

Design for Testability

Introduction to testing
integrated circuit designs

Michael Rathmair

2 lectures about testing circuits

■ PART 1: What to test


- Why is testing so important
- Which faults can happen
- Generation of test patterns

■ PART 2: How to test

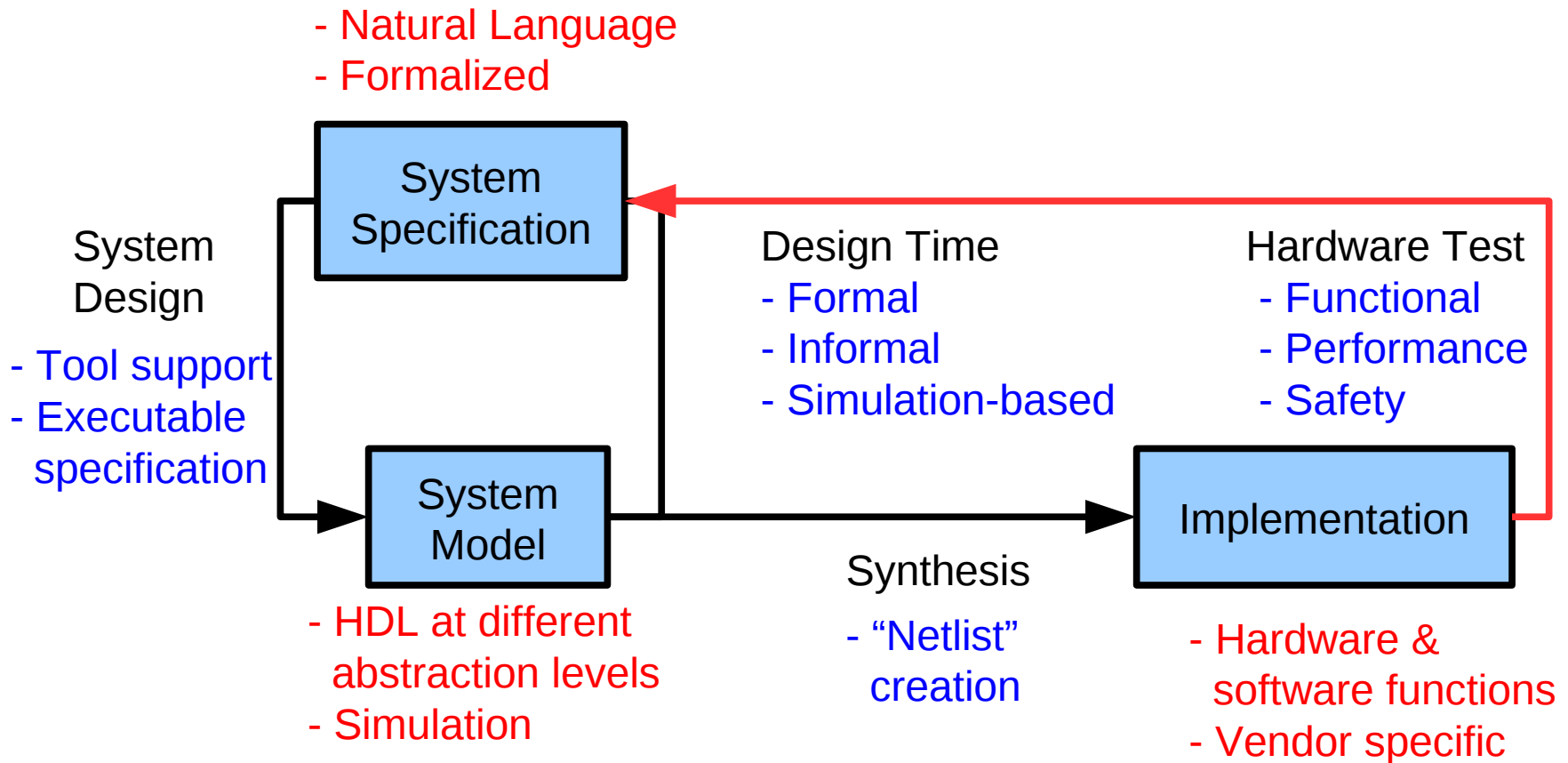
- How to test a design
- How good is my test
- How to test hardware



Introduction

- Reliability vs. Availability and Safety vs. Security
- Hardware bug fixes after delivery are impossible
- 50% of all ASICs do not work after initial design and fabrication [Keut91] 
- Increased effort in a guarantee that the design is correct
- Goal is “first time right”

Overview – Design Flow



Bed of nails testing

- Nail probes contact specified testpads during operation

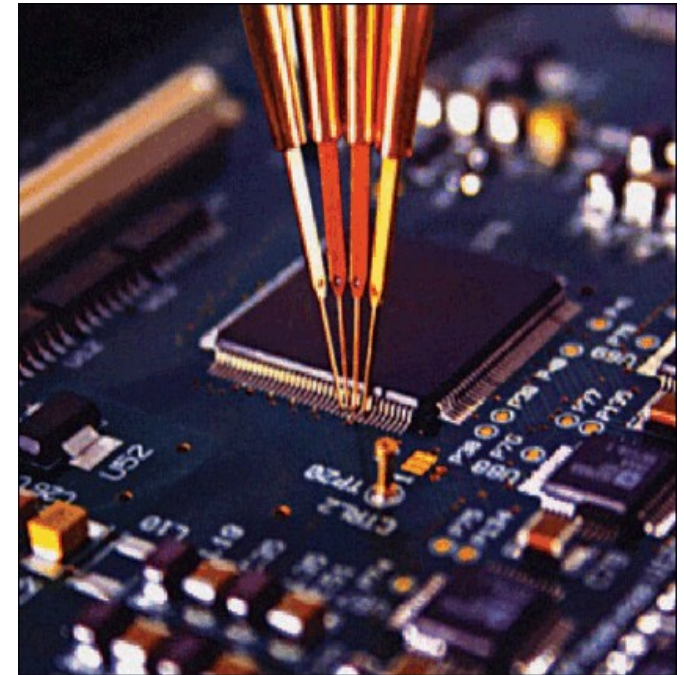
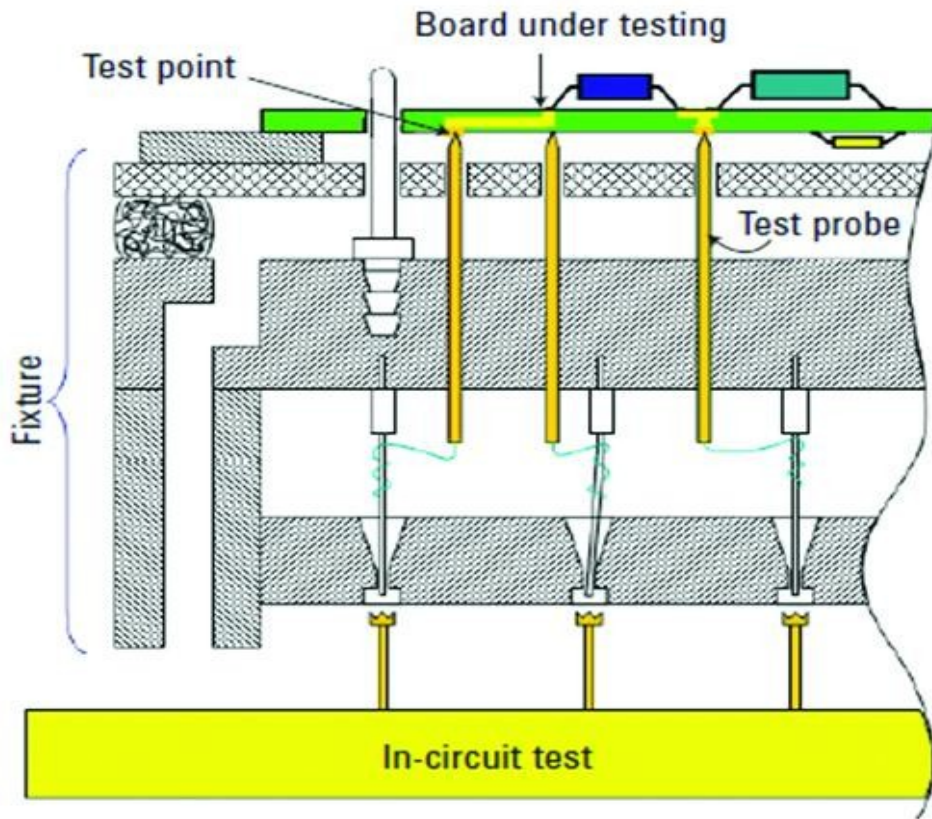
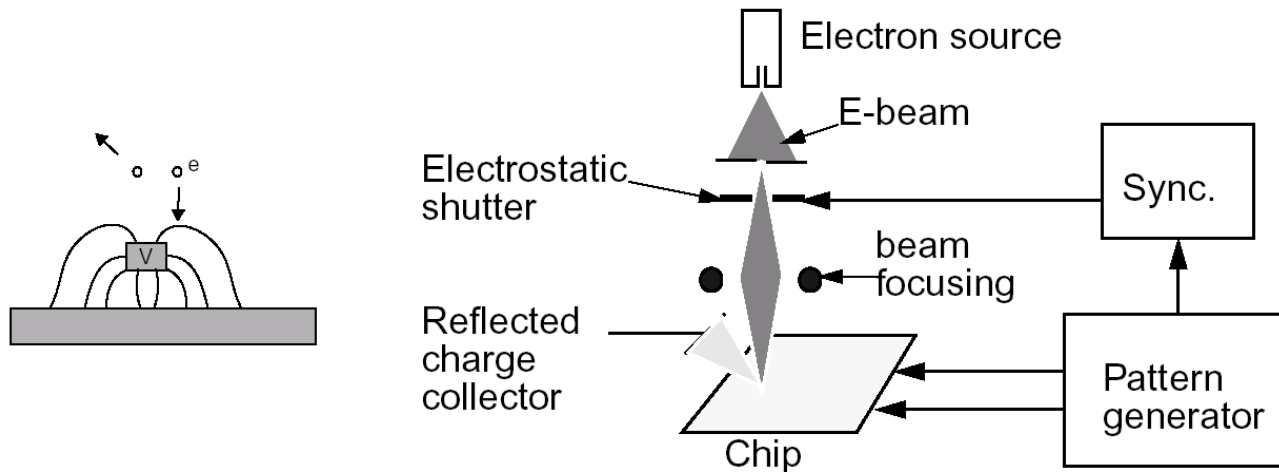


Figure 3-2: The probes are moved with three-axis, high powered linear motors to achieve electrical contact with the component under test. Photo courtesy of SPEA.

E- Beam testing

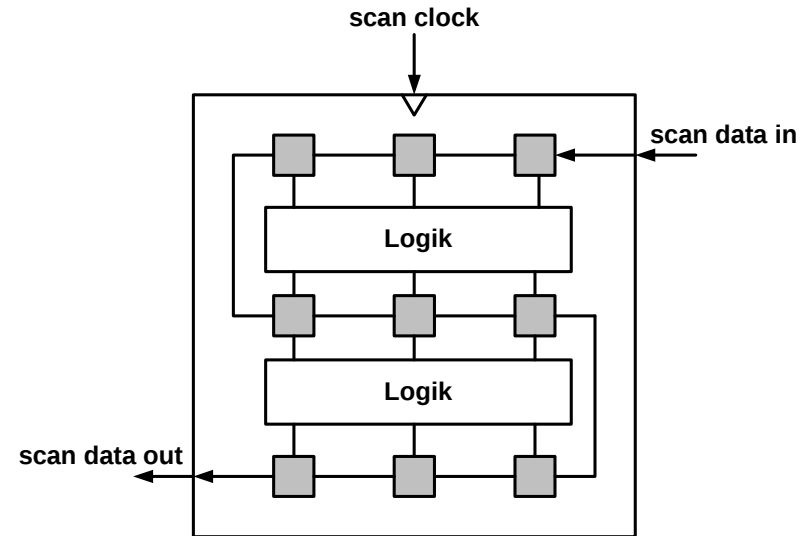
- The reflected component of an e-beam on the chip surface is a function of the surface voltage potential



- A single area/point on the chip can be observed at a high temporal accuracy ($\sim 100\text{ps}$)
- A complete scan of the chip results in a voltage contrast image at a dedicated point in time

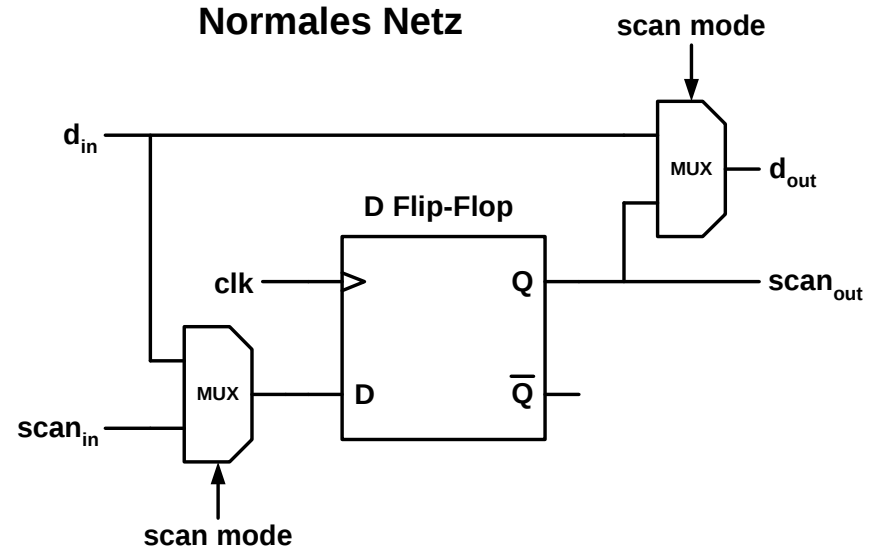
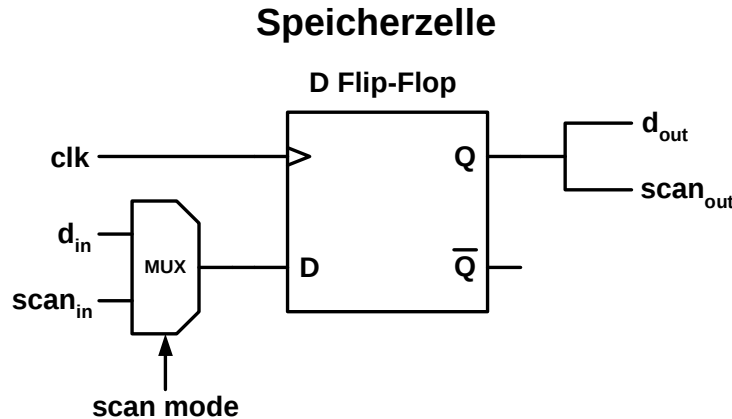
Scan Path Testing

- For increased observability, internal memory elements (Flip-flops) are chained to a serial shift register.
- Testsequence:
 - 1) Turn on scan mode
 - 2) Serial load of initial testdata (enhanced controllability)
 - 3) System operation for n cycles
 - 4) Serial read of results
- Advantages:
 - Single testvectors can be applied and due to enhanced controllability and observability testcases can be generated automatically (ATPG).
 - Existing tools (chip design) for integration of the scan path test technique into chip designs.



