COMSM0115 Design Verification:

Coverage

Kerstin Eder

(Acknowledgement: Avi Ziv from the IBM Research Labs in Haifa has kindly permitted the re-use of some of his slides.)

University of BRISTOL

Department of COMPUTER SCIENCE

Last Time

- Verification Cycle
- Verification Methodology &
- Verification Plan

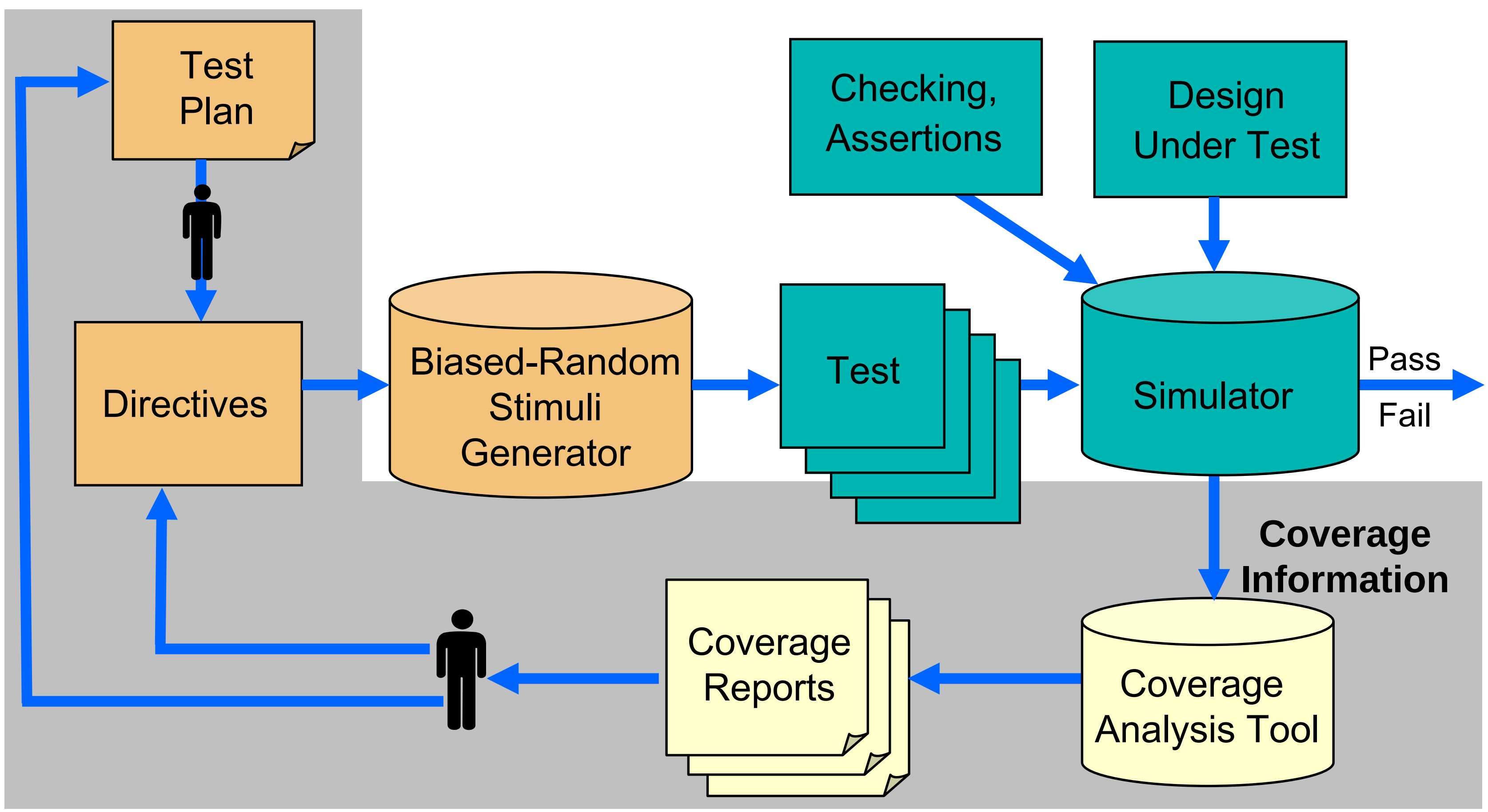
Previously: Verification Tools

Coverage is part of the Verification Tools.

Outline

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

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 written to quantify their behavioral spaces

Coverage Goals

- Measure the "quality" of a set of tests
- Supplement test specifications by pointing to untested areas
  - 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 a stopping criteria for unit testing
- Improve understanding of the design

Coverage Types

- Code coverage
- Structural 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
  - Both software and hardware

Code Coverage - Limitations

Code coverage can answer the question:

"Is there a piece of code that has not been verified?"

– Method used in software engineering for some time.

Main problem:

- **False positive answers can look identical to true positive answers.**

  False positive: A bad design is thought to be good.
- **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
- Mutation
  - Models that check directly for common bugs

