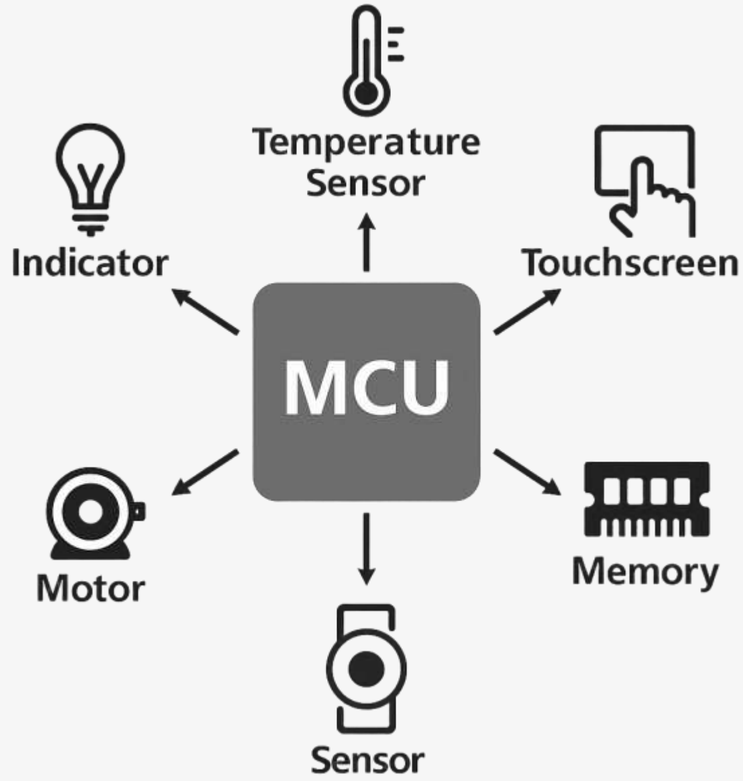
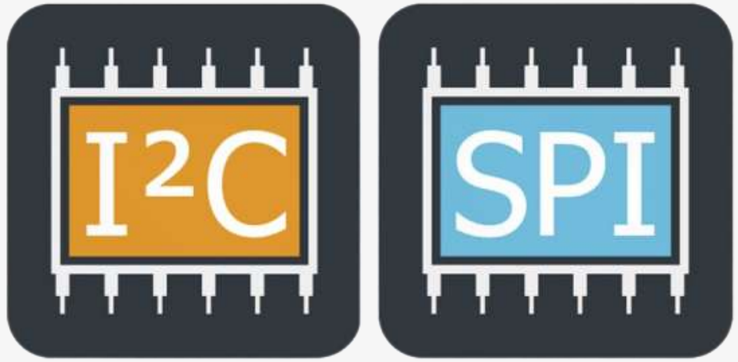
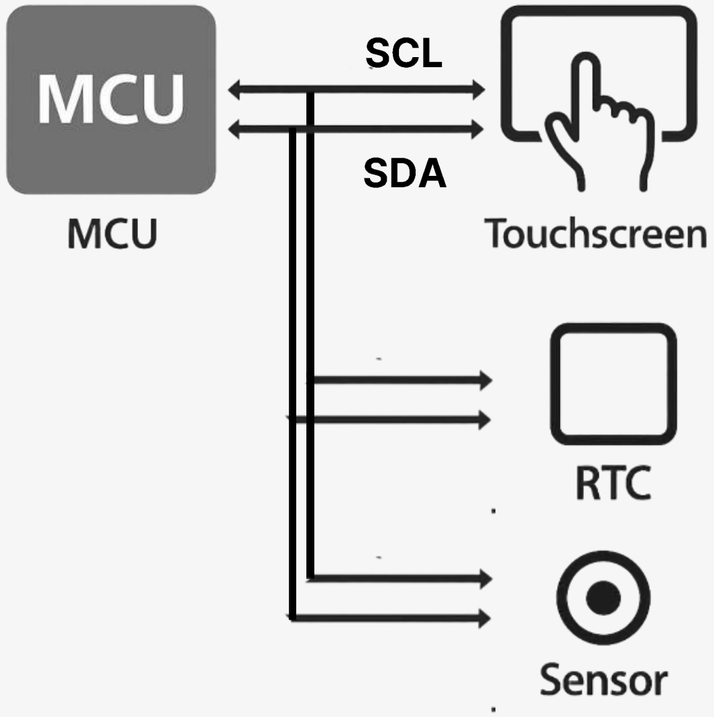
현대 MCU는 여러 기능 칩을 하나의 시스템으로 통합하는 구조

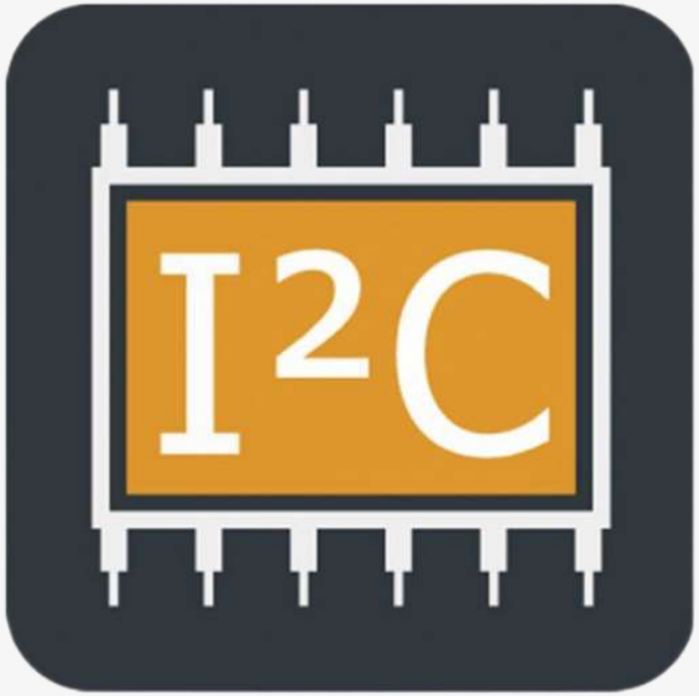


칩간의 정보교환은 어떻게 할까?



칩과 칩사이 (비교적) 짧은 wrie를 통해 데이터를 주고 받을 수 있음





I2C 기반 보드 간 직렬 통신 인터페이스 설계

발표자 오고은

목차

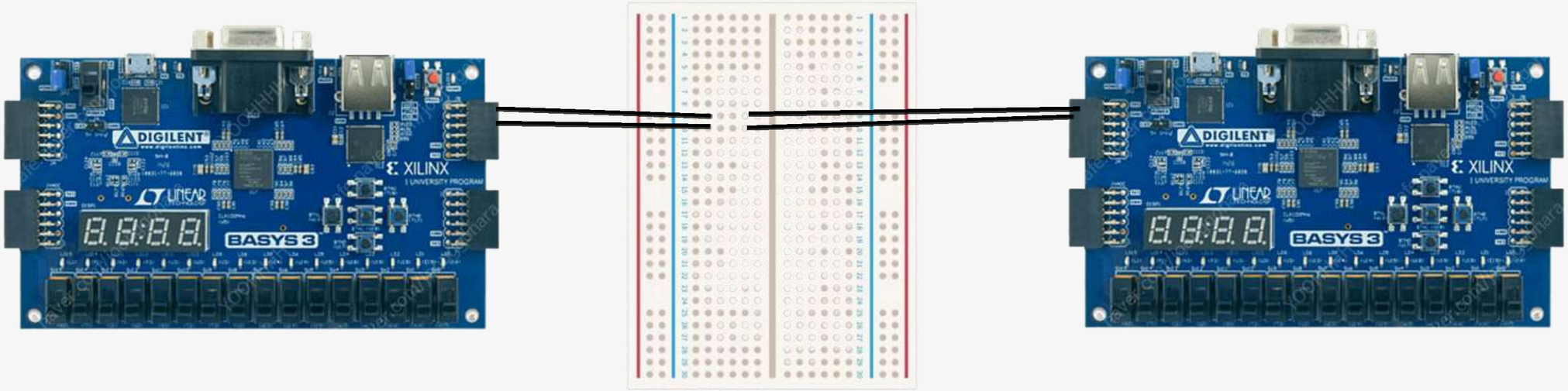
프로젝트 목표

READ, WRITE 이 가능한 I2C 통신 인터페이스 설계

microblaze와 AXI4 버스에 연결

보드간 보드 가 통신가능한 인터페이스를 만들고 간단한 애플리케이션 구현

synopsys vcs를 통해 검증



I2C 통신 프로토콜

I2C란?

칩과 칩 사이의 짧은 거리 통신을 위한 직렬 통신 방식

SCL, SDA 두개의 라인으로 통신

SCL은 Master가 생성하는 CLOCK(동기식 통신), SDA는 데이터를 주고 받는 통로

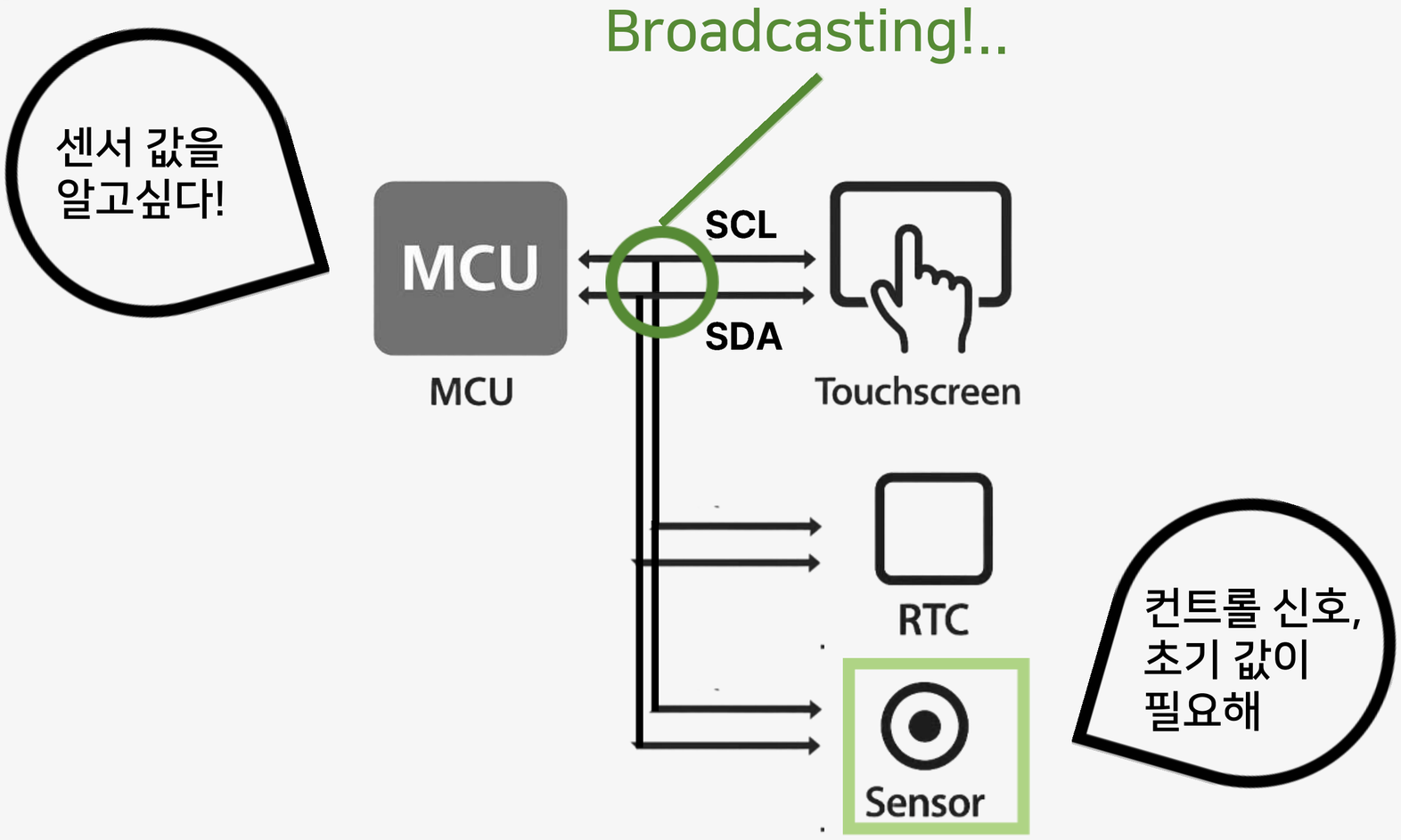
하나의 마스터와 여러 슬레이브가 통신 가능 (각 슬레이브가 고유한 HW address를 보유)

