# SoC를 이해하기 위해 알아야할 순차 기본 구성 블럭

# Agenda

- SoC 기본 순차논리 구성 블럭 이해하기
  - 플립플랍
  - counter와 시프트 레지스터
  - 레지스터 맵과 메모리
  - FSM 이해하기
- 기본 블럭 시뮬레이션 해보기 - 실습
- Q&A

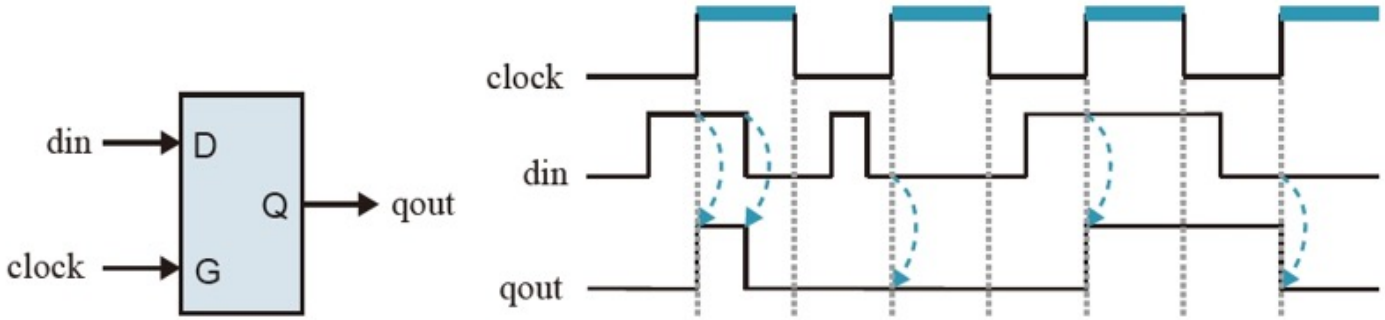
# SoC 기본 순차논리

- 순차회로
  - 현재의 입력, 과거의 입력, 회로에 기억된 상태값에 의해 출력이 결정
  - 과거의 입력, 현재의 상태값을 저장하는 저장소자(래치, 플립플롭)와 조합논리회로로 구성
  - 데이터 레지스터, 시프트 레지스터, 계수기(counter), 직렬/병렬 변환기, 유한상태머신 (Finite State Machine; FSM), 주파수 분주기, 펄스 발생기 등
- 래치와 플립플롭
  - 래치 (latch) : 클록신호의 레벨(즉, 0 또는 1)에 따라 동작하는 저장소자
  - 플립플롭 (Flop-flop) : 클록신호의 상승 또는 하강에지에 동기되어 동작하는 저장소자
  - always 구문 내부에 if 조건문을 이용하여 모델링
- 순차회로의 모델링
  - always 블록을 이용한 행위수준 모델링, 게이트 프리미티브 및 하위모듈 인스턴스,

    연속 할당문 등 다양한 Verilog HDL 구문들이 사용됨
  - 할당문의 형태 (nonblocking 또는 blocking)에 따라 회로의 동작과 구조가 달라짐

# SoC 기본 순차논리 - D 래치

- 클록 신호의 레벨 (0 또는 1)에 따라 통과모드 또는 유지모드로 동작
  - 통과 (transparent) 모드 : 입력 D가 출력 Q로 통과됨
  - 유지 (hold) 모드 : 출력 Q의 값이 유지됨
  - positive (level-sensitive) latch, negative (level-sensitive) latch

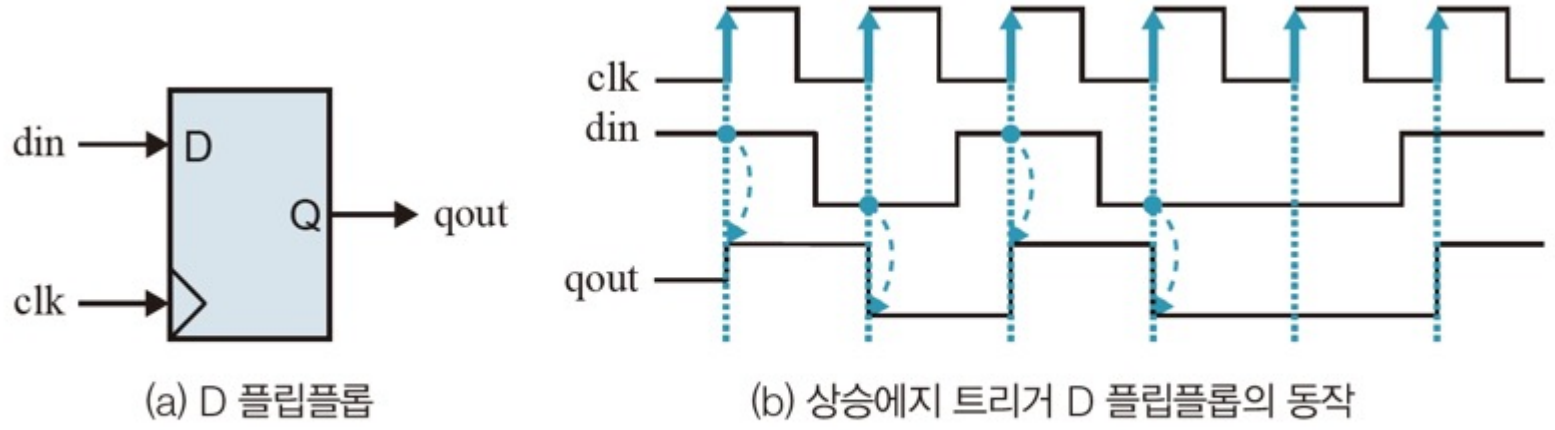



(a) positive D 래치의 기호 (b) positive D 래치의 동작

[그림 11–1] positive D 래치

# SoC 기본 순차논리 - D 플립플롭

- **클록 신호의 천이에지 (transition edge)에서 동작**
  - 한 클록 주기 동안 정보를 저장함
  - positive edge-triggered, negative edge-triggered
  - always 블록의 감지신호목록에 클록신호만 포함됨




(a) D 플립플롭 (b) 상승에지 트리거 D 플립플롭의 동작

[그림 11-6] 상승에지 트리거 D 플립플롭

# 래치와 플립플롭의 차이점

| 특성 | 래치 | 플립플롭 |
| --- | --- | --- |
| 클록 감응성 | 레벨에 반응 | 에지에 반응 |
| 데이터 변경 | 클록 활성 레벨 동안 지속적 | 클록 에지에서만 |
| 안정성 | 글리치에 취약 | 글리치에 강함 |
| 동기화 | 비동기 동작 가능 | 주로 동기식 시스템에서 사용 |
| 회로 복잡성 | 상대적으로 단순 | 더 복잡 (일반적으로 두 개의 래치로 구성) |
| 전력 소비 | 상대적으로 낮음 | 상대적으로 높음 |
| 주요 응용 | 간단한 저장, 비동기 시스템 | 동기식 디지털 시스템, 레지스터, 카운터 |
| Verilog 모델링 | always @(*) 또는 always @(CLK) | always @(posedge CLK) 또는 always @(negedge CLK) |

# SoC 기본 순차논리 – 래치와 플립플롭 주요 응용분야

| 적용 분야 | 래치 | 플립플롭 |
|---|---|---|
| 메모리 요소 | ✓ | ✓ |
| 카운터 | ✓ | ✓ |
| 레지스터 | ✓ | ✓ |
| 시프트 레지스터 | ✓ | ✓ |
| 전력 게이팅 회로 | ✓ | |
| 클록 게이팅 회로 | ✓ | |
| 고속 회로 설계 | ✓ | |
| 주파수 분주기 | | ✓ |
| 데이터 전송 | | ✓ |
| 입력 동기화 | | ✓ |
| 바운스 제거 스위치 | | ✓ |
| 제어 회로 및 알람 | ✓ | ✓ |
| 주파수 합성기 | | ✓ |
| 유한 상태 기계 (FSM) | ✓ | ✓ |

# SoC 기본 순차논리 - 계수기

- 계수기(counter)
  - 클록 펄스가 인가될 때마다 값이 증가 또는 감소되는 회로
  - 주파수 분주기, 타이밍 및 제어 신호 생성 등 디지털 회로 설계에 폭넓게 사용

**동기식 계수기**

- 모든 플립플롭이 하나의 공통 클록신호에 의해 구동되며, 모든 플립플롭의 상태변경이 동시에 일어남
- 장점 : 설계와 검증이 용이하며, 계수 속도가 빠름
- 단점 : 비동기식 카운터에 비하여 회로가 복잡함

