Functional & Temporal Verification

ST – Integrated Systems

Unicad Training Center
Introduction / Verification Techniques / …

ENSIMAG – 2A
Circuits Numériques Vérification Fonctionnelle
Année 2014-2015
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional Verification Overview</td>
<td>04</td>
</tr>
<tr>
<td>TestBench</td>
<td>28</td>
</tr>
<tr>
<td>Code Coverage simulation</td>
<td>49</td>
</tr>
<tr>
<td>Cycle Based simulation</td>
<td>65</td>
</tr>
<tr>
<td>Property Checking</td>
<td>68</td>
</tr>
<tr>
<td>Equivalence Checking</td>
<td>78</td>
</tr>
<tr>
<td>Acceleration</td>
<td>87</td>
</tr>
<tr>
<td>Emulation</td>
<td>96</td>
</tr>
<tr>
<td>Temporal Verification</td>
<td>106</td>
</tr>
<tr>
<td>Conclusion</td>
<td>119</td>
</tr>
</tbody>
</table>
Session Rules

• **It’s an interactive session**
  • Be Participative and Constructive

• If you already know one domain
  • Let me know, we can go faster

• If you want more details on another domain
  • We can go into more details

**Be attentive!**

• It is not a training on tools…
Training Objective

• At the end of the class you will be able to:
  
  • Explain what is a Functional Verification
  • Describe the basic taxonomy of functional verification
  • List the different Verification Techniques with a list of tools and languages
  • Explain when to apply Verification techniques on a design
  
  • Explain what is a temporal verification and the link with the functional verification
Introduction / Verification Techniques / …

Overview

- Introduction
- RTL Verification
- Circuit Validation
- Verification Platform
- Static Timing Analysis
Module Objective

• To get an overview of the verification tools, techniques, methodology and languages

• Further modules describe more in details every verification technique
What is Verification?

- Functional Verification?
- Physical Verification?
- Manufacturing Verification?
- Package Verification?
What is Verification? Design Risk

Specifications

VALIDATION

Simulation

Block Diagram

Design ‘FE’

EQ, STA

Design ‘BE’

DRC, LVS

Wafer Fab

Graphic Database

ATPG, …
Verification versus Test

Design Flow Introduction Training

Specifications → Design → Netlist → Manufacturing → Silicon

- **Verification**: ensures the implementation is conform to specifications in term of functionality
- **Testing**: Ensure functionality has been preserved by industrial process on silicon
Why such a noise around verification?

- **Errare Humanum Est**
  - Designers are humans and write with bugs …

- Average is 0.5%-1% bug rate per number of line
  - 50000 lines of code -> 500 bugs.

- Potential wins are high
  - Saving masks expenses
  - Time to market!
  - Ensure high quality of chips - customer satisfaction
Project Complexity Risks
Where Are Your Project’s Biggest Risks?

Productivity Risks
- Changing specs & verification plans
- Ability to create massive # of tests
- Scarce metrics to guide what to do next
- Poor block, chip, system verification links
- Poor interfaces and limited reuse
- Lengthy system level and sw based tests

Quality Risks
- Lack of metrics for ‘done’
- Incomplete Spec & Plan
- Unit-level bugs
- System-level bugs
- Scenarios never thought of

Predictability Risks
- No Verification Plan
- Unable to track progress to plan
- Unexpected iterations
- Limited ‘slip’ visibility
- Multiple respins
- Field recalls

Time

Scheduled Tapeout
Final Tapeout

Confidence

95%+
Functional Verification Today

- Verification now accounts for about 70% of the design

Class of Designs:
- Small designs: focus on synthesis
- Concurrent designs: focus on Timing

Class of Designs:
- Block based designs: focus on floorplan
- Platform based designs: focus on system level
Design Levels of Hierarchy

A DVD Player Chip

Disk Interface
Audio Decoding
TV Output

Control

Bit Stream framing
VLD
IDCT
Video Decoding

System
Sub - System
Block
Gate
Transistor
Design Abstraction...