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 (sub-)expression in Boolean expression 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
```

What do we need to do to get the statement coverage to 100%?

- Why has this never occurred?
- Is it a condition that can never occur? Was is simply forgotten?
- (Dead code can be “ok”!) WHY?

Path/Branch Coverage

Measures all possible ways to execute a sequence of statements.

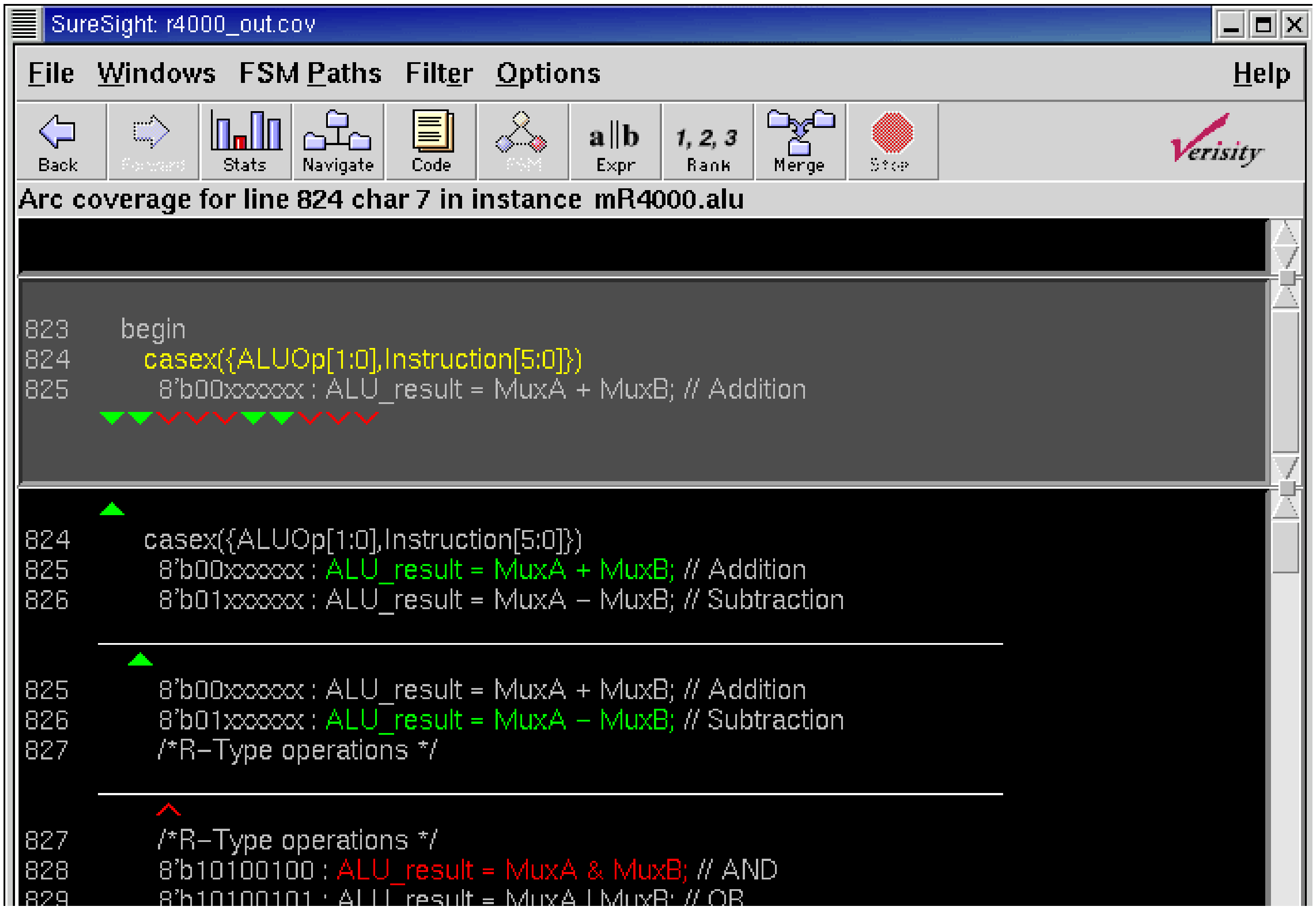
- Are all **if/case** branches taken?
- How many execution paths?

```
✓ 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
  ☑      ☐      ☑      ☑
```

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)

Branch Coverage Report



Expression/Condition Coverage

Measures the various ways paths through the code are executed.

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

```
✓ 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
```

Note: Only 50% expression coverage!

- Analysis: Understand WHY part of an expression was not executed
- **Reaching 100% expression coverage is extremely difficult.**

Data Flow Models

- Coverage models that are based on flow of data during execution
- Each coverage task has two attributes
  - **Define** – where a value is assigned to a variable (signal, register, …)
  - **Use** – where the value is being used
- Types of dataflow models
  - C-Use – Computational use
  - P-Use – Predicate use
  - All Uses – Both P and C-Uses

