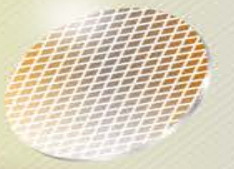


Design Verification Technologies Overview

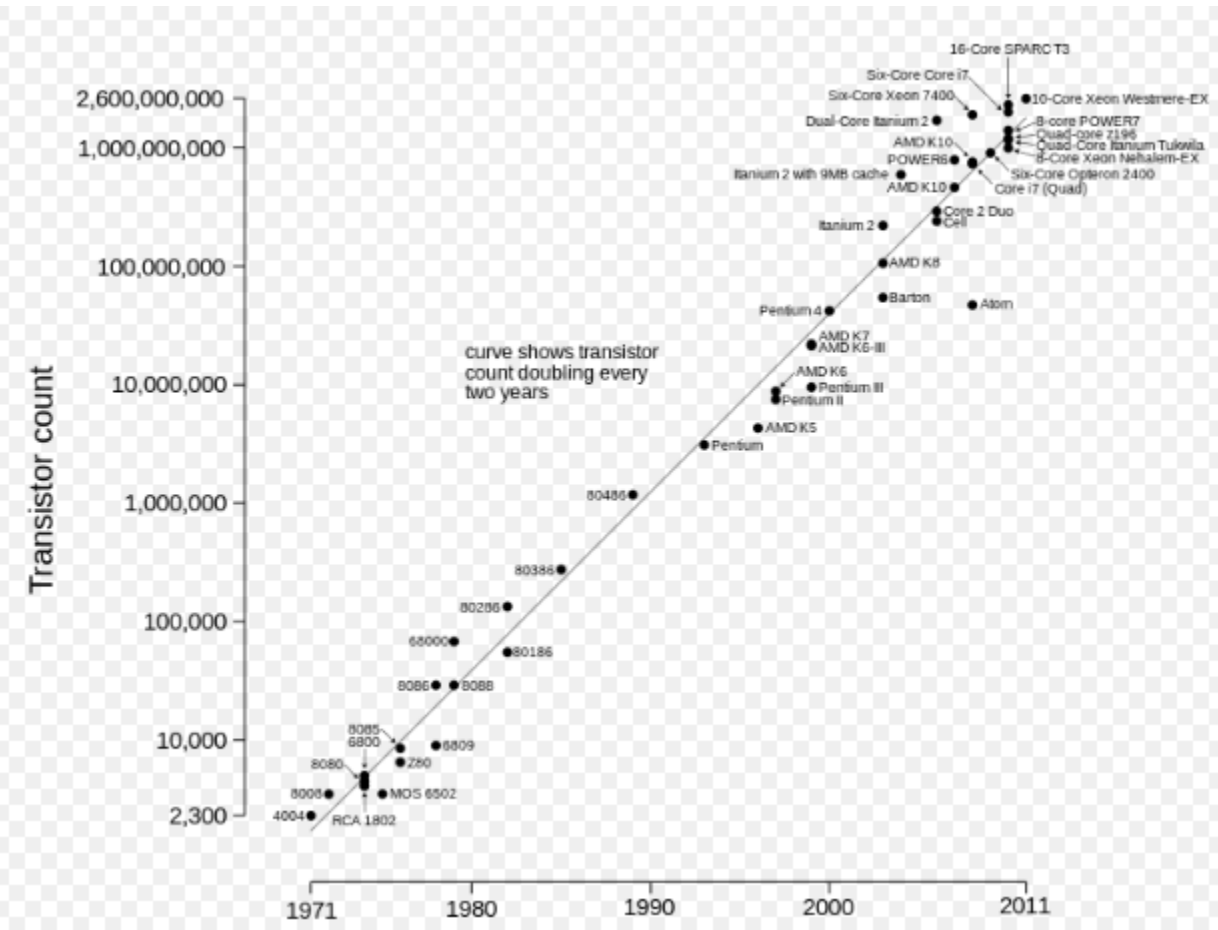
Yean-Ru Chen

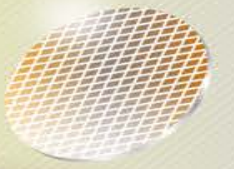
chenyr@mail.ncku.edu.tw



Design complexity increase

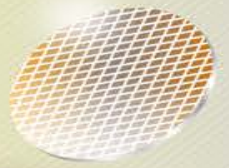
- Famous Moore's Law:
of transistors:
double/2 years
(proposed by G. Moore)
→
double/1.5 years
(proposed by D. House)





The facts result in...

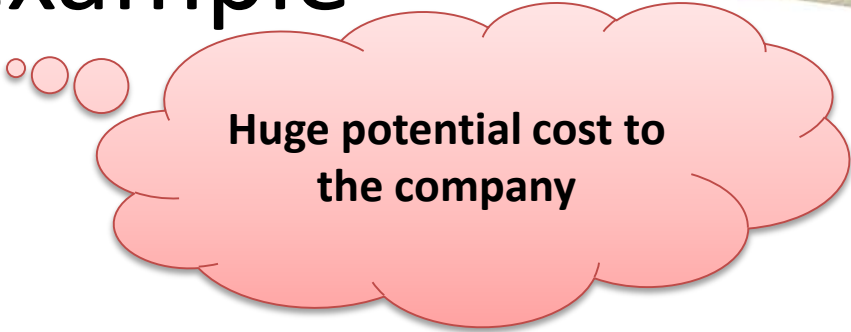
- “Verification” is the biggest bottleneck in the current design flow. It usually takes about 70% of the resources in the flow.
- To software, fixing a bug by releasing a patch after the product sold is not difficult.
- But to hardware, there is very little tolerance for a bug existing in a chip!!
 - No patch
 - Usually cause huge damage (\$)



