HARDWARE DESCRIPTION LANGUAGE FOR DIGITAL DESIGN

數位設計硬體描述語言

Synthesis

Materials partly adapted from "Digital System Designs and Practices Using Verilog HDL and FPGAs," M.B. Lin.

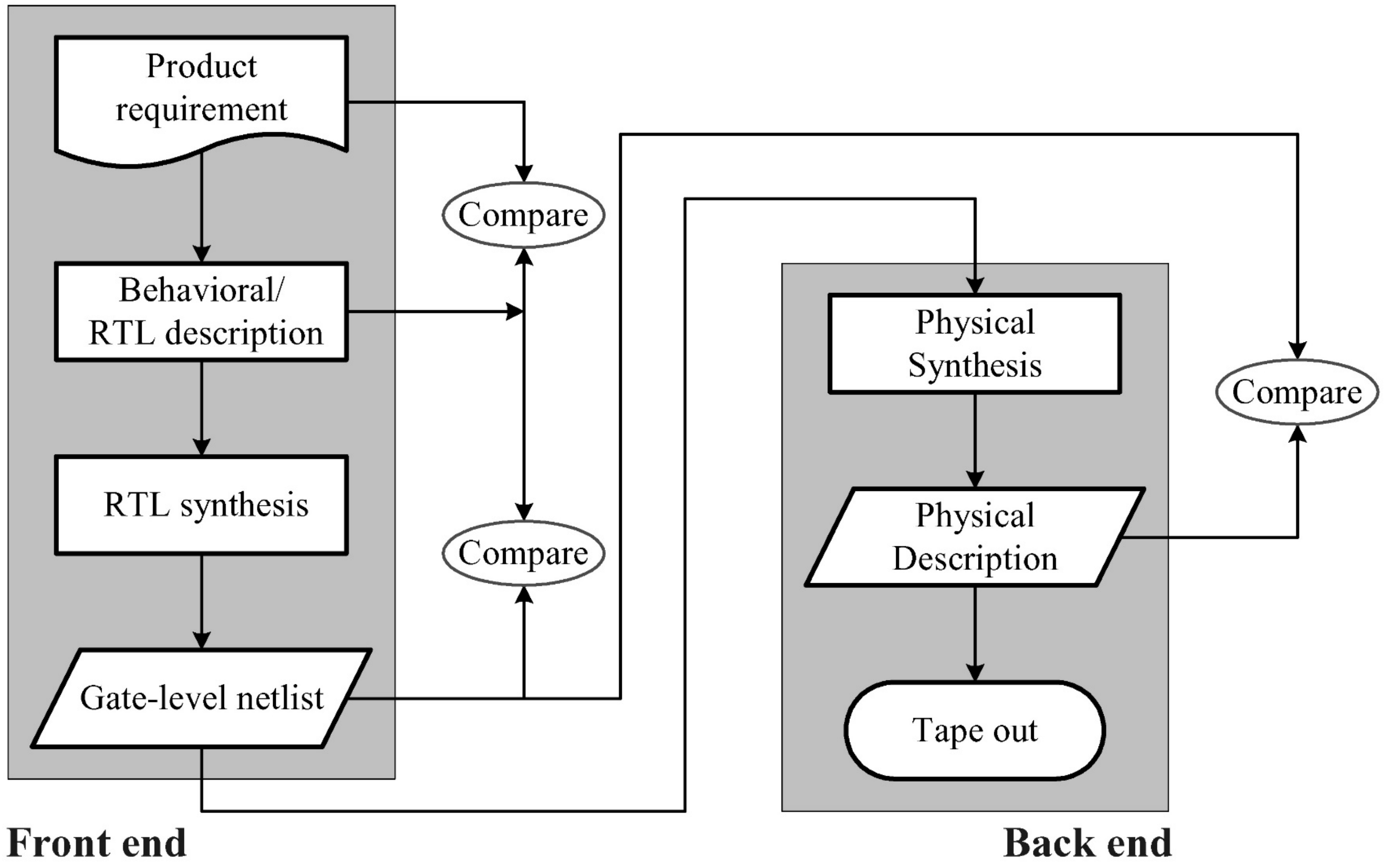
NCKU EE
LPHP Lab

OUTLINE

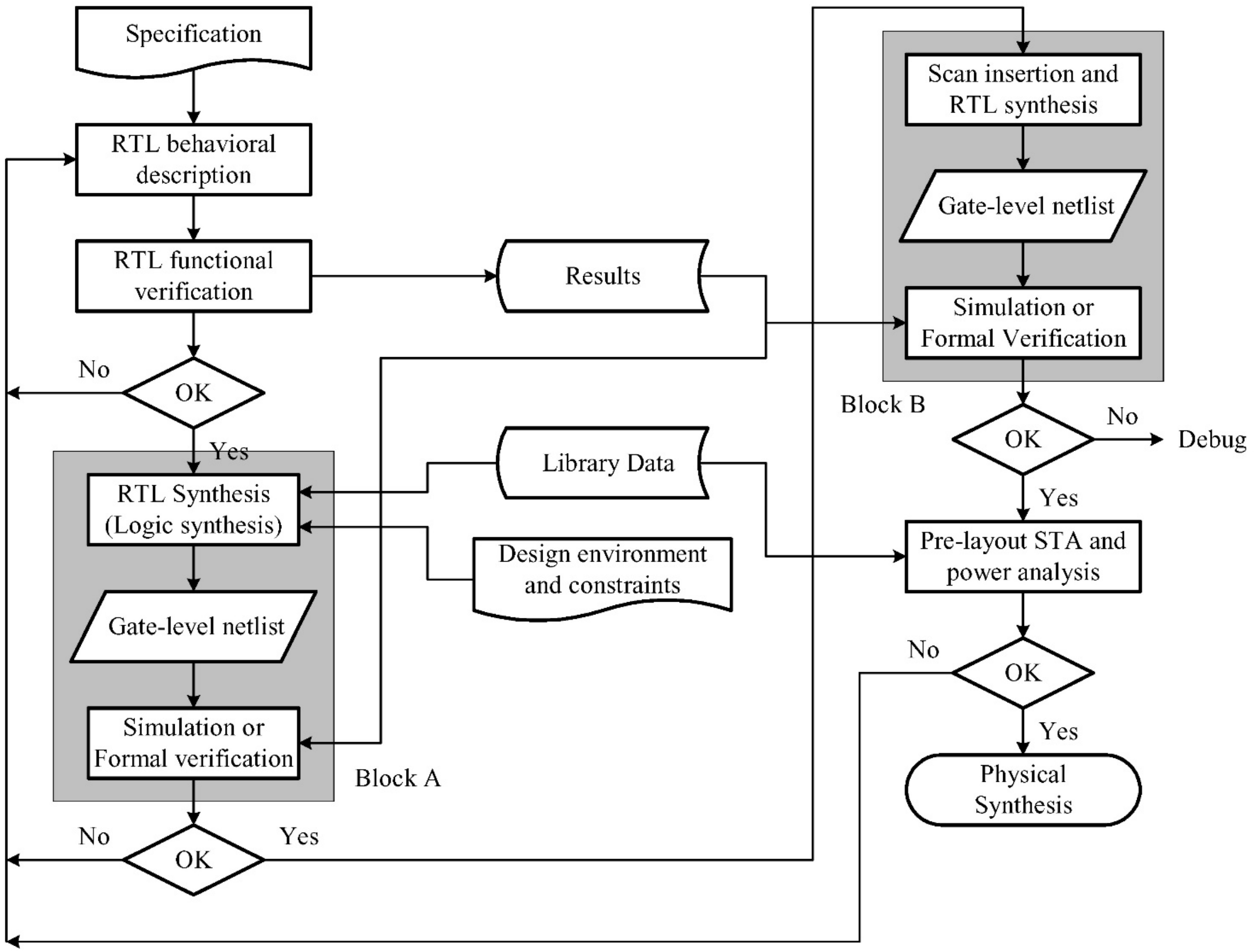
- Design flows
- Design environment and constraints
- Logic synthesis
- Language structure synthesis
- Coding guidelines

