

---

# RISC-V / APB 기반 MCU 설계

25.10.28 채준희

# 목차



1. 프로젝트 개요

2. RV32I Multi Cycle CPU

3. AMBA BUS

4. UART 설계 및 검증

5. C언어 Test code

6. 동작영상

7. 고찰



---

# 프로젝트 개요

## 프로젝트 목표

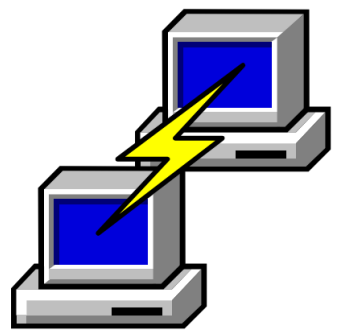
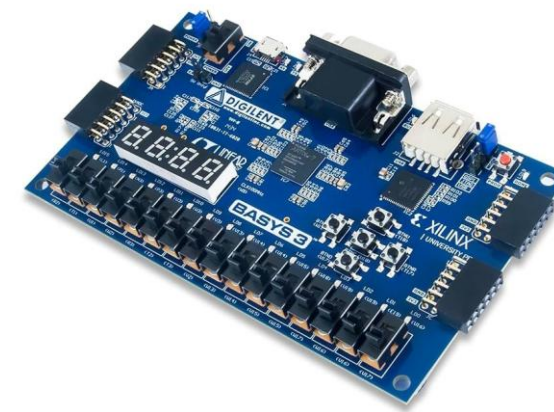
RV32I Multi Cycle cpu 와 AMBA Bus Protocol을 사용하여 여러 Peripheral 들을 사용할 수 있는 MCU 설계  
SystemVerilog Testbench Peripheral 기능 검증  
C 기반 펌웨어 작성 및 컴파일

## 핵심 기능 및 특징

하드웨어의 회로에 의해서만 동작하는 것이 아닌 ROM에 담겨있는 펌웨어에 따라 동작을 결정함  
버스를 이용하여 여러 주변장치들을 연결

## 개발환경 및 도구

- Hardware: Digilent Basys 3 FPGA Board
- Software: Xilinx Vivado, PuTTY
- Language: SystemVerilog, C