A famous but sad example

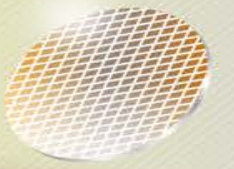
- Pentium FDIV Bug

- https://zh.wikipedia.org/wiki/Pentium_FDIV_bug



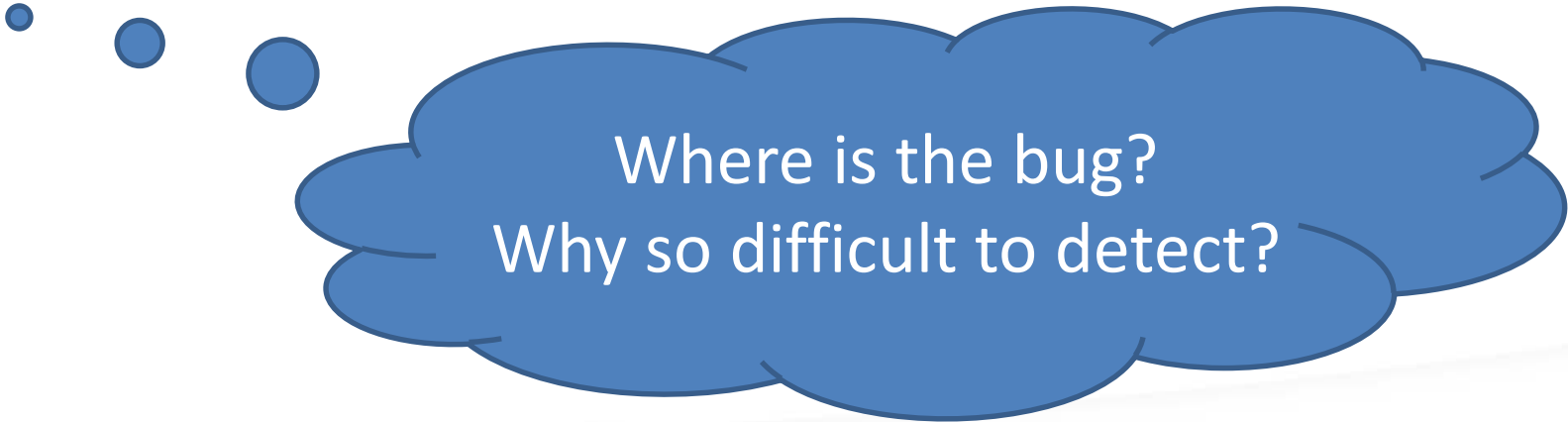
Huge potential cost to
the company

- Pentium FDIV bug（奔騰浮點除錯誤）是英特爾公司的舊版本Pentium浮點運算器的一個錯誤。錯誤起源於奔騰系列的FDIV（浮點除）指令。
- 發現
- 1994年10月，美國弗吉尼亞州Lynchburg College數學系教授Thomas Nicely發現用電腦處理長除法時一直出錯。他用一個數字去除以824,633,702,441時，答案一直是錯誤的。事後發現原因是英特爾為了加速運算，將整個乘法表燒錄在處理器上面，但是2048個乘法數字中，有5個輸入錯誤。這些錯誤其實不容易顯現，在運算過程中，它會自動修復錯誤，只有幾個二進位的數字組，才會造成完全錯誤的結果。
- 影響
- 根據工程師指出，大約90億個長除法中會有一個錯誤。依照計算，那個MTBF（平均故障間隔）時間，大概是七百年發生一次，所以幾乎是不可能發生。但是同樣有人聲稱實際上遭遇到這個錯誤的頻率要高得多。英特爾公司後來召回了有缺陷的產品。

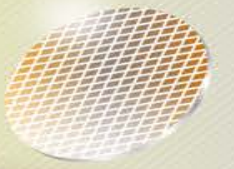


And then...

- Intel hired 1000+ PhDs for strategic CAD research, especially in the design verification area.
- However, there were still several bugs found in later Pentium chips
 - e.g. CMPXCHG8B instruction, Dan-0411, ... and etc.

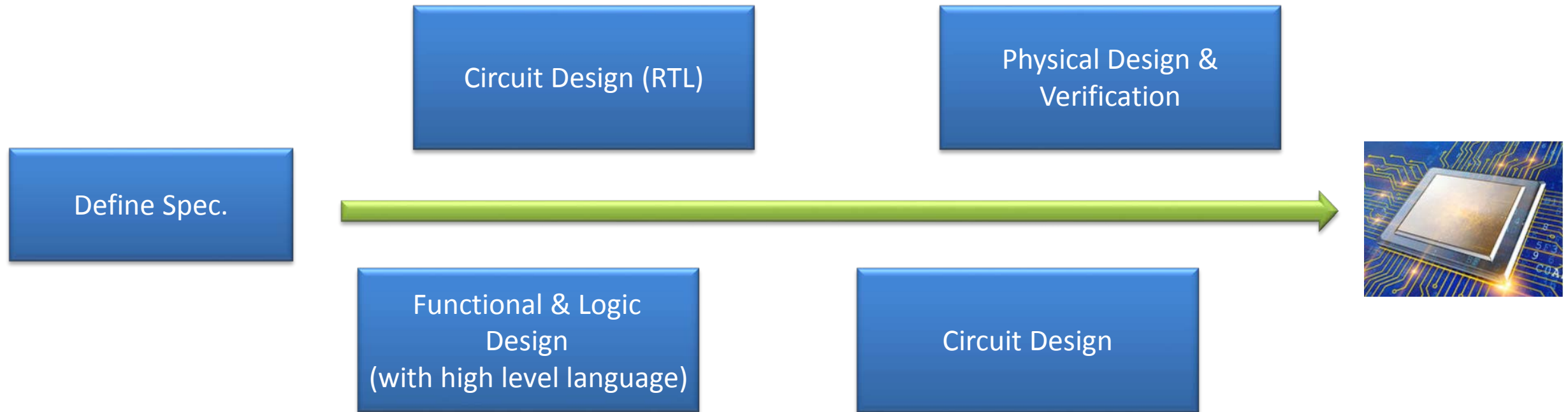


Where is the bug?
Why so difficult to detect?



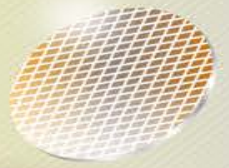
Typical design flow

- Let's look at typical design flow first...



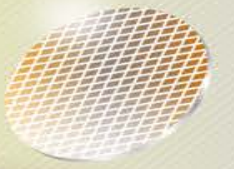
However, bugs can be anywhere...

Find as many bugs as possible! Find them as early as possible!



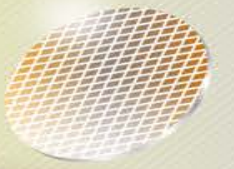
Try to find ALL bugs by simulation

- Use “simulation” approach...
 - Apply input pattern, exam output response
 - Suppose a circuit has 100 inputs (this is considered tiny in modern design)
 - Total number of input combinations: $2^{100} = 10^{30} = 10^{24}\text{M}$ (i.e. 10^{24} 百萬)
 - Requires (in the worse case) 10^{24}M test patterns to exhaust all the input scenarios
- For an 1 MIPS simulator (每秒百萬指令)
 - runtime = 10^{24} seconds = $3 * 10^{16}$ years



What is Verification?

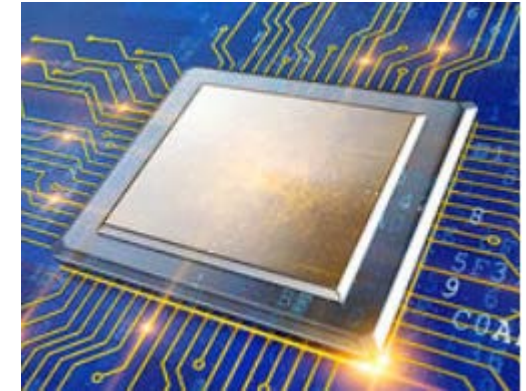
- Generally speaking, verification also covers
 - Functional verification
 - Timing verification
 - Physical verification
 - Manufacturing defect testing, etc
- Strictly speaking, verification is usually referred only to *“functional verification”*.



What is functional verification?