DESIGN FLOW

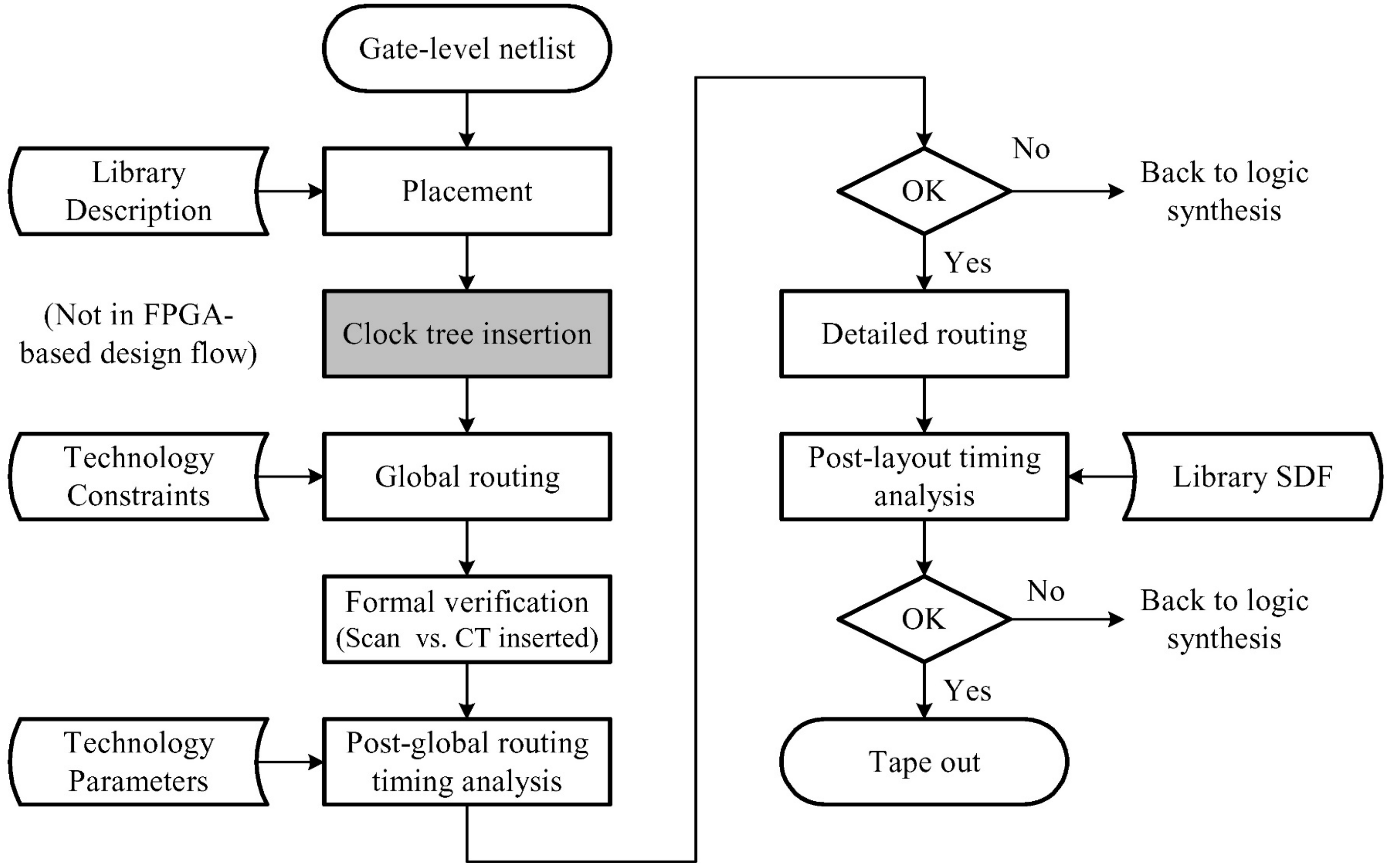
AN ASIC/VLSI DESIGN FLOW



AN RTL SYNTHESIS FLOW


Specification
RTL behavioral description
RTL functional verification
Results
No
OK
Yes
RTL Synthesis (Logic synthesis)
Library Data
Design environment and constraints
Gate-level netlist
Simulation or Formal verification
Block A
No
OK
Yes
Scan insertion and RTL synthesis
Gate-level netlist
Simulation or Formal Verification
Block B
OK
No
Debug
Yes
Pre-layout STA and power analysis
No
OK
Yes
Physical Synthesis

