RISC-V Multicycle 및 AMBA APB UART 설계/검증

황석현

2025.10.28

목차

- 프로젝트 목표
- RISC-V MultiCycle
  - 구조도, 명령어 type별 동작 확인
- AMBA APB
  - UART_peri 설계, 검증, C TestCode, 동작영상
- 고찰

프로젝트 목표

- **RISC-V Multi-cycle 설계**
  : RISC-V RV32I를 이해하고 Multi-cycle 설계.

- **시스템 확장**
  : AMBA APB를 연동하고 APB Slave인 UART 모듈을 설계하여 CPU와 외부 통신이 가능하도록 구현.

- **블록 레벨 검증**
  : SystemVerilog Class 기반 검증 환경을 구축하여 APB-UART 모듈 검증.

- **하드웨어 검증**
  : CPU에서 실행되는 C언어 테스트 코드를 작성하여 실제 보드에서 동작 확인.

프로젝트 환경

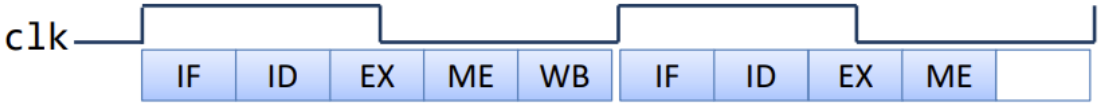
개발 언어
: System Verilog, C

Simulation Tool
: Xilinx VIVADO

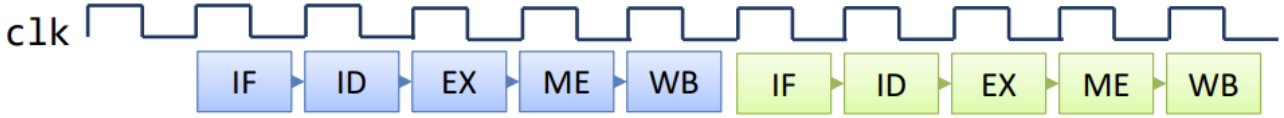
Hardware
: basys3

MultiCycle

- **Single cycle execution**



- **Multi cycle execution**



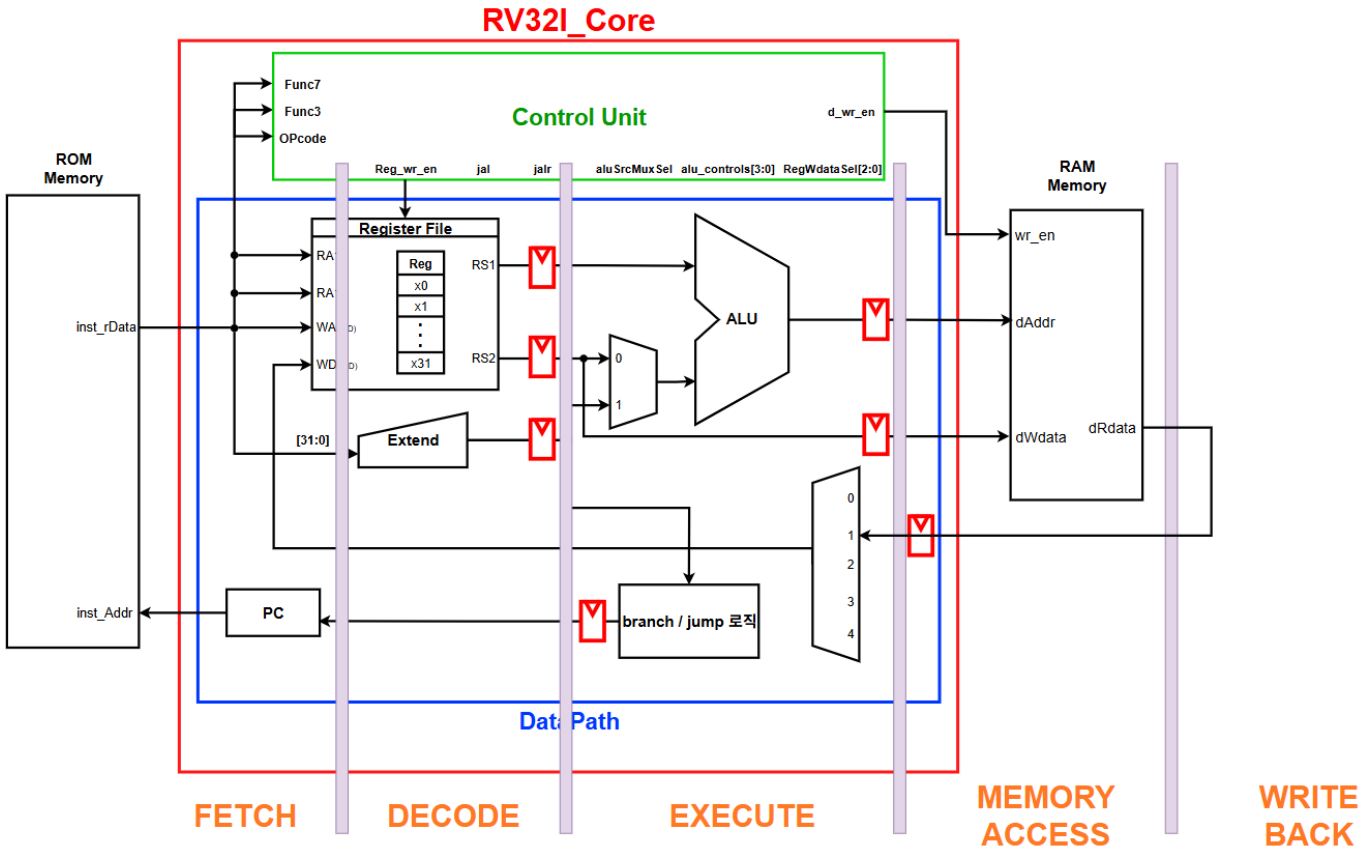
Multicycle : 하나의 명령어를 여러 clock cycle에 걸쳐 실행하는 CPU 구조.

