SoC 를 위한 *Peripheral* 설계

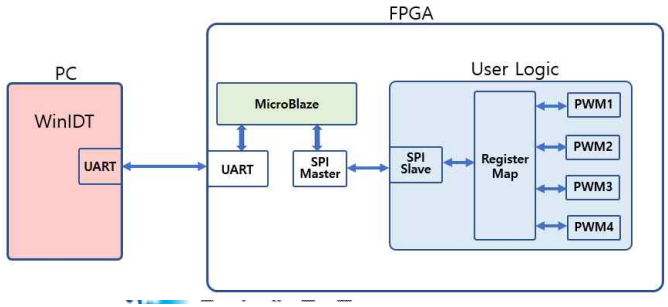
[*Serial Bus* ➔ *SPI* 구현]

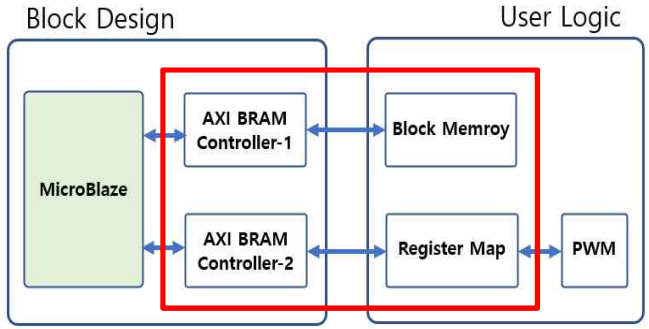
[*Reference*]

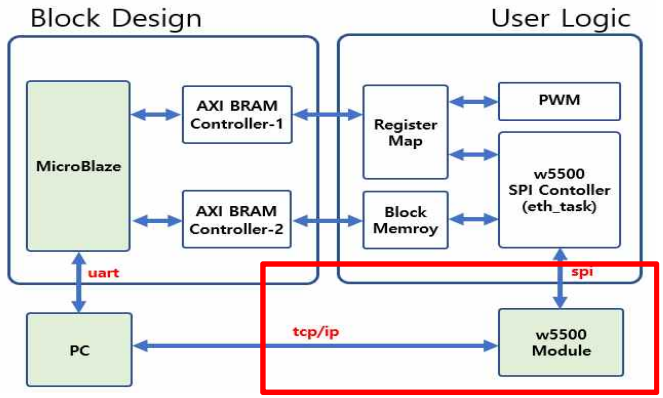
- *https://ko.wikipedia.org/wiki/%EC%A7%81%EB%A0%AC_%ED%86%B5%EC%8B%A0*
- *https://ko.wikipedia.org/wiki/UART*
- *MicroBlaze.v15 [IHIL]*
- *https://hanbulkr.tistory.com/5*
- *https://electriceng.tistory.com/422*

2024-06-13

Table of Contents



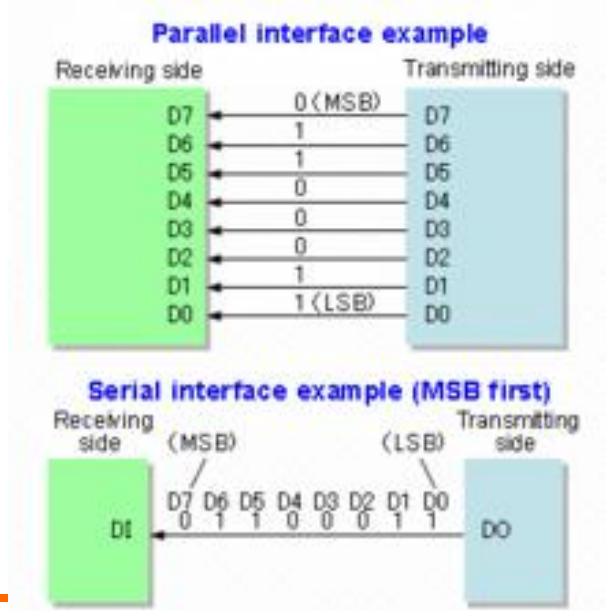




Serial Bus

➢ *Serial Bus*(직렬 통신)

- **직렬 통신(*Serial Bus*, 시리얼 버스)**은 연속적으로 통신 채널이나 컴퓨터 버스를 거쳐 **한 번에 하나의 비트 단위로 데이터를 전송**하는 과정
- 여러 개의 병렬 채널을 갖춘 링크 위에서 동시에 여러 개의 비트를 보내는 병렬 통신과 대조적
- 컴퓨터에서 데이터 처리가 병렬로 되는데, 통신을 위해 병렬 통신을 하려면 여러 개의 채널이 필요
- 거리와 비용을 고려하면 채널이 많을 경우 병렬 통신은 문제가 될 수 있다. 결국 병렬로 처리되는 데이터를 통신할 때 **시간으로 나누어 차례대로 전송(*Serial Bus*)함으로써 문제를 해결**
- **직렬 통신**에서 데이터가 계속되어 전송되면, **각 비트를 구별할 방법이 필요**
- *Digital Circuit*에서 수신된 데이터의 비트가 시간적으로 어디서부터 시작이고 끝인지를 알 필요
- **데이터 비트를 복구**하기 위해 데이터의 시간적 위치를 알리기 위해 동기신호를 보내는 경우와 동기 신호 없이 신호 자체에서 데이터 비트를 복원하는 방식으로 구분
  - 동기 방식 : 데이터 신호와는 별도로 동기신호를 함께 전송
  - 비동기 방식 : 데이터 신호만을 보내고 각각의 방식에 따라 데이터 비트를 찾아냄
    - 집적회로(IC)들은 여러 개의 핀을 갖추고 있을 수록 더 비싸서, 수많은 IC들은 핀들의 수를 줄이기 위해 속도가 중요하지 않을 시점에 직렬 버스를 사용하여 데이터를 전송
    - 값싼 직렬 버스의 예 : *UART, SPI, I²C* 등…



Serial Bus ➔ 종류(1) : SPI

- *Serial Bus :* ***SPI(Serial Peripheral Interface Bus)***
  - **동기식(*Synchronous*) 시리얼 통신 방식**. 데이터 수신의 타이밍을 위하여 ***Clock*** **라인을 사용**.
  - 직렬 주변기기 인터페이스 버스(*Serial Peripheral Interface Bus*) 또는 SPI 버스는 아키텍처 전이중 (Full Duplex) 통신 모드로 동작하는 모토로라 아키텍처에 이름을 딴 동기화 직렬 데이터 연결 표준
  - 아래와 같은 **4가지 선으로 구성**
    - ✓ ***MOSI*** : 마스터 출력, 슬레이브 입력 (마스터로부터의 출력). Master Out, Slave In
    - ✓ ***MISO*** : 마스터 입력, 슬레이브 출력 (슬레이브로부터의 출력). Master In, Slave Out
    - ✓ ***SCK*** : 직렬 클럭 (마스터로부터의 출력). 데이터 전송 타이밍 동기화를 위한 Clock
    - ✓ ***SS(CS)*** : 슬레이브 셀렉트 (***active low***, 마스터로부터의 출력). ***Slave Select***(또는 ***Chip Select***)는 데이터 수신할 기기 선택을 위한 신호로 사용
  - 마스터가 SS를 통해 신호를 전송할 슬레이브를 선택
  - 마스터는 *MOSI* 를 통해서 ***SCLK***에 동기화된 신호를 전송
  - *I2C*가 *2*개의 선이 필요한 것과 달리 *SPI*는 선을 추가로 더 필요

Serial Bus ➔ 종류(2) : UART

➢ *Serial Bus : **UART**(Universal Asynchronous serial Receiver and Transmitter)*

- 비동기식(*Asynchronous*) 통신. 데이터 수신의 타이밍을 위하여 **Clock 라인을 사용하지 않음**.
- 비동기식 통신이기 때문에 보내는 데이터를 어떻게 해석할지가 중요. 데이터 해석을 위해서는 아래와 같은 정보가 필요. 송수신 측에서는 아래와 같은 규칙들을 지정하고 데이터를 해석.
  - ✓ ***Baud Rate(보레이트)*** : 초당 많은 심볼(Symbol, 의미 있는 데이터 묶음)을 얼마만큼 전송할지에 대한 정보
  - ✓ **데이터 길이** : 송수신 데이터 길이(8bit or 7bit)
  - ✓ **패리티 비트** : 패리티 비트 사용 않는 경우, Even(짝수) 패리티 사용, Odd(홀수) 패리티 사용
  - ✓ **정지 비트** : 정지 비트(1 or 2개)

비트 수	1	2	3	4	5	6	7	8	9	10	11
	시작 비트 (Start bit)	5–8 데이터 비트								패리티 비트 (parity bit)	종료 비트 (Stop bit(s))
	Start	Data 0	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7	Parity	Stop

- 시작 비트 : 통신의 시작을 의미하며 한 비트 시간 길이 만큼 유지한다. 지금부터 정해진 약속에 따라 통신을 시작
- 데이터 비트 : 5~8비트의 데이터 전송을 한다. 몇 비트를 사용할 것인지는 해당 레지스터 설정에 따라 결정
- 패리티 비트 : 오류 검증을 하기 위한 패리티 값을 생성하여 송신하고 수신쪽에 오류 판단한다. 사용 안함, 짝수, 홀수 패리티 등의 세가지 옵션으로 해당 레지스터 설정에 따라 선택할 수 있다. '사용 안함'을 선택하면 이 비트가 제거
- 끝 비트 : 통신 종료를 알린다. 세가지의 정해진 비트 만큼 유지해야 한다. 1, 1.5, 2비트로 해당 레지스터 설정에 따라 결정

Serial Bus ➔ 종류(3) : I²C

- *Serial Bus : I²C*
  - 동기식(Synchronous) 시리얼 통신 방식. 데이터 수신의 타이밍을 위하여 Clock 라인을 사용.
  - 필립스에서 개발한 직렬 버스로 마더보드, 임베디드 시스템, 휴대 전화 등에 저속의 주변 기기를 연결하기 위해 사용
  - I²C 는 풀업 저항이 연결된 직렬 데이터(SDA)와 직렬 클럭(SCL)이라는 두 개의 양 방향 오픈 컬렉터 라인을 사용
  - 최대 전압은 +5 V이며, 일반적으로 +3.3 V 시스템이 사용되지만 다른 전압도 가능
  - 마스터(Master)와 슬레이브(Slave)가 존재
  - 아래와 같은 2가지 선으로 구성
    - ✓ SDA : Data 송수신
    - ✓ SCK : Clock 전송
  - 슬레이브마다 지정된 주소 값을 가지고 데이터를 주고 받는데, 데이터를 주고 받을 때 반드시 주소 값을 붙여서 전송
  - SPI가 여러 개의 선이 필요한 것과 달리 2개의 선만 가지고 통신이 가능

SPI(Serial Peripheral Interface Bus)

Serial Bus ➔ SPI(1)

➢ *Serial Bus : SPI(Serial Peripheral Interface Bus) (1)*

- SPI통신은 I2C통신과 같은 통신방법의 한 종류이지만 통신하는 방법이 다름
  - I2C통신은 한사람이 보낼때는 다른사람이 받고만 있어야하는 무전기와 같은 통신방식
  - SPI 통신은 한사람이 데이터를 보내면서 데이터를 받을 수 있는 전화같은 방식
- 어느 통신방법을 쓸 것인가에 대하여는 사용자들의 몫이기 때문에 장단점등을 잘 확인하여 사용

❖ SPI 통신의 장점

1. 완전한 전이중(Full duplex) 통신 : **양방향 통신**이다. 위에서 설명한 것처럼 말하면서 들을 수 있음
2. 전송되는 비트에 대한 완전한 프로토콜 유연성 : 최대 16비트까지 맘대로 길이를 조절 가능
3. 전송기가 불필요 : 흔히 말하는 트랜시버를 사용할 필요가 없음
4. 매우 **단순한 하드웨어 인터페이스** 처리 : 아주 **단순한 센서나 메모리에서 많이 사용**
5. IC 패키지에 4개의 핀만 사용
6. 최대 클럭이 제한되지 않아 속도 제한이 없음
7. Push-Pull 출력(Open Drain이 아닌)을 사용하여 상호간에 같은 전압을 사용하여 시그널 정합성과 고속 지원
8. I2C 보다 낮은 소비 전력
9. 슬레이브는 마스터가 보내주는 클럭만을 사용하고 정확성이 떨어져도 문제 없음

Serial Bus ➔ SPI(2)

➢ *Serial Bus : SPI(Serial Peripheral Interface Bus) (1)*

- PI통신은 I2C통신과 같은 통신방법의 한 종류이지만 통신하는 방법이 다름
  - I2C통신은 한사람이 보낼때는 다른사람이 받고만 있어야하는 무전기와 같은 통신방식
  - SPI 통신은 한사람이 데이터를 보내면서 데이터를 받을 수 있는 전화같은 방식
- 어느 통신방법을 쓸 것인가에 대하여는 사용자들의 몫이기 때문에 장단점등을 잘 확인하여 사용

❖ SPI 통신의 단점

1. 하드웨어 슬레이브 인식이 없음
2. 슬레이브에 의한 하드에워 흐름제어가 없음
3. 오류 검사 프로토콜이 정의되어 있지 않음 (에러 체킹 지원)
4. 노이즈 스파이크에 영향을 받는 경향이 있음
5. RS-232, CAN 버스보다 비교적 더 짧은 거리에서 동작 (칩간 통신에서만 주로 사용)
6. 하나의 마스터 장치만 지원
7. 인밴드(디폴트 SPI wire)를 통해 주소가 지원되지 않아 다수의 슬레이브를 사용 시 별도의 아웃밴드(칩 셀렉트 라인)를 통해 슬레이브를 선택
8. Hot 플러그를 지원하지 않음

Serial Bus ➔ SPI(3)

- *SPI(Serial Peripheral Interface Bus) ➔ Applications*

  (1) EEPROM, 플래시 메모리 (2) MMC or SD Card (3) LCD (4) RTC(Real Time Clock)

