



LSI LOGIC DESIGN

CHAPTER 3

LSI Logic Design

JUNE 02, 2020

PHAM TUONG HAI

QUALITY ASSESSMENT & TRAINING DEPARTMENT

RENESAS DESIGN VIETNAM CO., LTD.

RENESAS ELECTRONICS CORPORATION

CHAPTER 3. LSI Logic Design

3.1. LSI Design Flow.

3.2. System Design.

3.3. Logic Design.

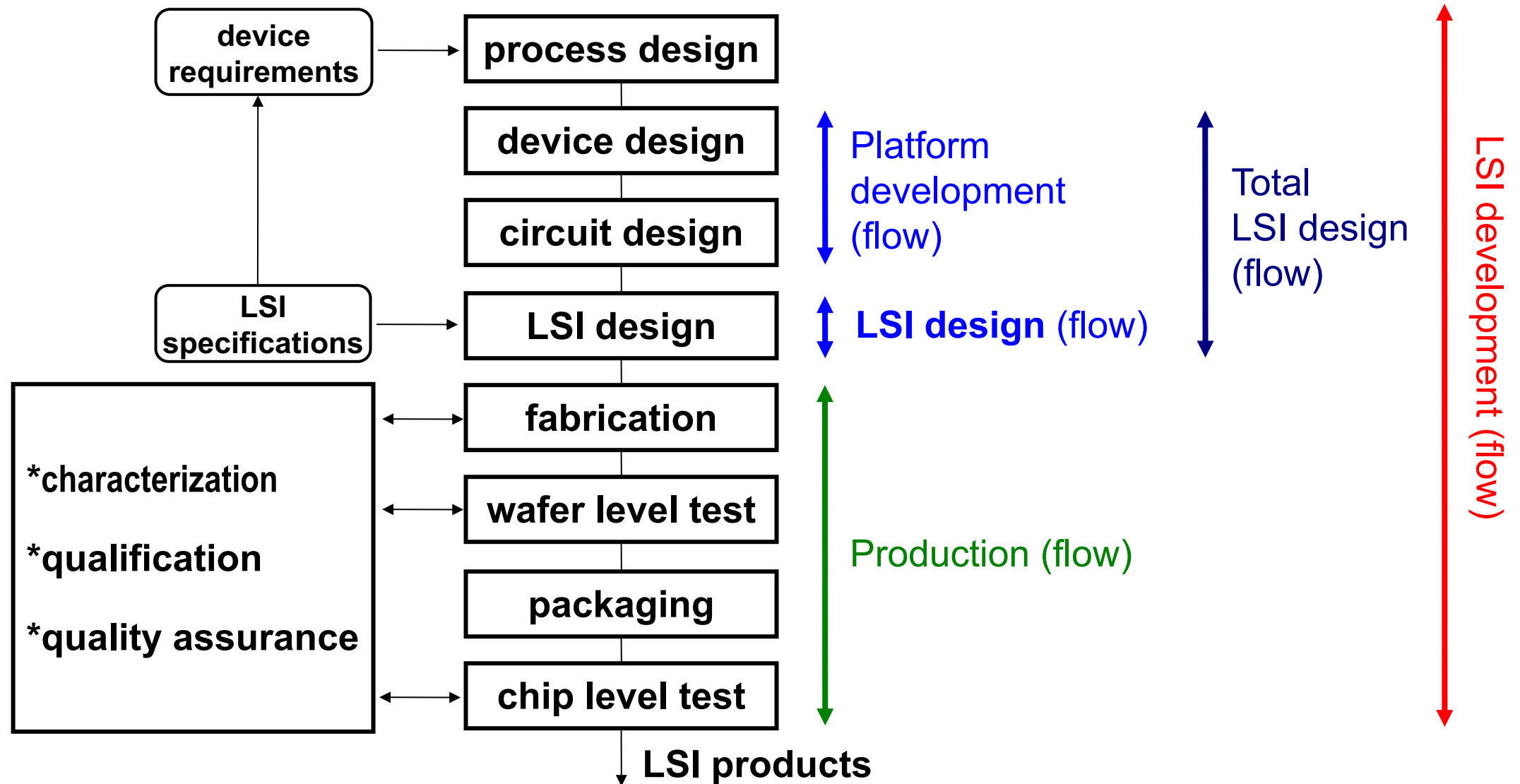
3.4. RTL Design and Verification.

3.5. Logic Synthesis and Cell-based Design.

3.6. Gate Level Design and Verification.

3.1 LSI Design Flow

LSI Development Flow (Review)

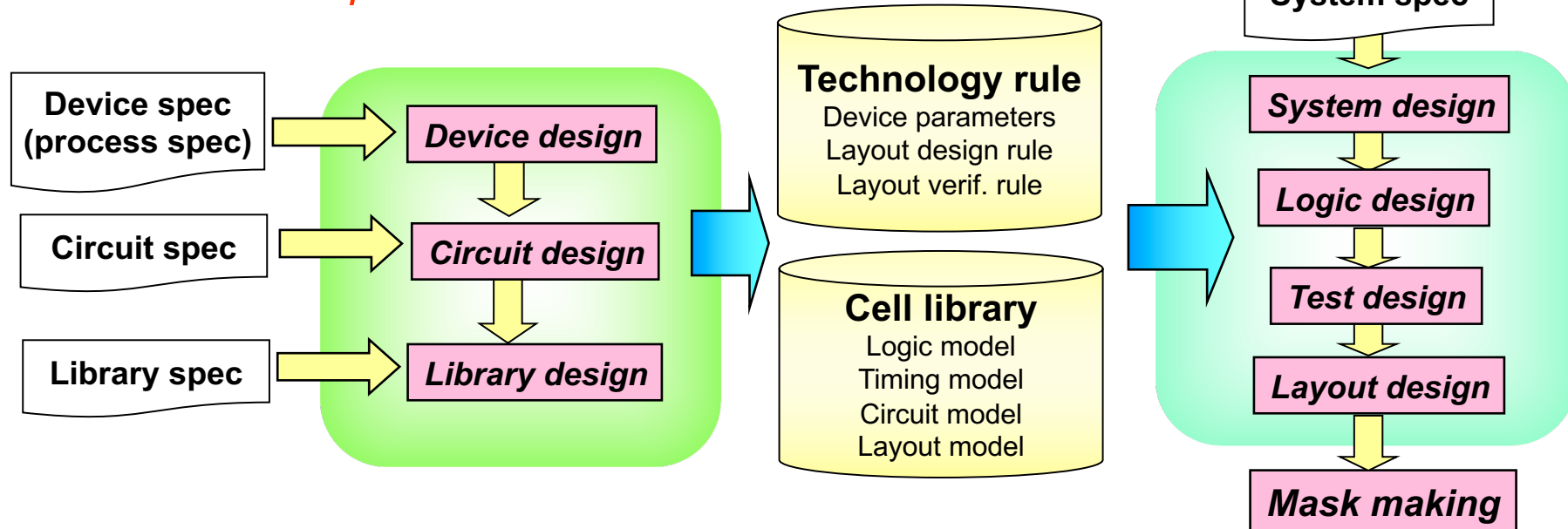


Total LSI Design Flow (Review)

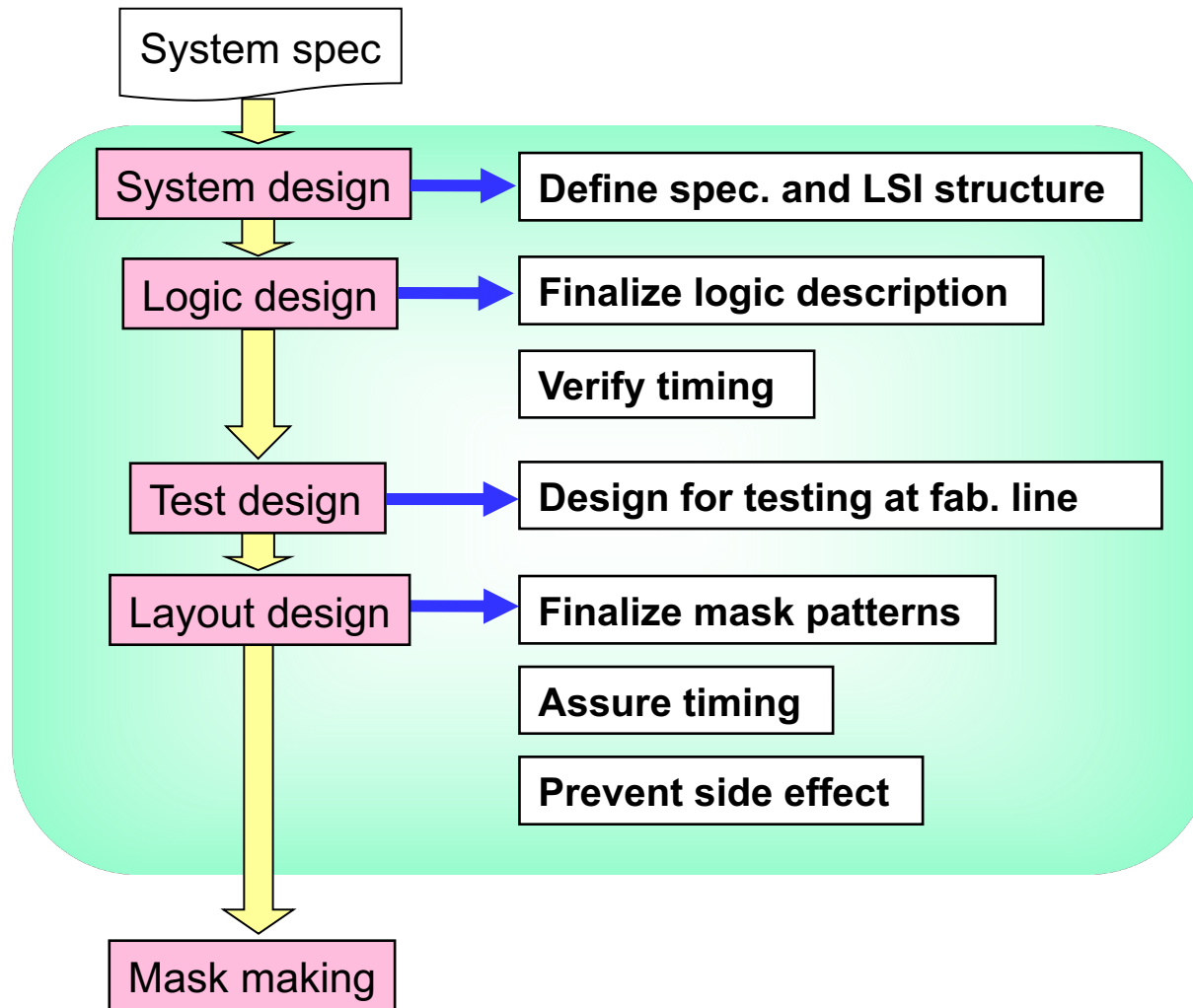
To design LSIs using design environment established by platform development

☞ Quality of the platform determines quality of all products

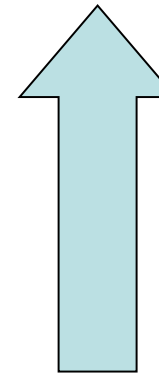
Platform Development Flow



LSI Design Flow (1/2)

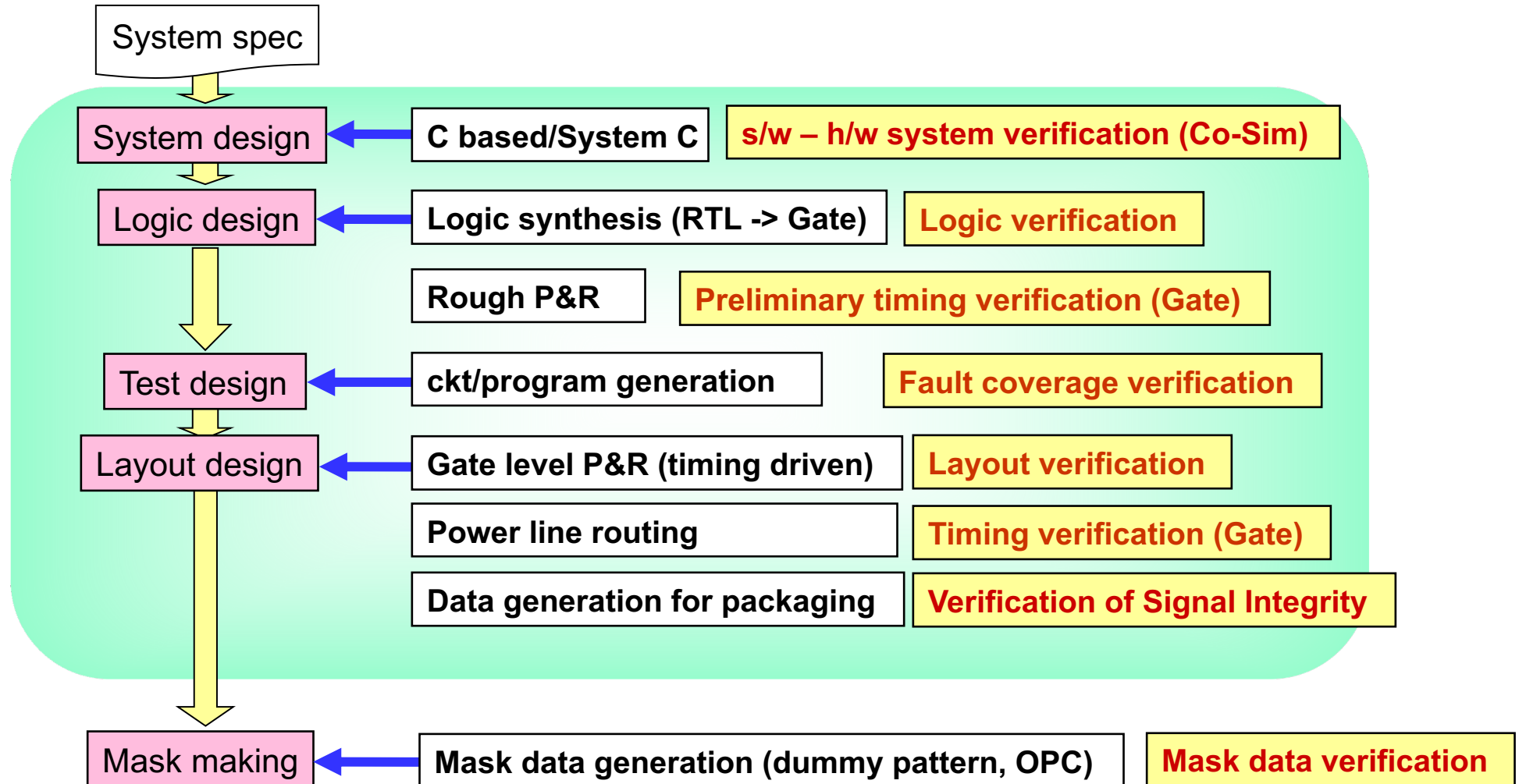


- Design input & output
- Design review
- Documentation



**Minimize
Design Errors**

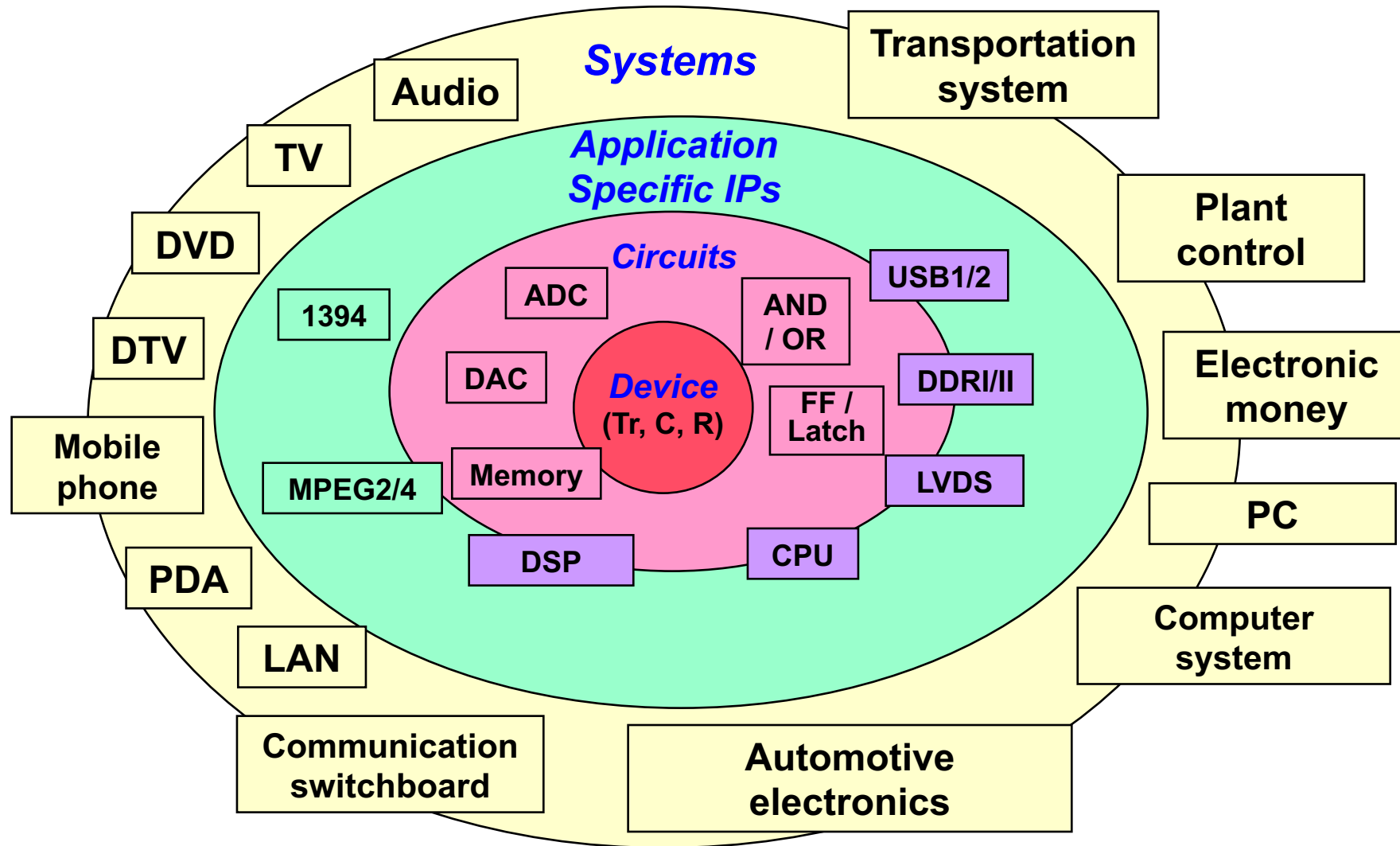
LSI Design Flow (2/2)



