Design Verification Technologies Overview with Brief Model Checking Intro.

Yean-Ru Chen

chenyr@mail.ncku.edu.tw

Objectives

- Understandings in
  - What is design verification problem
  - Complexity of verification problem
  - Different verification technologies
  - Basic knowledge of formal model checking methodology

This set of slides is partially from the lecture notes made by Prof. C.-Y. Huang, NTU

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

Huge potential cost to the company

- **Pentium FDIV Bug**
  - https://zh.wikipedia.org/wiki/Pentium_FDIV_bug

- **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。

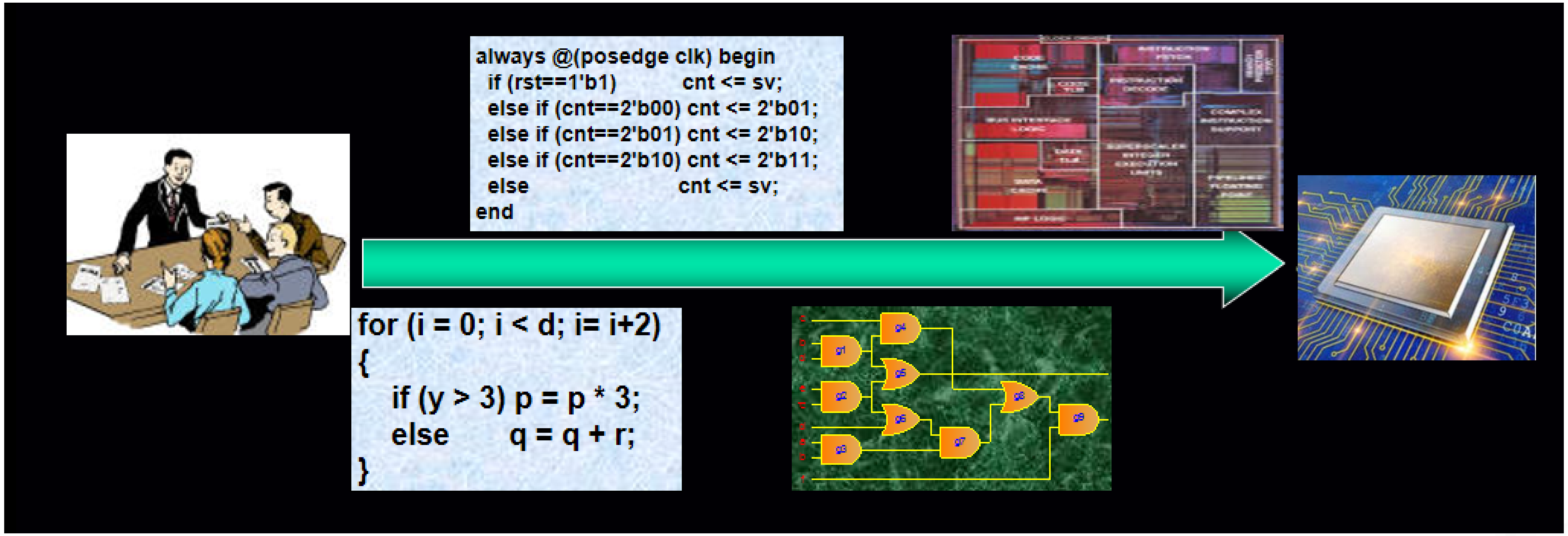
And then…

- 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

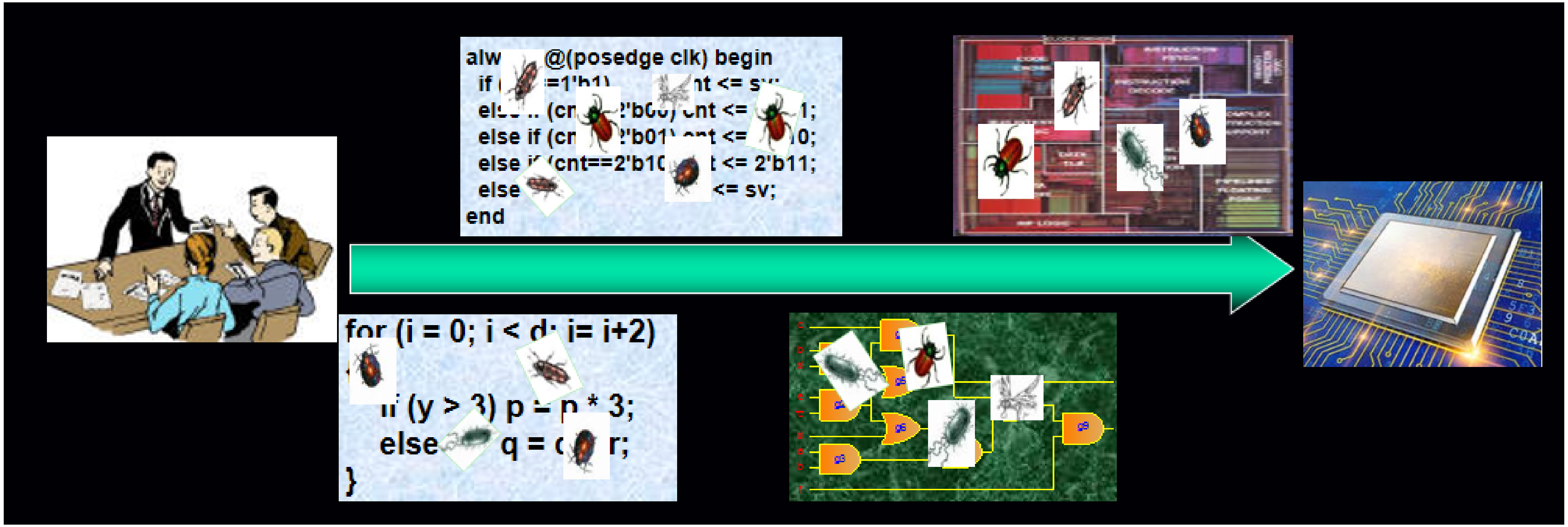
- Let’s look at typical design flow first…



```
always @(posedge clk) begin
  if (rst==1'b1)          cnt <= sv;
  else if (cnt==2'b00) cnt <= 2'b01;
  else if (cnt==2'b01) cnt <= 2'b10;
  else if (cnt==2'b10) cnt <= 2'b11;
  else                        cnt <= sv;
end
```

```
for (i = 0; i < d; i= i+2)
{
    if (y > 3) p = p * 3;
    else        q = q + r;
}
```

Bugs can be anywhere…



```
alw[illegible] @(posedge clk) begin
  if ([illegible]=1'b1) [illegible]nt <= sv;
  el[illegible] (cn[illegible]'b00) cnt <= [illegible]1;
  else if (cn[illegible]'b01) cnt <= [illegible]0;
  else i[illegible](cnt==2'b10[illegible]t <= 2'b11;
  else [illegible] <= sv;
end
```

```
for (i = 0; i < d; i= i+2)
[illegible]
  if (y > 3) p = p * 3;
  else [illegible] q = [illegible]r;
}
```

- 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: $2^{100} = 10^{30} = 10^{24}$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

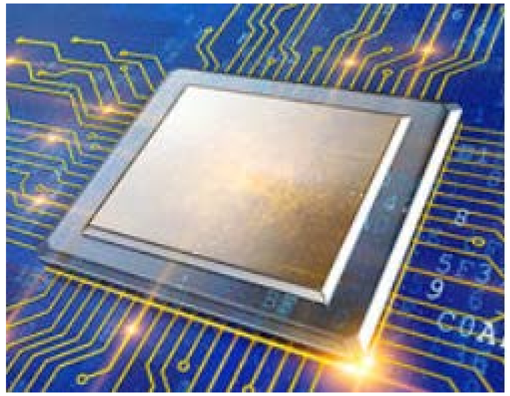
所以 formal 不可能是用 exhaustive simulation做成的!

What is Verification?

