

# SPI & I2C 설계

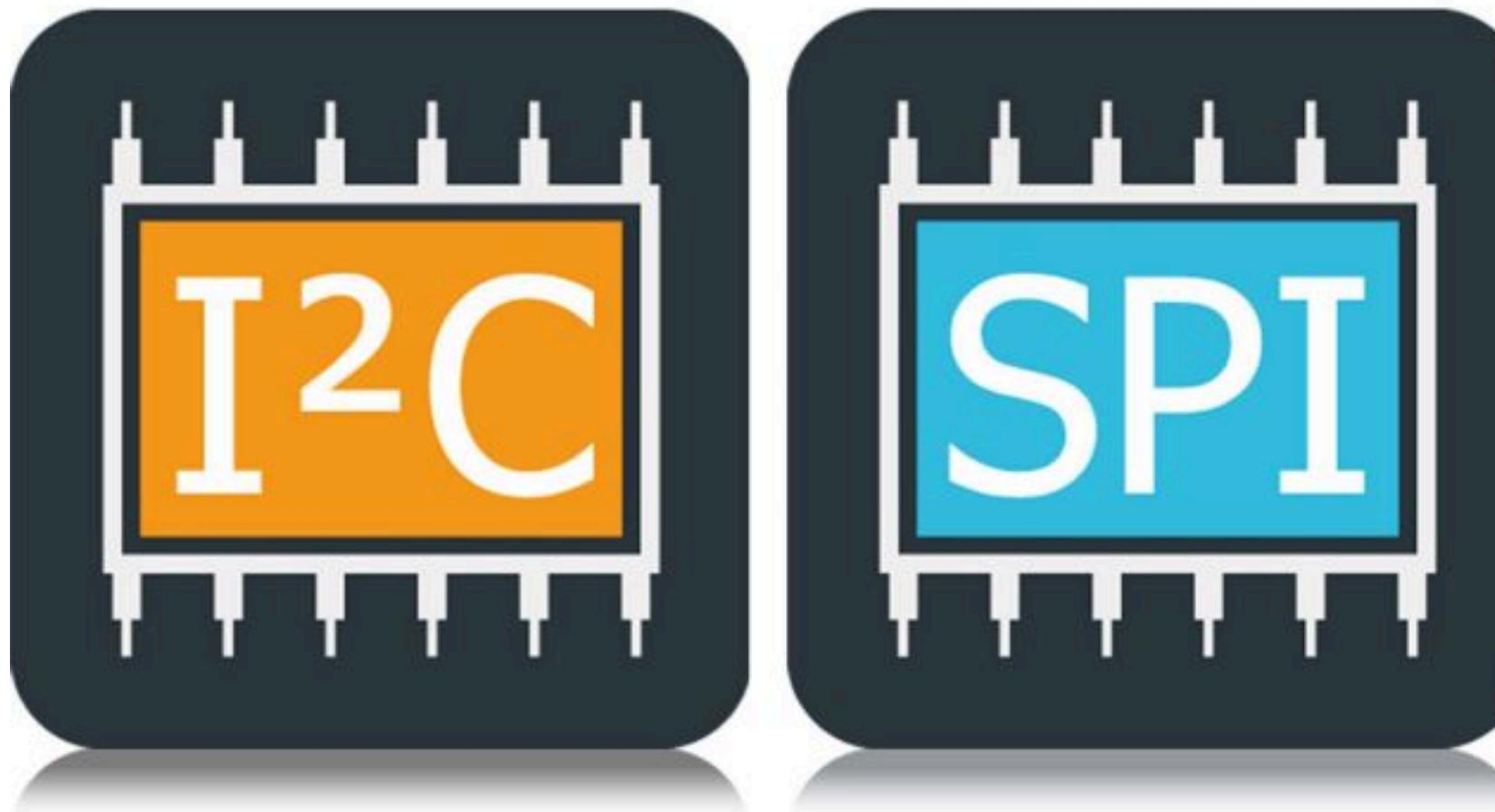
발표자 : 안유한

# 목차

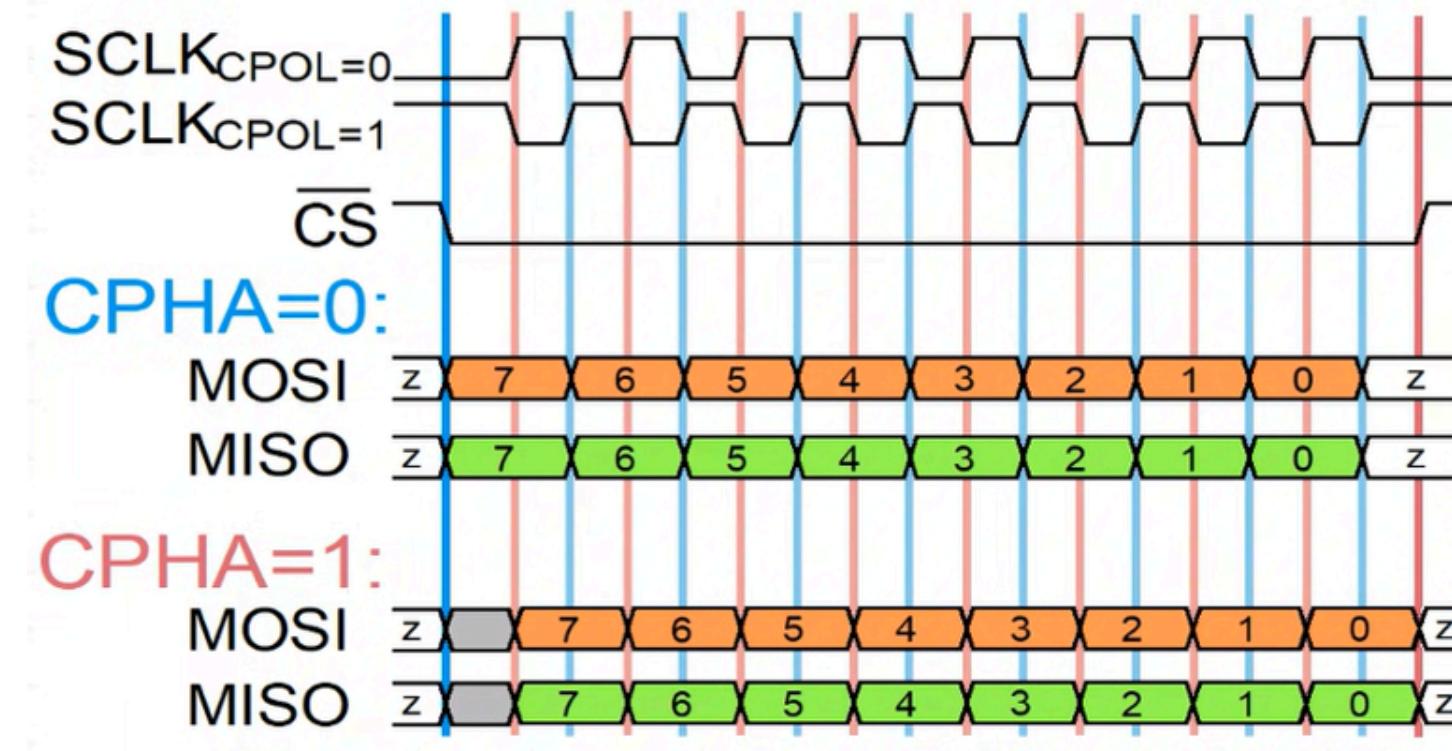
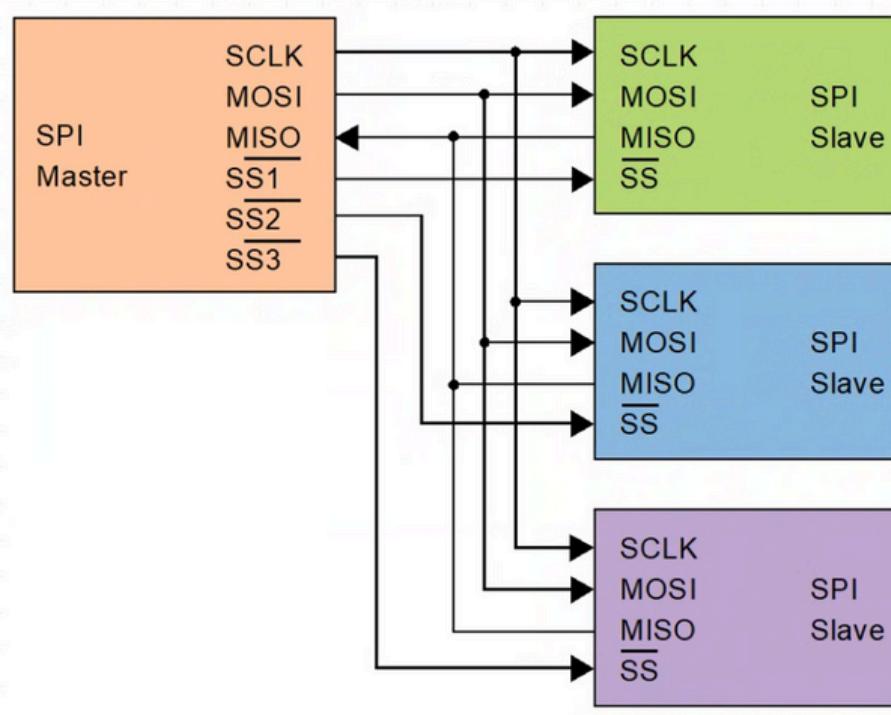
- 01**      프로젝트 목표
- 02**      SPI 설계
- 03**      I2C 설계
- 04**      트러블슈팅
- 05**      느낀점