- Objects where to apply verification

- Model of the circuit at system level
- Functional specifications: internal diagram (stacks, bus path, state machines, …) at sub-system level
- Register Transfer Level: HDL structural model (VHDL or Verilog)
- Derived from the RTL by synthesis or custom based can be: Pre layout gate level or Post layout gate level
- Layout view of the circuit, full custom
Processor example:

- **Architecture:**
  - Defines the contract between hardware and software
  - Determines the instruction set (and more)
  - Change architecture => change software

- **Micro-architecture:**
  - Details the implementation of an architecture
  - Change micro-architecture => keep software
Steps to design a circuit

- Specification
- Architecture
- System level
- Model
- Mapping
- Floorplan
- Placement
- Routing
- Verification

FE .... To BE
Design Flow

Front End
- Architecture
- RTL Entry
- Gate Level Netlist (Pre layout)
  - Gate Level Netlist (Post layout)
    + Parasitics
    - Layout
    - Transistor

Back End
- Model
  - Synthesis
- Floorplan
  - Place & Route
- Extraction

Transistor Model
What is Functional Verification?...
What is Functional Verification

• Definition
  • To ensure the implementation of the function correctly fit with the specification of the function

• What we want to do:
Design Intent Diagram 1

- Conceived by the human mind
- written in a natural language
- Captured in ‘written’ specifications
• Definition of product requirements
• Written in a natural language
• May be an associated design specification…

Functional Specification
Design Intent Diagram 3

- Hardware source code
  - Transaction level model
  - Synthesizable RTL

- Software source code
  - Transaction level model
  - Embedded production software
Design Intent Diagram

- Design Intent
- Functional Specification
- Implementation

Coincident Design Intent
• What we **want to do:**
• What we **can** do:

  - Functional Specifications
  - Golden Reference
  - Implementation
  - Design

  Human Interpretation!

  Verification (Tool)

- **Functional verification process:**
  - Derive a golden implementation (‘representation’)
  - Check actual versus golden implementation
**Goal of Functional Verification**

1. **RTL Sign Off**
   - To guarantee the RTL is aligned with the specifications

2. To guarantee the RTL quality level is maintained throughout the design flow
   - Any model (gate, transistor, layout) behaves as the RTL

---

Golden Reference  
RTL  
Gate  
Transistor  

**Functional Verification**
Where do bugs come from?

- Specifications
  - Incorrect or ambiguous specifications
  - Misinterpretation of specification

- Communication
  - Missed cases
  - Protocol non-conformance
  - Resource conflicts

- Interfaces
  - Unclear SW/HW interface
  - Unclear Analog / Digital interfaces

- Integration
  - Badly wired core/s
  - unexpected order/timing/dependencies
RTL Sign-off Vision

- Focus on RTL verification (versus the golden reference)
- Only check that further implementations in the design flow preserve this RTL functionality

- Target
  - Reduce time to market (RTL = high level abstraction)
  - Maintain reusable design & verification
The Verification Plan
Functional Verification Concept

- First step:
  - Define **What** must be verified

- Second step:
  - Define **How** to verify

- Third step:
  - List **Which** tool(s)?

*Need to be formalized for complex designs!*
Verification Plan!

The Verification Plan is the specification document for the verification process.

- The Verification Plan defines:
  - What TO DO/ What NOT to do
  - Resources: How many people are needed
  - Duration: How long does it take in the design flow

- The Plan must clearly define WHEN the verification will be completed to the REQUIRED DEGREE OF CONFIDENCE.
The goal is to describe in details
- The Acceptance criteria
  - What to cover (including error recovery)
- The Methods
  - How to make an effective verification (tools, standards…)
- The Deliverables
  - How to proof design compliancy versus specifications
- The Quality measures
  - Quantifiable data: throughput, latency, verification completeness

The Verification Plan is the basis for the whole verification process. It is the reference document all along the design flow
The Verification Plan must also define:

• **Must-Have** properties:
  • To ensure the circuit functions correctly vs the specifications and meets the demand of the market
  • Any cut here must be analyzed not to impact the objective of the design

• **Should-Have** properties:
  • To bring extra-capabilities (vs competition)

• **Nice-to-Have** properties:
  • Purely optional (if time permits…a must / competition)
Verification Level

1. TEST APPROACH (global)
   - Architecture verification (AVP)
   - Implementation verification (IVP)
     - μarch of the design
     - RTL level verification
   - Random (a mix of previous ones)

2. CHECKING MECHANISM (local)
   - Self-checking
   - Comparison vs. reference model
   - On-the-fly checking (human interaction)
The Verification Plan should start with the “global” verification strategy (1-Test approach) then, for each testbench, define the “local” verification strategy (2).

