ENCS5337: Chip Design Verification

Spring 2023/2024

Coverage

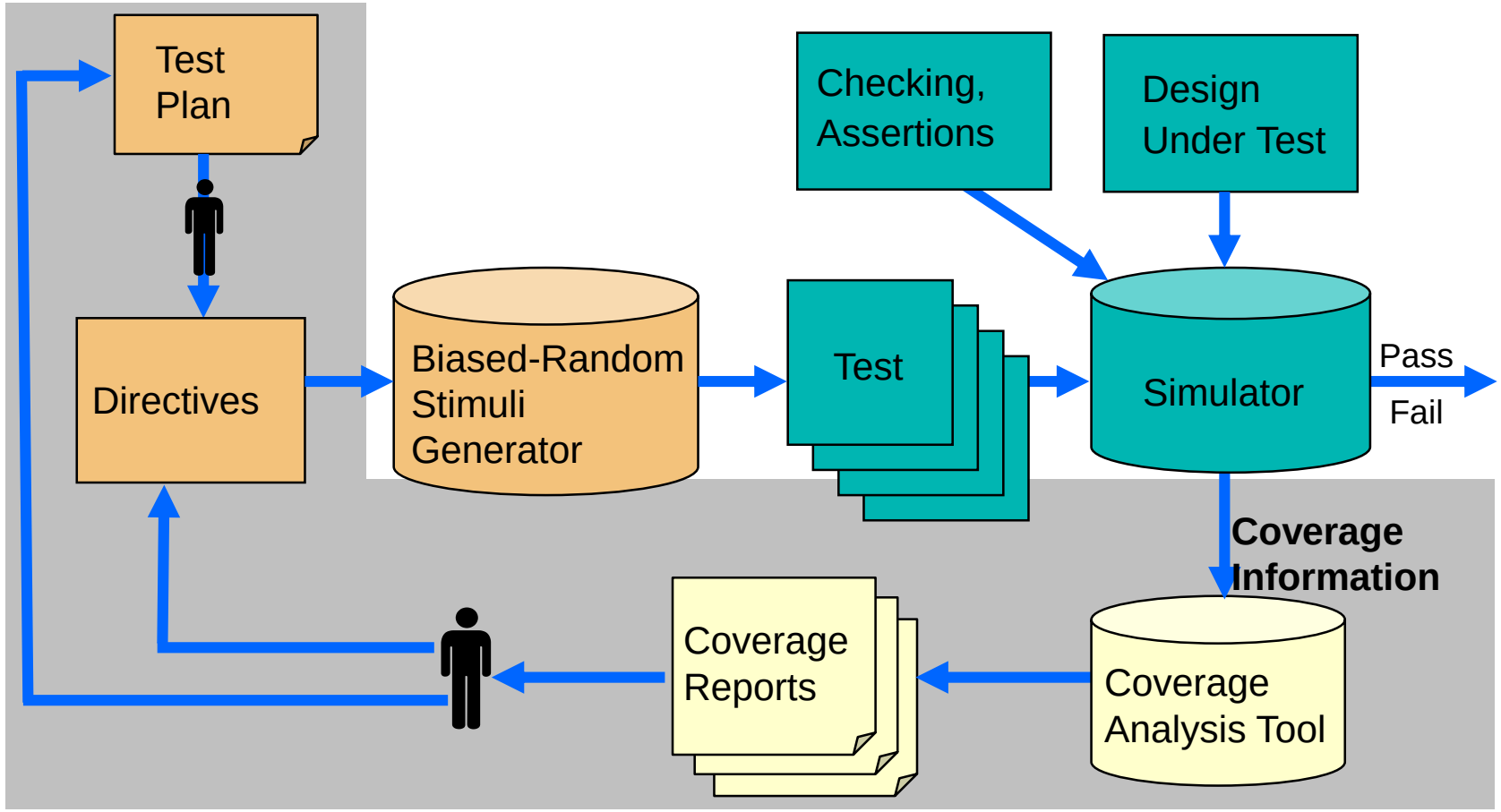
Dr. Ayman Hroub

Many thanks to Dr. Kerstin Eder for most of the slides

Outline

- Introduction to coverage
- Code coverage models
- Structural coverage models
- Functional coverage
- Case study and lessons to learn

