COMS31700 Design Verification:

Verification Cycle, Verification Methodology & Verification Plan

Kerstin Eder

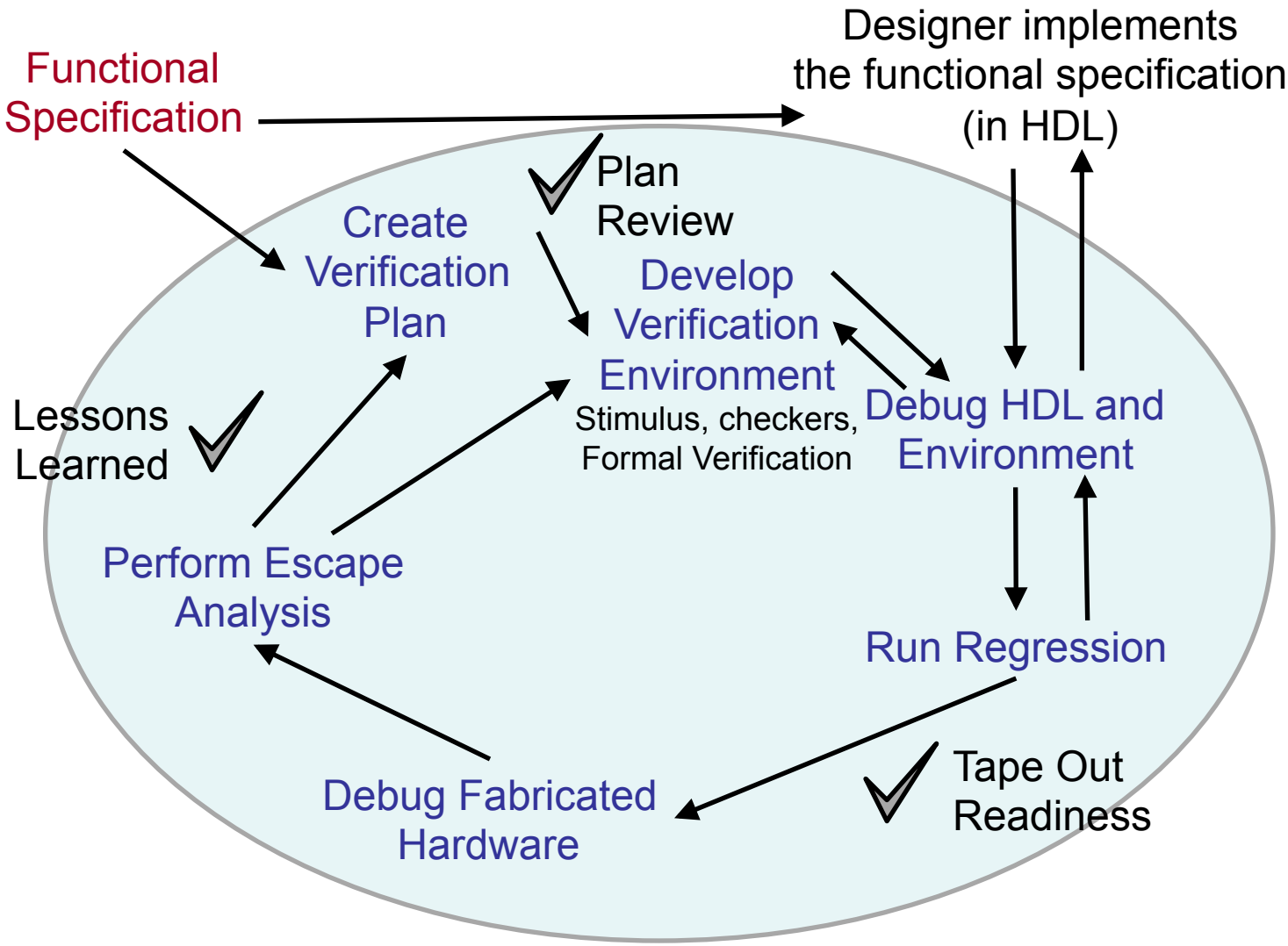
(Acknowledgement: Avi Ziv from the IBM Research Labs in Haifa has kindly permitted the re-use of some of his slides.)

University of BRISTOL

Department of COMPUTER SCIENCE

The Verification Cycle

The Verification Cycle



Functional Specifications

- The **functional specification** describes the desired product
- It contains the specification of:
  - The **function** that it must perform.
  - The **interfaces** with which it communicates.
  - The **conditions** that affect the design.
- Designers implement the specification in HDL
- Verification engineers incorporate the functional specification into the **verification plan** and environment.
  - This may seem redundant, but it is the **foundation** of verification, i.e. **the specification for the verification.**

Create Verification Plan

- **Functions to be verified:** list the functions that will be verified at this level of verification.
  - Functions not covered: any functions that must be verified at a different level of the hierarchy.
- **Resources required** (people) and **schedule details**: tie the plan to the program management by estimating the cost of verification.
- **Required tools:** list the software and hardware necessary to support the described environment.
- **Specific tests and methods:** define the type of environment that the verification engineers will create.
- **Completion criteria:** Define the measurements that indicate that verification is complete.

