Verification

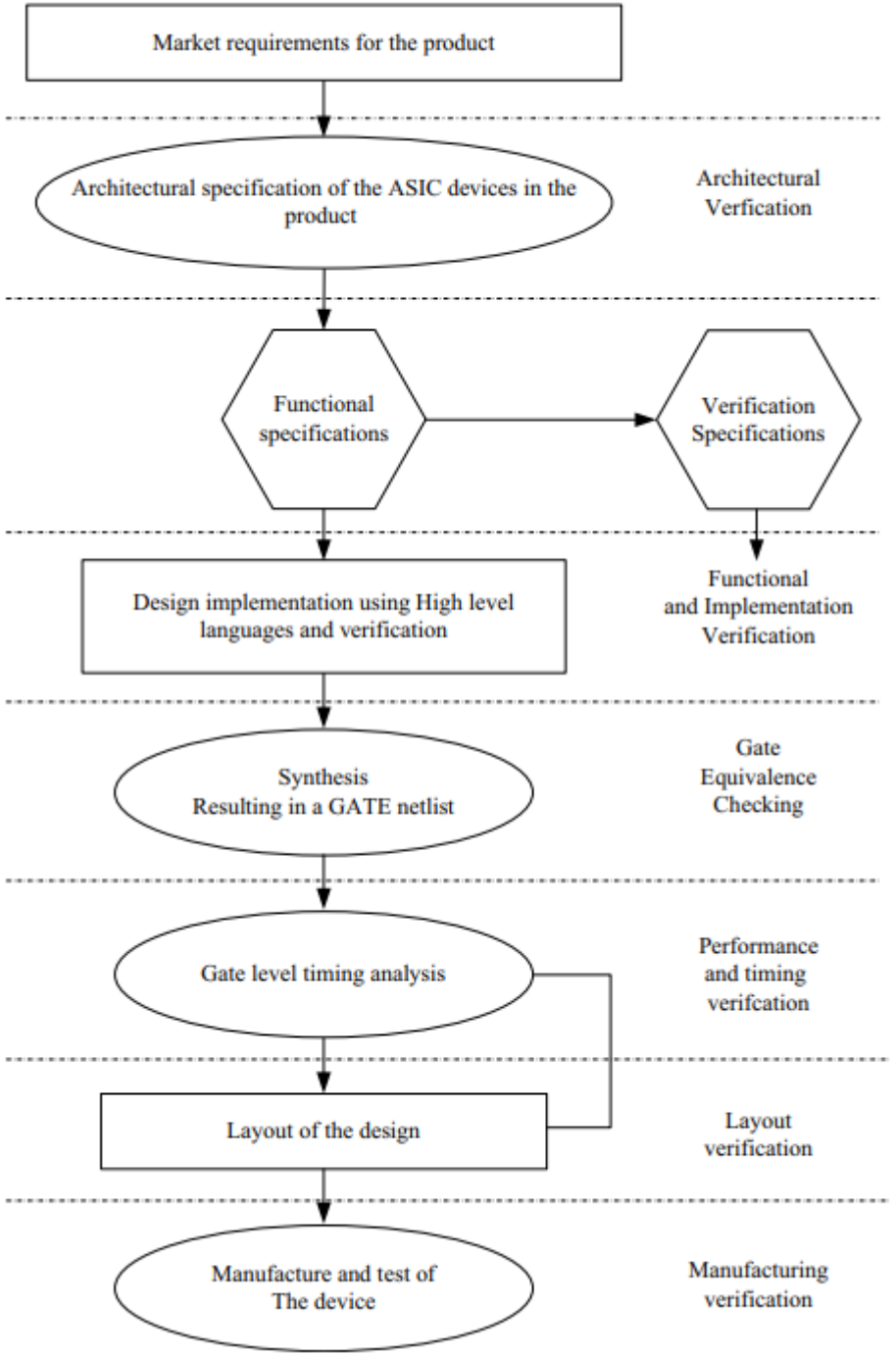
Importance of Verification

A typical project team usually has an equal number of design engineers and verification engineers. Sometimes verification engineers outnumber design engineers by as many as two to one. This stems from the fact that to verify a design, one must first understand the specifications as well as the design and, more important, devise a different design approach from the specifications. It should be emphasized that the approach of the verification engineer is different from that of the design engineer. If the verification engineer follows the same design style as the design engineer, both would commit the same errors and not much would be verified.

From the project development cycle, we can understand the difficulty of design verification. Statistical data show that around 70% of the project development cycle is devoted to design verification. A design engineer usually constructs the design based on cases representative of the specifications. However, a verification engineer must verify the design under all cases, and there are an infinite number of cases (or so it seems). Even with a heavy investment of resources in verification, it is not unusual for a reasonably complex chip to go through multiple tape-outs before it can be released for revenue.

The impact of thorough design verification cannot be overstated. A faulty chip not only drains budget through respin costs, it also delays time-to-market, impacts revenue, shrinks market share, and drags the company into playing the catch-up game. Therefore, until people can design perfect chips, or chip fabrication becomes inexpensive and has a fast turnaround time, design verification is here to stay.

Overview of the IC Design Process



Functional Verification

Functional verification is the activity where the design or product is tested to make sure that all the functions of the device are indeed working as stated. This activity ensures that the features functions as specified. It is noted that some of the device features may or may not be visible to the user and may be internal to the design itself. However, it is imperative that all the features are verified to operate correctly as specified. This verification activity usually consumes the most time in the design cycle.

