

PROJECT

RISC-V AMBA Peripheral

유지훈



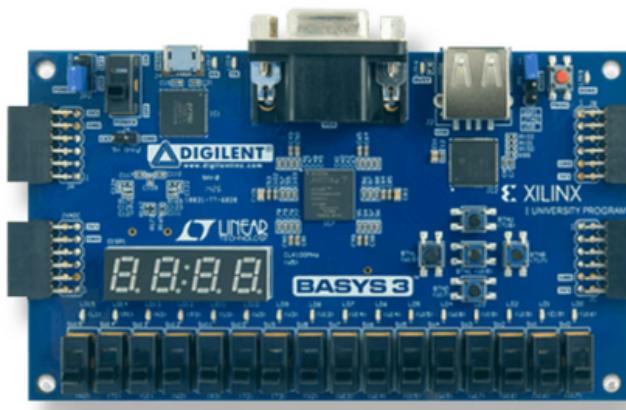
대한상공회의소
서울기술교육센터

목차

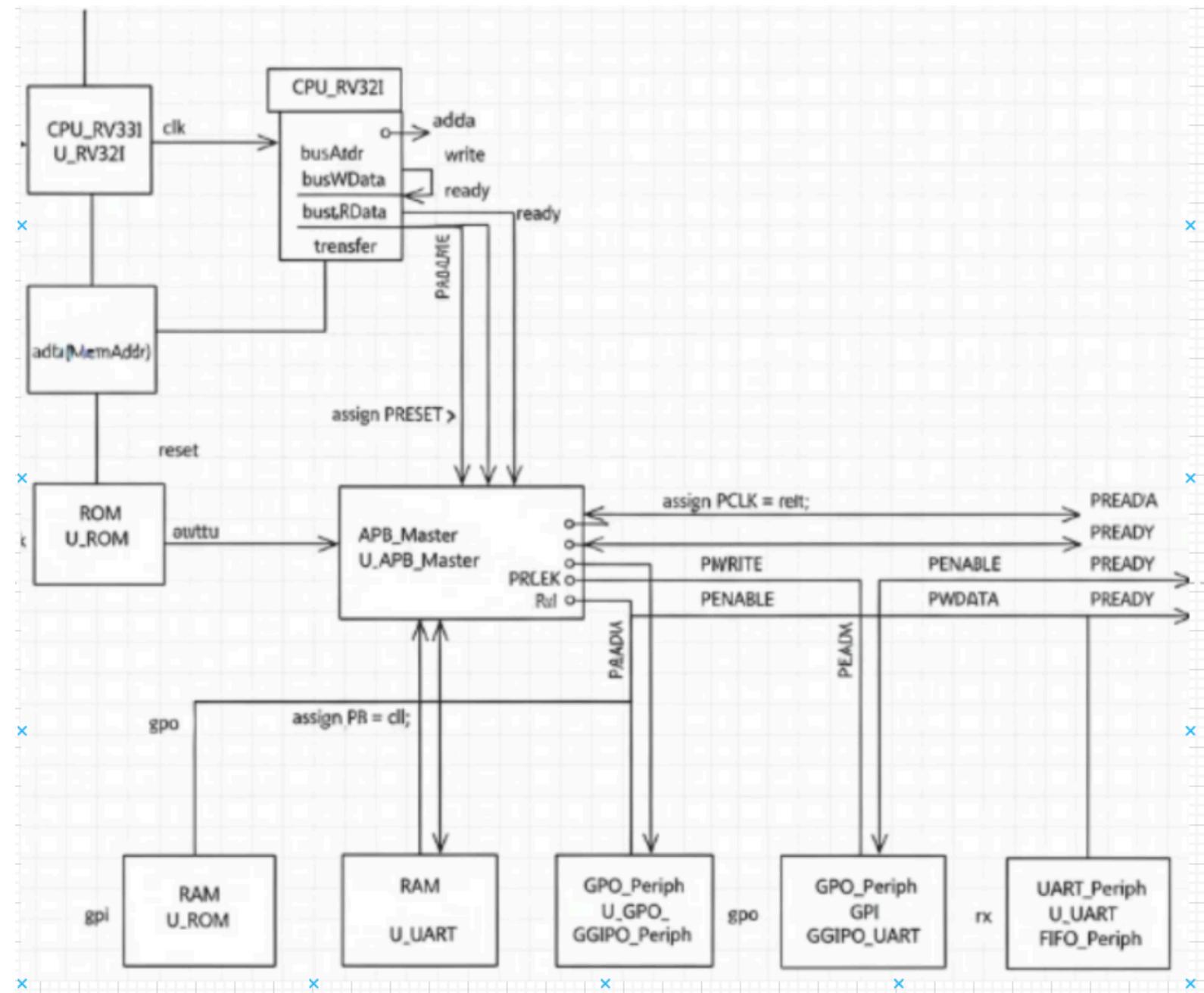
- | | | | |
|----|----------------|----|-----------------|
| 01 | 프로젝트 개요 | 05 | AMB Bus |
| 02 | MCU | 06 | UART Peripheral |
| 03 | Multi-cycle | 07 | C Application |
| 04 | 각 Type 별 동작 확인 | 08 | 고찰 |

01 프로젝트 개요

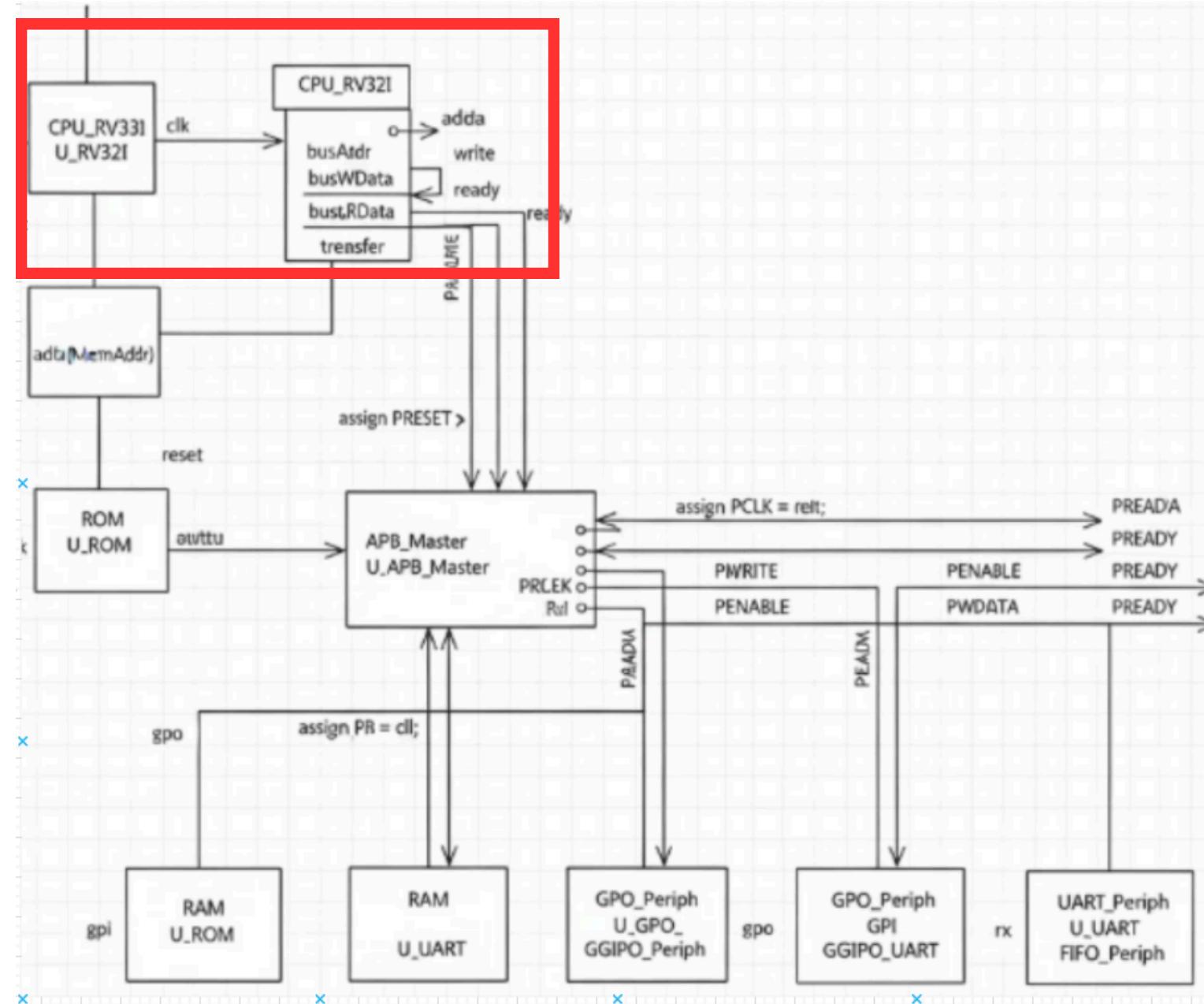
- AMBA APB Bus 기반의 UART-FIFO 통신 모듈을 설계 및 검증
- RISC-V 명령어 기반 RV32I CPU를 자체 구현하고 APB Read/Write 테스트 수행
- UVM-Lite 환경에서 TX/RX 트랜잭션을 랜덤 시나리오로 검증
- RV32I CPU와 UART를 연결하여 C 언어 기반 애플리케이션 정상 동작 확인