  (5) 각종 센서류

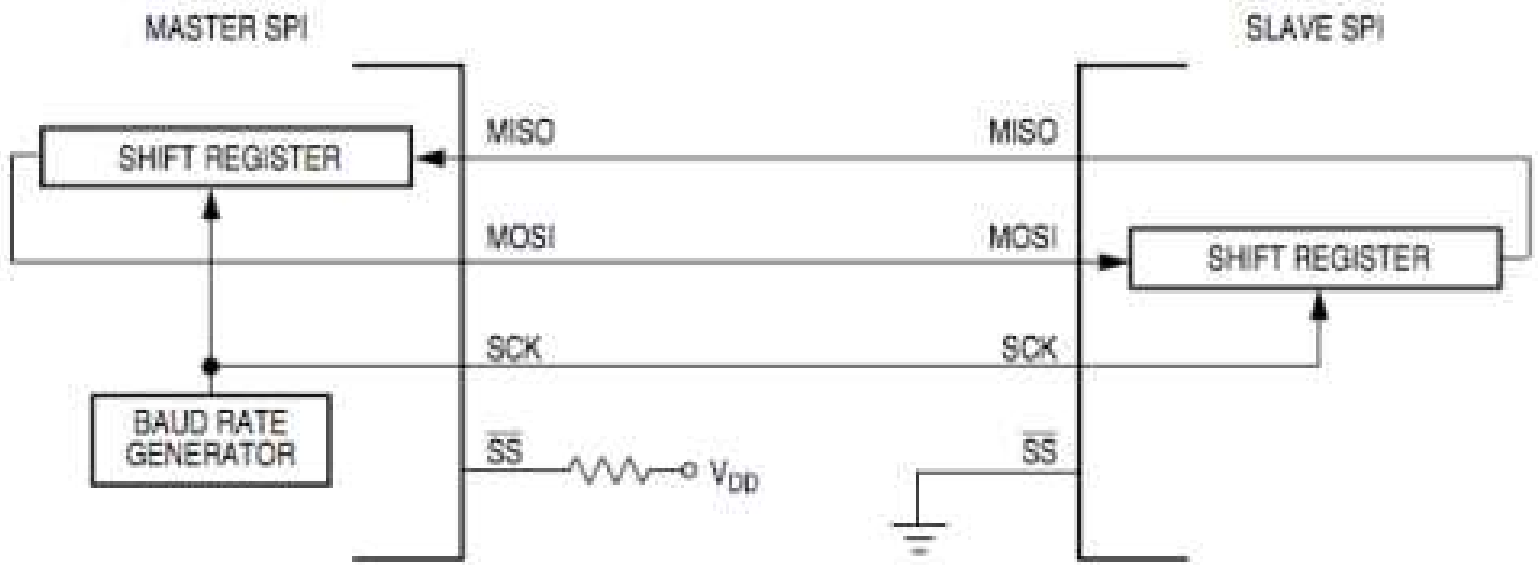
- *SPI(Serial Peripheral Interface Bus) ➔ SPI Pin Description*

1. I2C통신에서는 데이터를 전송하는 SDA(Serial Data)선과 클럭을 전송하는 SCL(Serial Clock)선 두개만 필요
2. SPI통신에서는 송신과 수신을 위한 MOSI(Master Out Slave IN)선과 MISO(Master In Slave Out)선, 클럭(SCLK)선, SS(Slave Select)선 이렇게 4개가 필요
3. ***MOSI(Master Out Slave IN)*** ➔ 이름 그대로 마스터에서 나와서 슬레이브에게 가는 데이터를 전송하는 선
4. ***MISO(Master In Slave Out)*** ➔ 슬레이브에게 나와서 마스터에게 가는 데이터를 전송하는 선 (I2C통신이 전이중통신이 안됐다면 SPI는 선이 구분되어 있기 때문에 송수신이 동시에 가능.다만 SPI통신은 Master-Slave 관계(주인-종)처럼 Slave는 Master에게 통신은 마음대로 하지 못함.)
5. SPI의 데이터 송수신은 동시에 이루어지기 때문에 데이터 송신이 곧 수신. 이 데이터 송수신의 타이밍을 결정짓는 것은 Master장치
6. ***SCLK***, CLK(Clock) 은 동기화 신호를 위한 Clock
7. ***SS***(Slave Select) ,CS(Chip Select)선 ➔ Master장치가 여러개의 Slave 장치와 통신할 때 하나를 선택하여 통신을 하는데 그 때 선택을 하는 역할의 신호선, 한마디로 어디로 보낼지 결정하는 신호선

Serial Bus ➔ SPI(4)

➢ *SPI(Serial Peripheral Interface Bus)* ➔ *SPI* 통신의 동작 구조(1)

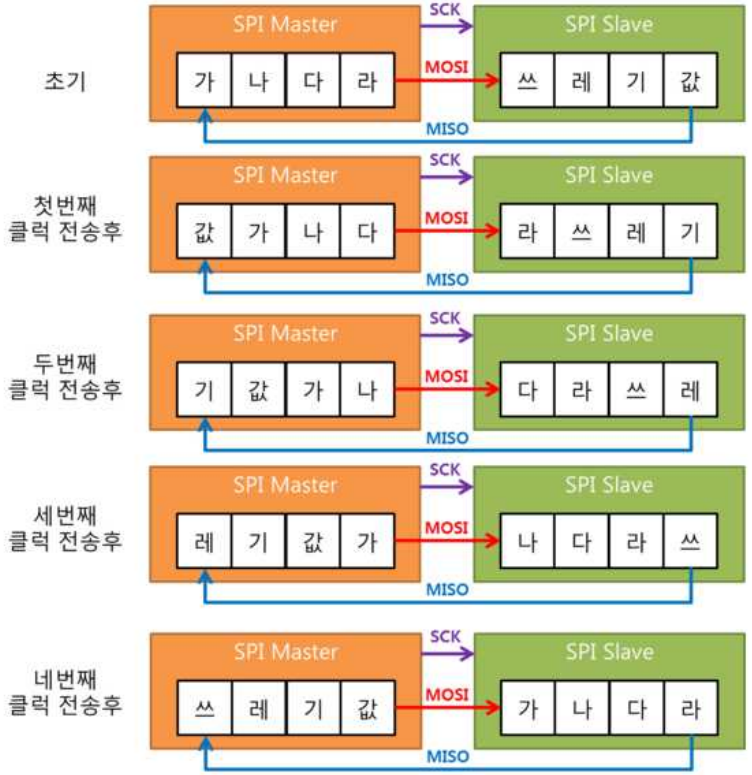
1. SPI 장치들은 Shift Register를 가지고 있음
2. 기본적으로 MSB부터 전송되는데 특정 컨트롤러는 LSB부터 전송을 수행시키는 방법도 지원
3. 그림처럼 SHIFT REGISTER에 의해서 데이터가 이동
4. 마스터에서 어떤 값을 슬레이브로 보낸다고 가정했을 때, 보내는 1Clock의 신호 마다 1bit의 data가 이동
5. 밀어내기 식으로 data가 들어간다고 생각하면 됨

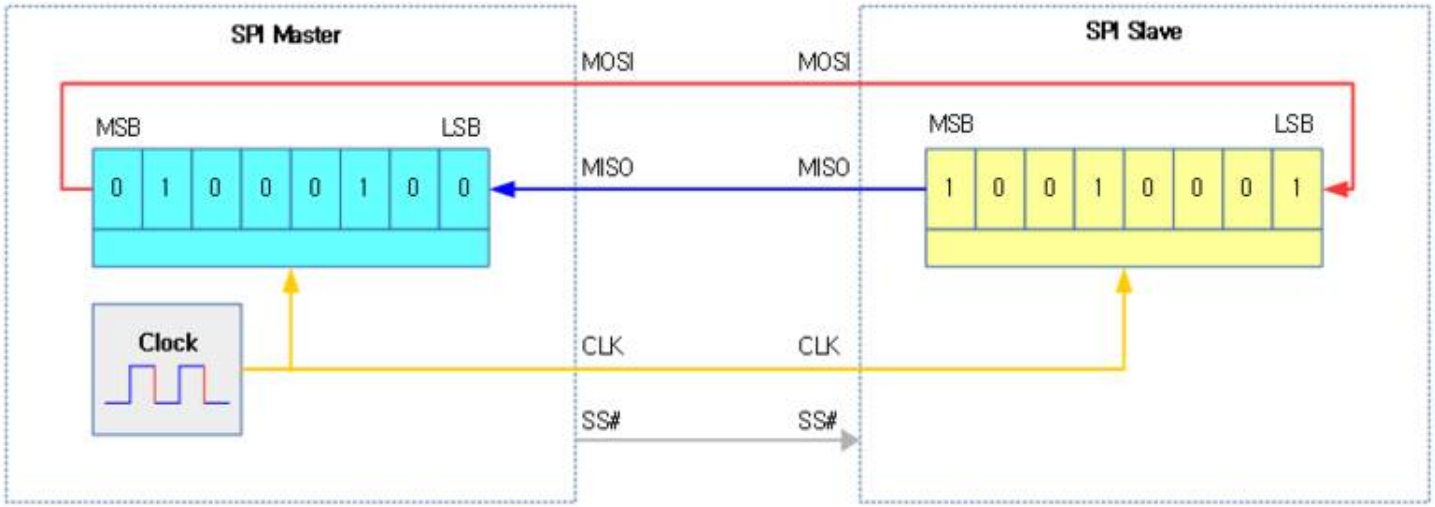


Serial Bus ➔ SPI(5)

➢ *SPI(Serial Peripheral Interface Bus)* ➔ *SPI* 통신의 동작 구조(2)

1. SPI 장치들은 Shift Register를 가지고 있음
2. 기본적으로 MSB부터 전송되는데 특정 컨트롤러는 LSB 부터 전송을 수행시키는 방법도 지원
3. 그림처럼 SHIFT REGISTER에 의해서 데이터가 이동
4. 마스터에서 어떤 값을 슬레이브로 보낸다고 가정했을 때, 보내는 1Clock의 신호 마다 1bit의 data가 이동
5. 밀어내기 식으로 data가 들어간다고 생각하면 됨





Serial Bus ➔ SPI(6)

➢ *SPI(Serial Peripheral Interface Bus)* ➔ *SPI* 전송모드

❖ ***3 wire SPI***

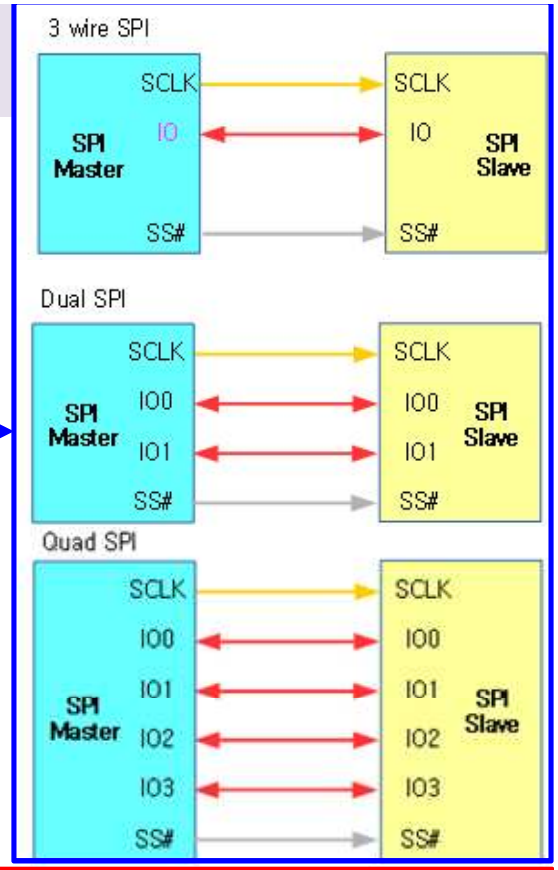
- Half Duplex 입출력 wire를 1개 지원
- 저속 EEPROM 및 센서류에서 사용

❖ ***Dual SPI***

- Half Duplex 입출력 wire를 2개 지원한다. (단방향 속도가 2배)

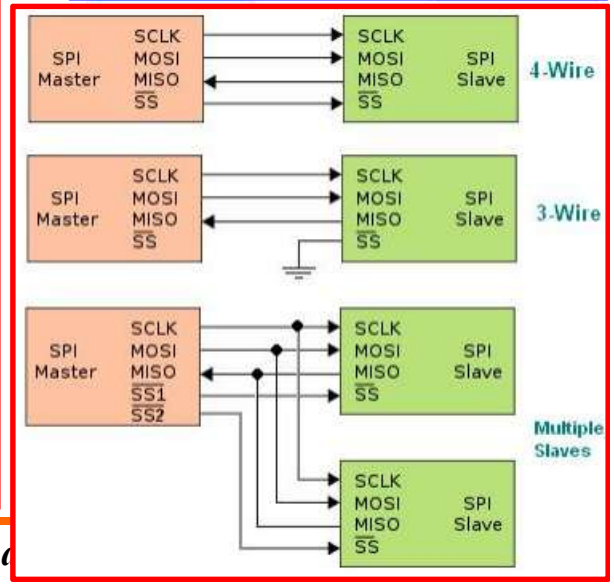
❖ ***Quad SPI***

- Half Duplex 입출력 wire를 4개 지원한다. (단방향 속도가 4배)
- SPI-Nor 플래시 및 SPI-Nand 플래시에 자주 사용



❖ 다른 예시 ***3-Wire, 4-Wire, Multiple Slaves***

- 1번이 4-Wire 방식, 2번이 3-Wire 방식, 3번이 다중 슬레이브 연결 시의 결선도
- 1번의 경우, 일반적으로 연결하는 방식이며, SS를 통해 data 전송 알림 신호를 보내 data를 전달하는 방식.
- 2번의 경우, Slave장치 에서 SS를 GND에 묶어두면 Slave장치는 항상 data를 받을 수 있는 상태가
- 3번의 경우, Slave를 여러개 연결할 경우 SS1, SS2를 통해 data 값을 전달하는 방식

