

Chapter 12: Synthesis

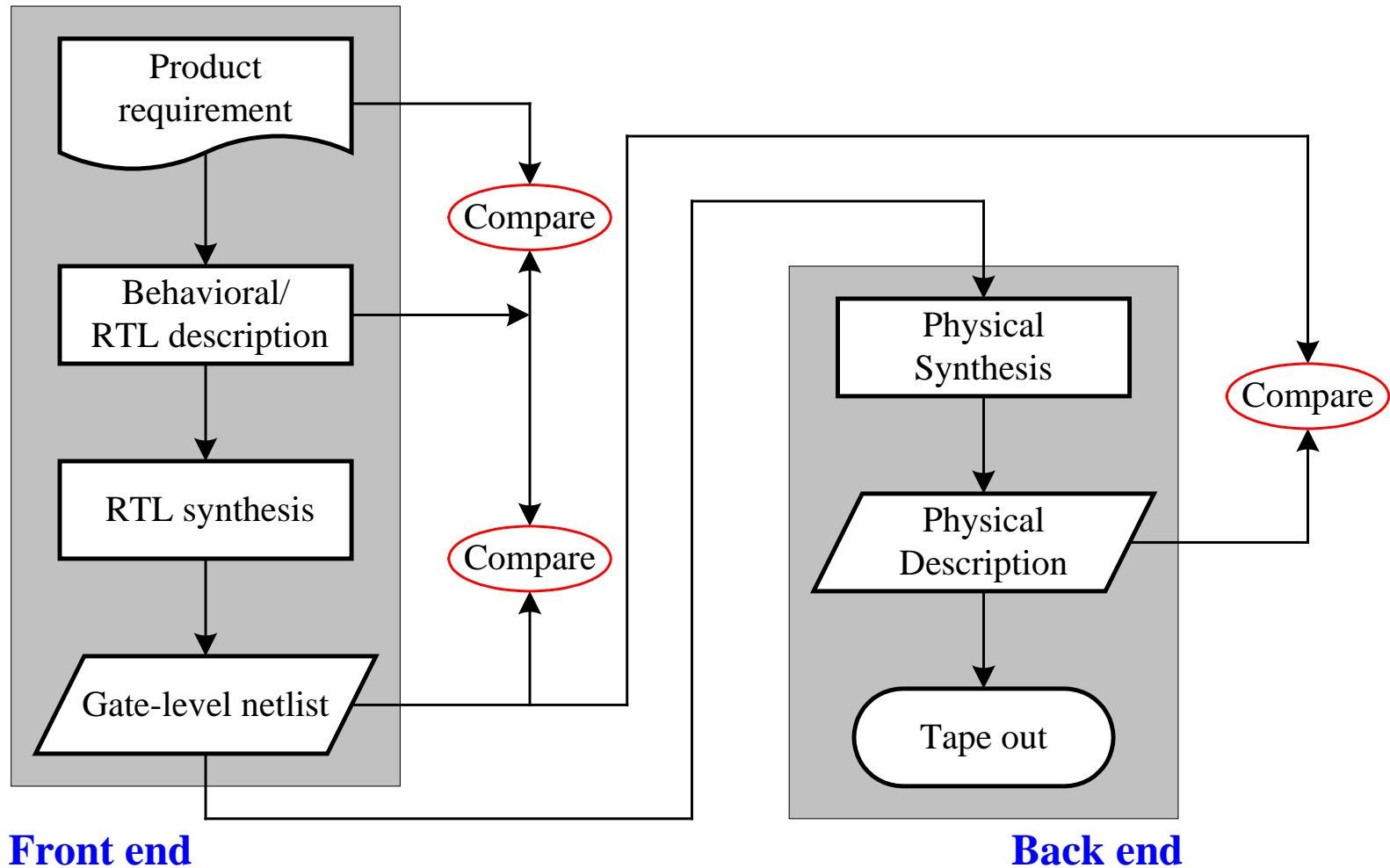
Prof. Soo-Ik Chae

Objectives

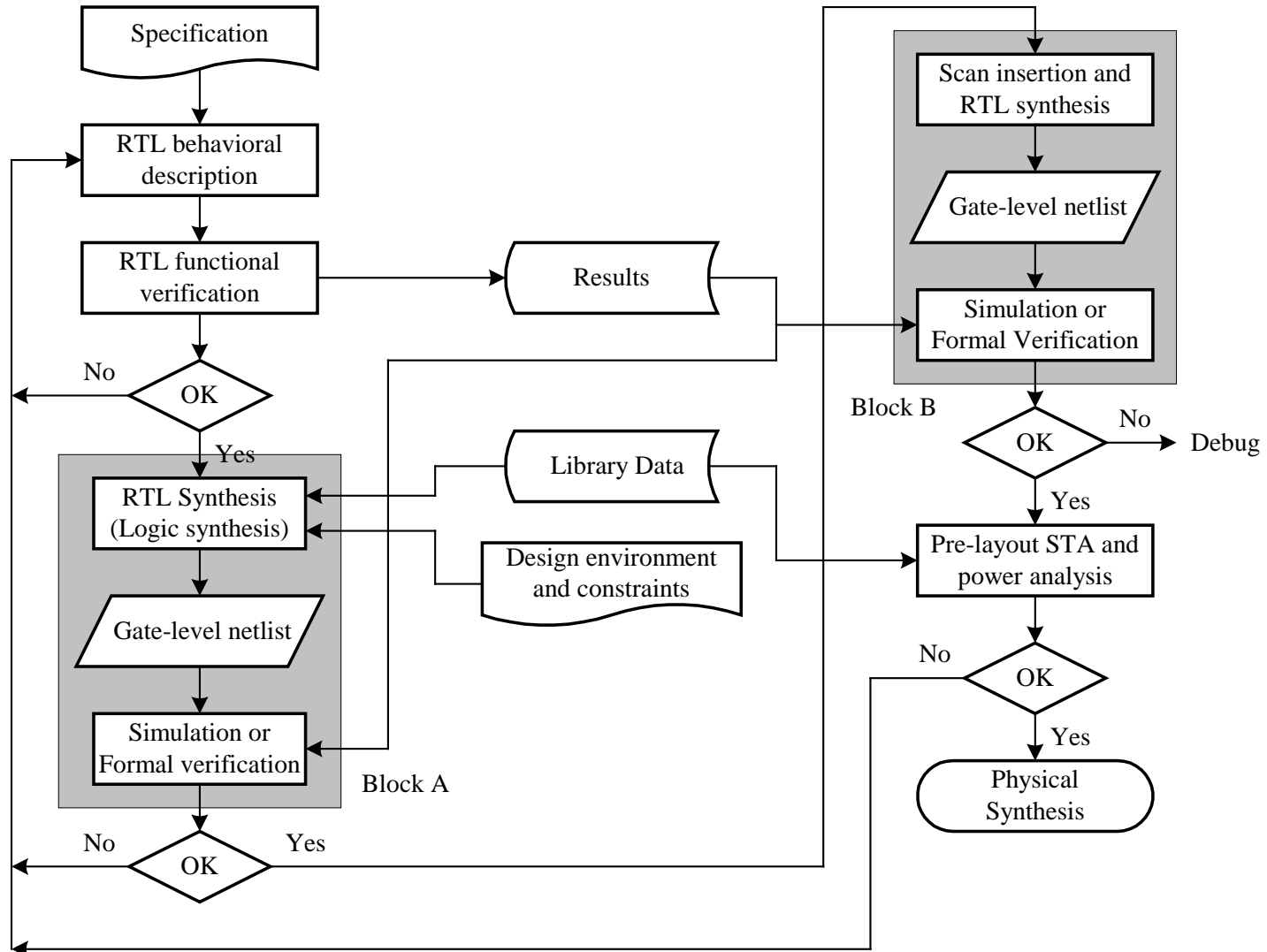
After completing this chapter, you will be able to:

- ❖ Describe ASIC/VLSI design flow
- ❖ Understand the RTL and physical synthesis flow
- ❖ Understand the principle of logic synthesis tools
- ❖ Understand issues of language translation
- ❖ Describe the considerations of clock signals
- ❖ Describe the considerations of reset signals
- ❖ Describe the partition issues for synthesis

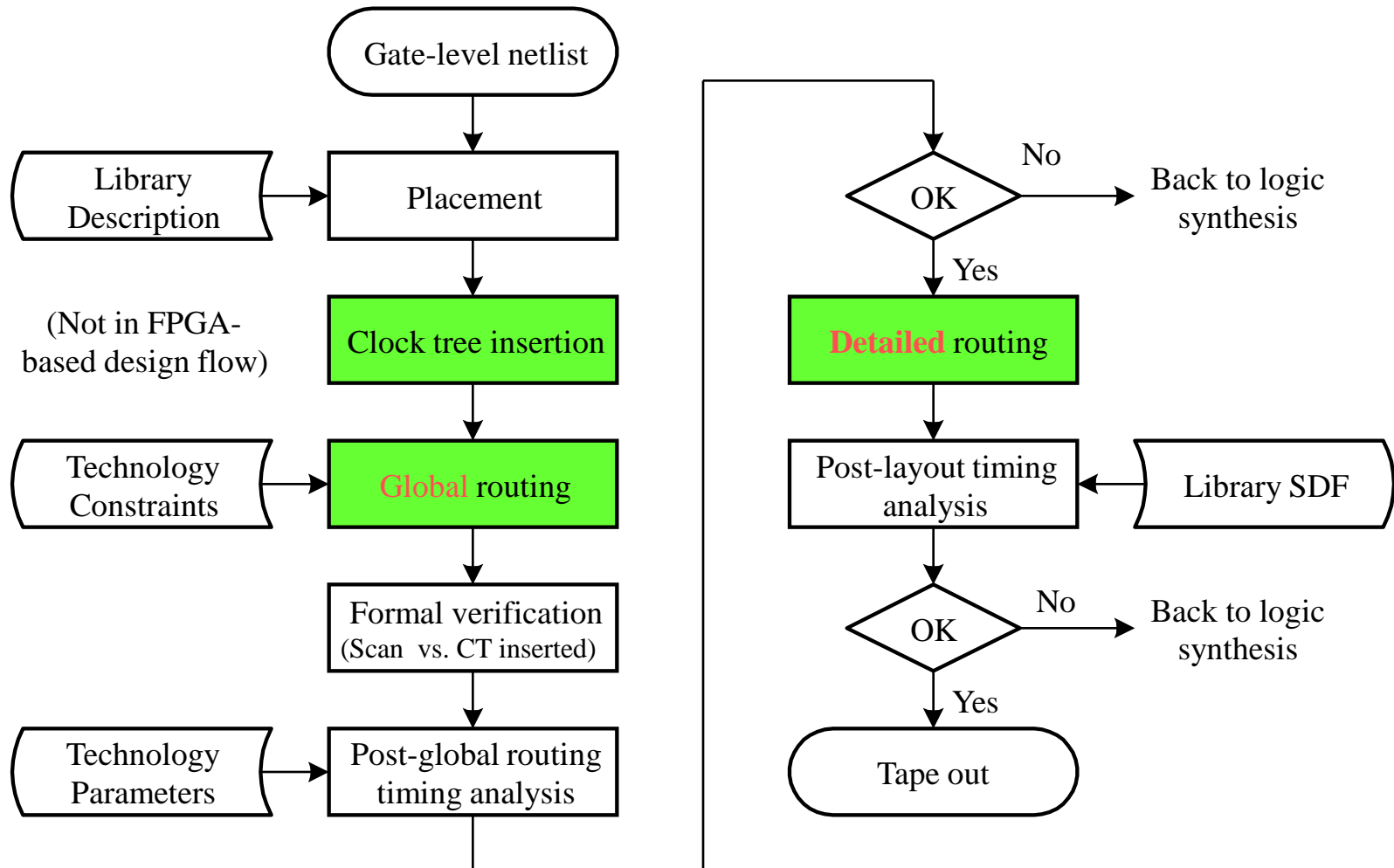
An ASIC/VLSI Design Flow



An RTL Synthesis Flow



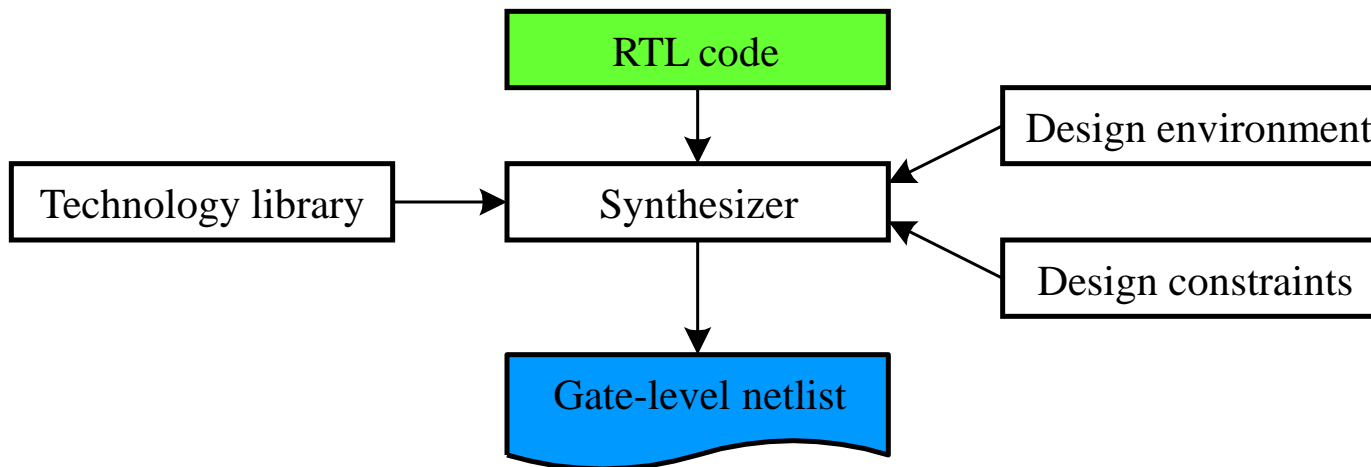
A Physical Synthesis Flow



Logic Synthesis Environment

❖ The following must be provided to synthesis tools:

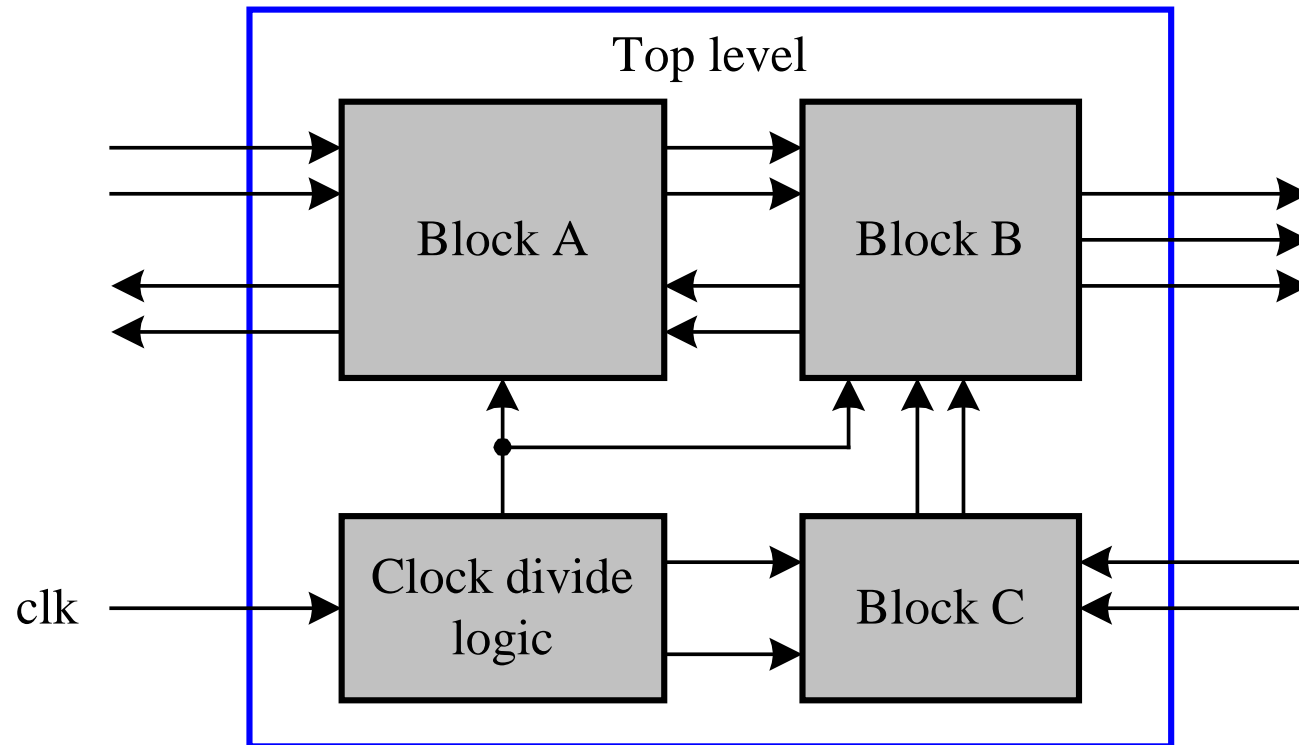
- design environment
- design constraints
- RTL code
- technology library



Design Environment

- ❖ Specify those directly influence design synthesis and optimization results
- ❖ The **external operating conditions** (PVT) include
 - manufacturing process
 - worst case: setup-time violations
 - best case: hold-time violations
 - operating conditions: voltage and temperature
- ❖ **I/O port attributes** contain
 - **drive** strength of input port
 - capacitive **loading** of output port
 - design rule constraints: fanin, fanout
- ❖ **Statistical wire-load model** provides a wire-load model for processing the pre-layout static timing analysis.

Design Environment



Design Constraints

❖ Clock signal specification

- period, duty cycle
- transition time, skew

❖ Delay specifications

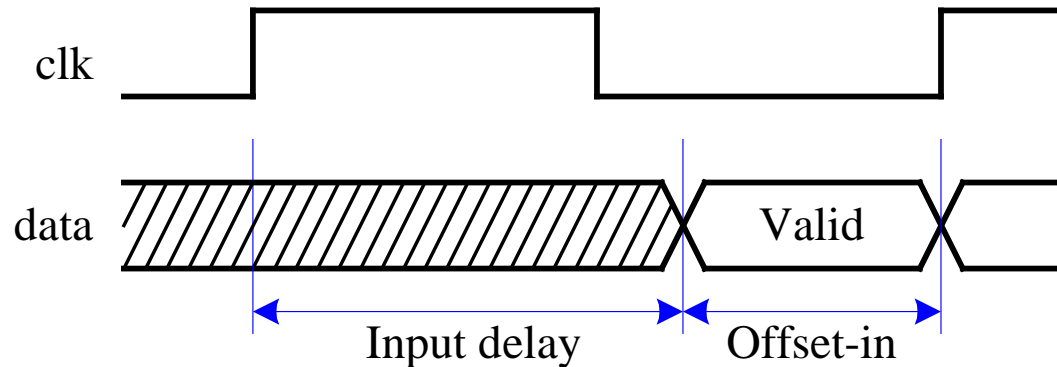
- input delay, output delay
- maximum, minimum delay for combinational circuits