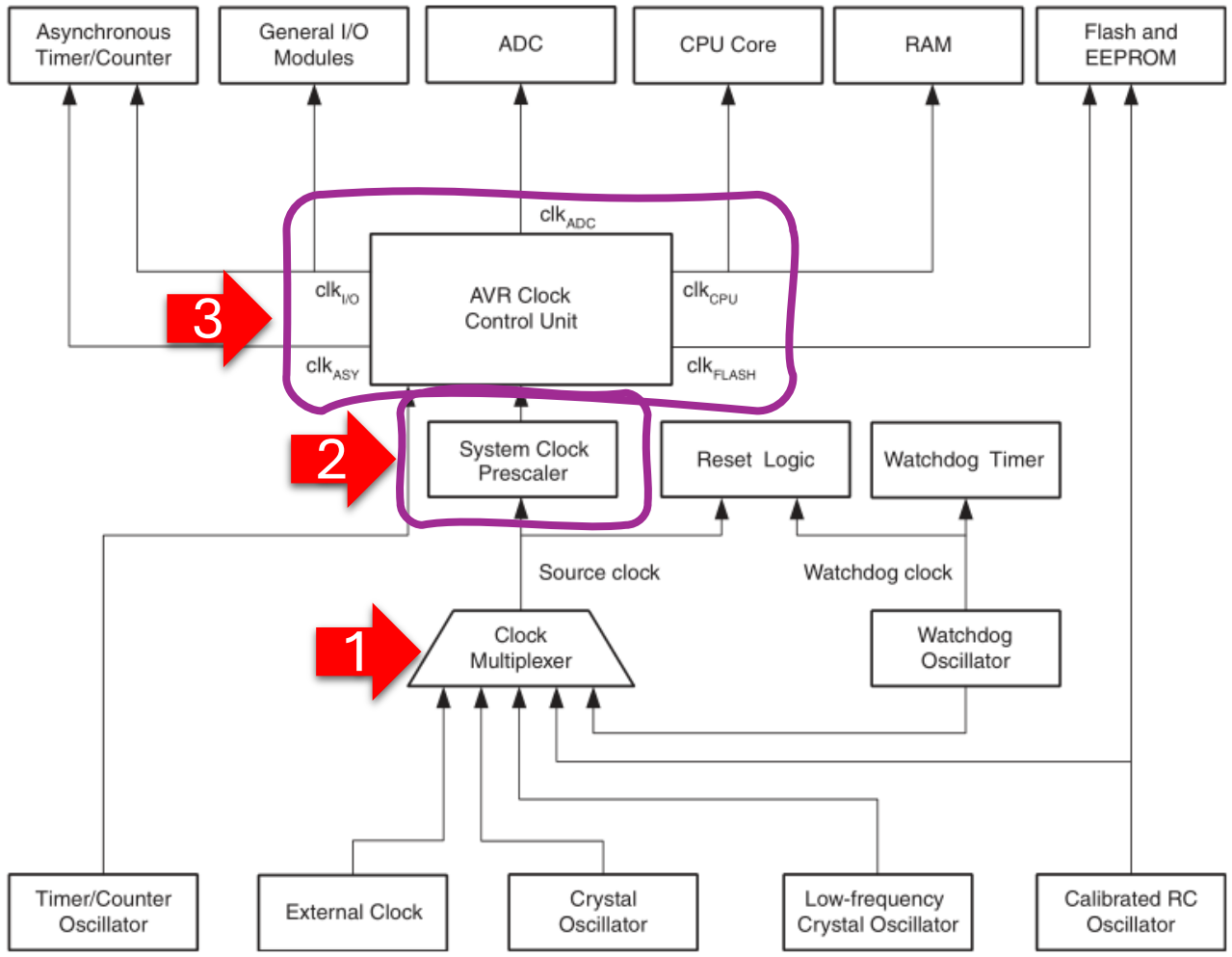
**비동기식 계수기**

- 첫단의 플립플롭에 클록신호가 인가되면, 플립플롭의 출력이 다음 단의 플립플롭을 트리거시키는 방식으로 동작 → 리플 계수기(ripple counter)라고도 함
- 장점 : 동기식 계수기에 비해 회로가 단순해짐
- 단점 : 각 플립플롭의 전파 지연이 누적되어 최종단의 출력까지 전파되어 속도가 느림

# SoC 기본 순차논리 – 계수기 주요 응용분야

- SoC 주변장치에서의 활용
  1. 주파수 분주: 높은 주파수의 클록 신호를 낮은 주파수로 나누는 데 사용됩니다.
  2. 타이밍 생성: 특정 시간 간격으로 이벤트를 트리거하는 데 사용됩니다.
  3. 제어 신호 생성: 다른 주변장치나 모듈을 제어하기 위한 신호를 생성합니다.
  4. 이벤트 카운팅: 특정 이벤트의 발생 횟수를 세는 데 사용됩니다.

주파수 분주기 사용 예



# SoC 기본 순차논리 – 계수기 실습:주파수 분주기 실습

- 주파수 1/10 분주기 , duty 50% 실습

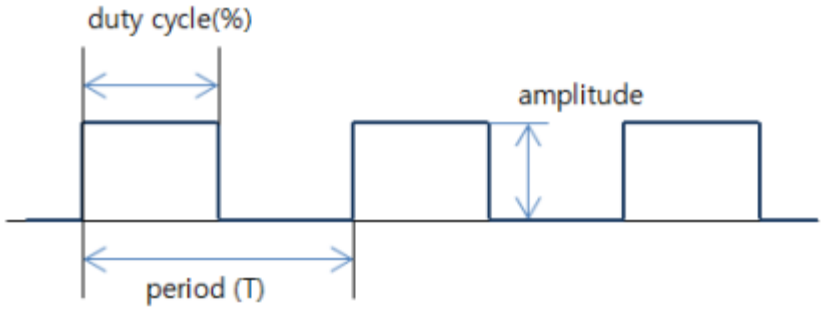



그림 1. 주파수, 주기, duty cycle

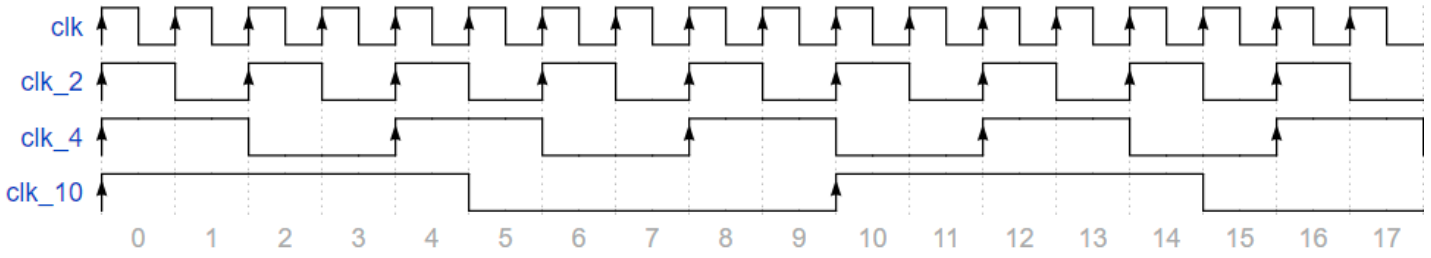
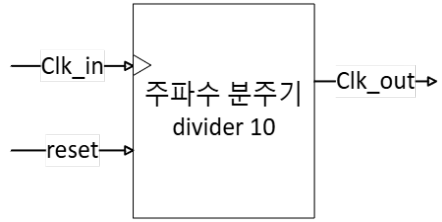



그림 2. 주파수 분주기




```
1   `timescale 1ns / 1ps
2   //fdivider_10.v
3   module freq_divider_by_10 #(
4       parameter DIVISOR = 10
5   )(
6       input wire clk_in,
7       input wire reset,
8       output reg clk_out
9   );
10
11      localparam WIDTH = $clog2(DIVISOR);
12      reg [WIDTH-1:0] count;
13
14      always @(posedge clk_in or posedge reset) begin
15          if (reset) begin
16              count <= 0;
17              clk_out <= 1'b0;
18          end else begin
19              if (count == DIVISOR - 1) begin
20                  count <= 0;
21              end else begin
22                  count <= count + 1'b1;
23              end
24
25              // DIVISOR의 절반에 도달할 때마다 출력 토글
26              if (count == (DIVISOR / 2) - 1) begin
27                  clk_out <= 1'b1;
28              end else if (count == DIVISOR - 1) begin
29                  clk_out <= 1'b0;
30              end
31          end
32      end
33
34  endmodule
```

# 계수기 실습:주파수 분주기 실습 – 시뮬레이션 벡터

- 아래 코드를 주석을 참조하여 각각의 경우에 맞는 벡터를 생성해 보시오.

```
1   `timescale 1ns / 1ps
2   //tb_fdivider_10.v
3   module freq_divider_by_10_tb;
4       reg clk_in;
5       reg reset;
6       wire clk_out;
7
8       freq_divider_by_10 #(
9           .DIVISOR(10)
10      ) divider (
11          .clk_in(clk_in),
12          .reset(reset),
13          .clk_out(clk_out)
14      );
```

1

2

```
    // 클록 생성 (10ns 주기, 100MHz)
    always #5 clk_in = ~clk_in;

    initial begin
        clk_in = 0;
        reset = 1;
        #20 reset = 0;  // 20ns 후 리셋 해제

        // 200 사이클 동안 시뮬레이션
        repeat(200) @(posedge clk_in);

        $finish;
    end
```

3

4

5

# 계수기 실습:주파수 분주기 실습– testbench code결과

```
1   `timescale 1ns / 1ps
2   //tb_fdivider_10.v
3   module freq_divider_by_10_tb;
4       reg clk_in;
5       reg reset;
6       wire clk_out;
7
8       freq_divider_by_10 #(
9           .DIVISOR(10)
10      ) divider (
11          .clk_in(clk_in),
12          .reset(reset),
13          .clk_out(clk_out)
14      );
15
16      // 클록 생성 (10ns 주기, 100MHz)
17      always #5 clk_in = ~clk_in;
18
19      initial begin
20          clk_in = 0;
21          reset = 1;
22          #20 reset = 0;  // 20ns 후 리셋 해제
23          // 200 사이클 동안 시뮬레이션
24          repeat(200) @(posedge clk_in);
25          $finish;
26      end
27      // 결과 모니터링
28      always @(posedge clk_in) begin
29          $display("Time=%0t, clk_in=%b, clk_out=%b", $time, clk_in, clk_out);
30      end
31  endmodule
```