필요성 : Singlecycle은 1 clock 안에 모든 작업을 수행해야 하기에 1 clock cycle의 주기가 굉장히 길어짐.

- **Single-Cycle 대비 장점**

  1. 긴 명령어(ex. lw, sw)에 clock 주기를 맞출 필요가 없어 더 빠른 클럭 사용 가능.
  2. ALU, 메모리 등 주요 자원을 여러 사이클에 걸쳐 재사용하므로 하드웨어 면적을 줄일 수 있음.

RV32I MultiCycle - Block Diagram



- **주요 실행 단계**

  IF : 메모리에서 다음 명령어 인출.

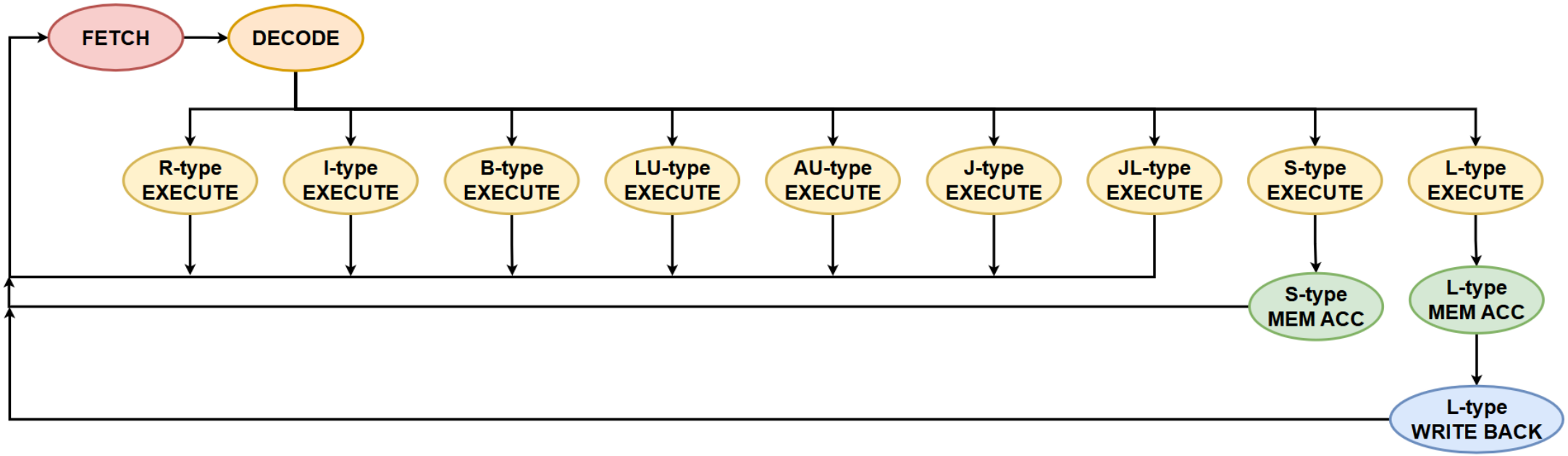
  ID : 명령어 해석.

  EX : 계산 수행.

  MEM : 메모리에 데이터를 쓰거나 읽기.

  WB : 최종 결과를 레지스터에 저장.

RV32I MultiCycle - FSM

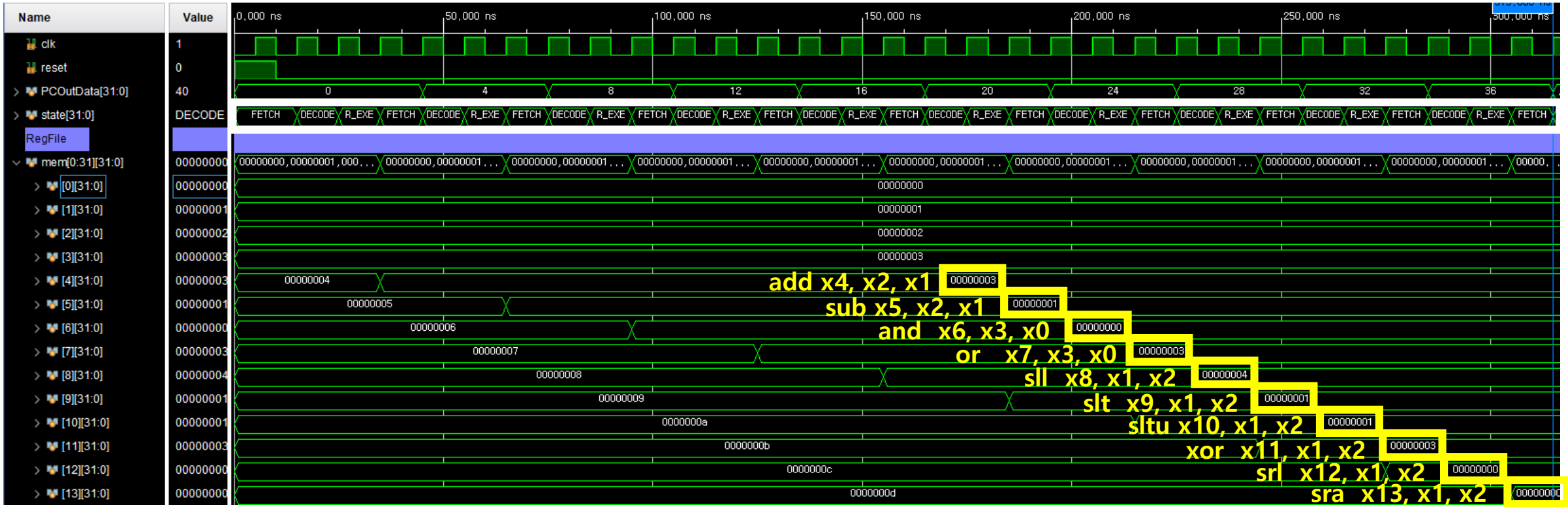


모든 명령어는 FETCH, DECODE 상태를 공통으로 거침.