직렬 통신 방식 종류와 특징

Feature	UART	SPI	I2C
Type	Asynchronous	Synchronous	Synchronous
전송방식	1:1	1:다수	1:다수
line 수	tx, rx	MOSI, MISO, SCLK, CS	SCL, SDA(주소, data)
이중통신	full-duplex	full-duplex	half-duplex
(상대적인)속도	느리다(spi보다)	빠르다(uart, i2c 보다)	느리다(spi 보다)
(상대적인)구현난이도	simple	simple	complex

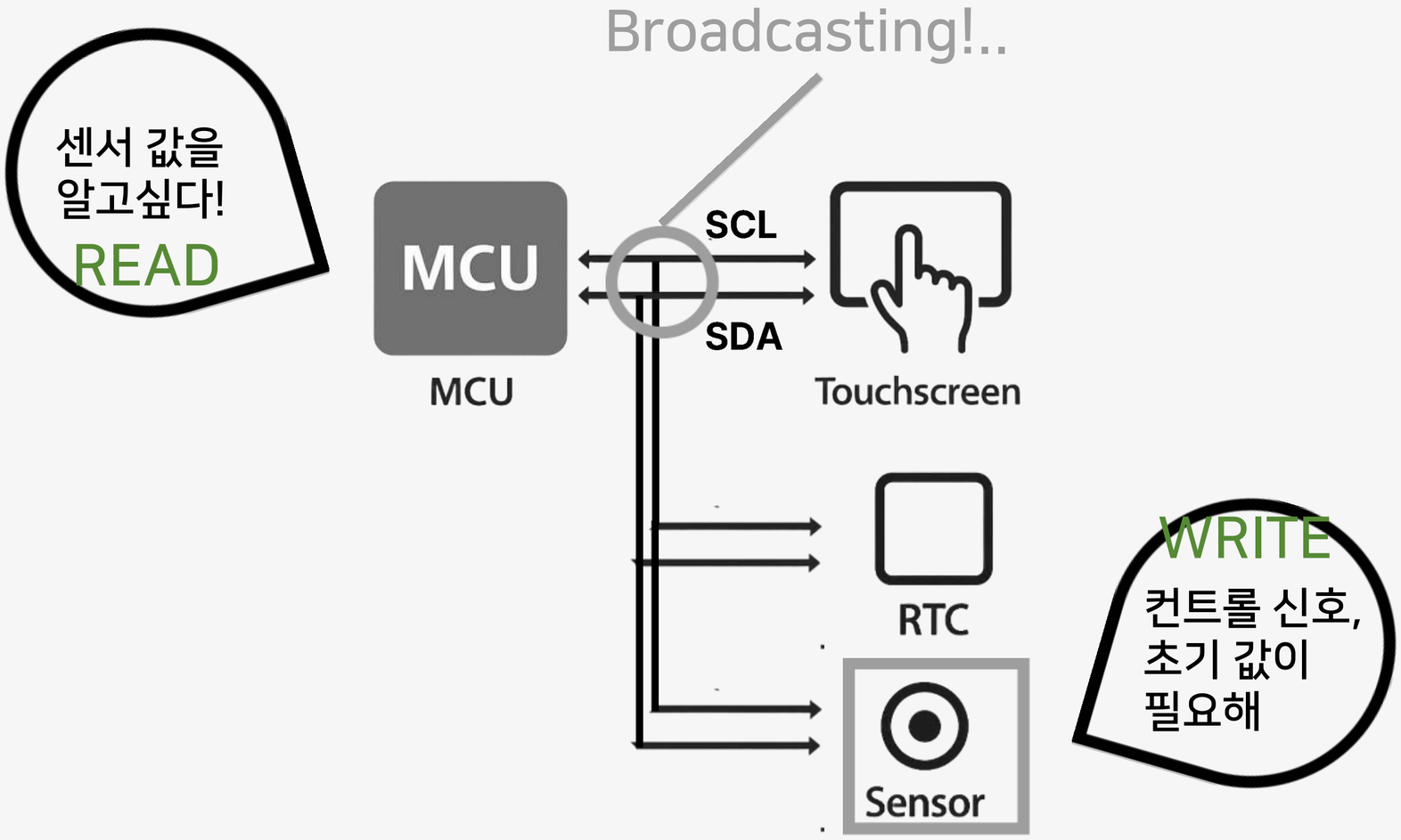
I2C 통신 프로토콜

- I2C 통신 방식



I2C 통신 프로토콜

- I2C 통신 방식



I2C 통신 프로토콜

START

통신 시작을 알림

모든 Slave가 주목

이후 전송되는 ADDR이 일치하는 슬레이브만 통신 준비

WRITE

ADDR의 LSB가 0일때

슬레이브는 ACK 신호를 보내 수신 완료를 알림

마스터는 ACK 수신 후 계속 전송 여부를 결정

STOP

통신 종료를 알림

슬레이브와 마스터 모두 IDLE 상태로 복귀

READ

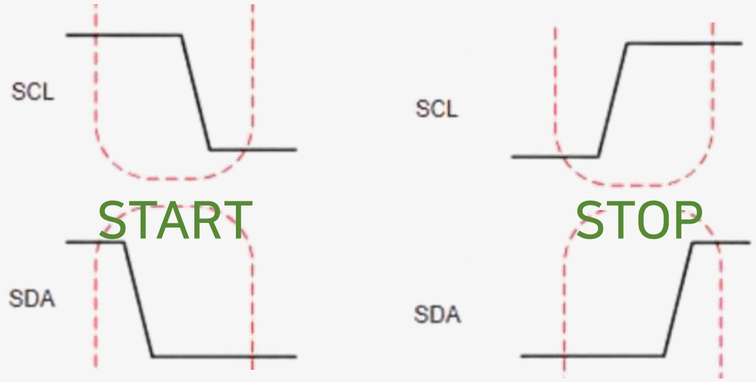
ADDR의 LSB가 1일때

마스터는 ACK/NACK 신호를 보내 수신 완료를 알림

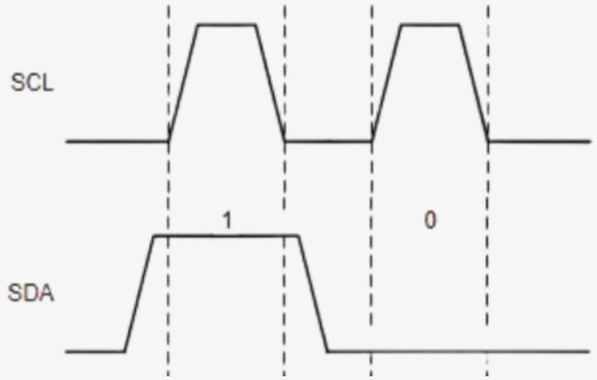
ACK : 계속 수신 NACK: 수신 종료

I2C 통신 프로토콜

START / STOP

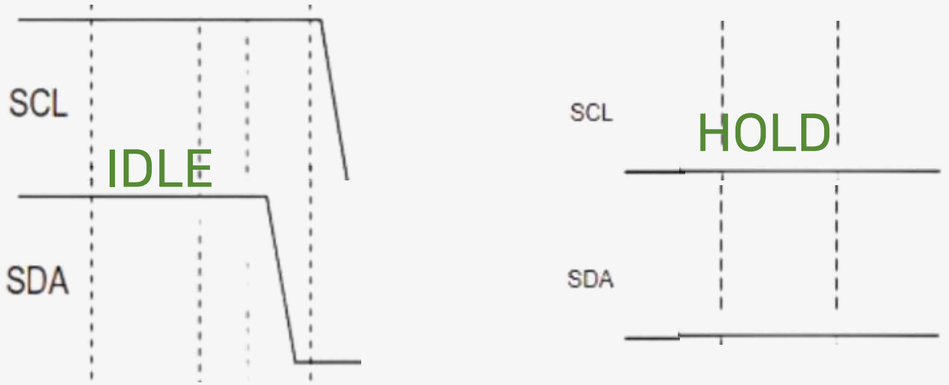


DATA 0 / 1 판별



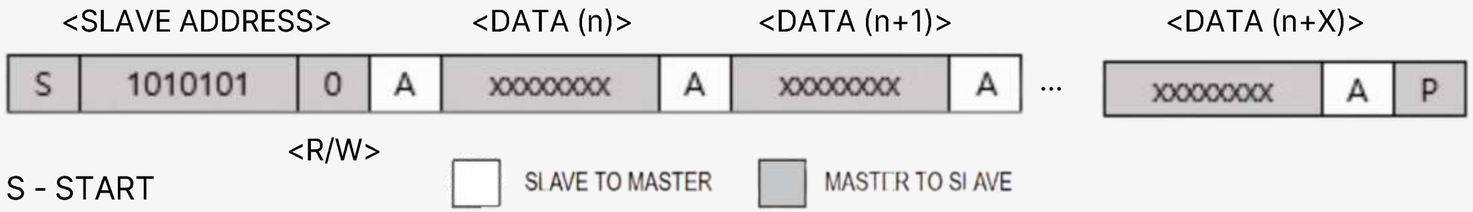
ACK / NACK

IDLE / HOLD



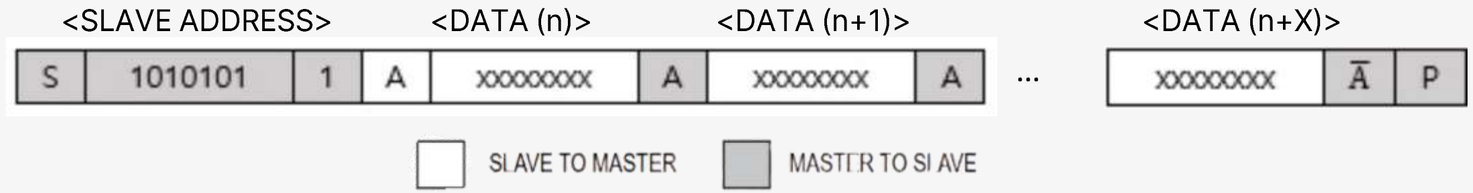
I2C 통신 프로토콜

WRITE - Slave Receiver Mode



S - START
A - Acknolwedge
P - STOP
R/W - READ/ WRITE

READ - Slave Transmiter Mode



S - START
A - Acknolwedge
!A - NOT Acknolwedge
P - STOP
R/W - READ/ WRITE

구현 목표 설정

커스텀 IP 설계

한 번에 1 Byte 를 송수신 할 수 있는 I2C 기반 IP 만들기

Speed mode는 Standard-mode로 설정

I²C Mode	Maximum Bit Rate
Standard-mode	100kbps
Fast-mode	400kbps
Fast-mode Plus	1Mbps
High-speed mode	3.4Mbps
Ultra-Fast mode	5Mbps