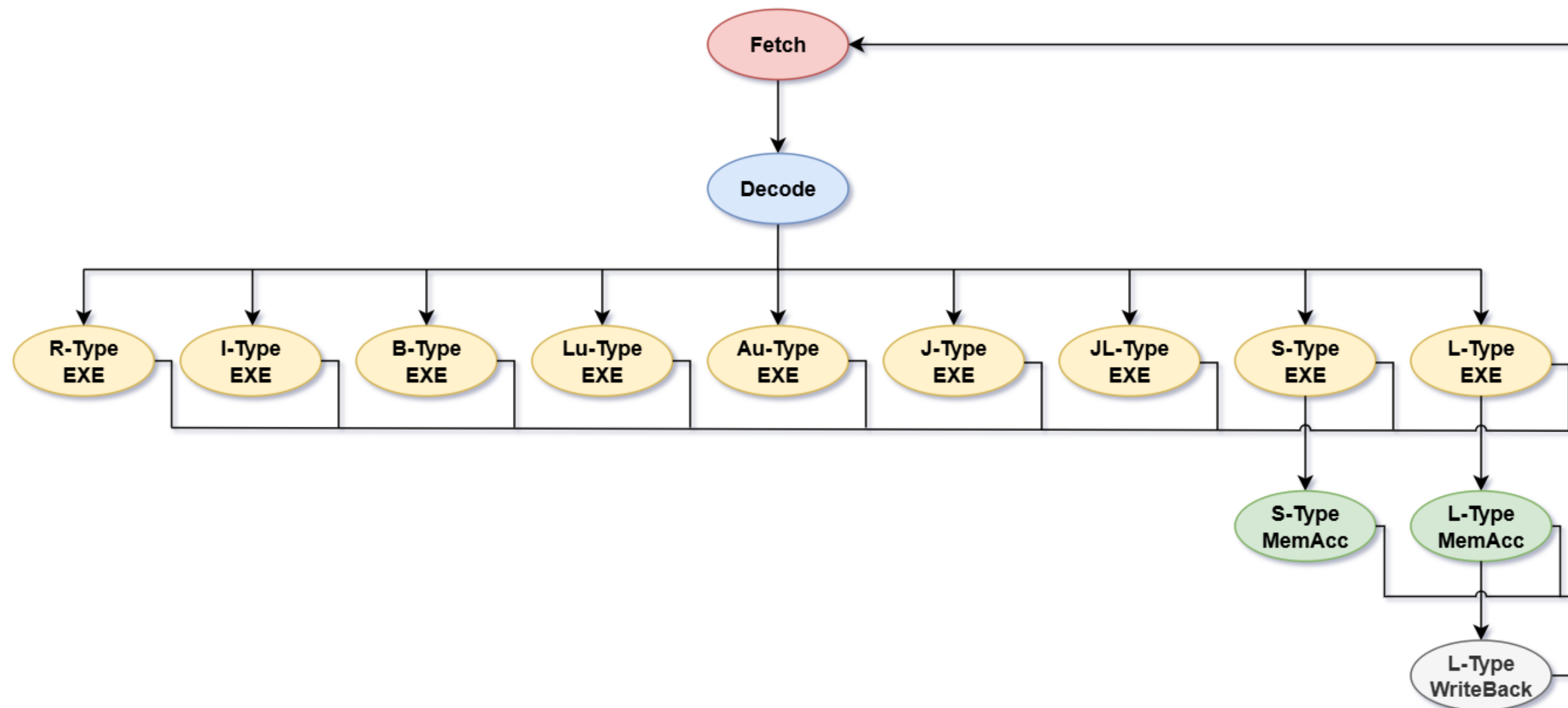


# RV32I Multi Cycle CPU

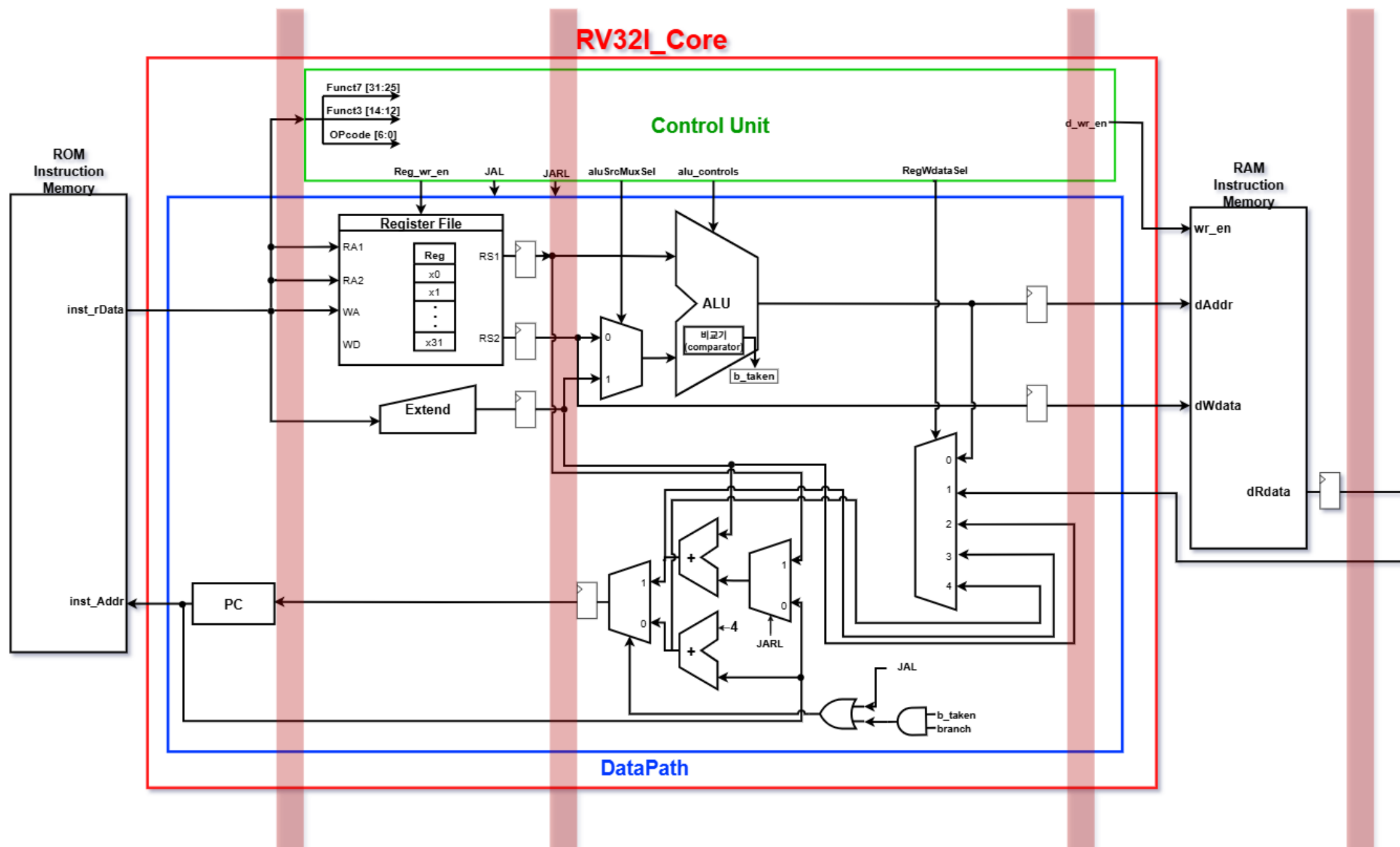
RISC-V 명령어 세트 아키텍처(ISA)의 기본 정수(Integer) 버전 32비트 명령어

여러 단계에 걸쳐 하나의 명령어를 실행하는 방식. 각 단계(Fetch, Decode, Execute, Memory, Write Back)마다 필요한 하드웨어만 활성화한다.

필요한 단계만 수행하기 때문에 클럭 효율이 높다



# RV32I Multi Cycle CPU



Fetch      Decode      Execute      MemAccess      Write Back

단계 (Phase)

역할 (Action)

RV32I 명령어와의 연관

Fetch

PC가 가리키는 주소에서 명령어(IR)를 가져오고 PC를 +4로 업데이트.

모든 명령어의 인출

Decode

IR의 레지스터 주소를 해독하고 레지스터 파일에서 데이터를 읽어옴.

R/I/S/B/U/J 타입 모두 해독.

Execute

명령어 유형에 따라 ALU 연산 수행. (R-Type, I-Type, Branch 조건 검사 등)

R-Type 연산, I-Type 주소/데이터 계산, B-Type 분기 결정.

Memory Access

메모리 접근이 필요한 명령어만 수행.

Load (L-Type): 메모리에서 데이터 인출. Store (S-Type): 데이터를 메모리에 기록.

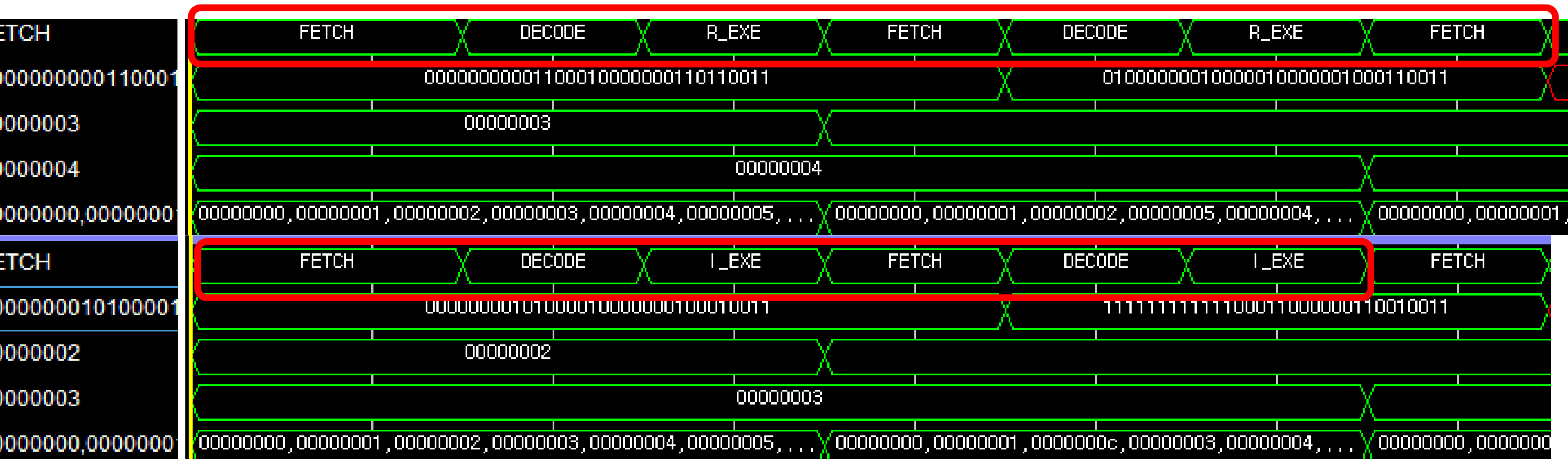
Write Back

결과를 레지스터 파일에 기록.

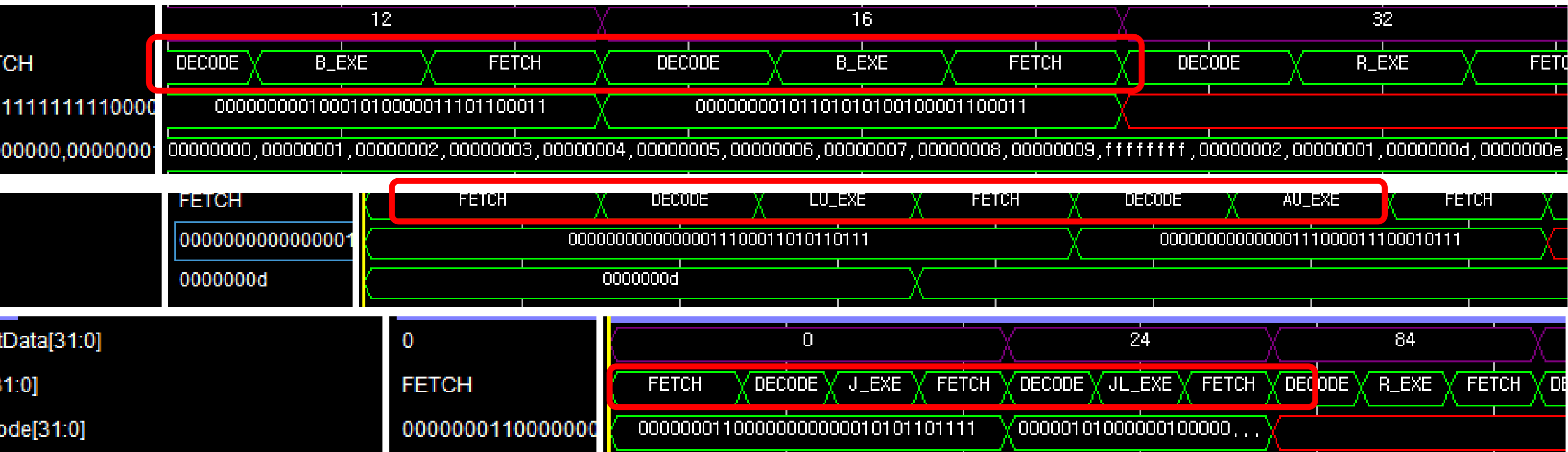
R-Type, Load (L-Type), U/J-Type의 결과를 레지스터에 기록.



