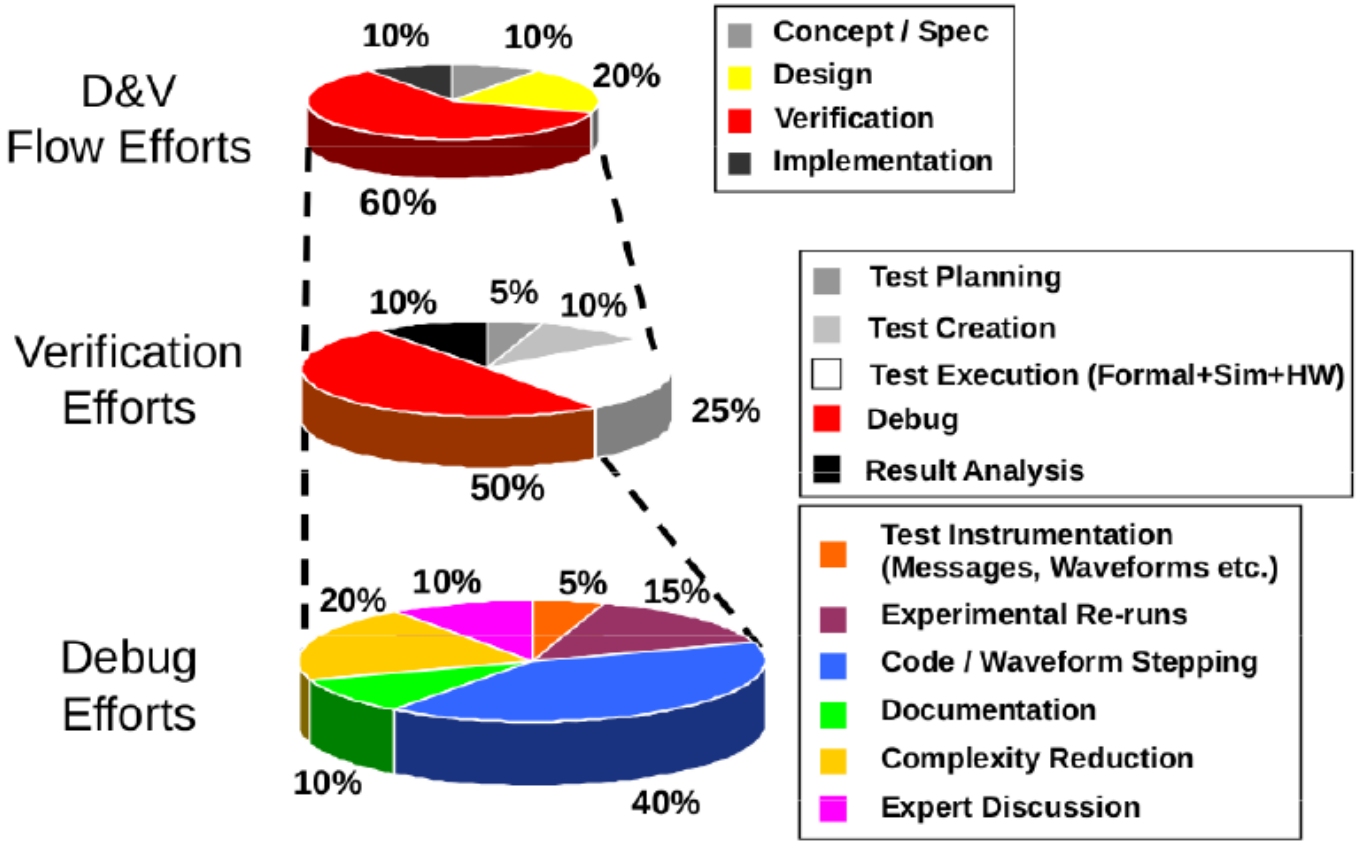
Chip Performance: Testing, Verification and Validation