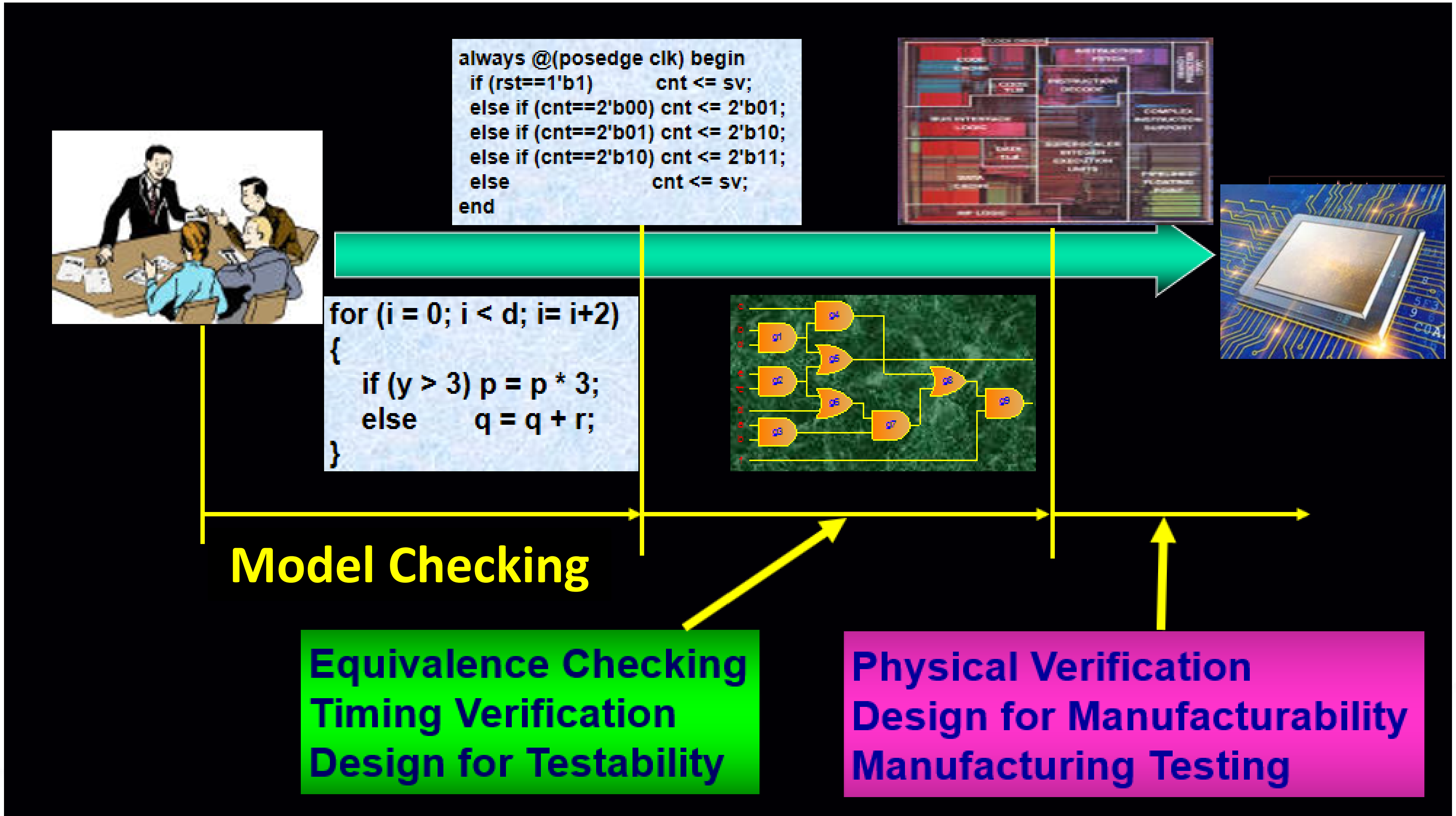
- Generally speaking, verification also covers
  - Functional verification
  - Security verification
  - Timing verification
  - Physical verification
  - Manufacturing defect testing, etc
- Strictly speaking, verification is usually referred only to “*functional verification*”.

What is functional verification?

- Functionally correctness
- Spec fully implemented



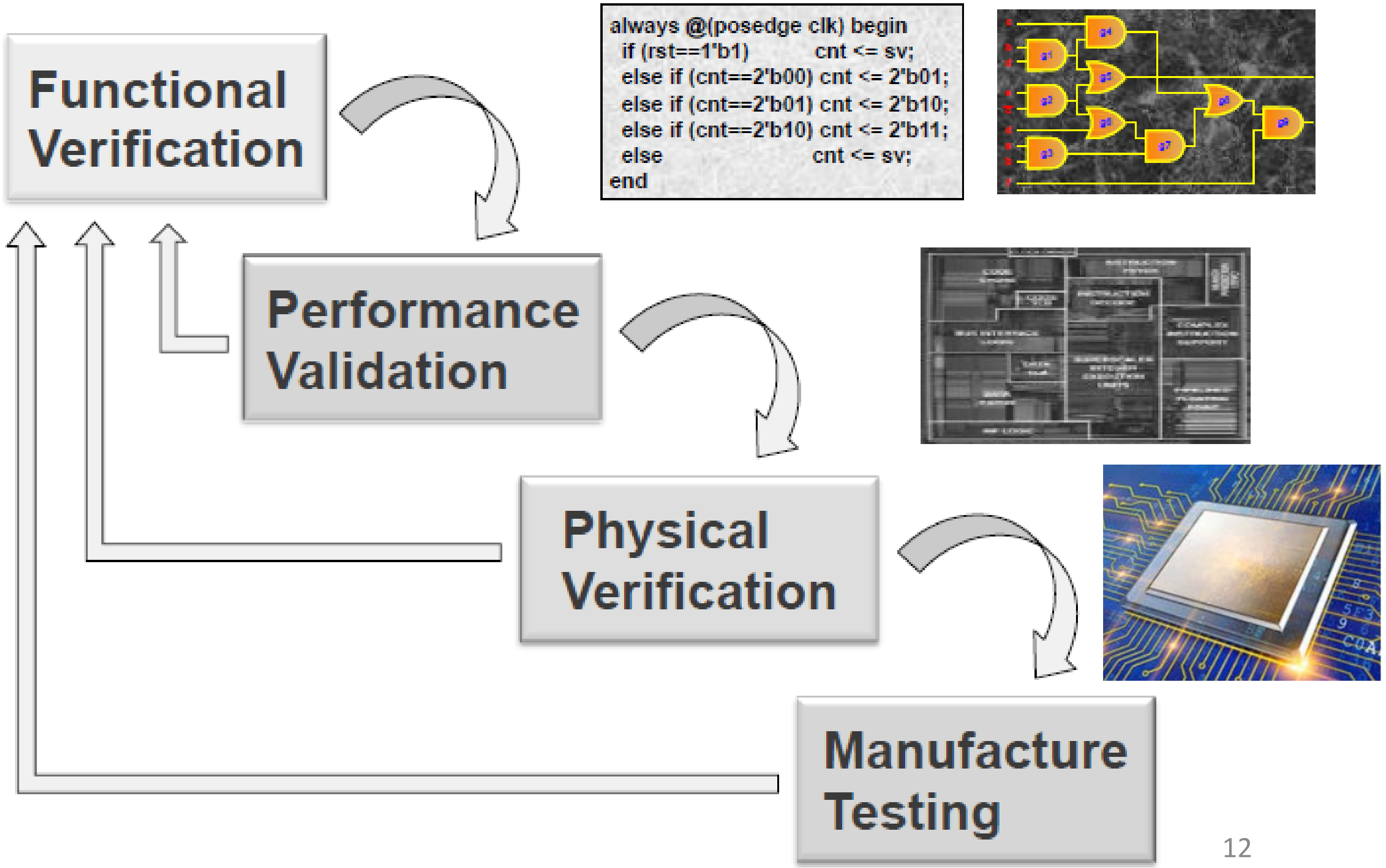
Typical verification paradigm


```
always @(posedge clk) begin
  if (rst==1'b1)          cnt <= sv;
  else if (cnt==2'b00) cnt <= 2'b01;
  else if (cnt==2'b01) cnt <= 2'b10;
  else if (cnt==2'b10) cnt <= 2'b11;
  else                        cnt <= sv;
end
```

```
for (i = 0; i < d; i= i+2)
{
   if (y > 3) p = p * 3;
   else       q = q + r;
}
```