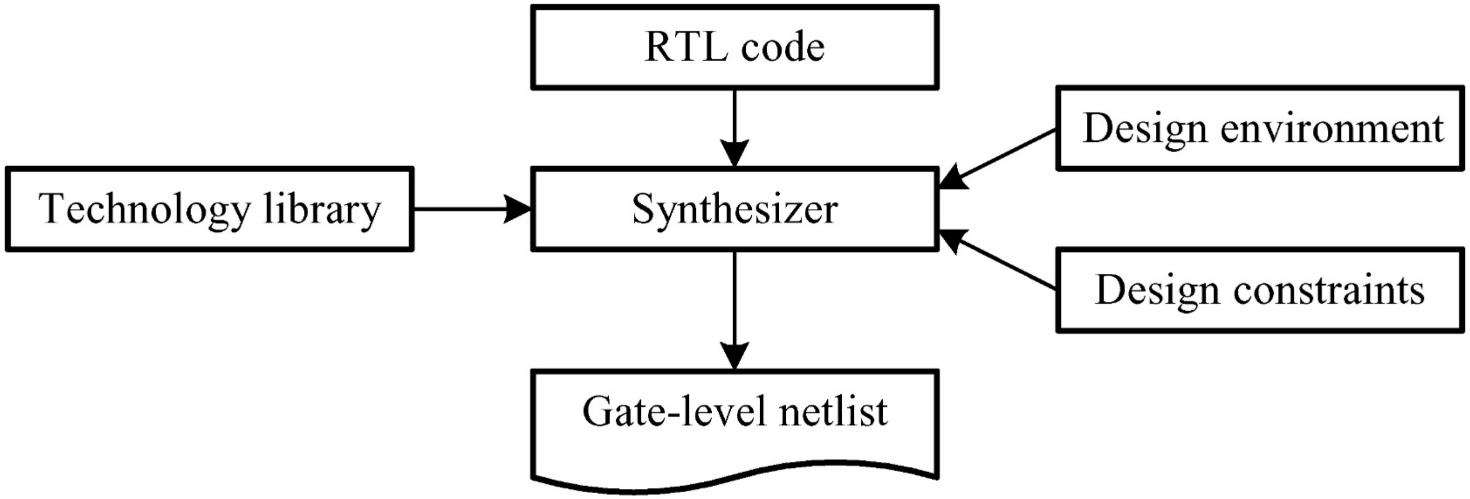
A PHYSICAL SYNTHESIS FLOW



DESIGN ENVIRONMENT AND CONSTRAINTS

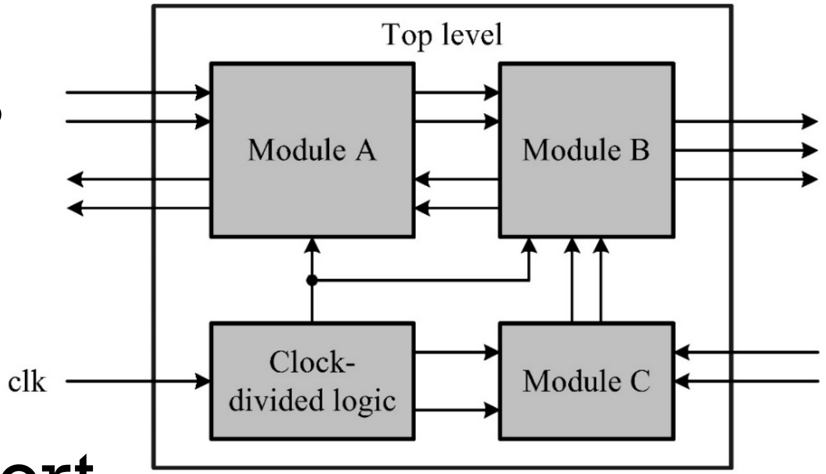
LOGIC SYNTHESIS ENVIRONMENT

- Design environment
- Design constraints
- RTL code
- Technology library



DESIGN ENVIRONMENT

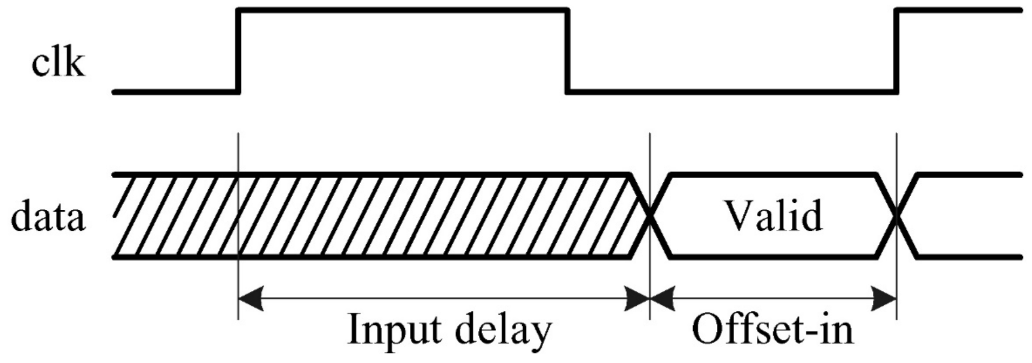
- The process parameters
  - technology library
  - operating conditions
- I/O port attributes
  - drive strength of input port
  - capacitive loading of output port
  - design rule constraints
- Statistical wire-load model
  - pre-layout static timing analysis



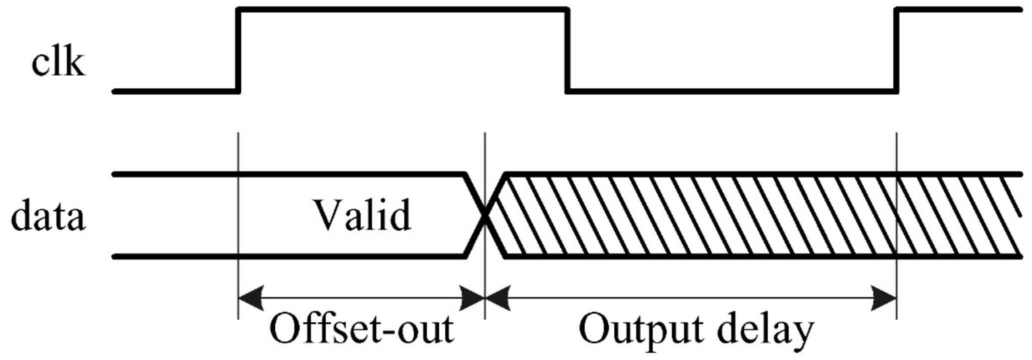
DESIGN CONSTRAINTS

- Clock signal specification
  - period
  - duty cycle
  - transition time
  - skew
- Delay specifications
  - maximum
  - minimum
- Timing exception
  - false path
  - multicycle path
- Path grouping

