Synthesis Of VHDL Code

Outline

1. Fundamental limitation of EDA software
2. Realization of VHDL operator
3. Realization of VHDL data type
4. VHDL synthesis flow
5. Timing consideration

1. Fundamental limitation of EDA software

- What is the synthesis of VHDL code
  - the process of realizing the VHDL description using primitive logic cells from the target device's library.
- Can "C-to-hardware" be done?
- EDA tools:
  - Core: the algorithm that perform the transformation or optimization
  - Shell: wrapping the algorithm
- What does theoretical computer science say?
  - Computability
  - Computation complexity

Computability

- A problem is computable if an algorithm exists.
- E.g., "halting problem":
  - can we develop a program that takes any program and its input, and determines whether the computation of that program will eventually halt?
- any attempt to examine the "meaning" of a program is uncomputable

Computation complexity

- How fast an algorithm can run (or how good an algorithm is)? – time complexity
  - "Interferences" in measuring execution time: types of CPU, speed of CPU, compiler etc.
- How much hardware resources are used – space complexity

Big-*O* notation

- $f(n)$ is $O(g(n))$:
  if n_0 and c can be found to satisfy:
  $f(n) < cg(n)$ for any n, $n > n_0$
- $g(n)$ is simple function: $1, n, \log_2 n, n^2, n^3, 2^n$
- Following are $O(n^2)$:

DFT n^2
FFT nlog2n

  - $0.1n^2$
  - $n^2 + 5n + 9$
  - $500n^2 + 1000000$

Interpretation of Big-*O*

- Filter out the “interference”: constants and less important terms
- n is the input size of an algorithm
- The “scaling factor” of an algorithm: What happens if the input size increases

E.g.,

input size	**Big-O function**					
n	n	$\log_2 n$	$n \log_2 n$	n^2	n^3	2^n
2	2 μs	1 μs	2 μs	4 μs	8 μs	4 μs
4	4 μs	2 μs	8 μs	16 μs	64 μs	16 μs
8	8 μs	3 μs	24 μs	64 μs	512 μs	256 μs
16	16 μs	4 μs	64 μs	256 μs	4 ms	66 ms
32	32 μs	5 μs	160 μs	1 ms	33 ms	71 min
48	48 μs	5.5 μs	268 μs	2 ms	111 ms	9 year
64	64 μs	6 μs	384 μs	4 ms	262 ms	600,000 year

- Intractable problems:
  - algorithms with $O(2^n)$
  - Not realistic for a larger n
  - Frequently tractable algorithms for sub-optimal solution exist
- Many problems encountered in synthesis are intractable

Theoretical limitation

- Synthesis software does not know your intention
- Synthesis software cannot obtain the optimal solution
- Synthesis should be treated as transformation and a “local search” in the “design space”
- Good VHDL code provides a good starting point for the local search
- The quality and efficiency of a design still rely on a human designer's experience, insight, ingenuity and imagination
  - the ultimate heuristics that cannot be coded into software

2. Realization of VHDL operator

- Logic operator
  - Simple, direct mapping
- Relational operator
  - =, /= fast, simple implementation exists
  - >, < etc: more complex implementation, larger delay
- Addition operator
- Other arithematic operators: support varies

- Operator with two constant operands:
  - Simplified in preprocessing
  - No hardware inferred
  - Good for documentation
  - E.g.,

```
constant OFFSET: integer := 8;
signal boundary: unsigned(8 downto 0);
signal overflow: std_logic;
. . .
overflow <= '1' when boundary > (2**OFFSET-1) else
            '0';
```

- Operator with one constant operand:
  - Can significantly reduce the hardware complexity
  - E.g., adder vs. incrementor
  - E.g

```
y <= rotate_right(x, z);  -- barrel shifter
y <= rotate_right(x, 3);  -- rewiring
y <= x(2 downto 0) & x(7 downto 3);
```

  - E.g., 4-bit comparator: x=y vs. x=0

$$(x_3 \oplus y_3)' \cdot (x_2 \oplus y_2)' \cdot (x_1 \oplus y_1)' \cdot (x_0 \oplus y_0)'$$

$$x_3' \cdot x_2' \cdot x_1' \cdot x_0'$$

An example 0.55 um standard-cell CMOS implementation

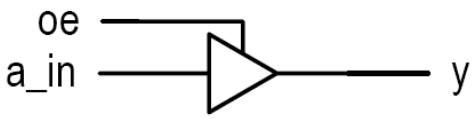
width	VHDL operator									
	nand	xor	$>_a$	$>_d$	=	$+1_a$	$+1_d$	$+_a$	$+_d$	mux
					area (gate count)					
8	8	22	25	68	26	27	33	51	118	21
16	16	44	52	102	51	55	73	101	265	42
32	32	85	105	211	102	113	153	203	437	85
64	64	171	212	398	204	227	313	405	755	171
					delay (ns)					
8	0.1	0.4	4.0	1.9	1.0	2.4	1.5	4.2	3.2	0.3
16	0.1	0.4	8.6	3.7	1.7	5.5	3.3	8.2	5.5	0.3
32	0.1	0.4	17.6	6.7	1.8	11.6	7.5	16.2	11.1	0.3
64	0.1	0.4	35.7	14.3	2.2	24.0	15.7	32.2	22.9	0.3

3. Realization of VHDL data type

- Use and synthesis of ‘Z’
- Use of ‘-’

Use and synthesis of ‘Z’

- Tri-state buffer:
  - Output with “high-impedance”
  - Not a value in Boolean algebra
  - Need special output circuitry (tri-state buffer)



oe	y
0	Z
1	a_in

- Major application:
  - Bi-directional I/O pins
  - Tri-state bus
- VHDL description:

```
y <= 'Z' when oe='1' else
     a_in;
```

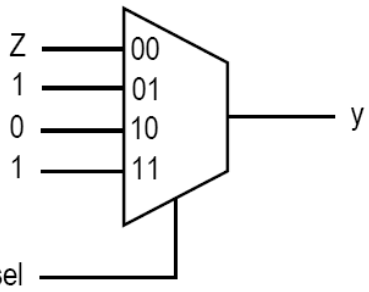
- 'Z' **cannot** be used as input or manipulated

```
f <= 'Z' and a;
```

```
y <= data_a when in_bus='Z' else
     data_b;
```

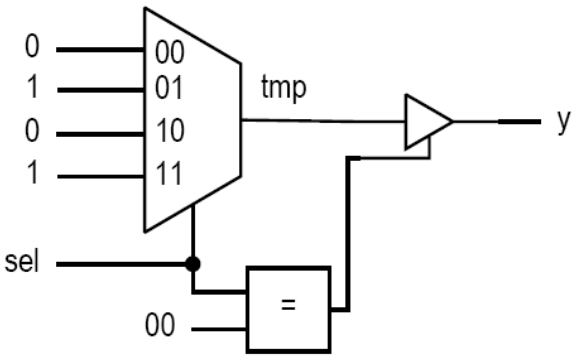
- Separate tri-state buffer from regular code:
  - **Less clear**:

```
with sel select
  y <= 'Z' when "00",
       '1' when "01"|"11",
       '0' when others;
```

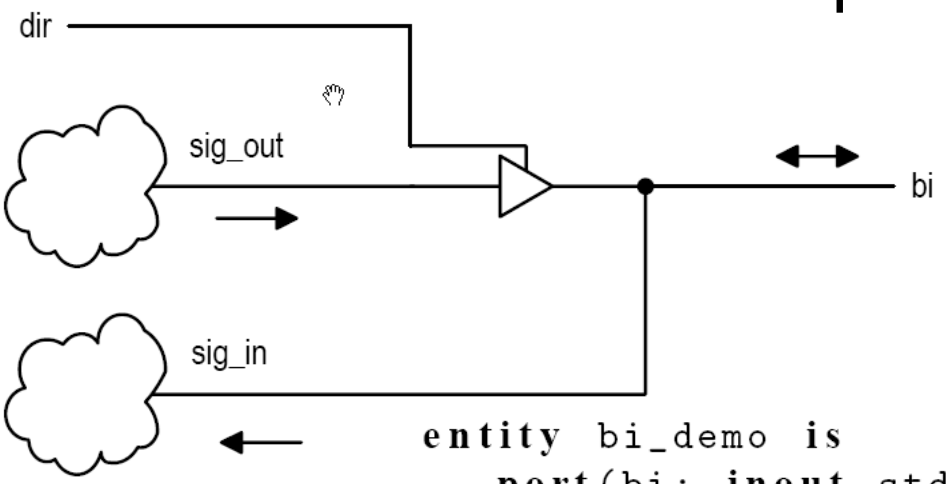


  - **better**:

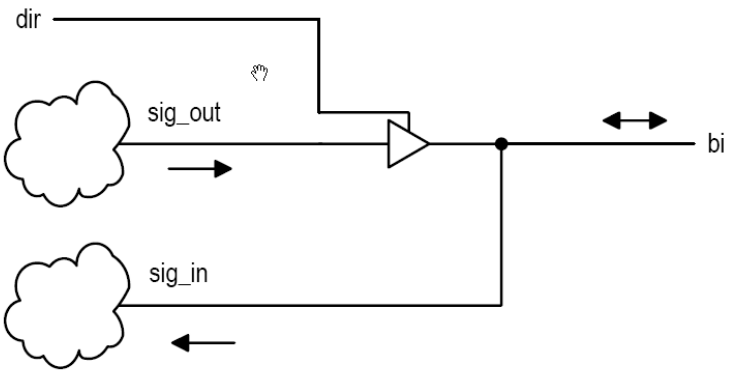
```
with sel select
  tmp <= '1' when "01"|"11",
         '0' when others;
  y <= 'Z' when sel="00" else
       tmp;
```

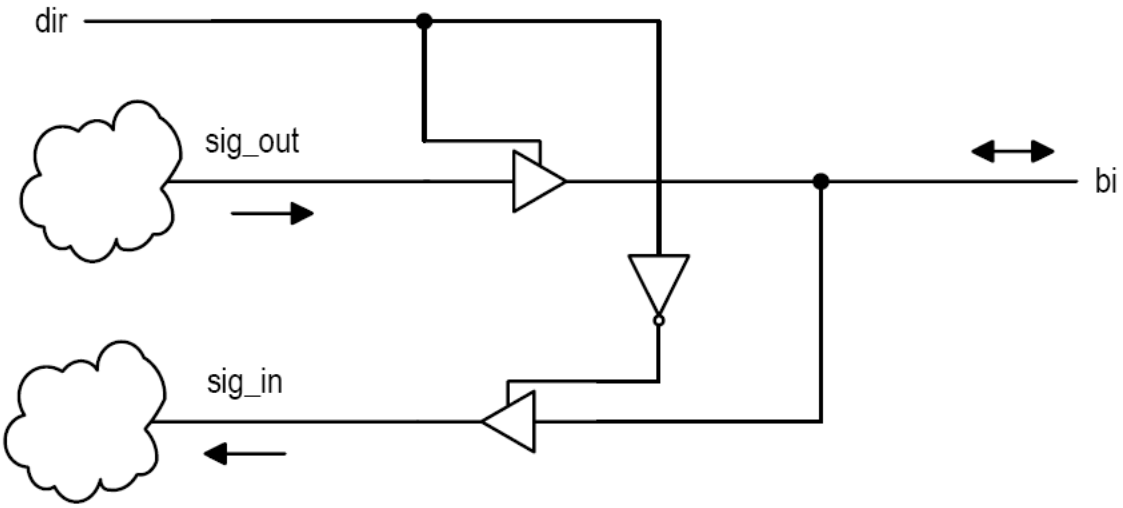


Bi-directional i/o pins



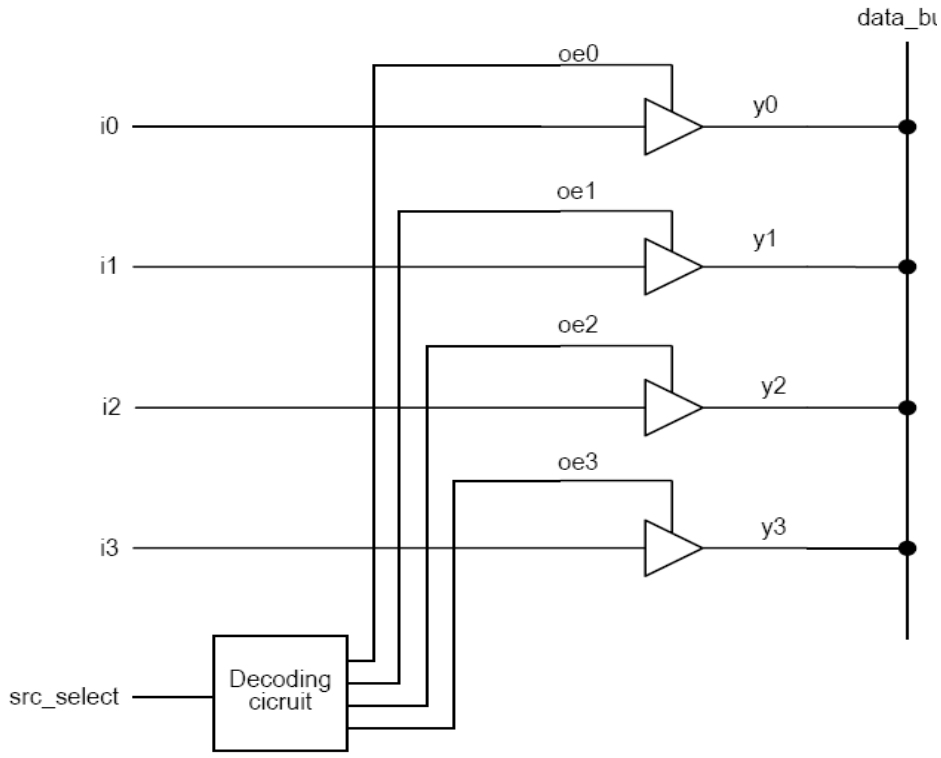

```
entity bi_demo is
   port(bi: inout std_logic;
 . . .

begin
   sig_out <= output_expression;
    . . .  <= expression_with_sig_in;
   . . .
   bi <= sig_out when dir='1' else 'Z';
   sig_in <= bi;
   . . .
```




```
sig_in <= bi when dir='0' else 'Z';
```

Tri-state bus




```
-- binary decoder
with src_select select
    oe <= "0001" when "00",
          "0010" when "01",
          "0100" when "10",
          "1000" when others; -- "11"
-- tri-state buffers
y0 <= i0 when oe(0)='1' else 'Z';
y1 <= i1 when oe(1)='1' else 'Z';
y2 <= i2 when oe(2)='1' else 'Z';
y3 <= i3 when oe(3)='1' else 'Z';
data_bus <= y0;
data_bus <= y1;
data_bus <= y2;
data_bus <= y3;
```

- Problem with tri-state bus
  - Difficult to optimize, verify and test
  - Technology dependent, and hence less portable.
- Alternative to tri-state bus: mux

```
with src_select select
    data_bus <= i0 when "00",
                i1 when "01",
                i2 when "10",
                i3 when others;   -- "11"
```


Use of '-'

- In conventional logic design
  - '-' as input value: shorthand to make table compact
  - E.g.,

input req	output code
1 0 0	10
1 0 1	10
1 1 0	10
1 1 1	10
0 1 0	01
0 1 1	01
0 0 1	00
0 0 0	00

input req	output code
1 – –	10
0 1 –	01
0 0 1	00
0 0 0	00

– '-' as output value: help simplification

– E.g.,
  '-' assigned to 1: a + b
  '-' assigned to 0: a'b + ab'

input $a\ b$	output f
0 0	0
0 1	1
1 0	1
1 1	–

Use '-' in VHDL

- As input value (against our intuition):
- Wrong:

```
y <= "10" when req="1--" else
     "01" when req="01-" else
     "00" when req="001" else
     "00";
```

- Fix #1:

```
y <= "10" when req(3)='1' else
     "01" when req(3 downto 2)="01" else
     "00" when req(3 downto 1)="001" else
     "00";
```

- Fix #2:

```
. . .
use ieee.numeric_std.all;
. . .
y <= "10" when std_match(req,"1--") else
     "01" when std_match(req,"01-") else
     "00" when std_match(req,"001") else
     "00";
```


- Wrong:

```
with req select
   y <= "10" when "1--",
        "01" when "01-",
        "00" when "001",
        "00" when others;
```

- Fix:

```
with req select
   y <= "10" when "100"|"101"|"110"|"111",
        "00" when "010"|"011",
        "00" when others;
```

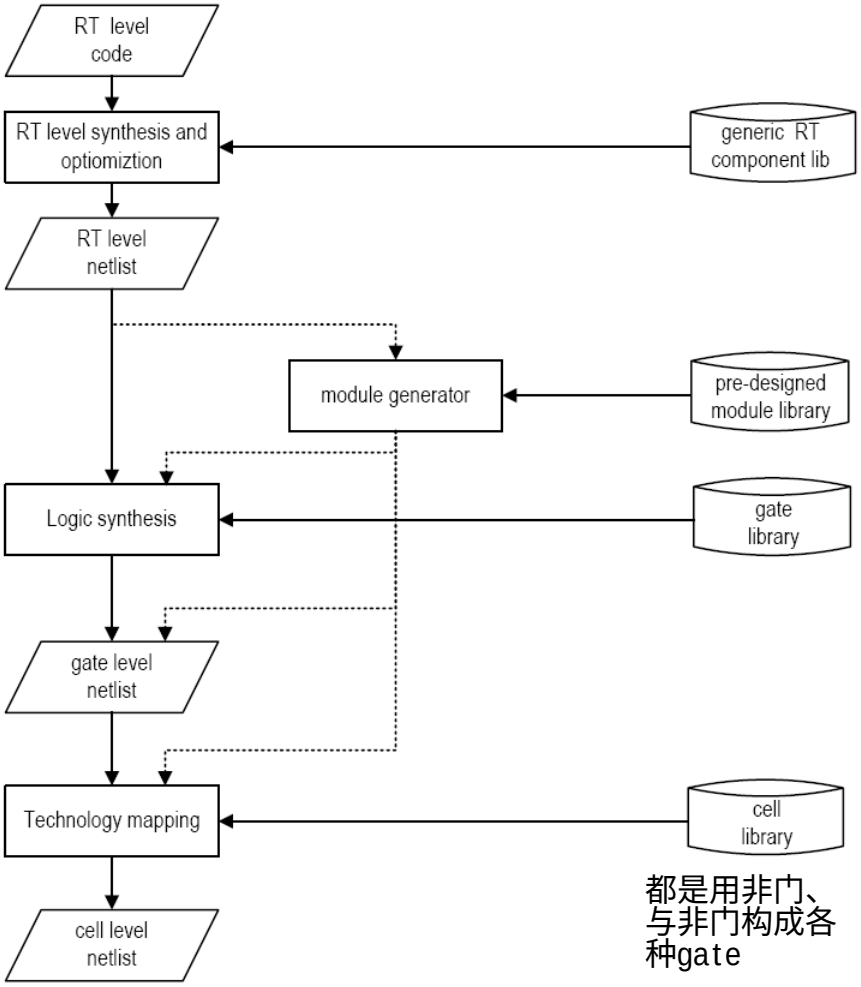
- '-' as an output value in VHDL
- May work with some software

```
sel <= a & b;
with sel select
    y <= '0' when "00",
         '1' when "01",
         '1' when "10",
         '-' when others;
```

input $a\ b$	output f
0 0	0
0 1	1
1 0	1
1 1	–

4. VHDL Synthesis Flow

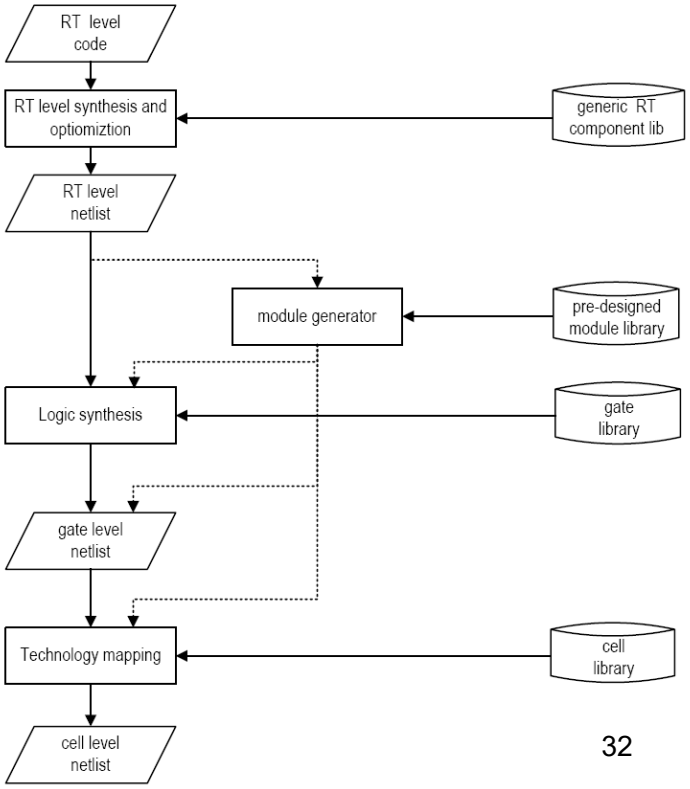
- Synthesis:
  - Realize VHDL code using logic cells from the device's library
  - a refinement process
- Main steps:
  - RT level synthesis
  - Logic synthesis
  - Technology mapping



都是用非门、与非门构成各种gate

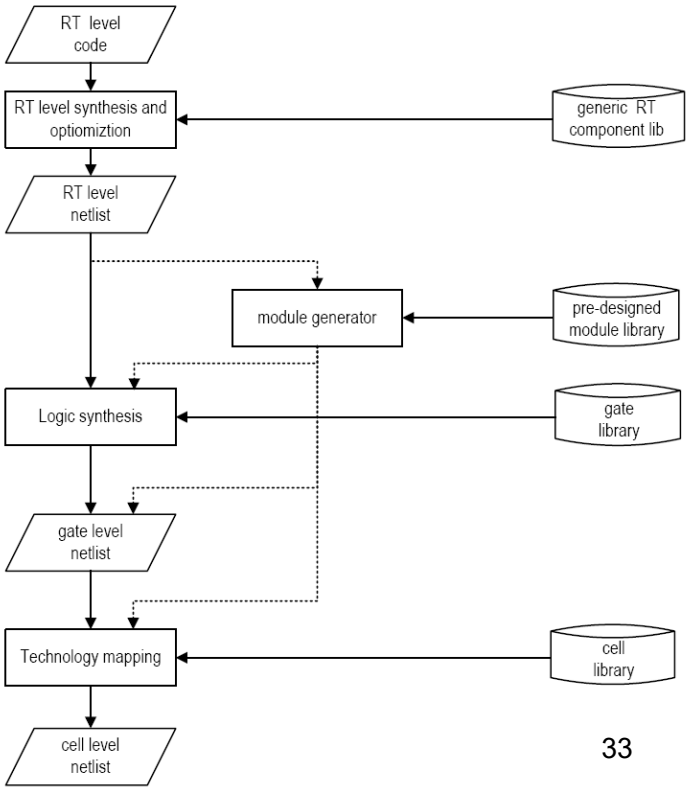
RT level synthesis

- Realize VHDL code using RT-level components
- Somewhat like the derivation of the conceptual diagram
- Limited optimization
- Generated netlist includes
  - "regular" logic: e.g., adder, comparator
  - "random" logic: e.g., truth table description



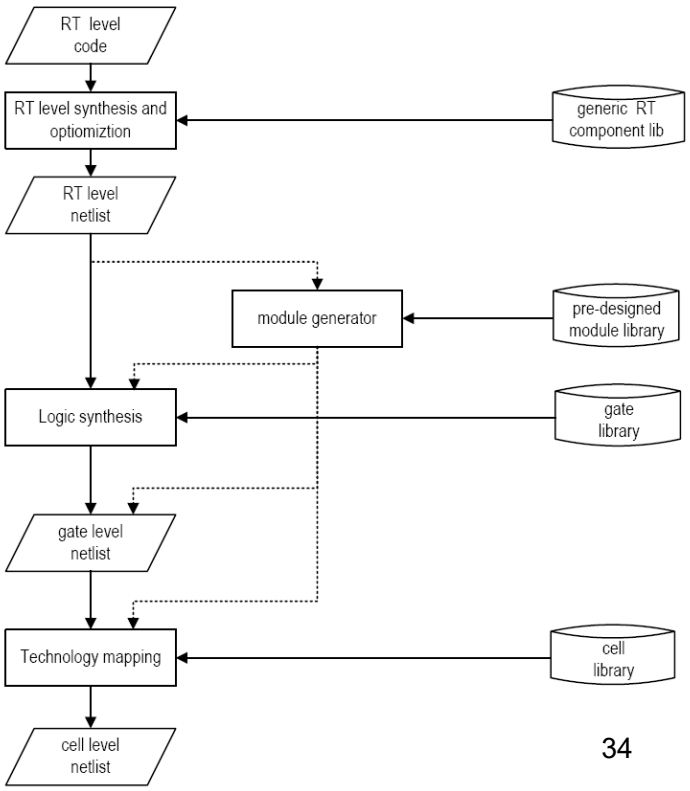
Module generator

- "regular" logic can be replaced by pre-designed module
  - Pre-designed module is more efficient
  - Module can be generated in different levels of detail
  - Reduce the processing time

