Lecture 07~08

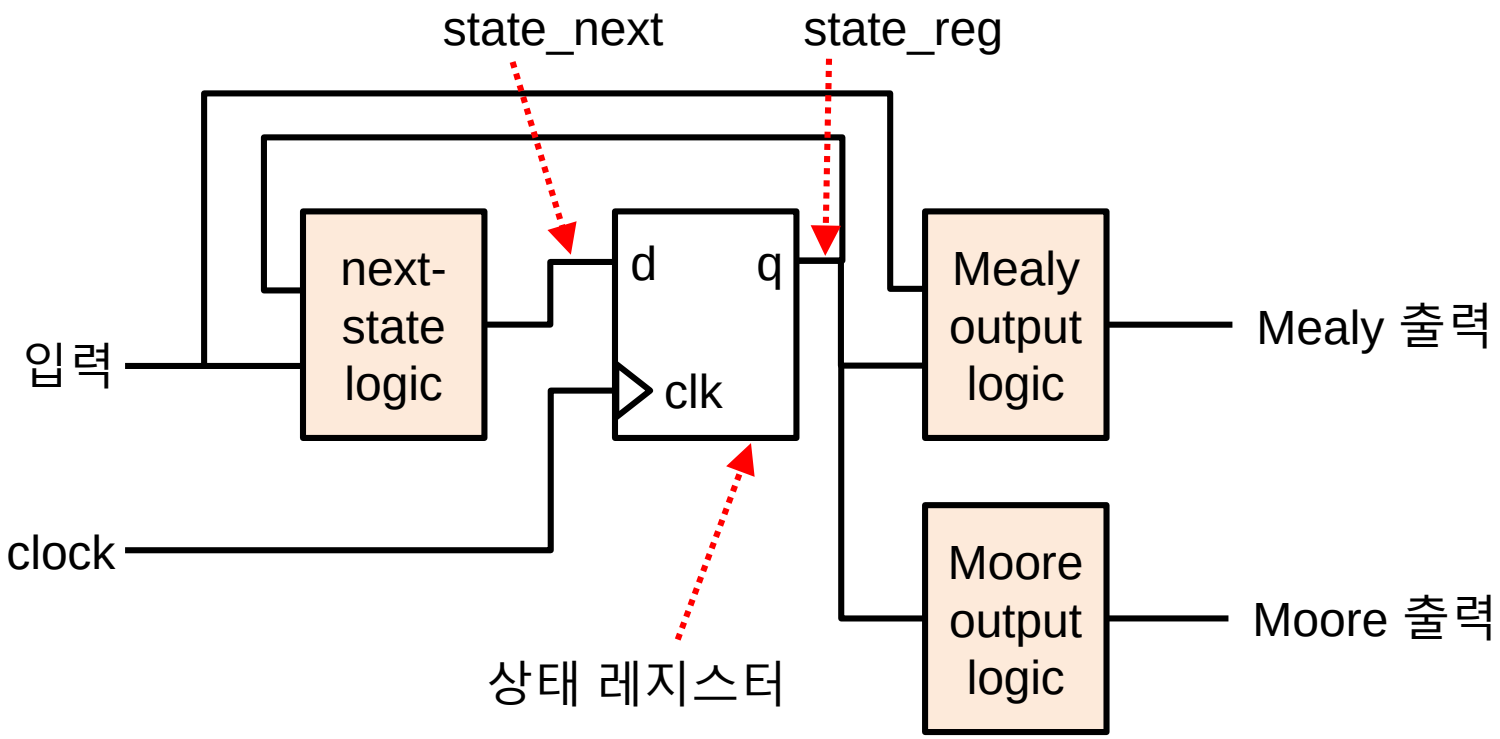
FSM 설계

FSM(유한 상태 기계)

- FSM : Finite State Machine
  - 순서회로
  - 복잡한 next-state 로직
    - 일반적으로 알고리즘 흐름도 통해 next-state 로직 설계함
  - 큰 디지털 시스템
    - Data path : 메모리, 기본 혹운 복잡한 연산을 하는 회로
    - Control path : data path의 컨트롤 신호 생성을 위한 FSM
  - 실용성 높음
  - FSM 종류
    - Mealy : 출력 = *f*(상태, 입력)
    - Moore : 출력 = *f*(상태)

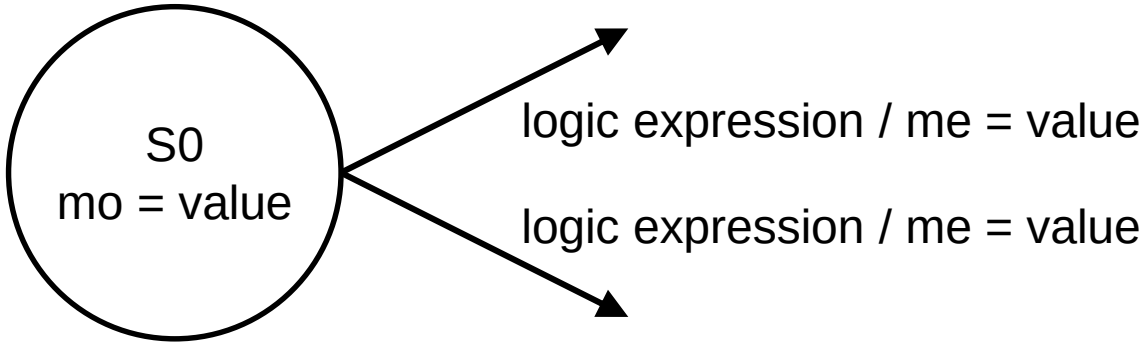
FSM(유한 상태 기계)

- FSM 블록도



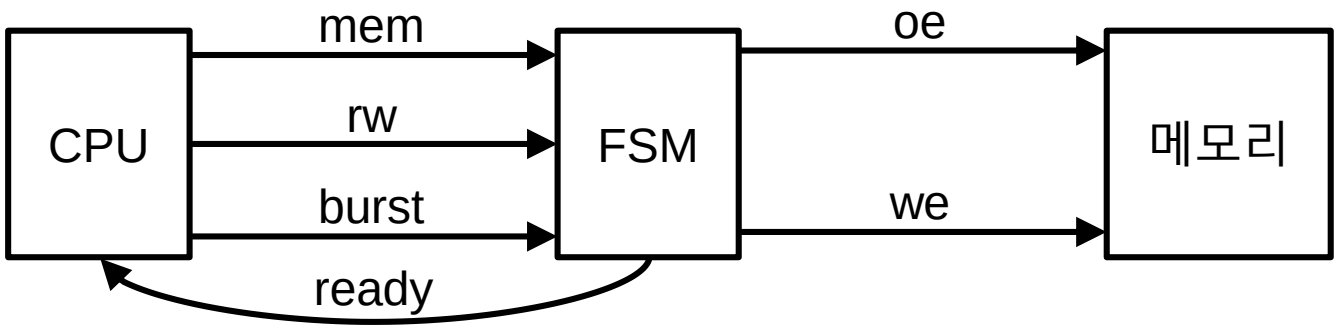
FSM 기술

- 상태도(state diagram) 혹은 ASM도로 기술
  - 입력, 출력 (Algorithmic State Machine)
  - 모두 상태
  - 상태 바꾸는 로직 (한 상태에서 다른 상태로 바꾸는 법 기술)
- 상태도 기술
  - 노드(node)로 표시
  - S0 : 상태 이름
  - mo = value : Moore 출력
  - 화살표 : 다른 상태로 바꿈
  - logic expression / me = value : 상태 바꾸는 로직 및 Mealy 추력

