Lecture 05~06

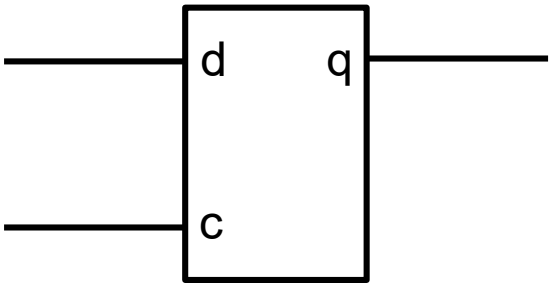
순서회로 설계

순서회로

- 메모리(또는 상태)를 포함함
- 같은 입력 → 회로 상태에 따라 출력 달라질 수 있음
- Verilog 설계 시 사용 가능 문장
  - assign
  - always @*
    ```
    begin
        blocking assignment;
        blocking assignment;
        …
    end
    ```
  - always @(posedge clk or negedge rstb)
    ```
    if (~rstb) begin
        non-blocking assignment;
        non-blocking assignment;
        …
    end
    else begin
        non-blocking assignment;
        non-blocking assignment;
        …
    end
    ```

Latch(래치)

- 1비트 정보 저장하는 기본 메모리 소자
- Clock 신호 없음
- Asynchronous(비동기식)
- Level-sensitive
- 회로 설계 시 거의 사용하지 않음

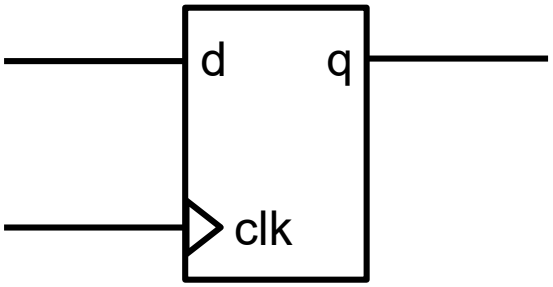


D latch(래치)

c	q^{next}
0	q
1	d

Flip-flop(플립플롭)

- 1비트 정보 저장하는 기본 메모리 소자
- Clock 신호 있음
- Synchronous(동기식)
- Edge-sensitive
- Latch(래치)보다 2배 정도 크지만 안정성 높음

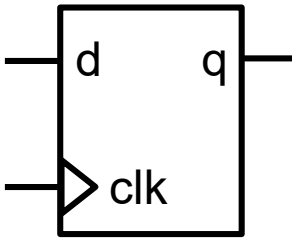


D FF(플립플롭)

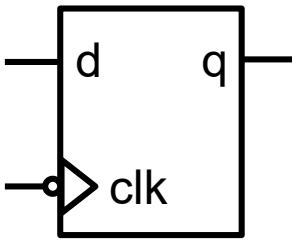
clk	q^{next}
0	q
1	q
↑(rising edge)	d

Flip-flop(플립플롭)

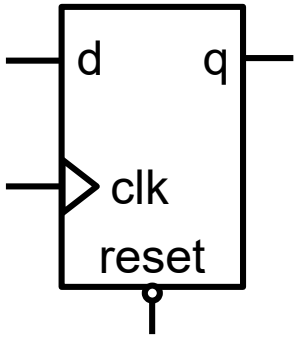
- Positive-edge-triggered D FF
- Negative-edge-triggered D FF
- 비동기식 reset D FF



clk	q^{next}
0	q
1	q
↑(rising edge)	d

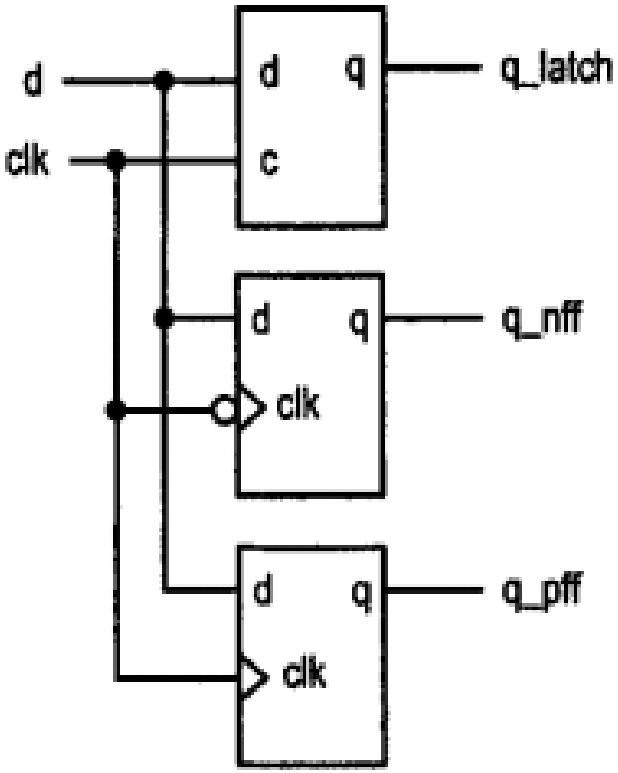


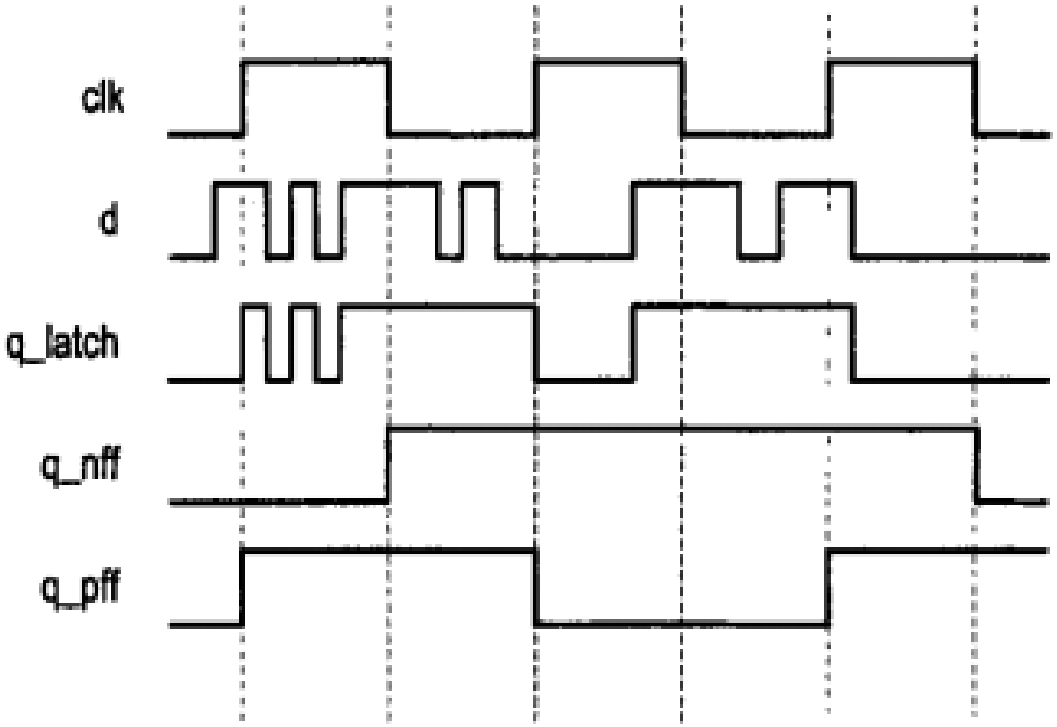
clk	q^{next}
0	q
1	q
↓(falling edge)	d



reset	clk	q^{next}
0	-	0
1	0	q
1	1	q
1	↑(rising edge)	d

Latch vs. flip-flop

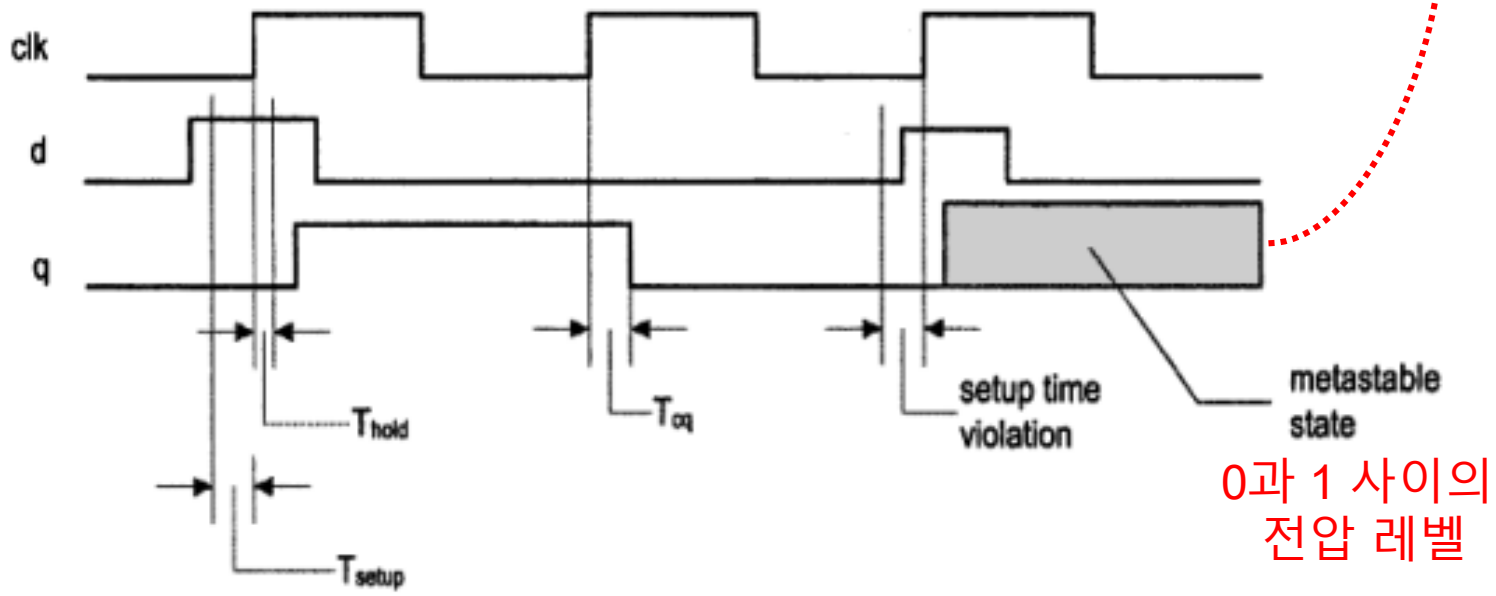




Flip-flop의 타이밍

- T_{cq} : clock-to-q delay
- T_{setup} : setup time
- T_{hold} : hold time

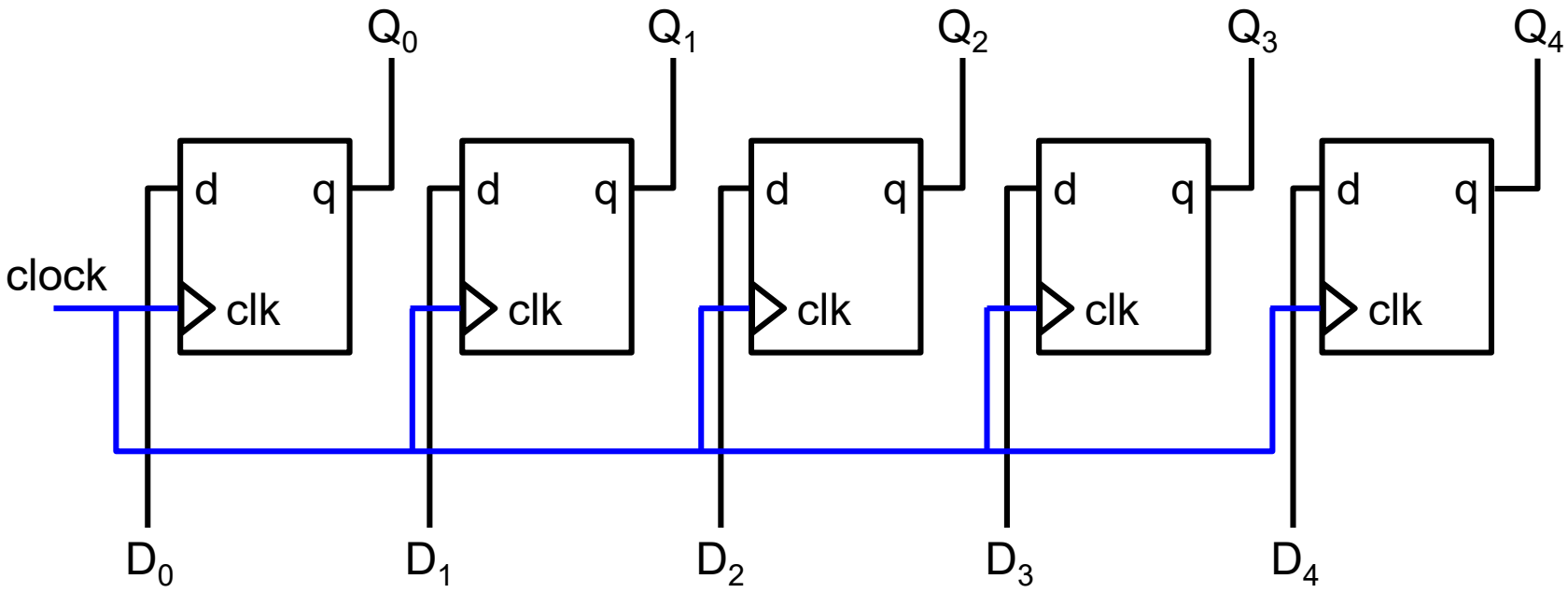
T_r(resolution time) 지난 후 0 혹은 1로 됨



0과 1 사이의
전압 레벨

Register(레지스터)