Once the RTL passes all the testbenches written to verify the MUST-HAVE features, the design can be implemented (physical implementation).

At netlist level, other techniques must be used in order to speed up the run time.
Conducting Verification…

1. Review the verification plan
   - This review must be done with the two main partners: designers + verification engineers
   - e.g. coverage target is a key part of plan

   - Define what must be verified
   - Remove ambiguities in the design specification

   - Define all input parameters (with random value)
   - List all features and coverage points (define per features all parameters and possible paths)
2. Use appropriate abstraction level

<table>
<thead>
<tr>
<th>Abstraction level</th>
<th>What to Verify</th>
</tr>
</thead>
<tbody>
<tr>
<td>Board level</td>
<td>External Connectivity</td>
</tr>
<tr>
<td>System level</td>
<td>Connectivity, Transaction,</td>
</tr>
<tr>
<td>SoC</td>
<td>Data Flow</td>
</tr>
<tr>
<td>Sub-system</td>
<td></td>
</tr>
<tr>
<td>Reusable component (IP)</td>
<td>Ad-hoc functionality</td>
</tr>
<tr>
<td>Unit (design bloc)</td>
<td>Ad-hoc functionality, Basic</td>
</tr>
<tr>
<td></td>
<td>operations</td>
</tr>
</tbody>
</table>
3. Independent checkers for sign-off
   • Ultimate verification to be conducted out of the design team

<table>
<thead>
<tr>
<th>Abstraction level</th>
<th>Who Verifies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Board level</td>
<td>Verification Team</td>
</tr>
<tr>
<td></td>
<td>Emulation Team</td>
</tr>
<tr>
<td>System level</td>
<td>Verification Team</td>
</tr>
<tr>
<td>SoC</td>
<td>\textit{(in collaboration with designers)}</td>
</tr>
<tr>
<td>Sub-system</td>
<td></td>
</tr>
<tr>
<td>Reusable component (IP)</td>
<td>Designer/Verif.</td>
</tr>
<tr>
<td>Unit</td>
<td>Designer</td>
</tr>
</tbody>
</table>
4. Use appropriate technology
   • Think about verification cost!
   • Use ad-hoc verification technics

<table>
<thead>
<tr>
<th></th>
<th>MPEG-2 Video Decoder: Simulation time (clock wall) to decode 1 picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL</td>
<td>2 hours</td>
</tr>
<tr>
<td>Gate</td>
<td>2 days</td>
</tr>
<tr>
<td>Back-annotated Gate</td>
<td>1 week ?</td>
</tr>
</tbody>
</table>
### Conducting Verification

- Appropriate technology…

<table>
<thead>
<tr>
<th>Abstraction level</th>
<th>Tools (examples)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Board level</td>
<td>Emulation, Prototyping</td>
</tr>
<tr>
<td>System level</td>
<td>Co-Verification, (Co-)Emulation, Coverage</td>
</tr>
<tr>
<td>SoC</td>
<td></td>
</tr>
<tr>
<td>Sub-system</td>
<td></td>
</tr>
<tr>
<td>Reusable component (IP)</td>
<td>SW Simulation, Verification Components, (Semi-)Formal, Coverage</td>
</tr>
<tr>
<td>Unit</td>
<td>Ad-hoc SW simulation Equivalence Checking, Coverage…</td>
</tr>
</tbody>
</table>
5. Rate failure consequences

- Verification effort to put on critical issues
- e.g.: debug mode of a processor:
  - A failure here will prevent any on-chip analysis on the prototype
- e.g. bus arbiter
  - A failure here would be disastrous: exhaustive verification is mandatory (even if it is costly vs software simulation)
6. Decide when to stop verifying the design
   • Based on quantifiable metrics from the verification plan
   • Verification will always have holes!!! NEVER 100%
   • Decision to stop must be checked by the stakeholders