Gate Equivalence verification

Equivalence verification is the activity to ensure that the schematic generated or created is actually a true representation of the design specified and verified functionally above. At this stage not much timing information is considered. Only logical and sequential relationships are considered.

Timing verification

Timing verification is the activity where the timing of the circuits is actually verified for various operating conditions after taking into account various parameters like temperature etc. This activity is usually done after all the functions of the feature are verified to be operating correctly. This is an important part of the verification activity.

Performance verification

Performance verification is the activity that could be carried out after an initial test device has been created as frequently happens with analog circuits or is carried on as a part of the functional verification activity. In some organizations, this term is also used for performance characterization of the device architecture

Layout verification

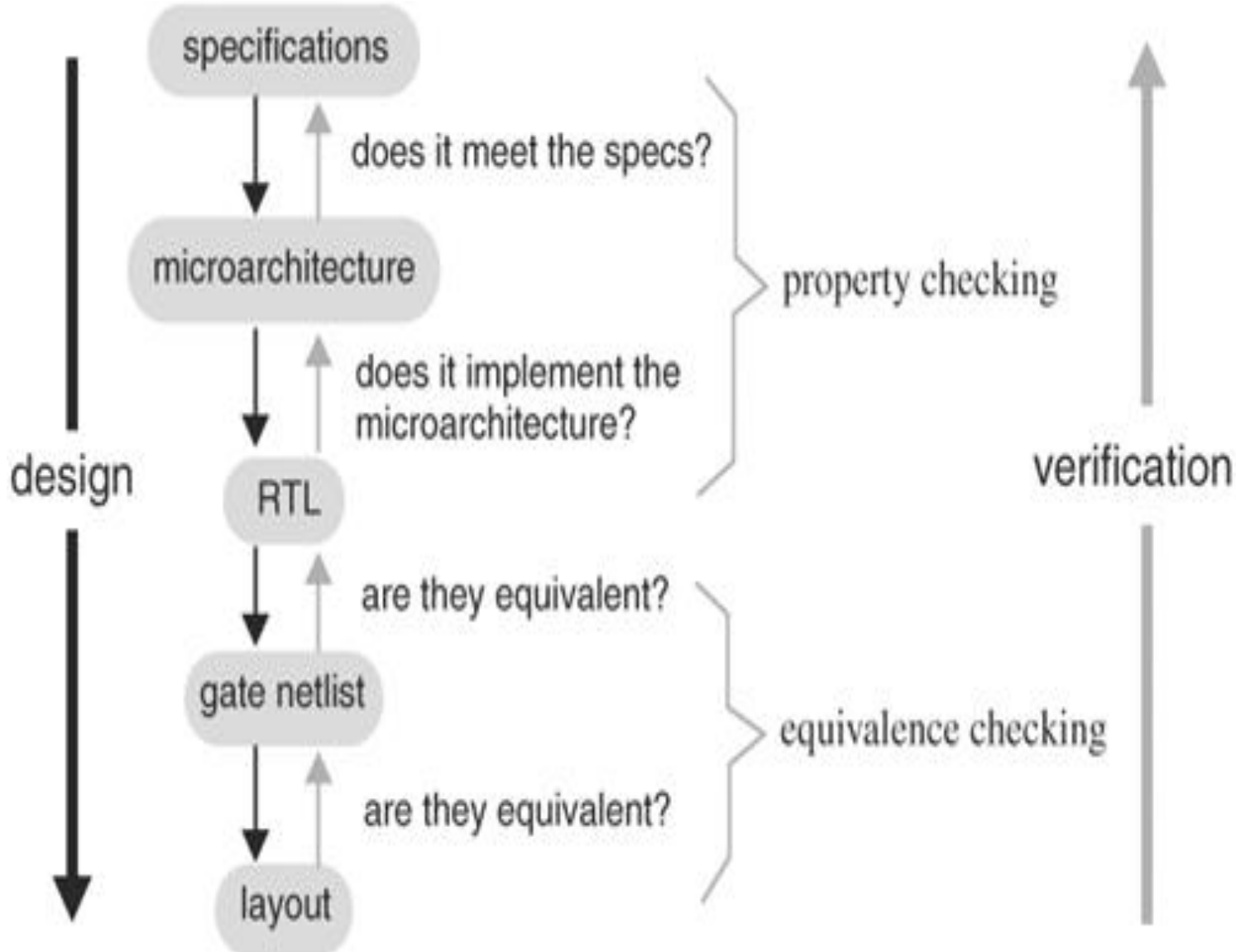
Layout verification is an activity to ensure that the layout of the design indeed matches the schematic that was actually verified in the timing verification above

Manufacturing verification

Manufacturing verification usually involves making sure that verification the device was manufactured correctly and no flaws were introduced as a result of the manufacturing process. During this process, the device that has been fabricated is tested using a tester which apply patterns to the device that has been fabricated. These patterns are determined using the gate simulations and ATPG tools.