Scan Path Testing

■ Scan path cell structure:



■ Disadvantages:

- Requires additional hardware (chip area)

For example:

- 20k Gates and 500 FF
- 1 scan FF = 10 Gates → 500 FF = 5000 Gates
- Overhead ~ 5%
- Decrease of the maximum clock frequency



Boundry scan

- IEEE 1149 Standard (JTAG Joint Test Action Group)
- Enables testability of interconnections between chips on a PCB
- Testability for internal chip interconnections SoC
- Unique chip ID
- Today JTAG is a standard equipment of integrated circuits
 - Processors
 - SoC FPGA
 - PCI Bridges
 - Memories
 - Network controllers (Ethernet, DSL, ...)
 - ...



Boundary scan

■ Standardized interface

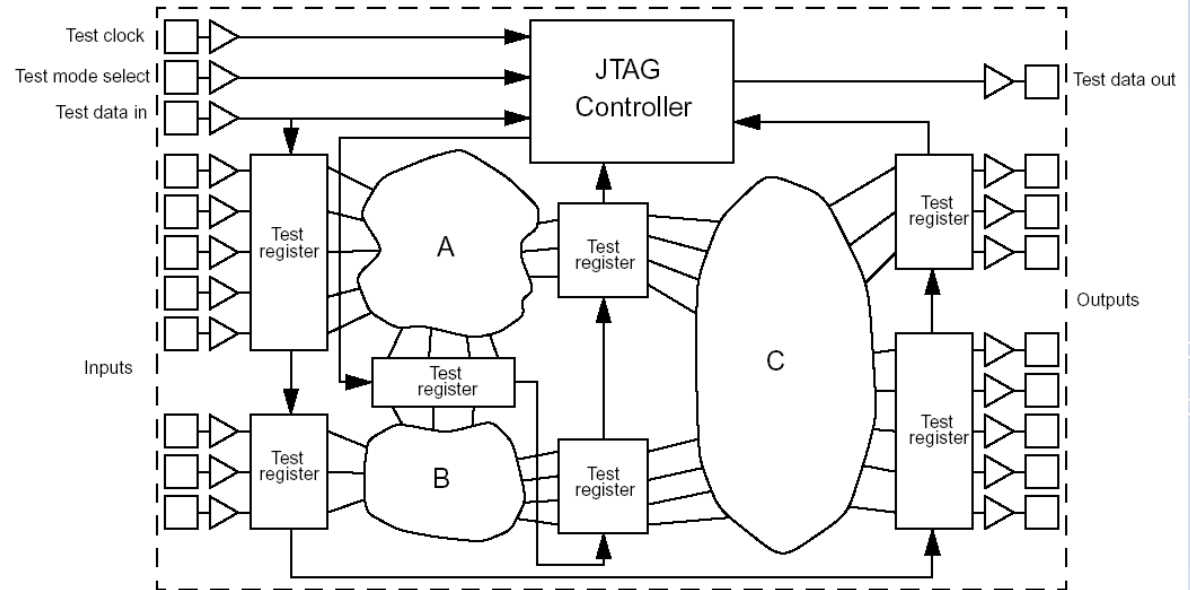
TCK – Test clock

TMS – Test mode select

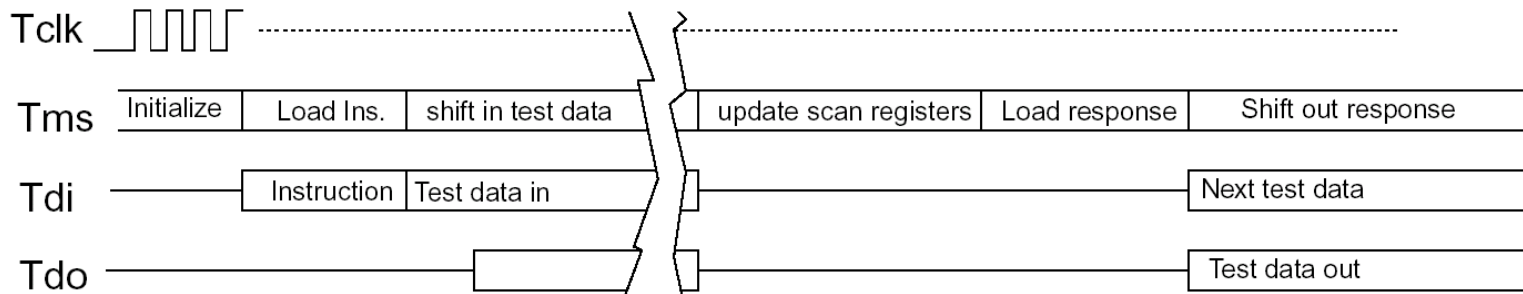
TDI – Test data in

TDO – Test data out

TRST* - Test reset (optional)

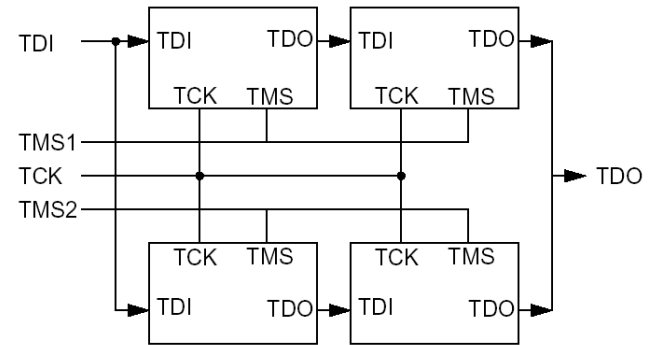
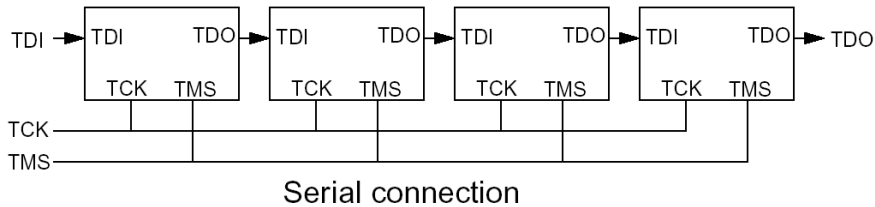


■ JTAG Protocol



Boundry scan

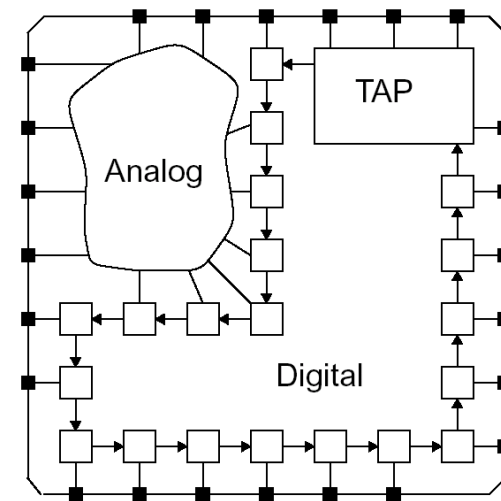
■ Connection of JTAG devices



Hybrid serial/parallel connection

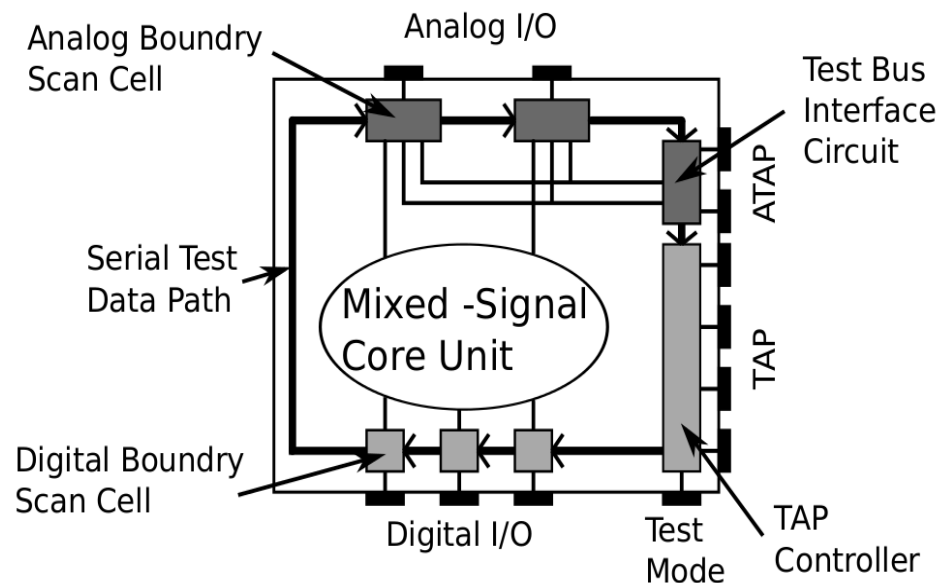
■ Mixed mode JTAG

- Registers are placed between analog and digital parts
- In case of very fast signal interconnections, signals may be routed to external ports



Mixed signal JTAG

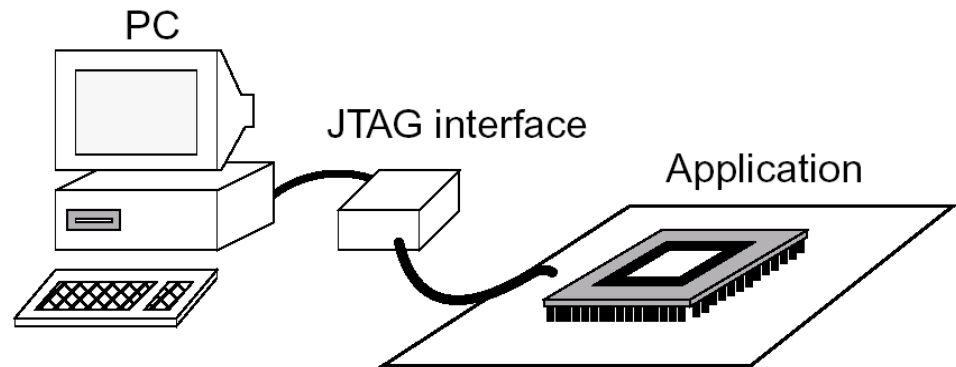
- Mixed signal testing – IEEE 1194.4
 - Boundary scan architecture extension for analog signals
 - ATAP (analog test access port)
 - Analog buses limit frequency bandwidth of the test process



Boundry scan

■ JTAG Testsystems

- PC based JTAG test interfaces
- Software support for analysis, debugging, fault location

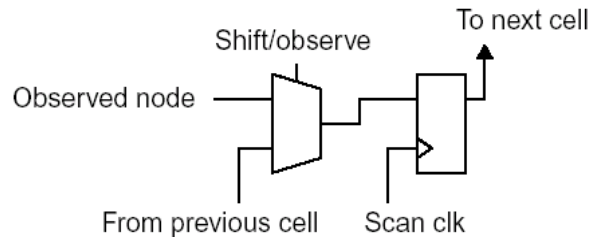


■ Alternative JTAG usage

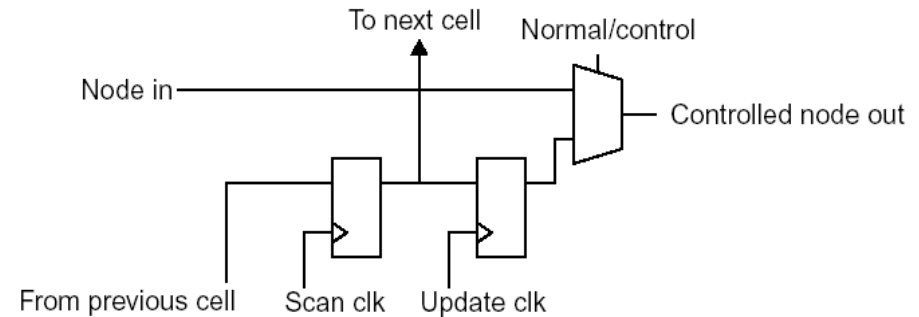
- User programming of components (JTAG software/firmware download) For example: Microcontroller, FPGA, CPLD, etc.
- Monitoring of internal functionalities during operation
- Debugging of processors (Register/Variable access, breakpoints, step into/over functions, etc.)
- Hardware emulation

