COMS30026 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.)

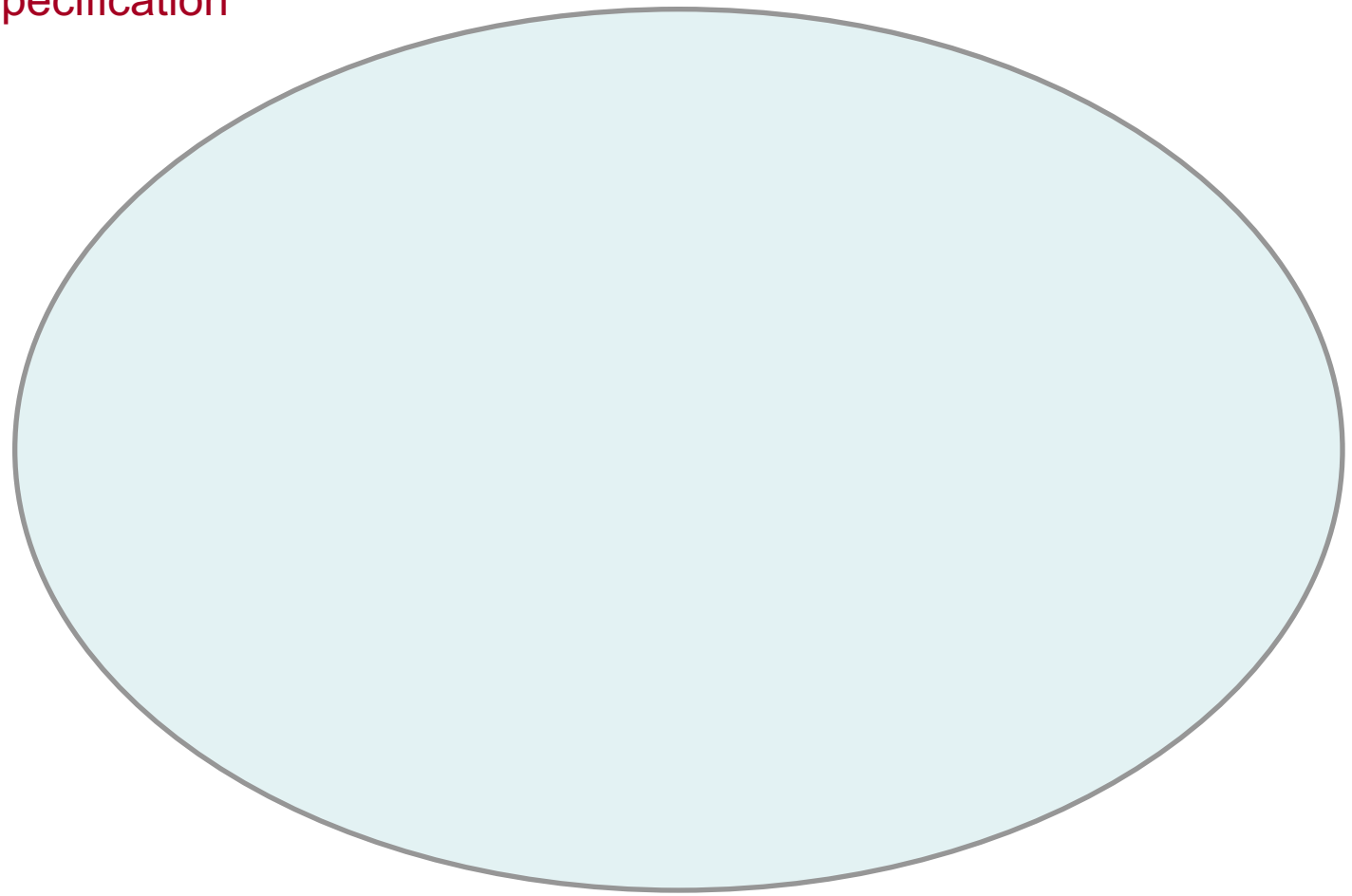
University of BRISTOL

Department of COMPUTER SCIENCE

The Verification Cycle

The Verification Cycle

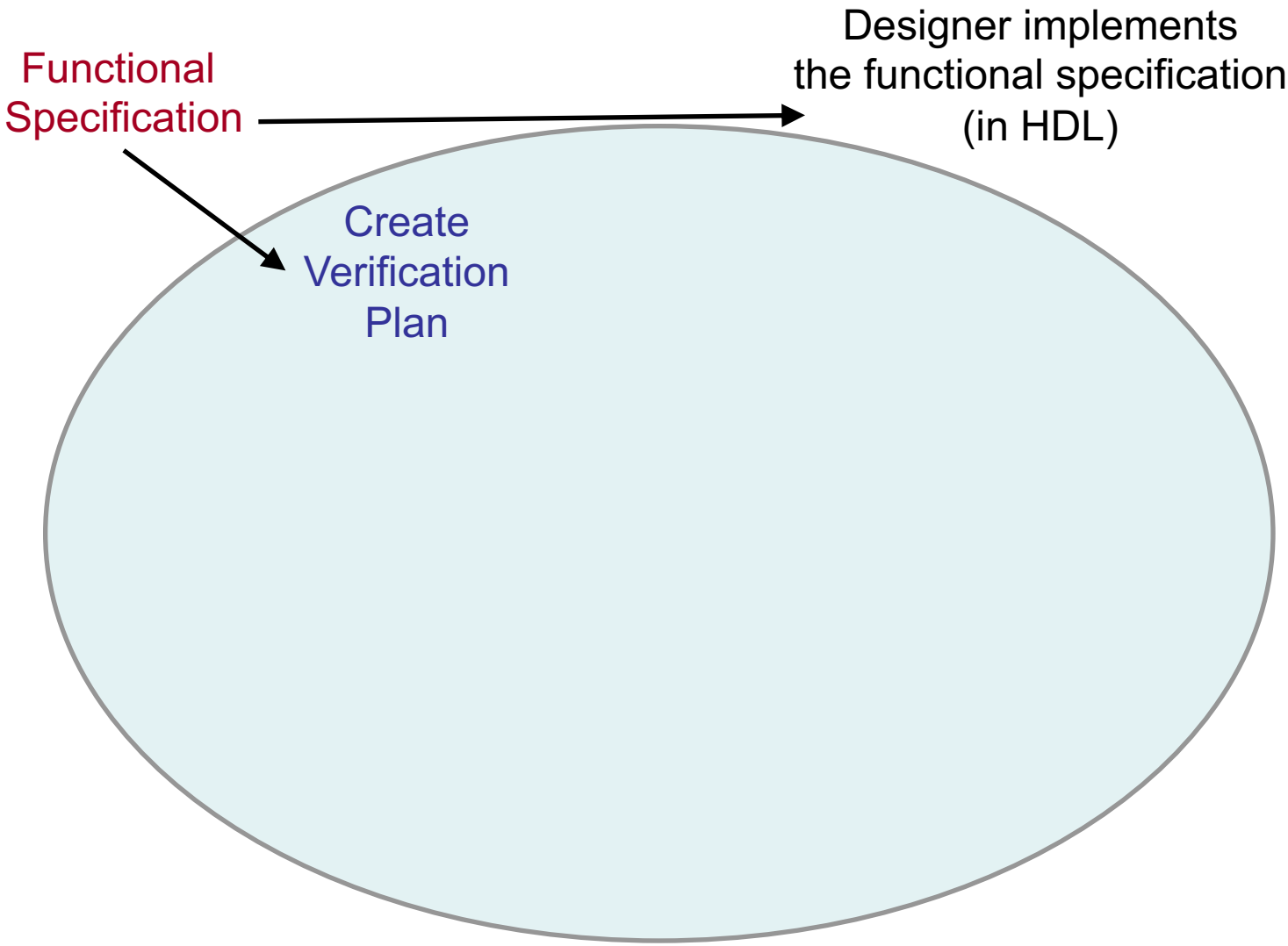
Functional Specification



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 process.**

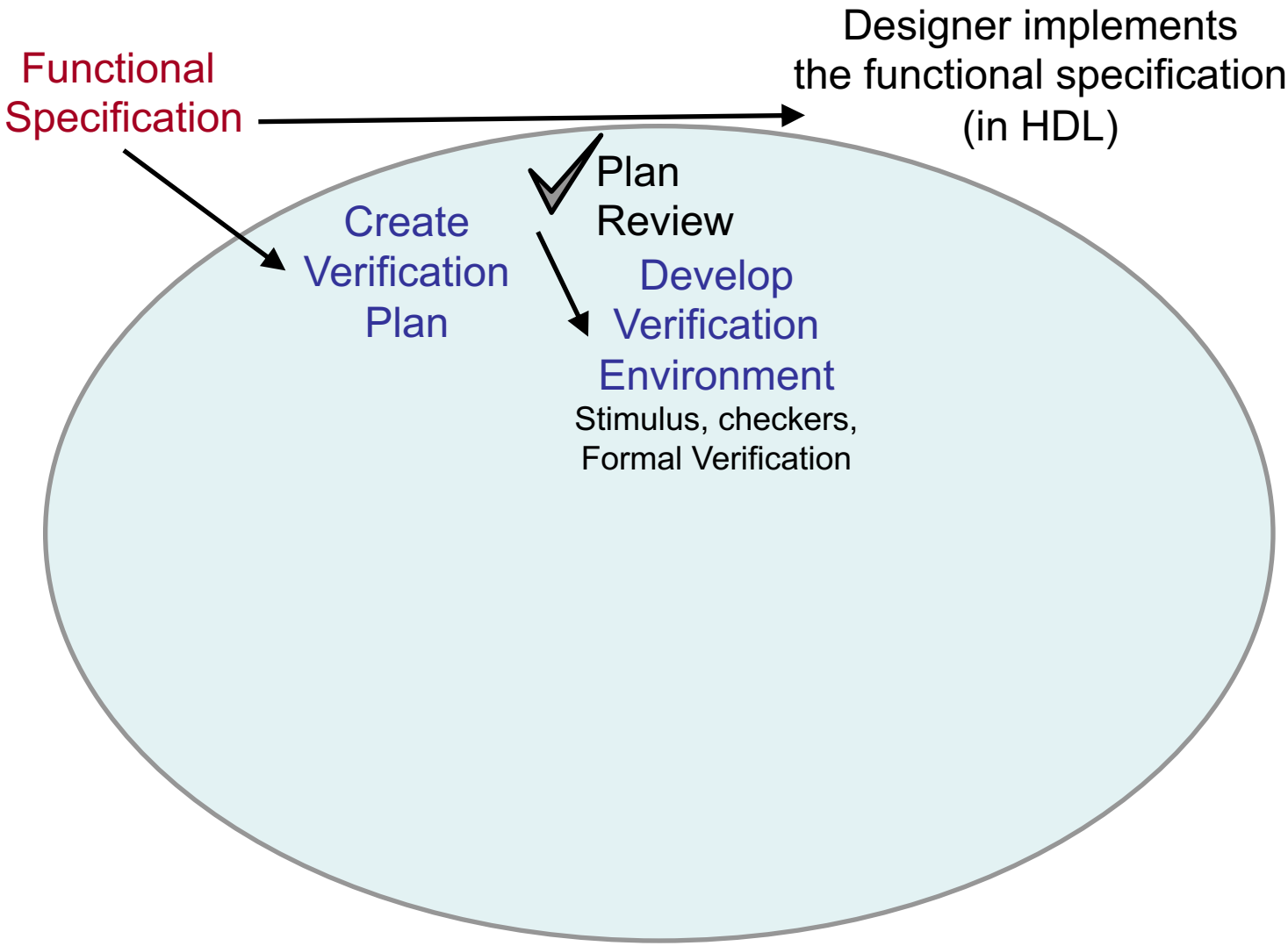
The Verification Cycle



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.
- **Specific tests and methods:**
  - Define the type of environment that the verification engineers will create.
- **Completion criteria:**
  - Define the measurements that indicate when verification is complete.
- **Required tools:**
  - List the software and hardware necessary to support the verification planned.