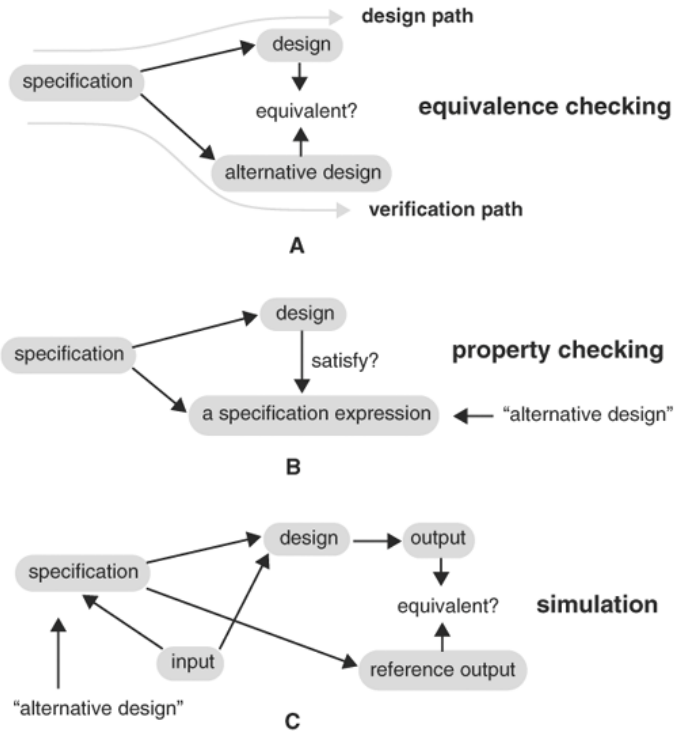
The difference between testing and verification

One important question often asked is the difference between testing and verification. Verification is the activity that is used to verify that the design does indeed meet its specified intent. Testing on the other hand is used to ensure that the design is indeed manufactured properly. Testing is accomplished by the use of test patterns that are generated using a process termed as ATPG (Automatic Test Pattern Generation). Verification on the other hand involves the generation of test vectors which are used to ensure that the design meets the specification.


specifications
does it meet the specs?
microarchitecture
does it implement the microarchitecture?
RTL
are they equivalent?
gate netlist
are they equivalent?
layout
design
property checking
equivalence checking
verification

There are **two types of design error**. The first type of error exists not in the specifications but **in the implementations, and it is introduced during the implementation process**. An example is **human error in interpreting design functionality.** To prevent this type of error, we can use a **software program** to synthesize an implementation directly from the specifications. Although this approach eliminates most human errors, errors can still result from **bugs in the software program**, or usage errors of the software program may be encountered. Furthermore, this synthesis approach is rather limited in practice for two reasons. First, many specifications are in the form of casual conversational language, such as English, as opposed to a form of precise mathematical language, such as Verilog or C++. We know that automatic synthesis from a loose language is infeasible. In fact, as of this writing, there is no high-level formal language that specifies both functional and timing requirements. A reason for this is that a high-level functional requirement does not lend itself to timing requirements, which are more intuitive at the implementation level. Therefore, **timing requirements such as delay and power, when combined with high-level functional specifications,** are so overtly inaccurate that people relegate timing specifications to levels of lower abstraction. Second, even if the specifications are written in a precise mathematical language, few synthesis software programs can produce implementations that meet all requirements. **Usually, the software program synthesizes from a set of functional specifications but fails to meet timing requirements.**

Another method the more widely used method to uncover errors of this type is through redundancy. That is, the same specifications are implemented two or more times using different approaches, and the results of the approaches are compared. In theory, the more times and the more different ways the specifications are implemented, the higher the confidence produced by the verification. In practice, more than two approaches is rarely used, because more errors can be introduced in each alternative verification, and costs and time can be insurmountable. The design process can be regarded as a path that transforms a set of specifications into an implementation. The basic principle behind verification consists of two steps. During the first step, there is a transformation from specifications to an implementation. Let us call this step *verification transformation*. During the second step, the result from the verification is compared with the result from the design to detect any errors. This is illustrated in Figure 1.3 (A). Oftentimes, the result from a verification transformation takes place in the head of a verification engineer, and takes the form of the properties deduced from the specifications. For instance, the expected result for a simulation input vector is calculated by a verification engineer based on the specifications and is an alternative implementation.



The basic principle of design verification. (A) The basic methodology of verification by redundancy. (B) A variant of the basic methodology adapted in model checking. (C) Simulation methodology cast in the form of verification by redundancy.

Obviously, if verification engineers go through the exact same procedures as the design engineers, both the design and verification engineers are likely to arrive at the same conclusions, avoiding and committing the same errors. Therefore, the more different the design and verification paths, the higher confidence the verification produces. One way to achieve high confidence is for verification engineers to transform specifications into an implementation model in a language different from the design language. This language is called *verification language*, as a counterpart to design language. Examples of verification languages include Vera, C/C++, and *e*. A possible verification strategy is to use C/C++ for the verification model and Verilog/VHSIC Hardware Description Language (VHDL) for the design model.

During the second step of verification, two forms of implementation are compared. This is achieved by expressing the two forms of implementation in a common intermediate form so that equivalency can be checked efficiently. Sometimes, a comparison mechanism can be sophisticated for example, comparing two networks with arrival packets that may be out of order. In this case, a common form is to sort the arrival packets in a predefined way. Another example of a comparison mechanism is determining the equivalence between a transistor-level circuit and an RTL implementation. A common intermediate form in this case is a binary decision diagram.