- D FF 하나 ↔ 1비트 register
- D FF N개 ↔ N비트 register



Latch(래치) 합성

- Verilog 코드

```
module d_latch
#(
    parameter N = 8
) (
    input               c,
    input       [N-1:0] d,
    output reg  [N-1:0] q
);

always @* begin
    if (c) q = d;
    else   q = q; // latch 합성의 경우 else 없어도 된다
end

endmodule
```

Flip-flop 합성

- Verilog 코드

D FF

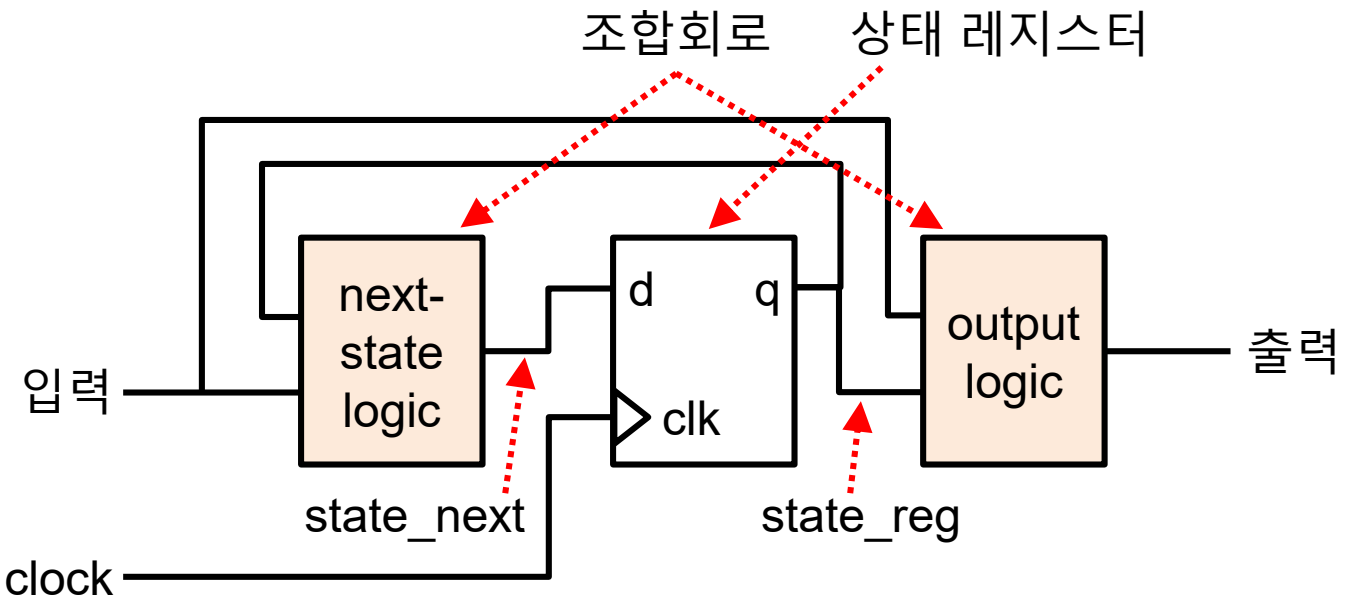
```
module d_ff
#(
    parameter N = 8
) (
    input               clk,
    input       [N-1:0] d,
    output reg  [N-1:0] q
);

always @(posedge clk) begin
    q <= d;
end

endmodule
```

비동기식 reset D FF

```
module d_ff_rstb
#(
    parameter N = 8
) (
    input               clk,
    input               rstb,
    input       [N-1:0] d,
    output reg  [N-1:0] q
);

always @(posedge clk or negedge rstb) begin
    if (~rstb) q <= 0;
    else       q <= d;
end

endmodule
```

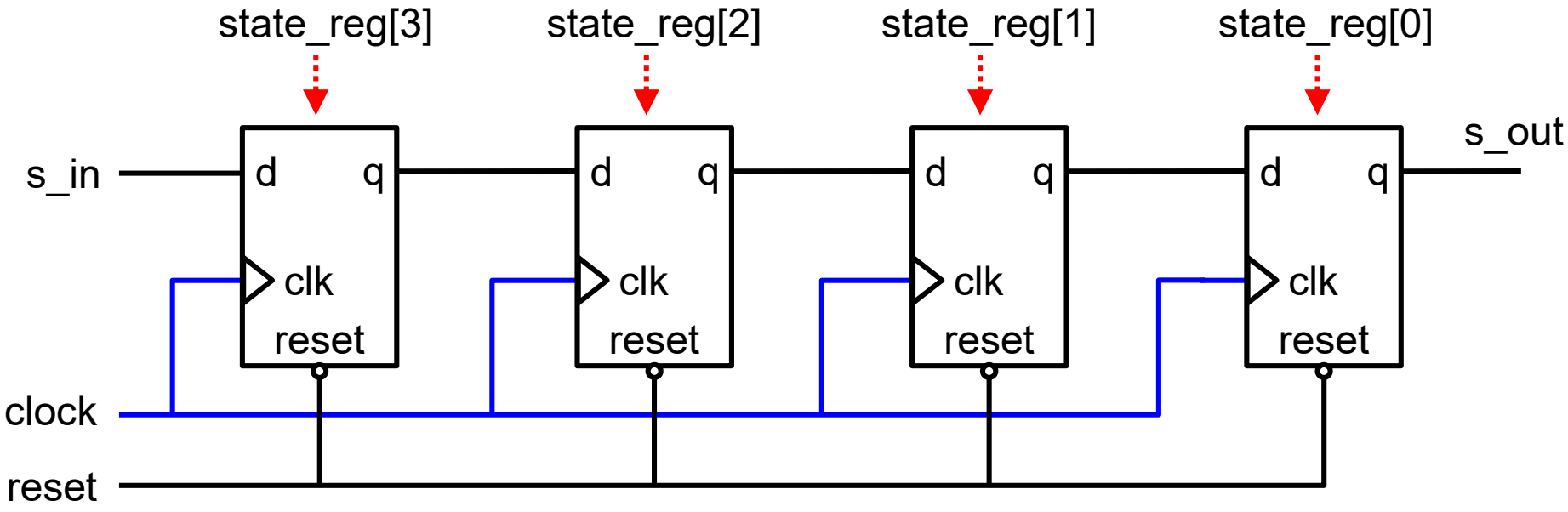
일반적인 순서회로 구조

- 좋은 설계 위한 팁: 입출력 레지스터 사용
- 최소 전파 지연 시간 = min(register→조합회로→register)
- 최대 동작 주파수 = 1/(최소 전파 지연 시간)



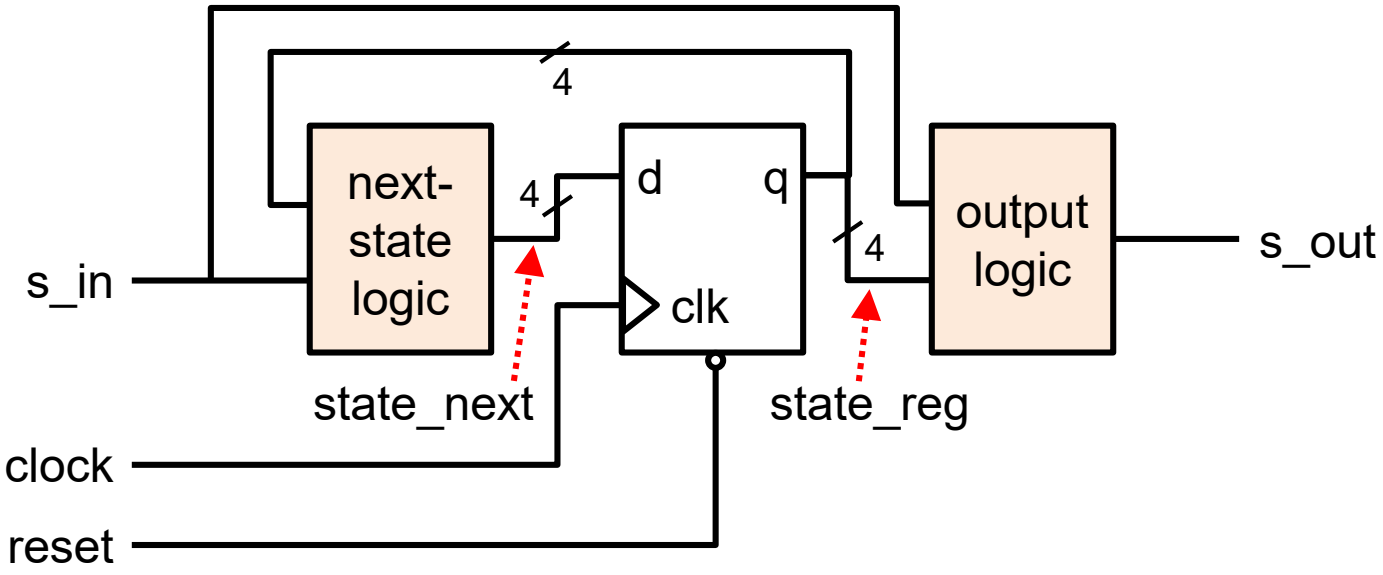
Free-running 시프트 레지스터

- 컨트롤 신호 없음
- Clock cycle 당 레지스터의 내용은 오른쪽 1비트 씩 시프트 됨



Free-running 시프트 레지스터

- Next-state logic : state_next = {s_in, state_reg[3:1]}
- Output logic : s_out = state_reg[0]



Free-running 시프트 레지스터

- Next-state logic : state_next = {s_in, state_reg[3:1]}
- Output logic : s_out = state_reg[0]

rstb	clk	s_in	state_next	state_reg	s_out
0	1st	1	1000	0000	0
1	2nd	1	1100	1000	0
1	3rd	1	1110	1100	0
1	4th	0	0111	1110	0
1	5th	0	0011	0111	0
1	6th	1	1001	0011	1
1	7th	1	1000	1001	1

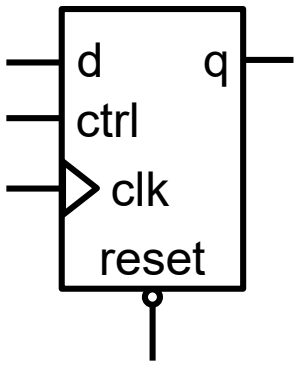
Free-running 시프트 레지스터

- Verilog 코드

```
module free_run_shift_reg
#(
    parameter N = 4
) (
    input  clk,
    input  rstb,
    input  s_in,
    output s_out
);

reg  [N-1:0] state_reg;
wire [N-1:0] state_next;

// state register
always @(posedge clk or negedge rstb) begin
    if (~rstb) state_reg <= 0;
    else       state_reg <= state_next;
end

// next-state logic
assign state_next = {s_in, state_reg[N-1:1]};

// output logic
assign s_out = state_reg[0];

endmodule
```