# 프로젝트 목표

- SPI/I2C 통신 프로토콜 이해 및 설계
- Microblaze 및 AXI4-Lite Bus 연결
- 보드간 통신 어플리케이션 구현
- Synopsys VCS를 통해 검증



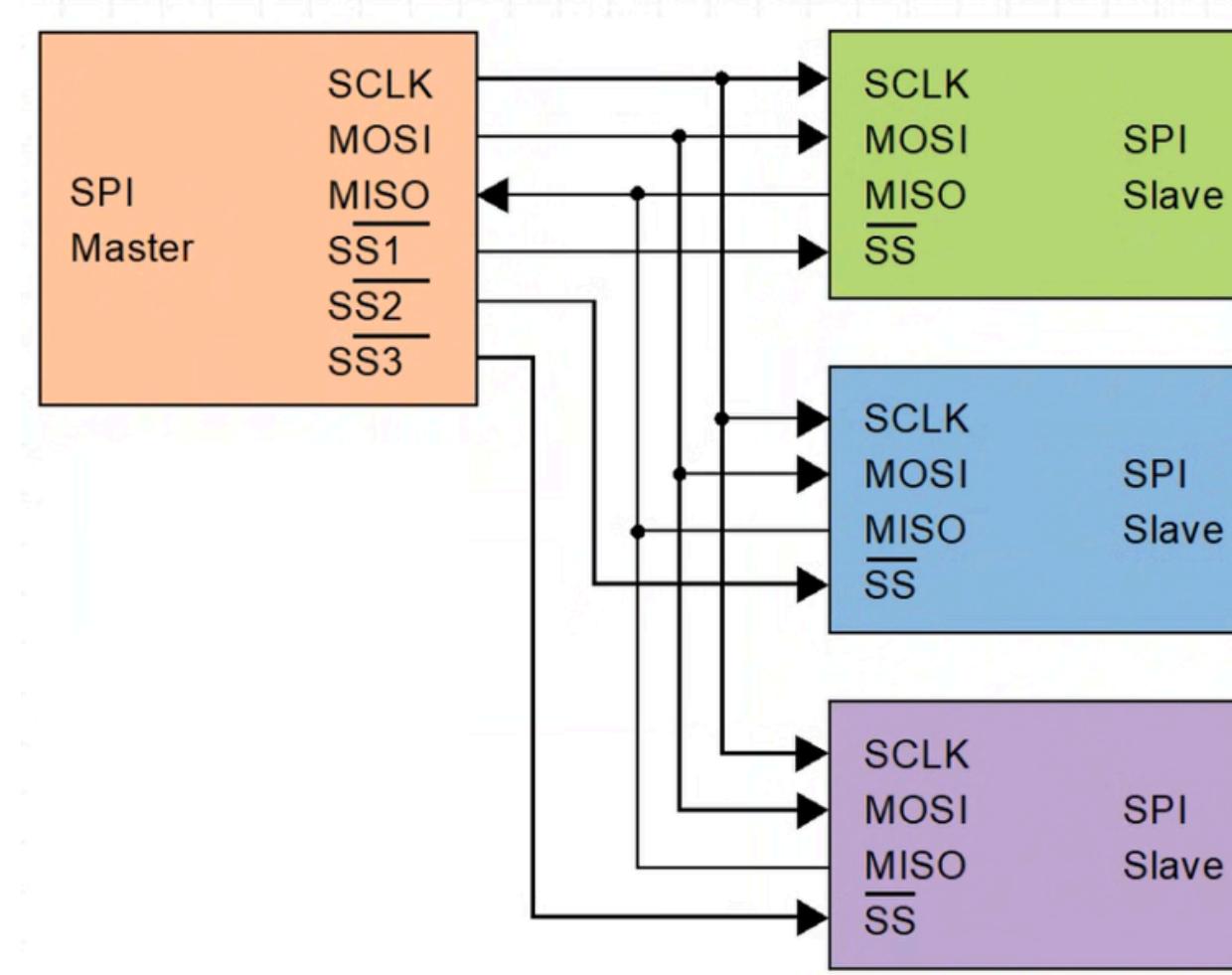
# SPI 설계



SPI는 Serial Peripheral Interface의 약자로, 하드웨어(주변 장치) 간 데이터 교환을 위한 동기식 직렬 통신 프로토콜

- 1:N(1 대 다수)의 통신을 지원한다.
- 한 번에 여러 장치와 데이터를 주고 받을 수는 있지만, 그만큼 통신선이 많이 필요하다는 단점이 있다.
- 동시에 여러 장치와 송수신이 가능
- I2C에 비해 속도가 빠르다
- 간단하게 배선 가능