Here we see that the classic simulation-based verification paradigm fits the verification principle. A simulation-based verification paradigm consists of four components: the circuit, test patterns, reference output, and a comparison mechanism. The circuit is simulated on the test patterns and the result is compared with the reference output. The implementation result from the design path is the circuit, and the implementation results from the verification path are the test patterns and the reference output. The reason for considering the test patterns and the reference output as implementation results from the verification path is that, during the process of determining the reference output from the test patterns, the verification engineer transforms the test patterns based on the specifications into the reference output, and this process is an implementation process. Finally, the comparison mechanism samples the simulation results and determines their equality with the reference output.

Verification through redundancy is a double-edged sword. On the one hand, it uncovers inconsistencies between the two approaches. On the other hand, it can also introduce incompatible differences between the two approaches and often verification errors. For example, using a C/C++ model to verify against a Verilog design may force the verification engineer to resolve fundamental differences between the two languages that otherwise could be avoided. Because the two languages are different, there are areas where one language models accurately whereas the other cannot. A case in point is modeling timing and parallelism in the C/C++ model, which is deficient. Because design codes are susceptible to errors, verification code is equally prone to errors. Therefore, verification engineers have to debug both design errors as well as verification errors. Thus, if used carelessly, redundancy strategy can end up making engineers debug more errors than those that exist in the designdesign errors plus verification errors resulting in large verification overhead costs.

Simulation-Based Verification

The most commonly used verification approach is simulation-based verification. As mentioned earlier, simulation-based verification is a form of verification by redundancy. The variant or the alternative design manifests in the reference output. During simulation-based verification, the design is placed under a test bench, input stimuli are applied to the test bench, and output from the design is compared with reference output. A test bench consists of code that supports operations of the design, and sometimes generates input stimuli and compares the output with the reference output as well. The input stimuli can be generated prior to simulation and can be read into the design from a database during simulation, or it can be generated during a simulation run. Similarly, the reference output can be either generated in advance or on the fly. In the latter case, a reference model is simulated in lock step with the design, and results from both models are compared.

Before a design is simulated, it runs through a linter program that checks static errors or potential errors and coding style guideline violations. A linter takes the design as input and finds design errors and coding style violations. It is also used to glean easy-to-find bugs. A linter does not require input vectors; hence, it checks errors that can be found independent of input stimuli. Errors that require input vectors to be stimulated will escape linting. Errors are static if they can be uncovered without input stimuli. Examples of static errors include a bus without a driver, or when the width of a port in a module instantiation does not match the port in the module definition. Results from a linter can be just alerts to potential errors. A potential error, for example, is a dangling input of a gate, which may or may not be what the designer intended. A project has its own coding style guidelines enforced to minimize design errors, improve simulation performance, or for other purposes. A linter checks for violations of these guidelines.

Next, input vectors of the items in the test plan are generated. Input vectors targeting specific functionality and features are called *directed tests*. Because directed tests are biased toward the areas in the input space where the designers are aware, bugs often happen in areas where designers are unaware; therefore, to steer away from these biased regions and to explore other areas, pseudorandom tests are used in conjunction with directed tests. To produce pseudorandom tests, a software program takes in seeds, and creates tests and expected outputs. These pseudorandomly generated tests can be vectors in the neighborhood of the directed tests. So, if directed tests are points in input space, random tests expand around these points.

After the tests are created, simulators are chosen to carry out simulation. A simulator can be an event-driven or cycle-based software or hardware simulator. An event simulator evaluates a gate or a block of statements whenever an input of the gate or the variables to which the block is sensitive change values. A change in value is called an *event*. A cycle-based simulator partitions a circuit according to clock domains and evaluates the subcircuit in a clock domain once at each triggering edge of the clock. Therefore, event count affects the speed a simulator runs. A circuit with low event counts runs faster on event-driven simulators, whereas a circuit with high event counts runs faster on cycle-based simulators. In practice, most circuits have enough events that cycle-based simulators out-perform their event-driven counterparts. However, cycle-based simulators have their own shortcomings. For a circuit to be simulated in a cycle-based simulator, clock domains in the circuit must be well defined. For example, an asynchronous circuit does not have a clear clock domain definition because no clock is involved. Therefore, it cannot be simulated in a cycle-based simulator.

A hardware simulator or emulator models the circuit using hardware components such as processor arrays and field programmable gate arrays (FPGAs). First, the components in a hardware simulator are configured to model the design. In a processor array hardware simulator, the design is compiled into instructions of the processors so that executing the processors is tantamount to simulating the design. In an FPGA-based hardware acceleration, the FPGAs are programmed to mimic the gates in the design. In this way, the results from running the hardware are simulation results of the design. A hardware simulator can be either event driven or cycle based, just like a software simulator. Thus, each type of simulator has its own coding style guidelines, and these guidelines are more strict than those of software simulators. A design can be run on a simulator only if it meets all coding requirements of the simulator. For instance, statements with delays are not permitted on a hardware simulator. Again, checking coding style guidelines is done through a linter.