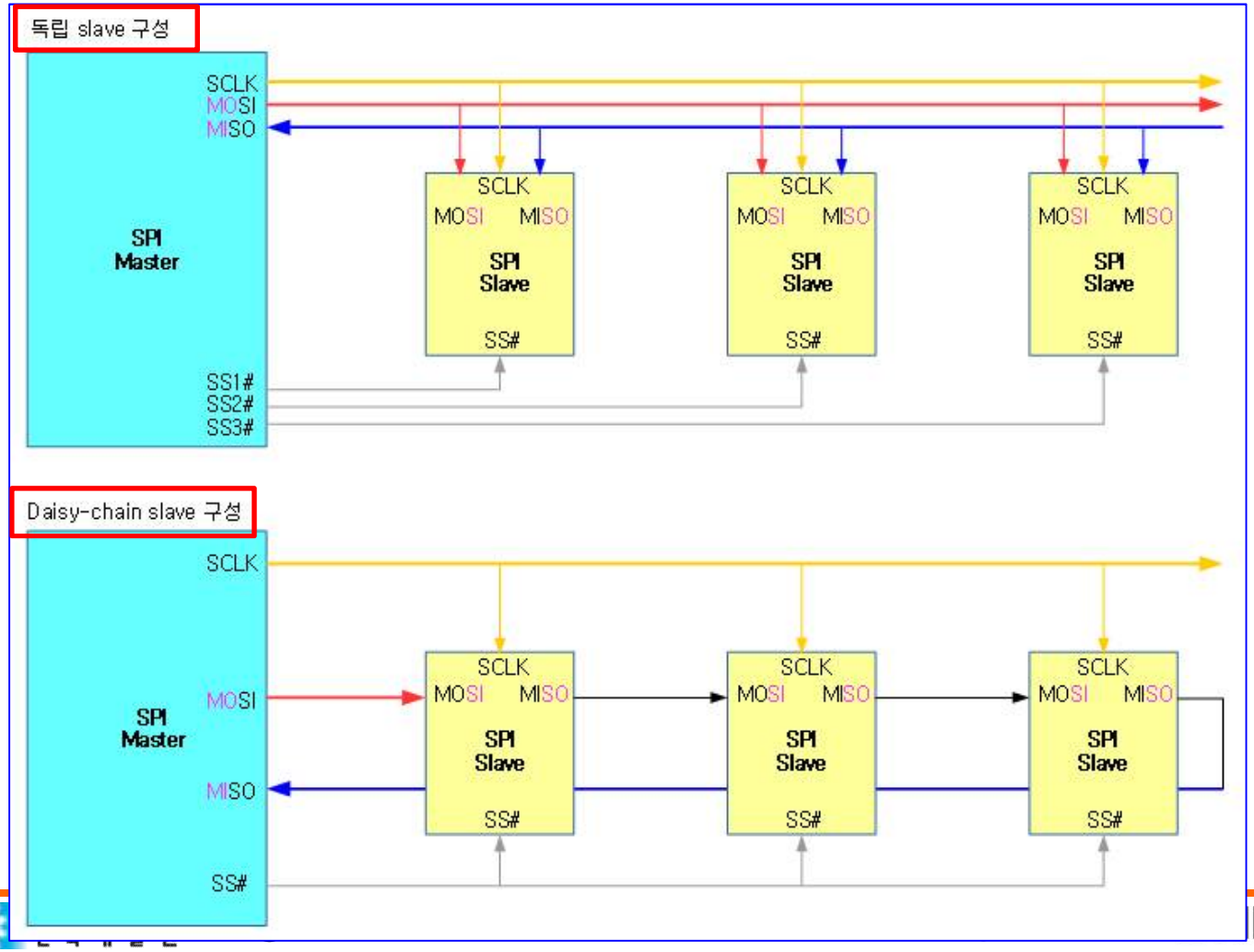


Serial Bus ➔ SPI(7)

➢ *SPI(Serial Peripheral Interface Bus)* ➔ *SPI* 연결 구성 방식

❖ SPI 버스를 구성하는 방법으로 다음과 같이 두 가지 방식을 사용

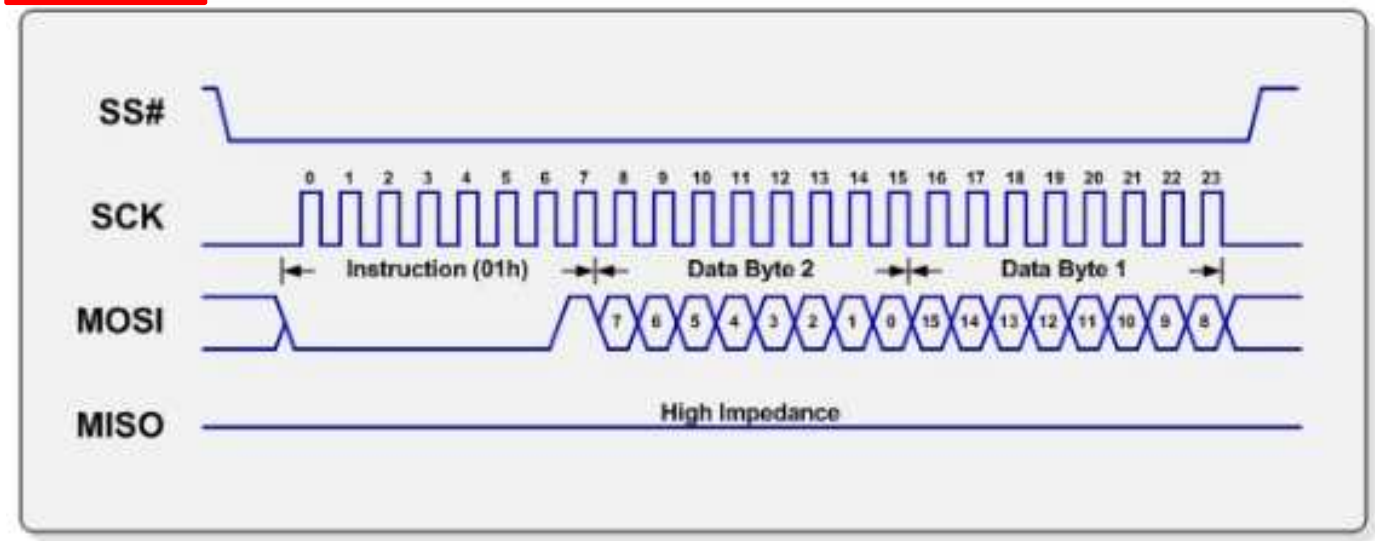
- **독립 slave** 구성 방식 ➔ SPI 디바이스를 선택하여 사용하는 방식
- ***Daisy-chain slave*** 구성 방식 ➔ 데이터 비트를 서로 밀어내는 방식



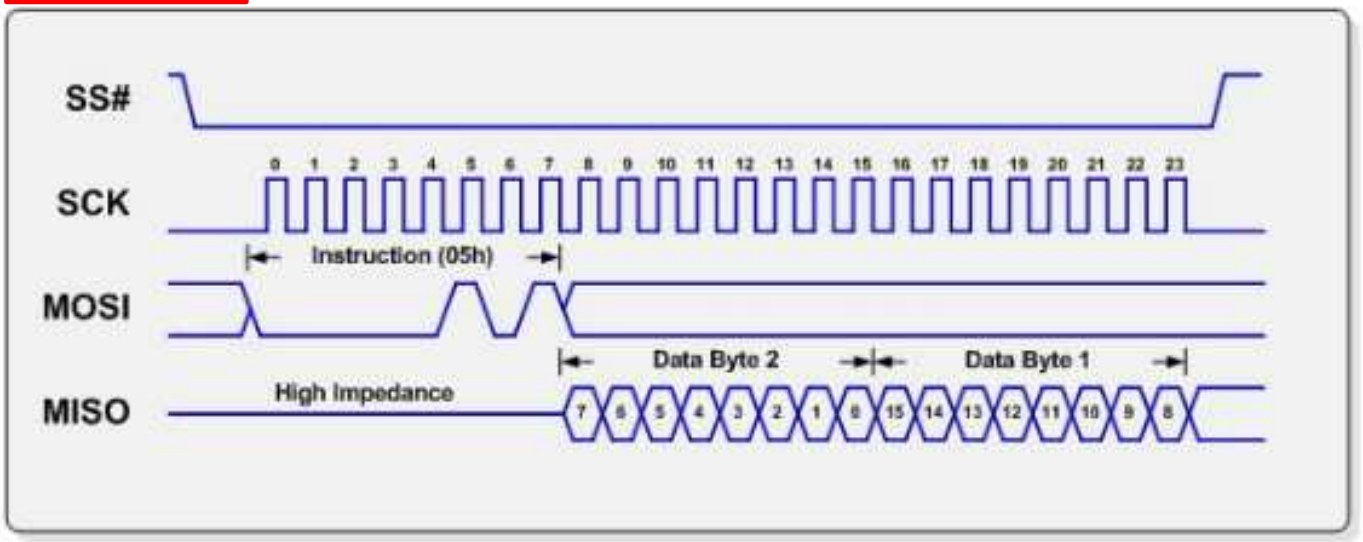
Serial Bus ➔ SPI(8)

➢ *SPI(Serial Peripheral Interface Bus)* ➔ *SPI* : 간단한 SPI의 Write 및 Read 동작(1)

Write



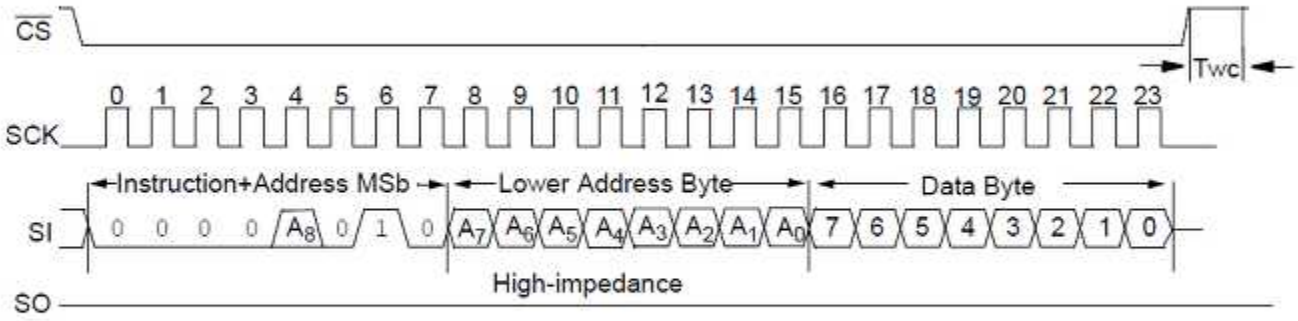
Read



Serial Bus ➔ SPI(9)

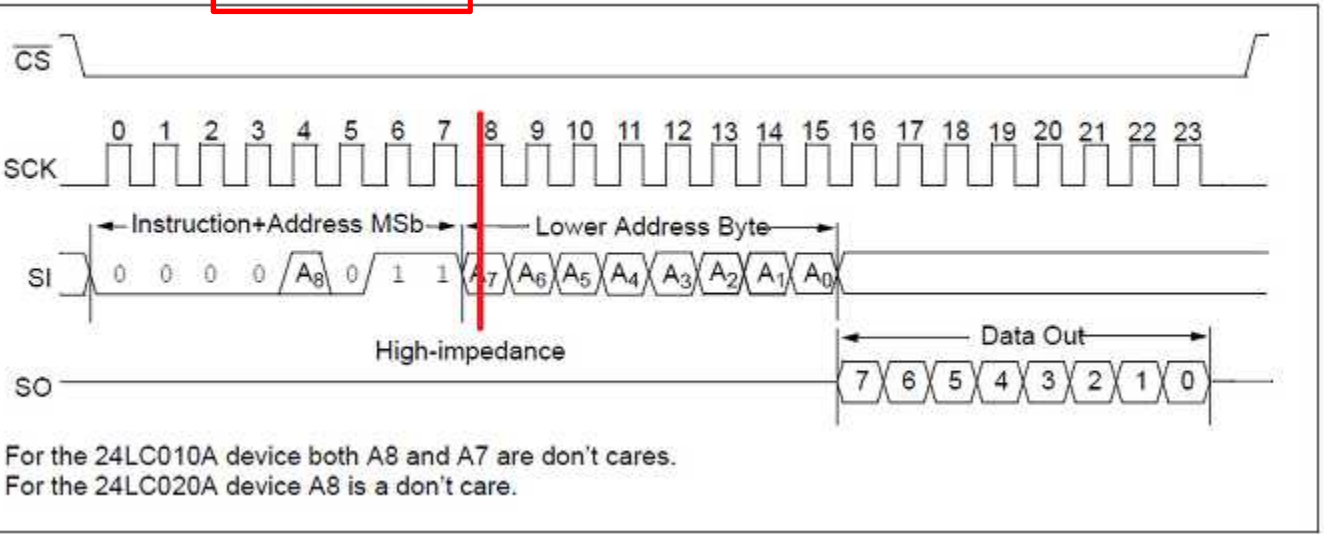
➢ *SPI(Serial Peripheral Interface Bus)* ➔ *SPI* : 간단한 SPI의 Write 및 Read 동작(2)

FIGURE 3-2: BYTE WRITE SEQUENCE



For the 24LC010A device both A8 and A7 are don't cares.
For the 24LC020A device A8 is a don't care.

FIGURE 3-1: READ SEQUENCE



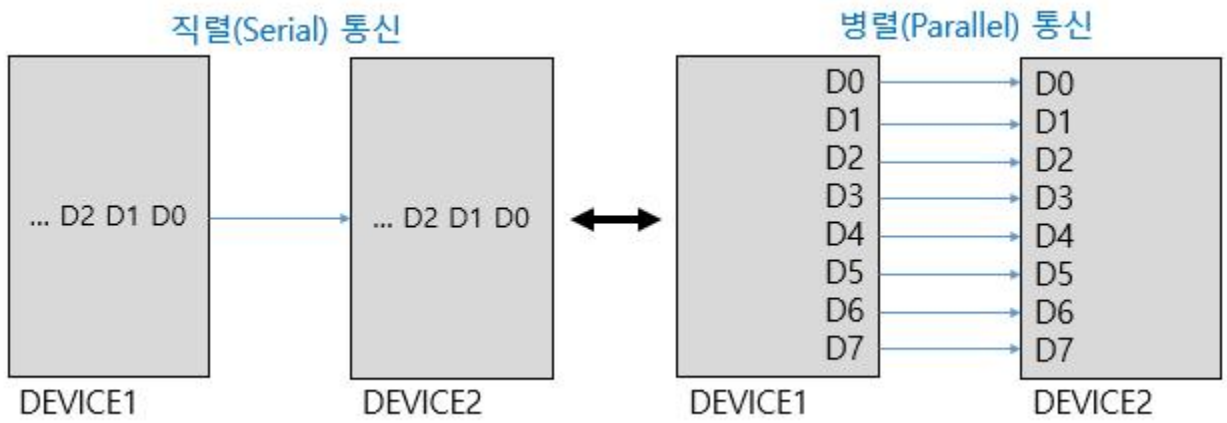
For the 24LC010A device both A8 and A7 are don't cares.
For the 24LC020A device A8 is a don't care.