Simulation-based Verification Environment



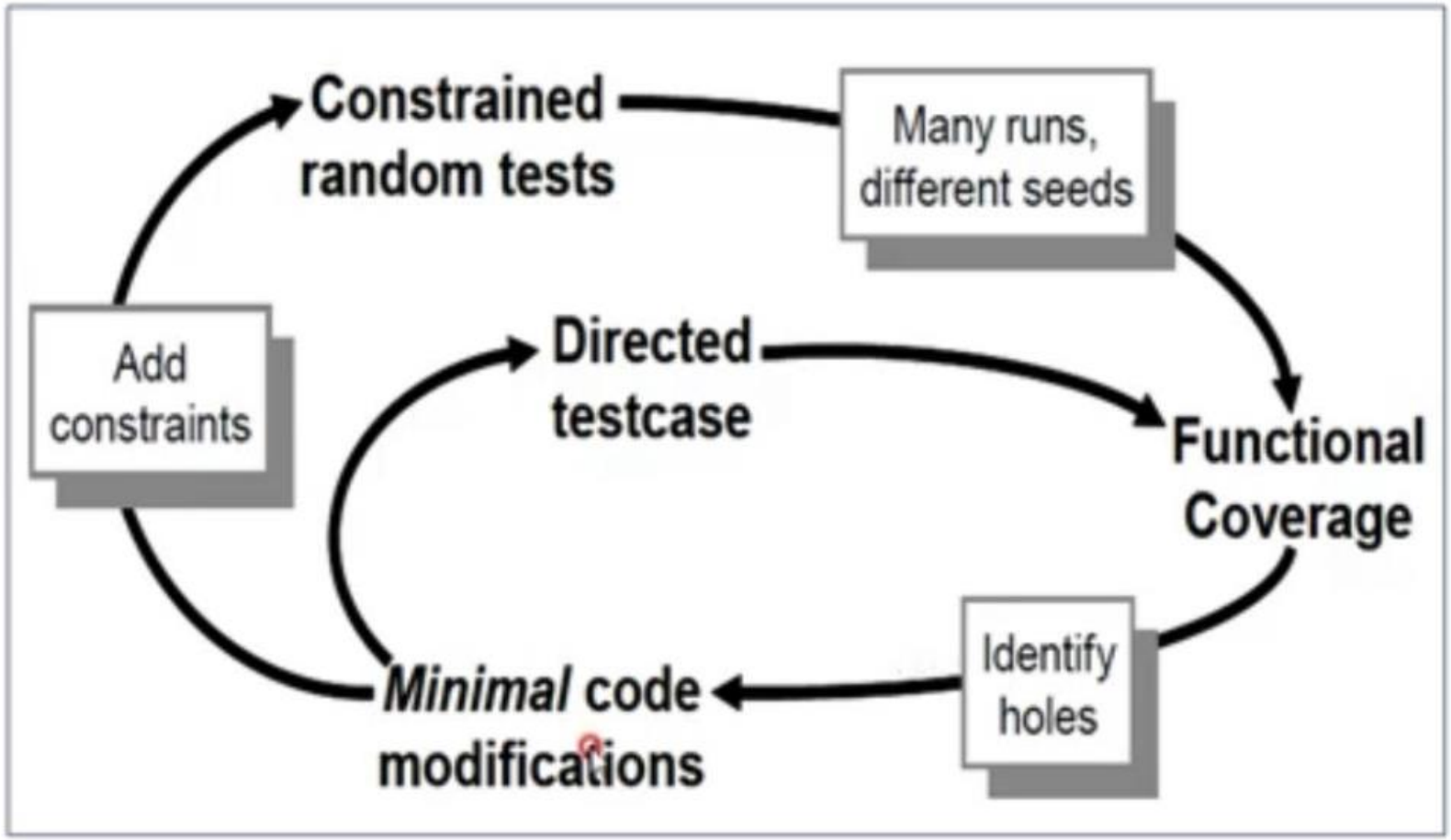
Why Coverage?

- Simulation is based on limited execution samples
  - Cannot run all possible scenarios, but
  - Need to know that all (important) areas of the DUV are verified
- Solution: Coverage measurement and analysis
- The main ideas behind coverage
  - Features (of the specification and implementation) are identified
  - Coverage models capture these features

Coverage Goals

- Measure the "quality" of a set of tests
  - NOTE: Coverage gives ability to see what **has not been** verified!
  - Coverage completeness does not imply functional correctness of the design! Why?
- Help create regression suites
  - Ensure that all parts of the DUV are covered by regression suite
- Provide stopping criteria for verification
- Improve understanding of the design

Coverage Convergence



Coverage Types

- Code coverage
- Functional coverage

- Other classifications
  - Implicit vs. explicit
  - Specification vs. implementation

Code Coverage - Basics

- Coverage models are based on the HDL code
  - Implicit, implementation coverage
- Coverage models are syntactic
  - Model definition is based on syntax and structure of the HDL
- Generic models – fit (almost) any programming language
  - Used in both software and hardware design

Code Coverage - Scope

Code coverage can answer the question:

“Is there a piece of code that has not been exercised?”

- **Useful for profiling:**
  - Run coverage on testbench to indicate what areas are executed most often.
  - **Gives insight on what to optimize!**
- Many types of code coverage report metrics/models.

Types of Code Coverage Models

- Control flow
  - Check that the control flow of the program has been fully exercised
- Data flow
  - Models that look at the flow of data in, and between, programs/modules
- Mutation
  - Models that check directly for common bugs by mutating the code and comparing results

Control Flow Models

- Routine (function entry)
  - Each function / procedure is called
- Function call
  - Each function is called from every possible location
- Function return
  - Each return statement is executed
- Statement (block)
  - Each statement in the code is executed
- Branch/Path
  - Each branch in branching statement is taken
    - `if, switch, case, when, …`
- Expression/Condition
  - Each input in a Boolean expression (condition) takes true and false values
- Loop
  - All possible number of iterations in (Bounded) loops are executed

Statement/Block Coverage

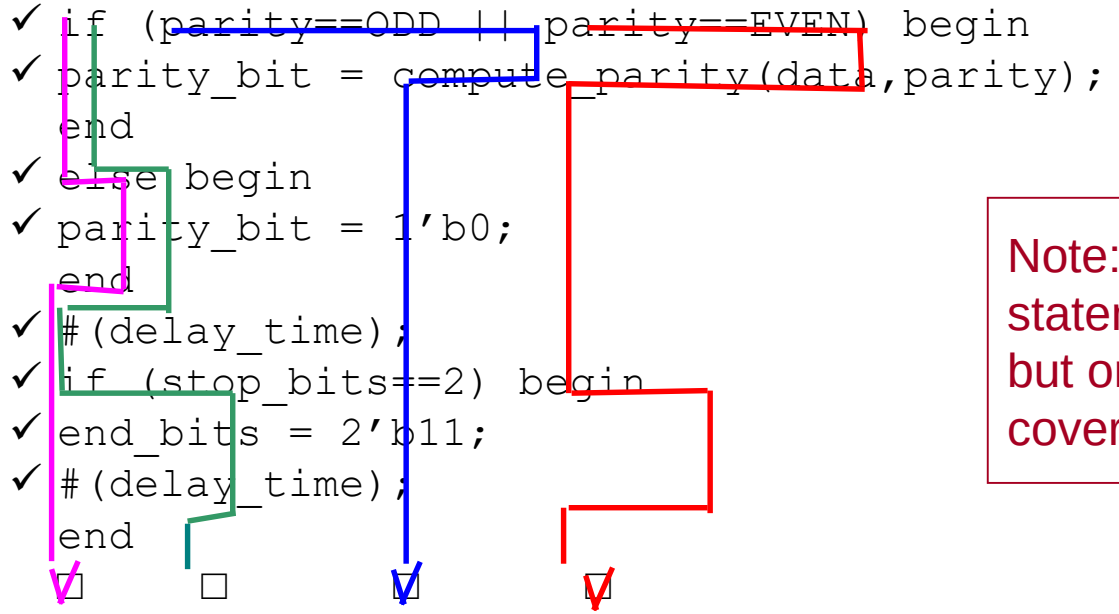
Measures which lines (statements) have been executed by the verification suite.