   • Verification is only complete when every item on the plan has been tested to an acceptable level, where "acceptable" and priorities of test have been agreed in advance and continually reviewed during the project
Verification Techniques
Verification Classes

- **Dynamic** Verification:
  - Transaction based
  - Requires stimuli to animate the design in a simulator
  - Based on testbench (model & stimuli)
  - e.g.: simulation, emulation (real-life checking)

- **Static** Verification:
  - Property and design functional description based
  - Do not requires stimuli
  - Exhaustive
  - e.g.: timing analysis, equivalence checking
Introduction / Verification Techniques / …

Verification Technique Panel…

- RTL Verification
  - Testbench
  - Code Coverage
  - Cycle based simulation
  - Property Checking

- Circuit Validation
  - Equivalence Checking

- Verification Platform
  - Acceleration
  - Emulation

- Functional Verification

- Temporal Verification
  - Static Timing Analysis
• **Testbench (TB)**
  - Transaction based (Dynamic)
  - Purpose: masters the simulation by:
    - Feeding stimuli to the design
    - Providing monitoring & checking the simulation result
  - Usage: mostly @ RTL level
    for very specific issues @ post layout gate
    (e.g. reset behavior)
• **Coverage**
  
  • Transaction based (Dynamic)
  
  • Purpose: used with TB to provide a way to measure test sequence completeness
  
  • Usage: mostly @ RTL
    
    basic sanity done @ pre layout gate level
• **Property Checking**
  • Property based (Static) - Assertion
  • Purpose: exhaustively verify the behavior of a device (e.g. protocol compliance)
  • Usage: architecture, RTL
Circuit Validation

• Equivalence Checking
  • Property based (Static)
  • Purpose: exhaustively compares different abstraction levels (faster than simulation)
  • Usage: @ any step in the design flow (mostly during back-end implementation)
            gate-gate, RTL-gate, RTL-RTL
Verification Platform…

• **Acceleration**
  - Transaction based (Dynamic)
  - Purpose: speed up software simulation. Used to exercise complex/long sequence from a TB
  - Technique: HW tool connected to a SW simulator
  - Usage: RTL, pre and post layout gate level
**Emulation**

- **Transaction based (Dynamic)**
- **Purpose:**
  - create a virtual verification platform to verify design behavior in situ
  - Low level SW development
- **Technique:** reconfigurable HW platform with simulator-like capabilities (waveform tracing, signal control, ...) than can run a synthesizable design
- **Usage:** RTL, pre layout gate level
Verification Methodology

• ‘UVM’ (Universal Verification Methodology)
  • A NEW verification methodology to handle SoC design!
  • Transaction based (Dynamic) – can include assertion
  • Purpose: a full methodology for functional verification based on SystemVerilog language
    • Providing a full reusable verification environment, monitoring & checking for simulation result
    • Design can be described in Verilog, SystemVerilog, VHDL or SystemC

• Usage: mostly @ RTL level
  also at behavioral or gate level
  • can also be used alongside assertion-based verification, hardware acceleration or emulation.

Turning Simulation into Verification!
...Temporal Verification...

• **Static Timing Analysis (STA)**
  - Property based (static…)
  - Purpose: verify timing requirements (performance versus the clock frequency)

• Usage: post layout gate level

• **Note:** this technique is not a functional verification, it is a temporal verification to insure that performances will be reached by the functional chip
Introduction / Verification Techniques /

...Verification Techniques Panel

- Architecture
- μ-Architecture
- RTL Entry
- Property Checking
- Testbench
- Coverage
- Emulation Prototyping
- Synthesis
- Floorplan
- Place & Route
- Extraction
- Gate Level Netlist (Pre layout)
- Gate Level Netlist (Post layout) + Parasitics
- Equivalence Checking
- Equivalence Checking
- Layout
- Transistor
- Static Timing Analysis

Functional Verification

ST Restricted
Testbench

Design Verification Techniques
Overview

- Introduction
- RTL Verification
- Circuit Validation
- Verification Platform
- Static Timing Analysis
RTL Verification

Testbench
Code Coverage
Cycle-based simulation
Property Checking
Verification Techniques / TestBench / Code Coverage...

