETARY INFORMATION WHICH IS THE PROPERTY
# // OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS
# // AND IS SUBJECT TO LICENSE TERMS.
# //
# // vsim -L csl_41 -assertdebug -do (do wave.do; assertion fail -enable -r *; assertion pass -enable -r *
# // assertion fail -logoff -r *; assertion pass -logoff -r *; run -all; assertion report -file
# // assertion_report_menthe3.txt /top -r *) -l ./transcript_ms_menthe3.wlf deleteonquit -gui -wlf ./
# // vsim_menthe3.wlf lib_ex_project.top
# ** Note: (vsim-3812) Design is being optimized...
# Loading sv_std.std
# Loading work.top(fast)
# Loading work.EnvironmentPackage(fast)
# Loading work.test(fast)
# Loading std.standard
# Loading ieee.std_logic_1164(body)
# Environment.new #0
# Driver.new #0 #0
# Driver.new #1 #0
# Driver.new #2 #0
# Driver.new #3 #0
# Driver.new #4 #0

92
# Driver.new #5 0
# Driver.i2c.new #0
# Environment.drv cfg #0
# in_use = 111111
# Environment.build #0
# Driver.build #0 #0
# Driver.build #1 #0
# Driver.build #2 #0
# Driver.build #3 #0
# Driver.build #4 #0
# Driver.build #5 #0
# Environment.run 05000000
# randomization failed:
  # tr[0].nwr = 0
  # tr[0].slvadd = 72
  # tr[0].memadd = 80
  # tr[0].data = a0
  # put tr[0] to agt2drv 05000000
  # get tr[0] from drv_i2c2agt 05000000
  # drv[0].id= 0
  # Driver.run #0 05000000
  # drv[1].id= 1
  # Driver.run #1 05000000
  # drv[2].id= 2
  # Driver.run #2 05000000
  # drv[3].id= 3
  # Driver.run #3 05000000
  # drv[4].id= 4
  # Driver.run #4 05000000
  # drv[5].id= 5
  # Driver.run #5 05000000
  # Driver.i2c.run 05000000
  # get tr[#0] from agt2drv_i2c 05000000
  # got tr[#0] from agt2drv_i2c 05000000
  # slvadd= 72
  # memadd= 80
  # data= a0
  # nwr= 0
  # start a slave data sending session
    # sending the slave address to the slave
    # slvadd[6]=1
    # slvadd[5]=0
    # slvadd[4]=0
    # slvadd[3]=1
    # slvadd[2]=0
    # slvadd[1]=0
    # slvadd[0]=0
    # send WR bit
    # check for ack
    # sending the slave data to the slave: h 80
    # slvdata[7]=1
    # slvdata[6]=0
    # slvdata[5]=0
    # slvdata[4]=0
    # slvdata[3]=1
    # slvdata[2]=0
    # slvdata[1]=0
    # slvdata[0]=0
    # check for the 2nd ack
    # start a slave data sending session
    # sending the slave address to the slave
    # slvadd[6]=1
    # slvadd[5]=0
    # slvadd[4]=0
    # slvadd[3]=1
    # slvadd[2]=0
    # slvadd[1]=0
    # slvadd[0]=0
    # send WR bit
    # check for ack
    # sending the slave data to the slave: h a0
    # slvdata[7]=1
    # slvdata[6]=0
    # slvdata[5]=1
    # slvdata[4]=0
    # slvdata[3]=0
    # slvdata[2]=0
    # slvdata[1]=0
    # slvdata[0]=0
    # check for the 2nd ack
    # put tr[#0] to drv_i2c2agt 0134172486
    # tr[#0] is put to drv_i2c2agt 0134172486
    # get tr[#1] from agt2drv_i2c 0134172486
    # got tr[#0] from drv_i2c2agt 0134172486
    # randomization failed:
    # tr[1].nwr = 1
    # tr[1].slvadd = 72
    # tr[1].memadd = 80
    # tr[1].data = a0
    # put tr[1] to agt2drv 0134172486