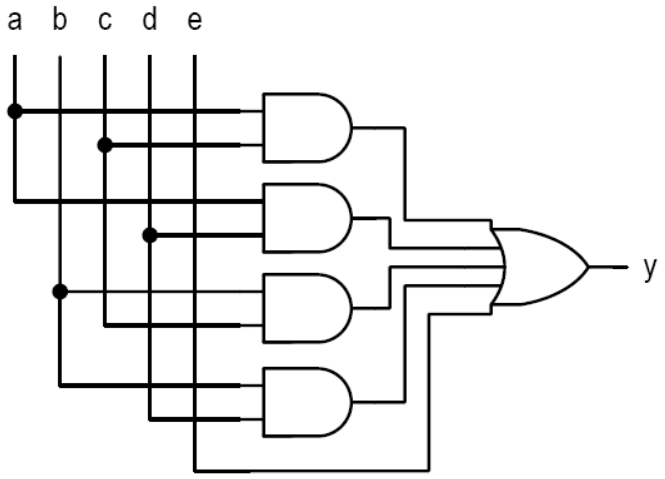


Logic Synthesis

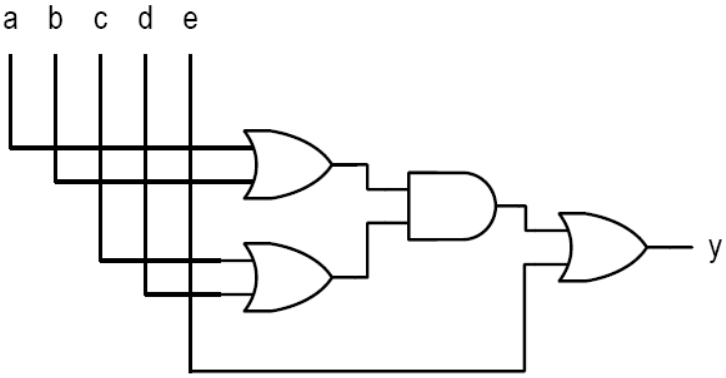
- Realize the circuit with the optimal number of "generic" gate level components
- Process the "random" logic
- Two categories:
  - Two-level synthesis: sum-of-product format
  - Multi-level synthesis



- E.g.,



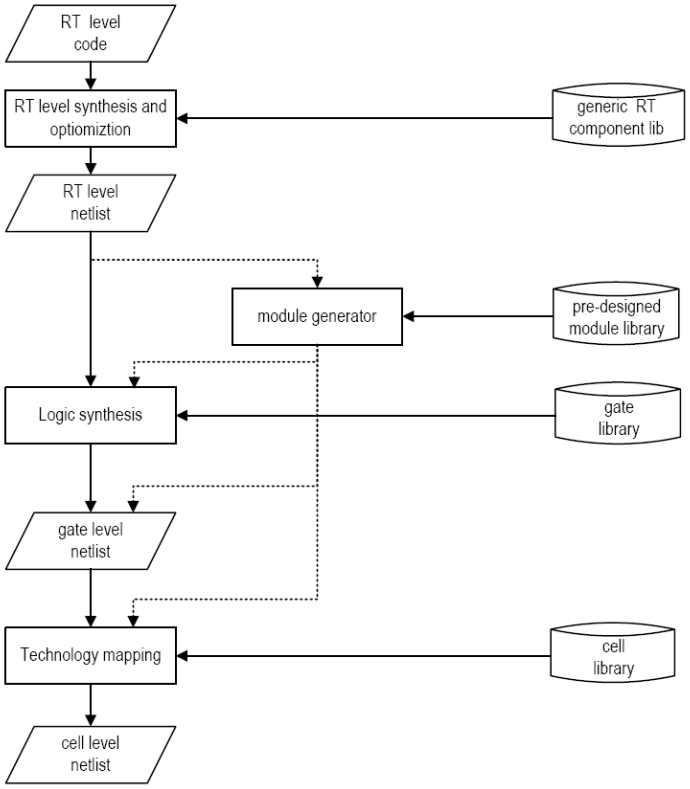
(a) Two-level implementation



(b) multi-level implementation

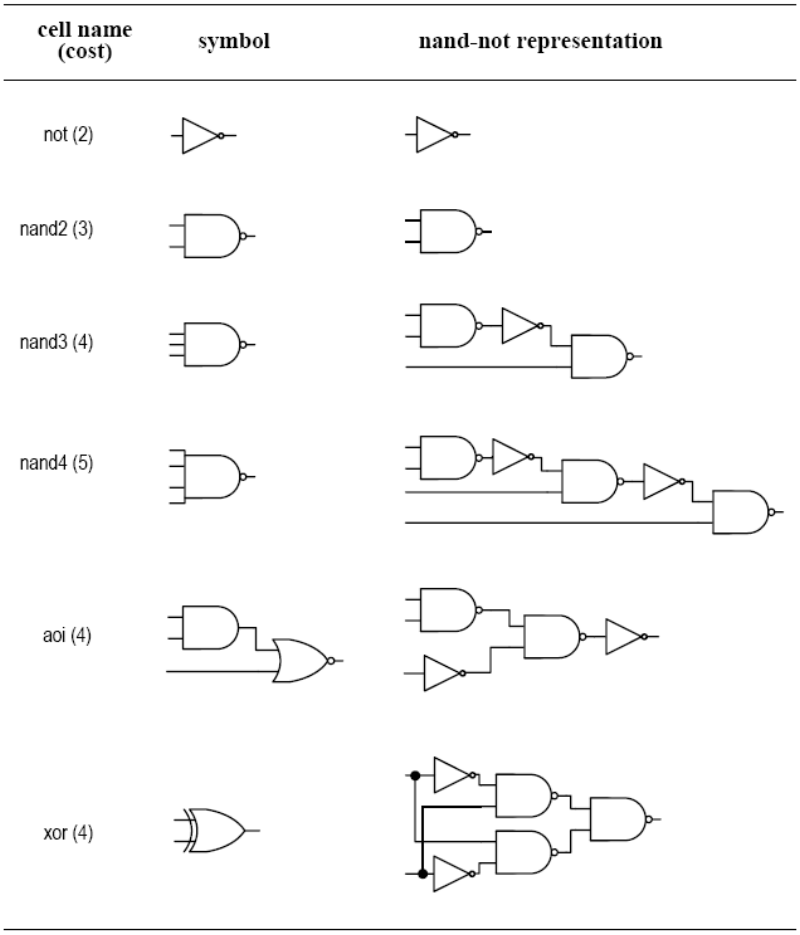
Technology mapping

- Map “generic” gates to “device-dependent” logic cells
- The technology library is provided by the vendors who manufactured (in FPGA) or will manufacture (in ASIC) the device

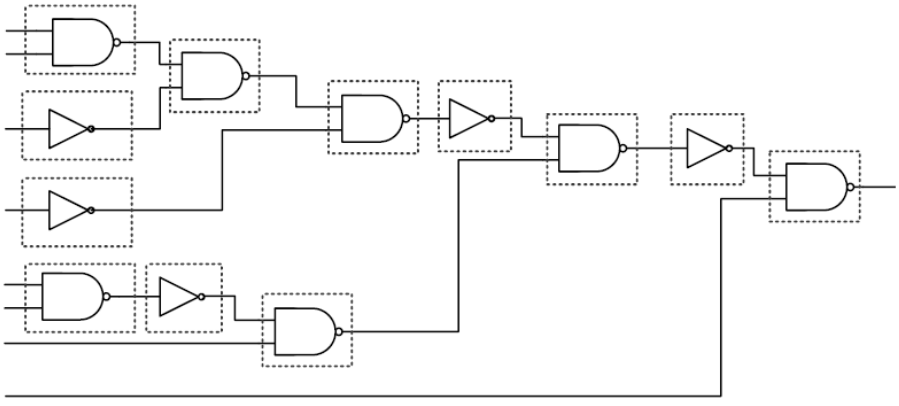


E.g., mapping in standard-cell ASIC

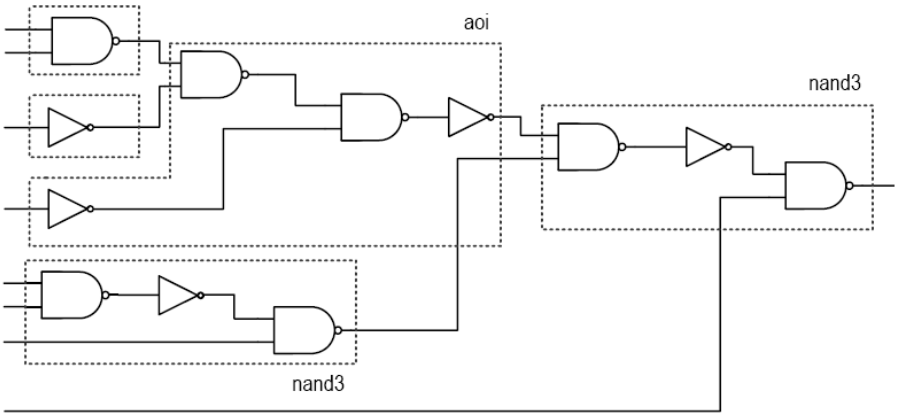
- Device library



- Cost: 31 vs. 17



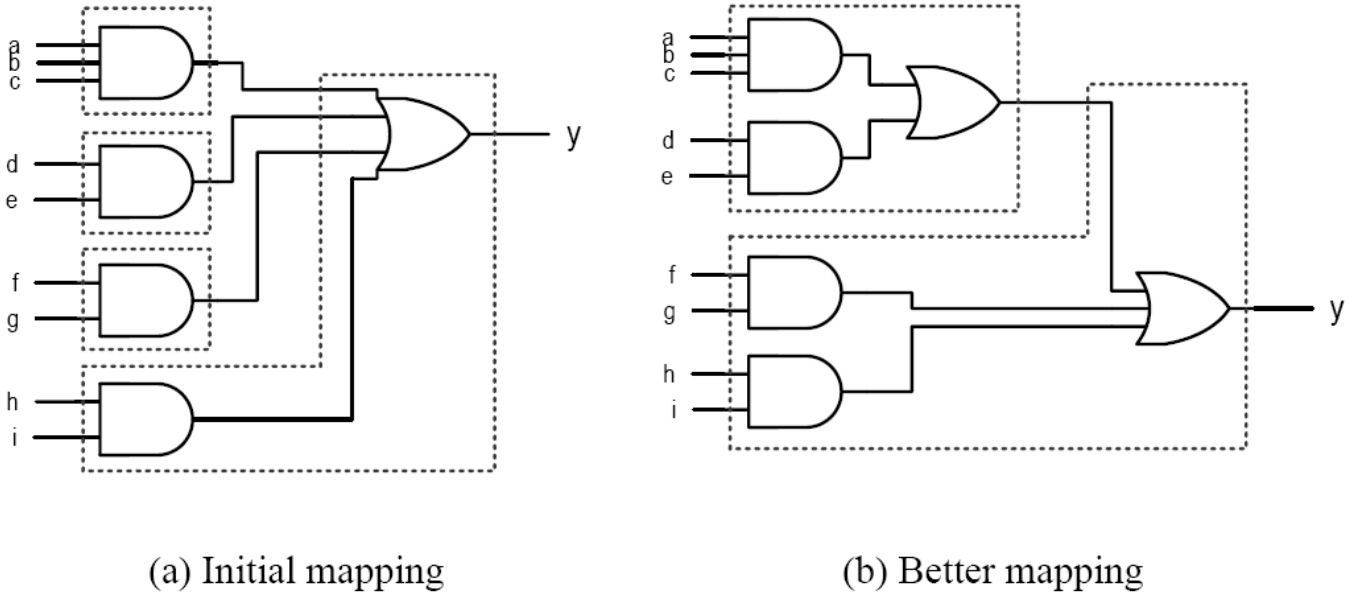
(a) Initial mapping



(b) Better mapping

E.g., mapping in FPGA

- With 5-input LUT (Look-Up-Table) cells

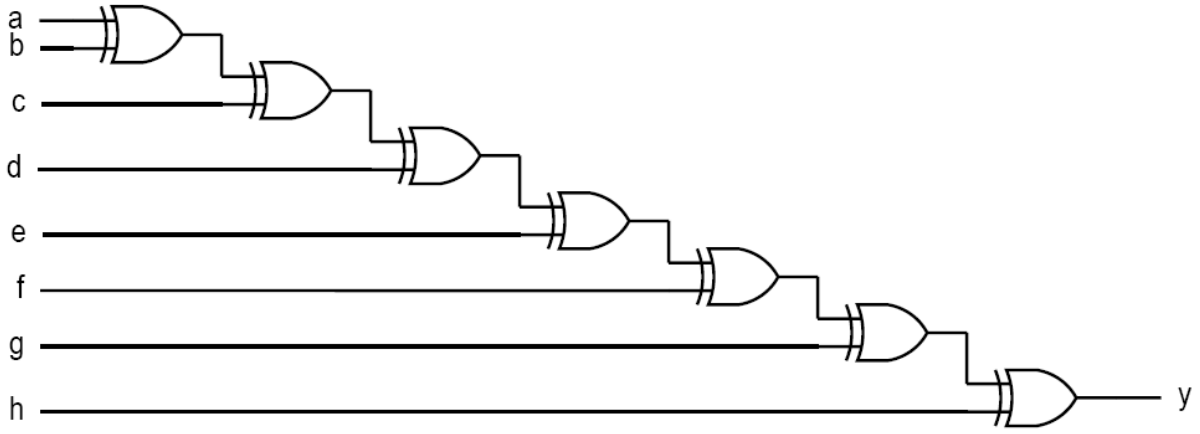


(a) Initial mapping

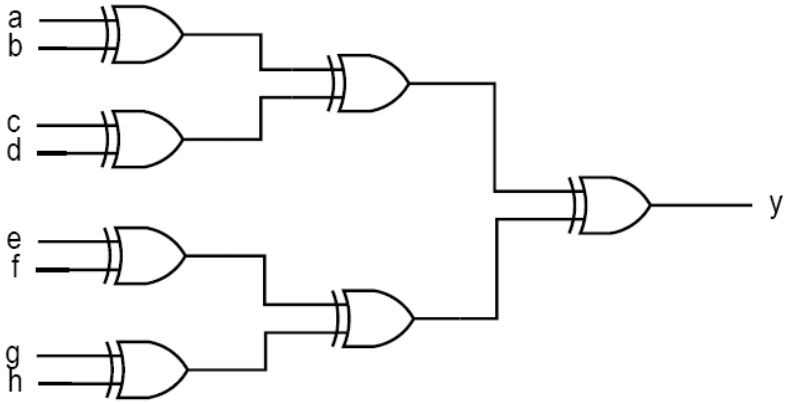
(b) Better mapping

Effective use of synthesis software

- Logic operators: software can do a good job
- Relational/Arith operators: manual intervention needed
- “layout” and “routing structure”:
  - Silicon chip is 2-dimensional square
  - “rectangular” or “tree-shaped” circuit is easier to optimize



(a) Cascading-chain structure



5. Combinational Circuit Timing consideration

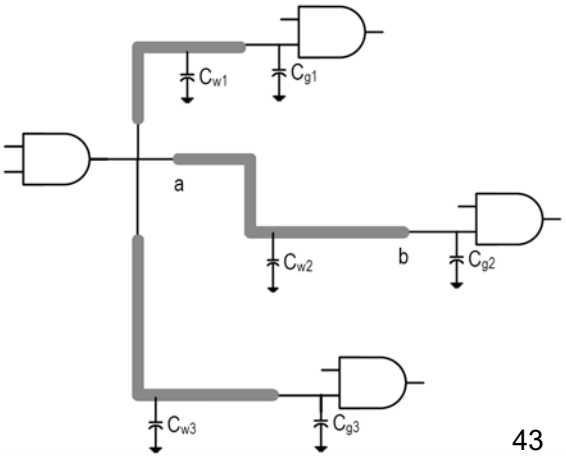
- Propagation delay
- Synthesis with timing constraint
- Hazards
- Delay-sensitive design

Propagation delay

- Delay: time required to propagate a signal from an input port to an output port
- Cell level delay: most accurate
- Simplified model:

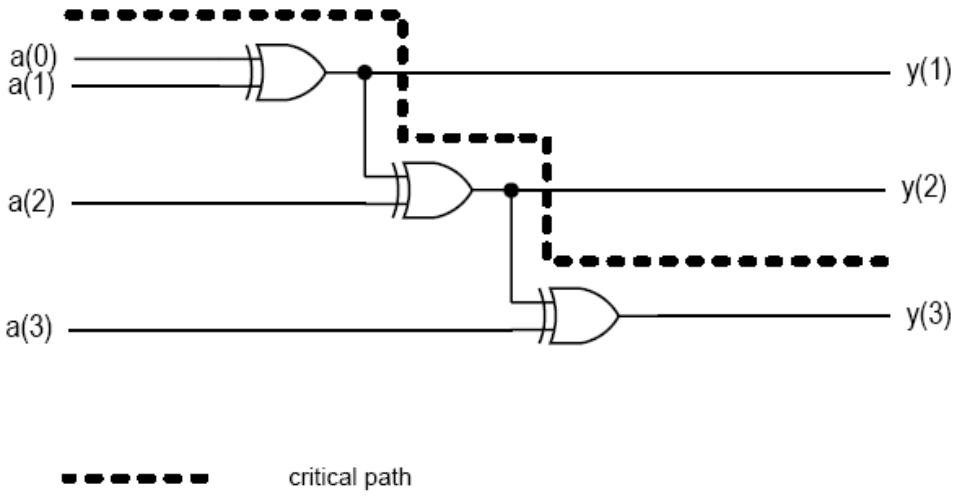
$$delay = d_{intrinsic} + r * C_{load}$$

- The impact of wire becomes more dominant as the transistor becomes smaller

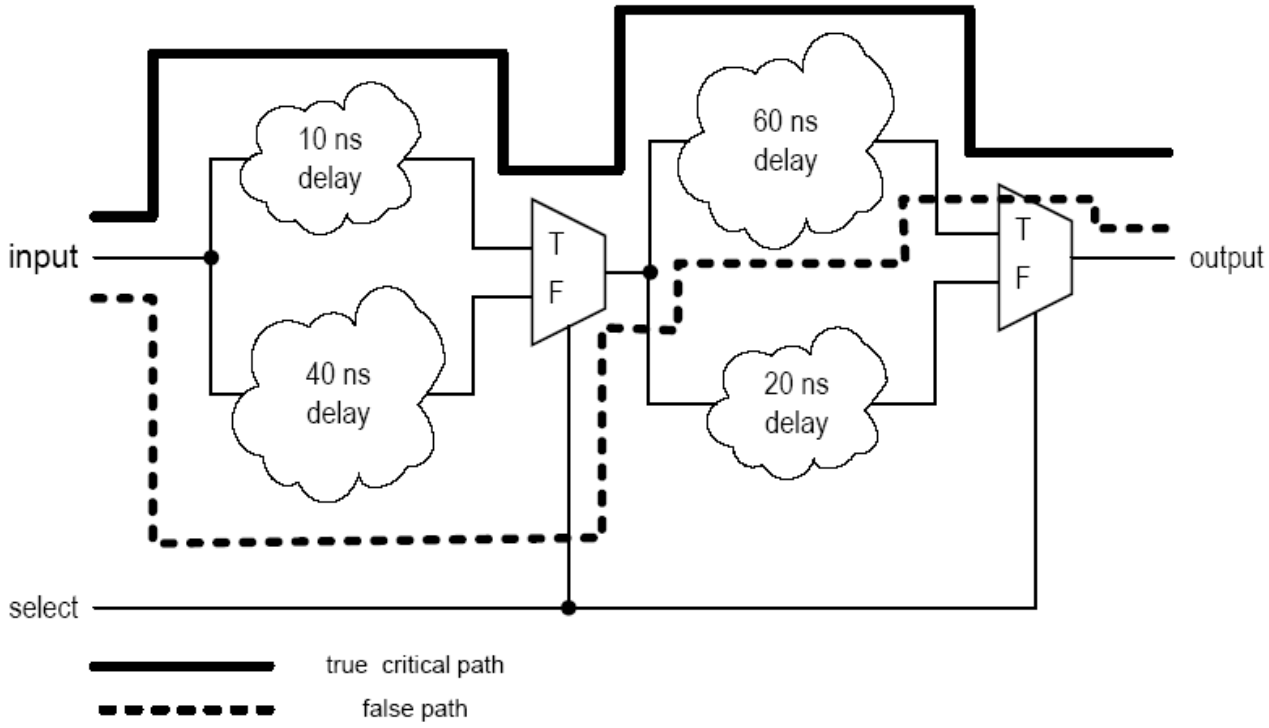


System delay

- The longest path (critical path) in the system
- The worst input to output delay
- E.g.,



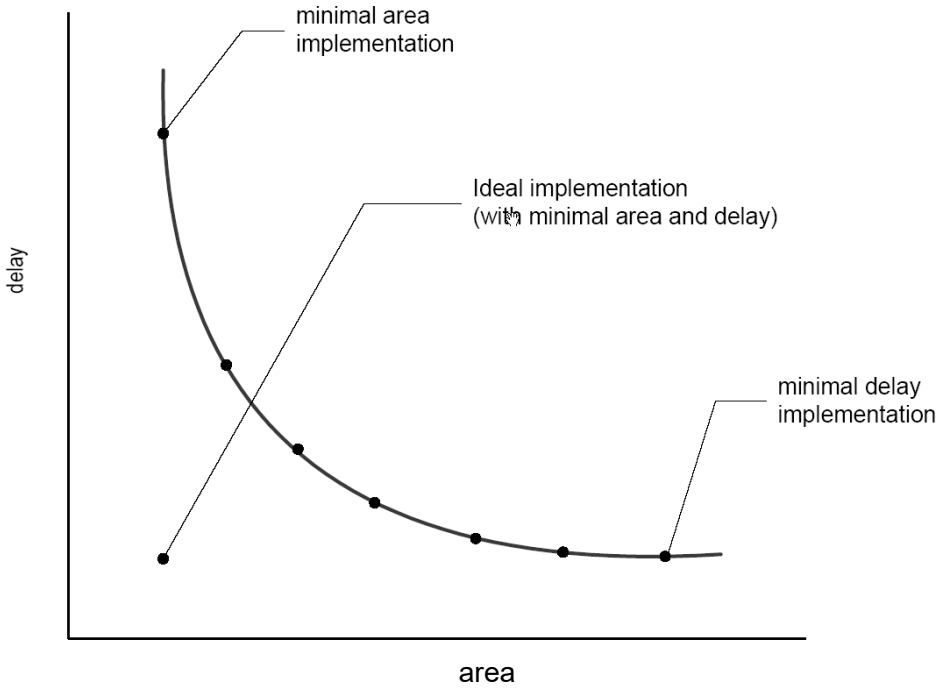
- “False path” may exists:



- RT level delay estimation:
  - Difficult if the design is mainly “random” logic
  - Critical path can be identified if many complex operators (such adder) are used in the design.