DECODE 후 명령어의 Opcode에 따라 R-type, Load, Store, Branch 등 서로 다른 상태로 분기.

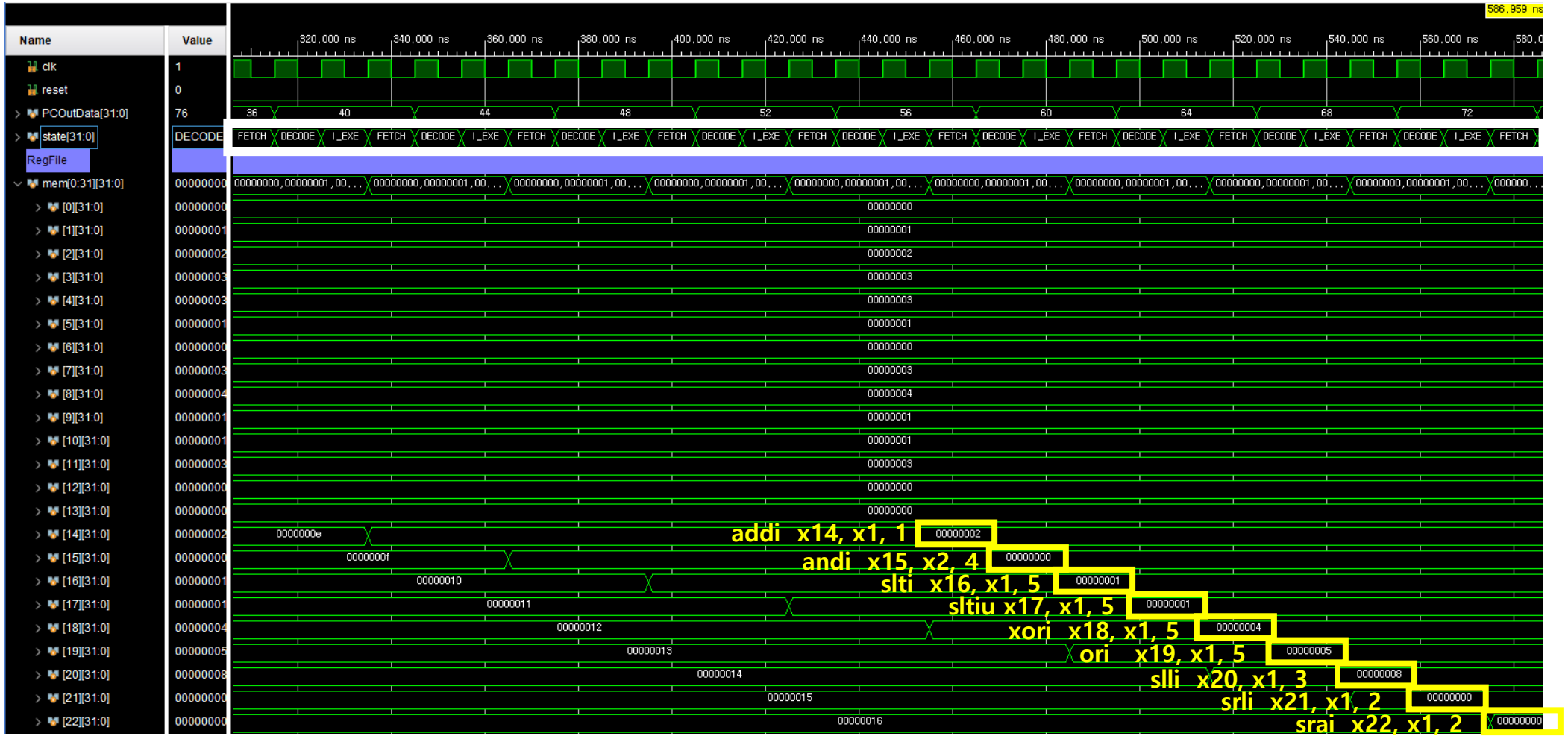
각 명령어 실행이 완료되면 다시 FETCH 상태로 복귀.