Develop Verification Environment

The verification environment is the set of software code and tools that enable the verification engineer to identify flaws in the design.

- The software code tends to be *specific to the design*,
- while the tools are more generic and are used across multiple verification projects

- Major components in the verification environment are **stimulus and checking** for simulation based environments, and **rules generation (properties)** for formal verification environments
- The environment is continually refined throughout the verification cycle
  - Refinements include fixes and additions to the software code

Debug HDL and Environment

- Run tests according to the verification plan and look for anomalies
- Examine the anomalies to reveal the failure source
  - Can be either in the verification environment or in the HDL design
- Fix the cause of the failure
  - Either the verification environment or the HDL design
- Once the problem is fixed, rerun the exact same test(s)
  - Aim to ensure that the update corrects the original anomaly and does not introduce new ones
- Update the verification plan based on lessons learnt

Run Regression

- Regression is the **continuous running of the tests** defined in the verification plan
- Often, verification teams leverage large workstation pools, or “farms”, to run an ever-increasing number of verification jobs
- Regression is used to uncover hard-to-find bugs and ensure that the quality of the design keeps improving
- With chip fabrication on the horizon, the verification team must reflect on the environment to ensure that
  - they have **applied all valid scenarios** to the design
  - and **performed all pertinent checks**

  This is the tape-out readiness checkpoint.

Debug Fabricated Hardware

- The design team releases the hardware to the fabrication facility when they meet all fabrication criteria
  - This process is also known as the tape-out.
- The design team receives the hardware once the chip fabrication completes
- The hardware is then mounted on test platforms or into the planned systems for these chips
- The hardware debug team performs the "**hardware bring-up**"
  - During hardware bring-up, further anomalies may present themselves.

Perform Escape Analysis

- Analysis of bugs that were found later than when they should have
- The goal is to fully understand the bug, as well as the reasons that it went undiscovered by the verification environments
- Important goal: **Reproduce the bug in a simulation environment, if possible.**
  - The lack of reproduction in the verification environment indicates that the design team may not understand the bug
  - It would then follow that the team cannot assert that the bug fix is correct without reproducing the original bug in verification.

Common Verification Breakdowns

- Verification based on the design itself instead of the specification
- Underdeveloped verification plans
- Underdeveloped specifications
- Lack of resources
- Tape-out based on schedule instead of pre-defined measures

This also applies to your A1 assignment.