# get tr[1] from drv_i2c2agt  @134172486
# got tr#1 from agt2drv_i2c  @134172486
# slvadd=  72
# memadd=  80
# data=  a0
# nwr=  1
# start a slave data sending session
# sending the slave address to the slave
# slvadd[6]=1
# slvadd[5]=0
# slvadd[4]=0
# slvadd[3]=1
# slvadd[2]=0
# slvadd[1]=0
# slvadd[0]=0
# send WR bit
# check for ack
# sending the slave data to the slave: h 80
# slvdata[7]=1
# slvdata[6]=0
# slvdata[5]=0
# slvdata[4]=0
# slvdata[3]=0
# slvdata[2]=0
# slvdata[1]=0
# slvdata[0]=0
# check for the 2nd ack
# start a slave data sending session
# sending the slave address to the slave
# slvadd[6]=1
# slvadd[5]=0
# slvadd[4]=0
# slvadd[3]=1
# slvadd[2]=0
# slvadd[1]=0
# slvadd[0]=0
# send WR bit
# check for ack
# reading the slave data from the slave: h a0
# slvdata[7]=0
# slvdata[6]=0
# slvdata[5]=0
# slvdata[4]=0
# slvdata[3]=0
# slvdata[2]=0
# slvdata[1]=0
# slvdata[0]=0
# check for the 2nd ack
# put tr#1 to drv_i2c2agt  @220654422
# tr#1 is put to drv_i2c2agt  @220654422
# get tr#2 from agt2drv_i2c  @220654422
# got tr[1] from drv_i2c2agt  @220654422
# randomization failed:
# tr[2].nwr = 0
# reading the slave data from the slave: h 00
# slvdata[7]=0
# slvdata[6]=0
# slvdata[5]=0
# slvdata[4]=0
# slvdata[3]=0
# slvdata[2]=0
# slvdata[1]=0
# slvdata[0]=0
# check for the 2nd ack
# put tr#43 to drv_i2c2agt  @26570458220
# tr#43 is put to drv_i2c2agt  @26570458220
# get tr#44 from agt2drv_i2c  @26570458220
# got tr[43] from drv_i2c2agt  @26570458220
# got tr[44] from agt2drv_i2c  @26570458220
# tr#44 is null!
# Here we are!
# Environment.wrapup  @27070468220
# Driver.wrapup  @27070468220
# Driver.wrapup  @27070468220
# Driver.wrapup  @27070468220
# Driver.wrapup  @27070468220
# Driver.wrapup  @27070468220
# Driver.wrapup  @27070468220
# Driverr_i2c.wrapup  @27070468220
# the end  @27070468220
# ** Note: $finish : . /tb/top_sv vhdl . sv(111)
# Time: 27070468230 ps Iteration: 0 Instance: /top/te
data written to memory 80: a0
data read from memory 80: 00
Scoreboard READ ERROR: ACTUAL 00 not equal EXPECTED a0
data written to memory 80: 40
data read from memory 80: 48
data written to memory 80: 18
Scoreboard READ ERROR: ACTUAL 18 not equal EXPECTED 48
data read from memory 80: 18
Scoreboard READ ERROR: ACTUAL 18 not equal EXPECTED 48
data written to memory 30: d4
data read from memory 30: 01
Scoreboard READ ERROR: ACTUAL 01 not equal EXPECTED d4
data written to memory 70: 00
data read from memory 70: 44
Scoreboard READ ERROR: ACTUAL 44 not equal EXPECTED 00
data written to memory 57: 02
data read from memory 57: 02
data written to memory 56: 00
data read from memory 56: 00
data written to memory 55: f8
data read from memory 55: f8
data written to memory 54: f0
data read from memory 54: f0
data written to memory 53: ff
data read from memory 53: ff
data written to memory 52: 52
data read from memory 52: 52
data read from memory 51: 68
Scoreboard READ ERROR: ACTUAL 68 not equal EXPECTED 00
data read from memory 50: 5d
Scoreboard READ ERROR: ACTUAL 5d not equal EXPECTED 00
adc error greater than 5%
data read from memory 51: 68
Scoreboard READ ERROR: ACTUAL 68 not equal EXPECTED 67
data read from memory 50: 5d
Scoreboard READ ERROR: ACTUAL 5d not equal EXPECTED 33
adc error less than 5%
data read from memory 51: 68
Scoreboard READ ERROR: ACTUAL 68 not equal EXPECTED 67
data read from memory 50: 5d
Scoreboard READ ERROR: ACTUAL 5d not equal EXPECTED 33
adc error less than 5%
data read from memory 51: 68
Scoreboard READ ERROR: ACTUAL 68 not equal EXPECTED 67
data read from memory 50: 5d
Scoreboard READ ERROR: ACTUAL 5d not equal EXPECTED 33
adc error less than 5%
data read from memory 51: 68
Scoreboard READ ERROR: ACTUAL 68 not equal EXPECTED 67