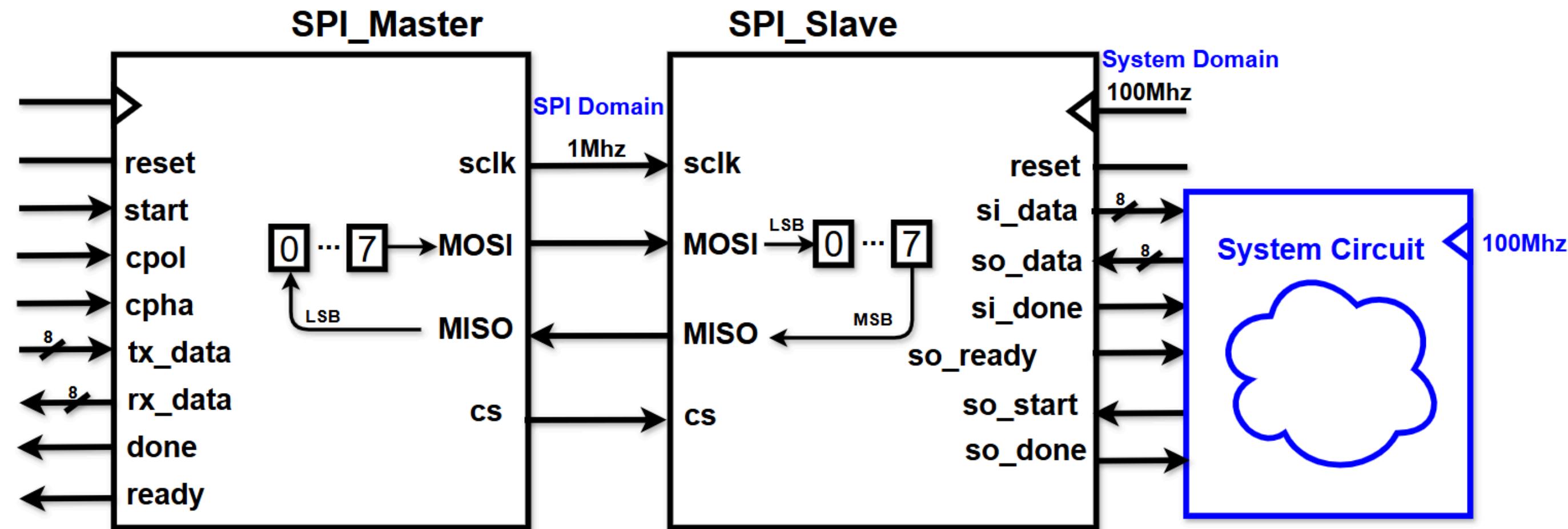
# SPI 설계



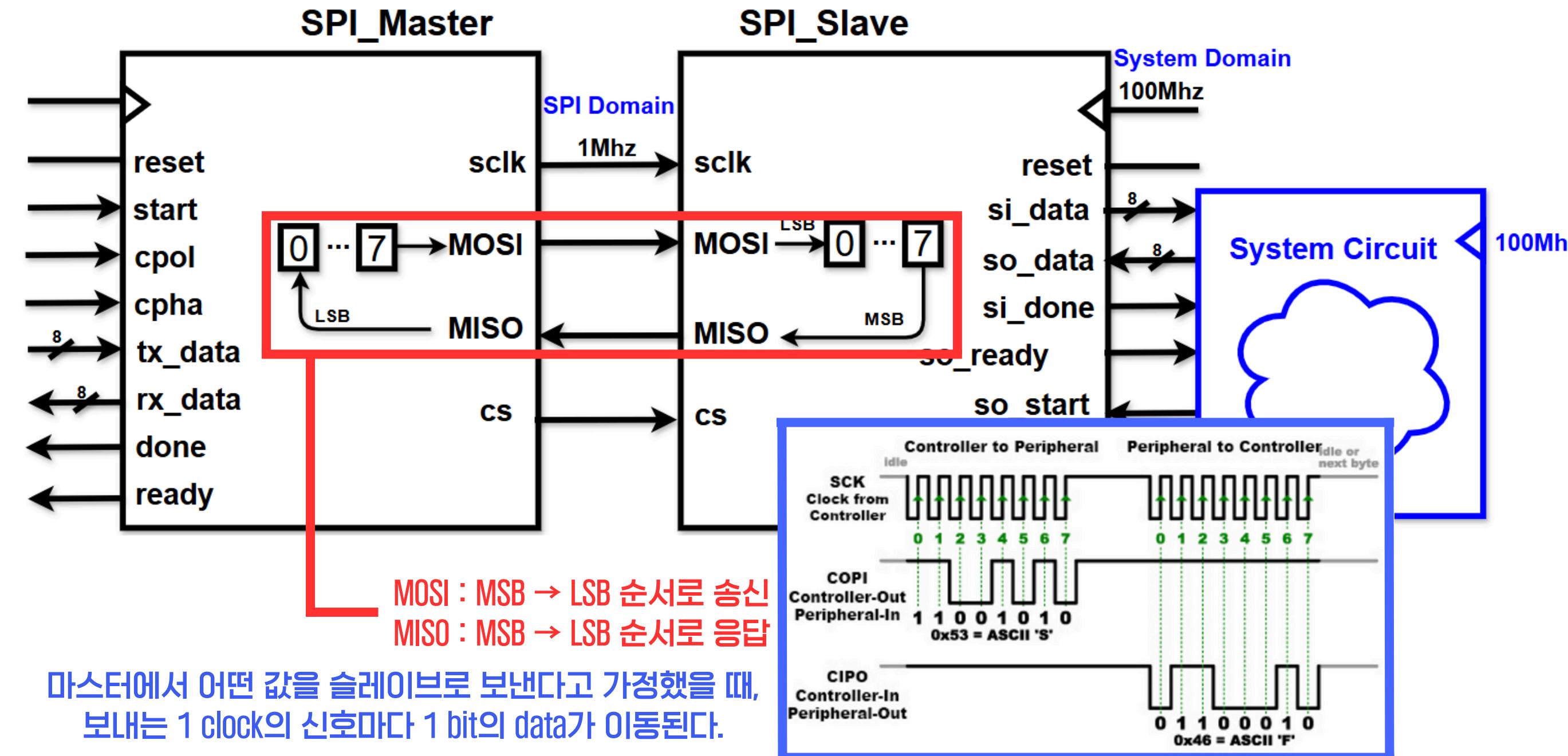
## SPI 구조

- SCLK·MOSI·MISO는 모든 슬레이브와 공통 연결된다.
- 마스터가 내는 클럭(SCLK)에 맞춰
  - MOSI는 마스터 → 슬레이브
  - MISO는 슬레이브 → 마스터
- 슬레이브 선택신호(SS)는 장치마다 하나씩 존재한다.
  - SS1, SS2, SS3 중 하나만 LOW로 내려가는 순간 통신
  - 선택되지 않은 슬레이브는 MISO를 High-Z로 둔다.
  - 신호 충돌이 일어나지 않는다.

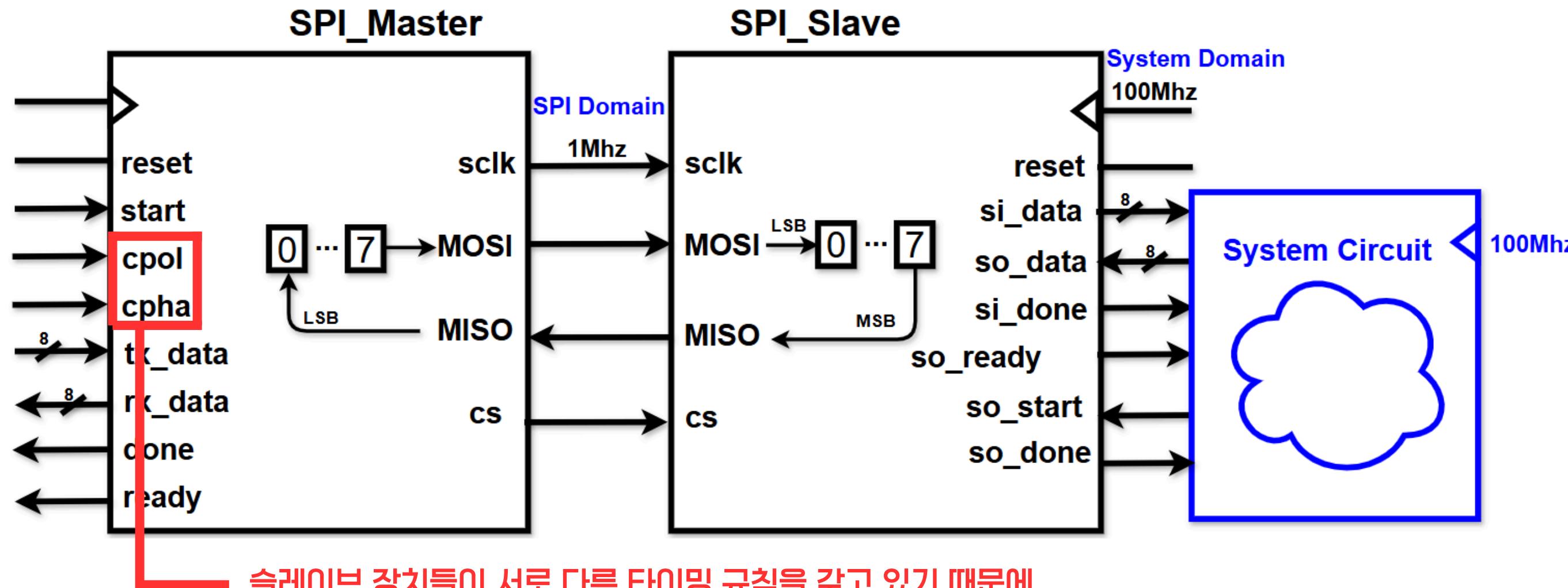
# SPI 설계



# SPI 설계



# SPI 설계

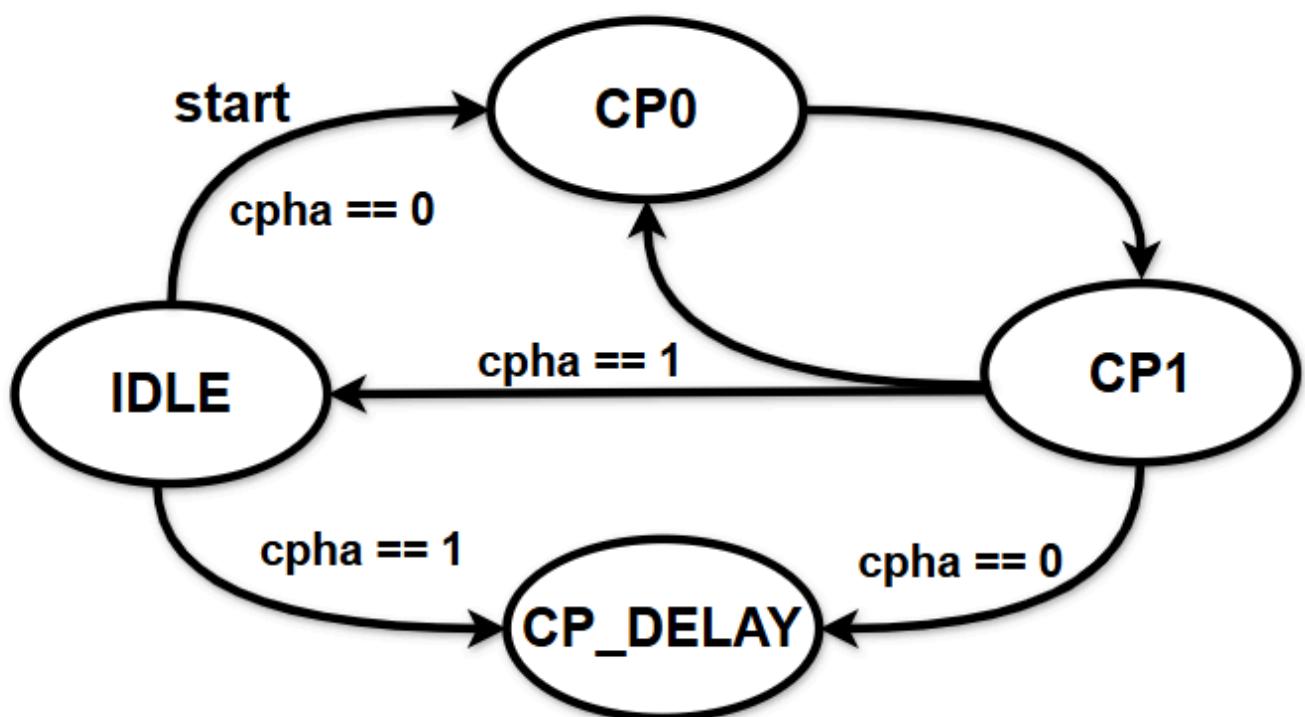


슬레이브 장치들이 서로 다른 타이밍 규칙을 갖고 있기 때문에,  
마스터가 타이밍을 맞춰줄 수 있도록 CPOL/CPHA 두 설정이 필수적으로 존재