02 MCU



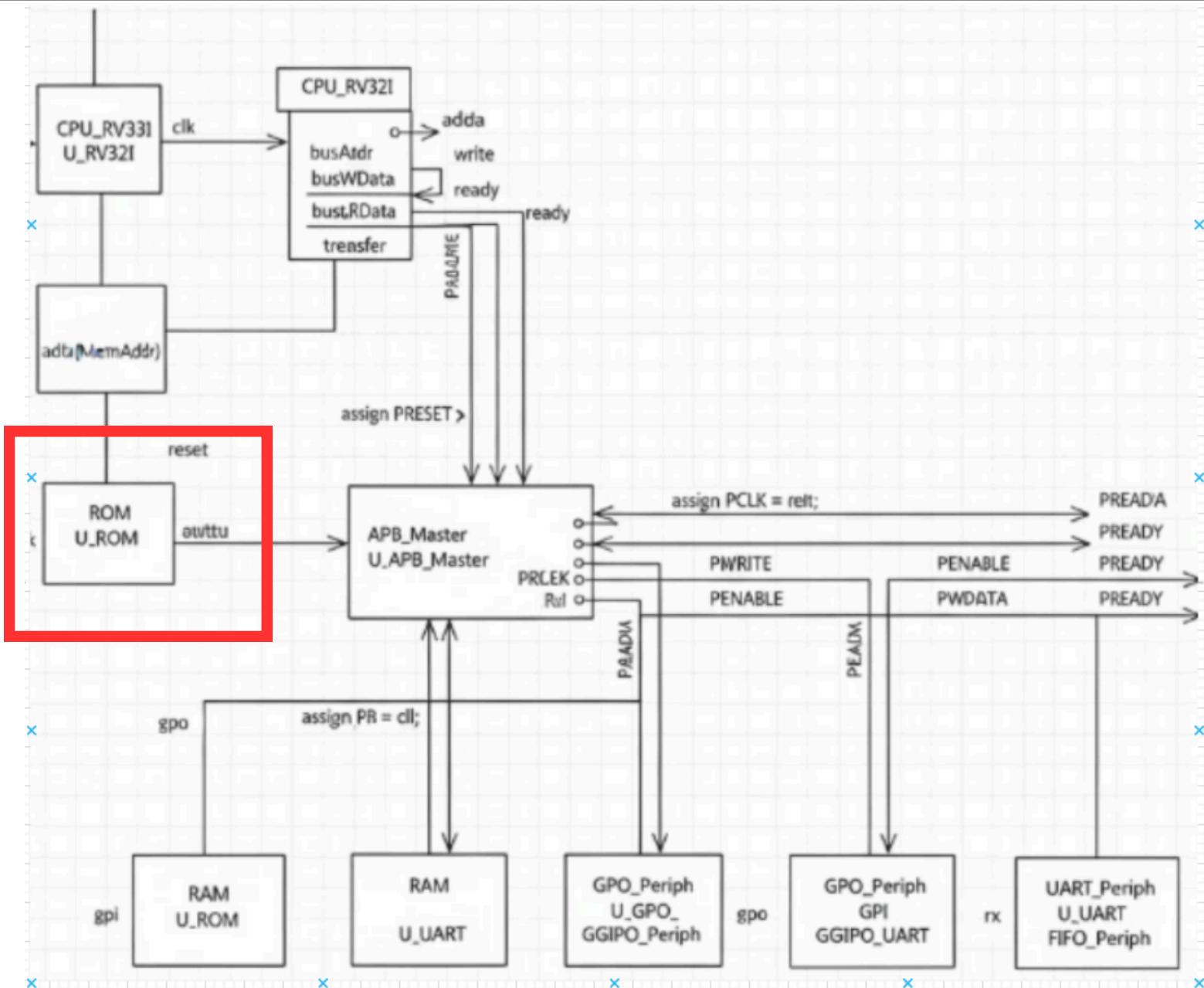
02 MCU



CPU_RV32I (U_RV32I)

- RISC-V 명령어 세트를 기반으로 하는 32비트 단일 사이클 CPU.
- 내부적으로 busAddr, busWData, busRData, write, transfer, ready 신호를 통해 APB Master와 통신
- 명령어는 ROM에서 읽고, 데이터 입출력은 APB를 통해 수행

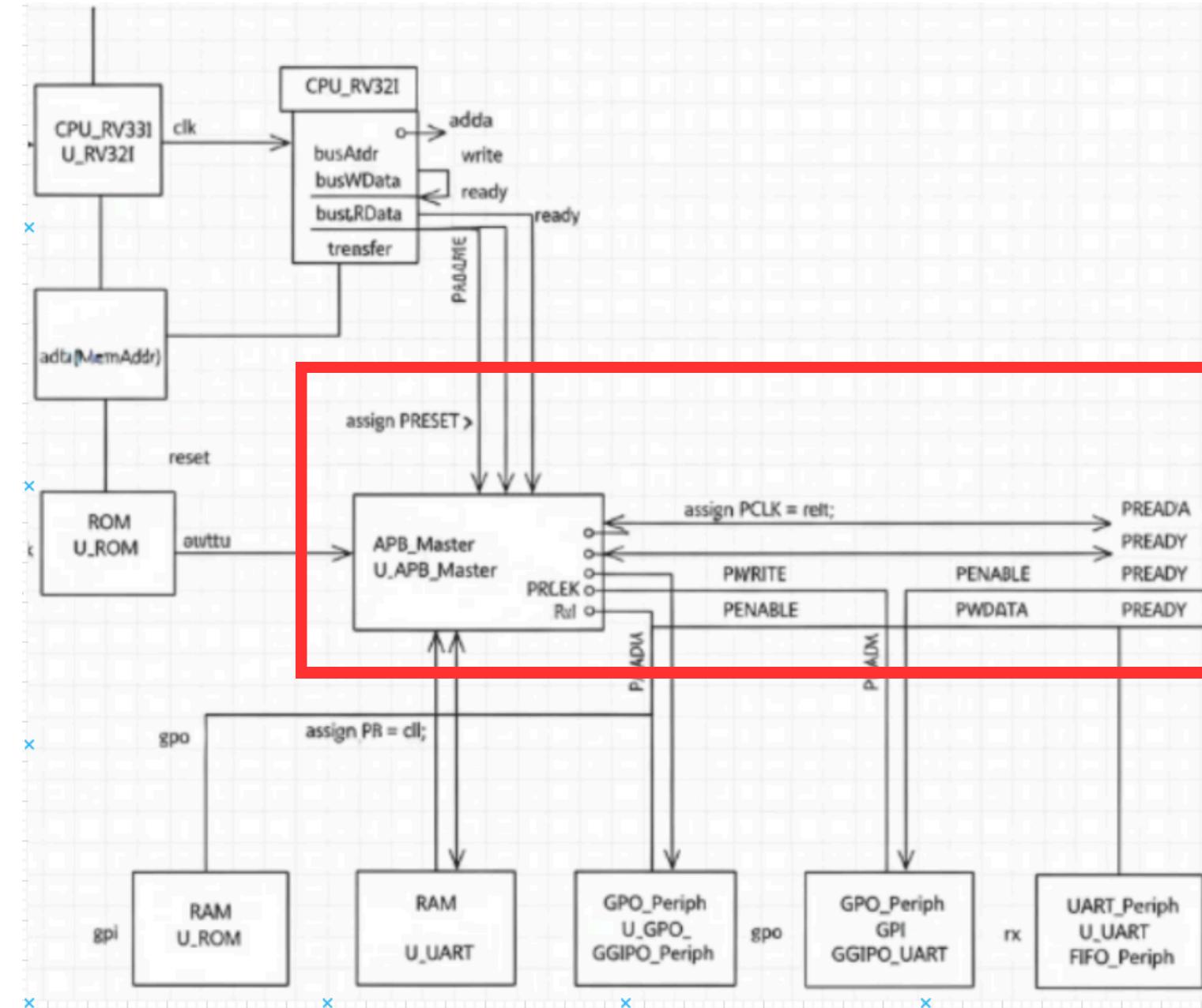
02 MCU



ROM (U_ROM)

- 프로그램(펌웨어) 코드가 저장된 영역
- CPU가 부팅 시 여기서 명령어를 Fetch하여 실행한다.
- 주소 신호(instrMemAddr)를 받아 명령어를 출력한다

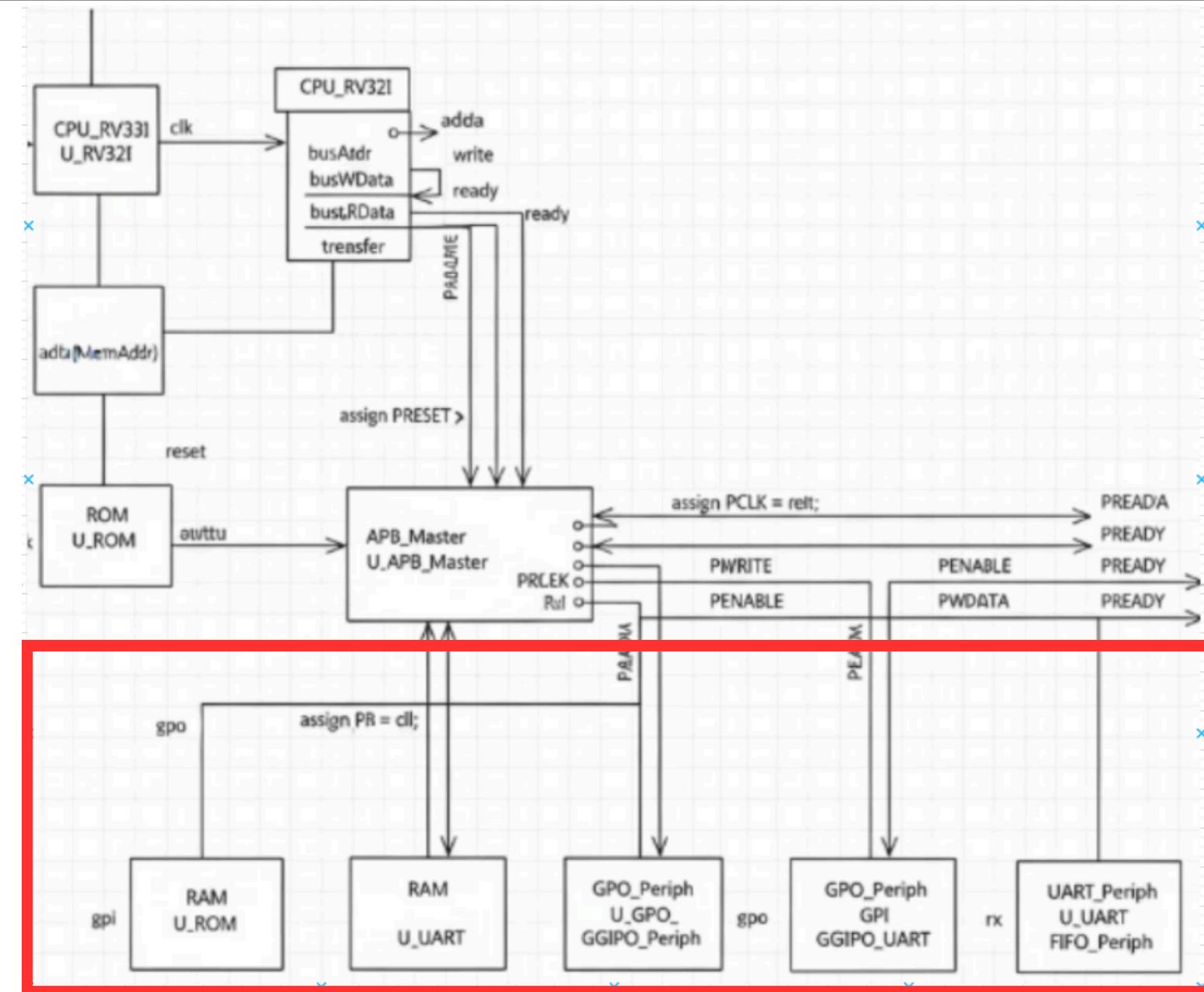
02 MCU



APB Master (U_APB_Master)