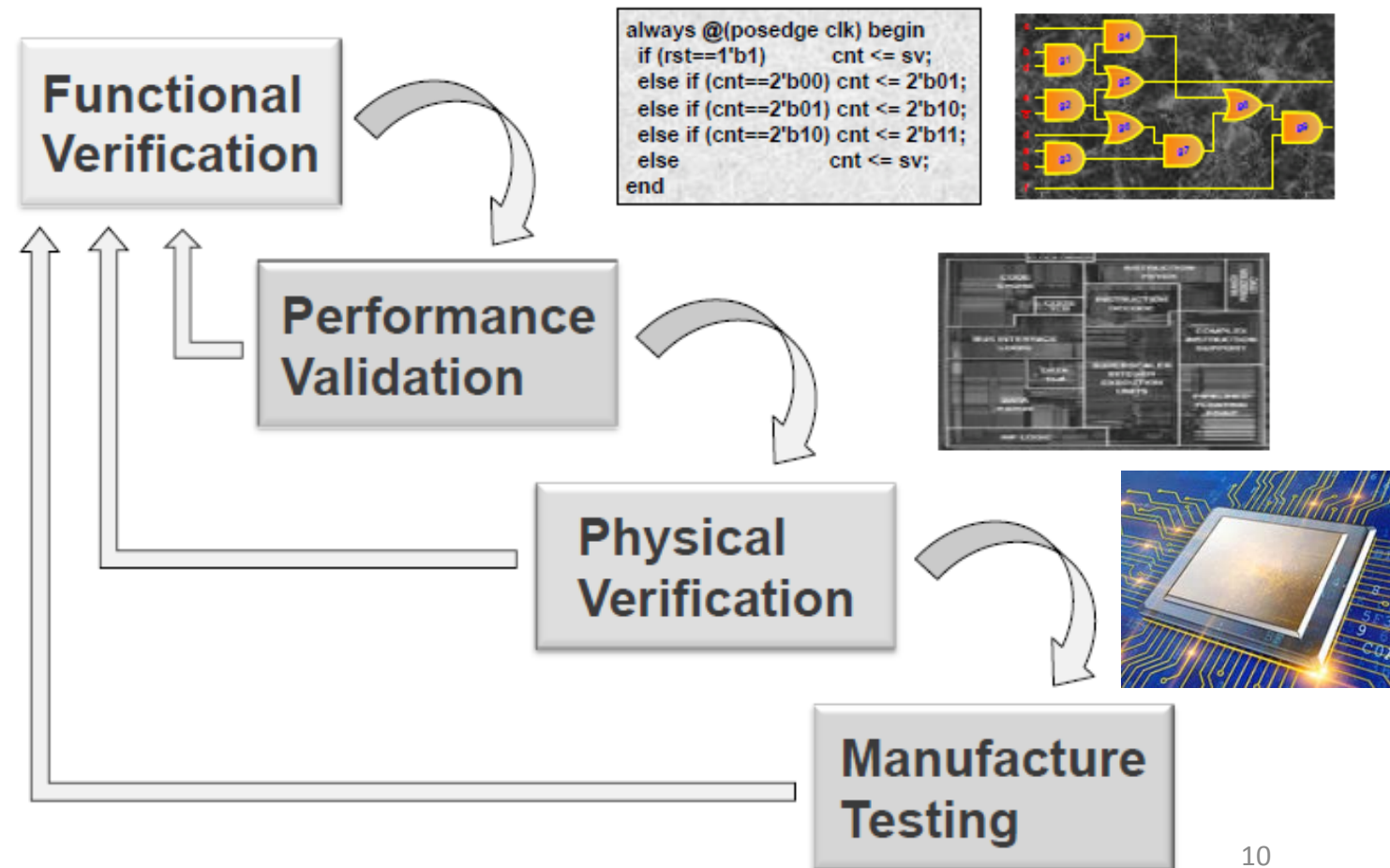
- Functionally correctness
- Spec fully implemented

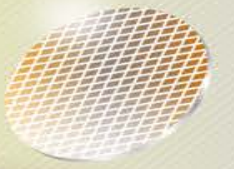




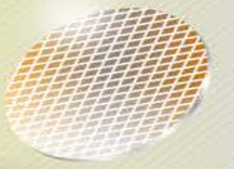
Four verification tasks in SoC verification flow

- Usually, functional error had the max %
 - More than 70% errors are related to functionality.

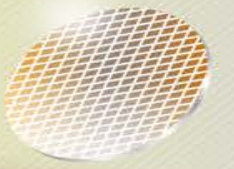




From this on,
we will focus on
“functional verification” only.

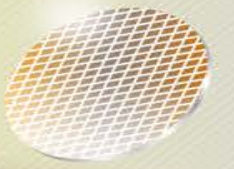


Functional Verification Overview



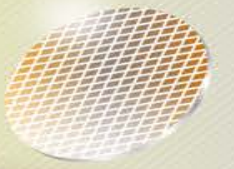
Agenda

- Simulation-based techniques
- Assertion-based verification
- Emulation
- FPGA prototyping
- Virtual prototyping
- Model checking (so-called Property checking)
- Formal equivalence checking
- Theorem proving
- Semi-formal verification



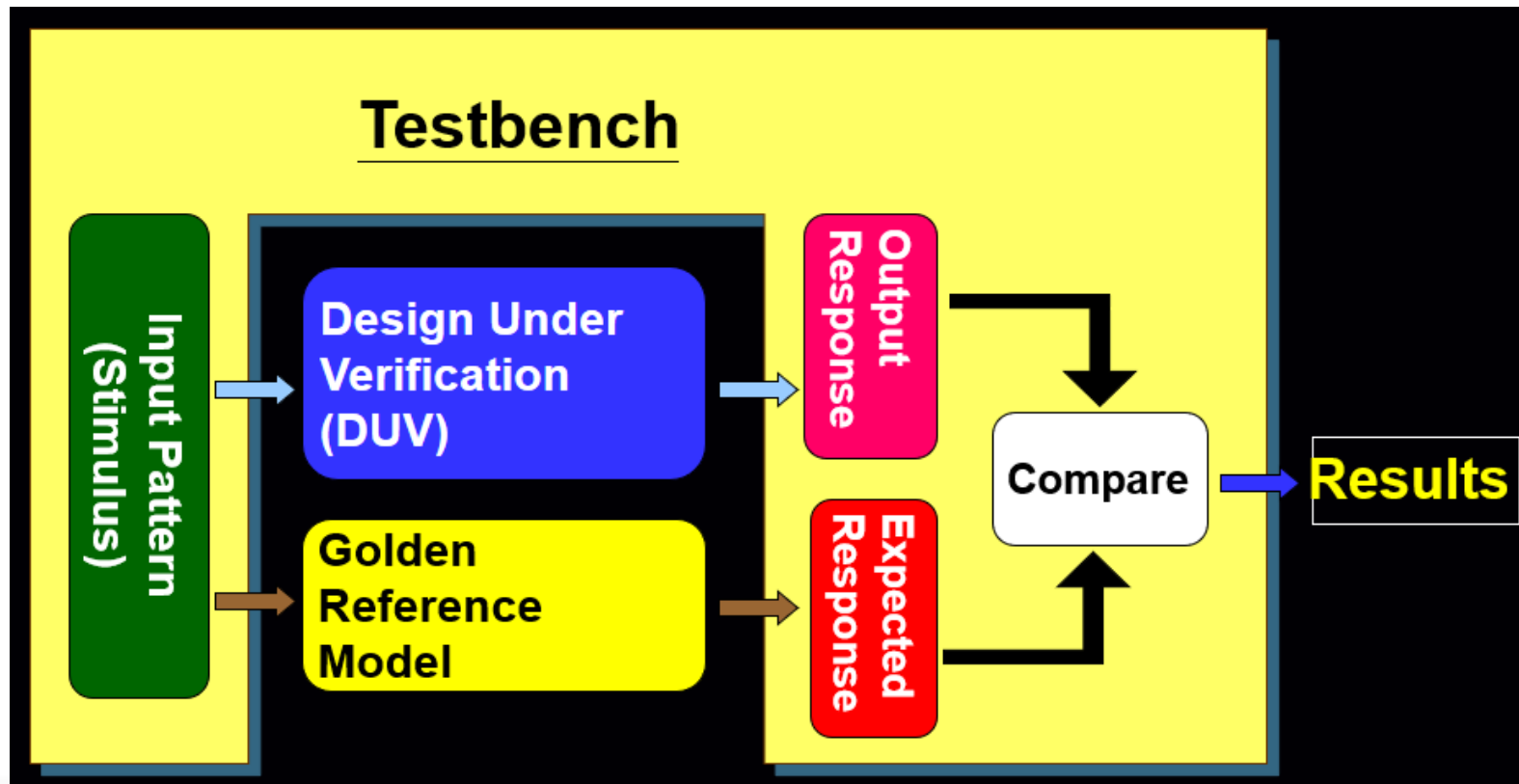
Agenda

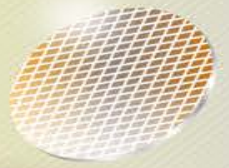
- Simulation-based techniques
- Assertion-based verification
- Emulation
- FPGA prototyping
- Virtual prototyping
- Model checking (so-called Property checking)
- Formal equivalence checking
- Theorem proving
- Semi-formal verification



Simulation-based techniques

- Most people verify their designs by simulation and debug by examining the output responses.