마스터가 슬레이브에게 SCL 선을 통해100kHz 주기의 클럭을 공급

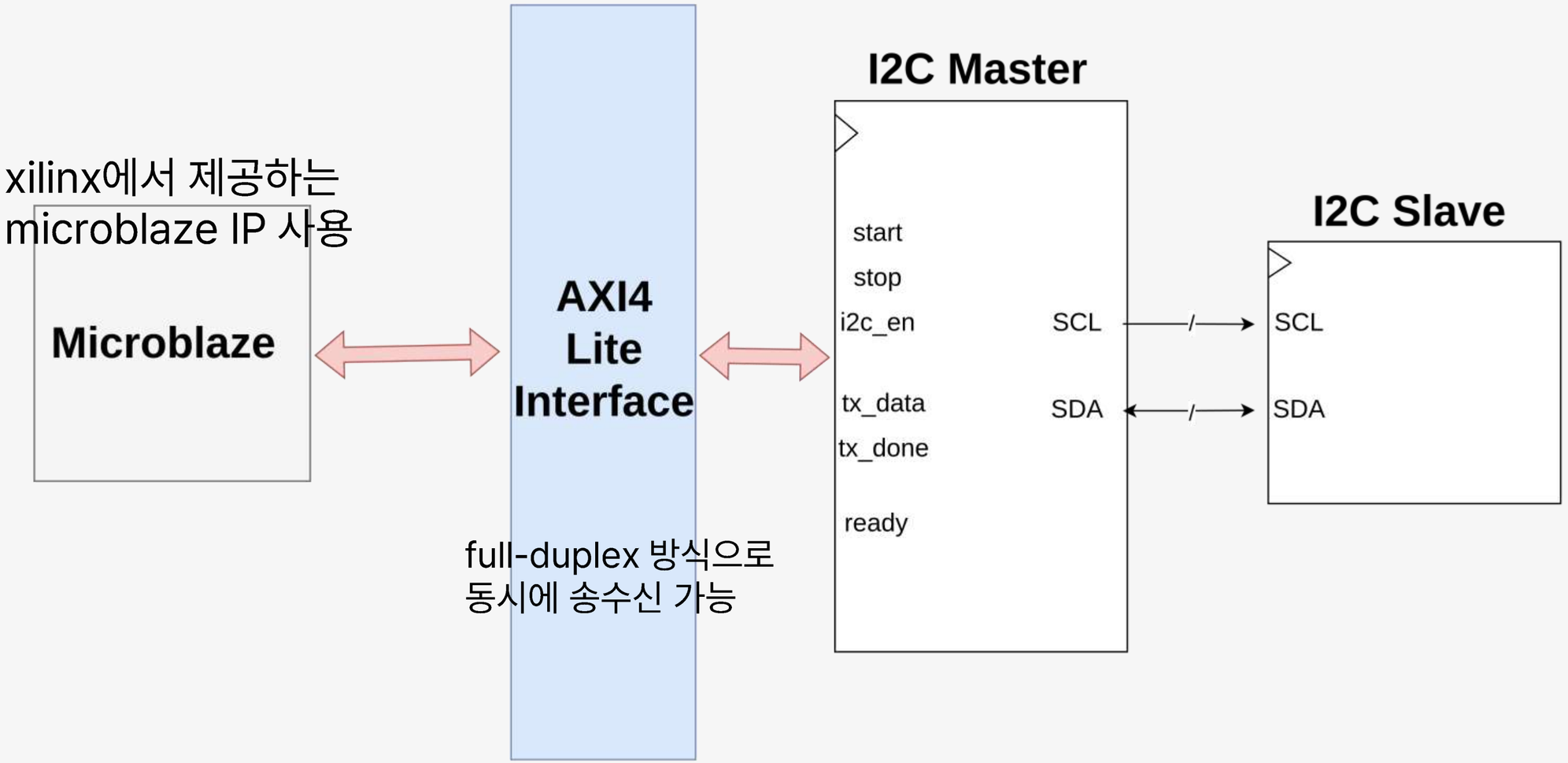
애플리케이션 구현 & 검증

설계한 Master IP를 Vivado IP Integrator를 이용하여 Microblaze SOC에 통합

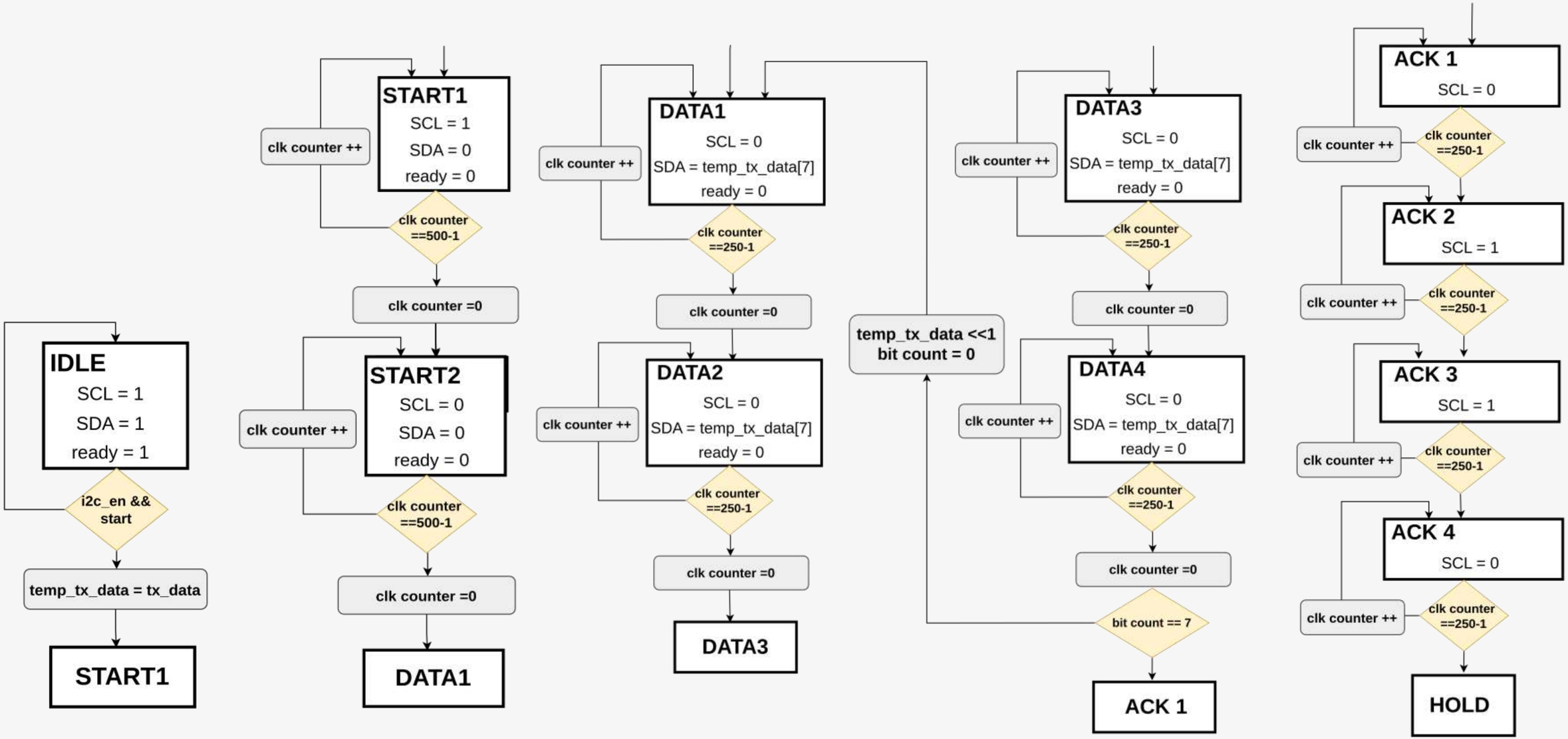
슬레이브 모듈을 합성한 보드와 물리적 wire 를 연결 하여 통신 환경 구성