data read from memory 50: 5d
Scoreboard READ ERROR: ACTUAL 5d not equal EXPECTED 33
adc error less than 5%
data read from memory 51: 68
Scoreboard READ ERROR: ACTUAL 68 not equal EXPECTED 67
data read from memory 50: 5d
Scoreboard READ ERROR: ACTUAL 5d not equal EXPECTED 33
adc error less than 5%
data read from memory 51: 68
Scoreboard READ ERROR: ACTUAL 68 not equal EXPECTED 67
data read from memory 50: 5c
Scoreboard READ ERROR: ACTUAL 5c not equal EXPECTED 33
adc error less than 5%
data read from memory 51: 68
Scoreboard READ ERROR: ACTUAL 68 not equal EXPECTED 67
data read from memory 50: 5c
Scoreboard READ ERROR: ACTUAL 5c not equal EXPECTED 33
adc error less than 5%
data read from memory 51: 68
Scoreboard READ ERROR: ACTUAL 68 not equal EXPECTED 67
data read from memory 50: 5c
Scoreboard READ ERROR: ACTUAL 5c not equal EXPECTED 33
adc error less than 5%
data written to memory 52: 00
data read from memory 52: 00
data written to memory 55: f6
data written to memory 55: b9
data written to memory 55: 42
data written to memory 55: b7
data written to memory 55: ff
data written to memory 55: d3
data written to memory 55: 0a

A6 diagnostics.txt
data written to memory 55: 11
data written to memory 55: 74
data written to memory 55: bb
data written to memory 55: 8b
data written to memory 55: 37
data written to memory 57: 36
data written to memory 57: 0c
data written to memory 55: 0f
data written to memory 55: b6
data written to memory 55: cd
data written to memory 55: f3
data written to memory 55: 65
data written to memory 55: bf
data written to memory 52: 52
data read from memory 52: 52
scoreboard read error: actual 36 not equal expected 67
data read from memory 50: c9
scoreboard read error: actual c9 not equal expected 32
adc error grater than 5%
data read from memory 51: 36
scoreboard read error: actual c9 not equal expected 2d
adc error less than 5%
data read from memory 51: 36
data read from memory 50: c9
scoreboard read error: actual c9 not equal expected 2d
adc error less than 5%
data read from memory 51: 36
data read from memory 50: c9
scoreboard read error: actual c9 not equal expected 2d
adc error less than 5%
data read from memory 51: 36
data read from memory 50: c9
scoreboard read error: actual c9 not equal expected 2d
adc error less than 5%
data read from memory 51: 36
data read from memory 50: c9
scoreboard read error: actual c9 not equal expected 2d
adc error less than 5%
data read from memory 51: 36
data read from memory 50: c9
scoreboard read error: actual c9 not equal expected 2d
adc error less than 5%
data read from memory 51: 36
data read from memory 50: c9
scoreboard read error: actual c9 not equal expected 2d
adc error less than 5%
data read from memory 51: 36
data read from memory 50: c9
scoreboard read error: actual c9 not equal expected 2d
adc error less than 5%
data read from memory 51: 36
data read from memory 50: c9
scoreboard read error: actual c9 not equal expected 2d
adc error less than 5%
data read from memory 51: 36
data read from memory 50: c8
scoreboard read error: actual c8 not equal expected 2c
adc error less than 5%
data written to memory 52: 00
data read from memory 52: 00
******
report:
******
scoreboard error: written addr 55 not read
scoreboard error: written addr 57 not read
scoreboard report failure: error count = 38
Bibliography