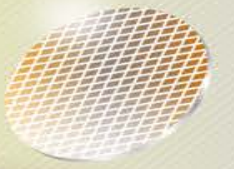
Sample SystemVerilog testbench

Test For Simple Request by CPU0

To test that simple requests are handled correctly for CPU0, drive bit 0 of the request signal and then monitor bit 0 of the grant signal. Finally, de-assert both bits of the request signal and check that both bits of the grant signal are properly de-asserted.

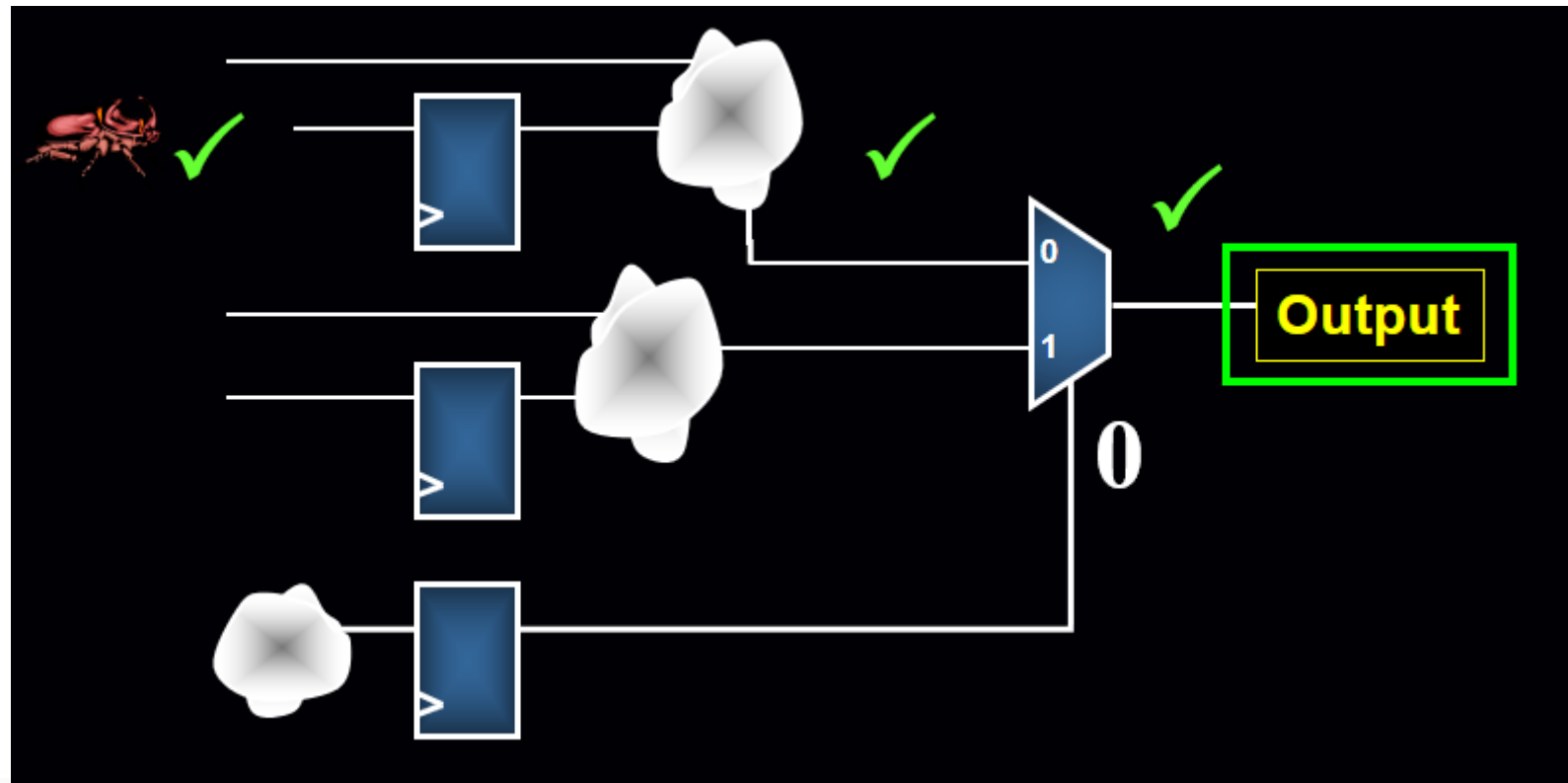
```
@(posedge clk) request_p <= 2'b01;
@(posedge clk);
expect(@(posedge clk) grant_p == 2'b01);
@(posedge clk) request_p <= 2'b00;
@(posedge clk);
expect(@(posedge clk) grant_p == 2'b00);
```

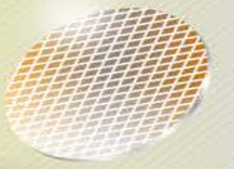
Ref: VCS/VCSi System Verilog Testbench Tutorial, 2005, SNPS



Observability problem

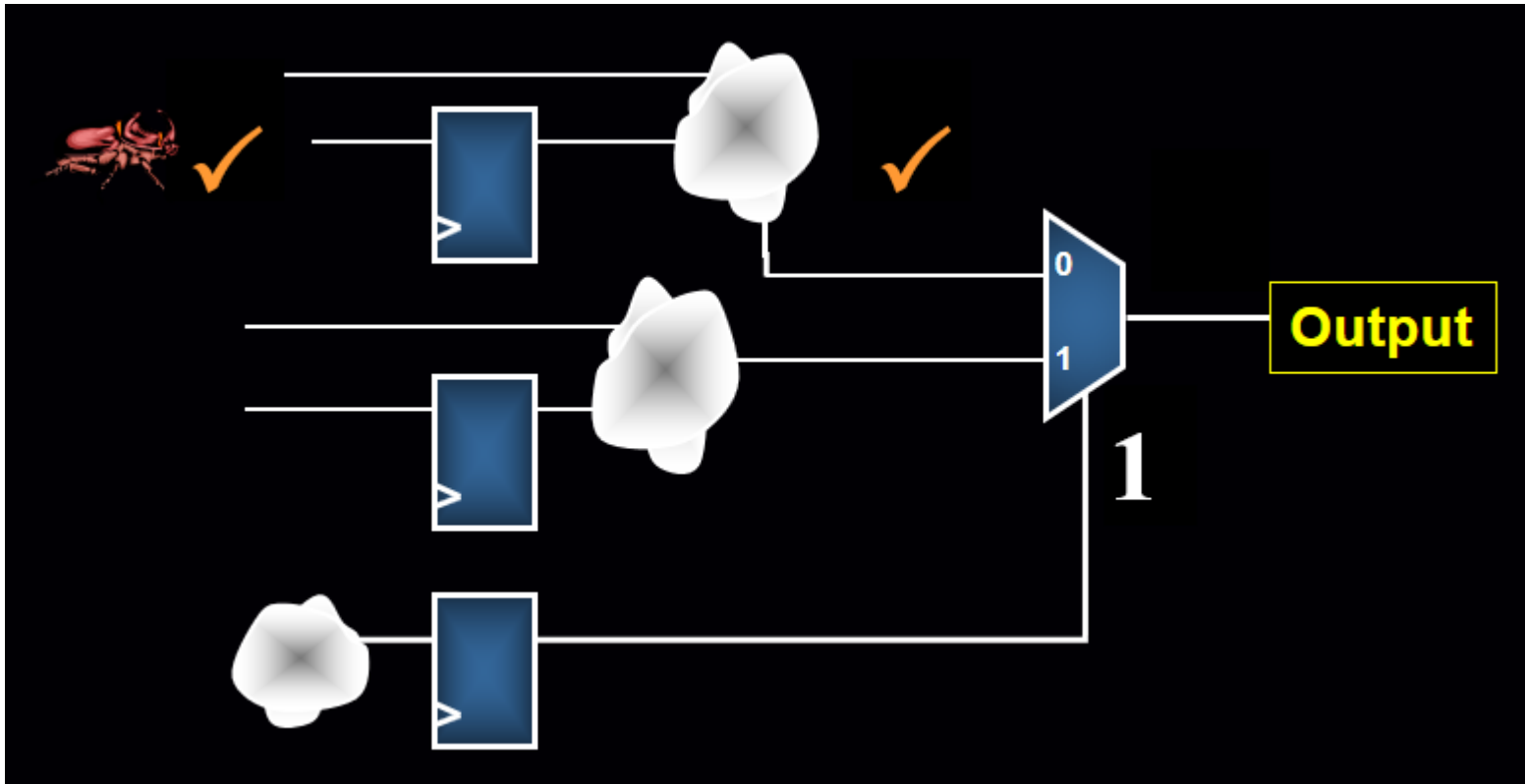
- Bugs are detected at the observation points, like primary outputs and registers. For example, bug can be detected at output:

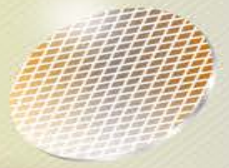




Observability problem

- For example, bug can not be detected at output:





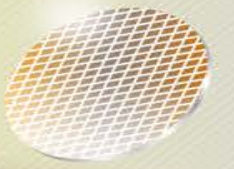
However...