vitis tool 을 이용하여 C언어 프로그램을 통한 I2C 송수신 제어

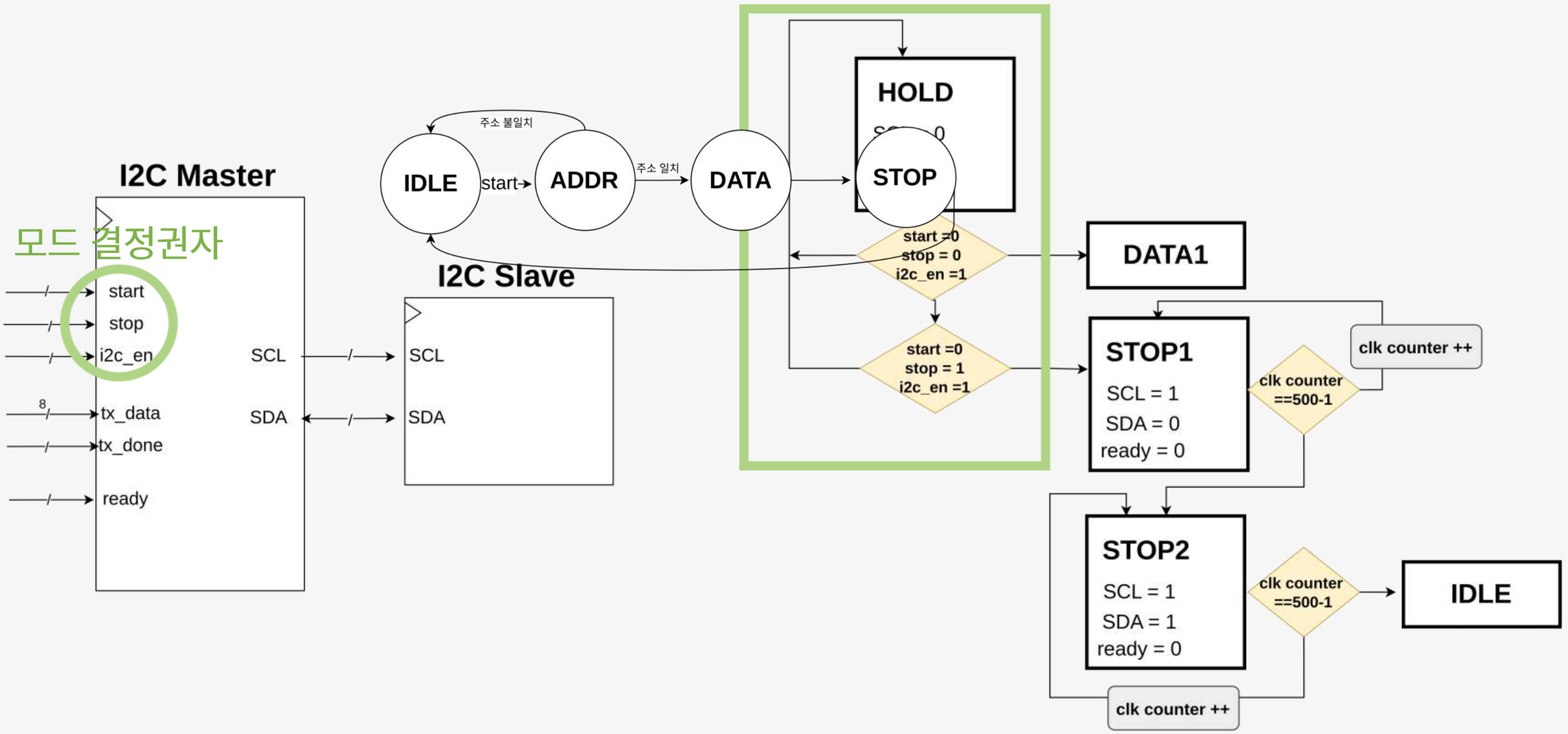
Schematic


xilinx에서 제공하는
microblaze IP 사용
Microblaze
AXI4
Lite
Interface
full-duplex 방식으로
동시에 송수신 가능
I2C Master
start
stop
i2c_en
tx_data
tx_done
ready
SCL
SDA
I2C Slave
SCL
SDA

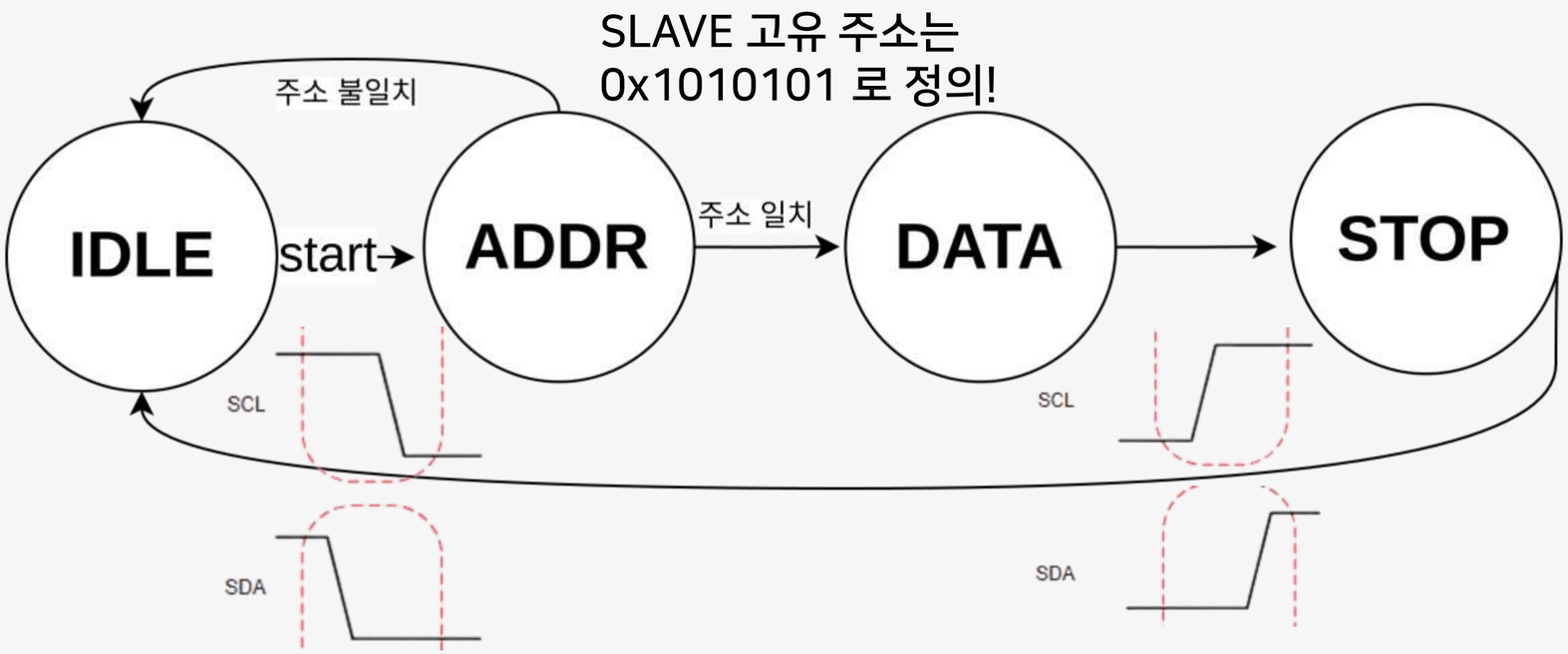
WRITE Mode


IDLE
SCL = 1
SDA = 1
ready = 1
i2c_en && start
temp_tx_data = tx_data
START1
START1
SCL = 1
SDA = 0
ready = 0
clk counter ++
clk counter ==500-1
clk counter =0
START2
SCL = 0
SDA = 0
ready = 0
clk counter ++
clk counter ==500-1
clk counter =0
DATA1
DATA1
SCL = 0
SDA = temp_tx_data[7]
ready = 0
clk counter ++
clk counter ==250-1
clk counter =0
DATA2
SCL = 0
SDA = temp_tx_data[7]
ready = 0
clk counter ++
clk counter ==250-1
clk counter =0
DATA3
temp_tx_data <<1
bit count = 0
DATA3
SCL = 0
SDA = temp_tx_data[7]
ready = 0
clk counter ++
clk counter ==250-1
clk counter =0
DATA4
SCL = 0
SDA = temp_tx_data[7]
ready = 0
clk counter ++
clk counter ==250-1
clk counter =0
bit count == 7
ACK 1
ACK 1
SCL = 0
clk counter ++
clk counter ==250-1
ACK 2
SCL = 1
clk counter ++
clk counter ==250-1
ACK 3
SCL = 1
clk counter ++
clk counter ==250-1
ACK 4
SCL = 0
clk counter ++
clk counter ==250-1
HOLD

WRITE Mode

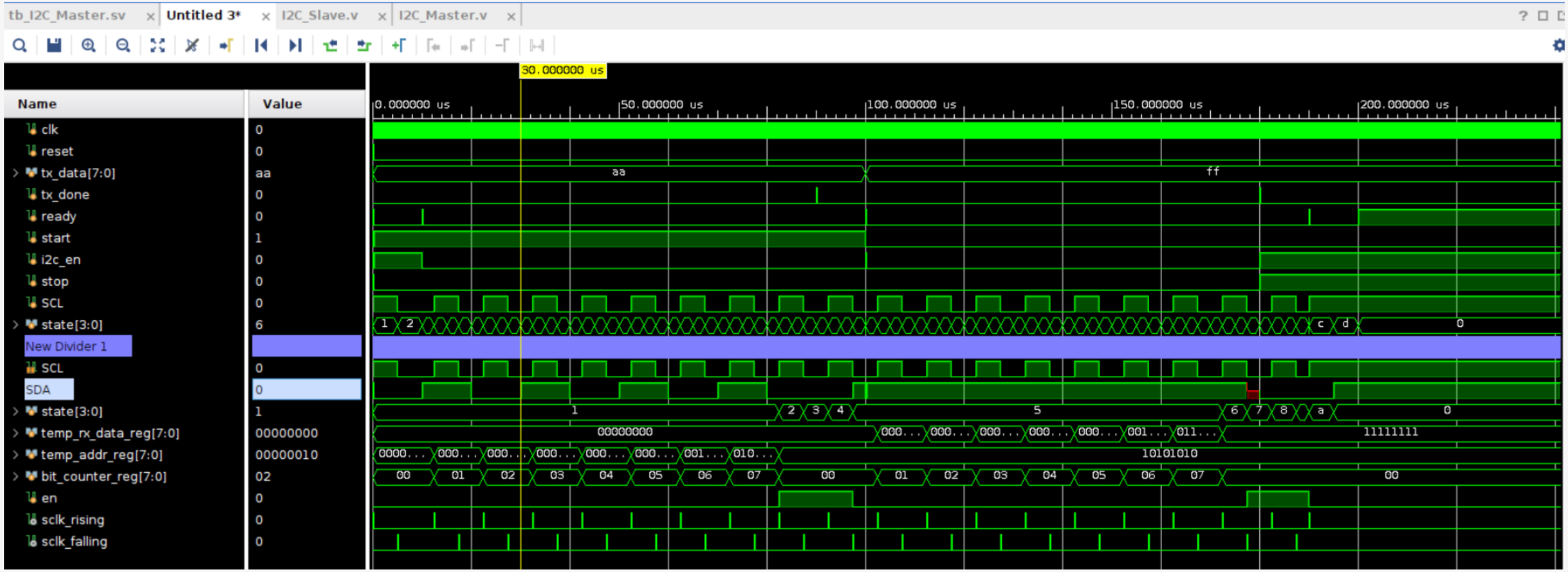

I2C Master
모드 결정권자
start
stop
i2c_en
tx_data
tx_done
ready
SCL
SDA
8
I2C Slave
SCL
SDA
IDLE
start
ADDR
주소 일치
주소 불일치
DATA
HOLD
STOP
start =0
stop = 0
i2c_en =1
start =0
stop = 1
i2c_en =1
DATA1
STOP1
SCL = 1
SDA = 0
ready = 0
clk counter ++
clk counter
==500-1
STOP2
SCL = 1
SDA = 1
ready = 0
clk counter ++
clk counter
==500-1
IDLE