The quality of simulating a test on a design is measured by the coverage the test provides. The coverage measures how much the design is stimulated and verified. A coverage tool reports code or functional coverage. Code coverage is a percentage of code that has been exercised by the test. It can be the percentage of statements executed or the percentage of branches taken. Functional coverage is a percentage of the exercised functionality. Using a coverage metric, the designer can see the parts of a design that have not been exercised, and can create tests for those parts. On the other hand, the user could trim tests that duplicate covered parts of a circuit.

When an unexpected output is observed, the root cause has to be determined. To determine the root cause, waveforms of circuit nodes are dumped from a simulation and are viewed through a waveform viewer. The waveform viewer displays node and variable values over time, and allows the user to trace the driver or loads of a node to determine the root cause of the anomaly. When a bug is found, it has to be communicated to the designer and fixed. This is usually done by logging the bug into a bug tracking system, which sends a notification to the owner of the design. When the bug is logged into the system, its progress can be tracked. In a bug tracking system, the bug goes through several stages: from opened to verified, fixed, and closed. It enters the opened stage when it is filed. When the designer confirms that it is a bug, he moves the bug to the verified stage. After the bug is eradicated, it goes to the fixed stage. Finally, if everything works with the fix, the bug is resolved during the closed stage. A bug tracking system allows the project manager to prioritize bugs and estimate project progress better.

Design codes with newly added features and bug fixes must be made available to the team. Furthermore, when multiple users are accessing the same data, data loss may result (for example, two users trying to write to the same file). Therefore, design codes are maintained using revision control software that arbitrates file access to multiple users and provides a mechanism for making visible the latest design code to all.

The typical flow of simulation-based verification is summarized in Figure 1.4. The components inside the dashed enclosure represent the components specific to the simulation-based methodology. With the formal verification method, these components are replaced by those found in the formal verification counterparts.

Simulation-Based Verification versus Formal Verification

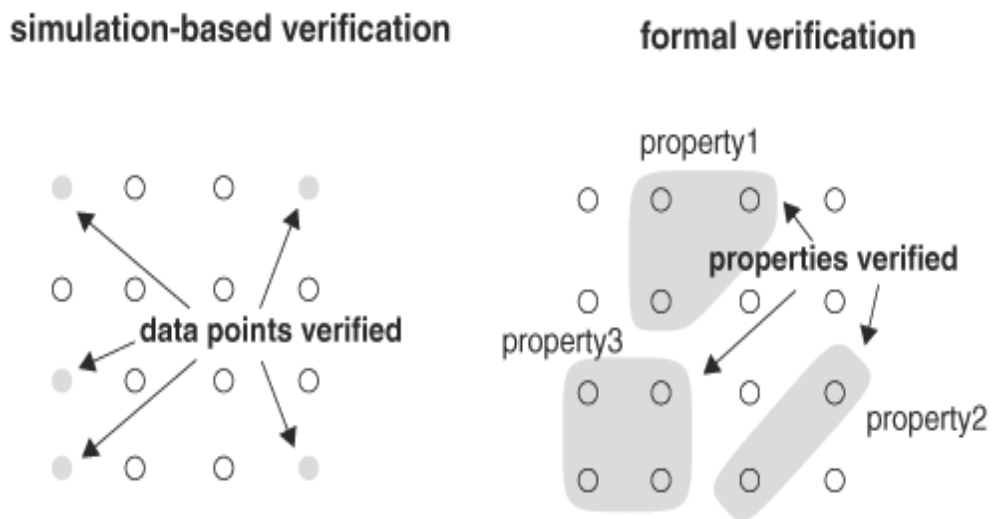
The most prominent distinction between simulation-based verification and formal verification is that the former requires input vectors and the latter does not. The mind-set in simulation-based verification is first to generate input vectors and then to derive reference outputs. The thinking process is reversed in the formal verification process. The user starts out by stating what output behavior is desirable and then lets the formal checker prove or disprove it. Users do not concern themselves with input stimuli at all. In a way, the simulation-based methodology is input driven and the formal methodology is output driven. It is often more straightforward to think in the input-driven way, and this tendency is reflected in the perceived difficulty in using a formal checker.

Another selling point for formal verification is completeness, in the sense that it does not miss any point in the input spacea problem from which simulation-based verification suffers. However, this strength of formal verification sometimes leads to the misconception that once a design is verified formally, the design is 100% free of bugs. Let's compare simulation-based verification with formal verification and determine whether formal verification is perceived correctly.

Simulating a vector can be conceptually viewed as verifying a point in the input space. With this view, simulation-based verification can be seen as verification through input space sampling. Unless all points are sampled, there exists a possibility that an error escapes verification. As opposed to working at the point level, formal verification works at the property level. Given a property, formal verification exhaustively searches all possible input and state conditions for failures. If viewed from the perspective of output, simulation-based verification checks one output point at a time; formal verification checks a group of output points at a time (a group of output points make up a property).

With this perspective, the formal verification methodology differs from the simulation-based methodology by verifying *groups* of points in the design space instead of *points*. Therefore, to verify completely that a design meets its specifications using formal methods, it must be further proved that the set of properties formally verified collectively constitutes the specifications. The fact that formal verification checks a group of points at a time makes formal verification software less intuitive and thus harder to use.