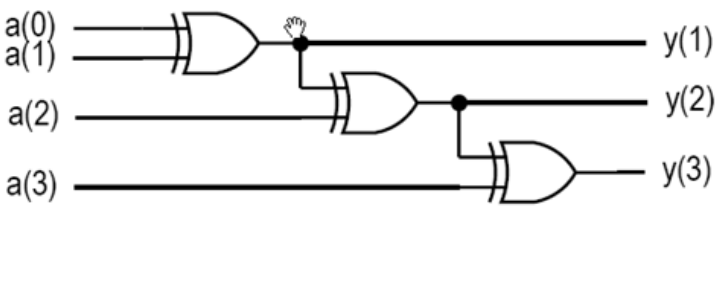
Synthesis with timing constraint

- Area-delay trade-off curve

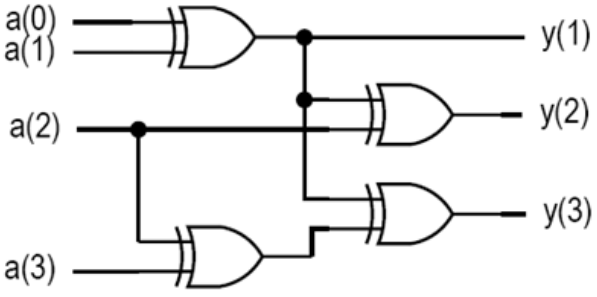


Synthesis with timing constraint

- Multi-level synthesis is flexible
- It is possible to reduce delay by adding extra logic



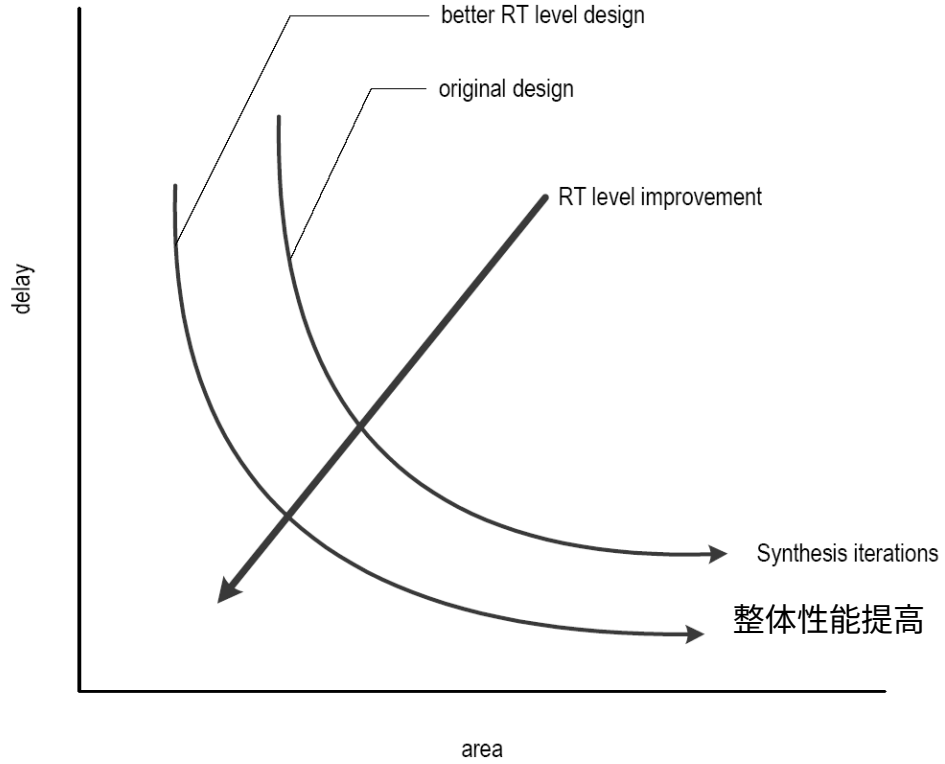
(a) Optimized for area



(b) Optimized for delay

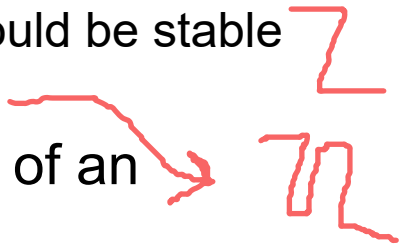
- Synthesis with timing constraint
  1. Obtain the minimal-area implementation
  2. Identify the critical path
  3. Reduce the delay by adding extra logic
  4. Repeat 2 & 3 until meeting the constraint

- Improvement in “architectural” level design (better VHDL code to start with)

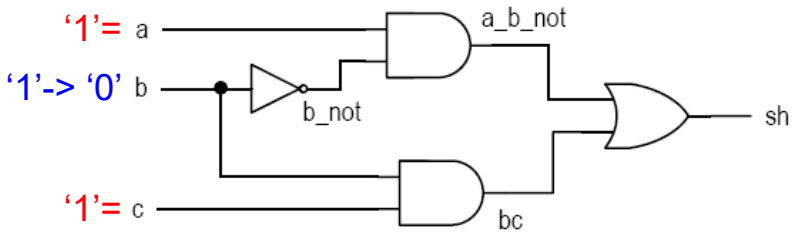


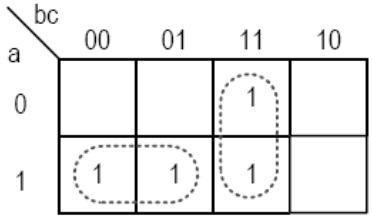
Timing Hazards

- Propagation delay: time to obtain a stable output
- Hazards: the fluctuation occurring during the transient period
  - Static hazard: glitch when the signal should be stable
  - Dynamic hazard: a glitch in transition
- Due to the multiple converging paths of an output port



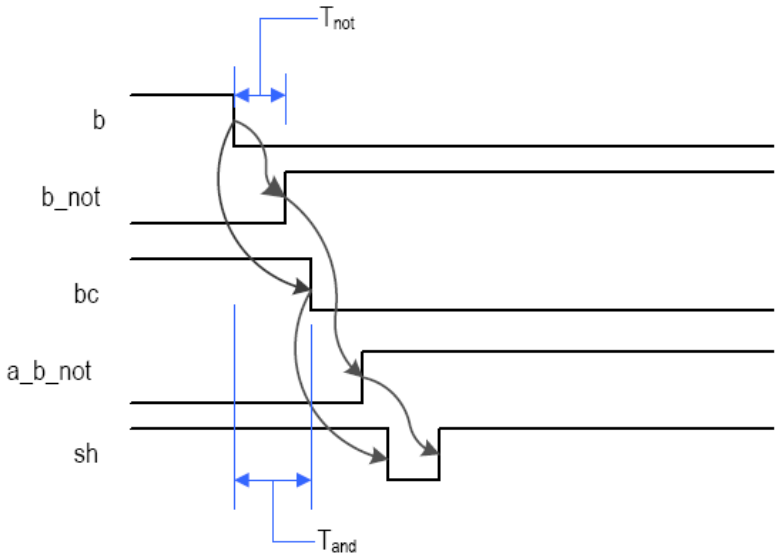
- E.g., static-hazard (sh=ab'+bc; a=c=1)





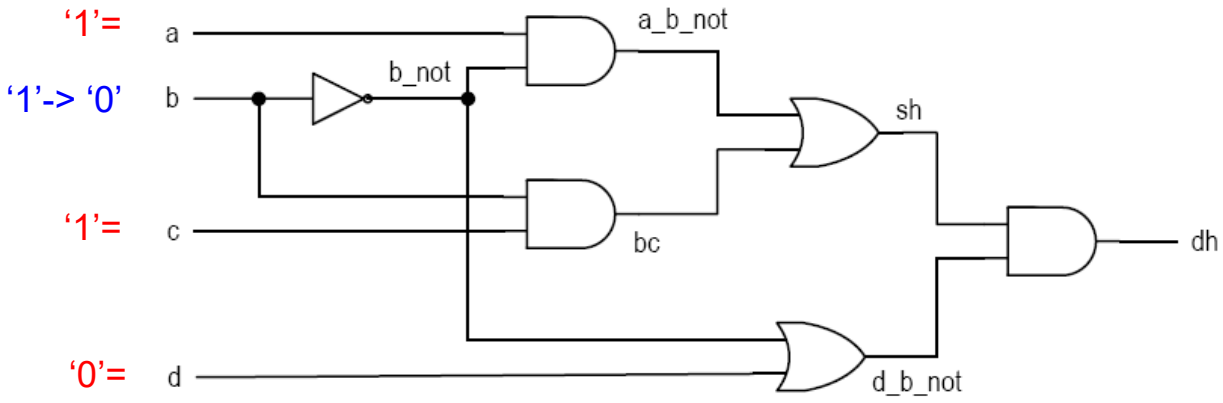
$$sh = a \cdot \bar{b} + b \cdot c$$

(a) Karnaugh map and schematic

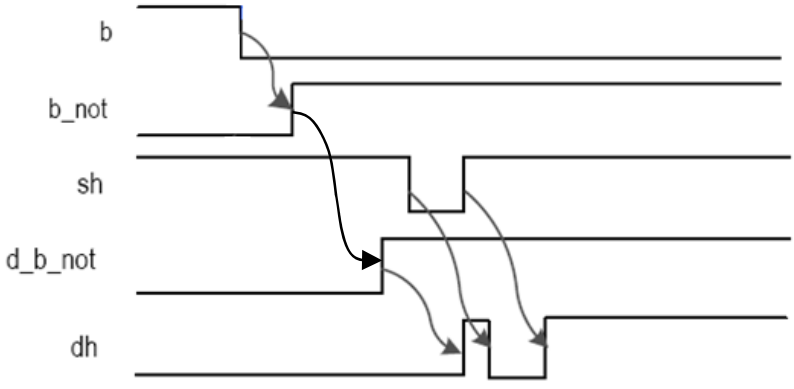


(b) Timing diagram

- E.g., dynamic hazard (a=c=1, d=0)



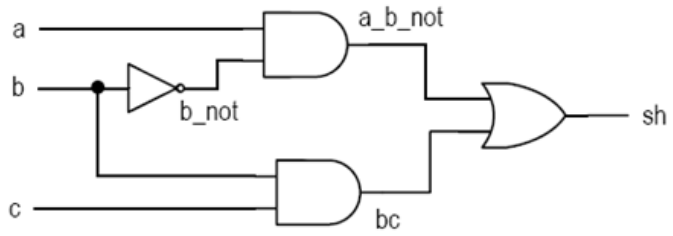
(a) Schematic

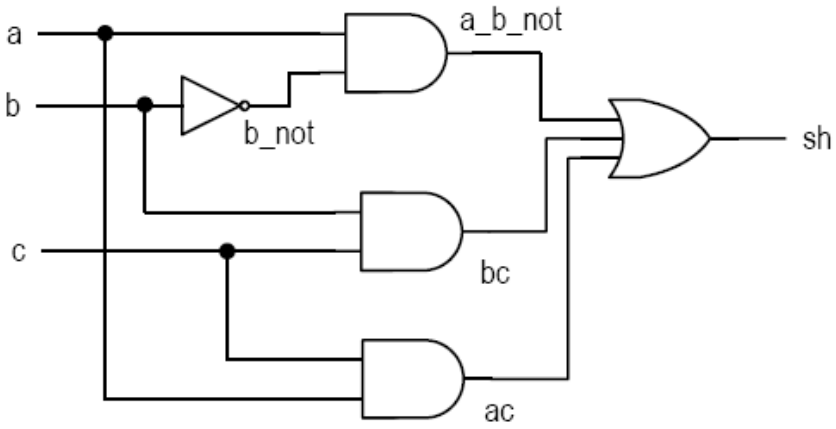


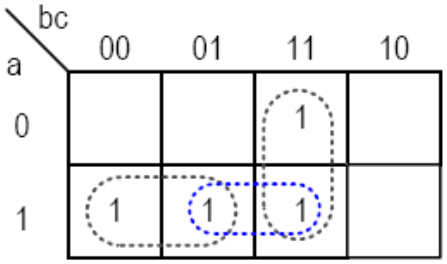
(b) Timing diagram

Dealing with hazards

- Some hazards can be eliminated in theory
- E.g.,







添加约束项

(c) Revised Karnaugh map and schematic to eliminate hazards

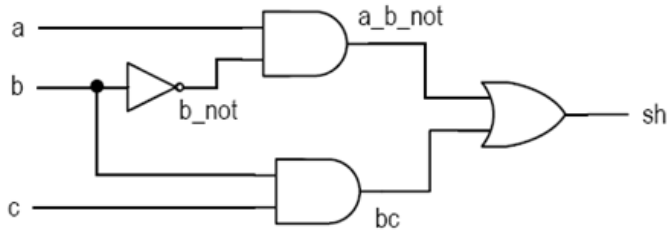
- Eliminating glitches is very difficult in reality, and almost impossible for synthesis
- Multiple inputs can change simultaneously (e.g., 1111=>0000 in a counter)
- How to deal with it?
- Ignore glitches in the transient period and retrieve the data after the signal is stabilized
- "Wait until the output is stabilized" – the motivation behind the synchronous design methodology.

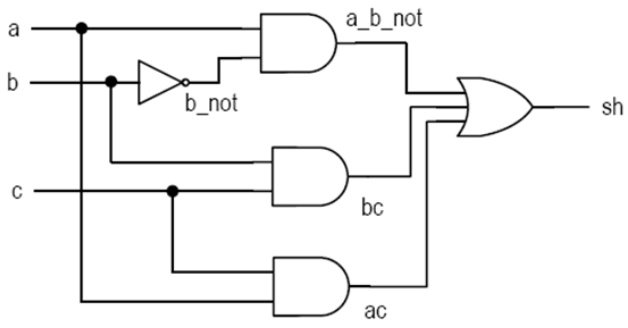
确保时钟周期足够长

Delay sensitive design and its danger

- Boolean algebra
  - the theoretical model for digital design and most algorithms used in synthesis process
  - algebra deals with the stabilized signals
- Delay-sensitive design
  - Depend on the transient property (and delay) of the circuit
  - Difficult to design and analyze

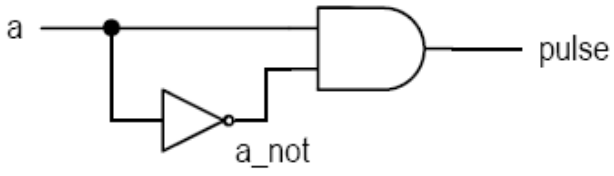
- E.g., hazard elimination circuit: ac term is not needed

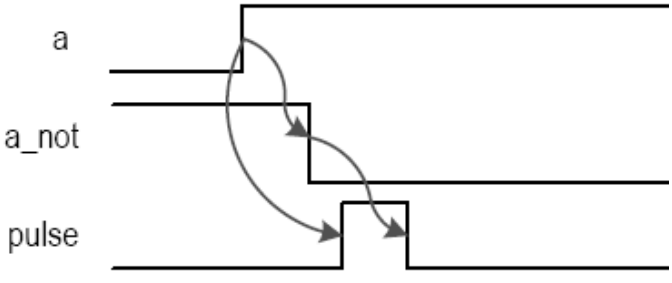




- E.g., edge detection circuit (pulse=a a')

ac这一项会在综合时候被自动去掉，因为冗余



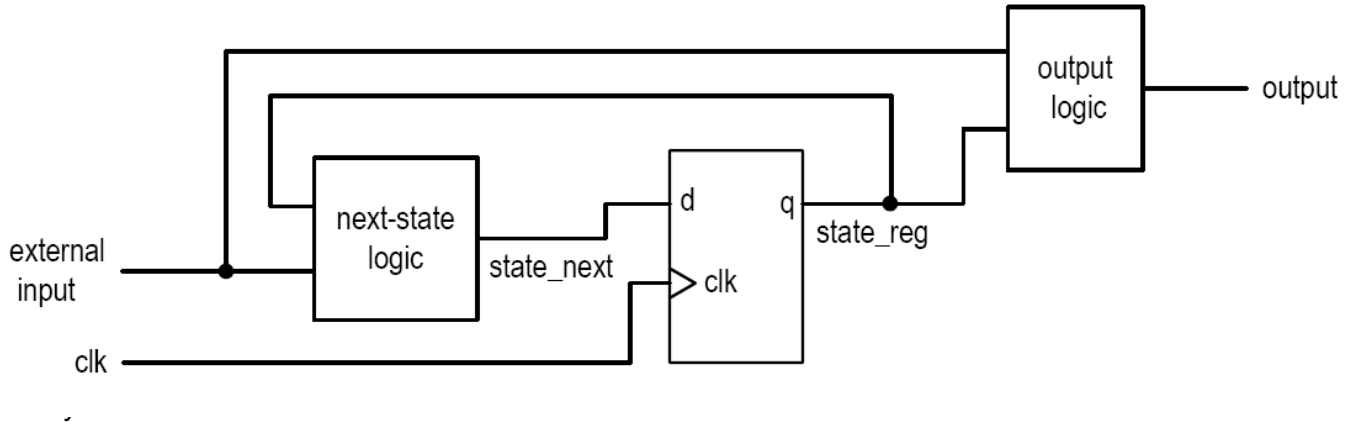


- What's can go wrong:
  - E.g., pulse <= a **and** (not a);
  - During logic synthesis, the logic expressions will be rearranged and optimized.
  - During technology mapping, generic gates will be re-mapped
  - During placement & routing, wire delays may change
  - It is bad for testing verification
- If delay-sensitive design is really needed, it should be done manually, not by synthesis

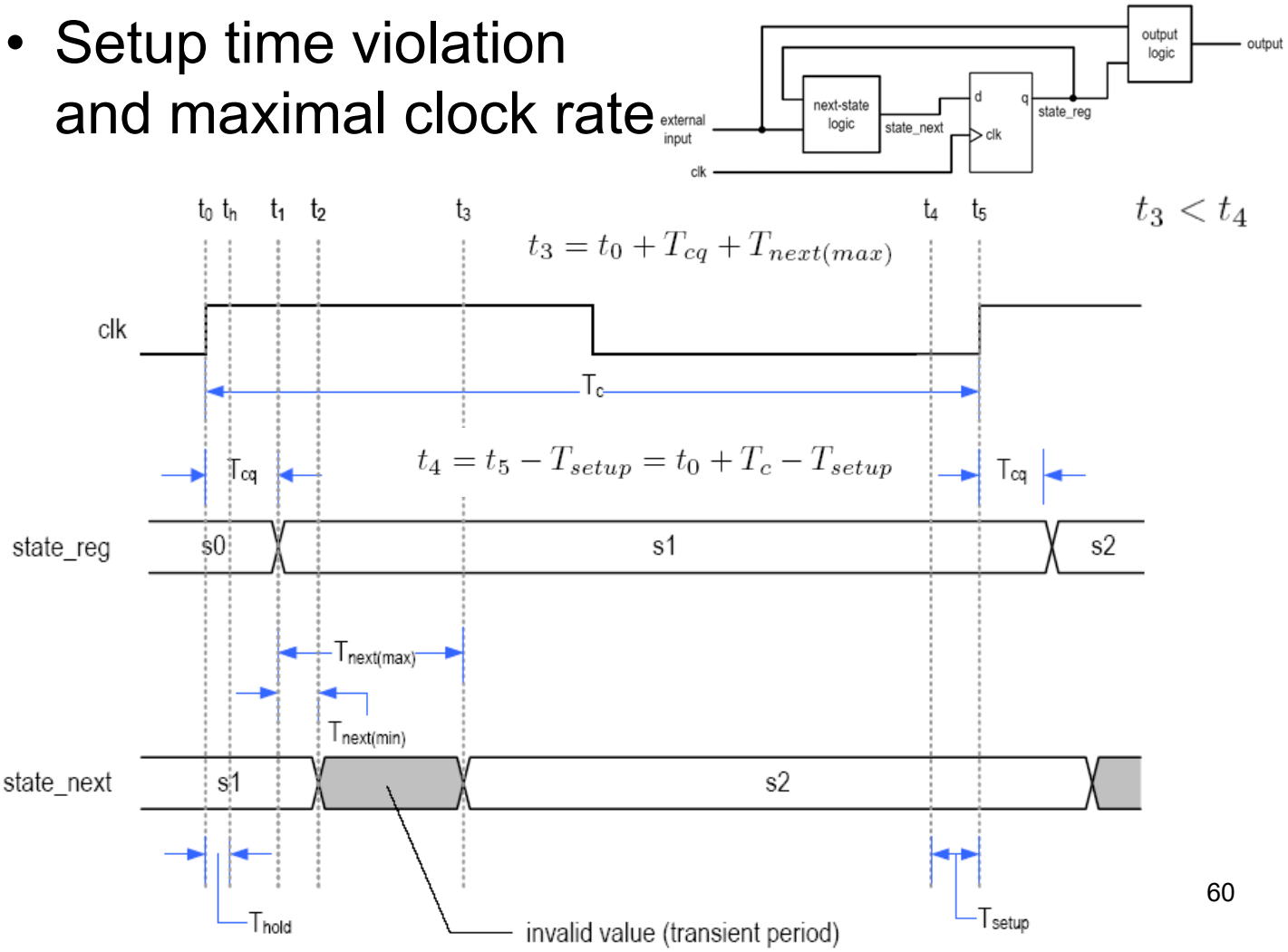
6. Sequential Circuit Timing analysis

- Combinational circuit:
  - characterized by propagation delay
- Sequential circuit:
  - Has to satisfy setup/hold time constraint
  - Characterized by maximal clock rate (e.g., 200 MHz counter, 2.4 GHz Pentium II)
  - Setup time and clock-to-q delay of register and the propagation delay of next-state logic are embedded in clock rate

- state_next must satisfy the constraint
- Must consider effect of
  - state_reg: can be controlled
  - synchronized external input (from a subsystem of same clock)
  - unsynchronized external input
- Approach
  - First 2: adjust clock rate to prevent violation
  - Last: use “synchronization circuit” to resolve violation



- Setup time violation and maximal clock rate



$$t_3 = t_0 + T_{cq} + T_{next(max)}$$

$$t_4 = t_5 - T_{setup} = t_0 + T_c - T_{setup}$$

$$t_3 < t_4$$

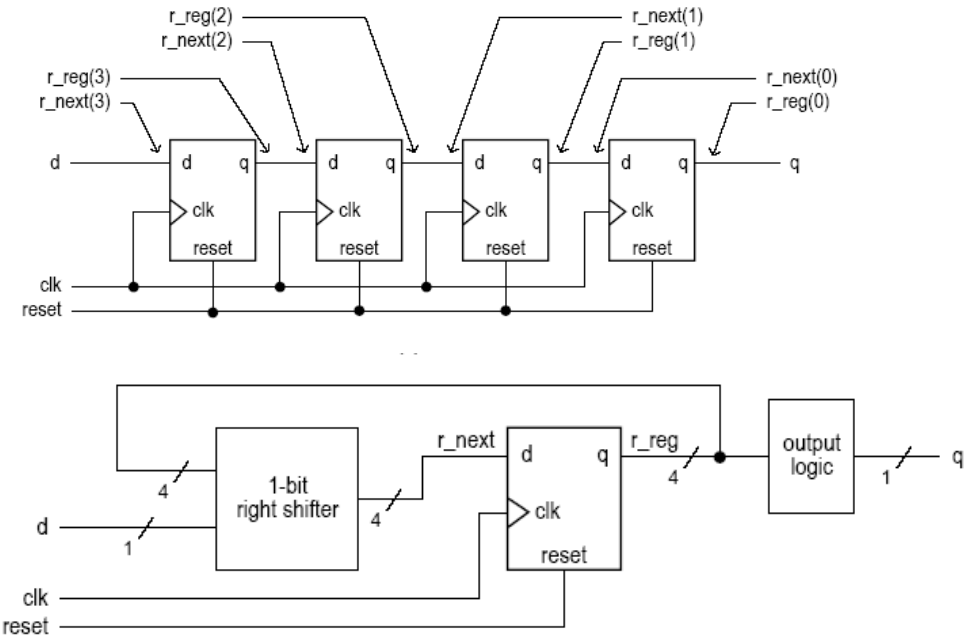
$$t_0 + T_{cq} + T_{next(max)} < t_0 + T_c - T_{setup}$$

$$T_{cq} + T_{next(max)} + T_{setup} < T_c$$

$$T_{c(min)} = T_{cq} + T_{next(max)} + T_{setup}$$

寄存器延时+next state logic延时+setup time

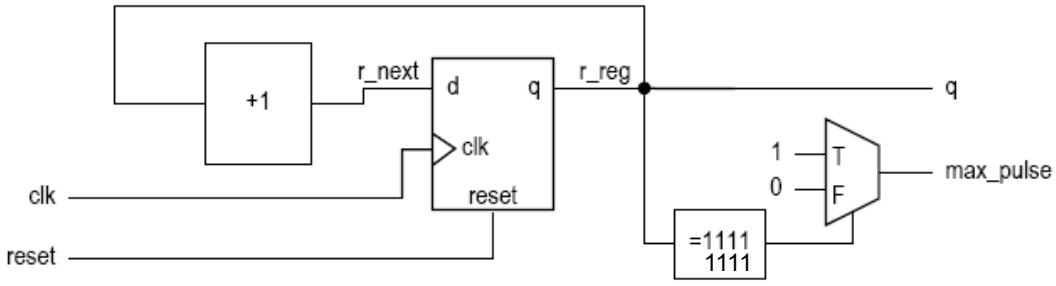
- E.g., shift register; let T_{cq} = 1.0ns T_{setup} = 0.5ns



$$T_{c(min)} = T_{cq} + T_{setup} = 1.5 \text{ ns}$$

$$f_{max} = \frac{1}{T_{cq} + T_{setup}} = \frac{1}{1.5 \text{ ns}} \approx 666.7 \text{ MHz}$$

- E.g., Binary counter; let T_{cq} = 1.0ns T_{setup} = 0.5ns



width	VHDL operator									
	nand	xor	$>_a$	$>_d$	=	$+1_a$	$+1_d$	$+_a$	$+_d$	mux
area (gate count)										
8	8	22	25	68	26	27	33	51	118	21
16	16	44	52	102	51	55	73	101	265	42
32	32	85	105	211	102	113	153	203	437	85
64	64	171	212	398	204	227	313	405	755	171
delay (ns)										
8	0.1	0.4	4.0	1.9	1.0	2.4	1.5	4.2	3.2	0.3
16	0.1	0.4	8.6	3.7	1.7	5.5	3.3	8.2	5.5	0.3
32	0.1	0.4	17.6	6.7	1.8	11.6	7.5	16.2	11.1	0.3
64	0.1	0.4	35.7	14.3	2.2	24.0	15.7	32.2	22.9	0.3

$$f_{max} = \frac{1}{T_{cq} + T_{8_bit_inc(area)} + T_{setup}} = \frac{1}{1\text{ ns} + 2.4\text{ ns} + 0.5\text{ ns}} \approx 256.4\text{ MHz}$$

$$f_{max} = \frac{1}{T_{cq} + T_{16_bit_inc(area)} + T_{setup}} = \frac{1}{1\text{ ns} + 5.5\text{ ns} + 0.5\text{ ns}} \approx 142.9\text{ MHz}$$

$$f_{max} = \frac{1}{T_{cq} + T_{32_bit_inc(area)} + T_{setup}} = \frac{1}{1\text{ ns} + 11.6\text{ ns} + 0.5\text{ ns}} \approx 76.3\text{ MHz}$$

还是要从组合逻辑提高电路性能

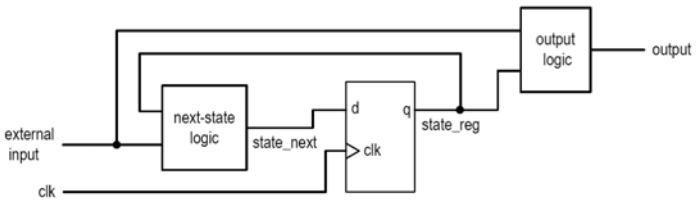
$$f_{max} = \frac{1}{T_{cq} + T_{8_bit_inc(delay)} + T_{setup}} = \frac{1}{1\text{ ns} + 1.5\text{ ns} + 0.5\text{ ns}} \approx 333.3\text{ MHz}$$

