Requirements Driven Design Verification
Flow Tutorial

Ateş BERNA – Managing Partner
Ahmet JORGANXHI – Design & Verification Eng.
About Electra IC

• Founded in 2014
• Headquarter in Istanbul, Turkey
• Branch office in Ankara
• Total 20 people
• ASIC/FPGA D&V Services
• ASIC/FPGA/EmbSys Training Services
Introduction

Main Objectives of Tutorial

- Requirement Capture
- Advanced Verification Methodology
- RTL Analysis

Advanced Verification Methodology

- Verification with Assertion and Self-Checking Mechanism
- Verifying Assertions with SVAUnit

RTL Analysis

- Functional and Code Coverage
Requirements are definitions of “what” hardware must do.

How?
The \{output or verifiable aspect\} shall
{always, unconditionally, only}
{assert, deassert, set to value}
{before, after, when, during, within}
{xnsec, the next rising edge of a clock, read/write asserts low}
when {inputs are set to a combination of high/low, a sequence of events has occurred or a timed period elapses}
EIC_IP_CORE_FR_001:
The \{dscrt_out\} shall
\{always\}
\{assert to logic HIGH\}
\{within 40 nanoseconds\}
when \{dscrt_in is asserted to logic HIGH\}
Requirement Tracing

Requirements Document

100%

Micro Architecture Design Document

100%

Design Source Codes
Introduction to Verification

Is the Design Under Test working correctly?

Design Under Test
Goals of Verification

Verification goals

Ensure that design behaves as expected

Ensure that design does not show any unexpected behavior under illegal and error conditions
## Verification Methodologies Comparison

<table>
<thead>
<tr>
<th>Features</th>
<th>Classic VHDL</th>
<th>SV/UVM</th>
<th>UVVM/OSVVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>OOP</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Ease of use for whose who knows VHDL</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Ease of use for whose who knows Verilog</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Code Coverage</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Functional Coverage</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Development Tool Advantages</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>IEEE Standard</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Extensive Verification IP Support</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
</tbody>
</table>
SV/UVM Verification Methodology

- Sequence Methodology
- Factory Mechanism
- Config Mechanism
- UVM Phase
- Modularity and Re-Usability
Verification Procedure Document

• Verification Environment
• For each testcase:
  – Description
  – Coverpoints
  – Test Steps
Verification Procedure Document

Input Pins → AXI4-Lite Slave Interface → SPI Master Interface → Output Pins
Verification Procedure Document

Base Test

Environment

- AXI VIP Config
- SPI VIP Config
- Reg Model
- uvm_config_db
- SPI Sequence
- SPI Coverage
- AXI Sequence
- AXI Coverage

SPI Agent:
1. Driver
2. Sequencer
3. Monitor

AXI Agent:
1. Driver
2. Sequencer
3. Monitor

Interface

Design Under Test
Verification Procedure Document

How to

- checkout files
- configure verification environment
- run test case
[EIC_IP_CORE_FR_002] *spi_cs_n* shall assert to logic LOW and remain for 24.5*T_{spi_sclk} ± 1% when any of the Active State conditions of *spi_mosi* are satisfied. (*T_{spi_sclk} = 10 MHz*)

[EIC_IP_CORE_FR_005] *spi_sclk* shall assert to clock signal with the period 10 MHz when any of the Active State conditions of *spi_mosi* are satisfied.
[EIC_IP_CORE_FR_008] *spi_mosi* shall assert to opcode and data value where:

- **Opcode**: 0x04
- **Data**: SPI CONF DATA REG 2(15:0)

within $500 \times T_{s_{axi_aclk}} \pm 1\%$ when SPI CONF DATA REG 2(31) is logic HIGH. ($T_{s_{axi_aclk}} = 100$ MHz, SPI CONF DATA REG is register at address 0x8)
Verification Procedure Document

TC_ProjectName_FunctionalElement_Feature_CHECK

TC_PRJX_SPI_PORT_CHECK
TC_PRJX_SPI_PORT_CHECK

Description

Coverpoints

Test Steps
This test case verifies active state of output ports of SPI Master Interface which are listed below.