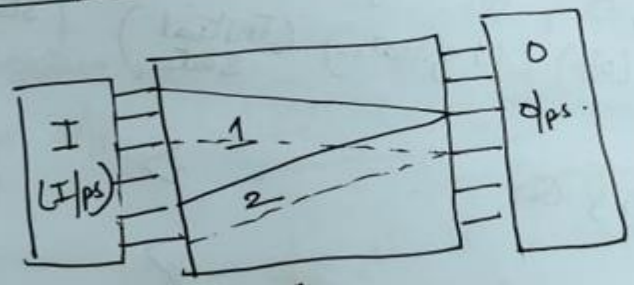
A major disadvantage of formal verification software is its extensive use of memory and (sometimes) long runtime before a verification decision is reached. When memory capacity is exceeded, tools often shed little light on what went wrong, or give little guidance to fix the problem. As a result, formal verification software, as of this writing, is applicable only to circuits of moderate size, such as blocks or modules.


simulation-based verification
data points verified
formal verification
property1
properties verified
property3
property2

Formal Verification

- **Equivalence Checking**
  - Original Model (Specification Model)
  - Final Model (Synthesized / optimized)
    - Sometimes synthesized model is run thro' the optimizer and we check for equivalence
- **Model Checking (Properties)**
  - We check whether the model satisfies the properties
- **Theorem Proving Approach**
  - We try to prove the implementation is equivalent to the specification

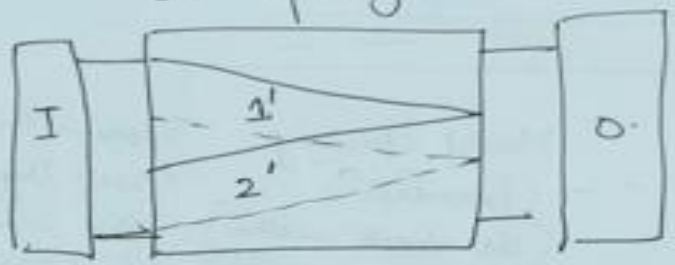
Logic Equivalence Checking



Implementing some logic

Reference design (Golden Design)

On Optimization



Equivalence Checking

If $1 \cong 1'$?

$2 \cong 2'$?

~~Equivalence of FSM~~

FSM Equivalence

$M : \langle I, O, Q, Q_0, F, H \rangle$ (FSM)

(I/P) (O/P) (Set of states) (Initial state) (State Transition fn) (O/P fn mapping state to O/P)

$M_1 : \langle I, O_1, Q_1 \text{ - - - } \rangle$

$M_2 : \langle I, O_2, Q_2 \text{ - - - } \rangle$

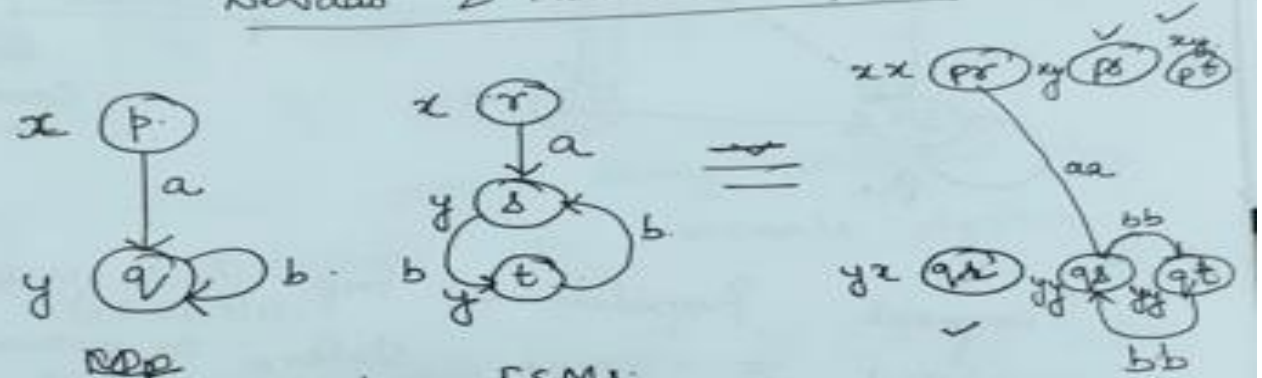
$M_1 \times M_2 : \langle I, O_1 \times O_2, Q_1 \times Q_2, Q_{01} \times Q_{02}, F_1 \times F_2, H_1 \times H_2 \rangle$

product FSM

Does $M_1 \times M_2$ give same set of O/ps for all the I/ps

for same i/ps
M1 X M2 should give same o/p

Devdas & Newton 1987



If these two FSMs are equivalent?

Here we define the o/ps as the ones

Final States : States which have diff o/ps (v)