1. R-type Simulation



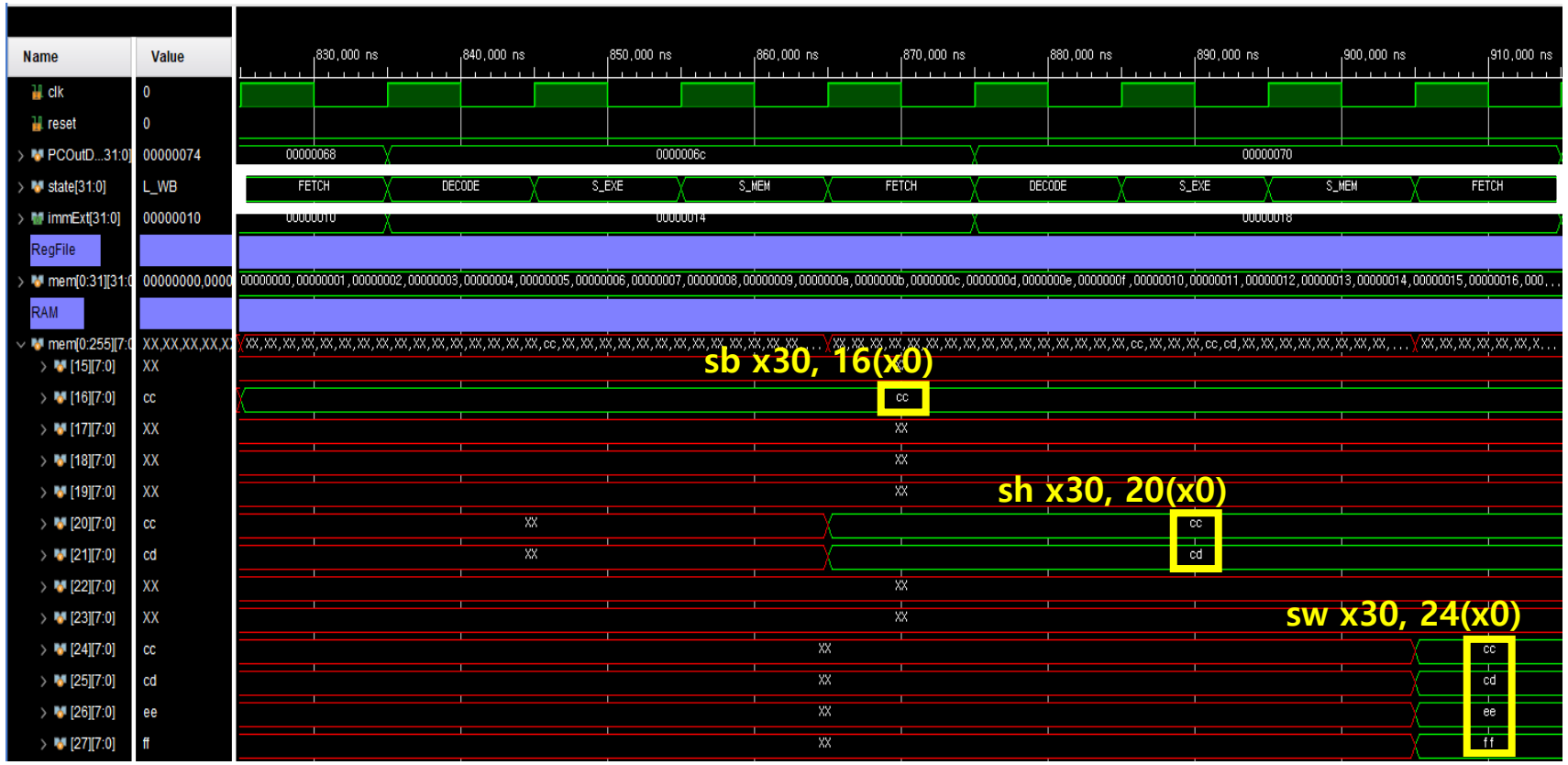
FETCH → DECODE → R-type EXECUTE → (Back to FETCH)

2. I-type Simulation



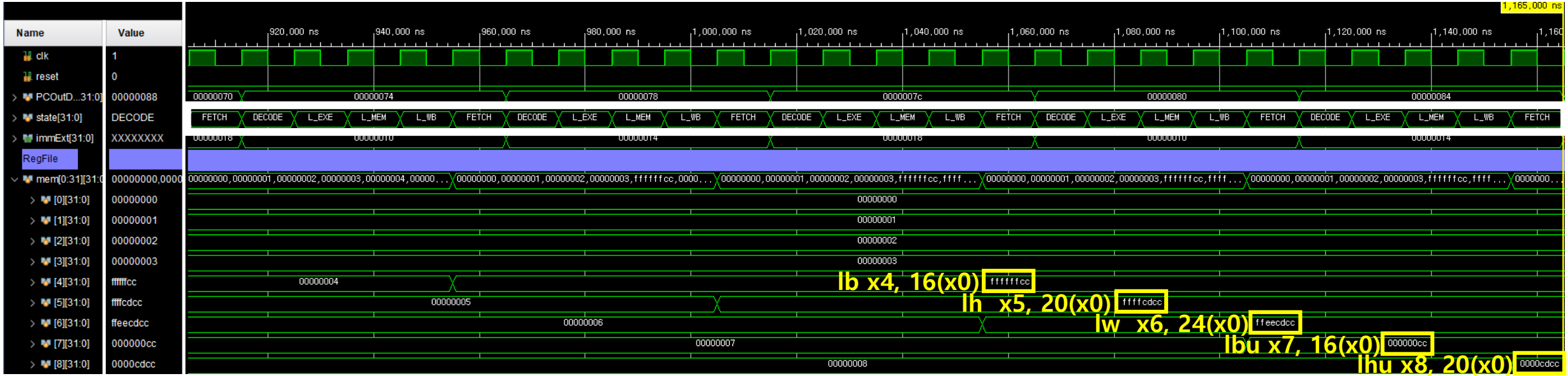
FETCH → DECODE → I-type EXECUTE → (Back to FETCH)

3. S-type Simulation



FETCH → DECODE → S-type EXECUTE → S-type MEM ACC → (Back to FETCH)

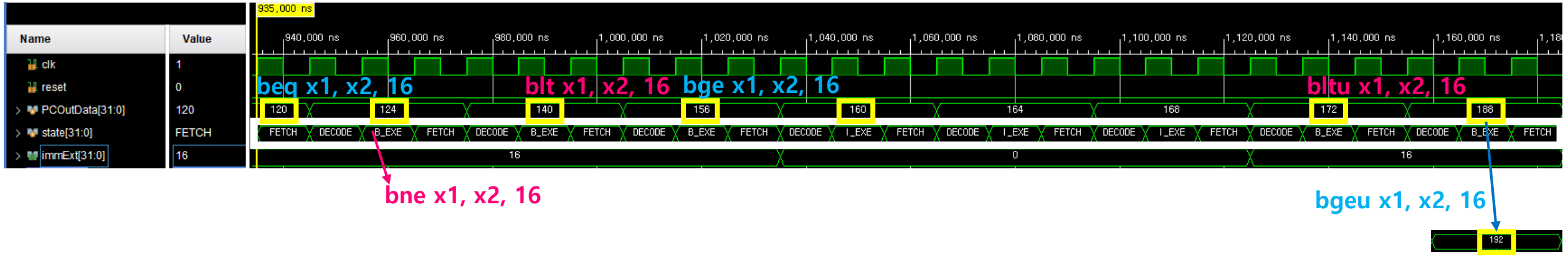
4. L-type Simulation



FETCH → DECODE → L-type EXECUTE → L-type MEM ACC
→ L-type WRITEBACK→ (Back to FETCH)