❖ Timing exception

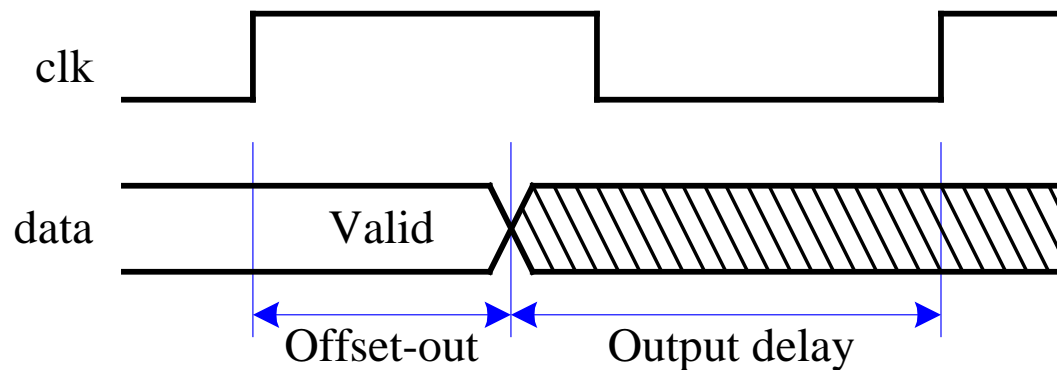
- **false path** : instruct the synthesis to ignore a particular path for timing optimization
- **multicycle path**: inform the synthesis tool regarding the number of clock cycles that a particular path requires to reach its endpoint

❖ Path grouping: bundle together critical paths in calculating a cost function

Input Delay and Output Delay

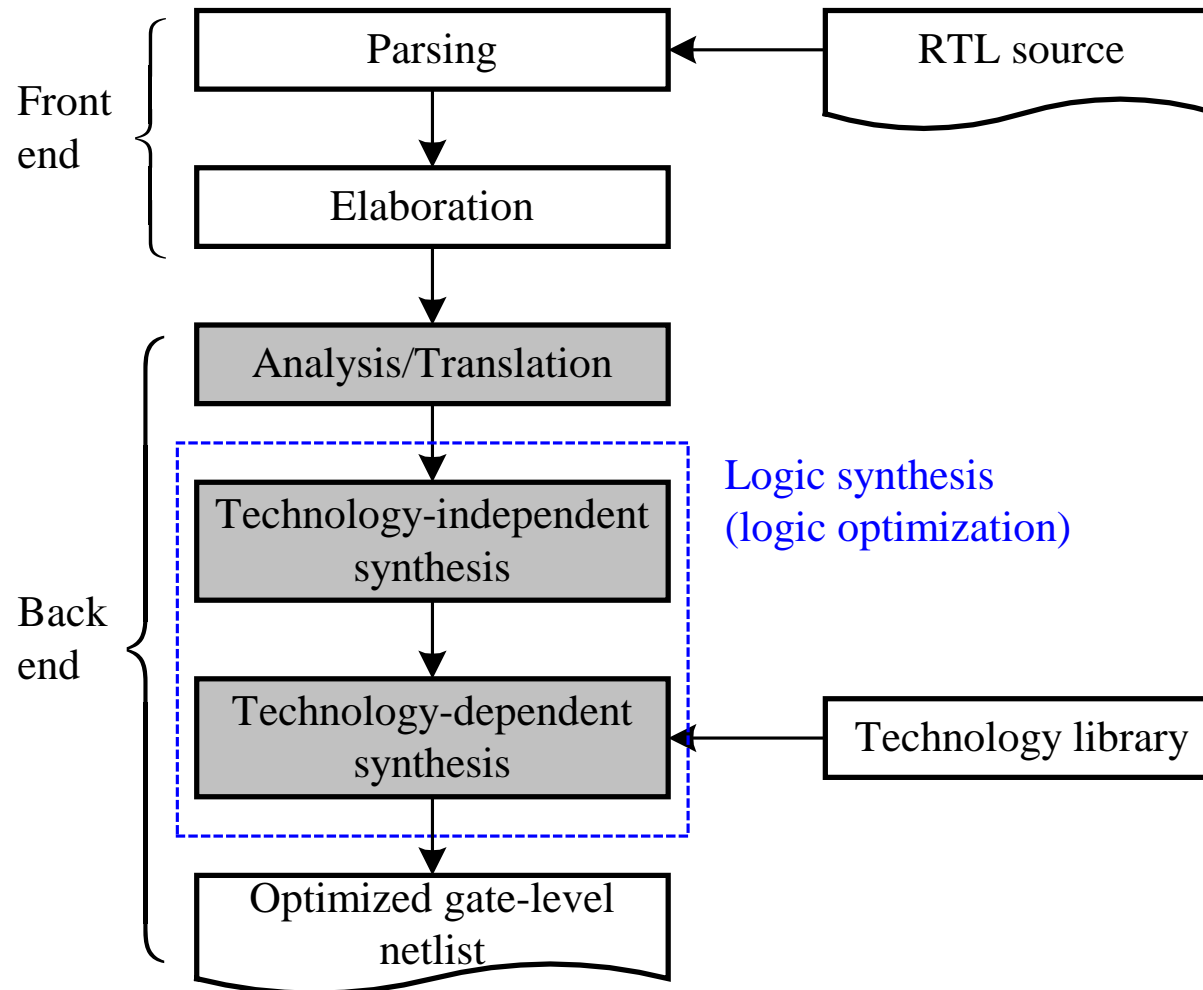


(a) The definition of input and offset-in delays



(b) The definition of offset-out and output delays

The Architecture of Logic Synthesis Tools



Logic Synthesis Tools: Front end

❖ Parsing phase

- checks the syntax of the source code
- creates internal components

❖ Elaboration phase (to construct a complete description of the input circuit)

- connects the internal components
- unrolls loops
- expands generate-loops
- sets up parameters passing for tasks and functions
- and so on

Logic Synthesis Tools: Back end

- ❖ **analysis/translation** prepares for technology-independent logic synthesis.
 - managing the design hierarchy
 - extracting finite-state machine (FSM)
 - exploring resource sharing
 - and so on.
- ❖ **logic synthesis** (**logic optimization**) creates a new gate network which computes the functions specified by a set of Boolean functions, one per primary output.
- ❖ **netlist generation** generates a gate-level netlist.

Logic Synthesis (Logic Optimization)

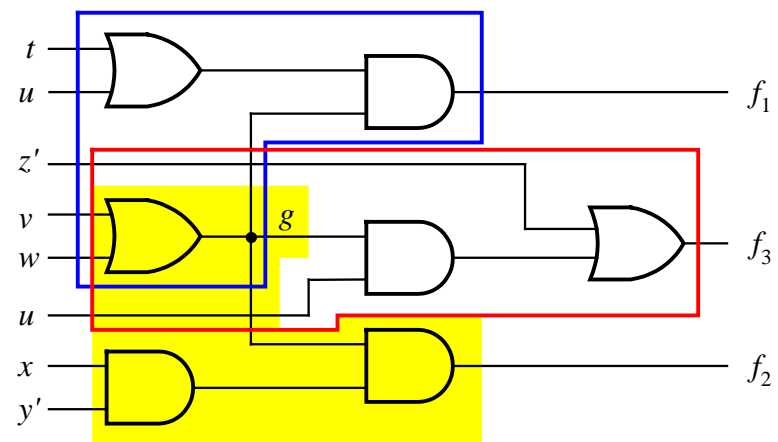
- ❖ Major concerns when synthesizing a logic gate network:
 - **functional metric**: such as fanin, fanout, and others.
 - **non-functional metric**: such as area, power, and delay.
- ❖ Two phases of logic synthesis:
 - technology-independent logic optimization
 - technology-dependent logic optimization
- ❖ The process of translating from technology-independent to the technology-dependent gate network is called **library binding**.