Universal 시프트 레지스터

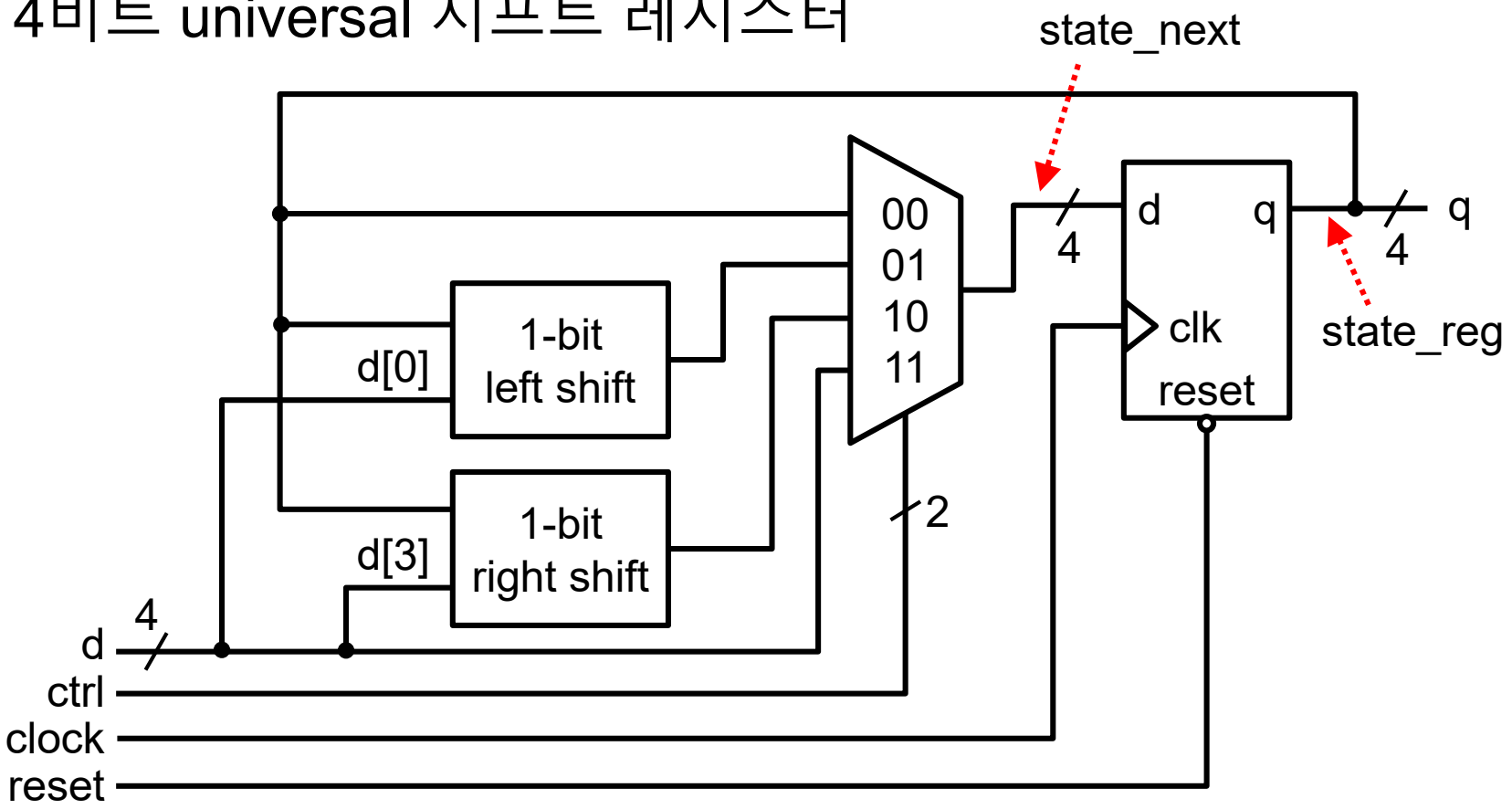
- 컨트롤 신호 있음
- 컨트롤 신호에 따라 4가지 동작 모드



컨트롤 신호 (ctrl)	동작 모드
00	레지스터 값 유지
01	왼쪽 1비트 시프트
10	오른쪽 1비트 시프트
11	입력 값을 가지고 옴

Universal 시프트 레지스터

- 4비트 universal 시프트 레지스터



Universal 시프트 레지스터

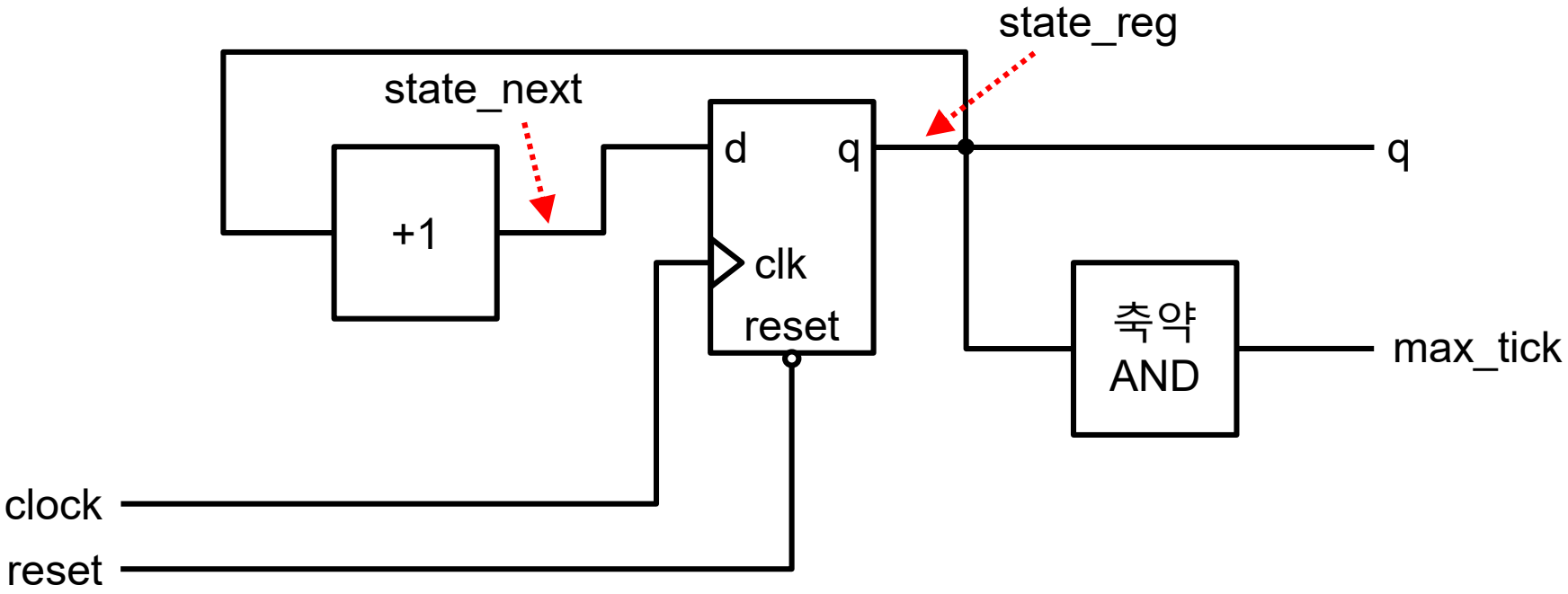
- Verilog 코드

```
module universal_shift_reg
#(
    parameter       N = 4
) (
    input           clk,
    input           rstb,
    input   [1:0]   ctrl,
    input   [N-1:0] d,
    output  [N-1:0] q
);
```

```
reg [N-1:0] state_reg;
reg [N-1:0] state_next;

// state register
always @(posedge clk or negedge rstb) begin
   if (~rstb) state_reg <= 0;
   else       state_reg <= state_next;
end

// next-state logic
always @* begin
   case (ctrl)
      2'b00: state_next = state_reg;
      2'b01: state_next = {state_reg[N-2:0], d[0]};
      2'b10: state_next = {d[3], state_reg[N-1:1]};
      default: state_next = d;
   endcase
end

// output logic
assign q = state_reg;

endmodule
```

Free-running 카운터

- 컨트롤 신호 없음
- Clock cycle 당 하나 씩 증가



Free-running 카운터

- Verilog 코드

```
module free_run_counter
#(
    parameter       N = 4
) (
    input           clk,
    input           rstb,
    output [N-1:0]  q,
    output          max_tick
);

reg [N-1:0] state_reg;
wire [N-1:0] state_next;

// state register
always @(posedge clk or negedge rstb) begin
    if (~rstb) state_reg <= 0;
    else       state_reg <= state_next;
end

// next-state logic
assign state_next = state_reg + 1'b1;

// output logic
assign q = state_reg;
assign max_tick = &state_reg;

endmodule
```

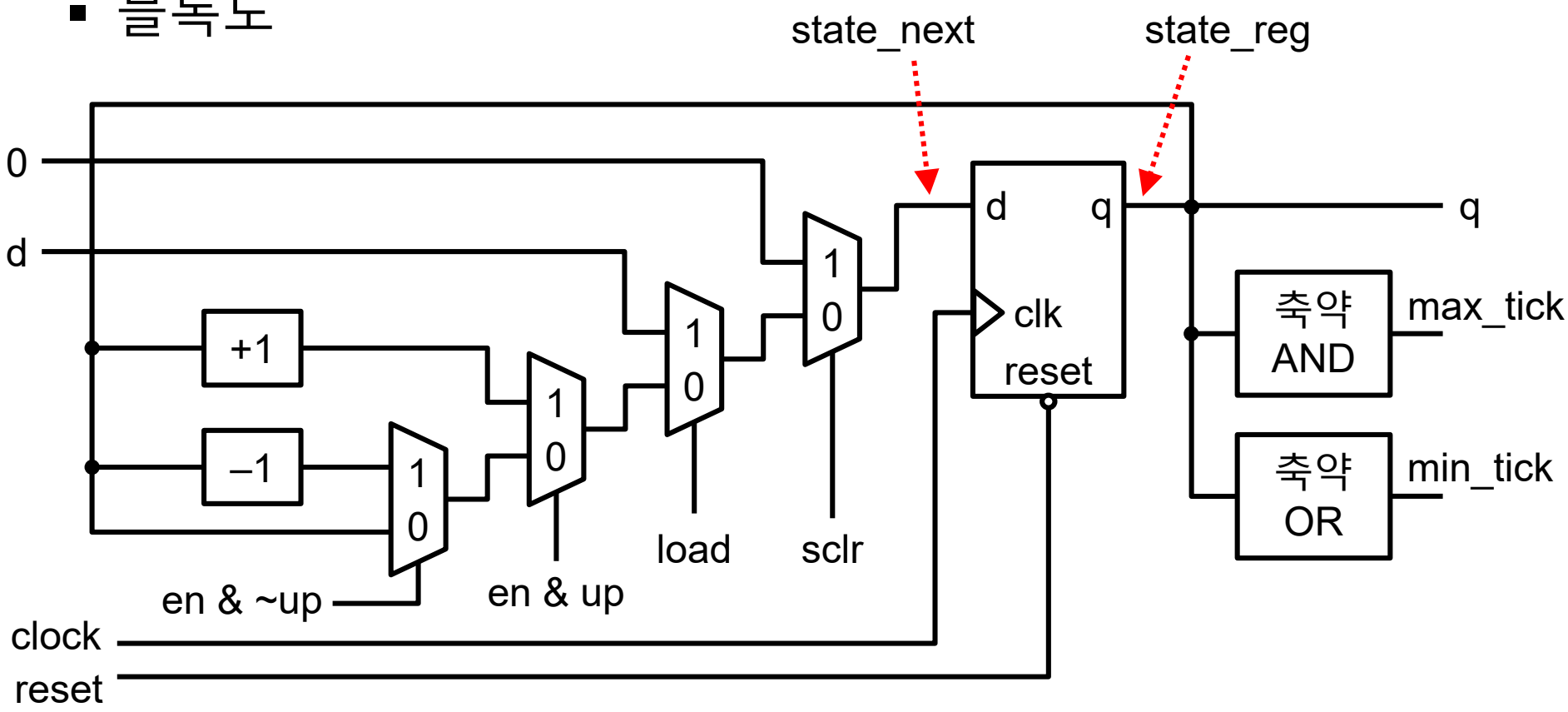
Universal 카운터

- 컨트롤 신호 있음
- 컨트롤 신호에 따라 5가지 동작 모드

컨트롤 신호				동작 모드
sclr	load	en	up	
1	-	-	-	동기식 clear
0	1	-	-	입력 값을 가지고 옴
0	0	0	-	카운터 값 유지
0	0	1	1	하나 씩 증가
0	0	1	0	하나 씩 감소