Tassadaq Hussain
Professor Namal University

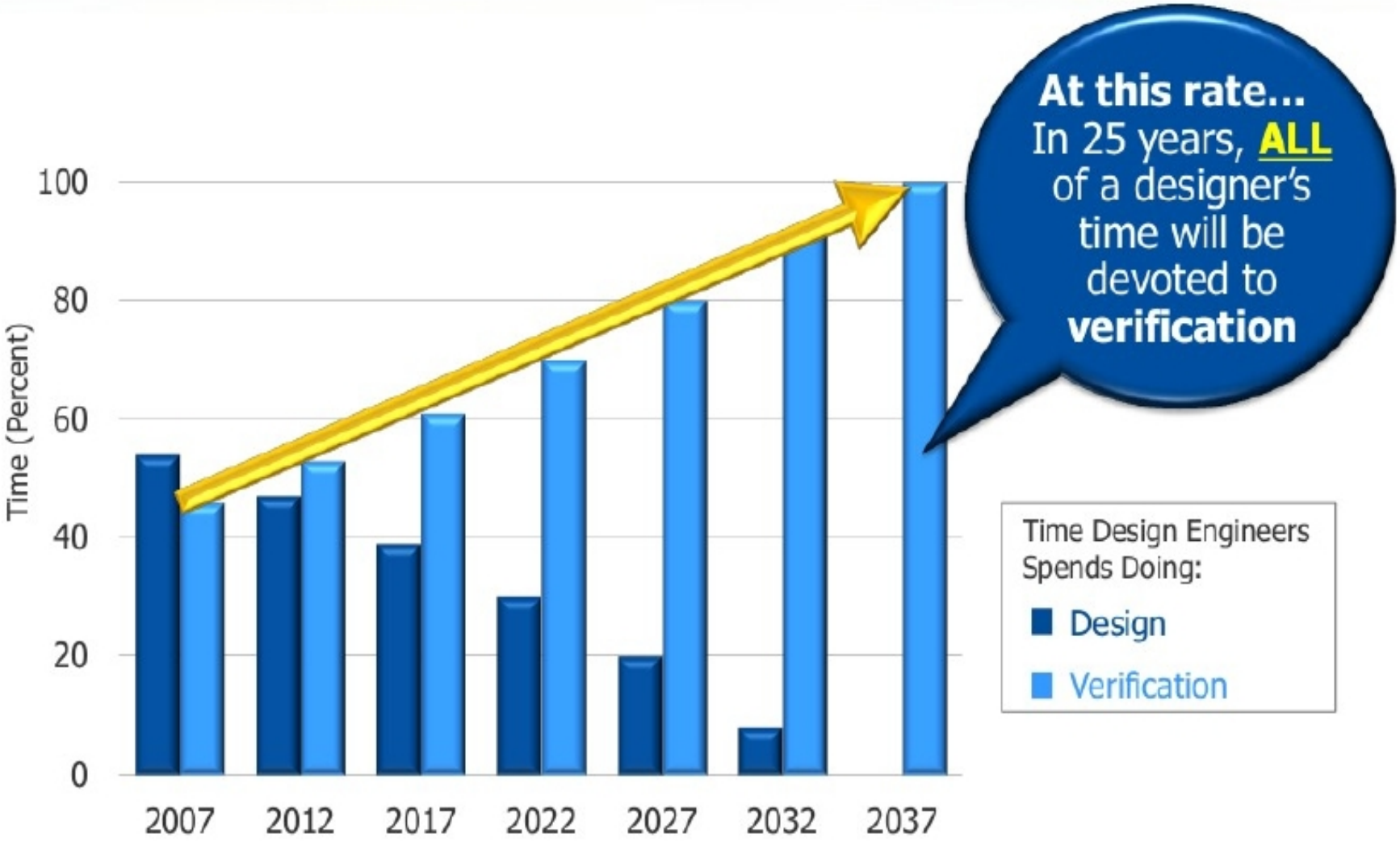
Outline

- **Importance of Verification**
- Hardware Software Testing
- Functional Verification
- Formal Verification
- Assertion Based Verification

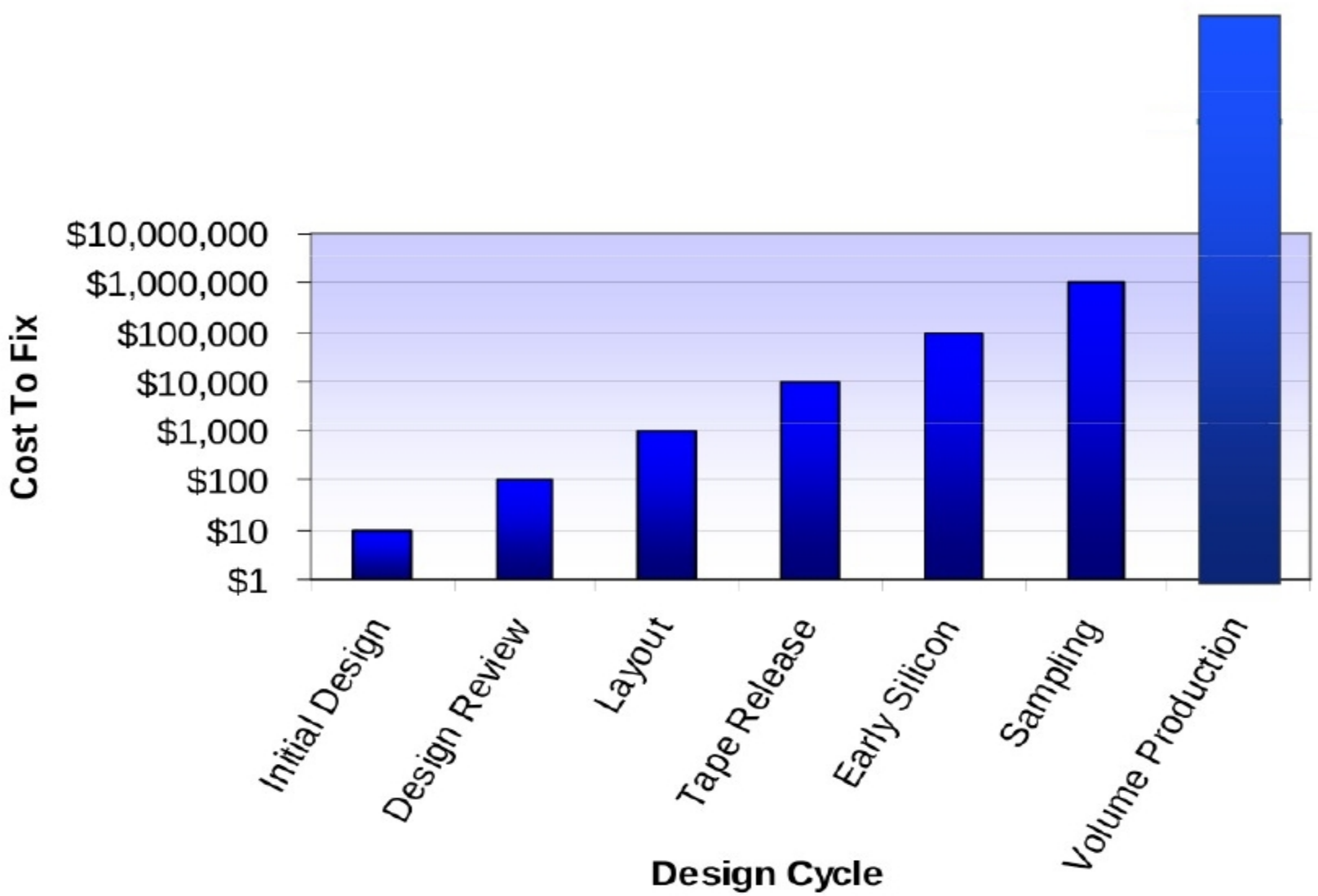
Design, Verification and Implementation



Industry Pulse: Trends in Functional Verification



Late-Stage Bugs Are More Costly



Design for Test (DFT):