3.2 System Design

System Design

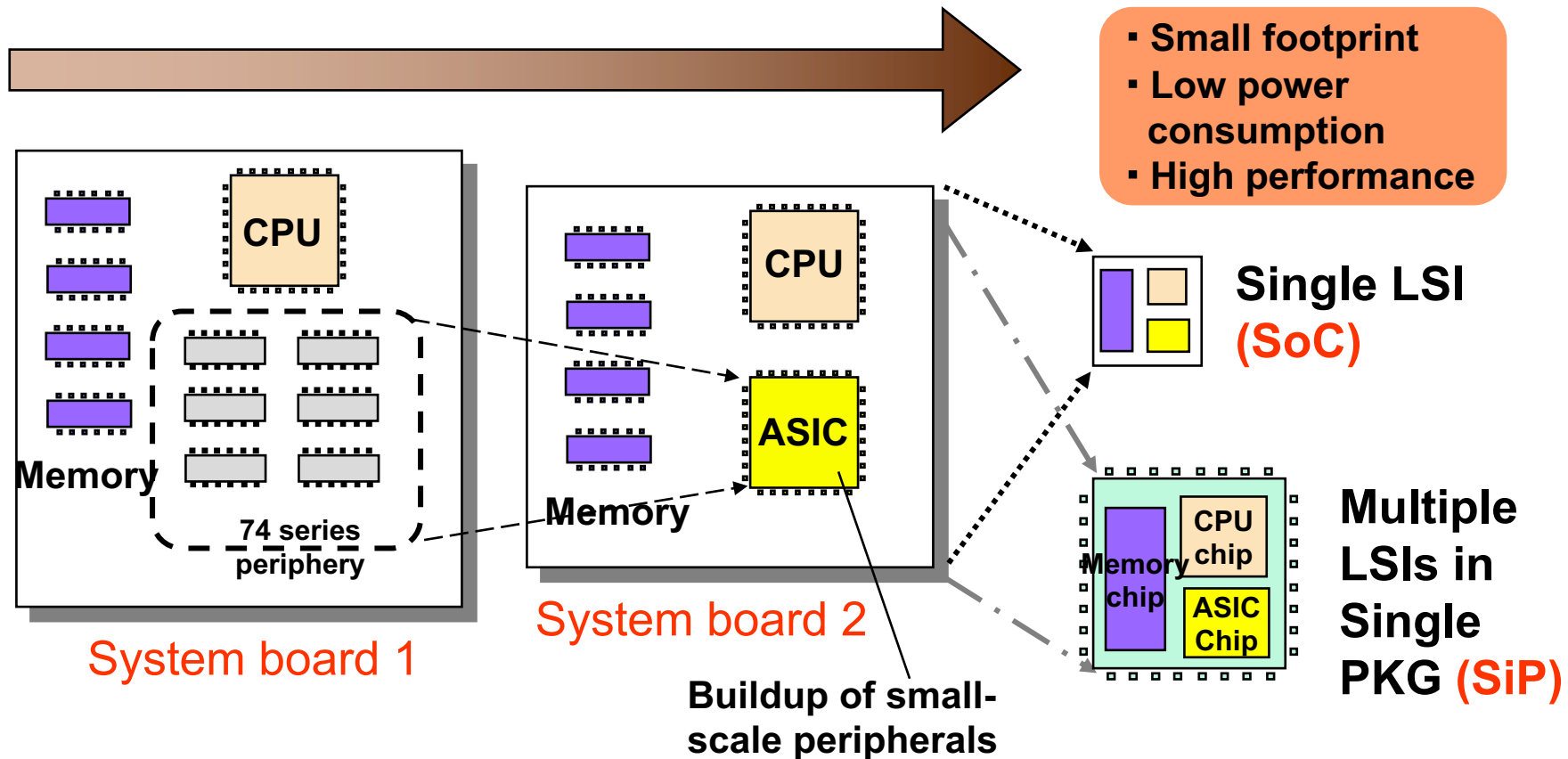
Systems & Components



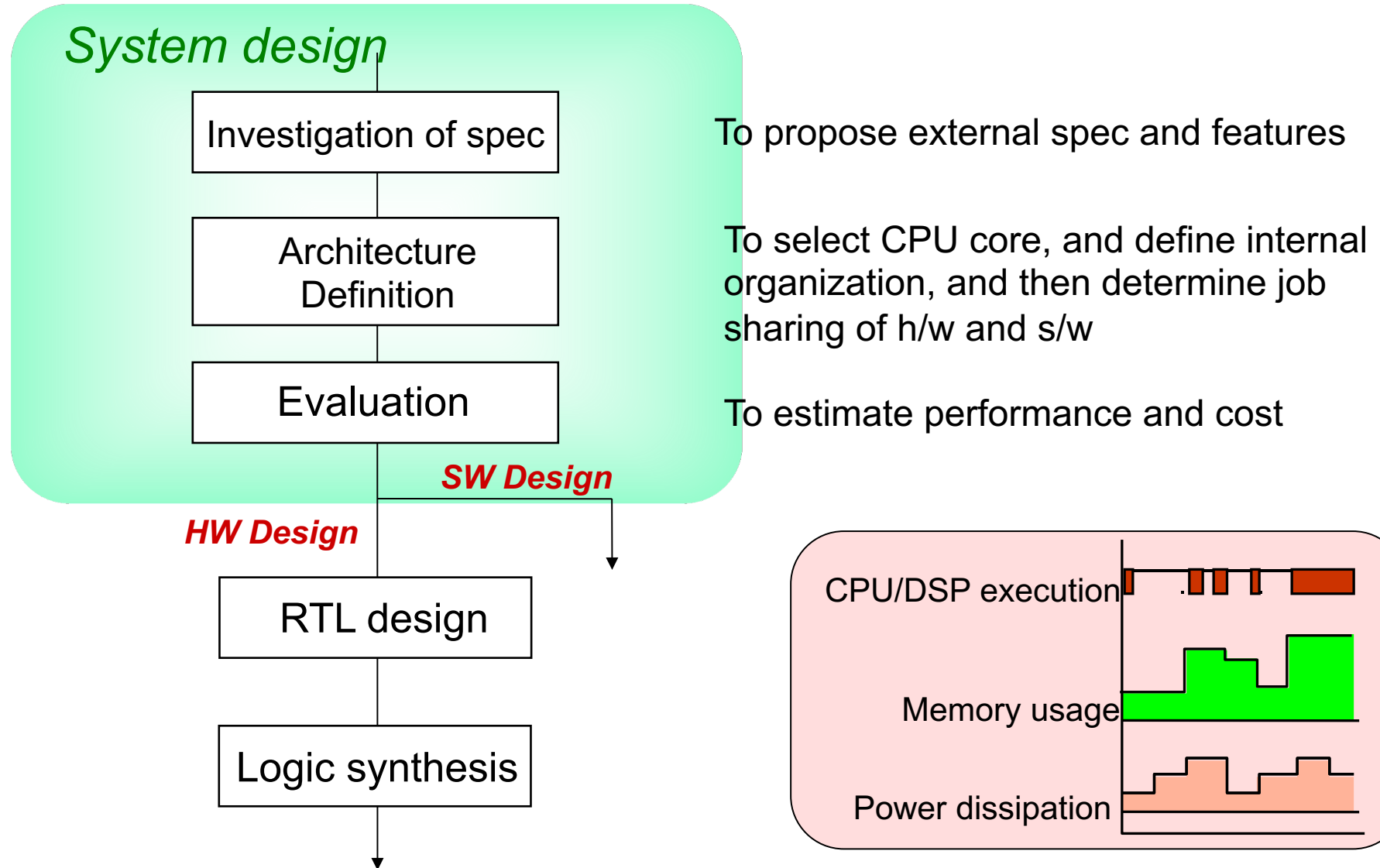
System Integration

SoC (System-on-Chip): System implementation in single LSI

SiP(System-in-Package): System implementation in 1 package



System Design Flow



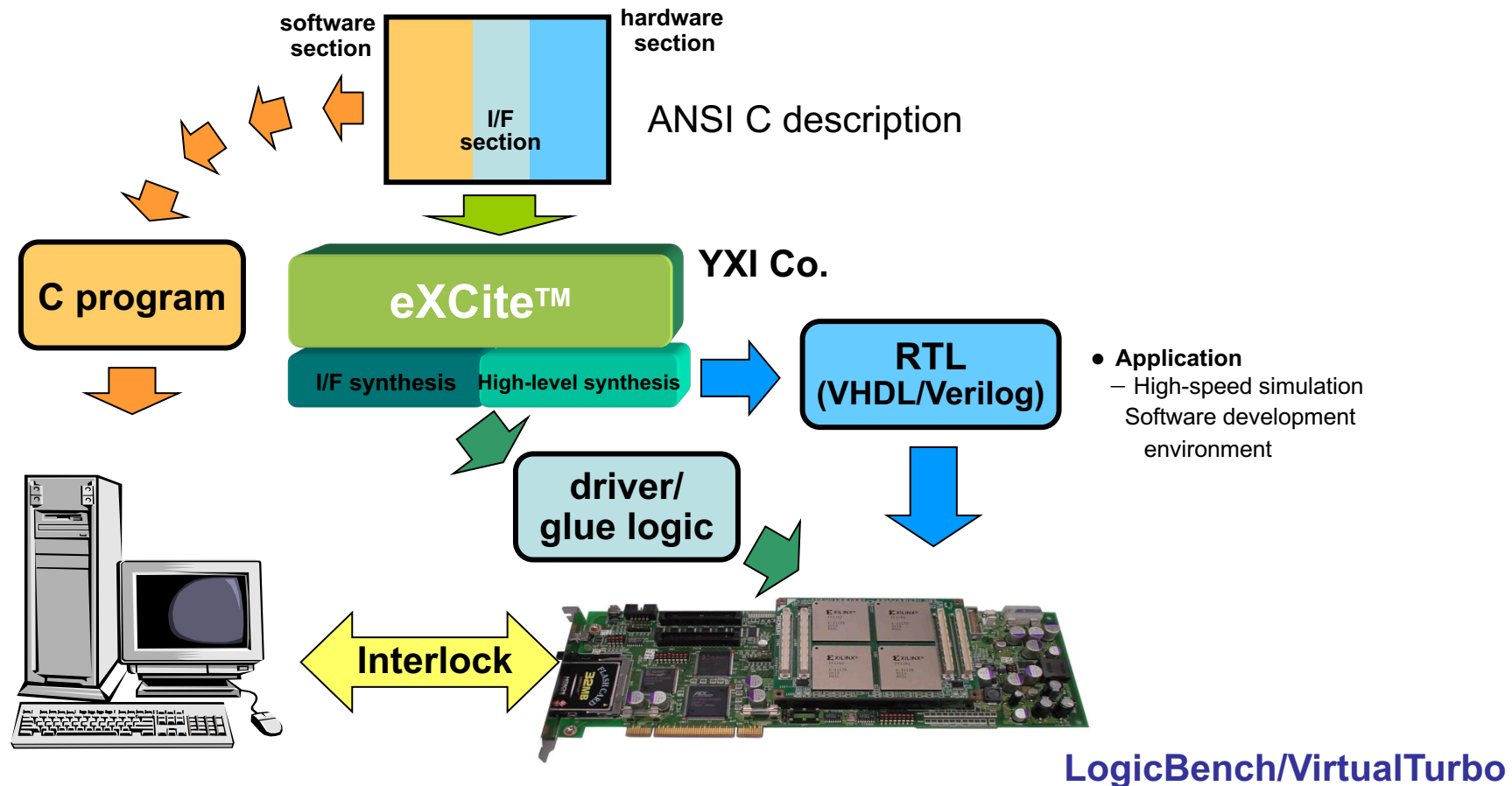
Requirements for System Design

- To optimize LSI specifications by watching **market trend** and hearing **customers' needs**, such as:
 - performance, features, power dissipation, reliability,
 - price (target die size, package selection, process).
- To remove over-spec and spec bugs.
- To avoid specification changes during HW design phase because HW design becomes critical:
 - Demand for time-to-market is increasing.
 - Complexity is increasing.
 - Need for performance is increasing.
 - Fabrication cost is increasing (more than \$1M for one mask set).

Early Prototype

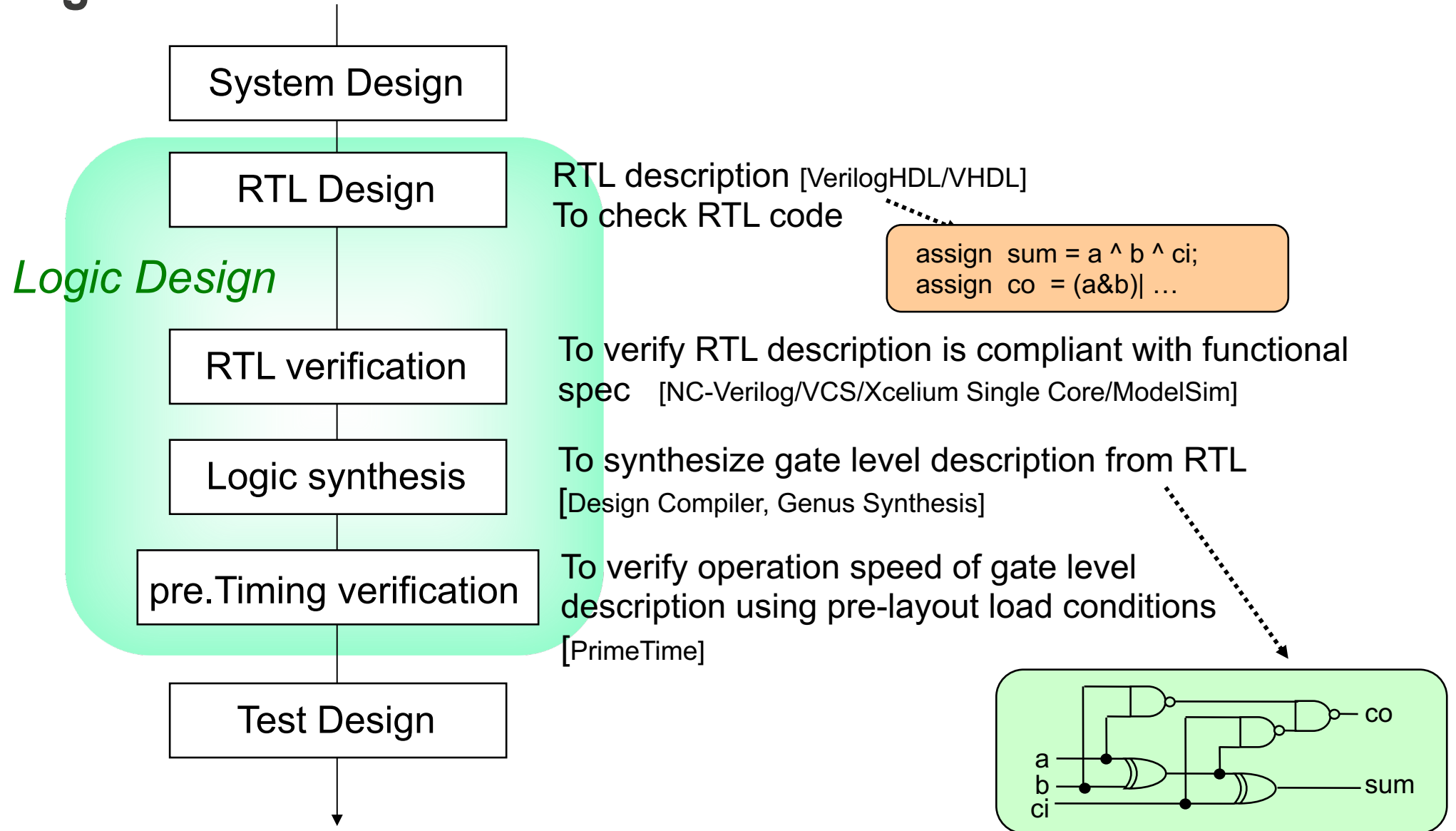
Functional description (algorithm) in ANSI C is converted to form FPGA-based early prototype

- Platform for software development, and system verification prior to initiation of RTL design.