Universal 카운터

- 블록도



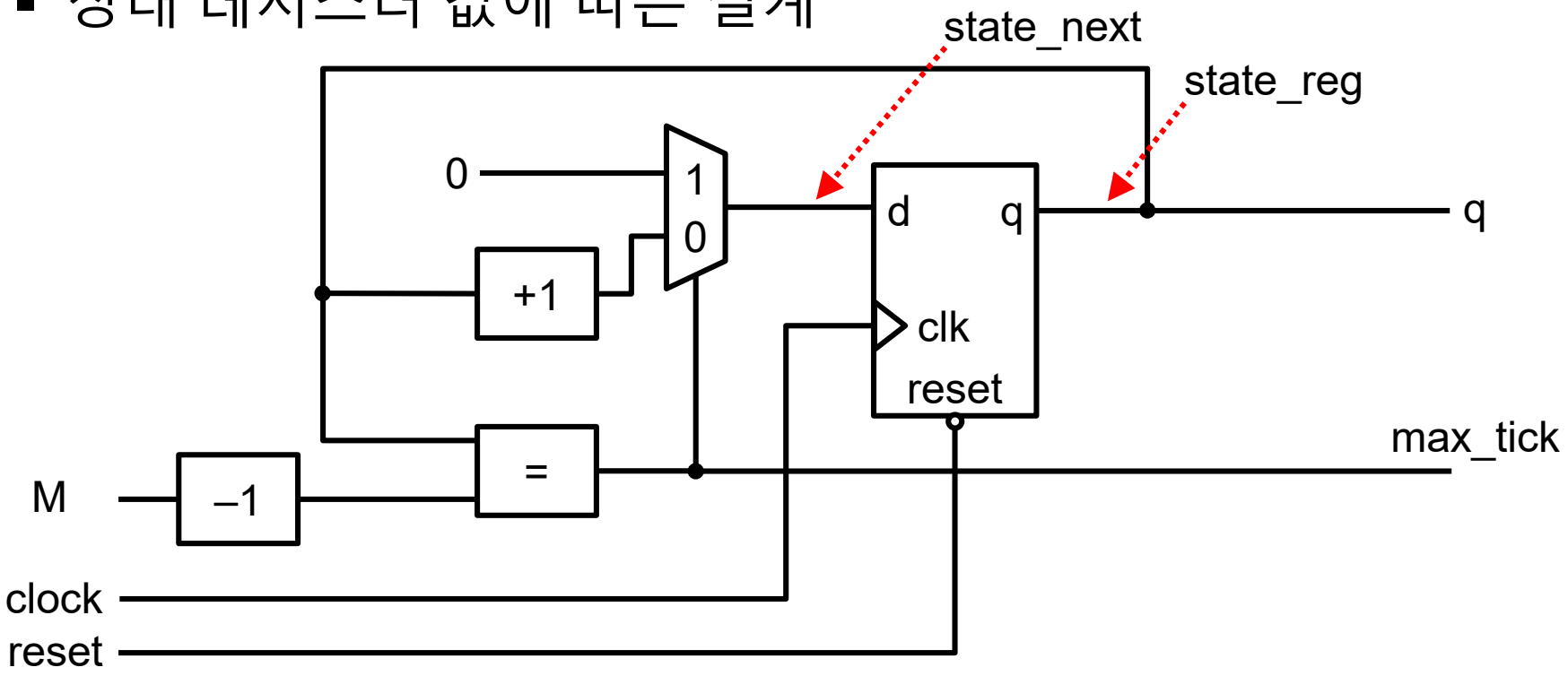
Universal 카운터

- Verilog 코드

```
module universal_counter
#(
   parameter N = 4
) (
   input           clk, rstb,
   input           sclr, load, en, up,
   input  [N-1:0] d,
   output [N-1:0] q,
   output          max_tick, min_tick
);

reg [N-1:0] state_reg;
reg [N-1:0] state_next;

// state register
always @(posedge clk or negedge rstb) begin
   if (~rstb) state_reg <= 0;
   else       state_reg <= state_next;
end

// next-state logic
always @* begin
if (sclr)          state_next = 0;
else if (load)     state_next = d;
else if (en&up)    state_next = state_reg + 1;
else if (en&~up)   state_next = state_reg - 1;
else               state_next = state_reg;
end

// output logic
assign q = state_reg;
assign max_tick = &state_reg;
assign min_tick = |state_reg;

endmodule
```

Mod-M 카운터

- 0, 1, 2, …, M – 1, 0, 1, 2, …, M – 1
- 상태 레지스터 값에 따른 설계



Mod-M 카운터

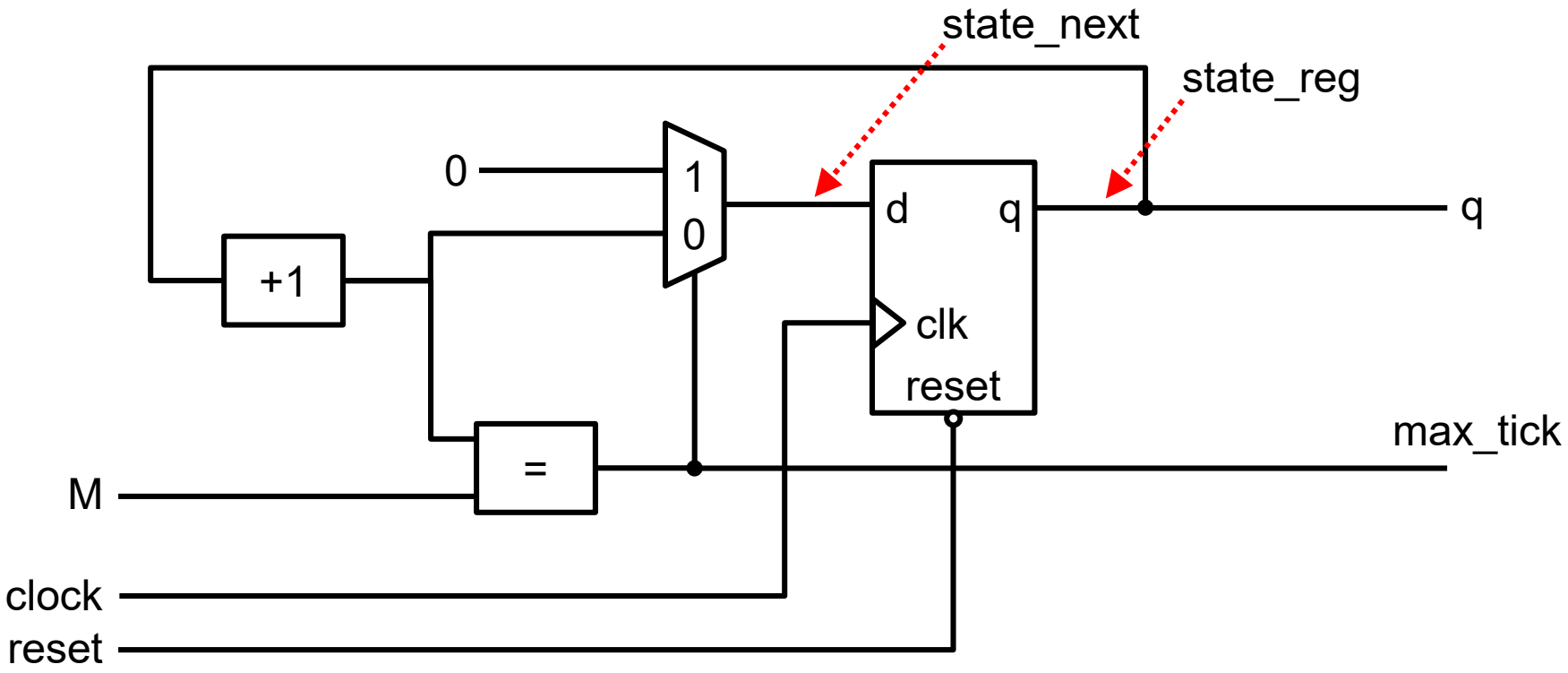
- 상태 레지스터 값에 따른 설계

```
module modm_counter_1
#(
   parameter M = 10,
   parameter N = $clog2(M)
) (
   input           clk, rstb,
   output [N-1:0] q,
   output          max_tick
);
```

```
reg [N-1:0] state_reg;
wire [N-1:0] state_next;

// state register
always @(posedge clk or negedge rstb) begin
   if (~rstb) state_reg <= 0;
   else       state_reg <= state_next;
end

// output logic
assign q = state_reg;
assign max_tick = (state_reg == (M-1)) ? 1'b1 : 1'b0;

// next-state logic
assign state_next = max_tick ? 0 : state_reg+1;

endmodule
```

Mod-M 카운터

- 상태 레지스터의 **다음 값**에 따른 설계



Mod-M 카운터

- 상태 레지스터의 **다음 값**에 따른 설계

```
module modm_counter_2
#(
   parameter M = 10,
   parameter N = $clog2(M)
) (
   input           clk, rstb,
   output [N-1:0] q,
   output          max_tick
);
```

```
reg  [N-1:0] state_reg;
wire [N-1:0] state_next, state_inc;

// state register
always @(posedge clk or negedge rstb) begin
   if (~rstb) state_reg <= 0;
   else       state_reg <= state_next;
end

// output logic
assign q = state_reg;
assign max_tick = (state_inc == M) ? 1'b1 : 1'b0;

// next-state logic
assign state_inc = state_reg+1;
assign state_next = max_tick ? 0 : state_inc;

endmodule
```

Mod-M 카운터

- 합성결과

상태 레지스터 값에 따른 설계

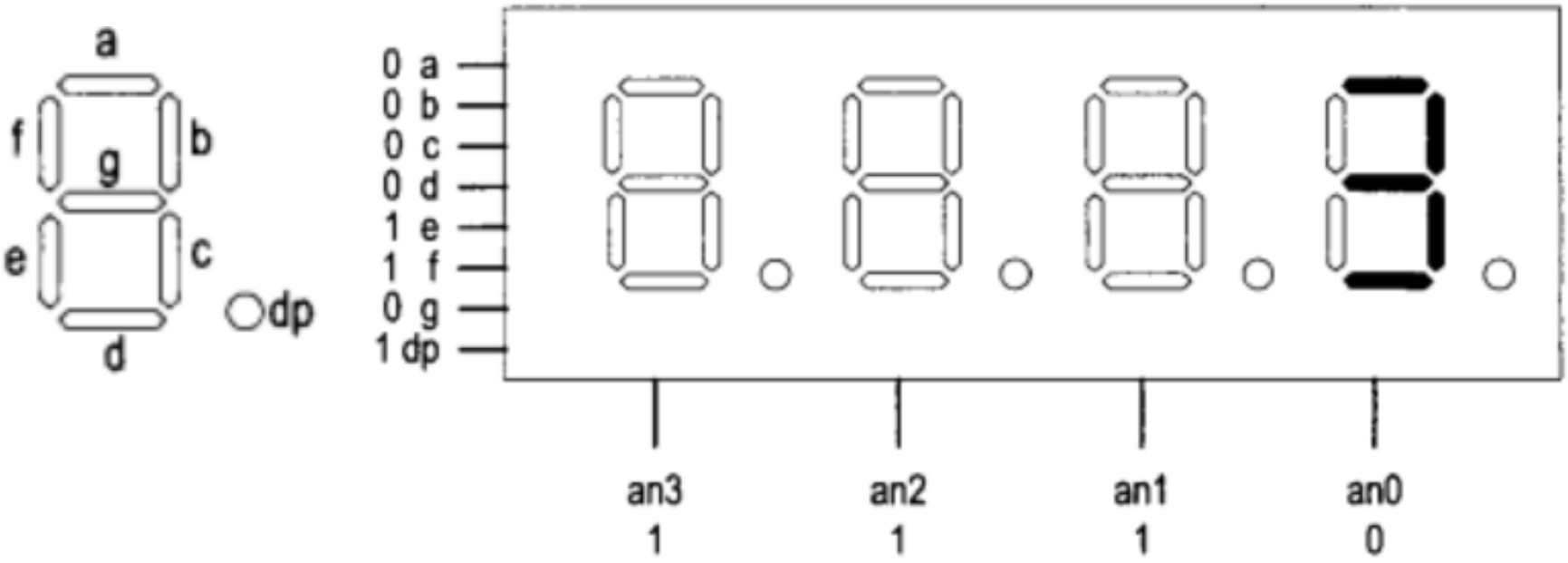
```
Number of wires:                 25
Number of wire bits:             34
Number of public wires:           6
Number of public wire bits:      15
Number of memories:               0
Number of memory bits:            0
Number of processes:              0
Number of cells:                 28
  DFFSR                           4
  NAND                            8
  NOR                             8
  NOT                             8

Chip area for module '\modm_counter_1': 160.000000
  of which used for sequential elements: 72.000000 (45.00%)
```

상태 레지스터의 다음 값에 따른 설계

```
Number of wires:                 22
Number of wire bits:             31
Number of public wires:           6
Number of public wire bits:      15
Number of memories:               0
Number of memory bits:            0
Number of processes:              0
Number of cells:                 25
  DFFSR                           4
  NAND                            5
  NOR                             8
  NOT                             8

Chip area for module '\modm_counter_2': 148.000000
  of which used for sequential elements: 72.000000 (48.65%)
```