UART(Universal Asynchronous serial Receiver and Transmitter)

Serial Bus ➔ UART(1)

➢ *Serial Bus : UART(Universal Asynchrounous serial Receiver and Transmitter) (1)*

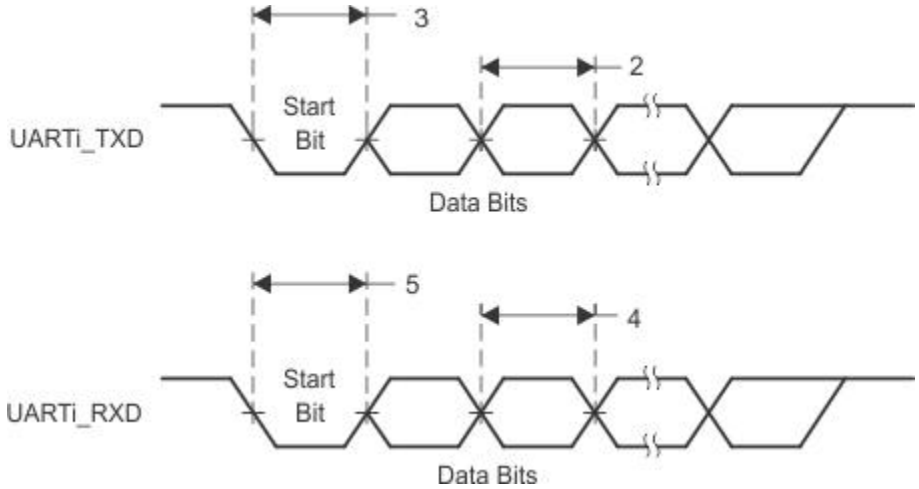
- 직렬 통신 : 1개의 입출력 핀을 통해 8개 비트를 한 번에 전송하는 방법
- 병렬(Parallel) 통신 : n비트의 데이터를 전송하기 위해 n개의 입출력 핀을 사용하는 방법



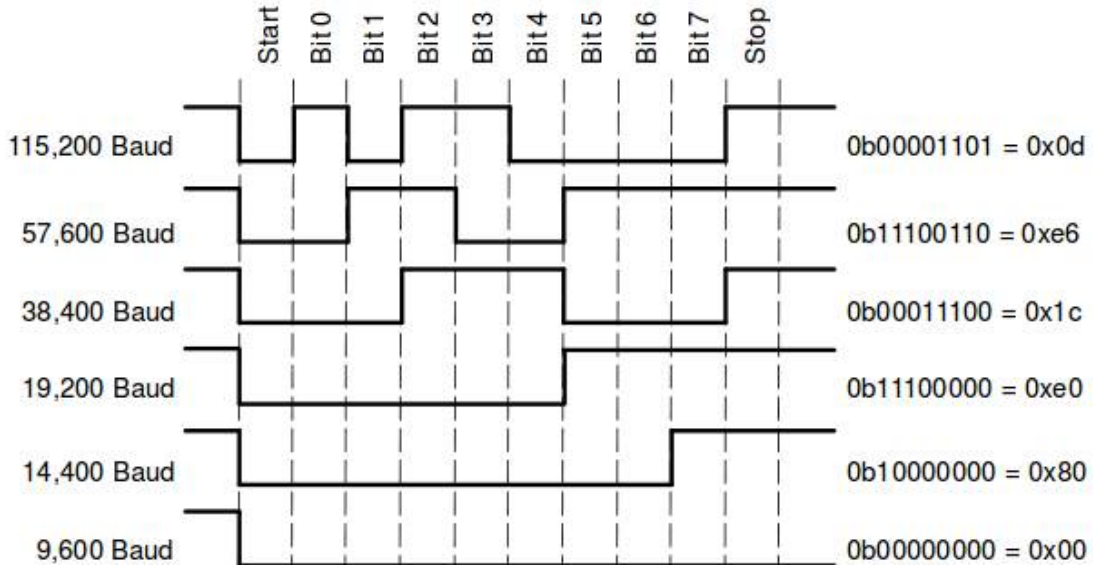
- MCU 등에서는 직렬 통신 방식이며, 그 중 많이 쓰이는 방법 중 하나가 UART
- 시리어 통신을 사용하기 위해서는 보내는 쪽(Tx)와 받는 쪽(Rx)에서 약속을 정해야하는데, 이를 프로토콜(Protocol)이라고 함
- 디지털 회로에서 0과 1의 값만 처리할 수 있으므로 0은 GND, 1은 VCC로 데이터 전송하고, 받는 쪽에서는 GND와 VCC를 0과 1의 이진 값으로 변환하여 사용
- 보내는 쪽(Tx)와 받는 쪽(Rx)이 원할하게 데이터를 처리하려면, 데이터를 보내는 속도에 대하여 약속(프로토콜, Protocol)이 정해져 있어야 함

Serial Bus ➔ UART(2)

➢ *Serial Bus : UART(Universal Asynchrounous serial Receiver and Transmitter) (2)*



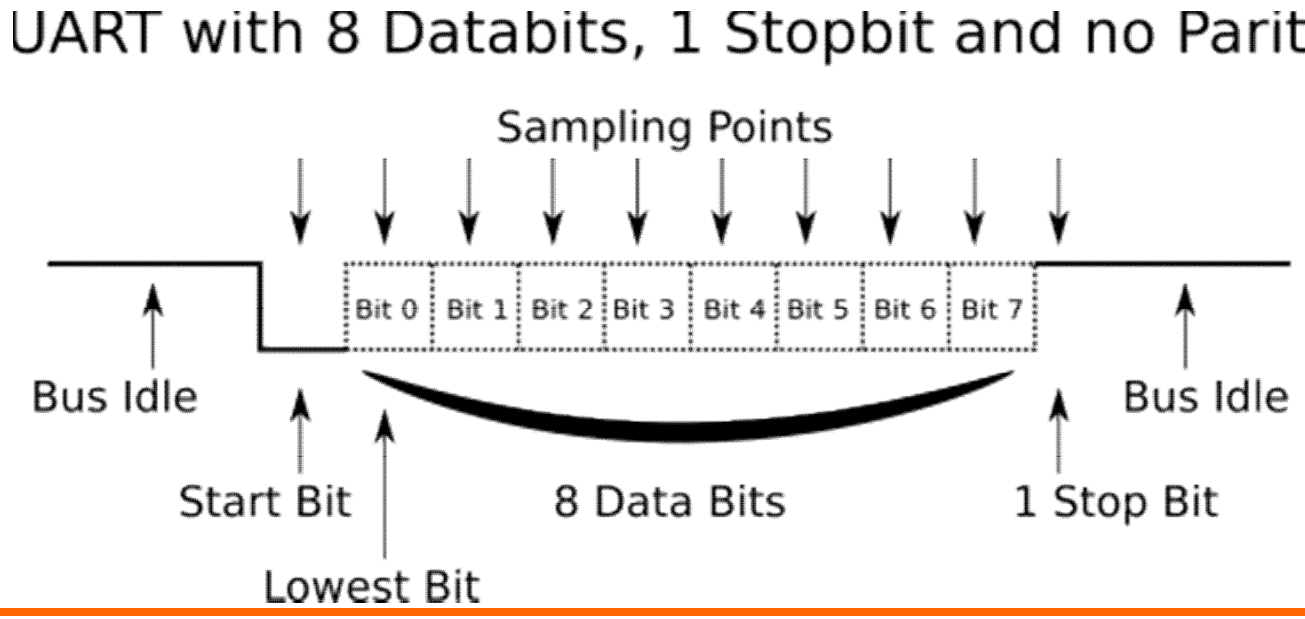
- UART에서는 보내는 쪽(Tx)와 받는 쪽(Rx) 에서 데이터를 보내는 속도를 보레이트(Baud Rate)로 정함



Serial Bus ➔ UART(3)

➢ *Serial Bus : UART(Universal Asynchrounous serial Receiver and Transmitter)(3)*

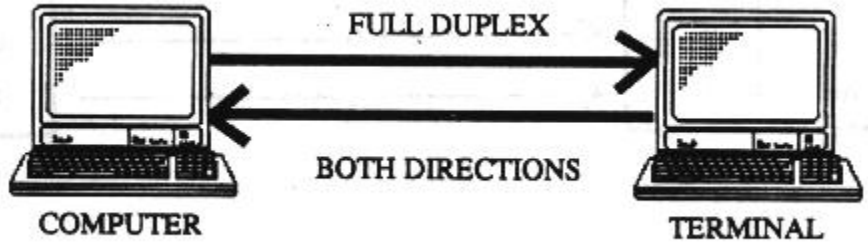
- Tx 와 Rx 가 동일한 속도로 데이터를 주고 받는다고 하여 정확하게 통신이 되지는 않음
- Tx는 항상 데이터를 보내는 것이 아니며, 필요한 경우에만 데이터를 전송하므로, Rx 는 Tx가 언제 데이터를 보내는지와 Tx에서 보내는 데이터의 시작이 어디서부터인지 알아낼수 있는 방법이 필요
  ➔ '0'의 시작비트(Start Bit)와 '1'의 정지 비트(Stop Bit)를 사용
- UART는 바이트(1byte=8bit) 단위 통신을 주로 사용, 시작비트와 정지 비트가 추가되어 10비트의 데이터를 전송(패리티 비트를 사용하지 않는 경우)

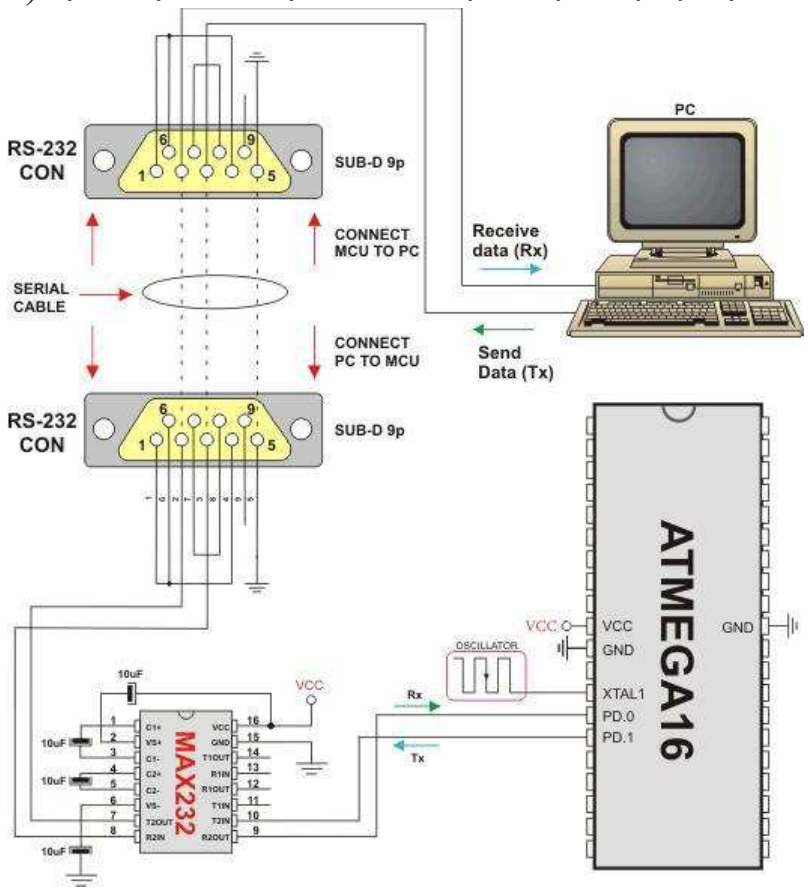


Serial Bus ➔ UART(4)

➢ *Serial Bus : UART(Universal Asynchrounous serial Receiver and Transmitter)(3)*

- UART 통신은 전이중 방식(Full Duplex) 통신으로 송신과 수신을 동시에 진행할 수 있으며, 이를 위해서 2개의 범용 입출력 핀이 필요(시리얼포트 ➔ Tx, Rx)
- 컴퓨터와 ATMEGA 16을 연결하는 경우, 컴퓨터와 연결에 있어서 RS232 연결을 사용하는데, RS232에서 사용하는 신호 레벨은 UART의 신호 레벨(TTL)과 달라 별도의 변환 장치를 사용하여 레벨을 변환시켜주어야 사용 가능
- RS232 통신에서 많이 쓰이는 IC는 MAX232





SoC 를 위한 *Peripheral* 설계

[*Serial Bus* ➔ *SPI* 구현]

[*Reference*]

- *https://ko.wikipedia.org/wiki/%EC%A7%81%EB%A0%AC_%ED%86%B5%EC%8B%A0*
- *https://ko.wikipedia.org/wiki/UART*
- *MicroBlaze.v15 [IHIL]*
- *https://hanbulkr.tistory.com/5*
- *https://electriceng.tistory.com/422*