- ✓ `if (parity==ODD || parity==EVEN) begin`
- ❑ `parity_bit = compute_parity(data,parity);`
- `end`
- ✓ `else begin`
- ✓ `parity_bit = 1'b0;`
- `end`
- ✓ `#(delay_time);`
- ✓ `if (stop_bits==2) begin`
- ✓ `end_bits = 2'b11;`
- ✓ `#(delay_time);`
- `end`

Path/Branch Coverage

Measures all possible ways to execute a sequence of statements.

- Are all **`if/case`** branches taken?
- How many execution paths?



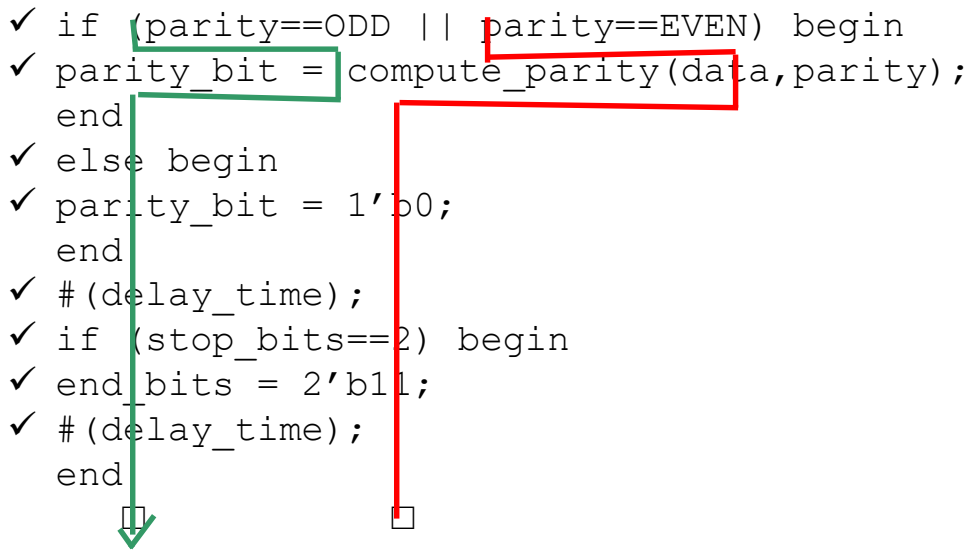
Note: 100% statement coverage but only 75% path coverage!

- **Dead code: default** branch on exhaustive **case**
- Don't measure coverage for code that was not meant to run! (tags)

Expression/Condition Coverage

Measures the various ways Boolean expressions and subexpressions are executed.

- Where a branch condition is made up of a Boolean expression, we want to know which of the inputs have been covered.



Note: Only 50% expression coverage!

- **Analysis:** Understand WHY part of an expression was not executed

- Reaching 100% expression coverage is extremely difficult.

Code Coverage Models for Hardware

- Toggle coverage
  - Each (bit) signal changed its value from 0 to 1 and from 1 to 0
- All-values coverage
  - Each (multi-bit) signal got all possible values
  - Used only for signals with small number of values
    - For example, state variables of FSMs

Code Coverage Strategy

- Set **minimum % of code coverage** depending on available verification resources and importance of preventing post tape-out bugs.

- Generally, 90% or 95% goal for statement, branch or expression coverage.
  - Some feel that less than 100% does not ensure quality.
  - Beware: Reaching full code coverage closure can cost a lot of effort!
  - This effort could be more wisely invested into other verification techniques.
- Avoid setting a goal lower than 80%.

Structural Coverage

- Implicit coverage models that are based on common structures in the code
  - FSMs, Queues, Pipelines, …
- The structures are extracted automatically from the design and pre-defined coverage models are applied to them

State-Machine Coverage

- State-machines are the essence of RTL design
- FSM coverage models are the most commonly used structural coverage models
- Types of coverage models
  - State
  - Transition (or arc)
  - Path

Code Coverage - Limitations

- Coverage questions not answered by code coverage tools
  - Did every instruction take every exception?
  - Did two instructions access the register at the same time?
  - How many times did cache miss take more than 10 cycles?
  - Does the implementation cover the functionality specified?
  - …(and many more)

- Code coverage indicates how thoroughly the test suite exercises the source code!
  - Can be used to identify outstanding corner cases
- Code coverage lets you know if you are not done!
  - It does not indicate anything about the functional correctness of the code!
- 100% code coverage does not mean very much. ☹
- Need another form of coverage!