RTL Verification

- RTL Verification
- Testbench
- Code Coverage
- Cycle-based simulation
- Property Checking
- Emulation
- Functional Verification
- Temporal Verification
- Static Timing Analysis
How can I ensure my design is compliant with the golden reference?
• **Definition:**
  - **Test-Bench (TB)** is the top level component of a simulation.
  - It drives the input of the **Design Under Test (DUT)**.
  - It can monitors/compare the output of the DUT.
Objective:

- To model the environment of the DUT as close as possible to the real environment of the chip -> use components!
• Testbench features:
  • Embed all necessary components to drive and observe the DUT in all modes of operation
  • Catch errors/violations and report them:
    • A good TB stops the simulation on error
  • Allow full visibility of the design during the simulation at RTL or netlist level
    • All signal or nets can be traced
    • Can be re-used after Place&Route for back-annotated simulation (including all delays)

• The testbench is not the test!
Where in the Flow

- RTL = main usage
- RTL simulation = mandatory
  - It’s the only way to check the functionality vs the golden specs
- Gate level: focus on timing on specific issues (e.g. reset stage)
• TestBench is a **dynamic technique** that is based on a functional simulation

• Usually completed with code coverage in order to check the design under test is well covered

• The TestBench must reflect the functional design environment of the DUT through stimuli created on every design input/output
Inputs

- Design Under Test (DUT)
- Key signal drivers (clocks, resets)
- Components (models)
- Test (=stimuli vectors)
• Design Under Test (DUT)
  • RTL/gate description (VHDL, Verilog)
  • Same code as for synthesis (except for hard blocks where a model for simulation must be provided)

• Note: Analog-Digital mixed signal co-verification can be conducted with ‘mixed’ simulators analog+digital. Every blocks are first of all verified in stand-alone mode
Inputs: Signals…

- Key signal drivers:
  - Clock, reset, configuration signals
  - Interface drivers: pull-up/down devices may be in the test bench to simulate the board behavior
Inputs: Models...

- Models emulate the design environment

**Diagram:**

- **PCI Model**
- **DUT: PCI – USB Translator**
- **USB Model**

**Testbench**

**Input connected models:**
- stimuli providers

**Output connected models:**
- monitors
- protocol checkers
- performance analyzers
• Bus Functional Model (BFM) – system level
  • To be used when stimuli are applied through complex protocols (PCI, …)
  • HDL based (close interaction with the DUT)
  • Example of a master BFM

Write 0x37fc @ 0xab00
• BFM usage with CPU: HW/SW co-verification
  • In conjunction with an Instruction Set Simulator (ISS)
  • Models CPU
  • Embedded software (executable image) is loaded and simulated by the ISS
  • It can have other pieces modeled (i.e. Cache, pipeline and peripherals)

• Often delivered with a debugger
• Provided by the processor vendor
• Performance verification: Validation is close to the board level
• Definition
  • Implementation of the verification plan
  • Composed of several test cases

• Objective
  • Exercise the DUT under **all possible functional modes**

• Content
  • Stimuli + expected answer (allowing automated checking)
• Golden test
  • Test case consists in applying these golden functional vectors and check the DUT output (using specific checkers)
  • In some domains (video, audio), a standard exists

• Random or directed-random test
  • A way to exercise the DUT when no reference (addon)
  • E.g: directed random for data communication
    • inject 10% of short packets (size < 10 bytes)
    • Hit protected address 3% of the time

• CRV: Constraint Random Verification (for SoC designs)
Inputs: Test Languages

- Originally using **HDL** (same as RTL function)

- Today Using **HVL** ‘**Hardware Verification Language**’
  - **Goal**: enhance verification engineer productivity
    - building more exhaustive TB with no sacrifice on speed
    - simplifying TB writing: less lines = easier maintenance
  - **How**: extension of traditional languages (HDL, C)
    - Enhanced observability/controllability of signals
    - Random/ Direct-random embedded test generator
    - Higher level of description
  - **Reusability** (Verification Components)
  - **Strong Debugging Environment**
  - **What**: mainly **e, system-verilog** (growing)
Signing off the test

- Who write the Test?