- CPOL은 클럭의 기본 상태를 결정                  CPHA = 0 → 첫 번째 엣지에서 샘플링
- CPHA는 데이터의 읽기 시점을 결정하는 방식      CPHA = 1 → 두 번째 엣지에서 샘플링

# SPI 설계

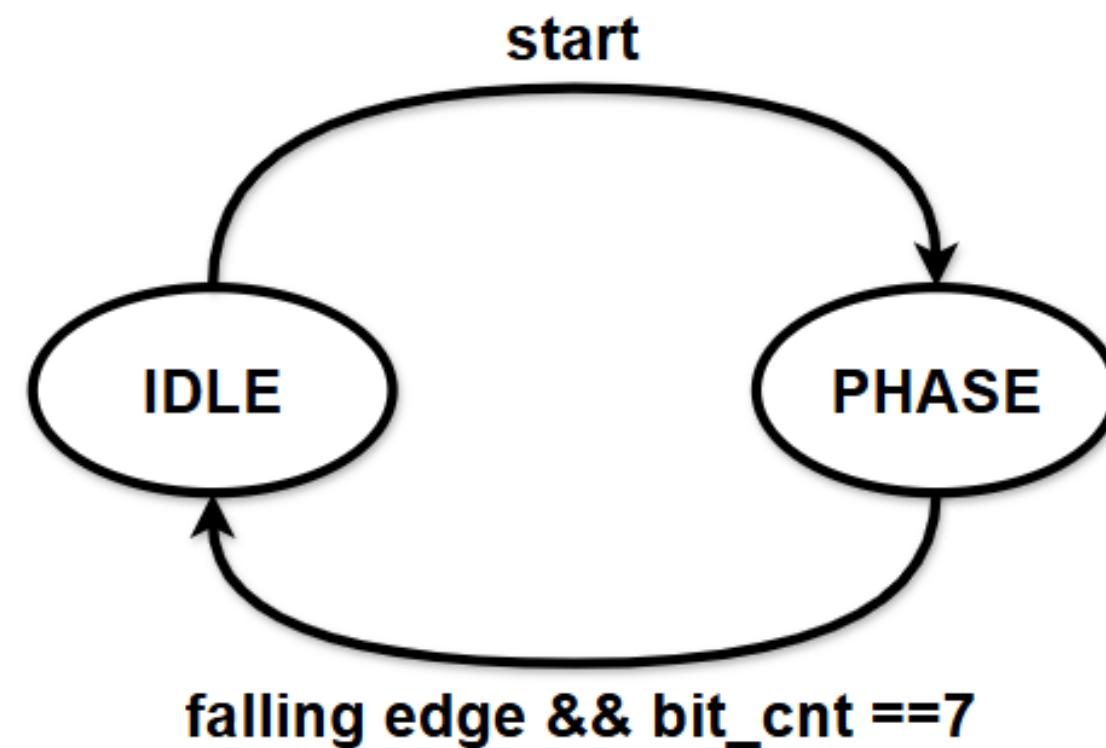
## Master FSM



- **IDLE**
  - SPI 통신을 준비하는 기본 대기 상태
- **CP0**
  - SCLK 반 주기 동안 기다리다가 MISO 입력을 샘플링하고 다음 단계로 이동
- **CP1**
  - SCLK 반 주기 동안 MOSI 비트를 시프트
  - 비트 하나 전송이 끝났는지 체크
  - 마지막 비트면 종료 처리
  - CPHA값에 따라 IDLE or CP\_DELAY로 이동
- **CP\_DELAY**
  - SPI Mode에 따라 종료 타이밍을 조정하는 보정 단계

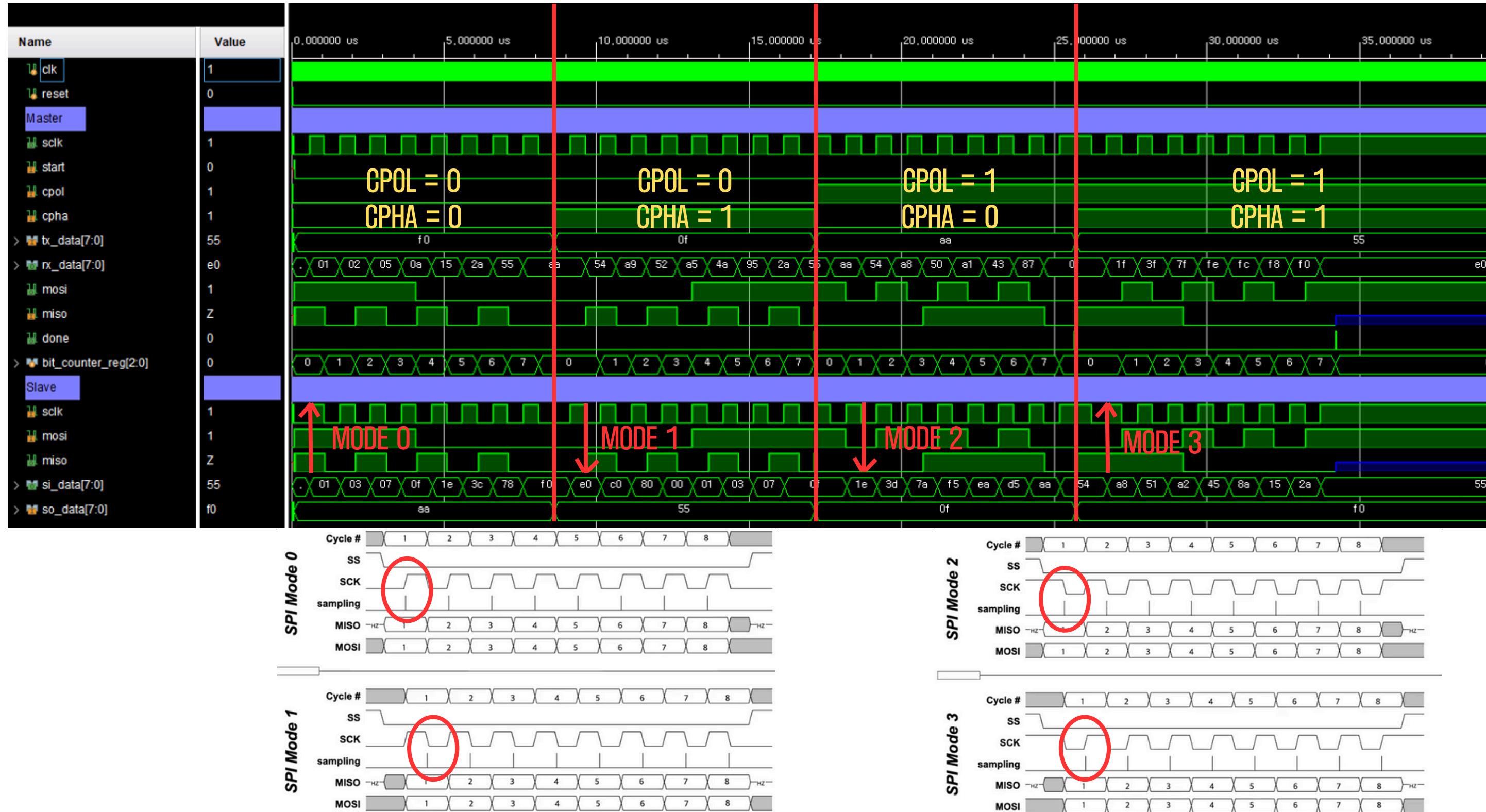
# SPI 설계

## Slave FSM

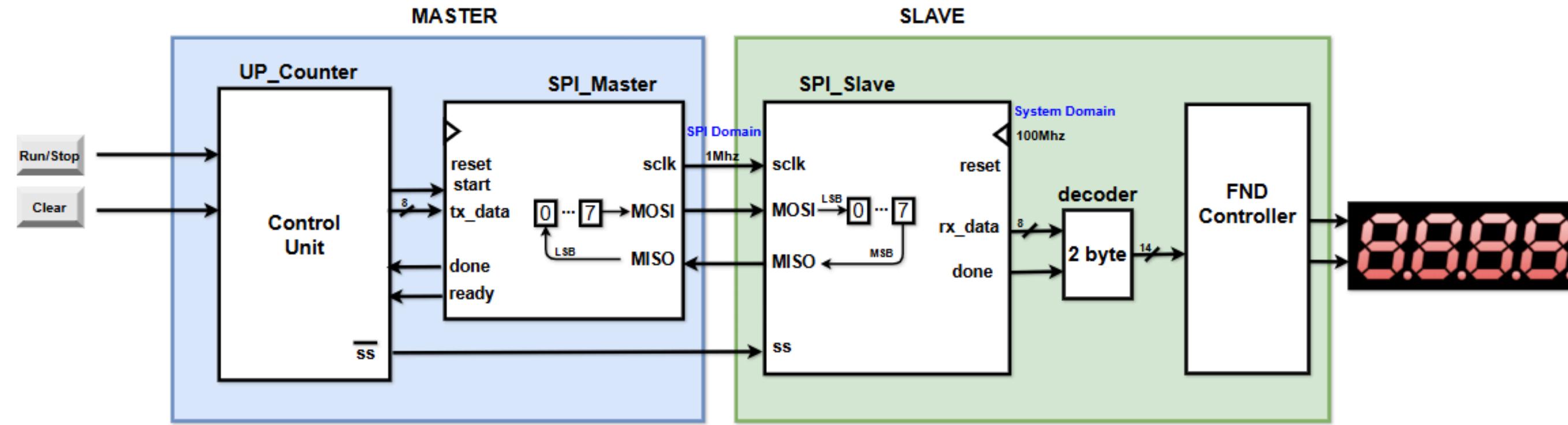


- **IDLE**
  - SPI Master가 슬레이브를 선택하지 않았으므로 MOSI 신호를 받지 않고 대기함
  - CS가 Low로 떨어지면 비트 수신 준비
- **PHASE**
  - Sclk rising edge마다 MOSI 비트를 하나씩 시프트 하여 수신
  - 총 8비트(0~7)를 채우면 수신 완료
  - CS가 High로 올라가면 통신종료 후 IDLE 돌아감