```
process (a, b)
begin
  s <= a + b;
end process

process (clk)
begin
  if (reset)
    a <= 0; b <= 0;
  else
    a <= in1; b <= in2;
  end if
end process
```

Mutation Coverage

- Mutation coverage is designed to detect simple (typing) mistakes in the code
  - Wrong operator
    - + instead of –
    - >= instead of >
  - Wrong variable
  - Offset in loop boundaries
- A mutation is considered covered if we found a test that can distinguish between the mutation and the original code
  - Strong mutation – the difference is visible in the primary outputs
  - Weak mutation – the difference is visible inside the DUV

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.
  - A failure in low-level code may affect multiple high-level callers.
  - Hence, set a higher level of code coverage for unit testing than for system testing.
- Generally, 90% 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%.

Literature: [J Barkley. Why Statement Coverage Is Not Enough. A practical strategy for coverage closure., TransEDA.]

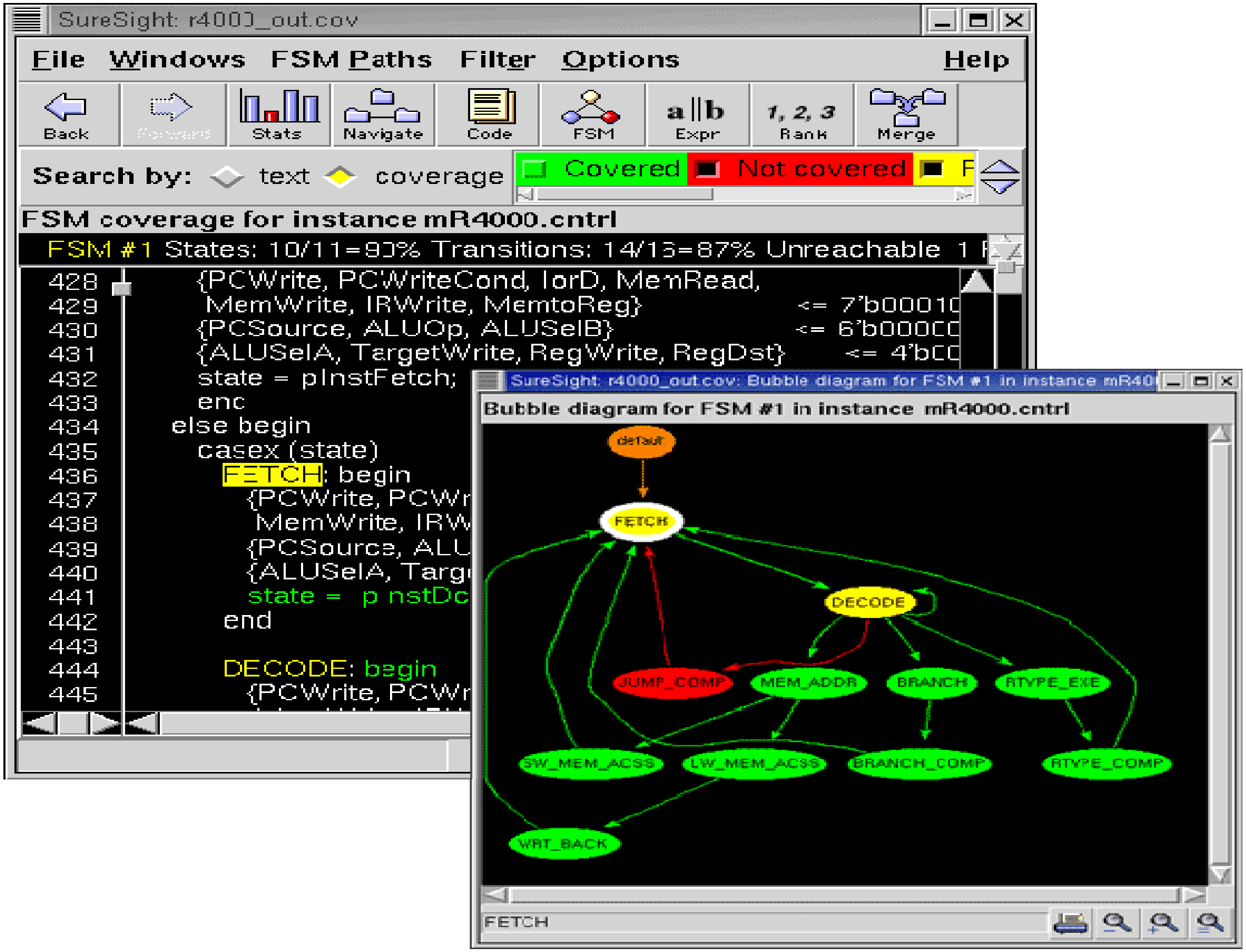
Structural Coverage

- 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
- Users may refine the models
  - Define illegal events

State-Machine Coverage

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

FSM Coverage Report



Code Coverage - Limitations

- Coverage questions not answered by code coverage tools
  - Did every instruction take every exception?
  - Did two instructions accessed 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
- Functional coverage models are specific to a given design or family of designs
- Models cover
  - The inputs and the outputs
  - Internal states
  - Scenarios
  - Parallel properties
  - Bug Models

Functional Coverage Model Types

- Discrete set of coverage tasks
  - Set of unrelated or loosely related coverage tasks
  - In many cases, natural extension of assertions into coverage
  - Often used for corner cases
- 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

[O Lachish, E Marcus, S Ur and A Ziv. Hole Analysis for Functional Coverage Data. In proceedings of the 2002 Design Automation Conference (DAC), June 10-14, 2002, New Orleans, Louisiana, USA.]

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

[G Nativ, S Mittermaier, S Ur and A Ziv. Cost Evaluation of Coverage Directed Test Generation for the IBM Mainframe. In Proceedings of the 2001 International Test Conference, pages 793-802, October 2001.]

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 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?