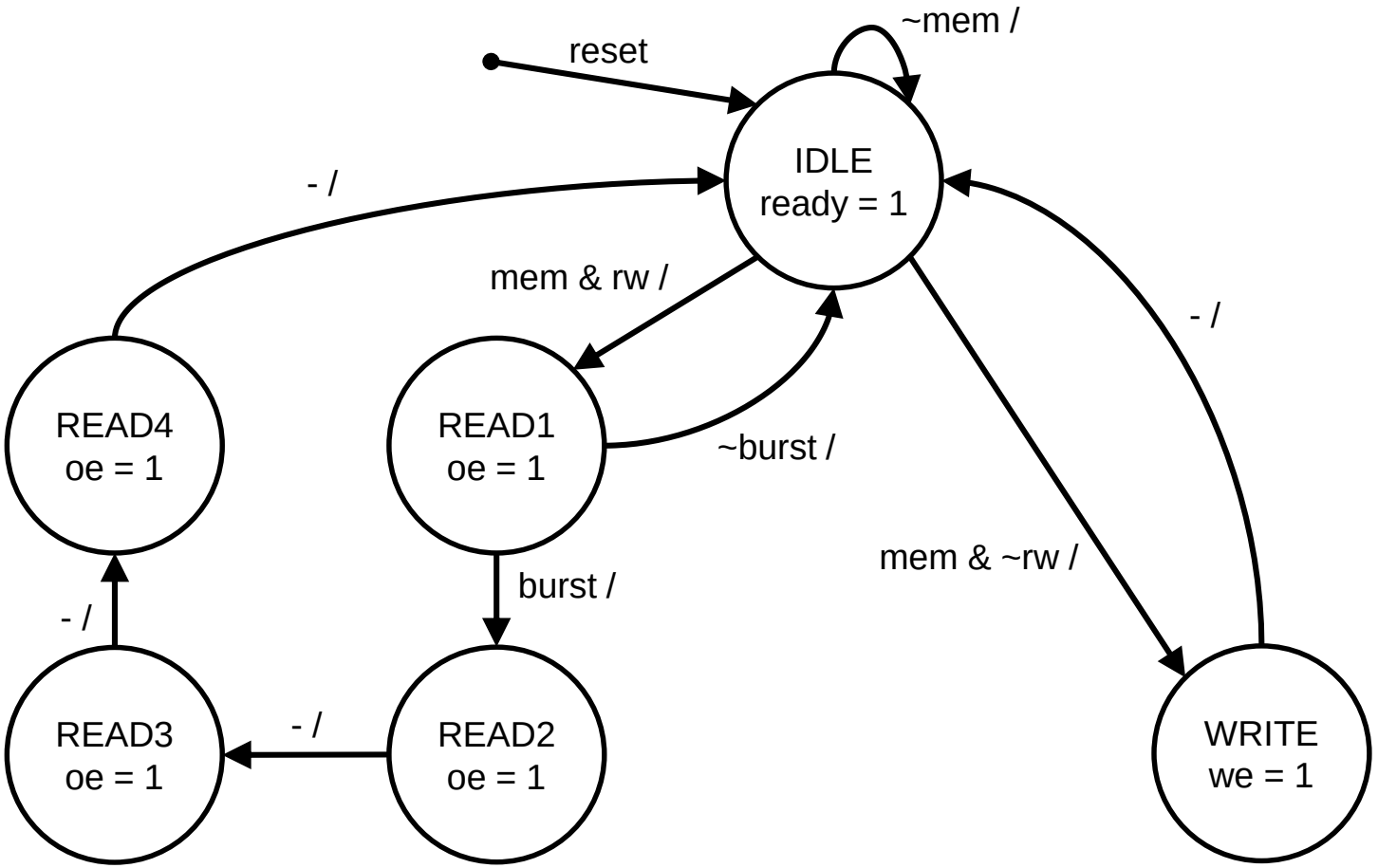


FSM 기술

- 예: 간소화 메모리 컨트롤러
  - CPU
    - mem : 1로 되면 메모리 접속 필요
    - rw : 1로 되면 메모리 읽기, 0로 되면 메모리 쓰기
    - burst : 1로 되면 메모리 연속 읽기 (4번)
  - 메모리
    - we : 1로 되면 메모리에 다이터 쓰기 가능
    - oe : 1로 되면 메모리에서 데이터 읽기 가능

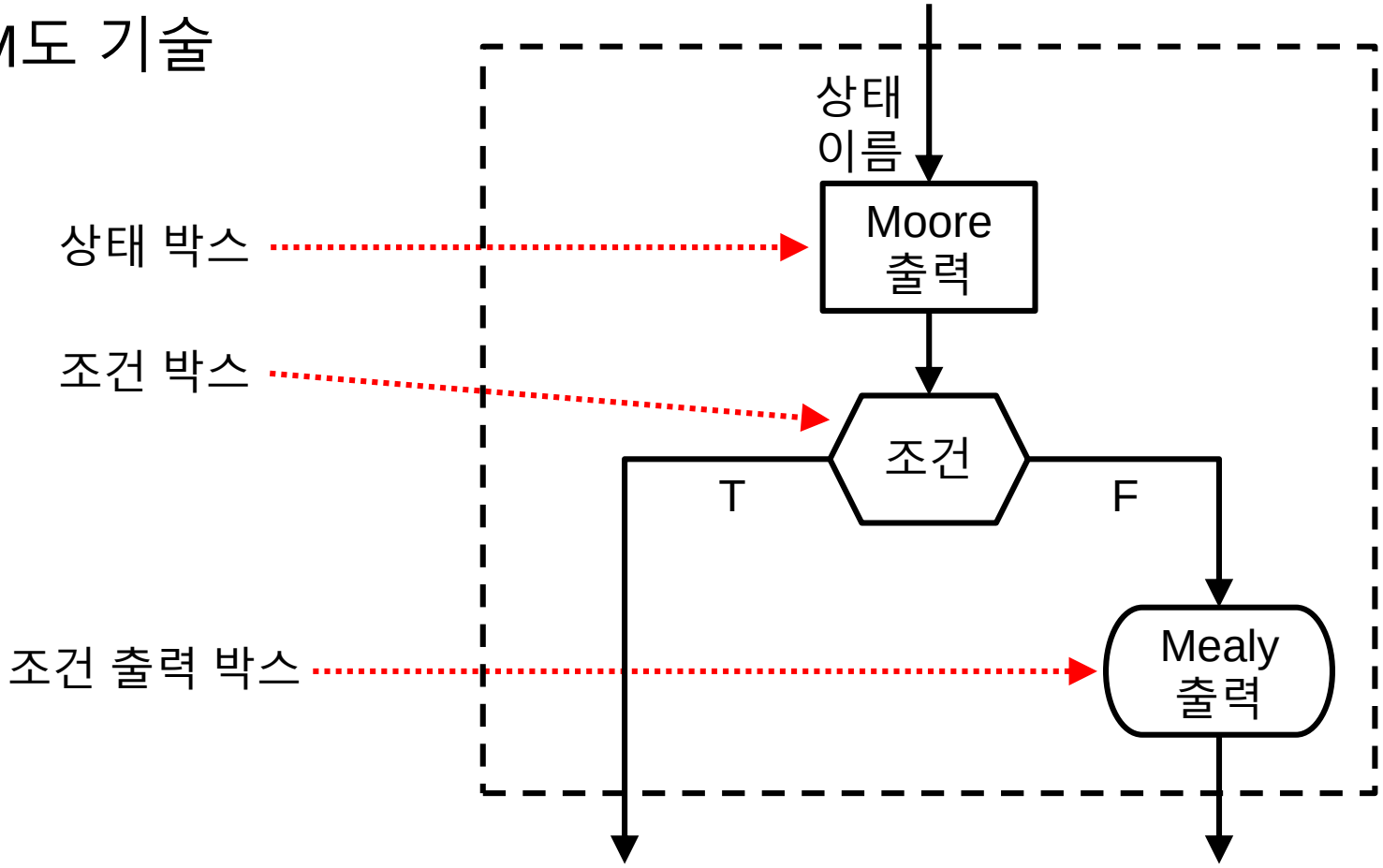


FSM 기술


reset
~mem /
IDLE
ready = 1
- /
mem & rw /
- /
READ4
oe = 1
READ1
oe = 1
~burst /
mem & ~rw /
burst /
- /
READ3
oe = 1
- /
READ2
oe = 1
WRITE
we = 1