WRITE Mode - SLAVE FSM



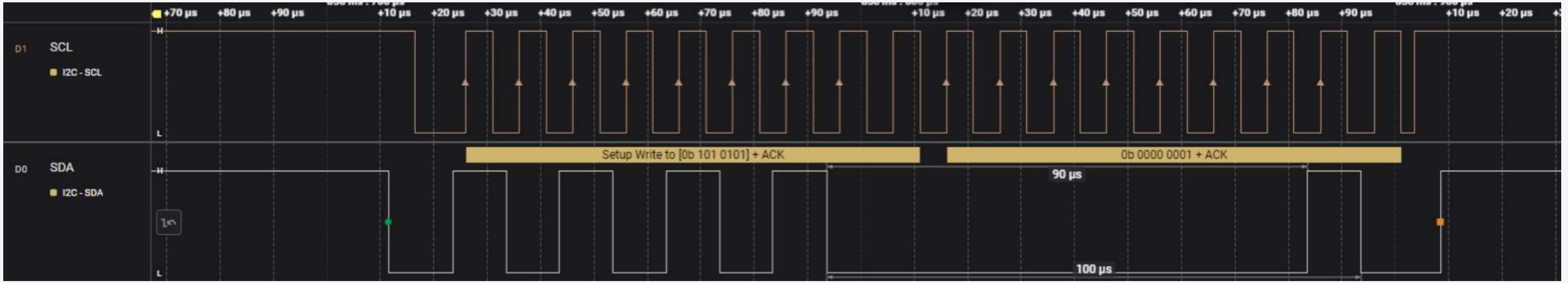
WRITE Mode

결과



WRITE Mode

- 결과

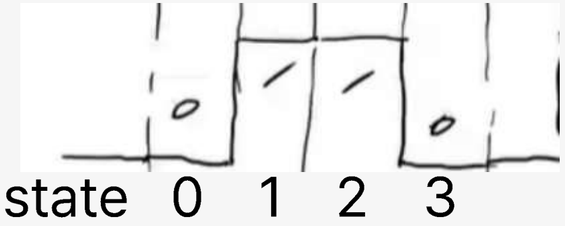


ASM 설계: READ Mode

READ Mode 추가 시 고려할 점

현재 설계된 FSM은 SCL 생성을 위해 하나의 STATE를 4개로 나누어 비효율적이다

READ Mode로 확장하게 된다면 고려해야할 STATE가 더욱 늘어나 상태 관리가 어렵고, 디버깅이 난해하다



-> 그렇다면 STATE 를 줄이자! 하지만 어떻게?...

I2C Master 내부에 SCL 을 생성하는 CLOCK Generator 를 구현하여

SCL 생성을 자동화 한다면 STATE를 줄일 수 있다

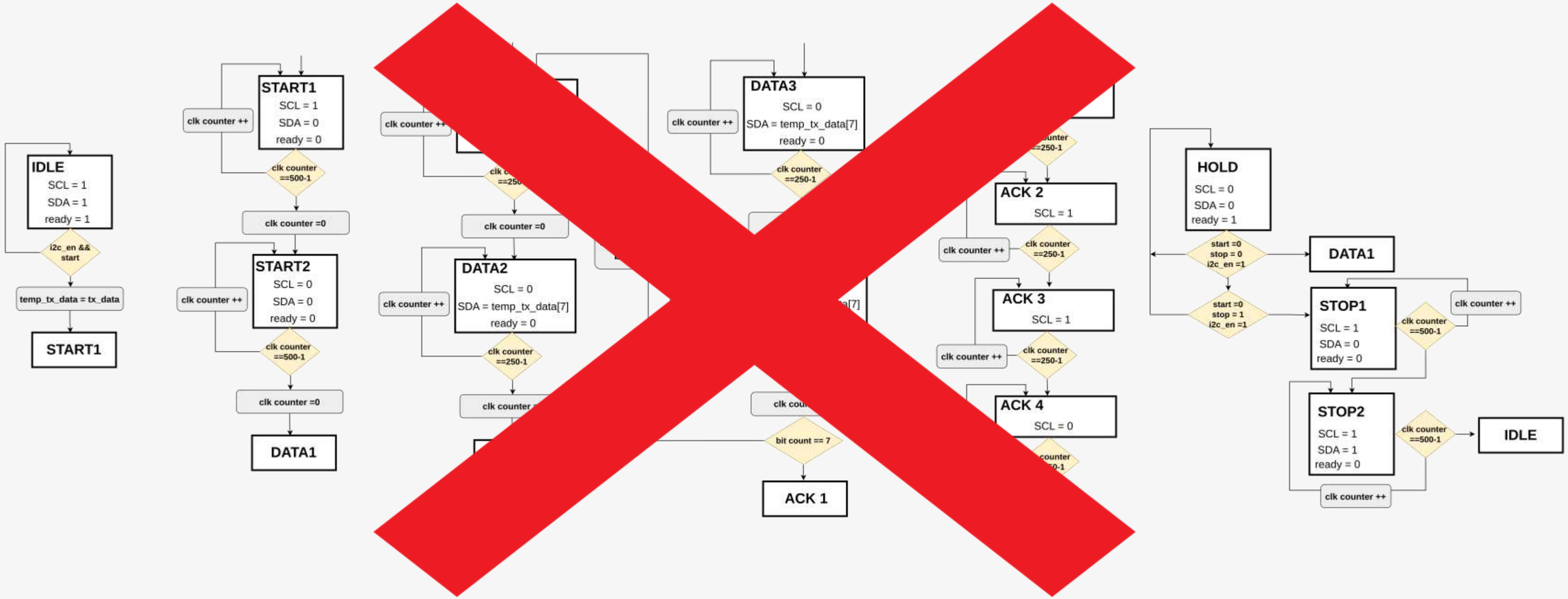
-> IDLE, HOLD, START, STOP 과 같이 주기적인 SCL(100kHz) 가 필요 없는 경우는?

en를 주어 clock 을 생성/비생성 하자

IDLE, HOLD, START, STOP 에서는 상태 분배를 통해 SCL 을 정의 하자!