# SPI 설계 Simulation



# SPI 설계



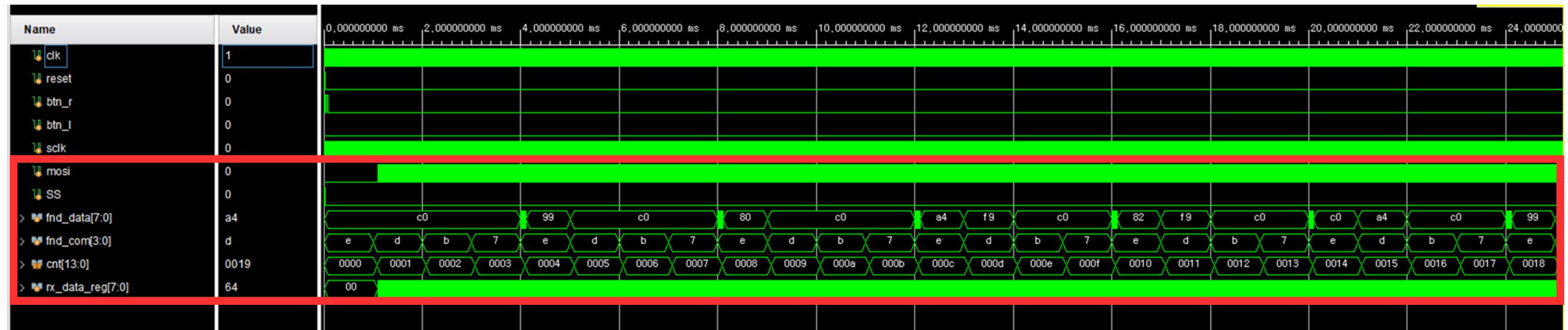
## Master MODULE

- 카운터 값과 버튼 입력을 기반으로 SPI 전송 시작(Start) 전송 데이터(tx\_data) Slave 선택 신호(ss)를 제어한다.
- Up\_Counter의 Count값인 8bit 데이터를 Slave에 전달,

## Slave MODULE

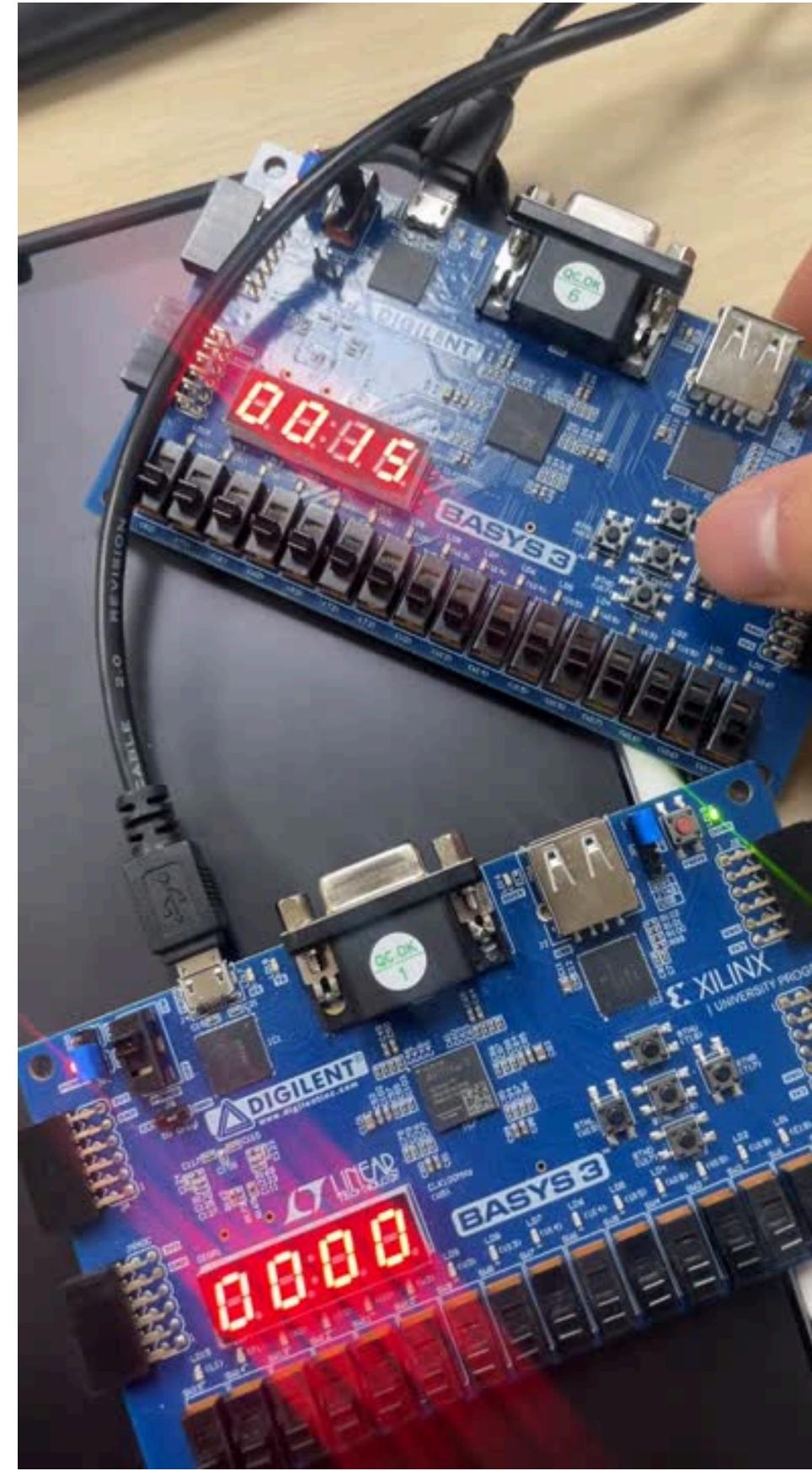
- 카운터에서 들어온 8bit 데이터를 직렬 데이터(MOSI)로 변환하여 전송한다.
- SCLK 생성, SS 제어, MISO 수신, CPOL/CPHA 반영, 전송 완료(done)

# SPI 설계 Simulation



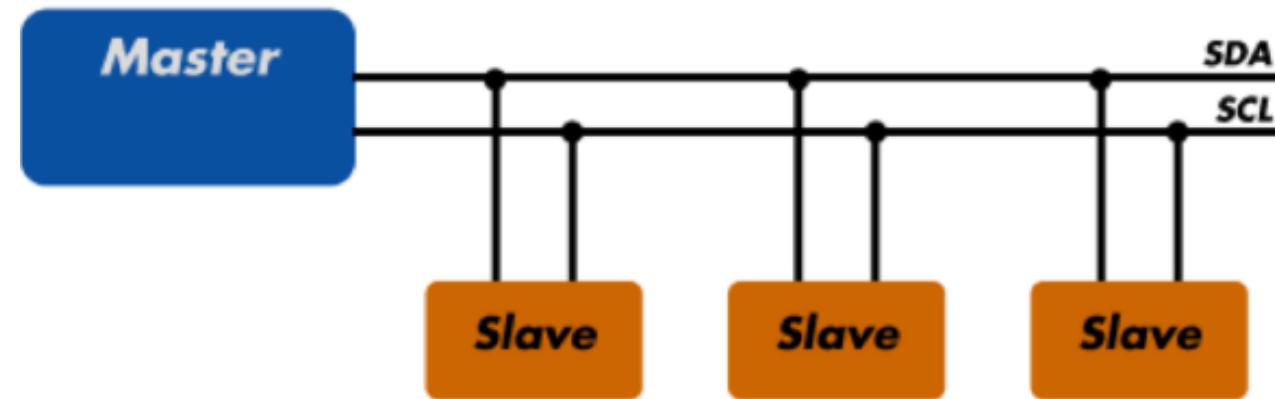
cnt → 슬레이브쪽에서 전달 받은 tx\_data 2개 → [13:0] cnt 값

# 동작 영상



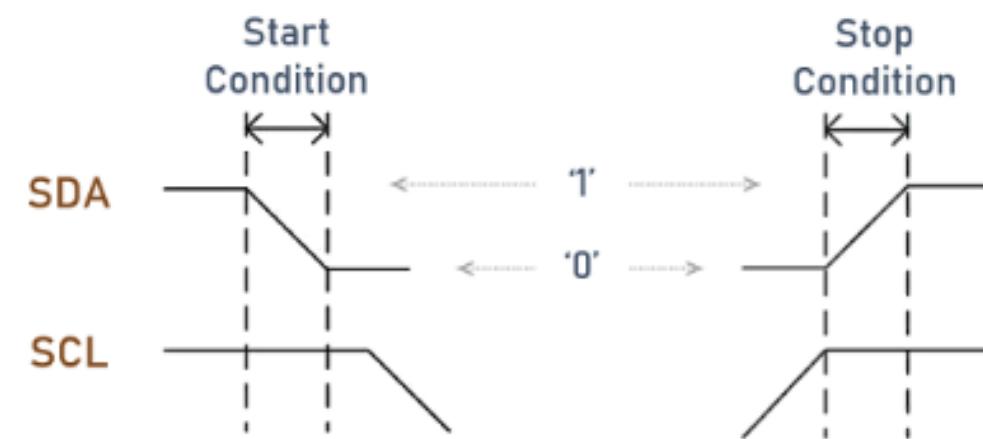
# I2C 설계

I2C는 Inter-Integrated Circuit의 약자로, 데이터 라인 (SDA)과 하나의 클럭 라인 (SCL)을 이용하는 동기식 시리얼 통신 방식이다.