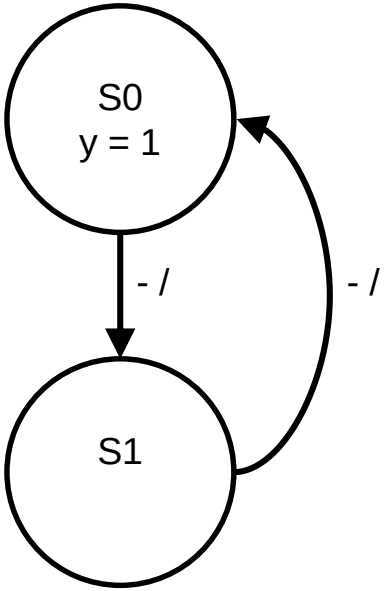
FSM 기술

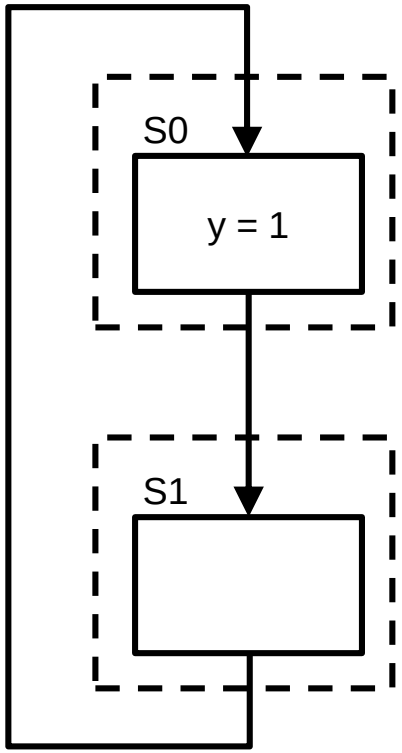
- ASM도 기술



FSM 기술

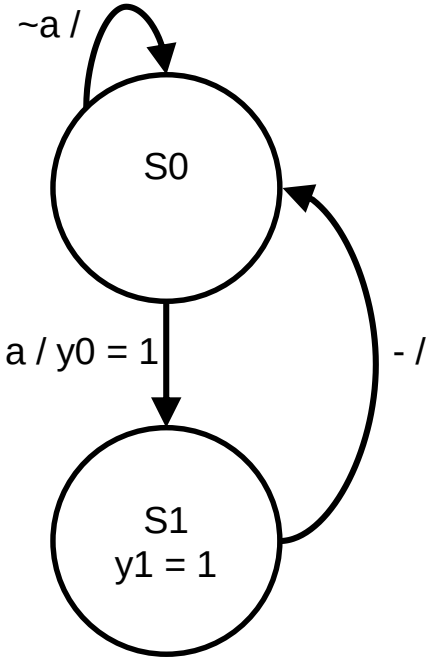
- 예: 상태도 및 ASM도

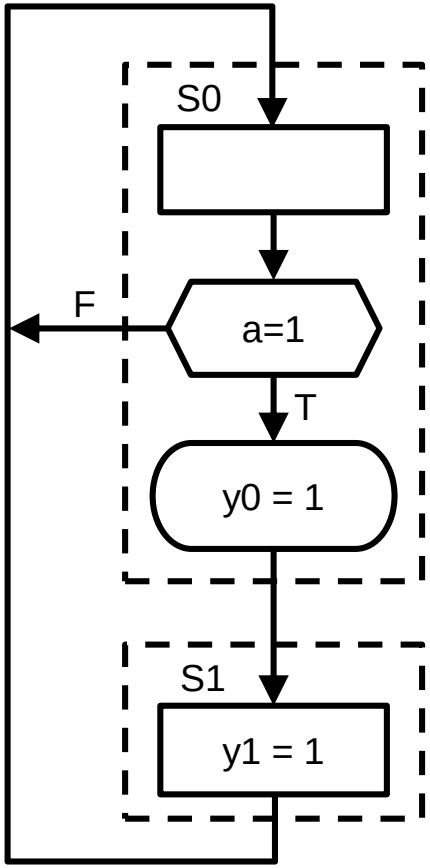




FSM 기술

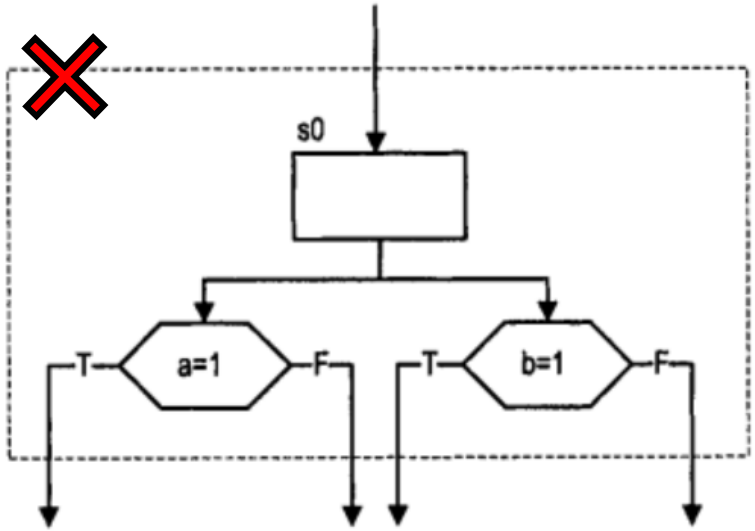
- 예: 상태도 및 ASM도

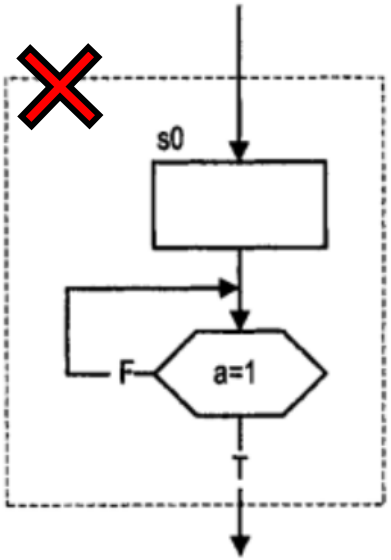


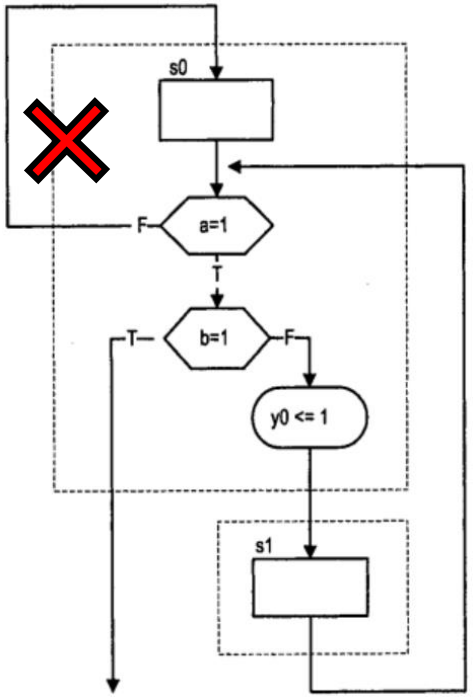


FSM 기술

- ASM도 사용 시 유의 사항
  - 다른 상태로 바꿀 때 상태 박스에 먼저 들어가 야 함
  - 한 입력 조합에 대해 동시의 여러 출력 경로 활성화하면 안 됨