- CPU와 주변장치 사이의 중앙 버스 컨트롤러.
- CPU의 제어 신호(addr, write, transfer)를 해석
- APB 신호(PADDR, PWDATA, PWRITE, PSEL, PENABLE)로 변환
- 각 슬레이브로부터의 응답(PRDATA, PREADY)을 받아 CPU로 전달

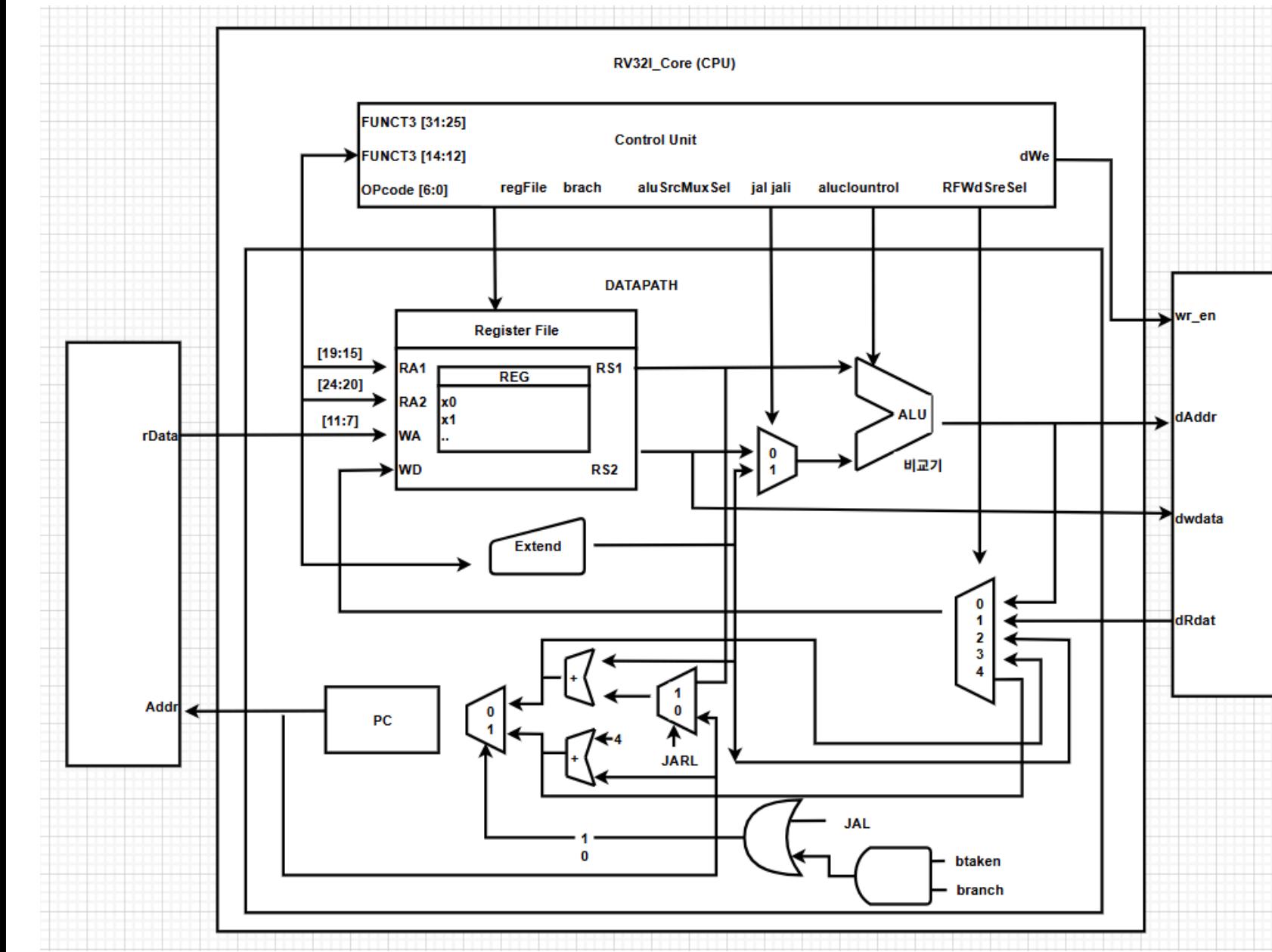
02 MCU



APB Slave Peripherals

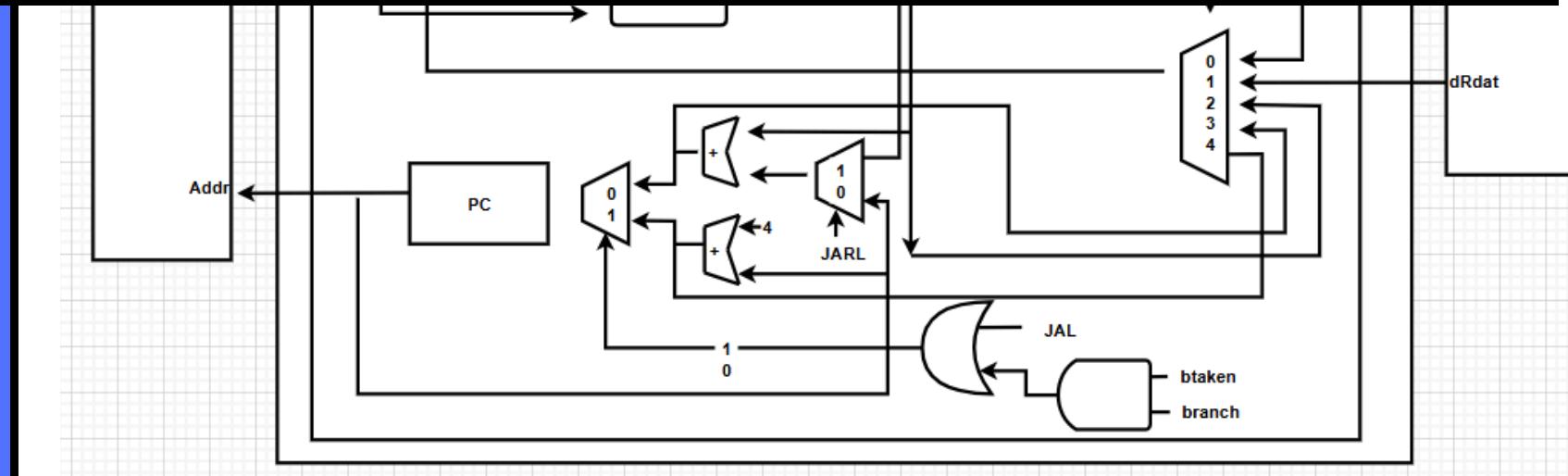
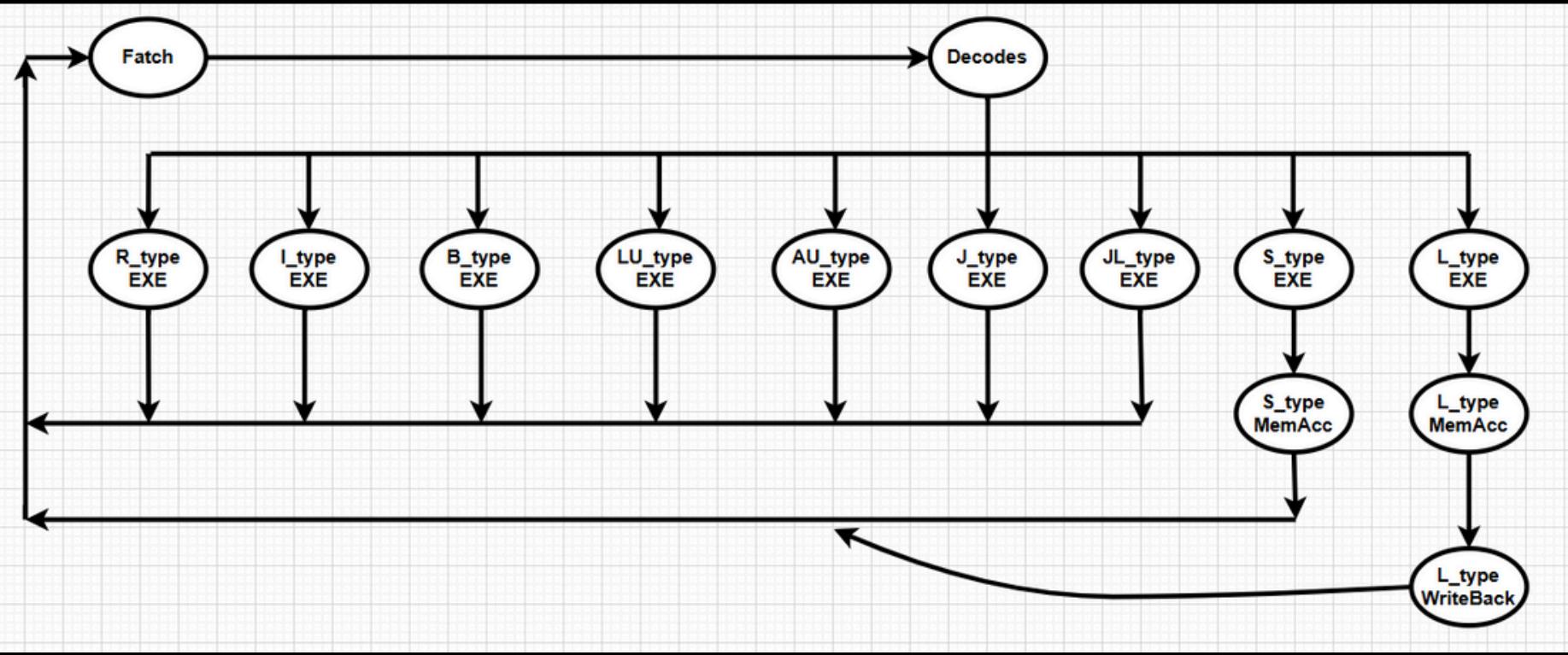
- APB Master에 여러 주변장치가 연결된 형태.
- 각 장치는 동일한 APB 인터페이스 구조
(PADDR, PWDATA, PRDATA, PWRITE, PREADY)