JTAG scan cells structure

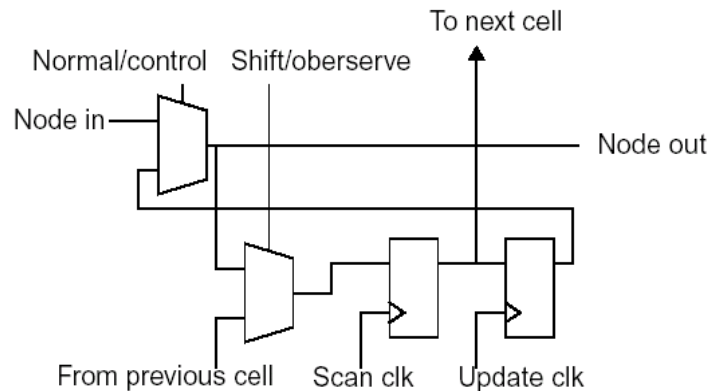
Observing scan cell



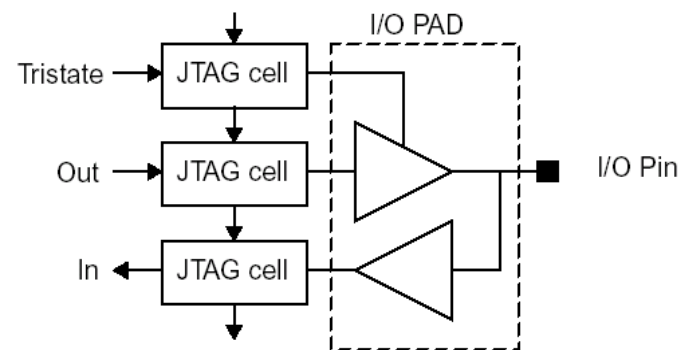
Controlling scan cell



Observing and controlling scan cell



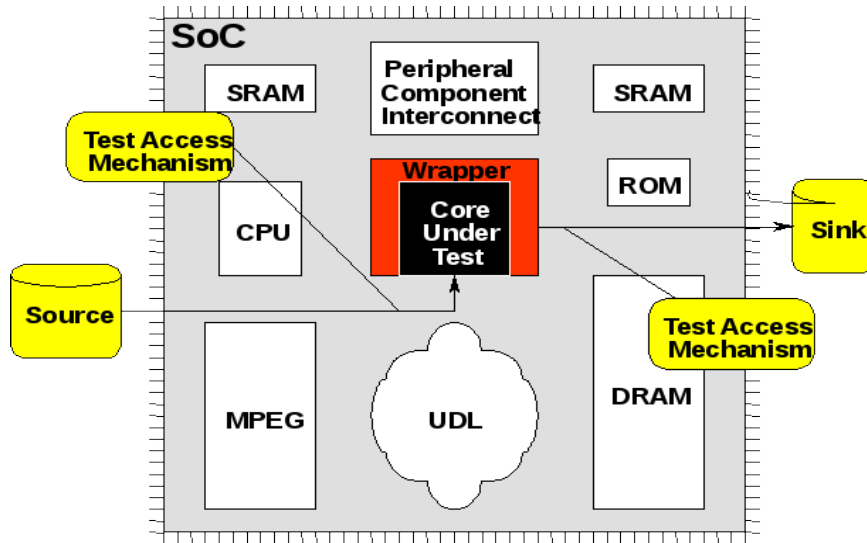
JTAG cells required for Input/Output pin



Built-In self test - BIST

- Described methods are not well suited for testing memory devices.
 - For example: 100MHz Test equipment $2N^2$ testvectors:
64k \rightarrow 1.4 min, 256k \rightarrow 23 min, 1M \rightarrow 6 h
- Also large and complex ASIC devices require increased time and effort for test sequences.
- Solution: BIST (built in self test)
- Offline and online BIST
 - For offline BIST the normal functionality of the chip is interrupted.
 - If a component is in standby due to redundancy it may perform self tests.
 - At online BIST sequences the chip is tested during operation.
 - e.g. Each 10th input pattern is a test sequence.

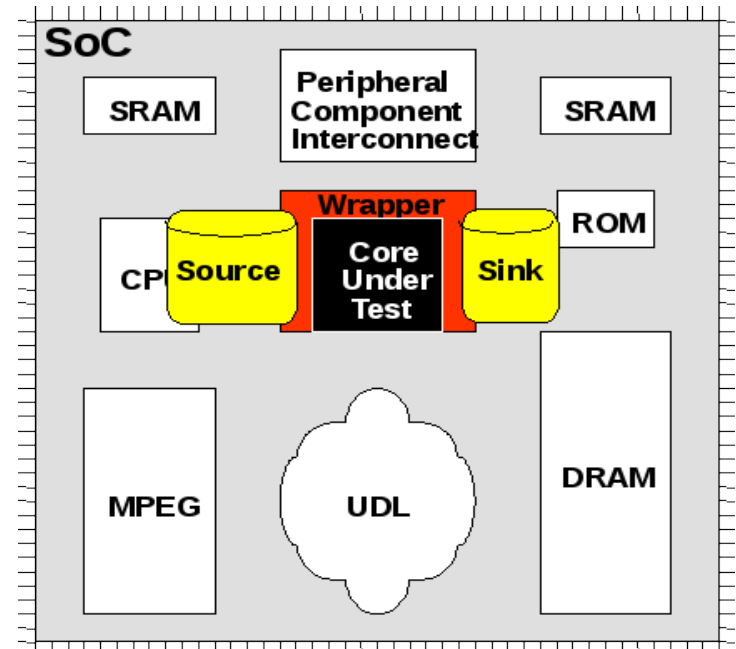
Motivation for BIST



- Need for cost efficient testing
- Increasing difficulties with TPG
- Growing volume of test pattern data
- Cost of ATE
- Gap between tester and DUT speeds

■ Drawbacks

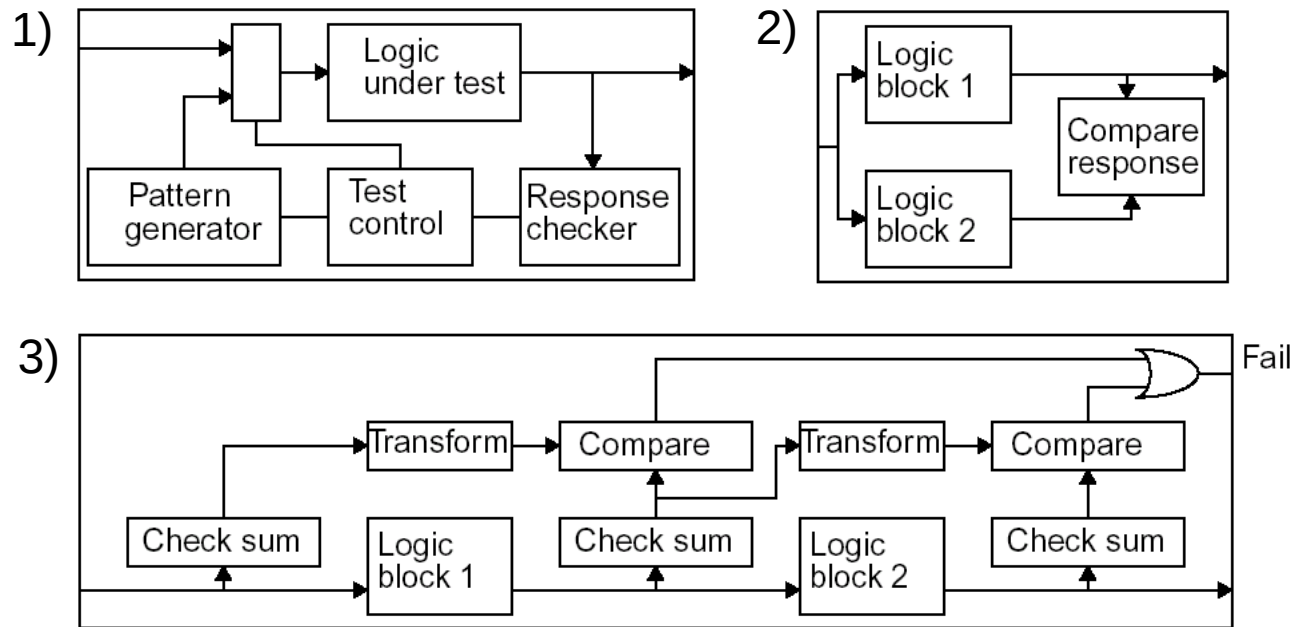
- Additional silicon area
- Decreased reliability due to area increase
- Performance impact due to additional circuitry
- Additional design time



Built-In self test - BIST

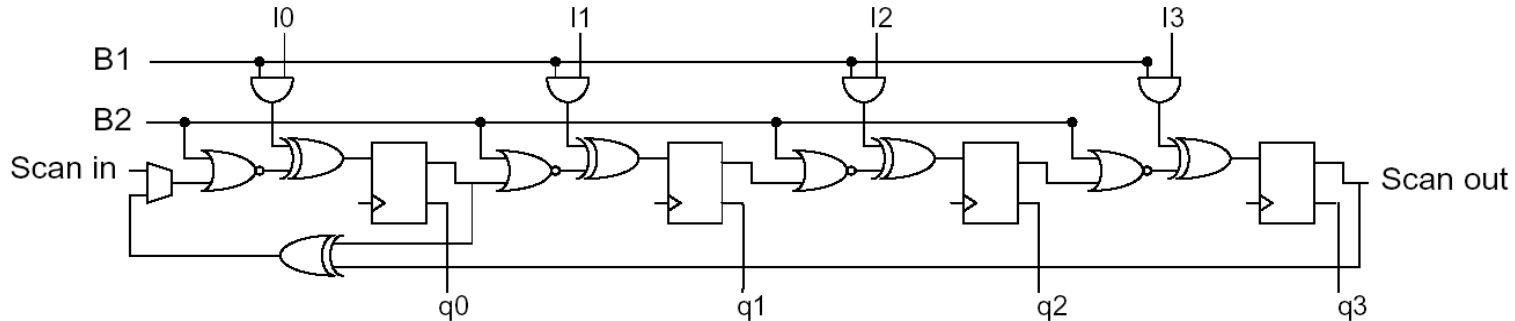
■ BIST Structures:

- 1) Test pattern generators and response analysis on the chip
- 2) Redundancy check of components
- 3) Predefined and saved result codes for functionalities

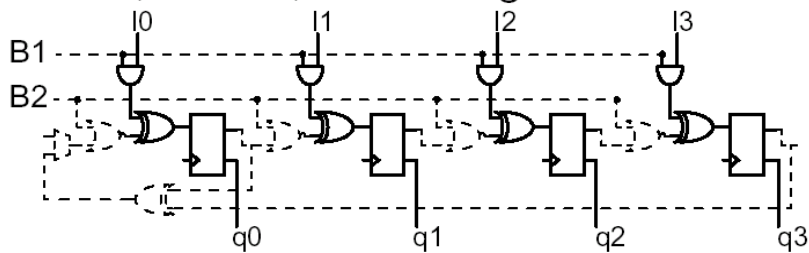


BILBO cell (built in logic block observer)

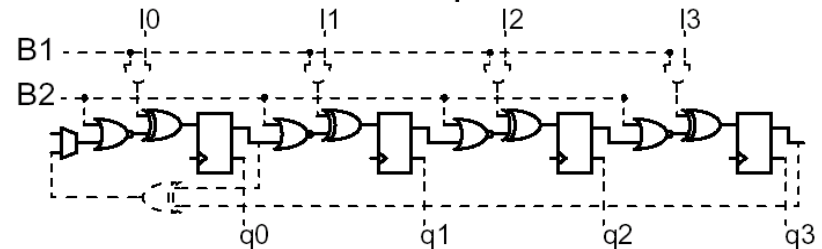
- Scan path cells can be implemented that they can operate as SCAN, LFSR or MISR cells → BILBO