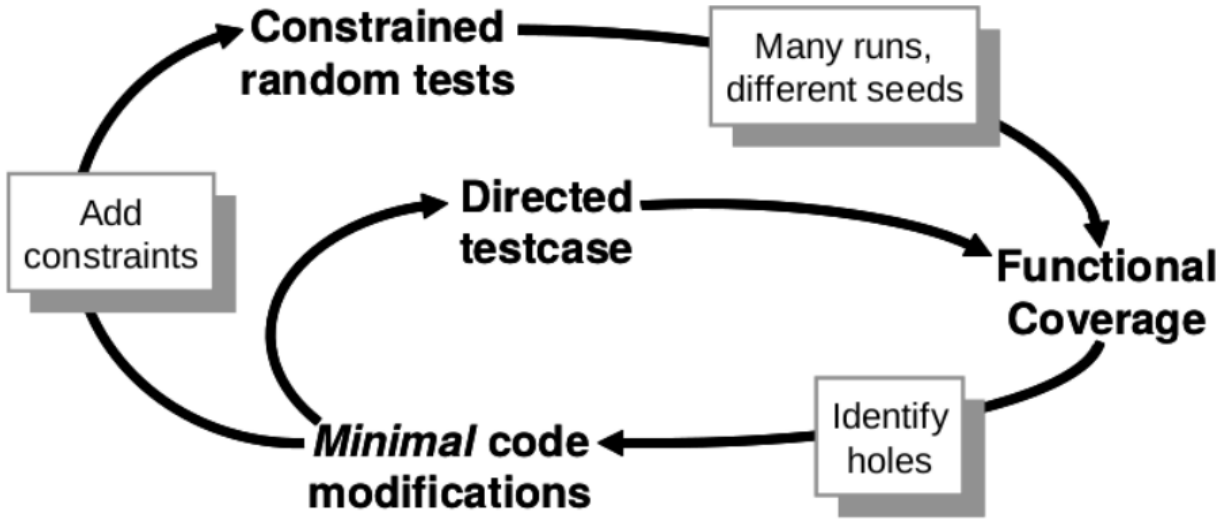
- Design for Test (DFT) is an approach or methodology used during the design phase to enhance the testability of the integrated circuit.
- It employed to facilitate the testing process by incorporating specific design features that enable easier and more efficient testing of the DUT. The primary goal of DFT is to improve test coverage, reduce test time, and minimize the cost of testing.
- It scan applies chains or boundary scan (JTAG) to enable the efficient shifting and observation of test patterns into the circuit.
- DFT may involve breaking down the design into smaller partitions that can be tested independently. This allows for parallel testing and faster fault localization.

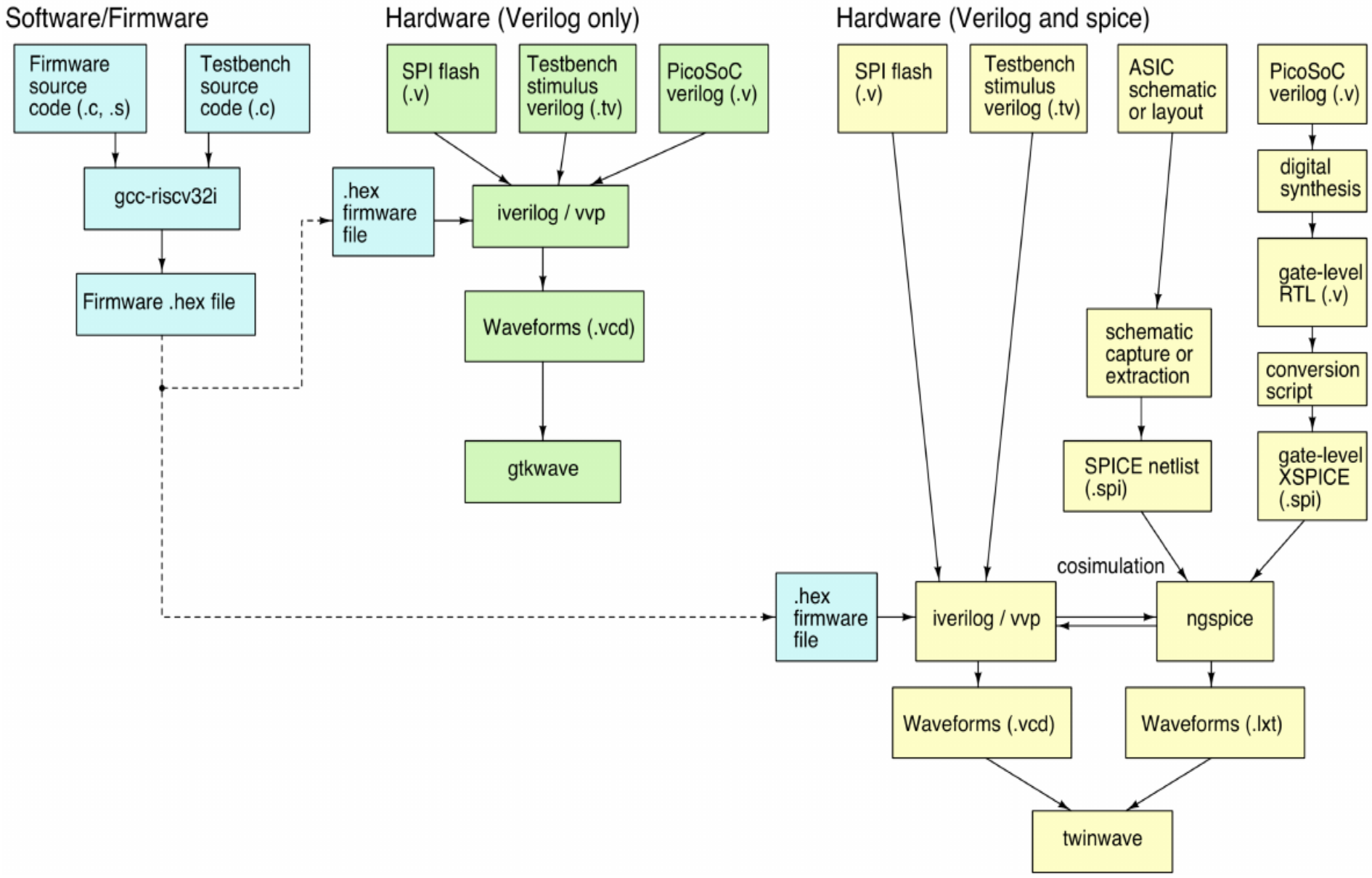
Design Under Test (DUT)

- The Design Under Test (DUT) refers to the actual electronic design or integrated circuit that needs to be tested.
- It represents the target design that is being evaluated for its functionality, performance, and adherence to specifications. The DUT is the device or system that undergoes testing to ensure its proper operation and to identify any defects or errors.