Summary

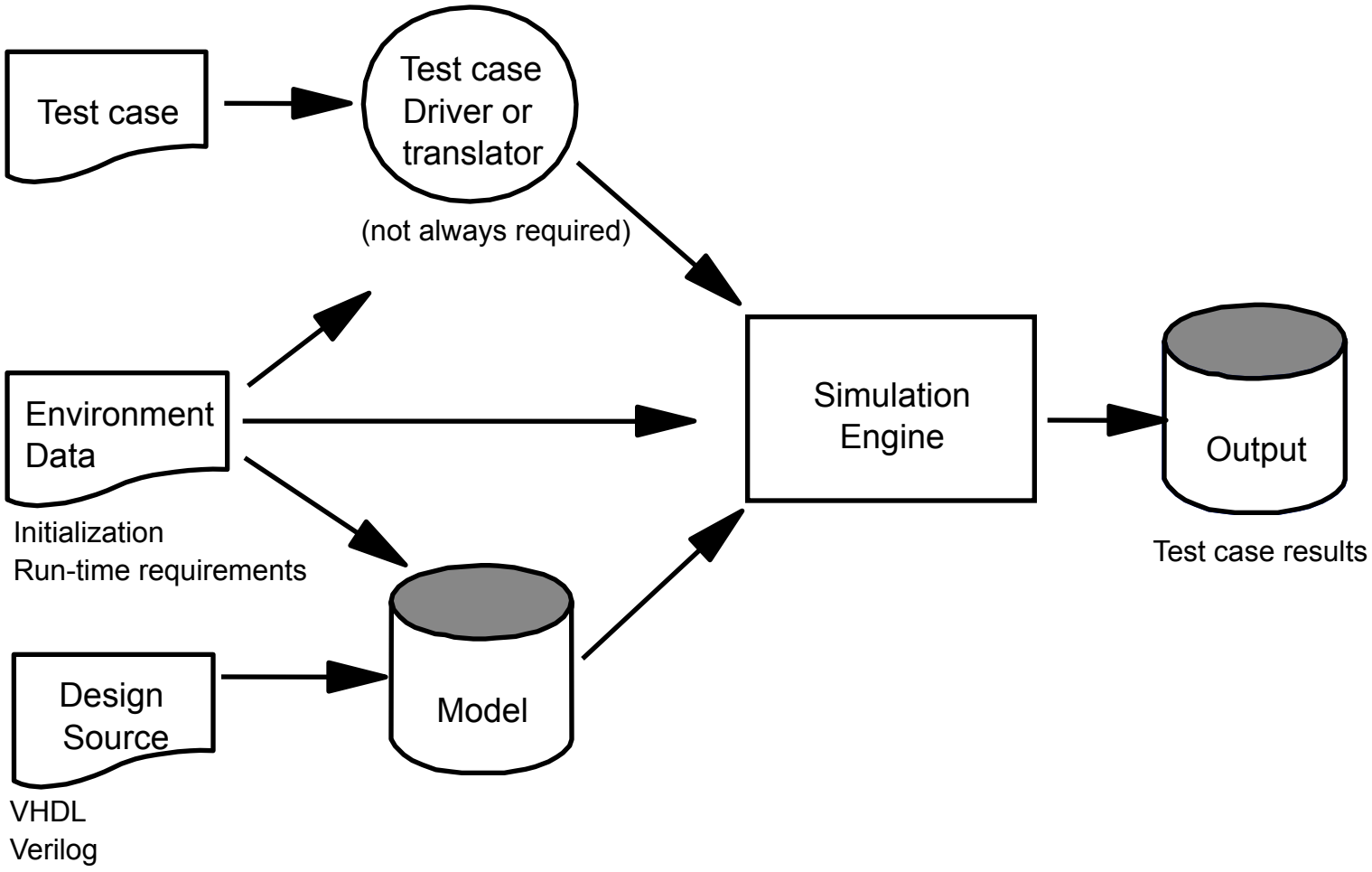
- Functional verification is a necessary step in the development of today's complex digital designs
- Verification engineers must understand the specification and internal microarchitecture of the design under verification
  - They couple this knowledge with programming skills, RTL comprehension, and **a detective's ability** to find the scenarios that uncover bugs.
- The two main challenges in the verification process:
  - Creation of a comprehensive set of stimulus
  - Identification of incorrect behavior when encountered
- **The foundation for a successful verification is the well-defined verification cycle**
  - **The process includes creation of test plans, writing and running verification tests, debugging, and analysis of the holes in the verification environments**