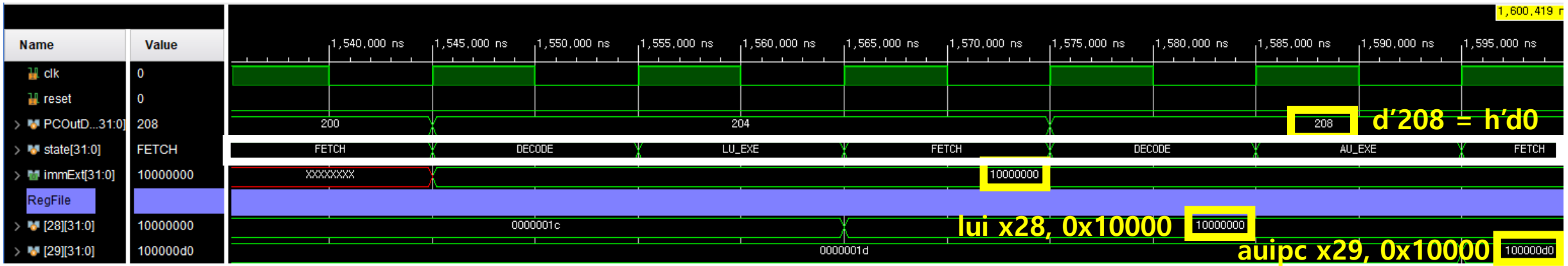
5. B-type Simulation

Branch O
Branch X



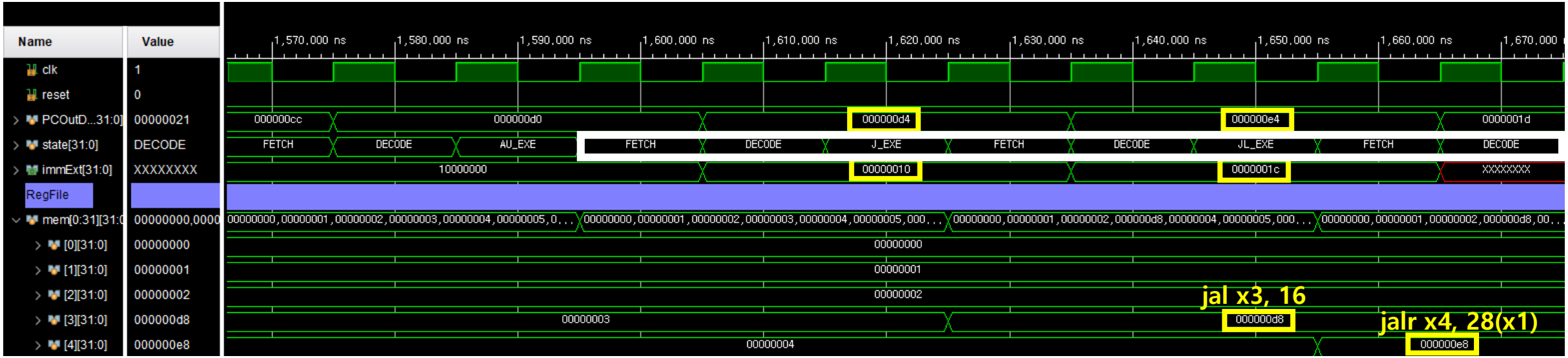
FETCH → DECODE → B-type EXECUTE → (Back to FETCH)

6. U-type Simulation



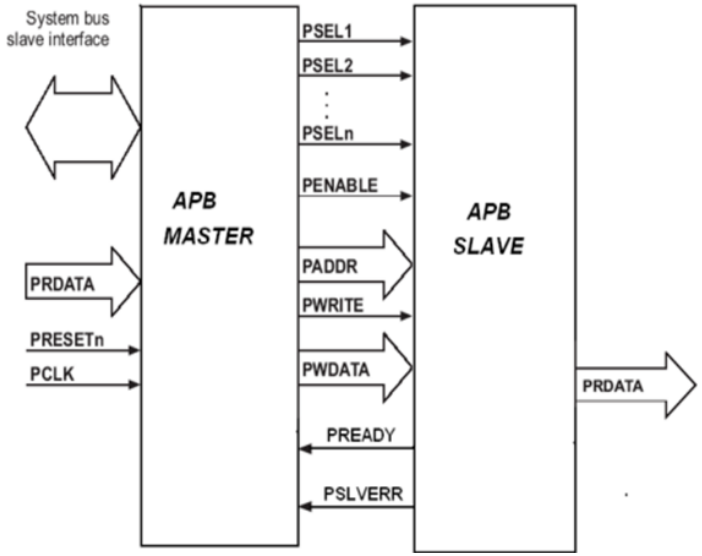
LU : FETCH → DECODE → LU-type EXECUTE → (Back to FETCH)
AU : FETCH → DECODE → AU-type EXECUTE → (Back to FETCH)