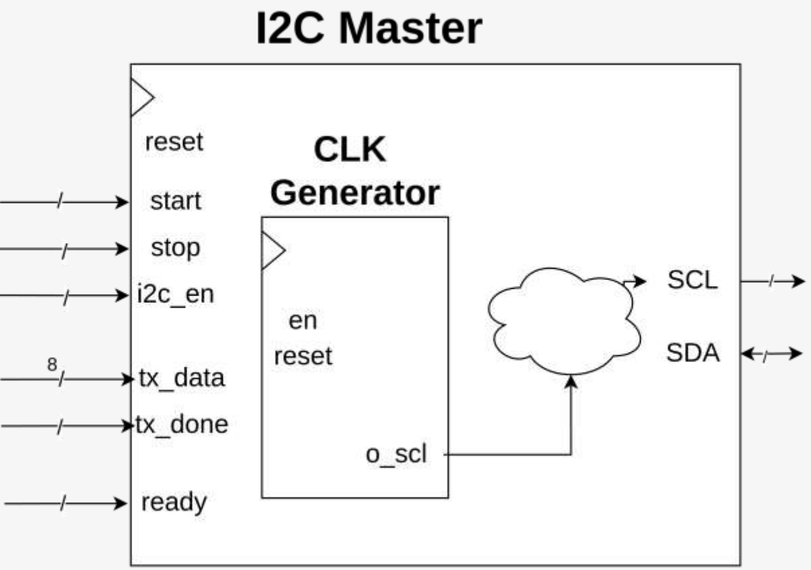
ASM 설계: READ Mode

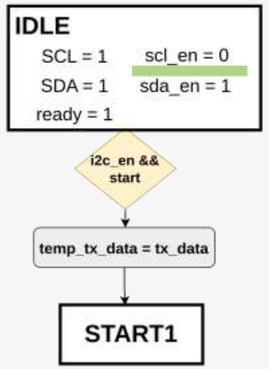
- READ Mode 추가 시 고려할 점

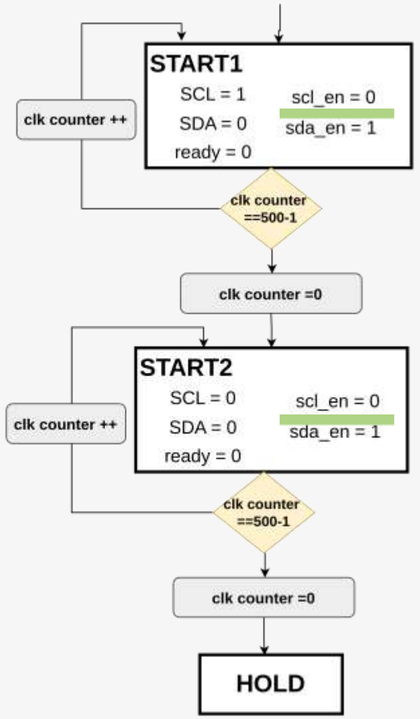


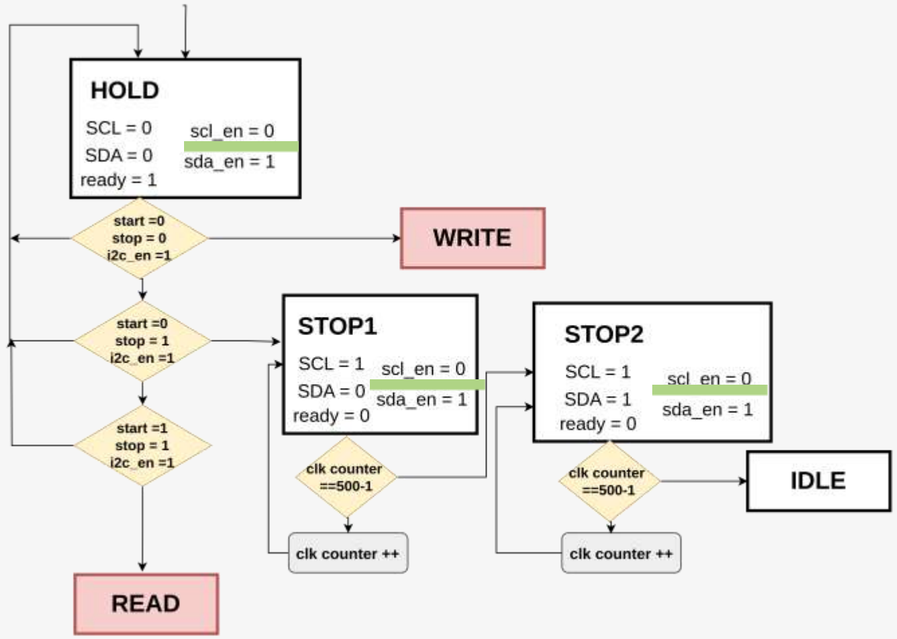
ASM 설계: READ Mode

- READ Mode 추가 시 고려할 점







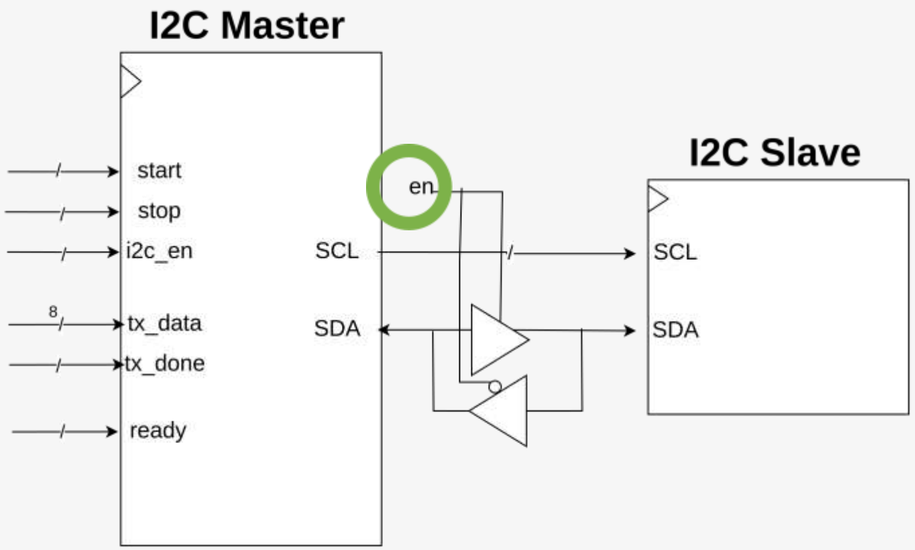


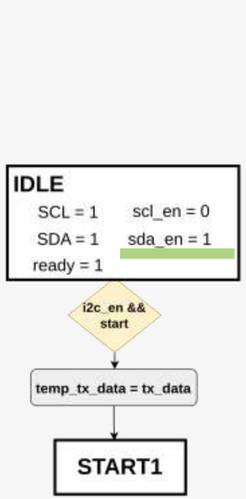
내부 클럭 생성 로직을 추가하여 STATE 줄이기

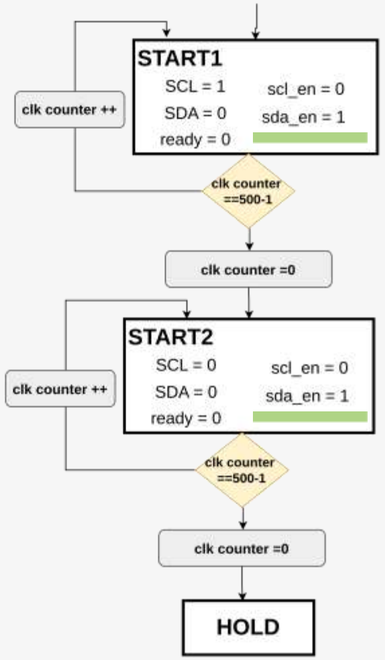
scl en 을 제어하여 외부로 나가는 SCL 정의

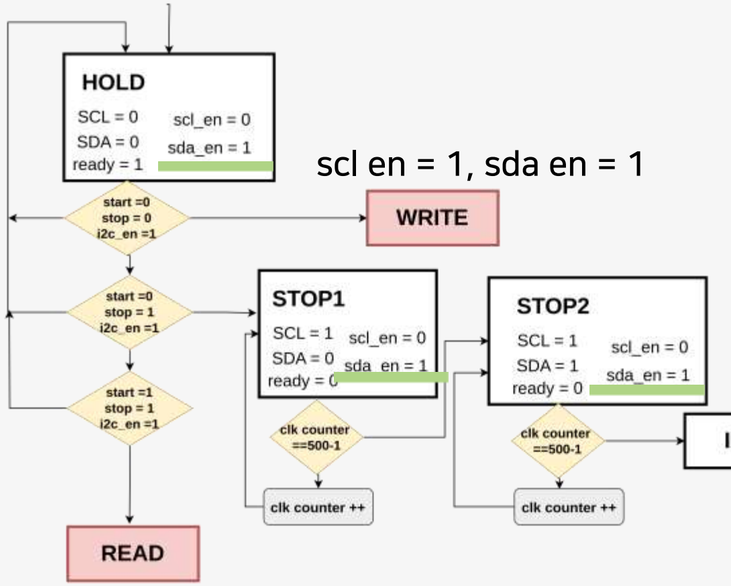
ASM 설계: READ Mode

- READ Mode 추가 시 고려할 점









scl en = 1, sda en = 1

scl en = 1, sda en = 0

SDA en 로직을 추가하여 SDA wire 입출력모드 제어

```
assign SCL = scl_en ? gen_scl : internal_scl;
assign SDA = sda_en ? o_data : 1'bz;
```

ASM 설계: READ Mode

- 결과

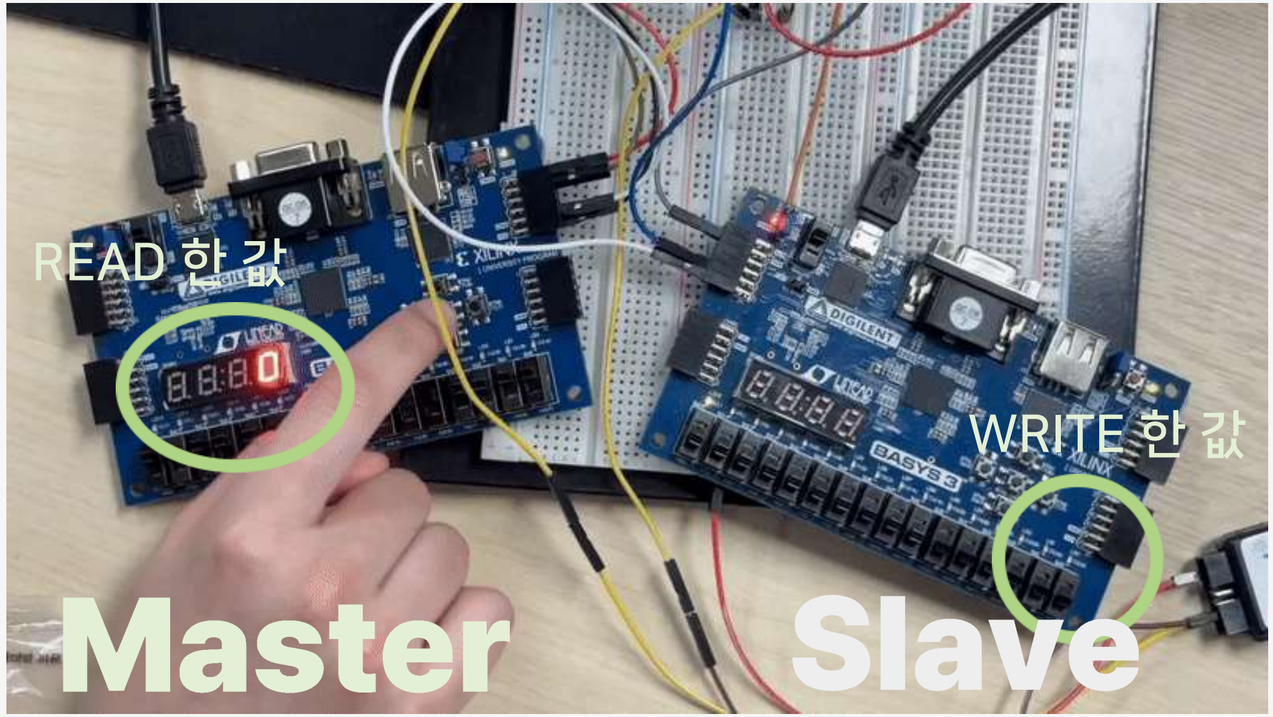
```
data I2C(I2C, 0x01);
```

```
void data_I2C(I2C_Typedef *I2Cx, uint32_t data){
    I2Cx->DATA = data;
}
```

```
typedef struct{
    volatile uint32_t CR;
    volatile uint32_t DATA;
    volatile uint32_t SR;
}I2C_Typedef;

#define I2C_BASEADDR 0x44A00000u

#define I2C ((I2C_Typedef *) I2C_BASEADDR)

void start_I2C(I2C_Typedef *I2Cx);
void stop_I2C(I2C_Typedef *I2Cx);
void data_I2C(I2C_Typedef *I2Cx, uint32_t data);
void set_I2C(I2C_Typedef *I2Cx);
void set_en(I2C_Typedef *I2Cx);


uint32_t is_ready(I2C_Typedef *I2Cx);
uint32_t is_txDone(I2C_Typedef *I2Cx);

int main()
{
    set_I2C(I2C);
    while(is_ready(I2C) == 0);

    I2C->CR = (1 << 2) | (1 << 0);  // start + en
    I2C -> CR &= ~(1<<0);
    while(is_ready(I2C) ==0);
    usleep(1);
    I2C -> DATA = 0xaa;
    I2C->CR = (I2C->CR & ~(1 << 2)) | (1 << 0);
    data_I2C(I2C, 0x01);
    while(is_ready(I2C)==0);
    I2C->CR &= ~(1 << 0);

    return 0;
}
```

XILINX VITIS

Launching SystemDebu...ystem: (36%)



Burst 전송이 가능하게 만들어 볼까? ...

WRITE/READ Burst Mode

Burst Mode 추가 시 고려할 점

"ACK Signal" 의 중요성

READ 모드에서는 데이터 수신 후 마스터가 슬레이브에게 "ACK"를 전송한다.