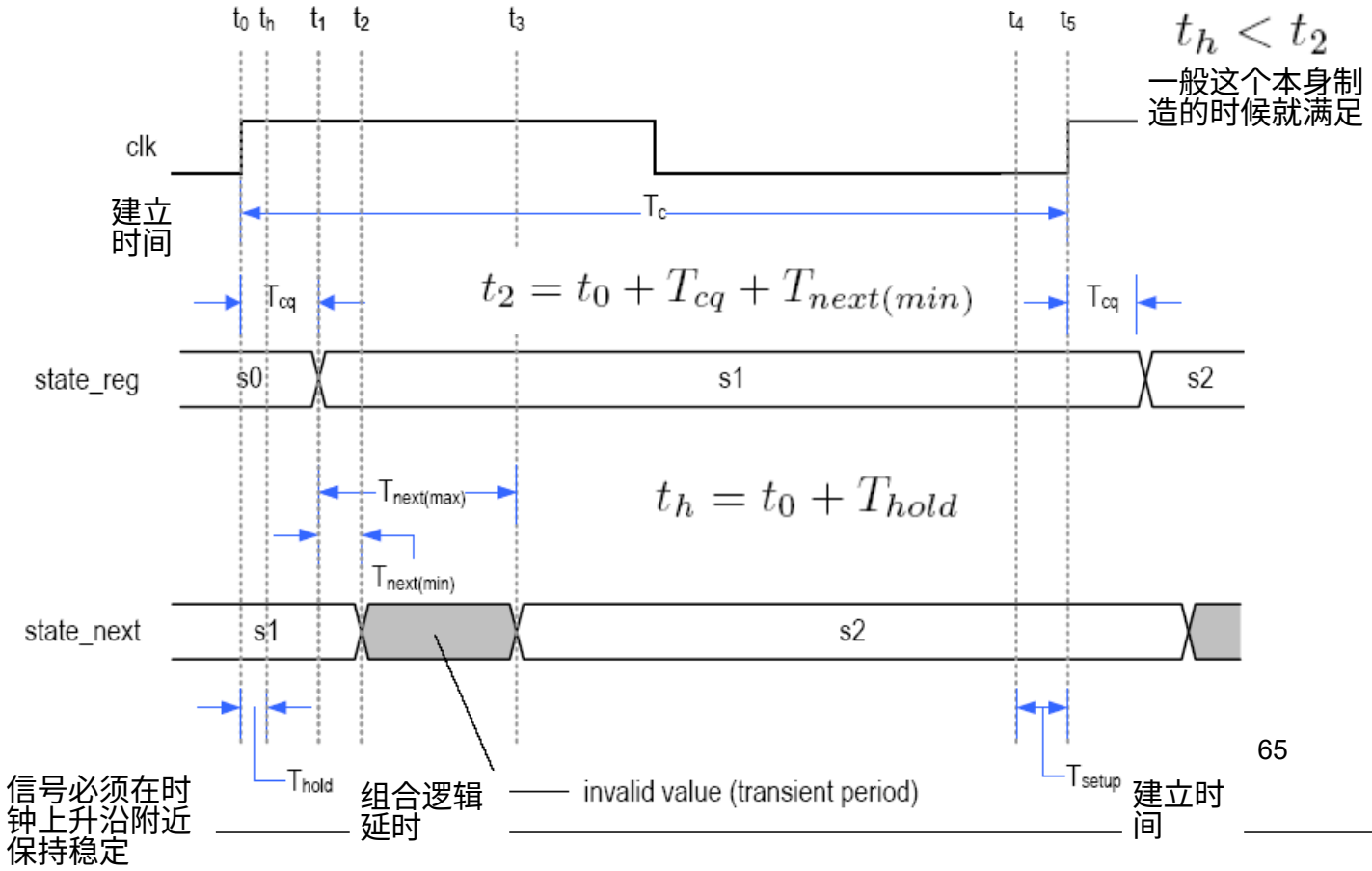
$$f_{max} = \frac{1}{T_{cq} + T_{16_bit_inc(delay)} + T_{setup}} = \frac{1}{1\text{ ns} + 3.3\text{ ns} + 0.5\text{ ns}} \approx 208.3\text{ MHz}$$

and

$$f_{max} = \frac{1}{T_{cq} + T_{32_bit_inc(delay)} + T_{setup}} = \frac{1}{1\text{ ns} + 7.5\text{ ns} + 0.5\text{ ns}} \approx 111.1\text{ MHz}$$

- Hold time violation





$t_h < t_2$

一般这个本身制造的时候就满足

65

$$t_2 = t_0 + T_{cq} + T_{next(min)}$$

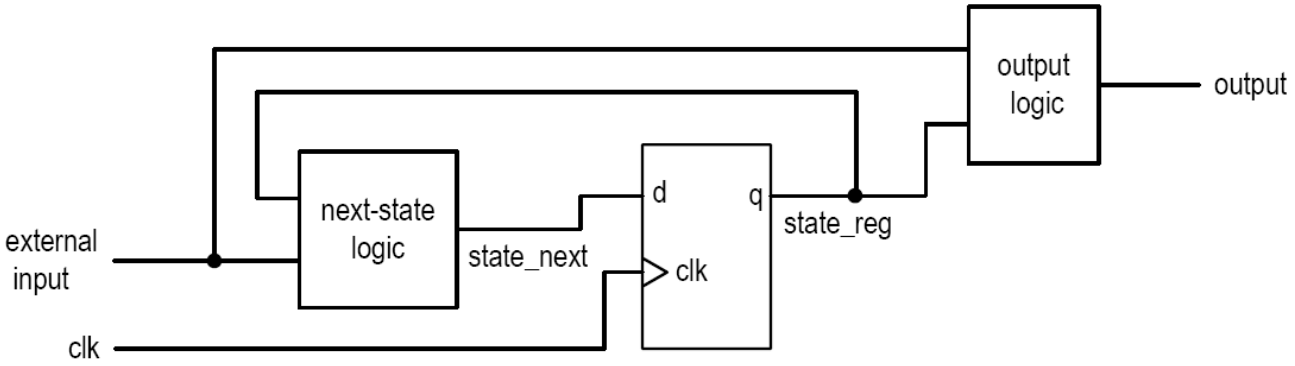
$$t_h = t_0 + T_{hold}$$

$$t_h < t_2$$

$$T_{hold} < T_{cq} + T_{next(min)}$$

$$T_{hold} < T_{cq}$$

Output delay



$$T_{co} = T_{cq} + T_{output}$$

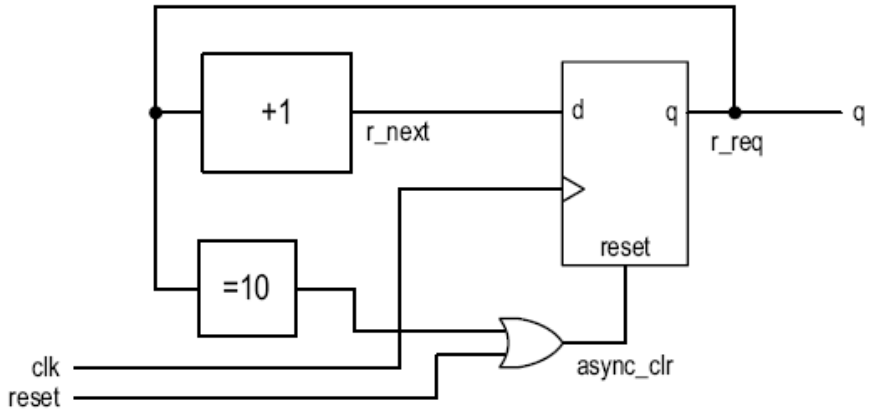
7. Poor design practice and remedy

- Synchronous design is the most important methodology
- Poor practice in the past (to save chips)
  - Misuse of asynchronous reset
  - Misuse of gated clock
  - Misuse of derived clock

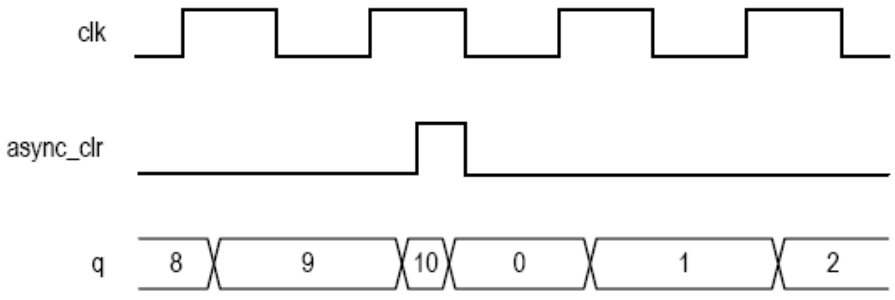
Misuse of asynchronous reset

- Poor design: use reset to clear register in normal operation.
- e.g., a poorly mod-10 counter
  - Clear register immediately after the counter reaches 1010

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity mod10_counter is
   port(
      clk, reset: in std_logic;
      q: out std_logic_vector(3 downto 0)
      );
end mod10_counter;
```

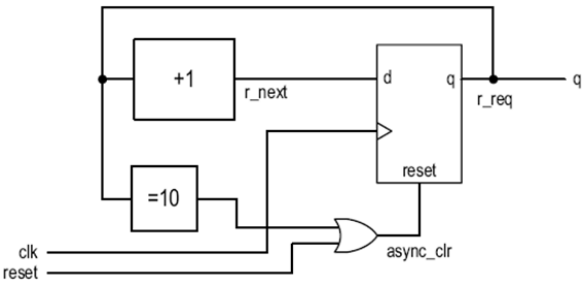
(a) Block diagram



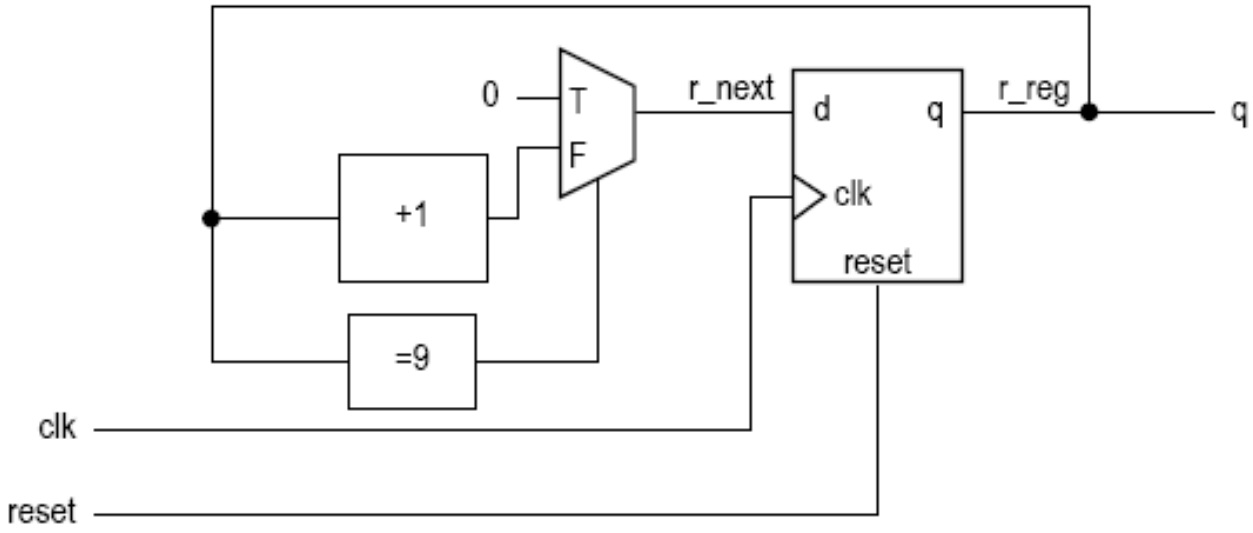
(b) Timing diagram

Figure 9.1 Decade counter using asynchronous reset

```
architecture poor_async_arch of mod10_counter is
   signal r_reg: unsigned(3 downto 0);
   signal r_next: unsigned(3 downto 0);
   signal async_clr: std_logic;
begin
   -- register
   process (clk, async_clr)
   begin
      if (async_clr='1') then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
      end if;
   end process;
   -- asynchronous clear
   async_clr <= '1' when (reset='1' or r_reg="1010") else
                '0';
   -- next state logic
   r_next <= r_reg + 1;
   -- output logic
   q <= std_logic_vector(r_reg);
end poor_async_arch;
```

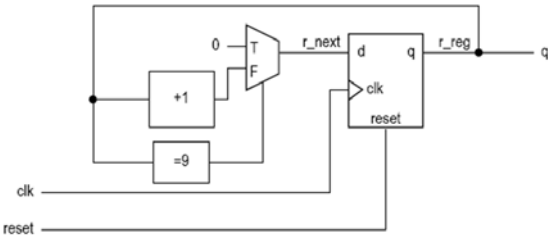


- Problem
  - Glitches in transition 1001 (9) => 0000 (0)
  - Glitches in aync_clr can reset the counter 毛刺导致误reset
  - How about timing analysis? (maximal clock rate)
- Asynchronous reset should only be used for power-on initialization



- Remedy: load “0000” synchronously

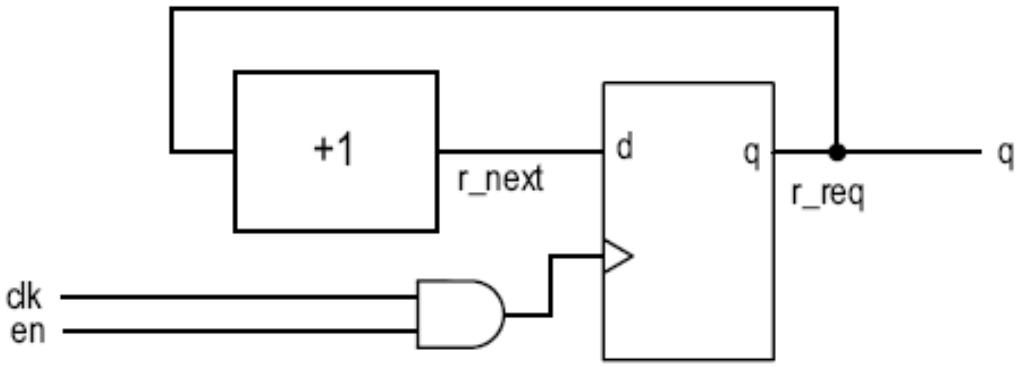
```
architecture two_seg_arch of mod10_counter is
   signal r_reg: unsigned(3 downto 0);
   signal r_next: unsigned(3 downto 0);
begin
   -- register
   process (clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
      end if;
   end process;
   -- next state logic
   r_next <= (others=>'0') when r_reg=9 else
             r_reg + 1;
   -- output logic
   q <= std_logic_vector(r_reg);
end two_seg_arch;
```



Misuse of gated clock

- Poor design: use a and gate to disable the clock to stop the register to get new value
- E.g., a counter with an enable signal

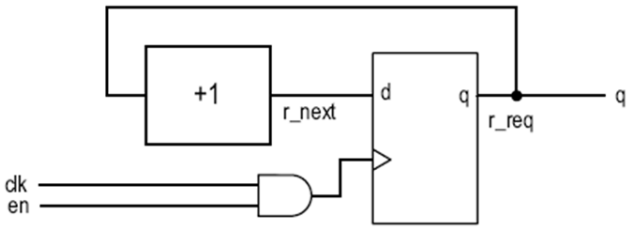
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity binary_counter is
   port(
      clk, reset: in std_logic;
      en: in std_logic;
      q: out std_logic_vector(3 downto 0)
      );
end binary_counter;
```

时钟的宽度可能会被影响
以及产生毛刺现象
可能组合逻辑出来的不被软件判断为时钟信号
时钟震荡会消耗很多能量

Figure 9.2 Disabling FF with gated clock

```
architecture gated_clk_arch of binary_counter is
   signal r_reg: unsigned(3 downto 0);
   signal r_next: unsigned(3 downto 0);
   signal gated_clk: std_logic;
begin
   -- register
   process (gated_clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif (gated_clk'event and gated_clk = '1') then
         r_reg <= r_next;
      end if;
   end process;
   -- gated clock
   gated_clk <= clk and en;
   -- next state logic
   r_next <= r_reg + 1;
   -- output logic
   q <= std_logic_vector(r_reg);
end gated_clk_arch;
```



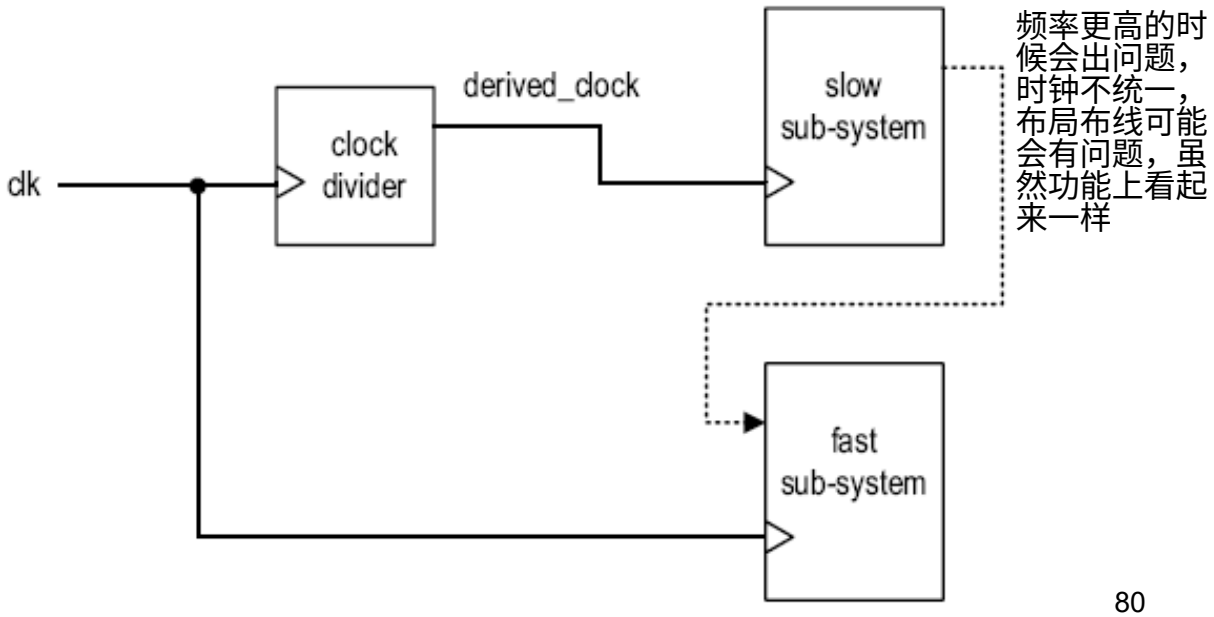
- Problem
  - Gated clock width can be narrow
  - Gated clock may pass glitches of en
  - Difficult to design the clock distribution network

- Remedy: use a synchronous enable

```
architecture two_seg_arch of binary_counter is
   signal r_reg: unsigned(3 downto 0);
   signal r_next: unsigned(3 downto 0);
begin
   -- register
   process (clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
      end if;
   end process;
   -- next state logic
   r_next <= r_reg + 1 when en='1' else
             r_reg;
   -- output logic
   q <= std_logic_vector(r_reg);
end two_seg_arch;
```

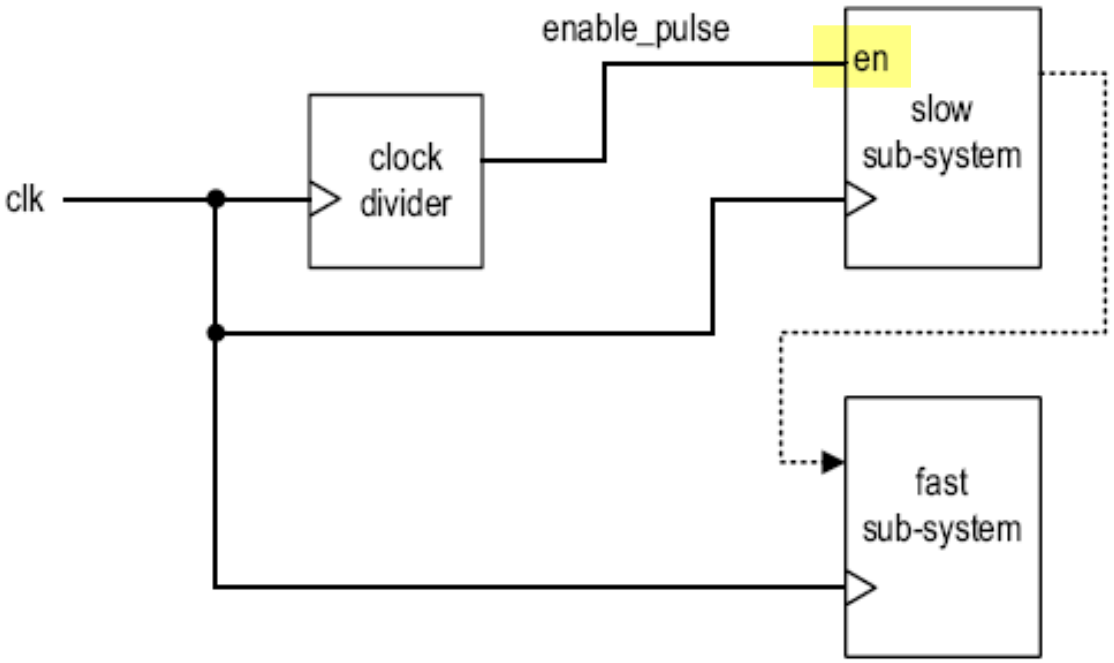

Misuse of derived clock

- Subsystems may run at different clock rate
- Poor design: use a derived slow clock for slow subsystem

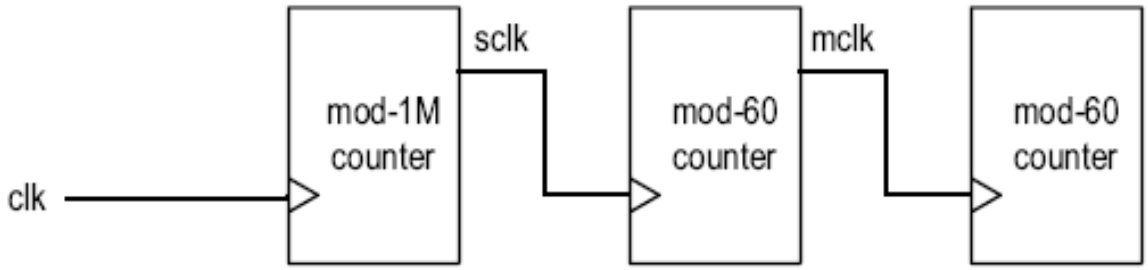


- Problem
  - Multiple clock distribution network
  - How about timing analysis? (maximal clock rate)

- Better use a synchronous one-clock enable pulse

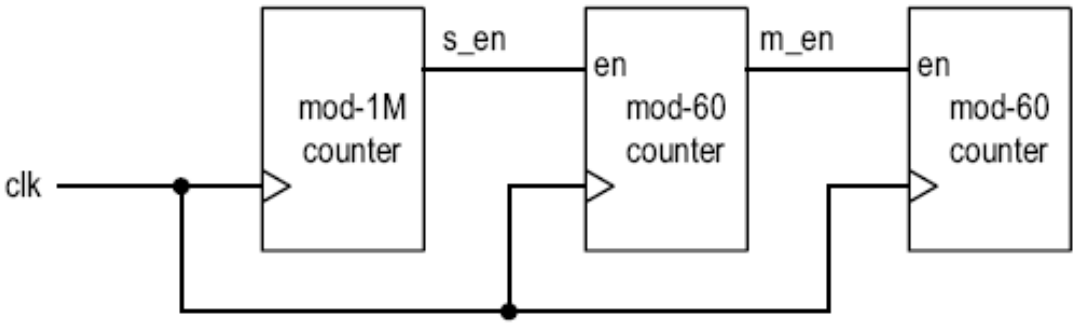


- E.g., second and minutes counter
  - Input: 1 MHz clock
  - Poor design:



(a) Design with derived clock

  - Better design



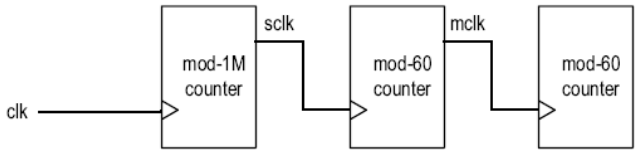
(b) Design with a single synchronous clock

- VHDL code of poor design

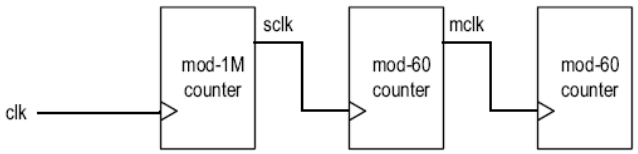
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity timer is
   port(
      clk, reset: in std_logic;
      sec,min: out std_logic_vector(5 downto 0)
      );
end timer;

architecture multi_clock_arch of timer is
   signal r_reg: unsigned(19 downto 0);
   signal r_next: unsigned(19 downto 0);
   signal s_reg, m_reg: unsigned(5 downto 0);
   signal s_next, m_next: unsigned(5 downto 0);
   signal sclk, mclk: std_logic;
begin
```



```
-- register
process (clk, reset)
begin
   if (reset='1') then
      r_reg <= (others=>'0');
   elsif (clk'event and clk='1') then
      r_reg <= r_next;
   end if;
end process;
-- next state logic
r_next <= (others=>'0') when r_reg=99999 else
          r_reg + 1;
-- output logic
sclk <= '0' when r_reg < 500000 else
        '1';
```

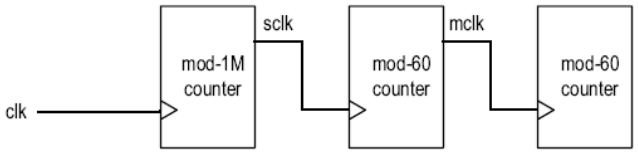


(a) Design with derived clock



(a) Design with derived clock

```
   -- second divider
   process (sclk, reset)
   begin
      if (reset='1') then
         s_reg <= (others=>'0');
      elsif (sclk'event and sclk='1') then
         s_reg <= s_next;
      end if;
   end process;
   -- next state logic
   s_next <= (others=>'0') when s_reg=59 else
             s_reg + 1;
   -- output logic
   mclk <= '0' when s_reg < 30 else
           '1';
   sec <= std_logic_vector(s_reg);
```

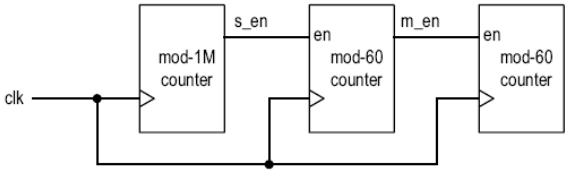



(a) Design with derived clock