Example: Cross-Product Coverage Model 2

Size of coverage space:

- Coverage space is formed by **cross-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?

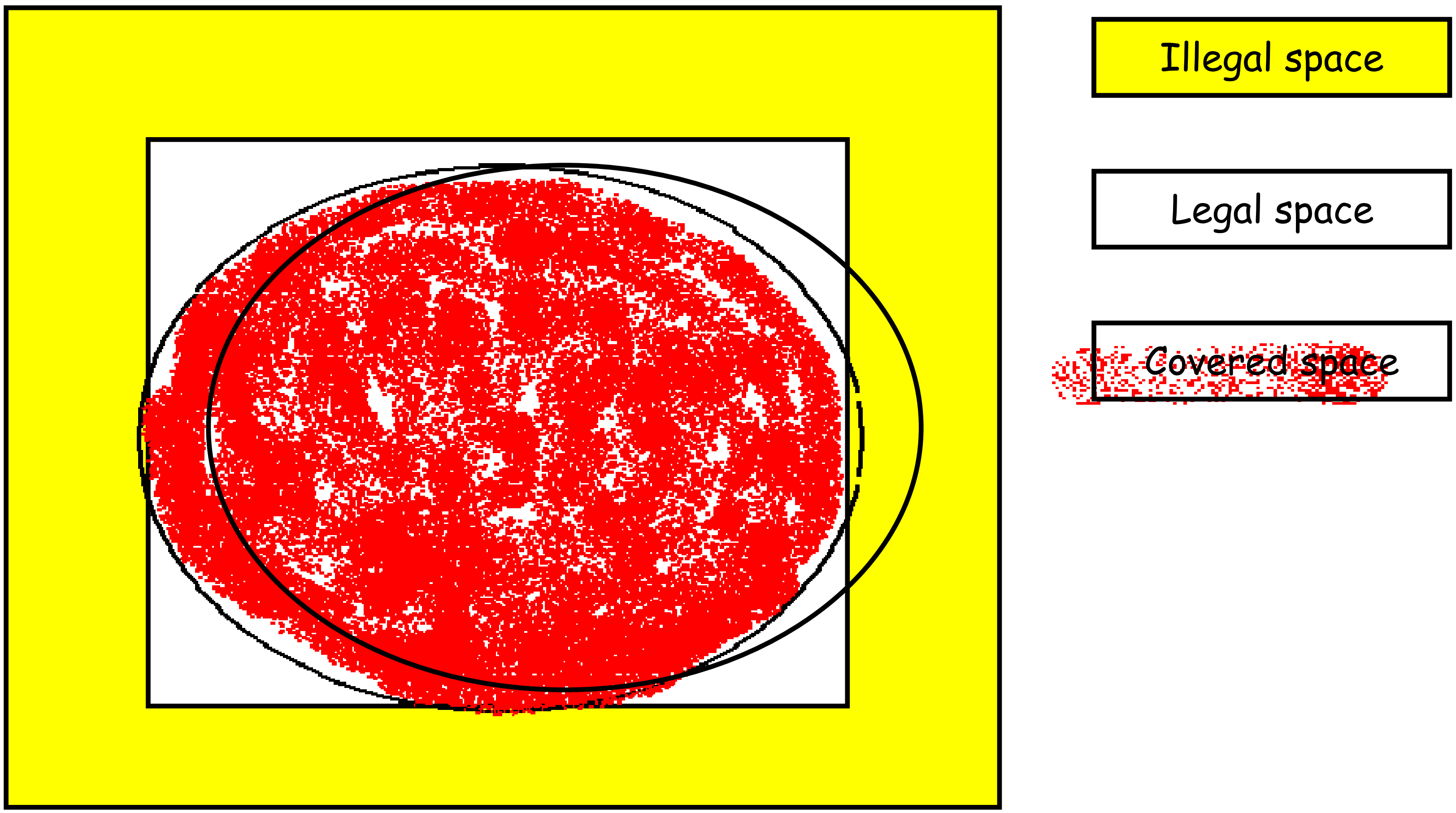
- Restrictions on the coverage model are:
  - possible responses for each command
  - unimplemented command/response combinations
  - some commands are only executed in pipe 1
- After applying restrictions, there are 1968 legal coverage tasks left.
- Make sure you identify & apply restrictions before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

Defining the Legal and Interesting Spaces

In Practice:

- The design and verification team create initial spaces based on their understanding of the design
  - Boundaries between legal and illegal are often not well understood
- Coverage feedback modifies the space definition
- Sub-models are used to economically check and refine the spaces
  - Easy to define as these are sub-crosses!
- Interesting spaces tend to change often due to shift in focus in the verification process

Legal Spaces Are Self-correcting



Cross-Product Coverage more formally

- Functional cross-product coverage models can be defined using **multi-dimensional coverage spaces.**
- A **functional coverage space** C_m is defined as the Cartesian product over m signal domains $D_0; \ldots; D_{m-1}$.
  - $C_m = D_0 X \ldots X D_{m-1}$
- Let $||D_k|| = d_k$ denote the **size of domain D_k.**
- The functional coverage space Cm contains

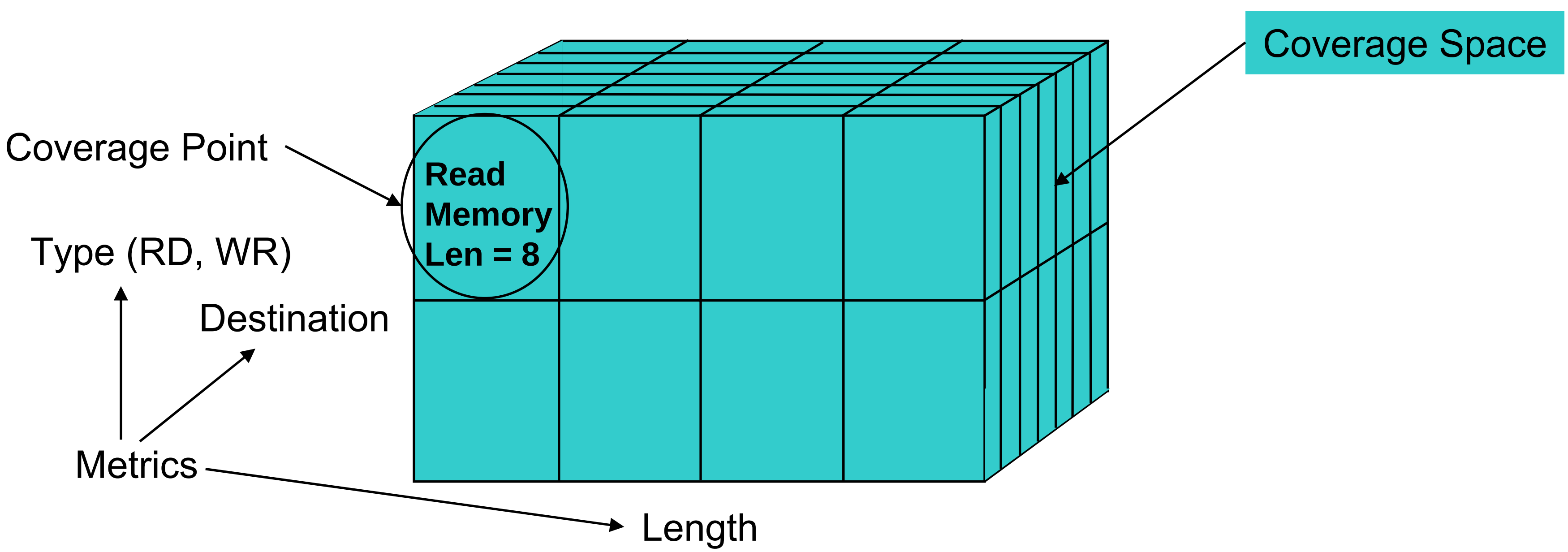
  $||C_m|| = ||D_0|| * \ldots * ||D_{m-1}|| = d$ distinct **coverage points** $p_0; \ldots; p_{d-1}$.
- A **coverage point** p_i with $i \varepsilon \{0; \ldots; d-1\}$ is characterized by an ***m*-tuple of values**

  $p_i = (v_0; \ldots; v_{m-1})$**,** where $p_i[k] = v_k$ and each $v_k \varepsilon D_k$, for $k \varepsilon \{0; \ldots; m-1\}$.

Formalization facilitates automation and identification of coverage gaps (coverage analysis).

Coverage Terminology

- **cov·er·age model** *n. 1. A set of legal and interesting coverage points in the coverage space.*
- **cov·er·age point** *n. 1. A point within a multi-dimensional coverage space. 2. An event of interest that can be observed during simulation.*



Cross-Product Models In *e*

Verification Languages such as *e* support cross-product coverage models:

- The story is hidden in the event
- The attributes and their values are defined in the coverage items
- Legal and interesting space are defined using the illegal and ignore constructs
  - Restrictions can be defined on the coverage items and the cross itself