- Remember, our DUV now is NOT yet an actual hardware; it is a software code!!
- Let's look at the differences between design verification and manufacturing testing.

Verification vs. Testing

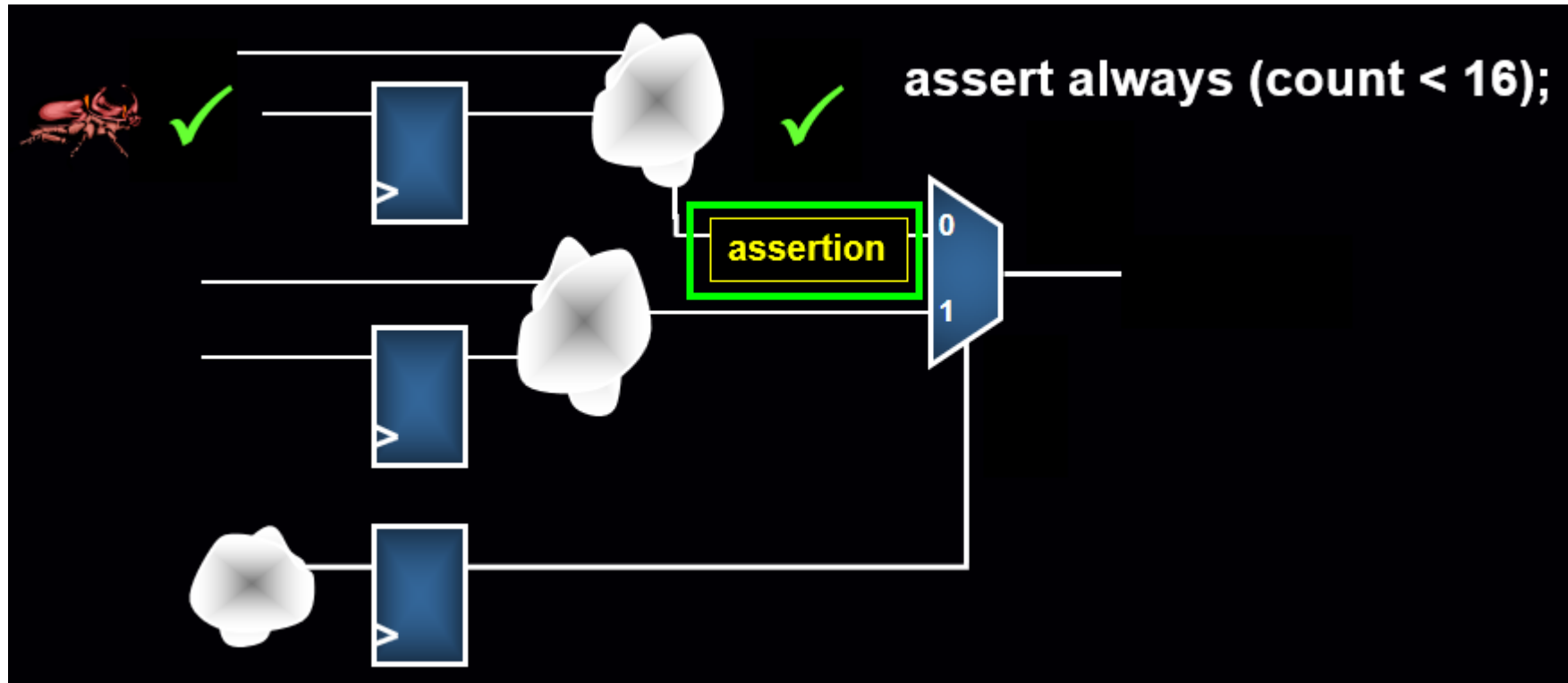
	Verification	Testing
Objective	Design (SW)	Chip (HW)
Environment	Simulator, debugger (tools)	Test equipments (HW)
Observation points	Any signal in the design	Chip outputs

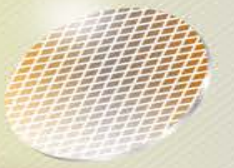
- Unlike testing, ideally, verification does not have the observability problem



Observability problem

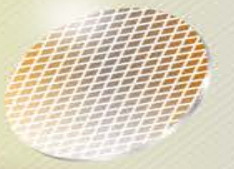
- Bug can be detected by putting assertion at suitable place:





Increase the observability of the bugs by using assertion

- Insert “*assertions*” in the middle of the circuit, and flag the simulator whenever the violation occurs
 - Increase the observability of the bugs.



Assertion-based verification (ABV)

- Assertions are not new.
- Designers frequently add verification code in Verilog designs

```
//synopsys translate_off  
    if(full & new_req) $display ("Error: buffer overflow.");  
//synopsys translate_on
```

- VHDL includes the keyword assert
 - [label] assert VHDL_expression [report message][severity level]



What is an Assertion?

- A concise description of complex behavior
- SystemVerilog Assertion Syntax:

Property Label: The user-defined name of the property. This name will be shown in the Property Table

`output_no_underflow:`

Verification Directive: Instructs the compiler that this property is an assertion, i.e. you expect the expression to be true at all times

`assert property (`

Clocking: Defines when the property is evaluated/sampled

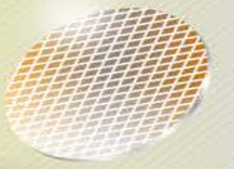
`@(posedge clk)`

`disable iff (!rstn)`

Disable Condition: When to ignore/abort evaluation of the property. Optional

`!(read && empty)`

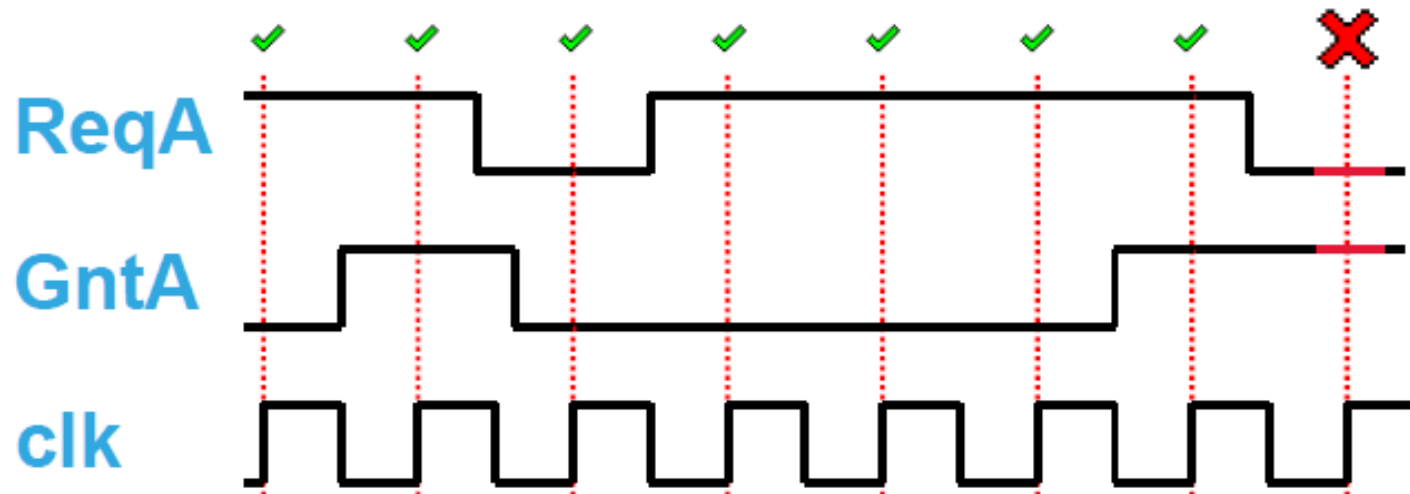
`);`
Property Expression: In the case of an assertion, this is what you expect to be true at all times. In this example, the assertion would fire if `read` and `empty` are high at the same time, which would make `!(read && empty)` evaluate to false