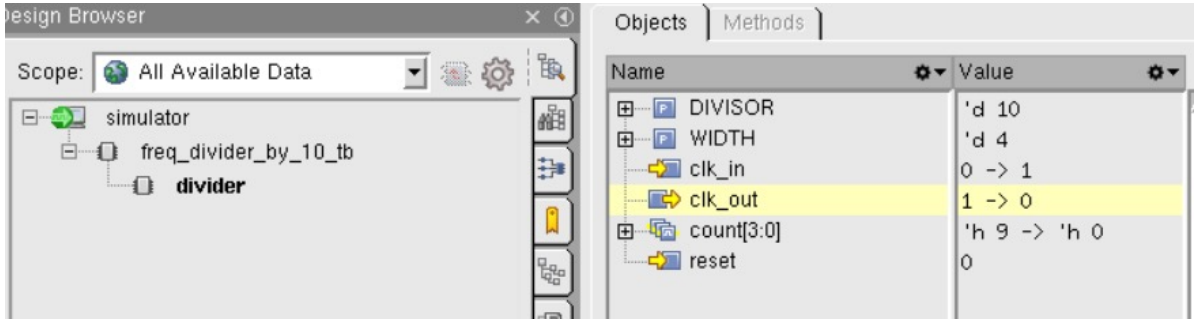
1

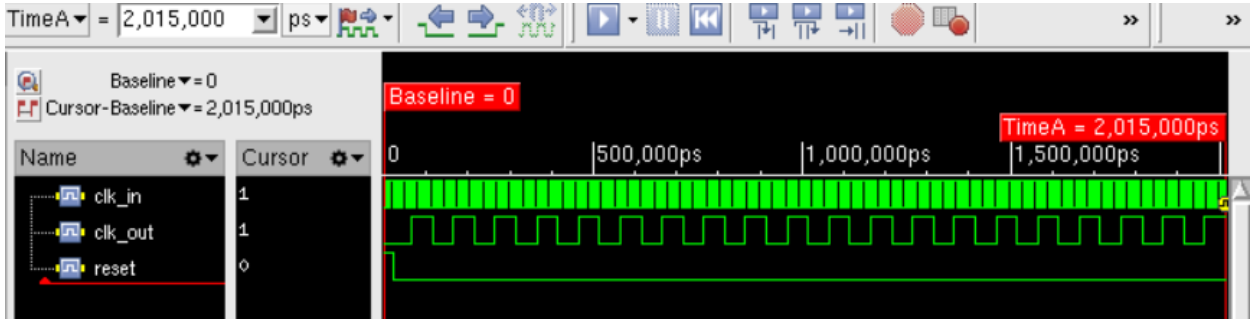
2

3

4

# 계수기 실습:주파수 분주기 시뮬레이션 결과



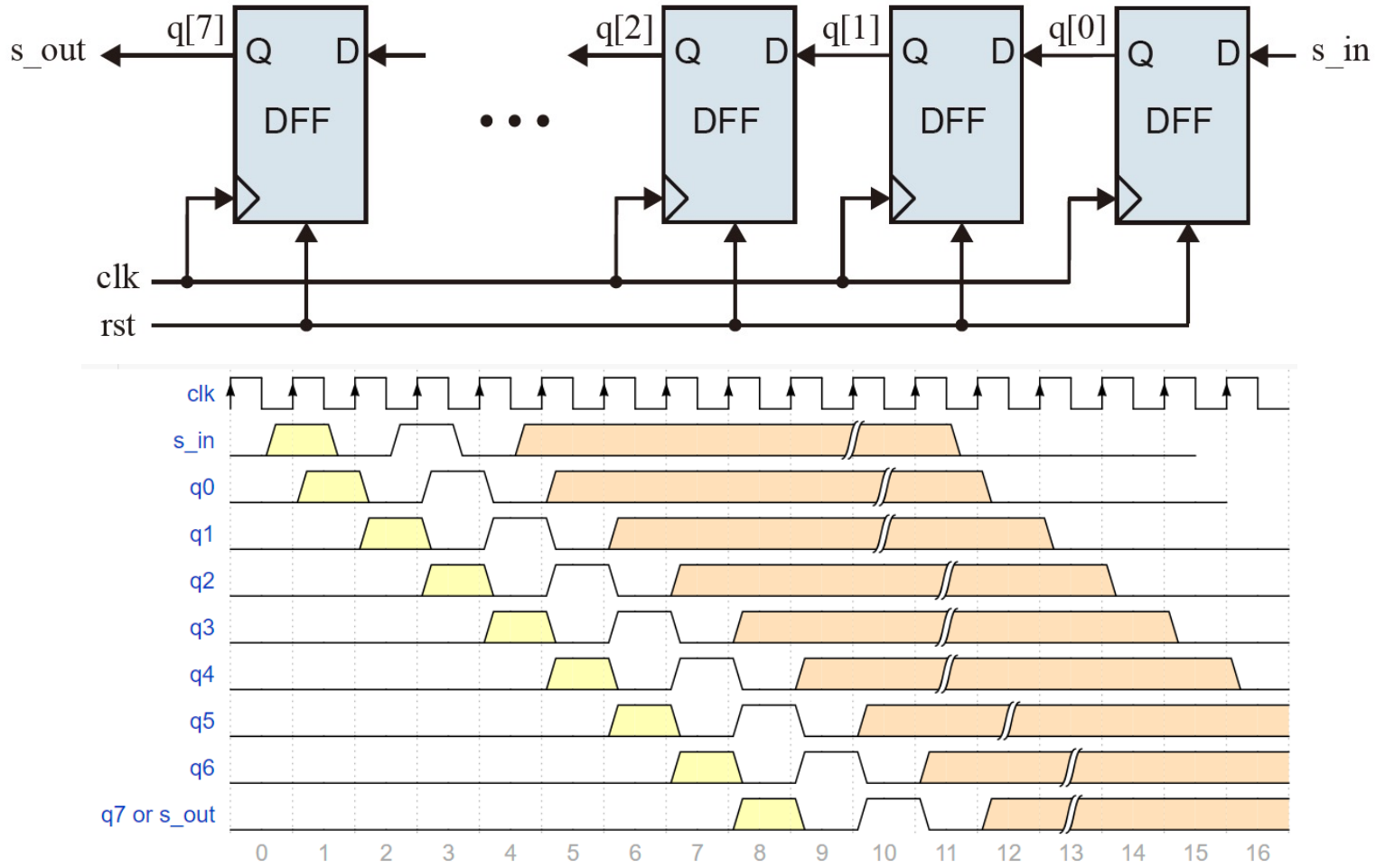




# SoC 기본 순차논리 - 시프트 레지스터

- 시프트 레지스터 (shift register)
  - 클록신호가 인가될 때마다 저장된 플립플롭에 저장된 데이터가 왼쪽 또는 오른쪽으로 이동되는 회로
  - 여러 개의 플립플롭이 직렬로 연결된 구조
  - 형태
    - 직렬입력 - 직렬출력 (Serial-In, Serial-Out)
    - 직렬입력 - 병렬출력 (Serial-In, Parallel-Out)
    - 병렬입력 - 직렬출력 (Parallel-In, Serial-Out)
    - 병렬입력 - 병렬출력 (Parallel-In, Parallel-Out)
    - 왼쪽 시프트, 오른쪽 시프트, 양방향 시프트
  - nonblocking 할당문, 시프트 연산자, 결합 연산자, 반복문 등으로 모델링

# SoC 기본 순차논리 - 직렬입력-직병렬출력 시프트 레지스터

- 8비트 직렬입력-직력/병렬 출력 시프트 레지스터



# SoC 기본 순차논리 - 시프트 레지스터

| 기능 | 설명 | 활용 예시 |
| --- | --- | --- |
| **SIPO** | 외부에서 직렬로 들어오는 데이터를 병렬로 변환하여 GPIO 출력으로 제공 | SPI/$I^2C$ 등 직렬 인터페이스를 통해 데이터를 받아 GPIO 포트 확장 |
| **PISO** | 내부의 병렬 데이터를 직렬 형태로 변환하여 외부에 전송 | 여러 개의 스위치나 센서 등으로부터 읽은 GPIO 상태를 직렬 인터페이스를 통해 프로세서나 다른 장치로 전송 |

# SoC 기본 순차논리 -8비트 직렬 입력-병렬 출력 실습

- 아래 코드를 작성하여 시뮬레이션 해 보시오.

```
2   `timescale 1ns / 1ps
3
4   module shift_register_sipo #(
5       parameter WIDTH = 8
6   )(
7       input clk,
8       input reset,
9       input serial_in,
10      output reg [7:0] parallel_out
11  );
12
13  always @(posedge clk or posedge reset) begin
14      if (reset) begin
15          parallel_out <= 8'b0;
16      end else begin
17          // 왼쪽으로 한 비트씩 시프트하고 가장 오른쪽에 입력 삽입
18          parallel_out <= {parallel_out[6:0], serial_in};
19      end
20  end
21
22  endmodule
```