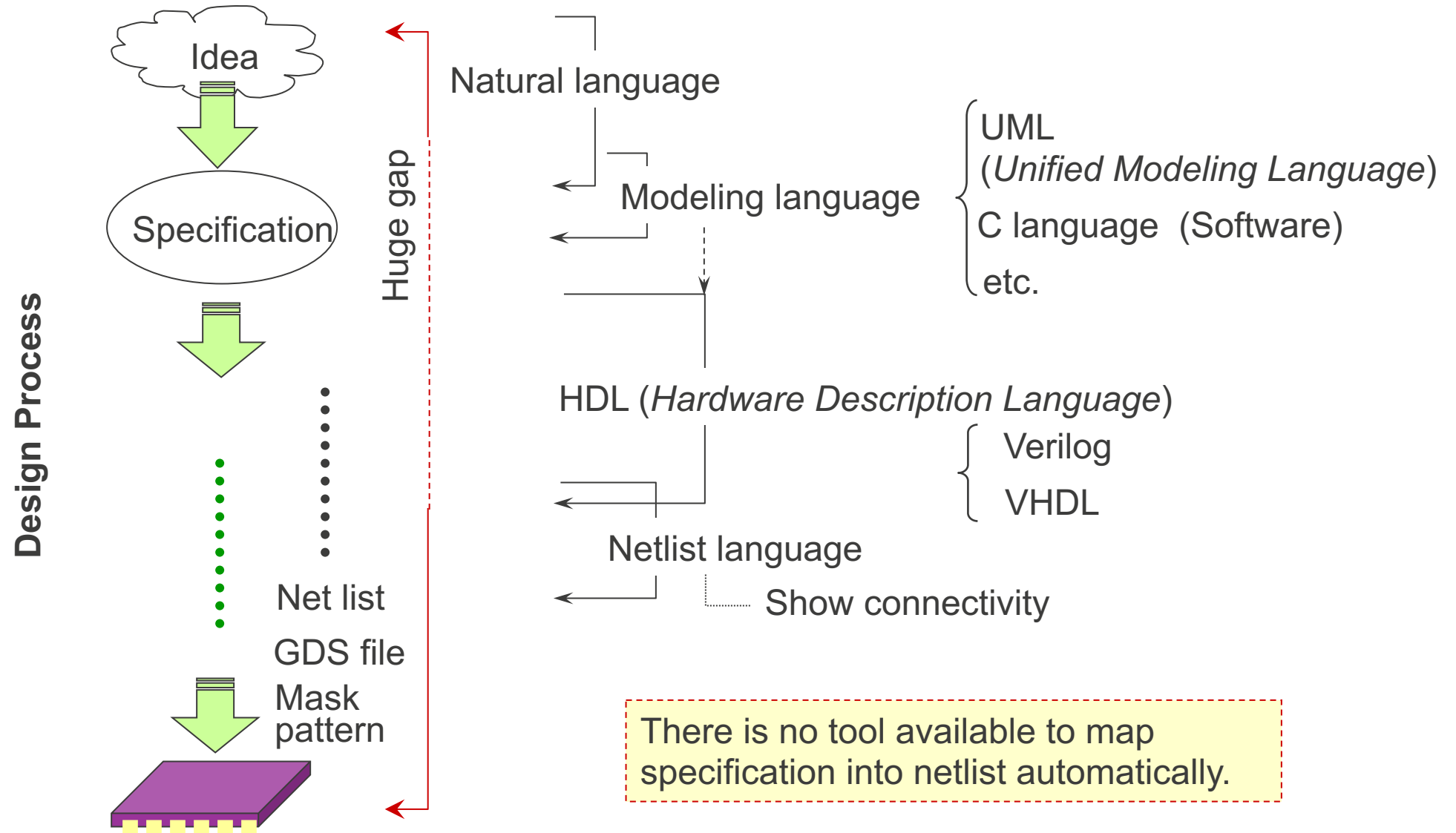


3.3 Logic Design

Logic Design Flow

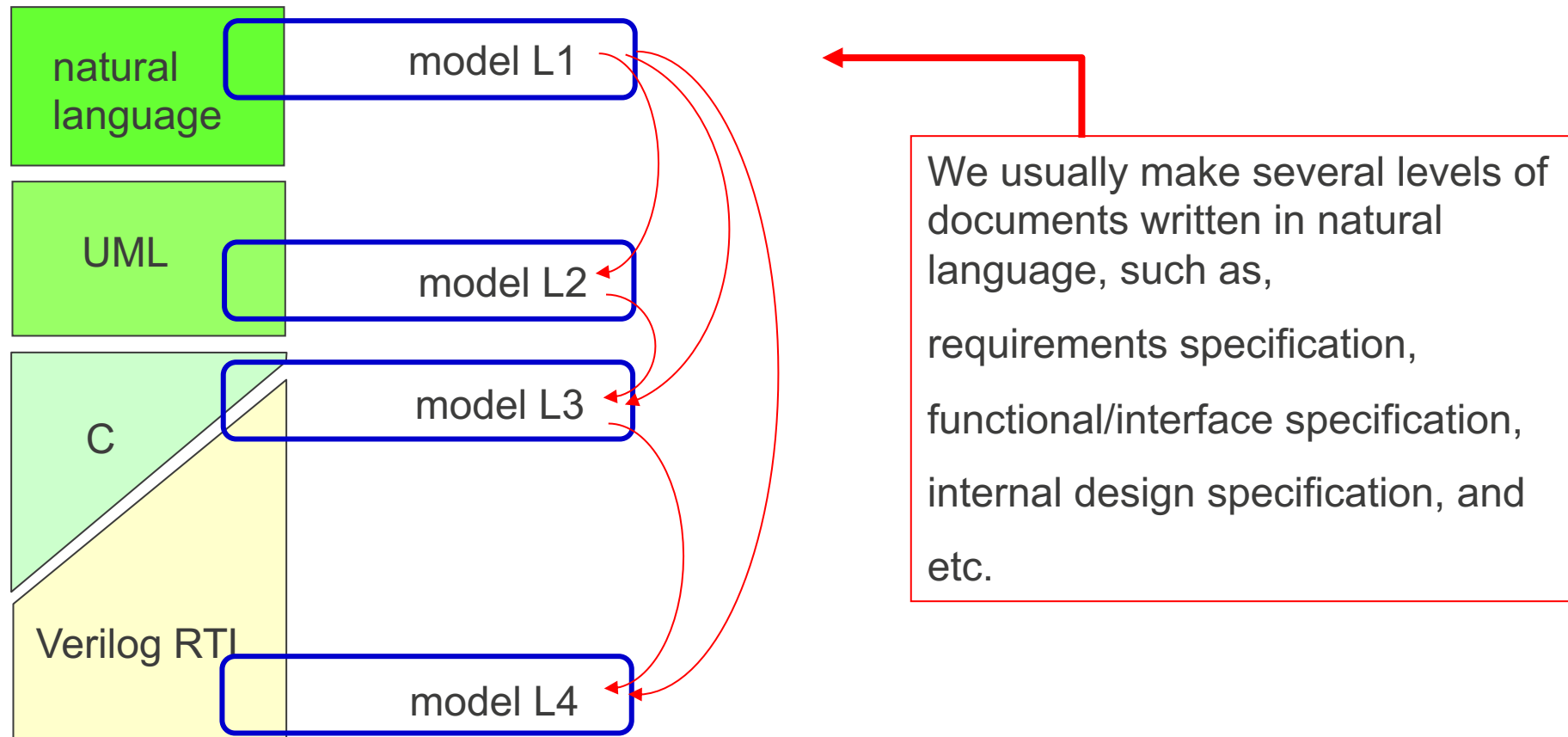


Design and Modeling (1/3)



Design and Modeling (2/3)

In general, the flow goes from high abstraction level to low abstraction level as shown below. Currently each model must be mapped into the next level manually. Some steps may be skipped, for example it is not always necessary to create UML or/and C models.



Design and Modeling (3/3)

Abstraction level

Description

Difficulty to write the code

Highest



Lowest

Focusing on **functionality only**.
Timing and hardware resource
needed can be ignored.

Order of events such as signal
change is taken into account.
(Sequence accurate)

Processing time of signal change is
taken into account. (Elapsed time
accurate)

Precise timing is taken into account.
(Cycle time accurate)

Not only functionality and timing, but
also **hardware resources needed are
described**.

Easy



Just focus on functionality only.
(*Short development time, less
bugs*)

Many aspects must be
considered at the same time.
(*Long development time, more
bugs*)

Difficult

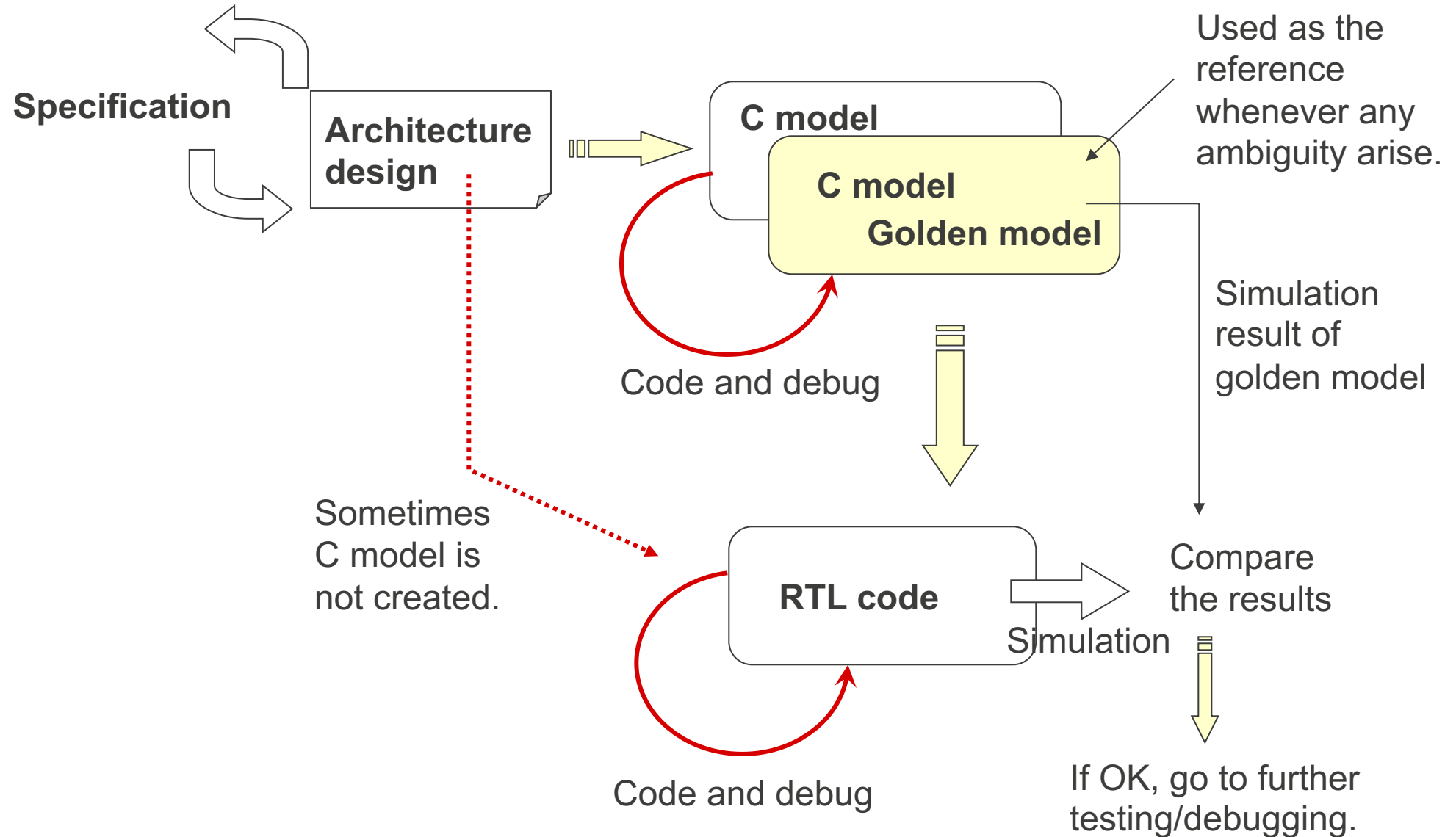
Modeling Level Example

**1: If a model has no concurrency and untimed, several behavioral synthesis tools are available, but not well developed yet.*

**2: Several behavioral synthesis tools are available, but not well developed yet*

Description level				Synthesis	Simulation	Example	Description language
Timing			Explicit hardware resource				
Sequence	Elapsed time	Machine cycle					
accurate	No	No	No	*1 No	Yes	transaction level modeling <i>untimed</i>	C, C++ System C
	Yes					<i>timed</i>	System C
accurate	accurate	Yes	No	*2 No	Yes	Cycle accurate simulation model	System C Verilog VHDL
accurate	accurate	Yes	Yes	Yes	Yes	RTL model	Verilog VHDL
accurate	accurate	Yes	Yes	—	Yes	Gate level model	Verilog-netlist

Design Process Using Models



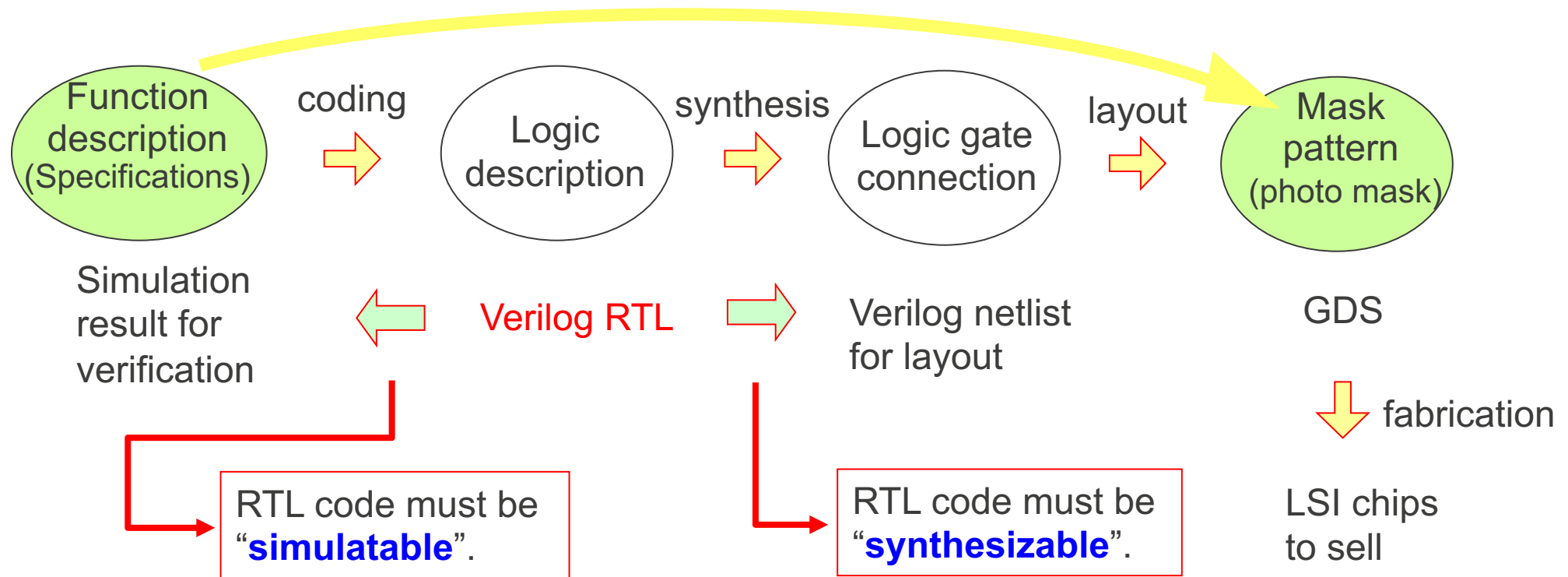
3.4 RTL Design and Verification

Objective of Logic Design

Objective of logic design

=

Get photo masks equivalent to the specifications, for fabrication purpose.

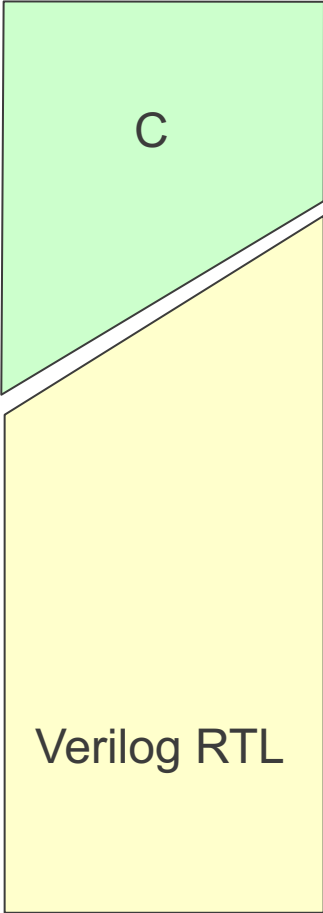
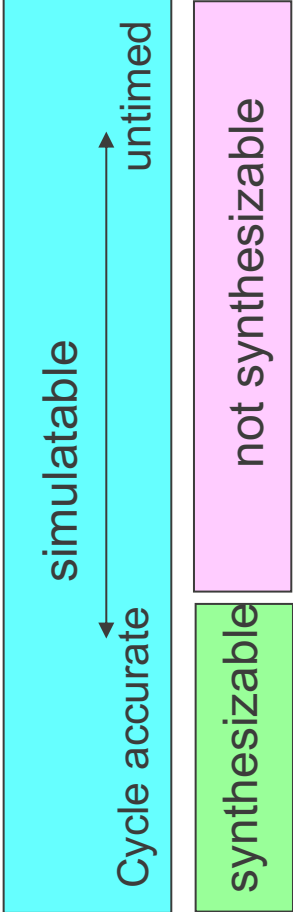


We have to write RTL code so that an EDA tool can *simulate* the code for verification.