Coverage

- Measures which design features have been exercised
- Code coverage metrics:
  - e.g. line, path, toggle, FSM
- Functional coverage
  - e.g. Ensuring that all possible interactions with DUT have been tested




Software/Firmware
Hardware (Verilog only)
Hardware (Verilog and spice)
Firmware source code (.c, .s)
Testbench source code (.c)
gcc-riscv32i
Firmware .hex file
SPI flash (.v)
Testbench stimulus verilog (.tv)
PicoSoC verilog (.v)
.hex firmware file
iverilog / vvp
Waveforms (.vcd)
gtkwave
SPI flash (.v)
Testbench stimulus verilog (.tv)
ASIC schematic or layout
PicoSoC verilog (.v)
digital synthesis
gate-level RTL (.v)
conversion script
gate-level XSPICE (.spi)
schematic capture or extraction
SPICE netlist (.spi)
cosimulation
.hex firmware file
iverilog / vvp
ngspice
Waveforms (.vcd)
Waveforms (.lxt)
twinwave

Outline

- Importance of Verification
- **Hardware Software Testing**
- Functional Verification
- Assertion Based Verification
- Formal Verification

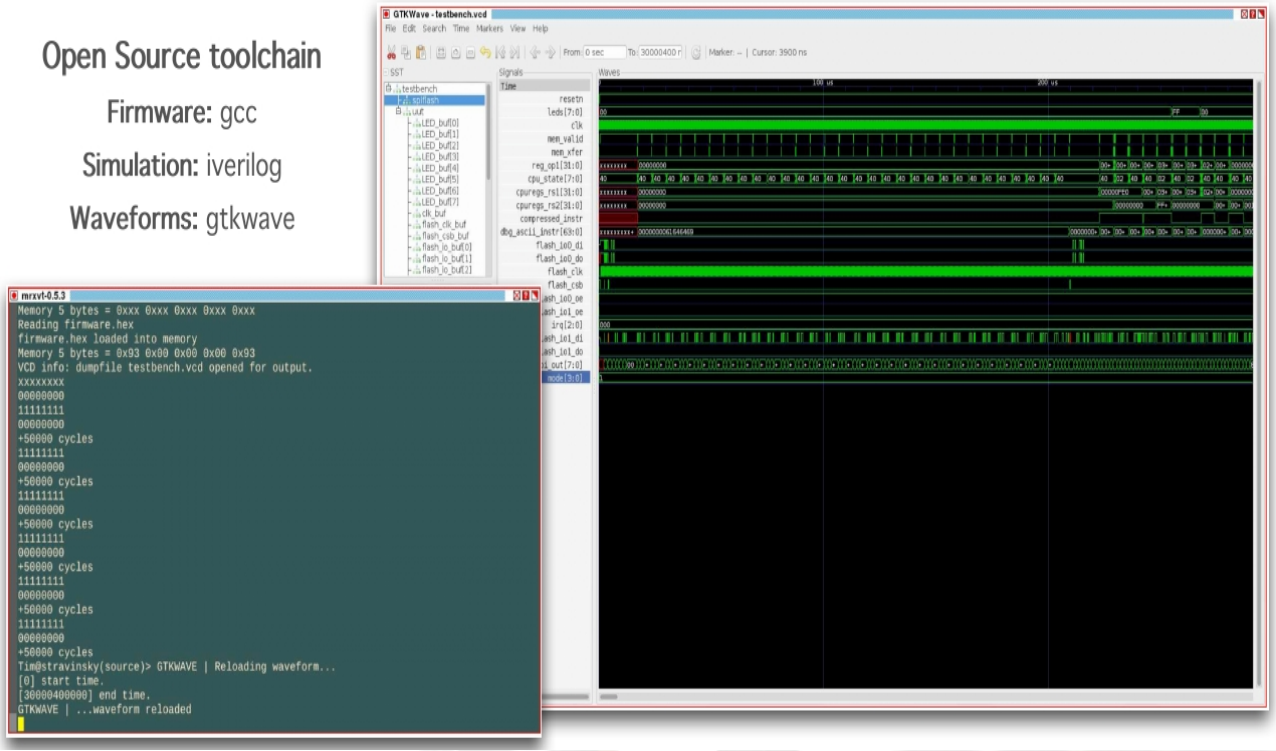
Emulation

- Emulation is a technique used to replicate the behavior of a target hardware system, allowing the execution of both hardware and software for testing purposes.