- Sign off evidence: the design will meet the specifications if it passes the test

- Before starting the design phase, the test must be reviewed and signed-off by:
  - The design team
  - The verification team
  - The product manager
Goal
- To verify the functionality of the Design Under Test (DUT)

Tool:
- Any HDL simulator (ncsim, questa, vcs, …)

Method
- Run the simulation with self-checking TB
- A visual inspection is time consuming, not repeatable and not reliable
- Simulation should end if error
  - saving run time!
Checking the Result

• Popular way
  • Comparing vectors

On the fly comparison
(preferred)

Post simulation comparison
(e.g.: diff’ing memory content)
(e.g. post process log files)
Limitations...

- TB cannot be exhaustive
  - Exhaustive tests do not exist! ...
    - All simulation-based verification suffers from the issue that you can never run enough test vectors to exhaustively test the whole design
    - How to improve???

- Run time issue
  - Too huge to be easily run at top level...
    - Long run time, can be more than a week at Layout level!
Technique: debugging the TB...

- **Goal:**
  - Ensure the TB is reliable before running the simulation!

<table>
<thead>
<tr>
<th>Verification</th>
<th>Design</th>
<th>Code Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good Pass</td>
<td>Ship a good design</td>
<td>False Positive</td>
</tr>
<tr>
<td>Fail Debug</td>
<td>Debug the testbench</td>
<td>False Negative</td>
</tr>
<tr>
<td>Good Fail</td>
<td>Ship a bad design</td>
<td></td>
</tr>
<tr>
<td>Bad Debug</td>
<td>Debug the design</td>
<td></td>
</tr>
</tbody>
</table>

[Image of caution sign]
• **Method:**
  - Visual inspection for limited portion of the design (using waveform viewers)
  - To inject known errors and to observe the result
...Directed Tests are Not Enough

• Tedious to write and maintain along evolution
  • Thousands of lines of testbench code
  • Any Design spec changes cause major verification delays
• Difficult to insure all required verification cases
  • Many different ways to reach a certain state
  • Can't implement all identified tests in test plan within project schedule
Supplementing with Random

- Random or directed-random test
  - A way to exercise the DUT when no reference (addon)
  - E.g: directed random for data communication
    - inject 10% of short packets (size < 10 bytes)
    - Hit protected address 3% of the time

- CRV: Constraint Random Verification (for SoC designs)
  - Improves test coverage
    - Better with the ability to ‘constrain’ the random vectors
    - Self checking no longer optional

- How can we tell if test goals are met?
Automated Verification – Elements

- **Stimuli generation**
  - Generates input to stimulates the design under test (dut)

- **Self-checking**
  - Verification environment determines correctness at run time

- **Coverage model and goals**
  - Ensures goals are met
  - Shows coverage of non-goal areas
  - Allows cross-correlation to identify new goal areas
Automated Verification

Verification Environment

- Automatic Stimulus Generation
- Physical Layer
- Data and Assertion Checkers
- Device
- Coverage Monitor
- Generation
- Coverage
- Self Checking

Stimulus Scenarios

ST Restricted
Limitations...

- TB cannot be exhaustive
  - Exhaustive tests do not exist! ...
    - You write the TB: are you sure to describe ALL possible configuration? Of course not!!!
    - Directed test are not enough!

- Run time issue
  - Too huge to be easily run at top level…
    - Long run time, can be more than a week at Layout level!
• Simulation runtime at gate level
  • Too huge: almost impossible to run the full test

<table>
<thead>
<tr>
<th>Time to decode a picture (MPEG-2 Decoder)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL</td>
<td>2 hours</td>
</tr>
<tr>
<td>Gate</td>
<td>1 day</td>
</tr>
<tr>
<td>Gate+parasitic</td>
<td>1 week</td>
</tr>
</tbody>
</table>

• Test features at gate level
  • To exercise specific tests that cannot be covered by any method…
  • e.g. reset sequence, power down
• The TB and all components can be used within any HDL-simulator
  • Cadence (ncsim), Mentor (Questa), Synopsys (VCS) …

• Components can be described in HDL, C, HVL
  • HVL as system-verilog is the most in fashion today!
Deliverables

• Result of the test
  • Should clearly state the passing test cases and the failing ones (if any and why!)
  • Coverage report to deliver as an evidence for test completeness

• Sign-off criterion
  • Execution of test cases on RTL and gate with no difference

• Good verification = high coverage with no test failing
Code Coverage

Design Verification Techniques

Unicad Training Center
Key points about simulation:

How much of my design has been simulated?