```
   -- minute divider
   process (mclk, reset)
   begin
      if (reset='1') then
         m_reg <= (others=>'0');
      elsif (mclk'event and mclk='1') then
         m_reg <= m_next;
      end if;
   end process;
   -- next state logic
   m_next <= (others=>'0') when m_reg=59 else
             m_reg + 1;
   -- output logic
   min <= std_logic_vector(m_reg);
end multi_clock_arch;
```

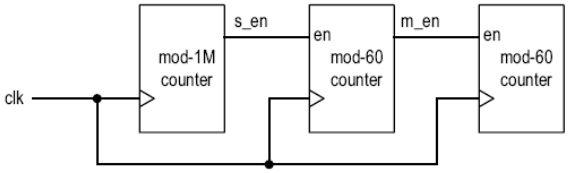
- Remedy: use a synchronous 1-clock pulse

```
architecture single_clock_arch of timer is
   signal r_reg: unsigned(3 downto 0);
   signal r_next: unsigned(3 downto 0);
   signal s_reg, m_reg: unsigned(5 downto 0);
   signal s_next, m_next: unsigned(5 downto 0);
   signal s_en, m_en: std_logic;
begin
   -- register
   process (clk, reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
         s_reg <= (others=>'0');
         m_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
         s_reg <= s_next;
         m_reg <= m_next;
      end if;
   end process;
```



(b) Design with a single synchronous clock


```
   -- next state logic/output logic for mod-1000000 counter
   r_next <= (others=>'0') when r_reg=999999 else
             r_reg + 1;
   s_en <= '1' when r_reg = 500000 else
           '0';
   -- ext state logic/output logic for second divider
   s_next <= (others=>'0') when (s_reg=59 and s_en='1') else
             s_reg + 1     when s_en='1' else
             s_reg;
   m_en <= '1' when s_reg=30 and s_en='1' else
           '0';
   -- next state logic for minute divider
   m_next <= (others=>'0') when (m_reg=59 and m_en='1') else
             m_reg + 1     when m_en='1' else
             m_reg;
   -- output logic
   sec <= std_logic_vector(s_reg);
   min <= std_logic_vector(m_reg);
end single_clock_arch;
```



(b) Design with a single synchronous clock

A word about power

- Power is a major design criteria now
- In CMOS technology
  - Dynamic power is proportional to the switching frequency of transistors
  - High clock rate implies high switching freq
- Clock manipulation
  - Can reduce switching frequency
  - But should not be done at RT level

- Development flow:
  1. Design/synthesize/verify a regular synchronous subsystems

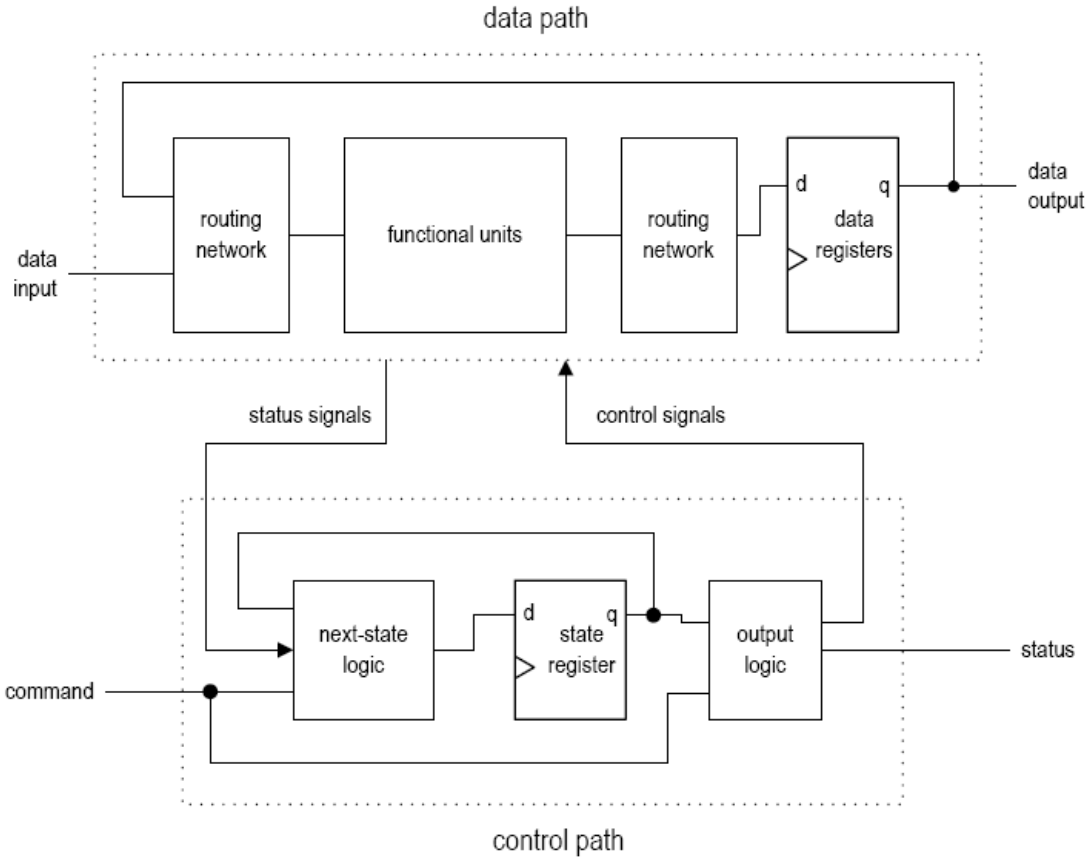
  2(a). Derived clock: use special circuit (PLL etc.) to obtain derived clocks

  2(b). Gated clock: use “power optimization” software tool to convert some register into gated clock

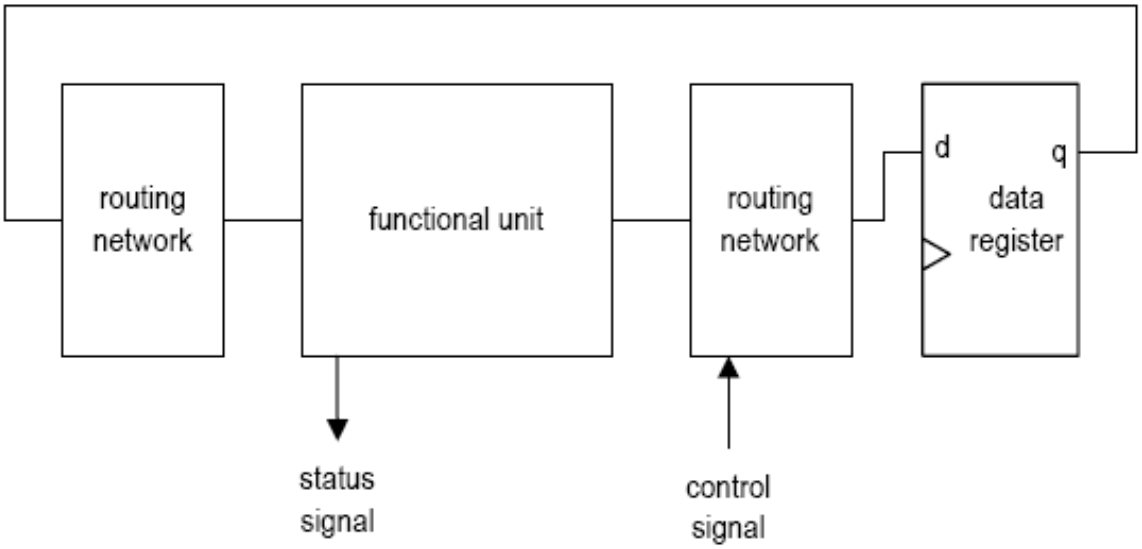
8. Clock rate and Performance of FSMD

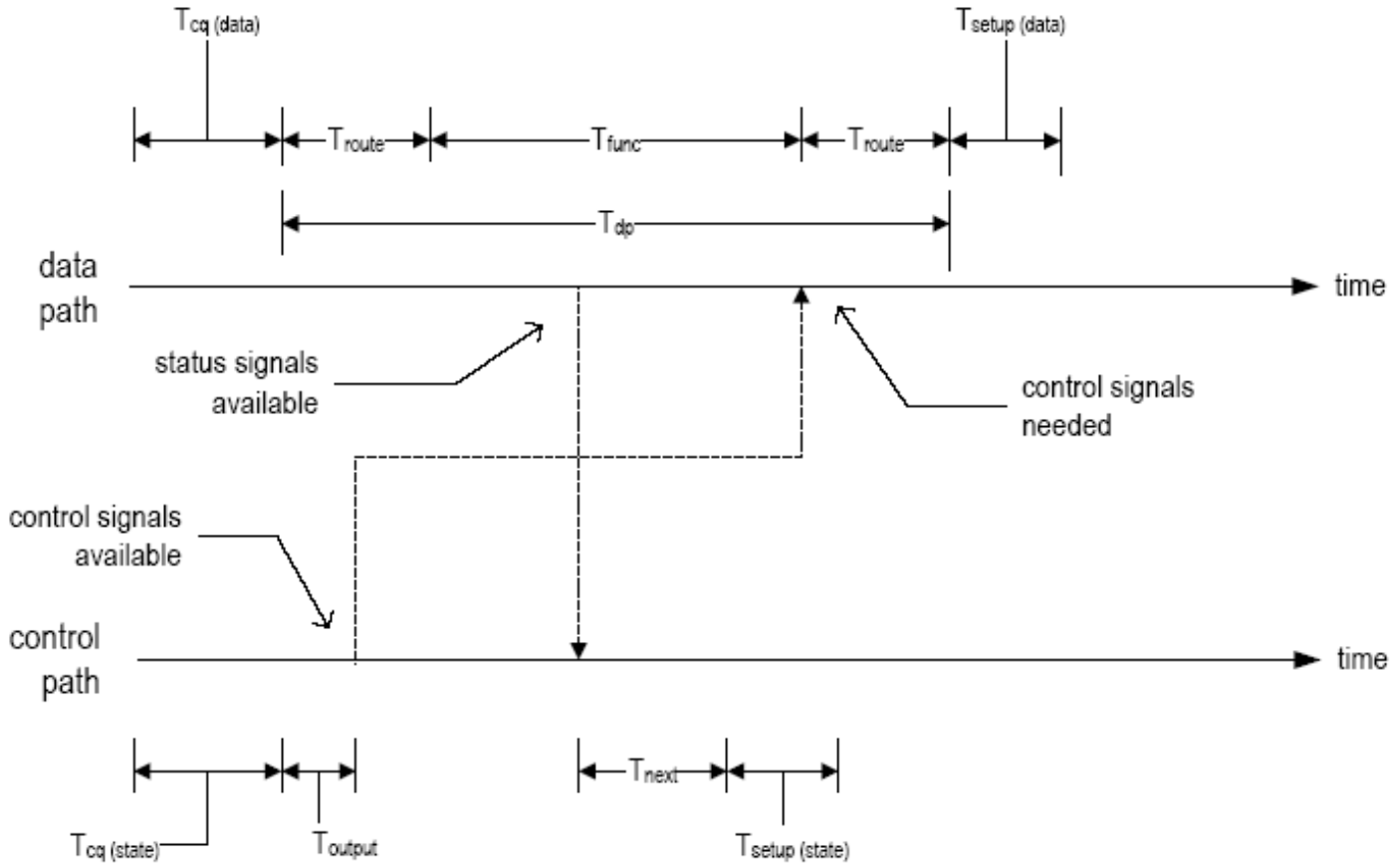
- Maximal clock rate
  - More difficult to analyze because of two interactive loops
  - The boundary of the clock rate can be found

Basic Block Diagram of FSMD



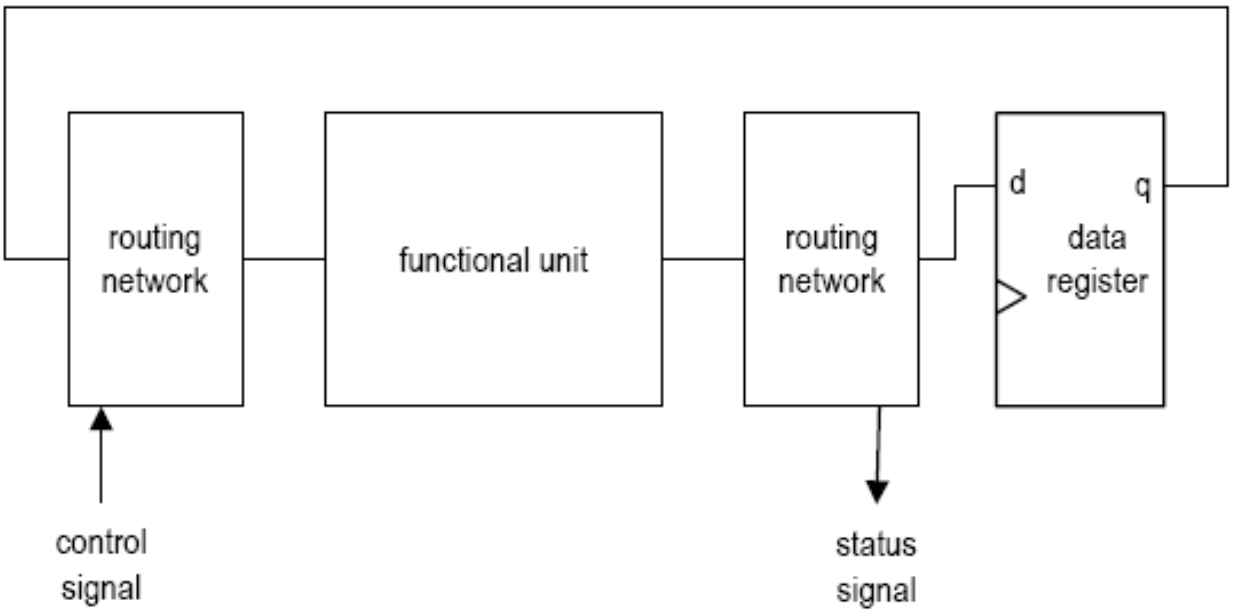
- Best-case scenario:
  - Control signals needed at late stage
  - Status signal available at early stage

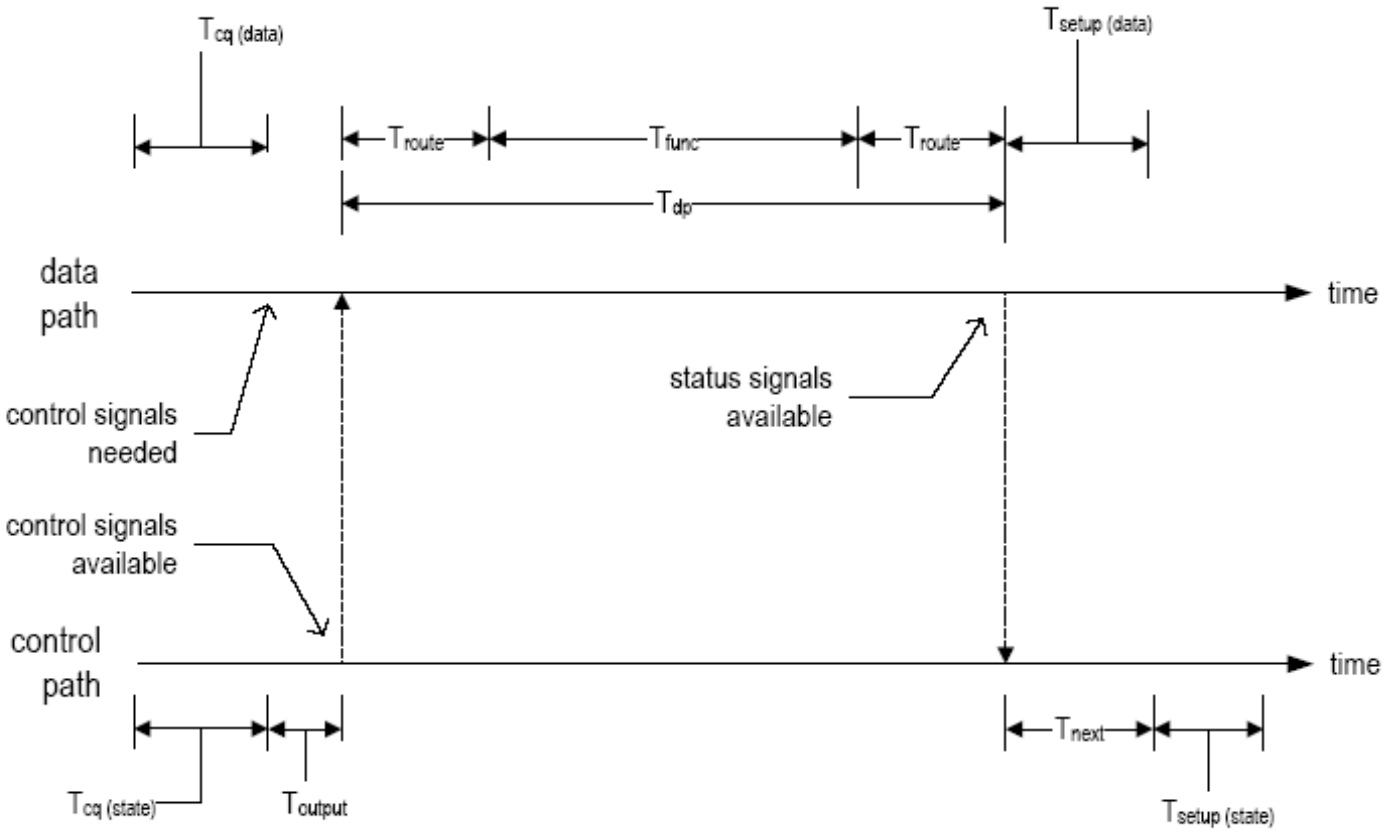




$$T_c = T_{cq(data)} + T_{dp} + T_{setup(data)}$$

- Worst-case scenario:
  - Control signals needed at early stage
  - Status signal available at late stage





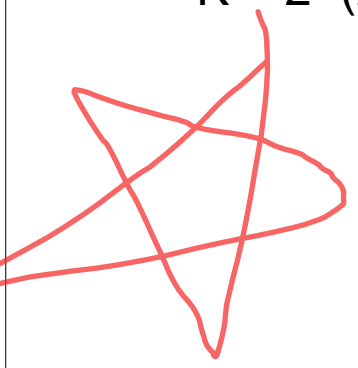
$$T_c = T_{cq(state)} + T_{output} + T_{dp} + T_{next} + T_{setup(state)}$$

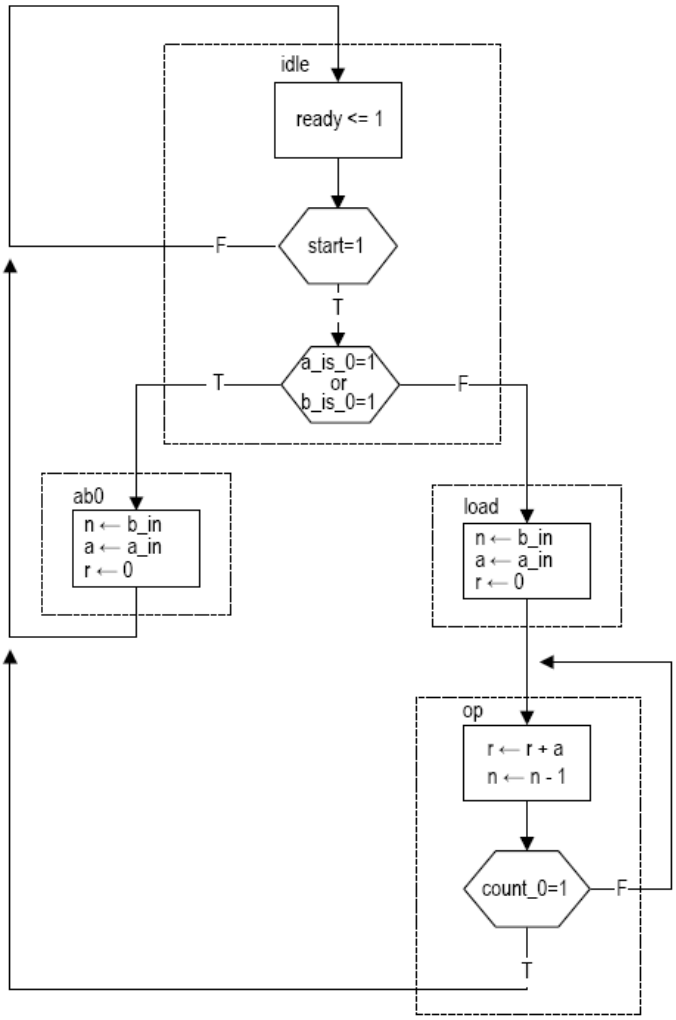
$$T_{cq} + T_{dp} + T_{setup} \leq T_c \leq T_{cq} + T_{output} + T_{dp} + T_{next} + T_{setup}$$

$$\frac{1}{T_{cq} + T_{output} + T_{dp} + T_{next} + T_{setup}} \leq f \leq \frac{1}{T_{cq} + T_{dp} + T_{setup}}$$

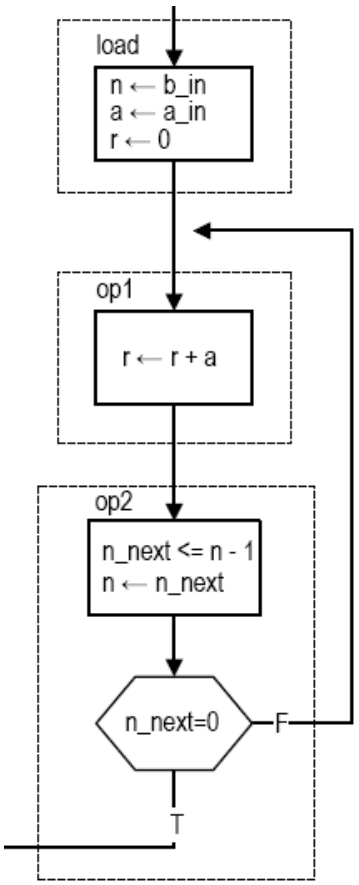
- Performance of FSMD
  - Tc: Clock period
  - K: # clock cycles to compete the computation
  - Total time = K * Tc
- K determined by algorithm, input patterns etc.

- 8-bit input
  - Best: b=0, K=2
  - Worst: b=255, K=257
- N-bit input:
  - Worst:
    K = 2+(2^n-1)





- 8-bit input
  - Best: b=0, K=2
  - Worst: b=255,
    K=2 + 255*2
- N-bit input:
  - Worst:
    K=2+2*(2^n-1)



Clock and Synchronization

Outline

1. Why synchronous?
2. Clock distribution network and skew
3. Multiple-clock system
4. Meta-stability and synchronization failure
5. Synchronizer
6. Enable tick crossing clock domain
7. Handshaking
8. Data transfer crossing clock domains

1. Why synchronous

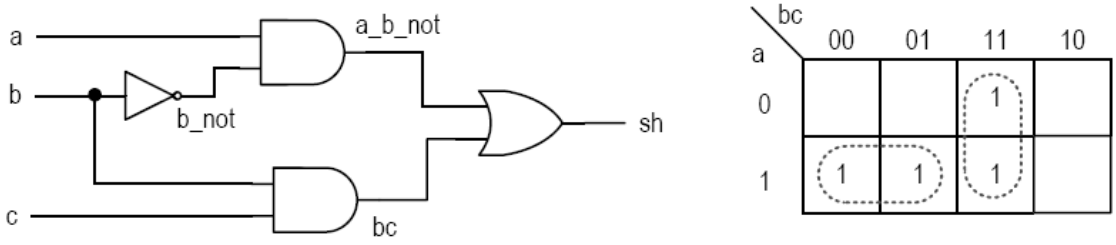
Timing of a combinational digital system

- Steady state
  - Signal reaches a stable value
  - Modeled by Boolean algebra
- Transient period
  - Signal may fluctuate
  - No simple model
- Propagation delay: time to reach the steady state

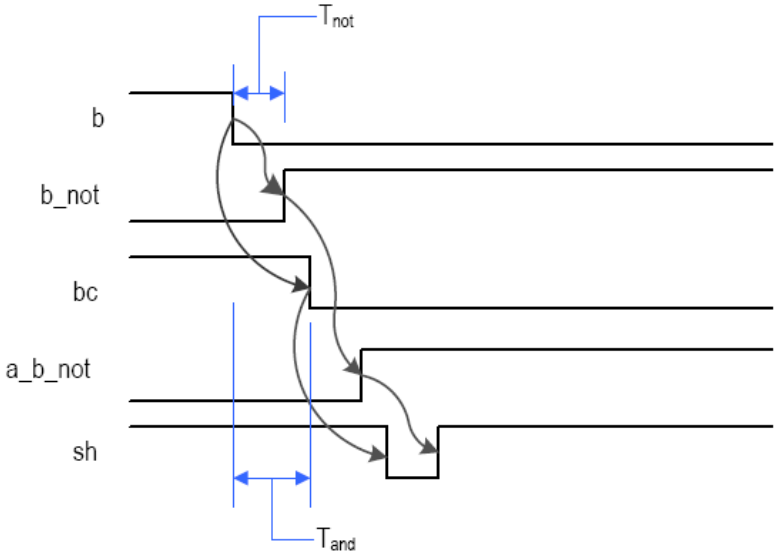
Timing Hazards

- Hazards: the fluctuation occurring during the transient period
  - Static hazard: glitch when the signal should be stable
  - Dynamic hazard: a glitch in transition
- Due to the multiple converging paths of an output port

- E.g., static-hazard (sh=ab'+bc; a=c=1)

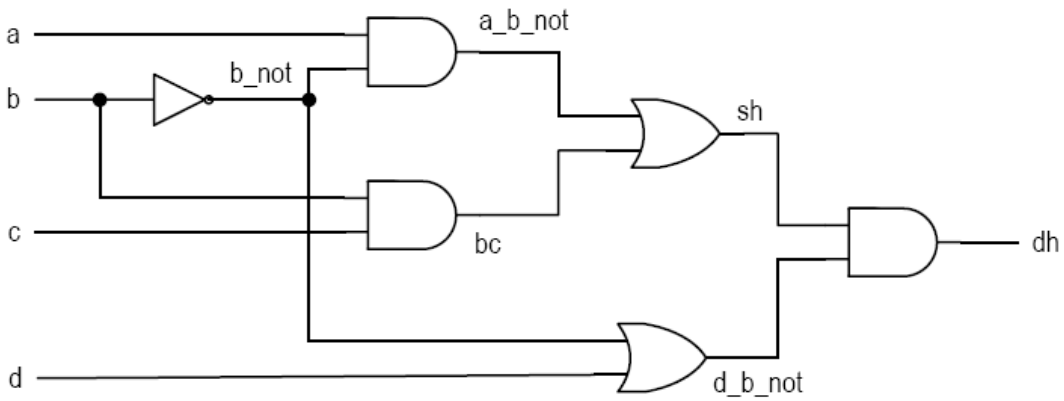


(a) Karnaugh map and schematic

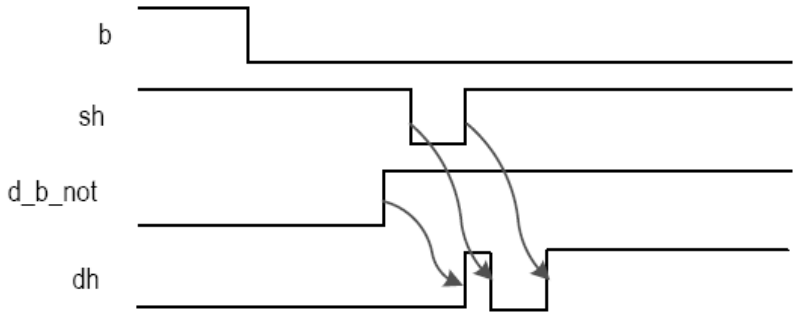


(b) Timing diagram

- E.g., dynamic hazard (a=c=1, d=0)

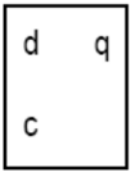


(a) Schematic



(b) Timing diagram

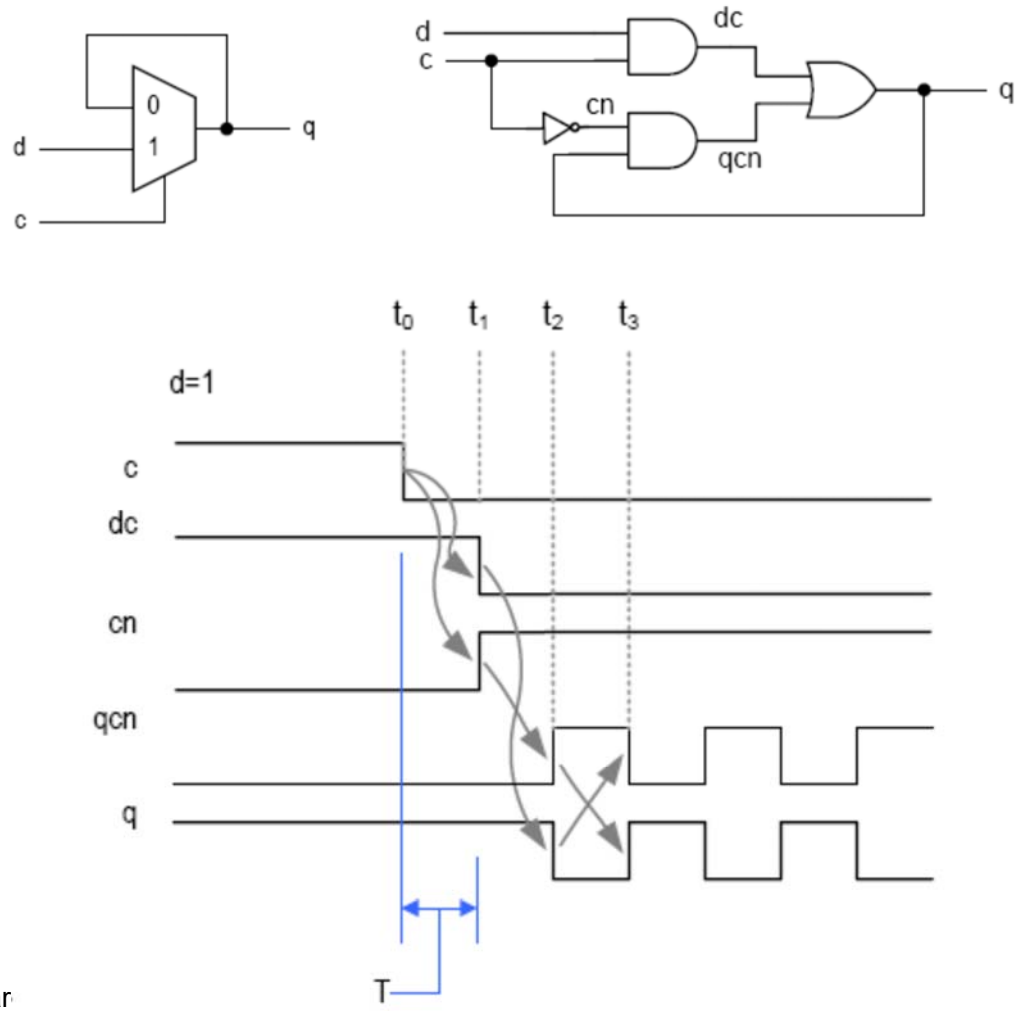
E.g., Hazard of circuit with closed feedback loop (async seq circuit)



c	q*
0	q
1	d

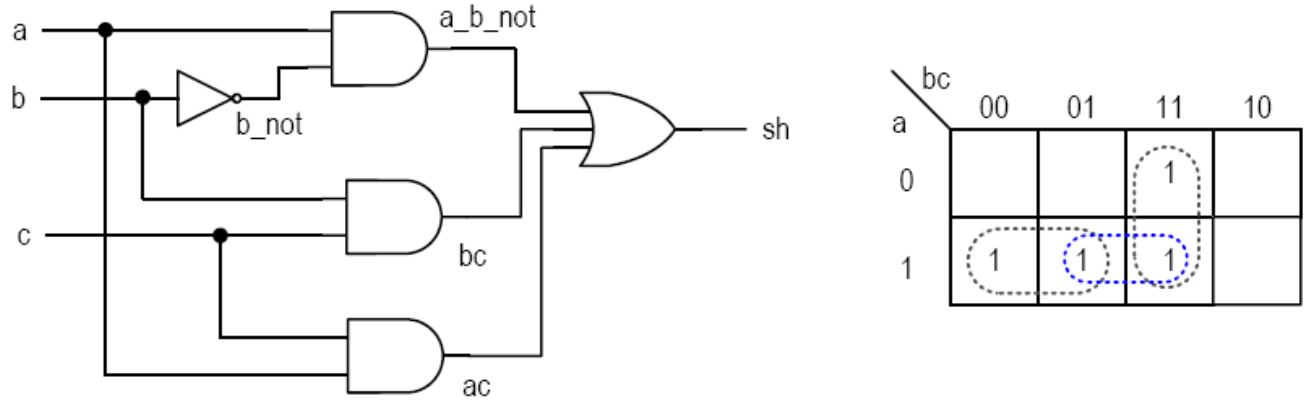
(a) D latch