- spi_cs_n
- spi_sclk
- spi_mosi

As test scenario after applied reset, AXI4 Write Transfer to SPI CONF DATA REG 2 register which is located at address 0x8 with 31\textsuperscript{st} of Write Data set to 1 and (30:0) set to a random value will be initiated. Then, at required time, active states of spi_cs_n, spi_sclk and spi_mosi output ports are verified.

**Pass/Fail Criteria:** Assertions and checkers of requirement checking mechanism of test class should pass.
Verification Procedure Document

Coverpoints

- spi_master_cg: spi_cs_cp
- spi_master_cg: spi_opcode_cp
- spi_master_cg: spi_trans_type_cp
- axi4_slave_cg: axi4_trans_type_cp
- reg_cg: spi_conf_reg2_cp
## Verification Procedure Document

<table>
<thead>
<tr>
<th>Action</th>
<th>Expected Result</th>
<th>Assertion</th>
<th>Requirement Links</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Apply clock signal with period of 100 MHz to sys_clk input.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Set reset_n to logic LOW.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Wait for 4 to 10 sys_clk cycles.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Set reset_n to logic HIGH</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action</td>
<td>Expected Result</td>
<td>Assertion</td>
<td>Requirement Links</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>-----------------</td>
<td>-----------</td>
<td>-------------------</td>
</tr>
<tr>
<td>5. Initiate AXI4 Write Transfer to address 0x8 with 31st bit of write data set to 1 and (30:0) bits of write data set to random value.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Verification Procedure Document

<table>
<thead>
<tr>
<th>Action</th>
<th>Expected Result</th>
<th>Assertion</th>
<th>Requirement Links</th>
</tr>
</thead>
<tbody>
<tr>
<td>6. Perform steps 6.1-6.3 in parallel manner</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| 6.1 Within $500*sys\_clk\pm 1\%$ cycles check spi\_mosi.               | spi\_mosi should assert respectively to opcode and data byte in order: Opcode  : 0x04  
Data Byte : (15:0) bits of write data initiated in step 5.             |                                                                          | EIC\_IP\_CORE\_FR\_008    |
<table>
<thead>
<tr>
<th>Action</th>
<th>Expected Result</th>
<th>Assertion</th>
<th>Requirement Links</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2 Within 500<em>sys_clk± 1% check spi_cs_n. Keep checking for 24.5</em>Tspi_sclk± 1%.</td>
<td>spi_cs_n should assert and remain at logic LOW.</td>
<td>eic_ip_core_fr_003_spi_cs_n_check</td>
<td>EIC_IP_CORE_FR_003</td>
</tr>
<tr>
<td>6.3 Within 500*sys_clk± 1% check spi_sclk.</td>
<td>spi_sclk should assert to clock signal with period of 10 MHz.</td>
<td>eic_ip_core_fr_005_spi_sclk_check</td>
<td>EIC_IP_CORE_FR_005</td>
</tr>
</tbody>
</table>
Do all requirements properly linked in various components?
Requirement Verification

Verification with:

– Assertion
– Self-Checking
Verification with Assertion

“s_addr_ready shall assert to 0x1 after one clock cycle
slv_en is asserted to 0x1”
Verification with Assertion

cover property (S_ADDR_READY_ASSERT);

property S_ADDR_READY_ASSERT;
@ (posedge bus_clk)
($rose(sl_v_en)) \implies (s_addr_ready);
endproperty

assert property (S_ADDR_READY_ASSERT)
else $error("EIC_IP_CORE_FR_100 has failed.");
“i2c_sda shall assert to I2C Write Transfer Sequence within 5us after reset_n is set to logic HIGH with the following conditions:”

- **Slave Address**: 7b1101010
- **Register Address**: 0xC4
- **Write Data Byte in order**: 0xD0, 0xC1 and 0xAA
Verification with Self-Checking Mechanism

Task which verifies:
1. Time when i2c_sda is deasserted after reset.
2. Asserted value of the i2c_sda
Verification with Self-Checking Mechanism