Technology-Independent Logic Optimization

❖ Technology-independent logic synthesis

- **Simplification** rewrites a single function in the network to the literals of that network.
- **Restructuring network** creates new function nodes that can be used as **common factors** and **collapses** sections of the network into a single node.
- **Restructuring delay** changes the factorization of a subnetwork to **reduce the number of function nodes** through which delay-critical signal must pass.

- The network is decomposed into nodes with no nodes more than k inputs, where k is determined by the fan-in of each LUT.
- The number of nodes is reduced by combining some of them taking into account the special features of LUTs.

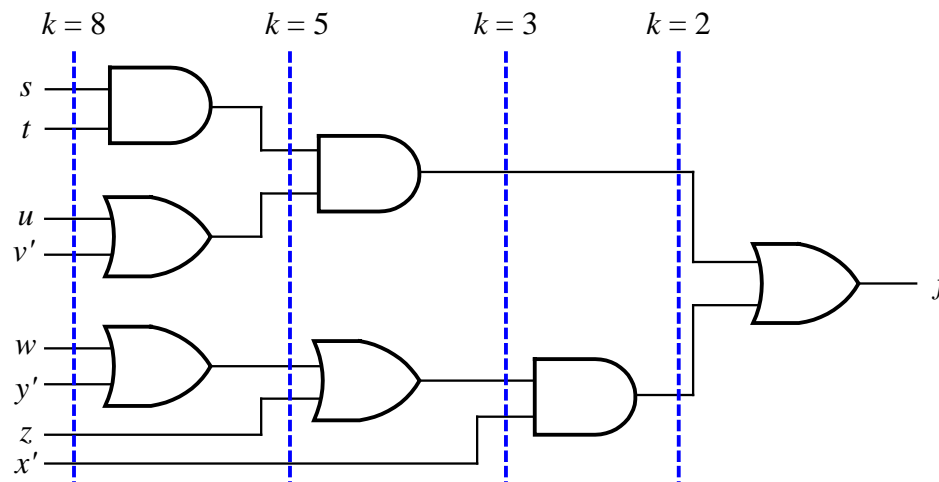


- only 3 LUTs are needed.

Technology Mapping

❖ FlowMap method

- Using a k -feasible cut algorithm breaks the network into LUT-sized blocks.
- Using heuristics to maximize the amount of logic fit into each cut to reduce the number of logic elements or LUTs required.



Three LUTs are required.

Synthesis-Tool Tasks

- ❖ Synthesis tools at least perform the following critical tasks:
 - Detect and eliminate redundant logic
 - Detect combinational feedback loops
 - Exploit don't-care conditions
 - Detect unused states
 - Detect and collapse equivalent states
 - Make state assignments
 - Synthesize optimal, multilevel logic subject to constraints.

The Key Point for Successful Logic Synthesis

Think hardware.

Language Structure Translations

- ❖ Language structure translation
 - Synthesizable operators
 - Synthesizable constructs
 - assignment statement
 - if .. else statement
 - case statement
 - loop structures
 - always statement
 - Memory synthesis approaches

Synthesizable Operators

Arithmetic	Bitwise	Reduction	Relational
+: add -: subtract *: multiply /: divide %: modulus **: exponent	~ : NOT & : AND : OR ^ : XOR ~^, ^~ : XNOR	& : AND : OR ~& : NAND ~ : NOR ^ : XOR ~^, ^~ : XNOR	>: greater than <: less than >=: greater than or equal <=: less than or equal
			Equality
			==: equality !=: inequality
		Logical	
	case equality	&&: AND	
	===: equality	: OR	
	!==: inequality	!: NOT	
			Miscellaneous
			{ , }: concatenation {const_expr{ }}: replication ? : : conditional

Synthesizing if-else Statements

❖ Features of if-else statement:

- The if-else statement infers a priority-encoded, cascaded combination of multiplexers.
- For combinational logic, we need to specify a complete if...else structure, otherwise, **a latch** will be inferred.
- For sequential logic, we need not specify a complete if ...else structure, otherwise, we will get as a notice **removing redundant expression** from synthesis tools.

```
always @(enable or data)
  if (enable) y = data; //infer a latch
```

```
always @(posedge clk)
  if (enable) y <= data;
  else y <= y; // a redundant expression
```

Synthesizing case Statements

- ❖ Features of case statement:
 - A case statement infers a multiplexer.
 - The consideration of using a complete or incomplete specified statement is the same as that of if...else statement.

Latch Inference --- Incomplete if-else Statements

// creating a latch example.

```
module latch_infer_if(enable, data, y);
```

```
input enable, data;
```

```
output y;
```

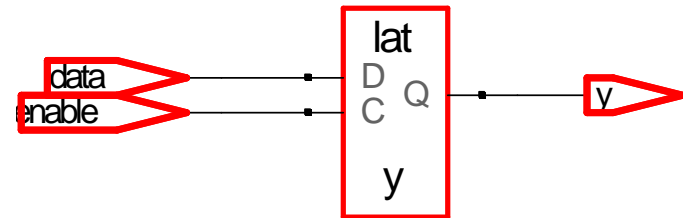
```
reg y;
```

// the body of testing program.

```
always @(enable or data)
```

```
    if (enable) y = data; //due to lack of else part, synthesizer infer a latch for y.
```

```
endmodule
```



Coding style:

- Avoid using any latches in your design.
- Assign outputs for all input conditions to avoid inferred latches.