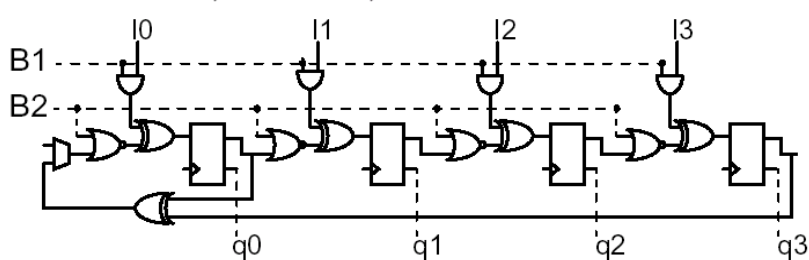
B1,B2 = 11, Normal register mode



B1,B2 = 00, Scan path mode

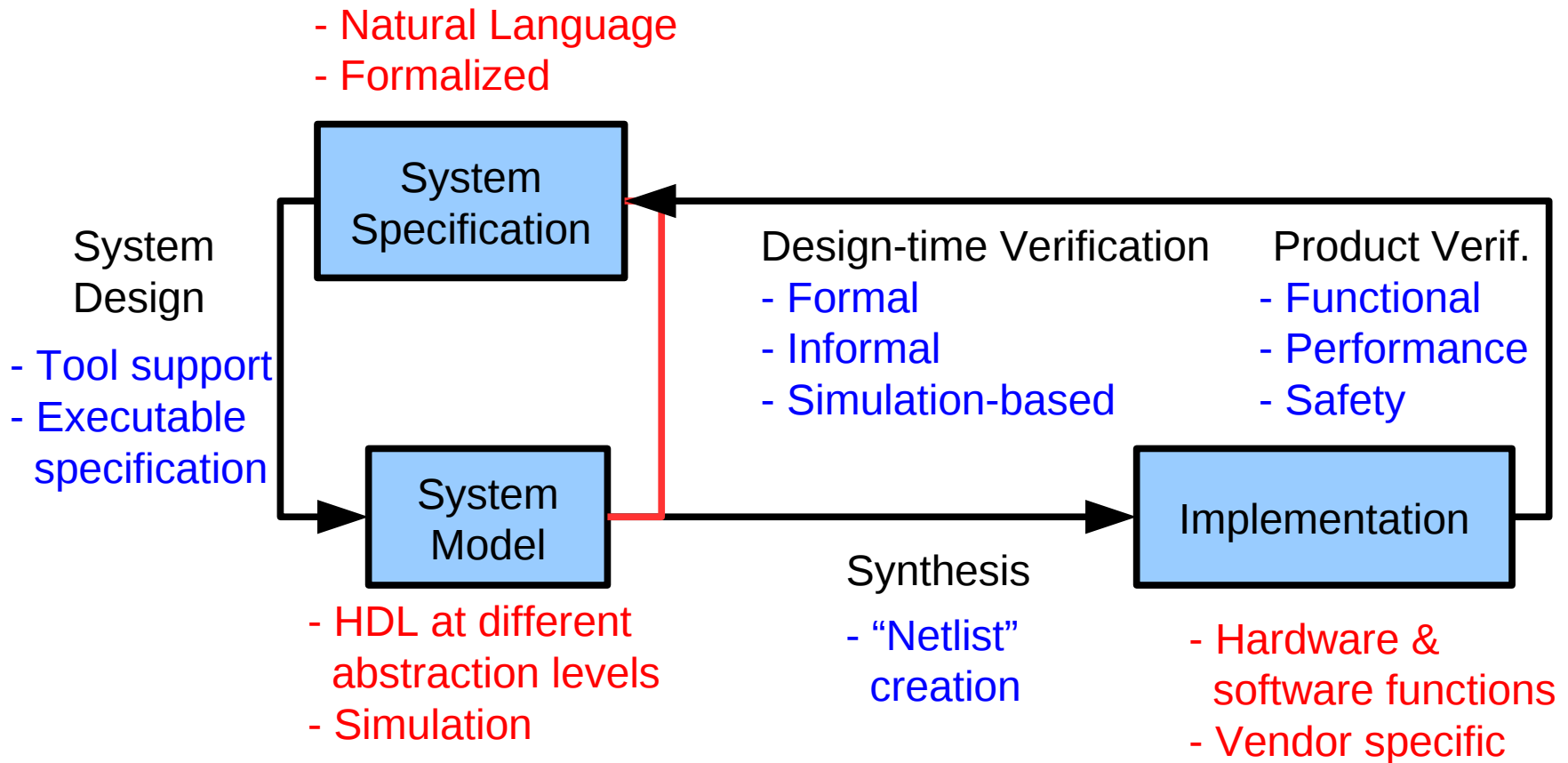


B1,B2 = 10, LFSR mode



B1,B2 = 01, Reset of BILBO

Overview – Design Flow



Definitions

- **Hardware Verification:**

Hardware verification is the proof that a circuit or a system (the design) behaves according to a given set of requirements (the specification).

- **Verification Process:**

For every verification process you must mention what you verified the design against to! You can not just say: “I verified the design”

- **Formal Specification:**

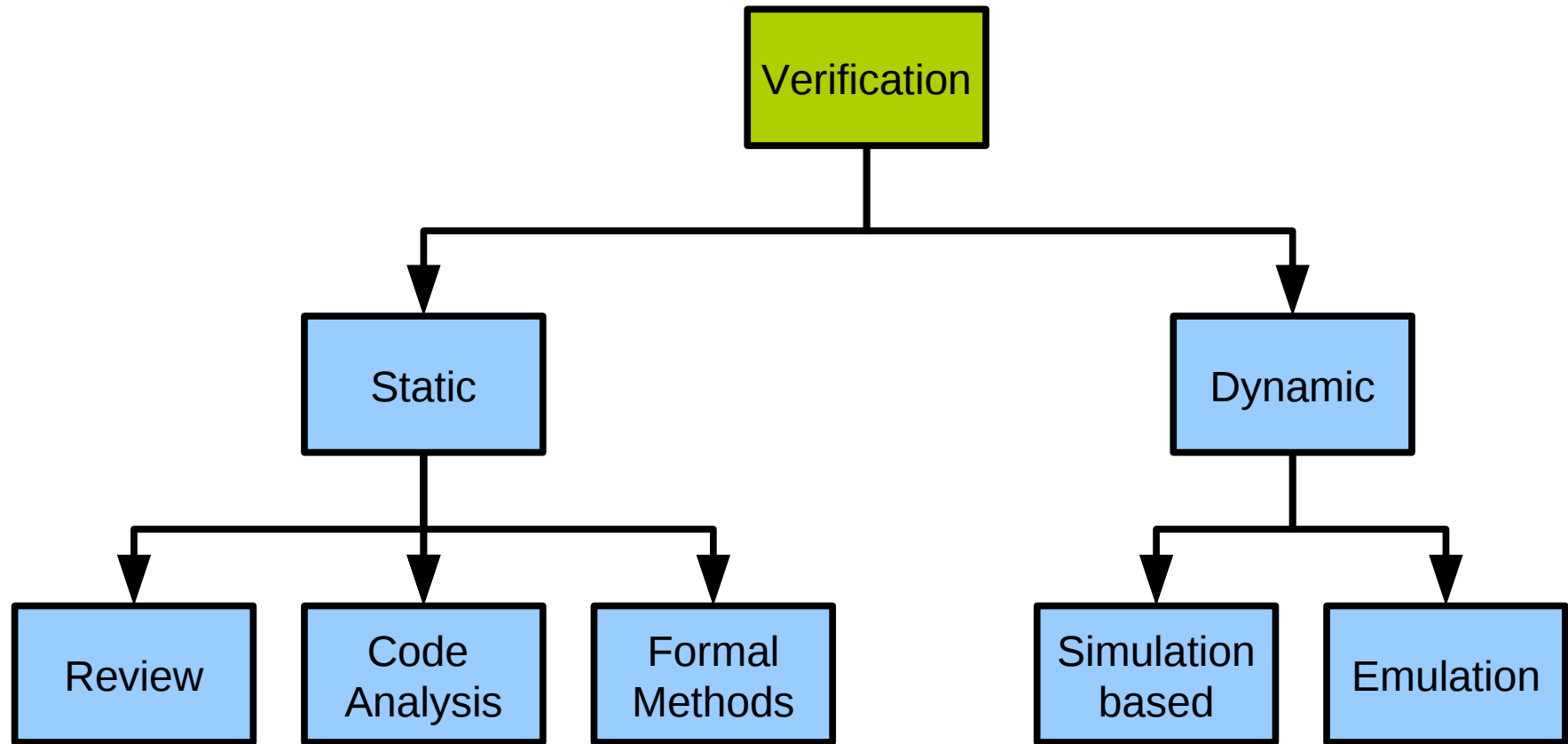
A formal specification is a **concise description** of the behavior and properties of a system written in a **mathematically-based language**, stating what a system is supposed to do in the context it is supposed to operate **as abstractly as possible**, thereby eliminating distraction detail and providing a general description resistant to future system modifications.

Hardware Verification

- Reasons for adoption to circuit (system) design:
 - Quality standard
 - Design as an engineering discipline
 - Compatibility with existing design flows
 - ...

- Limitations
 - Allows only to detect design faults
 - Specifications can be incomplete or contradictory
 - Implementation models can be inconsistent, wrong or too coarse
 - Verification tools can contain faults

Verification tasks



Verification methods

- **Informal Methods – increase the confidence**
 - Functional Simulation – Design Verification (VHDL, SPICE, Matlab, etc.)
 - Assertion based verification
 - Implementation Reviews

- **Formal Methods – results in a proof**
 - Equivalence Checking
 - Symbolic FSM Traversal
 - Structural Equivalence Checking
 - BDD based Equivalence Checking
 - Property Checking
 - Higher-order Logic Theorem Proving
 - Propositional Temporal Logic (PTL) Model Checking

Basic approaches

■ **Black-box Approach**

- The DUT has some I/Os and a defined static/predictable/control able function.
- The function is well documented (... or not :-))
- The black box can be a full system, chip, module or a single macro

■ **White-box Approach**

- Internal facilities and code structures are known.
- The testbench is designed to use this internal knowledge.

■ **Gray-box Approach**

- A limited number of facilities are known and can be utilized for testing. Most of the stuff is black-box
- e.g. some internal signals or interfaces which offer additional information are available at the module's output

Testbench (TB)

- 3 main purposes
 - Generate stimuli
 - Apply stimuli at the DUT at correct time
 - Collect DUT responses and compare this with expected values
- A testbench consists of
 - **Entity**: Has no external ports
 - **Architecture**: The top module. Instantiates the DUT. Instantiates all submodules inside the TB (signal generators, checkers, ...)
 - **Configuration**: Specifies the version of the model under test (optional)

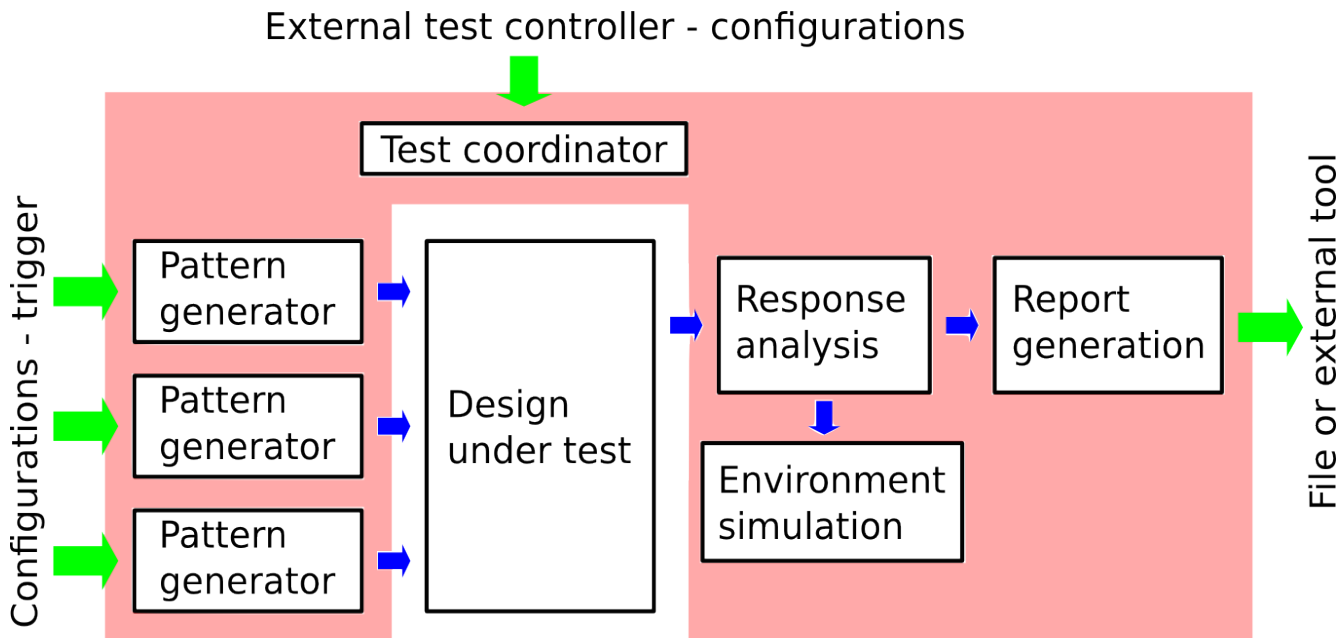
Testbench

```
ENTITY test_bench IS
END test_bench;

ARCHITECTURE tb1 OF test_bench IS
    test_component declaration
    internal signal declarations
BEGIN
    UUT: test_component instantiation

    signals to generate stimulus

    assert statements to verify responses
END tb1;
```



VHDL for TB generation

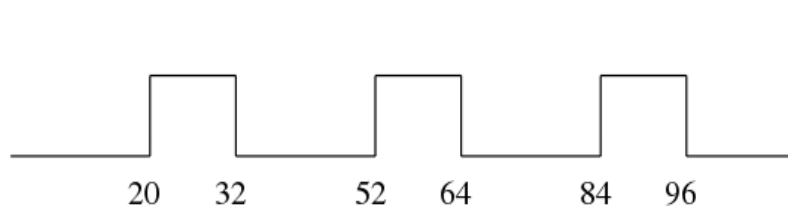
■ Requirements:

- Generating Waveforms
 - Modeling of time (wait statements)
 - Check the response (assert statement)
 - Test management and control (access to files and generation of reports)
 - Add debug messages – console outputs
- These VHDL functions are just for simulation – not for synthesis