- “I2C_sda shall assert to I2C Write Transfer Sequence within 5us after reset_n is set to logic HIGH with the following conditions:”
  - Slave Address: 7b1101010
  - Register Address: 0xC4
  - Write Data Byte in order: 0xD0, 0xC1 and 0xAA

- “I2C_sda shall assert to I2C Write Transfer Sequence within 5us after reset_n is set to logic HIGH with the following conditions:”
  - Slave Address: 7b1101010
  - Register Address: 0xC4
  - Write Data Byte in order: 0xD0, 0xC1 and 0xAA

- “I2C_sda shall assert to I2C Write Transfer Sequence within 5us after reset_n is set to logic HIGH with the following conditions:”
  - Slave Address: 7b1101010
  - Register Address: 0xC4
  - Write Data Byte in order: 0xD0, 0xC1 and 0xAA

```
wait(dut_if.reset_n == 1);
tinit = $time;
check_i2c_write
(EIC_IP_CORE_FR_102,7b1101010, 8hC4, 8hD0, 8hC1, 8hAA, tinit, 5us)
```
Verification with Self-Checking Mechanism

check_i2c_write
(Req. ID, Slave Addr., Register Addr., Write Data, Starting Time, Req. Time)

“i2c_sda shall assert to I2C Write Transfer Sequence within 5us after reset_n is set to logic HIGH with the following conditions:”

- Slave Address : 7b1101010
- Register Address : 0xC4
- Write Data Byte in order: 0xD0, 0xC1 and 0xAA

wait(m_i2c_seq.ev0.triggered);
tfinal = $time;
Verification with Self-Checking Mechanism

“i2c_sda shall assert to I2C Write Transfer Sequence within 5us after reset_n is set to logic HIGH with the following conditions:”

- **Slave Address**: 7b1101010
- **Register Address**: 0xC4
- **Write Data Byte in order**: 0xD0, 0xC1 and 0xAA

```c
wait(m_i2c_seq.ev0.triggered);
slv_addr = m_i2c_seq.data_buf;
wait(m_i2c_seq.ev0.triggered);
```
Verification with Self-Checking Mechanism

“\textit{i2c\_sda shall assert to I2C Write Transfer Sequence within 5us after reset\_n is set to logic HIGH with the following conditions:}”

\begin{itemize}
\item Slave Address : \texttt{7b1101010}
\item Register Address : \texttt{0xC4}
\item Write Data Byte in order: \texttt{0xD0, 0xC1 and 0xAA}
\end{itemize}

\begin{itemize}
\item Register Address $\rightarrow$ \texttt{reg\_addr}$\rightarrow$
\item First Write Data Byte $\rightarrow$ \texttt{wr\_data[0]}$\rightarrow$
\item Second Write Data Byte $\rightarrow$ \texttt{wr\_data[1]}$\rightarrow$
\item Third Write Data Byte $\rightarrow$\texttt{wr\_data[2]}$\rightarrow$
\end{itemize}
Verification with Self-Checking Mechanism

“i2c_sda shall assert to I2C Write Transfer Sequence within 5us after reset_n is set to logic HIGH with the following conditions:”

- Slave Address: 7b1101010
- Register Address: 0xC4
- Write Data Byte in order: 0xD0, 0xC1 and 0xAA

```verilog
if(({Slave Addr, 1b0} == slv_addr) && (Register Addr == reg_addr) && (Write Data [0] == wr_data[0]) && (Write Data [1] == wr_data[1]) && (Write Data [2] == wr_data[2]) && (tfinal – tinit <= Req. Time)) `uvm_info("req_pass", $sprint("Requirement: %s has passed", Req.ID), UVM_LOW) else `uvm_error("req_fail", $sprint("Requirement: %s has failed", Req.ID))
```
Control Register should assert to **write data byte of the AHB Write Transfer within 2 clk cycles**, after all of the following conditions are satisfied:

- `reset_n` is set to logic HIGH
- AHB Write Transfer to Control Register address is completed
Verification with Self-Checking Mechanism