Verification Methodology

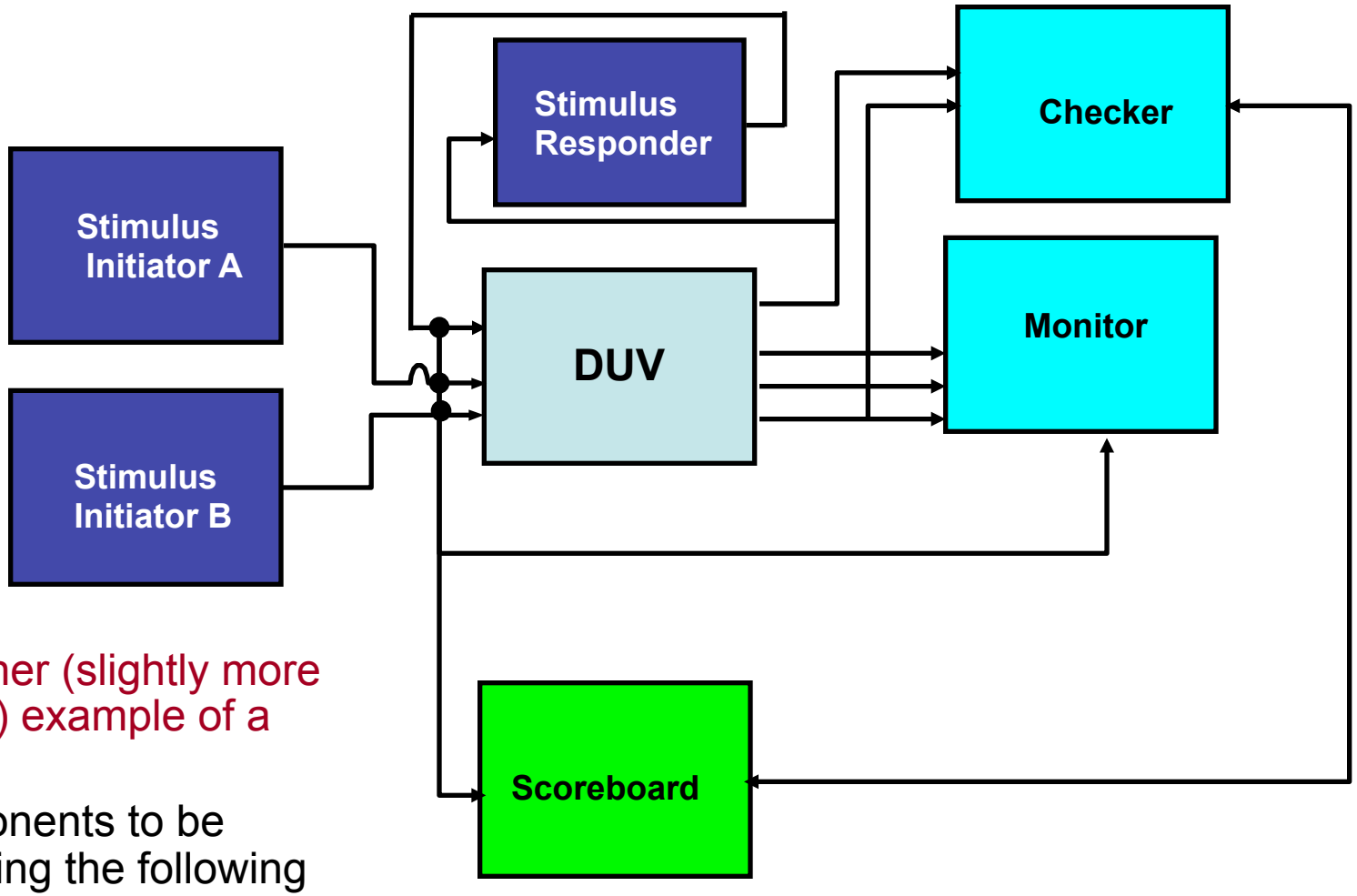
Outline

- Verification methodology evolution
- Basic verification environment
- Evolution of the **Verification plan**
- Contents of the **Verification plan**
  - Functions to be verified
  - Specific tests
  - Coverage goals
  - Test case scenarios (Tests list)
- (Calc1 Example)

Simulation-based Verification Environment Flow



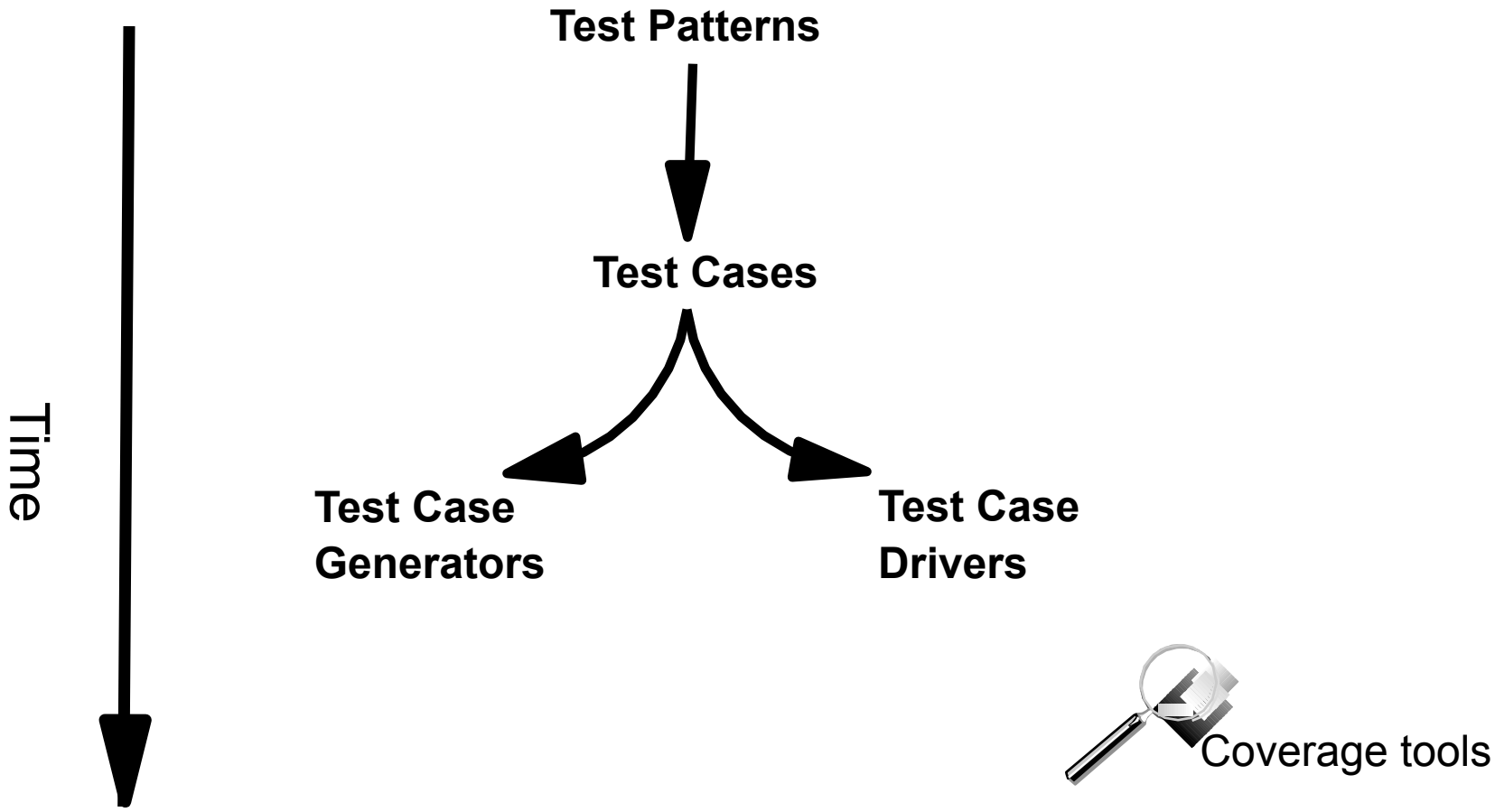
Simulation-based Verification Environment Structure