Signal generation examples

BEGIN

```
clk <= '1' AFTER 20 ns WHEN clk = '0' ELSE  
      '0' AFTER 12 ns;
```



```
a <= '0', '1' AFTER 8 ns, '0' AFTER 13 ns, '1' AFTER 50 ns;
```



```
two_phase: PROCESS
```

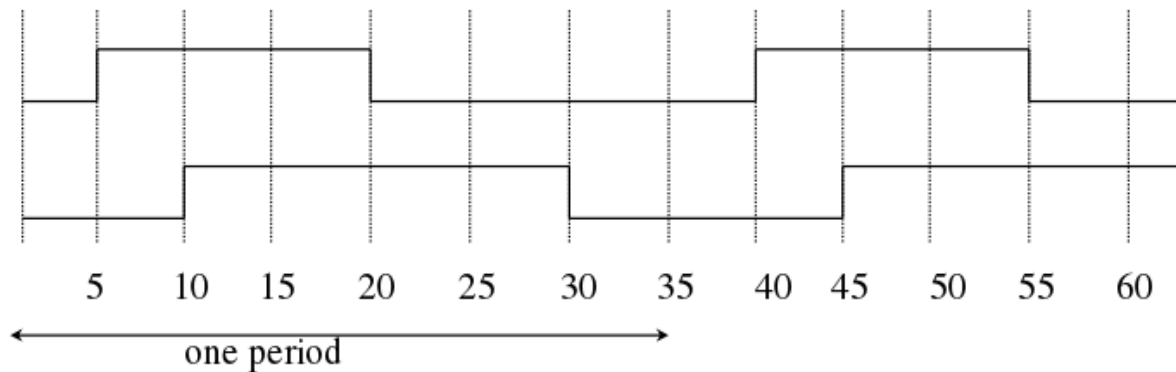
```
BEGIN
```

```
  clk1 <= '0', '1' AFTER 5 ns, '0' AFTER 20 ns;
```

```
  clk2 <= '0', '1' AFTER 10 ns, '0' AFTER 30 ns;
```

```
  WAIT FOR 35 ns;
```

```
END PROCESS two_phase;
```



VHDL assert statements

- Assert is a sequential statement, so it can be used in any VHDL process body.
- Assert **checks a boolean expression**
- If an assert violation arises during simulation the used report statement will prompt that on the console output
- Optional severity level can be added
 - *note, warning, error, failure* (will stop simulation)

```
p_RS_FF : process (S,R) is
begin
    assert S='1' NAND R='1'
    report "S=R=TRUE is not allowed";
    Severity error;

    if (S='1') then
        Q='1';
    end if;

    ...
```

UVM/ OVM

■ Why do we need a methodology?

- Building constrained testbenches is very complex and time consuming
- Verification engineers will divide and conquer
 - Various skills will be applied
 - New scope for reuse
- System modeling language complexity – How to cover verification?

■ Requirements on a verification methodology

- **Abstraction** – layered approach, verification IP generation, ...
- **Re-use** – reusing checkers, reusing test sequences, ...
- **Consistency** – verification IP configuration, how to connect the DUT, well defined generation of simulation phases, ...
- **Behavior of a testbench** – Macros, callbacks, factories, virtual functions, ...
- **Language abstraction** – Adoption for various state of the art HDLs



OVM/UVM

- OVM (open verification methodology)
 - Joint development of Mentor and Cadence
 - TLM communication
 - Reuse and customization through a **class factory (no new language)**
 - **Common** message reporting and formatting **interface**
- UVM (unified verification Methodology)
 - Based on OVM
 - **Industry wide, and standardized** by Accellera
 - Adopted for other languages (originally system Verilog)

Content

- **Functional Virtual Prototype (FVP)**
 - Golden functional model combined with testbench
 - Created on Transaction-Level (TLM communication) → Refined down to implementation level
- UVM provides collection of stimuli generators, interface monitors and application checkers for all levels
- **Abstraction of verification** → high performance
- **Assertions** → capture intended behavior
- **Functional coverage** → effectiveness of the verification
- **Automatic input stimuli generation** → ATPG

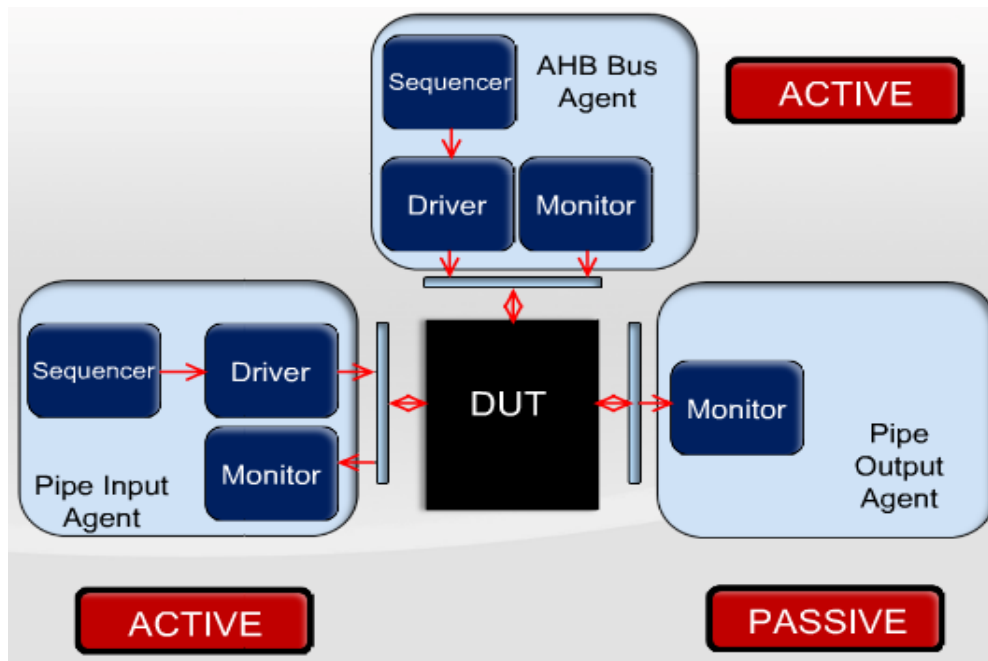
UVM Macros

- Utility macros to register classes with the verification factory and give access to the methods
- Field automation macros give access to common functions
- Report info macros for user feedback
- Implementation declaration macros. Enables the implementation of more than one instance of a dedicated TLM interface.
- etc. → see UVM manual



UVM Testbench architecture

- The AHB (advanced High-performance bus) agent drives the register read and write accesses
- Input and output agents
- Reuse of the monitor function block
- Ensure that each monitor selects the correct interface



UVM Phases

build_phase

Builds components top-down

connect_phase

Connects the components in the environment

end_of_elaboration_phase

Post elaboration activity

start_of_simulation_phase

Configuration of components before the simulation begins

run_phase

Test Execution

extract_phase

Collects test details after run execution

check_phase

Checks simulation results

report_phase

Reporting of simulation results

UVM Example

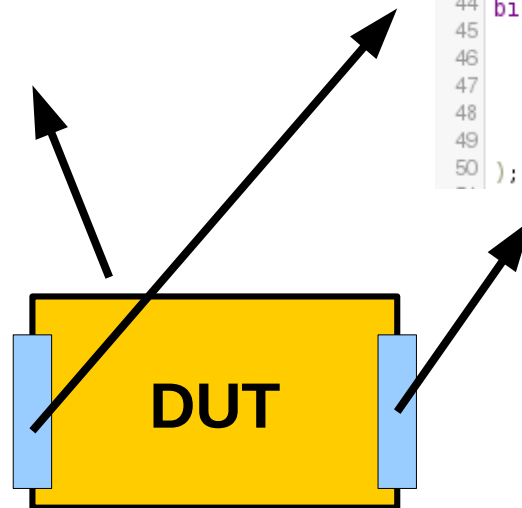
The design → design.sv

```
design.sv +
1 // Simple adder/subtractor module
2 module ADD_SUB(
3     input          clk,
4     input [7:0]    a0,
5     input [7:0]    b0,
6     // if this is 1, add; else subtract
7     input          doAdd0,
8     output reg [8:0] result0
9 );
10
11 always @ (posedge clk)
12 begin
13     if (doAdd0)
14         result0 <= a0 + b0;
15     else
16         result0 <= a0 - b0;
17 end
18
19 endmodule: ADD_SUB
20
```

```

21 //-----
22 // Interface for the adder/subtractor DUT
23 //-----
24 interface add_sub_if(
25     input bit clk,
26     input [7:0] a,
27     input [7:0] b,
28     input      doAdd,
29     input [8:0] result
30 );
31
32     clocking cb @(posedge clk);
33     output    a;
34     output    b;
35     output    doAdd;
36     input     result;
37 endclocking // cb
38
39 endinterface: add_sub_if
40
41 //-----
42 // Interface bind
43 //-----
44 bind ADD_SUB add_sub_if add_sub_if0(
45     .clk(clk),
46     .a(a0),
47     .b(b0),
48     .doAdd(doAdd0),
49     .result(result0)
50 );

```

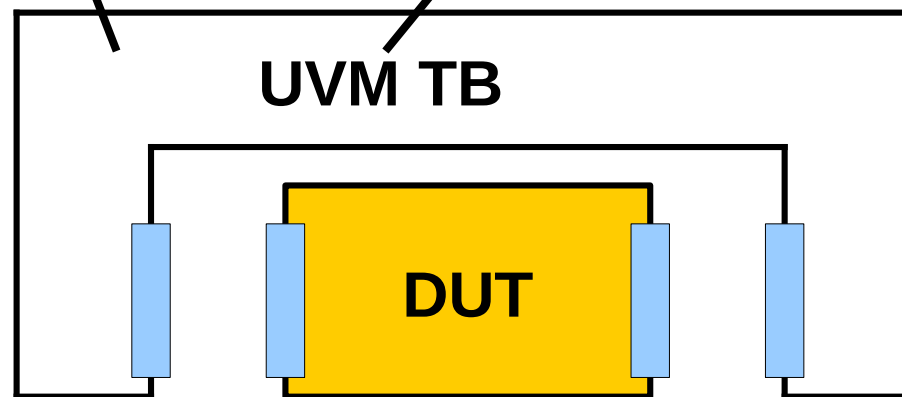


UVM Example

The UVM testbench → testbench.sv

```
4 //-----  
5 // environment env  
6 //-----  
7 class env extends uvm_env;  
8  
9     virtual add_sub_if m_if;  
10
```

```
42 //-----  
43 // module top  
44 //-----  
45 module top;  
46  
47     bit clk;  
48     env environment;  
49     ADD_SUB dut(.clk (clk));  
50  
51     initial begin  
52         environment = new("env");  
53         // Put the interface into the resource database.  
54         uvm_resource_db#(virtual add_sub_if)::set("env",  
55             "add_sub_if", dut.add_sub_if0);  
56         clk = 0;  
57         run_test();  
58     end  
59  
60     initial begin  
61         forever begin  
62             #(50) clk = ~clk;  
63         end  
64     end  
65  
66     initial begin  
67         // Dump waves  
68         $dumpvars(0, top);  
69     end  
70  
71 endmodule
```

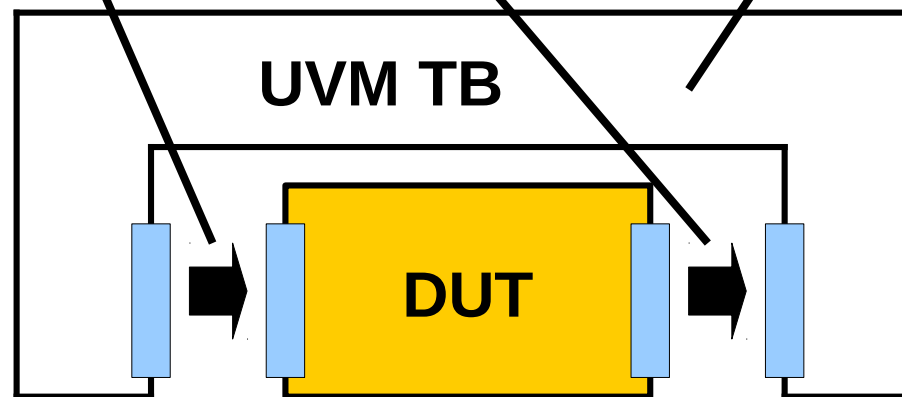


UVM Example

The UVM testbench → testbench.sv

```
15 function void connect_phase(uvm_phase phase);
16     `uvm_info("LABEL", "Started connect phase.", UVM_HIGH);
17     // Get the interface from the resource database.
18     assert(uvm_resource_db#(virtual add_sub_if)::read_by_name(
19         get_full_name(), "add_sub_if", m_if));
20     `uvm_info("LABEL", "Finished connect phase.", UVM_HIGH);
21 endfunction: connect_phase
```