SPI Master Implementation

SPI Master Implementation

➢ *spi_master_exam.xpr*

➢ ***SPI** (Serial Peripheral Interface Bus) **Master Implementation***

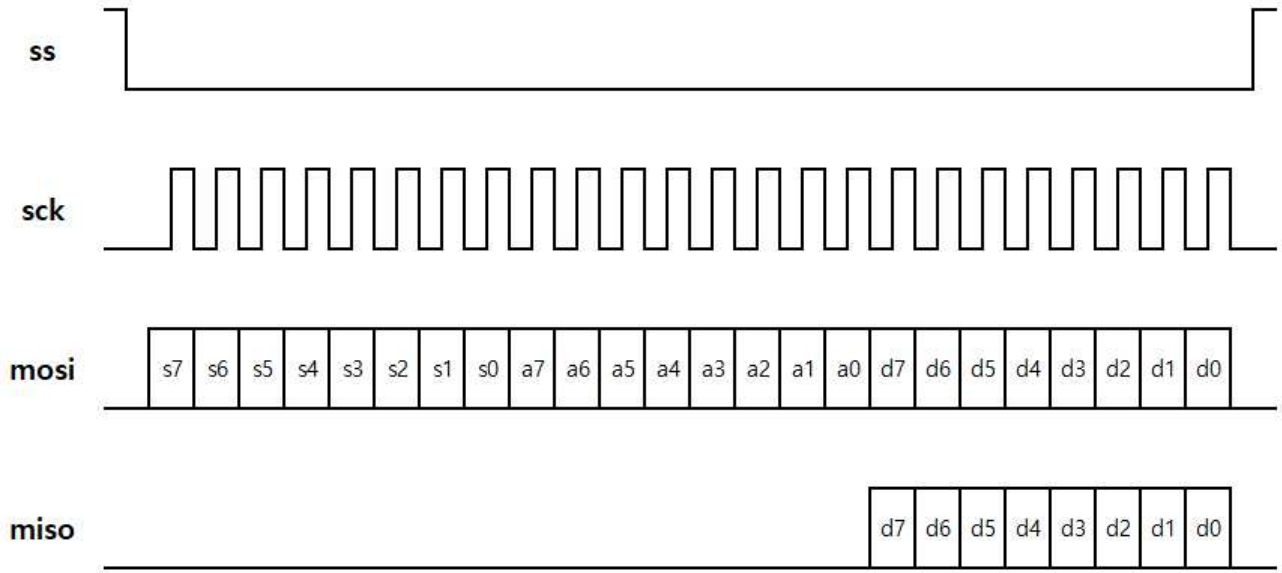
- SPI : Serial Parallel Interface ➔ Parallel 데이터를 Serial(1bit)로 변환하여 전송
- 1bit씩 전송하기 때문에 속도는 낮아지지만, 데이터 라인의 수를 줄일 수 있는 장점
- SPI는 저속의 데이터(주로 Control 용도의 데이터)를 전송하는데 사용

❖ ***Spec*** ➔ **SPI Master 를 구현하기 위하여 스펙**을 정의

- ***slave_id*** : ***0x64 (for write)***, ***0x65 (for read)***, (*write, read*는 *Master* 기준)
- ***Address*** : 8bits , ***Data*** : 8bits (연속해서 Access 하는 것은 지원하지 않음)

❖ ***SPI Timing*** ➔ SPI Master는 총 3 Byte 를 전송

- **Slave에 데이터를 저장(*for write*)**하기 위해서는 ***slave_id(0x64)***, ***address***, ***data***를 전송
- **Slave에서 데이터를 읽기(*for read*)** 위해서는 ***slave_id(0x65)***, ***address***를 전송➔ miso를 통하여 데이터를 read
- **mosi, miso는 sck에 동기** 되어 동작➔ **sck의 negative edge(or Low 구간)에서 데이터를 변경**하고, **sck의 positive edge (or High 구간)에서 데이터를 read**



SPI Master Implementation

➢ *spi_master_exam.xpr*

➢ *SPI Master Implementation* ➔ ***Code Implementation***

✓ ***Port*** 정의

- Module의 Input & Output Port를 정의

✓ ***State*** 정의

- State를 정의합니다.
- Verilog Code 를 그냥 구현하는 것보다 State Machine을 이용하여 구현

✓ ***Verilog Code Implementation***

- 정의된 스펙과 State를 참조해서 코드를 구현합니다.
- 각 State에서의 동작을 구현하고, State의 이동(상태 천이, State Transition)을 구현

✓ ***Simulation***

- 동작 검증을 위하여 Test bench를 구성하고 Simulation을 진행

SPI Master Implementation

➔ *Port 정의*

SPI Master Implementation

➢ *spi_master_exam.xpr*

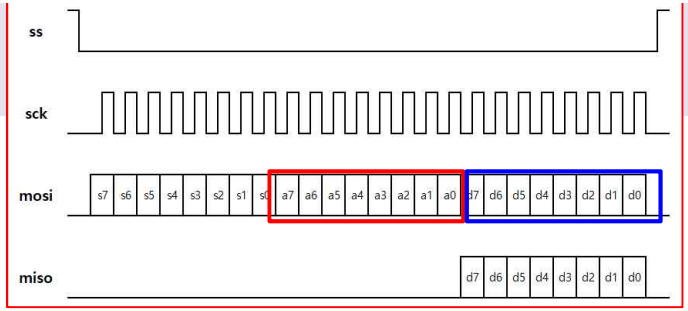
➢ *SPI Master Implementation* ➔ Code Implementation

✓ ***Port*** 정의 ➔ Module의 Input, Output Port를 정의

signal	in/out	size	description
reset	*input*	[0]	main reset, active low
clock	*input*	[0]	main clock, 100Mhz
freq	*input*	[9:0]	sck frequency 설정, 1 : 50Mhz, 1023 : 97.56 Khz, sck_freq = 100Mhz / (freq+1)
start_w	*input*	[0]	write를 위한 start flag, active high (pulse 로 인가함)
start_r	*input*	[0]	read를 위한 start flag, active high (pulse 로 인가함)
addr	*input*	[7:0]	read / write address
wdata	*input*	[7:0]	write data
rdata	*output*	[7:0]	read data
done	*output*	[0]	read / write done flag
ss	*output*	[0]	spi ss
sck	*output*	[0]	spi sck
mosi	*output*	[0]	spi mosi (master out slave in)
miso	*input*	[0]	spi miso (master in slave out)

Next Slide

SPI Master Implementation



➢ *SPI Master Implementation ➔ Code Implementation*

✓ ***Port*** 정의 ➔ Module의 Input, Output Port를 정의

- reset, clock : reset, clock 이 필요➔ clock은 보드상에 장착된 100Mhz를 사용
- freq : spi 통신의 frequency를 설정➔ spi_master 모듈을 사용하는 상위 모듈에서 freq를 설정

  ➔ frequency 설정은 main clock를 분주해서 설정➔ ***sck_freq = clock / (freq + 1)***
- start_w : spi write를 시작하기 위한 start flag ➔ spi_master 상위 모듈에서 spi write를 시작하라는 명령어를 pulse 로 줌 ➔ spi_master 모듈은 idle 상태에서 start_w 가 enable 되면 ready 상태
- start_r : spi read를 시작하기 위한 start flag ➔ spi_master 상위 모듈에서 spi read를 시작하라는 명령어를 pulse 로 줌 ➔ spi_master 모듈은 idle 상태에서 start_r 가 enable 되면 ready 상태
- addr : spi read/write 를 위한 address ➔ SPI Timing 의 a7 ~ a0 에 해당하는 값 : start_w / start_r 을 enable 하기 전에 값을 설정
- wdata : spi write 를 위한 data ➔ mosi를 통하여 d7 ~ d0까지 전송; start_w / start_r 을 enable 하기 전에 값을 설정
- rdata : spi read 동작시에 spi slave로 부터 읽은 데이터가 rdata에 저장
- done : spi read / write 동작 완료
- ***ss, sck, mosi, miso*** : spi master 신호➔ freq ~ wdata까지 설정되면 spi 전송

Continue

SPI Master Implementation

➢ *spi_master_exam.xpr*

➢ *SPI Master Implementation* ➔ *Code Implementation*

Continue

✓ Port 정의 ➔ Module의 Input, Output Port를 정의

❖ ***SPI write*** 동작

1. spi_master 모듈에 ***freq, addr, wdata***를 **입력**
2. ***start_w*** 을 active로 만들어 줌 (pulse로 인가)
3. spi_master 모듈은 ***ss, sck, mosi*** 에 해당 값을 **전송** ➔ 전송이 완료되면 done 값을 active로 만들어 줌
4. done 신호가 active가 되면 전송이 완료되었으므로 다음 동작을 진행

❖ ***SPI read*** 동작

1. spi_master 모듈에 ***freq, addr*** 를 **입력**
2. start_r 을 active로 만듬 (pulse로 인가)
3. ***spi_master*** 모듈은 **ss, sck, mosi 에 해당 값을 전송**하고 **miso를 통해 데이터를 읽어서 rdata에 저장**
4. 전송이 완료되면 done 값을 active로 만들어 줌
5. done 신호가 active가 되면 전송이 완료되었으므로 다음 동작을 진행

SPI Master Implementation

➢ *spi_master_exam.xpr*

➢ *SPI Master Implementation* ➔ Code Implementation

✓ Port 정의 ➔ Module의 Input, Output Port를 정의

spi_master . v (1)

```
1.  `timescale 1ns / 1ps

2.  module spi_master( reset , clock , freq ,
    start_w , start_r , addr  , wdata , rdata ,
    done  , ss  , sck  , mosi  , miso
3.  );
4.  input                reset  ;
5.  input                clock  ;
6.  input     [9:0]      freq   ;
7.  input                start_w ;
8.  input                start_r ;
9.  input     [7:0]      addr   ;
10.input      [7:0]      wdata  ;
11.output     [7:0]      rdata  ;
12.output                done   ;
13.output                ss     ;
14.output                sck    ;
15.output                mosi   ;
16.input                 miso   ;
```

❖ 1 ~ 16 : port 선언

✓ reset, clock : reset, clock 이 필요➔ clock은 보드상에 장착된 100Mhz를 사용

✓ freq : spi 통신의 frequency를 설정 ➔ spi_master 모듈을 사용하는 상위 모듈에서 freq를 설정 ➔ frequency 설정은 main clock를 분주해서 설정 ➔ sck_freq = clock / (freq + 1) ➔ [freq : 0 ~ 1023]

✓ start_w : spi write를 시작하기 위한 start flag ➔ spi_master 상위 모듈에서 spi write를 시작하라는 명령어를 pulse 로 줌 ➔ spi_master 모듈은 idle 상태에서 start_w 가 enable 되면 ready 상태

✓ start_r : spi read를 시작하기 위한 start flag ➔ spi_master 상위 모듈에서 spi read를 시작하라는 명령어를 pulse 로 줌 ➔ spi_master 모듈은 idle 상태에서 start_r 가 enable 되면 ready 상태

✓ addr : spi read/write 를 위한 address ➔ SPI Timing 의 a7 ~ a0 에 해당하는 값 : start_w / start_r 을 enable 하기 전에 값을 설정

✓ wdata : spi write 를 위한 data ➔ mosi를 통하여 d7 ~ d0까지 전송; start_w / start_r 을 enable 하기 전에 값을 설정

✓ rdata : spi read 동작시 spi slave로 부터 읽은 데이터가 rdata에 저장

✓ done : spi read / write 동작 완료

✓ ss, sck, mosi, miso : spi master 신호➔ freq ~ wdata까지 설정되면 spi 전송

SPI Master Implementation

➔ *Port 정의*

➔ ***State 정의***

SPI Master Implementation

➢ *spi_master_exam.xpr*

➢ *SPI Master Implementation* ➔ Code Implementation

❖ ***State*** 정의 ➔ **SM에서 사용할 State 정의**

✓ ***spi_master***는 총 **4개의 State**를 정의

state	*transition condition (state ➔ next state)*	*state description*
M_IDLE	start_w or start_r 이 enable 되면 (M_IDLE)➔ (M_READY) ready 상태로 이동	idle state
M_READY	spi 통신을 위한 준비가 끝나면 (M_READY)➔ (M_SEND) send 상태로 이동	ready for send
M_SEND	spi write or spi read가 완료되면(M_SEND) ➔ (M_DONE) done 상태로 이동	send
M_DONE	done flag 생성후 (M_DONE)➔ (M_IDLE) idle 상태로 이동	finish

1. idle : 통신이 없는 상태 ➔ SM의 초기 상태는 항상 idle
2. ready : 통신을 시작하면서 준비해야 하는 것들을 처리 ➔ **ss를 enable (0)**하고, **spi read/write flag(rw_flag)**를 설정
3. send : spi write, spi read 송수신
4. done : spi 송수신을 완료 ➔ **ss를 disable(1로)**하고, 전송이 완료를 알려주는 **done flag ➔ 1**

SPI Master Implementation

➢ *spi_master_exam.xpr*

➢ *SPI Master Implementation* ➔ Code Implementation

✓ State 정의 ➔ SM에서 사용할 State 정의

spi_master . v (2)

```
17. // ----------------------------------------
18. parameter           SLAVE_IDW = 8'h64; //8'b0110_0100 , 8'h64=8'd100
19. parameter           SLAVE_IDR = 8'h65; //8'b0110_0101 , 8'h65=8'd101
20. // ----------------------------------------
21. // 1) define state
22. reg      [1:0]      m_state ;
23. parameter           M_IDLE   = 2'd0 ; // 2'b00
24. parameter           M_READY  = 2'd1 ; // 2'b01
25. parameter           M_SEND   = 2'd2 ; // 2'b10
26. parameter           M_DONE   = 2'd3 ; // 2'b11

27. // ----------------------------------------
28. // 2) state flag
29. wire                s_idle   = (m_state == M_IDLE   ) ? 1'b1 : 1'b0 ;
30. wire                s_ready  = (m_state == M_READY  ) ? 1'b1 : 1'b0 ;
31. wire                s_send   = (m_state == M_SEND   ) ? 1'b1 : 1'b0 ;
32. wire                s_done   = (m_state == M_DONE   ) ? 1'b1 : 1'b0 ;
```

❖ 52 – 54 : 내부에서 사용하는 parameter 정의, slave_id for write/read

❖ 56 – 62 : state 정의

❖ 64 – 69 : state flag 정의

SPI Master Implementation

➔ *Port 정의*

➔ *State 정의*

➔ ***Code Implementation***

SPI Master Implementation

➢ *spi_master_exam.xpr*

➢ *SPI Master Implementation* ➔ *Verilog Code Implementation*

✓ 코드를 구현시, ***state flag***를 이용하여 구현 ➔ 각 state 에서 해당 신호가 어떻게 동작하는지를 구현

- 각 **state** 일 때, 해당 신호가 어떻게 되는지를 구현하고, 마지막에는 **(else** 조건일 때**)** 현재의 값을 유지하기 위하여 해당신호
- 코드 구현시, 각 **state** 마다 **counter**를 만들어서 사용하면 **state** 이동이나 기타 여러가지 면에서 편리

- ex) **ss** 신호 구현

```
ss   <=   s_idle    ? (express1 or data1) :
          s_ready   ? (express1 or data1) :
          s_send    ? (express1 or data1) :
          s_done    ? (express1 or data1) :
                    ss ;
```