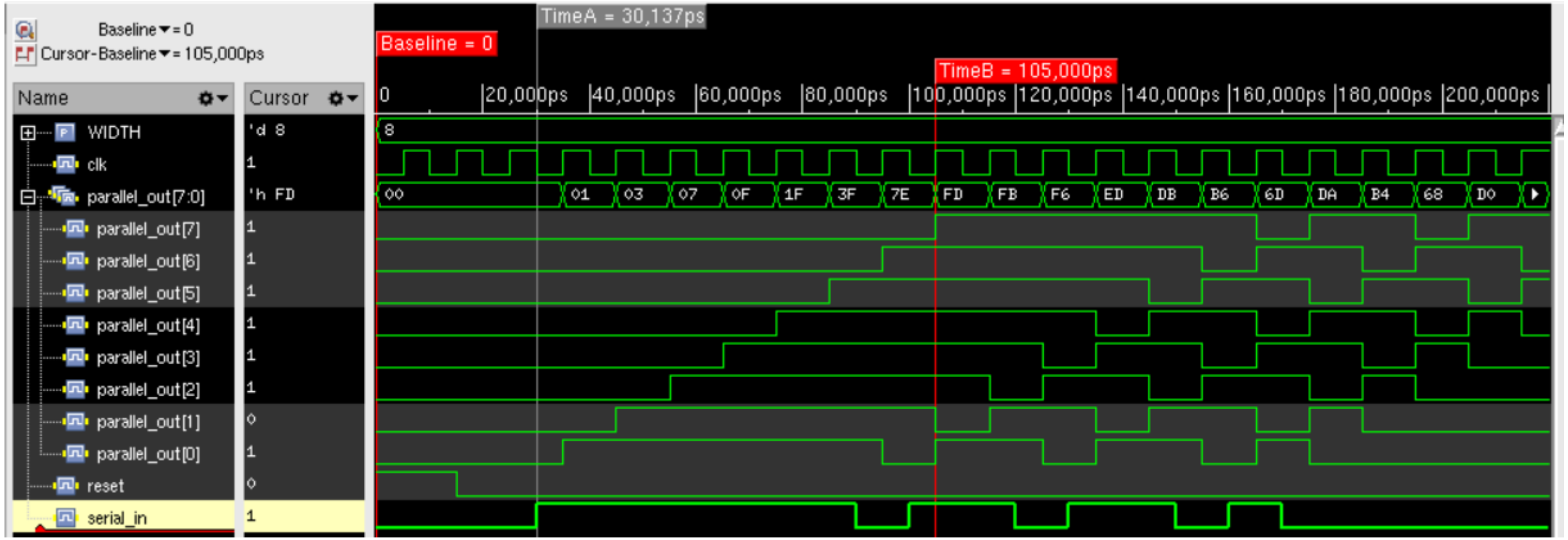
# 데이터 직렬입력- 8비트 직렬 입력-병렬 출력 – 시뮬레이션 벡터

- 아래 코드를 참조하여 시뮬레이션 해보시기 바랍니다.

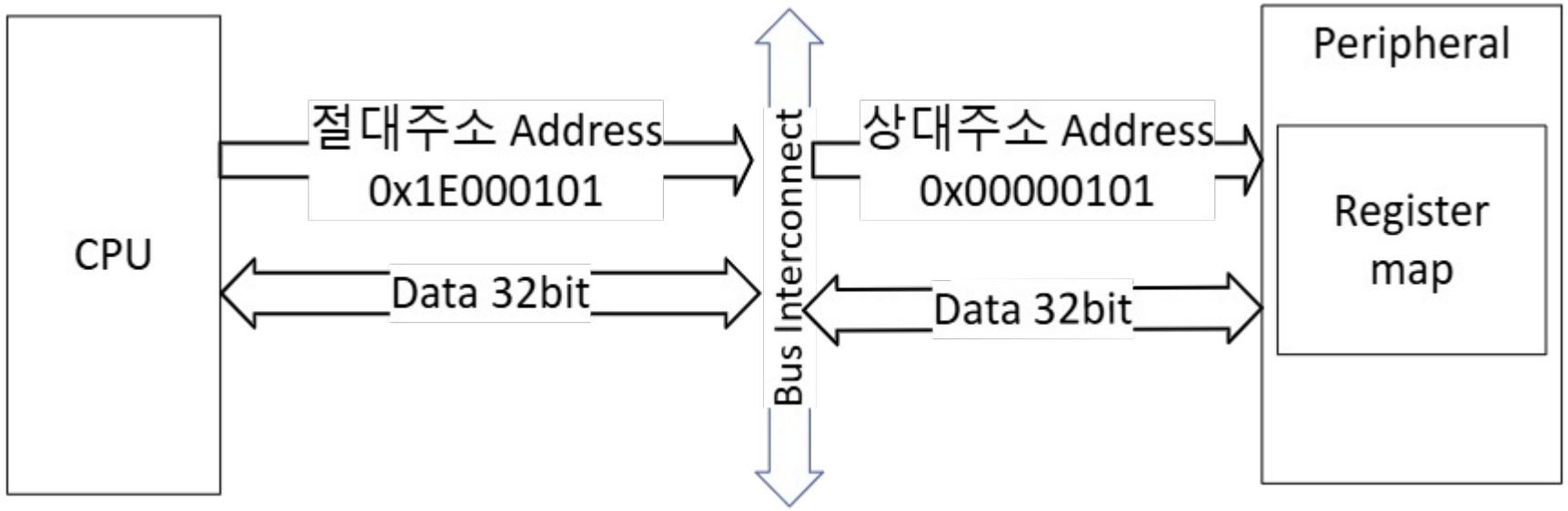
```
`timescale 1ns / 1ps

module tb_shift_register_sipo;
localparam WIDTH = 8;

reg clk;
reg reset;
reg serial_in;
wire [WIDTH -1 :0] parallel_out;
// 모듈 인스턴스화
shift_register_sipo uut (
    .clk(clk),
    .reset(reset),
    .serial_in(serial_in),
    .parallel_out(parallel_out)
);

// 클럭 생성 (100MHz)
initial begin
    clk = 0;
    forever #5 clk = ~clk; // 주기 10ns (100MHz)
end
```

```
// 입력 데이터 및 리셋 신호 설정
initial begin
    reset = 1'b1; serial_in = 0;
    #15 reset = 1'b0;
    // random vector 생성 test
    repeat(WIDTH*2) begin
        @(negedge clk);
        // 랜덤 비트 입력 (SoC 환경에서의 데이터 입력 상황 모사)
        serial_in = $random % 2;
    end
    // 추가 클럭 사이클 진행 후 종료
    #50 $finish;
end

// 결과 모니터링
initial begin
    $monitor("time=%0t | reset=%b | serial_in=%b | parallel_out=%b",
            $time, reset, serial_in, parallel_out);
end

endmodule
```

# 데이터 직렬입력- 8비트 직렬 입력-병렬 출력 – testbench code결과



# SoC 기본 순차논리 - 레지스터 맵과 메모리 맵




레지스터 맵(Register Map) 예시

| 주소 (Hex) | 이름 | 설명 | 읽기/쓰기 |
|---|---|---|---|
| **0x00** | CONTROL | 하드웨어 제어 | RW |
| **0x04** | STATUS | 하드웨어 상태 | R |
| **0x08** | DATA_IN | 입력 데이터 | R |
| **0x0C** | DATA_OUT | 출력 데이터 | W |

# SoC 기본 순차논리 - 레지스터 맵과 메모리 맵

- 메모리 맵(Memory Map)
- **메모리 맵**은 SoC에서 사용 가능한 메모리 공간을 정의한 것입니다.
  - CPU가 접근할 수 있는 모든 메모리 영역을 포함하며, 각 영역은 특정 용도로 나뉩니다.
- 특징
  - 주소 공간 분할: 메모리는 코드, 데이터, 스택, 외부 장치 등으로 분할됩니다.
- 장치 매핑:
  - 외부 장치(예: UART, SPI 등)는 메모리 주소에 매핑되어 CPU가 이를 제어할 수 있습니다.
- 효율적 설계:
  - 메모리 맵을 통해 데이터와 장치 간의 효율적인 접근이 가능합니다.

예시

| 주소 범위 (Hex) | 용도 | 설명 |
|---|---|---|
| **0x00000000~0x0000FFFF** | ROM | 부트 코드 저장 |
| **0x00010000~0x0001FFFF** | RAM | 실행 중인 프로그램 데이터 |
| **0x80000000~0x80000FFF** | UART | UART 제어 및 데이터 전송 |
| **0x90000000~0x90000FFF** | GPIO | GPIO 핀 설정 및 상태 확인 |

# SoC 기본 순차논리 레지스터 맵을 통한 메모리 관리

- 레지스터 맵(Register Map)은 하드웨어 모듈의 제어 및 상태 정보를 특정 메모리 주소에 매핑하여 소프트웨어가 이를 효율적으로 관리
- 하드웨어와 소프트웨어 간의 상호작용이 체계적으로 이루어지며, 메모리 관리가 간소화
- 레지스터 맵을 통한 메모리 관리의 주요 원칙

| 원칙 | 설명 |
|---|---|
| 주소 공간 분할 | 각 레지스터에 고유한 메모리 주소를 할당하여 하드웨어 모듈 간 충돌을 방지하고 체계적으로 관리. |
| 효율적인 접근 | CPU가 특정 주소를 통해 하드웨어를 제어하거나 상태를 읽음으로써 빠르고 간단한 데이터 처리 가능. |
| 읽기/쓰기 권한 설정 | 레지스터마다 읽기(Read), 쓰기(Write), 읽기-쓰기(Read-Write) 권한을 설정하여 잘못된 접근 방지. |
| 확장성 | 새로운 하드웨어 모듈 추가 시 기존 메모리 구조에 쉽게 통합 가능 (새로운 주소 공간만 추가). |
| 메모리 보호 | 권한 설정 및 주소 분리를 통해 소프트웨어가 불필요하거나 잘못된 영역에 접근하지 않도록 보호. |