Functional Coverage

- It is important to cover the **functionality** of the DUV.
  - Most functional requirements can't easily be mapped into lines of code!
- **Functional coverage models** are designed to assure that various aspects of the functionality of the design are verified properly, they link the requirements/specification with the implementation
- Functional coverage models are specific to a given design or family of designs
- Models cover
  - The inputs and the outputs
  - Internal states or microarchitectural features
  - Scenarios
  - Parallel properties

Functional Coverage Model Types

- Discrete set of coverage tasks
  - Set of unrelated or loosely related coverage tasks often derived from the requirements/specification
  - Often used for corner cases
    - Driving data when a FIFO is full
    - Reading from an empty FIFO
  - In many cases, there is a close link between functional coverage tasks and assertions
- Structured coverage models
  - The coverage tasks are defined in a structure that defines relations between the coverage tasks
    - Allow definition of similarity and distance between tasks
    - Most commonly used model types
      - Cross-product
      - Trees
      - Hybrid structures

Cross-Product Coverage Model

A cross-product coverage model is composed of the following parts:

1. A semantic **description** of the model (story)
2. A list of the **attributes** mentioned in the story
3. A set of all the **possible values** for each attribute (the attribute value **domains**)
4. A list of **restrictions** on the legal combinations in the cross-product of attribute values

Example: Cross-Product Coverage Model 1

Design: switch/cache unit

Motivation: Interactions of core processor unit command-response sequences can create complex and potentially unexpected conditions causing contention within the pipes in the switch/cache unit when many core processors (CPs) are active.

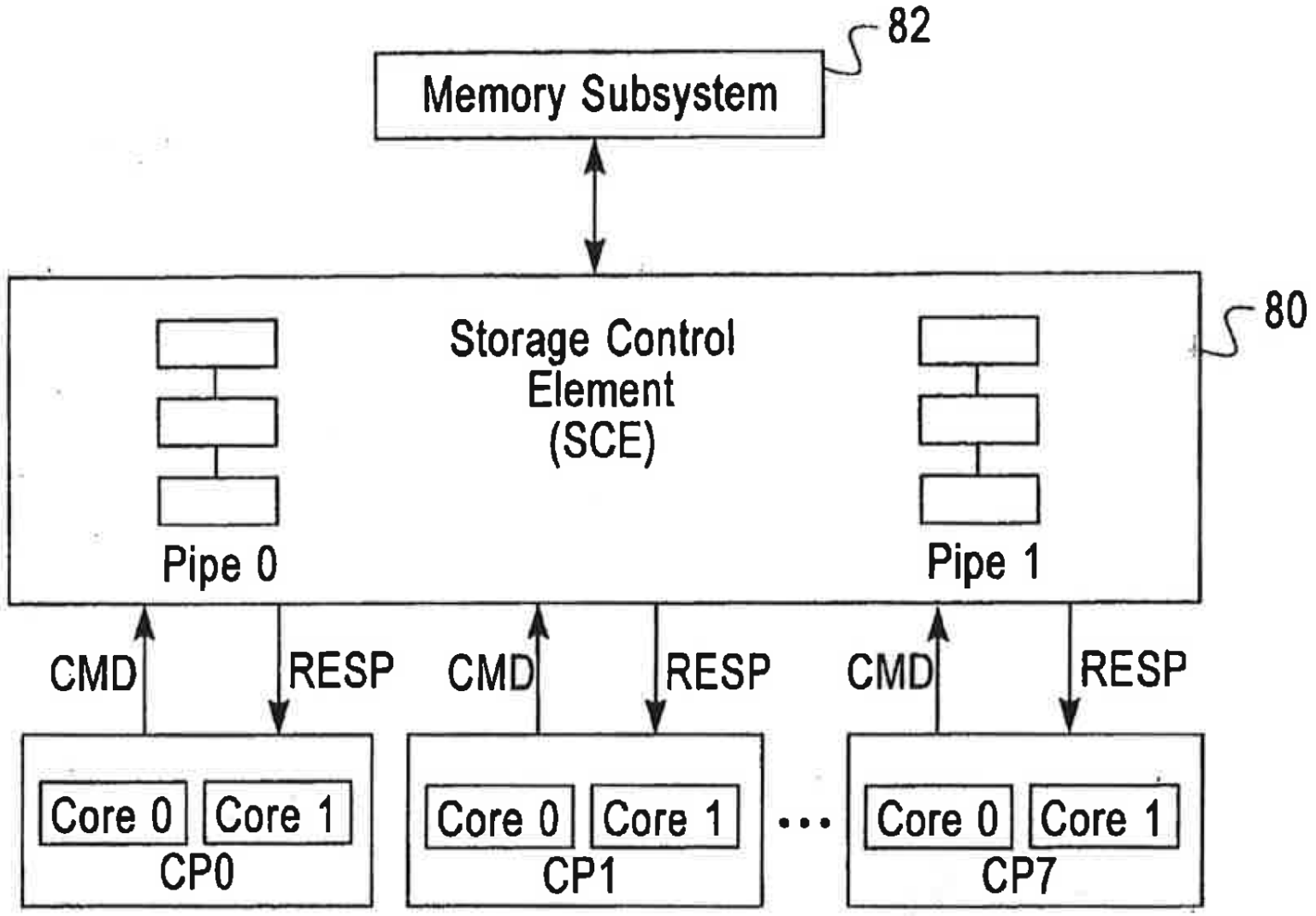
All conditions must be tested to gain confidence in design correctness.

Attributes relevant to command-response events:

- Commands - CPs to switch/cache [31]
- Responses - switch/cache to CPs [16]
- Pipes in each switch/cache [2]
- CPs in the system [8]
- (Command generators per CP chip [2])

How big is the coverage space, i.e. how many coverage tasks?

Switch/Cache Unit



Example: Cross-Product Coverage Model 2

Size of coverage space:

- Coverage space is formed by **cross-product (or, more formally, the Cartesian product) over all attribute value domains.**
- Size of cross-product is product of domain sizes:
  - 31x16x2x8x2 = 15872
- Hence, there are 15872 coverage tasks.