We have to write RTL code so that an EDA tool can *create a netlist* from the code.

Design Models

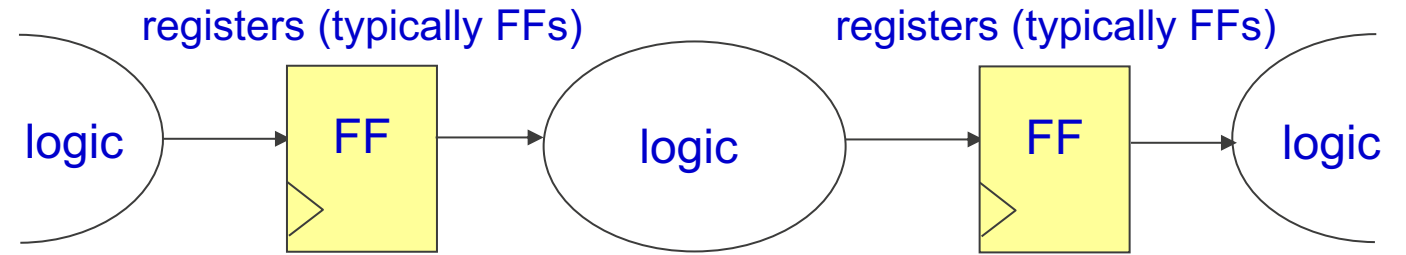
We must select a suitable abstraction level depending on the purpose of modeling the target logic.

Abstraction level	Applicable language	Feature	Comment
<div>high</div> <div>↑</div> <div>↓</div> <div>low</div>			<div>Currently, C code is not synthesizable.</div> <div>RTL code using features of Verilog which are not synthesizable is not synthesizable.</div> <div>RTL code using only synthesizable features of Verilog is synthesizable.</div>

Verilog RTL Design (1/2)

There are several levels of description in Verilog language. For front end design, we generally use **Register Transfer Level (RTL Level)**, which can describe how signals go into registers, memory elements, as shown beside.

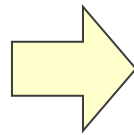
Renesas mainly uses Verilog as a high level hardware description language



Verilog RTL code for the truth table on the left

Truth table

a[3]	a[2]	a[1]	a[0]	y
1	x	x	x	q1
0	1	x	x	q2
0	0	x	0	q3
0	0	0	1	q4
0	0	1	1	q5



```
•
•
•
casez ( a[3:0] )
  4'b1??? : y = q1 ;
  4'b01?? : y = q2 ;
  4'b00?0 : y = q3 ;
  4'b0001: y = q4 ;
  4'b0011: y = q5 ;
  default : y = 4'bxxxx ;
endcase
•
•
•
```


Verilog RTL Design (2/2)

When writing RTL code, frontend designers do not have to concern on:

how to combine what kind of gates, and the
circuits of the gates which are necessary to realize the logic in the code.

Therefore, frontend designers can be free from the physical design and focus on the logic itself.

The mapping between RTL code and the circuit on the silicon is done by backend designers with a help of design automation tools.



RTL Programming Summary (1/18)

Follow the following steps to write Verilog RTL source program of a module.

Step 1. Write a header for a file you are going to create to help others understand what are written, and help yourself to remind what are written, in the file.

```
// +HC -----  
// File: snake_game_top.v  
// Module: snake_game_top  
// Function: Top module of snake game  
// -----  
// keywords: snake, game, lamp_field,  
// -----  
// Remarks:  
// -----  
// History: Version, Date, Author, Description  
// v1.0, 07.May.2011, Viet Viet, new release  
// v1.1, 12.Aug.2012, Minh Minh,  
//           game over signal timing corrected  
// -----  
// (C) Copyright 2011 Renesas Electronics Corp. All rights reserved.  
// -HC -----
```

File and module name

Functions

Key words for the file

Comments for reusers

Update history

Copyright notice

RTL Programming Summary (2/18)

Step 2. Give a unique name to a module.

```
////////////////////////////////////
```

```
// header
```

```
////////////////////////////////////
```

```
module module_name;
```

Give a unique meaningful name,
Use 5 to 16 lowercase characters.

```
endmodule
```

Step 3. Find how many inputs and outputs are needed for the module.
And give them unique meaningful names, and then list them up
in a port list in the order of clock, reset, input and output signals.

```
////////////////////////////////////
```

```
// header
```

```
////////////////////////////////////
```

```
module module_name ( clk, rst, in_a, in_b,,,, out_c, out_d,,,, );
```

```
endmodule
```

List up all the interface signals including
clock and reset in a module port list.

Give a unique meaningful name
for each signal, use 5 to 16
lowercase characters.

RTL Programming Summary (3/18)

Step 4. Declare parameters for constants, such as bit-width of signals used in the module, initial values of FFs, etc. to improve readability and reusability of the module.

```
////////////////////////////////////  
// header  
////////////////////////////////////  
module module_name ( clk, rst, in_a, in_b,,, out_c, out_d,,, ) ;  
// parameters  
    parameter BW_IN_B = 16 ; // Bit width of in_b,  
                                // BW_IN_B can be either 4, 8, 16, 32, or 64  
    parameter BW_OUT_D = 8 ; // Bit width of out_d,  
                                // BW_OUT_D can be either 2, 4, 8, 16, or 32  
  
    parameter ,,,,,,  
  
endmodule
```

Declare parameters and give comments for them.
Parameter ranges must also be declared.

RTL Programming Summary (4/18)

Step 5. Declare port using **input/output** keywords for all the signals listed up in a module port list. Use range specification for multi-bit signals.

```
////////////////////  
// header  
////////////////////  
module module_name ( clk, rst, in_a, in_b,,,, out_c, out_d,,,, );  
    parameter BW_IN_B = 16 ; // Bit width of in_b,  
                                // BW_IN_B can be either 4, 8, 16, 32, or 64  
    parameter BW_OUT_D = 8 ; // Bit width of out_d,  
                                // BW_OUT_D can be either 2, 4, 8, 16, or 32  
  
    parameter ,,,,,,
```

// ports

```
input clk, rst ; // clock and reset signal  
input in_a; // comment, ,,,,,,  
input [BW_IN_B -1 :0] in_b ; // ,,,,  
output out_c ; // ,,,,,,,  
output [BW_OUT_D -1 :0] out_d ; // ,,,,,,
```

Declare input and output
for all the signals listed in
a module port list.
Use range specification
for multi-bit signals.

```
endmodule
```

Give comments to describe each port.

RTL Programming Summary (5/18)

Step 6. Declare data type for **input ports using wire** keyword.

```
////////////////////
// header
////////////////////
module module_name ( clk, rst, in_a, in_b,,, out_c, out_d,,, );
// parameters
    parameter ,,,
    parameter ,,,,,,
// ports
    input clk, rst ;
    ,,,,
    ,,,
    output [BW_OUT_D -1 :0] out_d ;
// input data type
    wire ckl, rst ;
    wire in_a ;
    wire [BW_IN_B -1:0] in_b ;

endmodule
```

} Declare data type of **all inputs**
by using **wire** keyword. Use
the same range specification
for multi-bit signals.

RTL Programming Summary (6/18)

Step 7-1. Find if **output signals** can be described by using **continuous assign** statement or not. For those output signals which can be described as “assign out_c = some_expression ;”, declare their data type by using **wire** keyword.

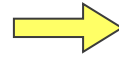
```
////////////////////////////////////
// header
,,,,,,,,
,,,,
// input data type
wire ckl, rst ;
wire in_a ;
wire [BW_IN_B -1:0] in_b ;
,,,
// output data type
wire out_c ;
wire [BW_OUT_D -1:0] out_d ;

endmodule
```

} Declare data type of **outputs** by using **wire** keyword if they appear on LHS of **continuous assign** statements.
Use the same range specification for multi-bit signals.

RTL Programming Summary (7/18)

Step 7-2. For those **output signals** which can not be defined by continuous assign statements and have to be defined by procedural assign statements, declare their data type by using **reg** keyword.



Use reg keyword if a signal **appears on LHS** of procedural assign statements.

```
//////////////////////////
// header
,,,,,,,,
// output data type
,,,,,
''
wire [BW_OUT_D -1 :0] out_d ;
reg out_f ; // non-FF
reg [BW_OUT_G -1 :0] out_g ; // FF
reg [BW_OUT_P -1 :0] out_p ; // non-FF

endmodule
```

Give comments to describe output arguments. And also give note to make it clear that if they are to be mapped into FF or not.

Use the same range specification for multi-bit signals.

RTL Programming Summary (8/18)

Step 8. Declare **internal signals** by using wire or reg keyword depending on if they appear on LHS of continuous assign or procedural assign statements.

```
//////////  
// header  
''  
''  
reg [BW_OUT_P-1:0] out_p ; // non-FF  
// internal signals  
wire intnl_sig_a ; //,,,  
wire [BW_SIG_B-1:0] intnl_sig_b ; // ,,,  
reg intnl_sig_c ; // non-FF  
reg [BW_SIG_D-1:0] intnl_sig_d ; // FF  
  
endmodule
```

Declare data type of internal signals.

Use wire keyword if they appear on LHS of continuous assign statements,

use reg keyword if they appear on LHS of procedural assign statements.

Use range specification for multi-bit signals.

Give comments for reg data type to make it clear that they are to be mapped into FF or not.

All internal signals must be commented to describe what they are used for.

RTL Programming Summary (9/18)

Step 9. Describe logic using continuous assign statements, procedures and module instantiation.

```
////////////////////
// header
////////////////////
module module_name ( clk, rst,
// parameters
    parameter ...,
    ...,
// ports
    input clk, rst ;

    ...,
// data type
    wire ...,
// internal signals
    wire intnl_sig_a;

    ...
    // start logic description

endmodule
```

```
assign sig_r = sig_s & sig_t ;
assign sig_u = ( sig_f )? sig_w : sig_v ;
```

```
m_name m_name_01 ( ,,,, ) ;
m_name m_name_02 ( ,,,, ) ;
```

```
function func_name ;
endfunction
```

```
always ,,,, begin
end
```

```
initial begin
end
```

```
task task_name ;
endtask
```

RTL Programming Summary (10/18)

The following set of code lines must be written near to each other if they are related to the same logic block. **Do not mix up unrelated code lines together.**

```
assign sig_r = sig_s & sig_t ;  
assign sig_u = ( sig_f )? sig_w : sig_v ;
```

```
m_name m_name_01 ( ,,,, ) ;  
m_name m_name_02 ( ,,,, ) ;
```

```
function func_name ;  
endfunction
```

```
,,  
always ,,,, begin  
end
```

```
task task_name ;  
endtask
```

RTL Programming Summary (11/18)

Step 9-1. Define FFs by using always constructs with posedge clk in a sensitivity list.

```
////////////////////  
// header  
''''  
''  
// start logic description
```

```
// always for FF  
always @ ( posedge clk or negedge rst_n ) begin  
    if ( rst_n == 1'b0 ) begin  
        sig_ff <= INTL ;  
    end  
    else begin  
        sig_ff <= next_sig_ff ;  
    end  
end
```

- (1) LHS must be declared by **reg** keyword.
- (2) Use **nonblocking** procedural assignment.
- (3) Write **change direction event of clock and reset** (for asynchronous reset) in the sensitivity list.

Write FFs using
always constructs.

```
endmodule
```

Apply delay to avoid racing, if the project policy says so.

RTL Programming Summary (12/18)

Step 9-2. Describe logic using continuous assign statements.

```
////////////////////  
// header  
",,"  
",  
// start logic description  
// always for FF  
always @ (posedge clk ,,, )  
",,"  
",  
// continuous assigns  
assign sig_w = sig_u & sig_y ; // comment  
assign sig_h = ( sig_en )? sig_s : sig t ; // comment  
assign ,,,,,  
  
endmodule
```

} Write logic blocks
using **continuous**
assign statements.

- (1) LHS must be declared by **wire** keyword.
- (2) Do not use @ nor #.

RTL Programming Summary (13/18)

Step 9-3. Describe logic by instantiating lower level modules.

```
//////////  
// header  
'''  
''  
// start logic description  
// always for FF  
''  
// continuous assigns  
assign sig_w = sig_u & sig_y ; // comment  
assign sig_h = ( sig_en )? sig_s : sig t ; // comment  
assign ,,,,,,  
// module instantiation  
m_name m_name_01 ( ,,,,, ) ;  
m_name m_name_02 ( ,,,,, ) ;  
  
endmodule
```

Syntax

```
module_name module_name_01  
    ( .port1_name(connecting_signal_name),  
      .port2_name(connecting_signal_name),  
      ,,,,,  
    );
```

(1) Output port must be connected to wire.
(2) Input port can be connected to wire or reg.



RTL Programming Summary (14/18)

Step 9-4. Define combinational logic blocks by functions.

```
////////////////////  
// header  
'''  
''  
assign ,,,,,  
// module instantiation  
m_name m_name_01 ( ,,,,, );  
''  
// functions  
function [BW_FUNC -1:0] func;  
input [BW_F_IN -1:0] f_in_a ;  
  
'''  
func = ,,,, ;  
endfunction  
  
endmodule
```

Syntax

```
function range_declaration function_name ;  
input range_declaration input_name ;  
input range_declaration input_name ;  
reg local_wk ;
```

```
function_name = ,,,, ;  
endfunction
```



- (1) Internal work must be declared by reg keyword.
- (2) Do not use @ nor #.
- (3) Use blocking procedural assignment only.
- (4) Declare range if func is multi-bit.

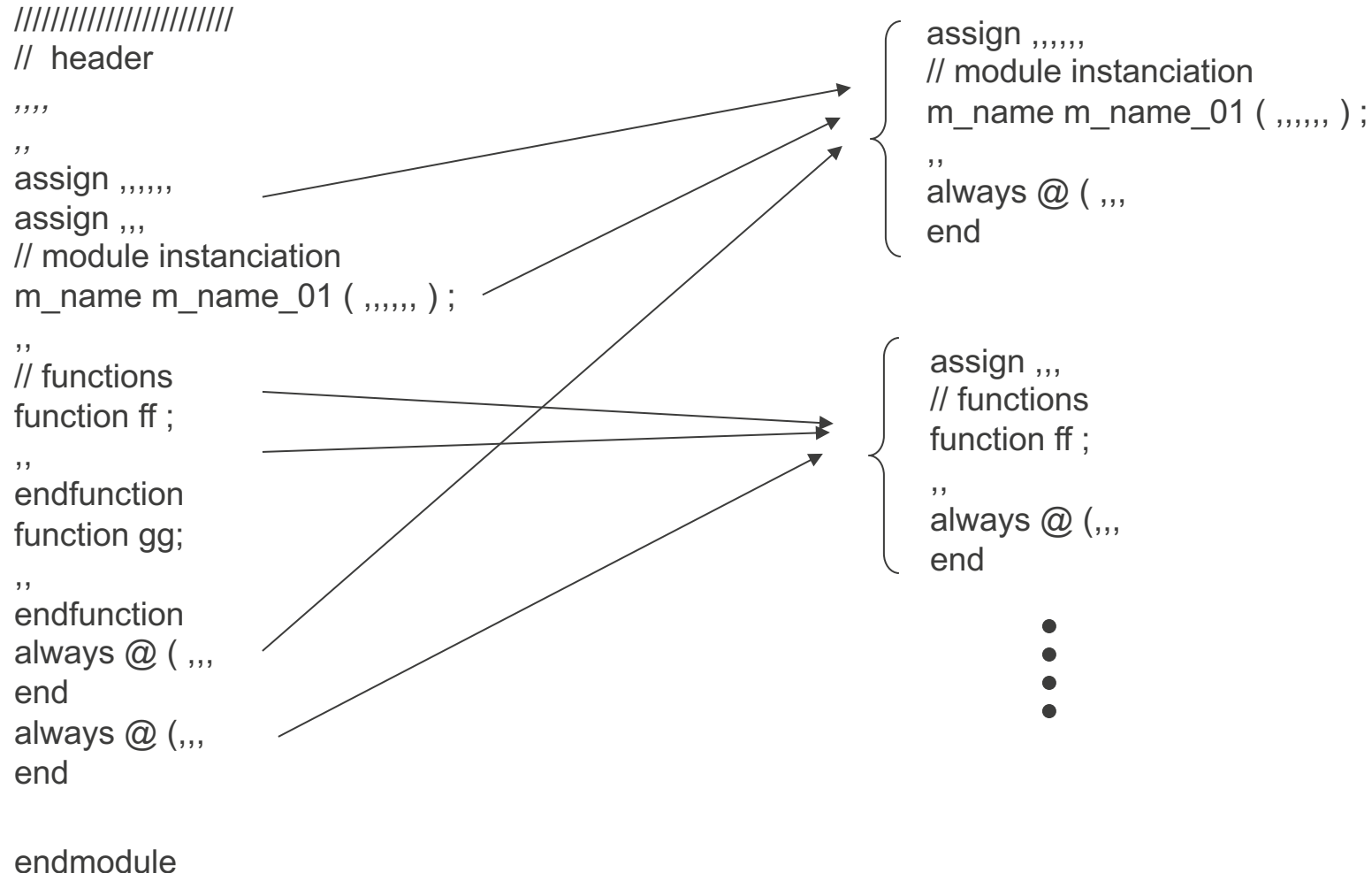
RTL Programming Summary (15/18)

Step 9-5. Define combinational logic blocks by always constructs.

```
//////////  
// header  
  
,,,  
,,  
assign ,,,,  
// module instantiation  
m_name m_name_01 ( ,,,, ) ;  
  
,,  
// functions  
  
,,  
// always for combinational logic  
always @ ( sig_a or sig_b ,,, ) begin  
if ( ,,,, ) ,,  
  
,,,  
sig_w = ,,, ;  
end  
  
endmodule
```


RTL Programming Summary (16/18)

Step 10. Review the code lines. If code lines related to the same logic block are distributed among other code lines, move them into one part so that the logic part can be seen at a glance.