- **Resources required** (people) and **schedule details**:
  - Goal is to link the plan to the project management by estimating the cost of verification.

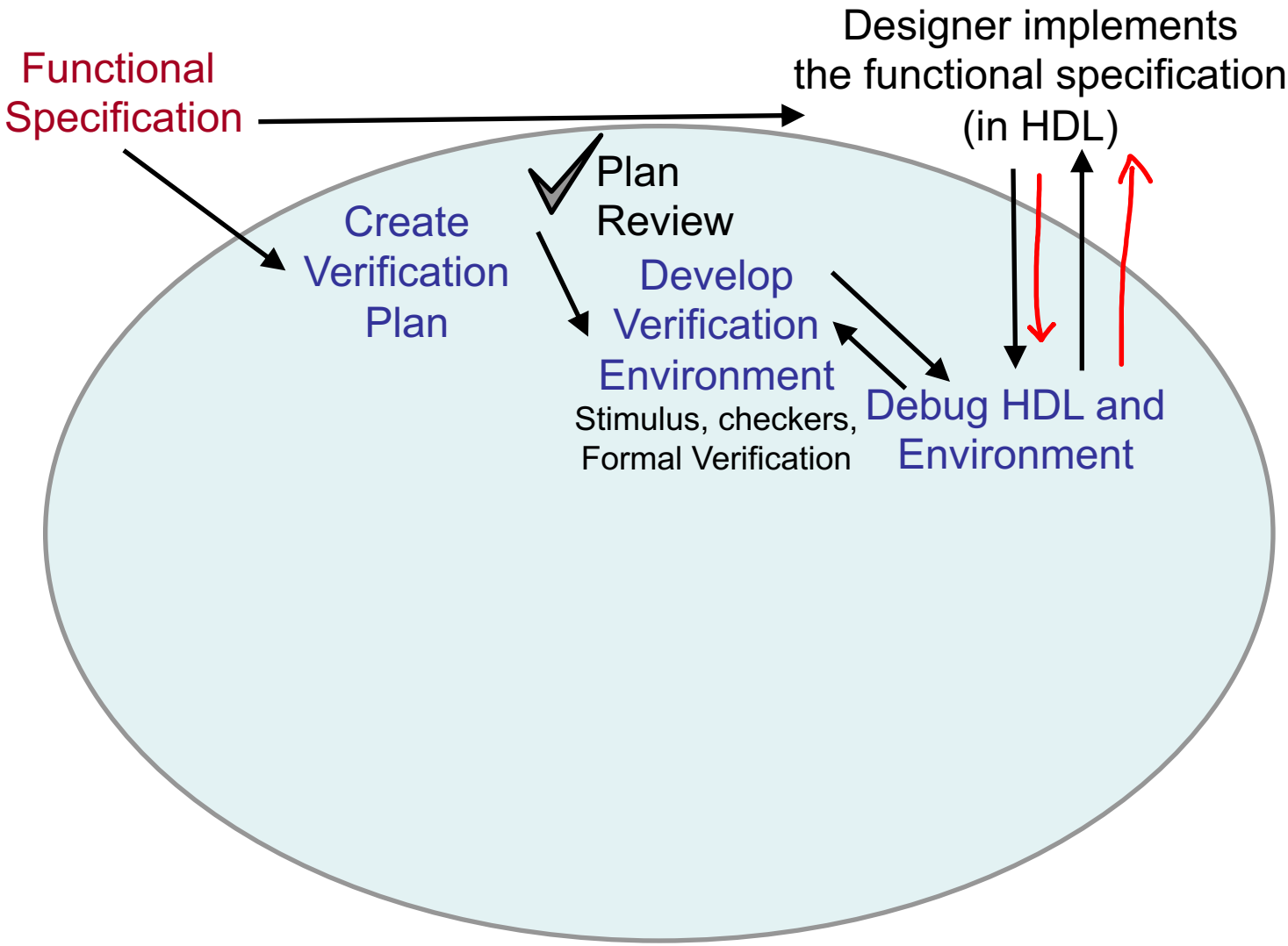
The Verification Cycle



Develop Verification Environment

- The verification environment includes the software code (e.g. testbench) 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
  - e.g. fixes and additions to the software code

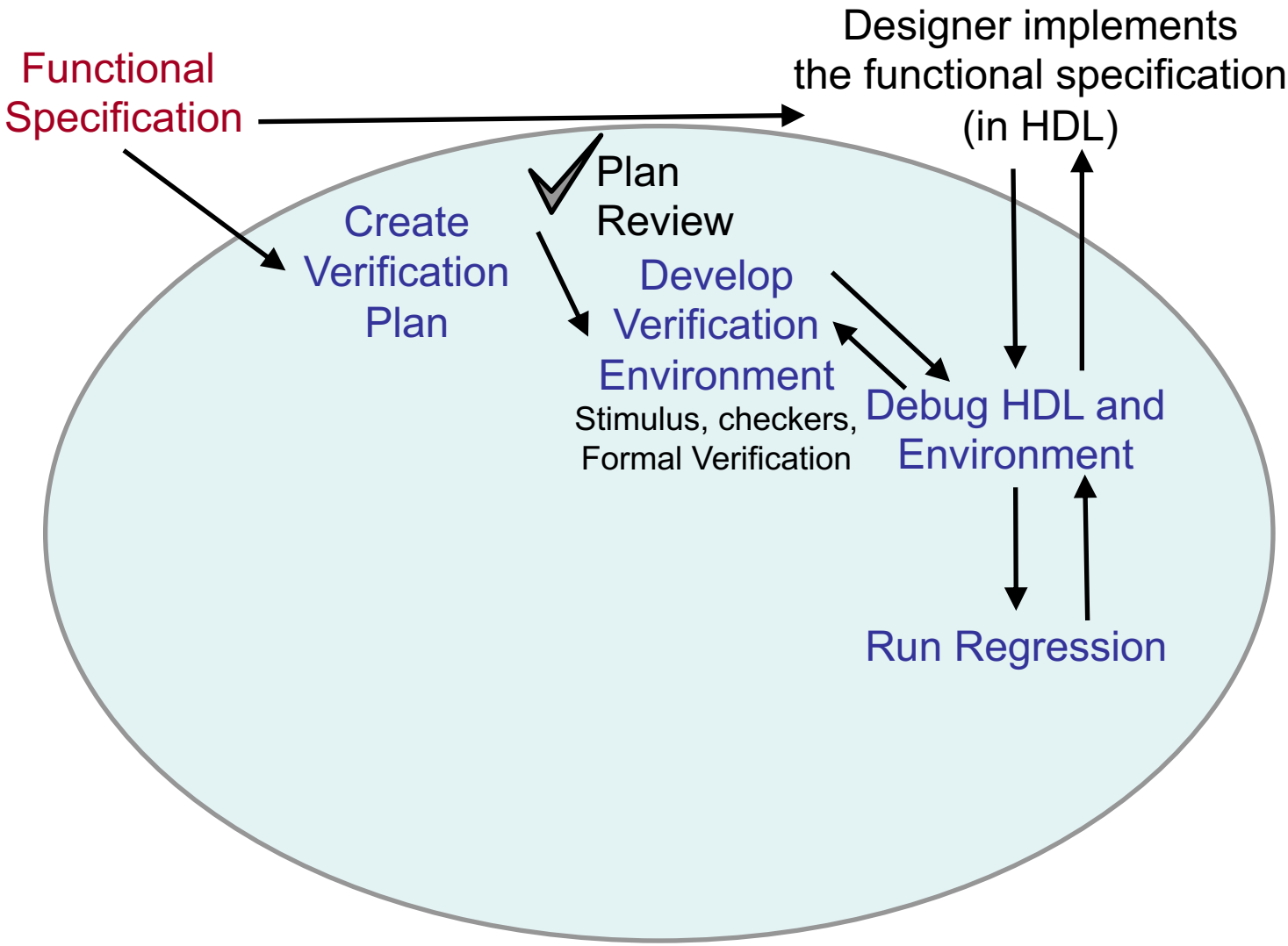
The Verification Cycle



Debug HDL and Environment

- Run tests according to the verification plan and look for anomalies
- Examine the anomalies to reveal the failure source, the root cause of the failure
  - Remember, bugs 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