# SoC 기본 순차논리 레지스터 맵을 통한 메모리 관리

| 항목 | 설명 |
| --- | --- |
| 효율적인 자원 분배 | 각 하드웨어 모듈에 고유한 메모리 공간을 할당하여 충돌을 방지하고, CPU가 효율적으로 접근 가능. |
| 소프트웨어와 하드웨어 통합 | 소프트웨어가 특정 메모리 주소를 통해 하드웨어를 제어하거나 상태를 읽음으로써 간단한 통합 가능. |
| 확장성 | 새로운 하드웨어 모듈 추가 시 기존 구조에 쉽게 통합 가능 (새로운 주소 공간만 추가). |
| 읽기/쓰기 권한 설정 | 레지스터마다 읽기(Read), 쓰기(Write), 읽기-쓰기(Read-Write) 권한을 설정하여 메모리 보호 가능. |
| 디버깅 및 유지보수 용이성 | 명확히 정의된 레지스터 맵 덕분에 소프트웨어와 하드웨어 간의 인터페이스가 직관적이고 문제 해결이 용이. |

# SoC 기본 순차논리 - 레지스터 맵 실습

- 아래 코드를 작성하여 시뮬레이션 해 보시오.

```
1   `timescale 1ns / 1ps
2   module SimpleRegister (
3       input  clk,              // 클럭 신호
4       input  reset,            // 리셋 신호
5       input  write_enable,     // 쓰기 활성화 신호
6       input  [7:0] write_data, // 쓰기 데이터
7       output  [7:0] read_data  // 읽기 데이터
8   );
9
10      // 8비트 레지스터 선언
11      reg [7:0] register;
12
13      // 클럭 상승 에지에서 동작
14      always @(posedge clk or posedge reset) begin
15          if (reset) begin
16              register <= 8'b0; // 리셋 시 레지스터 초기화
17          end else if (write_enable) begin
18              register <= write_data; // 쓰기 활성화 시 데이터 저장
19          end
20      end
21
22      // 읽기 데이터는 항상 현재 레지스터 값 출력
23      assign read_data = register;
24
25  endmodule
```

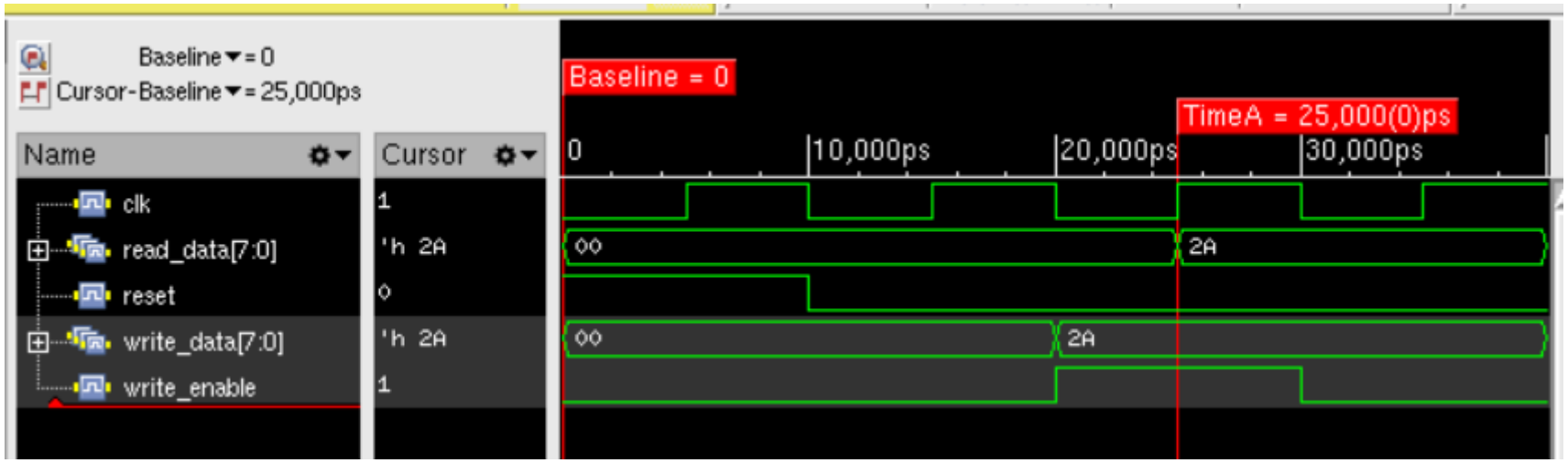
# 레지스터 실습 – 시뮬레이션 벡터

- 아래 코드를 참조하여 시뮬레이션 해보시기 바랍니다.

```
    reg clk;
    reg reset;
    reg write_enable;
    reg [7:0] write_data;
    wire [7:0] read_data;

    // SimpleRegister 인스턴스 생성
    SimpleRegister uut (
        .clk(clk),
        .reset(reset),
        .write_enable(write_enable),
        .write_data(write_data),
        .read_data(read_data)
    );
```

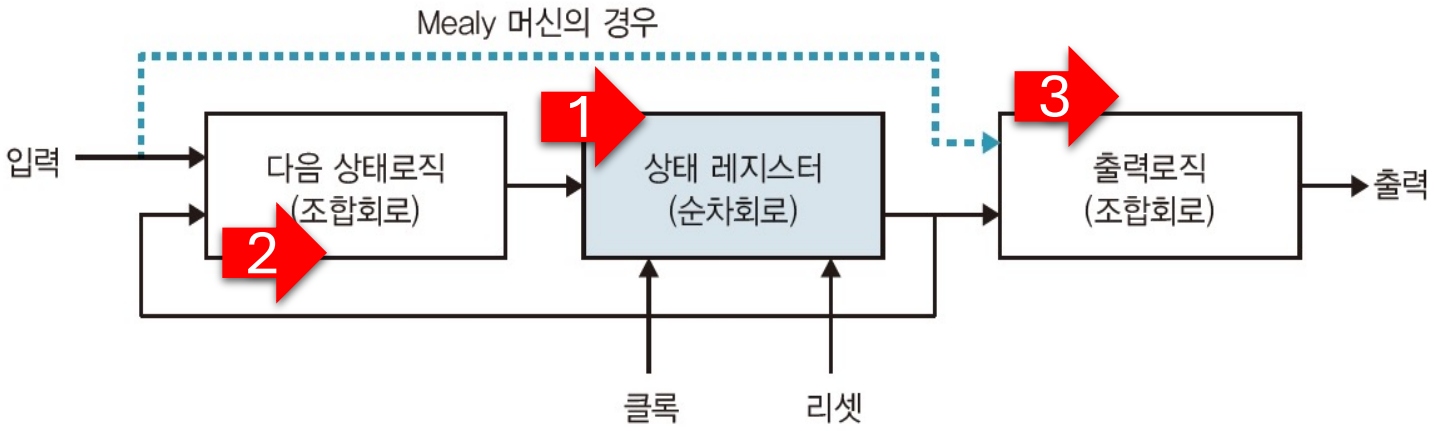
```
    // 클럭 생성기
    initial begin
        clk = 0;
        forever #5 clk = ~clk; // 10ns 주기의 클럭 생성
    end

    initial begin
        // 초기화 및 테스트 시퀀스 시작
        reset = 1; write_enable = 0; write_data = 8'b0;
        #10 reset = 0; // 리셋 비활성화

        // 쓰기 테스트: 값 42 저장
        #10 write_enable = 1; write_data = 8'd42;
        #10 write_enable = 0;

        // 읽기 테스트: 저장된 값 확인
        #10 $display("Read Data: %d", read_data);

        $finish; // 시뮬레이션 종료
    end
```