For example:

```
always @(enable or data)
```

```
    y = 1'b0; // initialize y to its initial value.
```

```
    if (enable) y = data;
```


Latch Inference --- Incomplete case Statements

// Creating a latch example

```
module latch_infer_case(select, data, y);
```

```
input [1:0] select;
```

```
input [2:0] data;
```

```
output reg y;
```

// The body of 3-to-1 MUX

```
always @(select or data)
```

```
  case (select)
```

```
    2'b00: y = data[select];
```

```
    2'b01: y = data[select];
```

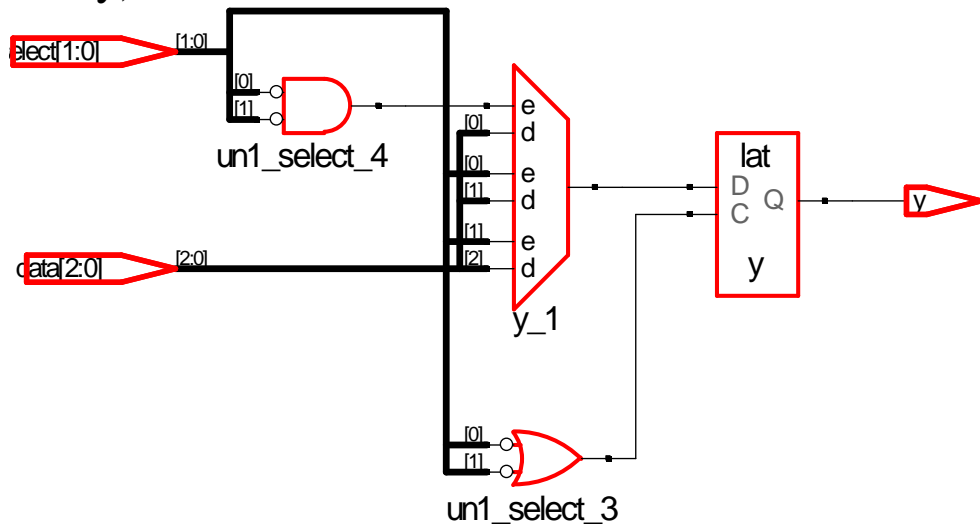
```
    2'b10: y = data[select];
```

// The following statement is used to avoid inferring a latch

```
//   default: y = 2'b11;
```

```
  endcase
```

```
endmodule
```



Ignored Delay Values --- An Incorrect Version

// a four phase clock example --- Generated **incorrect** hardware

```
module four_phase_clock_wrong(clk, phase_out);
```

```
input clk;
```

```
output reg [3:0] phase_out; // phase output
```

// the body of the four phase clock

```
always @(posedge clk) begin
```

```
    phase_out <= 4'b0000;
```

```
    phase_out <= #5 4'b0001;
```

```
    phase_out <= #10 4'b0010;
```

```
    phase_out <= #15 4'b0100;
```

```
    phase_out <= #20 4'b1000;
```

```
end
```

```
endmodule
```

1000 **phase_out[3:0]**

clk

Ignored Delay Values --- A Correct Version

// a four phase clock example --- synthesizable version

```
module four_phase_clock_correct(clk, phase_out);
```

```
input clk;
```

```
output reg [3:0] phase_out; // phase output
```

```
// the body of the four phase clock
```

```
always @(posedge clk)
```

```
case (phase_out)
```

```
4'b0000: phase_out <= #5 4'b0001;
```

```
4'b0001: phase_out <= #5 4'b0010;
```

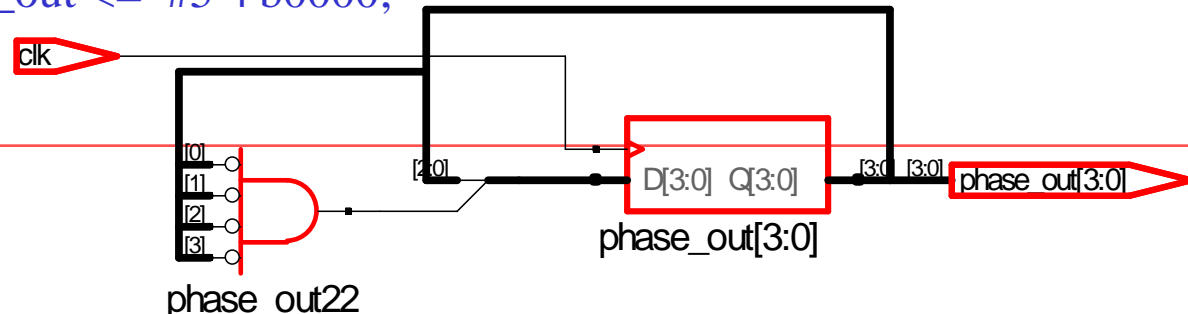
```
4'b0010: phase_out <= #5 4'b0100;
```

```
4'b0100: phase_out <= #5 4'b1000;
```

```
default: phase_out <= #5 4'b0000;
```

```
endcase
```

```
endmodule
```



Mixed Use of posedge/level Signals

```
// an example to illustrating the mixed usage of posedge/negedge signal.  
// The result cannot be synthesized. Try it in your system !!  
module DFF_bad (clk, reset, d, q);  
input  clk, reset, d;  
output reg q;  
// the body of DFF  
always @(posedge clk or reset)  
begin  
    if (reset) q <= 1'b0;  
    else      q <= d;  
end  
endmodule
```

Error: Can't mix posedge/negedge use with **plain** signal references.

Mixed Use of posedge/negedge Signals

// an example to illustrate the mixed usage of posedge/negedge signal.

// try it in your system !!

```
module DFF_good (clk, reset_n, d, q);
```

```
input  clk, reset_n, d;
```

```
output reg q;
```

// the body of DFF

```
always @(posedge clk or negedge reset_n)
```

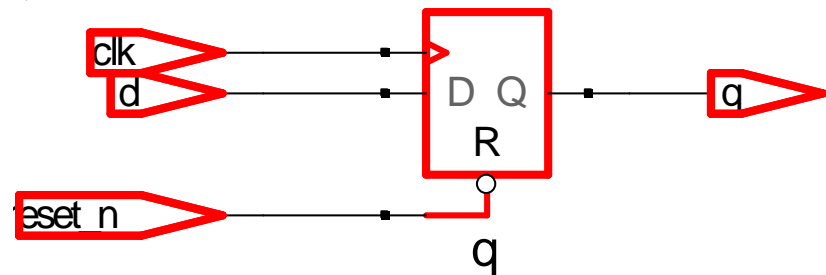
```
begin
```

```
    if (!reset_n) q <= 1'b0;
```

```
    else          q <= d;
```

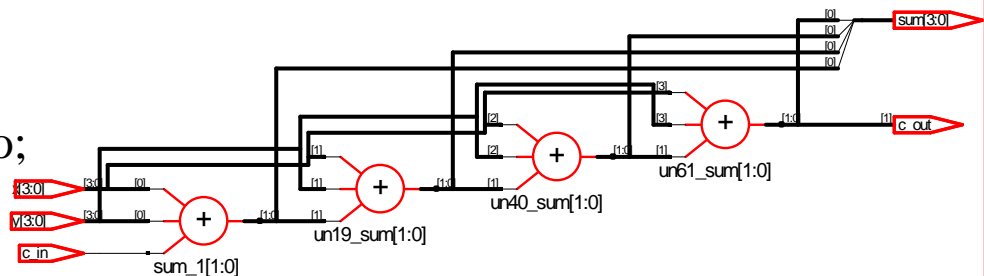
```
end
```

```
endmodule
```



Loop Structures

```
// an N-bit adder using for loop.
module nbit_adder_for( x, y, c_in, sum, c_out);
parameter N = 4;      // define default size
input  [N-1:0] x, y;
input  c_in;
output reg [N-1:0] sum;
output reg c_out;
reg    co;
integer i;
// specify the function of an n-bit adder using for loop.
always @(x or y or c_in) begin
    co = c_in;
    for (i = 0; i < N; i = i + 1)
        {co, sum[i]} = x[i] + y[i] + co;
    c_out = co; end
endmodule
```



Loop Structures --- An Incorrectly Synthesizable Example

// a multiple cycle example --- This is an **incorrect** version.