The Verification Cycle

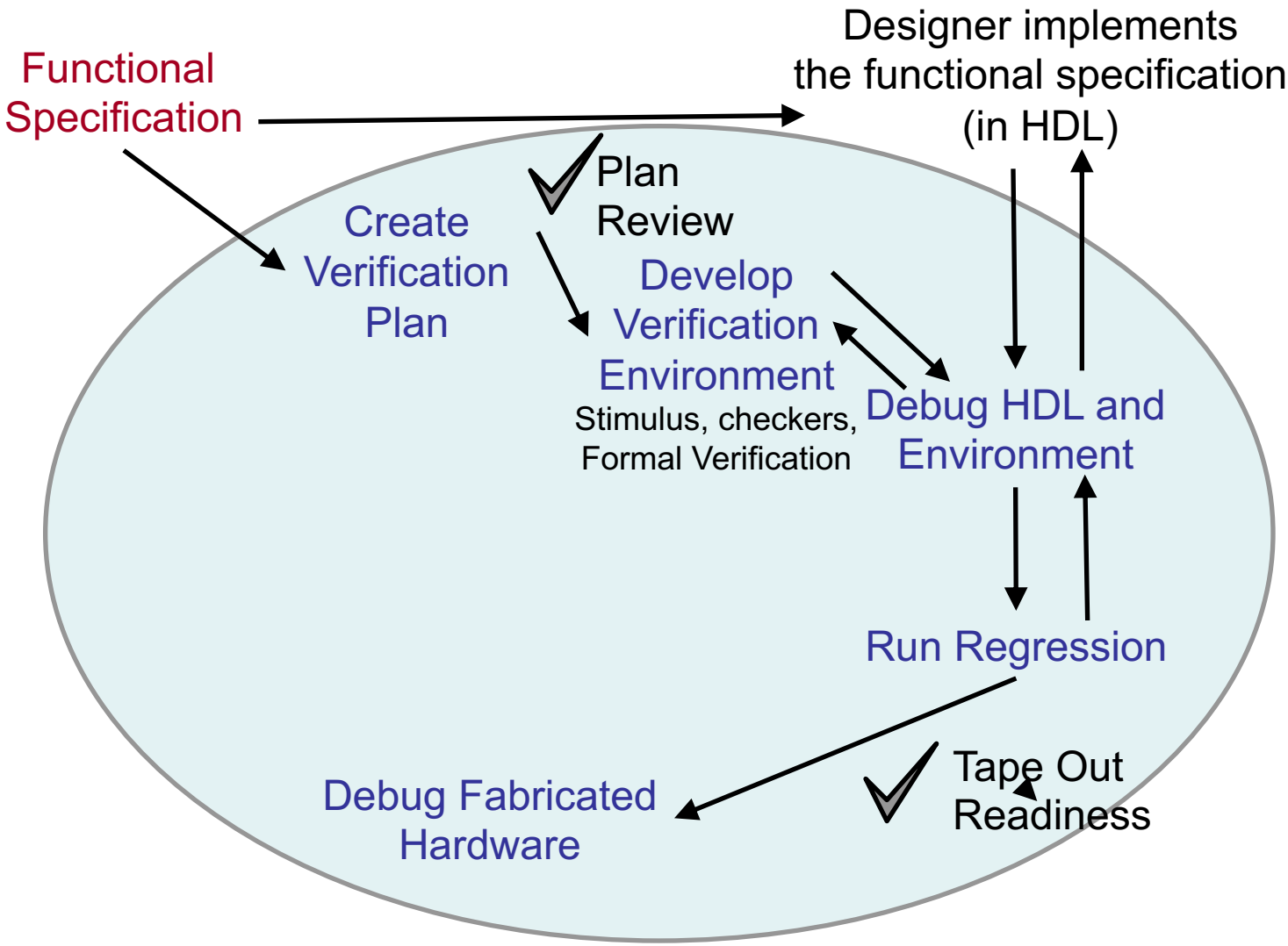


Run Regression

- 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**

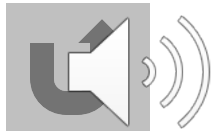
  This is the tape-out readiness checkpoint.

The Verification Cycle

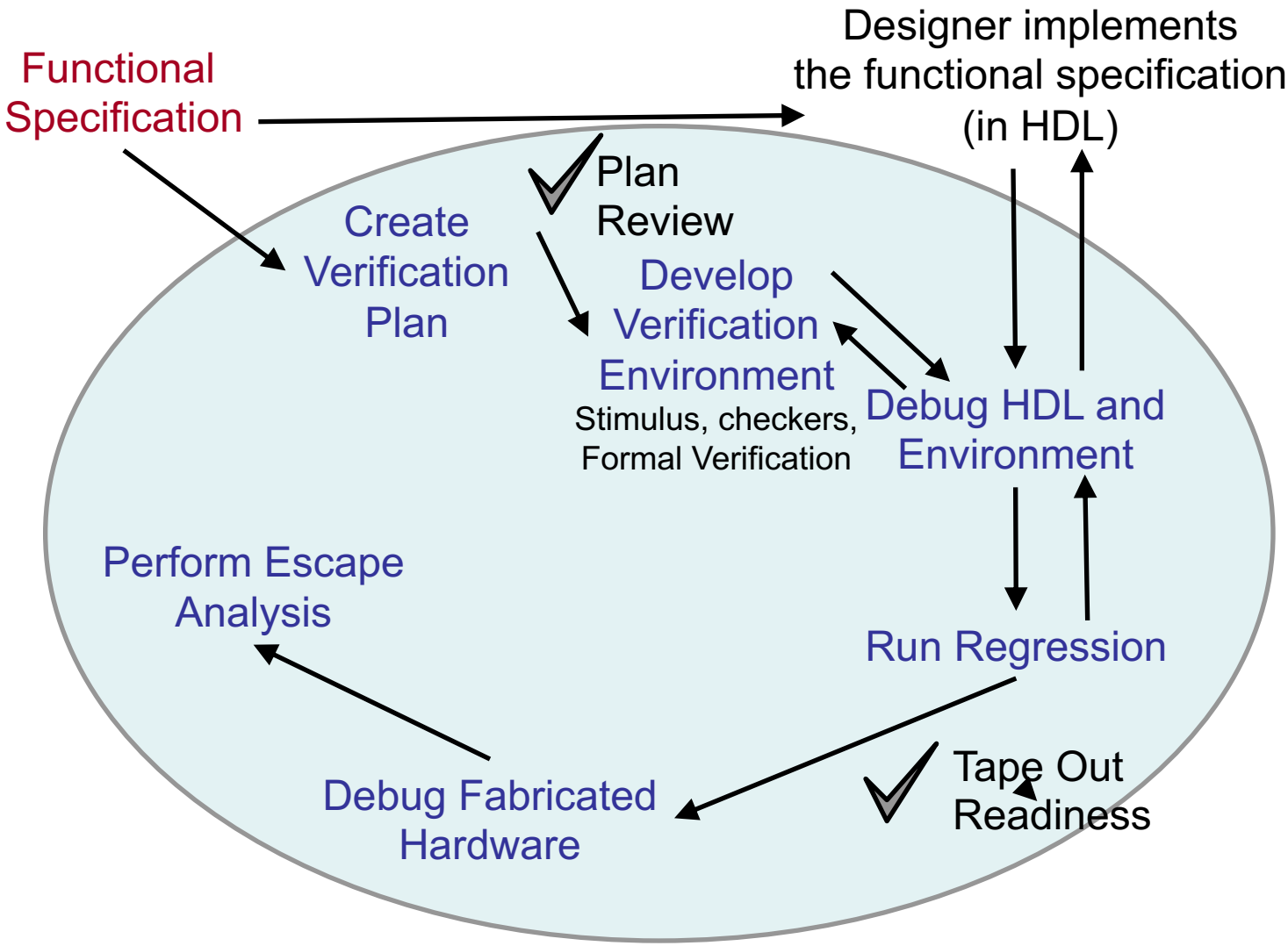


Debug Fabricated Hardware

- 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.