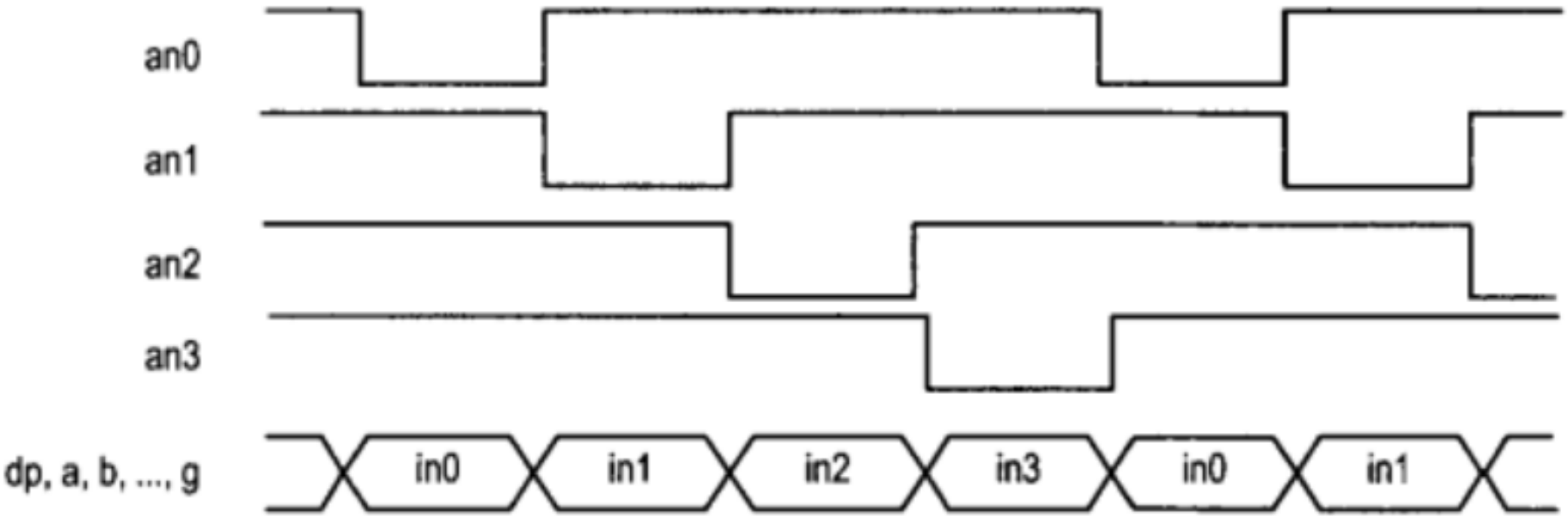
7-segment LED 제어

- Time-multiplexing 제어



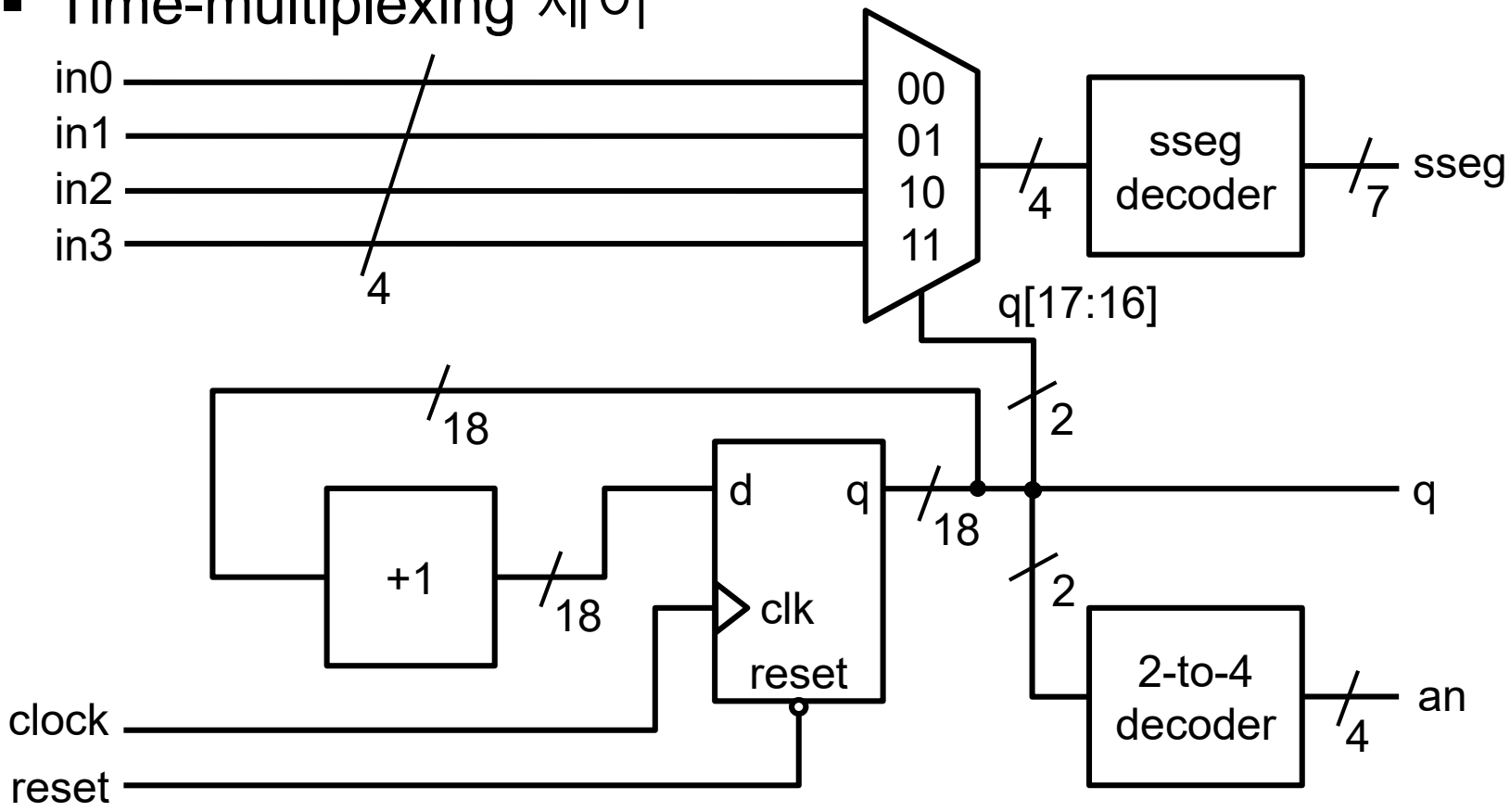
7-segment LED 제어

- Time-multiplexing 제어



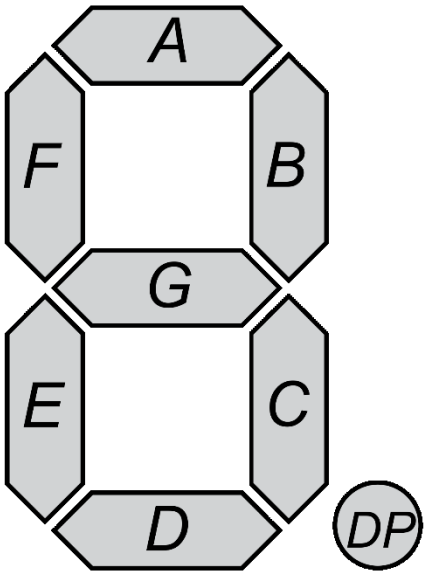
7-segment LED 제어

- Time-multiplexing 제어



7-segment LED 제어

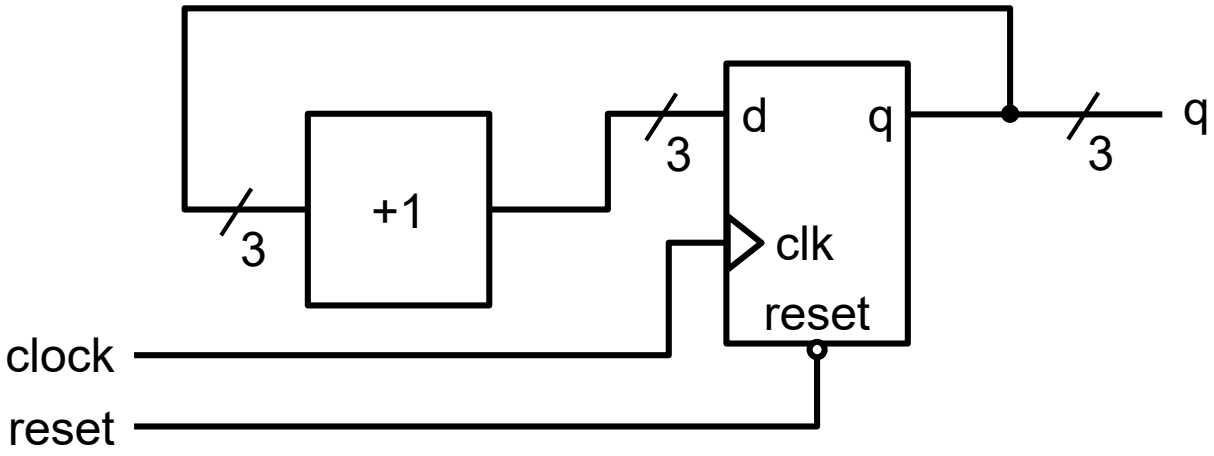
- sseg decoder



숫자	Active high							Active low						
	A	B	C	D	E	F	G	A	B	C	D	E	F	G
0	1	1	1	1	1	1	0	0	0	0	0	0	0	1
1	0	1	1	0	0	0	0	1	0	0	1	1	1	1
2	1	1	0	1	1	0	1	0	0	1	0	0	1	0
3	1	1	1	1	0	0	1	0	0	0	0	1	1	0
4	0	1	1	0	0	1	1	1	0	0	1	1	0	0
5	1	0	1	1	0	1	1	0	1	0	0	1	0	0
6	1	0	1	1	1	1	1	0	1	0	0	0	0	0
7	1	1	1	0	0	0	0	0	0	0	1	1	1	1
8	1	1	1	1	1	1	1	0	0	0	0	0	0	0
9	1	1	1	1	0	1	1	0	0	0	0	1	0	0

7-segment LED 제어

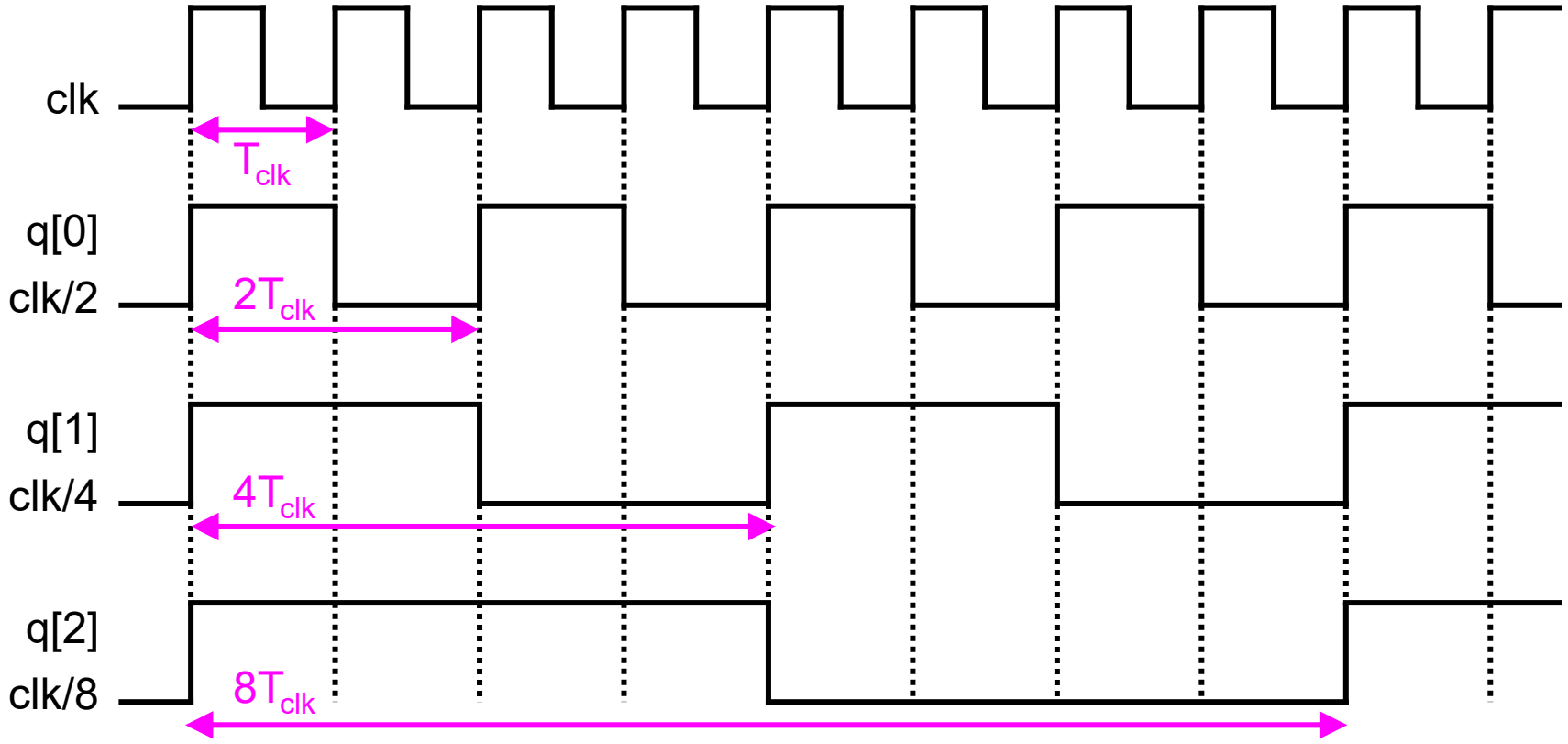
- 카운터 기반 clock 나누기
  - N비트 카운더 : q[N-1:0]
  - m 번째 비트의 주파수 $clk_m = clk/2^m$