```
library ieee;
use ieee.std_logic_1164.all;
entity dlatch is
   port(
      c: in std_logic;
      d: in std_logic;
      q: out std_logic
   );
end dlatch;

architecture demo_arch of dlatch i
   signal q_latch: std_logic;
begin
   process (c, d, q_latch)
   begin
      if (c='1') then
         q_latch <= d;
      else
         q_latch <= q_latch;
      end if;
   end process;
   q <= q_latch;
end demo_arch;
```



Dealing with hazards

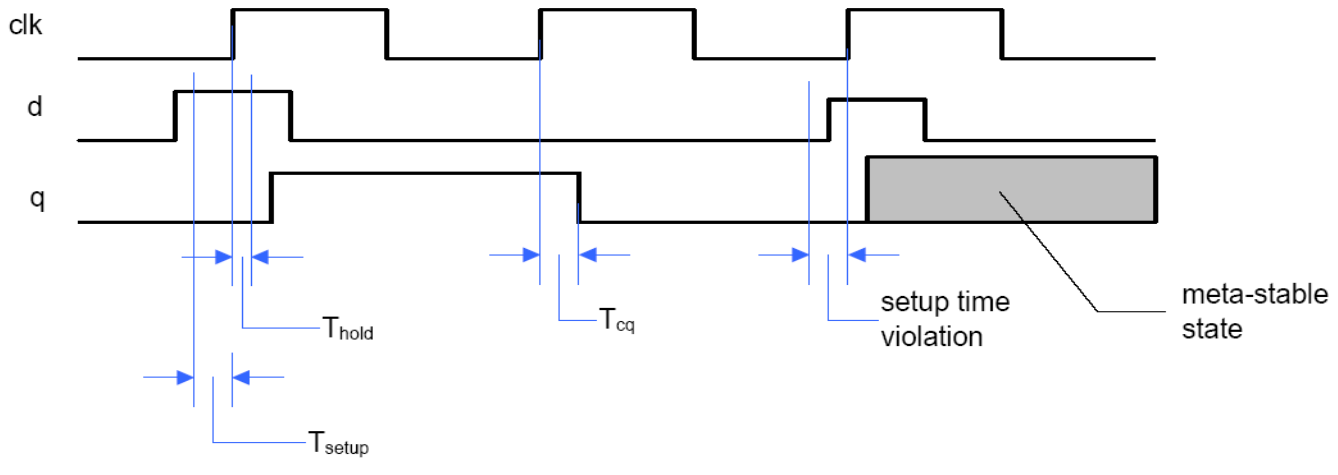
- In a small number of cases, additional logic can be added to eliminate race (and hazards).



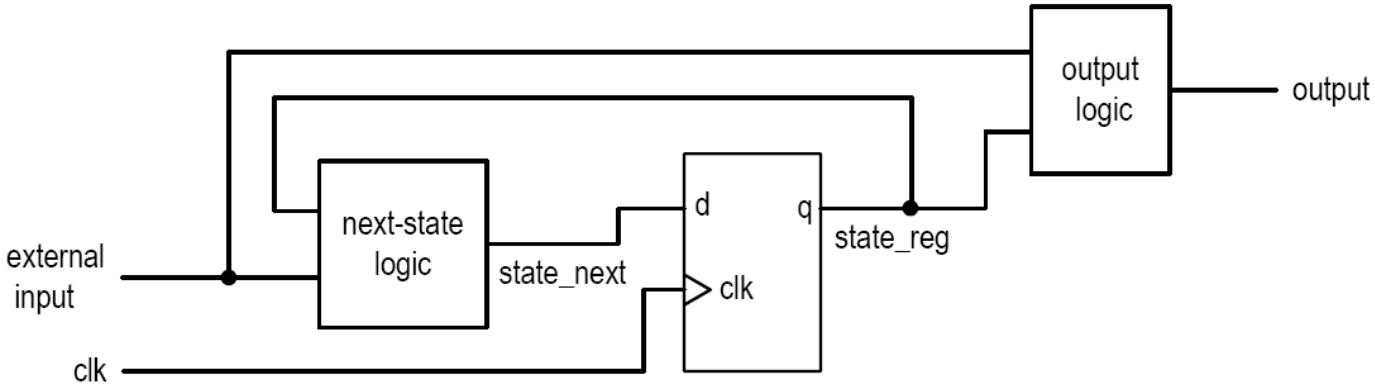
(c) Revised Karnaugh map and schematic to eliminate hazards

- This is not feasible for synthesis
- What's can go wrong:
  - During logic synthesis, the logic expressions will be rearranged and optimized.
  - During technology mapping, generic gates will be re-mapped
  - During placement & routing, wire delays may change
  - It is bad for testing verification

- Better way to handle hazards
  - Ignore glitches in the transient period and retrieve the data after the signal is stabilized
- In a sequential circuit
  - Use a clock signal to sample the signal and store the stable value in a register.
  - But register introduces new timing constraint (setup time and hold time)



- Synchronous system:
  - group registers into a single group and drive them with the same clock
  - Timing analysis for a single feedback loop



Synchronous circuit and EDA

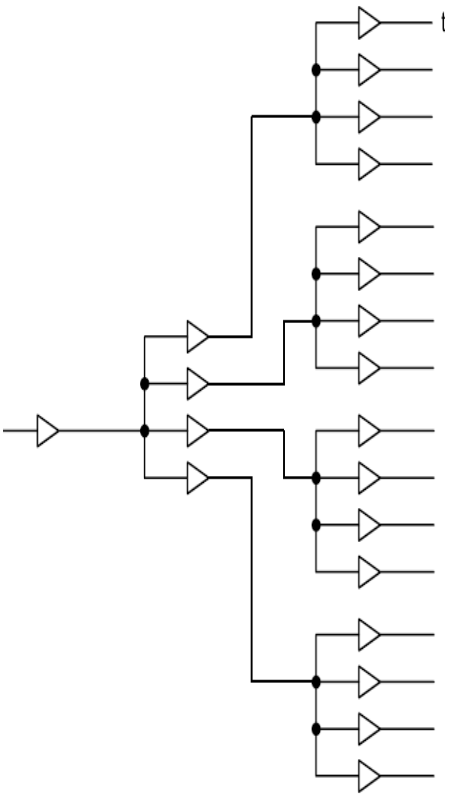
- Synthesis: reduce to combinational circuit synthesis
- Timing analysis: involve only a single closed feedback loop (others reduce to combinational circuit analysis)
- Simulation: support “cycle-based simulation”
- Testing: can facilitate scan-chain

2. Clock distribution network and skew

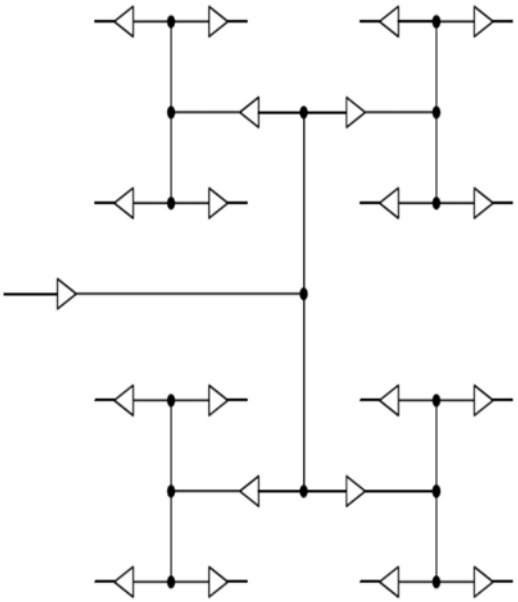
Clock distribution network

- Ideal clock: clock's rising edges arrive at FFs at the same time
- Real implementation:
  - Driving capability of each cell is limited
  - Need a network of buffers to drive all FFs
  - In ASIC: done by clock synthesis (a step in physical synthesis)
  - In FPGA: pre-fabricated clock distribution network

- Block diagram

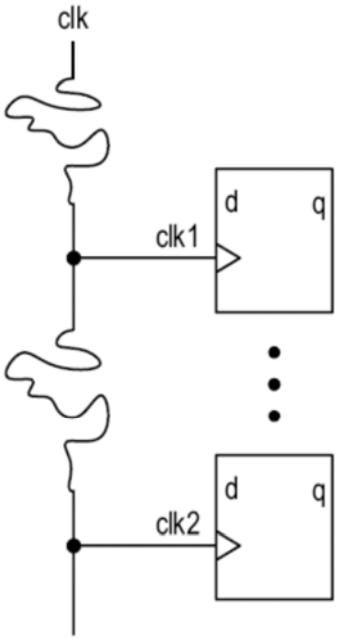


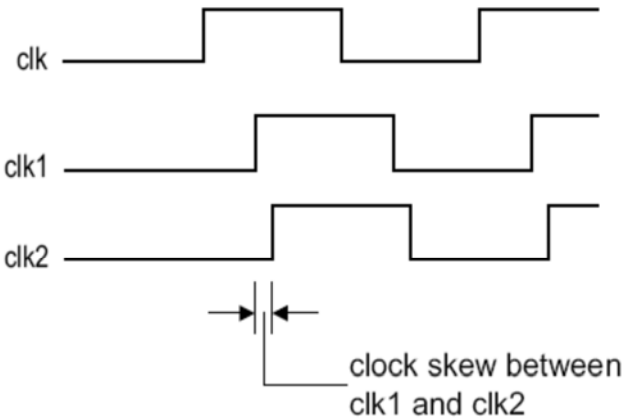
- Ideal H-routing



Clock skew (时钟偏移)

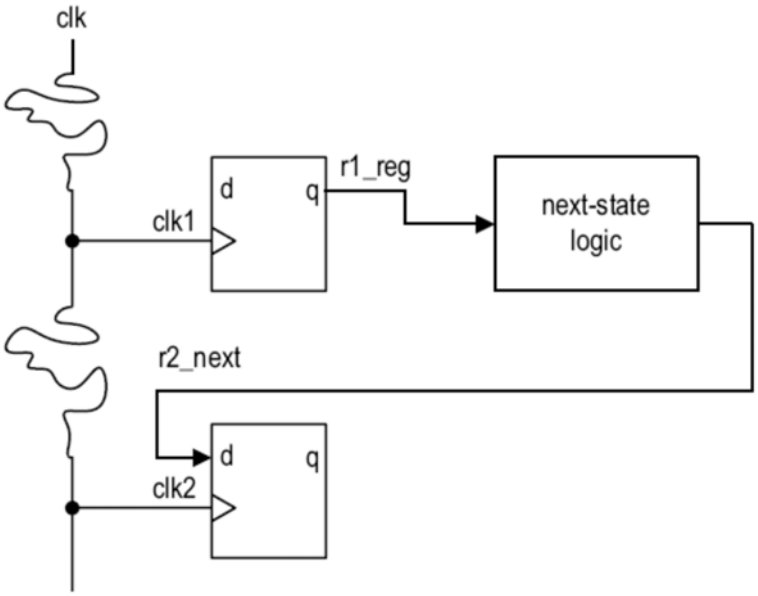
- Skew: time difference between two arriving clock edges

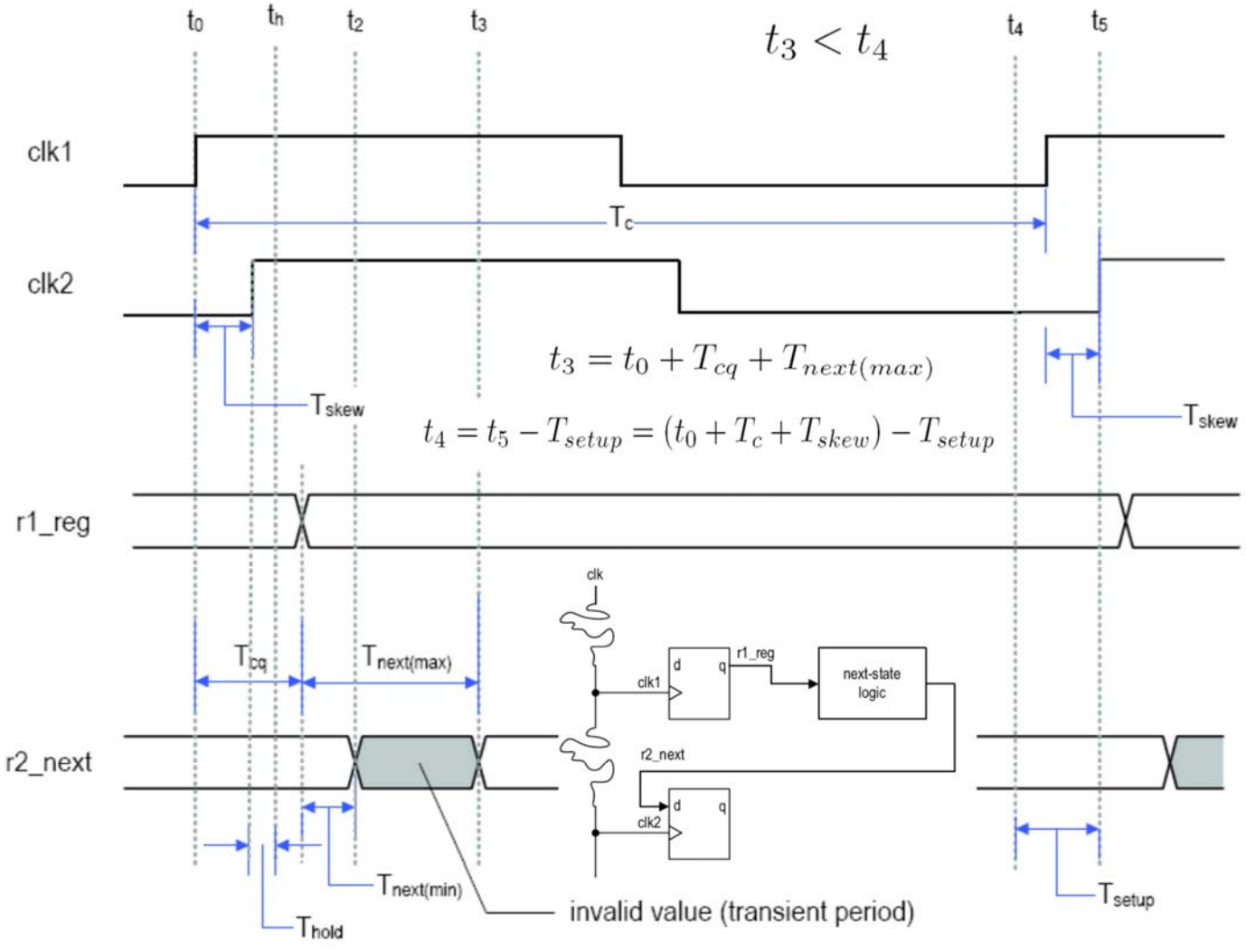




Timing analysis

- Setup time constraint (impact on max clock rate)
- Hold time constraint





$$t_3 < t_4$$

$$t_3 = t_0 + T_{cq} + T_{next(max)}$$

$$t_4 = t_5 - T_{setup} = (t_0 + T_c + T_{skew}) - T_{setup}$$

$$T_{cq} + T_{next(max)} + T_{setup} - T_{skew} < T_c$$

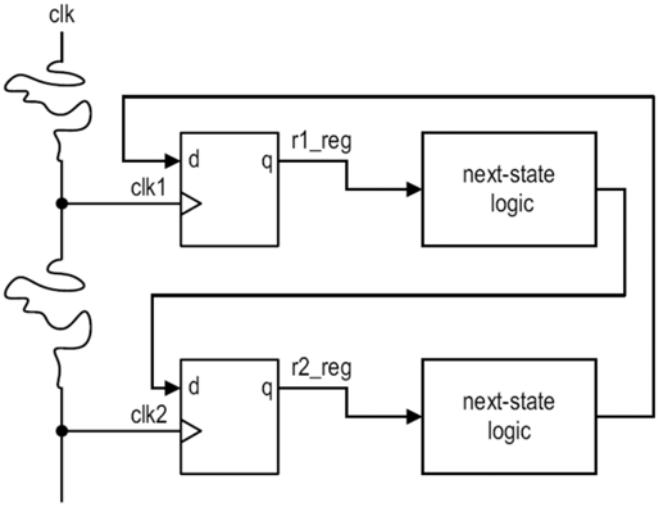
$$T_{c(min)} = T_{cq} + T_{next(max)} + T_{setup} - T_{skew}$$

- Clock skew actually helps increasing clock rate in this particular case

- If the clock signal travels from the opposite direction

$$T_{c(min)} = T_{cq} + T_{next(max)} + T_{setup} + T_{skew}$$

- Normally we have to consider the worst case since
  - No control on clock routing during synthesis
  - Multiple feedback paths



- Hold time constraint

$$t_h < t_2$$

$$t_2 = t_0 + T_{cq} + T_{next(min)}$$

$$t_h = t_0 + T_{hold} + T_{skew}$$

$$T_{hold} < T_{cq} + T_{next(min)} - T_{skew}$$

$$T_{hold} < T_{cq} - T_{skew}$$

- Skew may reduce hold time margin
- Hold time violation cannot be corrected in RT level

- Summary
  - Clock skew normally has negative impact on synchronous sequential circuit
  - Effect on setup time constraint: require to increase clock period (i.e., reduce clock rate)
  - Effect on hold time constraint: may introduce hold time violation
    - Can only be fixed during physical synthesis: re-route clock; re-place register and comb logic; add artificial delay logic
  - Skew within 10% of clock period tolerable

3. Multiple-clock system

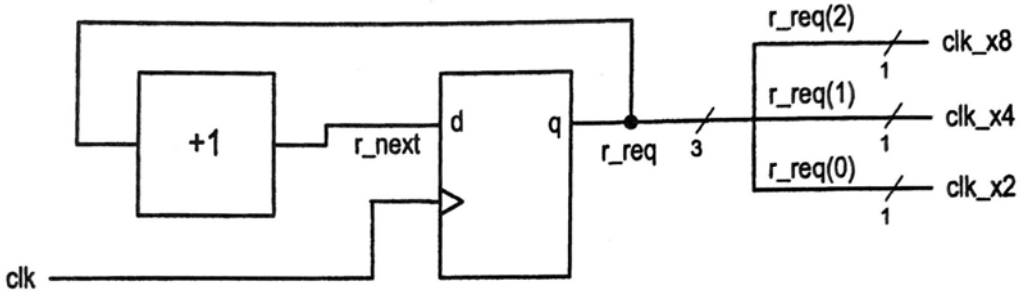
Why multiple clocks

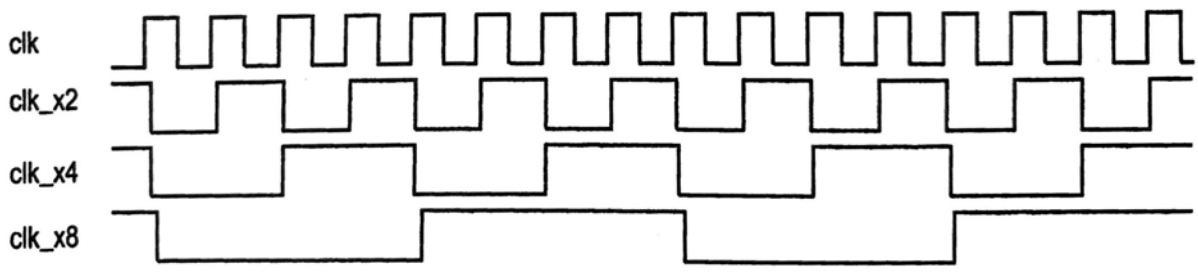
- Inherent multiple clock sources
  - E.g., external communication link
- Circuit size
  - Clock skew increases with the # FFs in a system
  - Current technology can support up to 10^4 FFs
- Design complexity
  - E.g., as sysetm w/ 16-bit 20 MHz processor, 1-bit 100 MHz serial interface, 1 MHz I/O controller
- Power consideration
  - Dynamic power proportional to switching freq

Derived vs Independent clocks

- Independent clocks:
  - Relationship between the clocks is unknown
- Derived clocks:
  - A clock is derived from another clock signals (e.g., different clock rate or phase)
  - Relationship is known
  - Logic for the derived clock should be separated from regular logic and manually synthesized (e.g., special delay line or PLL)
  - A system with derived clock can still be treated and analyzed as a synchronous system

Poorly conceived clock divider





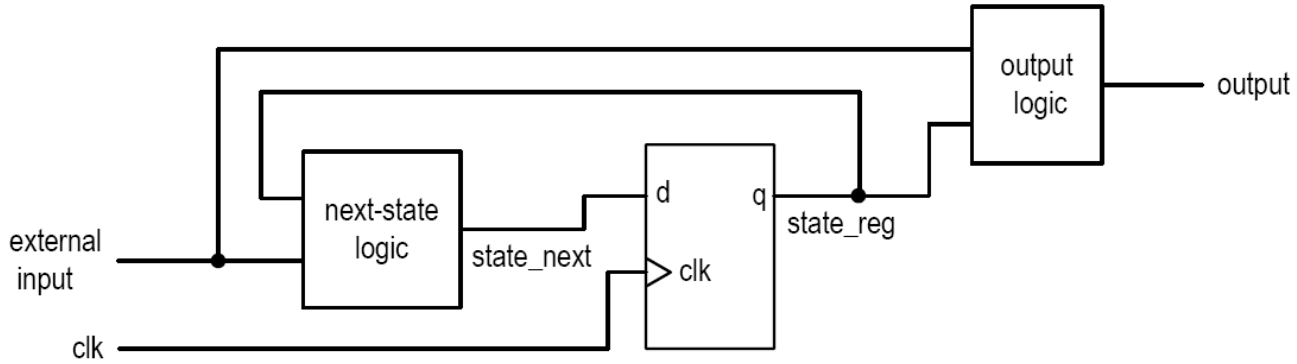
GALS

- Globally asynchronous locally synchronous system
  - Partition a system into multiple clock domains
  - Design and verify subsystem in same clock domain as a synchronous system