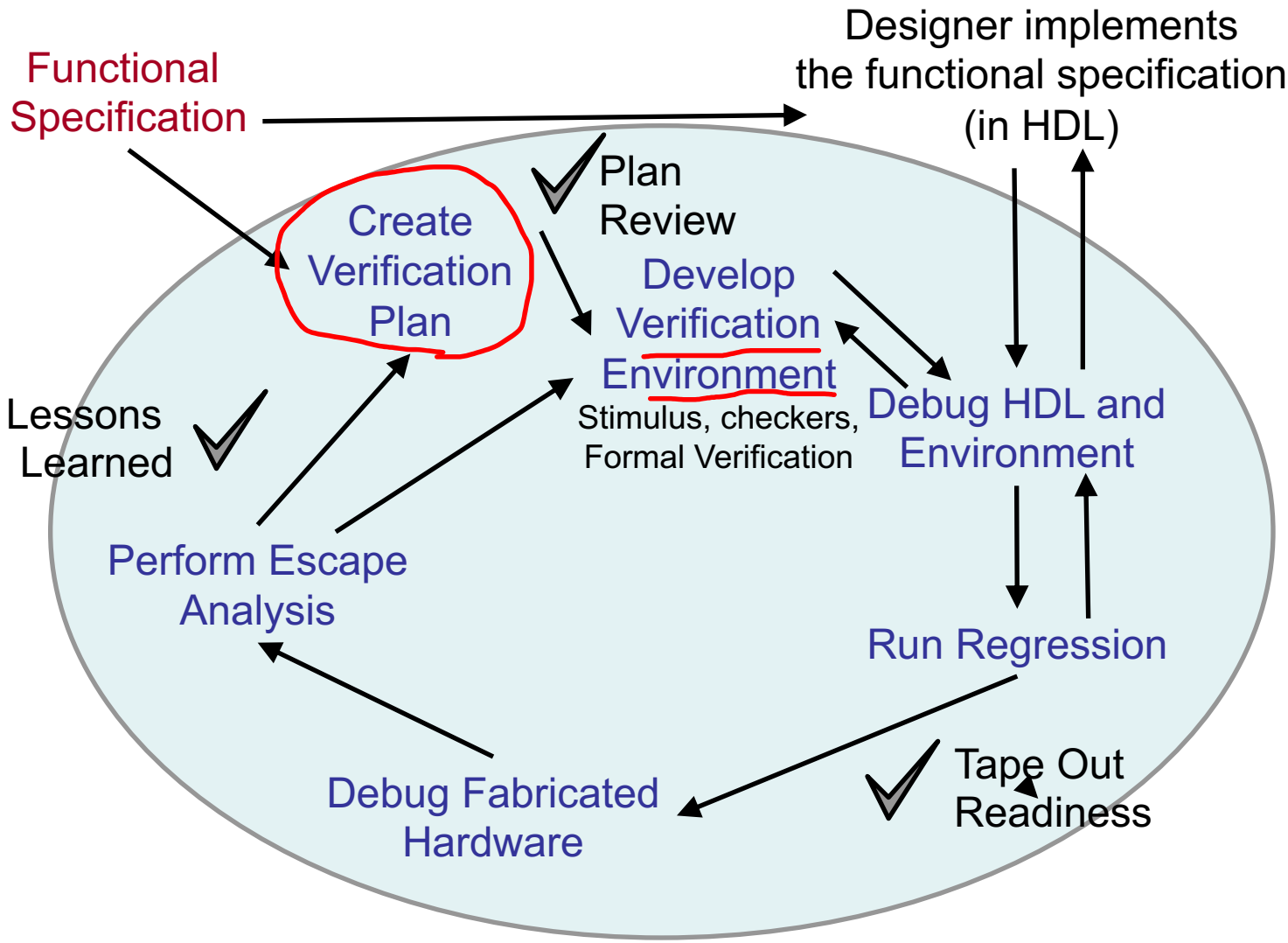
The Verification Cycle



Perform Escape Analysis

- Analysis of bugs that were found later than when they should have been found
- The goal is to fully understand the bug, as well as the reasons why it went undiscovered by the verification environment
- **Important goal: Reproduce the bug in a simulation environment, if possible.**
  - The lack of reproduction in the verification environment indicates that the design and/or the verification team may not understand the bug.
  - It would then follow that the team cannot claim that the bug fix is correct or complete unless they can reproduce the original bug in the verification environment and then re-run specific tests on the updated design to demonstrate that the bug fix has been effective and no new bugs have been introduced.
- Reproducing the bug in the verification environment is also critically important to make sure that this same bug won't escape again in any future versions of the design.

The Verification Cycle



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 practical too.

Summary

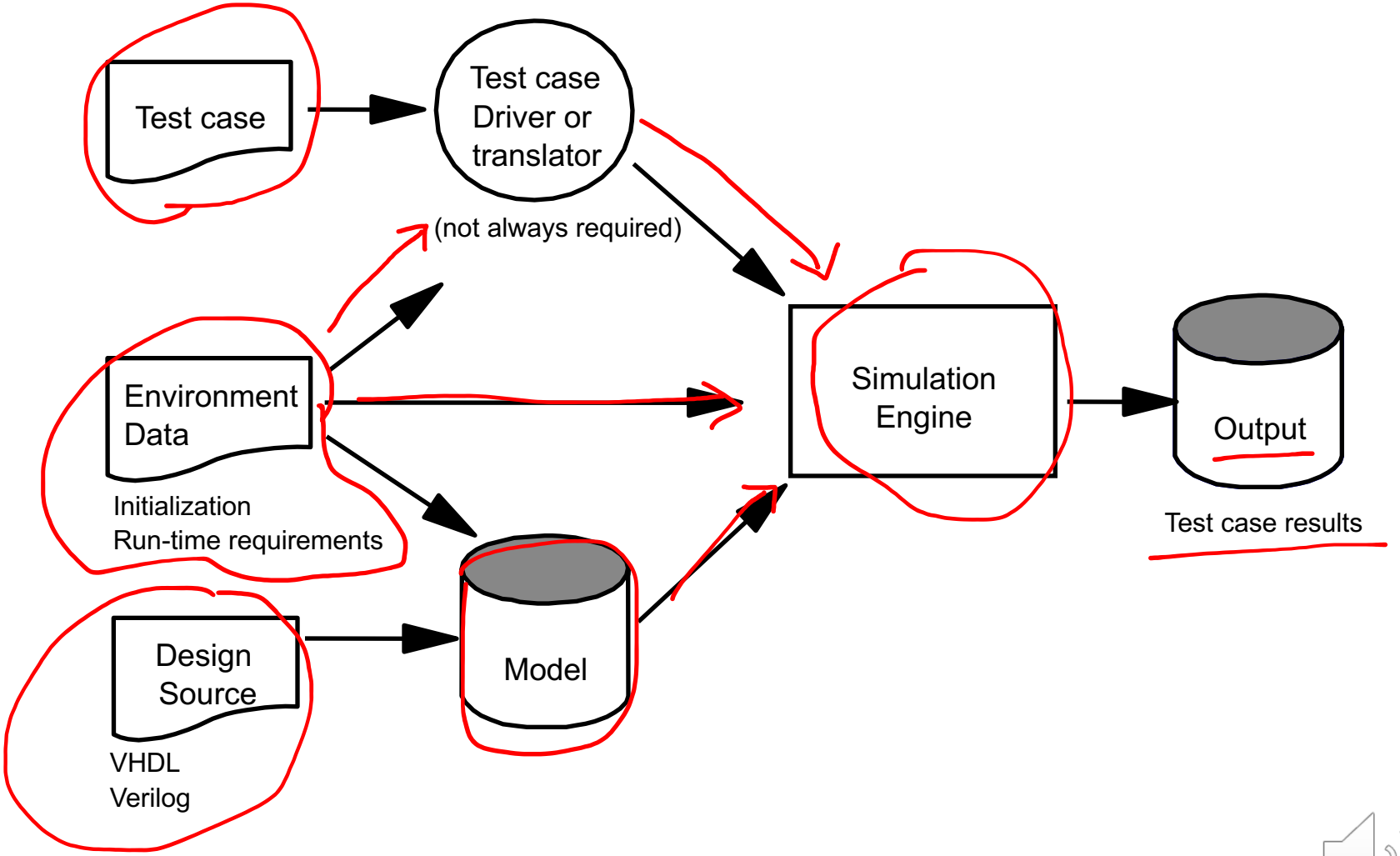
- Functional verification is a necessary step in the development of today's complex digital designs
- Verification engineers must understand the specification, architecture and internal microarchitecture of the design under verification
  - They couple this knowledge with programming skills, RTL comprehension, 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 stimuli
  - Identification of incorrect behavior when encountered

The foundation for successful verification is a well-defined verification cycle

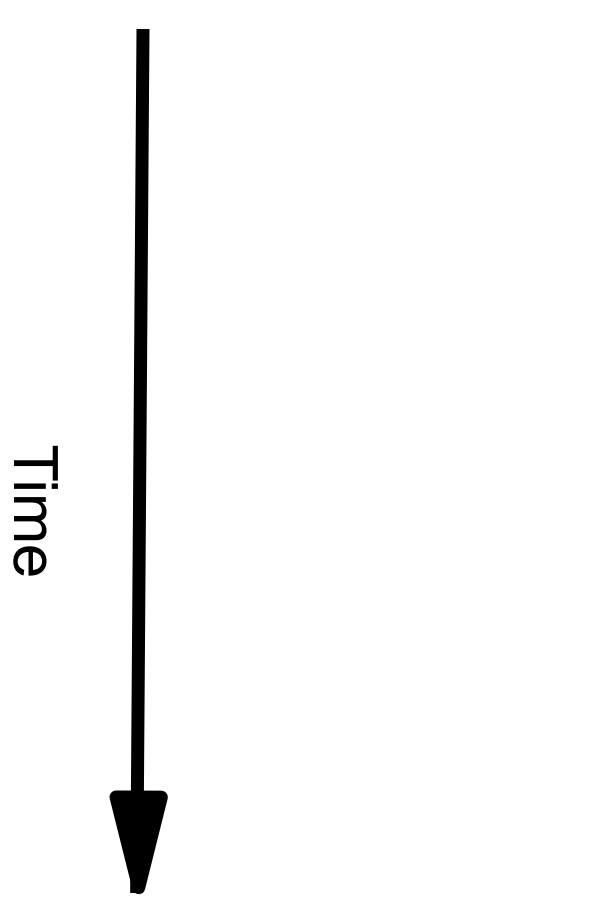
- The process includes creation of test plans, development of the verification environment, performing verification, debugging, and the analysis of any holes in the verification environment and/or the verification plan