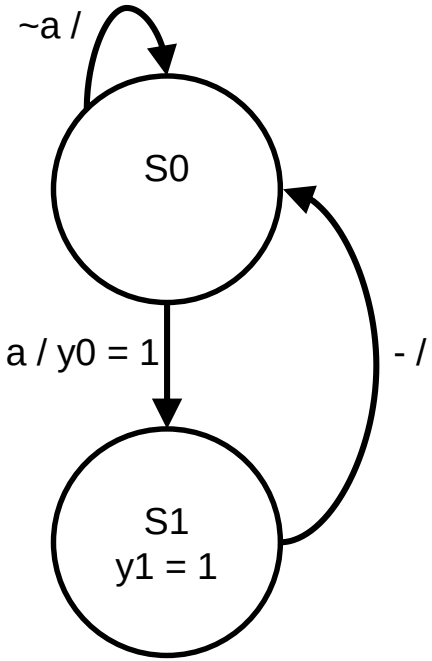


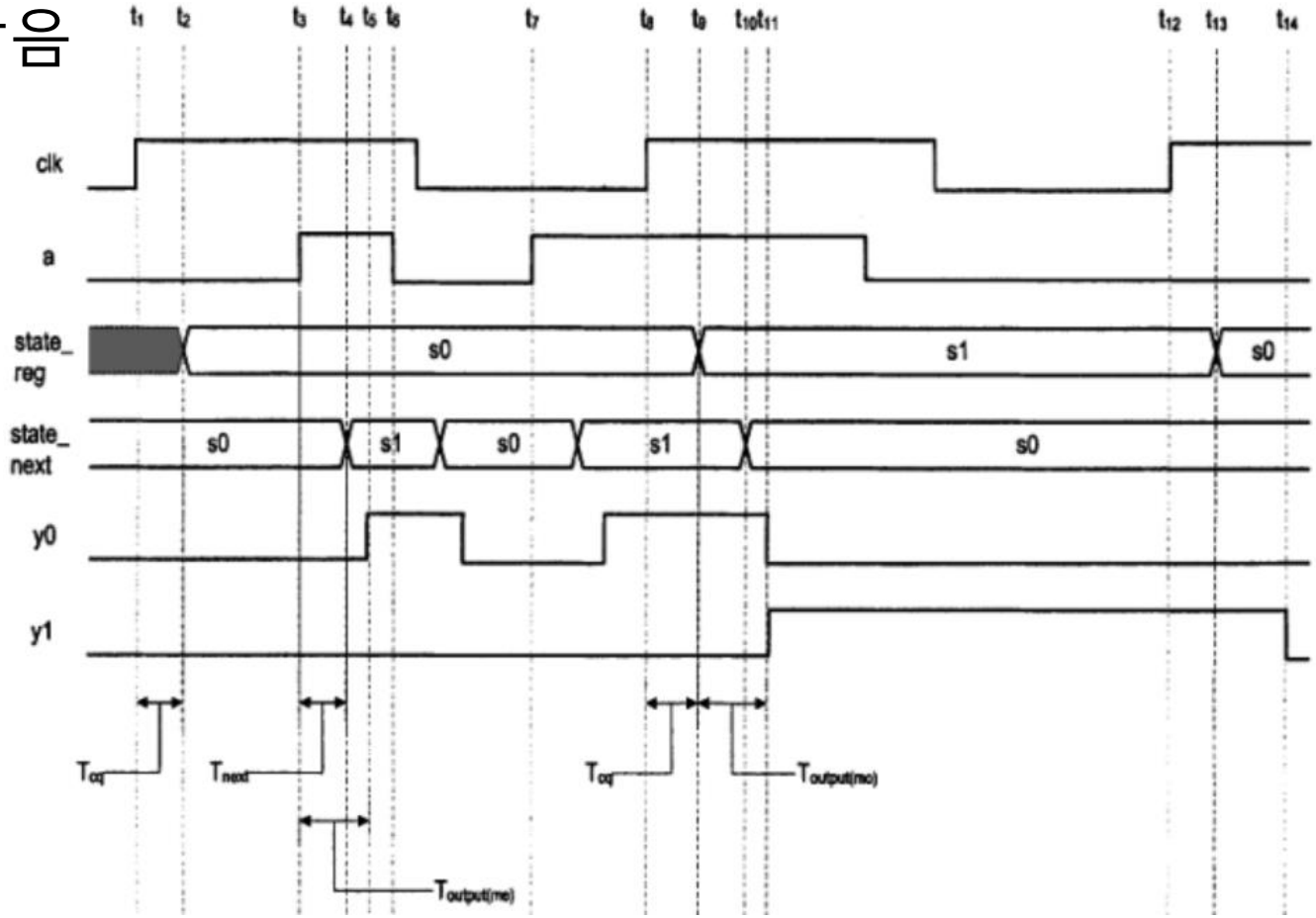




FSM 타이밍 분석

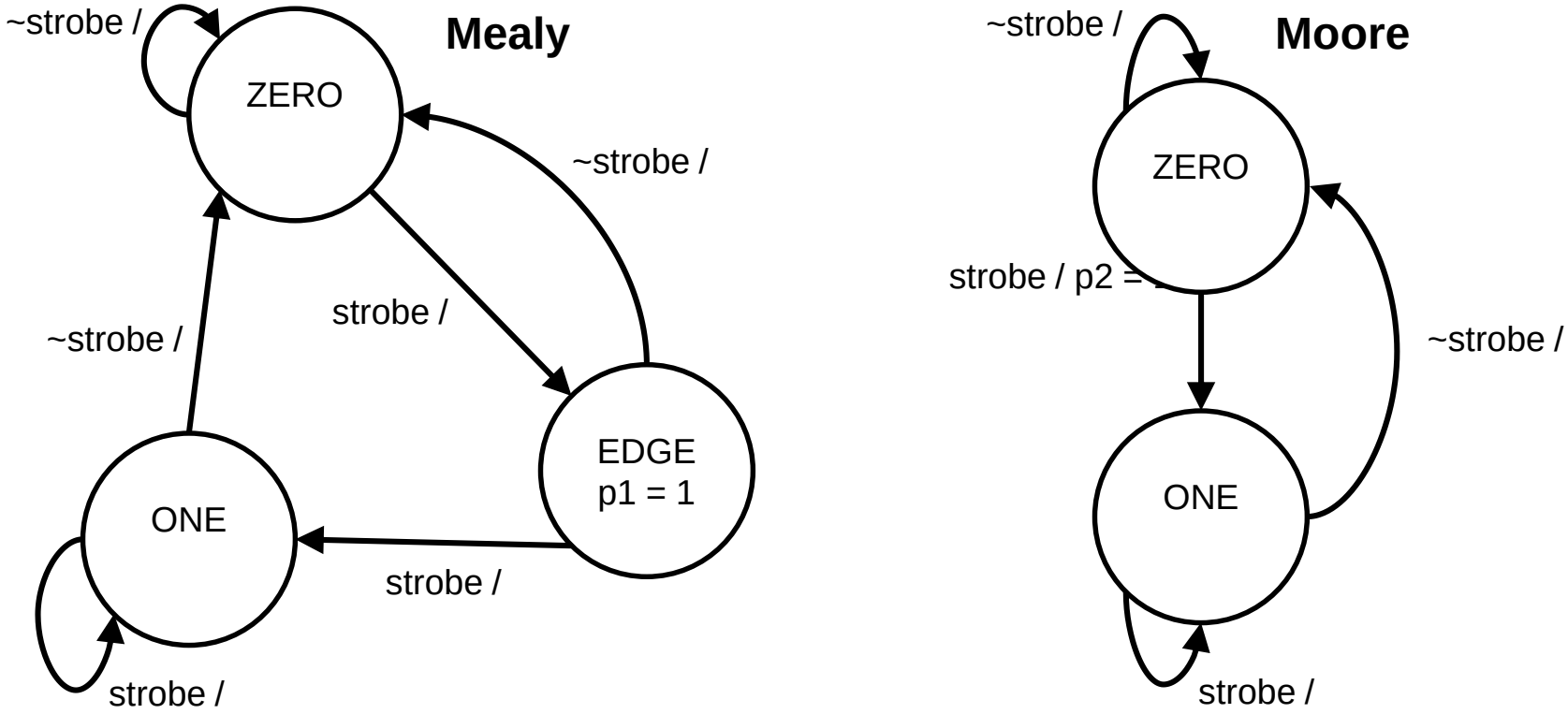
- 순서회로와 같음



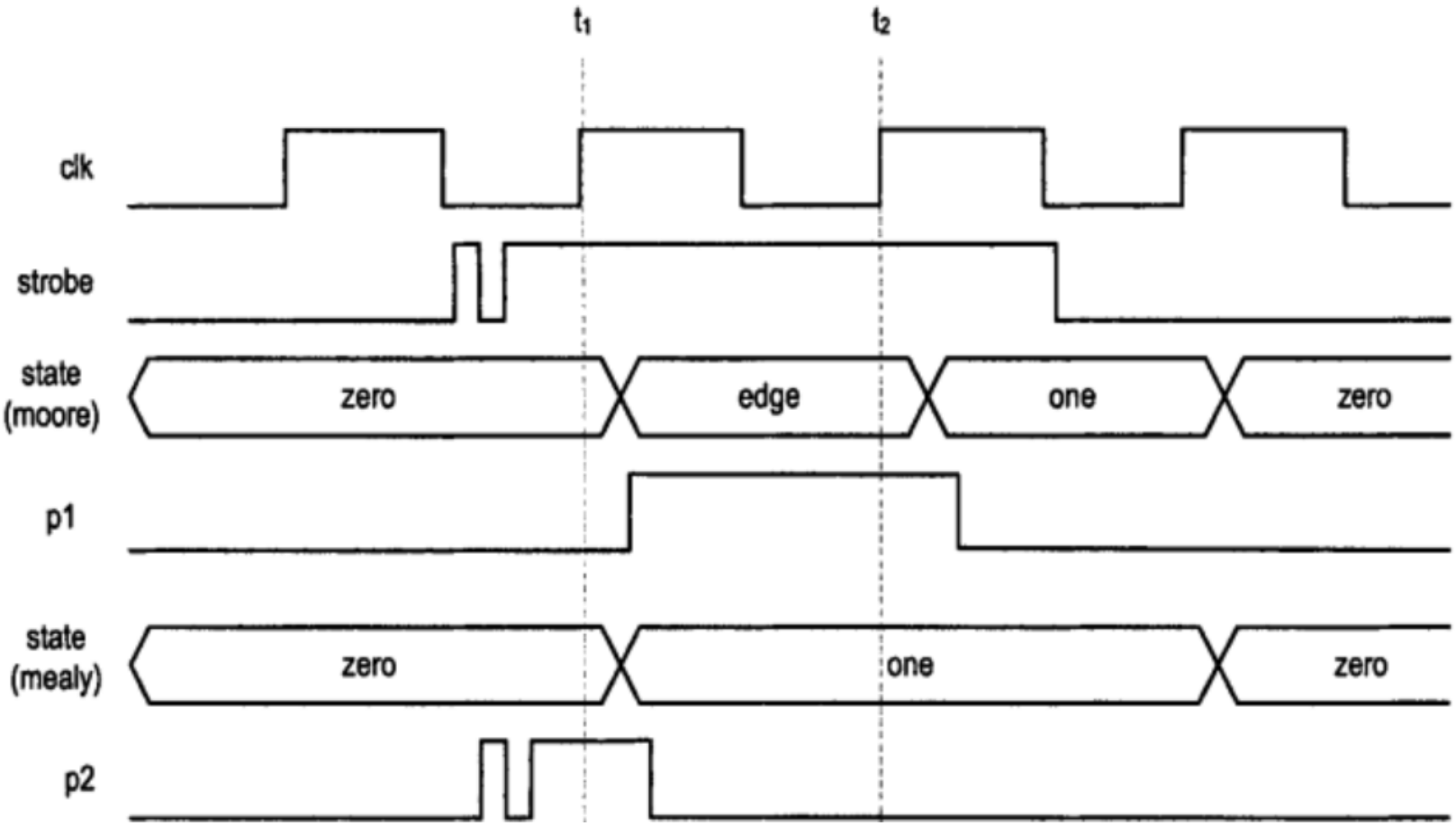


Mealy FSM vs. Moore FSM

- 예: 에지 검출 회로



Mealy FSM vs. Moore FSM



Mealy FSM vs. Moore FSM

- Mealy FSM
  - Moore FSM보다 상태 개수 적음 → 회로 크기 작음
  - 입력의 변경에 대한 빠른 응답 → “glitch”에 민감함
  - Moore FSM보다 edge-sensitive 제어 신호 생성에 적합함
- Moore FSM
  - Mealy FSM보다 상태 개수 많음 → 회로 크기 큼
  - Clock의 에지에만 상태 바꿔 출력 신호 생성함 → “glitch”에 민감하지 않음
  - Mealy FSM보다 level-sensitive 제어 신호 생성에 적합함

Mealy FSM vs. Moore FSM

- Verilog 코드 (Mealy FSM)

```
module edge_detector_mealy(
   input       clk, rstb,
   input       strobe,
   output reg p2
);

// state
localparam ZERO = 1'b0,
           ONE  = 1'b1;

reg state_reg, 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 and output logic
always @* begin
   state_next = state_reg;
   p2 = 1'b0;
   case (state_reg)
      ZERO: if (strobe) begin
         state_next = ONE;
         p2 = 1'b1;
      end
      ONE: if (~strobe) state_next = ZERO;
      default: state_next = ZERO;
   endcase
end

endmodule
```

Mealy FSM vs. Moore FSM

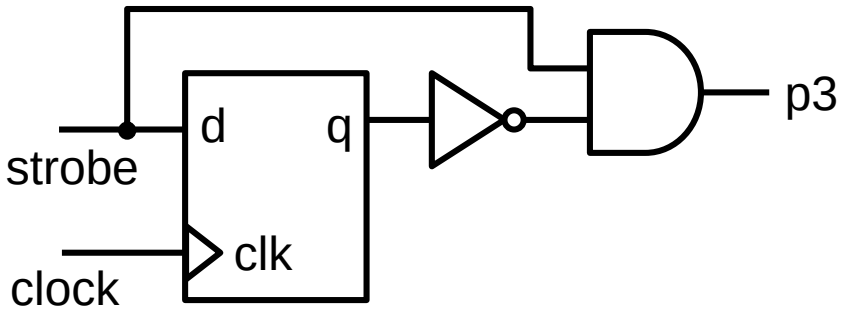
- Verilog 코드 (Moore FSM)

```
module edge_detector_moore(
   input       clk, rstb,
   input       strobe,
   output reg p1
);

// state
localparam [1:0] ZERO = 2'b00,
                 EDGE = 2'b01,
                 ONE  = 2'b10;

reg [1:0] state_reg, 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 and output logic
always @* begin
   state_next = state_reg;
   p1 = 1'b0;
   case (state_reg)
      ZERO: if (strobe) state_next = EDGE;
      EDGE: begin
         p1 = 1'b1;
         state_next = strobe ? ONE : ZERO;
      end
      ONE: if (~strobe) state_next = ZERO;
      default: state_next = ZERO;
   endcase
end

endmodule
```

실제 에지 검출 회로

- 회로 + Verilog 코드




```
module edge_detector_opt(
    input  clk, rstb,
    input  strobe,
    output p3
);

reg strobe_delay;

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

// output logic
assign p3 = strobe & ~strobe_delay;

endmodule
```

에지 검출 회로 비교

Mealy

```
Number of wires:                  8
Number of wire bits:              8
Number of public wires:           6
Number of public wire bits:       6
Number of memories:               0
Number of memory bits:            0
Number of processes:              0
Number of cells:                  4
  DFFSR                           1
  NOR                             1
  NOT                             2

Chip area for module '\edge_detector_mealy': 28.000000
  of which used for sequential elements: 18.000000 (64.29%)
```

Moore

```
Number of wires:                 14
Number of wire bits:             15
Number of public wires:           5
Number of public wire bits:       6
Number of memories:               0
Number of memory bits:            0
Number of processes:              0
Number of cells:                 12
  DFFSR                           2
  NAND                            3
  NOR                             3
  NOT                             4

Chip area for module '\edge_detector_moore': 72.000000
  of which used for sequential elements: 36.000000 (50.00%)
```

실제 사용한 회로

```
Number of wires:                  7
Number of wire bits:              7
Number of public wires:           5
Number of public wire bits:       5
Number of memories:               0
Number of memory bits:            0
Number of processes:              0
Number of cells:                  4
  DFFSR                           1
  NOR                             1
  NOT                             2

Chip area for module '\edge_detector_opt': 28.000000
  of which used for sequential elements: 18.000000 (64.29%)
```