INPUT DELAY AND OUTPUT DELAY



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

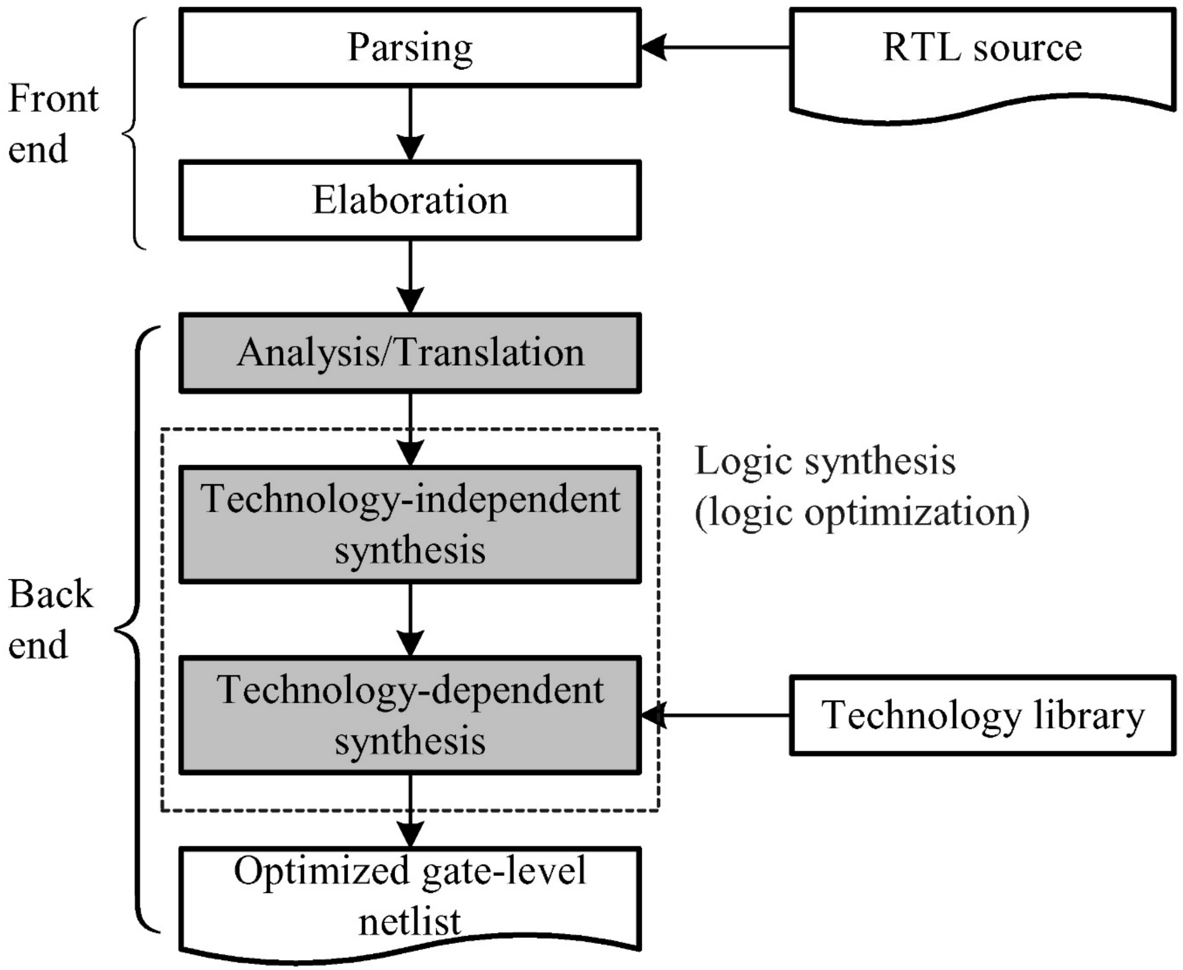


(b) The definition of offset-out and output delays

SYLLABUS

- Objectives
- Design flows
- Design environment and constraints
- Logic synthesis
  - Architecture of synthesizers
  - Technology-independent logic synthesis
  - Technology-dependent logic synthesis
- Language structure synthesis
- Coding guidelines

THE ARCHITECTURE OF SYNTHESIZER



Parsing
RTL source
Front end
Elaboration
Analysis/Translation
Technology-independent synthesis
Logic synthesis (logic optimization)
Back end
Technology-dependent synthesis
Technology library
Optimized gate-level netlist

THE ARCHITECTURE OF SYNTHESIZER

- Front end
  - Parsing phase
  - Elaboration phase
- Back end
  - analysis/translation
  - logic synthesis (logic optimization)
  - netlist generation

LOGIC SYNTHESIS

LOGIC SYNTHESIS (LOGIC OPTIMIZATION)

- Major concerns
  - functional metric: fanin, fanout, and others
  - non-functional metric: area, power, and delay
- Two phases of logic synthesis
  - technology-independent
  - technology-dependent
- Library binding

TECHNOLOGY-INDEPENDENT LOGIC OPTIMIZATION

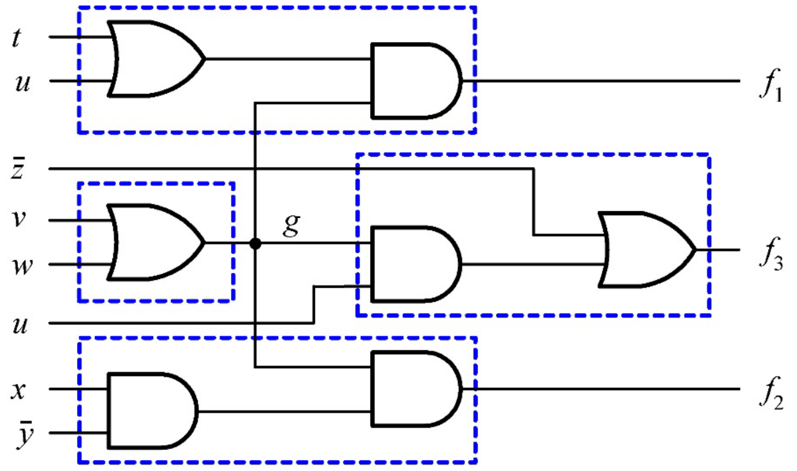
- Technology-independent logic synthesis
  - Simplification
  - Restructuring network
  - Restructuring delay