- Cycle-Accurate Simulation
- FPGA-Based Emulation

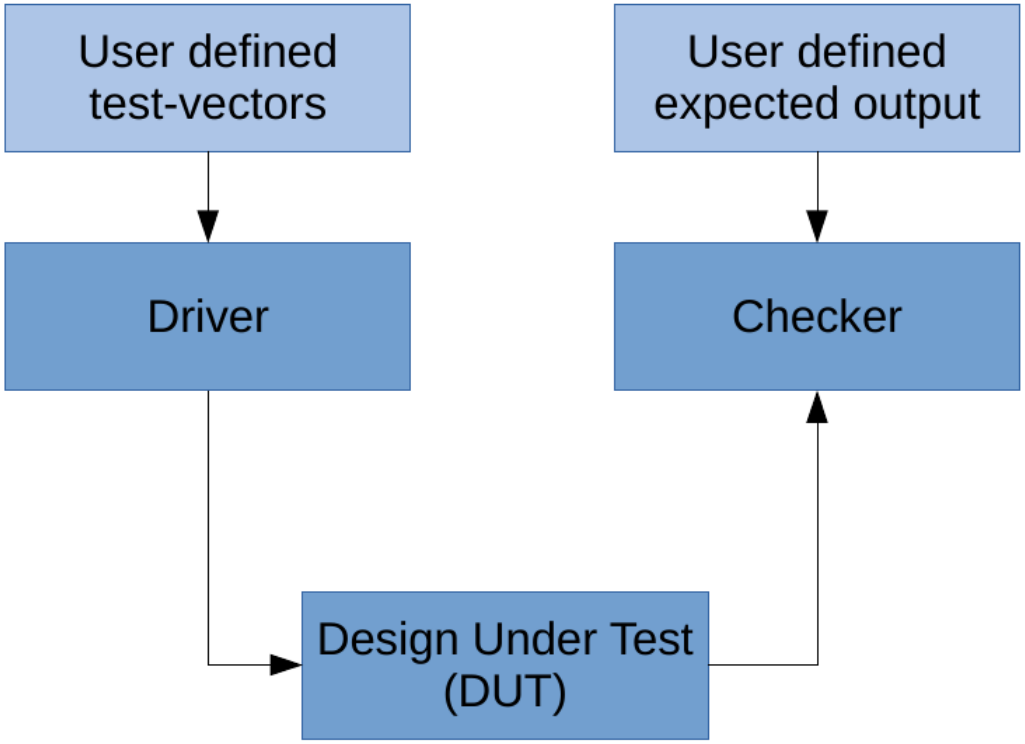
Software - Hardware Co-Simulation

- **Commercial Simulator**
  - Synopsys VCS
  - Mentor Graphic's ModelSim
- **Open Source**
  - Icarus Verilog
  - GHDL
  - Verilator
  - Yosys

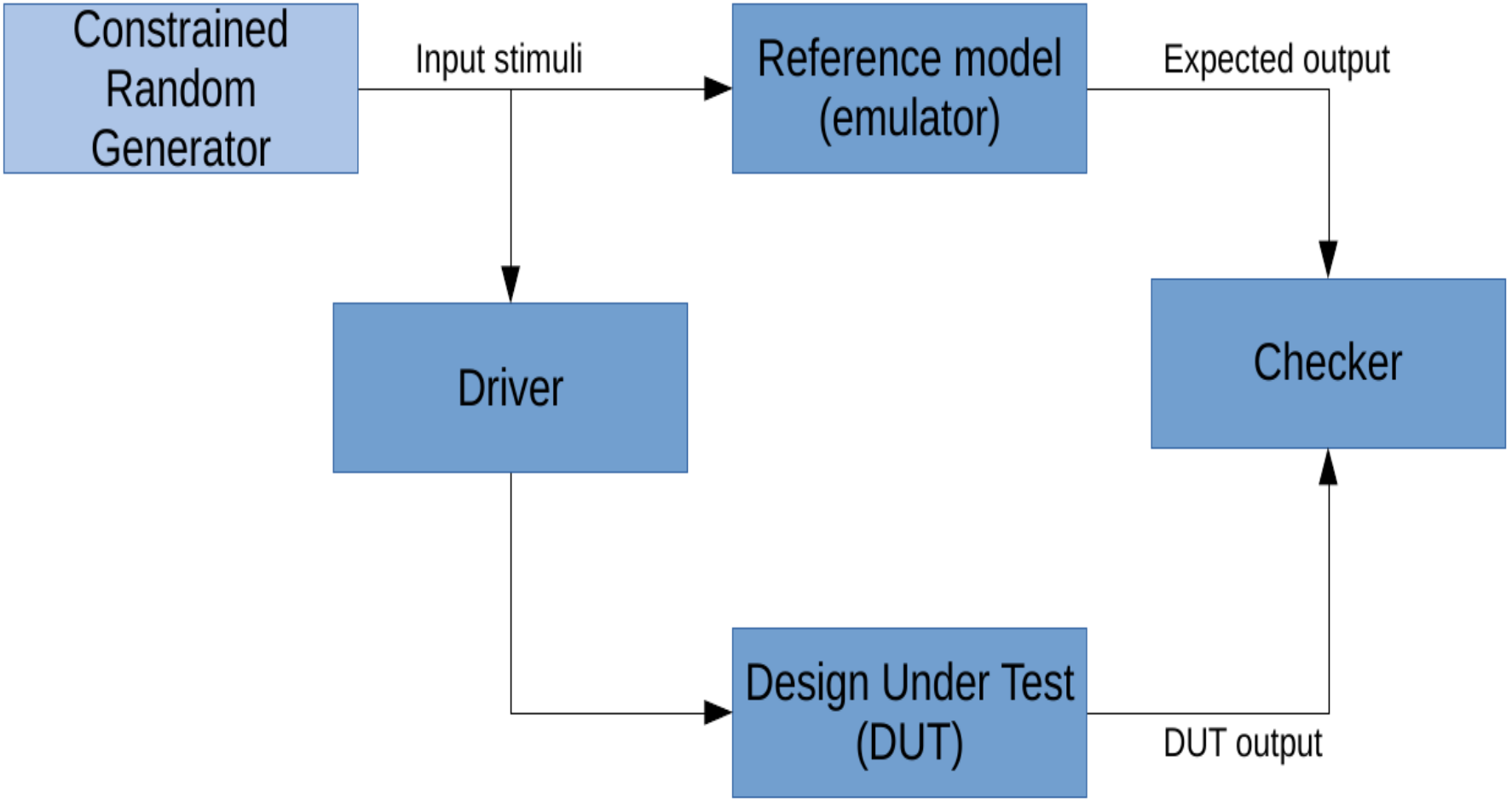


Test Cases (Directed)

- Pre-defined stimulus to test particular functionality
- Testbench implements specs to check outputs
- Very labour intensive
- Hard to model all cases