상태 할당

- 많이 사용하는 할당 제도
  - **2진수 (또는 순서)**
    - 상태 개수에 따른 순서 2진수 사용
    - N개의 상태 → $\lceil \log_2 N \rceil$ 비트 필요
  - **Gray 코드**
    - Gray 코드 사용
    - 한 상태에서 다음 상태로 변할 때 인접 코드 간 오직 한자리만 변화함
    - N개의 상태 → $\lceil \log_2 N \rceil$ 비트 필요
  - **One-hot**
    - 한 상태 코드에서 1비트 하나만 있음
    - N개의 상태 → N비트 필요 (00…0 코드 안 사용)
  - **Almost one-hot**
    - One-hot와 비슷한데 00…0 코드도 사용
    - N개의 상태 → (N – 1)비트 필요

상태 할당

- 예시: 간소한 메모리 컨트롤러

상태	2진수	Gray 코드	One-hot	Almost one-hot
IDLE	000	000	000001	00000
READ1	001	001	000010	00001
READ2	010	011	000100	00010
READ3	011	010	001000	00100
READ4	100	110	010000	01000
WRITE	101	111	100000	10000

더 안전한 FSM

- 사용하지 않은 상태 코드 많음
  - 실제로 동작할 때 사용하지 않은 상태에 걸릴 수 있음

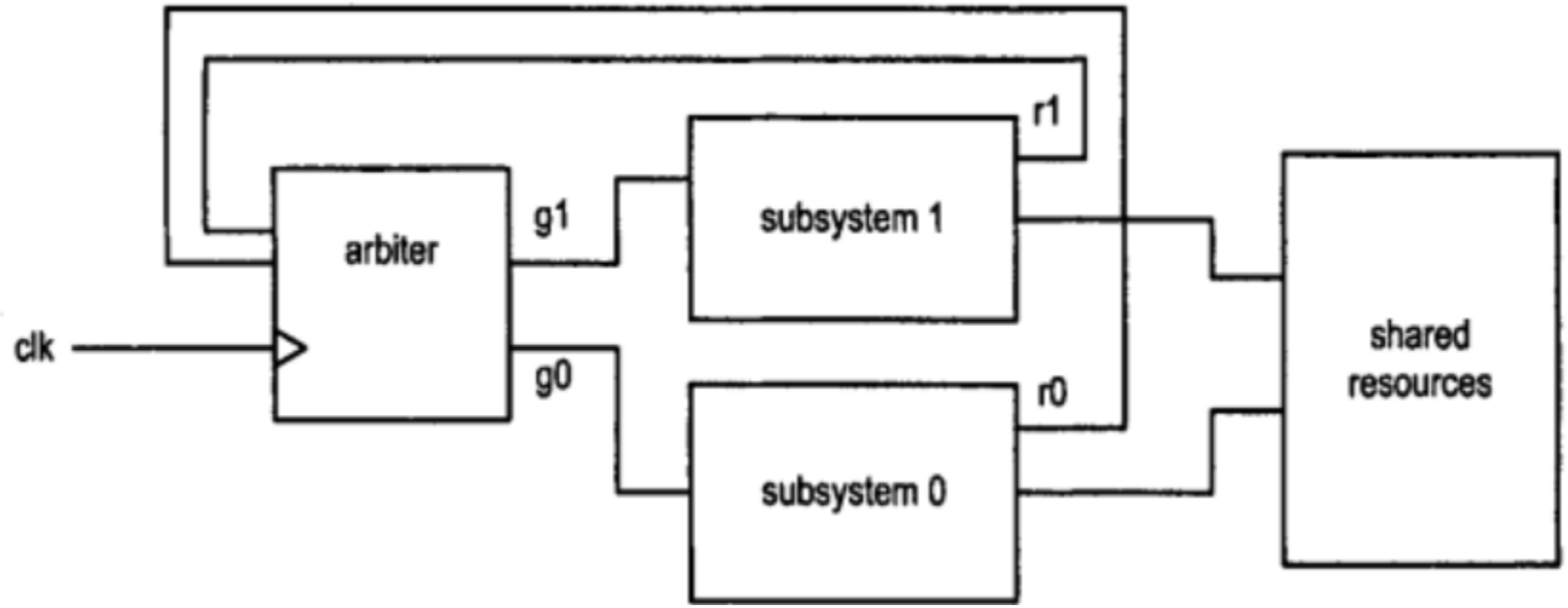
  → 예상대로 동작 못하거나 의미 없는 출력 만들게 됨
- 예방 방법
  - 사용하지 않은 상태에 걸리면 초기 상태(예, IDLE)로 돌아감
  - 사용하지 않은 상태를 처리하기 위한 추가 상태(예, ERROR) 만듬

```
case (state_reg)
   IDLE: ...
   ...
   default: state_next = IDLE;
endcase
```

```
case (state_reg)
   IDLE: ...
   ...
   ERROR: begin
      ...
      state_next = IDLE;
   end
   default: state_next = ERROR;
endcase
```

Arbiter

- 여러 시스템은 공용 자원(예, 메모리)을 같이 사용할 수 있게끔 해주는 컨트롤러

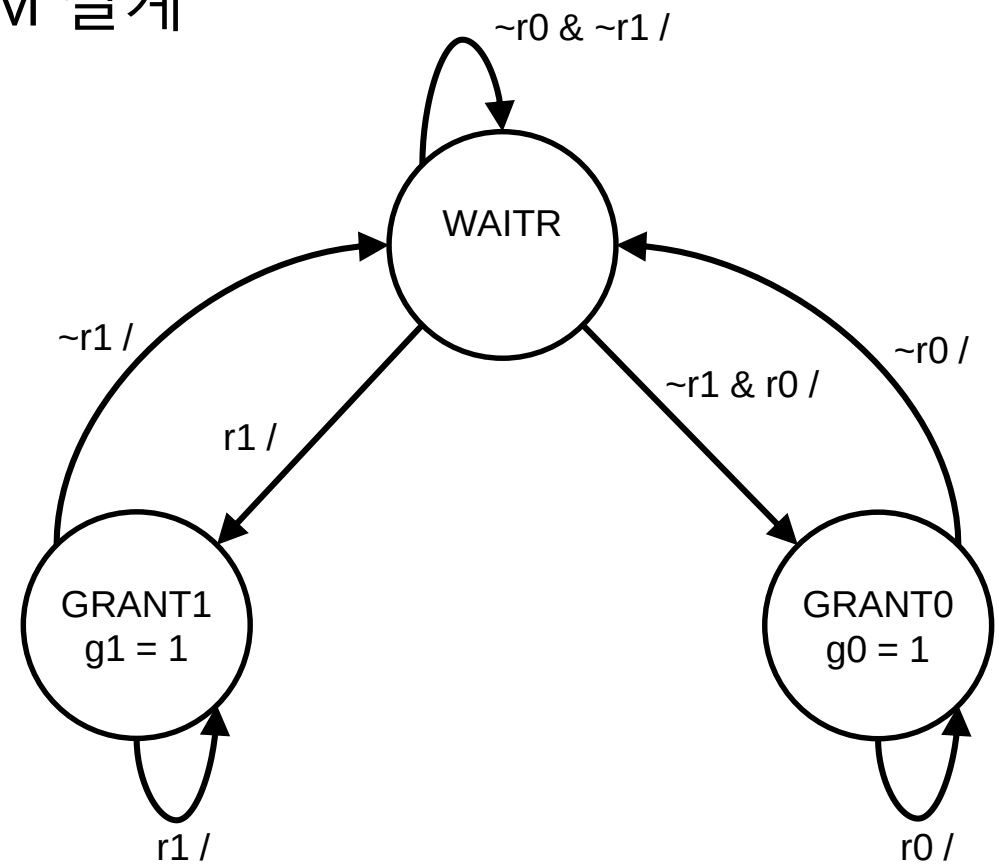


Arbiter

- Arbiter의 입출력 신호
  - **입력**
    - r0, r1 : request 신호
    - 예, r0 = 1 → subsystem0이 공용 자원을 사용하고 싶다는 의미
  - **출력**
    - g0, g1 : grant 신호
    - 예, g0 = 1 → subsystem0이 공용 자원을 사용하게 허가해준다는 의미