TECHNOLOGY MAPPING

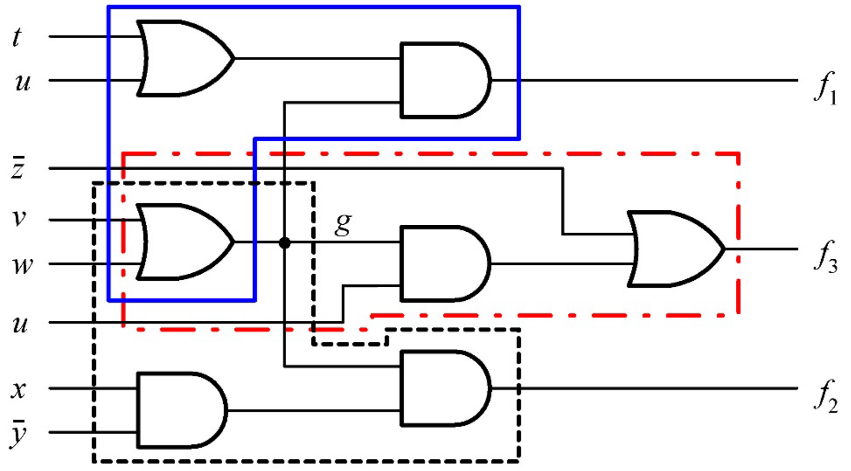
- A two-step approach
- FlowMap method

A TWO-STEP APPROACH

- Decompose the network
- Reduce the number of nodes



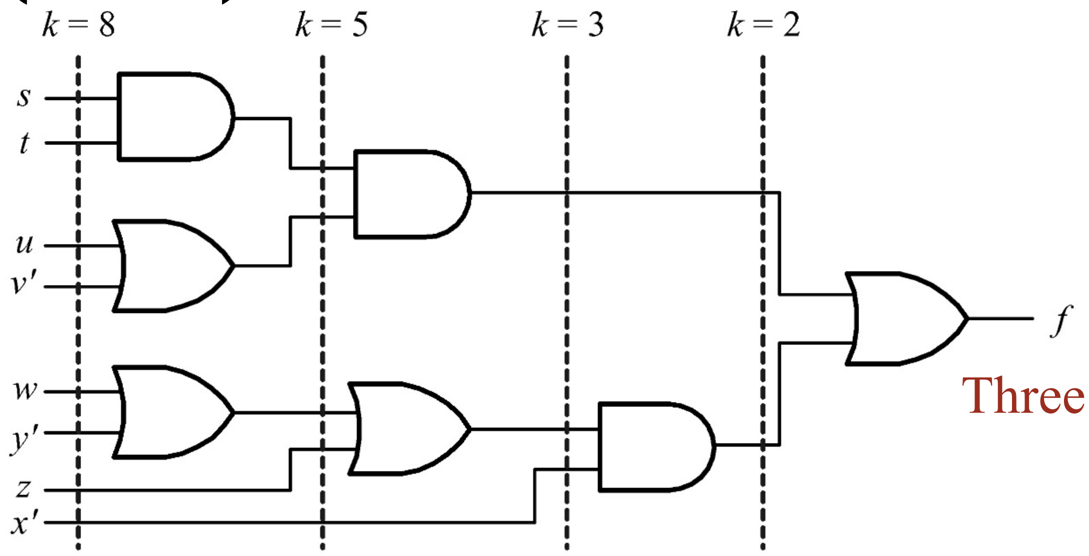
(a) Mapping I requires four LUTs.



(b) Mapping II only requires three LUTs.

FLOWMAP METHOD

- Break the network into LUT-sized blocks
- Reduce the number of logic elements (LUTs)



Three LUTs are required

LANGUAGE STRUCTURE SYNTHESIS

SYNTHESIS-TOOL TASKS

- 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 in a hardware mind

LANGUAGE STRUCTURE TRANSLATIONS

- 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	~ : NOT	&: AND	>: greater than
- : subtract	&: AND	|: OR	<: less than
* : multiply	| : OR	~&: NAND	>= : greater than or equal
/ : divide	^: XOR	~|: NOR	<=: less than or equal
% : modulus	~^, ^~: XNOR	^: XOR	**Equality**
**: exponent		~^, ^~: XNOR	==: equality
Shift		**Logical**	!=: inequality
<< : left shift	**case equality**	&&: AND	**Miscellaneous**
>> : right shift	===: equality	|| : OR	{ , }: concatenation
<<< : arithmetic left shift	!==: inequality	! : NOT	{const_expr{ }}: replication
>>>: arithmetic right shift			? : : conditional

SYNTHESIZING IF-ELSE STATEMENTS

- For combinational logic
  - Completely specified?
- For sequential logic
  - Completely specified?

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

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

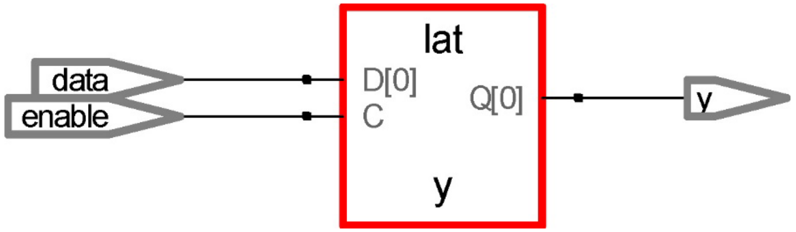
SYNTHESIZING CASE STATEMENTS

- A case statement
  - Infers a multiplexer
  - Completely specified?

LATCH INFERENCE --- INCOMPLETE IF-ELSE STATEMENTS

```
// creating a latch
module latch_infer_if(enable, data, y);
…
reg    y;

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



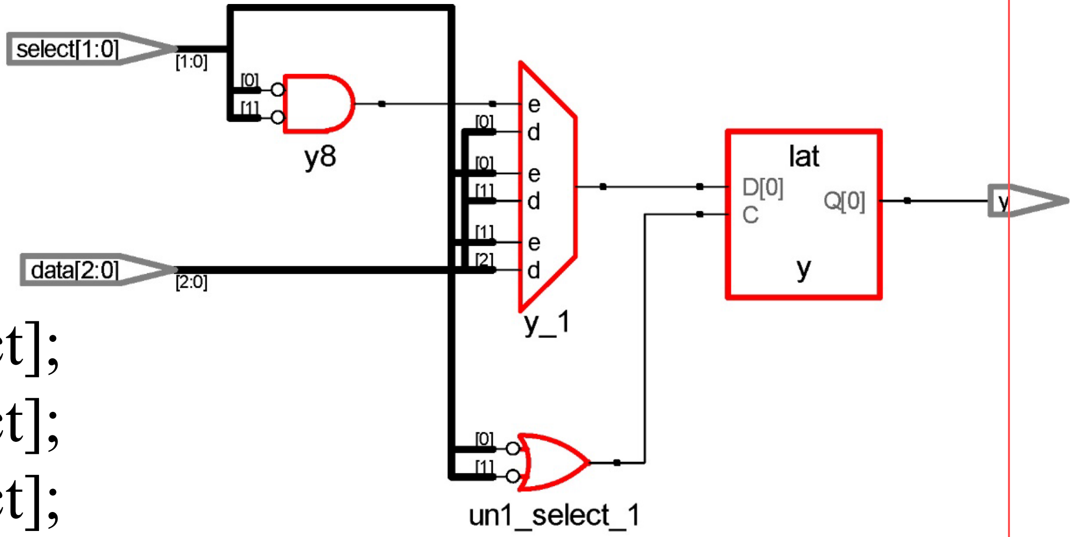
CODING STYLE

- Avoid using any latches in a 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
module latch_infer_case(select, data, y);
…
output reg y;
always @(select or data)
    case (select)
        2'b00: y = data[select];
        2'b01: y = data[select];
        2'b10: y = data[select];
        //      default: y = 2'b11;
    endcase
```