Example coverage task:

(Command=20, Response=01, Pipe=1, CP=5, CG=0)

Are all of these tasks reachable/legal?

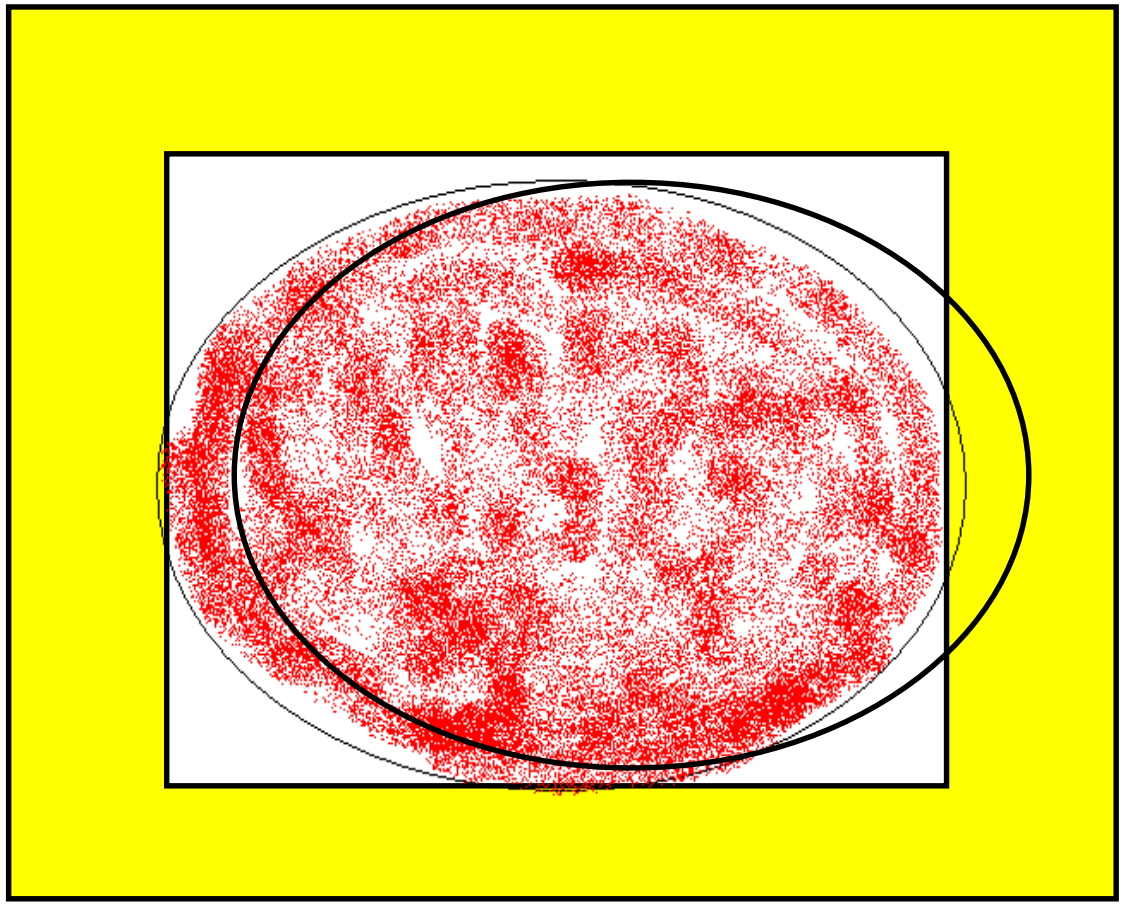
- Restrictions on the coverage model are:
  - possible responses for each command
  - unimplemented command/response combinations
  - some commands are only executed in pipe 1
- After applying restrictions, there are 1968 legal coverage tasks left.
- Make sure you identify & apply restrictions before you start!

Defining the Legal and Interesting Spaces

In Practice:

- Boundaries between legal and illegal coverage spaces are often not well understood
- The design and verification team create initial spaces based on their understanding of the design
- Coverage feedback modifies the space definition
- Interesting spaces tend to change often due to shift in focus in the verification process