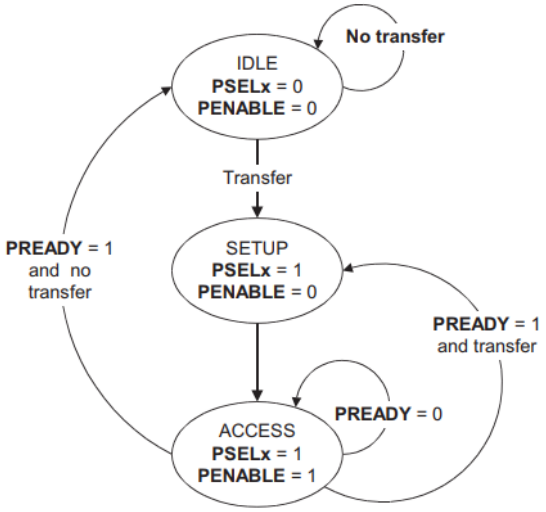
7. J-type Simulation



J : FETCH → DECODE → J-type EXECUTE → (Back to FETCH)
JL : FETCH → DECODE → JL-type EXECUTE → (Back to FETCH)

APB





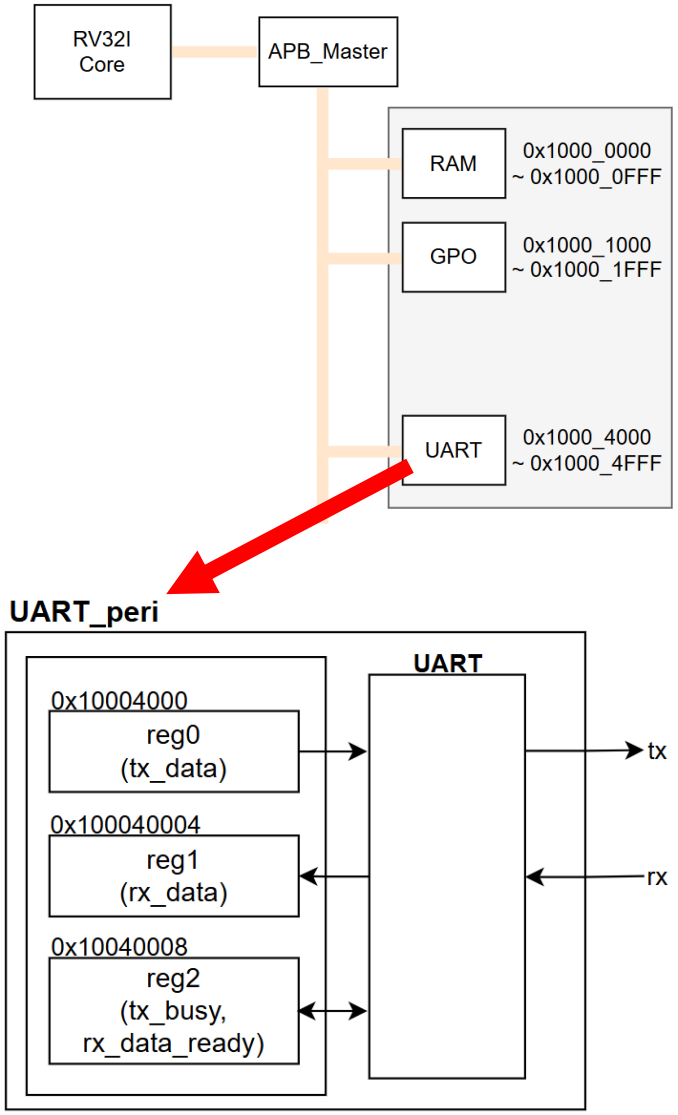
: AMBA(Advanced Microcontroller Bus Architecture) 일종
저전력/저성능 주변장치를 위한 간단한 버스 프로토콜

- **특징**
  간단한 신호 체계, Non-Pipeline 구조.
  모든 접근은 최소 2 cycle 사용

- **주요 신호**

PADDR	주소
PSELx	slave sel 신호
PENABLE	ACCESS 상태
PWRITE	쓰기 : 1 / 읽기 : 0
PWDATA / PRDATA	쓰기 / 읽기 데이터

APB – UART_peri 설계



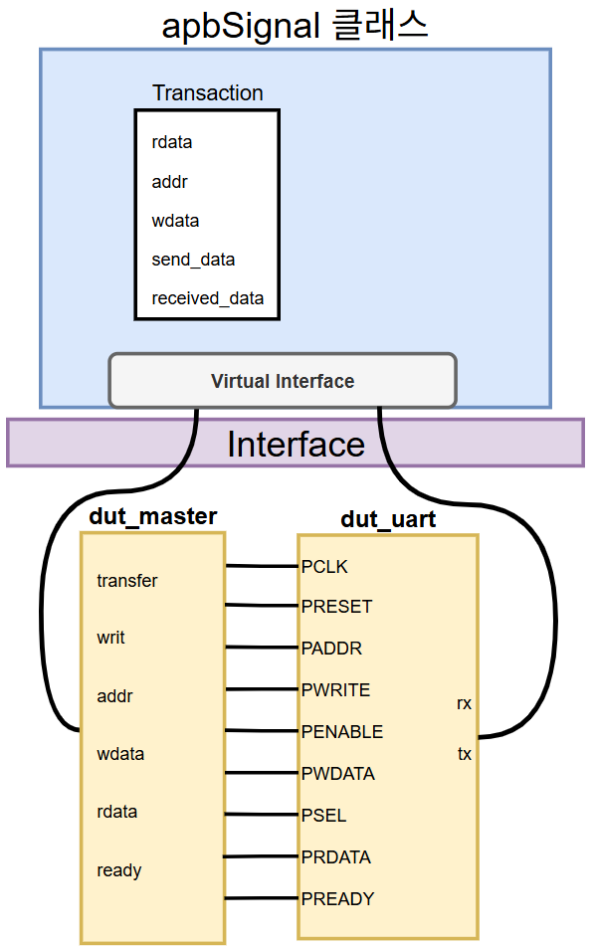
- **UART_peri**
  CPU가 APB 버스로 신호를 보냈을 때(PSEL), 함께 보낸 주소(PADDR) 확인.
  주소를 판단해서 올바른 레지스터 선택