Clock edge	q		
	q[2]	q[1]	q[0]
1st	0	0	0
2nd	0	0	1
3rd	0	1	0
4th	0	1	1
5th	1	0	0
6th	1	0	1
7th	1	1	0
8th	1	1	1

7-segment LED 제어

- 카운터 기반 clock 나누기



7-segment LED 제어

```
module sseg_multiplexing
#(
   parameter N = 18 // counter
) (
   input             clk, rstb,
   input       [3:0] in0, in1, in2, in3,
   input       [3:0] dp_in,
   output reg  [3:0] an,
   output reg  [7:0] sseg
);

reg [N-1:0] q_reg;
wire [N-1:0] q_next;
reg [3:0] in; reg dp;

// counter
always @(posedge clk or negedge rstb) begin
   if (~rstb) q_reg <= 0;
   else       q_reg <= q_next;
end

// next-state logic
assign q_next = q_reg+1;
```

```
// 4-to-1 multiplexer
always @* begin
   case (q_reg[N-1:N-2])
      2'b00: begin
         an = 4'b1110; in = in0; dp = dp_in[0];
      end
      2'b01: begin
         an = 4'b1101; in = in1; dp = dp_in[1];
      end
      2'b10: begin
         an = 4'b1011; in = in2; dp = dp_in[2];
      end
      default: begin
         an = 4'b0111; in = in3; dp = dp_in[3];
      end
   endcase
end

// sseg decoder
always @* begin
   ...
end

endmodule
```

최대 동작 주파수

t_0 clock의 rising edge 시작

t_1 state_reg 값 바꾸는 시점 ($t_0 + T_{cq}$)

t_2 state_next 값 바꾸기 시작 시점 ($t_1 + T_{next(min)}$)

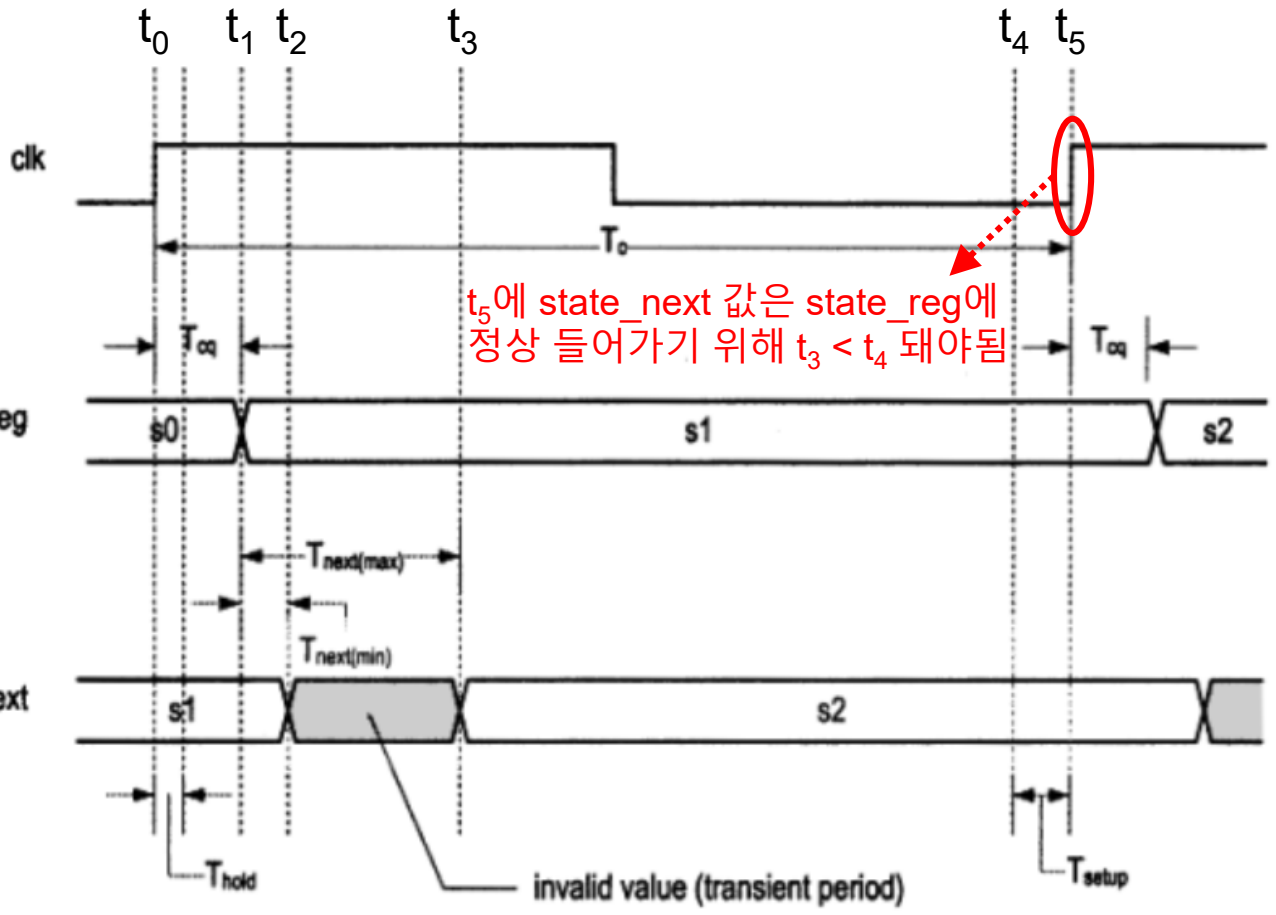
t_3 state_next 값 바꾸기 끝 시점 ($t_1 + T_{next(max)}$)

t_4 state_next 값 안정되어 야 하는 시점 ($t_4 + T_{setup} = t_5$)

t_5 다음 clock의 rising edge 시작

T_c clock cycle

동작 주파수 = $1/T_c$



최대 동작 주파수

- 최대 동작 주파수 추론

$$t_3 < t_4$$

$$t_3 = t_0 + T_{cq} + T_{next(max)} \qquad t_4 = t_5 - T_{setup} = t_0 + T_c - T_{setup}$$

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

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

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

최대 동작 주파수 = $1/T_{c(min)} = 1/(T_{cq} + T_{next(max)} + T_{setup})$

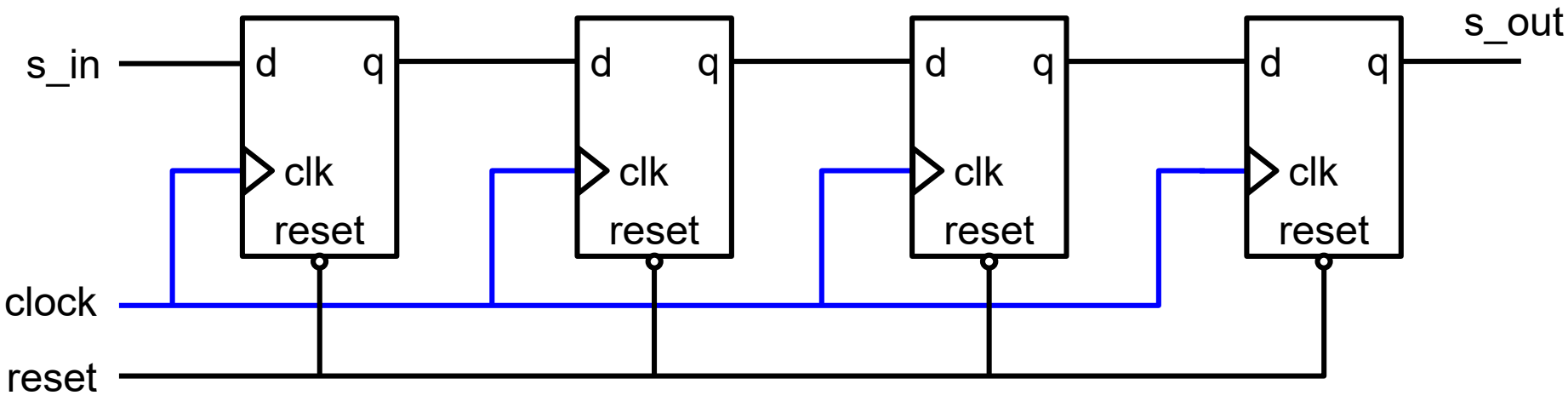
최대 동작 주파수

- 0.55-μm CMOS standard-cell technology

비트 크기	자주 쓰는 연산자										
	nand	xor	$>_a$	$>_d$	=	$+1_a$	$+1_d$	$+_a$	$+_d$	MUX	
8	8	22	25	68	26	27	33	51	118	21	Area (gate count)
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	
8	0.1	0.4	4.0	1.9	1.0	2.4	1.5	4.2	3.2	0.3	Delay (ns)
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	

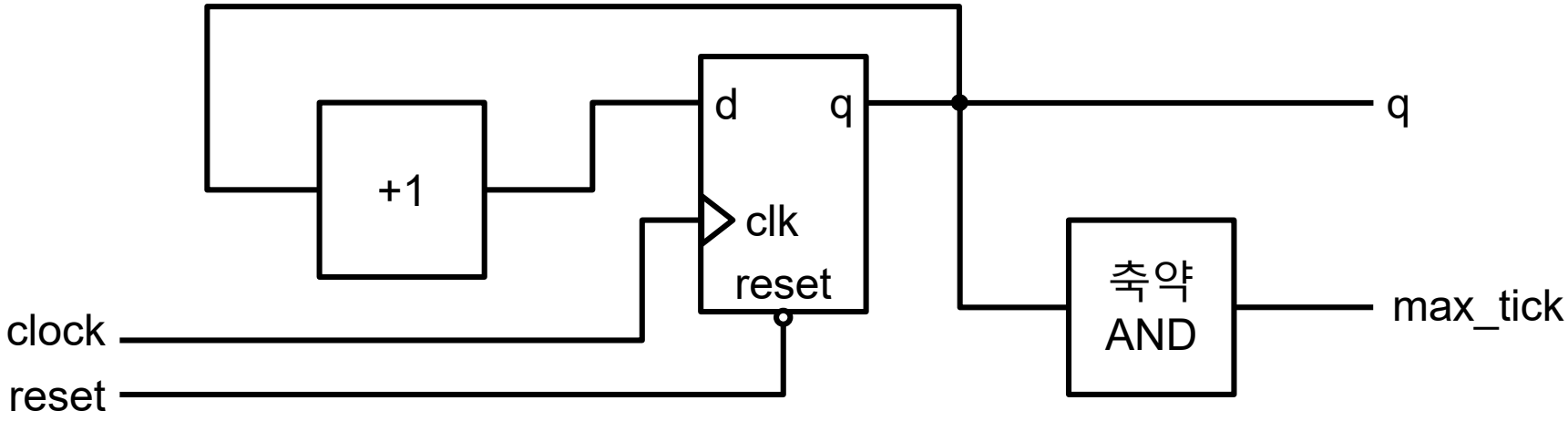
최대 동작 주파수

- 예시: free-running 시프트 레지스터
  - T_{cq} 및 T_{setup}는 flip-flop 특성이라 datasheet에 제작사 제공함
  - 예: T_{cq} = 1 ns, T_{setup} = 0.5 ns
  - next-state 로직 없어서 $T_{next(max)}$ = 0 ns
  - 최대 동작 주파수 = $1/(T_{cq} + T_{setup})$ = 1/1.5 = 666.7 (MHz)