Four verification tasks in SoC verification flow

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



```
always @(posedge clk) begin
  if (rst==1'b1)          cnt <= sv;
  else if (cnt==2'b00) cnt <= 2'b01;
  else if (cnt==2'b01) cnt <= 2'b10;
  else if (cnt==2'b10) cnt <= 2'b11;
  else                    cnt <= sv;
end
```

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

Functional Verification Overview

Agenda

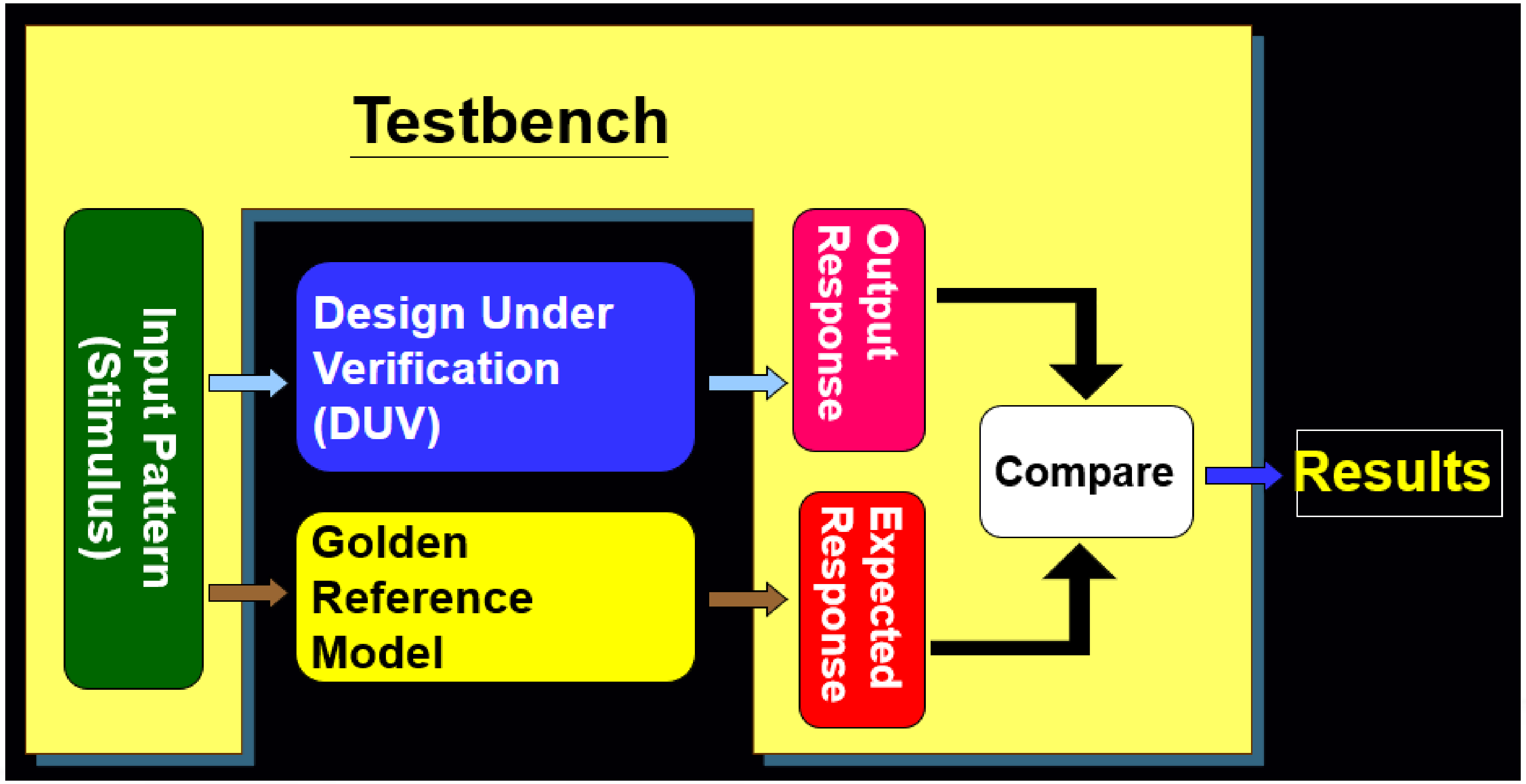
- Simulation-based techniques
- Assertion-based verification
- Emulation
- FPGA prototyping
- Virtual prototyping
- Model checking (so-called Property checking)
- Formal equivalence checking
- Theorem proving
- Semi-formal verification