How can I reduce my test development time?

How can increase my test efficiency?
• Verification completeness
  • Should be measured by the test coverage
  • Design should be extensively verified until ~100%
  • eg: Assume a 32-bit space … 4 Billion cases!

• Checking for holes in verification
  • Test should highlight holes so that subsequent test cases are added to increase the coverage
• Definition

• Code coverage reports how the design has been exercised during a given simulation

```vhdl
If (a = '1' or b = '0') then
    z <= data(7 downto 0) + base
else
    z <= base << 2;
endif
```

Source file

- Exercised
- Never exercised!
Motivation

• Goal
  • Measure the test cases completeness versus code
  • Indicate whether the test exercises any construct or combination of construct
  • Report holes in verification to refine existing test cases or code...

• Coverage indicates the maturity of test cases

⚠️

Code coverage does not prove that a design is functionally correct!
Where in the Flow

- **RTL Entry**
  - **RTL** = main usage
  - rich variety of metrics

- **Gate level**:
  - very restrictive
  - Simple metrics only (e.g. toggle coverage)

**Code Coverage**

- **Synthesis**
- **Floorplan**
- **Place & Route**
- **Extraction**

- **Gate Level Netlist**
  - (Pre layout)
  - (Post layout) + Parasitics

---

TestBench / Code Coverage / Cycle Based Simulation …
• The complete simulation environment
  • Design Under Test (DUT)
  • Testbench + components
  • Testcases

• Metrics
• Code coverage is a **dynamic technique** that is applied during a functional simulation

• Usually coming with a testbench in order to validate the tb coverage on the design under test

• The code coverage itself does not perform any functional verification on the design
1. **Instrument the design**
   
   - To add checkpoints at strategic locations to record whether a particular construct has been exercised

<table>
<thead>
<tr>
<th>Filename</th>
<th>Stat</th>
<th>Br</th>
<th>Cond</th>
<th>Path</th>
<th>Tog</th>
<th>Act</th>
<th>Trig</th>
<th>Sig/Var</th>
</tr>
</thead>
<tbody>
<tr>
<td>iolib.vhd</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>gct_const.pck.vhd</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SPS2_48X20M4D4_R0_SP.vhd</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>gct_core.vhd</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>gct.vhd</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>tb_gct.vhd</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

   - **Note:** testbench itself is not instrumented
2. Run the simulation
   • Using existing testcases
   • Cumulative traces are collected

```vhdl
If (a = '1' or b = '0') then
  z <= data(7 downto 0) + base
else
  z <= base << 2;
endif
```
3. Report coverage (metrics)
   • Coverage metrics are then displayed

![Coverage Metrics](image)
• Statement coverage
  • A counter is put on each statement
  • Counter values are annotated in the HDL code

These lines are not covered by any of the testcases
Signal coverage

Counts and reports the number of time a signal changes values

- e.g.: to check a signal goes through all possible values
- e.g.: to check a signal makes all possible transitions

<table>
<thead>
<tr>
<th>Count</th>
<th>c_state</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>150</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
</tr>
</tbody>
</table>

This value is not covered by any of the test case!
• **Branch coverage**
  • Very close to statement coverage
  • Determines if all possibilities from a control flow are executed

Assume test case forces \( b = a \)
  • 100% statement coverage
  • 50% branch coverage

Never exercise branch 'False'!
• Path coverage
  • Counts the paths in a process.