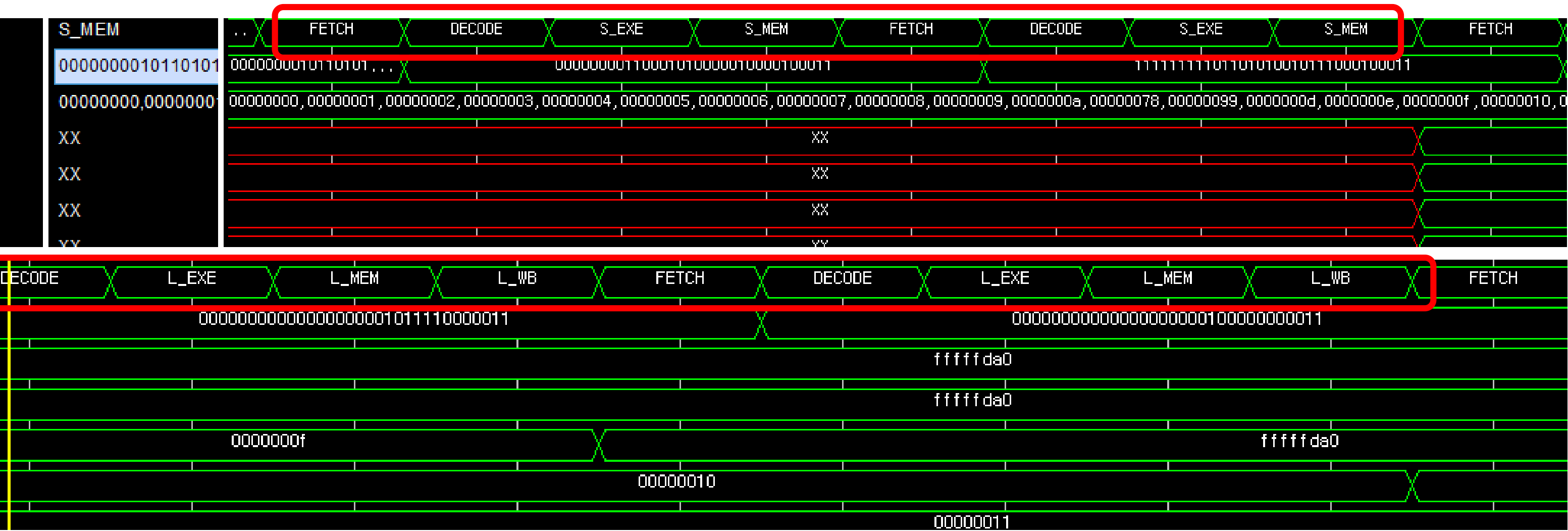
# RV32I Multi Cycle CPU



# RV32I Multi Cycle CPU



100





## APB BUS

- AMBA는 ARM에서 만든 시스템 온 칩 내부 구성요소들을 연결하는 표준 버스 규격
- AMBA 프로토콜 내에서 가장 낮은 대역폭을 가지며, 전력 소비를 최소화하고