03 Multi-cycle 구조



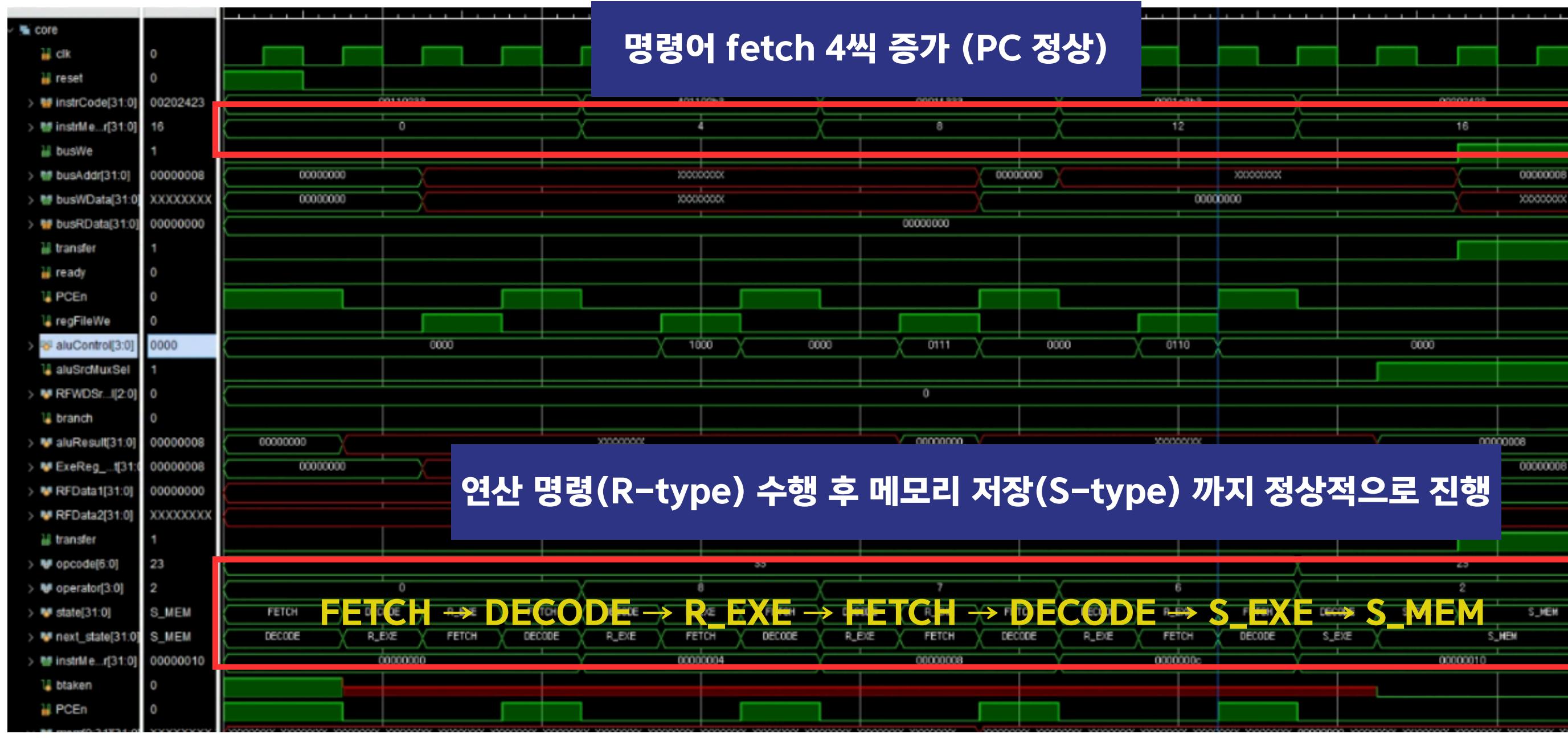
- **Program Counter (PC):** 현재 명령어 주소 저장
- **Register File:** rs1, rs2 읽기 / rd 쓰기
- **ALU:** 산술 및 논리 연산 수행
- **Imm Extend:** 즉시값을 Sign-extend
- **MUX:** ALU 입력, WriteBack 경로 제어
- **Branch Comparator:** BEQ, BNE 등 비교 수행
- **Control Unit:** FSM을 기반으로 제어 신호 생성

03 Multi-cycle 설명



단계	명칭	주요 동작	제어 신호
S0: IF (Fetch)	명령어 인출	PC가 가리키는 주소로 ROM에서 명령어를 읽고 PC += 4	PCEn=1, regFileWe=0
S1: ID (Decode)	명령어 해석	Opcode, funct3, funct7을 해석하고 레지스터 주소 (rs1, rs2, rd) 추출	aluSrcMuxSel, RFwdSrcSel 결정
S2: EXE (Execute)	ALU 연산 수행	R/I/S/B/U/J 타입에 따라 ALU 연산 또는 분기/주소 계산	aluControl, branch, jal, jalr
S3: MEM (Memory Access)	데이터 메모리 접근	Store(S-type): Data Write Load(L-type): Data Read	busWe=1, strb, PCEn=0
S4: WB (Write Back)	결과 저장	ALU 연산 또는 Load 데이터 결과를 rd에 저장	regFileWe=1, RFwdSrcSel=ALU/MEM

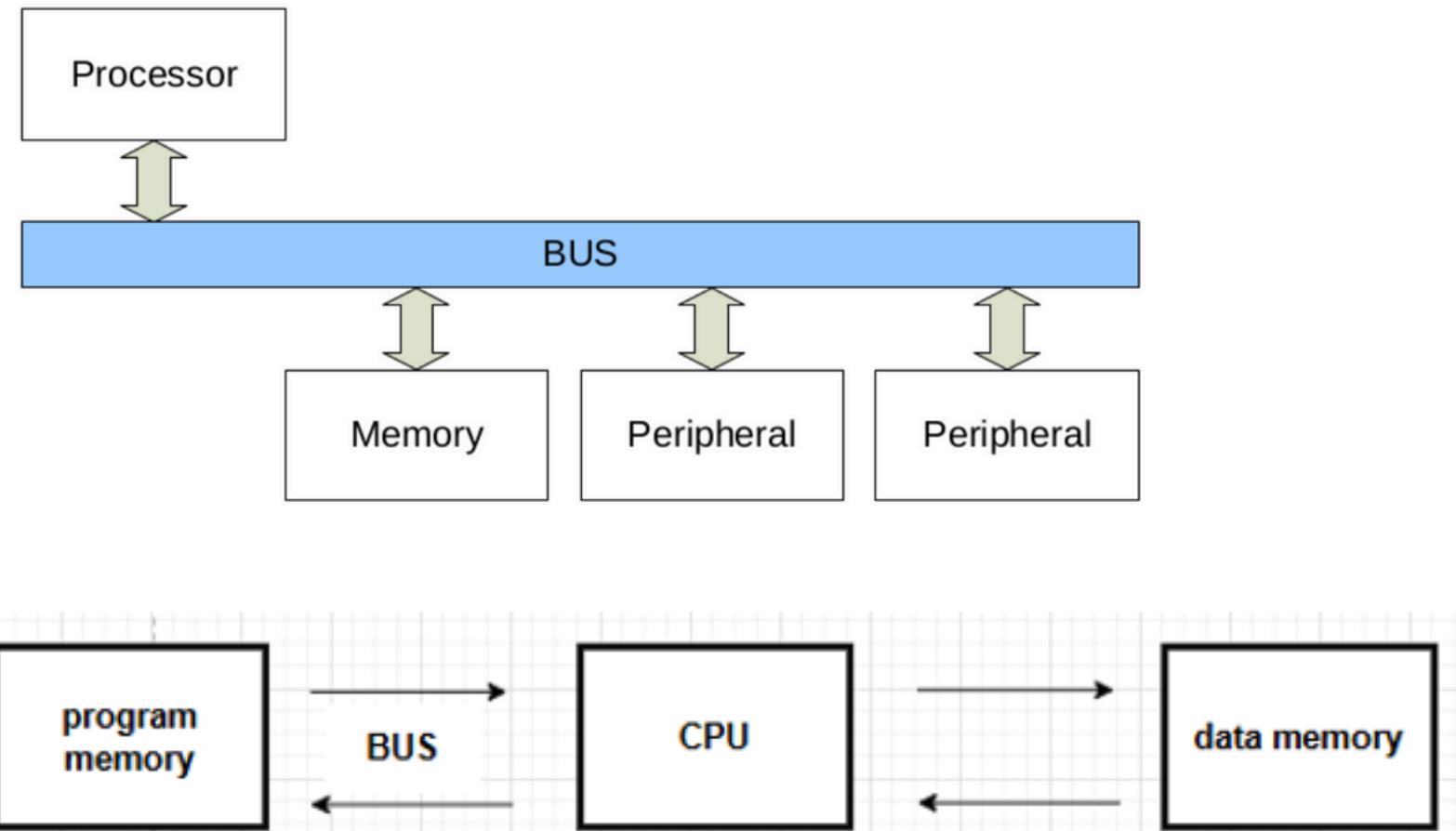
04 각 Type 별 동작 확인



05 AMB Bus 설계 목적

목적	설명
1. 단순한 제어	주소, 데이터, 읽기/쓰기 신호만으로 통신
2. 저전력 동작	불필요한 신호 토글 최소화
3. 저속 주변장치 적합	UART, Timer, GPIO, ADC처럼 느린 장치에 최적화
4. 일관된 인터페이스 제공	다양한 IP를 동일한 APB 프로토콜로 쉽게 연결 가능
5. 설계 재사용성 향상	표준화된 인터페이스로 SoC 통합 시 편리함