This is another (slightly more sopisticated) example of a testbench.

(New components to be covered during the following lectures.)

Verification Methodology Evolution



(Remember that there is also Formal Verification, but this is not covered here.)

Test Patterns

- Patterns that are created to test specific behaviors
- Each pattern handles a single scenario
- Tests patterns are hand generated
- DUV behavior is manually checked
  - For example, by viewing wave forms
- Expensive to create
- Expensive to maintain
- Expensive to execute

Test Cases

- Mostly change the checking of the test
  - Stimulus still hand generated and for a single scenario
- Checking evolves to
  - Self checking tests
    - The test knows at which signals to look and which values should be there
  - Automatic checking
    - The checking is independent of the stimulus
- Automatic checking opens the door for random stimuli (generation)
- Around this stage the **verification profession** was born

Test Case Generators

- Replace hand-crafted specific test patterns with machine generated random patterns
  - Single scenario → multiple scenarios
  - Specific target → more generic targets
  - Small number of tests → large number of tests
- Test case generators are tools that are external to the verification environment
  - Offline generation
  - For the environment, tests are hardcoded

Test Case Drivers

- The stimuli generation is embedded in the verification environment
- Stimuli are generated during the operation of the environment (and simulation)
- The driver can react to the state of the DUV
  - Can improve the quality of the stimuli and stress per cycle

Coverage

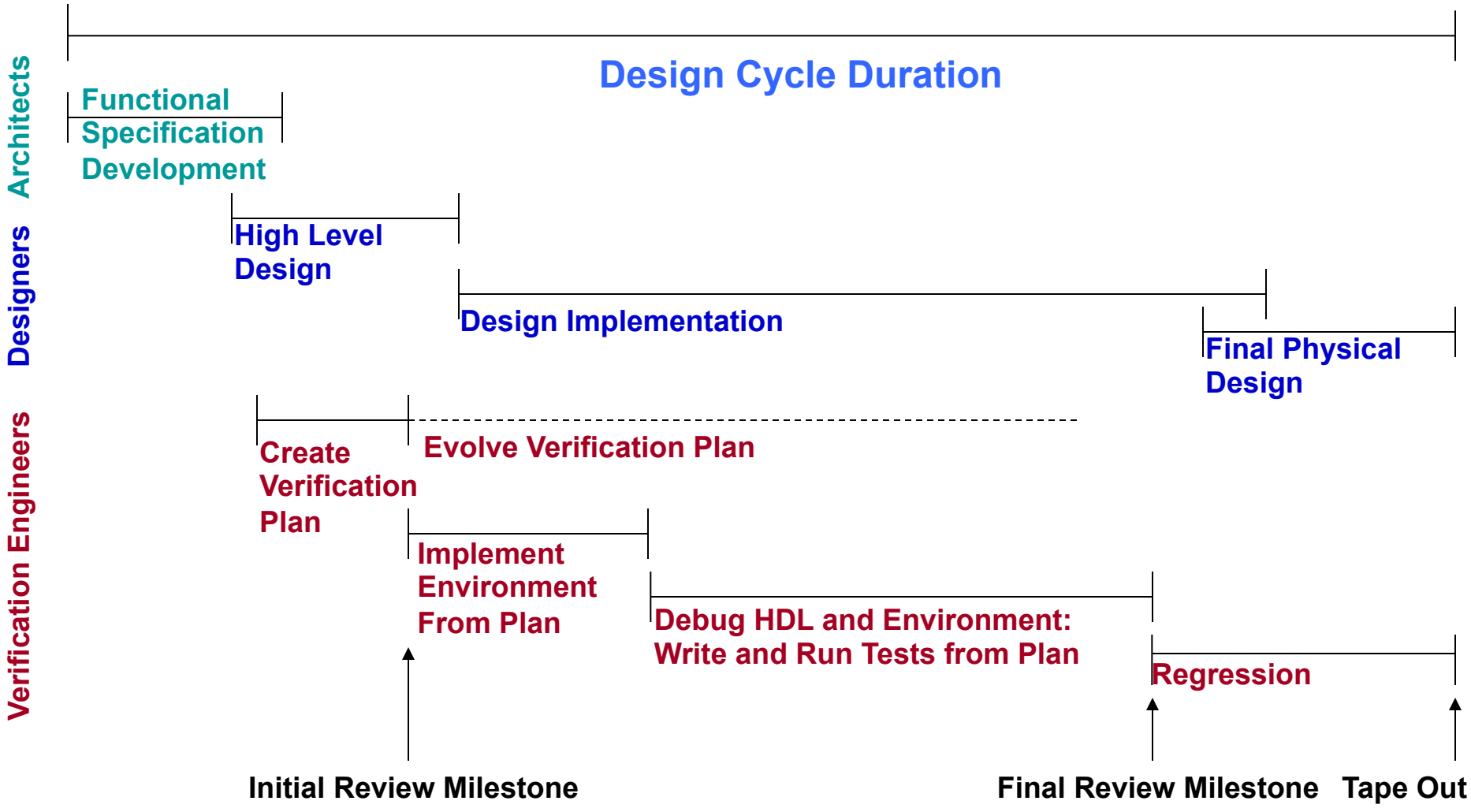
- The move from target-specific test cases to random stimuli generation reduced the ability of the verification team to **ensure that all interesting cases are verified**
- Coverage measurement and analysis are the "**automatic replacement**" for this
  - Replaces one-to-one matching with many-to-many
    - Many tests can potentially hit many interesting cases
- Coverage measures whether test cases hit the scenarios they are supposed to hit
  - And highlights untested areas
- **Coverage measures the effectiveness of the verification**