- 1:N(다수 슬레이브) 통신을 지원한다.
- 주소 기반으로 슬레이브 선택이 가능하다.
- **오픈드레인** 구조로 신호 충돌 없이 다중 장치 연결이 가능하다.
- SPI에 비해 속도는 느리지만 안정적이다.

오픈드레인은 ‘0은 직접 만들고, 1은 풀업저항이 만들도록 비워두는 출력 구조’이다. 여러 장치가 같은 신호선을 동시에 사용할 때 충돌을 방지하기 위해 사용된다.



- SDA가 먼저 LOW로 내려가면 START, SCL이 먼저 HIGH로 올라가면 STOP이며, 데이터는 SCL이 High일 때만 유효하다.

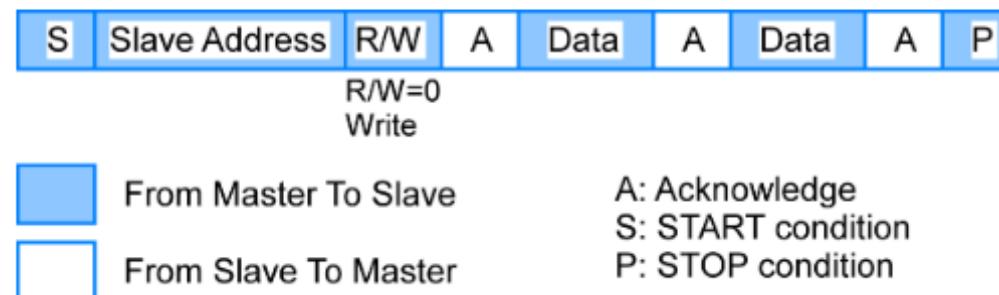
# I2C 설계

## I2C 구조



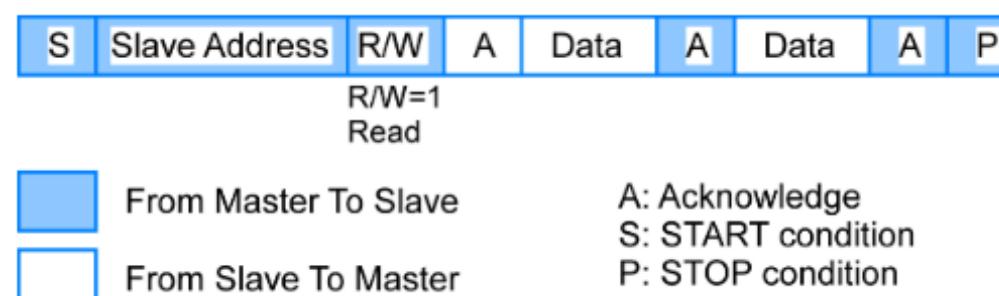
### 데이터 형식

- I2C의 한 프레임은 8비트 데이터 + 1비트 ACK로 구성된다.
- 마스터는 8비트 데이터를 보낸 뒤, 다음 클럭 구간에서 슬레이브의 ACK를 기다린다.
- 슬레이브는 8비트를 모두 받은 후,
- SCL이 HIGH인 동안 SDA를 LOW로 유지하여 ACK를 응답한다.
- 슬레이브가 SDA를 LOW로 당기면 → 정상 수신(ACK)
- SDA를 그대로 HIGH(?)로 두면 → 비정상 또는 거부(NACK)



### Write

- 마스터가 START → 슬레이브 주소 → R/W → 데이터 → ACK들을 주고받고 → STOP



### Read

- 마스터가 START → 슬레이브 주소 → R/W=1로 읽기 요청을 하고, 슬레이브가 데이터 전송 → 마스터는 ACK/NACK → STOP

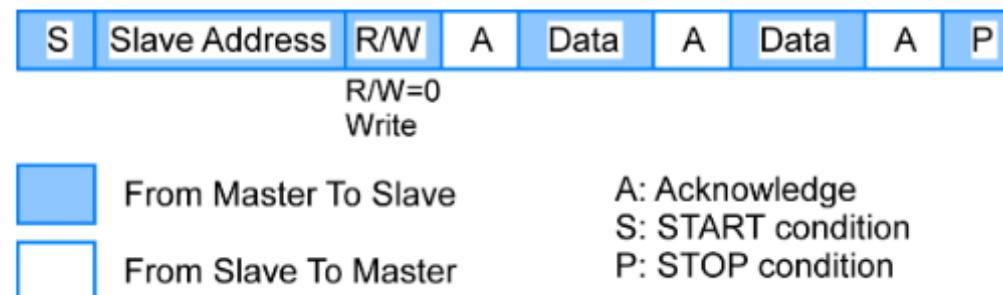
# I2C 설계

## I2C 구조



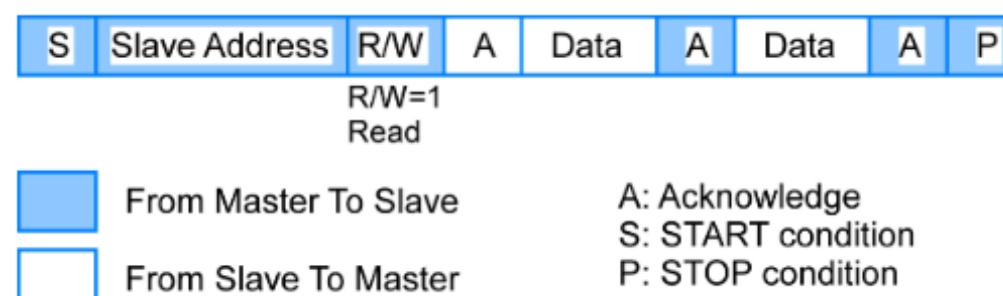
### 데이터 형식

- I2C의 한 프레임은 8비트 데이터 + 1비트 ACK로 구성된다.
- 마스터는 8비트 데이터를 보낸 뒤, 다음 클럭 구간에서 슬레이브의 ACK를 기다린다.
- 슬레이브는 8비트를 모두 받은 후,
- SCL이 HIGH인 동안 SDA를 LOW로 유지하여 ACK를 응답한다.
- 슬레이브가 SDA를 LOW로 당기면 → 정상 수신(ACK)
- SDA를 그대로 HIGH(?)로 두면 → 비정상 또는 거부(NACK)



### Write

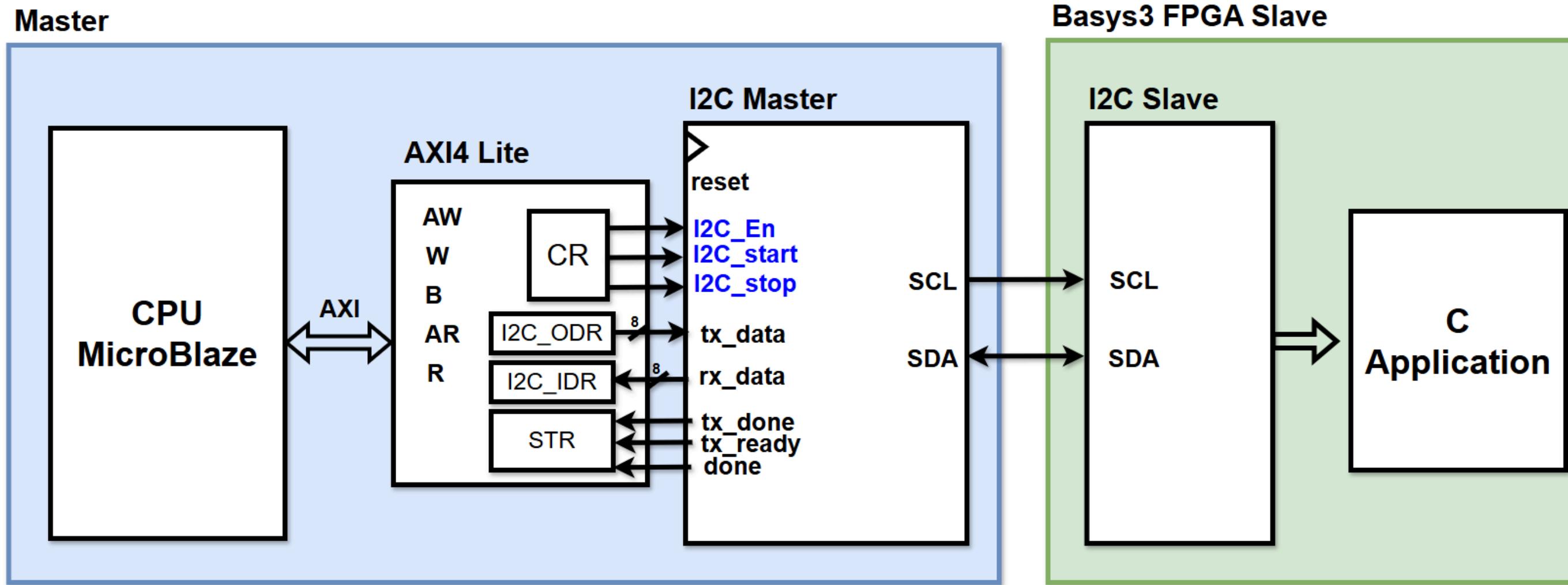
- 마스터가 START → 슬레이브 주소 → R/W → 데이터 → ACK들을 주고받고 → STOP