# 레지스터 맵과 메모리 맵 실습– testbench code결과



# SoC 기본 유한상태머신 회로

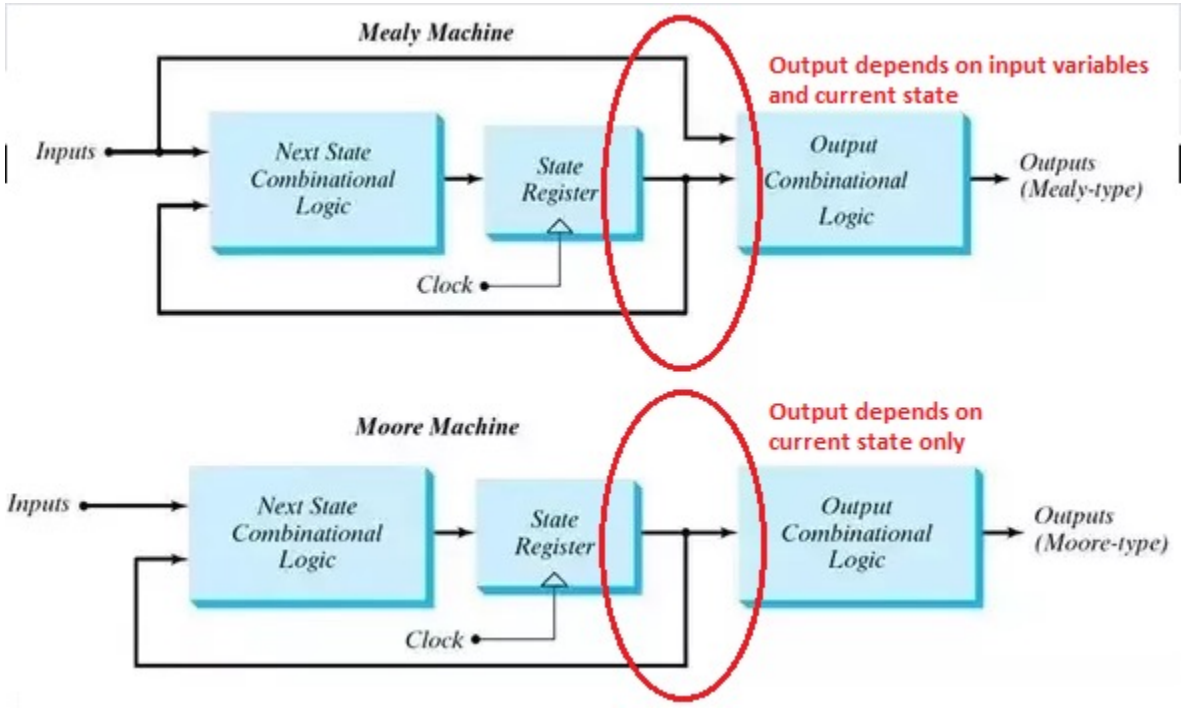
- 유한상태 머신(Finite State Machine; FSM)
  - 정해진 수의 상태를 가지고 상태들 간의 천이에 의해 출력을 생성하는 회로
  - 디지털 시스템의 제어회로 구성에 사용
  - **Moore 머신** : 출력이 단지 현재상태에 의해서 결정
  - **Mealy 머신** : 현재상태와 입력에 의해 출력이 결정




[그림 11-27] 유한상태머신의 구조

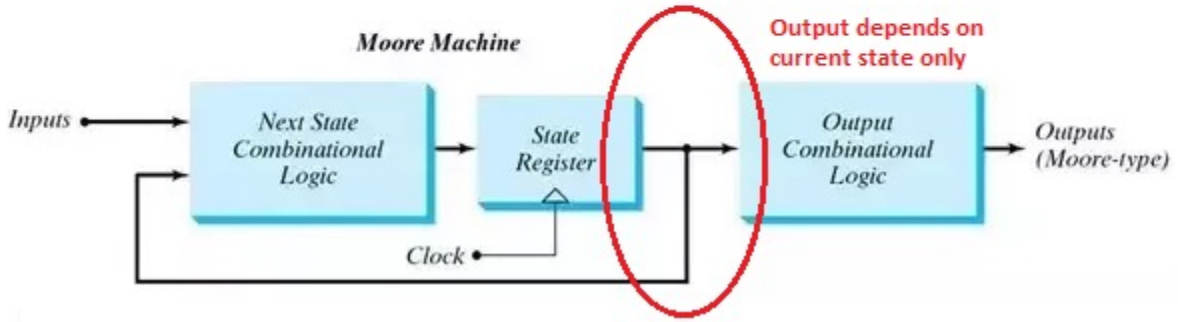
| 로직 종류 | 역할 및 특징 |
|---|---|
| **1. State Sequential Logic** | 클럭을 기준으로 현재 상태(state)를 저장하고 업데이트합니다. |
| **2. Next Combinational Logic** | 입력과 현재 상태에 따라 다음 상태(next_state)를 결정합니다. |
| **3. Output Logic** | Moore FSM에서는 현재 상태만으로 출력을 결정하며, Mealy FSM에서는 입력과 현재 상태 모두를 고려하여 출력을 결정합니다. |

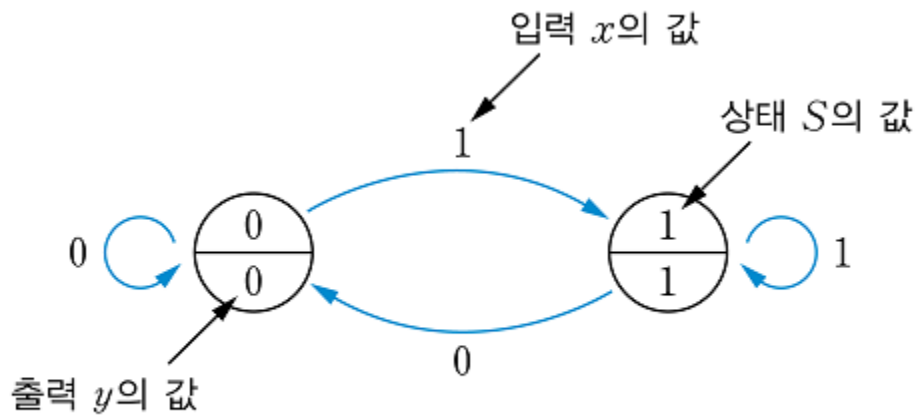
# SoC 기본 유한상태머신 회로




| 특징 | Mealy 머신 | Moore 머신 |
|---|---|---|
| **출력 결정 방식** | 현재 상태와 입력 | 현재 상태 |
| **출력 반응 속도** | 입력 변화에 즉각 반응 | 클럭 주기에 따라 반응 |
| **설계 복잡도** | 상대적으로 복잡 | 상대적으로 단순 |
| **상태 수** | 적음 | 많음 |

# SoC 기본 유한상태머신 회로 - Moore FSM 회로

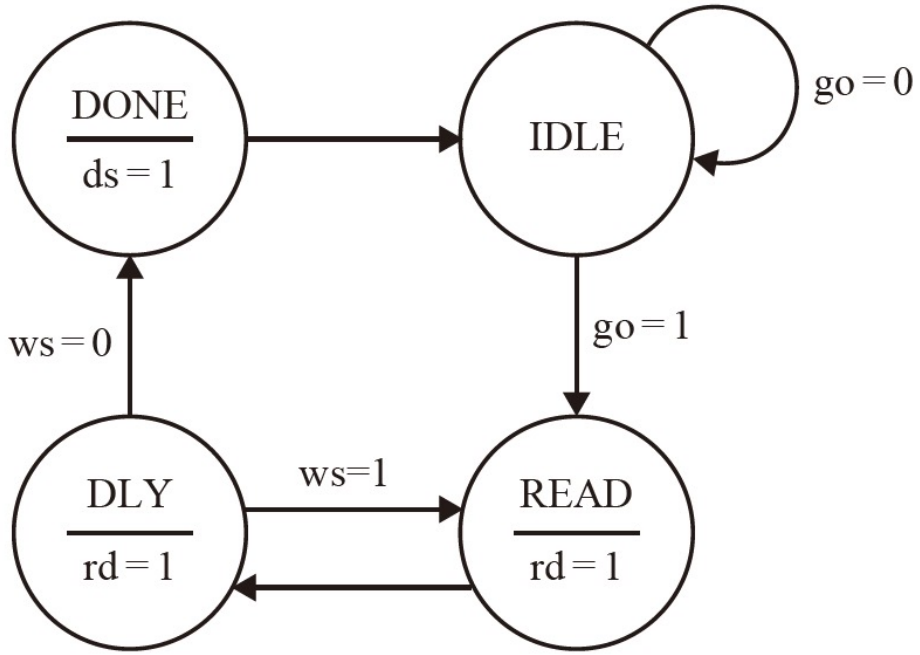



(d) 상태도

# SoC 기본 유한상태머신 회로 - Moore FSM 회로

- 4개의 상태를 갖는 FSM 모델링




[그림 11-28] 4개의 상태를 갖는 FSM 예

| 현재 상태 | 입력 조건 | 다음 상태 | 출력 |
|---|---|---|---|
| **IDLE** | go = 0 | IDLE | 없음 |
| **IDLE** | go = 1 | READ | 없음 |
| **READ** | ws = 1 | DLY | rd = 1 |
| **DLY** | ws = 0 | DONE | rd = 1 |
| **DONE** | N/A | IDLE | ds = 1 |

# SoC 기본 순차논리 - Moore FSM 회로 실습

```
1  `timescale 1ns / 1ps
2  module moore_fsm (
3      input clk,          // 클럭 신호
4      input reset,        // 리셋 신호 (비동기)
5      input go,           // 입력 신호 go
6      input ws,           // 입력 신호 ws
7      output reg rd,      // 출력 신호 rd
8      output reg ds       // 출력 신호 ds
9  );
10
11     // 상태 정의 (4개의 상태)
12     localparam IDLE = 2'b00,
13                READ = 2'b01,
14                DLY  = 2'b10,
15                DONE = 2'b11;
16
17     reg [1:0] state, next_state;  // 현재 상태와 다음 상태
18
19     // 상태 레지스터: 현재 상태를 저장
20     always @(posedge clk or posedge reset) begin
21         if (reset)
22             state <= IDLE;        // 리셋 시 초기 상태는 IDLE
23         else
24             state <= next_state; // 다음 상태로 전환
25     end
26
```

1

2

3

```
28     // 다음 상태 로직: 입력(go, ws)에 따라 상태 전환 결정
29     always @(*) begin
30         case (state)
31             IDLE:
32                 if (go)
33                     next_state = READ;   // go=1이면 READ로 전환
34                 else
35                     next_state = IDLE;   // go=0이면 IDLE 유지
36
37             READ:
38                 if (ws)
39                     next_state = DLY;    // ws=1이면 DLY로 전환
40                 else
41                     next_state = READ;   // ws=0이면 READ 유지
42
43             DLY:
44                 if (!ws)
45                     next_state = DONE;   // ws=0이면 DONE으로 전환
46                 else
47                     next_state = DLY;    // ws=1이면 DLY 유지
48
49             DONE:
50                 next_state = IDLE;       // DONE 이후에는 항상 IDLE로 복귀
51
52             default:
53                 next_state = IDLE;       // 기본값은 IDLE
54         endcase
55     end
```