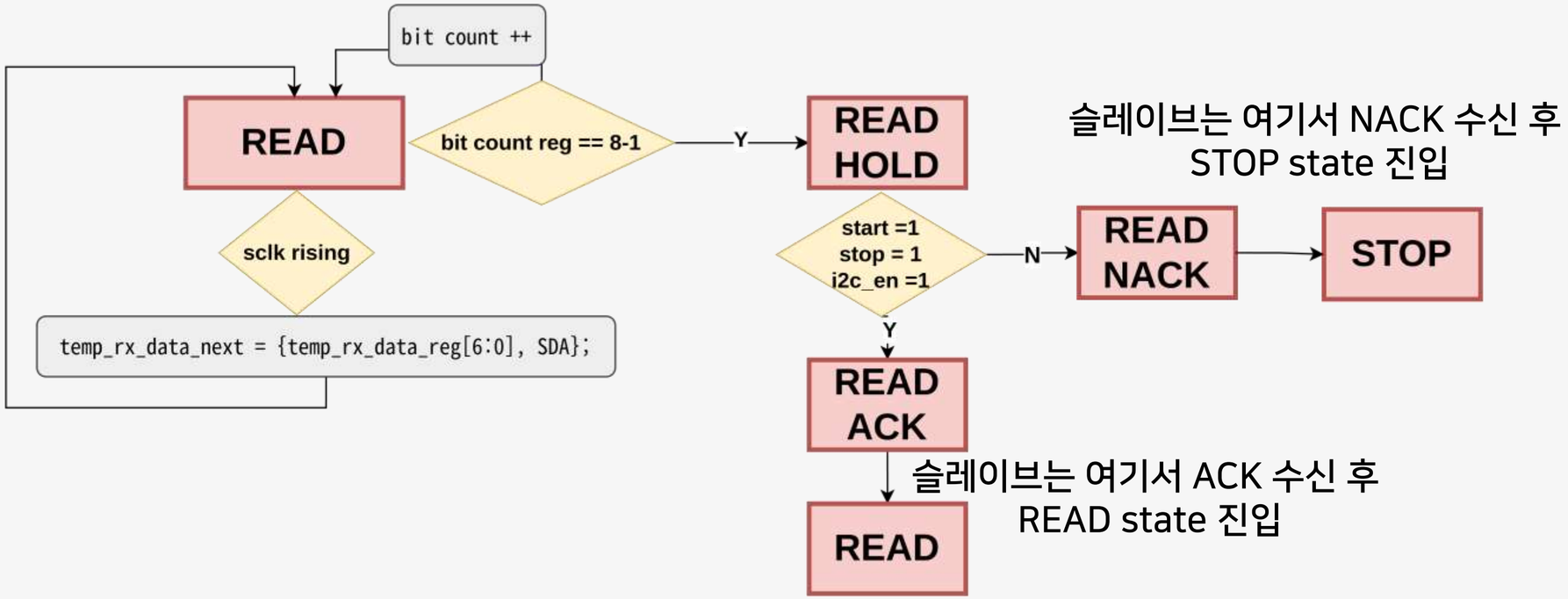
-> 계속 수신한다면 "ACK", 계속 수신을 종료한다면 "NACK"

-> 슬레이브 입장에서는 NACK를 받으면 STOP signal을 받을 준비를 하고,
마스터 입장에서는 NACK를 보낸 후 STOP signal을 보낸다.

-> 마스터가 바로 직전에 READ 상태 였다가 HOLD 상태에서 read signal을 받았다면 계속 전송인 경우이고,
슬레이브는 마스터로부터 받은 ACK를 통해 계속 수신함을 판단한다.

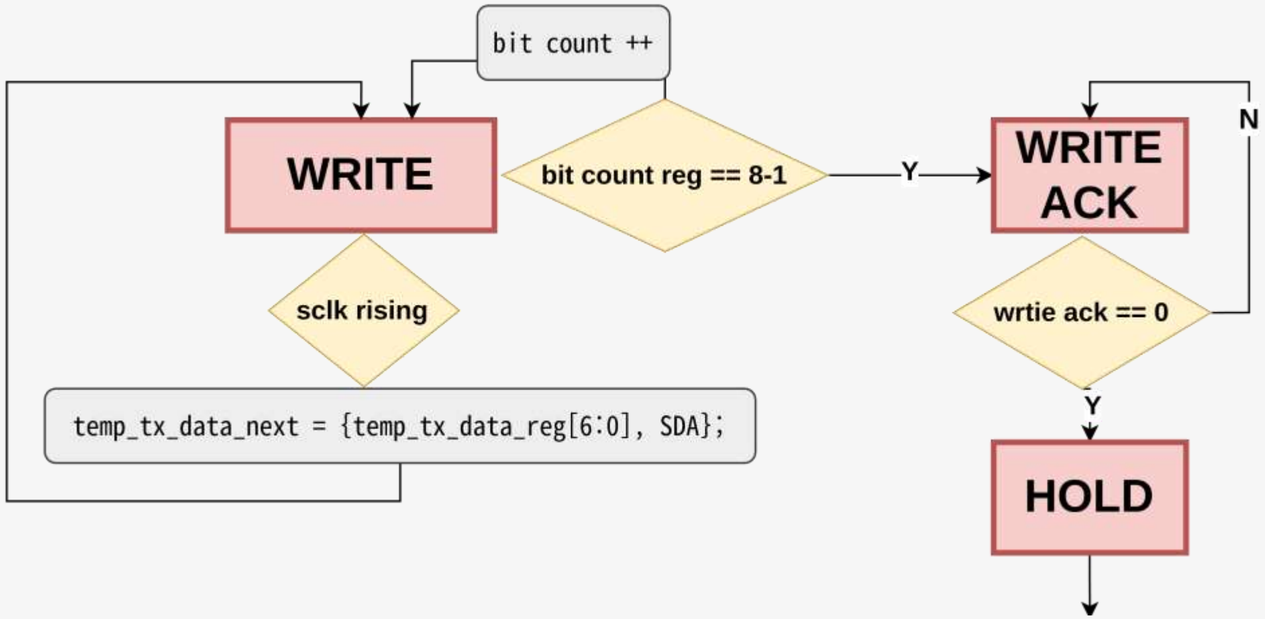
WRITE/READ Burst Mode

- READ ASM 수정



WRITE/READ Burst Mode

- WRITE ASM 수정



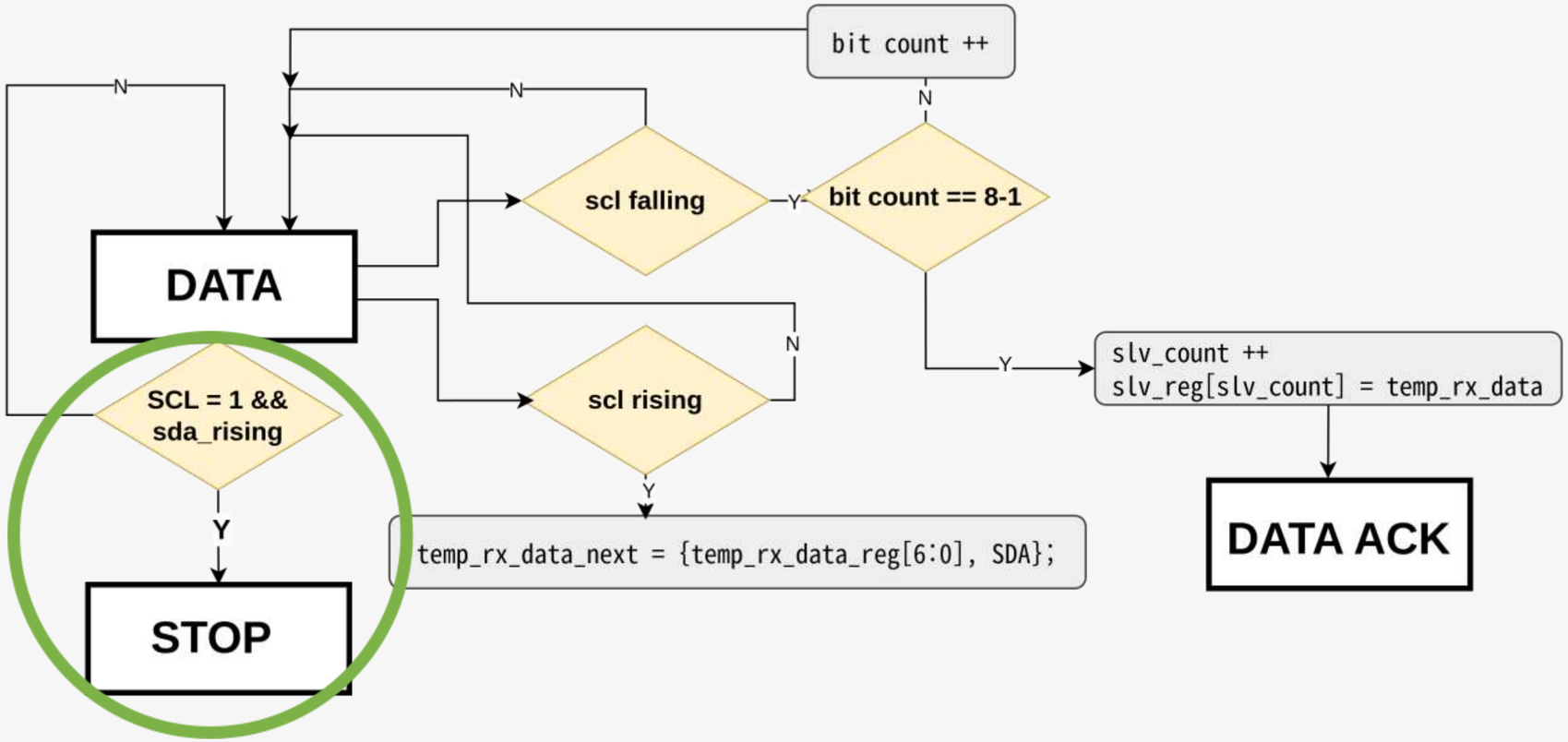
다음 모드는 HOLD 상태에서의
start, stop, i2c en의 조합으로 결정됨

슬레이브는 어떻게?