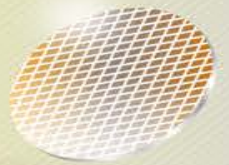


SystemVerilog Assertion example

- Should never see a Grant without a Request

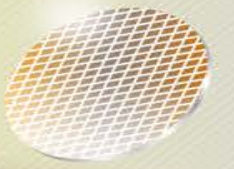
```
no_GntA_without_ReqA: assert property (  
    @(posedge clk) disable iff (!rstn)  
    !(GntA && !ReqA)  
);
```





Complex or simple assertions?

- We will see by some examples that we can describe very complex design properties by using assertion specification languages
 - However, this may scare off many users...
 - Another bigger problem: how do you know you write the correct property?
 - What if the property you wrote is wrong?
- Assertions should be simple!!
 - The main purpose should be finding bugs
 - Most assertions needed in the design are very simple assertions
 - IBM research founds that over 70% properties mostly used in verification are simple ones!

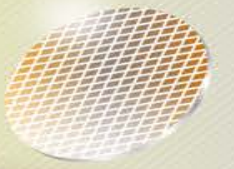


When to stop simulation?

- **Coverage Metrics**

DUT: usually RTL or up

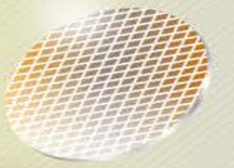
- A quantitative measure to assess the quality of a test suite
 - [Assume] Test completeness or verification confidence is proportional to the simulation coverage on certain attribute of the design
- Can also be a hint to generate more vectors on the area where the test is not sufficient
- Examples:
 - Line (statement), toggle, branch, expression, path
 - FSM state, transitions ... and etc.
- What can we do if the coverage is not “good” to your boss?



Wondering...

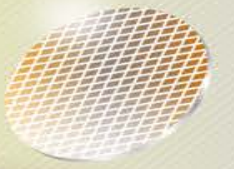
- Then why is simulation still the mainstream approach in verification?
- How can it be useful?
 - **Simulation is useful because...**
 - In early design cycle, most bugs are easy to find
 - Like something contaminates the ocean...
 - Use “design intent” to guide the testbench
 - Guided random test pattern generation
 - Real-life stimulus

Key point is:
Simulation approach is
easier to learn, simpler to
use, so it is the most popular.

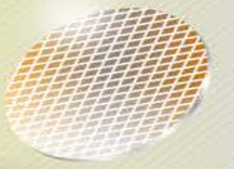


Agenda

- Simulation-based techniques
- Assertion-based verification
- Emulation
- FPGA prototyping
- Virtual prototyping
- **Model checking (so-called Property checking)**
- **Formal equivalence checking**
- Theorem proving
- Semi-formal verification

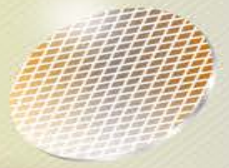


What is formal?



Formal verification

- A verification methodology to prove or disprove the correctness of intended design behaviors w.r.t. its golden specifications by using formal methods of mathematics.
- Different approaches of formal verification



Inductive proof (數學歸納證明法)

Prove: $\sum_{k=1}^n k = 1 + 2 + 3 + 4 + \dots + n = \frac{n(n+1)}{2}$

Step1: Show true for $n = 1$:

$$1 = \frac{(1)(1+1)}{2} \quad \text{so } 1 = 1.$$

Notice that it is also true for $n = 2$:

$$1 + 2 = \frac{(2)(2+1)}{2} \quad \text{so } 3 = 3.$$

Step 2: Suppose that it is true for $n = k$:

$$1 + 2 + 3 + \dots + k = \frac{k(k+1)}{2}$$

Step 3: Show it is true for $n = k + 1$:

$$1 + 2 + 3 + \dots + k + (k+1) = \frac{(k+1)(k+2)}{2}$$

Substitute step 2 in step 3 by replacing the first k terms with $\frac{k(k+1)}{2}$

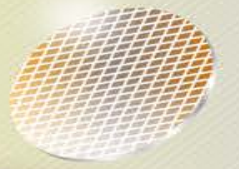
$$\text{This yields} \quad \frac{k(k+1)}{2} + (k+1) = \frac{(k+1)(k+2)}{2}$$

$$\text{Simplifying the left side:} \quad \frac{k(k+1)}{2} + \frac{2(k+1)}{2} = \frac{(k+1)(k+2)}{2}$$

$$\frac{k^2 + k + 2k + 2}{2} = \frac{(k+1)(k+2)}{2}$$

$$\frac{k^2 + 3k + 2}{2} = \frac{(k+1)(k+2)}{2}$$

$$\text{Factoring,} \quad \frac{(k+1)(k+2)}{2} = \frac{(k+1)(k+2)}{2}$$

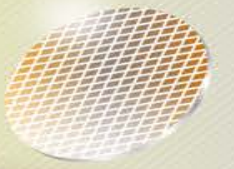


Deductive verification (演繹驗證)

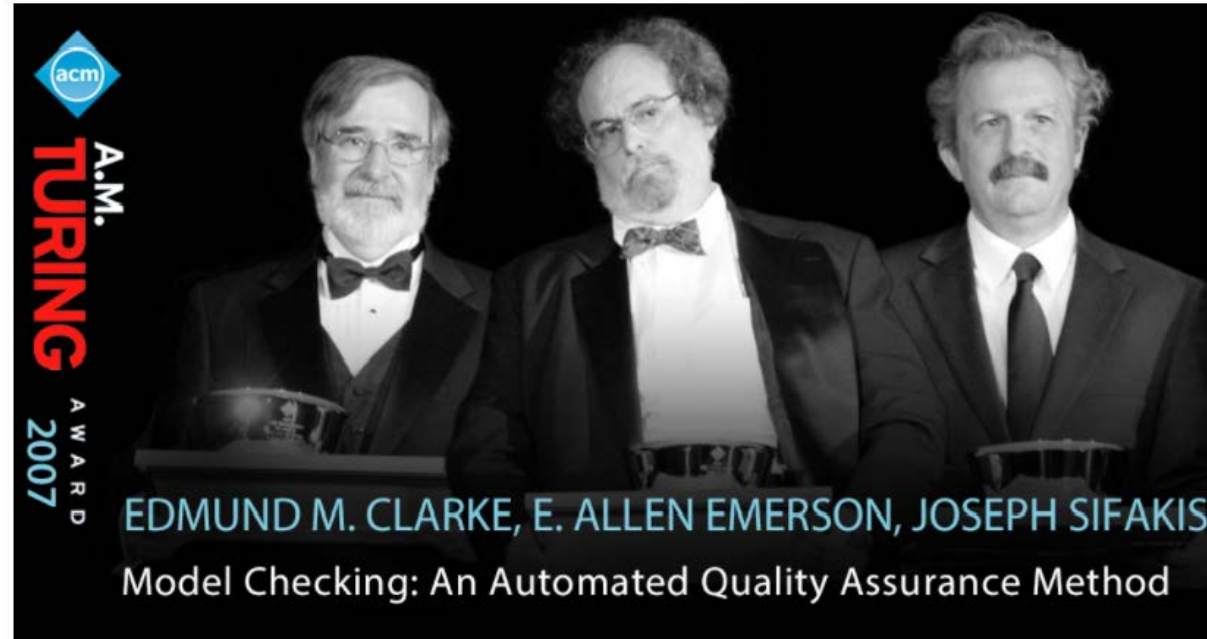
- Assume-guarantee reasoning

$$\frac{M_0 \parallel A \models P \quad M_1 \models A}{M_0 \parallel M_1 \models P}$$