Arbiter

- Moore FSM 설계



Arbiter

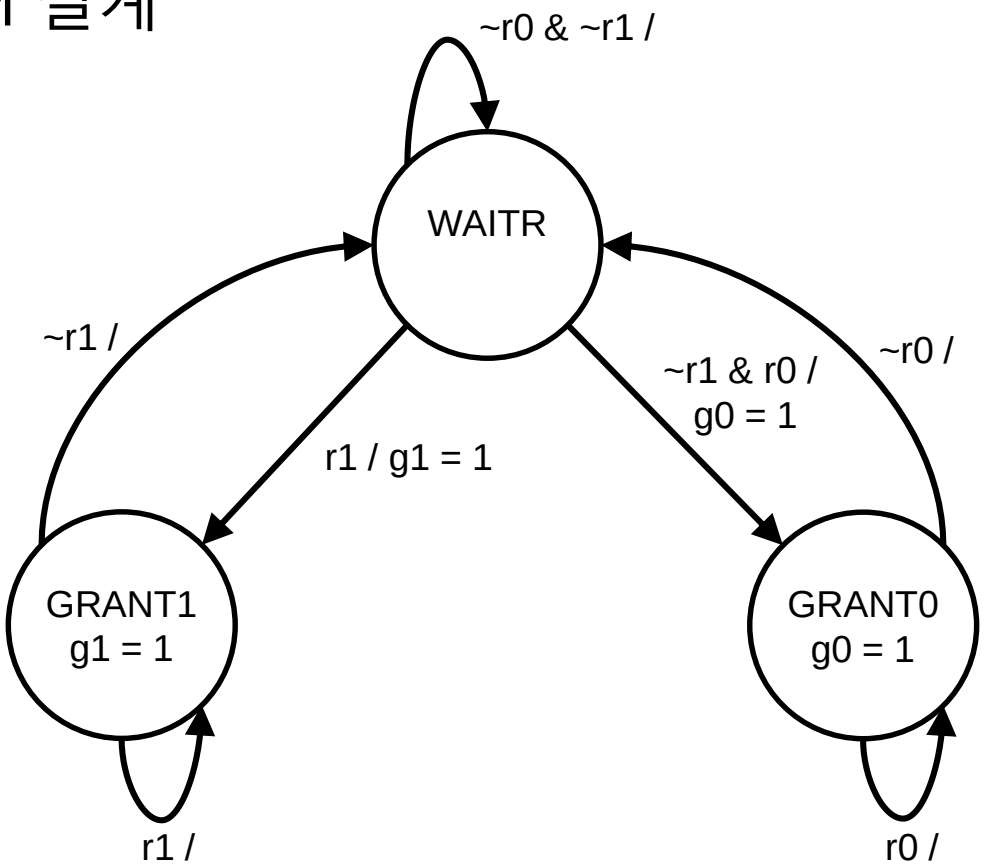
- Moore FSM 설계

```
module arbiter_moore(
   input            clk, rstb,
   input      [1:0] r,
   output reg [1:0] g
);

// state
localparam [1:0] WAITR  = 2'b00,
                 GRANT0 = 2'b01,
                 GRANT1 = 2'b10;

reg [1:0] state_reg, 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 and output logic
always @* begin
   state_next = state_reg;
   g = 2'b00;
   case (state_reg)
      WAITR: begin
         if (r[1]) state_next = GRANT1;
         else if (r[0]) state_next = GRANT0;
      end
      GRANT0: begin
         if (~r[0]) state_next = WAITR;
         g[0] = 1'b1;
      end
      GRANT1: begin
         if (~r[1]) state_next = WAITR;
         g[1] = 1'b1;
      end
      default: state_next = WAITR;
   endcase
end

endmodule
```

Arbiter

- Mealy FSM 설계



Arbiter

- Mealy FSM 설계

```
module arbiter_mealy(
   input            clk, rstb,
   input      [1:0] r,
   output reg [1:0] g
);

// state
localparam [1:0] WAITR  = 2'b00,
                 GRANT0 = 2'b01,
                 GRANT1 = 2'b10;

reg [1:0] state_reg, 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 and output logic
always @* begin
   state_next = state_reg;
   g = 2'b00;
   case (state_reg)
      WAITR: begin
         if (r[1]) begin
            state_next = GRANT1; g[1] = 1'b1;
         end
         else if (r[0]) begin
            state_next = GRANT0; g[0] = 1'b1;
         end
      end
      GRANT0: begin
         if (~r[0]) state_next = WAITR;
         g[0] = 1'b1;
      end
      GRANT1: begin
         if (~r[1]) state_next = WAITR;
         g[1] = 1'b1;
      end
      default: state_next = WAITR;
   endcase
end
endmodule
```

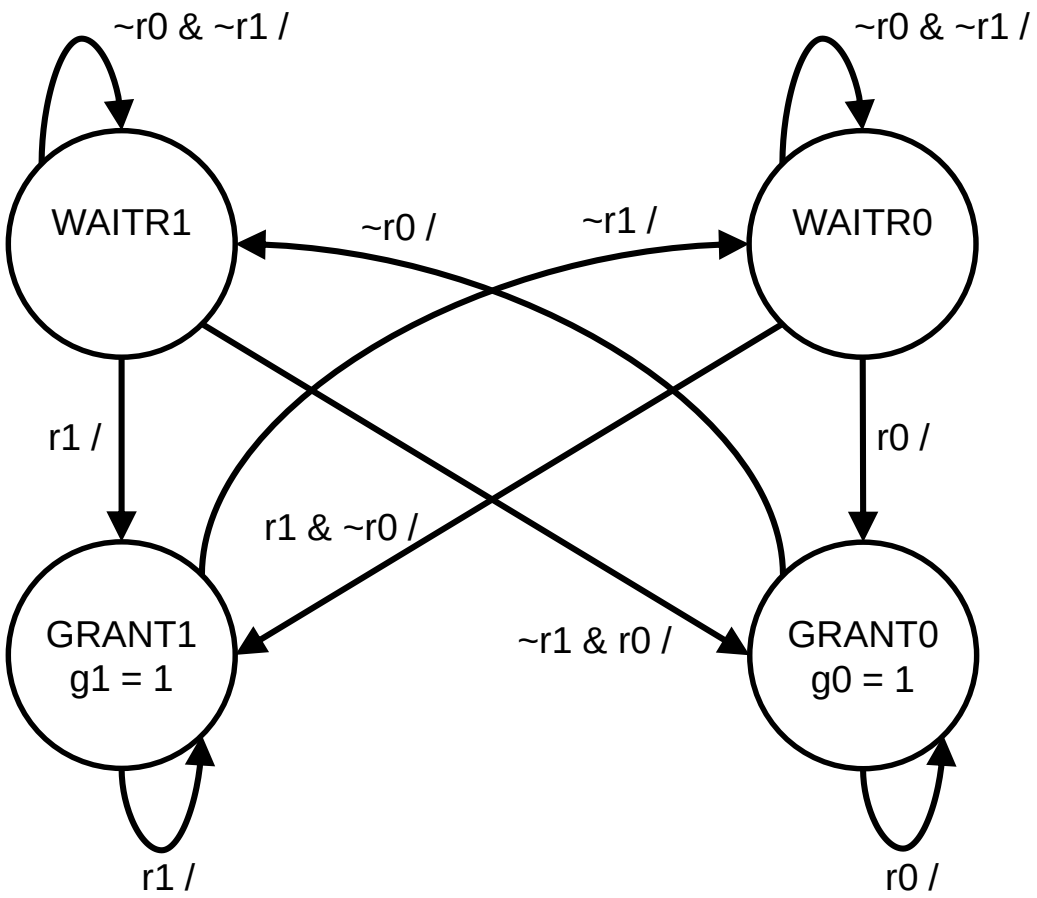
Arbiter

- 이전 설계의 단점
  - 코드에 따르지만 subsystem0과 subsystem1 중 하나 더 선호함
    (설계한 FSM의 경우 subsystem1을 더 선호함)
  - subsystem0과 subsystem1 동시의 공용 자원을 사용하려고 요청할 때 subsystem0은 허가 못 받음
- 공평한 arbiter 설계
  - 이전에 subsystem0은 공용 자원을 사용하면 다음에 subsystem1을 선호함
  - 이전에 subsystem1은 공용 자원을 사용하면 다음에 subsystem0을 선호함

  → WAITR 상태를 WAITR0과 WAITR1으로 나눔

Arbiter

- FSM 설계



Arbiter

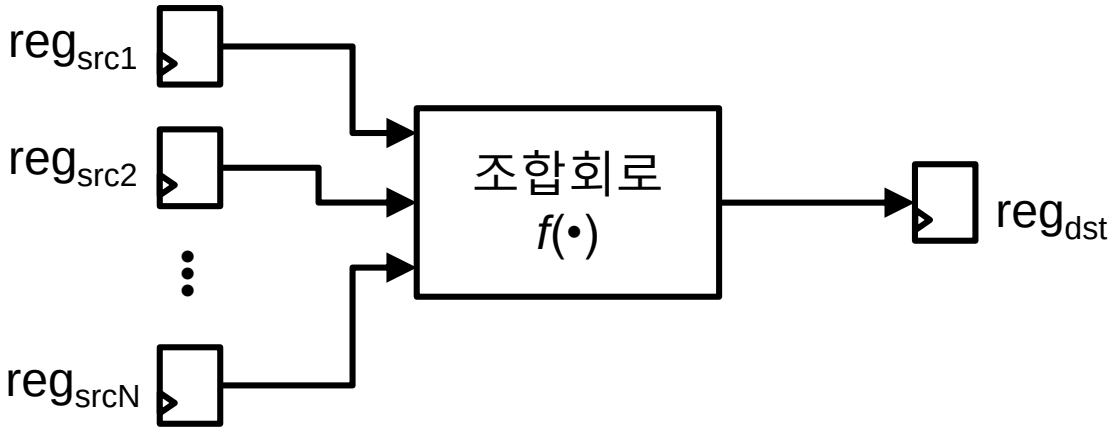
- FSM 설계

```
module arbiter_fair(
   input            clk, rstb,
   input      [1:0] r,
   output reg [1:0] g
);

// state
localparam [1:0] WAITR0 = 2'b00,
                 WAITR1 = 2'b01,
                 GRANT0 = 2'b10,
                 GRANT1 = 2'b11;

reg [1:0] state_reg, 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 and output logic
always @* begin
   state_next = state_reg;
   g = 2'b00;
   case (state_reg)
      WAITR0: begin
         if (r[0]) state_next = GRANT0;
         else if (r[1]) state_next = GRANT1;
      end
      WAITR1: begin
         if (r[1]) state_next = GRANT1;
         else if (r[0]) state_next = GRANT0;
      end
      GRANT0: begin
         if (~r[0]) state_next = WAITR1;
         g[0] = 1'b1;
      end
      GRANT1: begin
         if (~r[1]) state_next = WAITR0;
         g[1] = 1'b1;
      end
      default: state_next = WAITR0;
   endcase
end
endmodule
```