  - Design special interface between clock domains

4. Meta-stability and synchronization failure

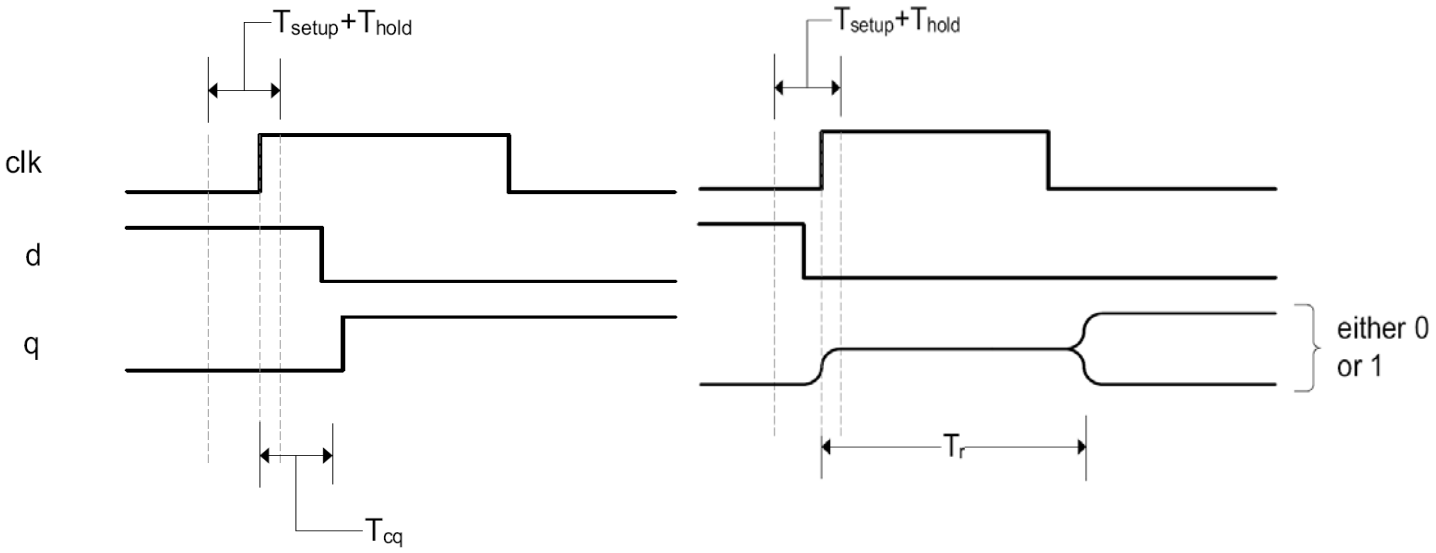
Timing analysis of a synchronous system

- To satisfy setup time constraint:
  - Signal from the state register
    - Controlled by clock
    - Adjust clock period to avoid setup time violation
  - Signal from external input
    - Same if the external input comes from another synchronous subsystem
    - Otherwise, have to deal with the occurrence of setup time violation.



Metastability

- What happens after timing violation?



– Output of FF becomes 1 (sampled old input value)
– Output of FF becomes 0 (sampled new input value)
– FF enters metastable state, the output exhibits an "in-between" value
  - FF eventually "resolves" to one of stable states
  - The resolution time is a random variable with distribution function (τ is decay constant)

$$P(T_r) = e^{-\frac{T_r}{\tau}}$$

  - The probability that metastability persists beyond T_r (i.e., cannot be resolved within T_r)

MTBF(T_r)

- Synchronization failure
  - an FF cannot resolve the metastable condition within the given time
- MTBF
  - Mean Time Between synchronization Failures
  - Basic criterion for metastability analysis
  - Frequently expressed as a function of T_r (resolution time provided)

- MTBF computation

- R_{meta}: The average rate at which an FF enters the metastable state.
- $P(T_r)$: The probability that an FF cannot resolve the metastable condition within T_r.

$$R_{meta} = w * f_{clk} * f_d$$

w is the *susceptible time window*.

$$P(T_r) = e^{-\frac{T_r}{\tau}}$$

$$AF(T_r) = R_{meta} * P(T_r) = w * f_{clk} * f_d * e^{-\frac{T_r}{\tau}}$$

$$\text{MTBF}(T_r) = \frac{1}{AF(T_r)} = \frac{e^{\frac{T_r}{\tau}}}{w * f_{clk} * f_d}$$

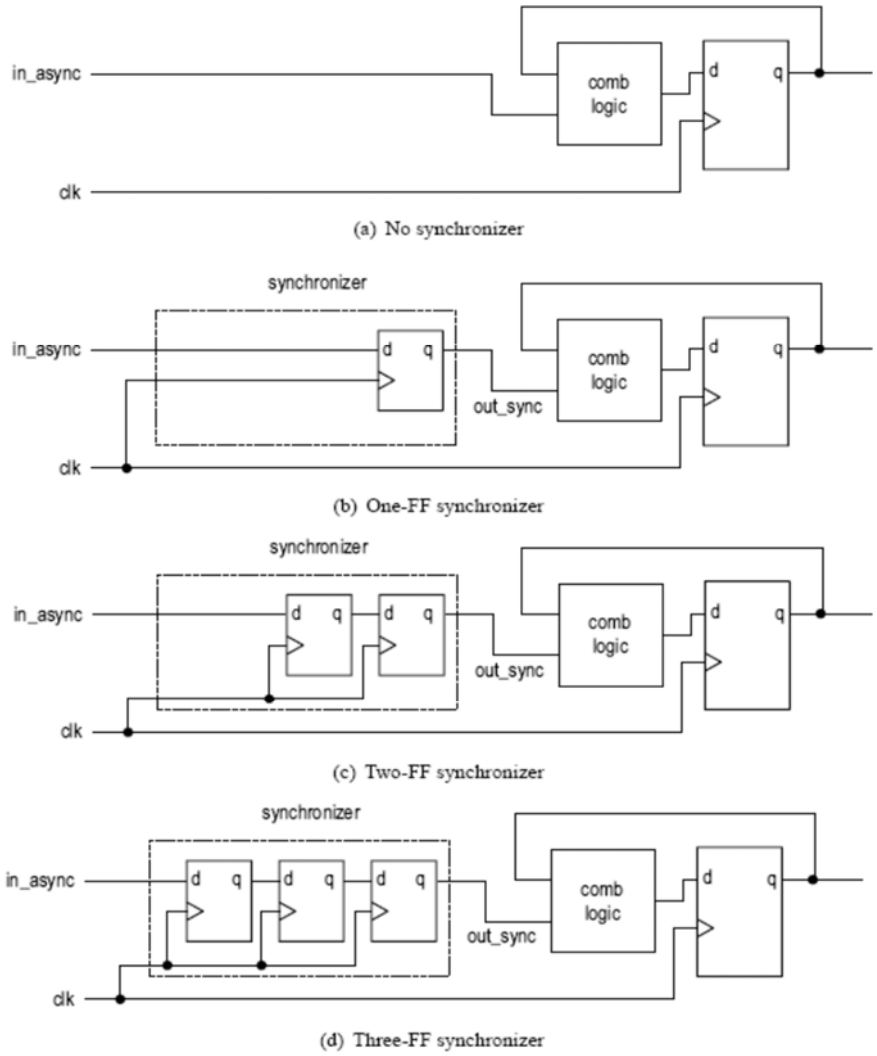
- E.g., w=0.1ns, τ=0.5ns, f_{clk}=50MHz, f_d=0.1f_{clk}

T_r	**MTBF**
0.0 ns	$4.00 * 10^{-05}$ sec (0.04 msec)
2.5 ns	$5.94 * 10^{-03}$ sec (5.94 msec)
5.0 ns	$8.81 * 10^{-01}$ sec (0.88 sec)
7.5 ns	$1.31 * 10^{+02}$ sec (131 sec)
10.0 ns	$1.94 * 10^{+04}$ sec (5.39 hours)
12.5 ns	$2.88 * 10^{+06}$ sec (3.33 days)
15.0 ns	$4.27 * 10^{+08}$ sec (1.36 years)
17.5 ns	$6.34 * 10^{+10}$ sec ($2.01 * 10^{3}$ years)
20.0 ns	$9.42 * 10^{+12}$ sec ($2.99 * 10^{5}$ years)
22.5 ns	$1.40 * 10^{+15}$ sec ($4.43 * 10^{7}$ years)
25.0 ns	$2.07 * 10^{+17}$ sec ($6.58 * 10^{9}$ years)
27.5 ns	$3.08 * 10^{+19}$ sec ($9.76 * 10^{11}$ years)
30.0 ns	$4.57 * 10^{+21}$ sec ($1.45 * 10^{14}$ years)
32.5 ns	$6.78 * 10^{+23}$ sec ($2.15 * 10^{16}$ years)
35.0 ns	$1.01 * 10^{+26}$ sec ($3.19 * 10^{18}$ years)

- Observations
  - MTBF is statistical average
  - Only T_r can be adjusted in practical design
  - MTBF is extremely sensitive to T_r
    - Good: synchronization failure can be easily avoided by providing additional resolution time
    - Bad: minor modification can introduce synchronization failure

5. Synchronizer

- Synchronization circuit:
  - Synchronize an asynchronous input with system clock
  - No physical circuit can prevent metastability
  - Synchronizer just provides enough time for the metastable condition to be “resolved”
- E.g.,
  - w = 0.1ns, τ = 0.5ns, f_{clk} = 50MHz, $f_d = 0.1 f_{clk}$
  - T_{setup} = 2.5s



No synchronizer

- $T_r = 0$
- MTBF(0) = 0.04 ms

One-FF synchronizer

- $T_r = T_c - (T_{comb} + T_{setup})$
- T_r depends on T_c , T_{setup} and T_{comb}
  - T_c: vary with system specification
  - T_{comb}: vary with circuit, synthesis (gate delay), placement & routing (wire delay)
- E.g.,
  - $T_r = 20 - (T_{comb} + 2.5) = 17.5 - T_{comb}$
  - T_{comb} = 1ns, T_r = 16.5ns; MTBF(16.5) = 272yr
  - T_{comb} = 12.5ns, T_r = 5ns; MTBF(5) = 0.88ns
- Not a reliable design

Two-FF synchronizer

- Add an extra FF to eliminate T_{comb}
  - $T_r = T_c - T_{setup}$
  - T_r depends on T_c only
  - Async input delayed by two clock cycles
- E.g.,
  - $T_r = 20 - 2.5 = 17.5$; MTBF(17.5) = 3,000yr
- Most commonly used synchronizer
- In ASIC technology
  - May have “metastability-hardened” D FF cell (large area)

Three-FF synchronizer

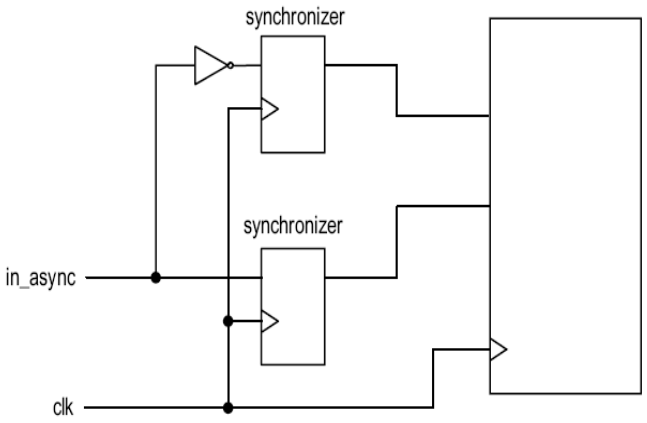
- Add an extra stage to increase resolution time
  - $T_r = 2(T_c - T_{setup})$
  - Async input delayed by three clock cycles
- E.g.,
  - T_r =2*(20 - 2.5); MTBF(30)=6 billion yr
- Hardly needed

Observation

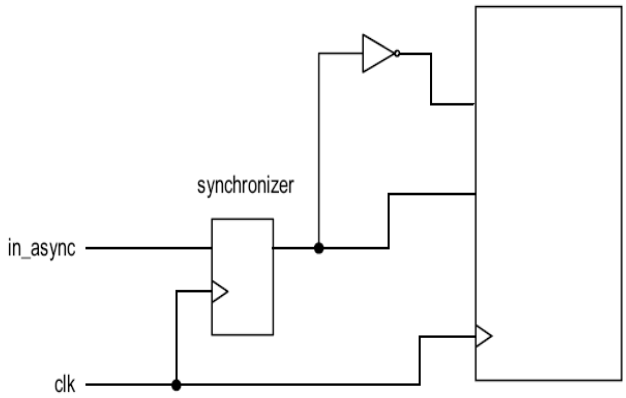
- T_r is in the exponent of MTBF equation
- Small variation in T_r can lead to large swing in MTBF

Proper use of synchronizer

- Use a glitch-free signal for synchronization
- Synchronize a signal in a single place
- Avoid synchronization multiple “related” signals.
- Reanalyze the synchronizer after each design change



(a) Synchronizing a signal in two places



(b) Synchronizing a signal in one place

Why synchronization is a “tricky” issue

- Metastability is basically an “analog” phenomena
- Metastability behavior is described by random variable
- Metastability cannot be easily modeled or simulated in gate level (only ‘X’)
- Metastability cannot be easily observed or measured in physical circuit (e.g., MTBF = 3 months)