Verification Methodology

Simulation-based Verification Environment Flow



Verification Methodology Evolution

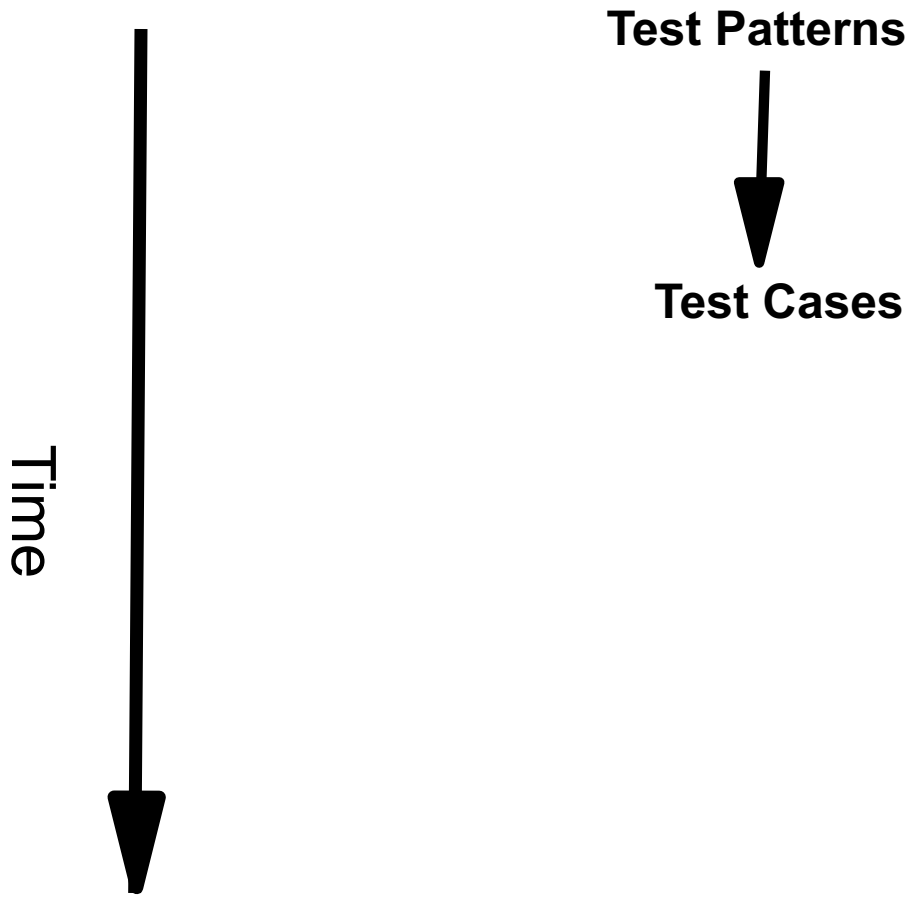


(Remember that there is also Formal Verification, but this is not covered here.)

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

Verification Methodology Evolution

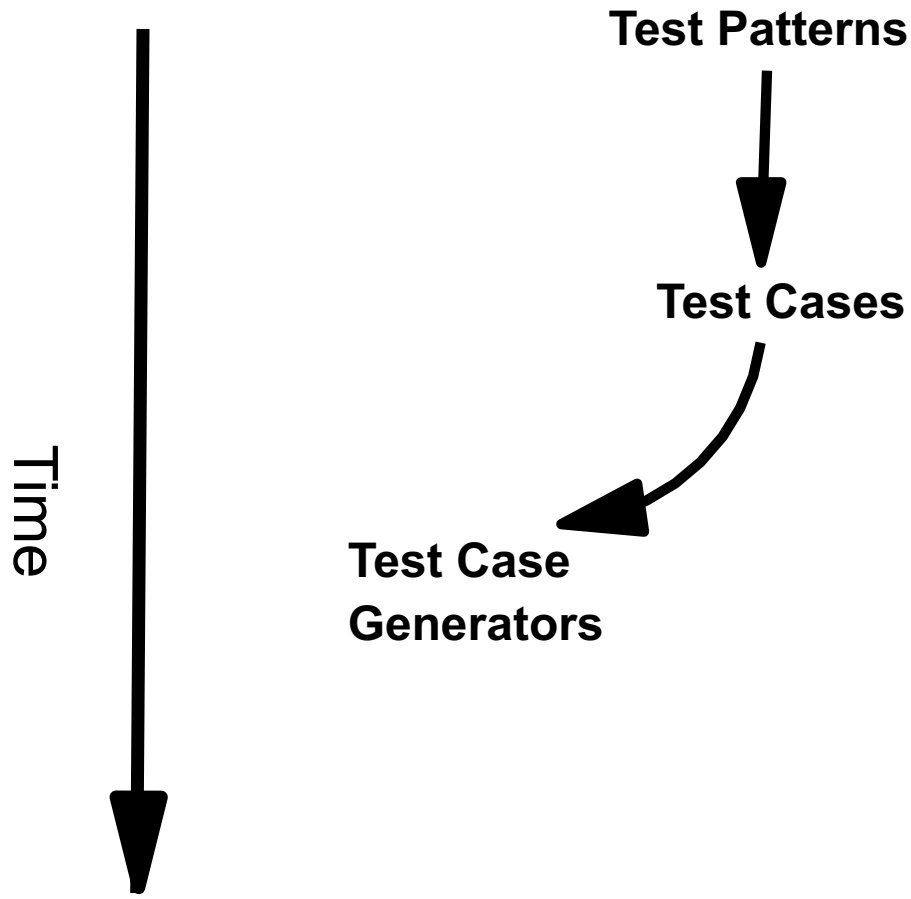


(Remember that there is also Formal Verification, but this is not covered here.)

Test Cases

- Stimulus is still hand generated and for a single scenario, but there is a significant change wrt. checking
- Checking is automated over two stages
  - Self checking tests
    - The test knows at which signals to look and which values should be there at the end of the test
  - 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

Verification Methodology Evolution

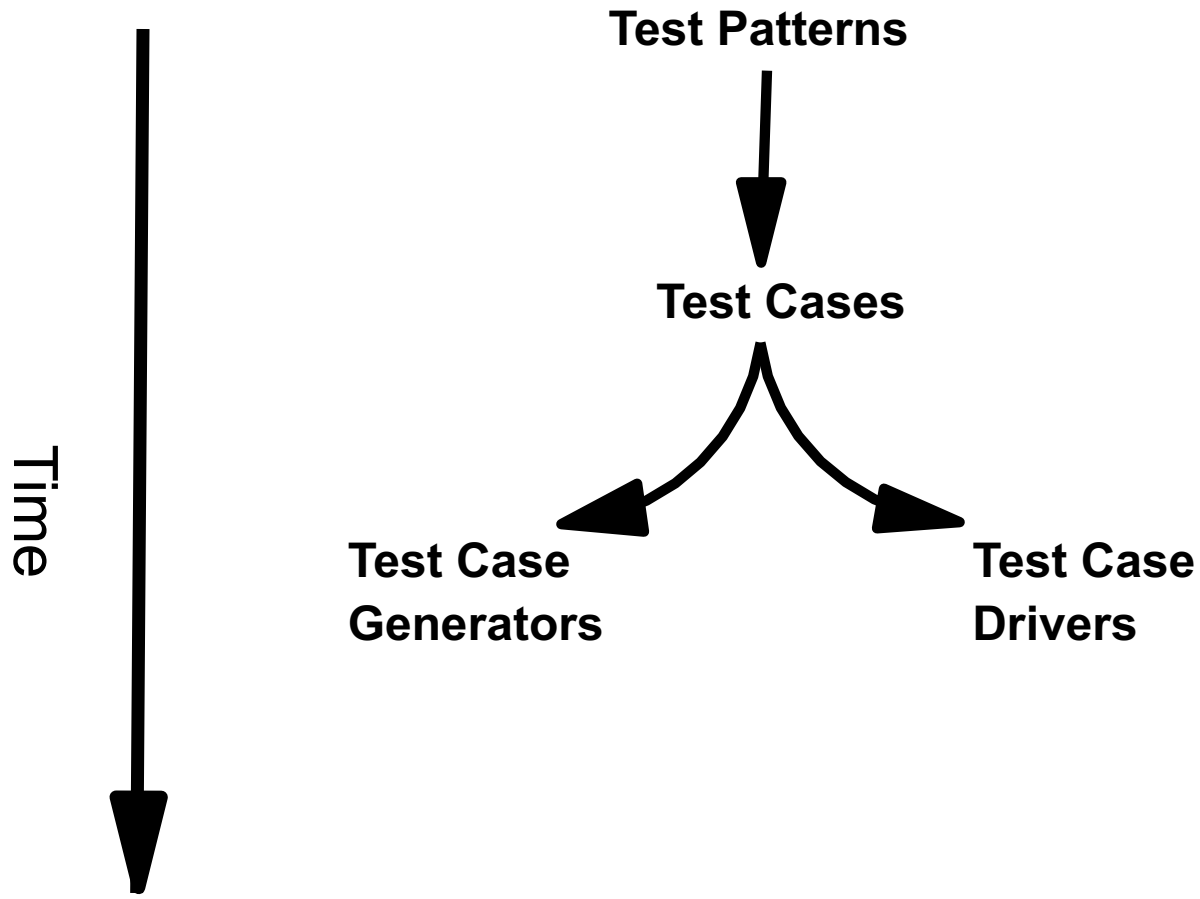


(Remember that there is also Formal Verification, but this is not covered here.)