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

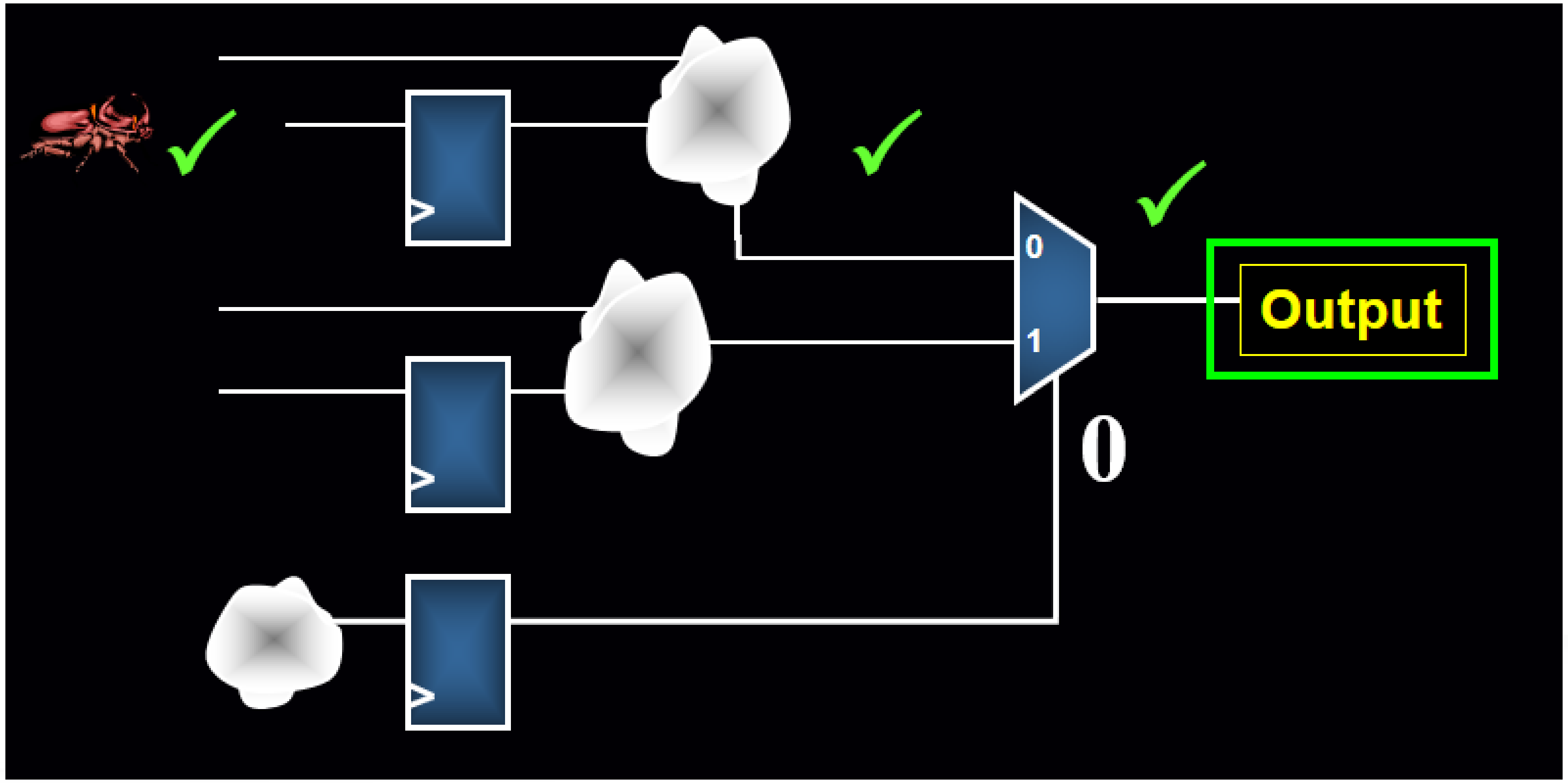
Simulation-based techniques

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



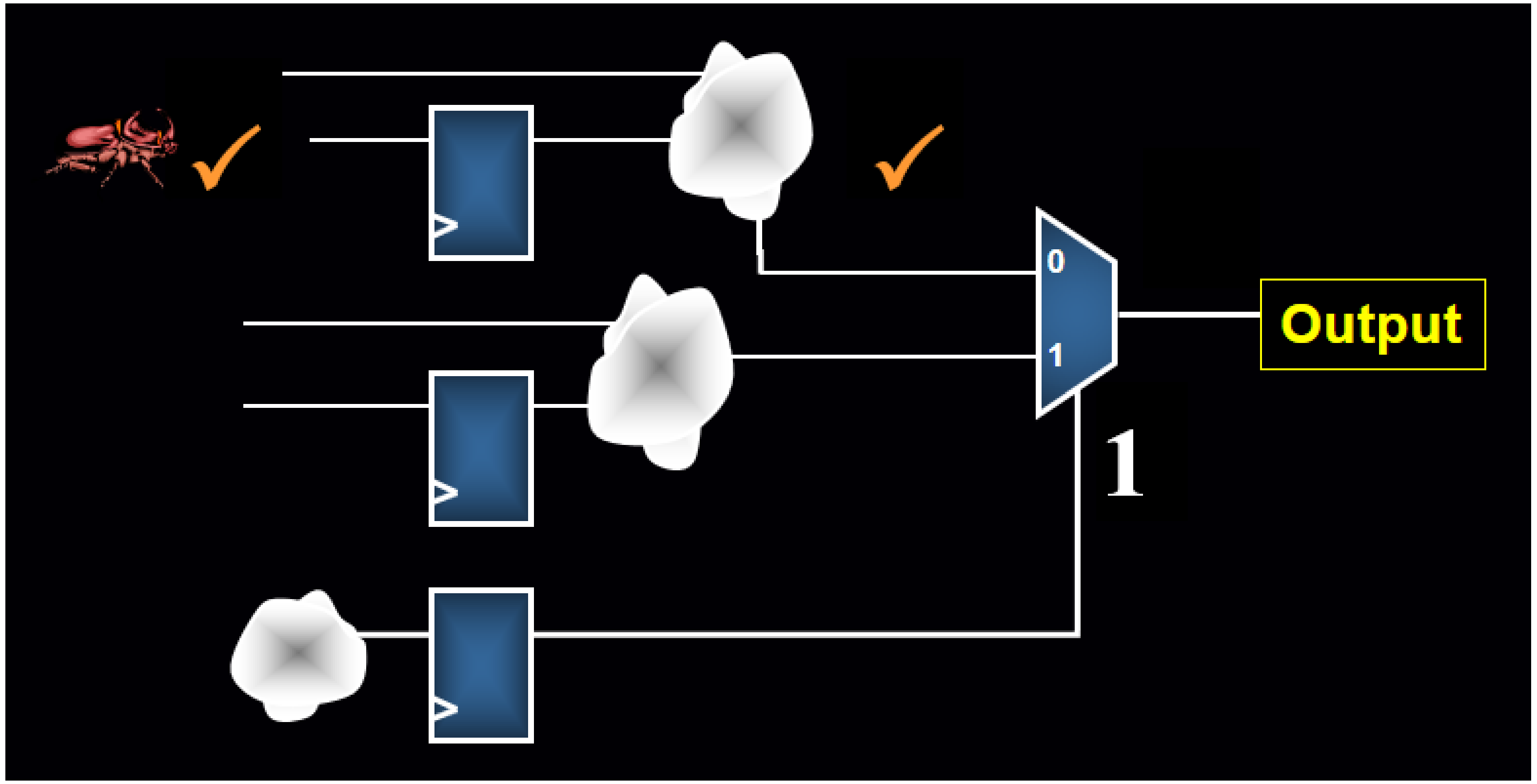
Observability problem

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



Observability problem

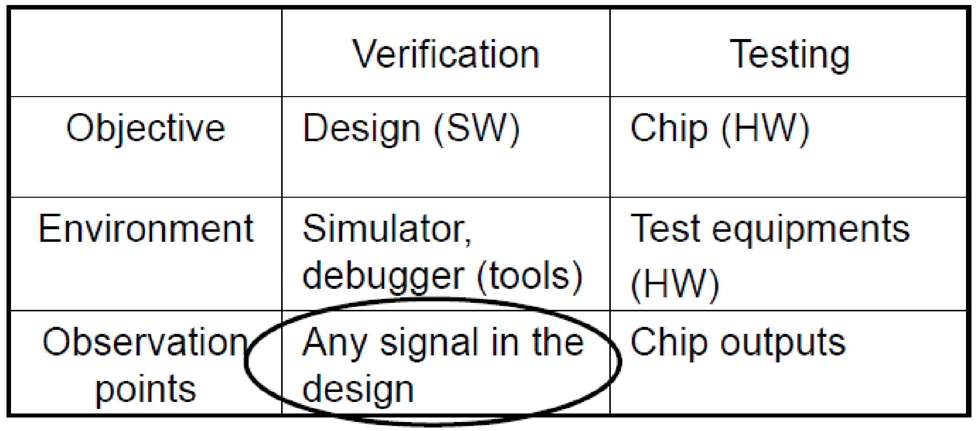
- For example, bug can not be detected at output:



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

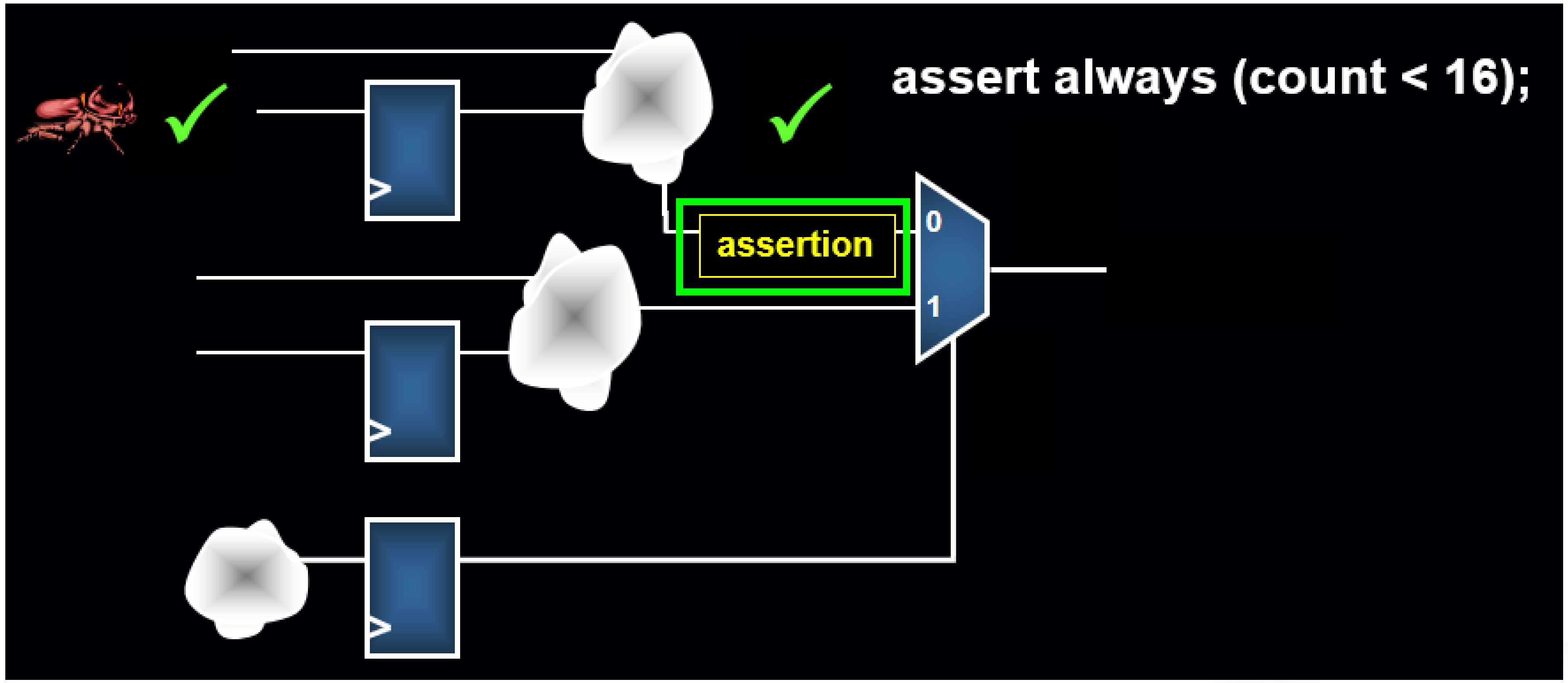


	Verification	Testing
Objective	Design (SW)	Chip (HW)
Environment	Simulator, debugger (tools)	Test equipments (HW)
Observation points	Any signal in the design	Chip outputs

- 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

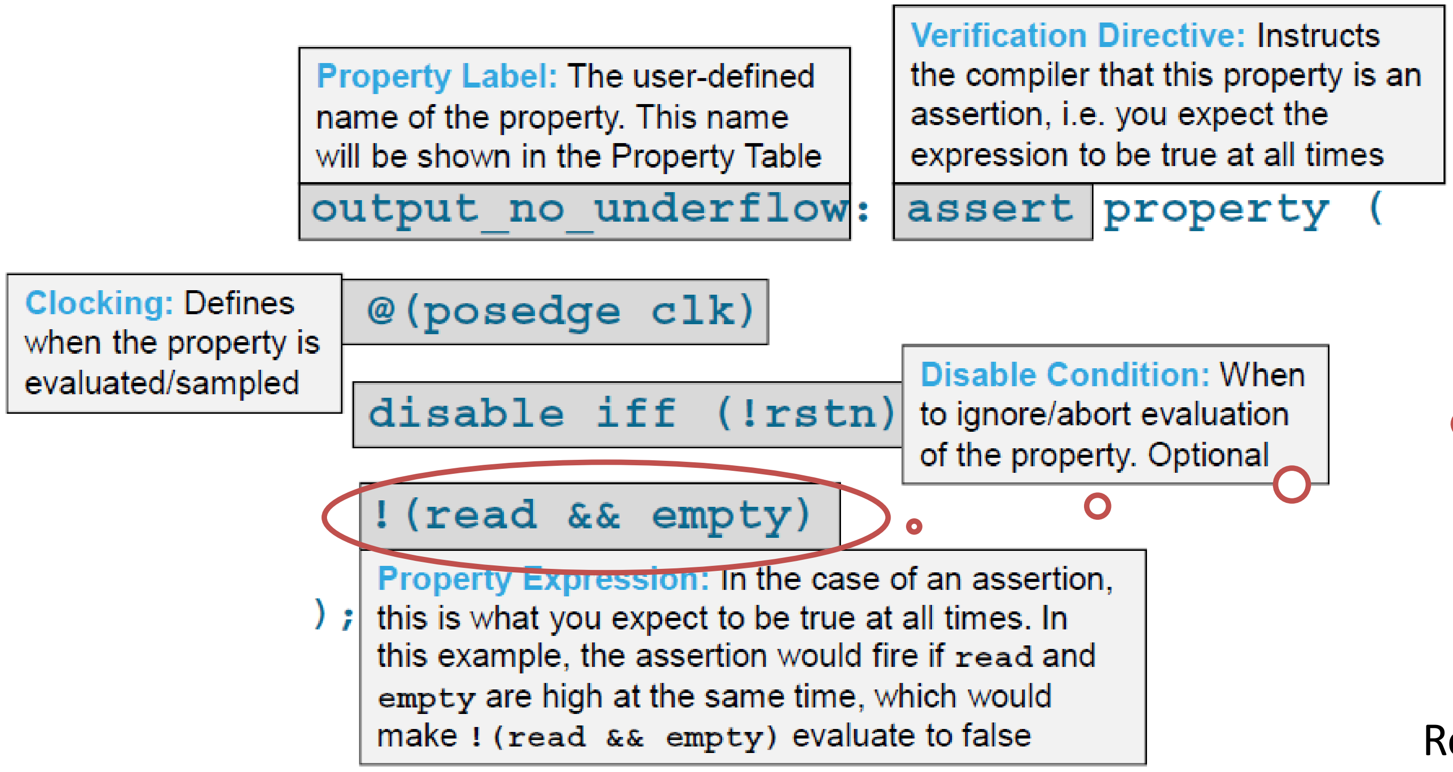
- Insert “*assertions*” in the middle of the circuit, and flag the simulator whenever the violation occurs
  – Increase the observability of the bugs.

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 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

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

Clocking: Defines when the property is evaluated/sampled

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

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

```
output_no_underflow: assert property (
    @(posedge clk)
    disable iff (!rstn)
    !(read && empty)
);
```

邏輯要寫正確寫得剛剛好

Ref: Cadence JasperGold SVA Introduction

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!

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.
  - No “corner case” to formal method-based verification.
- Different approaches of formal verification

Inductive proof (數學歸納證明法)

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

Step1: Show true for n = 1:

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

Notice that it is also true for n = 2:

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

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

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

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

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

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

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

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

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

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

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

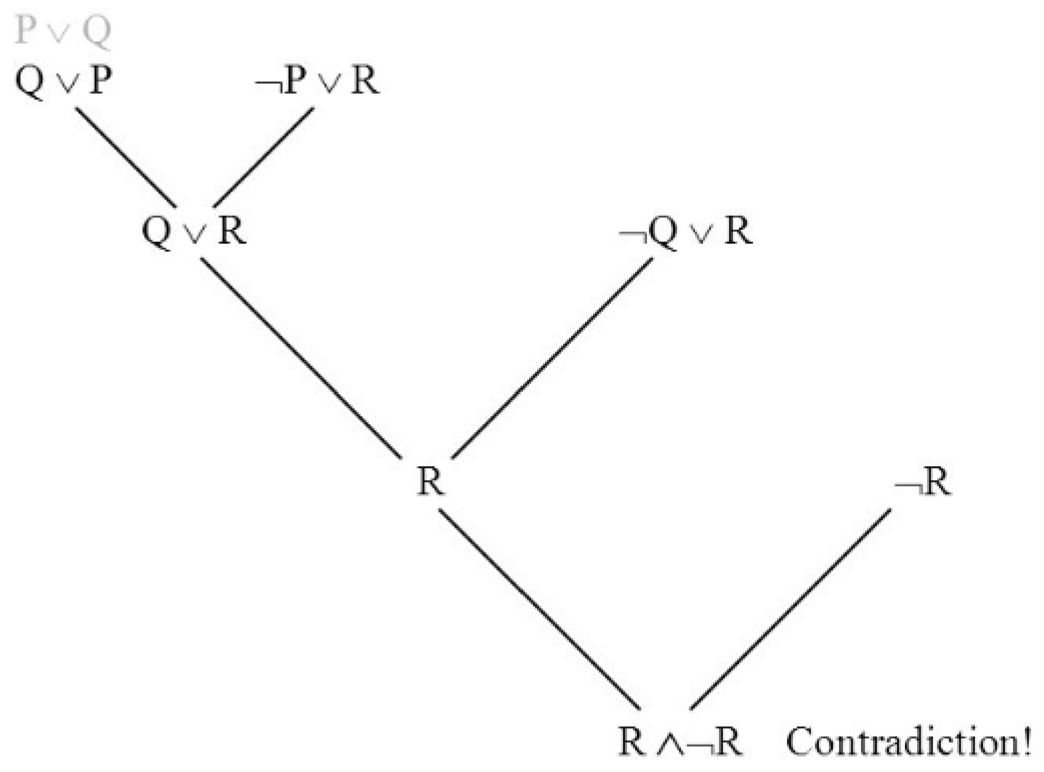
Deductive verification (演繹驗證)