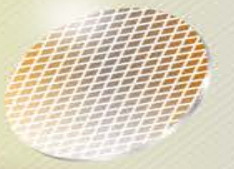
- Informally, if we can find an assumption A such that (1) $M_0 \parallel A$ satisfies P ; and (2) M_1 satisfies A , then $M_0 \parallel M_1$ satisfies P .



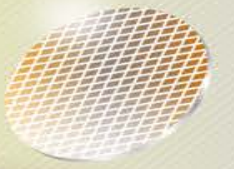
People you need to know!



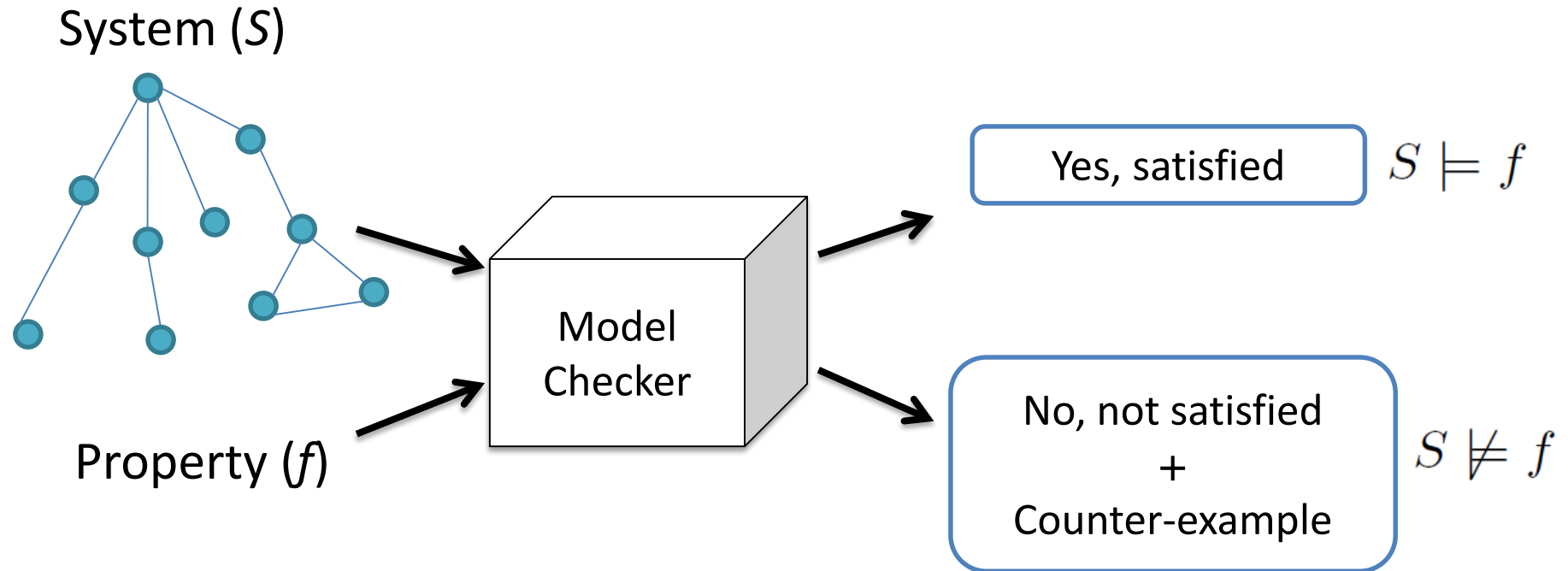
Edmund M. Clarke, E. Allen Emerson and
Joseph Sifakis

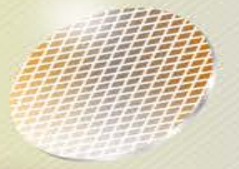


Model is a **formal representation** of
system behaviors.



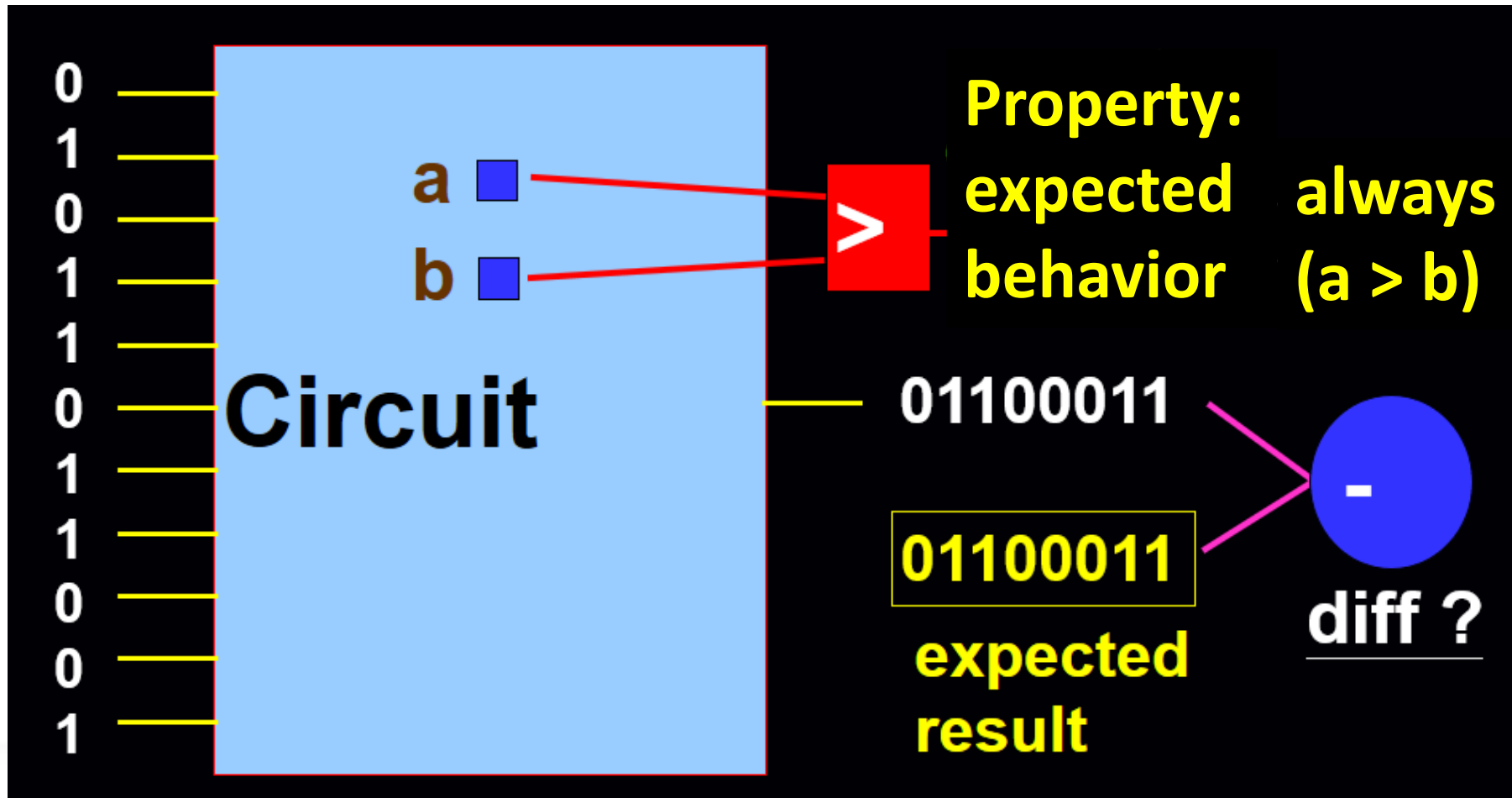
Model checking workflow (so called formal property checking)