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

Verification Methodology Evolution

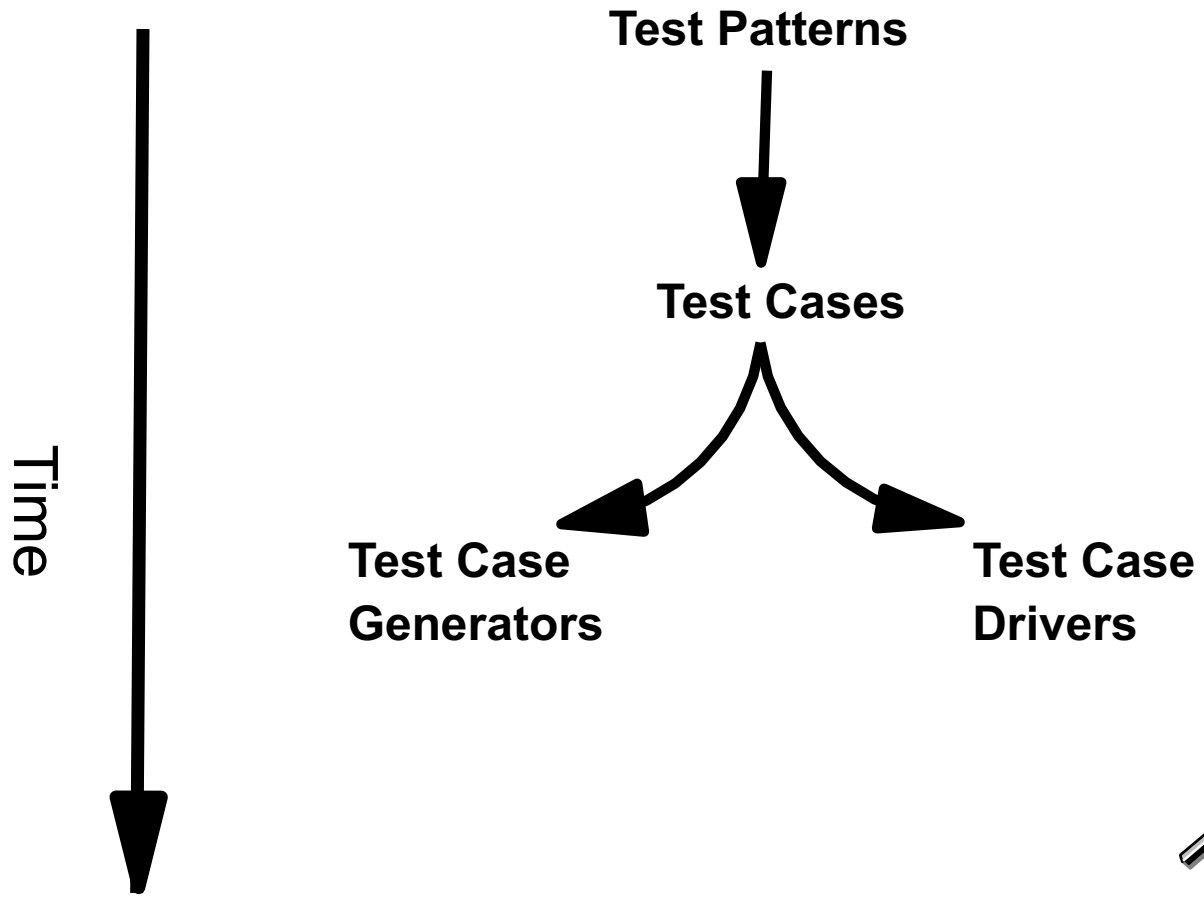


(Remember that there is also Formal Verification, but this is not covered here.)