Legal Spaces Are Self-correcting



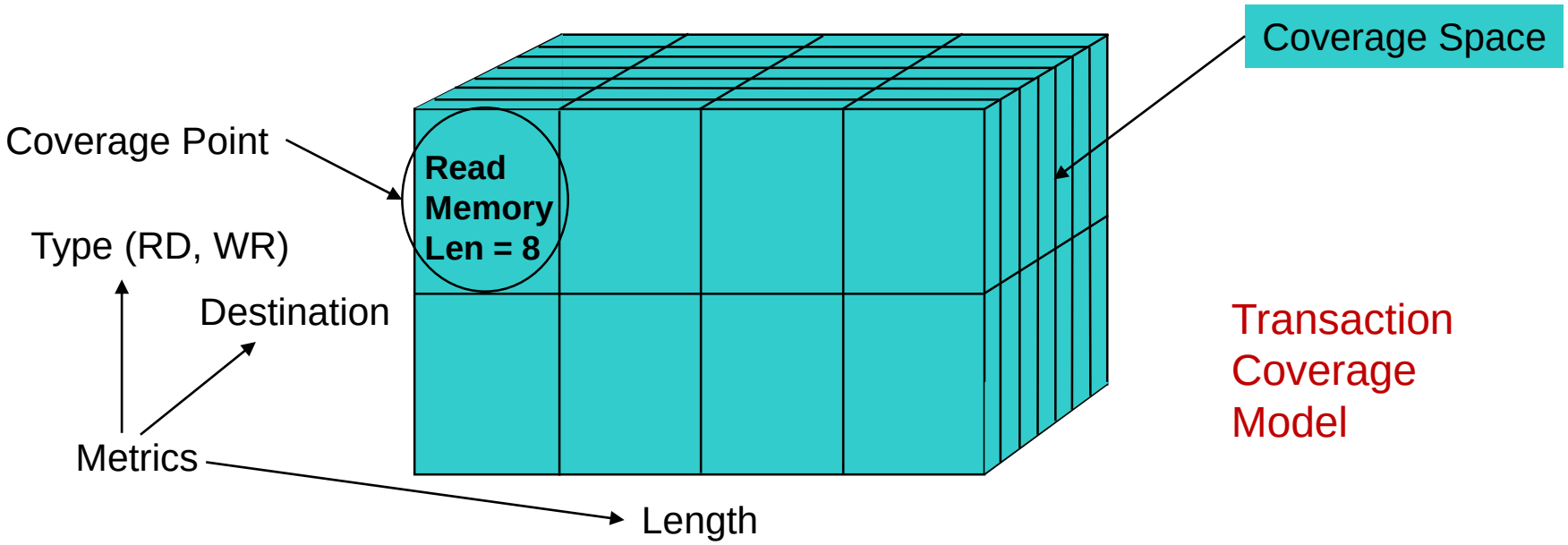
Illegal space

Legal space

Covered space

Coverage Terminology

- **coverage model** *is a set of legal and interesting coverage points in the coverage space.*
- **coverage point/task**
  - *A point within a multi-dimensional coverage space.*
  - *An event of interest that can be observed during simulation.*



“Coverage is a measure of effort, not achievement”

Summary: Functional Coverage

Determines whether the **functionality** of the DUV was verified.

- Functional coverage models are **user-defined.**
  - (specification driven)
  - This is a skill. It needs (lots of) experience!
  - Focus on **control signals.**

- **Strengths:**
  - High expressiveness: cross-correlation and multi-cycle scenarios.
  - Objective measure of progress against verification plan.
  - Can identify coverage holes by crossing existing items.
  - Results are easy to interpret.
- **Weaknesses:**
  - Only as good as the coverage metrics.
  - To implement the metrics, engineering effort is required and a lot of expertise.

Summary: Code Coverage

Determines if all the **implementation** was verified.

- Models are implicitly defined by the source code.
  - (implementation driven)
  - statement, path, expression, toggle, etc.

- **Strengths:**
  - Reveals unexercised parts of design.
  - May reveal gaps in functional verification plan.
  - No manual effort is required to implement the metrics. (Comes for free!)

- **Weaknesses:**
  - No cross correlations.
  - Can’t see multi-cycle/concurrent scenarios.
  - Manual effort required to interpret results.

Summary: Coverage Models

- Do we need both code and functional coverage? YES!

Functional Coverage	Code Coverage	Interpretation
Low	Low	There is verification work to do.
Low	High	Multi-cycle scenarios, corner cases, cross-correlations still to be covered.
High	Low	Verification plan and/or functional coverage metrics inadequate. Check for “dead” code.
High	High	High confidence in quality.

- Coverage models complement each other!
- No single coverage model is complete on its own.

Case Study

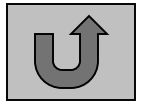
Interdependency in a PowerPC Processor

Interdependency in a PowerPC Processor

- Interdependencies between instructions in the pipeline of a processor create interesting testing scenarios
  - They activate many microarchitectural mechanisms, such as forwarding and stalling
  - Studies have shown that they are the source of many bugs in processor designs
  - Functionality at this level is often related to increasing processor performance

Lesson No. 1

- *Define coverage models in interesting areas in the design*
  - Bug prone, New logic, Complex algorithm
- In our case:
  - Register interdependency activates many pipeline mechanisms, such as forwarding and stalling
  - Coverage model aims to ensure that all forward and stall mechanisms are activated



First Approach – Black Box Model

- The motivation (story):

  Verify all dependency types of a resource (register) relating to all instructions

- The semantics of the coverage tasks:

  A coverage task is a quadruplet (I_i, I_k, R, DT), where Instruction I_i is followed by Instruction I_k, and both share Resource R with Dependency Type DT
- The attributes:
  - I_i, I_k - Instruction: add, sub, ...
  - R - Register (resource): G1, G2, ...
  - DT - Dependency Type:
    - WW, WR, RW, RR and ???

First Approach – Black Box Model

- The motivation (story):

  Verify all dependency types of a resource (register) relating to all instructions

- The semantics of the coverage tasks:

  A coverage task is a quadruplet (I_i, I_k, R, DT), where Instruction I_i is followed by Instruction I_k, and both share Resource R with Dependency Type DT

- The attributes:
  - I_i, I_k - Instruction: add, sub, ...
  - R - Register (resource): G1, G2, ...
  - DT - Dependency Type:
    - WW, WR, RW, RR and None

More Semantics

- The semantics provided so far is too coarse
  - What if I_i is the first instruction in the test and I_k is the 1000 instruction?
- Need to refine the semantics to improve probability of hitting interesting events
- Additional semantics
  - The distance between the instructions is no more than 5
  - The first instruction is at least the 6th

The Legal Space

- Not all combinations are valid
  - Not all instructions read from registers
  - Not all instructions write to registers
  - Fixed point instructions cannot share FP (floating point) registers
  - … and more

Space and Model Size

- PowerPC has
  - ~400 instructions
    - (actually this is an old number, current PowerPC has close to 1000 instructions)
  - ~100 registers
- Coverage space size is 400 x 400 x 100 x 5 = 80,000,000 tasks
- Even after all restrictions are applied, the model size is still 200,000 tasks