### Read

- 마스터가 START → 슬레이브 주소 → R/W=1로 읽기 요청을 하고, 슬레이브가 데이터 전송 → 마스터는 ACK/NACK → STOP

# I2C 설계

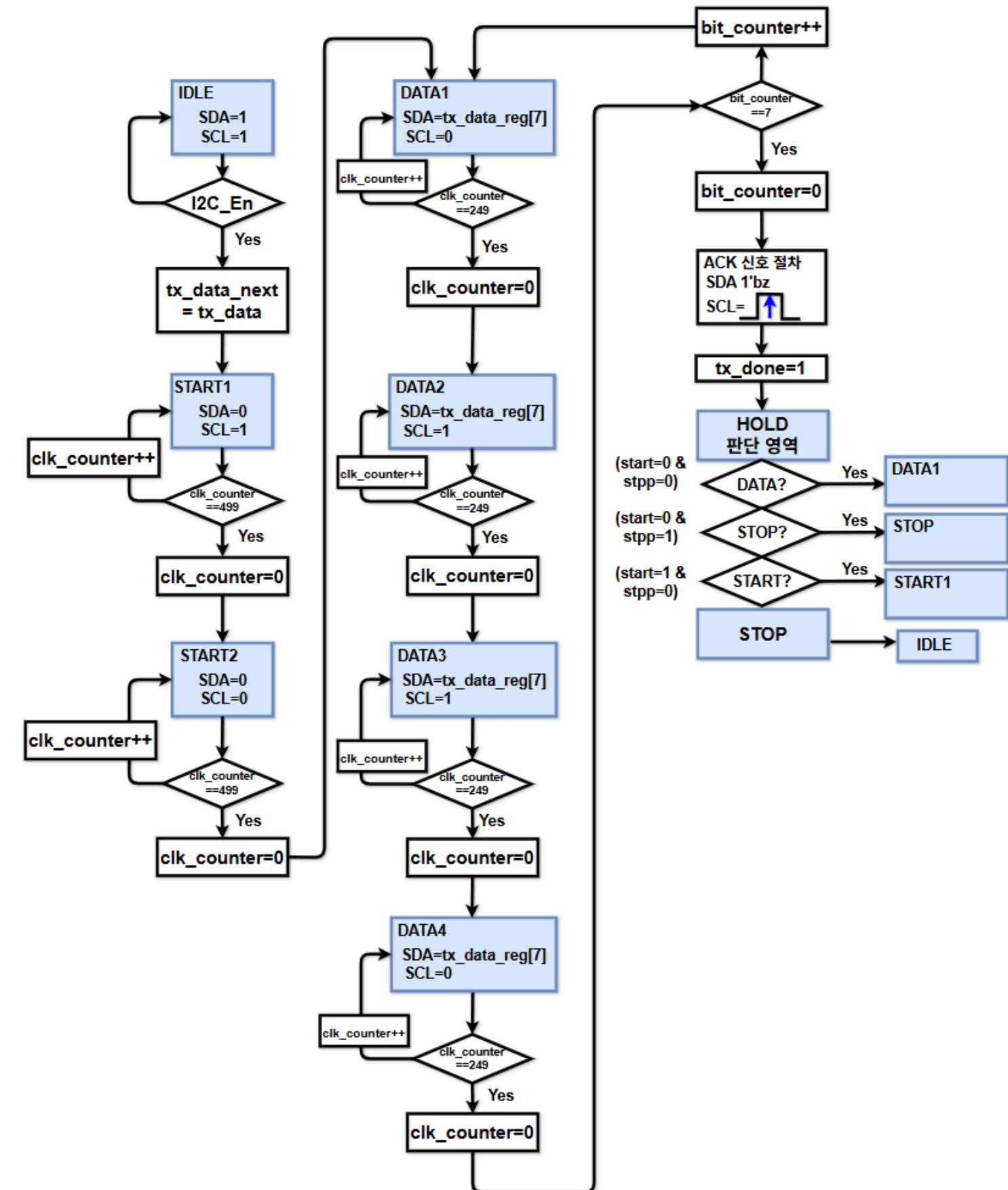


MicroBlaze가 AXI 레지스터를 통해 Master의 컨트롤, 입출력, 상태 레지스터를 제어하고,  
SCL/SDA로 Basys3 I2C Slave와 데이터를 교환하여 C 로직이 이를 처리하는 구조

# I2C 설계

## Master ASM (write)

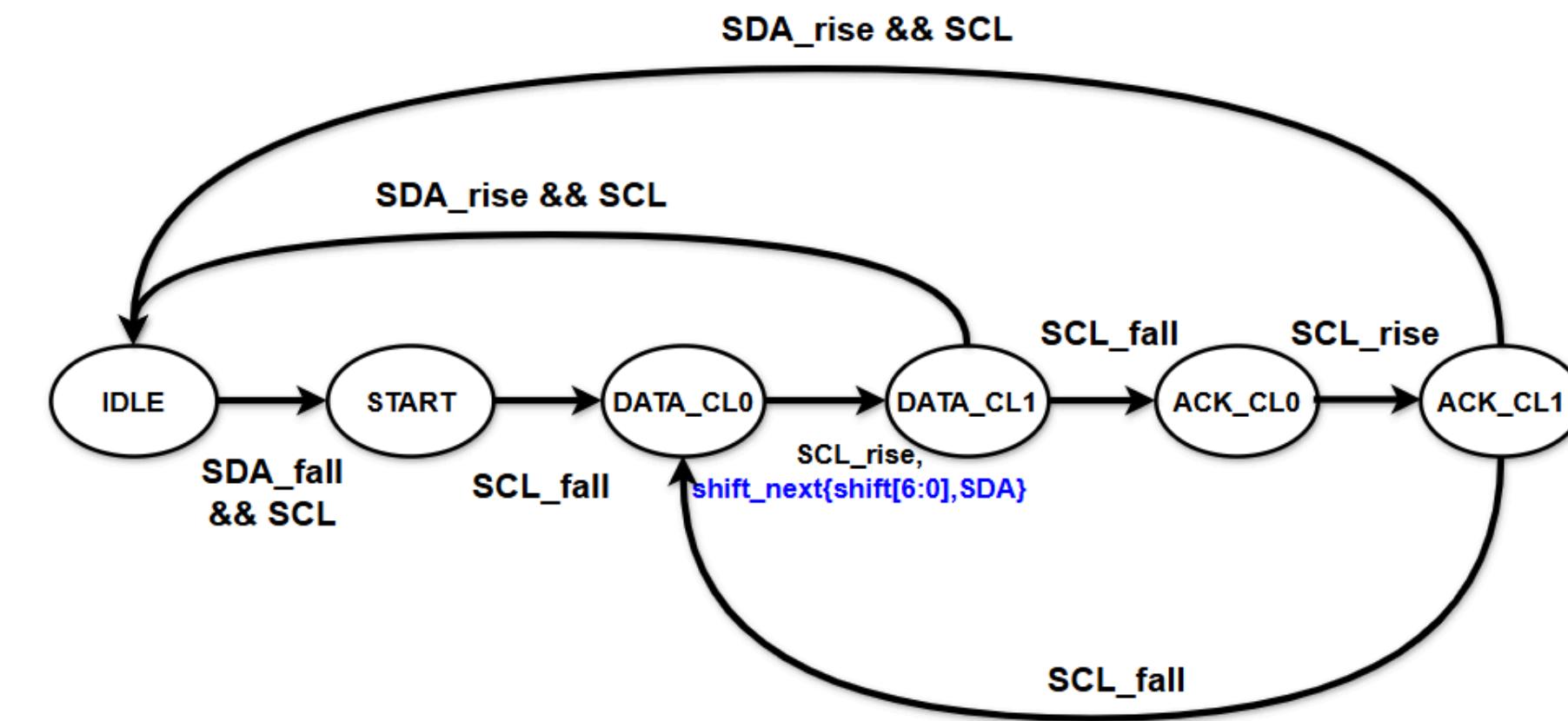
- IDLE**
  - 버스를 유휴 상태로 유지하고 명령을 대기하는 상태
  - SDA=1/SCL=1로 I2C 대기 조건을 유지한다
- START1**
  - SDA를 먼저 LOW로 내려 START 조건을 만드는 단계
  - I2C 통신의 시작 신호를 형성한다.
- START2**
  - START 신호를 안정화하는 단계
  - SDA Low 상태에서 SCL을 Low로 내려
  - “START 후 첫 비트 전송 준비” 상태를 만든다.
- DATA1~4**
  - 보낼 비트를 SDA에 올리고, SCL High에서 슬레이브가 샘플링하도록 하고, 안정 시간 유지 후 SCL Low로 내려 다음 비트를 준비하는 과정
  - SCL High에서 ACK 판별
- HOLD**
  - 다음 상태 제어 (DATA1, STOP, START1)
- STOP**