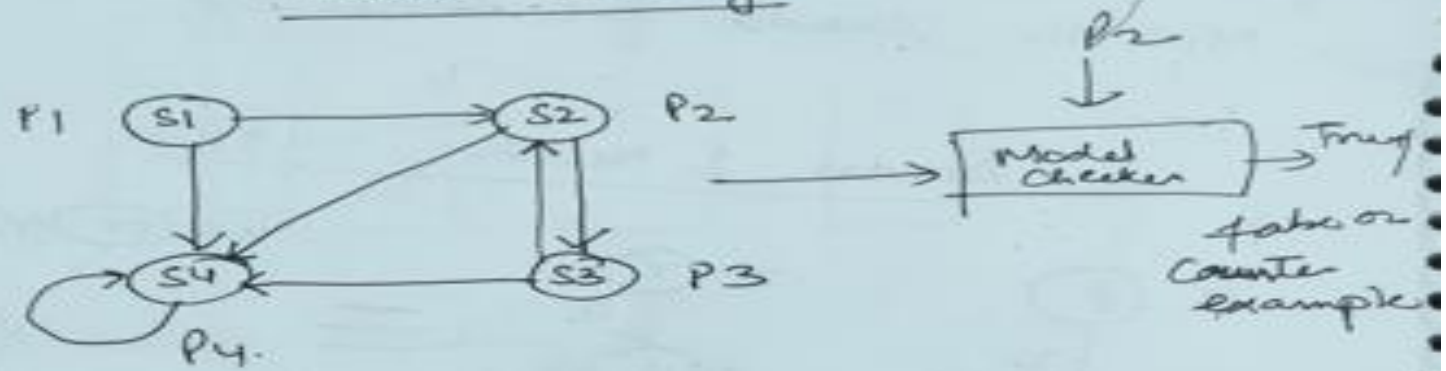
Dissimalar o/p State : (ps), (pt), (qr)

Danger ps But won't be encountered

for same i/ps, o/p shud be same — Basic Idea

Since in the above chr for same i/p we get same o/p.
↳ Equivalent chs

Model Checking



Temporal Properties - Prop (P1/P2/P3/P4) which are associated ∈ some states ∧ or overall systems (P)

P = P2 will eventually lead ∈ P4.

If P2 is satisfied at any pt of time then ultimately in this machine P4 is also satisfied.

These properties are presented ∈ the model checker.

Typical example -

o/p of a JK ff will give a seq. 101 at some pt of time

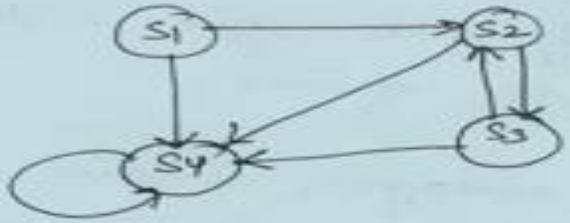
D ff will always remain in state of 1.

Are those prop. being satisfied by other properties or overall system.

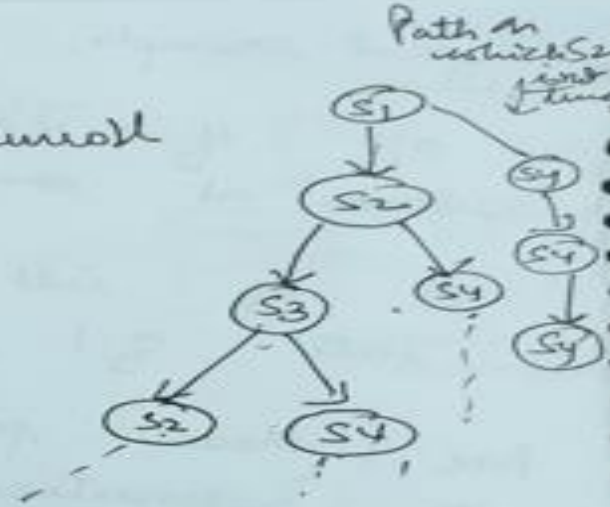
Properties on all paths P (A.P) is true
on some paths P will be true (E,P)

A for all path
E there exists some paths
G - Globally
F - Eventually

Always P (G,P) □P
Eventually P (F,P) ◇P



unroll



Looking at Properties

$\langle A, S_2 \rangle$ — False { On all paths, S_2 — Not true }

$\langle E, S_2 \rangle$ — True { on some paths S_2 }

~~Always~~ S4 succeeds S_2 — { Not true

Eventually S4 Not True

S 3 succeeds S_2 ✓

Eventually S4 — Not true

⇓

We are checking if the properties are true by unrolling the State Machine

But it leads to state space explosion

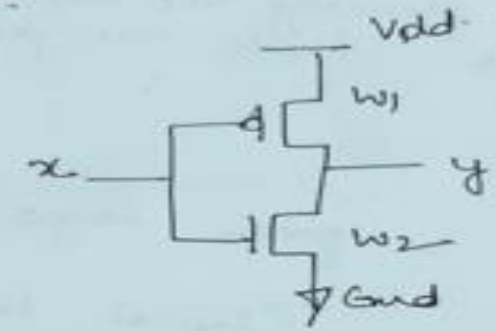
Heuristics involve pruning the tree & see which part of tree holds the properties

Theorem Proving

- Formula in a given logic for the specs.
  - (Typical logic can be)
    - First order Logic
    - Higher Order Logic
    - Temporal Logic etc
- Formula in a given logic for the implementation
- Assumptions about a particular domain (Typically Axioms or Background theories etc)

⇒ Proof (as o/p) that says spec ≡ implementation

Example:



Assumptions : Vdd (y) := (y = T)

Gnd (y) := (y = F)

N transistor (x, y1, y2)

:= (x → (y1 = y2))

likewise for P transistor

Implementation (x, y) := ∃ w1, w2, Vdd (w1)

∧ Ptrans (x, w, y)

∧ Ntrans (x, y, w2)

∧ Gnd (w2)

∧ - And.