SPI Master Implementation

➢ spi_master_exam.xpr

➢ SPI Master Implementation ➔ Verilog Code Implementation ➔ **spi_master . v (3)**

spi_master . v (3)

```
33. // 3) code implementation
34. reg                 startw_1d, startw_2d;
35. wire                startw_pedge = startw_1d & ~startw_2d;
36. always @(posedge clock or negedge reset)
37. begin
38.             if(~reset)      begin
39.                             startw_1d <= 1'b0;
40.                             startw_2d <= 1'b0;
41.             end
42.             else            begin
43.                             startw_1d <= start_w;
44.                             startw_2d <= startw_1d;
45.             end
46. end
47.
48. reg                 startr_1d, startr_2d;
49. wire                startr_pedge = startr_1d & ~startr_2d;
50. always @(posedge clock or negedge reset)
51. begin
52.             if(~reset)      begin
53.                             startr_1d <= 1'b0;
54.                             startr_2d <= 1'b0;
55.             end
56.             else            begin
57.                             startr_1d <= start_r;
58.                             startr_2d <= startr_1d;
59.             end
60. end
61. reg                 rw_flag;        // 0 : write, 1 : read
62. always @(posedge clock or negedge reset)
63. begin
64.             if(~reset)      rw_flag <= 1'b0;
65.             else                    rw_flag <= startw_pedge ? 1'b0 : startr_pedge ? 1'b1 : rw_flag ;
66. end
```

❖ 34 ~46 : start_w 신호의 positive edge[pulse]를 생성
➔ 이 신호는 idle 상태에서 ready 상태로 변경될 때 사용

❖ 48 ~ 60 : start_r 신호의 positive edge[pulse]를 생성
➔ 이 신호는 idle 상태에서 ready 상태로 변경될 때 사용

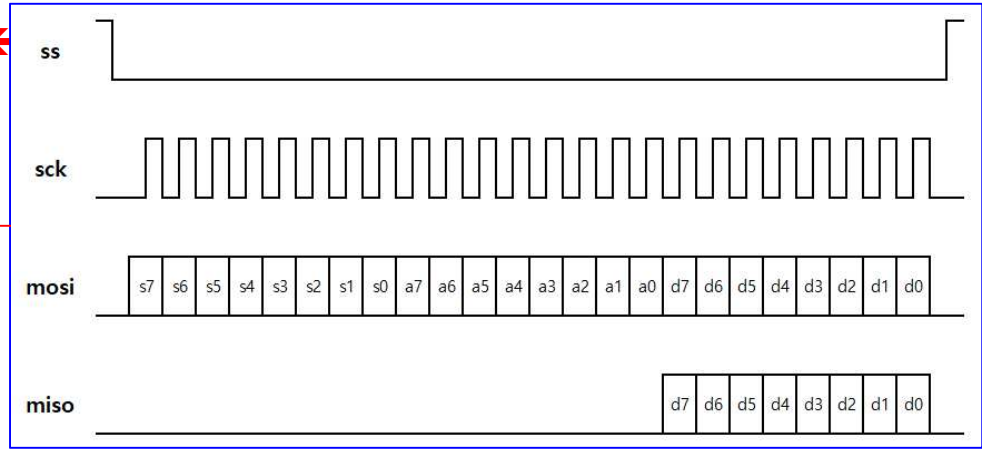
❖ 61 ~ 66 : *rw_flag* ➔ *0 : spi_write, 1 : spi_read*

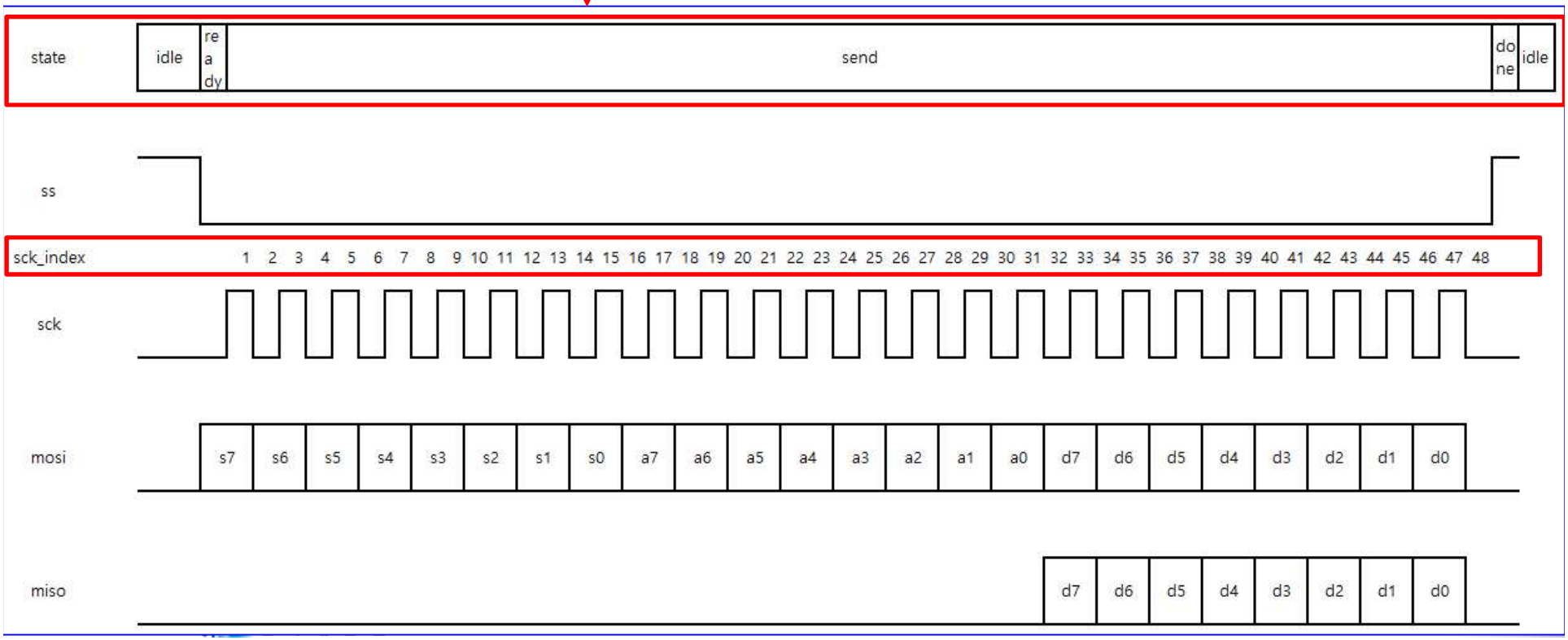
SPI Master Implementation

➢ *spi_master_exam.xpr*

➢ *SPI Master Implementation* ➔ *Verilog Code Implementation* ➔ ***Timing Diagram***

✓ ***SPI Timing*** 에 **state 정보**와 ***sck_index***를 추가 ➔





SPI Master Implementation

➢ spi_master_exam.xpr

➢ SPI Master Implementation ➔ Verilog Code Implementation ➔ **spi_master . v (4)**

spi_master . v (4)

```
67. reg         [9:0]           ready_cnt;
68. always @(posedge clock or negedge reset)
69. begin
70.             if(~reset)      ready_cnt <= 10'b0;
71.             else                            ready_cnt <= ~s_ready ? 10'b0 : ready_cnt+1'b1;
72. end
```

❖ 67 ~ 72 : ready state 에서 사용되는 counter

➔ 각각의 state 마다 카운터를 만들어 두면 상태 전이나 기타 용도로 사용

➔ ready_cnt = 0 일 때, [ss : 1 → 0]로 되고, sck 의 반주기가 지났을 때, [ready → send] 됨

```
73. reg         [3:0]           done_cnt;
74. always @(posedge clock or negedge reset)
75. begin
76.             if(~reset)      done_cnt <= 4'b0;
77.             else                            done_cnt <= ~s_done ? 4'b0 : done_cnt+1'b1;
78. end
```

❖ 73 ~ 78 : done state 에서 사용되는 counter

➔ done_cnt = 15 가 될 때 [done → idle] 되도록 setting

```
79. reg         [9:0]           sck_cnt;
80. always @(posedge clock or negedge reset)
81. begin
82.             if(~reset)      sck_cnt <= 10'b0;
83.             else                            sck_cnt <= ~s_send ? 10'b0 : (sck_cnt==freq) ? 10'b0 : sck_cnt+1'b1 ;
84. end
```

❖ 79 ~ 84 : sck 를 만들기 위한 counter ➔ freq(=10'd100)→10'h64 만큼 sck clock 만들어 줌

```
85. reg         [5:0]           sck_index;
86. always @(posedge clock or negedge reset)
87. begin
88.             if(~reset)      sck_index <= 6'b0;
89.             else                            sck_index <= ~s_send ? 6'b0 : (sck_cnt==10'b0) ? sck_index+1'b1 : sck_index;
90. end
```

❖ 85 ~ 90 : send state 에서 sck 갯수를 알려주는 counter

➔ sck 반주기 만큼 지났을 때 counter 가 증가 ➔ sck_index 값에 따라서 slave_id, addr, wdata 를 전송하고[*rdata → read*] ➔ sck_index 값 48 될 때 [send → done] 됨

SPI Master Implementation

➢ *spi_master_exam.xpr*

➢ *SPI Master Implementation* ➔ *Verilog Code Implementation* ➔ ***spi master . v (5)***

spi_master . v (5)

```
91. reg                         ss;
92. always @(posedge clock or negedge reset)
93. begin
94.             if(~reset)      ss <= 1'b1;
95.             else                            ss <=  s_idle  ? 1'b1 :
96.                             (s_ready & (ready_cnt==10'd0) ) ? 1'b0 :
97.                             (s_done  & (done_cnt==4'd15)  ) ? 1'b1 : ss;
98. end
```

❖ 136 – 143 : ss 신호를 생성

➔ ss 는 *Active Low* 로 동작

➔ 따라서 [*reset* ➔ *ss= 1*] , [*idle 상태* ➔ *ss = 1*]

➔ [*ready* 상태 ➔ *ss = 0(active)*] 으로 되었다가 [*done* 상태에서 ss = 1(*deactive*)]

➔ 그 외의 상태에서는 이전 값을 유지하기 위하여 ss

```
99. reg                         sck;
100.always @(posedge clock or negedge reset)
101.begin
102.            if(~reset) sck <= 1'b0;
103.            else            sck <= ~s_send ? 1'b0 : ((sck_index<6'd48) && (sck_cnt==10'b0)) ? ~sck : sck ;
104.end
```

❖ 99 ~ 104 : sck 신호를 생성

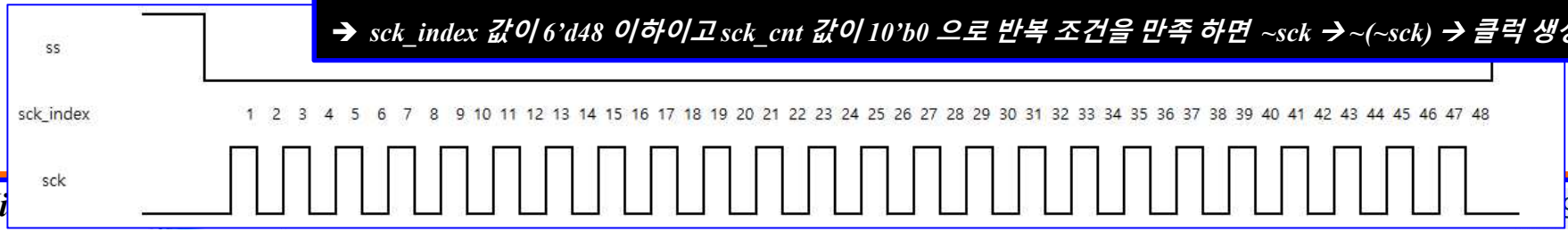
➔ sck 신호는 s_send(=1) 상태에서만 동작하고 나머지 상태에서는 0

➔ sck_index 는 sck 의 반주기를 counting ➔ spi 통신은 총 3 바이트(24 바이트, 반주기는 48 번)를 전송

➔ sck 를 생성하는 조건문은 대략적으로 생성하고, simulation 을 통하여 세부적인 Timing 을 맞추는 방법 이용하여 세부 타이밍 구현

➔ *sck 신호는 s_send(=1) 상태에서만 동작하고 나머지 상태에서는 0*

➔ *sck_index 값이 6'd48 이하이고 sck_cnt 값이 10'b0 으로 반복 조건을 만족 하면 ~sck ➔ ~(~sck) ➔ 클럭 생성*