WRITE/READ Burst Mode

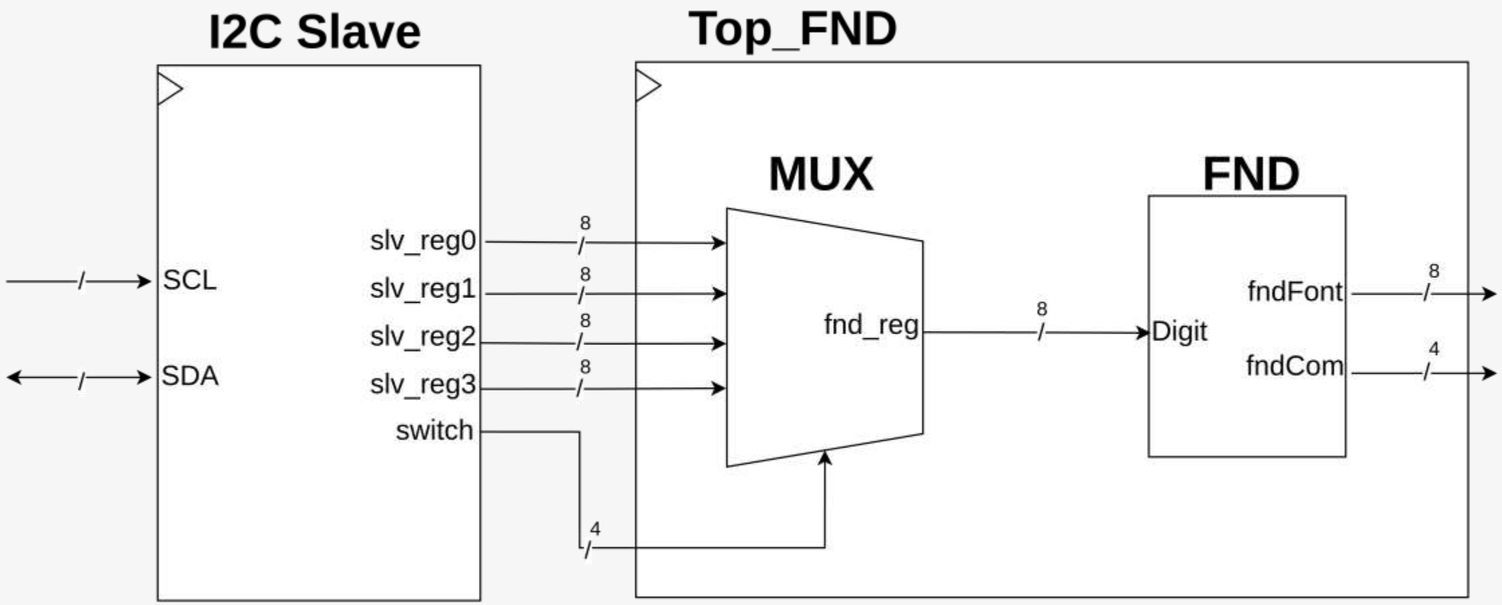
WRITE ASM 수정

슬레이브는 write mode 에서 stop signal을 감지하여 stop state로 진입할 수 있도록 퇴출구를 마련

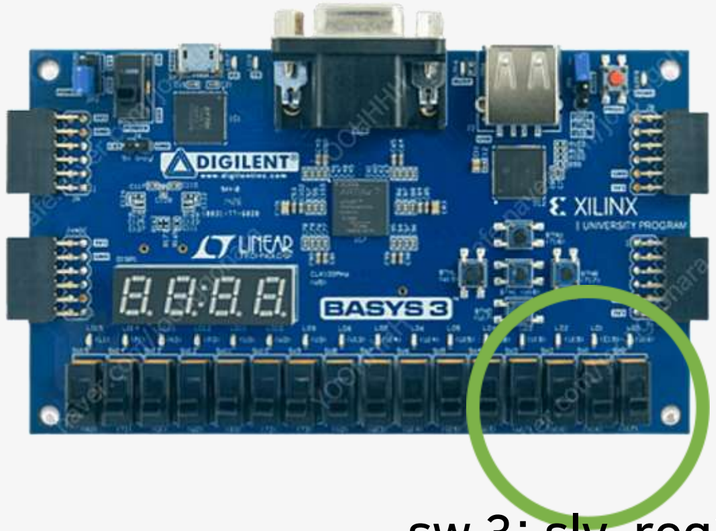


WRITE/READ Burst Mode

- I2C Slave 모듈 수정



SLAVE



sw 3: slv_reg3
sw2 : slv_reg2
sw1: slv_reg1
sw0: slv_reg0

SW 입력을 통해 I2C slave의 특정 레지스터 값을 FND 로 확인할 수 있도록 설계

Application 구현

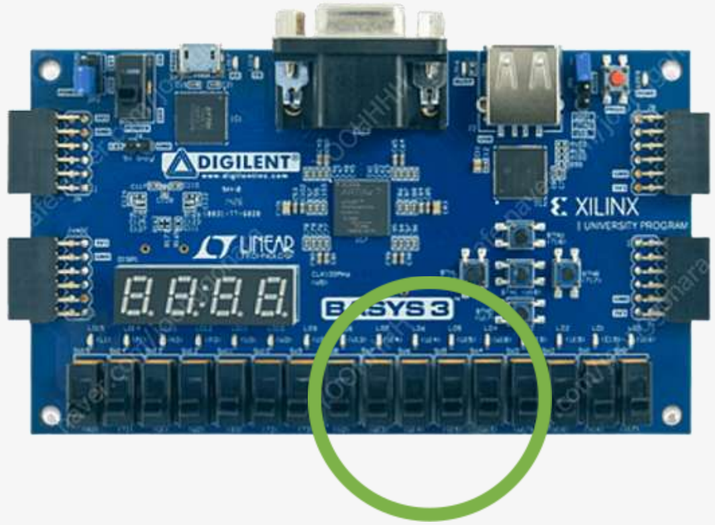
AXI4 I2C Intf Reigster Map

	bits	feature
0x00	[2:0]	CR (start, stop, i2c_en)
0x04	[7:0]	WDATA (tx_data)
0x08	[1:0]	SR(rx_done, tx_done)
0x0c	[7:0]	DATA1
0x10	[7:0]	DATA2
0x14	[7:0]	DATA3
0x18	[7:0]	DATA4

GPIO, FND IP ADD

- up button: data 전송
- down button: data 수신

받은 rx_data를 DATA1,2,3,4 register에 저장하고
SW 입력으로 register 저장 값을 FND로 READ 가능



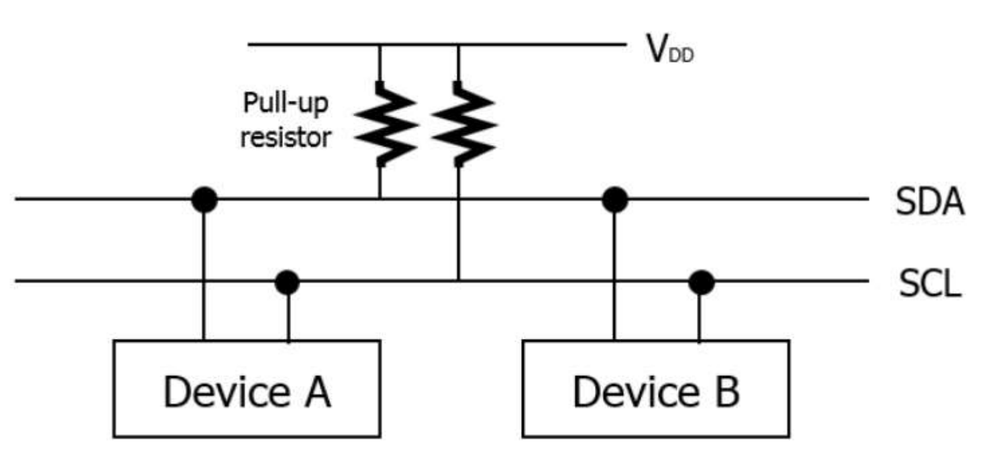
sw 7: DATA4
sw6 : DATA3
sw5: DATA2
sw4: DATA1

결과

트러블 슈팅

SLAVE 주소인 1010101에 접근 후, 0x01을 write(LSB=0) 한후 read하였는데

logic analyzer에서 확인된 값은 0x01이 아닌 0xff



address를 0xab가 아닌 0xaa를 주어야 했고,
 Master와 Slave 모두 SDA un-enable 될 시 풀업 되어 있어 SDA가 Logic High 된 것임을 확인

고찰

설계도만 보고 완전히 bottom부터 top까지 설계한 경험이 거의 처음이라 막막했지만

하나하나 설계에 성공하다 보니 점점 더 기능 확장에 욕심이 생겨

주어진 시간을 설계에 극한으로 할애했습니다.

WRITE 모드에 성공하다 보니 READ 모드 설계에 욕심이 생겼고

이어 READ 도 성공하니 연속 송수신 기능도 구현하고자하는 욕심이 생겼습니다.

이렇게 연속 송수신 기능까지 구현하고나니

애플리케이션을 어떻게 구현할지에 대한 주체적인 구상을 하게 되었고

덕분에 완전한 bottom level 부터 top level까지의 설계를 경험할 수 있었습니다.

고찰

기능을 확장해 가며 초기 설계 구조를 효율적으로 개선하고

확장된 기능을 유연하게 적용 가능하도록 모듈을 자유롭게 변형하는 능력을 기를 수 있었습니다.

이전까지는 단순 블록 단위의 설계 경험에 머물렀지만

이번 프로젝트를 통해 기능 확장, 시스템 통합, 임베디드 제어까지 경험하며

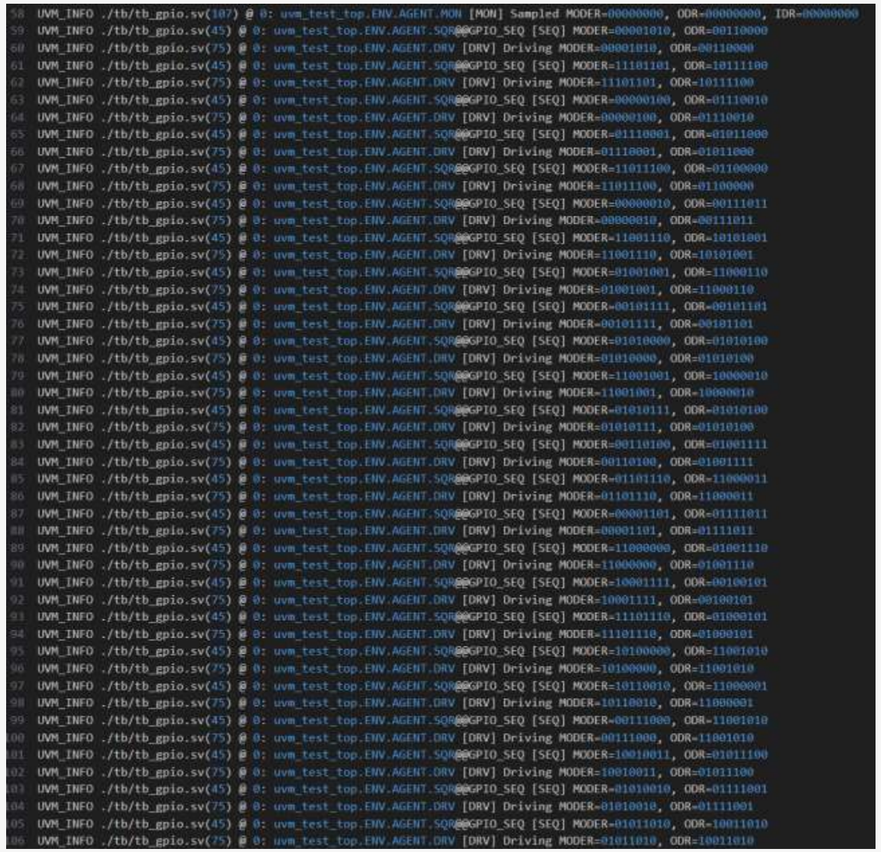
보다 시스템적인 시야에서 회로를 바라보는 시야를 갖게 되었고

자율적으로 원하는 기능을 설계하고 구현할 수 있다는 자신감을 얻게 되었습니다.

발표 들어주셔서 감사합니다.

부록 : UVM GPIO 검증

```
** Report counts by severity
UVM_INFO :  206
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[DRV]   100
[MON]     1
[RNTST]     1
[SEQ]   100
[TEST_DONE]     1
[UVM/RELNOTES]     1
[UVM/REPORT/CATCHER]     1
[UVMTOP]     1

$finish called from file "/tools/synopsys/vcs/W-2024.09-SP1/etc/uvm-1.2/base/uvm_root.svh", line 527.
$finish at simulation time                   0
                 V C S   S i m u l a t i o n   R e p o r t
Time: 000 ps
CPU Time:      0.230 seconds;       Data structure size:   0.6Mb
Sun May 25 15:18:13 2025

```



synopsys vcs를 이용한 uvm gpio 검증 결과