```
struct instruction {
   opcode: [NOP, ADD, SUB,
            AND, XOR];
   operand1 : byte;

   event stimulus;

   cover stimulus is {
      item opcode;
      item operand1;
      cross opcode, operand1
         using ignore = (opcode == NOP);
   };
};
```

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.** WHY?

- **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, manual effort is required.

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 of quality.

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

Case Study

The Coverage Process in Practice

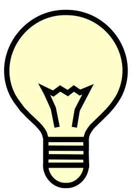
Examples:

- Verifying interdependency in a PowerPC processor
- Pipeline of Branch unit in S/390 system

(Thanks to Avi Ziv from IBM Research Labs in Haifa for sharing these.)

Example 1: 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 and work properly

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 follows 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 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

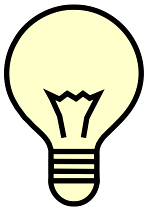
Lesson No. 2

- *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
- 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

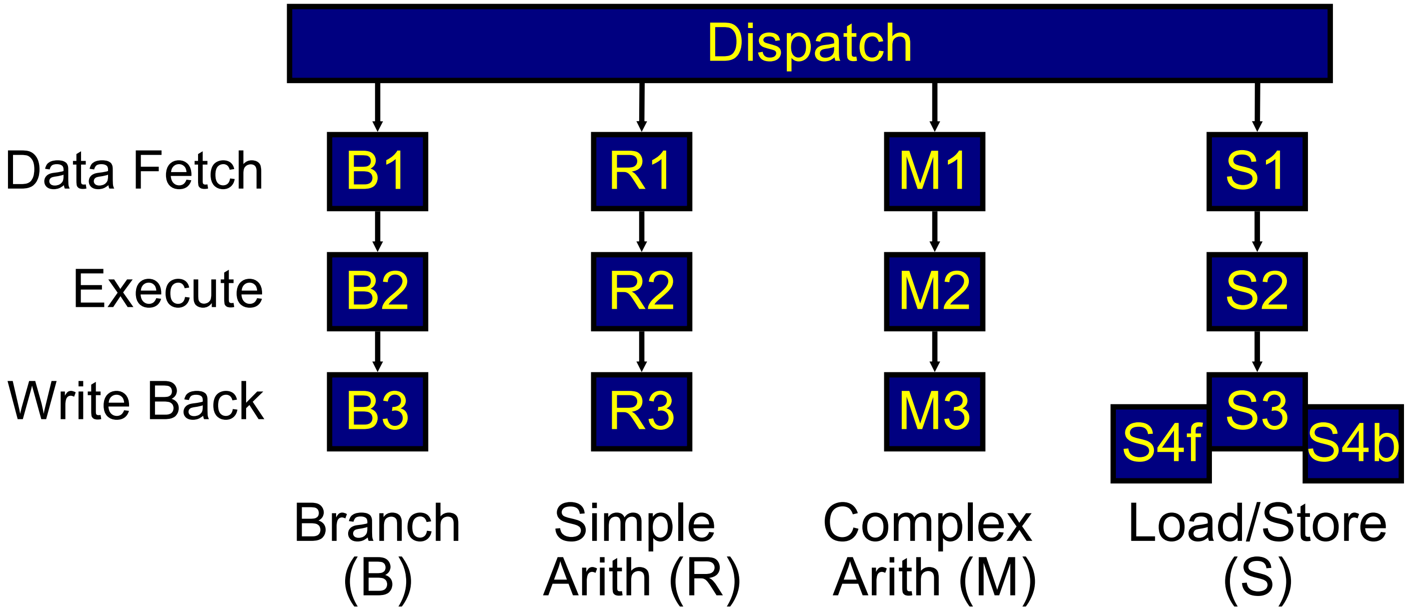


Lesson No. 3

- *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
    - 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

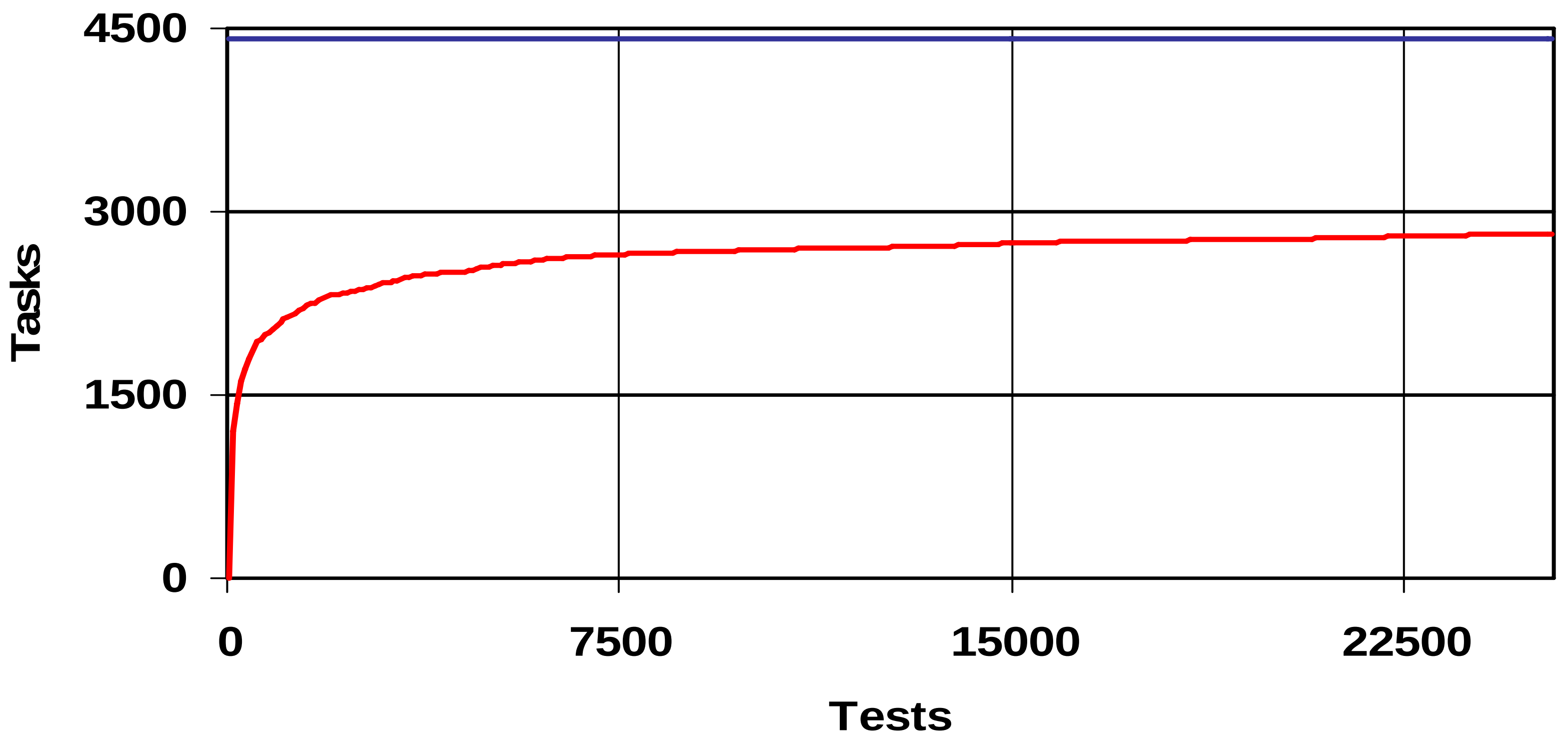
- Model contains 7 attributes
  - Type, pipe and stage of first instruction (I1 ,P1 ,S1)
  - Same attributes for second instruction (I2, P2, S2)
  - Type of dependency between the instructions
    - RR, RW, WR, WW, None
- Grouping is done in a similar way to the architectural model
- Many restrictions exist
  - I1 is simple fixed point → P1 is R or M
  - P1 is not S → S1 is 1, 2, or 3
- After restrictions, 4418 tasks are legal

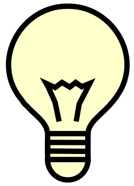
Coverage Measurement

- Make sure that you measure what you really want and what really happens
- Use simpler environment and models to test and debug the measurement system
  - Hierarchy of models
    - All instructions
    - All pipe stages