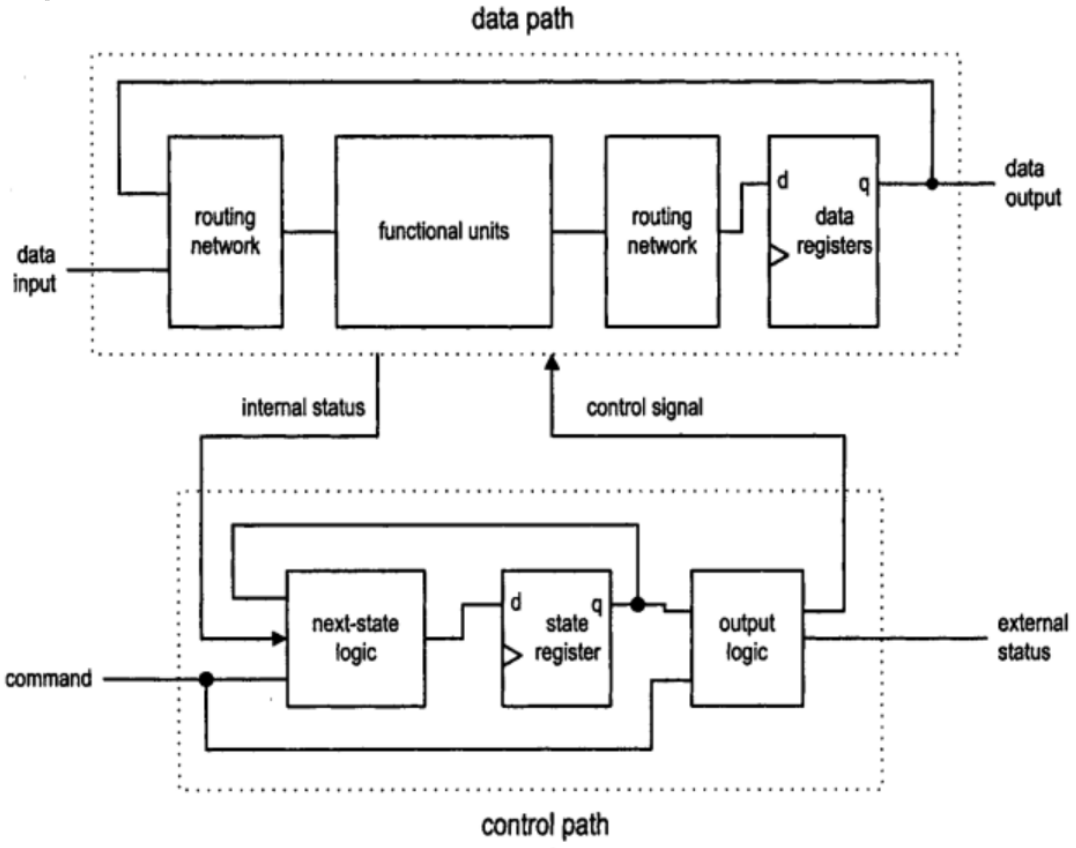
FSMD + RTL

- FSMD = FSM(control path) + Data path
- RTL = Register Transfer Level
  - 레지스터 : 알고리즘의 변수
  - 조합회로 : 알고리즘의 연산
  - $reg_{dst} = f(reg_{src1}, reg_{src2}, \ldots, reg_{srcN})$



FSMD + RTL

- FSMD 블록도

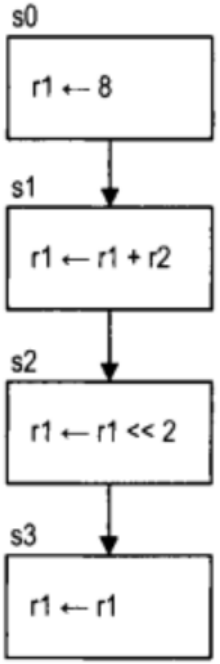


FSMD + RTL

- (FSMD + RTL)은 알고리즘을 하드웨어로 구현할 때 사용
- 상태도(state diagram)보다 ASM도 더 적합함
  - ASM + Data path = ASMD
- 예시:

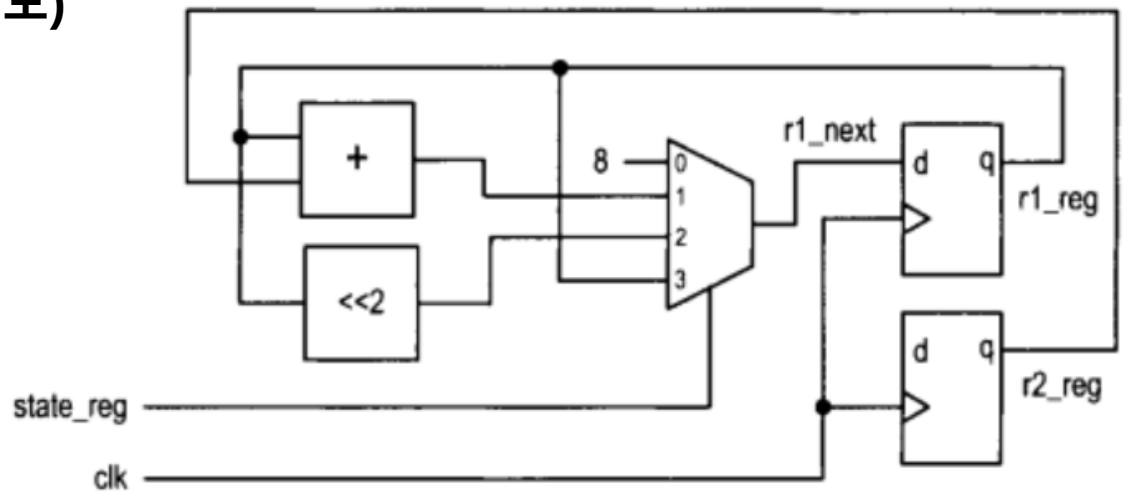
Pseudo 코드

```
r1 = 8
r1 = r1 + r2
r1 = r1<<2
r1 = r1
```



FSM (ASM도)

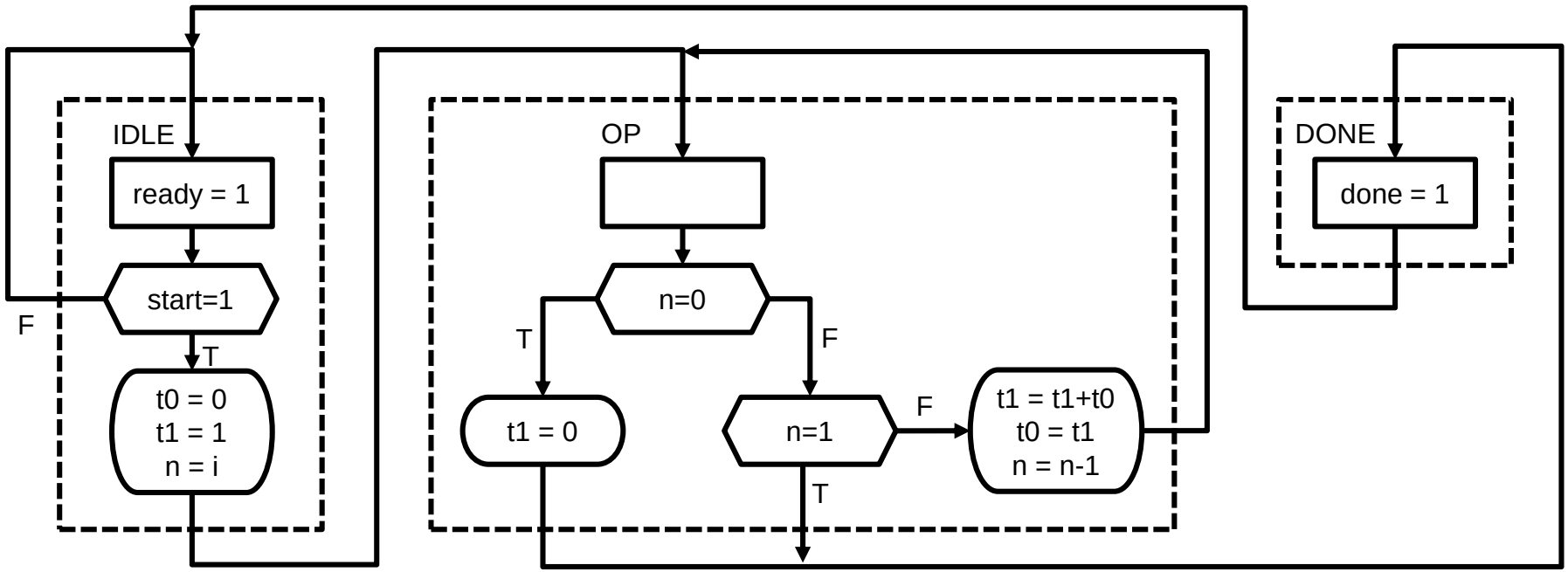
Data path



FSMD + RTL

- 예시: Fibonacci 수열

$$t(i) = \begin{cases} 0 & i = 0 \\ 1 & i = 1 \\ t(i-1) + t(i-2) & i > 1 \end{cases}$$



FSMD + RTL

- 예시: Fibonacci 수열

```
module fibonacci(
   input              clk, rstb,
   input              start,
   input       [4:0]  i,
   output reg         ready, done,
   output reg [19:0]  f
);

// state localparam [1:0] IDLE = 2'b00,
                          OP   = 2'b01,
                          DONE = 2'b10;

reg  [1:0] state_reg, state_next;
reg [19:0] t0_reg, t1_reg, t0_next, t1_next;
reg  [4:0] n_reg, n_next;
```

```
// state register
always @(posedge clk or negedge rstb) begin
   if (~rstb) begin
      state_reg <= 0;
      t0_reg <= 0;
      t1_reg <= 0;
      n_reg <= 0;
   end
   else begin
      state_reg <= state_next;
      t0_reg <= t0_next;
      t1_reg <= t1_next;
      n_reg <= n_next;
   end
end
```