Lesson No. 2

- *Define a model of realistic size*
  - Ensure good coverage can be achieved with simulation resources
  - Group similar cases together to reduce model size
- In our case:
  - Original space size is
    (400 x 400 x 100 x 5) = 80,000,000 tasks
  - Many instructions behave similarly in the pipe
    - For example add and sub
  - Many registers are activated in the same way
    - All general purpose registers, all floating-point registers
  - Grouping similar instructions together helps to reduce the model size to a manageable size

Coverage Results

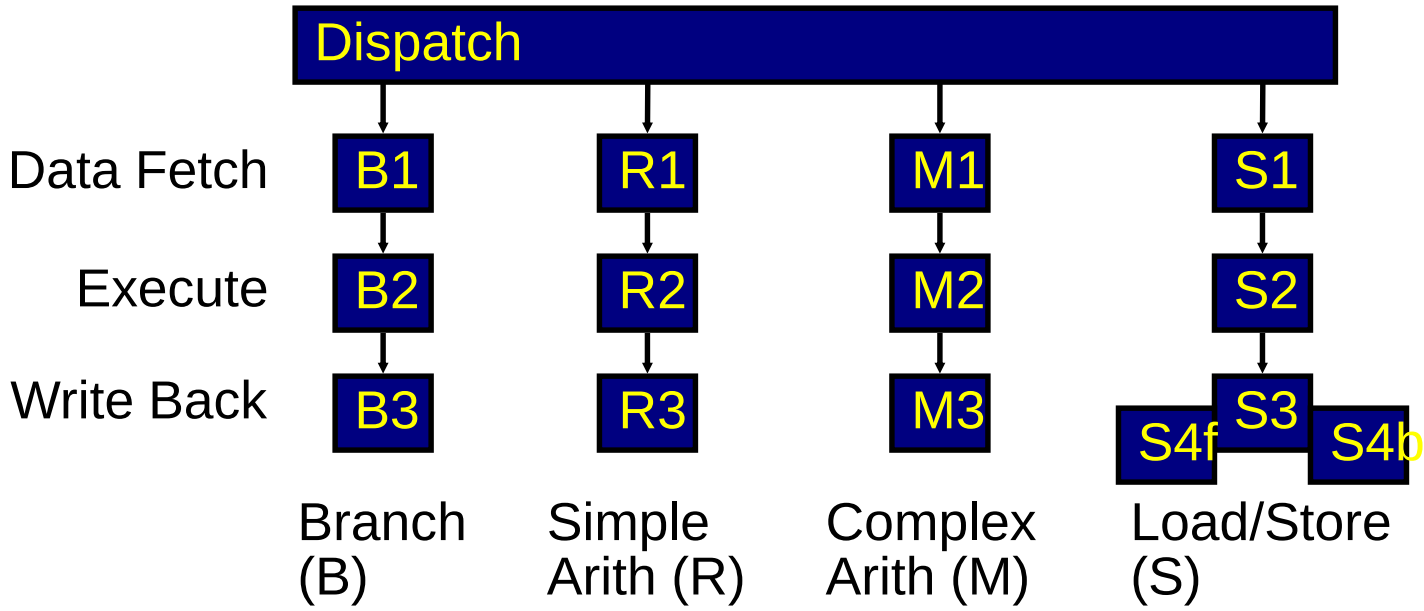
- A random test generator was used to generate tests that achieved 100% coverage architecture-level requirements coverage
- Testing the generated tests against the forwarding and stalling mechanisms of a specific processor showed that many such mechanisms were not activated by the tests

Lesson No. 3

- *Define coverage models at the proper level of abstraction for the coverage tasks*
- In our case:
  - Forwarding and stalling are **microarchitectural** mechanisms, so the coverage model should be defined at the microarchitectural level
- In general:
  - Microarchitecture is the place to look for coverage models
    - This is where the complexity of the design hides
      - Architecture is not detailed enough
      - Implementation is too messy

Grey Box Model

- Microarchitectural model for a specific Processor
  - Multithreaded
  - In-order execution
  - Up to four instructions dispatched per cycle