// Please try to correct it!

```
module multiple_cycle_example_a(clk, reset_n, data_a, data_b, total);
```

```
parameter N = 8;
```

```
parameter M = 4;
```

```
input  clk, reset_n;
```

```
input  [M-1:0] data_a;
```

```
input  [N-1:0] data_b;
```

```
output [N-1:0] total;
```

```
reg    [N-1:0] total;
```

```
integer i;
```

// what does the following statement do?

```
always @(posedge clk or negedge reset_n)begin
```

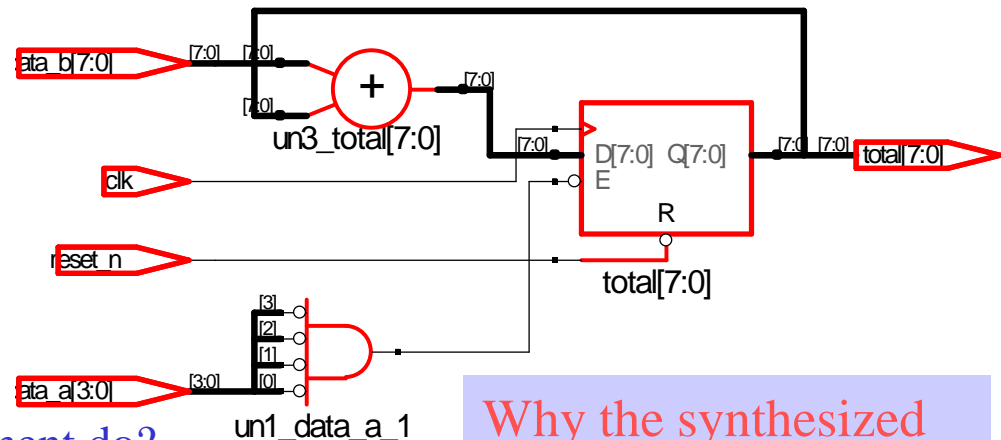
```
    if (!reset_n) total <= 0;
```

```
    else for (i = 0; i < M; i = i + 1)
```

```
        if (data_a[i] == 1) total <= total + data_b;
```

```
end
```

```
endmodule
```



Why the synthesized result is like this?
Try to explain it!

Memory Synthesis Approaches

- ❖ Random logic using flip-flops or latches
 - is independent of any software and type of ASIC.
 - is independent of easy to use but inefficient in terms of area.
- ❖ Register files in datapaths
 - use a synthesis directive or hand instantiation.
- ❖ RAM standard components
 - are supplied by an ASIC vendor.
 - depend on the technology.
- ❖ RAM compilers
 - are the most area-efficient approach.

A flip-flop may take up 10 to 20 times the area of a 6-transistor static RAM cell.

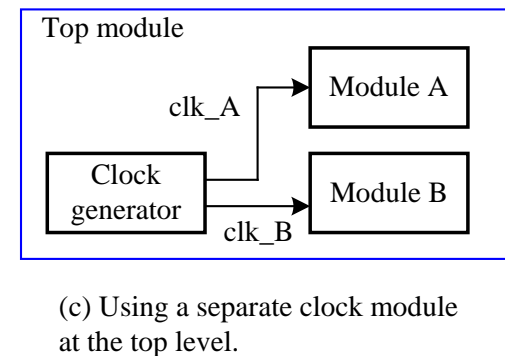
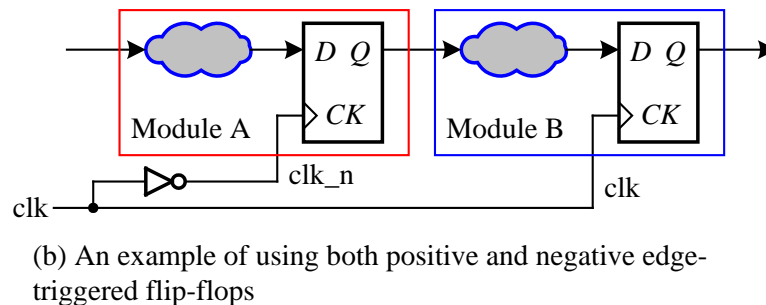
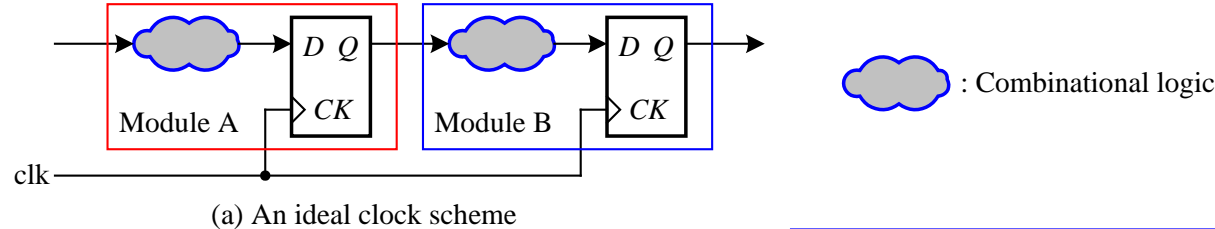
Coding Guidelines for Synthesis

❖ Goals of coding guidelines:

- Testability
- Performance
- Simplification of static timing analysis
- Gate-level behavior that matches that of the original RTL codes.

Guidelines for Clocks

- ❖ Using single global clock
- ❖ Avoiding using gated clocks
- ❖ Avoiding mixed use of both positive and negative edge-triggered flip-flops
- ❖ Avoiding using internally generated clock signals



Guidelines for Resets

- ❖ The basic design issues of resets are:
 - Asynchronous or synchronous?
 - An internal or external power-on reset?
 - More than one reset, hard vs. soft reset?
- ❖ The basic writing styles for both asynchronous and synchronous reset are as follows:

```
always @(posedge clk or posedge reset)
    if (reset) .....
    else .....
```

Asynchronous reset

```
always @(posedge clk)
    if (reset) .....
    else .....
```

Synchronous reset

- ❖ The only logic function for the reset signal should be a direct clear of all flip-flops.