설계 복잡성을 줄이는 데 초점을 맞춘 버스

저속 저전력 주변 장치들을 연결하는 데 최적화

- Non-pipelined으로 한 번에 하나의 전송만 처리하며 주소와 데이터가 한 클럭 사이클에 동시에 처리되지 않음

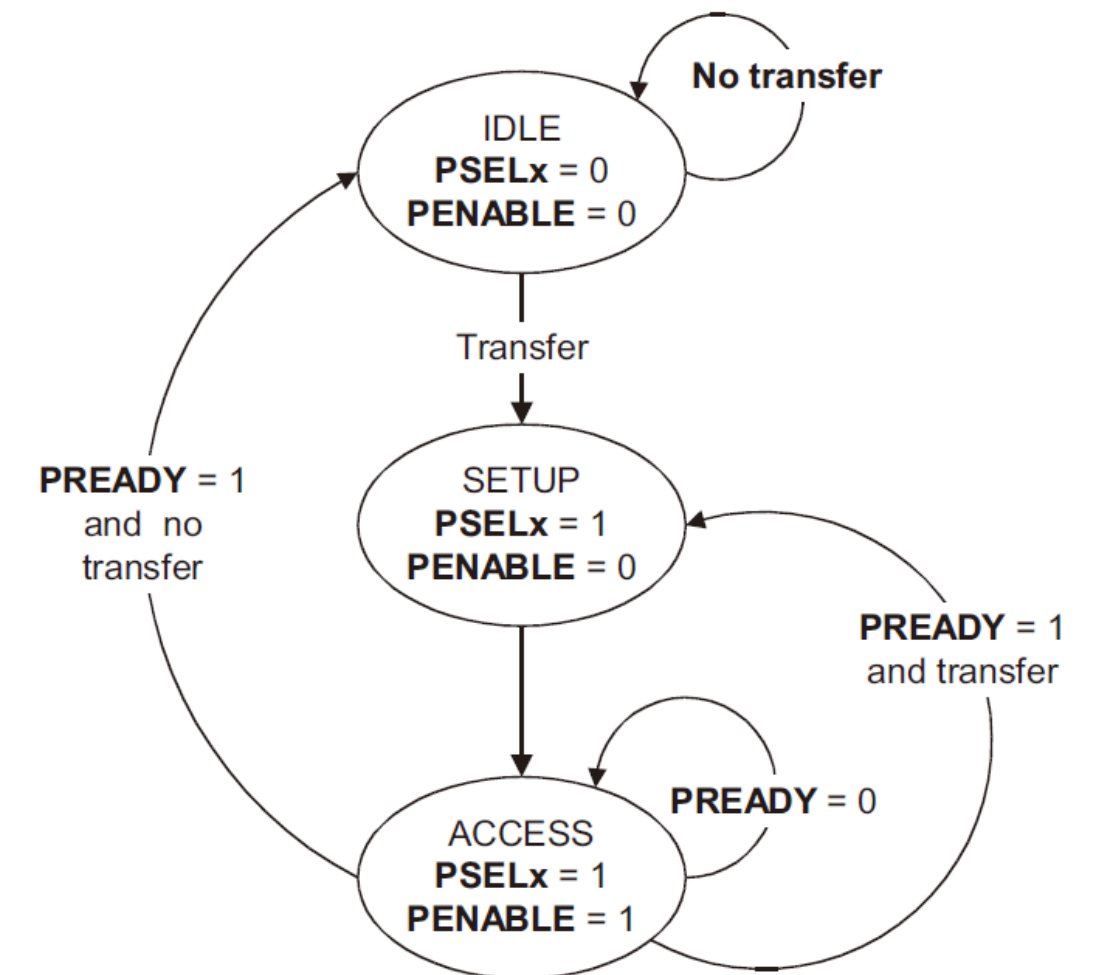
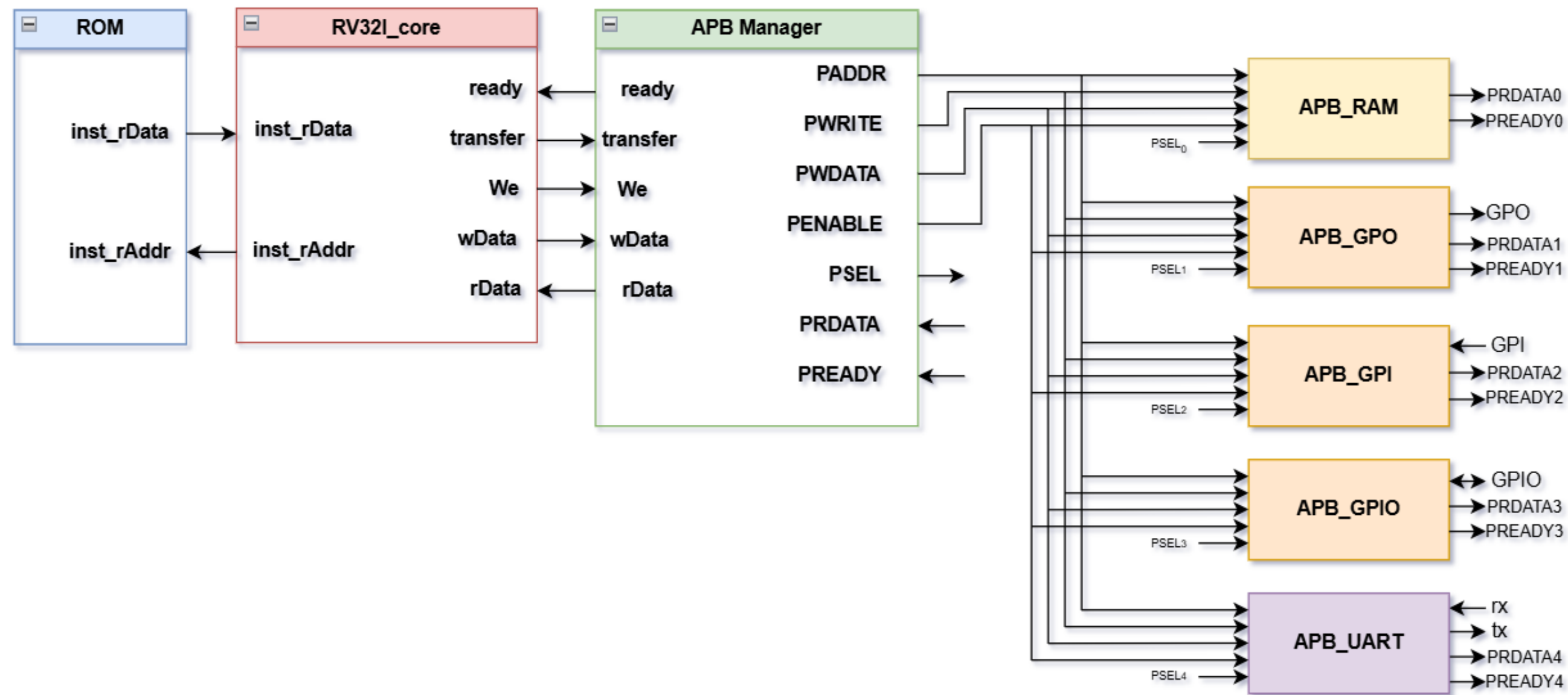
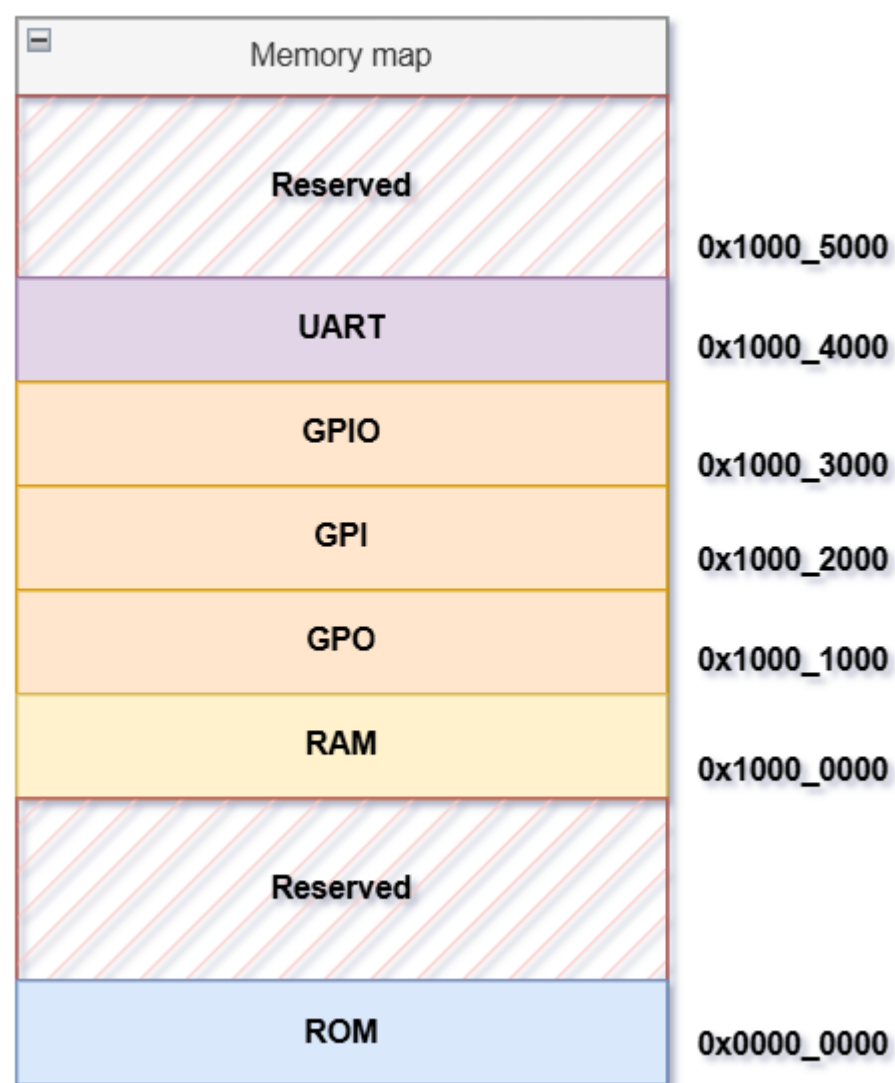