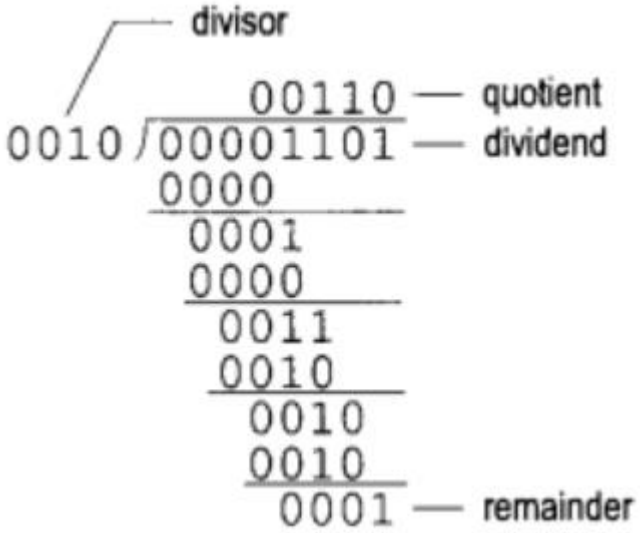
FSMD + RTL

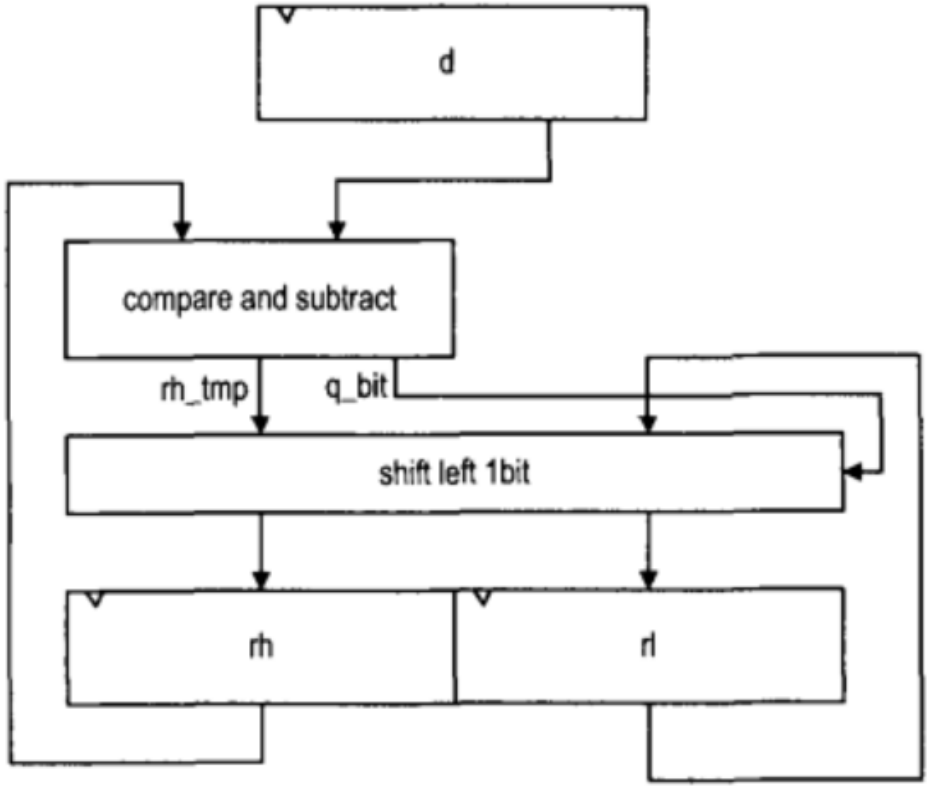
- 예시: Fibonacci 수열

```
// next-state logic and output logic
always @* begin
   state_next = state_reg;
   t0_next = t0_reg;
   t1_next = t1_reg;
   n_next = n_reg;
   ready = 1'b0;
   done = 1'b0;
   case (state_reg)
      IDLE: begin
         ready = 1'b1;
         if (start) begin
            t0_next = 0;
            t1_next = 1;
            n_next = i;
            state_next = OP;
         end
      end
      OP: begin
         if (n_reg==0) begin
            t1_next = 0;
            state_next = DONE;
         end
         else if (n_reg==1)
            state_next = DONE;
         else begin
            t1_next = t1_reg + t0_reg;
            t0_next = t1_reg;
            n_next = n_reg - 1;
         end
      end
      DONE: begin
         done = 1'b1;
         state_next = IDLE;
      end
      default: state_next = IDLE;
   endcase
   f = t1_reg;
end

endmodule
```

FSMD + RTL

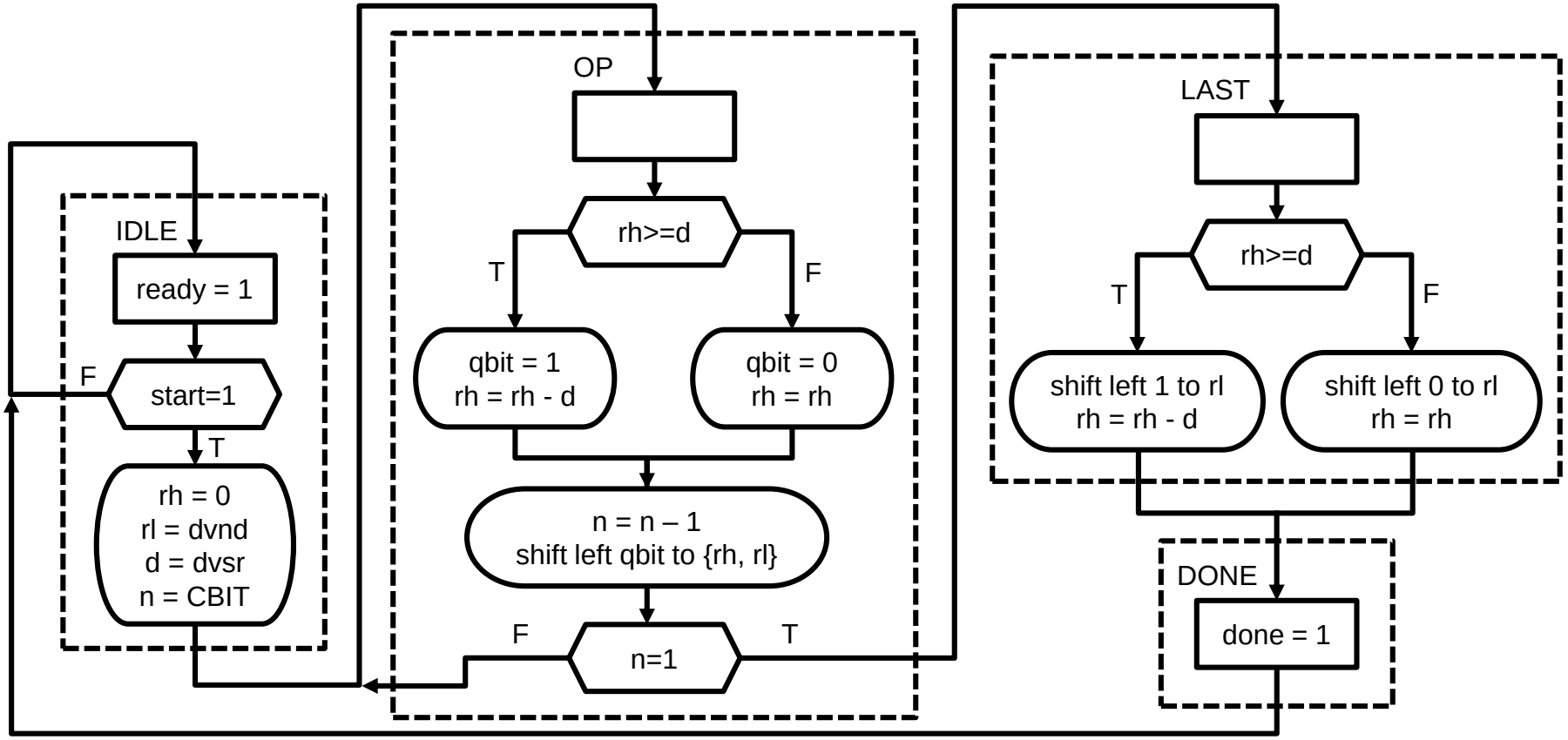
- 예시: 나누기





FSMD + RTL

- 예시: 나누기



FSMD + RTL

- 예시: 나누기

```
module divider
#(
   parameter N = 8,
             CBIT = $clog2(N)
) (
   input               clk, rstb,
   input               start,
   input      [N-1:0]  dvsr, dvnd,
   output reg          ready, done,
   output reg [N-1:0]  quo, rmd
);

// state
localparam [1:0] IDLE = 2'b00,
                 OP   = 2'b01,
                 LAST = 2'b10,
                 DONE = 2'b11;
```

```
reg [1:0] state_reg, state_next;
reg [N-1:0] rh_reg, rh_next, rl_reg, rl_next, rh_tmp;
reg [N-1:0] d_reg, d_next;
reg [CBIT-1:0] n_reg, n_next;
reg qbit;

// state register
always @(posedge clk or negedge rstb) begin
   if (~rstb) begin
      state_reg <= 0;
      rh_reg <= 0;
      rl_reg <= 0;
      d_reg <= 0;
      n_reg <= 0;
   end
   else begin
      state_reg <= state_next;
      rh_reg <= rh_next;
      rl_reg <= rl_next;
      d_reg <= d_next;
      n_reg <= n_next;
   end
end
```

FSMD + RTL

- 예시: 나누기

```
// next-state logic and output logic
always @* begin
   state_next = state_reg;
   rh_next = rh_reg; rl_next = rl_reg;
   d_next = d_reg; n_next = n_reg;
   ready = 1'b0; done = 1'b0;
   case (state_reg)
      IDLE: begin
         ready = 1'b1;
         if (start) begin
            rh_next = 0; rl_next = dvnd;
            d_next = dvsr; n_next = N;
            state_next = OP;
         end
      end
      OP: begin
         rl_next = {rl_reg[N-2:0], qbit};
         rh_next = {rh_tmp[N-2:0], rl_reg[N-1]};
         n_next = n_reg - 1;
         if (n_next==0) state_next = LAST;
      end
      LAST: begin
         rl_next = {rl_reg[N-2:0], qbit};
         rh_next = rh_tmp;
         state_next = DONE;
      end
      DONE: begin
         done = 1'b1;
         state_next = IDLE;
      end
      default: state_next = IDLE;
   endcase
   quo = rl_reg; rmd = rh_reg;
end

// compare and subtract
always @* begin
   if (rh_reg>=d_reg) begin
      rh_tmp = rh_reg - d_reg; qbit = 1;
   end
   else begin
      rh_tmp = rh_reg; qbit = 0;
   end
end
endmodule
```