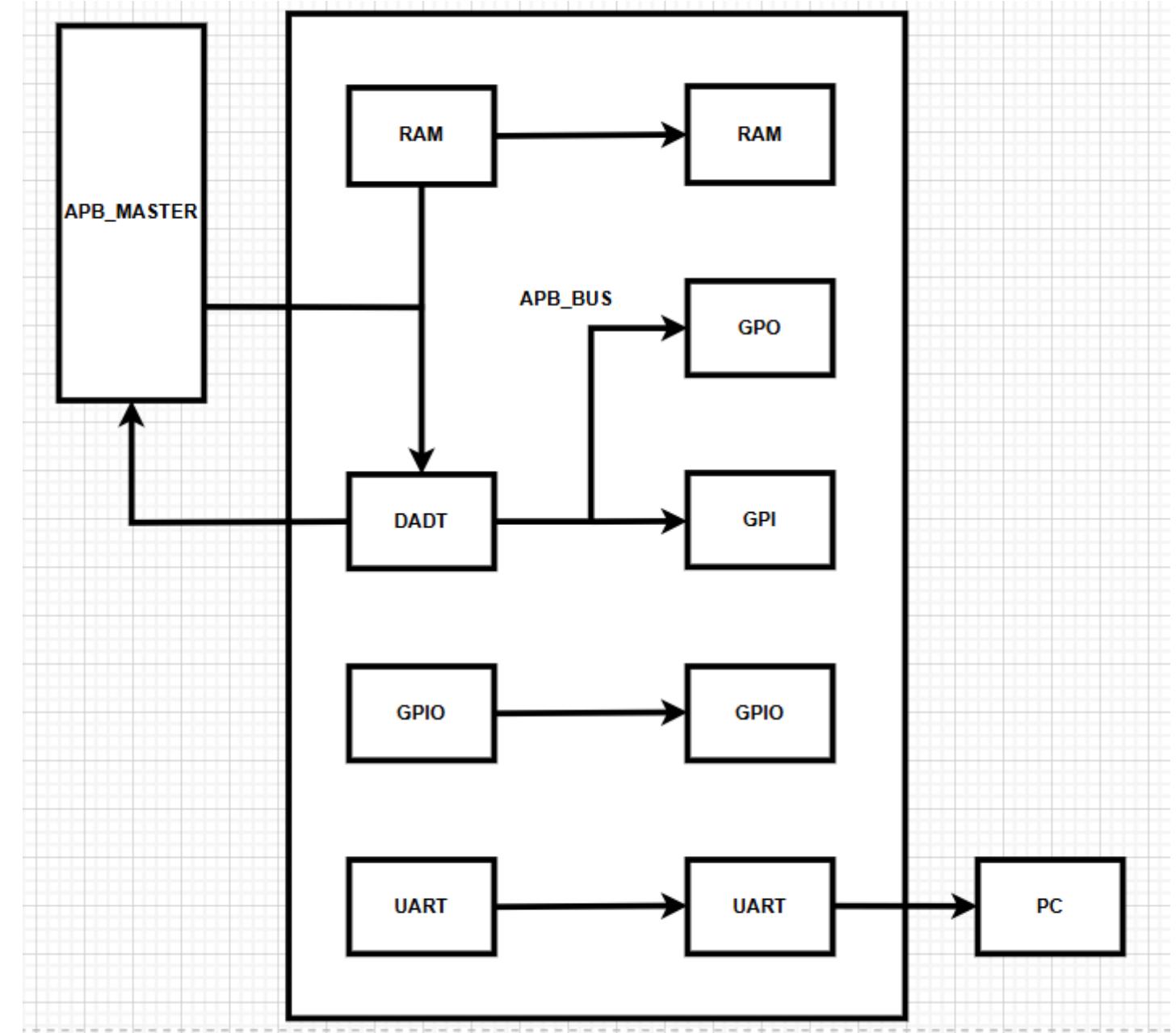
05 AMB Bus 설계 목적



구분	하버드 버스	AMBA 버스
정의	CPU 내부 구조 개념	SoC 내부 연결 표준
목적	명령어/데이터 병렬 접근	주변장치 간 통신 표준화
버스 종류	Instruction Bus / Data Bus	APB, AHB, AXI 등
사용 위치	CPU 코어 내부	CPU ↔ 주변장치 간
설계 주체	CPU 아키텍처	ARM사 정의 프로토콜
프로토콜 존재	X (단순 신호 연결)	O (PADDR, PWRITE, PSEL, PENABLE,

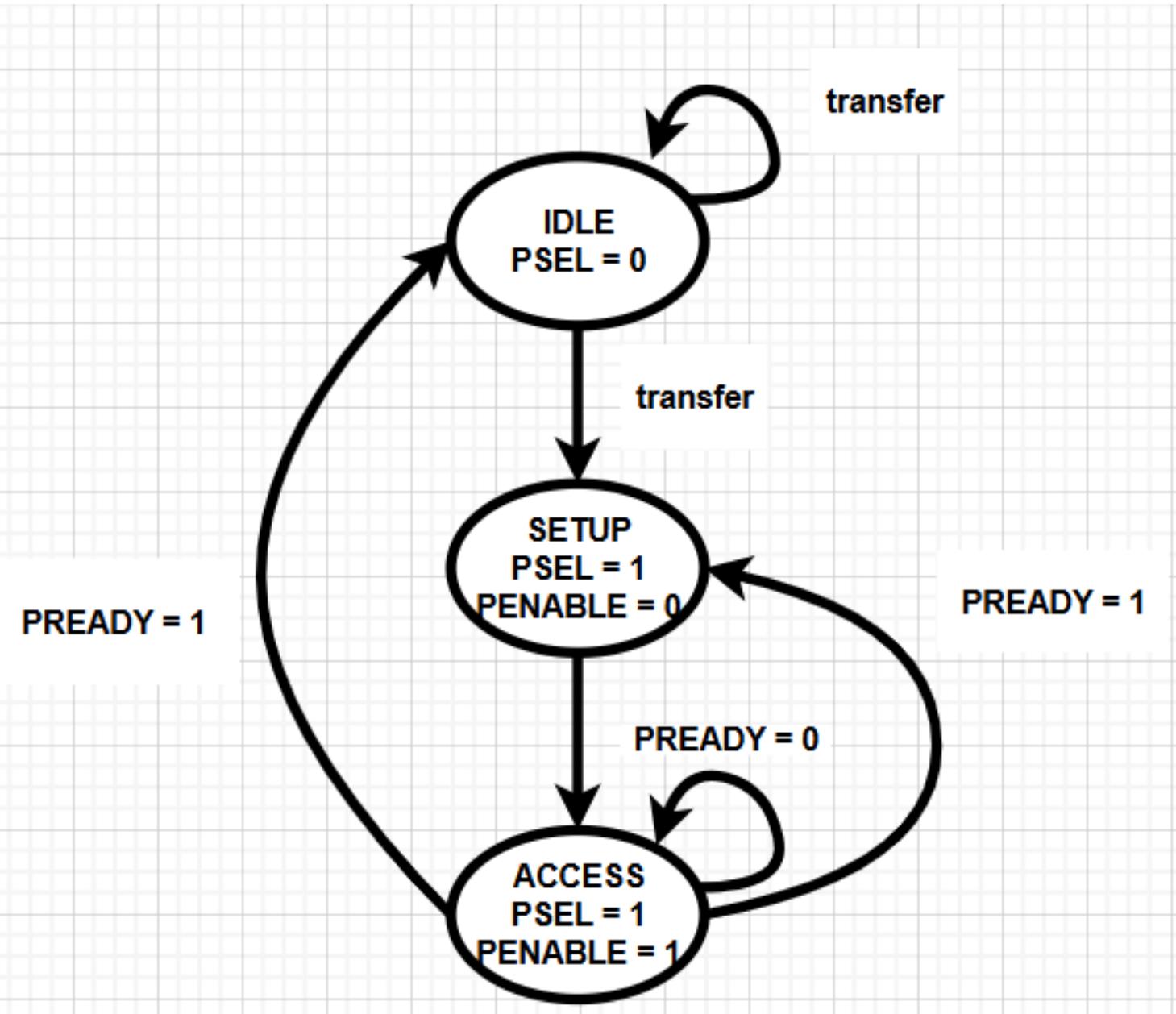
- 내부적으로 하버드 구조를 기반으로 명령어와 데이터 버스를 분리하여 병렬 처리를 수행하며,
- 외부 주변장치 접근은 ARM AMBA 표준 중 하나인 APB 버스를 통해 통신하는 구조

05 APB Bus 구조



APB Master는 CPU로부터 주소, 데이터, 제어신호를 입력받아
내부 디코더(Address Decoder + Data Router)를 통해 각
Slave (RAM, GPO, GPI, UART 등)에 접근
각 Slave는 PADDR, PWDATA, PWRITE, PSEL,
PENABLE 신호를 통해
Master와 통신하며, PRDATA와 PREADY로 응답을 반환

05 APB Bus FSM



IDLE: 버스가 대기 상태이며, PSEL=0으로 주변장치 비활성화

SETUP: 전송 준비 단계로, Master가 PSEL=1, PENABLE=0
상태에서 주소와 데이터를 설정

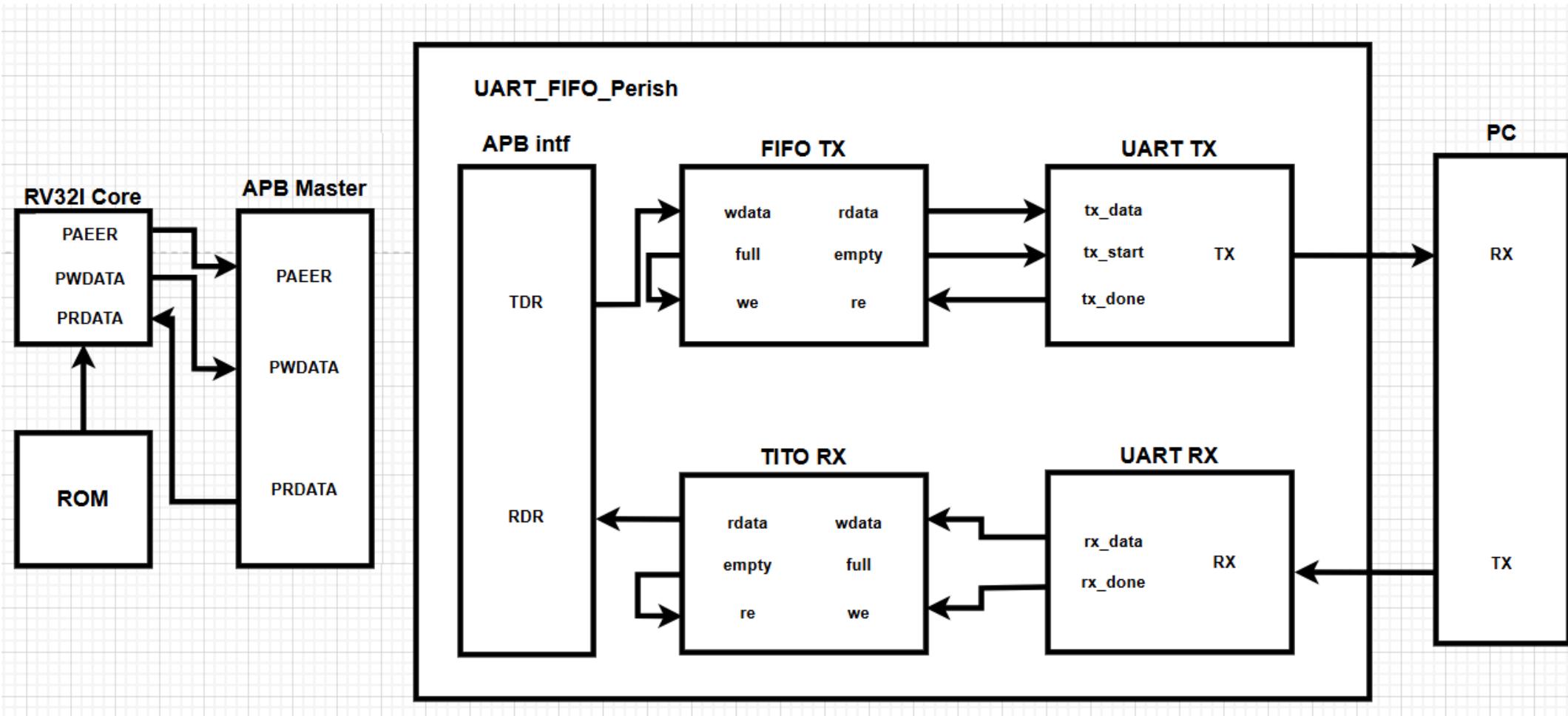
ACCESS: 실제 데이터 전송 단계로, PENABLE=1이 되어
Slave와 데이터 교환 수행