  - Controlled simulation

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

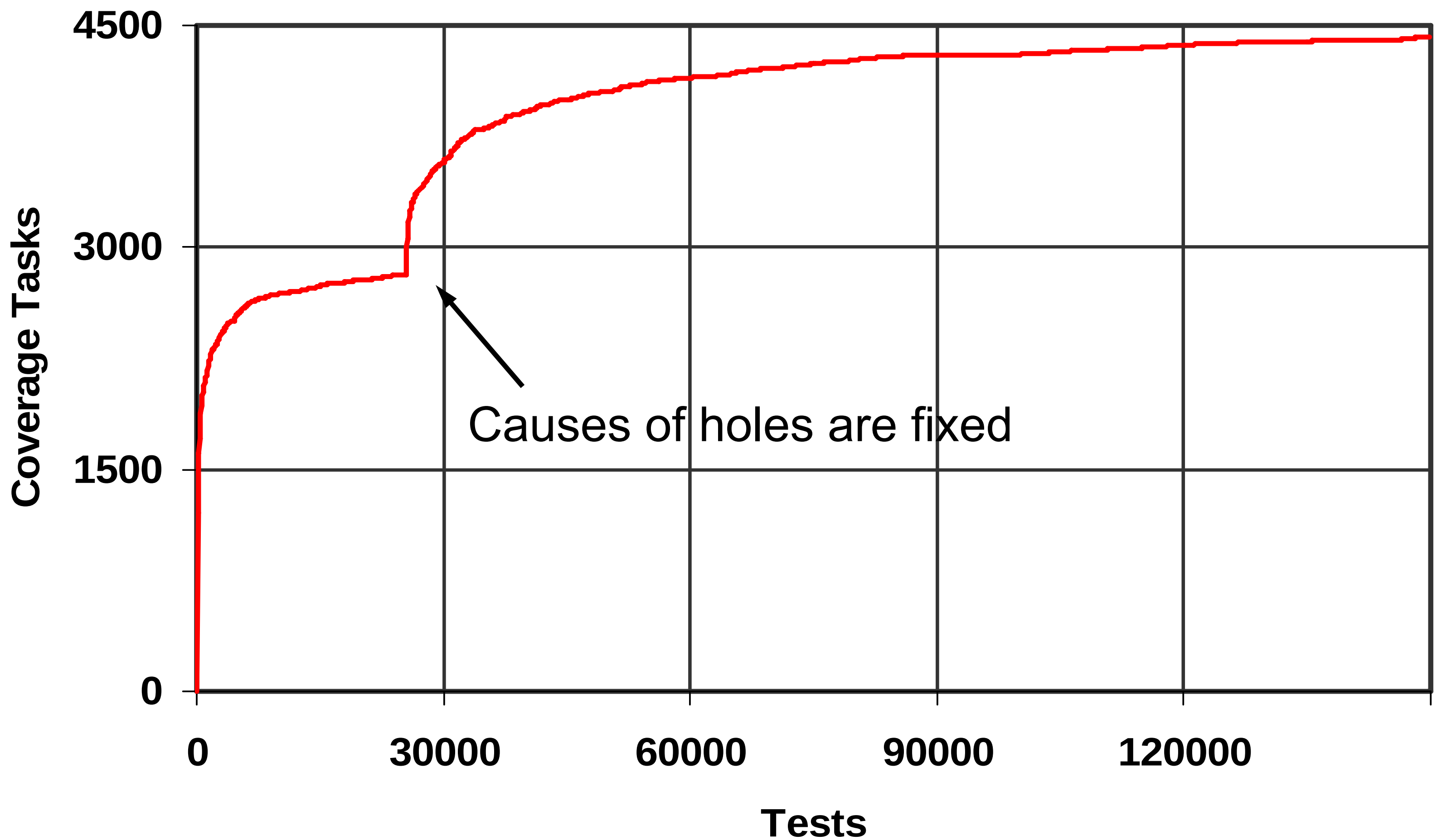
Analysis of Interdependency Model

- 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 (Why?)
  - Snapshot model

Example 2: S/390 Branch Unit

- Unit handles branch prediction and execution of branch instructions
- Contains
  - Nine stage complex pipe
    - More than one instruction at the same time in some stages
    - Instructions can enter the pipe at two places
  - Branch history tables
  - and more
- 2 PY spent on verification
- Done by experts with experience with similar designs
- About 100,000 tests per day

Coverage Models for Branch Unit

- Several models defined
  - Access to branch tables
  - Flow of a branch in the pipe
  - State of the pipe
- State of the pipe model
  - Attributes contain
    - Location and type of each branch in the pipe in a given cycle
    - Reset signal
  - Model size:
    - Without restrictions ~ 15,000,000
    - With restrictions ~ 1400

Lesson No. 6

- *Define families of coverage models that represent different views of the design*
  - Help capture all the functionality with a small number of coverage tasks
  - Analysis of one model can help understanding behavior of another
- In our case:
  - Two views of pipe functionality
  - Model for the flow of a single instruction in the pipe
  - Model for all instructions in the pipe at a given time

Lesson No. 7

- *Look for models that have a view different from the view of the designer*
  - Model definition can lead to better understanding of the design
  - Coverage can lead to unexpected scenarios
- In our case:
  - Designer's view is the flow of instructions in the pipe
  - Model for global pipe state led to accurate analysis of number of instructions in the pipe

Coverage Closure

Risks of Using Coverage

- Low coverage goals
- Some coverage models are ill-suited to deal with common problems
  - Missing code
- Generating simple tests to cover specific uncovered tasks
  - There is merit in generating tests outside the coverage!
- Using coverage without commitment to using the results

Coverage Closure