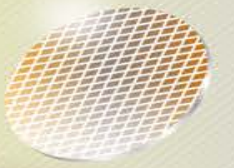




Formal verification in SoC

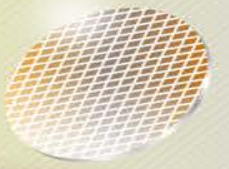
- Model checking: a kind of “proof” techniques



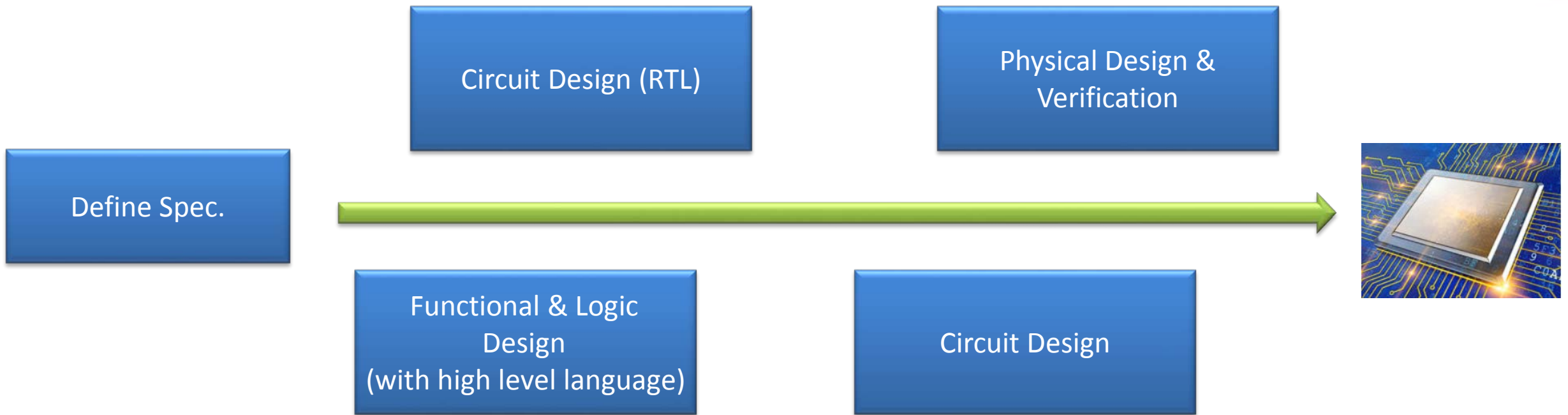


Generic formal verification definition

- Given:
 - Design: implementation from the given spec.
 - Property: expected behavior
- Prove:
 - Property always hold under all circumstance. That is, no input sequence can make property fail.



Formal verification approaches in SoC



Idea → Algorithm → RTL → Gate → Layout → Chip

Property Checking

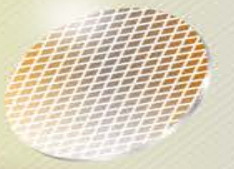
Equivalence Checking

(correct implementation) (correct revision/optimization)



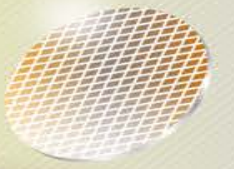
Simulation vs. Model checking (property checking)

- Input constraints
 - Sim: specify the legal inputs
 - Formal: specify the exact set of legal inputs
- Robustness
 - Sim
 - What you sim is what you get.
 - The bug trace can be very long
 - You feel “confident” about the design if you don’t see any bug for enough sim patterns.
 - Formal
 - Can assure mathematical certainty of a property if it is proven
 - Usually generate the shortest bug trace
 - Proof can be aborted due to the complexity issue



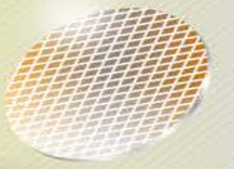
Simulation vs. Formal

- Simulation
 - Easy to use
 - Can run on large circuit
 - Can detect easy bugs quickly
 - Almost impossible to handle corner case bug
- Formal (model checking)
 - Higher learning curve for designers
 - Cannot perform exhaustive search on large designs
 - An target on corner case bug (“corner” w.r.t. simulation wordings)
- Semi-formal --- combines the advantages of both



Common verification issues in SoC

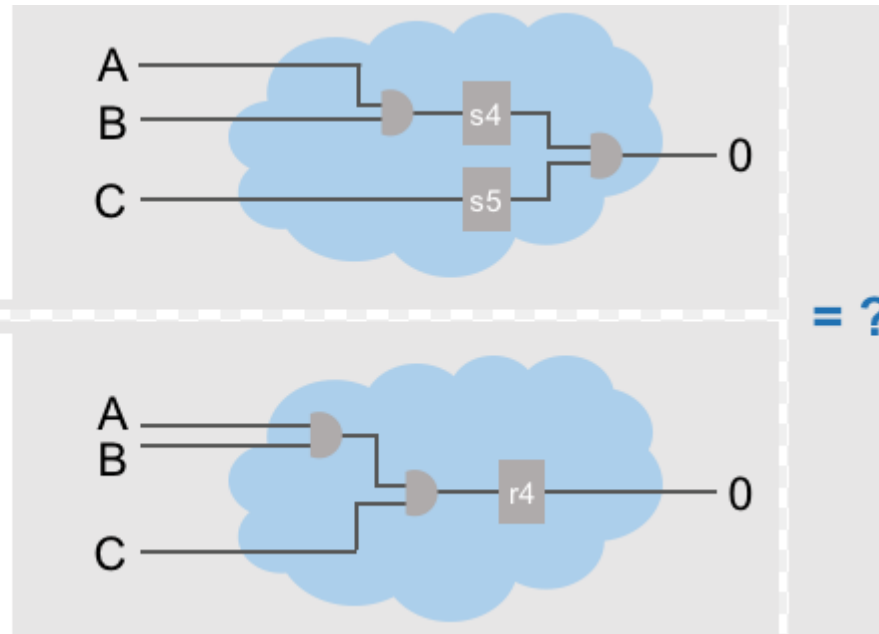
- Determine a simulation hole is
 - An un-covered case due insufficient testbench, or
 - A dead code that never be reachable?
- Data integrity verification for bus protocol bridge
- Detect unknown-value propagation in RTL level
- Check connections in SoC level
- Control status register verification
- Security data path verification
- Sequential equivalent checking
- Root cause detection for post-silicon bugs
- ...



Equivalence checking

```

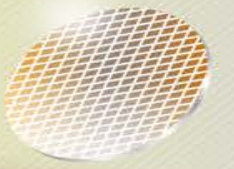
@posedge clk
begin
    S4 <= A && B;
    S5 <= C;
end
assign O = S4 && S5;
    
```



```

@posedge clk
    S4 <= A && B && C;
assign O = S4;
    
```

Example from OneSpin



Conclusions

- Various functional verification options
 - Simulation/emulation/FPGA
 - Formal (property/equivalence checking)
 - Virtual prototyping, theorem proving, ...
- Verification is a methodology, not a pushbutton tool
- Orders of improvement on formal verification engines in the past two decades
 - BDD, SAT, SMT, etc.