# SoC 기본 순차논리 - Moore FSM 회로 실습

4

```
56      // 출력 로직: 현재 상태에 따라 출력 결정 (Moore 머신)
57      always @(*) begin
58          case (state)
59              IDLE: begin
60                  rd = 0;
61                  ds = 0;
62              end
63              READ: begin
64                  rd = 1;  // READ 상태에서 rd 활성화
65                  ds = 0;
66              end
67              DLY: begin
68                  rd = 1;  // DLY 상태에서 rd 유지 활성화
69                  ds = 0;
70              end
71              DONE: begin
72                  rd = 0;
73                  ds = 1;  // DONE 상태에서 ds 활성화
74              end
75              default: begin
76                  rd = 0;
77                  ds = 0;
78              end
79          endcase
80      end
81
82  endmodule
```

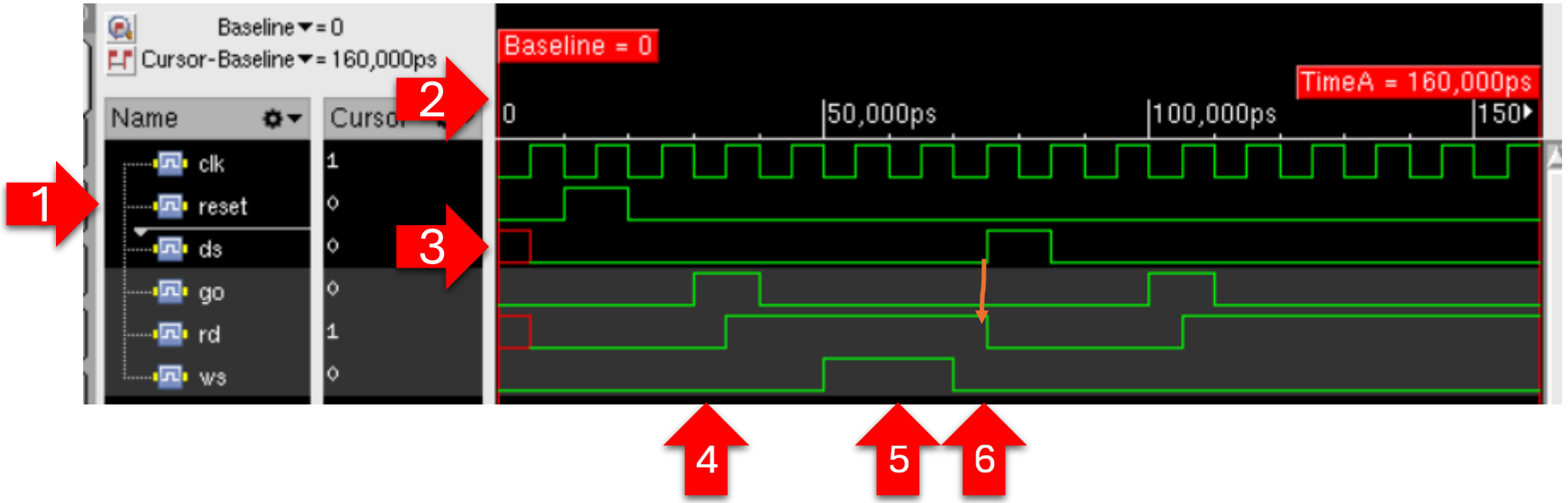
# Moore FSM 회로 실습 – 시뮬레이션 벡터

- 아래 코드를 주석을 참조하여 각각의 경우에 맞는 벡터를 생성해 보시오.

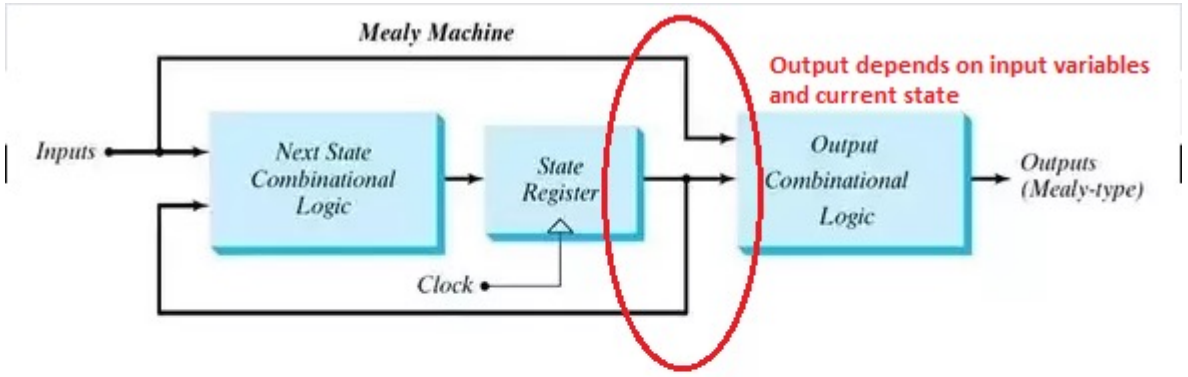
```
1   `timescale 1ns / 1ps
2
3   module tb_moore_fsm;
4
5       // 테스트 벤치에서 사용할 신호 정의
6       reg clk;          // 클럭 신호
7       reg reset;        // 리셋 신호
8       reg go;           // 입력 신호 go
9       reg ws;           // 입력 신호 ws
10      wire rd;          // 출력 신호 rd
11      wire ds;          // 출력 신호 ds
12
13      // DUT (Device Under Test) 인스턴스화
14      moore_fsm uut (
15          .clk(clk),
16          .reset(reset),
17          .go(go),
18          .ws(ws),
19          .rd(rd),
20          .ds(ds)
21      );
22
23      // 클럭 생성: 10ns 주기로 클럭 신호 변경
24      always #5 clk = ~clk;
```

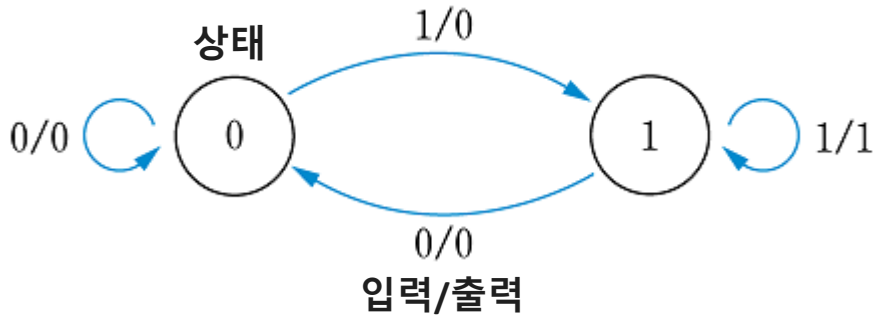
```
27      // 초기화 및 테스트 시퀀스 작성
28      initial begin
29          // 초기 상태 설정
30          clk = 0;
31          reset = 0;
32          go = 0;
33          ws = 0;
34          // 리셋 활성화 (비동기 리셋)
35          #10 reset = 1;
36          #10 reset = 0;
37          // IDLE -> READ 상태로 전환 (go=1)
38          #10 go = 1;
39          #10 go = 0;
40          // READ -> DLY 상태로 전환 (ws=1)
41          #10 ws = 1;
42          // DLY -> DONE 상태로 전환 (ws=0)
43          #20 ws = 0;
44          // DONE -> IDLE 상태로 복귀 (자동 복귀)
45          #20;
46          // 추가 테스트: 다시 READ 상태로 전환
47          #10 go = 1;
48          #10 go = 0;
49          // 종료 조건
50          #50 $stop;
51      end
52      // 모니터링: 출력 값 확인
53      initial begin
54          $monitor("Time=%0t | clk=%b | reset=%b | go=%b | ws=%b | rd=%b | ds=%b | state=%b",
55                   $time, clk, reset, go, ws, rd, ds, uut.state);
56      end
57
58  endmodule
```