Coverage closure is the process of:

- Finding areas of coverage not exercised by a set of tests.
  - Coverage Holes!
- Creating additional tests to increase coverage by targeting these holes.

Controllability Problems

If the cases to be hit contain internal states/signals of the DUV, directed tests that exercise all combinations are hard to find.

- Processor pipeline verification: Control logic, Internal FSMs

- Generate biased random tests automatically. [RTPG]
    - **ISG**
  - Typically tests are filtered to retain only those that add to coverage.
  - Coverage analysis indicates **hard-to-reach** cases.
  - Don’t waste engineers time on what **automation** can achieve.
- Combine automatically generated stimulus with coverage.
- Gives rise to **Coverage DRIVEN Verification Methodology**

BUT:

- **Hard-to-reach cases (may) need manual attention.**
  - Bias tests towards certain conditions or corner cases.
  - Supplying bias requires significant engineering skill.
    - Often only trial-and-error approach.

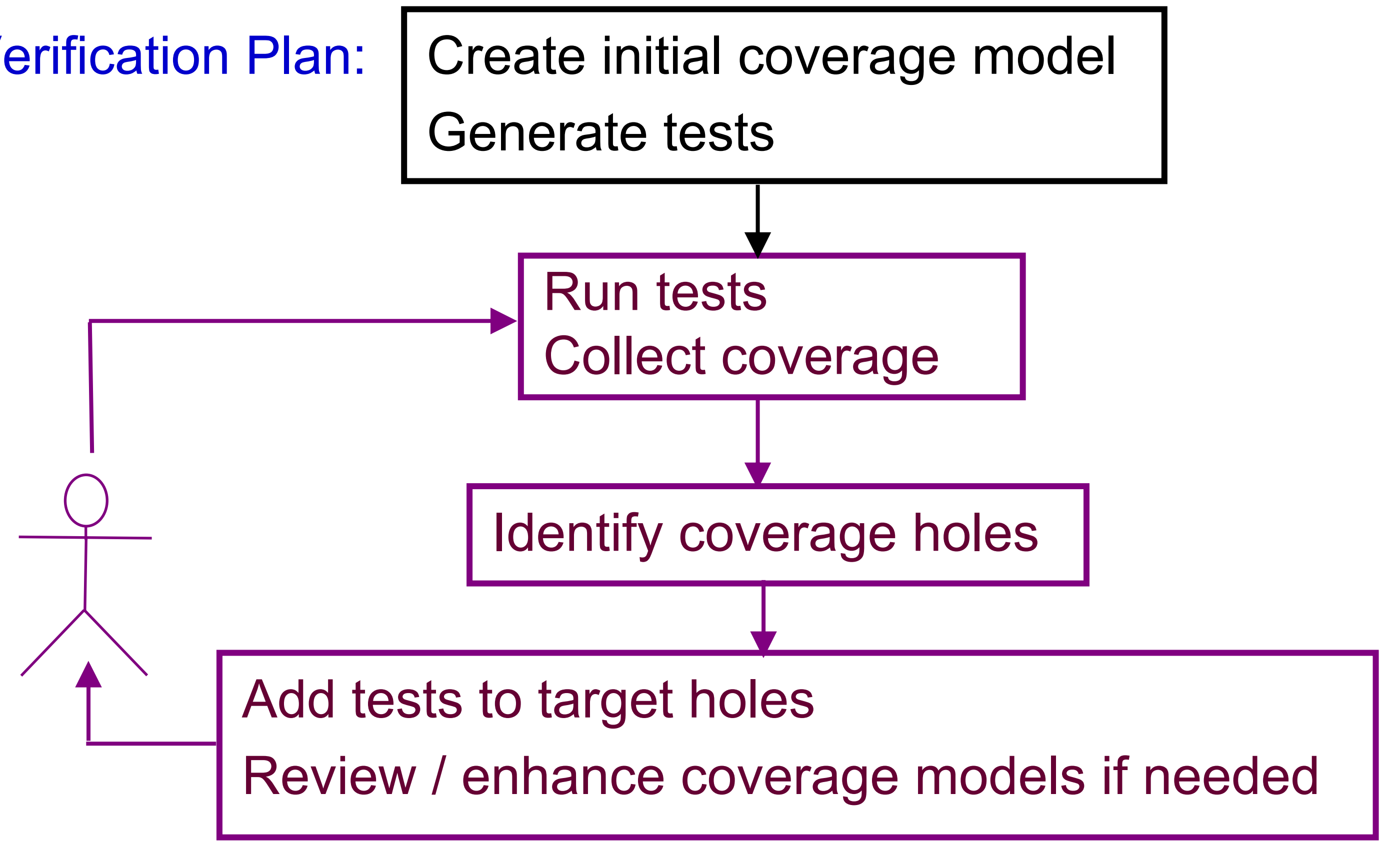
Biasable Random Test Generation

[A Adir, E Almog, L Fournier, E Marcus, M Rimon, M Vinov and A Ziv. Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification. In IEEE Design & Test of Computers, 2004.]

- **Test template description language**
  - expressiveness of programming language
  - virtually unlimited control over events to be generated
- **Architectural modelling framework**
  - building blocks at high enough level to describe processors
    - VLIW architectures
- **Generation engine based on constraint satisfaction**
  - generic constraint solver customized for pseudo-random test generation

Coverage DRIVEN Verification **Methodology**

From Verification Plan:



Current research: How can we automate this further?

80/20 Split

In practice: 80/20 (20/80) split wrt coverage progress.

Good news:)

- 80% of coverage is achieved (relatively quickly/easily) driving randomly generated tests.
- This takes about 20% of total time/effort/sim runs spent on verification.

Bad news:(

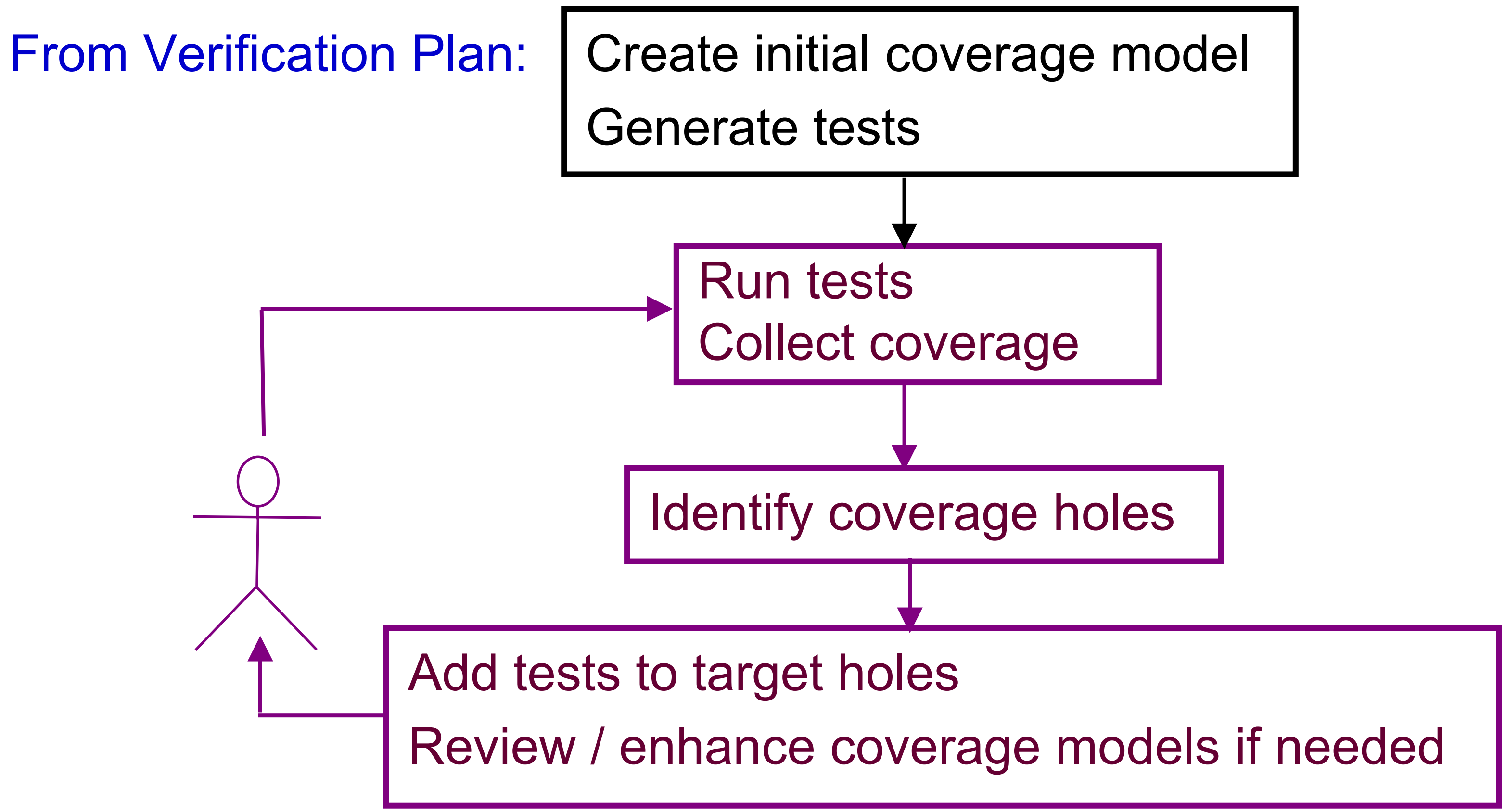
- Gaining the remaining 20% coverage,