- MTBF is very sensitive to circuit revision

6. Enable tick crossing clock domains

Signals crossing clock domains

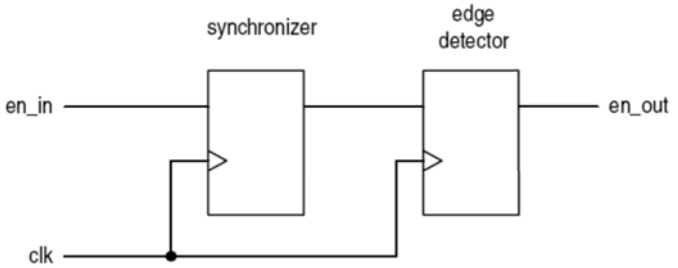
- Synchronizer
  - Just ensures that the receiving system does not enter a metastable state
  - Not guarantee the “function” of the received signal
- Consideration
  - One signal
  - Multiple signals (“bundled data”)

Domain-crossing of an enable signal

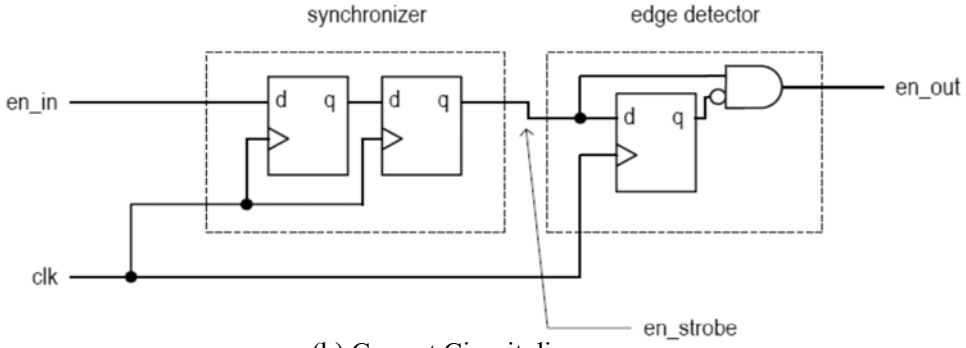
- An enable tick
  - One-clock-cycle wide
  - To be sample in a single clock edge
  - E.g., enable input of a counter; read/write signal of a FIFO buffer
  - Can also be used to retrieve bundled data

“Wide” enable signal

- From a slow clock domain to a fast clock domain (e.g., 1 MHz to 10 MHz)

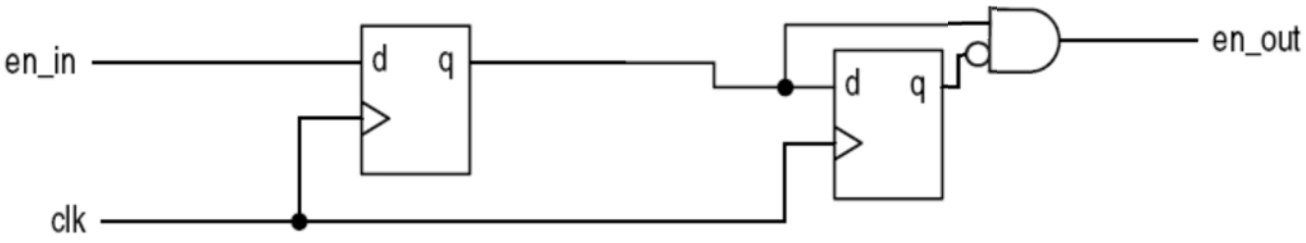


(a) Block diagram



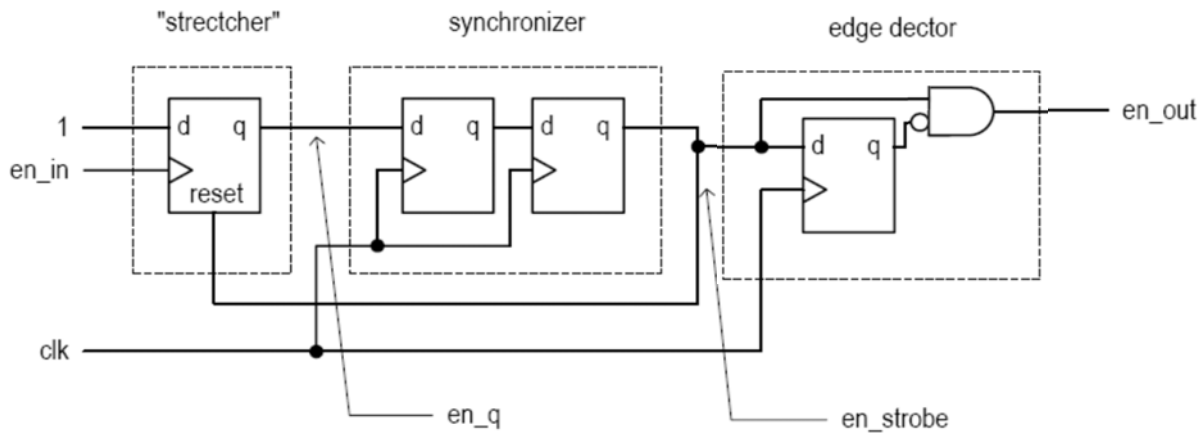
(b) Correct Circuit diagram

- Will this work?



“Narrow” enable signal

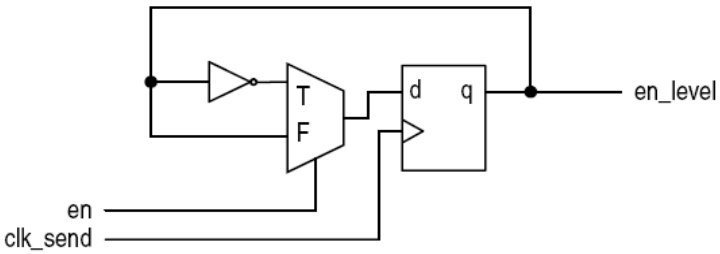
- From a fast clock domain to a slow clock domain (e.g., 10 MHz to 1 MHz)
- The enable pulse is probably too narrow to be detected
- Need to “stretch” the pulse
  - Cannot be done by a normal sequential circuit
  - Need to use “tricks”



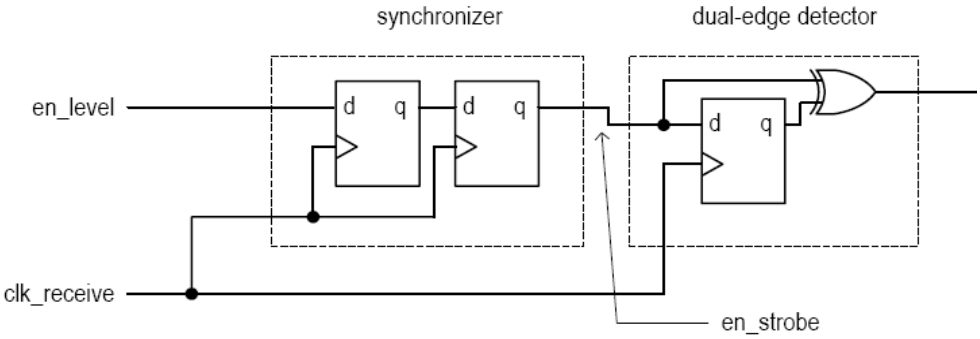
- en_q asserted at the rising edge of en_in
- en_q then synchronized
- en_strobe then clears stretcher
- en_q may last over two clock cycles and thus an edge-detector is needed
- Can this scheme be used for wide-pulse?

Level-alternating scheme

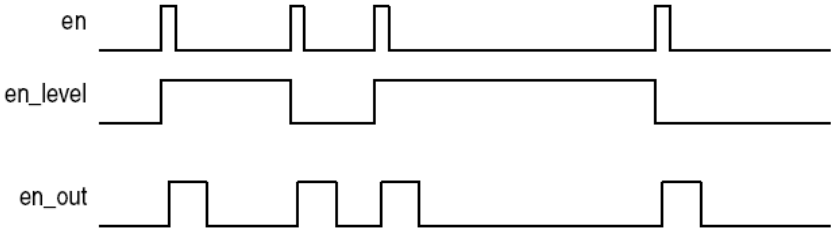
- Output interface of sender and input interface of receiver modified for domain crossing
- Output interface converts an “edge-sensitive” enable pulse to a level-alternating signal
  - Use a T-FF
- Input interface converts the level-alternating signal back to “edge-sensitive” enable pulse
  - Use a dual-edge detector
- Eliminate the ad-hoc stretcher and follow the synchronous design methodology



(a) Block diagram of the sending subsystem



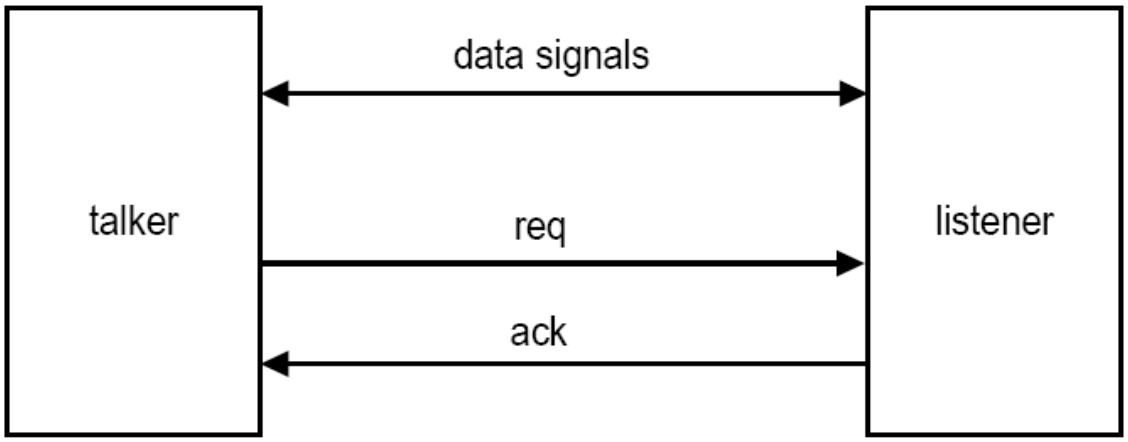
(b) Block diagram of the receiving subsystem

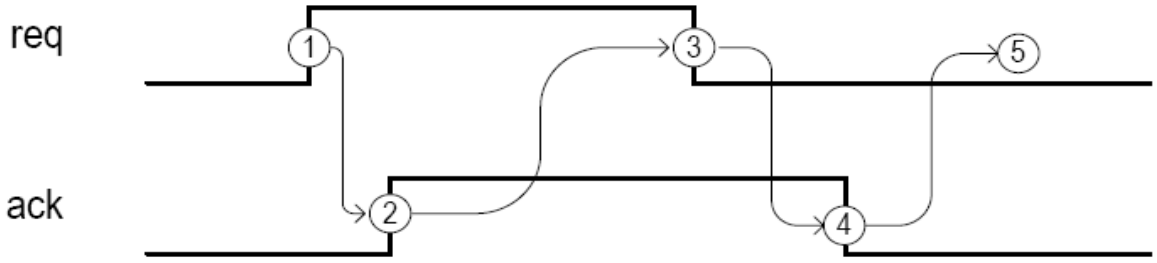


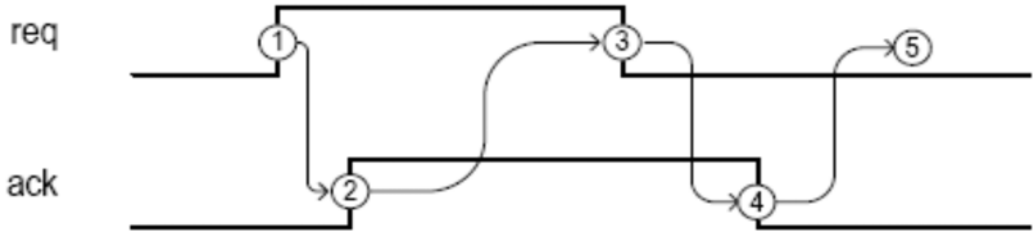
(c) Simplified timing diagram

7. Handshaking

- How to control the rate of data (or number of enable ticks) between two clock domains? (e.g., 10 MHz system to 1 MHz system)
- Does the sending system have prior knowledge about the processing speed of receiving system?
- Handshaking scheme
  - Use a feedback signal
  - Make minimal assumption about the receiving system

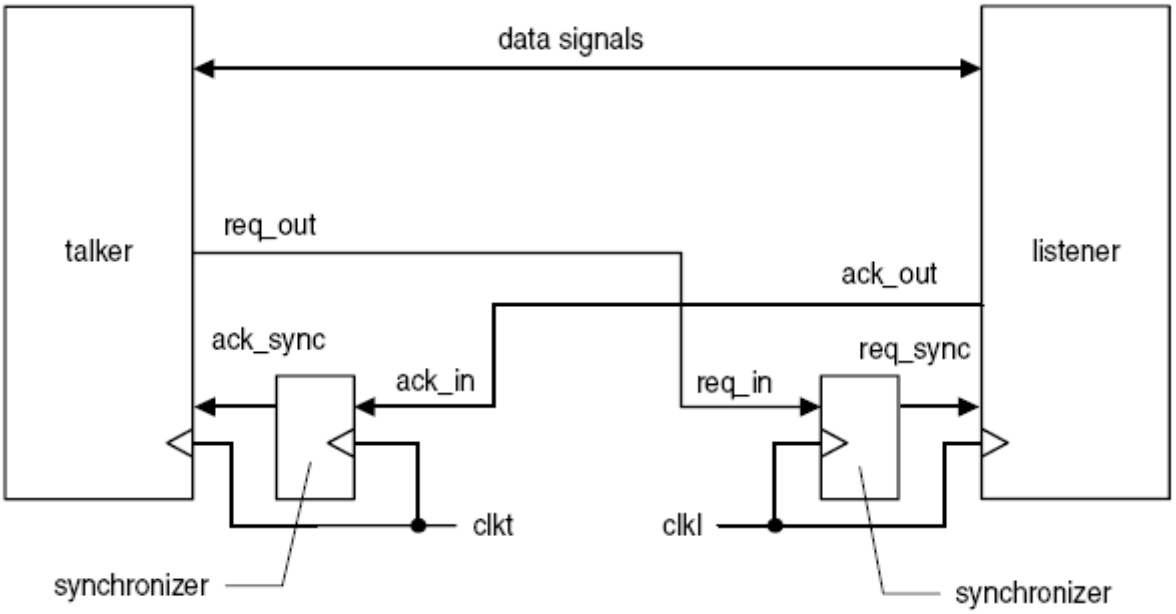




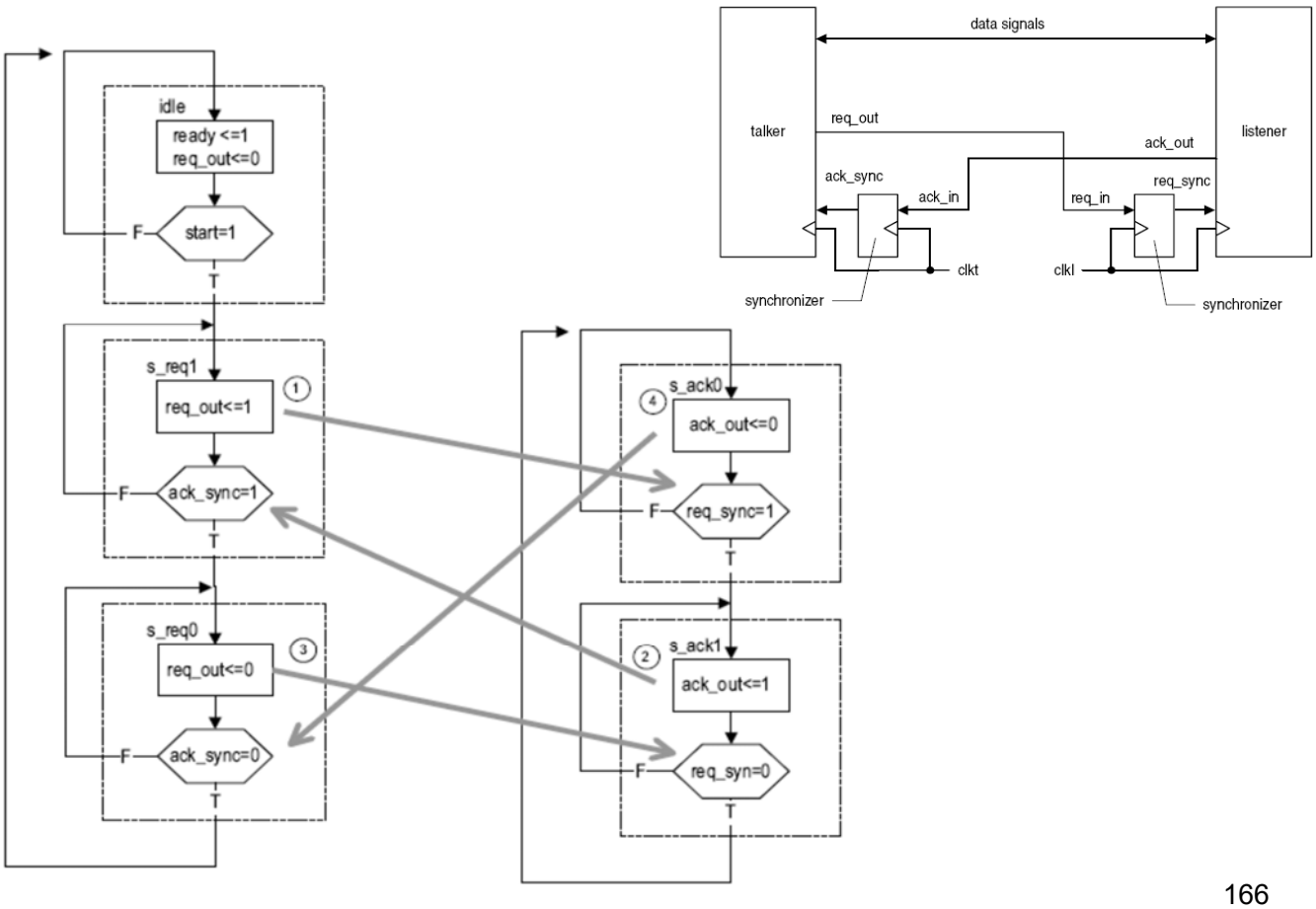


- Four phases:
  - Phase 1: talker activates req
  - Phase 2: listener activates ack
  - Phase 3: talker de-activates req
  - Phase 4: listener de-activates ack
  - Talker can start a new request

- Need synchronizer if talker listener in different clock domains

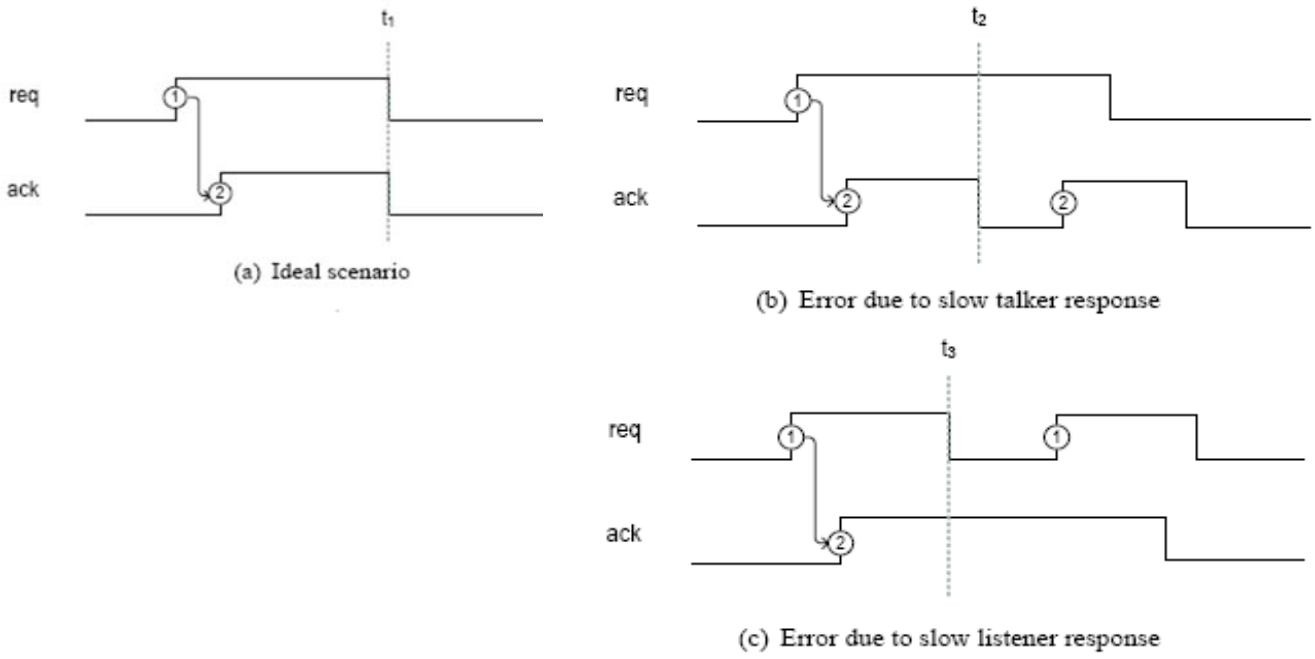


- Talker FSM and listener FSM



- Implementation:
  - Talker: FSM and synchronizer for ack_out
  - Listener: FSM and synchronizer for req_out
- Pass an enable tick using handshaking
  - The enable tick functions as the start signal in talker
  - The listener generates a Mealy output which is asserted when req_sync is asserted in the s_ack0 state (i.e., a rising-edge detection circuit for req_sync)

- Can we remove the second part of handshaking?

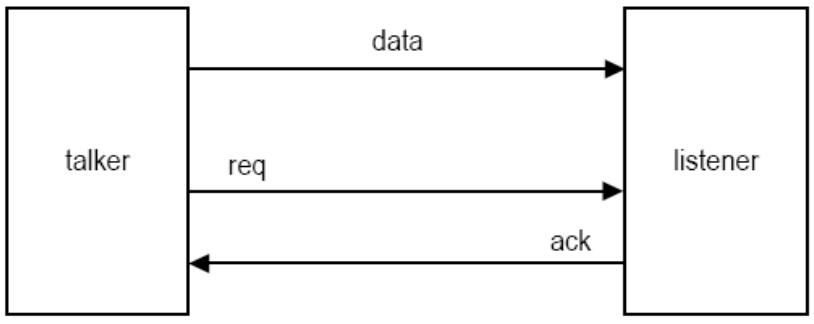


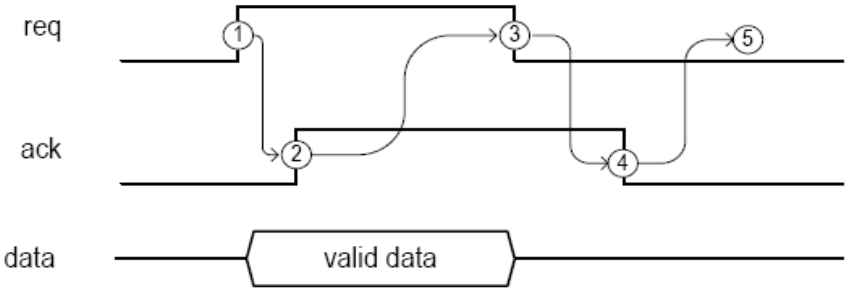
- Two-phase handshaking protocol
  - We can modify the 4-phase protocol so that talker/listener not returning to 0
  - May not be proper for certain applications

8. Data transfer crossing clock domains

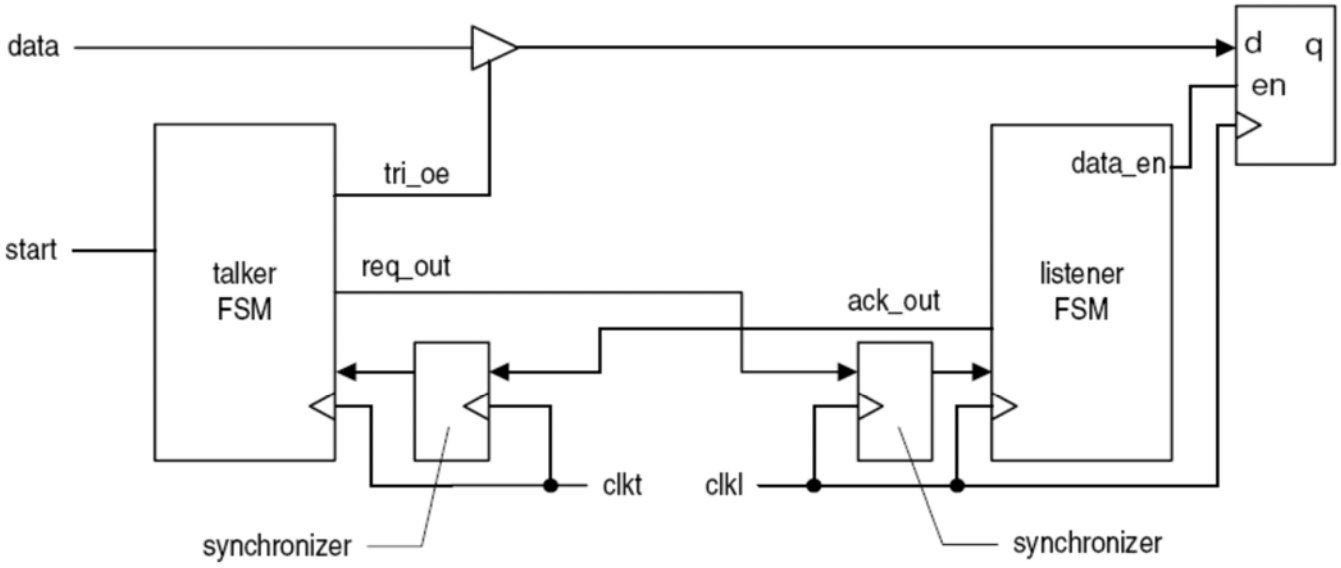
- It is difficult to synchronize a multiple-bit signal (e.g., signal changes from 11 to 00)
- Use req/ack and handshaking protocol to coordinate data transfer
  - Only one signal needs to be synchronized in each domain
  - All other signals are bundled as “data”

- Push operation (talker sending data)
  - Conceptual diagrams

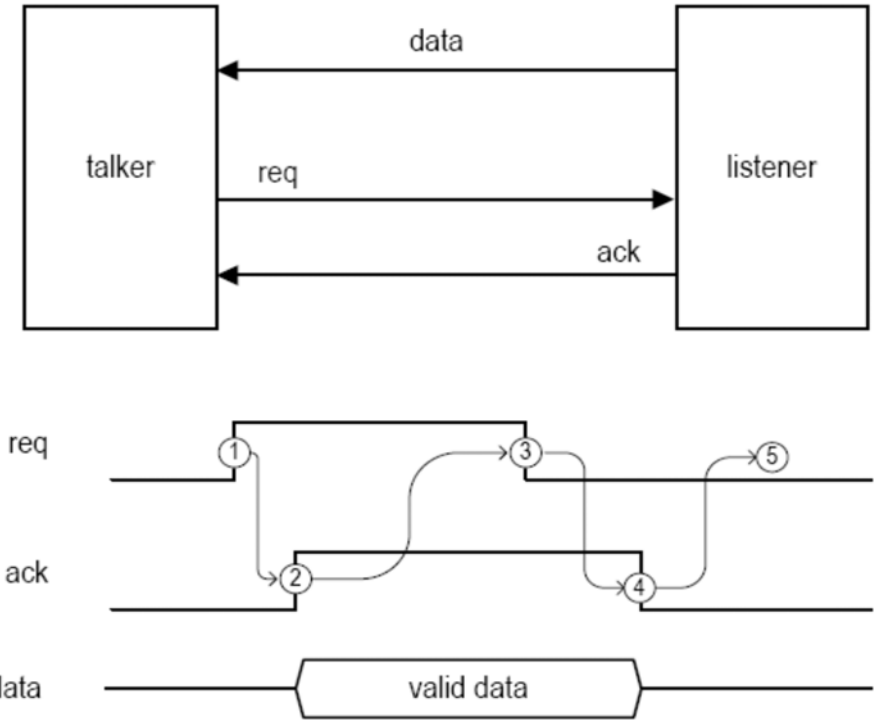




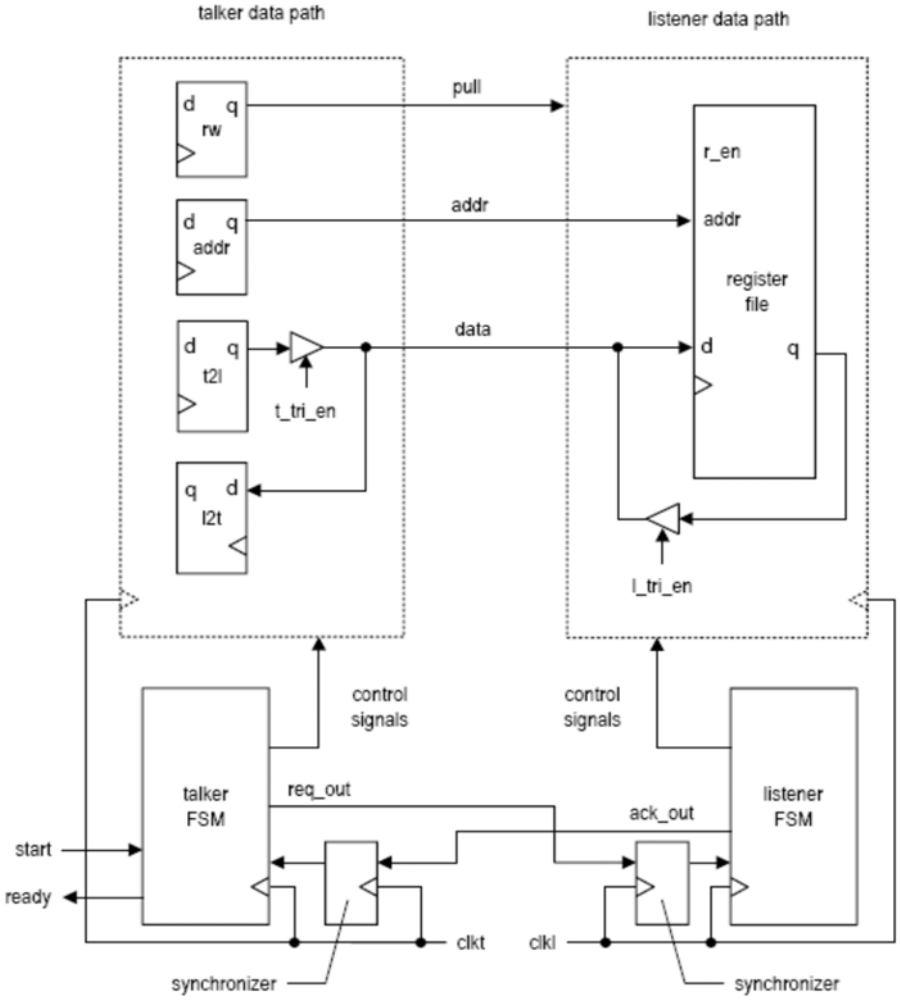
- More detailed diagram
  - Talker activates req_out and tri_en (i.e., placing data on data bus) at the same time.
  - req_out is delayed one or two clocks when synchronized in listener
  - data is stabilized when data_en is asserted (i.e., no timing violation)



- Pull operation (taller retrieving data)
  - Conceptual diagrams

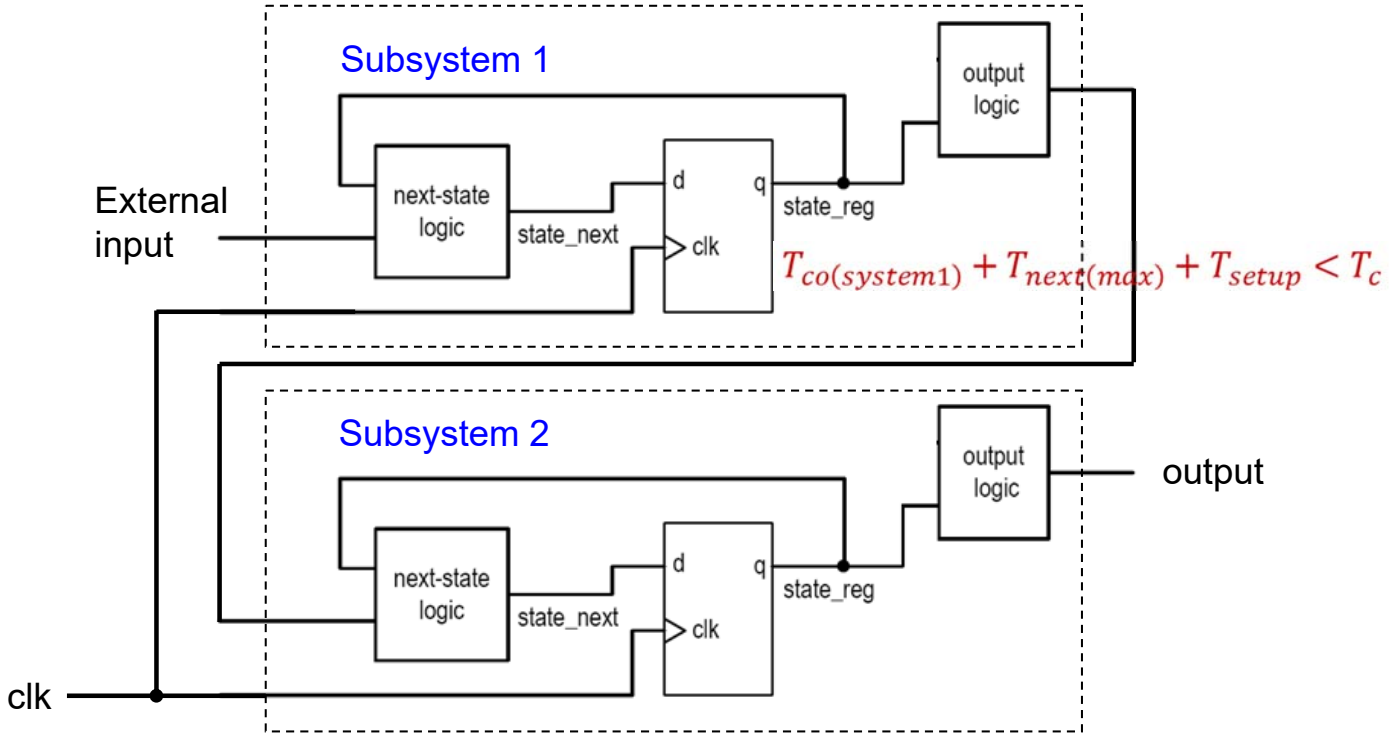


- Bidirectional operation is possible; e.g.,



- Performance:
  - How many clock cycle for one data transfer?
- Other methods for data transfer
  - FIFO (synchronization needed for empty and full status signal)
  - Shared memory (synchronization needed for arbitration circuit)
  - Dual-port memory (meta-stable condition may occur in the internal arbitration circuit)

Input-related timing consideration



$T_{cq} + T_{next(max\ of\ state_reg\text{-}to\text{-}state_next)} + T_{setup} < T_{c1}$

$T_{co(system1)} + T_{next(max\ of\ ext_input\text{-}to\text{-}state_next)} + T_{setup} < T_{c2}$