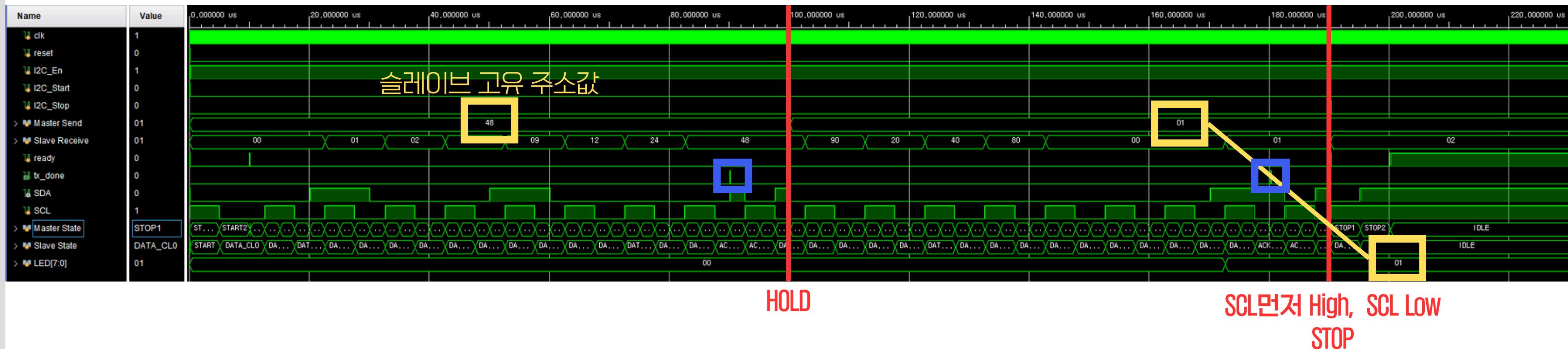
# I2C 설계

## Slave FSM (write)

- IDLE
  - SDA=1, SCL=1 상태에서 Start 조건(SDA falling while SCL=1)을 기다리는 대기 상태
- START
  - SCL falling edge를 기다려 첫 번째 데이터 비트를 받을 준비
- DATA\_CL0~1
  - SCL rising edge에서 SDA 값을 읽고 shift 레지스터로 시프트
  - 8비트(0~7)가 다 채워졌는지 bit\_counter로 체크
  - 8비트 수신 완료되면 ACK 단계로 이동
- ACK\_CL0~1
  - SCL rising edge에서 슬레이브가 SDA를 0으로 ACK 생성
  - 이후 다음 바이트 수신 준비 위해 DATA\_CL0 복귀하거나 SCL신호에 따라 IDLE로 이동

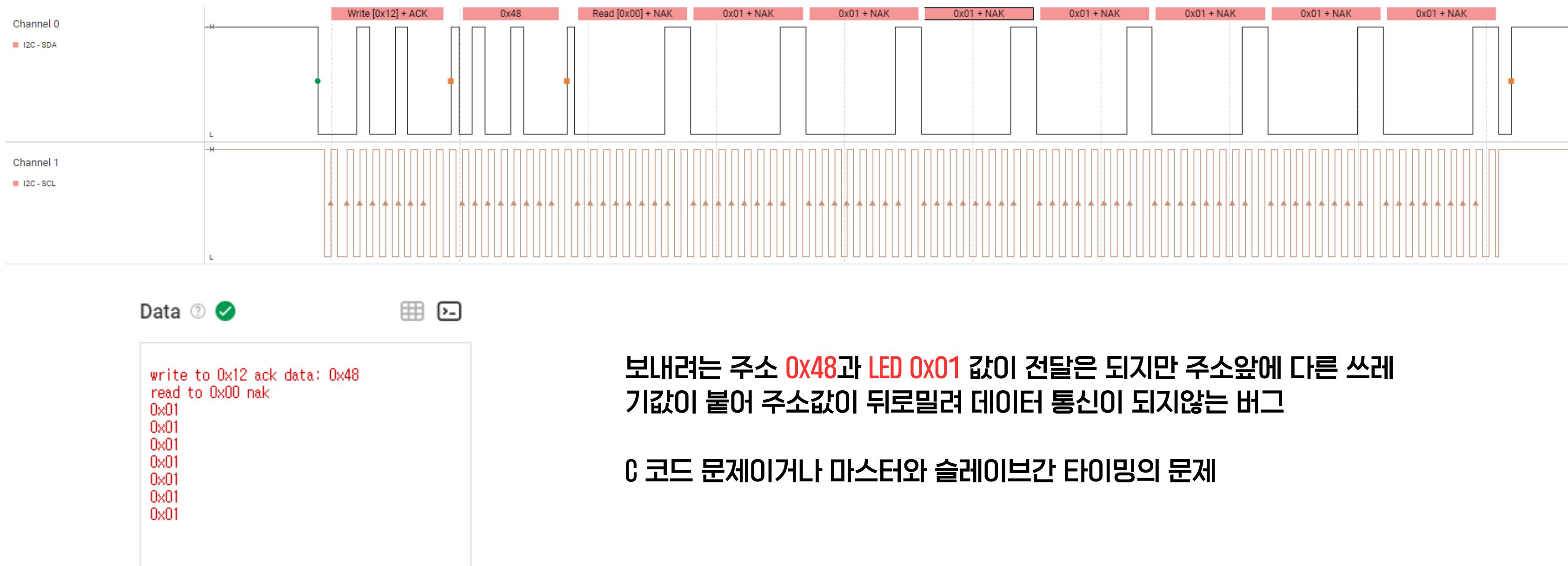


# I2C 설계 Simulation



# I2C 설계 C Application

- 각 스위치를 결 때 주소와 LED값 (0x01, 0x02...)를 보내 슬레이브 보드에서 전달 받고 LED ON



# 트러블 슈팅

문제 : SPI FND 구현에서 보드 하나로 동작을 하였을때는 잘 동작하는 반면 보드 두개 사용시 숫자가 카운트하다가 중간에 쓰레기 값이 자주 확인됨.

Master는 매우 빠른 속도로 2바이트 전송 Slave는 내부 clk 기준 done 펄스를 처리 두 이벤트가 같은 clk 사이클에서 겹치며 타이밍이 꼬인 것으로 추정

해결 : 슬레이브측에 싱크로나이저 회로를 사용하여 타이밍 문제를 안정화하여 데이터를 잘 수신받음

```
always_ff @(posedge clk, posedge reset) begin
    if (reset) begin
        sclk_s0 <= 0; sclk_s1 <= 0;
        ss_s0   <= 1; ss_s1   <= 1;
        mosi_s0 <= 0; mosi_s1 <= 0;
    end else begin
        sclk_s0 <= sclk;
        sclk_s1 <= sclk_s0;

        ss_s0   <= ss_n;
        ss_s1   <= ss_s0;

        mosi_s0 <= mosi;
        mosi_s1 <= mosi_s0;
    end
end

wire sclk_clean = sclk_s1;
wire ss_clean   = ss_s1;
wire mosi_clean = mosi_s1;
```

# 프로젝트 후 느낀점

- 이번 SPI-I2C 통신 설계 과정을 진행하면서 디지털 통신 프로토콜이 단순한 회로 연결로 끝나는 것이 아니라 정확한 타이밍 이해, FSM 설계 규칙 준수, 그리고 실제 환경의 요소까지 고려해야 한다는 사실을 깊이 체감할 수 있었습니다. 특히 SPI와 달리 I2C는 START/STOP 조건이 명확하면서도 실제 구현에서는, 동기 타이밍, 마스터 FSM과 펌웨어동작 차이 등이 겹치면서 이론대로만 동작하지 않는다는 점을 분명히 느꼈습니다.
- 아쉬운 점으로는 Write 안정화도 어렵게 진행되어서 Read사이클까지 완성하지 못 한 점과 마스터 FSM과 C코드의 타이밍차이를 완전히 제어하지 못했고 시간 부족으로 UVM 검증을 진행하지 못한 점이 아쉬웠습니다.
- 이후 남은 시간에 동작구현과 검증을 할 계획입니다.

# 감사합니다

발표자 : 안유한