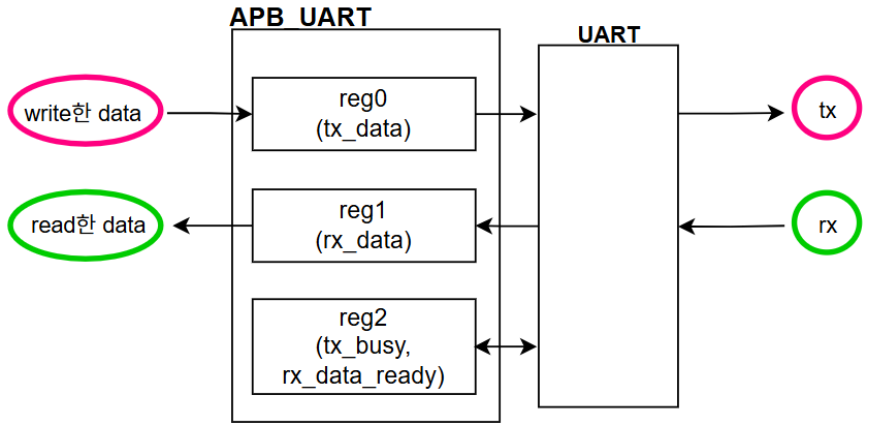
- **송신 레지스터 (reg0 / 0x10004000)**
  CPU가 apb_write로 쓴 데이터(PWDATA) 임시 저장.

- **수신 레지스터 (reg1 / 0x10004004)**
  외부 rx 핀에서 받은 데이터를 8비트로 완성해 저장.
  CPU는 apb_read 명령으로 수신한 데이터를 읽음.

- **UART**
  TX (송신) : reg0의 8비트 데이터를 가져와 tx 핀으로 1bit씩 쪼개 전송.
  RX (수신) : rx 핀으로 1bit씩 들어오는 신호를 8개 모아 reg1에 저장.

APB – 검증





1. TX 검증 : "APB로 쓴 값" vs "tx핀으로 받은 값"

보낸 값: 테스트벤치가 APB 버스로 reg0에 쓴 8bit 데이터. (t.wdata)

받은 값: DUT가 이 데이터를 tx 핀으로 1bit씩 쏘면 TB가 tx핀에서 8bit로 조립한 값. (t.received_data)

2. RX 검증 : "rx핀으로 쏜 값" vs "APB로 읽은 값"

보낸 값: 테스트벤치가 rx 핀으로 쏜 8bit 데이터. (t.send_data)

받은 값: DUT가 1bit 신호들을 8bit로 조립해 reg1에 저장하면 TB가 APB 버스로 reg1 데이터를 읽은 값. (t.rdata)

APB – 검증

```
PASS! TX: APB_wdata 0f == UART_receivedData 0f
PASS! RX: UART_send 0a == APB_rdata 0a

done PASS!
PASS! TX: APB_wdata 00 == UART_receivedData 00
PASS! RX: UART_send 7e == APB_rdata 7e

done PASS!
PASS! TX: APB_wdata 16 == UART_receivedData 16
PASS! RX: UART_send 63 == APB_rdata 63

done PASS!
PASS! TX: APB_wdata 4d == UART_receivedData 4d
PASS! RX: UART_send eb == APB_rdata eb

done PASS!
PASS! TX: APB_wdata 14 == UART_receivedData 14
PASS! RX: UART_send 31 == APB_rdata 31

done PASS!
PASS! TX: APB_wdata 49 == UART_receivedData 49
PASS! RX: UART_send 6c == APB_rdata 6c

done PASS!
```

```
PASS! TX: APB_wdata 53 == UART_receivedData 53
PASS! RX: UART_send e1 == APB_rdata e1

done PASS!
PASS! TX: APB_wdata 26 == UART_receivedData 26
PASS! RX: UART_send f8 == APB_rdata f8

done PASS!
PASS! TX: APB_wdata fe == UART_receivedData fe
PASS! RX: UART_send cb == APB_rdata cb

done PASS!
========== Test Report ==========
==       | Total | Pass  | Fail  ==
---------------------------------
== TX    |   512 |   512 |     0 ==
== RX    |   512 |   512 |     0 ==
== Done  |   512 |   512 |     0 ==
=================================
```