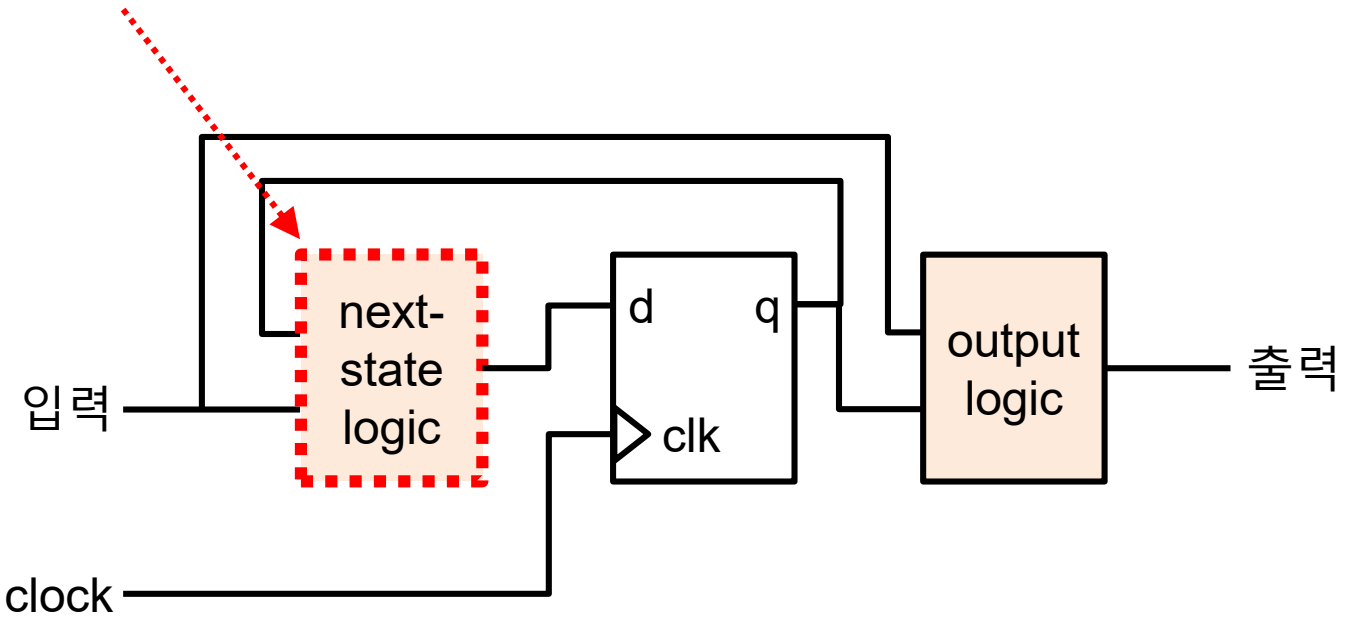
최대 동작 주파수

- 예시: free-running 카운터
  - 8비트 incrementor의 경우 $T_{next(max)}$ = 2.4 ns
  - 최대 동작 주파수(f_{max}) = 1/(T_{cq} + $T_{next(max)}$ + T_{setup}) = 1/3.9 = 256.4 (MHz)
  - 16비트 incrementor의 경우 $T_{next(max)}$ = 5.5 ns, f_{max} = 142.9 MHz
  - 32비트 incrementor의 경우 $T_{next(max)}$ = 11.6 ns, f_{max} = 76.3 MHz



최대 동작 주파수

- 최대 동작 주파수 올리는 방법
  - 최대 동작 주파수 = $1/(T_{cq} + T_{next(max)} + T_{setup})$
  - T_{cq} 및 T_{setup}는 flip-flop의 특성이라 건드릴 수 없음
  - $T_{next(max)}$는 next-state 로직 회로에 따르니 회로 잘 설계하면 줄일 수 있음



Hold time violation

t_0 clock의 rising edge 시작

t_1 state_reg 값 바꾸는 시점 ($t_0 + T_{cq}$)

t_2 state_next 값 바꾸기 시작 시점 ($t_1 + T_{next(min)}$)

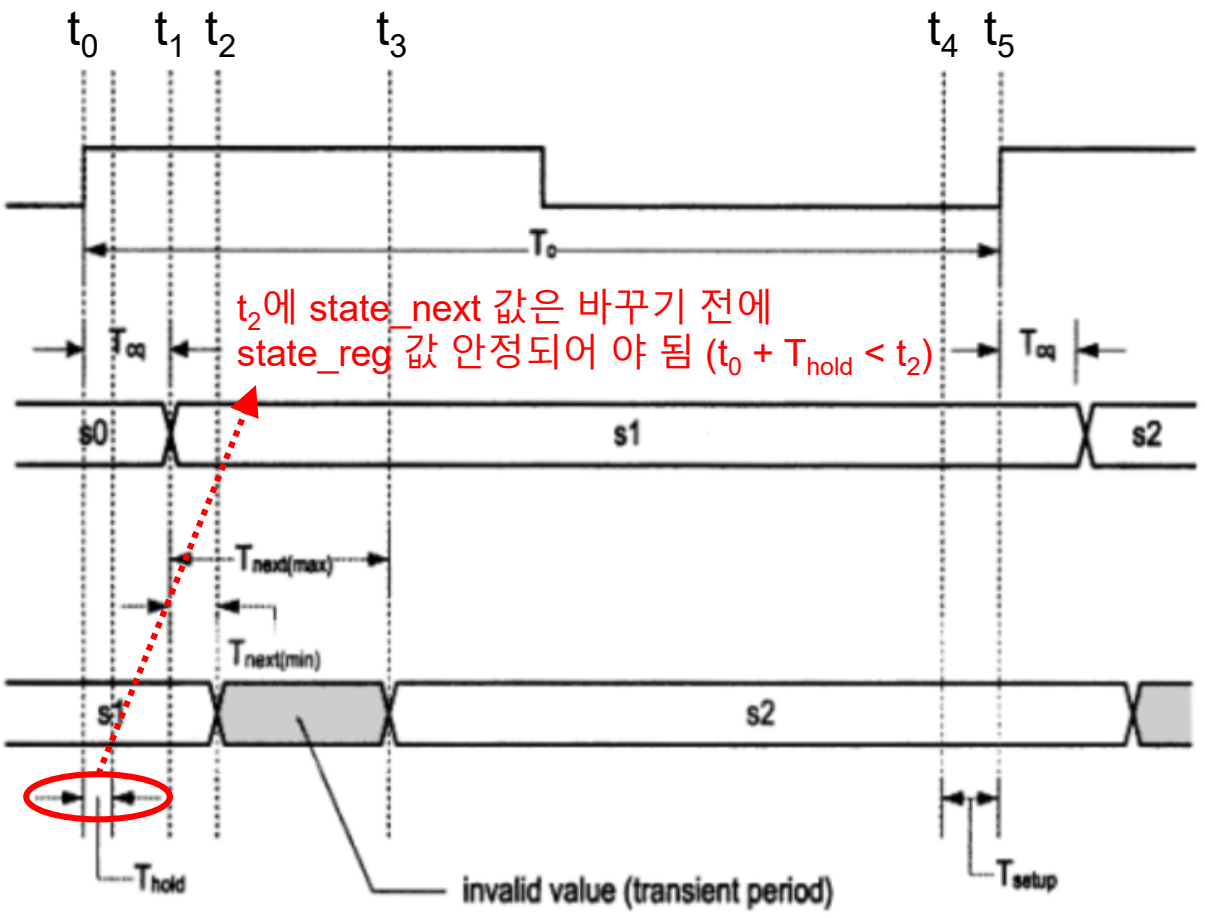
t_3 state_next 값 바꾸기 끝 시점 ($t_1 + T_{next(max)}$)

t_4 state_next 값 안정되어 야 하는 시점 ($t_4 + T_{setup} = t_5$)

t_5 다음 clock의 rising edge 시작

T_c clock cycle

동작 주파수 = $1/T_c$



Hold time violation

- Hold time 분석

$$t_0 + T_{hold} < t_2$$

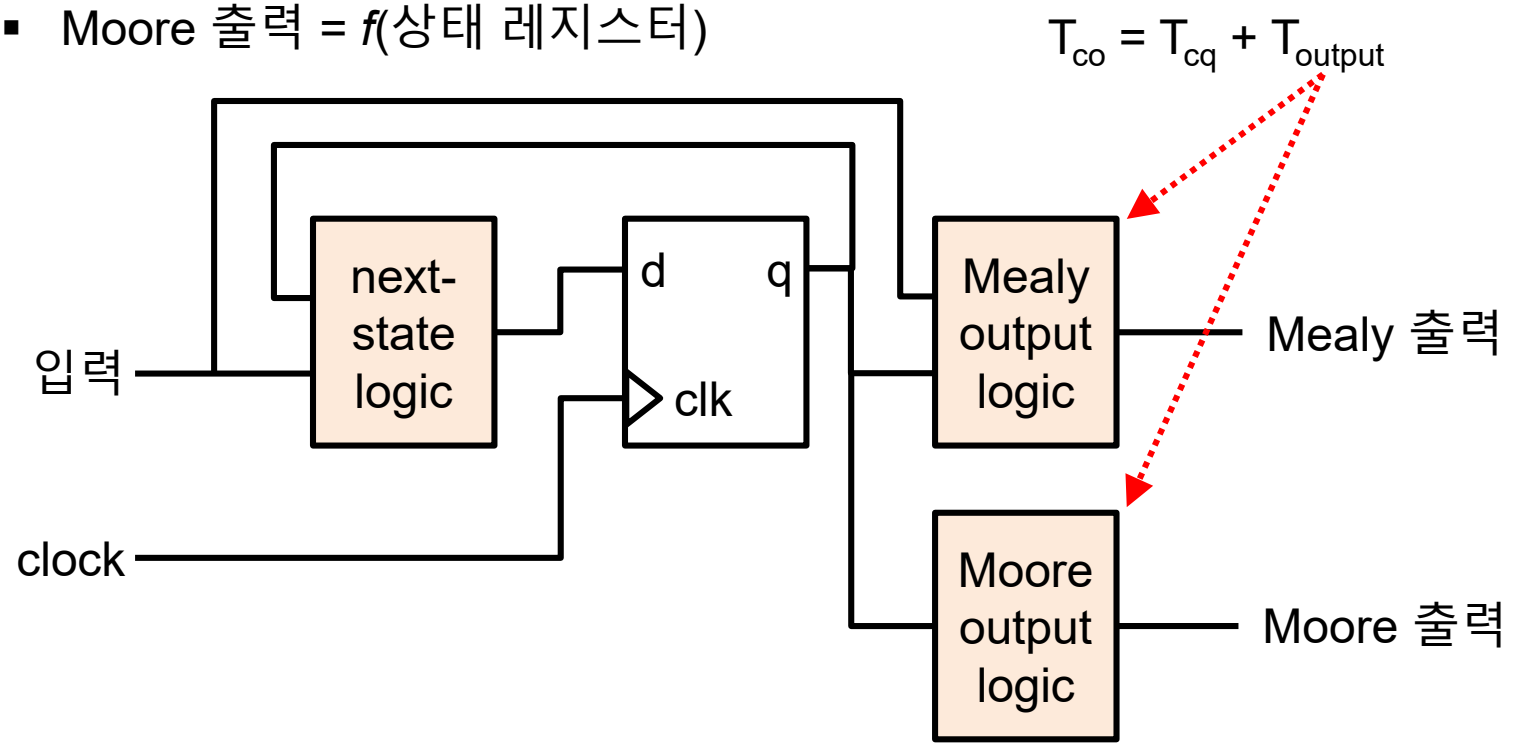
$$t_0 + T_{hold} < t_0 + T_{cq} + T_{next(min)}$$

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

- $T_{next(min)}$는 next-state 로직 회로에 따르니 회로 잘 설계하면 줄일 수 있음
- Free-running 시브트 레지스터의 경우
  - $T_{next(min)} = 0$ ns
  - $T_{hold} < T_{cq}$
  - 위 조건을 제작사 보증함

출력 관련 타이밍

- 출력 로직 회로
  - Mealy 출력 = *f*(상태 레지스터, 입력)
  - Moore 출력 = *f*(상태 레지스터)