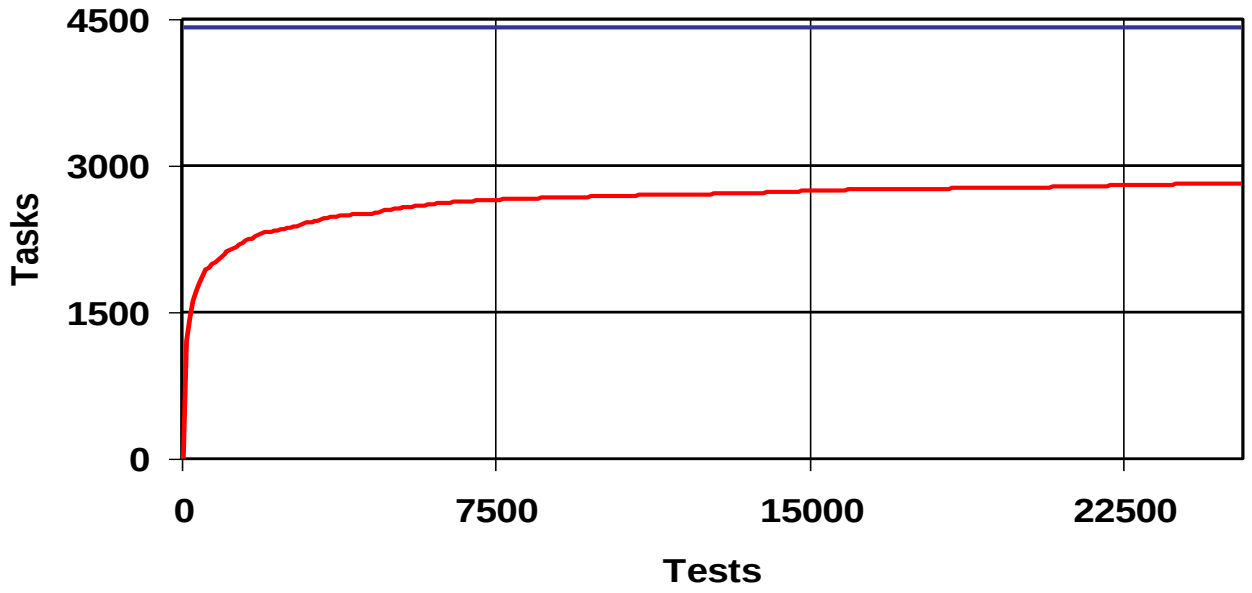


Model Details

- Model contains 7 attributes
  - Type, pipe and stage of first instruction (I1 ,P1 ,S1)
  - Same attributes for second instruction (I2, P2, S2)
  - Type of dependency between the instructions
    - RR, RW, WR, WW, None
- Grouping is done in a similar way to the architectural model
- Many restrictions exist, e.g.
  - if I1 is simple fixed point, then P1 is R (Simple Arithmetic) or M (Complex Arithmetic)

Analysis of Interdependency Model

- After 25,000 tests 2810 / 4418 tasks were covered (64%)



Lesson No. 4

- *Coverage analysis is more than a single number*
- In our case:
  - 64% is not bad but
  - Progress report shows that coverage is progressing slowly
  - Hole analysis finds big areas that are covered very lightly
  - Analysis found some problems in test generators

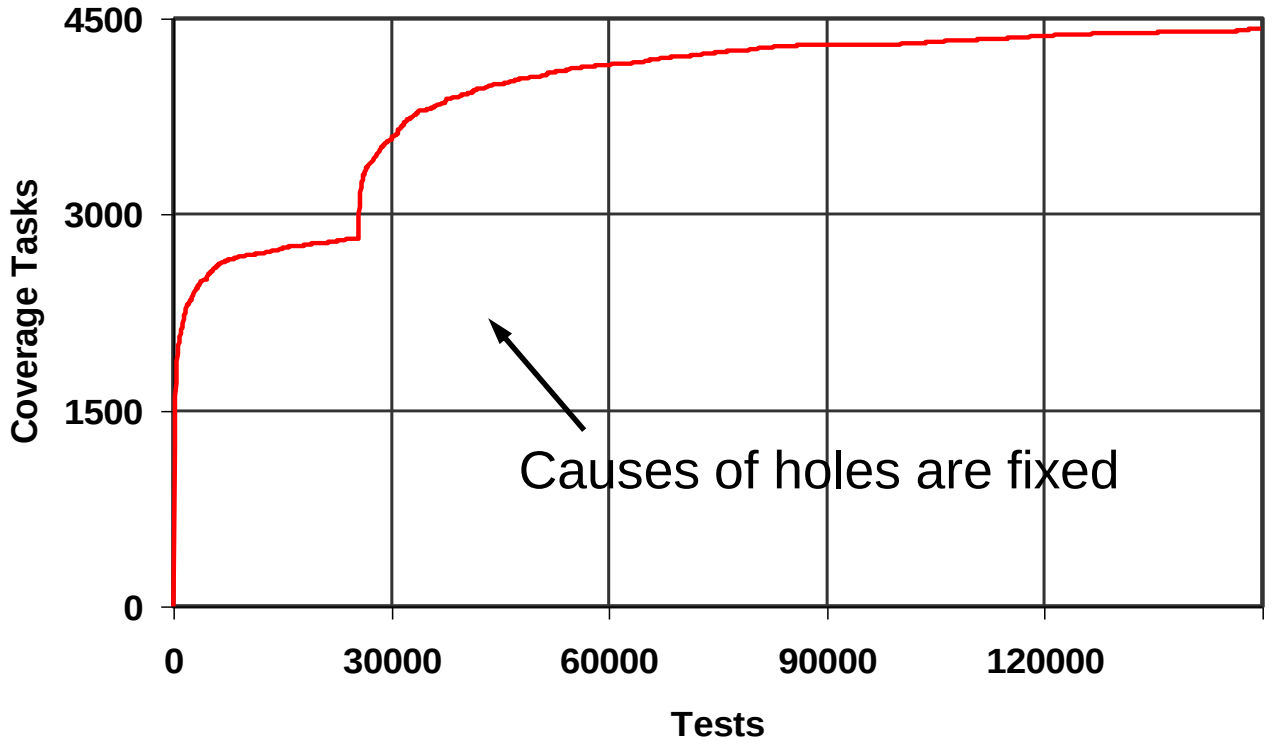
Analysis of Interdependency Model

- Coverage hole analysis detected two major areas that are lightly covered
  - Stages S4f and S4b that are specific to thread switching are almost always empty
    - Reason: not enough thread switches during tests
  - The address-base register in the store-and-update instruction is not shared with other registers in the test
    - Reason: bug in the test generator that didn't consider the register as a modified register

Lesson No. 5

- *Look for large uncovered areas*
  - Can indicate problems in the testing
  - Or missing restrictions
- Constantly update the coverage models
  - Makes coverage picture clearer
- In our case:
  - Two large holes caused by problems in the test generator and test specification

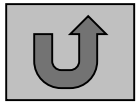
Coverage Progress



Architecture vs. Microarchitecture

- Architecture
  - No implementation details
  - Easy to share between designs
  - Temporal model

- Microarchitecture
  - Pipe implementation knowledge is needed
  - Access to microarchitectural mechanisms is needed
    - White box or at least grey box
    - More for observability than for controllability



Summary: Coverage

- Coverage is an important verification tool.
  - Code coverage: statement, path, expression
  - Functional coverage models: story, attributes, values, restrictions
- Combination of coverage models required in practice.
  - Code coverage alone does not mean anything!
- Verification Methodology should be **coverage driven**.