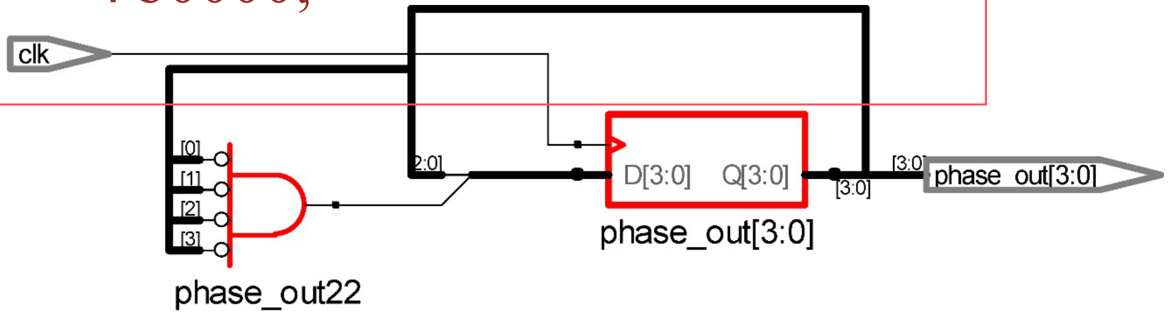


IGNCORED DELAY VALUES --- AN INCORRECT VERSION

```
// a four phase clock example --- incorrect
module four_phase_clock_wrong(clk, phase_out);
…
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
```

IGNORED DELAY VALUES --- A CORRECT VERSION

```
// a four phase clock example --- synthesizable version
…
output reg [3:0] phase_out;  // phase output
always @(posedge clk)
   case (phase_out)
      4'b0000: phase_out <=  4'b0001;
      4'b0001: phase_out <=  4'b0010;
      4'b0010: phase_out <=  4'b0100;
      4'b0100: phase_out <=  4'b1000;
      default:   phase_out <=  4'b0000;
   endcase
```



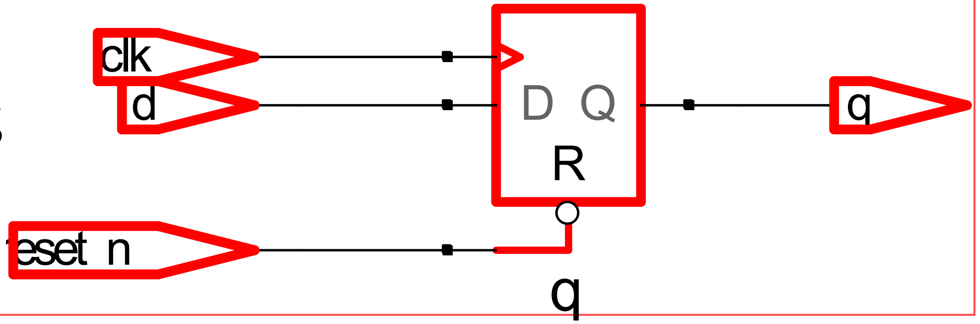
MIXED USE OF POSEDGE/LEVEL SIGNALS

```
// the mixed usage of posedge/negedge signal
// The result cannot be synthesized
module DFF_bad (clk, reset, d, q);
…
// the body of DFF
always @(posedge clk or reset)
begin
    if (reset) q <= 1'b0;
    else       q <= d;
end
```

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

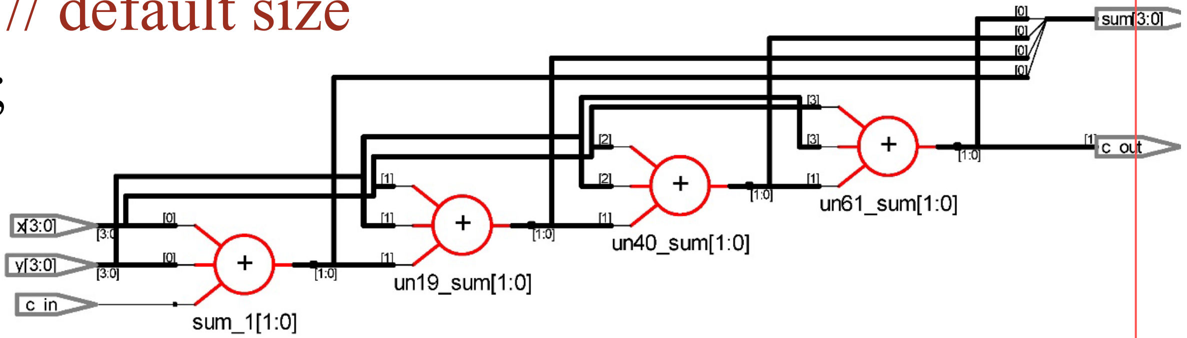
MIXED USE OF POSEDGE/NEGEDGE SIGNALS

```
// the mixed usage of posedge/negedge signal
module DFF_good (clk, reset_n, d, q);
…
// the body of DFF
always @(posedge clk or negedge reset_n)
begin
    if (!reset_n)  q <= 1'b0;
    else           q <= d;
end
```



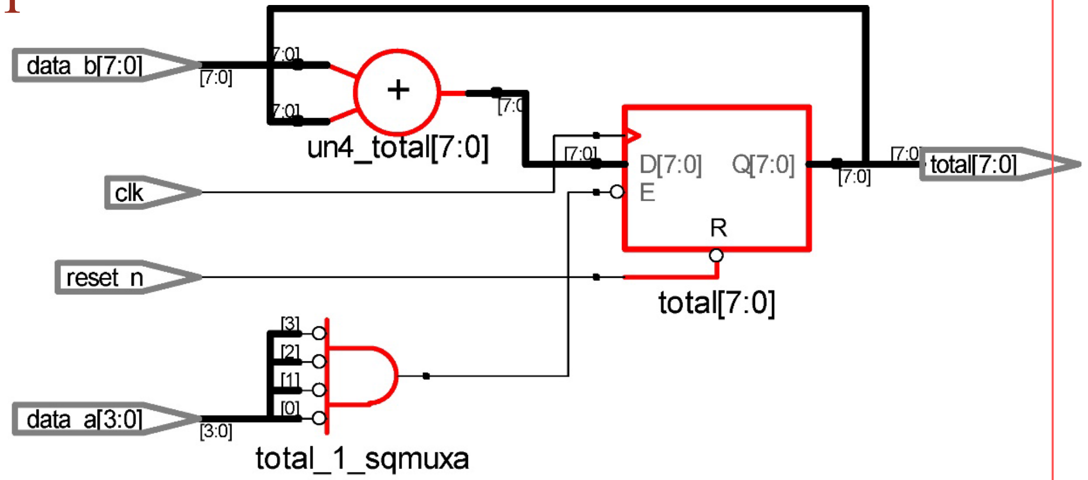
LOOP STRUCTURES

```
// an N-bit adder using for loop.
module nbit_adder_for( x, y, c_in, sum, c_out);
parameter N = 4;      // default size
input   [N-1:0] x, y;
…
integer i;
…
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
```



LOOP STRUCTURES --- AN INCORRECTLY SYNTHESIZABLE EXAMPLE

```
// a multiple cycle example --- This is an incorrect version.
…
parameter N = 8;
parameter M = 4;
input   clk, reset_n;
…
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
```



Q: Why the synthesized result is like this?