SPI Master Implementation

➢ spi_master_exam.xpr

➢ SPI Master Implementation ➔ Verilog Code Implementation ➔ **spi_master . v (6)**

spi_master . v (6)

```
105. reg                    mosi;
106.always @(posedge clock or negedge reset)
107.begin
108.            if(~reset) mosi <= 1'b0;
109.            else  mosi <= s_idle ? 1'b0 : // Line 18~19 ➔ SLAVE_IDW = 8'h64; //8'b0110_0100 , SLAVE_IDR = 8'h65; //8'b0110_0101
110.                        (s_ready & (ready_cnt==10'd10) ) ? (~rw_flag ? SLAVE_IDW[7] : SLAVE_IDR[7]) :
111.                        (s_send  & (sck_index==6'd1 ) & (sck_cnt==10'b0)) ? (~rw_flag ? SLAVE_IDW[6] : SLAVE_IDR[6]) :
112.                        (s_send  & (sck_index==6'd3 ) & (sck_cnt==10'b0)) ? (~rw_flag ? SLAVE_IDW[5] : SLAVE_IDR[5]) :
113.                        (s_send  & (sck_index==6'd5 ) & (sck_cnt==10'b0)) ? (~rw_flag ? SLAVE_IDW[4] : SLAVE_IDR[4]) :
114.                        (s_send  & (sck_index==6'd7 ) & (sck_cnt==10'b0)) ? (~rw_flag ? SLAVE_IDW[3] : SLAVE_IDR[3]) :
115.                        (s_send  & (sck_index==6'd9 ) & (sck_cnt==10'b0)) ? (~rw_flag ? SLAVE_IDW[2] : SLAVE_IDR[2]) :
116.                        (s_send  & (sck_index==6'd11) & (sck_cnt==10'b0)) ? (~rw_flag ? SLAVE_IDW[1] : SLAVE_IDR[1]) :
117.                        (s_send  & (sck_index==6'd13) & (sck_cnt==10'b0)) ? (~rw_flag ? SLAVE_IDW[0] : SLAVE_IDR[0]) :
118.                        (s_send  & (sck_index==6'd15) & (sck_cnt==10'b0)) ?  addr[7] : //top_module input ➔ addr(8'h55) ➔ 8'b0101_0101
119.                        (s_send  & (sck_index==6'd17) & (sck_cnt==10'b0)) ?  addr[6] :
120.                        (s_send  & (sck_index==6'd19) & (sck_cnt==10'b0)) ?  addr[5] :
121.                        (s_send  & (sck_index==6'd21) & (sck_cnt==10'b0)) ?  addr[4] :
122.                        (s_send  & (sck_index==6'd23) & (sck_cnt==10'b0)) ?  addr[3] :
123.                        (s_send  & (sck_index==6'd25) & (sck_cnt==10'b0)) ?  addr[2] :
124.                        (s_send  & (sck_index==6'd27) & (sck_cnt==10'b0)) ?  addr[1] :
125.                        (s_send  & (sck_index==6'd29) & (sck_cnt==10'b0)) ?  addr[0] :
126.                        (s_send  & (sck_index==6'd31) & (sck_cnt==10'b0)) ? (~rw_flag ? wdata[7] : 1'b0) : //wdata(8'haa) ➔ 8'b1010_1010
127.                        (s_send  & (sck_index==6'd33) & (sck_cnt==10'b0)) ? (~rw_flag ? wdata[6] : 1'b0) : // top_module input
128.                        (s_send  & (sck_index==6'd35) & (sck_cnt==10'b0)) ? (~rw_flag ? wdata[5] : 1'b0) :
129.                        (s_send  & (sck_index==6'd37) & (sck_cnt==10'b0)) ? (~rw_flag ? wdata[4] : 1'b0) :
130.                        (s_send  & (sck_index==6'd39) & (sck_cnt==10'b0)) ? (~rw_flag ? wdata[3] : 1'b0) :
131.                        (s_send  & (sck_index==6'd41) & (sck_cnt==10'b0)) ? (~rw_flag ? wdata[2] : 1'b0) :
132.                        (s_send  & (sck_index==6'd43) & (sck_cnt==10'b0)) ? (~rw_flag ? wdata[1] : 1'b0) :
133.                        (s_send  & (sck_index==6'd45) & (sck_cnt==10'b0)) ? (~rw_flag ? wdata[0] : 1'b0) :
134.                        (s_send  & (sck_index==6'd47) & (sck_cnt==10'b0)) ? 1'b0 : mosi ;
135.end
```

❖ 105 ~ 135 : mosi 신호를 생성

➔ idle 상태에서는 '0', send 상태에서는 *sck_index* 값에 따라 *slave_id, addr, wdata* 값을 전송 ➔ simulation 을 통하여 데이터 값들이 제대로 전송되는지 확인 작업 필요

❖ *sck_index* ➔ 홀수 == '1'

❖ *sck_cnt* ➔ sck_cnt = freq 까지 count up

❖ *rw_flag* ➔ '0' : write, '1' : read

SPI Master Implementation

➢ spi_master_exam.xpr

➢ SPI Master Implementation ➔ Verilog Code Implementation ➔ **spi_master . v (7)**

spi_master . v (7)

```
136.reg         [7:0]           rdata;
137.always @(posedge clock or negedge reset)
138.begin
139.            if(~reset)      rdata <= 8'b0;
140.            else            begin
141.                            rdata[7] <= (s_send  & (sck_index==6'd32 ) & (sck_cnt==10'b0)) ? miso : rdata[7] ;
142.                            rdata[6] <= (s_send  & (sck_index==6'd34 ) & (sck_cnt==10'b0)) ? miso : rdata[6] ;
143.                            rdata[5] <= (s_send  & (sck_index==6'd36 ) & (sck_cnt==10'b0)) ? miso : rdata[5] ;
144.                            rdata[4] <= (s_send  & (sck_index==6'd38 ) & (sck_cnt==10'b0)) ? miso : rdata[4] ;
145.                            rdata[3] <= (s_send  & (sck_index==6'd40 ) & (sck_cnt==10'b0)) ? miso : rdata[3] ;
146.                            rdata[2] <= (s_send  & (sck_index==6'd42 ) & (sck_cnt==10'b0)) ? miso : rdata[2] ;
147.                            rdata[1] <= (s_send  & (sck_index==6'd44 ) & (sck_cnt==10'b0)) ? miso : rdata[1] ;
148.                            rdata[0] <= (s_send  & (sck_index==6'd46 ) & (sck_cnt==10'b0)) ? miso : rdata[0] ;
149.            end
150.end
```

❖ 136 ~ 150 : miso 신호로 부터 rdata 값을 read
➔ sck_index 와 sck_cnt 값에 따라 해당 비트를 얻음
➔ simulation 을 통하여 데이터를 읽는 포인트가 맞는지 확인

```
151.reg                         done;
152.always @(posedge clock or negedge reset)
153.begin
154.            if(~reset) done <= 1'b0;
155.            else            done <= (startw_pedge | startr_pedge) ? 1'b0 :
156.                        (s_done & (done_cnt==4'd15) ) ? 1'b1 : done;
157.end
```

❖ 151 ~ 157 : done 신호를 생성
➔ done 는 한번 active 되면 또 다른 전송이 시작되기 전까지(start_w, start_r 신호가 다시 입력될 때까지) active 상태를 유지 : 상위 모듈에서 전송이 완료되었는지 확인할 때까지 active 상태로 유지하기 위함

```
158.// ----------------------------------------
159.// 4) state transition
160.always @(posedge clock or negedge reset)
161.begin
162.            if(~reset) m_state <= 2'b0;
163.            else            m_state <= (s_idle  & (startw_pedge | startr_pedge) ) ? M_READY :
164.                            (s_ready & (ready_cnt==freq) ) ? M_SEND  :
165.                            (s_send  & (sck_index==6'd48) & (sck_cnt==10'b0)) ? M_DONE  :
166.                            (s_done  & (done_cnt==4'd15)) ? M_IDLE  : m_state ;
167.end
168.endmodule
```

❖ 158 ~ 168 : 상태 천이 구현
➔ idle 상태에서 start_w, start_r 신호가 들어오면 ready 상태 ➔ ready 상태에서 sck 반주기만큼 지나면 send 상태 ➔ send 상태에서는 sck 가 24 clock(반주기 기준 =sck_index=48)이 되면 done 상태 ➔ done 상태에서 done_cnt 가 15 가 되면 다시 idle 상태

SPI Master Implementation

➔ *Port 정의*

➔ *State 정의*

➔ *Code Implementation*

➔ ***Test Bench***

SPI Master Implementation

➢ *spi_master_exam.xpr*

➢ *SPI Master Implementation* ➔ *Verilog Code Implementation* ➔ *Test Bench :* ***spi_master_tb . v (1)***

spi_master_tb . v (1)

```
1.  `timescale 1ns / 1ps

2.  module spi_master_tb();

3.  reg                     reset, clock;
4.  initial     begin
5.              reset = 0;
6.              clock = 0;
7.
8.  #10000     reset = 1;
9.  end

10. always  #5  clock = ~clock;                         // 100 Mhz

11. reg         [14:0]      cnt;
12. always @(posedge clock or negedge reset)
13. begin
14.             if(~reset)      cnt <= 15'b0;
15.             else                        cnt <= cnt+1'b1;
16. end

17. reg                     start_w;
18. always @(posedge clock or negedge reset)
19. begin
20.             if(~reset)      start_w <= 1'b0;
21.             else                        start_w <= (cnt==15'd1000) ? 1'b1 : (cnt==15'd1010) ? 1'b0 : start_w ;
22. end

23. reg                     start_r;
24. always @(posedge clock or negedge reset)
25. begin
26.             if(~reset)      start_r <= 1'b0;
27.             else                        start_r <= (cnt==15'd6500) ? 1'b1 : (cnt==15'd6510) ? 1'b0 : start_r ;
28. end
```

❖ 3 ~ 10 : reset, clock 생성

❖ 11 ~ 16 : start_w, start_r 신호를 생성하기 위한 *counter*

❖ 17 ~ 22 : *start_w* 신호 생성

❖ 17 ~ 22 : *start_r* 신호 생성

SPI Master Implementation

➢ spi_master_exam.xpr

➢ SPI Master Implementation ➔ Verilog Code Implementation ➔ Test Bench : **spi_master_tb . v (4)**

spi_master_tb . v (2)

```
29. wire        [7:0]       rdata   ;
30. wire                    done  , ss  , sck  , mosi , s_idle , s_ready , s_send , s_done  ;
31. wire        [9:0]       sck_cnt;
32. wire                    startw_pedge;
33. wire        [5:0]       sck_index;
34. wire                    rw_flag;
35. wire                     startw_1d;
36. wire                    startw_2d;
37. spi_master   spi_master_u1(
38.             .reset   (reset   ), .clock   (clock   ),
39.             .freq    (10'd100 ), //sck_freq = 100Mhz/(freq+1) ➔ (freq=100) ➔ sck_freq= 100MHz/(100+1) = 0.99Mhz
40.             .start_w (start_w ), .start_r (start_r ), .startw_1d(startw_1d), .startw_2d(startw_2d),
41.             .startw_pedge(startw_pedge),
42.             .addr(8'h55),
43.             .wdata(8'haa),
44.             .rdata   (rdata   ),
45.             .done    (done    ),
46.             .sck_cnt(sck_cnt),
47.             .sck_index(sck_index),
48.             .ss      (ss      ),
49.             .sck     (sck     ),
50.             .mosi    (mosi    ),
51.             .s_idle    (s_idle),
52.             .s_ready  (s_ready),
53.             .s_send   (s_send),
54.             .s_done   (s_done),
55.             .rw_flag(rw_flag),
56.             .miso    (1'b0    )
57. );
58. endmodule
```

- ❖ 29 ~ 56 : spi_master module추가, freq : 100(10'h64), addr : 0x55(8'b0101_0101) , wdata : 0xaa (8'b1010_1010)

➔ spi read simulation을 진행하려면, miso(1bit) 에 적당한 데이터를 입력해 주고, miso에 입력한 데이터가 rdata에 맞게 저장되는지 확인

➔ simulation을 간단하게 진행하기 위하여 miso에 1'b0을 입력 ➔ rdata = 8'h00

➔ spi read 의 정확한 simulation은 spi slave controller 구현 후 검증

SPI Master Implementation

➔ Port 정의

➔ State 정의

➔ Code Implementation

➔ Test Bench

➔ **Simulation Check**

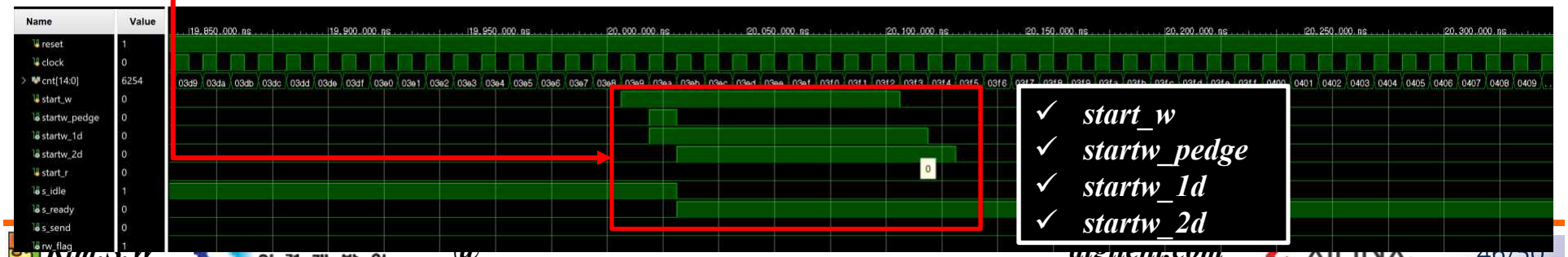
SPI Master Implementation

➢ *spi_master_exam.xpr*

➢ *SPI Master Implementation* ➔ *Verilog Code Implementation* ➔ *Test Bench : **spi_master_tb . v (4)***

- ***Simulation Result Check (1)***

✓ 전체적인 state
✓ start_w 가 enable 되면, s_idle, s_ready, s_send, s_done이 차례대로 enable
✓ start_r가 enable 될 때에도, s_idle, s_ready, s_send, s_done이 차례대로 enable

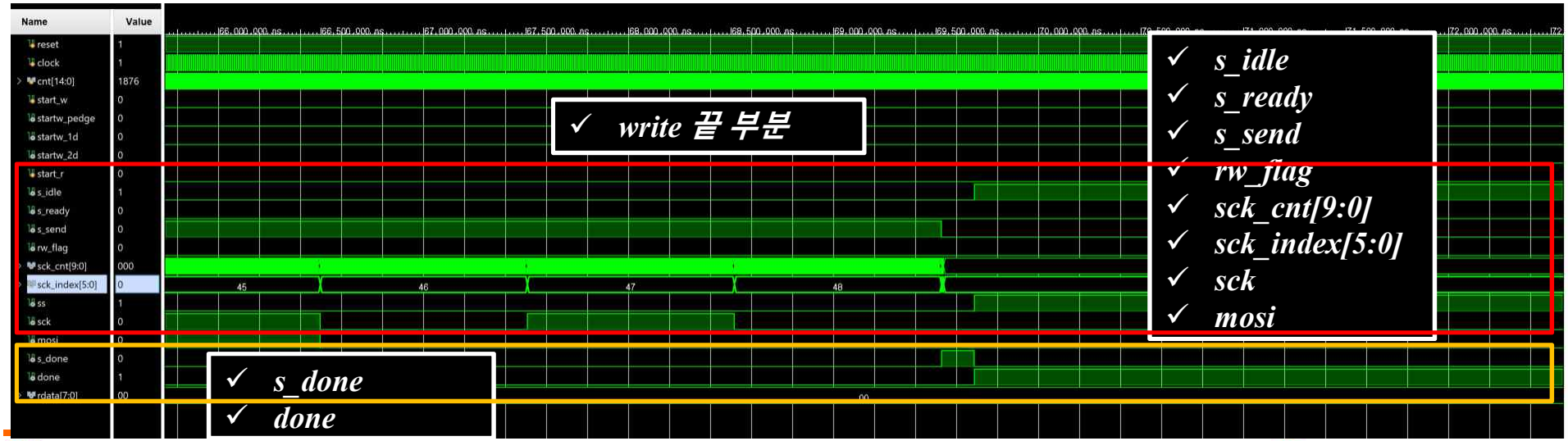


SPI Master Implementation

➢ *spi_master_exam.xpr*

➢ *SPI Master Implementation* ➔ *Verilog Code Implementation* ➔ *Test Bench :* ***spi_master_tb . v (4)***