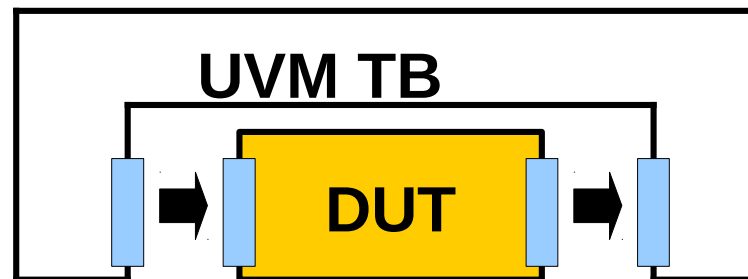
```
23 task run_phase(uvm_phase phase);
24     phase.raise_objection(this);
25     `uvm_info("LABEL", "Started run phase.", UVM_HIGH);
26     begin
27         int a = 8'h2, b = 8'h3;
28         @(m_if.cb);
29         m_if.cb.a <= a;
30         m_if.cb.b <= b;
31         m_if.cb.doAdd <= 1'b1;
32         repeat(2) @(m_if.cb);
33         `uvm_info("RESULT", $sformatf("%0d + %0d = %0d",
34             a, b, m_if.cb.result), UVM_LOW);
35     end
36     `uvm_info("LABEL", "Finished run phase.", UVM_HIGH);
37     phase.drop_objection(this);
38 endtask: run_phase
```



UVM Example

Results

```
run -all;
# KERNEL: UVM_INFO @ 0: reporter [RNTST] Running test ...
# KERNEL: UVM_INFO /home/runner/testbench.sv(16) @ 0: env [LABEL] Started connect phase.
# KERNEL: UVM_INFO /home/runner/testbench.sv(20) @ 0: env [LABEL] Finished connect phase.
# KERNEL: UVM_INFO /home/runner/testbench.sv(25) @ 0: env [LABEL] Started run phase.
# KERNEL: UVM_INFO /home/runner/testbench.sv(34) @ 250: env [RESULT] 2 + 3 = 5
# KERNEL: UVM_INFO /home/runner/testbench.sv(36) @ 250: env [LABEL] Finished run phase.
# KERNEL: UVM_INFO /home/build/vlib1/vlib/uvm-1.2/src/base/uvm_objection.svh(1271) @ 250: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
# KERNEL: UVM_INFO /home/build/vlib1/vlib/uvm-1.2/src/base/uvm_report_server.svh(862) @ 250: reporter [UVM/REPORT/SERVER]
# KERNEL: --- UVM Report Summary ---
# KERNEL:
# KERNEL: ** Report counts by severity
# KERNEL: UVM_INFO :      8
# KERNEL: UVM_WARNING :    0
# KERNEL: UVM_ERROR :     0
# KERNEL: UVM_FATAL :     0
# KERNEL: ** Report counts by id
# KERNEL: [LABEL]      4
# KERNEL: [RESULT]     1
# KERNEL: [RNTST]      1
# KERNEL: [TEST_DONE]   1
# KERNEL: [UVM/RELNOTES] 1
# KERNEL:
# RUNTIME: Info: RUNTIME_0068 uvm_root.svh (521): $finish called.
# KERNEL: Time: 250 ns, Iteration: 57, Instance: /top, Process: @INITIAL#51_0@.
# KERNEL: stopped at time: 250 ns
```



Regression Testing

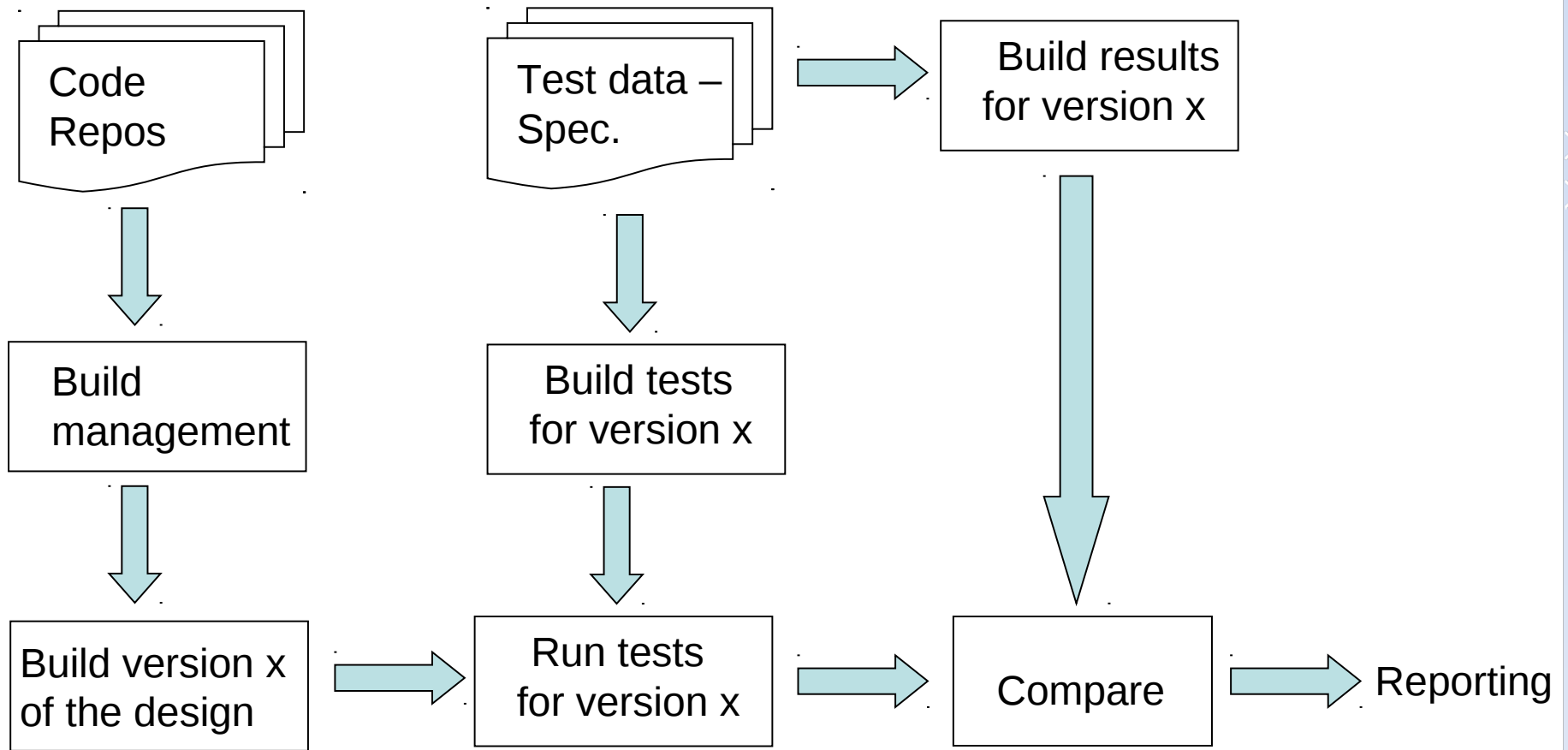
- Regression testing is an automated test sequence which checks that updates or bugfix of a subsystem do not affect other functionality.
- Regression tests are applied for example during night and once a week.
- Regression tests are highly automated
- Regression test sequences are usually large and consume high computational power.
- Aim to check functionality and the architecture.
- **Reasons for doing a regression test**
 - Fix a bug
 - Add/Delete/Change functionality
 - Change platform
 - Create any variants of the system

} Creation of a new release

Test automation for regression tests

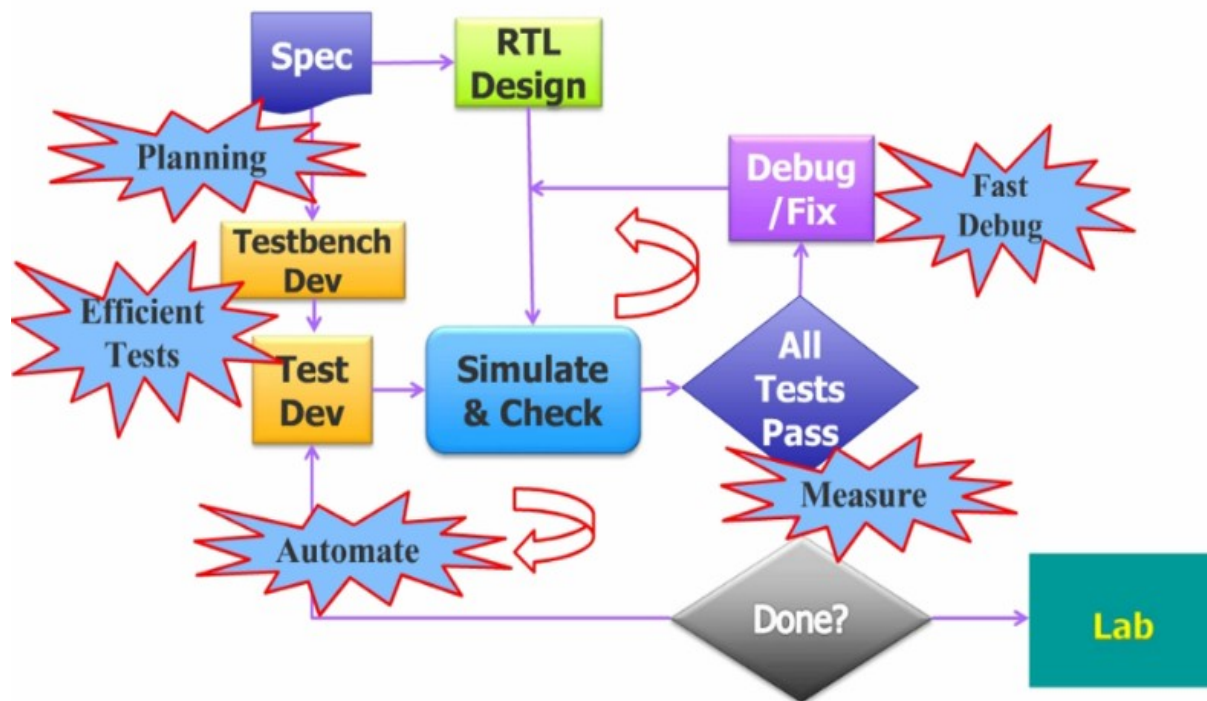
- **Format problems due to variances in print methods**
 - Use of date and timestamps
 - Line shifts and blanks
 - Logging outputs
 - Variability in message headers
- **Design challenges for automated testing**
 - Use methods/tools that extract the relevant information from output/logging streams.
 - Compare this concentrated information with the predefined result database.
 - Implement functionality which returns relevant information as non string datatype.
- Complex protocols (e.g. PCIe, USB, ...) are complex to test
- Integration of 3PIP (black- gray box testing)

Regression testing



Motivation for Coverage Measures

- “We don't want to debug in the Lab, simulation is much more productive !”
- “When is my design good enough to go to the Lab ?”



[Mentor Graphics]

Coverage

- Coverage is a measure for completeness of verification
- **2 Basic types of coverage:**
 - **Code coverage:** Can be automatically extracted by the tools for the designed code
 - No changes to your current simulation approach → just turn it on!
 - Have I tested everything I wrote ?
 - Is there some unused code ?
 - **Functional coverage:** Developed by the user to link the testbench to the design's functionality
- Code and functional coverage are not the same and must be evaluated individually to get the full picture.

Coverage results

- Executing each statement
- Taking every branch
- Seeing every term of expressions
- Seeing all FSM state-transitions

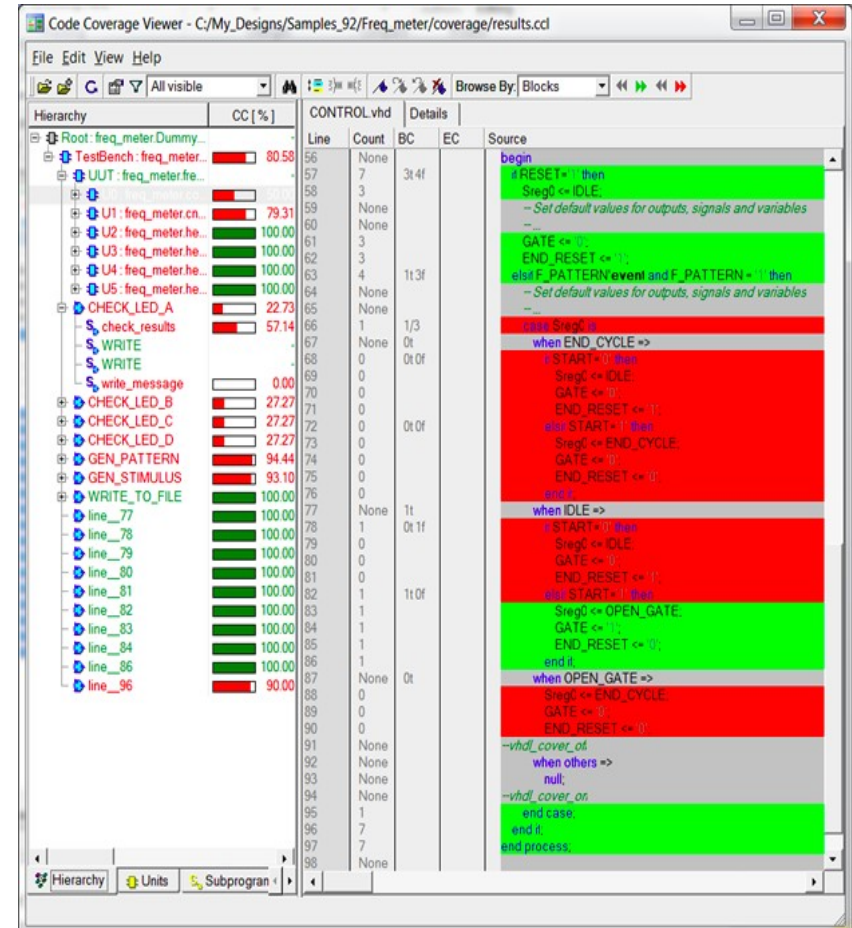
**GOAL IS TO REACH
100% COVERAGE**

```

357 case state_r is
      when STOPPED =>
10   if detect_start = '1' then
2     state <= STARTED;
      end if;

      when STARTED =>
120  if detect_stop = '1' then
0     state <= STOPPED;
120  elsif scl_r1 Condition never evaluated to TRUE
45    shift_en <= '1';
45    if byte_done = '1' then
3     state <= CHECK_DEVA;
      end if;
  
```

| Name | Coverage | Status | % of Goal |
|--------------------------|----------|--------|-----------|
| /tb/duta | | | |
| TYPE c1_cg | 50.0% | | 50.0% |
| CVP c1_cg::x | 50.0% | | 50.0% |
| bin_0 | 2 | 100.0% | |
| bin_1 | 0 | 0.0% | |
| INST Vtb/duta/c1_cg_inst | 50.0% | | 50.0% |
| CVP x | 50.0% | | 50.0% |
| bin_0 | 2 | 100.0% | |
| bin_1 | 0 | 0.0% | |
| /tb/dutb | | | |
| TYPE c1_cg | 50.0% | | 50.0% |
| CVP c1_cg::x | 50.0% | | 50.0% |
| bin_0 | 0 | 0.0% | |
| bin_1 | 2 | 100.0% | |
| INST Vtb/dutb/c1_cg_inst | 50.0% | | 50.0% |
| CVP x | 50.0% | | 50.0% |
| bin_0 | 0 | 0.0% | |
| bin_1 | 2 | 100.0% | |



Condition/Decision coverages

- **Decision coverage** is a synonym for **branch coverage**
- **Condition coverage**
 - Each (part) condition of a boolean expression must be changed for the test.
 - Each boolean variable must be set to both true and false states
 - Drawback: It may happen that boolean condition is never covered and has no impact on the decision.
- **Modified condition decision coverage (MC/DC)**
 - Additionally it must be shown that each partial condition has an impact on the branch (decision)
 - Create a test to show that a variable can modify the branch independent of all other conditions.