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

Verification Methodology Evolution



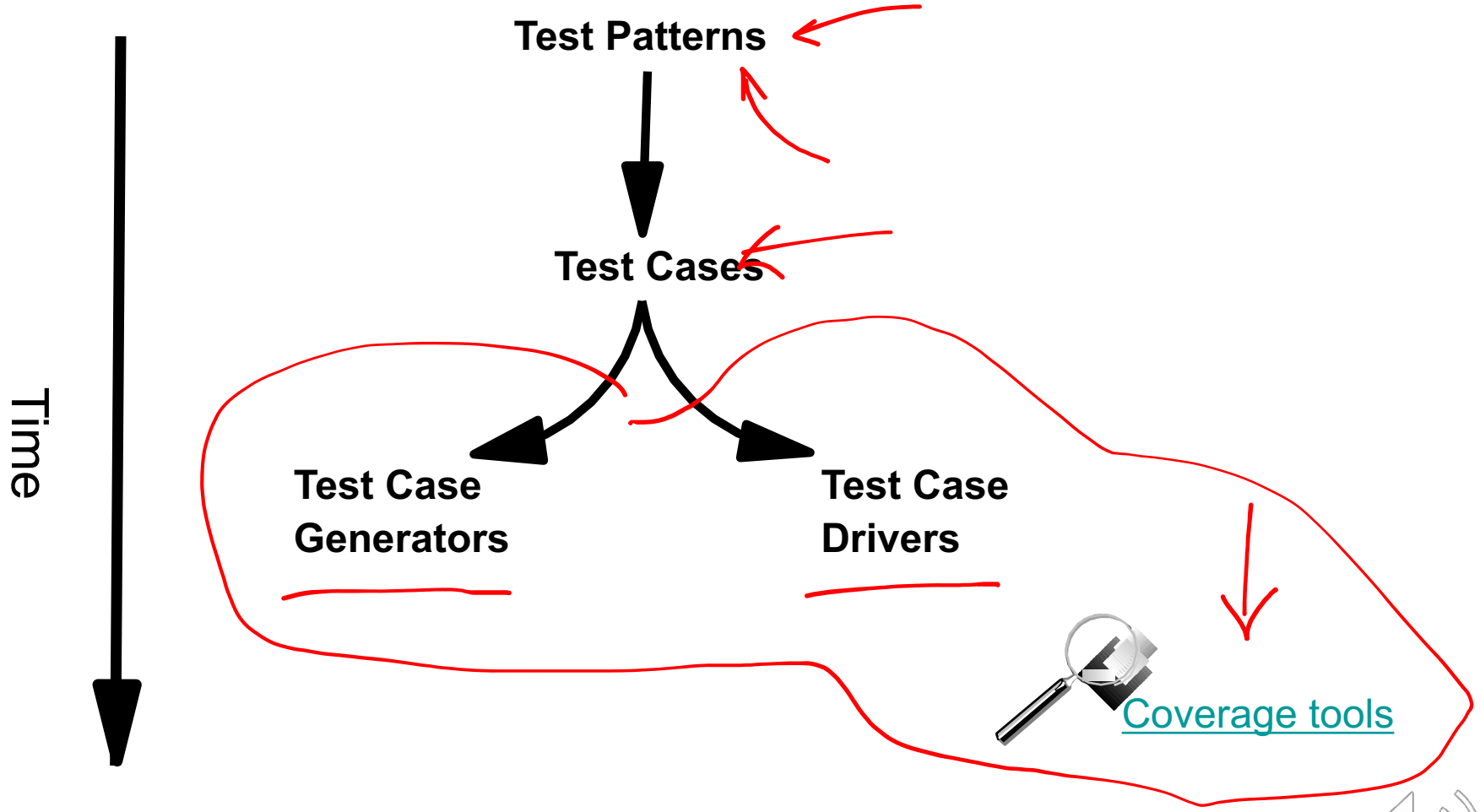
Coverage tools

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

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 Methodology Evolution



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

Verification Plan

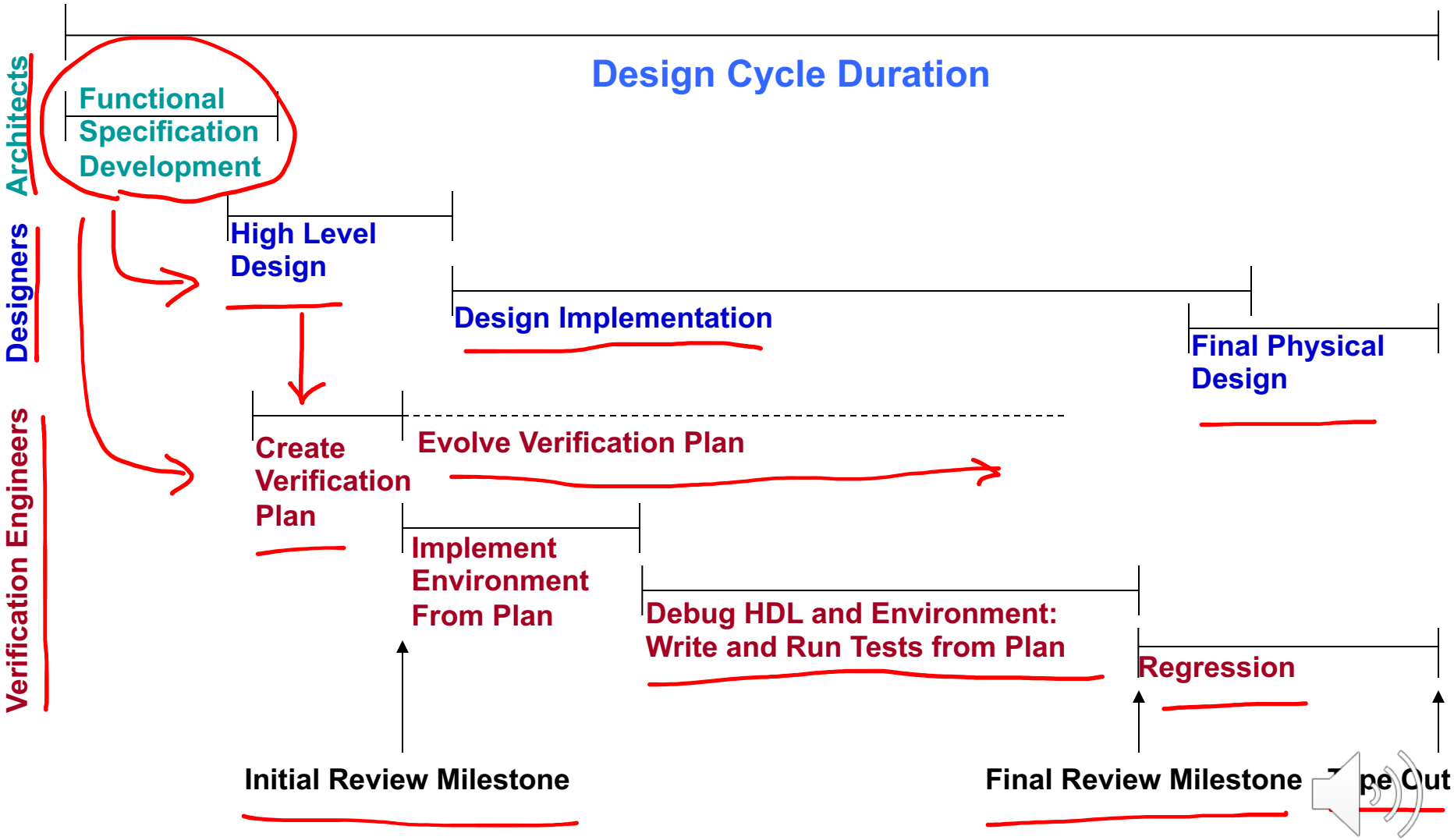
Outline

- Evolution of the **Verification plan**
- Contents of the **Verification plan**
  - Functions to be verified
  - Specific tests
  - Coverage goals
  - Test case scenarios (Test list)
- calc1 DUV example

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



Contents of the Verification Plan

- Description of the verification levels
- Functions to be verified
- Specific tests and methods
- Test scenarios (Matrix)
- Coverage requirements
- Completion criteria
- Resource requirements
- Required tools
- Schedule
- 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 which level
- Each level and function selected need to have their own verification plan

calc1 DUV Verification Plan

- The design description 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 verification requirements and developing a verification plan.
- Please note:
  - For the next live session, you can bring a verification plan – no matter how sketchy.
  - *We will review some of these verification plans.*
  - *The feedback can be used to improve these plans.*

Verification Levels for calc1

- calc1 is simple enough to be verified only at the top level of the DUV
  - In addition, we do not have enough details on the internal components (black box)
- 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
- Functions not verified at this level
  - Fully verified at a lower level
  - Not applicable to this level
- Assign Priority for each function
  - Critical functions
  - Secondary functions

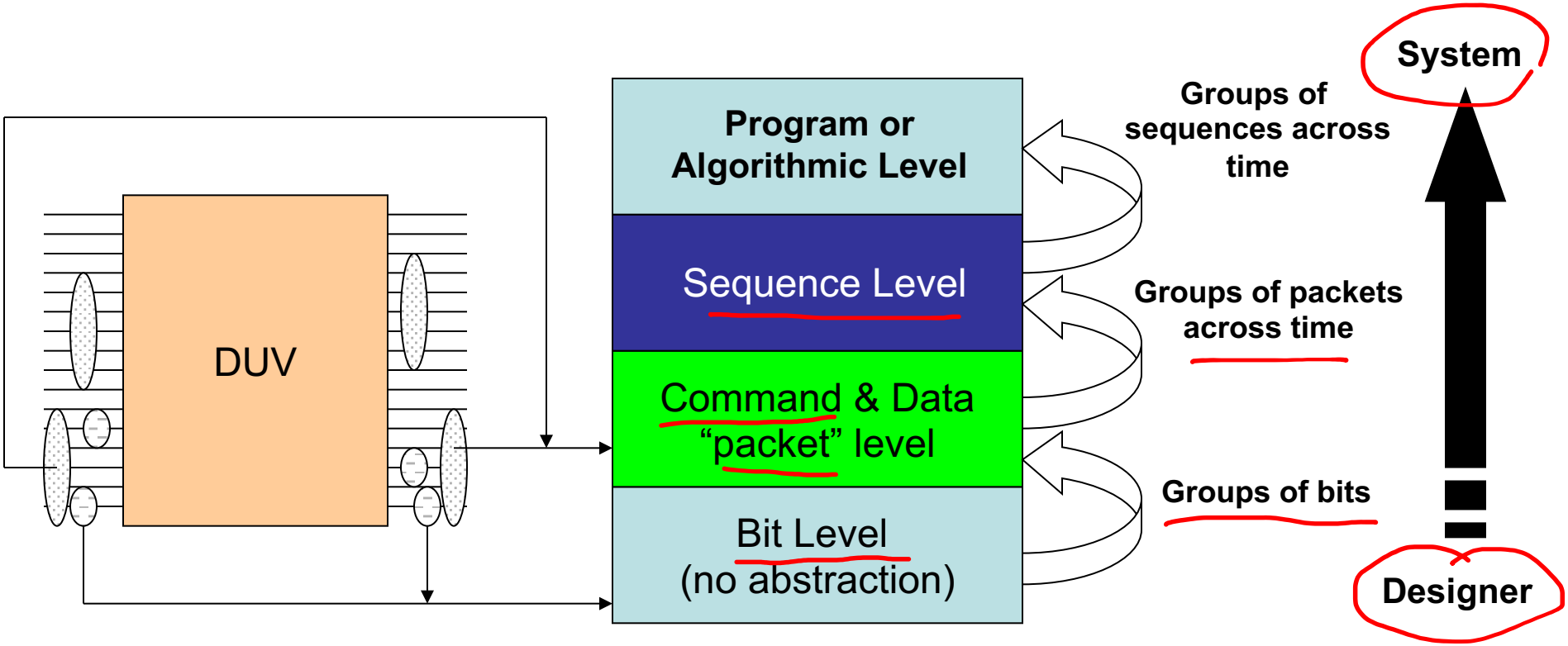
Required Tools

- Specification and list of the verification toolset
  - Simulation engines
  - Debuggers
  - Verification environment authoring tools
  - Formal verification tools
  - … and more
- For **calc1 practical**
  - Simulation engine
  - Waveform viewer
  - Verification environment authoring tool

Decisions to be made

1. What level of observability and controllability?
   - Black box
   - White box
   - Grey box
2. Verification Strategy
   - Directed testing
   - Constrained pseudo-random test generation
   - Formal Verification
3. Checking
   - I/O checking for data correctness
   - Behavioral rules for timing compliance, priority logic, etc
4. Abstraction level
   - From bit-level representation
   - via transactions (packet sequences or instruction streams)
   - up to the algorithmic level.

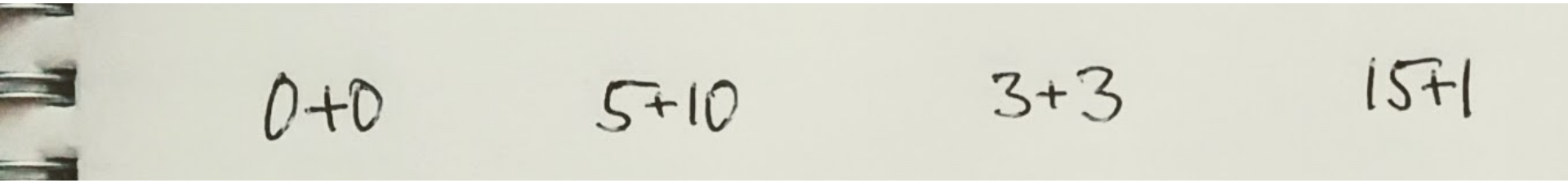
Abstraction Levels



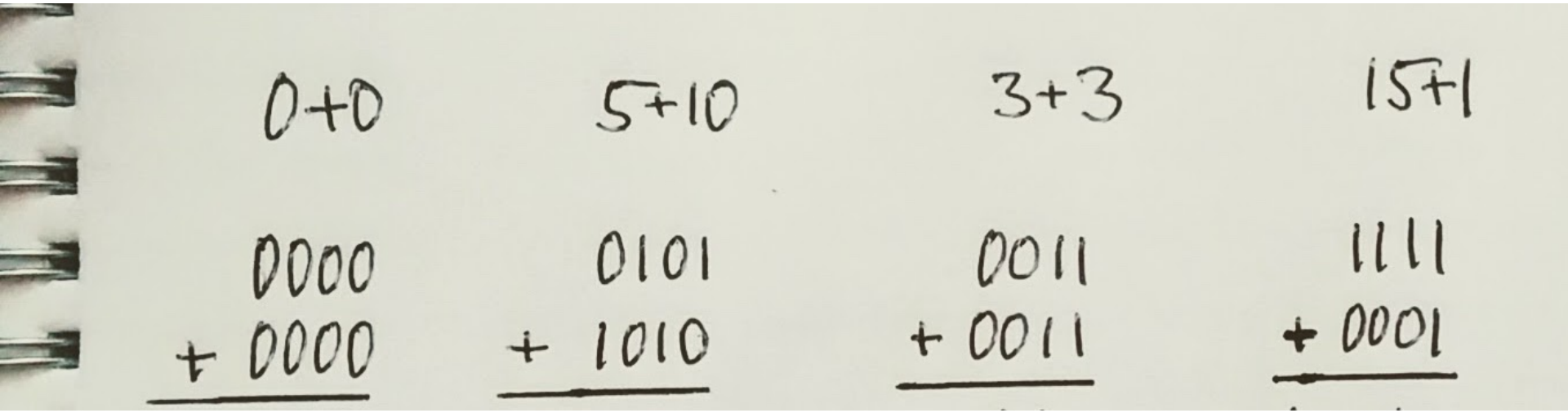
Which level(s) will you use for the calc1 DUV?