- ***Simulation Result Check*** *(2)*

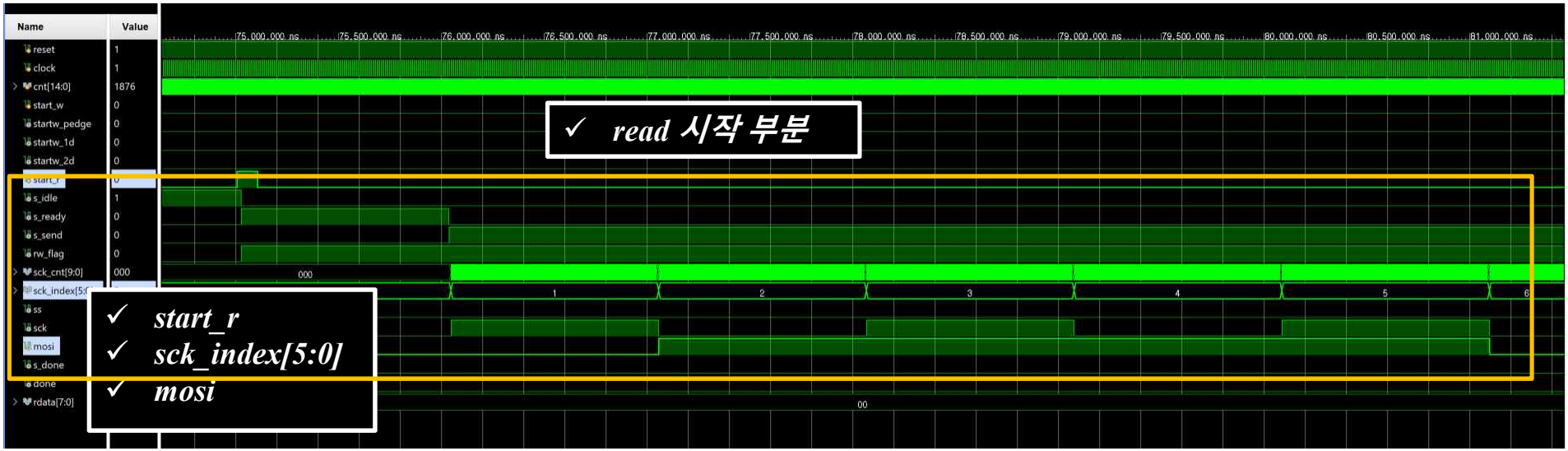


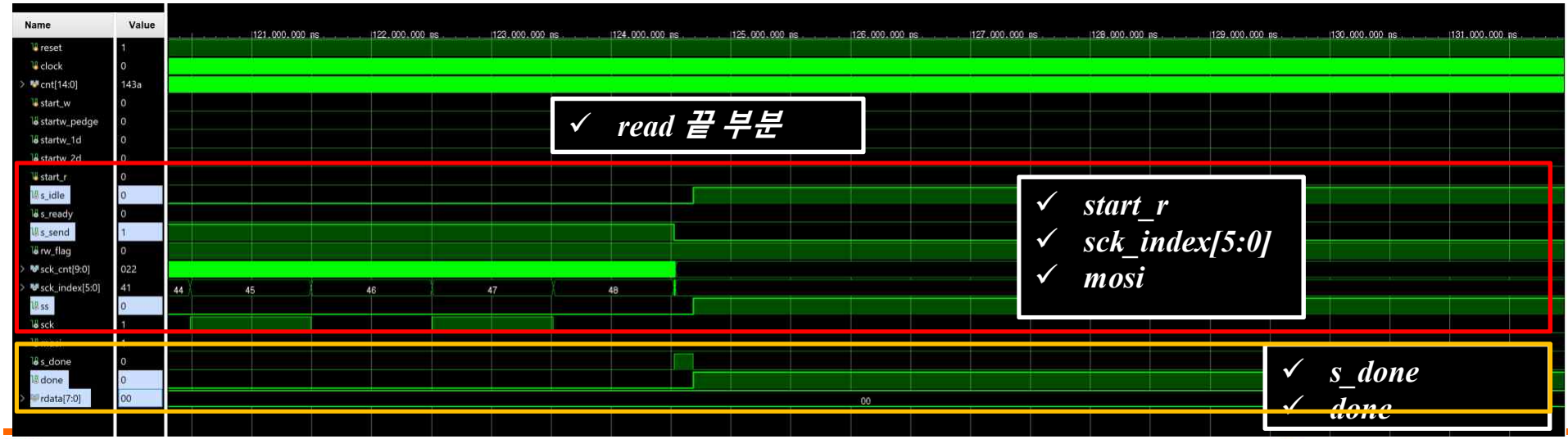
SPI Master Implementation

➢ *spi_master_exam.xpr*

➢ *SPI Master Implementation* ➔ *Verilog Code Implementation* ➔ *Test Bench :* ***spi_master_tb . v (4)***

- ***Simulation Result Check (3)***





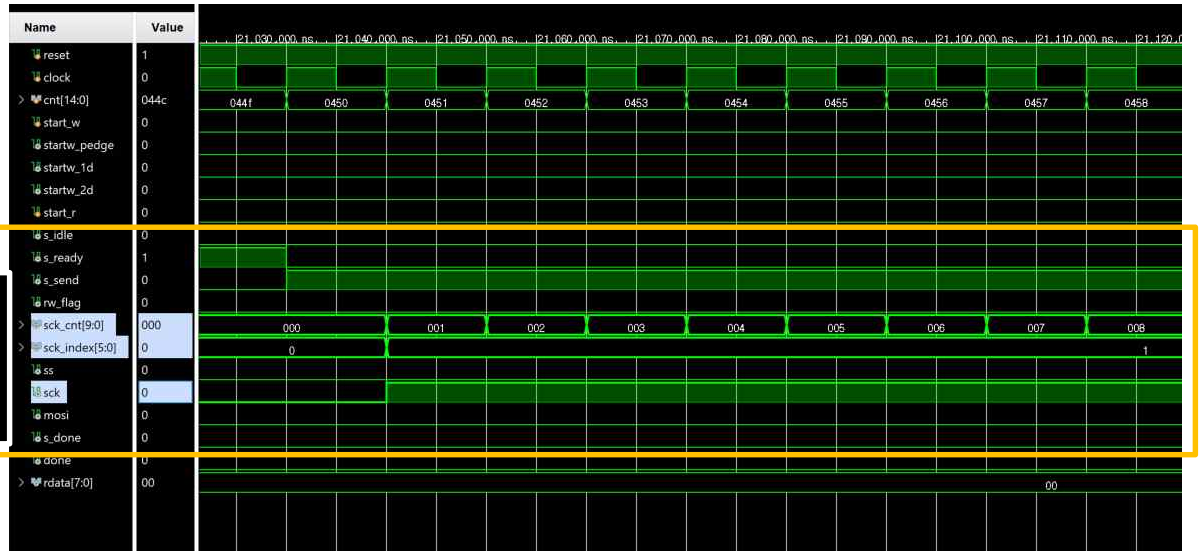
SPI Master Implementation

➢ *spi_master_exam.xpr*

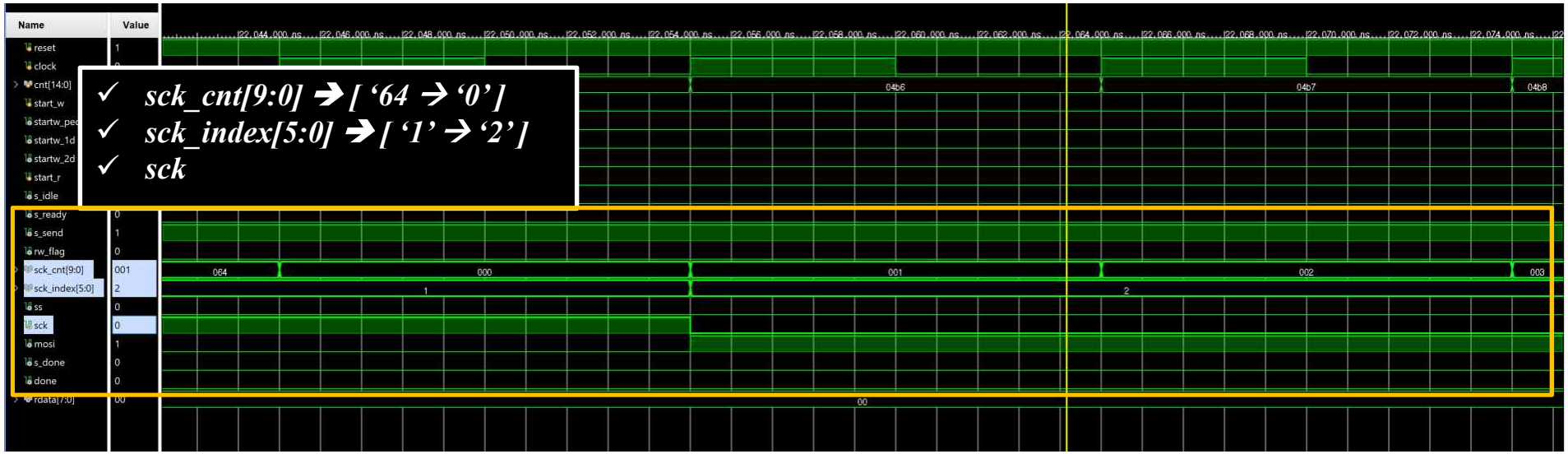
➢ *SPI Master Implementation* ➔ *Verilog Code Implementation* ➔ *Test Bench :* ***spi_master_tb . v (4)***

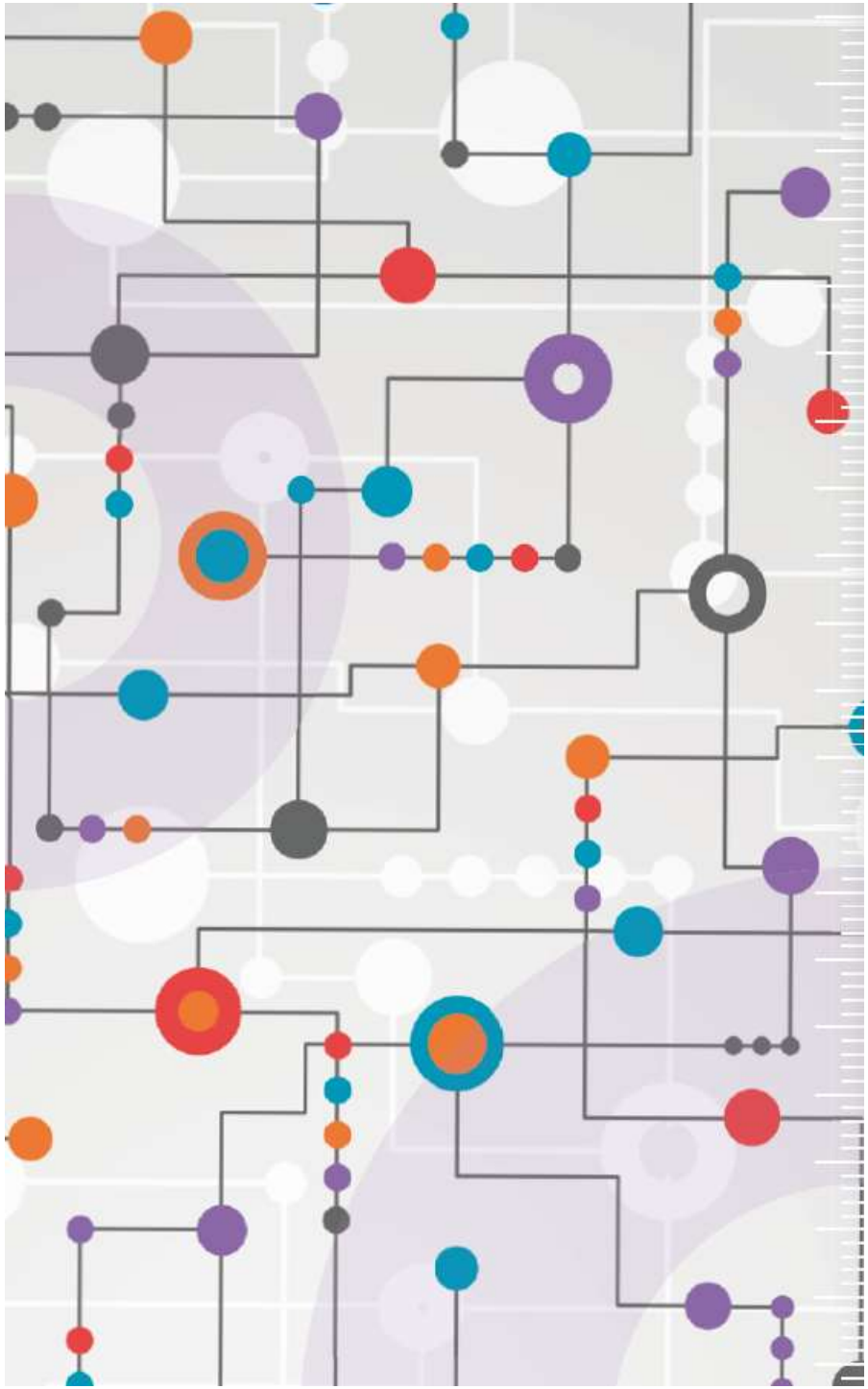
- ***Simulation Result Check*** *(4)*

✓ *sck_cnt[9:0]*
✓ *sck_index[5:0]* ➔ *['0' → '1']*
✓ *sck*



✓ *sck_cnt[9:0]* ➔ *['64 → '0']*
✓ *sck_index[5:0]* ➔ *['1' → '2']*
✓ *sck*





수고하셨습니다.