Figure 3-1 State diagram



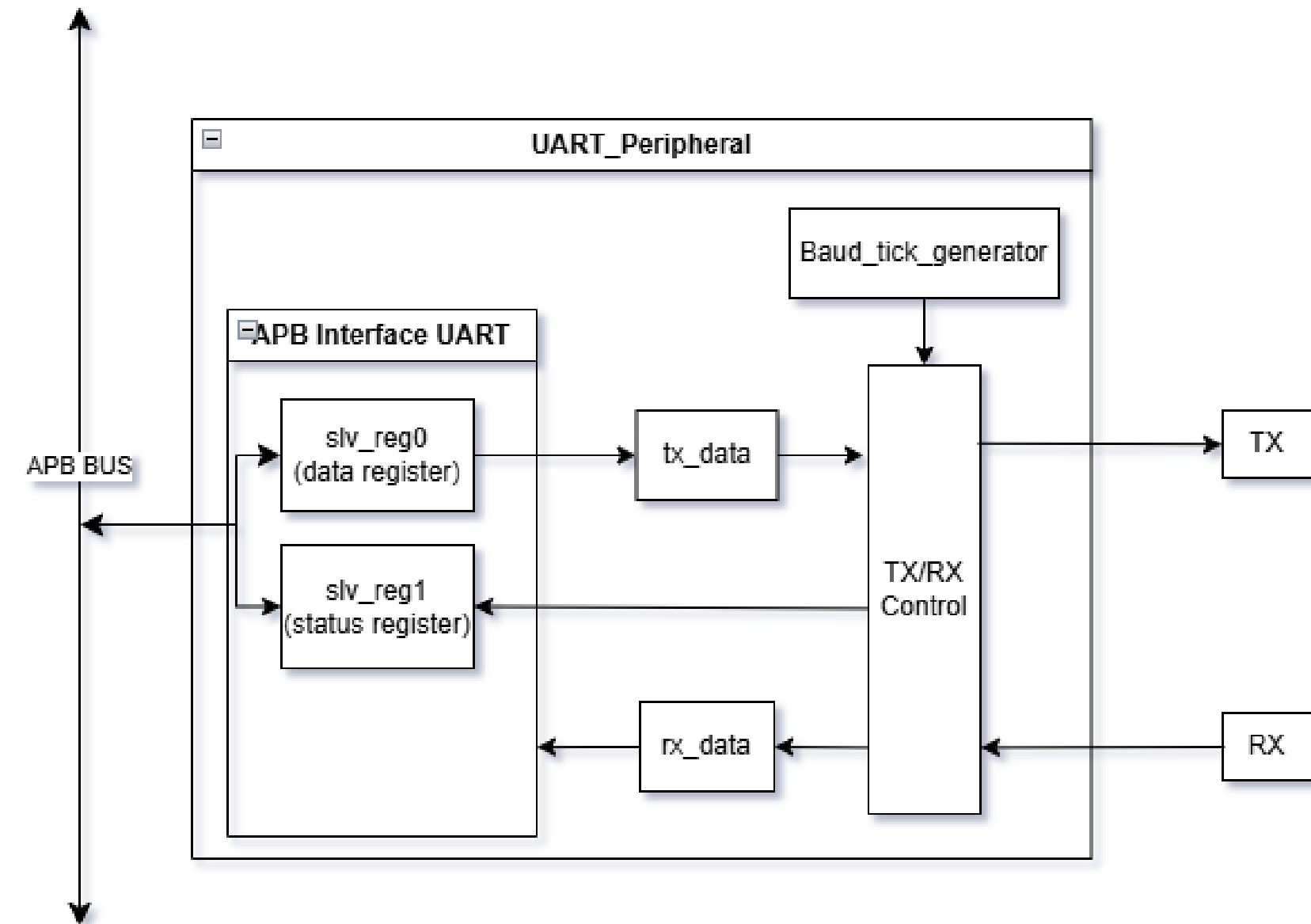
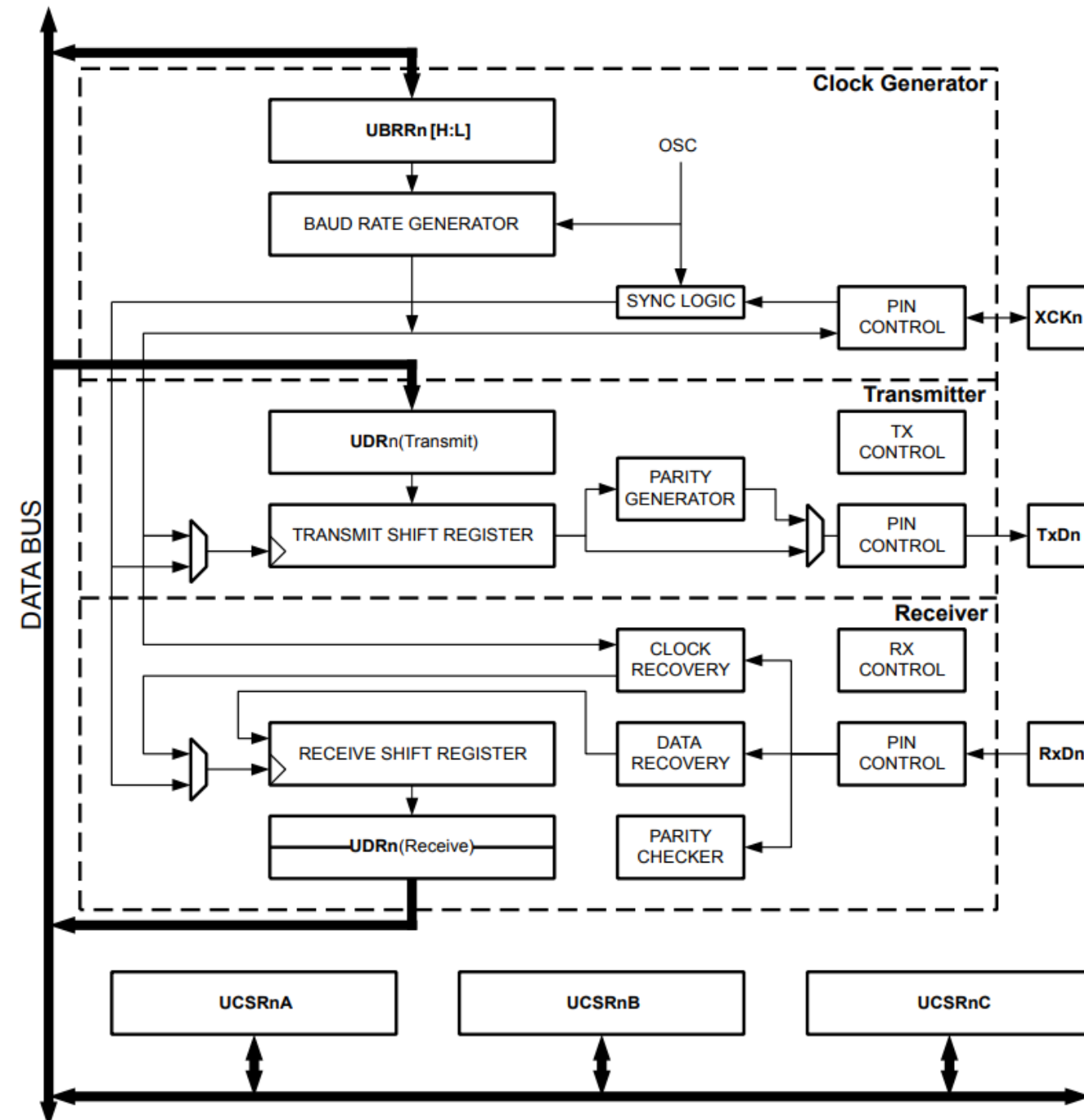
# APB BUS