APB - C Test Code

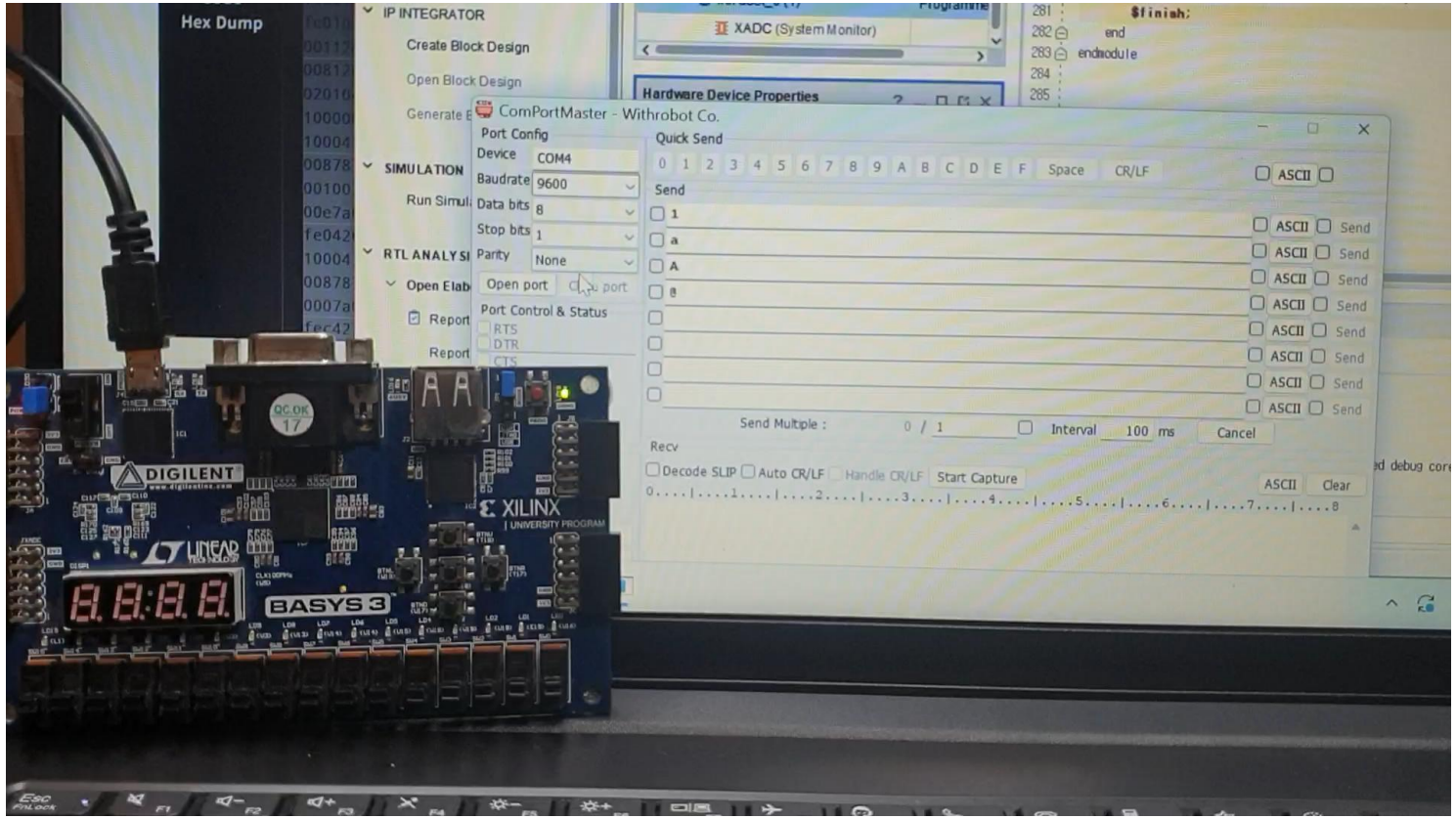
```c
#include <stdint.h>

// --- APB 버스 기본 주소 ---
#define APB_BASE_ADDR 0x10000000

// --- 1. GPO (LED 제어) 주소 정의 ---
#define GPO_OFFSET 0x1000
#define GPO_BASE_ADDR (APB_BASE_ADDR + GPO_OFFSET)
#define GPO_CR (*(volatile uint32_t*)(GPO_BASE_ADDR + 0x00))
#define GPO_ODR (*(volatile uint32_t*)(GPO_BASE_ADDR + 0x04))

// --- 2. UART 주소 정의 ---
#define UART_OFFSET 0x4000
#define UART_BASE_ADDR (APB_BASE_ADDR + UART_OFFSET)

// TX/RX/Status 레지스터 정의
#define UART_TX_DATA (*(volatile uint32_t*)(UART_BASE_ADDR + 0x00))  // 0x0 (TX)
#define UART_RX_DATA (*(volatile uint32_t*)(UART_BASE_ADDR + 0x04))  // 0x4 (RX)
#define UART_STATUS  (*(volatile uint32_t*)(UART_BASE_ADDR + 0x08))  // 0x8 (Status)

#define UART_STATUS_TX_BUSY (1 << 0)   // [0]번 비트: tx_busy
#define UART_STATUS_RX_READY (1 << 1)  // [1]번 비트: rx_data_ready

// --- 함수 프로토타입 ---
void System_init();
char uart_getc();        // 1글자 수신
void uart_putc(char c);  // 1글자 송신
```

```c
int main() {
    System_init();
    int temp;
    UART_STATUS = 0x01;

    while (1) {
        temp = 0;

        UART_STATUS = 0x00;  // tx_start = 1
        temp++;
        UART_STATUS = 0x01;  // tx_start = 0
        for (int j = 0; j < 1000000; j++) {
            if (UART_STATUS == 0x02 |
                UART_STATUS == 0x03)  // reg[1] = 1이면 (= rx_done을 받으면)
            {
                UART_TX_DATA = UART_RX_DATA;  // rx값을 tx값에 대입
                GPO_ODR = UART_RX_DATA;       // GPO의 output을 rx값으로
                for (int k = 0; k < 2000000; k++) {
                    temp++;
                }
            }
        }
    }
    return 0;
}

void System_init() {
    GPO_CR = 0xFFFF;
    GPO_ODR = 0x0000;
}
```

- 입력한 문자에 해당하는 ascii 값에 맞게 LED가 켜짐.
- Comportmaster에 그 값이 echo되어 주기적으로 display.

APB - C Test Code

동작영상



1 : 8'b0011_0001
a : 8'b0110_0001
A : 8'b0100_0001
@ : 8'b0100_0000

고찰

- 보드 동작(C) ≠ 설계 정확성(TB)

  C코드로 보드 테스트 시, 입력한 단어의 ascii 값에 맞게 LED가 켜져 설계에 맞는 동작을 확인함.

  그러나, testbench를 통해 검증해본 결과 데이터가 불일치하는 문제가 발생함.

  하드웨어에서 정상적으로 작동하더라도 사실은 옳지 않은 동작일 수 있음을 깨달음.

  설계(RTL) + 검증(TB) + 펌웨어가 모두 맞물려야 하나의 시스템이 완성됨을 느낌.

- Multicycle 구조를 Pipelined 구조로 개선하고 데이터 처리 로직을 구현하여 CPU 성능을 향상하고 싶음.

감사합니다.