Verification Plan

Evolution of the Verification Plan

- The source of the verification plan is the Functional Spec document
  - Must understand the DUV before determining how to verify it
  - Confront unclear and ambiguous definitions
  - *Incomplete and changing continuously*
- Other factors may affect its content

Design and Verification Process Interlock



Calc1 Verification Plan

- The design description from A1 details the intent of the Calc1 design.
  - It is the verification engineer's job to demonstrate that the actual design implementation matches the intent.
- Even for a relatively simple design like Calc1, it is still best not to jump into test case writing before thinking through the entire verification plan requirements.
- Please note:
  - For the feedback session on A1, please bring your verification plan with you.
  - *You can obtain feed forward on your verification plan up to 5 days before the deadline.*

Contents of the Verification Plan

- Description of the verification levels
- Functions to be verified
- Resource requirements
- Required tools
- Schedule
- Specific tests and methods
- Coverage requirements
- Completion criteria
- Test scenarios (Matrix)
- Risks and dependencies

Description of Verification Levels

- The first step in building the verification plan is to decide on which levels to perform the verification
- The decision is based on many factors, such as
  - The complexity of each level
  - Resources
  - Risk
  - Existence of a clean interface and specification
- The decision should include which functions are verified at lower levels and which at the current level
- Each level and piece selected need to have its own verification plan

Verification Levels for Calc1

- Calc1 is simple enough to be verified only at the top level
  - In addition we do not have enough details on the internal components
- In more realistic world we may decide to verify the ALU and shifter alone
  - For example, using formal verification

Functions to be verified

- This section lists the specific functions of the DUV that the verification team will exercise
  - Omitted functions may slip away and not be verified
- Assign Priority for each function
  - Critical functions
  - Secondary functions
- Functions not verified at this level
  - Fully verified at a lower level
  - Not applicable to this level