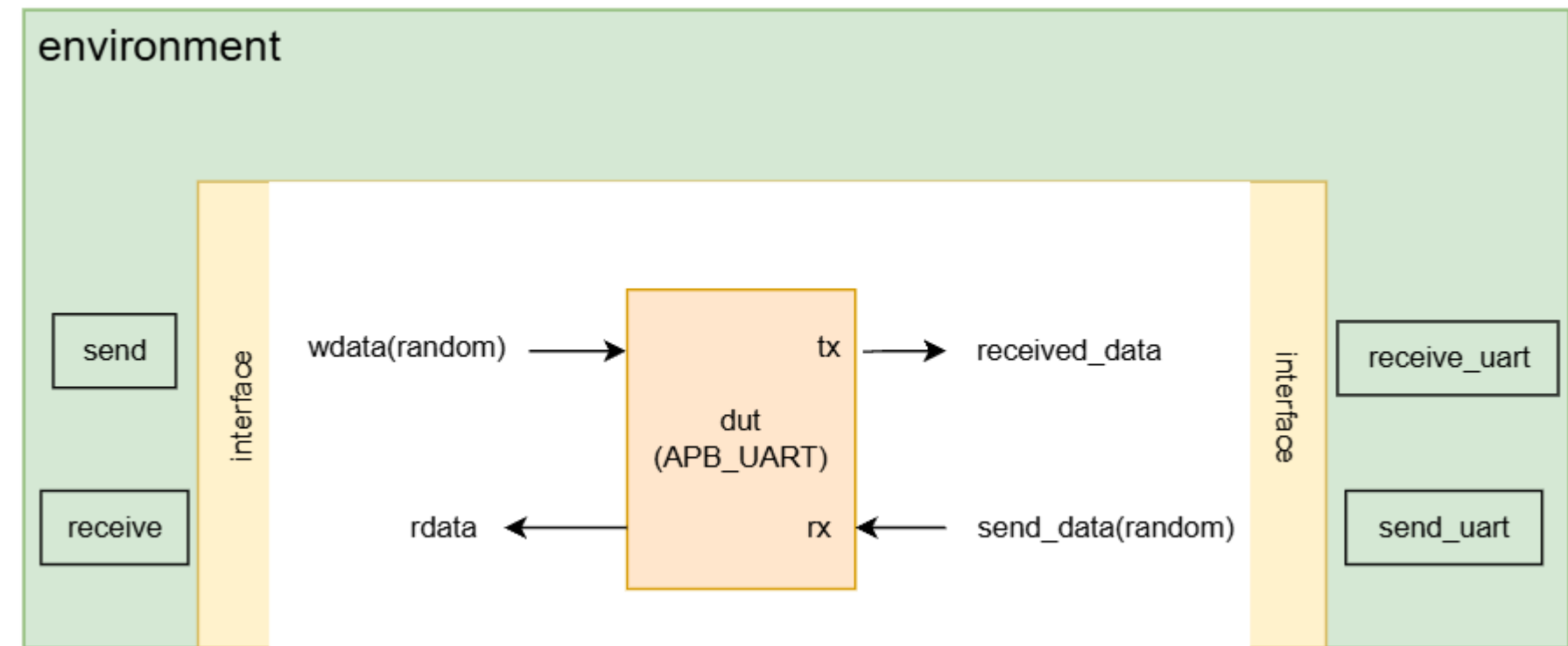
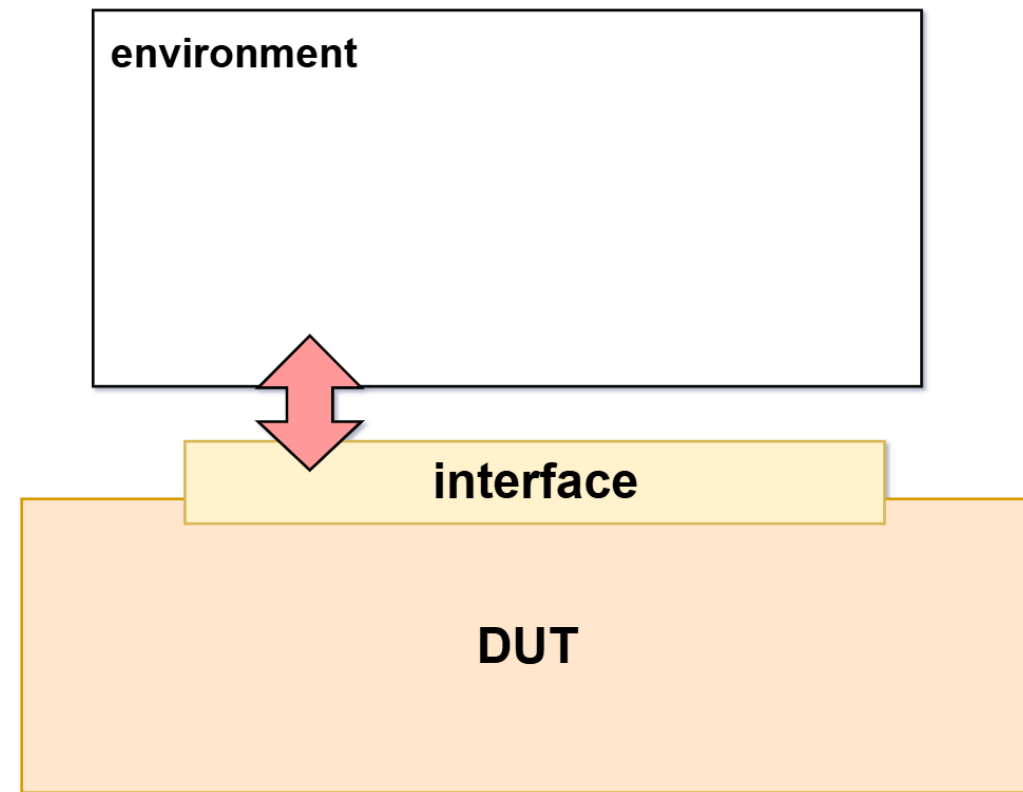
# UART Peripheral