RTL Programming Summary (17/18)

Step 11. Review the code lines. If code lines creating flags are placed far from the logic block where the original signal is generated, move them near to the original signal generating block.

```
assign sig_a = ,,, , ;
```

```
// module instantiation
```

```
m_name m_name_01 ( ,,, , ) ;
```

```
''
```

```
// functions
```

```
function gg;
```

```
''
```

```
endfunction
```

```
always @ ( ,,,
```

```
end
```

```
//
```

```
assign en_flag = | sig_a;
```

```
always @ ( ,,,
```

```
if ( en_flag ) begin
```

```
''
```

```
''
```

```
end
```

```
endmodule
```

Move this flag signal generating code line into the code block generating sig_a.

RTL Programming Summary (18/18)

Step 12. Review the code lines to check if readability is good and if they are easy to understand and to reuse. Improve the understandability, readability, and reusability.

RTL Verification – Where Bugs come from?

Misunderstanding. Poor imagination

Customer's
expectation

Implicit requirements

Specification

Ambiguous and/or
insufficient description

Interfaces and
internal logics

Ambiguous and/or
insufficient description

RTL programs

Incorrect coding,
tool dependent coding

Misunderstanding.
Poor imagination.

Misunderstanding.
Insufficient/wrong
programming knowledge.
Careless miss

Example:

For three events A, B and C, the logic was designed for those events coming in sequence of A-B-C, or B-A-C, or B-C-A or C-A-B or C-B-A.

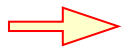
However a poor imaginative designer cannot think of the sequence A-C-B. And this sequence may result in failure of the logic.

RTL Verification – Desktop Checking (1/3)

Checking a logic on a desk by “hand” (eyes) is still an effective way to verify logics if it is done in a way that every assumption a designer made is checked whether they are really true or not.

You must do desktop check to make sure that you really understand the issues in your design.

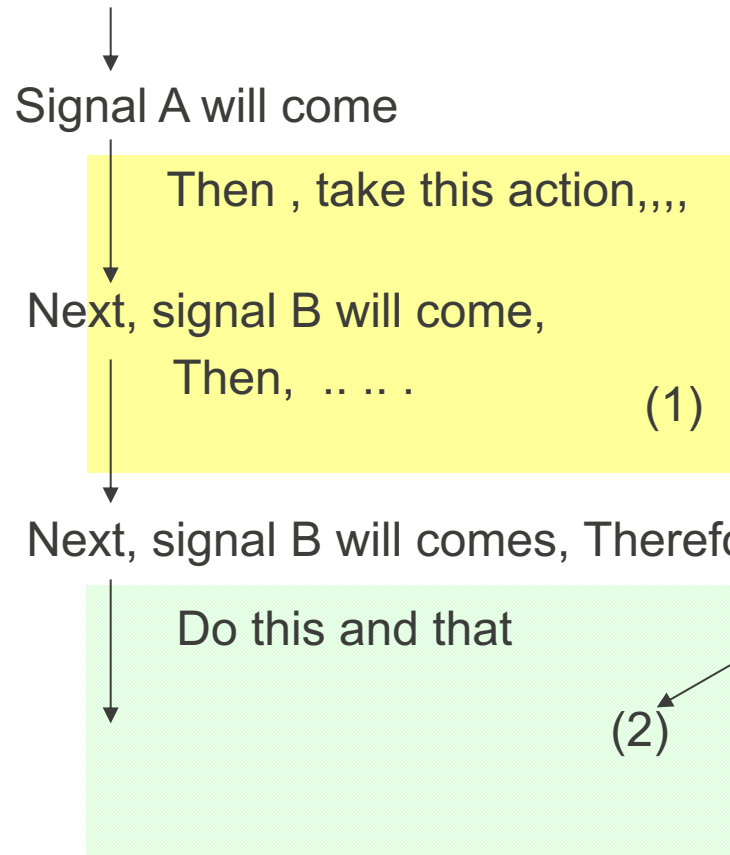
Sometimes RTG (Random Test Generator) can checkout bugs by providing test data which a designer may never imagine. However, this does not mean RTG can replace desktop checking.



Logics designed by an engineer with poor imagination have a lot of bugs.

RTL Verification – Desktop Checking (2/3)

You will code the sequence you think of. \Rightarrow There is little chance for bugs to sneak into such sequences.



Sometimes, engineers cannot take specific cases into consideration and there is a big risk of bugs being implemented in such cases.

When processing (2), the environment may vary from that of (1). However, very often a designer tends to forget about the possibility that his assumption becomes invalid.

Always check whether the basic assumption you have made is still true or not .

RTL Verification – Desktop Checking (3/3)

Especially, following points must be checked all the time during the design activities.

What happens might **not happen in the way you hope**.

(The input which you expect may not come when you think it will come).

Events may happen in a **different sequence to that you expected**.

(Event A will come before Event B, when you assume B will come before A.

Or Event A and B happen at the same time, when you expect them to come in a sequence).

A presupposed condition **may not realized**.

Some blocks may have no power when you think all the blocks are powered up.

Some blocks may not be initialized, when you think all the blocks are initialized.

A clock signal is dead, when you think the clock is always alive.

Some voltage may be supplied, even after the power is off.

A value which must be fixed, **may not be defined yet**.

(Not initialized, Not given a predetermined value, Unstable, , , ,)

RTL Verification – RTL Checker (Style Check)

Checkout naming and coding style which are prohibited or not recommended.

RTL description

```
.....  
always @(i) begin  
  casez (i)  
    3'b???1: o = 1'b0;  
    3'b?10: o = 1'b1;  
    3'b100: o = 1'b0;  
  endcase  
end  
.....
```

(case, 3'b000 is missing)

**RTL
Checker
[SpyGlass]**

Checking rule

Check results

```
.....  
Error  
Case statement is missing  
cases and has no default  
("casez (i)")  
.....
```

- Rule creation is important
- Sometimes customization is needed

RTL Verification – Static Functional Verification

Given a set of properties that describe the behavior of the design, static functional verification tools can prove proper functionality of the design using formal verification techniques.

What are checked out?

Properties given by a designer hold true or not.
If not true, it gives a counterexample.

Available tools

Solidify (Averant), 0-In (Mentor)

Although static functional verification cannot handle large size logic, it is suitable for testing logics which need huge combination of input patterns such as ECC, pattern matching logics, or FIFO, etc.

Static verification does not do logic simulation, therefore no test input is needed. It checks if properties given are true or not.

RTL Verification – Assertion Based Verification

Define properties of signals in target modules using a language such as PSL (Property Specification Language)

- ➡ Run simulation on a simulator supporting assertion, such as NC-Verilog (LDV5.0 or later)
- ➡ When violation detected, error report is given.

“Must” and “must not” can be declared.

Signal A must rise 5 clock cycles after signal B falls.

Signal A and B must not be 1 at the same time.

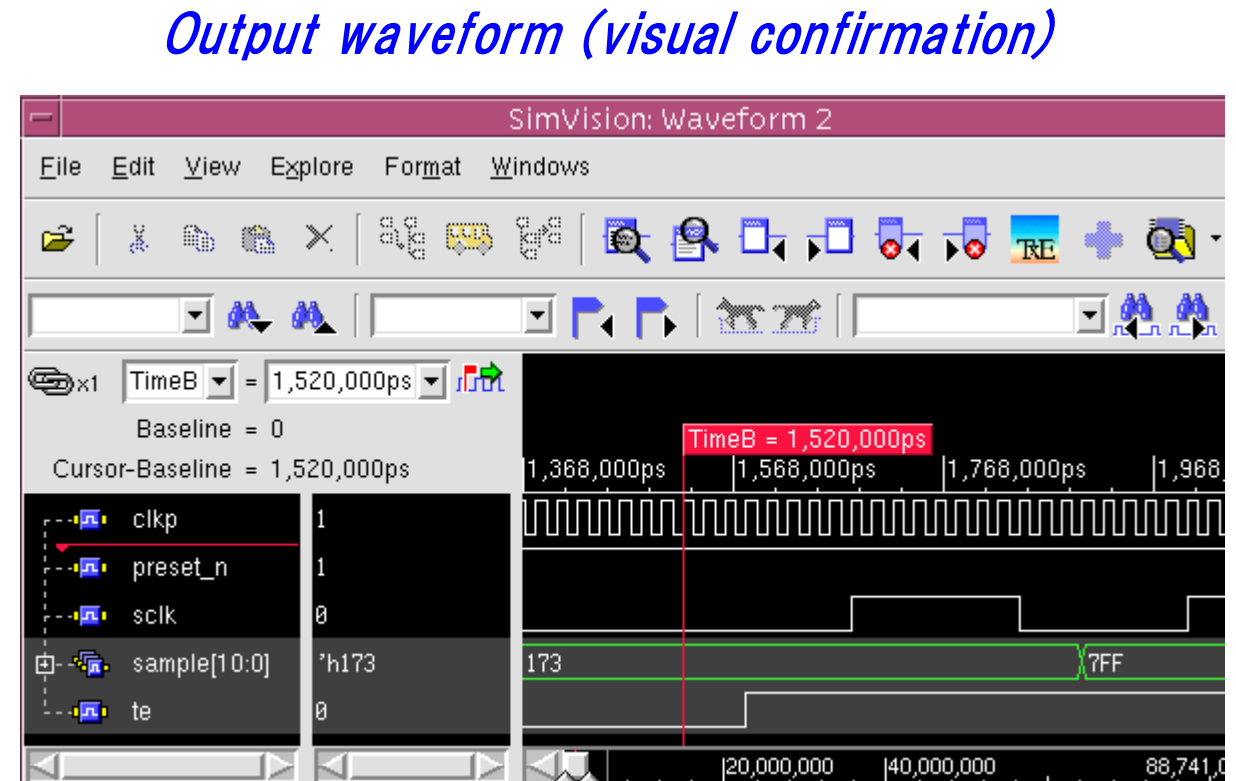
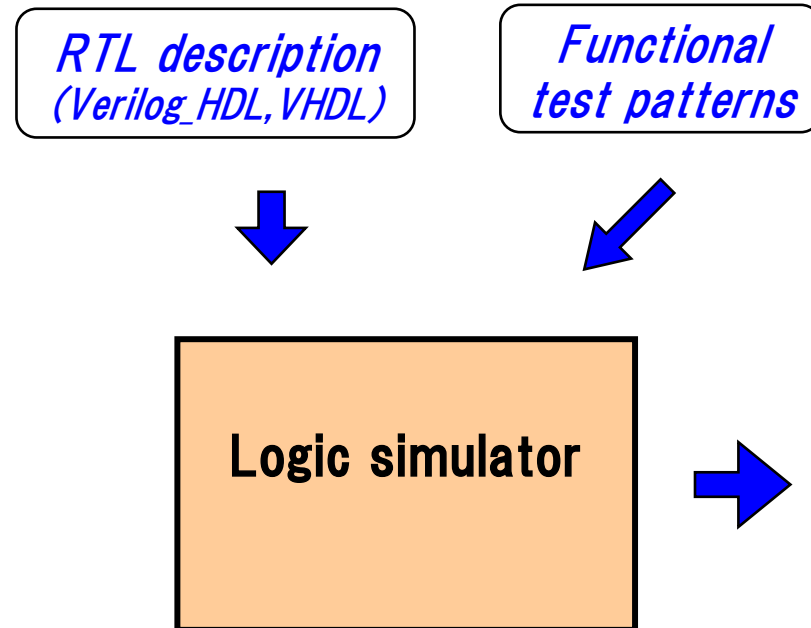
⋮

➡ Especially useful for checking improper reuse of modules.

➡ Expected to improve reusability or reduce troubles using IPs.

RTL Verification – Functional Logic Simulation (1/4)

To check expected simulation output by applying functional test patterns to RTL description



RTL Verification – Functional Logic Simulation (2/4)

By using RTL simulator, we can see if the logic works as intended by checking the outputs.

RTL simulation takes time. Therefore, to make it time efficient, test data must be carefully prepared to test critical or corner cases.

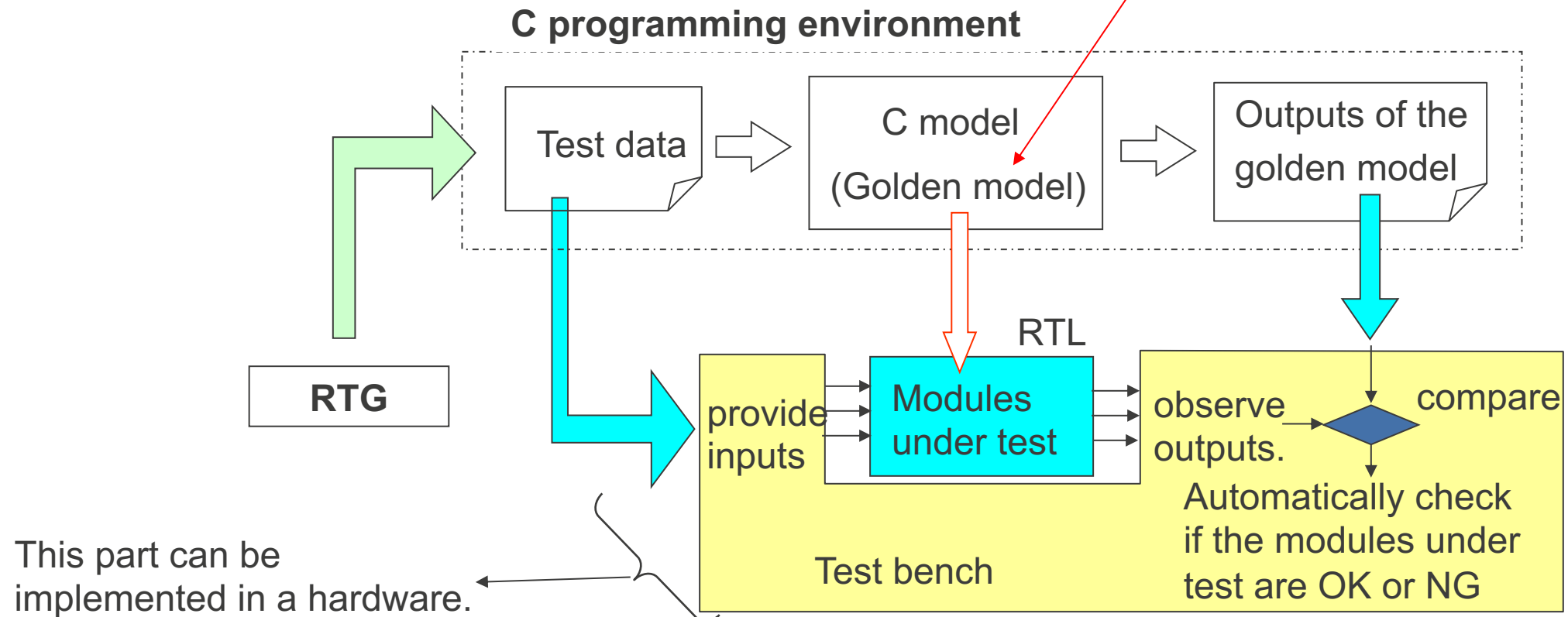
There are several ways to help RTL simulation time short.