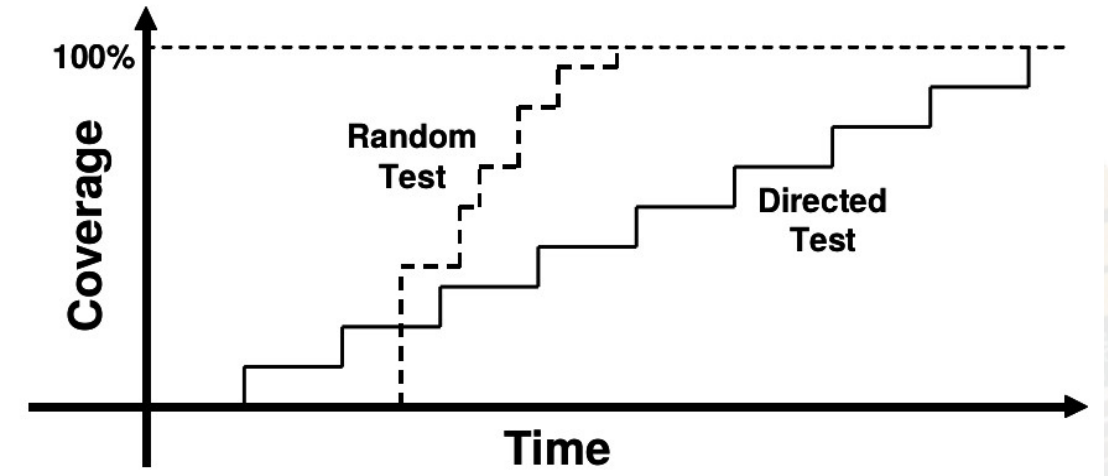
Constrained Random Verification



Which functionality has been tested?

Randomizing Testing Data

- Input data
- Device configuration
- Delays
- Transaction length



Outline

- Importance of Verification
- Hardware Software Testing
- **Functional Verification**
- Assertion Based Verification
- Formal Verification

Functional Verification: TestBench

- A “testbench” is the code used to create determined input sequence to a design and then observe the response
  - Written in VHDL, Verilog, C++, Python...
- Verification challenge:
  - Find what input patterns to supply to Design Under Test and what is expected for the output of a properly working design

CocoTB

- **Library for digital logic verification in Python**

– Coroutine co-simulation testbench

– Python interface to RTL simulators

– Human-friendly alternative to Verilog/VHDL

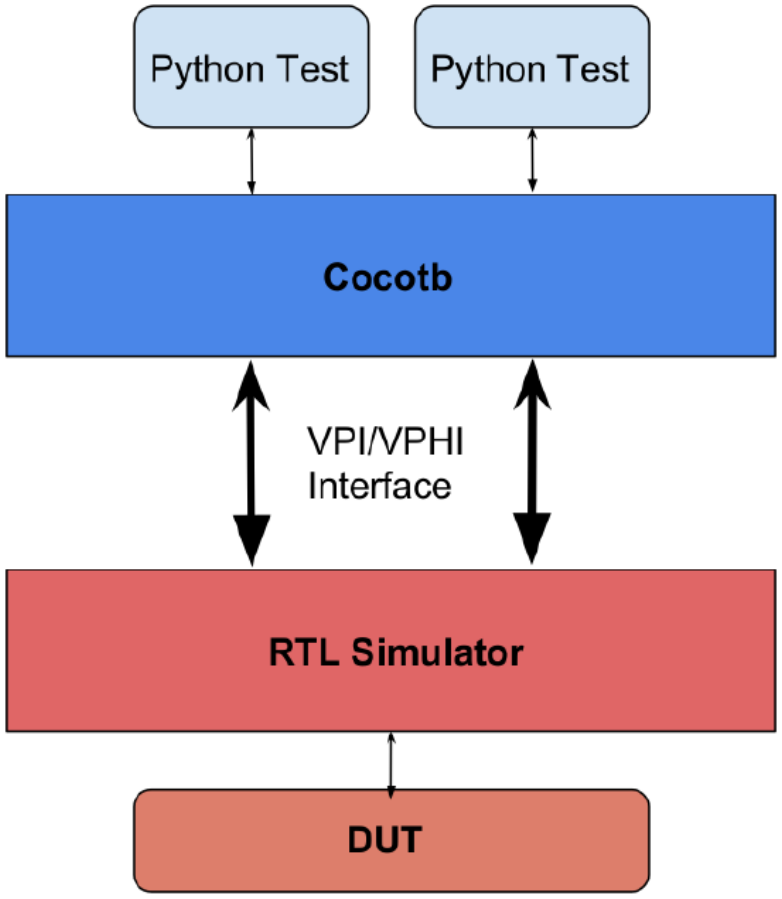
– Verification testbenches are software!

CocoTB

- Benefits of Python
  - Extremely simple and easy to learn, but very powerful
  - Large standard library and huge ecosystem
  - Easy to find Python developers
    - https://github.com/cocotb/cocotb

Cocotb Basic Arch

- DUT runs in standard simulator
- Cocotb provides interface between simulator and Python
- Python testbench code can:
  - Reach into DUT hierarchy and change values
  - Wait for simulation time to pass
  - Wait for a rising or falling edge of a signal



Examples

Verilog code (add.v)

```
module add(input wire [7:0] a,
           input wire [7:0] b,
           output wire [8:0] c);
    assign c = a + b;
endmodule
```

Python code (add_test.py)

```
import cocotb
from cocotb.triggers import Timer

@cocotb.test()
def add_test(dut):
    dut.a <= 3
    dut.b <= 5
    yield Timer(1, "ns")
    assert dut.c == 8
```

Makefile

```
VERILOG_SOURCES = add.v
TOPLEVEL=add
MODULE=add_test
include $(shell cocotb-config --makefiles)/Makefile.inc
include $(shell cocotb-config --makefiles)/Makefile.sim
```

Examples

```
Running tests with Cocotb v1.2.0rc1 from /usr/local/lib/python2.7/dist-packages
Seeding Python random module with 1569503113
Found test add_test.add_test
Running test 1/1: add_test
Starting test: "add_test"
Description: None
Test Passed: add_test
Passed 1 tests (0 skipped)
*******************************************************************************
** TEST                 PASS/FAIL  SIM TIME(NS)  REAL TIME(S)  RATIO(NS/S) **
*******************************************************************************
** add_test.add_test    PASS            1.00          0.00      1017.32  **
*******************************************************************************

*******************************************************************************************
**                                    ERRORS : 0                                         **
*******************************************************************************************
**                                  SIM TIME : 1.00 NS                                   **
**                                 REAL TIME : 0.00 S                                    **
**                           SIM / REAL TIME : 533.89 NS/S                               **
*******************************************************************************************

Shutting down...
```

Cocotb Triggers

- Co-simulation
  - DUT and testbench simulated independently
  - Communication through VPI/VHPI interfaces, represented by cocotb “triggers”
  - When Python code is executing, simulation time is not advancing
  - When a trigger is yielded, testbench waits until condition is satisfied before resuming execution