Atmega128

Figure 25-1 USART Block Diagram<sup>(1)</sup>



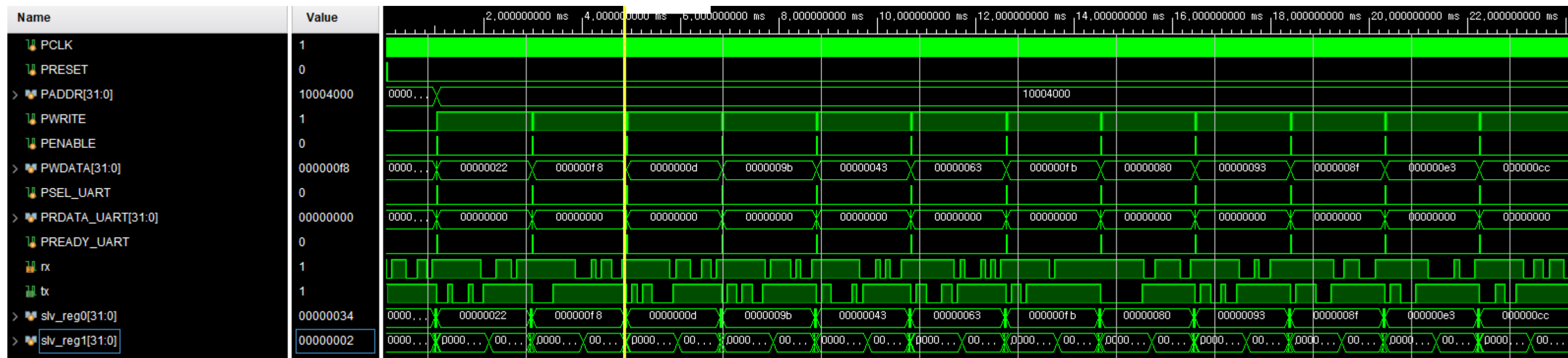
# SystemVerilog Verification



**send\_data == rdata -> rx pass**

**wdata == received\_data -> tx pass**

# SystemVerilog Verification



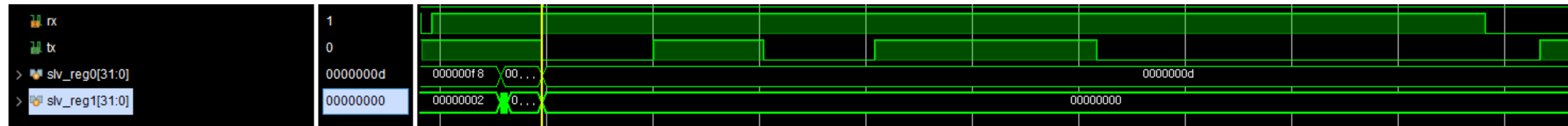
```
981951026000[SEND_UART], transfer = 1, write = 1, addr = xxxxxxxx, wdata = 00000010, rdata = 000000a7, received_data = 12, send_data = f4
981951026000[RECEIVE], transfer = 1, write = 0, addr = xxxxxxxx, wdata = 00000010, rdata = 000000a7, received_data = 12, send_data = f4
981951055000[SEND], transfer = 1, write = 1, addr = xxxxxxxx, wdata = 00000010, rdata = 000000f4, received_data = 12, send_data = f4
982836496000[RECEIVE_UART], transfer = 1, write = 1, addr = xxxxxxxx, wdata = 00000010, rdata = 000000f4, received_data = 10, send_data = f4
```

rx PASS!

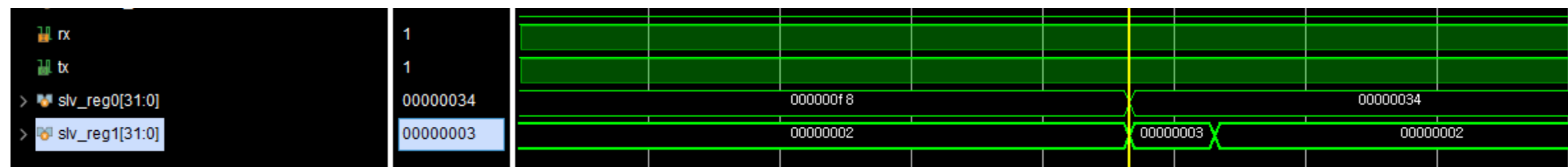
tx PASS!

```
=====
===== Test Report =====
=====
===== Total cycle : 512 =====
==      Total    tx : 512    rx : 512      ==
==      Pass     tx : 512    rx : 512      ==
==      Fail     tx :   0    rx :   0      ==
=====
===== Test bench is finish =====
=====
```

# SystemVerilog Verification



tx 종일때 status register[1]-> 0확인



rx 종료 직후 status register [0]->1확인

# C언어 Test code

주기적으로 uart로 현재 상태, GPO출력  
GPIO로 세팅 모드에 진입하여 uart 로 모드를 전환할 수 있는 펌웨어 설계

## 메모리주소 정의

```
#define APB_BASE_ADDR 0x10000000
#define GPO_OFFSET 0x1000
#define GPI_OFFSET 0x2000
#define GPIO_OFFSET 0x3000
#define UART_OFFSET 0x4000

#define GPO_BASE_ADDR (APB_BASE_ADDR + GPO_OFFSET)
#define GPI_BASE_ADDR (APB_BASE_ADDR + GPI_OFFSET)
#define GPIO_BASE_ADDR (APB_BASE_ADDR + GPIO_OFFSET)
#define UART_BASE_ADDR (APB_BASE_ADDR + UART_OFFSET)

// GPO
#define GPO_CR (*(volatile uint32_t*)(GPO_BASE_ADDR + 0x00))
#define GPO_ODR (*(volatile uint32_t*)(GPO_BASE_ADDR + 0x04))

// GPI (사용하지 않지만 정의는 유지)
#define GPI_CR (*(volatile uint32_t*)(GPI_BASE_ADDR + 0x00))
#define GPI_IDR (*(volatile uint32_t*)(GPI_BASE_ADDR + 0x04))

// GPIO (사용하지 않지만 정의는 유지)
#define GPIO_CR (*(volatile uint32_t*)(GPIO_BASE_ADDR + 0x00))
#define GPIO_ODR (*(volatile uint32_t*)(GPIO_BASE_ADDR + 0x04))
#define GPIO_IDR (*(volatile uint32_t*)(GPIO_BASE_ADDR + 0x08))

// UART
#define UART_DR (*(volatile uint32_t*)(UART_BASE_ADDR + 0x00))
#define UART_SR (*(volatile uint32_t*)(UART_BASE_ADDR + 0x04))

// Status Register (SR) 비트 정의
#define UART_SR_RXNE (1 << 0)
#define UART_SR_TXC (1 << 1)

#define TX_DELAY_COUNT 5000
```

```
uint8_t UART_receive() {
    while (!(UART_SR & UART_SR_RXNE)) {
    }
    return (uint8_t)UART_DR;
}
```