Try to explain it!

MEMORY SYNTHESIS APPROACHES

- A flip-flop
  - 10 to 20 times the area of a 6-transistor static RAM cell
- Random logic using flip-flops or latches
  - Independent of any software
  - Independent of the type of ASIC
  - Inefficient in terms of area
- Register files in datapaths
  - use a synthesis directive
  - hand instantiation

MEMORY SYNTHESIS APPROACHES

- RAM standard components
  - supplied by an ASIC vendor
  - depend on the technology
- RAM compilers
  - the most area-efficient approach

CODING GUIDELINES

CODING GUIDELINES

- Coding Guidelines for Synthesis
- Guidelines for Clocks
- Guidelines for Resets
- Partitioning for Synthesis

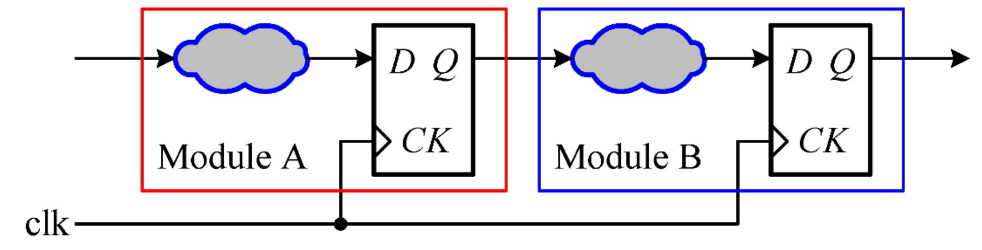
CODING GUIDELINES FOR SYNTHESIS

- Goals of coding guidelines
  - Testability
  - Performance
  - Simplification of static timing analysis
  - Matching gate-level behavior with that of the original RTL codes

GUIDELINES FOR CLOCKS

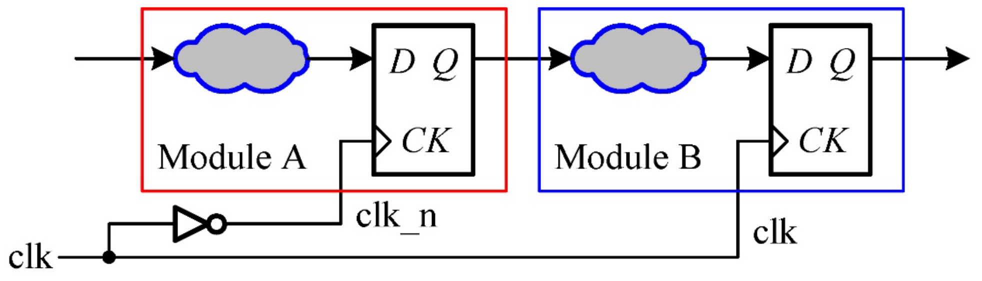
- Using single global clock
- Avoiding using gated clocks
- Avoiding mixed use of both positive and negative edge-triggered flip-flops
- Avoiding using internally generated clock signals

GUIDELINES FOR CLOCKS

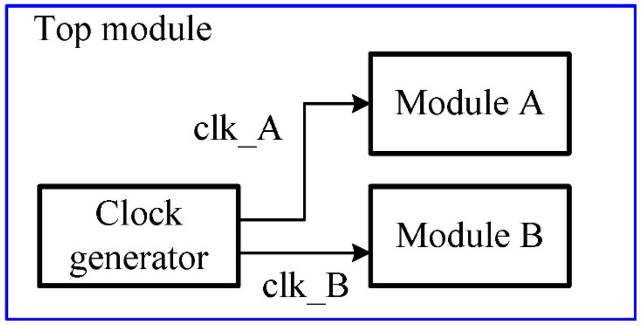


(a) An ideal clock scheme

: Combinational logic



(b) An example of using both positive and negative edge-triggered flip-flops



(c) Using a separate clock module at the top level.

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?

GUIDELINES FOR RESETS

- The basic writing styles:

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

Asynchronous reset

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

Synchronous reset

- The reset signal should be a direct clear of all flip-flops

GUIDELINES FOR RESETS

- Asynchronous reset
  - Hard to implement
  - Does not require a free-running clock
  - Does not affect flip flop data timing
  - Makes STA more difficult
  - Makes the automatic insertion of test structure more difficult
- Synchronous reset
  - easy to implement
  - Requires a free-running clock

GUIDELINES FOR RESETS

- Avoid internally generated conditional resets

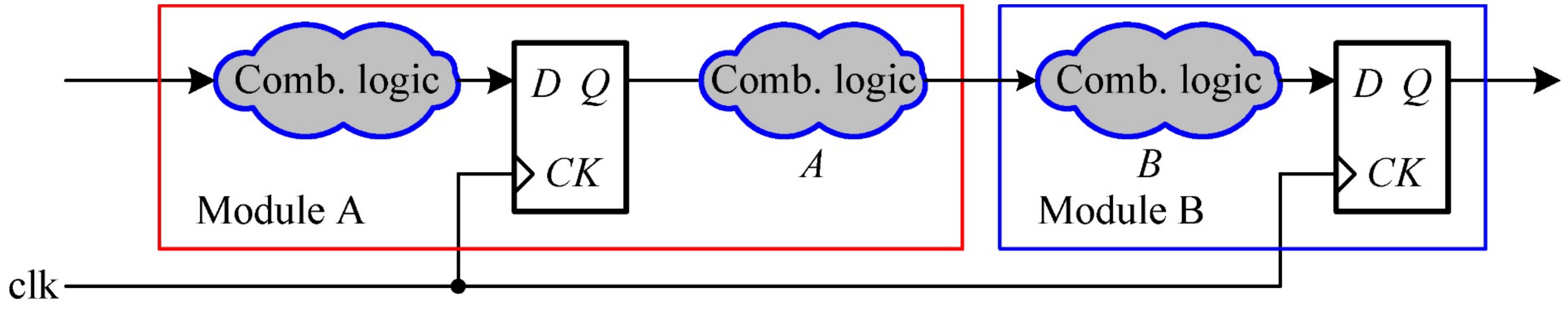
```
always @(posedge gate or negedge reset_n or posedge timer_load_clear)
    if (!reset_n || timer_load_clear) timer_load <= 1'b0;
    else timer_load <= 1'b1;
```