- (a) To reduce test data preparation time: Random Test Generator (RTG)
- (b) To reduce simulation time: Early prototyping with FPGA or emulator

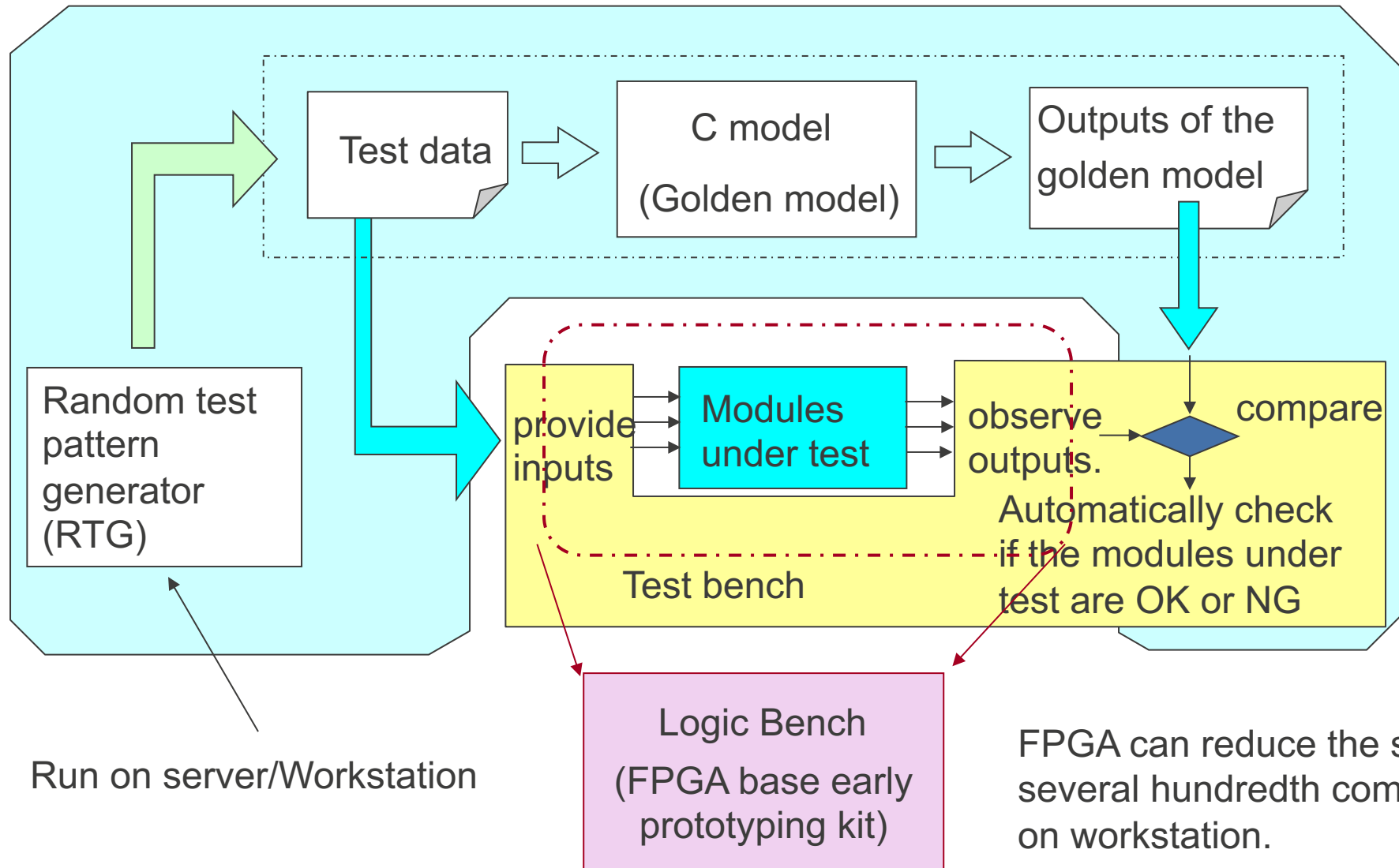
RTL Verification – Functional Logic Simulation (3/4)

RTG creates so many data patterns, therefore it is not feasible to see the result and check if it is OK or not by hand. Therefore, usually we use a **golden model** as a reference and compare the result as shown below.

A model which is believed to be correct and can be used as a reference for RTL code is called the golden model.



RTL Verification – Functional Logic Simulation (4/4)

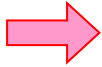


RTL Verification – Test Bench vs Target Code

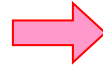
Target code

Test bench

Good



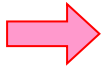
Wrong



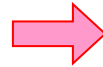
Good code may be rejected



Wrong



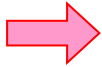
Wrong



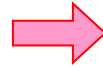
Wrong code may be accepted.
Bugs may be overlooked



Wrong



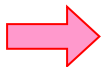
Good



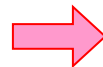
Wrong code is rejected



Good



Good



Good code is accepted



Do not think
“My code is
correct,
because the
simulation
result is good.”

Damage of wrong test bench



Damage of wrong target code

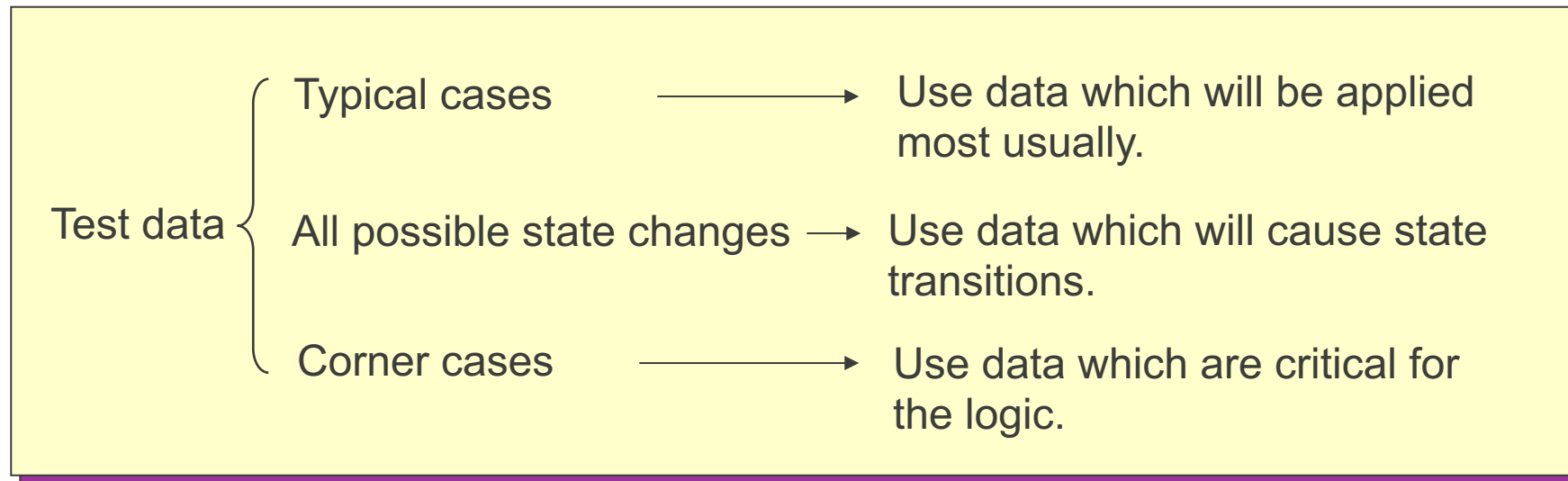
Test bench is much more vital for our products to assure that our products are bug free.

RTL Verification – Test Data

While programming RTL code, it is important to assume various data as input and make your code prepared for such data. Your code **will not work properly** for data which you **did not expect to come**. This means **your imaginative power decides the quality of your program**.

It is very important for an engineer to be able to select or determine proper data to test a module he/she designed.

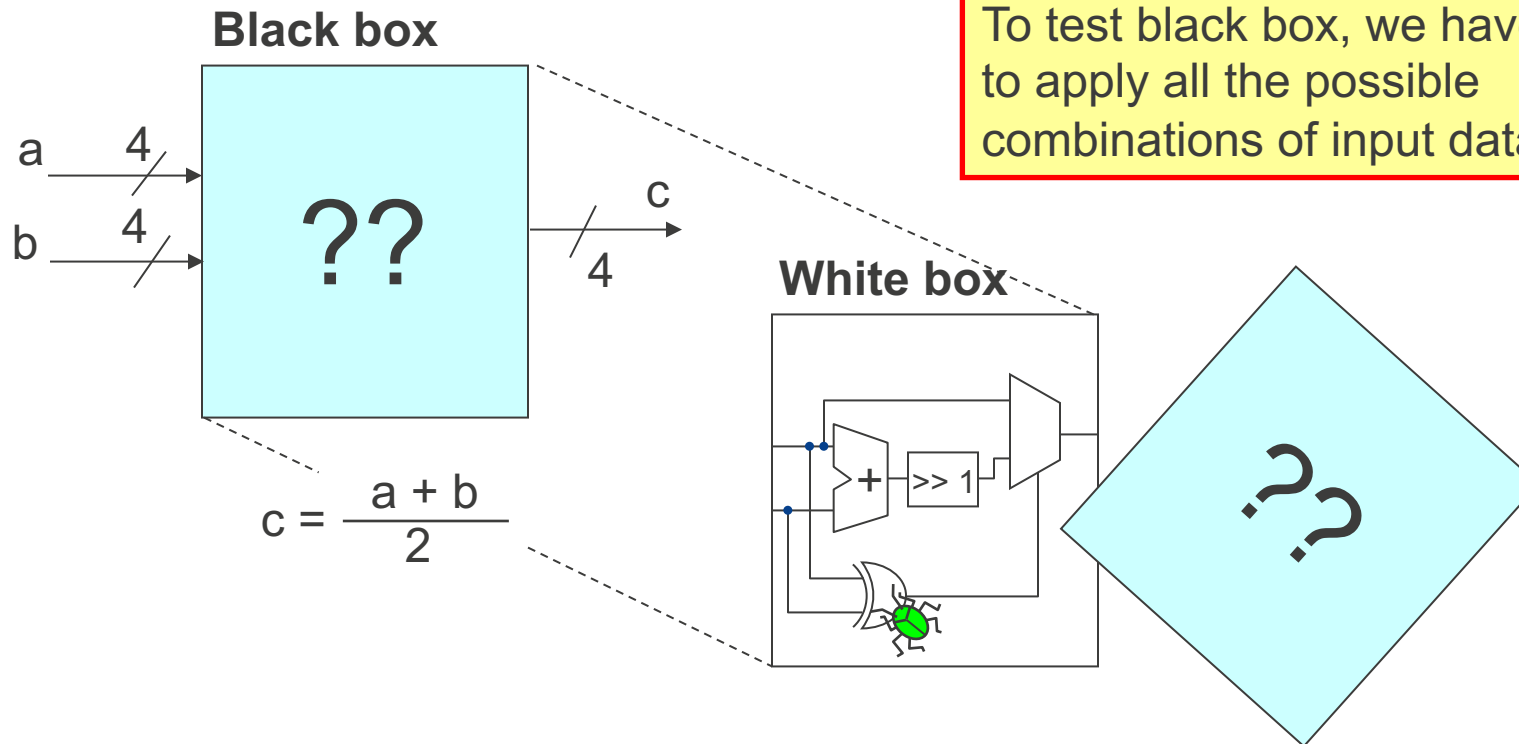
Test data shall be selected so that all the possible paths of your code are covered.



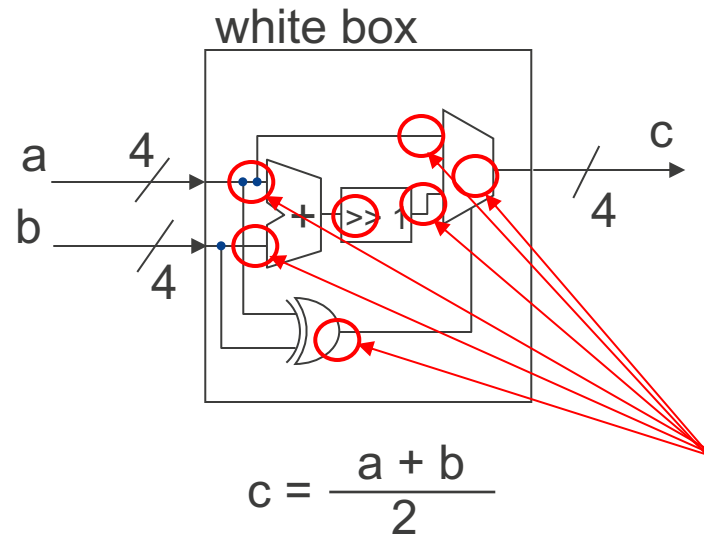
RTL Verification – Black Box vs White Box Test (1/2)

Black box test : Test **without** knowledge of internal structure and logic.

White box test : Test **with** knowledge of internal structure and logic.



RTL Verification – Black Box vs White Box Test (2/2)



To test white box, we do not have to apply all the possible combinations of input data. We can apply selective input data to check specific part of the design.

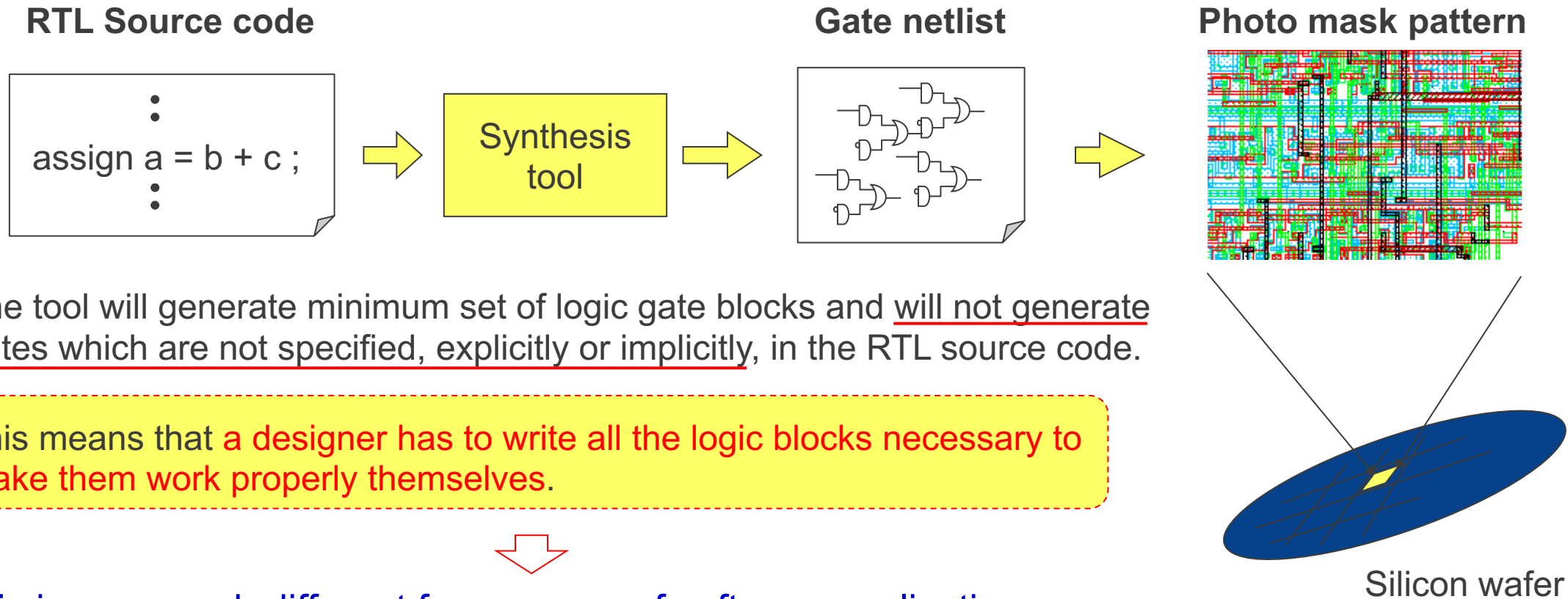
If the target is white box, we can create selective input data to activate specific part of the logic.

And once checked with some data, it may not have to be checked by using all the possible values. **Typical and corner case data may be enough to be applied.**

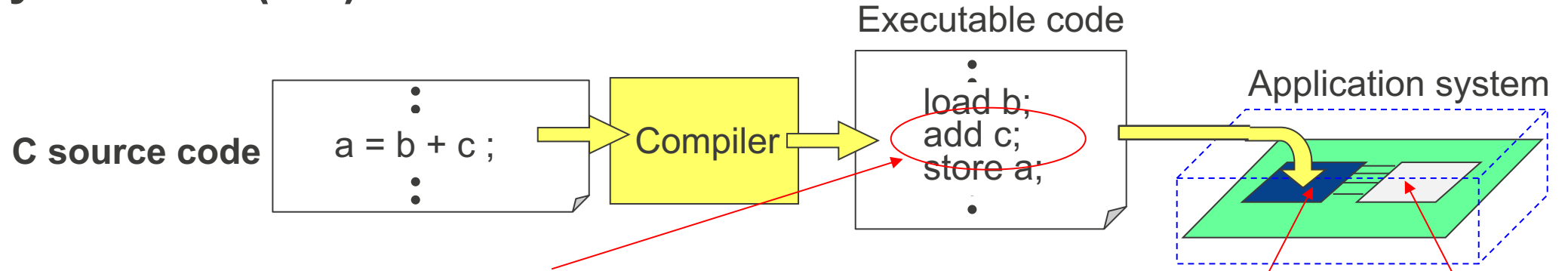
3.5 Logic Synthesis and Cell-based Design

Logic Synthesis (1/3)

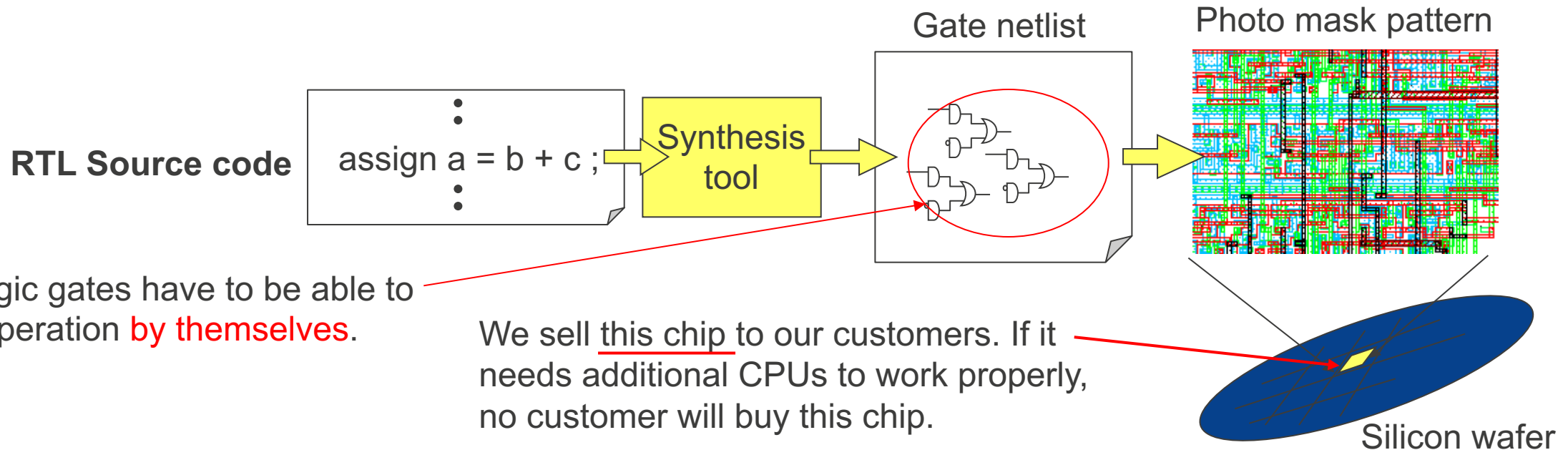
A synthesis tool uses a Verilog RTL source program as an input and generates a gate netlist which can be directly mapped to circuits on a silicon wafer.