  - i.e. filling the remaining coverage holes (which often needs to be done manually and requires a lot of skill plus design understanding),
- can take as much as 80% of the total time/effort/sim runs spent on verification.

Benefits of Coverage DRIVEN Verification **Methodology**

Benefits:

- Shortens implementation time
  - (Initial setup time)
  - Random generation covers many “easy” cases
- Improves quality
  - Focus on goals in verification plan
  - Encourages exploration/refinement of coverage models
- Accelerates verification closure
  - Refine/tighten constraints to target coverage holes

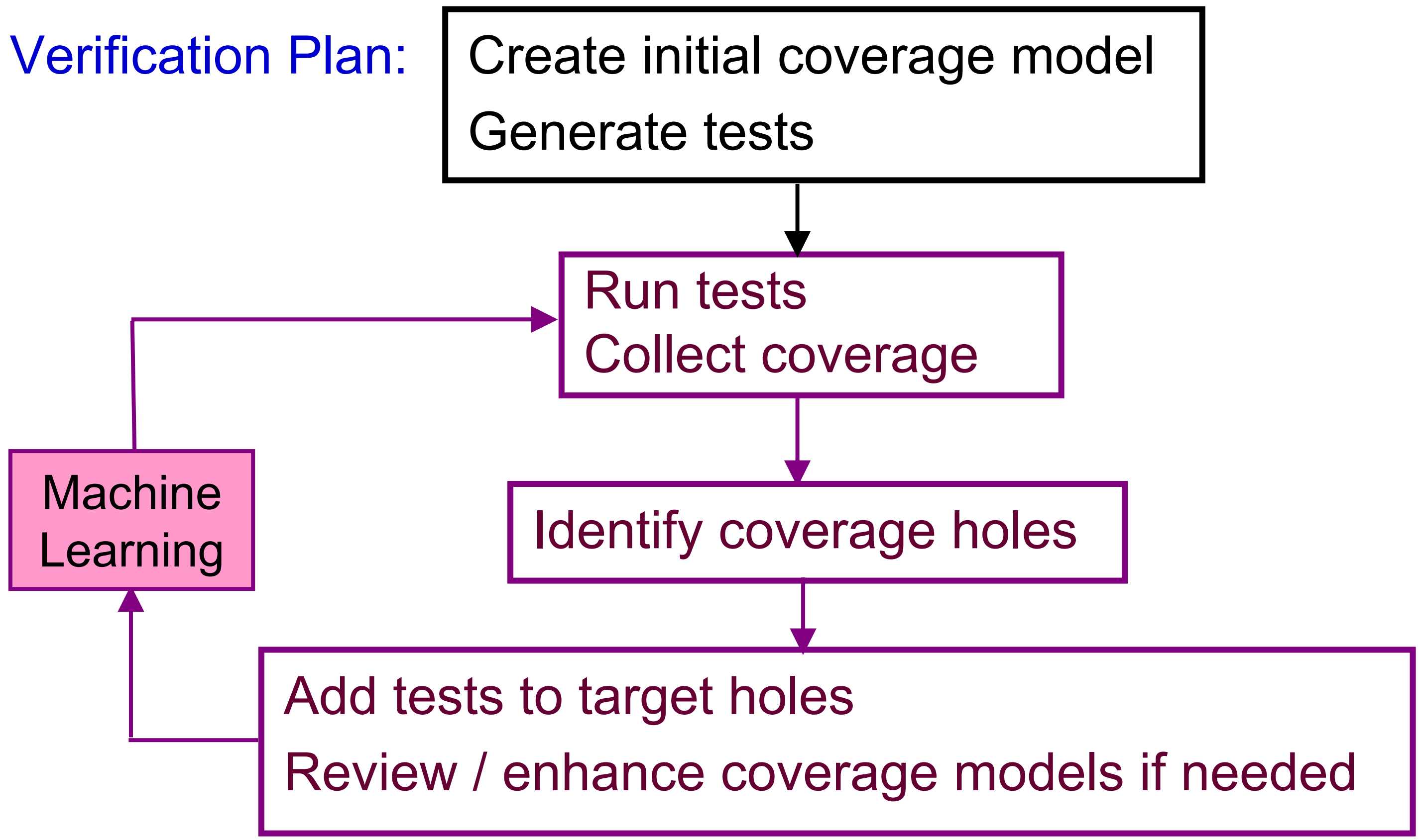
Coverage DRIVEN Verification **Methodology**



Current research: How can we automate this further?

Coverage DIRECTED Test Generation

From Verification Plan:



Current research: How can we automate this further?

CDG: Coverage DIRECTED Test Generation

How can we make better use of coverage data to **automate** stimulus generation?

Latest Research:

Coverage DIRECTED (stimulus/test) generation [IBM]

- BY CONSTRUCTION
  - Require description of design as FSM.
  - Use formal methods to derive transition coverage.
  - Automatically translate paths through FSM to test vectors.
  - Fall over in practice: FSMs are prohibitively large!
- BY FEEDBACK
  - (Exploit Machine Learning techniques)
  - GAs/GP - Need to find suitable encoding (e.g. of instructions).
  - Bayesian Networks - Need to design and train BN.
  - Data Mining in coverage spaces

No significant breakthrough in CDG yet!

Summary: Coverage

- Coverage is an important verification tool.
  - Code coverage: statement, path, expression
  - Structural coverage: FSM
  - Functional coverage models: story, attributes, values, restrictions
  - (Assertion coverage will be introduced during the lecture on Assertion-based Verification.)
- Combination of coverage models required in practice.
  - Code coverage alone does not mean anything!
- Verification Methodology should be **coverage driven**.
- **Automation:** Research into coverage directed test generation
- **Delays in coverage closure** are the main reason why verification projects fall behind schedule!