Register Class

class Control_Reg extends uvm_reg;

function new(string name = “Control_Reg”)
    super.new(name, 16, UVM_NO_COVERAGE)
endfunction

virtual function void build();
    Control_Reg = uvm_reg_field::type_id::create(“Control_Reg”);
    Control_Reg.configure(this, 16, 0, “RW”, 0, 16’h0000, 1, 1, 1);
endfunction
Verification with Self-Checking Mechanism

In the test class register block is defined

REG block m_reg_block;

Control Register should assert to write data byte of the AHB Write Transfer within 2 clk cycles, after all of the following conditions are satisfied:

- reset_n is set to logic HIGH
- AHB Write Transfer to Control Register address is completed

wait(dut_if.reset_n == 1);
m_reg_block.Control_Register_h.write(status, write_value);
Verification with Self-Checking Mechanism

Control Register should assert to write data byte of the AHB Write Transfer within 2 clk cycles, after all of the following conditions are satisfied:
• reset_n is set to logic HIGH
• AHB Write Transfer to Control Register address is completed

get_value = m_reg_block.Control_Register_h.get(status);

Control Register should assert to write data byte of the AHB Write Transfer within 2 clk cycles, after all of the following conditions are satisfied:
• reset_n is set to logic HIGH
• AHB Write Transfer to Control Register address is completed

repeat(2) @(posedge bus_if.clk);
m_reg_block.Control_Register_h.read(status, read_value);
Verification with Self-Checking Mechanism

Control Register should assert to write data byte of the AHB Write Transfer within 2 clk cycles, after all of the following conditions are satisfied:

- reset_n is set to logic HIGH
- AHB Write Transfer to Control Register address is completed

If(read_value == get_value)
  `uvm_info("req_pass", $sprint("Requirement EIC_IP_CORE_FR_103 has passed"), UVM_LOW)
else
  `uvm_error("req_fail", $sprint("Requirement: EIC_IP_CORE_FR_103 has failed", Req. ID))
Verification of Assertion with SVA Unit

Source:
SystemVerilog Assertion Verification with SVAUnit
Retrieved from: https://www.amiq.com
Verification of Assertion with SVA Unit

Is assertion working as specified?
Verification of Assertion with SVA Unit

<table>
<thead>
<tr>
<th>SVA Unit checks</th>
<th>CHECK_SVA_EXISTS</th>
<th>CHECK_SVA_ENABLED</th>
<th>CHECK_SVA_PASSED</th>
<th>CHECK_SVA_FAILED</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2C_CLK_SVA</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Coverage Analysis

We need metrics for...
Coverage Analysis

• Functional Coverage
• Code Coverage
• Other Coverage Types
  – Linting (quality of RTL code)
  – Clock Domain Crossing (metastability)
Functional Coverage

Design Behaviours + Do you hit them? + Do they violate? = Functional Coverage
Functional Coverage

cover property (S_ADDR_READY_ASSERT);

property S_ADDR_READY_ASSERT;
@ (posedge bus_clk)
($rose(slv_en)) => (s_addr_ready);
endproperty

assert property (S_ADDR_READY_ASSERT)
else $error("EIC_IP-CORE_FR_100 is failed.");
Functional Coverage

interconnect_cg

mas_trans_cp
- bins single_wr
- bins single_rd
- bins burst_wr
- bins burst_rd

Crossed

slv_en_cp
- bins slv_en_0
- bins slv_en_1
- bins slv_en_2
Functional Coverage

Test Plan (.xml document)

Functional Coverage Analysis Report (.html document)

Assertions

Covergroup
Coverpoints
# Functional Coverage

## Coverage Summary by Testplan Section:

<table>
<thead>
<tr>
<th>Scope</th>
<th>Coverage</th>
<th>% of Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.1 TC_PRJX_SPI_PORT_CHECK</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>3.2.1.1 TC_PRJX_SPI_PORT_CHECK_1cp</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>3.2.1.2 TC_PRJX_SPI_PORT_CHECK_2a</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>3.2.1.3 TC_PRJX_SPI_PORT_CHECK_3cd</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
</tbody>
</table>
Code Coverage