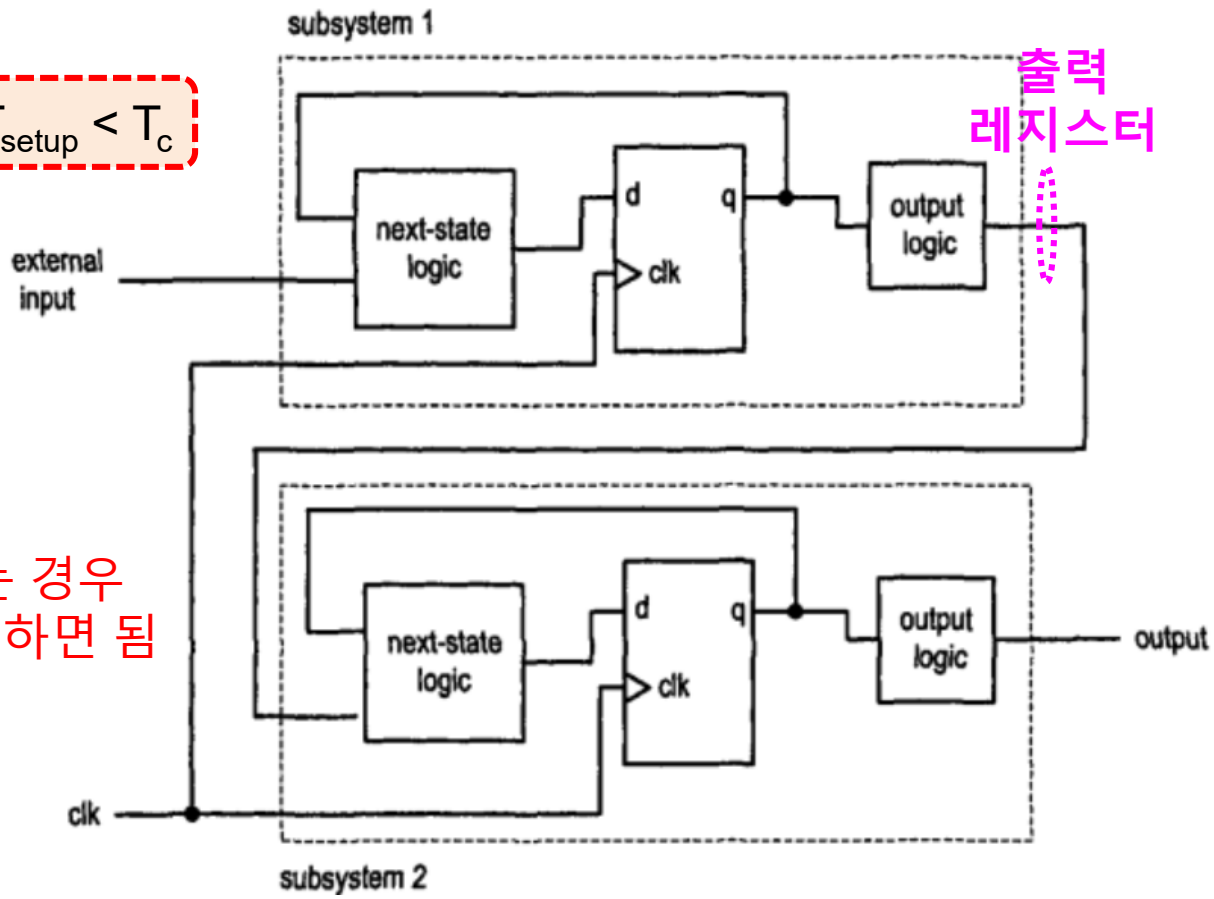


입력 관련 타이밍

- 회로 1 → 회로 2

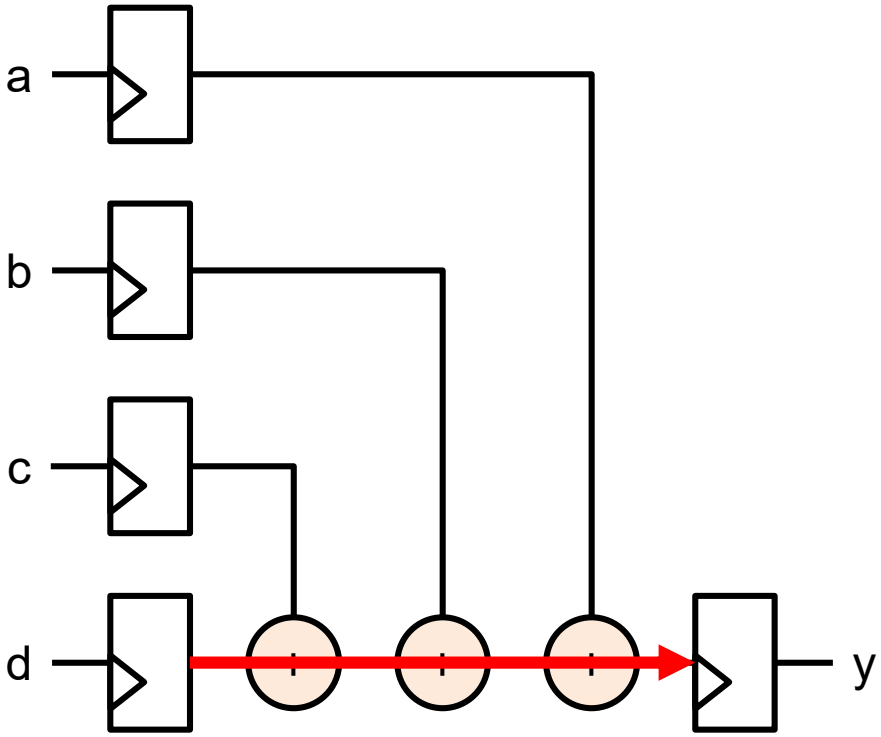
$T_{co(subsystem\ 1)} + T_{next(max)} + T_{setup} < T_c$

실제는 위 조건을 만족 못하는 경우
출력 레지스터를 추가로 사용하면 됨



Vivado 타이밍 분석

- y = a + b + c + d

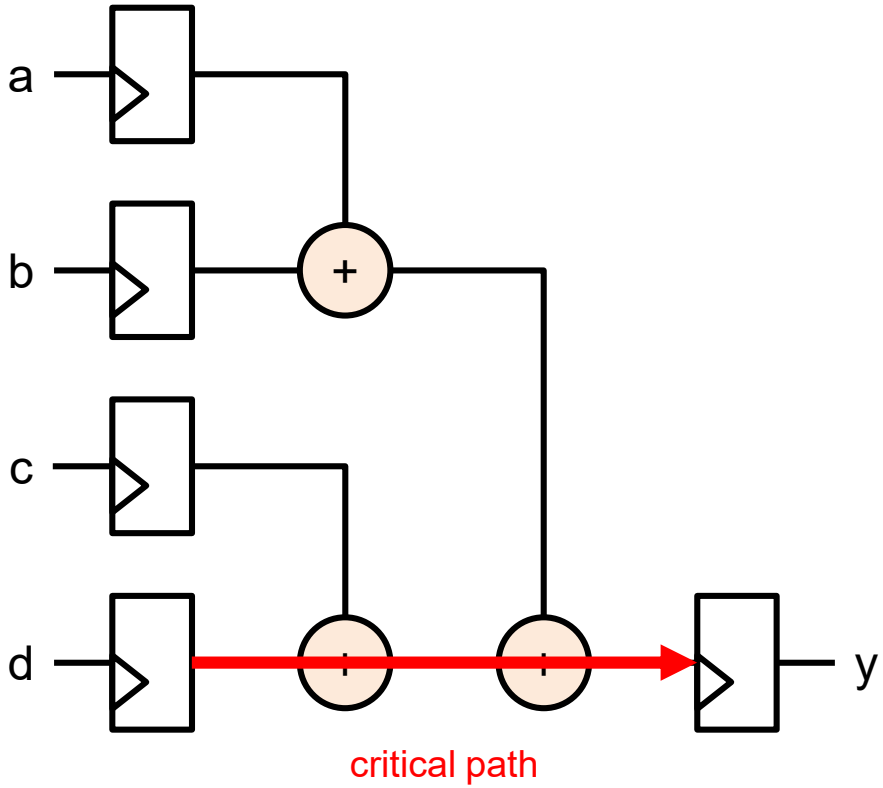


지연 시간 제일 느린 경로(critical path)

```
module adder_tree(
   input clk, rstb,
   input [15:0] a, b, c, d,
   output reg [17:0] y
);

reg [15:0] a_reg, b_reg, c_reg, d_reg;

always @(posedge clk or negedge rstb)
   if (~rstb) begin
      a_reg <= 0; b_reg <= 0; c_reg <= 0; d_reg <= 0;
      y <= 0;
   end
   else begin
      a_reg <= a; b_reg <= b; c_reg <= c; d_reg <= d;
      y <= a_reg + b_reg + c_reg + d_reg;
   end

endmodule
```

Vivado 타이밍 분석

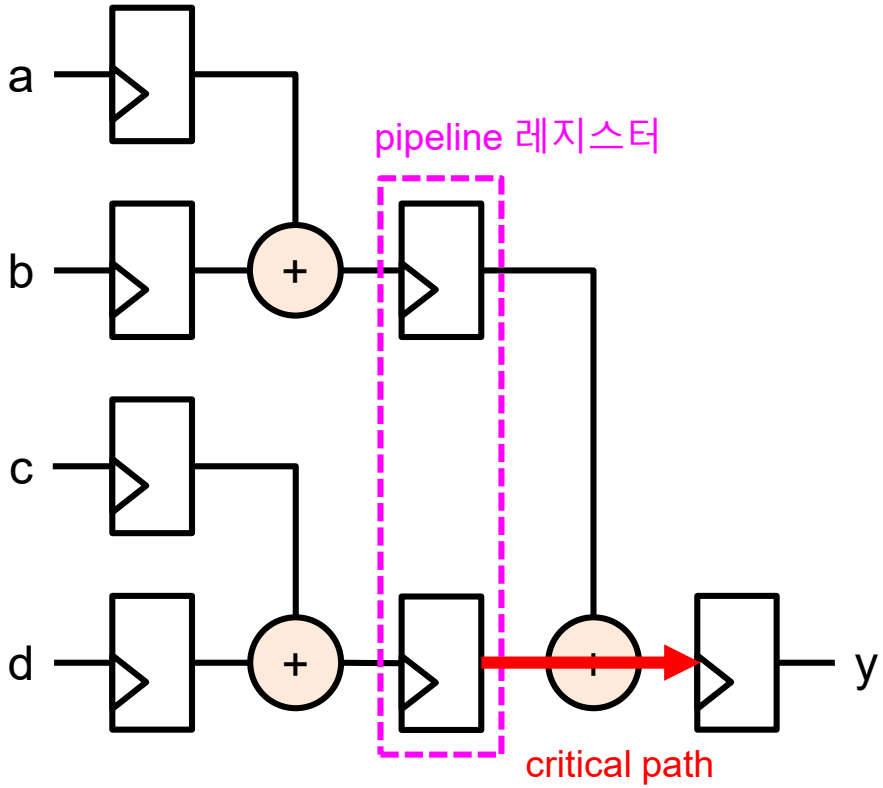
- Critical path 해결: 레이아웃 위한 설계




```
module adder_tree(
    input clk, rstb,
    input [15:0] a, b, c, d,
    output reg [17:0] y
);

reg [15:0] a_reg, b_reg, c_reg, d_reg;

always @(posedge clk or negedge rstb)
    if (~rstb) begin
        a_reg <= 0; b_reg <= 0; c_reg <= 0; d_reg <= 0;
        y <= 0;
    end
    else begin
        a_reg <= a; b_reg <= b; c_reg <= c; d_reg <= d;
        y <= (a_reg + b_reg) + (c_reg + d_reg);
    end

endmodule
```

Vivado 타이밍 분석

- Critical path 해결: pipeline 레지스터 추가




```
module adder_tree(
    input clk, rstb,
    input [15:0] a, b, c, d,
    output reg [17:0] y
);

reg [15:0] a_reg, b_reg, c_reg, d_reg;
reg [16:0] ab, cd;

always @(posedge clk or negedge rstb)
    if (~rstb) begin
        a_reg <= 0; b_reg <= 0; c_reg <= 0; d_reg <= 0;
        ab <= 0; cd <= 0;
        y <= 0;
    end
    else begin
        a_reg <= a; b_reg <= b; c_reg <= c; d_reg <= d;
        ab <= a_reg + b_reg; cd <= c_reg + d_reg;
        y <= ab + cd;
    end

endmodule
```