Cocotb Triggers

Some available triggers:

- Timer(time, unit)
- RisingEdge(signal)
- ClockCycles(signal, num)

Coroutines

- Cocotb uses a cooperative multitasking architecture
- Tests can call functions
  - If they want to consume simulation time, must be **coroutines**

```
import cocotb
from cocotb.triggers import Timer

@cocotb.coroutine
def Driver(signal1, signal2, v1, v2):
    signal1 <= v1
    signal2 <= v2
    yield Timer(1, "ns")

@cocotb.coroutine
def Checker(signal, expected):
    yield Timer(1, "ns")
    assert signal == expected

@cocotb.test()
def add_test(dut):
    cocotb.fork(Driver(dut.a, dut.b, 3, 5))
    cocotb.fork(Checker(dut.c, 8))
    yield Timer(1, "ns")
```

Verilator

- Fastest free Verilog simulator
  - Suitable for large designs
  - Compiles Verilog to C++
  - Multithreaded since 4.0
- Support for synthesizable SystemVerilog
- Testbenches written in C++
- https://www.veripool.org/wiki/verilator

Verilator - Example

- Translate to C++ class
  - **verilator -cc Convert.v**

Convert.v

```
module Convert;
   input clk
   input [31:0] data;
   output [31:0] out;

   initial $display("Hello flip-flop");
   always_ff @ (posedge clk)
      out <= data;
endmodule
```


obj_dir/VConvert.h (made by Verilator)

```
#include "verilated.h"

class VConvert {
   bool      clk;
   uint32_t  data;
   uint32_t  out;

   void eval();
   void final();
}
```

DotProd - Verilator

```
`timescale 1ns/1ps

`default_nettype none

module dotprod(input wire clk,
               input wire rst,
               input wire [7:0] Ai,
               input wire [7:0] Bi,
               output wire [25:0] out);

    reg [25:0] accum;

    always @(posedge clk)
    begin
        if (rst) accum <= 26'd0;
        else     accum <= accum + Ai * Bi;
    end

    assign out = accum;

endmodule // dotprod
```

DotProd Testbench - Verilator

```cpp
#include "Vdotprod.h"
#include "verilated.h"

// Generates two random vectors A and B
struct RandomVectors {
    vector<unsigned char> A;
    vector<unsigned char> B;

    RandomVectors(int N) : A(N), B(N) {
        generate(A.begin(), A.end(), rand);
        generate(B.begin(), B.end(), rand);
    }
};

// Control coverage
struct SampleCoverage {
    vector<unsigned int> bins;

    SampleCoverage() : bins(1025, 0) {}

    void sample(int N) { bins[N]++; }

    void printCoverage(ostream& out) {
        int bins_hit = 0;
        for (size_t i = 0; i < bins.size(); i++)
            if (bins[i])
                bins_hit++;
        out << "Bins covered: " << bins_hit << " / " << 1024 << endl;
        out << "Coverage = " << bins_hit / 1024.0f * 100.0f << "%" << endl;
    }
};
```

```cpp
// Advance one cycle of RTL simulation
void runCycle(Vdotprod *dut, int halfCycle=5) {
    dut->clk = 1;
    for (int i = 0; i < halfCycle; i++)
        dut->eval();

    dut->clk = 0;
    for (int i = 0; i < halfCycle; i++)
        dut->eval();
}
```

Outline

- Importance of Verification
- Hardware Software Testing
- Functional Verification
- **Assertion Based Verification**
- Formal Verification

What is: Assertions

- Capture the knowledge about how a design should operate
- Increase the observability of a design
- Each assertion specifies both legal and illegal behavior of a circuit structure inside the design
- Assertions in English:
  - The result should not overflow
  - Signal “a” must be less than...

HDL Assertion

- An assertion is a formal statement or condition that specifies a desired property or behavior of a digital design.
- Assertions are used for formal verification and serve as a means of specifying and checking design requirements, correctness properties, and constraints.
- They play a crucial role in ensuring that a digital design operates as intended. Here are key aspects of assertions in HDLs:

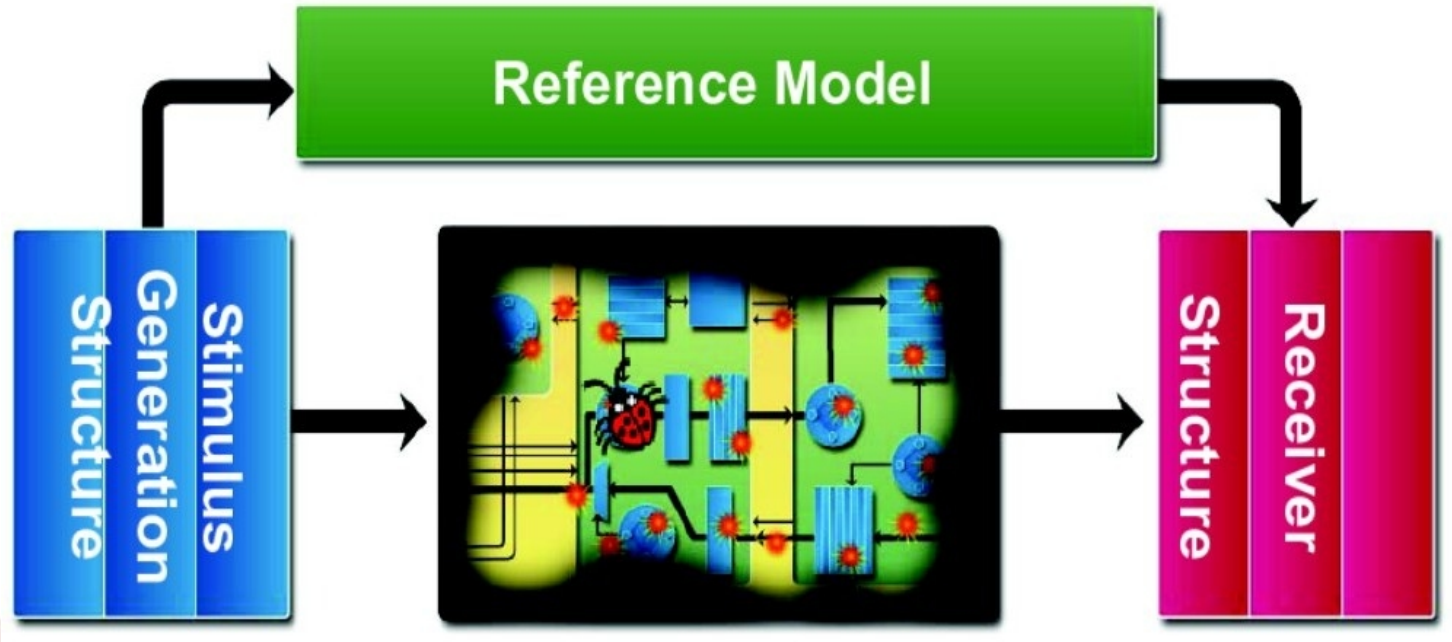
Open Assertion Library

- Library of assertions across many HDL languages
  - Verilog, VHDL, SystemVerilog...
- Standardized, supported by major simulators
- Support common assertion structures using module/parameter instantiations

```
ovl_never #(`OVL_ERROR, `OVL_ASSERT, "Register A < Register B",`OVL_COVER_ALL)
                                    valid_checker_inst(clk, reset_n, regA < regB )
```

Assertion Based Verification

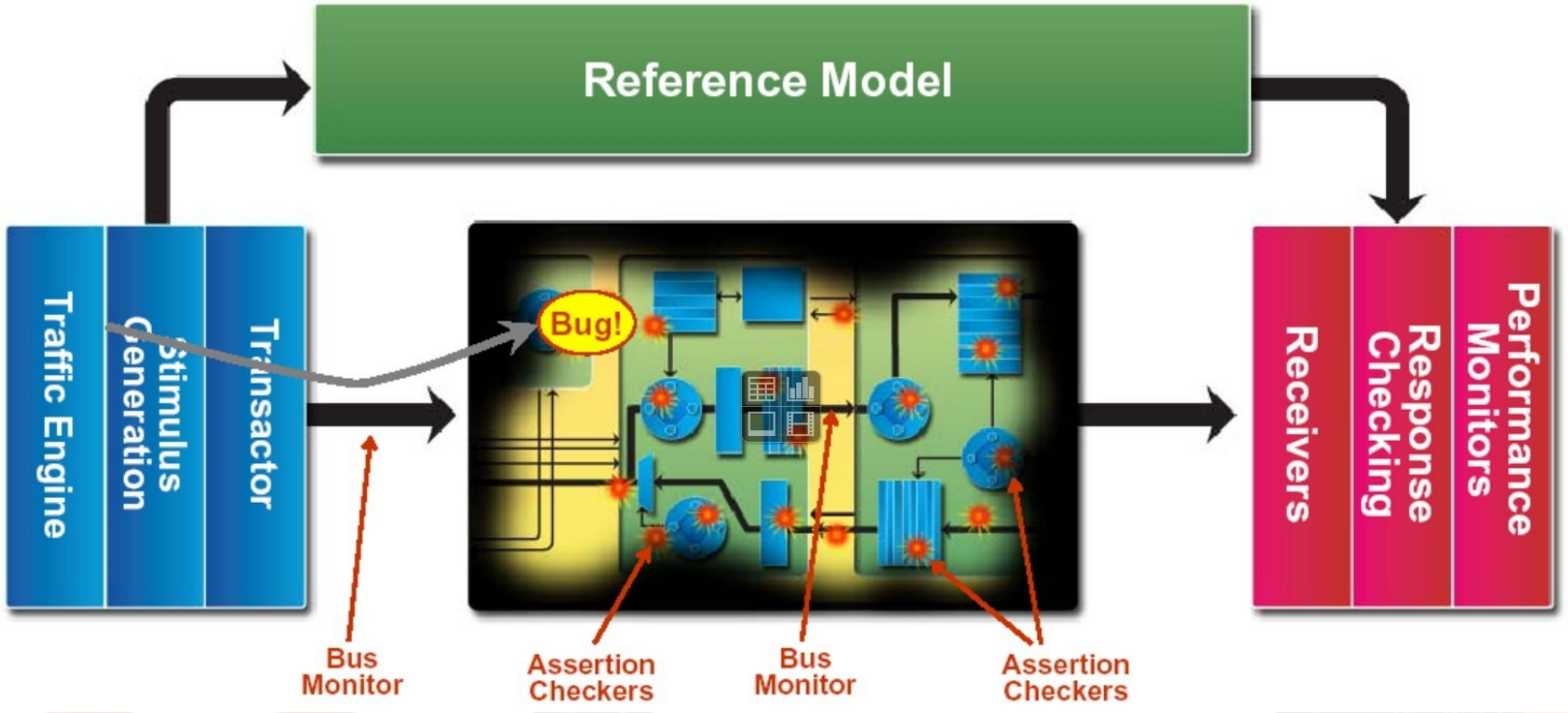
- Methodology to detect bugs quickly and close to source
- Increased observability



Mentor Graphics. “Assertion Based Verification ”

Assertion Based Verification

- Observe bug when propagated to output
- Catch bugs at or near source of the



Mentor Graphics. “Assertion Based Verification”

SystemVerilog Assertions (SVA)

- Powerful language to describe assertions
- Two types of assertions
  - Immediate assertions
  - Concurrent assertions
- Perform test on an aspect of the design
  - Pass or fail statements can be executed
- Typically ignored during synthesis
- Partial SVA support in Verilator

Outline

- Importance of Verification
- Hardware Software Testing
- Functional Verification
- Assertion Based Verification
- **Formal Verification**

Formal Verification

- Can “Prove” correctness about design blocks
- Exhaustively check state-space of a design for
- Violations of properties (assertions) e.g. “error signal should never be high”
- Does not execute design, proves property mathematically
- Good at finding corner case behaviors
- Requires high-effort, specialized knowledge
- May be impractical for large designs

When is Verification Done?

- Never truly done on complex designs
- Functional verification can only show presence of errors, not their absence
- Given enough time, errors will be uncovered