Guidelines for Resets

❖ Asynchronous reset

- is harder to implement.
 - since reset is a special signal like clock, it requires a tree of buffers to be inserted at place and route.
- does not require a free-running clock.
- does not affect flip flop data timing.
- makes static timing analysis (or cycle-based simulation) more difficult.
- makes the automatic insertion of test structure more difficult.

❖ Synchronous reset

- is easy to implement.
 - It is just another synchronous signal to the input.
- requires a free-running clock
 - in particular, at power-up, for reset to occur.

Guidelines for Resets

- ❖ Avoid internally generated conditional resets.

```
always @(posedge gate or negedge reset_n or posedge timer_load_clear)
    if (!reset_n || timer_load_clear) timer_load <= 1'b0;
    else timer_load <= 1'b1;
```

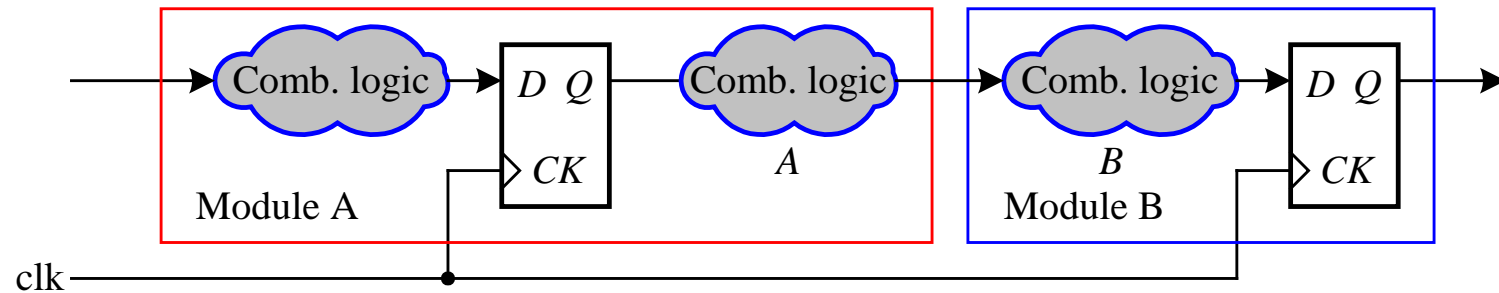
- ❖ When a conditional reset is required:

- to create a separate signal for the reset signal.
- to isolate the conditional reset logic in a separate logic block.

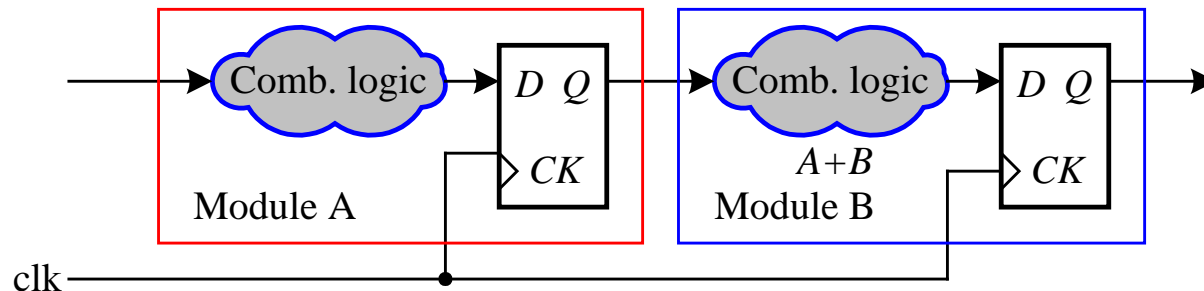
```
assign timer_load_reset = !reset_n || timer_load_clear;
always @(posedge gate or posedge timer_load_reset)
    if (timer_load_reset) timer_load <= 1'b0;
    else timer_load <= 1'b1;
```

Partitioning for Synthesis

❖ Keep related logic within the same module.



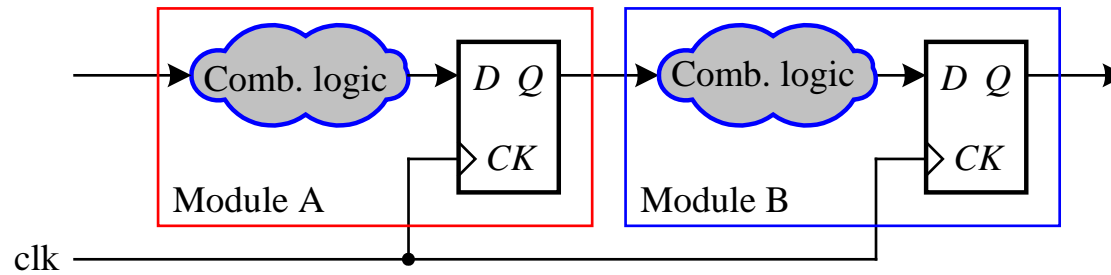
(a) Bad style



(b) Good style

Partitioning for Synthesis

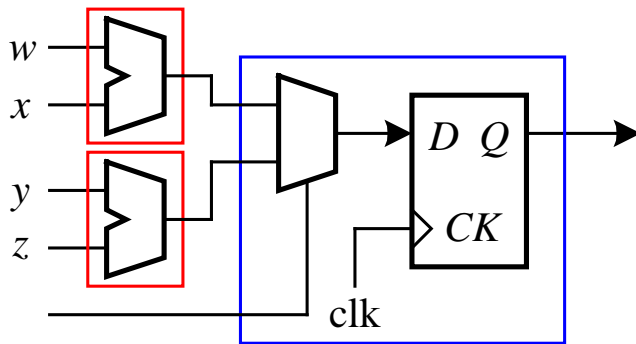
❖ Register all outputs.



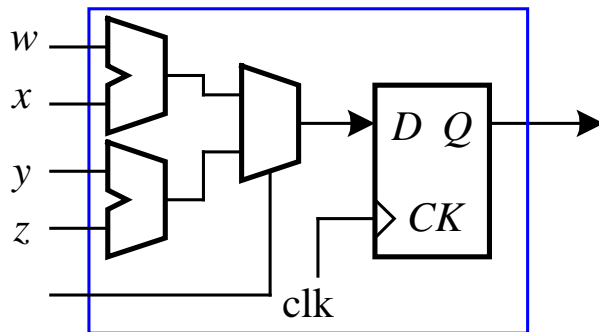
❖ Separating structural logic from random logic.

Partitioning for Synthesis

❖ Synthesis tools tend to maintain the original hierarchy.



(a) Resources in different modules cannot be shared.



(b) Resources in the same module can be shared.