Logic Synthesis (2/3)



This add instruction can do add operation **with a help of a CPU** implemented in an application system. Without a CPU, it cannot do anything.

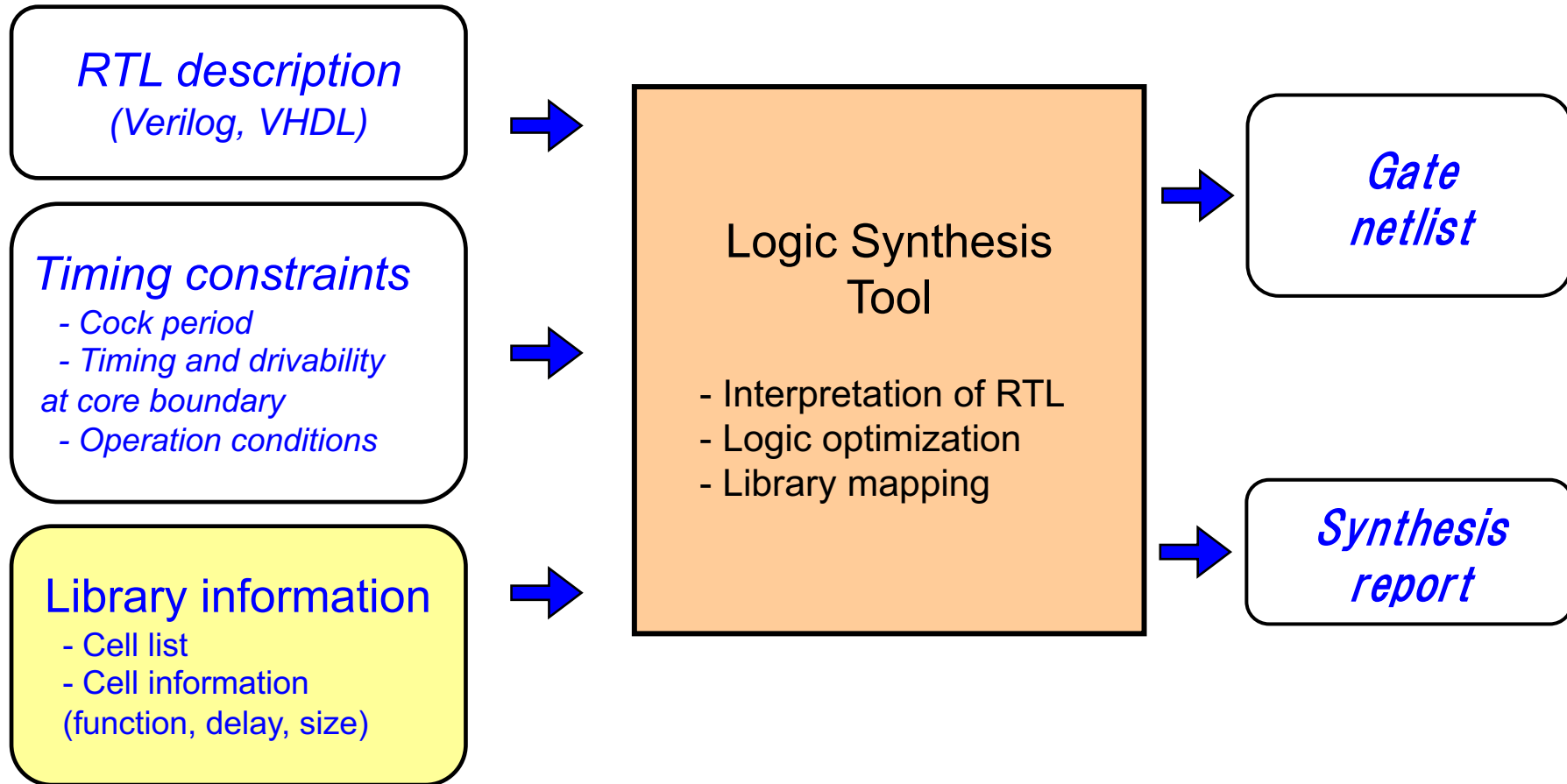


These logic gates have to be able to do add operation **by themselves**.

We sell this chip to our customers. If it needs additional CPUs to work properly, no customer will buy this chip.

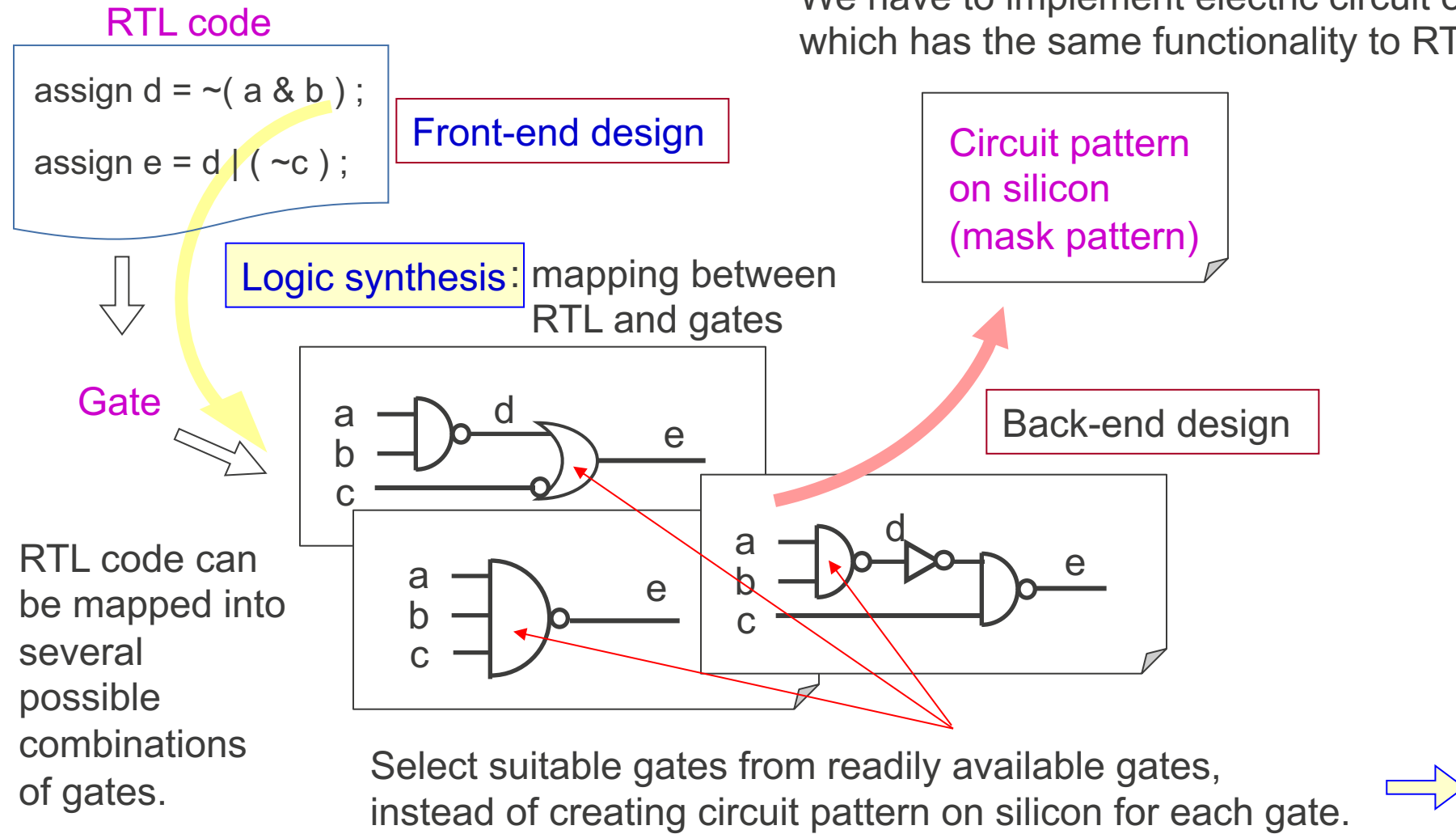
Logic Synthesis (3/3)

Synthesis: RTL description is converted to gate-level description.



Logic Synthesis and Cell-Based Design (1/4)

We have to implement electric circuit on silicon to get a chip which has the same functionality to RTL code we designed.



Logic Synthesis and Cell-Based Design (2/4)

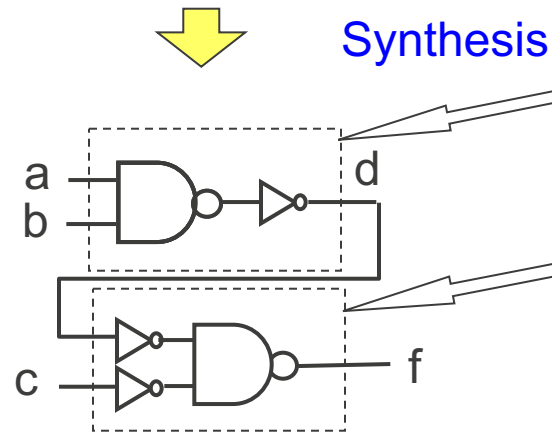
Example.

Verilog RTL code

Synthesis process

```
assign d = a & b ; // <1>  
assign f = d | c ; // <2>
```

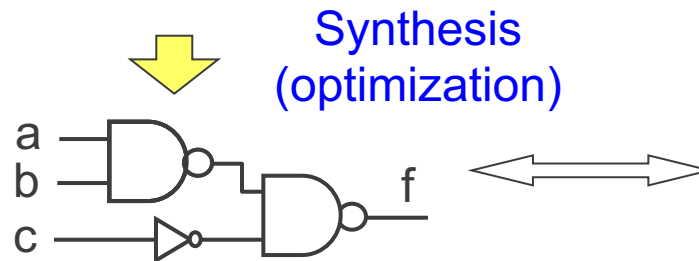
(1) Each assign statement is mapped to a gate logic block.



```
{ not01d1 INST1(.a1(E), .zn(D)) ;  
  nand02d1 INST2(.a1(A), .a2(B), .zn(E)) ;  
}  
  
{ not01d1 INST3(.a1(C), .zn(G)) ;  
  not01d1 INST4(.a1(D), .zn(H)) ;  
  nand02d1 INST5(.a1(G), .a2(H), .zn(F)) ;  
}
```

connection

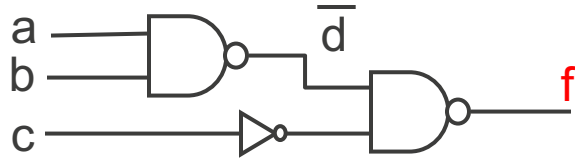
(2) Optimize and determine which circuit shall be used on a silicon.



```
not01d1 INST1(.a1(C), .zn(G)) ;  
nand02d1 INST2(.a1(A), .a2(B), .zn(E)) ;  
nand02d1 INST3(.a1(E), .a2(G), .zn(F)) ;
```

Cell name

Logic Synthesis and Cell-Based Design (3/4)

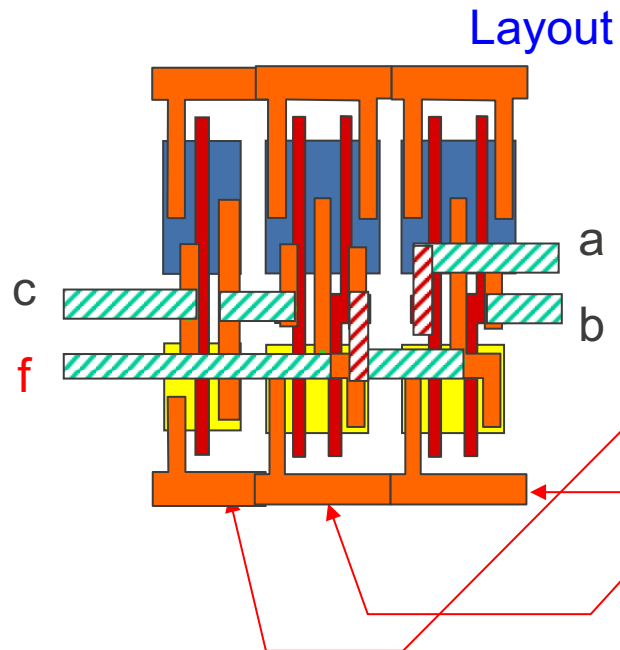


NAND gate is much smaller and efficient than AND and OR gates.

Cells are select from a standard cell library depending on a design constraint such as minimize area and/or power consumption, maximize speed, etc.

Example:

If *f* has to drive four gates, then nand02d4 is used instead of nand02d1 as shown below.



```
not01d1 INST1(.a1(C), .zn(G)) ;  
nand02d1 INST2(.a1(A), .a2(B), .zn(E)) ;  
nand02d4 INST3(.a1(E), .a2(G), .zn(F)) ;
```

The drivability of this cell is 4.

Verilog gate level programming

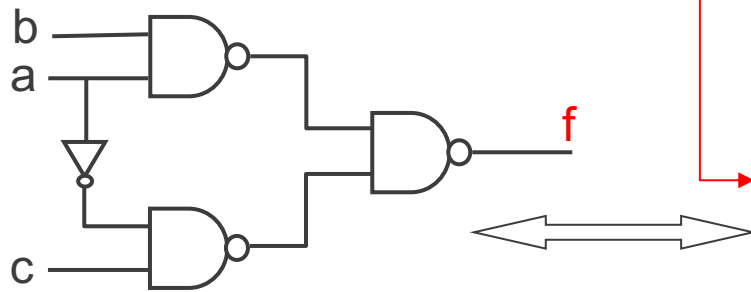
Logic Synthesis and Cell-Based Design (4/4)

While synthesizing, procedures are mapped into a logic gate blocks procedure by procedure base.

Verilog RTL code

```
always @ ( a or b or c ) begin  
  if (a) begin f = b ; end  
  else begin f = c ; end  
end
```

↓ synthesis



Synthesis process

(1) Each procedure is mapped to a logic gate block.

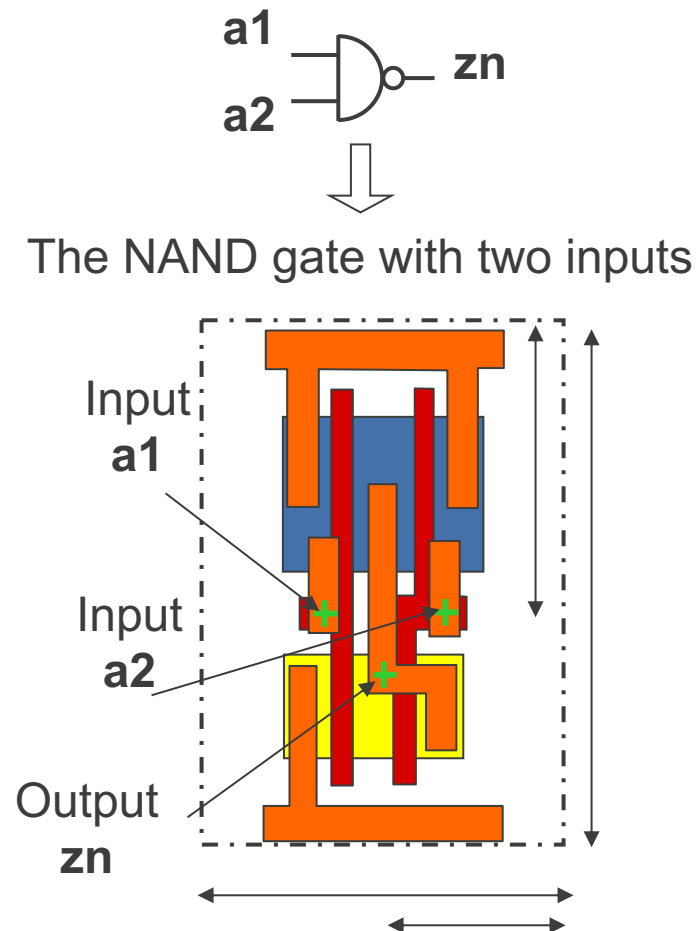
(2) Optimize and determine which circuit shall be used on a silicon.

```
not01d1 INST1(.a1(B), .zn(D)) ;  
nand02d1 INST2(.a1(A), .a2(B), .zn(E)) ;  
nand02d1 INST3(.a1(D), .a2(C), .zn(G)) ;  
nand02d1 INST4(.a1(E), .a2(G), .zn(F)) ;
```

3.6 Gate Level Design and Verification

Implement The Logic on Silicon (1/4)

EDA tools handle the following information for implementing the logic on to the silicon



1. The information connection between cells
2. The feature information of the cell
(Function, input/output, the drive power, etc.)
3. The information on the structure of the cell
(Width, the height, the terminal position, etc.)
4. The information on the inner structure of the cell
Diffusion layer size/position, gate, inner wiring, etc.
5. Layout rules
(Minimum distance between cells, etc.)
6. Wiring rules
(Minimum pitch, multi-via, etc.)