전송이 완료되면 PREADY=1이 되어 IDLE 상태로 복귀

장점:

APB Bus는 복잡한 파이프라인이 없이 안정적인 데이터 전송이
가능

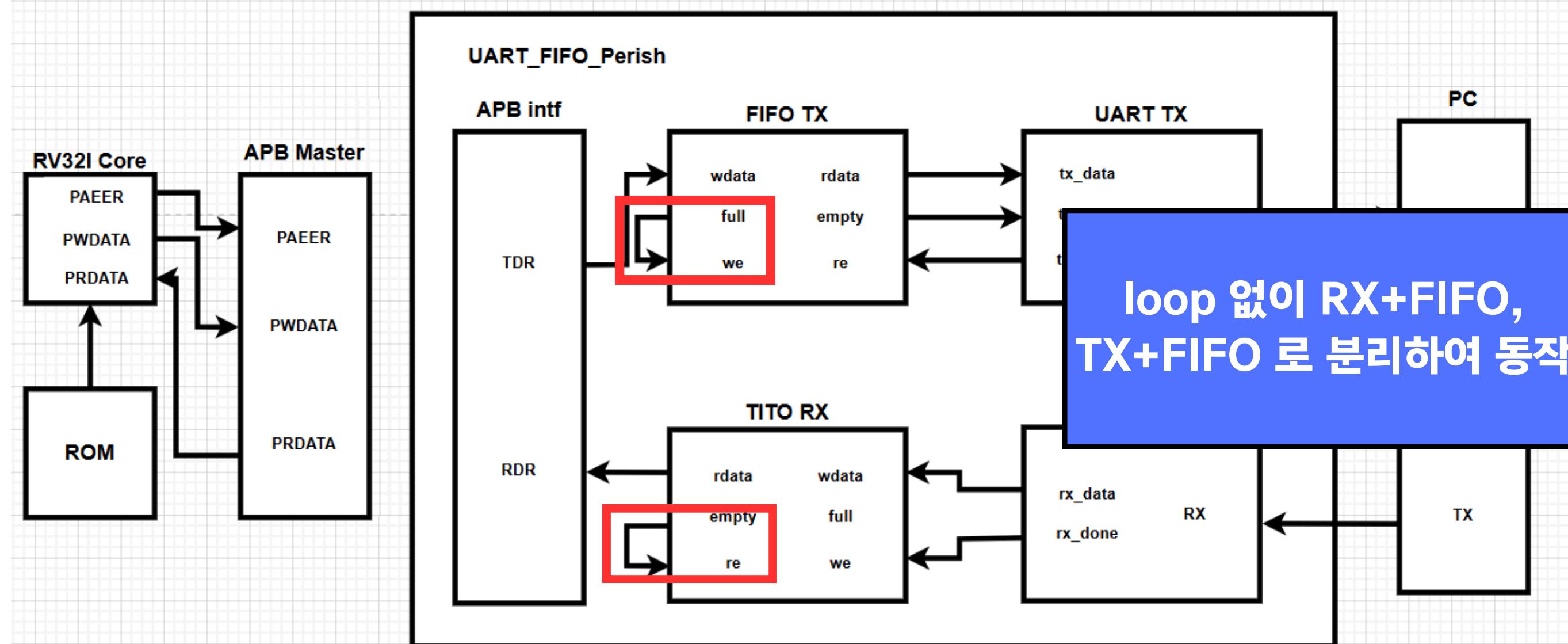
06 UART Peripheral



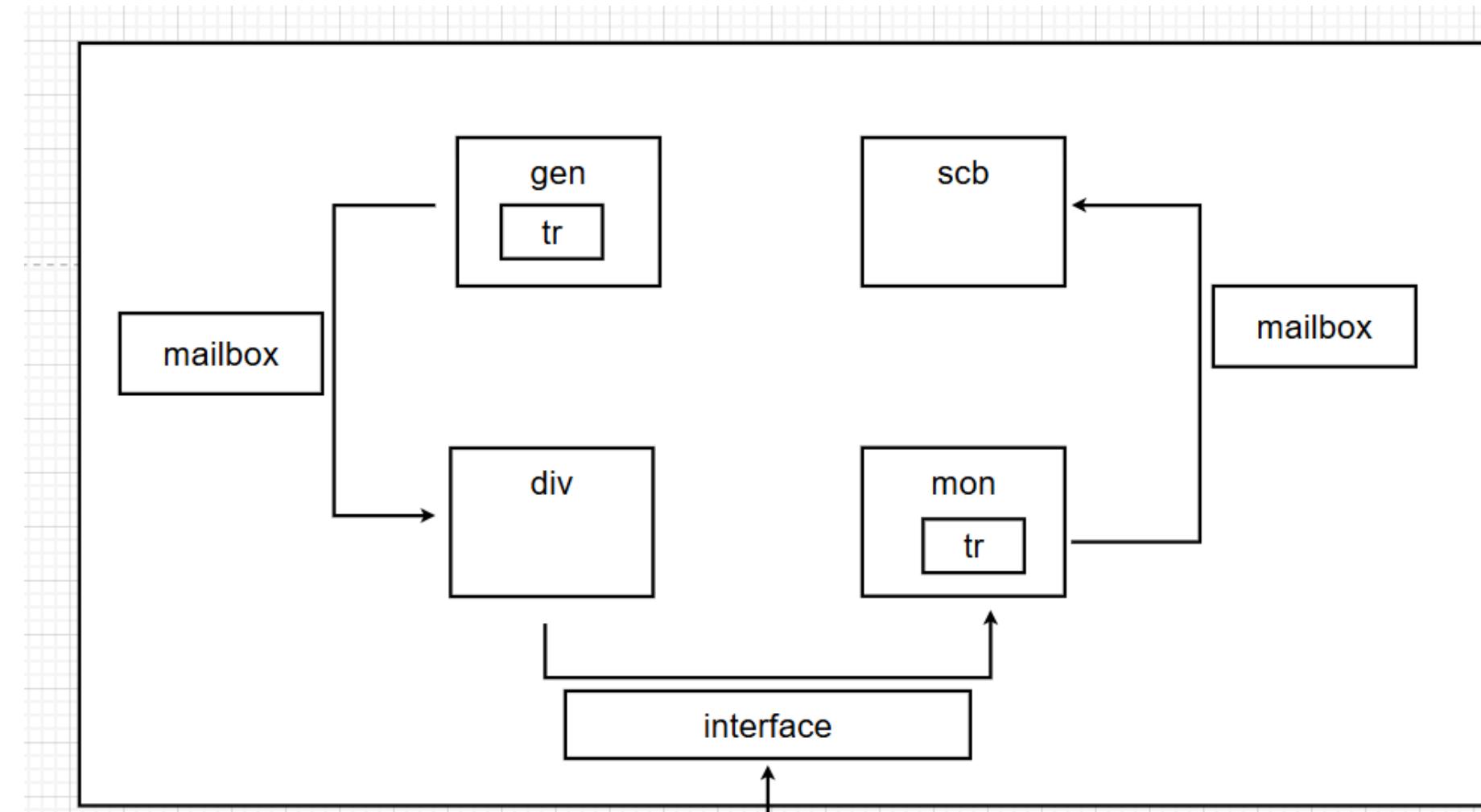
APB Bus 인터페이스를 통해 CPU와 PC 간 데이터 송수신을 수행하는 주변장치 내부적으로 TX/RX 모듈과 각 FIFO 버퍼로 구성되어 있으며, APB Master가 TDR(slv_reg1)과 RDR(slv_reg2)을 통해 데이터를 제어

- **FIFO TX :** CPU에서 전송할 데이터를 저장하고, **UART_TX**로 전달
- **UART TX :** FIFO로부터 데이터를 읽어 직렬 데이터로 변환 후 **PC**로 송신
- **UART RX :** **PC**에서 수신한 직렬 데이터를 병렬 데이터로 변환
- **FIFO RX :** 수신된 데이터를 저장 후 CPU가 APB를 통해 읽음

06 UART Peripheral 검증



06 UART Peripheral 검증



- **gen (generator)** : 송수신 데이터 생성
- **drv (driver)** : UART 프로토콜에 따라 bit 단위 전송
- **mon (monitor)** : DUT의 수신 데이터를 모니터링
- **scb (scoreboard)** : 송신/수신 데이터 일치 여부 검증

RX:

bit 단위의 직렬 데이터를 받아 병렬 데이터로
복원하며, 수신된 byte가 APB Bus의 PRDATA
와 일치하면 PASS

TX:

PWDATA에 입력된 byte를 bit 단위로 변환하
여 송신하고, 전송된 bit를 샘플링하여 복원한
데이터가 PWDATA와 일치하면 PASS

06 UART Peripheral 검증 결과

```
[1000504] RX_SEND | data=ec  
[108853495000] APB_READ | RXDATA=dd  
[PASS] RX match | sent=dd read=dd  
[109999296000] RX_SEND | data=1c  
[109999325000] APB_READ | RXDATA=ec  
[PASS] RX match | sent=ec read=ec  
[111145122000] RX_SEND | data=f8  
[111145145000] APB_READ | RXDATA=1c  
[PASS] RX match | sent=1c read=1c  
[112290948000] RX_SEND | data=07  
[112290975000] APB_READ | RXDATA=f8  
[PASS] RX match | sent=f8 read=f8  
[113436774000] RX_SEND | data=13  
[113436805000] APB_READ | RXDATA=07  
[PASS] RX match | sent=07 read=07  
[114582600000] RX_SEND | data=01  
[114582625000] APB_READ | RXDATA=13  
[PASS] RX match | sent=13 read=13
```

```
=====  
RX SCOREBOARD REPORT  
Total: 100  
PASS : 100  
FAIL : 0
```

```
[SCB] WRITE 0x8 TXDATA=00000030  
[4775000] DRV | addr=8 write=1 wdata=00000030 rdata=00000000  
[4800000] GEN | addr=c write=0 wdata=000000bb rdata=00000000  
[4820000] MON | addr=c write=0 wdata=00000030 rdata=00000000  
[PASS] 0xC RXDATA empty OK (0x00)  
[4825000] DRV | addr=c write=0 wdata=000000bb rdata=00000000  
[4850000] GEN | addr=0 write=1 wdata=00000043 rdata=00000000  
[4870000] MON | addr=0 write=1 wdata=00000043 rdata=00000000  
[SCB] Write ignored (read-only reg 0)  
[4875000] DRV | addr=0 write=1 wdata=00000043 rdata=00000000  
[4900000] GEN | addr=8 write=0 wdata=00000067 rdata=00000000  
[4920000] MON | addr=8 write=0 wdata=00000043 rdata=00000030  
[SCB] READ 0x8 ignored (write-only)  
[4925000] DRV | addr=8 write=0 wdata=00000067 rdata=00000030  
[4950000] GEN | addr=4 write=1 wdata=00000011 rdata=00000000  
[4970000] MON | addr=4 write=1 wdata=00000011 rdata=00000000  
[SCB] WRITE 0x4: data=00000011  
[4975000] DRV | addr=4 write=1 wdata=00000011 rdata=00000000
```

```
=====  
SCOREBOARD REPORT  
PASS: 100  
FAIL: 0
```