<table>
<thead>
<tr>
<th>A=1</th>
<th>B = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A=1</th>
<th>B = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

Not covered

Functional Verification
• Finite State Machine coverage
  • State
  • Transition
  • Path
• Condition coverage
  • Shows which combinations in an expression are responsible for entering (or not) a branch
  • Hard to reach 100% for this criteria!

\[
\text{if } ((a = 1) \text{ or } (b = 2)) \text{ then } z <= d;
\]

Assume test case sets a and b either to 0 or 1

- 100% statement coverage
- 100% branch coverage
- 50% condition coverage

<table>
<thead>
<tr>
<th>a=1</th>
<th>b=2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>False</td>
</tr>
<tr>
<td>F</td>
<td>Covered</td>
</tr>
<tr>
<td>T</td>
<td>Covered</td>
</tr>
</tbody>
</table>
• Trigger coverage

Counter on the signals of any waiting list: action on all signal change must be validated

<table>
<thead>
<tr>
<th>Line Text</th>
<th>Count</th>
<th>Signal Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>122 p_fsm: process(a, b, c)</td>
<td>200</td>
<td>only ‘a’ changed</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>only ‘b’ changed</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>only ‘c’ changed</td>
</tr>
</tbody>
</table>

Construct entered 214 times
Covered 2/3 cases: 66% coverage
The Code Coverage completes the TB:

- Refine the test case to reach higher coverage
  => Functional Verification coverage!

- Define the minimum number of test cases to get the higher coverage (regression issue)
  => Simulation run time!

- Spot the most frequently called routines
  => Help optimizing the source code to reduce simulation time!

- Help analyzing the code quality
  => Remove unused code!
  • e.g. if (A and not(A)) : no branch condition flagged
Limitations…

- Restriction on the design style
  - Better coverage if no functions nor procedures
  - Loops should be used with caution

- Simulation runtime impact (in %)
  - e.g. faster on small block (1300 gates)

<table>
<thead>
<tr>
<th>Statement</th>
<th>Branch</th>
<th>Condition</th>
<th>Path</th>
<th>Trigger</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>25%</td>
<td>25%</td>
<td>12000%!</td>
<td>25%</td>
<td>25%</td>
<td>~2000%</td>
</tr>
</tbody>
</table>
Strategy

• RTL Level: Metrics are dependent
  • Unuseful to look for 100% path coverage if statement and branch covers less than 100%
    1. Start with Statement and Branch coverage
    2. Increase test cases to reach 100%
    3. Go on with Condition only

• Gate level coverage
  • No more construct exist at gate level!
    • Statement, branch, conditions,… : all useless!
  • Only remaining metric
    • Toggle (simplified metric from signal coverage where signal toggling (1 to 0 and 0 to 1 transitions) is counted
    • Usually done on flip-flops only
• Plug-in tools
  • Benefit: rich variety of metrics and report capabilities
  • Drawback: simulation speed! (run through an API)

• Simulator native capability
  • Reverse benefit and drawback!
Deliverables

• Coverage reports
  • Sorted by metric, indicates completeness of test
  • Warning: Reminder… coverage is not function!

• Minimum set of test cases to get the higher coverage
  • Suppress redundant test cases
  • Result is to reduce simulation run time with the best coverage!
And what’s more on tools?
Verification Closure Flow

Verification Plan

Deploy
Analyze
Control

Tests

Verification Environment
DUT
Specman Elite + Simulation

Verification Closure?

Yes
Done!

No

Functional Verification
Simulate Failure Triage

- Rapid identification of failures
  - Multiple simulations analyzed together
  - Sort and group by error message, time to failure,

- Eliminate extraneous common failures

- Select shortest failing run
  - Easier, faster debugging

- Explore coverage contribution common to failing runs
  - May point to the functionality causing the failure
Track and Report Coverage

- Executable coverage plan
  - Coverage levels are annotated in the plan

- Total coverage measurement

- Tracks progress against milestones
  - Allows aspects/phases, with different coverage goals

- HTML reports
  - For email, project website
Advanced Coverage Analysis

• Correlate coverage and scenarios
  • Run more of the most effective

• Group coverage holes
  • Clear picture of uncovered areas
  • Easy to characterize required new scenarios

• Correlate various coverage metrics types
  • Discrepancies between coverage levels in various metrics
Controlling Closure

- Refine/enhance scenarios
  - To target coverage holes revealed during analysis

- Refine coverage model
  - Based on failure analysis and the correlation between different coverage metrics

- Create next session description
  - Daily/weekly/…
  - To target all or part of the verification plan goals