- Theorem proving for CNF resolution

Example 1:

$$\begin{tikzcd}
\text{(A or B)} \arrow[dr] & & \text{(!A or C)} \arrow[dl] \\
& \text{(B or C)} &
\end{tikzcd}$$

Example2:

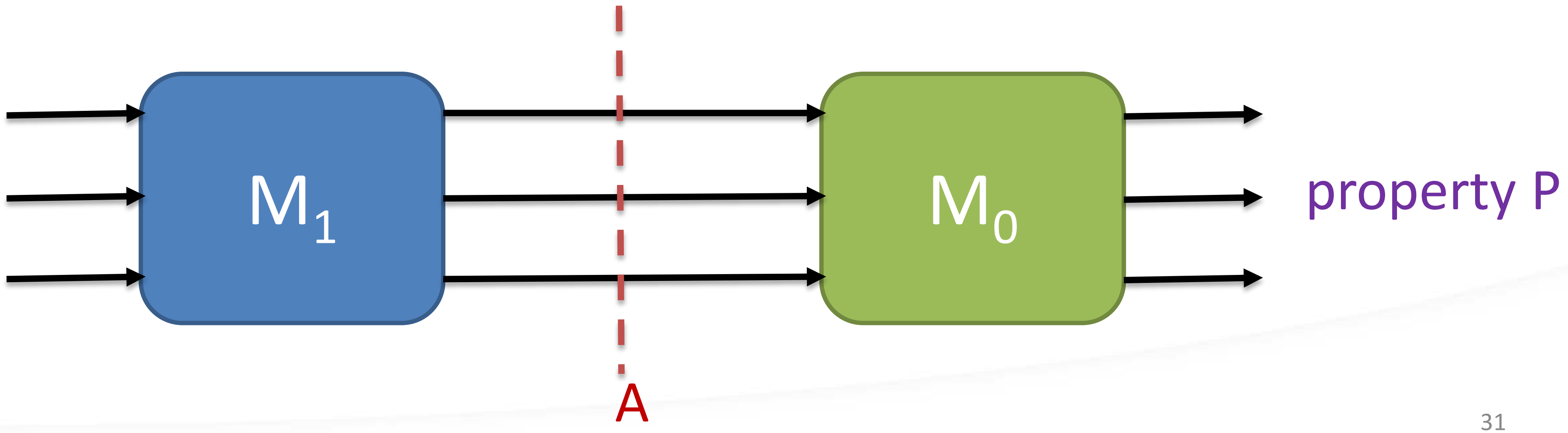


Deductive verification (演繹驗證)

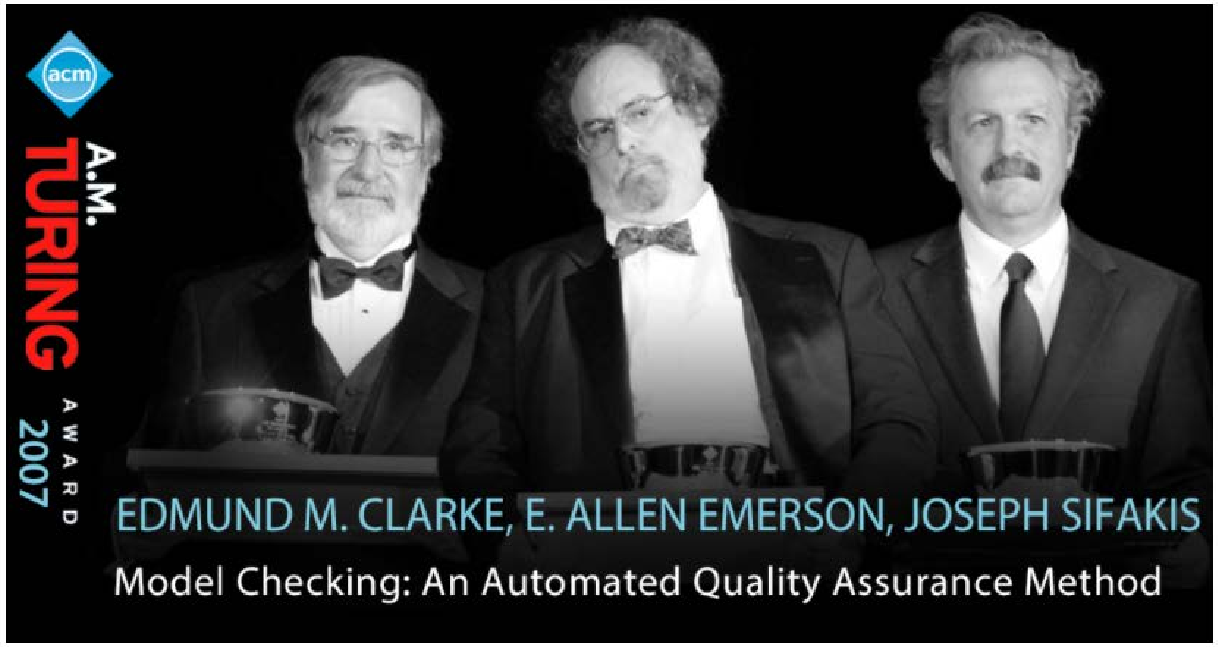
- Assume-guarantee reasoning

$$\frac{M_0 \| A \models P \qquad M_1 \models A}{M_0 \| M_1 \models P}$$

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



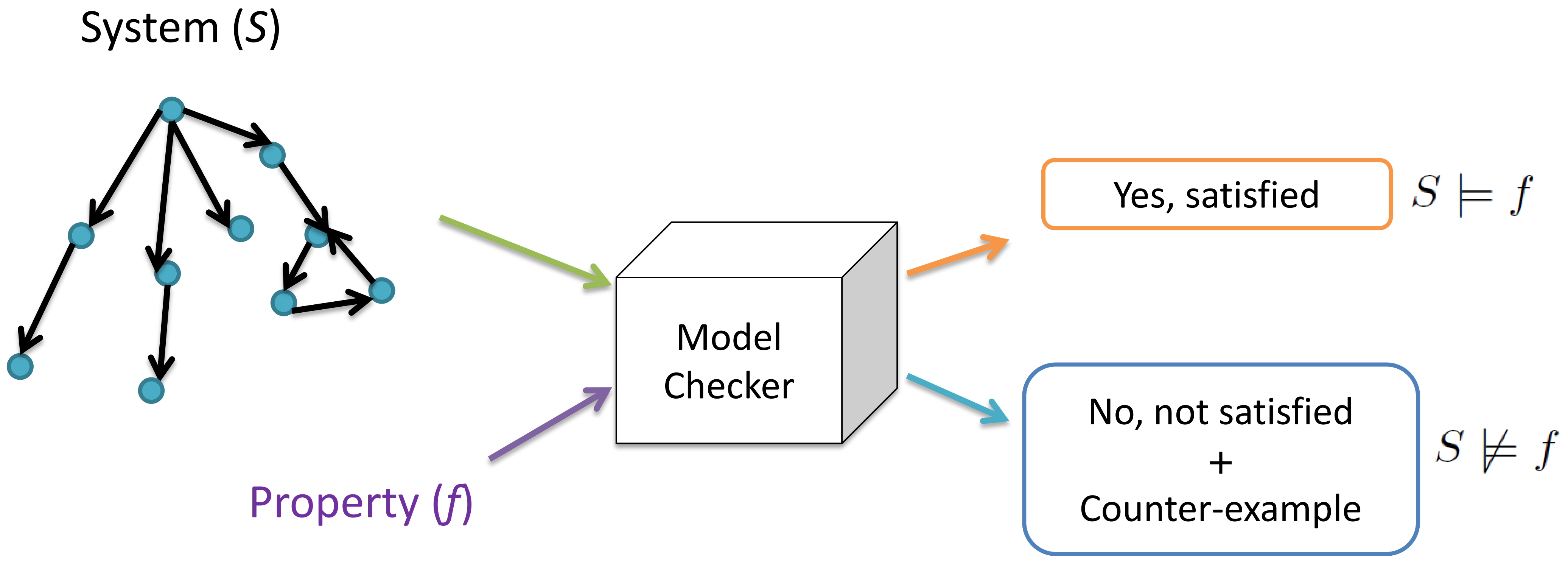
People you need to know!