uart 상태 레지스터를 읽어와 rx상태 확인 ->rx가 끝날때까지 대기

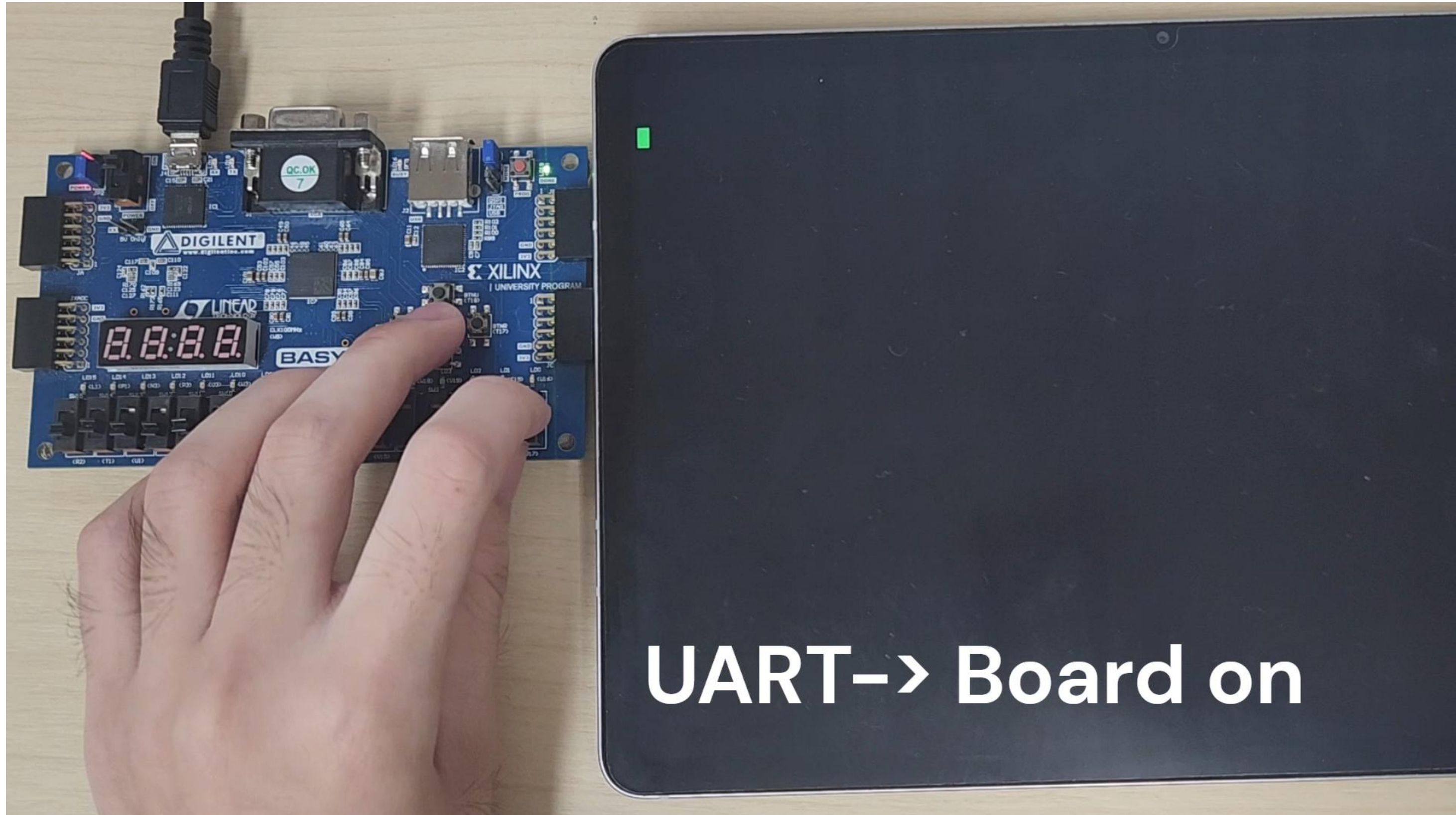
```
void UART_send(uint8_t data) {
    UART_DR = data;
    delay(TX_DELAY_COUNT);
}
```

uart 데이터 레지스터에 값 쓰기 tx시간 만큼 대기

```
if (GPI_IDR == 0x01) {
    current_led_pattern = uart_led();
}
```



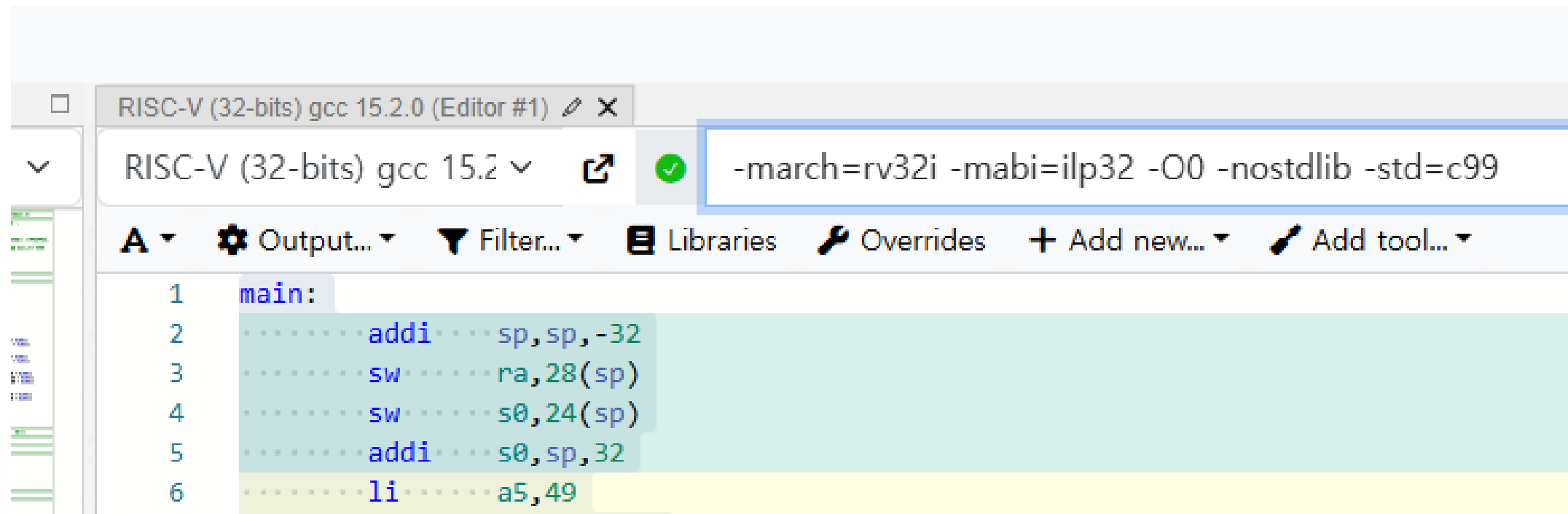
## 동작영상





# Trouble shooting

컴파일러에서 RV32I set이 아닌 명령어 컴파일되는 문제



The screenshot shows a code editor window titled "RISC-V (32-bits) gcc 15.2.0 (Editor #1)". The editor displays assembly code for a function named `main`. The code consists of six lines:

```
1  main:
2      .... addi    sp, sp, -32
3      .... sw     ra, 28(sp)
4      .... sw     s0, 24(sp)
5      .... addi    s0, sp, 32
6      .... li     a5, 49
```

Below the code, there is a toolbar with icons for "Output...", "Filter...", "Libraries", "Overrides", "Add new...", and "Add tool...". Above the toolbar, there is a dropdown menu showing "RISC-V (32-bits) gcc 15.2" with a green checkmark icon. To the right of the dropdown, the compiler options are displayed: `-march=rv32i -mabi=ilp32 -O0 -nostdlib -std=c99`.

c코드에서 다른 명령어세트를 유발하는 코드 우회,  
컴파일러 옵션으로 RV32I만 사용하도록 지정

---




## 고찰

이번 프로젝트의 설계방향은 최대한 간단한 하드웨어를 설계하고 그것들을 복잡한 소프트웨어로 제어하는 것으로 잡았었습니다.

그런데 RV32I 만 사용가능 했기 때문에 코드를 작성하면서 복잡한 기능들을 구현하기엔 부족함을 느꼈고 설계방향과는 조금 다르게 하드웨어와 소프트웨어의 조화로운 조합이 필요하다는 것을 배웠습니다.

CPU에 APB Bus를 붙이고 주변장치를 직접 설계·연결하며 MCU에 가까운 시스템을 구현해보는 경험을 하면서 임베디드 시스템의 구조와 동작 원리를 체감할 수 있었습니다.



---



## 참고자료

<https://ww1.microchip.com/downloads/en/DeviceDoc/doc2467.pdf> - Atmega128 datasheet

<https://developer.arm.com/documentation/ih0024/latest/> - APB Protocol Specification