Decision/condition coverage example

```
extern int a,b;  
void to_test (void) {  
    if (a && b)  
        printf("Hello");  
}
```

1) 100% Statement Coverage

```
void test1 (void) {  
    a=1; b=1; to_test();  
}
```

3) 100% Decision/Condition coverage

```
void test3 (void) {  
    a=0; b=0; to_test();  
    a=1; b=1; to_test();  
}
```

2) 100% Decision (branch) coverage

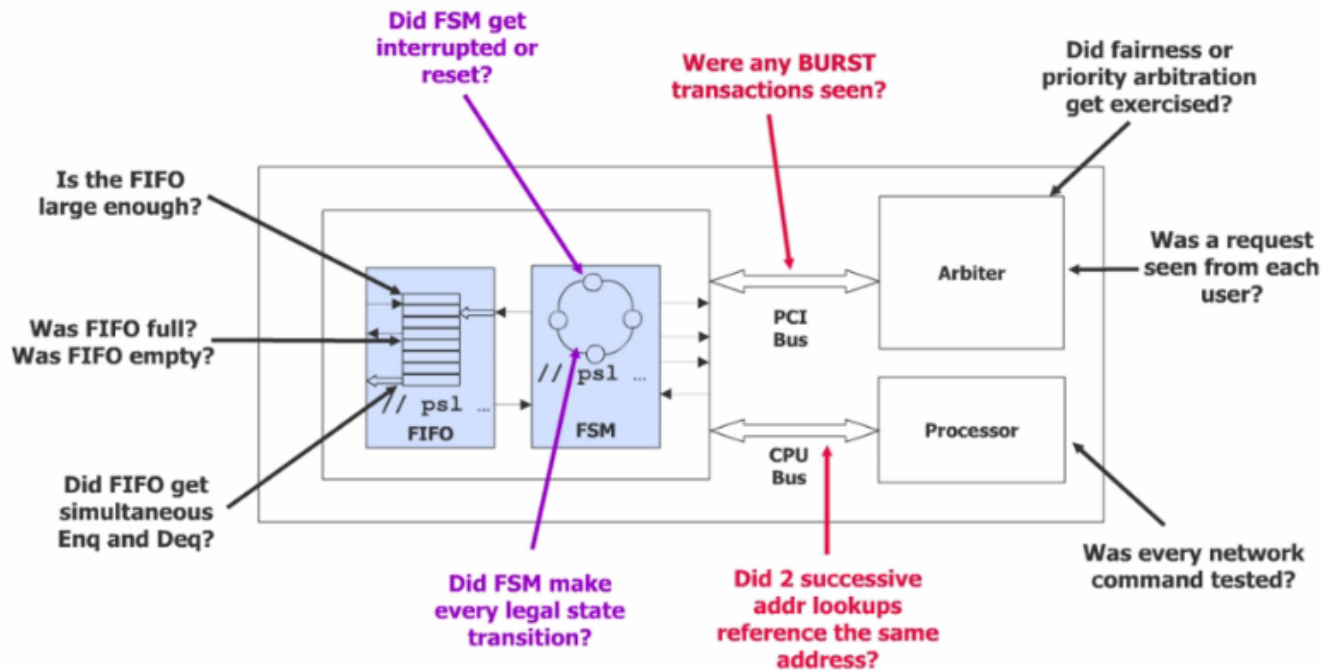
```
void test2 (void) {  
    a=0; b=1; to_test();  
    a=1; b=1; to_test();  
}
```

4) 100% MC/DC

```
void test4 (void) {  
    a=1; b=1; to_test();  
    a=0; b=1; to_test();  
    a=1; b=0; to_test();  
}
```

Minimal test set for 100% SC, DC, CC, MC/DC = {1/1 , 0/1 , 1/0}

Functional Coverage

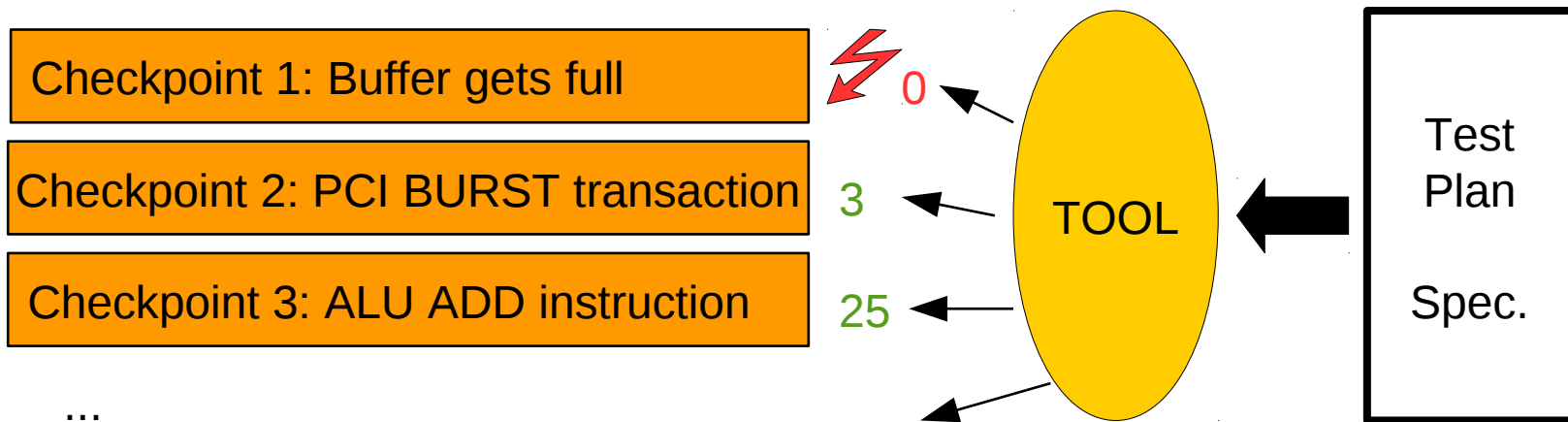


[Mentor Graphics]

- Must be specified manually and can not be inferred from the design.
- Automation from tools supported in specifying functional checkpoints.

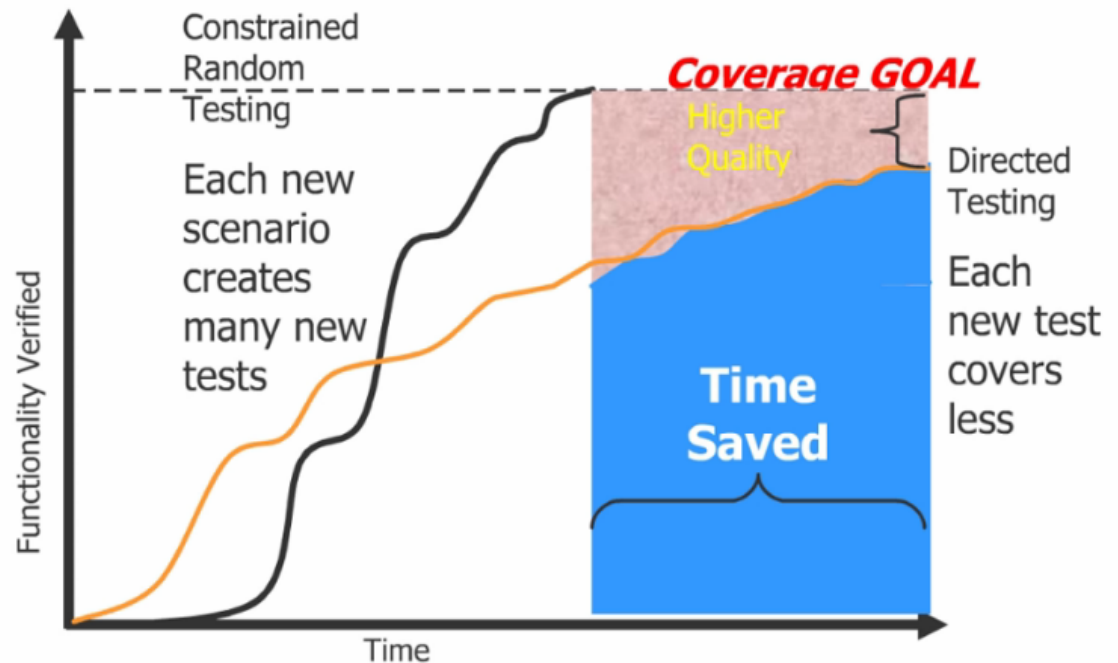
Functional Coverage

- Motivation Questions for functional coverage:
 - “Have I verified all functional requirements?”
 - “Have I covered the full verification plan?”
 - “Have I executed all corner cases in my design?”
 - ...
- Verifies the designed functionality
- Counts how many times something “meaningful” happens during the test



Constrained random testing

- Use a constrained random solver to generate test stimulus automatically.
- The solver will pick test values from a constrained solution space.
- A feedback mechanism will improve the effectiveness of this methodology.

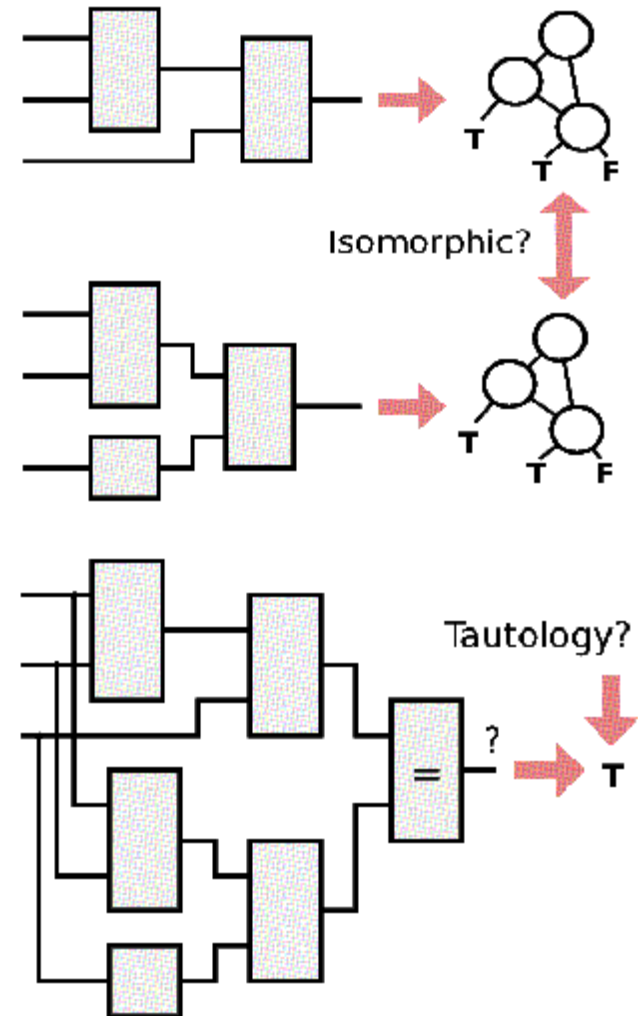


Specification, Proof Goals and Proof

- Implicit Hardware Verification
 - Establishing equivalence
 - Test for isomorphism

- Explicit Hardware Verification
 - Methods based on mathematical logic
 - Proof goal is stated explicitly
 - SAT Problem

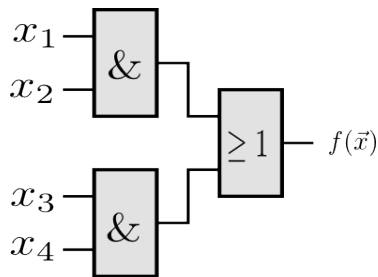
$$\models Impl \rightarrow Spec$$



Equivalence Checking Application

Representations at different levels of abstraction result in equal ROBDD structures