• **Statement**—Did we cover every statement?
• **Branch**—Did we cover every IF branch and CASE entry?
• **Finite State Machine**—Did we cover all states and transitions?
• **Expression**—Did we fully test our single-bit expressions?
• **Condition**—Did we test all the conditions in our IF statements?
process (a)
begin
  z <= '0'; if (a = 0) then z <= '1';
end process;
if (a = '0') then
    z <= '1';
else
    z <= '0';
end if;

case a is
    when '0' => z <= '1';
    when '1' => z <= '0';
    when others => z <= '0';
end case;
Code Coverage

Transition Coverage

State Coverage

FSM Coverage

IDLE

S1
Z = 1

S2
Z = 0

Reset = 1
A = 0
A = 1
B = 1
A = 1
if (a or b) then
  Z <= C + 1;
end if;

• Condition 1 -> a = 1
• Condition 2 -> b = 1
### Code Coverage

#### Expression Coverage

\[ Z \leq A \text{ or } B; \]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Code Coverage

Why corresponding statement is not covered?

Why corresponding IF branch or CASE entry is not covered?

Why corresponding state or transition is not covered?

Why corresponding single-bit expression is not covered?

Why corresponding condition in related IF is not covered?
Code Coverage

• Reason of the code coverage holes:
  – Unused Codes Lines
  – Missing Test Scenario
Code Coverage

Code Coverage Improvement Graphics

Reg_01.10.2018
Reg_03.10.2018
Reg_06.10.2018
Reg_10.10.2018

Statement  Branch  Expression
RTL Analysis

Requirements Capture
- New RTL
- 3rd Party IP
- Reuse IP

Verification
- RTL Static Analysis
- Clock Domain Crossing Analysis

Synthesis

P&R
RTL Analysis

Static Analysis

- Coding Style Analysis
- Run Simulation and Synthesis Analysis
- Long if-then-else Analysis
- Language Check
- Clock Domain Analysis
- Run Basic Checks
- Clock and Reset Signals Check
- Run Reset Analysis
- FSM Analysis
- Run Structural Checks
RTL Analysis – Static Checks

Package: DO-254

Description: This package includes checks that aid in compliance with the DO-254 military standard.

Checks:

- GRST (Gated reset)
- HCCC (Do not hard-code constants)
- REGU (Register all module outputs)
- UNREACHABLE_STATE (Report unreachable states)
- MCD (No case default)
- MDA (Missing case default assignment)
- CSL (Complete sensitivity lists)
- MEB (Missing else block)
- ETB (Empty then block)
- LEC (Little endian checks)
- CCLP (Infinite and 0-count loops)
- SLCC (Separate lines for commands)
- INT_TRI (Internal tristate objects)
- COMMENT_END_STMTS (Comment end statements)
- MCA (Missing case item assignment)
- MIA (Missing if assignment)
- LATCH_CREATED (Report on every latch created or inferred)
- COMMENT_NET_DEC (Comment net declarations)
- STATE_VAR_NAME (Run name analysis on FSM state variable names)
- COMMENT_PORT_DEC (Comment port declarations)
- RST (All flip-flops resettable)
- RESET_POLARITY (Check reset polarity consistency)
- FOREIGN_LANGUAGE_KWD (Report usage of Foreign Language)
RTL Analysis – FSM Checks

```
FLU5: begin
  // To Fix Uncomment Section below
  if (!if_stall && !id_freeze) begin
    state <= #1 IDLE;
    except_type <= #1 `OR1200_EXCEPTION_NONE;
    extend_flush_last <= #1 1'b0;
    end
  else state <= #1 FLU6;
  end

  state <= #1 FLU5; // To fix comment it
end

FLU6: begin
  if (!if_stall && !id_freeze) begin
    state <= #1 IDLE;
    except_type <= #1 `OR1200_EXCEPTION_NONE;
    extend_flush_last <= #1 1'b0;
    end
  else state <= #1 FLU8; // To fix uncomment it
end
```
RTL Analysis – Long Path Analysis
2016 Industry research reports
Clocking/CDC Errors are the #2 cause of respins
RTL Analysis - CDC
Questions