# Moore FSM 회로 실습– testbench code결과



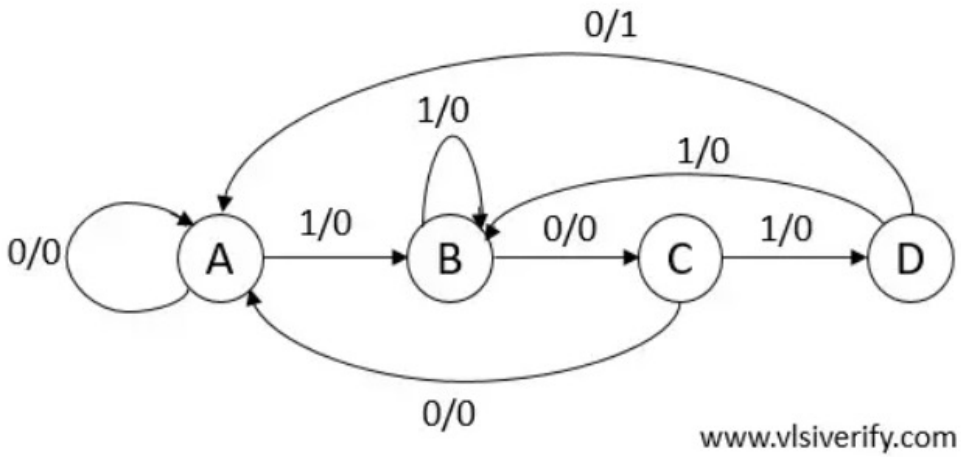
# SoC 기본 순차논리 - Mealy FSM 회로 실습



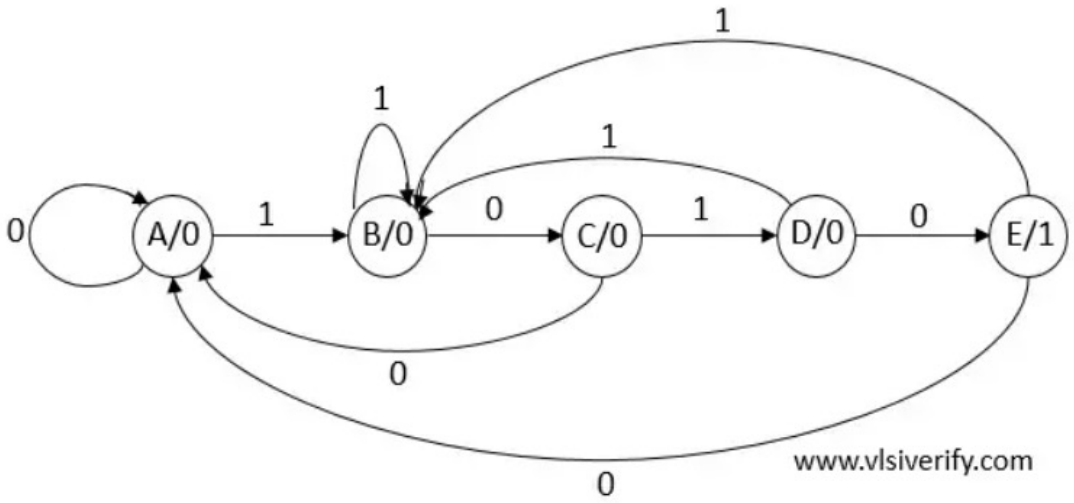



# SoC 기본 유한상태머신 회로 - Mealy FSM 회로

- 1010 detector Mealy vs Moore




1010 Non-Overlapping Mealy Sequence Detector




1010 Non-Overlapping Moore Sequence Detector

# SoC 기본 순차논리 - Mealy FSM 회로

1

```
`timescale 1ns / 1ps
module seq_detector_1010(input clk, rst_n, x, output z);
  parameter A = 4'h1;
  parameter B = 4'h2;
  parameter C = 4'h3;
  parameter D = 4'h4;

  reg [3:0] state, next_state;
```

2

```
  always @(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
      state <= A;
    end
    else state <= next_state;
  end
```

| 상태 | 의미 (탐지한 입력 시퀀스) |
|---|---|
| A | 초기 상태 (아무것도 감지되지 않음) |
| B | 입력에서 '1'을 감지함 |
| C | 입력에서 '10'을 감지함 |
| D | 입력에서 '101'을 감지함 |

3

```
always @(state or x) begin
  case(state)
    A: begin
         if(x == 0) next_state = A;
         else       next_state = B;
       end
    B: begin
         if(x == 0) next_state = C;
         else       next_state = B;
       end
    C: begin
         if(x == 0) next_state = A;
         else       next_state = D;
       end
    D: begin//This state only differs when
      if(x == 0) next_state = A;
         else       next_state = B;
       end
    default: next_state = A;
  endcase
end
```

4

```
assign z = (state == D) && (x == 0)? 1:0;
```

# Mealy FSM 회로 실습 – 시뮬레이션 벡터

```
1   `timescale 1ns/1ps
2   module TB;
3     reg clk, rst_n, x;
4     wire z;
5
6     seq_detector_1010 sd(clk, rst_n, x, z);
7     initial clk = 0;
8     always #2 clk = ~clk;
```

1 2 3

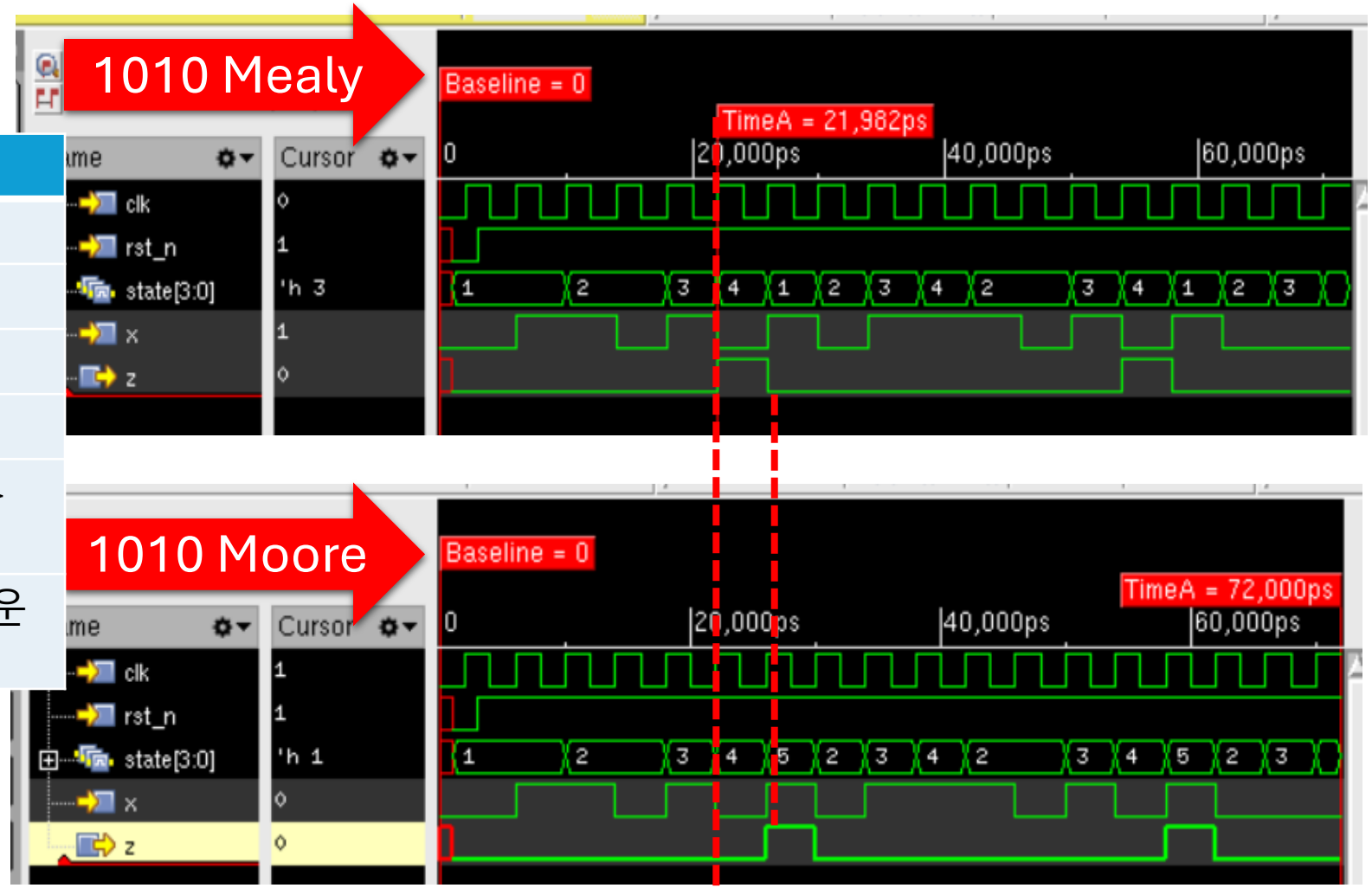
```
10    initial begin
11      x = 0;
12      #1 rst_n = 0;
13      #2 rst_n = 1;
14
15      #3 x = 1;
16      #4 x = 1;
17      #4 x = 0;
18      #4 x = 1;
19      #4 x = 0;
20      #4 x = 1;
21      #4 x = 0;
22      #4 x = 1;
23      #4 x = 1;
24      #4 x = 1;
25      #4 x = 0;
26      #4 x = 1;
27      #4 x = 0;
28      #4 x = 1;
29      #4 x = 0;
30      #10;
31      $finish;
32    end
33
34    initial begin
35      // Dump waves
36      $dumpfile("dump.vcd");
37      $dumpvars(0);
38    end
39  endmodule
```

4

| 시간(ns) | 입력 x | 설명 |
|---|---|---|
| **초기값** | x=0 | 초기 상태 |
| **#3** | x=1 | '1' 감지 시작 |
| **#7** | x=0 | '10' 감지 |
| **#11** | x=1 | '101' 감지 |
| **#15** | x=0 | '1010' 패턴 완성 → 출력(z) 활성화 |
| **이후** | 다양한 값 | 중첩되지 않는 새로운 패턴 감지 테스트 |

# Mealy FSM 회로 실습– testbench code결과

| 시간(ns) | 입력 x | 설명 |
|---|---|---|
| 초기값 | x=0 | 초기 상태 |
| #3 | x=1 | '1' 감지 시작 |
| #7 | x=0 | '10' 감지 |
| #11 | x=1 | '101' 감지 |
| #15 | x=0 | '1010' 패턴 완성 → 출력(z) 활성화 |
| 이후 | 다양한 값 | 중첩되지 않는 새로운 패턴 감지 테스트 |




Mealy와 Moore의 차이 : 상태도에 따른 출력 1clock의 차이가 발생함

# Thanks