Library information

The frontend library

Information necessary to choose the cells
Information about delay and drivability

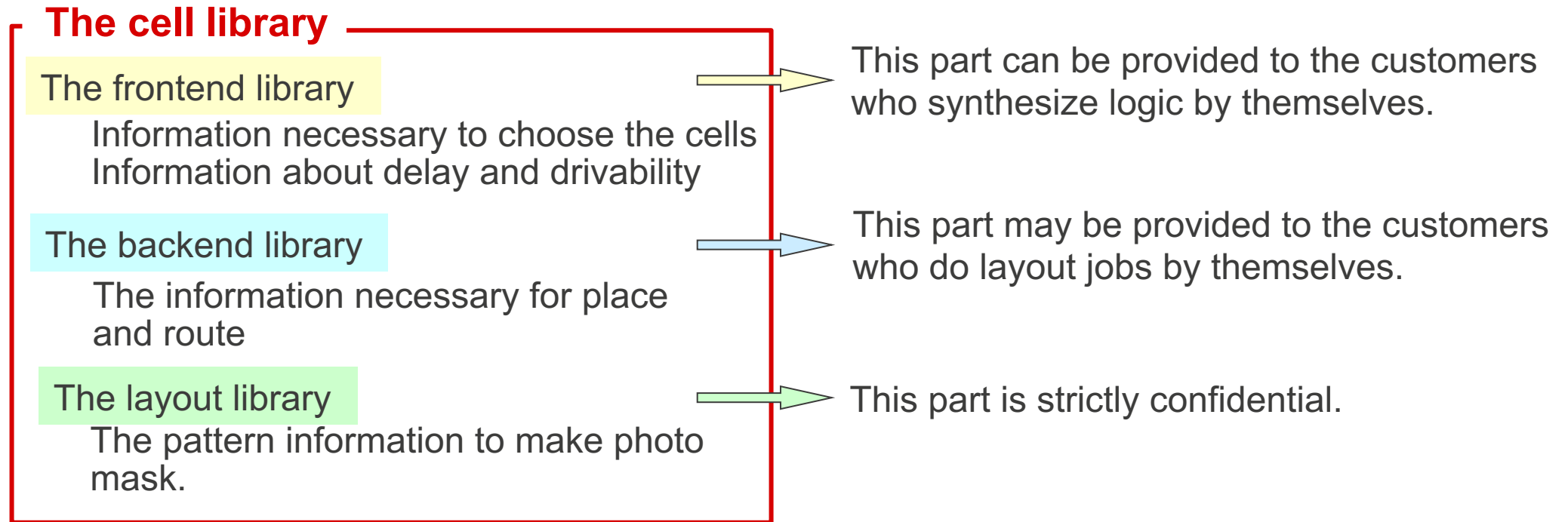
The backend library

The information necessary for place and route

The layout library

The pattern information to make photo mask.

Implement The Logic on Silicon (2/4)



The cell library contains a lot of information which is highly dependent on particular process generation and fabrication line.

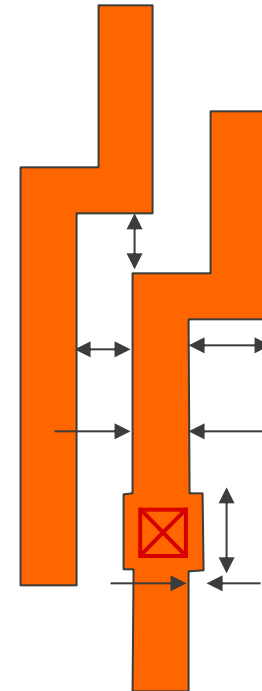
In particular, macro cells and custom cells have no compatibility between fabrication lines, therefore avoid using these cells whenever possible.

Implement The Logic on Silicon (3/4)

- 1.The information connection between cells
- 2.The feature information of the cell
(Function, input/output, the drive power, etc.)
- 3.The information on the structure of the cell
(Width, the height, the terminal position, etc.)
- 4.The information on the inner structure of the cell
Diffusion layer size/position, gate, inner wiring, etc.
- 5.Layout rules
(Minimum distance between cells, etc.)
- 6.Layout rules
(wiring rule, etc.)

The design rule

Minimum spacing, wiring width, and other conditions defined specific to a fabrication line.



Implement The Logic on Silicon (4/4)

1. The information connection between cells
2. The feature information of the cell
(Function, input/output, the drive power, etc.)
3. The information on the structure of the cell
(Width, the height, the terminal position, etc.)
4. The information on the inner structure of the cell
Diffusion layer size/position, gate, inner wiring, etc.
5. Layout rules
(Minimum distance between cells, etc.)
6. Layout rules
(wiring rule, etc.)

The Netlist

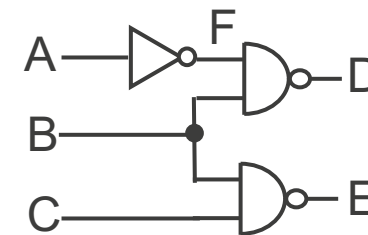
```
not01d1 INST1(.a1(A), .zn(F));  
nand02d1 INST2(.a1(F), .a2(B), .zn(D));  
nand02d1 INST3(.a1(C), .a2(B), .zn(E));
```

Name of the
cell used

Name of the
wire

Name of the
I/O of the cell

Connection



Gate Level Design

```
not01d1 INST1(.a1(A), .zn(F)) ;  
nand02d1 INST2(.a1(F), .a2(B), .zn(D)) ;  
nand02d1 INST3(.a1(C), .a2(B), .zn(E)) ;
```

Generating a total logic or modifying a part of logic by selecting/connecting primitive gates as the above is called “gate level design”.

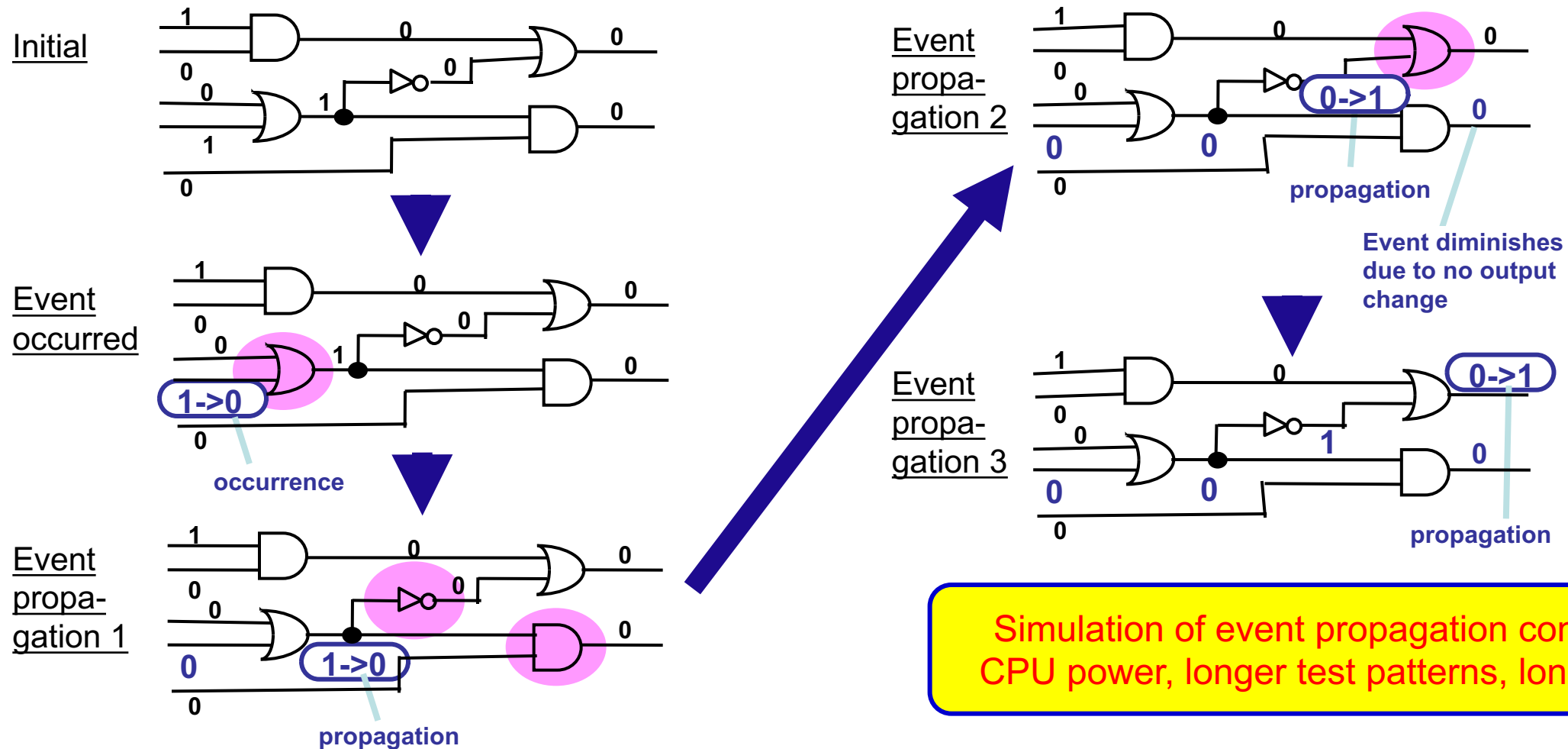
This situation may happen in Design for Testability (DFT), Static Timing Analysis (STS), Layout design.

Gate level design has some disadvantages as below:

- (1) Visibility of a code is very poor, hard to understand at a glance.
- (2) Portability of a design in net list is very bad because of the cells used in the design.
- (3) There are few compatibilities among the process generations.
- (4) It will be difficult to enjoy the benefit of the evolution of DA tools (such as optimization)

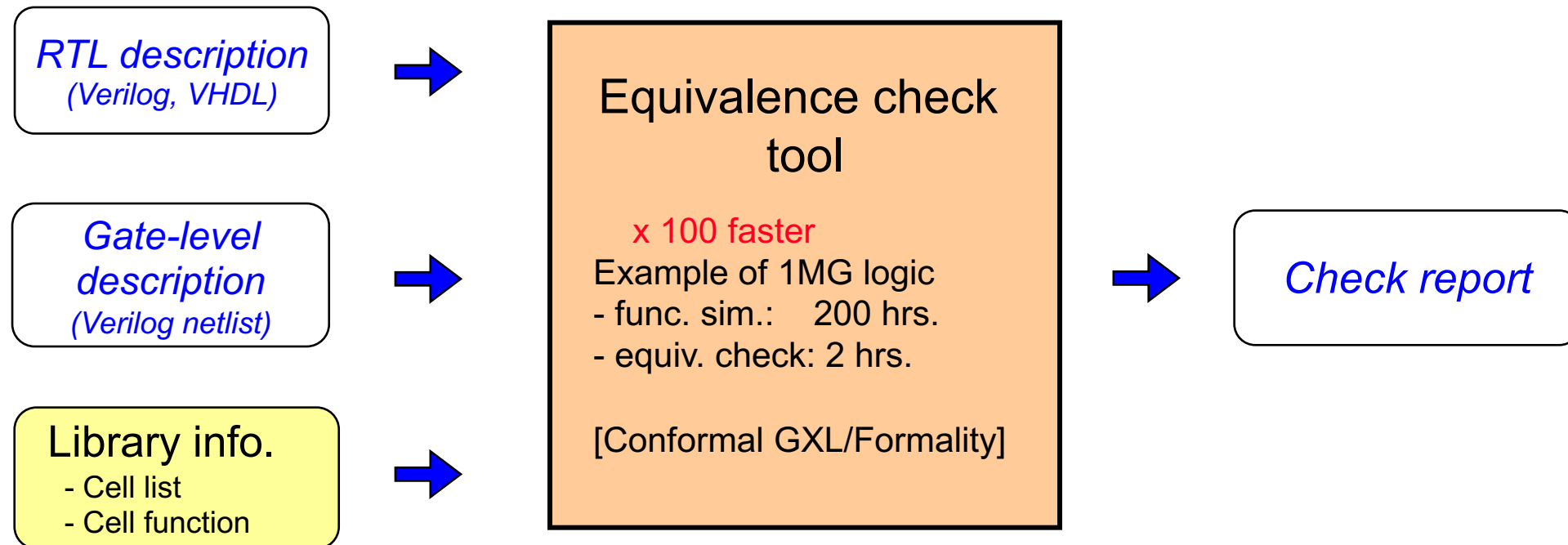
Gate Level Verification – Functional Simulation

Gate level simulation (or logic simulation): event driven method by test patterns



Gate Level Verification – Equivalence Check

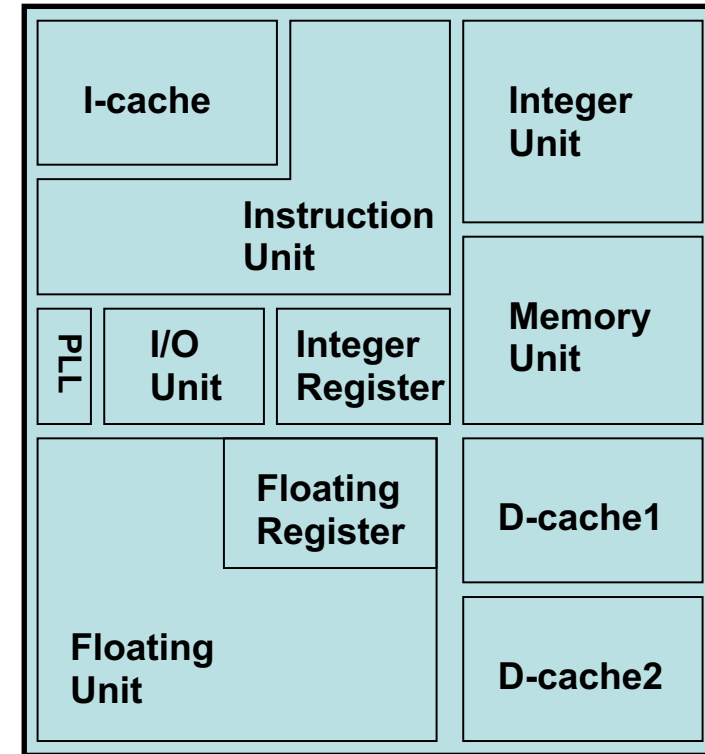
- Equivalence of two logic descriptions is mathematically verified by formality check: no need for test patterns.
- Equivalence check is exhaustive and faster than functional simulation.



Floor Planning

For pre-layout timing verification

- Place functional blocks by considering signal flow
- Consider package specifications, minimize pin count, and shorten wiring length
- Bonding pad layout must be well considered



Example for floor plan of processor chip

Timing Verification – Methods

Pre-layout timing verification

STA: Static Timing Analysis

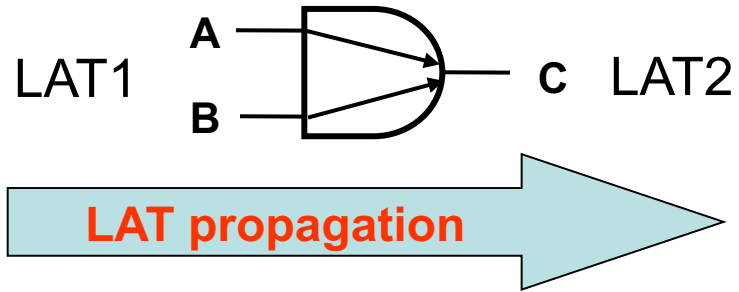
- Structure of gate-level logic description is analyzed and then compared with timing constraints
- Test patterns are not required
- Synchronous paths can be 100% verified
- Asynchronous logic and other special circuit cannot be analyzed
- Special paths must be specified manually

DTA: Dynamic Timing Analysis

- Logic simulation handling delay timing is executed
- Verification of all paths is unrealistic because huge test patterns are required

Verify synchronous paths with STA, and use DTA for limited paths that cannot be checked by STA. Always do synchronous design except for unavoidable case!

Timing Verification – STA Basic Algorithm



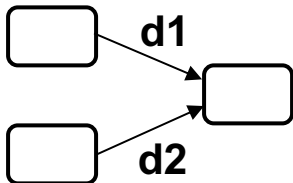
STA tool propagates LAT by selecting the maximum of LATs and adding delay of ARC to the LAT.

LAT: Latest Arrival Time

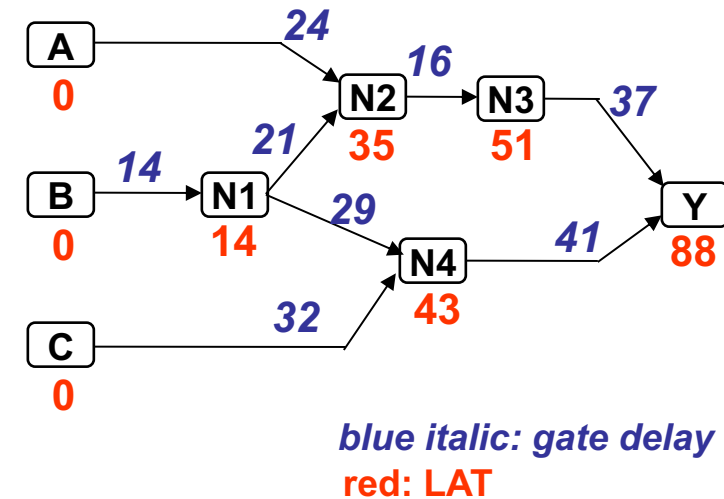
Sum : $d1 + d2$



Max : $\max(d1, d2)$



Example



Timing Verification – Design Process

- If the timing doesn't meet the requirement, replace cells and re-route wiring for better result.
- When it isn't possible to meet the timing requirement by just replacing and re-routing, then we have to go back to RTL code and have to rewrite the code.

[Renesas.com](https://www.renesas.com)