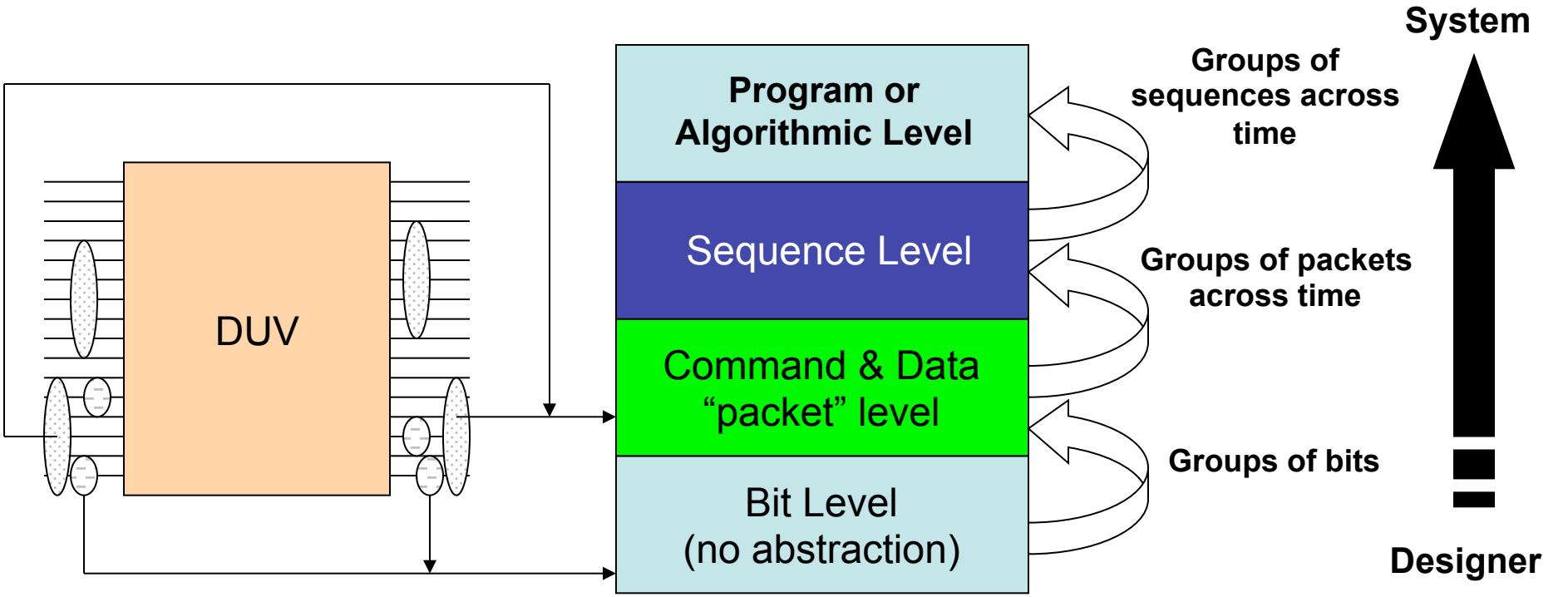
Required Tools

- Specification and list of the verification toolset
  - Simulation engines
  - Debuggers
  - Verification environment authoring tools
  - Formal verification tools
  - … and more
- For **Calc1 A1**
  - Simulation engine
  - Waveform viewer
  - Verification environment authoring tool

Specific Tests and Methods

- What type of Verification?
  - Black box
  - White box
  - Grey box
- Verification Strategy
  - Formal Verification
  - Deterministic
  - Random based
- Abstraction level
  - From bit-level to algorithmic level
  - From bit-streams to transactions (packets)
- Checking
  - Simple I/O checking for data correctness
  - Behavioral rules for timing

Abstraction Levels



Which level(s) will you use for A1?