07 C Application

```
int main(){
    int led=0x01, dir=LEFT;
    char r;

    sys_init();

    while(1){
        if(uart_rx_ready()){
            r=(char)UART_RXD;
            if(r=='l'){dir=LEFT; led=0x01;}
            else if(r=='r'){dir=RIGHT; led=0x80;}
        }

        led_write(led);
        delay(200);

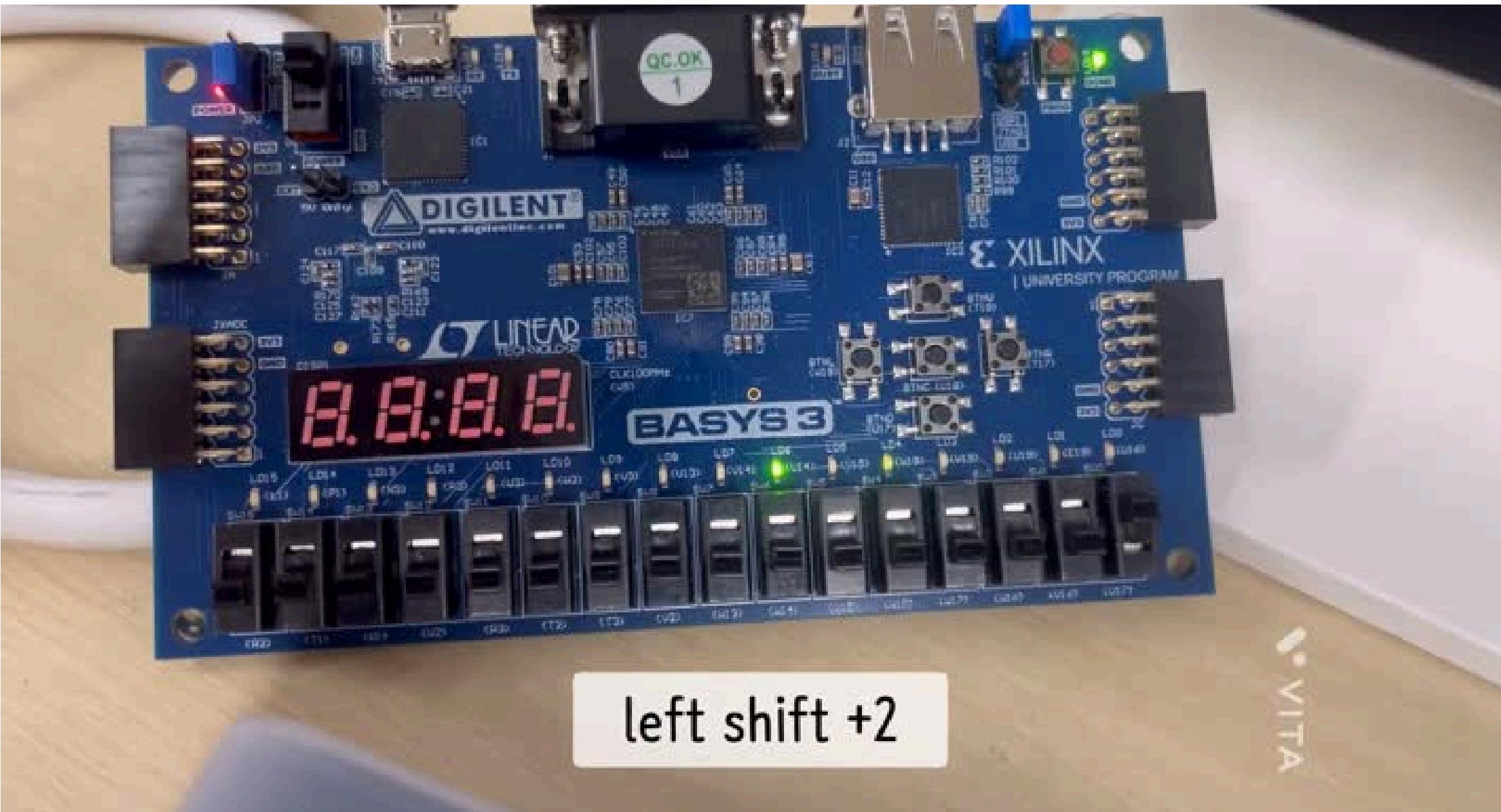
        if(dir==LEFT) led_left(&led);
        else         led_right(&led);
    }
    return 0;
}
```

```
void led_left(uint32_t* p){
    *p = (*p << 2);
    if(*p > 0x80) *p = 0x01;
}

void led_right(uint32_t* p){
    *p = (*p >> 2);
    if(*p == 0x00) *p = 0x80;
}
```

단계	ledData (이진)	설명
초기	0000 0001	1번 LED 켜짐
<<2	0000 0100	두 칸 왼쪽 이동 (1→3번 LED)
>>6	0000 0000	0 (오버플로우 없음)
OR	0000 0100	결과: 3번 LED 켜짐
초기	1000 0000	8번 LED 켜짐
>>2	0010 0000	두 칸 오른쪽 이동 (8→6번 LED)
<<6	0000 0000	(왼쪽으로 회전 없음)
OR	0010 0000	결과: 6번 LED 켜짐

07 C Application 동작영상



08 고찰 (트러블슈팅)

```
always_ff @(posedge PCLK or posedge PRESET) begin
    if (PRESET) begin
        prev_uart_tx_busy <= 1'b0;
    end else begin
        prev_uart_tx_busy <= uart_tx_busy;
    end
end

wire fifo_tx_rd_en = uart_tx_busy & ~prev_uart_tx_busy;
wire tx_start_signal = ~fifo_tx_empty & ~uart_tx_busy;

always_comb begin
    tx_wr_en = 1'b0;
    rx_rd_en = 1'b0;
    if (PSEL && PENABLE) begin
        case (PADDR[3:0])
            4'h0: tx_wr_en = PWRITE & ~fifo_tx_full;
            4'h4: rx_rd_en = ~PWRITE & ~fifo_rx_empty;
            default: ;
        endcase
    end
end

logic [31:0] status_reg;
always_comb begin
    status_reg = 32'b0;
    status_reg[0] = uart_tx_busy;
    status_reg[1] = ~fifo_rx_empty;
    status_reg[2] = fifo_rx_full;
    status_reg[3] = fifo_tx_full;
end

always_comb begin
    PREADY = 1'b1;
    PRDATA = 32'b0;
    if (PSEL && PENABLE && !PWRITE) begin
        case (PADDR[3:0])
            4'h4: PRDATA = {24'b0, fifo_rx_rdata};
            4'h8: PRDATA = status_reg;
            default: PRDATA = 32'hDEAD_BEEF;
        endcase
    end
end
```

APB 인터페이스와 UART 제어를 한 모듈에
묶으면서

조합논리 기반으로 제어 신호를 처리해 타이
밍 문제가 발생했다

08 고찰 (트러블슈팅 해결)

```
always_ff @(posedge PCLK, posedge PRESET) begin
    if (PRESET) begin
        slv_reg0[31:4] <= 0; //USTATEREG
        slv_reg1      <= 0; //USTATEREG_RX
        slv_reg3[31:8] <= 0;
        slv_reg2      <= 0;
        state_reg     <= IDLE;
        we_reg        <= 0;
        re_reg        <= 0;
        PRDATA_reg    <= 32'bx;
        PREADY_reg    <= 1'b0;
    end else begin
        slv_reg1      <= slv_reg1_next;
        slv_reg2      <= slv_reg2_next;
        state_reg     <= state_next;
        we_reg        <= we_next;
        re_reg        <= re_next;
        PRDATA_reg    <= PRDATA_next;
        PREADY_reg    <= PREADY_next;
    end
end

case (state_reg)
    IDLE: begin
        PREADY_next = 1'b0;
        if (PSEL && PENABLE) begin
            if (PWRITE) begin
                state_next = WRITE;
                we_next = 1'b1;
                re_next = 1'b0;
                PREADY_next = 1'b1;
            case (PADDR[3:2])
                2'd0: ;
                2'd1: slv_reg1_next = PWDATA;
                2'd2: begin
                    slv_reg2_next = PWDATA;
                end
                2'd3: ;
            endcase
        end else begin
            state_next = READ;
            PREADY_next = 1'b1;
            we_next = 1'b0;
            case (PADDR[3:2])
                2'd0: begin
                    PRDATA_next = slv_reg0;
                    re_next = 1'b0;
                end
            endcase
        end
    end
end
```