∃ - for some cases (finding like)

Spec (x, y) := (y = ¬ x)

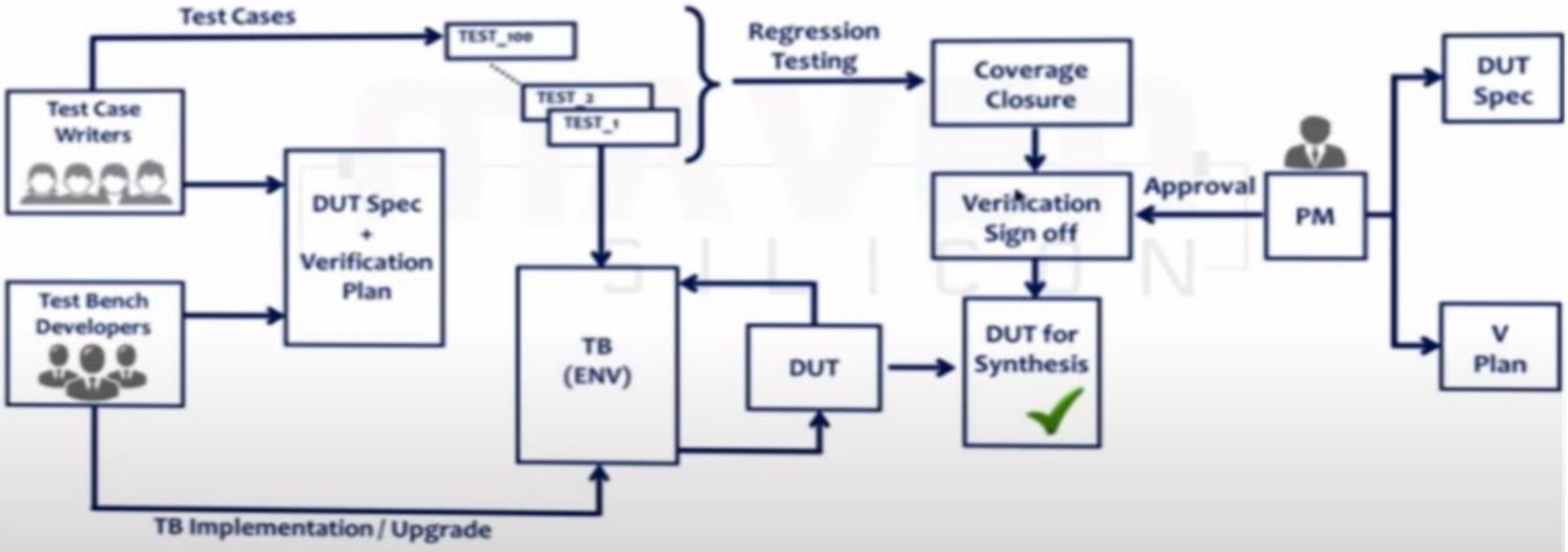
given the Assumptions and the Implementation we have to do the derivation and proof spec are holding

Descrip^{n} in this appen becomes very complex and hence not finding so much use in Industry

Not background in formal Logic
Limited scope of Automa^{n}

* Equivalence Checking is adopted

Verification Process



Concept of layered test Bench

- In this example:
- **Layer 1 (Stimulus generation):** Generates input stimuli such as clock, reset, and data signals.
- **Layer 2 (Monitor):** Observes signals of interest, such as inputs and outputs of the DUT, and displays them during simulation.
- **Layer 3 (Scoreboard):** Compares the actual outputs of the DUT with expected outputs and reports any mismatches.

```
`timescale 1ns / 1ps
module dffr (
    output reg Q,
    output reg QB,
    input d,
    input clk,
    input rst
);
    always @(posedge clk) begin
        if (rst)
            Q <= 1'b0;
        else
            Q <= d;
        QB <= ~Q;
    end
endmodule

// Layer 1: Stimulus generation
module stimulus;
    reg clk, rst, d;
    initial begin
        clk = 1'b0;
        forever #5 clk = ~clk;
    end
initial begin
        rst = 1'b1; // Initialize reset
        d = 1'b0;   // Input data
        #20 rst = 1'b0;
        #10 d = 1'b1;
        #20 $finish; units
    end
```

```
    // Instantiate DUT (Design Under Test)
    dffr dut (
        .Q(),
        .QB(),
        .d(d),
        .clk(clk),
        .rst(rst)
    );
endmodule

// Layer 2: Monitor
module monitor;
    reg Q, QB;
    reg clk, rst, d;
    initial begin
        $monitor("Time=%0t, clk=%b, rst=%b,
d=%b, Q=%b, QB=%b", $time, clk, rst, d, Q,
QB);
    end

    // Connect DUT outputs to monitor
    always @* begin
        Q = 0; // Initialize
        QB = 0;
    end
endmodule
```

```
// Layer 3: Scoreboard
module scoreboard;
    reg Q_exp, QB_exp;
    reg Q, QB;
    initial begin
        // Expected outputs
        Q_exp = 1'b0;
        QB_exp = 1'b1;
 // Compare expected and actual outputs
        $assert(Q == Q_exp) else $error("Error: Q mismatch");
        $assert(QB == QB_exp) else $error("Error: QB mismatch");
    end
Endmodule


// Top-level testbench
module testbench;
    // Instantiate layers
    stimulus stim();
    monitor mon();
    scoreboard sb();
endmodule
```