Verification Level

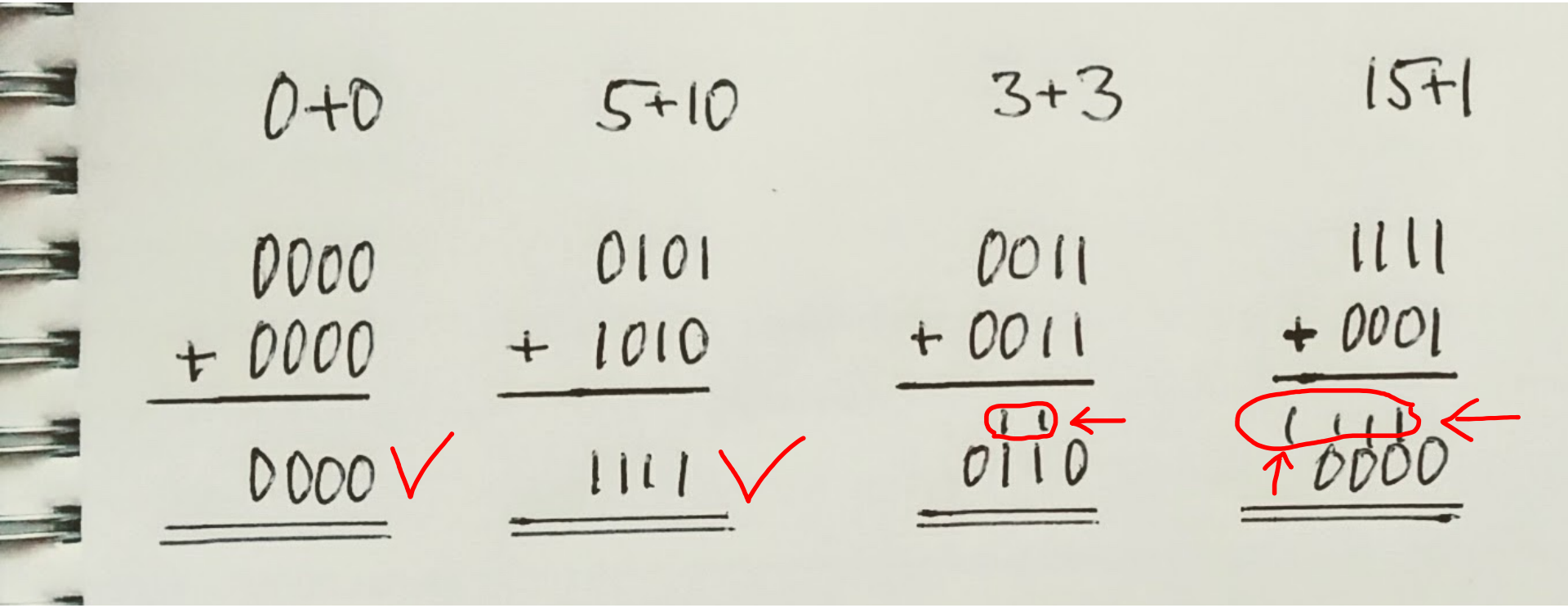
Integer vs bit-level data



Integer vs bit-level data



Integer vs bit-level data



Using a different level of abstraction allows us to see how these numbers differ in terms of how they exercise the logic in the design.

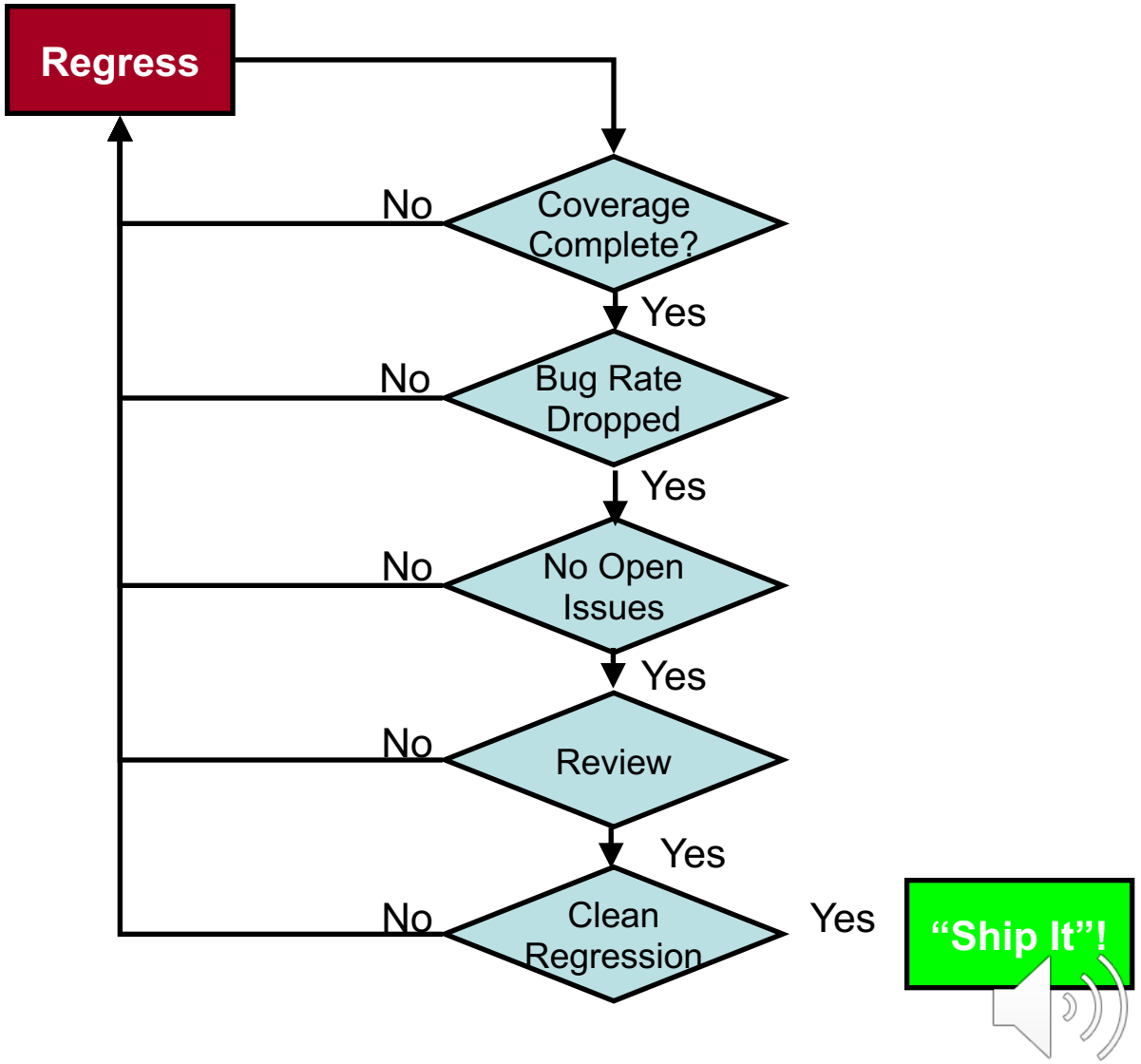
Coverage Requirements

- 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 generator has created a specific or varying range of data
  - The environment has driven varying degrees of legal concurrent stimulus
- **Coverage is the feedback mechanism that evaluates the quality of the stimuli**
  - Some aspects of coverage can be achieved using direct testing
  - Mandatory in all random-based verification environments

Completion Criteria

These might include:

- Coverage targets
- Other 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
  - direct testing and pseudo-random techniques
- 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	Create a scenario that allows checking **the basic command-response protocol** on each of the four ports for each command
1.2	Create a scenario that allows checking the **operation of each command** (on each port?)
1.3	Create a scenario that allows checking **overflow and underflow** for add and subtract commands
1.4	Create a scenario that ...

These generic tests should be broken to more specific tests

- Test case 1.1.1.1 : ... protocol for add command on channel 1
- …
- Test case 1.1.2.4 : ... 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 individual port, check that each command can be followed by another command without leaving the state of the design dirty
	2.1.2	For all ports combined, check that each command can be followed by another 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
	2.3.2	Add two numbers that reach the maximum value
	2.3.3	Subtract two numbers that underflow
	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

- Consider verification process in the context of the complexity of the design and the overall design project
- Maturity and closure of the architecture and microarchitecture
- Availability of resources
  - Not just those used for verification, but also those beyond your control.
  - New tools and associated learning curve
- Deliveries
  - Internal
  - External
- Dependencies
  - Design availability
  - Quality of lower levels of verification
  - Tools and third-party verification IP

Summary

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