- When a conditional reset is required:

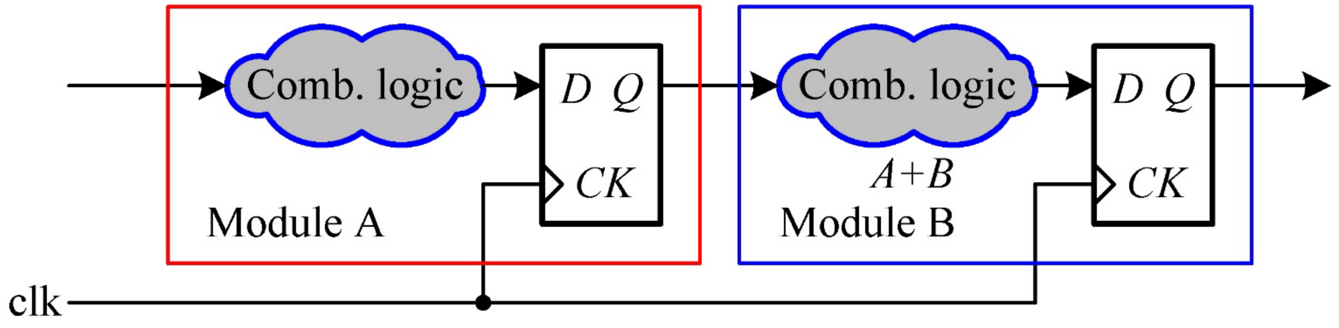
```
assign timer_load_reset = !reset_n || timer_load_clear;
always @(posedge gate or posedge timer_load_reset)
    if (timer_load_reset) timer_load <= 1'b0;
    else timer_load <= 1'b1;
```

PARTITIONING FOR SYNTHESIS

- Keep related logic within the same module



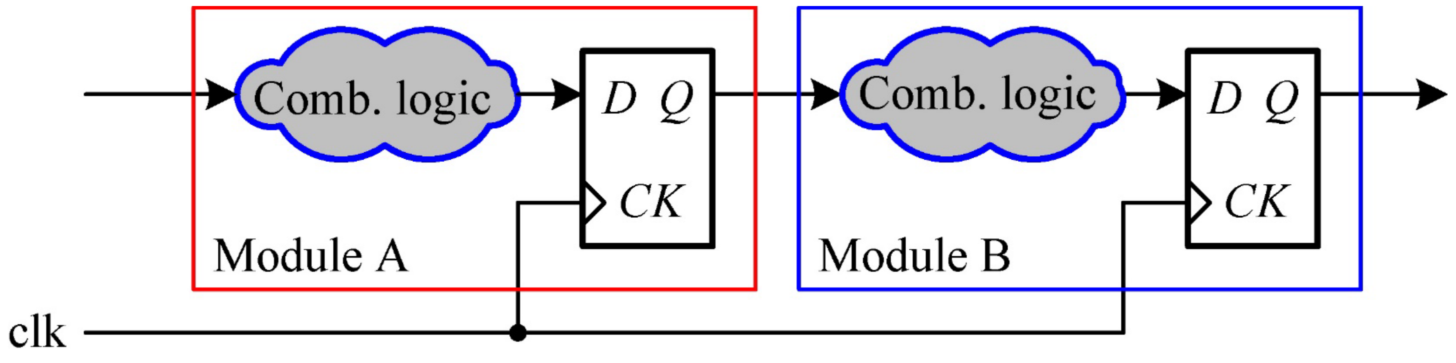
(a) Bad style



(b) Good style

PARTITIONING FOR SYNTHESIS

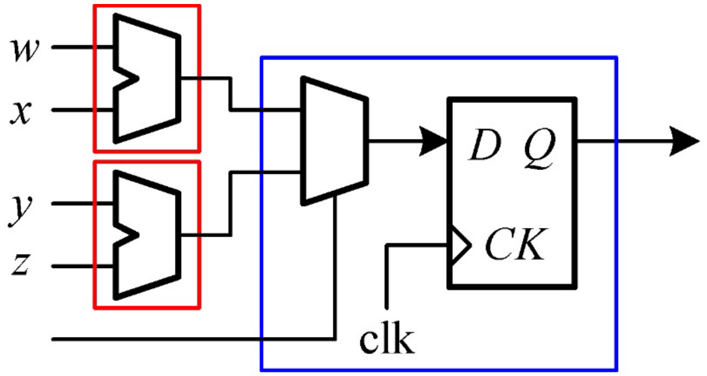
- Register all outputs



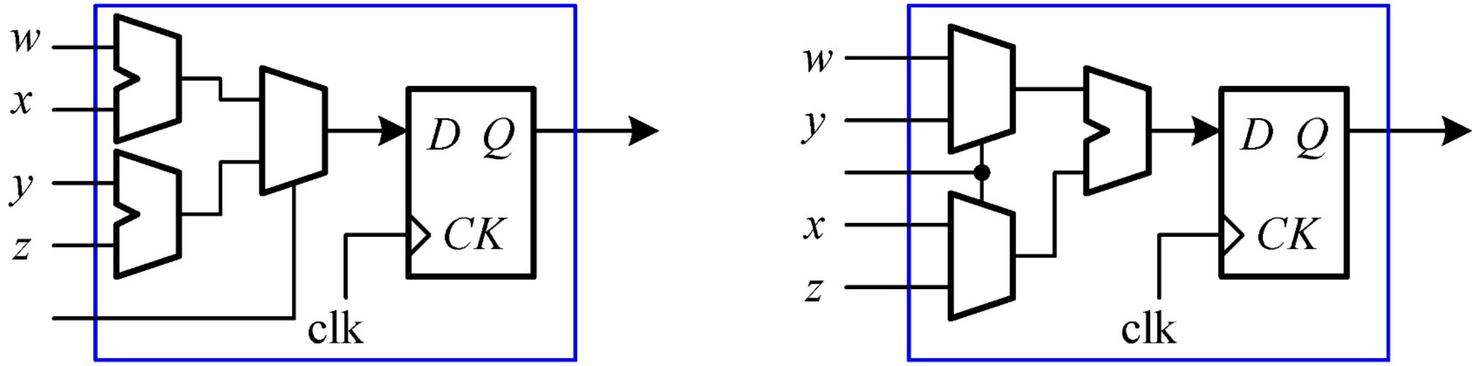
- Separating structural logic from random logic

PARTITIONING FOR SYNTHESIS

- Maintaining the original hierarchy



(a) Resources in different modules cannot be shared.



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