Gate Level



Formal Specification

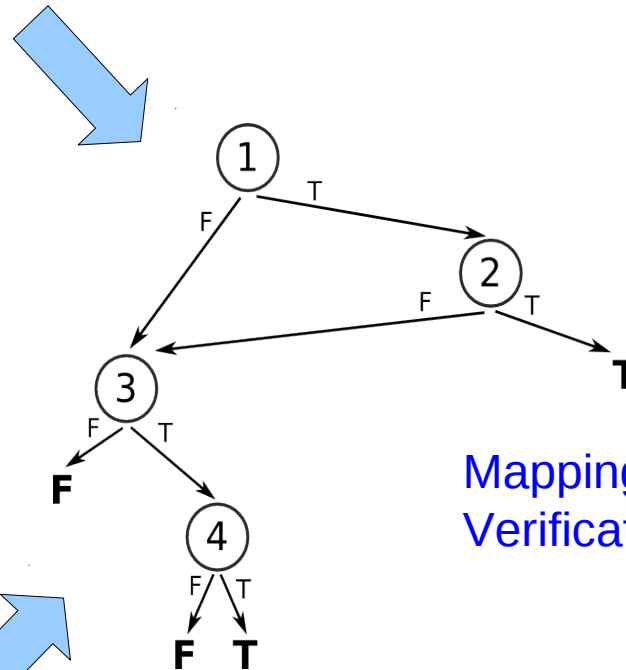
$$f(x_1, x_2, x_3, x_4) = (x_1 \text{ AND } x_2) \text{ OR } (x_3 \text{ AND } x_4)$$

High Level exec. Spec.

```
if (x1 && x2)
  y=TRUE;
else
  y=FALSE;
if (x3 && x4)
  y=TRUE;
else
  y=FALSE;
```

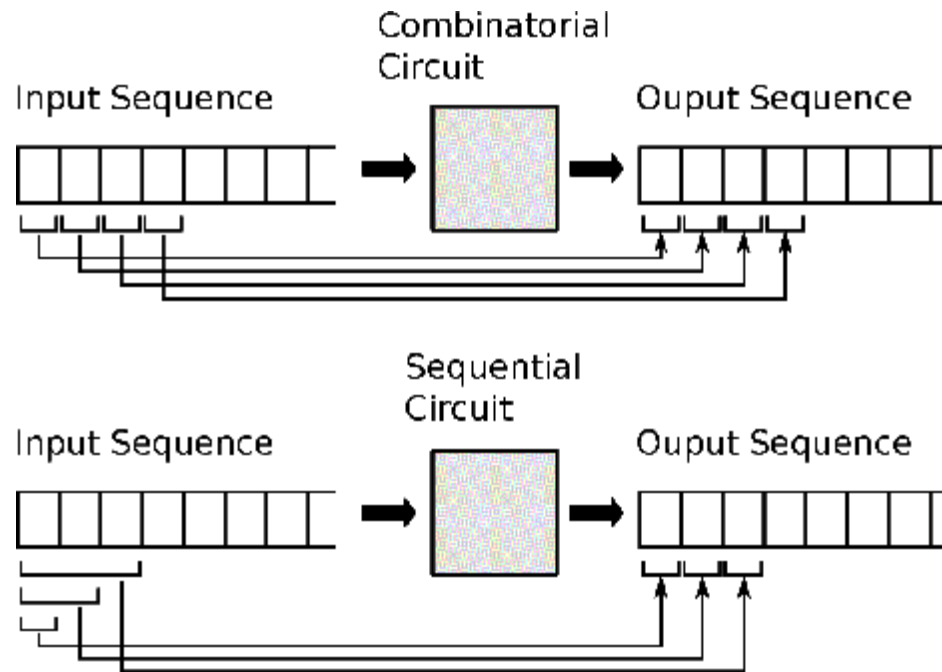
VHDL

```
Y <= (X1 AND X2) OR (X3 AND X4);
```



Mapping by
Verification Tool

Approaches sequential circuits



- FSM representations
- The input of a finite automata are finite sequences of values.

Automata for finite Sequences

Abstract automata definition:

A finite deterministic automata \mathcal{M} (Finite State Machine FSM) is a 6-tuple $\mathcal{M} := (Q, \Sigma, \Delta, \delta, \lambda, q^0)$. Q is the finite set of states, Σ is the input alphabet, Δ is the output alphabet, δ is the state transition function with $\delta : Q \times \Sigma \rightarrow Q$, λ is the output function and q^0 is the initial state.

For a Mealy automata λ is defined as $\lambda : Q \times \Sigma \rightarrow \Delta$ and for a Moore automata as $\lambda : Q \rightarrow \Delta$.



Product Automata

■ Definition

Given two automata $\mathcal{M} := (Q, \Sigma, \Delta, \delta, \lambda, q^0)$ and $\mathcal{M}' := (Q', \Sigma, \Delta, \delta', \lambda', q'^0)$, with an equal input/output alphabet Σ and Δ .
The product automata $\mathcal{M}^P := \mathcal{M} \times \mathcal{M}'$ is defined as $\mathcal{M}^P := (Q \times Q', \Sigma, \mathbb{B}, \delta^P, \lambda^P, (q^0, q'^0))$ with $\delta^P : (Q \times Q') \times \Sigma \rightarrow (Q \times Q')$ and $\lambda^P : (Q \times Q') \times \Sigma \rightarrow \mathbb{B}$

■ Properties

- The states of the product automaton are the state pairs of both
- The output function partially computes the comparison of the two original automata. The **output is a boolean value which indicates if both original automata outputs are equal (T) or not (F)**
- Synchronous Product

Equivalence

- State Equivalence:

Two states of Mealy automata q and q' are equivalent
 $q \sim q'$ if $\forall a, a \in \Sigma. \lambda(q, a) = \lambda'(q', a)$

- Two automata are called equivalent if for an arbitrary input sequence applied at both automata the same output sequence results \rightarrow Equal behavior.
- Automata equivalence:

For checking the equivalence of two automata \mathcal{M} and \mathcal{M}'
 we build the product automaton $\mathcal{M}^P := \mathcal{M} \times \mathcal{M}'$

with $\mathcal{M}^P := (Q \times Q', \Sigma, \mathbb{B}, \delta^P, (q^0, q'^0))$

Two automata are equivalent if

the initial states are equivalent $q^0 \sim q'^0$

and $(q, q') \in \sim \Leftrightarrow \forall a, a \in \Sigma, \lambda^P((q, q'), a) = \mathbf{T}$

and $\delta^P((q, q'), a) \in \sim$ with $\sim \subseteq Q \times Q'$

Example

Have the following two automata an equal behavior for all possible input sequences?

$$\mathcal{M}_1 := (Q, \Sigma, \Delta, \delta, \lambda, q^0)$$

$$\mathcal{M}_2 := (Q', \Sigma, \Delta, \delta', \lambda', q'^0)$$

with

$$Q = \{0, 1, 2\}$$

$$Q' = \{0, 1\}$$

$$\Sigma = \{a, b\}$$

$$\Delta = \{\alpha, \beta\}$$

$$q^0 = \{0\}$$

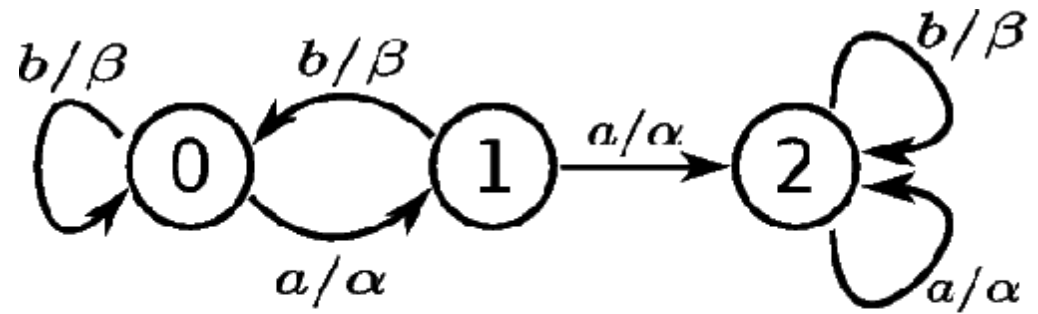
$$q'^0 = \{0\}$$



Example

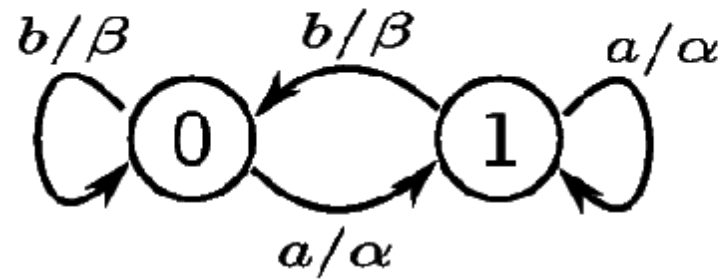
δ and λ

| state | next_state | condition | output |
|-------|------------|-----------|----------|
| 0 | 1 | a | α |
| 0 | 0 | b | β |
| 1 | 2 | a | α |
| 1 | 0 | b | β |
| 2 | 2 | a | α |
| 2 | 2 | b | β |



δ' and λ'

| state | next_state | condition | output |
|-------|------------|-----------|----------|
| 0 | 1 | a | α |
| 0 | 0 | b | β |
| 1 | 1 | a | α |
| 1 | 0 | b | β |



Example

Build the product automaton

$$\mathcal{M}^P := (Q^P, \Sigma, \delta^P, \mathbb{B}, (q^0, q'^0))$$

$$\mathcal{M}^P = \mathcal{M}_1 \times \mathcal{M}_2$$

$$Q^P = Q \times Q'$$

$$Q^P = \{(0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)\}$$

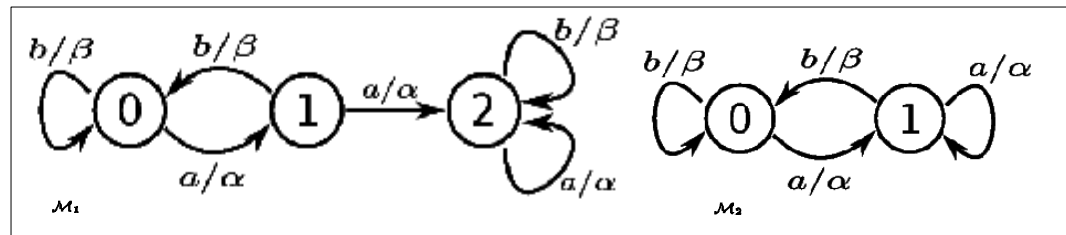
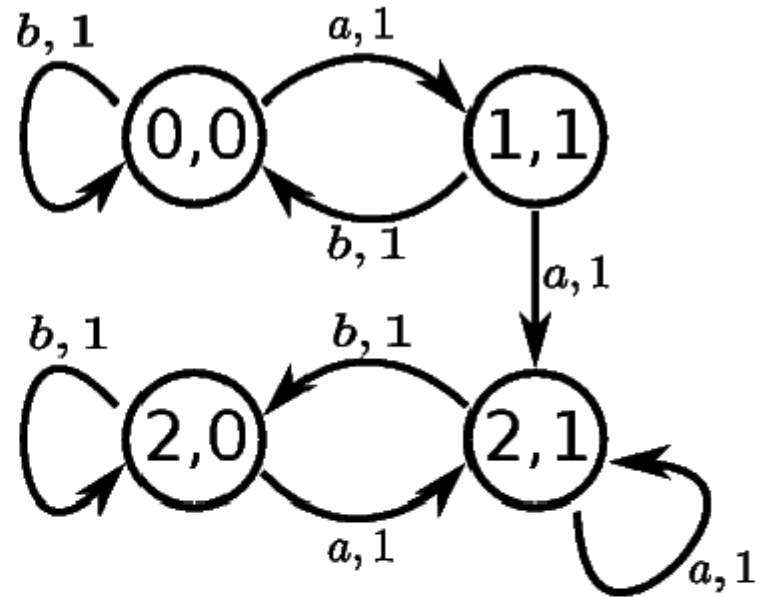
$$\Sigma = \{a, b\}$$

$$\delta^P : (Q \times Q') \times \Sigma \rightarrow (Q \times Q')$$

$$\lambda^P : (Q \times Q') \times \Sigma \rightarrow \mathbb{B}$$

Example

| state | next_state | condition | output |
|-------|------------|-----------|--------|
| (0,0) | (1,1) | a | 1 |
| (0,0) | (0,0) | b | 1 |
| (0,1) | - | a | - |
| (0,1) | - | b | - |
| (1,0) | - | a | - |
| (1,0) | - | b | - |
| (1,1) | (2,1) | a | 1 |
| (1,1) | (0,0) | b | 1 |
| (2,0) | (2,1) | a | 1 |
| (2,0) | (2,0) | b | 1 |
| (2,1) | (2,1) | a | 1 |
| (2,1) | (2,0) | b | 1 |



$$\forall a, a \in \Sigma, \lambda^P((q, q'), a) = \mathbf{T}$$

=> Automata \mathcal{M}_1 and \mathcal{M}_2 are equivalent
And have an equal I/O behavior for all possible input sequences

Thank you for your attention !

Interested ?

- Master thesis
- Seminary work
- etc.

Michael.Rathmair@tuwien.ac.at