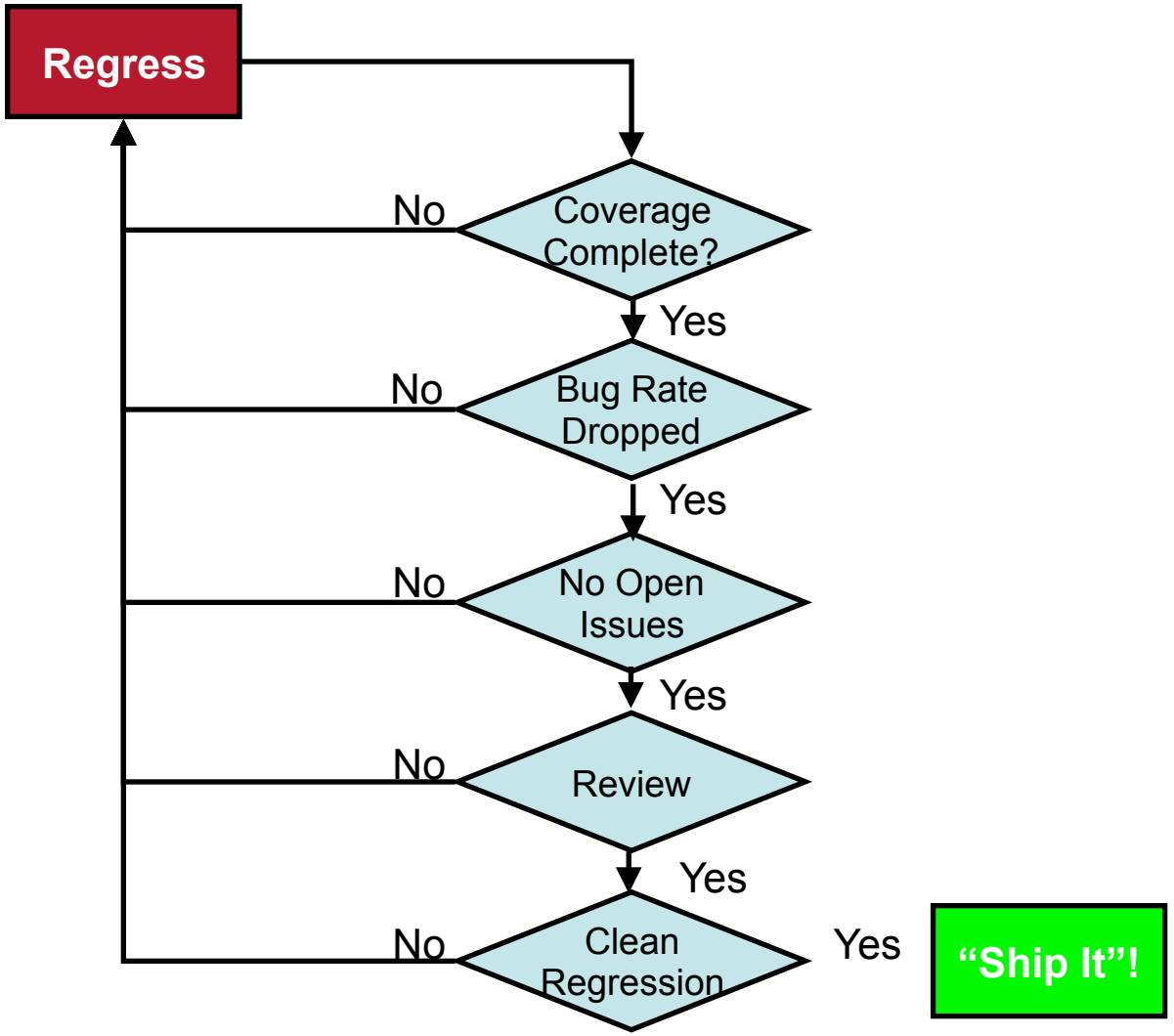
Coverage Requirements

- Traditionally, **coverage is the feedback mechanism that evaluates the quality of the stimuli**
  - Required in all random-based verification environments
  - Some aspects of coverage are directly achieved in deterministic testing
- Coverage is defined as events (or scenarios) or families of events that span the functionality and code of the DUV
  - The environment has exercised all types of commands and transactions
  - The stimulus has created a specific or varying range of data types
  - The environment has driven varying degrees of legal concurrent stimulus
- Soon: **Coverage metrics**

Completion Criteria

These might include:

- Coverage targets
- Target metrics, e.g. bug rate drop
- Resolution of open issues
- Review
- Regression results



Test Scenarios (Matrix)

- Specifies test scenarios that will be used throughout the verification process
  - deterministic or random
  - Scenarios are connected to items in the coverage requirements
- Start with a basic set for the basic functionality
  - Add more tests to plug holes in coverage, reach corner cases, etc.
- Examples for calc1 design

Test Scenarios for Calc1: Basic Tests (partial list)

Test #	Description
1.1	Check the basic command-response protocol on each of the four ports for each command
1.2	Check the operation of each command (on each port?)
1.3	Check overflow and underflow for add and subtract commands
1.4	Check …

NOTE: *"Check that X"* should be read as "Create a scenario that allows checking X".

- These generic tests should be broken to more specific tests
  - Test case 1.1.1.1 : Check the protocol for add command on channel 1
  - …
  - Test case 1.1.2.4 : Check the protocol for sub command on channel 4

Test Scenarios for Calc1: Advanced Tests (partial list)

Topic	Test #	Description
Command sequences	2.1.1	For each port, check that each command can be followed by other command without leaving the state of the design dirty
	2.1.2	For all ports combined, check that each command can be followed by other command without leaving the state of the design dirty (concurrent commands)
Fairness	2.2	Check that there is fairness among the channels
Corner cases	2.3.1	Add two numbers that overflow by 1
	2.3.2	Add two numbers that reach the maximum value
	2.3.3	Subtract two numbers that underflow by 1
	2.3.4	Subtract two equal numbers (result is 0)
	2.3.5	Shift (left and right) 0 places
	2.3.6	Shift completely out (left and right)

Risks and Risk Management

- Complexity of design project
- Architecture and microarchitecture closure
- Resources
  - Not just verification
- New tools
- Deliveries
  - Internal
  - External
- Dependencies
  - Design availability
  - Quality of lower levels verification
  - Tools and verification IP

Summary

- Verification Cycle
  - Foundation for successful verification
- Verification Methodology
  - Evolution of:
    - Test patters
    - Test cases
    - Test case generators/drivers
- Verification Plan
  - The specification for the verification process.