acm
A.M. TURING AWARD 2007
EDMUND M. CLARKE, E. ALLEN EMERSON, JOSEPH SIFAKIS
Model Checking: An Automated Quality Assurance Method

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

Model is a formal representation of system behaviors.

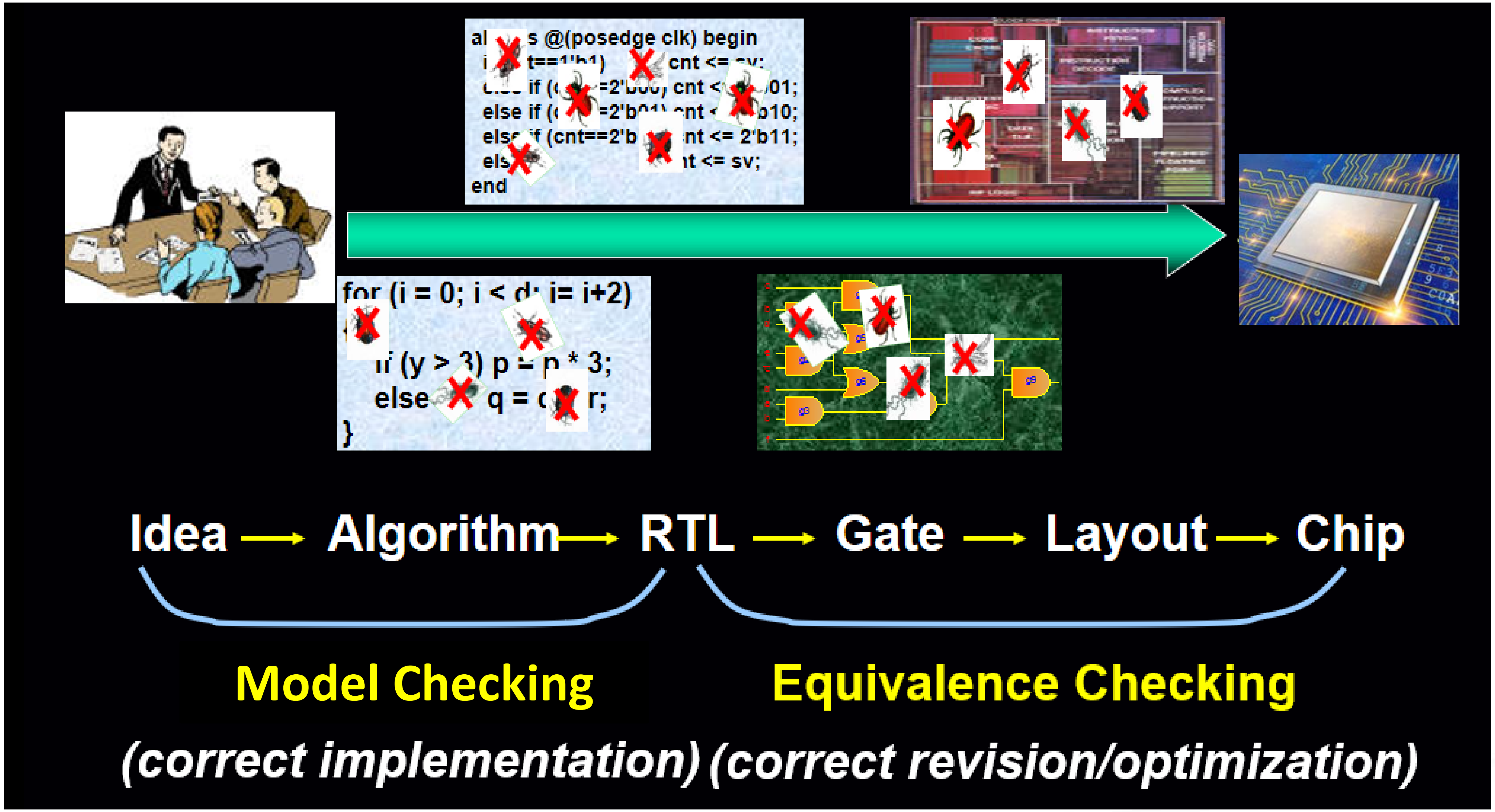
Model checking workflow (so called formal property checking)



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
Model Checking
(correct implementation)
Equivalence Checking
(correct revision/optimization)

Ref. from Prof. C.-Y. (Ric) Huang

Model checking (So-called property checking)

- Objective: Make sure the (RTL) implementation is correct – w.r.t. the given spec..
- Typical flow
  - 1. Specify constraints (but I suggest to add constraints later…one by one… non-primary-input constraints should be verified first!)
  - 2. Compose assertions/monitors
  - 3. Run model checking
  - 4. Debugging, if any assertion is violated
  - 5. Write more constraints/assertions/monitors if necessary

Formal verification == 100% verification?

- Note: formal verification can “prove” a property always true
  - For all input combinations until infinite time
- If we can prove ALL properties true (or find counter-examples for failed properties)
  - Is it 100% verification?
- Do you write enough properties?
  - (who’s responsibility?)

So how can model checking verify all states specified in DUV?

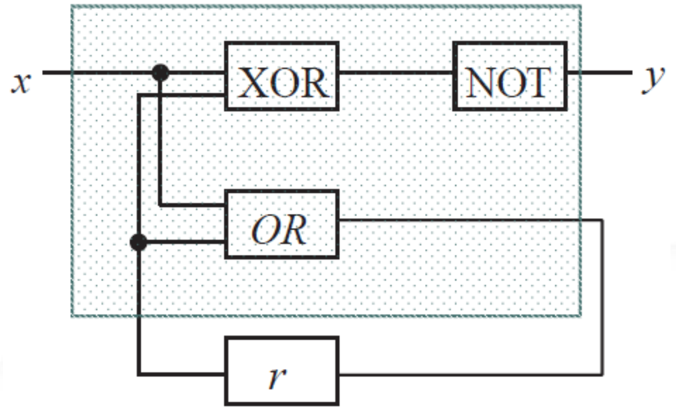
Sequential Hardware Circuits

- Control function for *y*: $\lambda_y = \neg(x \oplus r)$
- Register evaluation changes according to the circuit function:

$\delta_r = x \vee r$

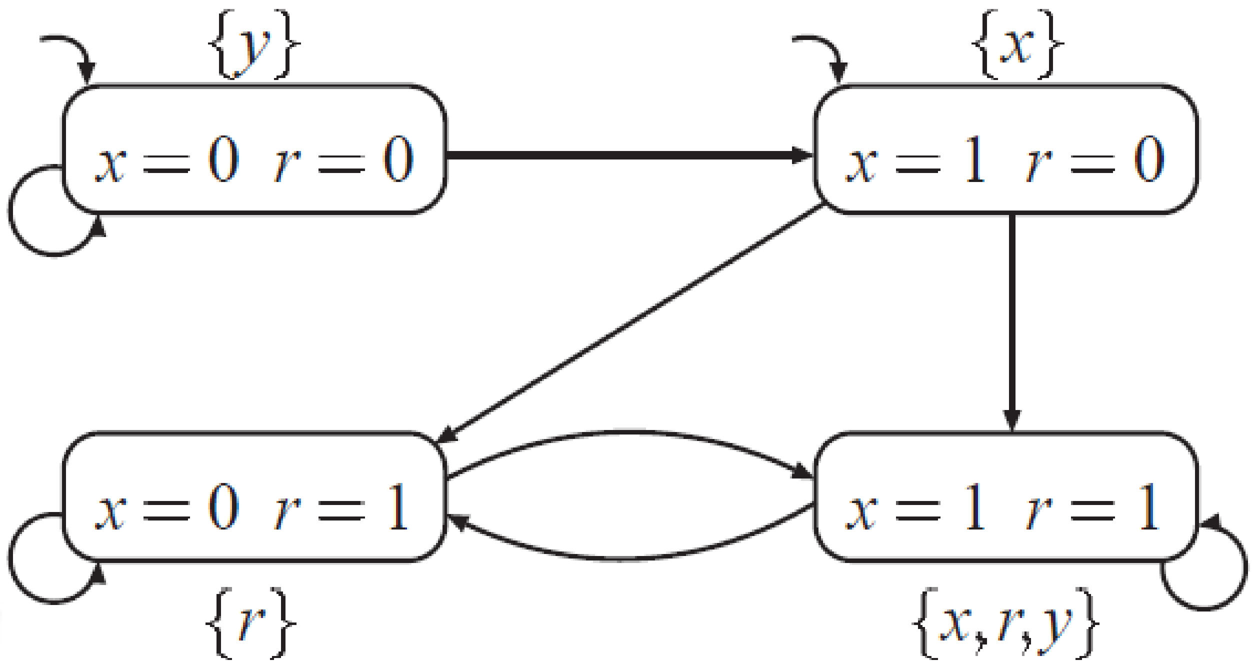
  - Once the register evaluation is [*r* = 1], *r* keeps the value.
  - Initial value of *r* is [*r* = 0]
- State space *S*: $S = Eval(x, r)$
  - *x* is free, and thus the initial states of TS are *I*

$$I = \{ \langle x = 0, r = 0 \rangle, \langle x = 1, r = 0 \rangle \}$$

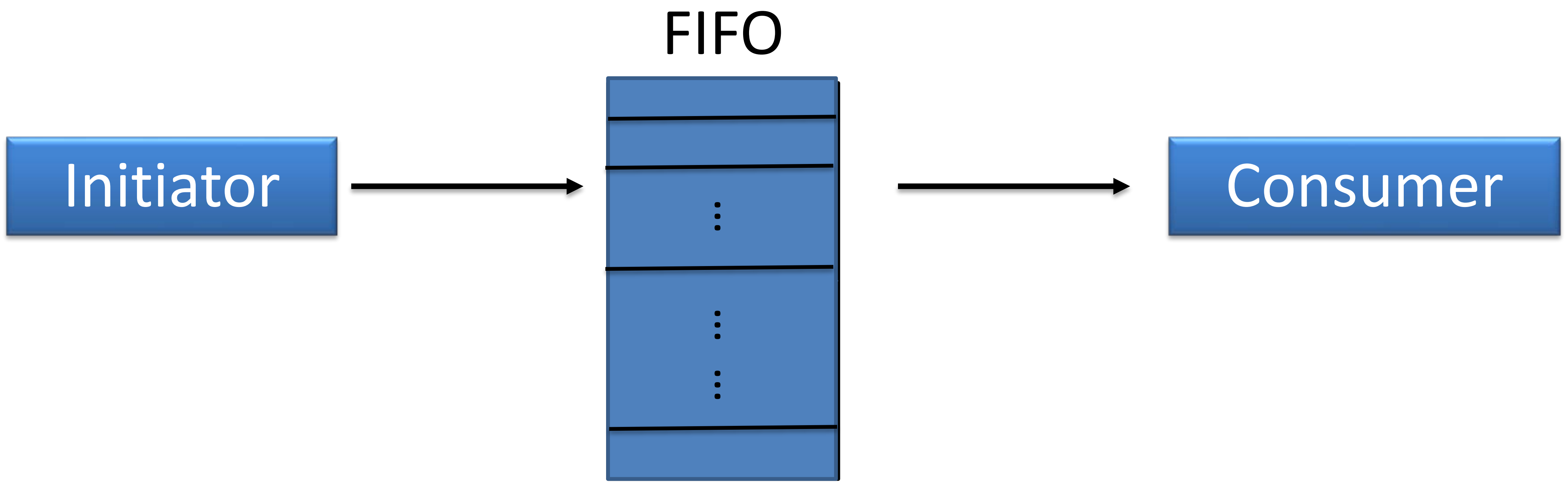


Circuit to Transition State Graph

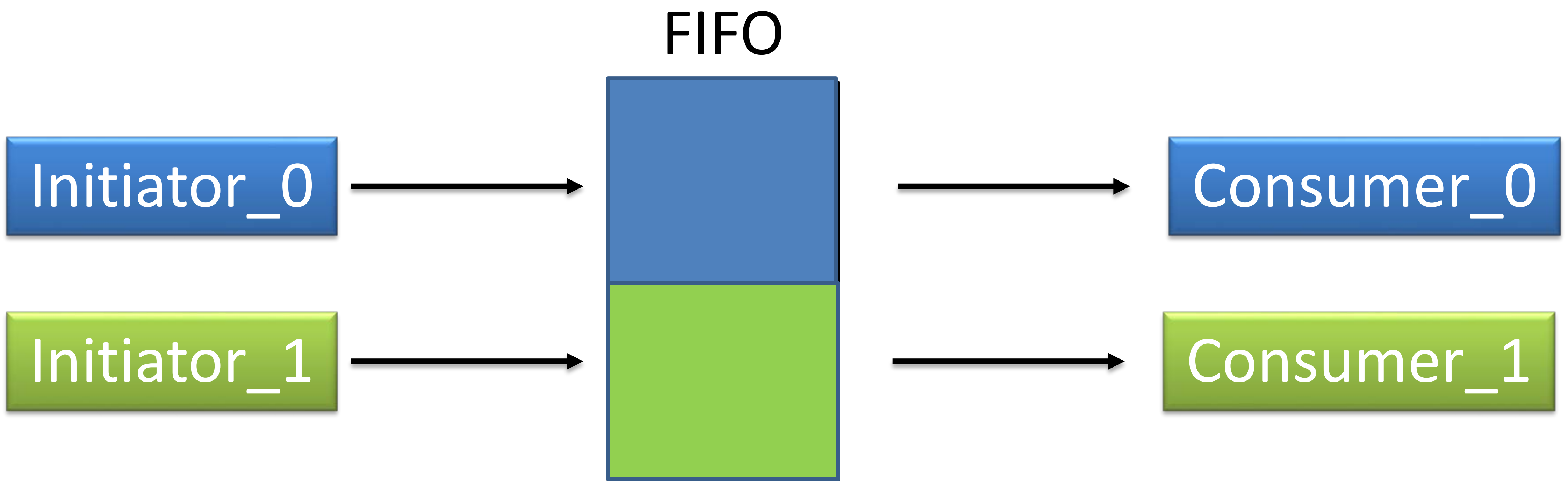
- Consider the labeling *L*
  - AP = { *x*, *y*, *r* }
    - E.g. the state $\langle x = 0, r = 1 \rangle$ is labeled with {r}. No *y* since $\neg(x \oplus r)$ results in value 0 for this state.
- The TSG of this circuit



Data Integrity vs. Data Size



Abstraction Trap!


FIFO
Initiator_0
Initiator_1
Consumer_0
Consumer_1