**APB 인터페이스와 UART 제어를 분리하여
FSM 기반으로 제어 신호를 처리함으로써 타이밍
문제를 개선**

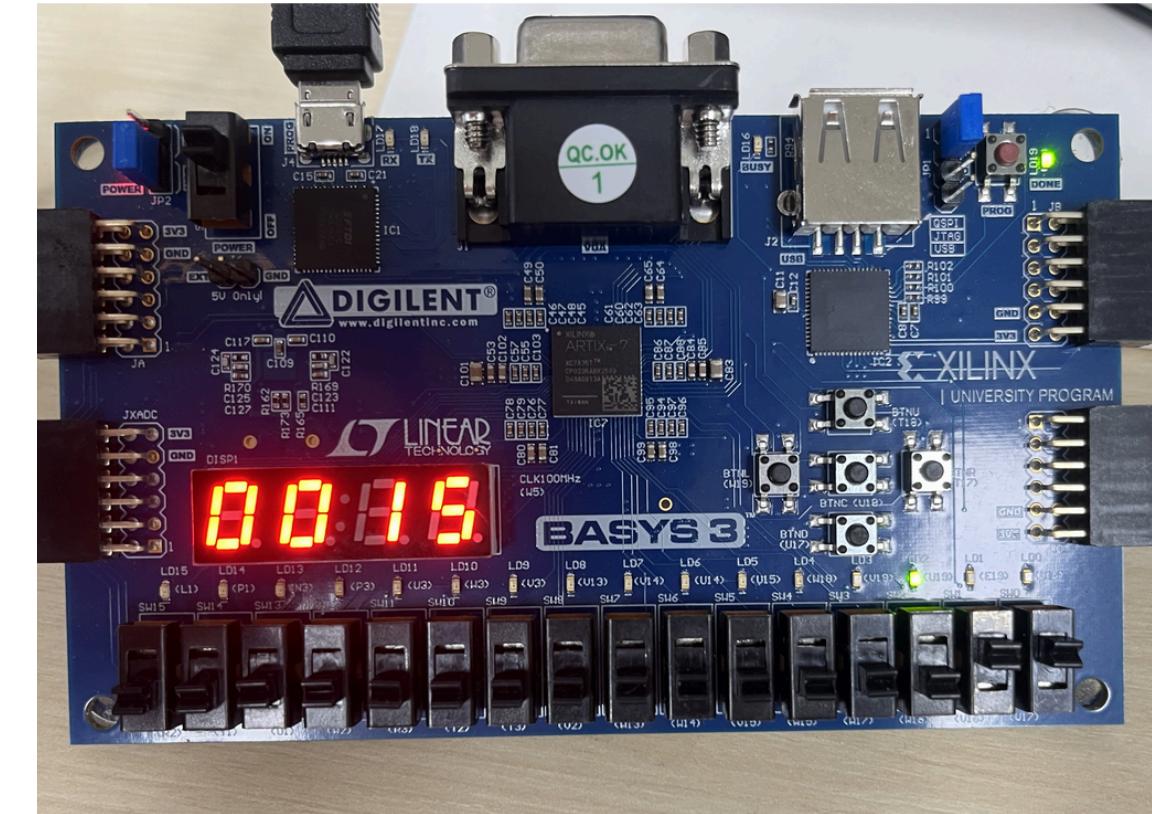
08 고찰 (개선사항)

```
// LED 비트 → 숫자 변환 (1~8)
int led_to_num(uint32_t led){
    switch(led){
        case 0x01: return 1;
        case 0x02: return 2;
        case 0x04: return 3;
        case 0x08: return 4;
        case 0x10: return 5;
        case 0x20: return 6;
        case 0x40: return 7;
        case 0x80: return 8;
        default: return 0;
    }
}

// 오프셋 수정! (0x04 → 0x00)
#define FND_ODR (*uint32_t*)(FND_BASE_ADDR + 0x00)

#define UART_USR_RX_EMPTY (1<<0)

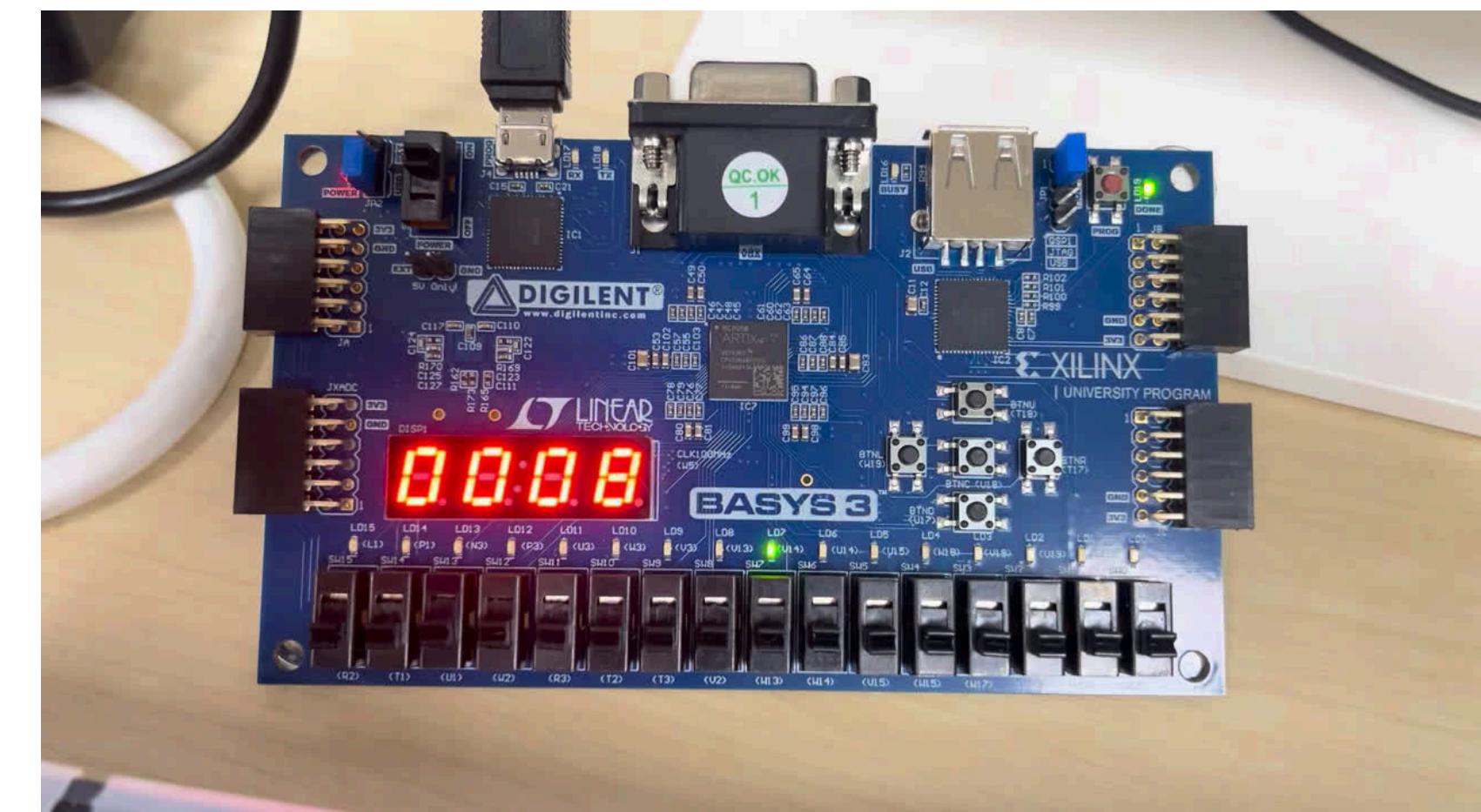
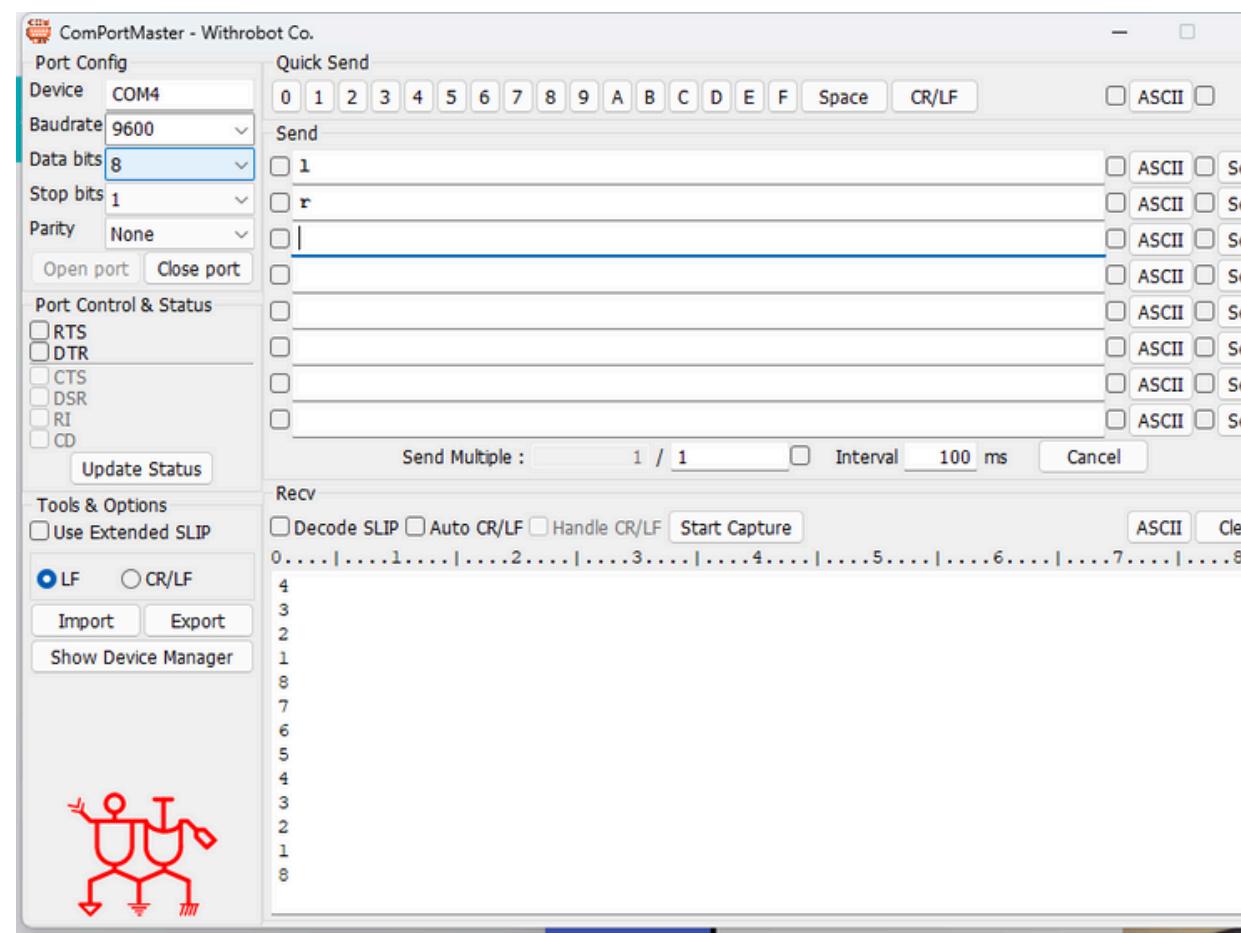
void delay(uint32_t t);
```



FND가 계속 15로 고정되는 이유는

FND_ODR 오프셋을 +0x04로 뒀서 FND_Periph의 slv_reg0에
write가 전혀 안 들어감

08 고찰 (개선사항 동작 영상)



THANK YOU

감사합니다