RISC-V based AMBA APB Peripheral Design

하만 세미콘 아카데미 2기 김은성

2025.10.27

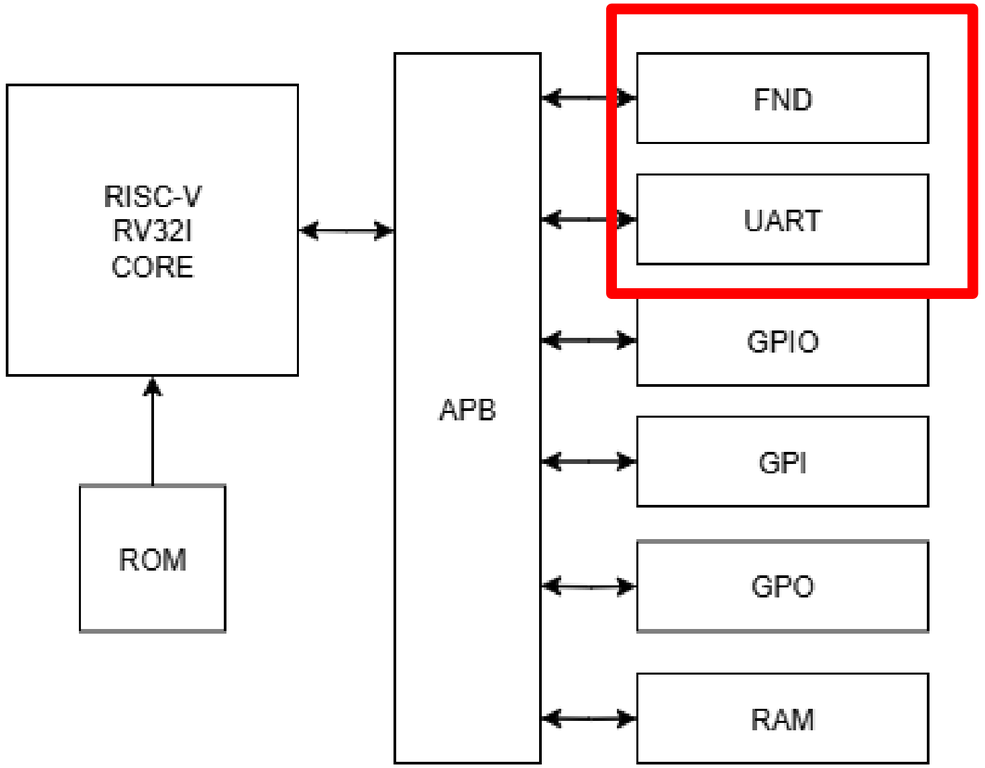
Contents

Introduction

Introduction

Project Goal

- **RISC-V Multi-Cycle CPU Core** 설계 및 구현
- AMBA APB Protocol 기반하여 **다양한 Peripheral 연결**
  - UART, FND Controller …
- Systemverilog Testbench를 활용하여 **UART Peripheral 검증**
- **C 언어 기반 Application code** 작성하고 컴파일하여 ROM에 올린 후 CPU에서 작동 확인



Introduction

RISC-V

- **R**educed **I**nstruction **S**et **C**omputer
- 오픈 소스 명령어 집합 구조 (Instruction Set Architecture)
- 하드웨어 및 소프트웨어 설계에 유연성을 제공

RV32I

- 'RV' **RISC-V**, '32' **32 bit** , 'I' **Integer** (정수)
- **RV32I Instruction Formats** : RISC-V의 기본 명령어 집합

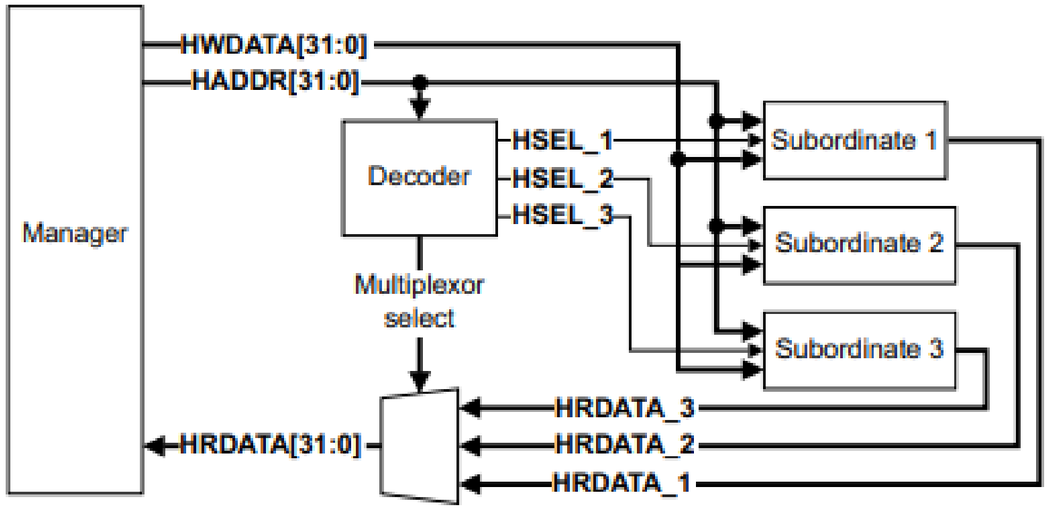
Multicycle

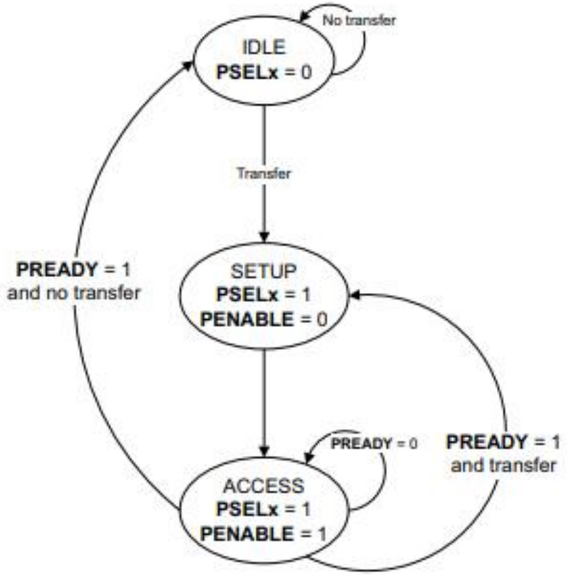
- 하나의 명령어를 여러 개의 짧은 클럭 사이클에 걸쳐 나누어 실행하는 CPU 구현 방식
- 각 명령어 단계(Fetch, Decode, Execute 등)가 한 클럭 사이클씩 차지

Introduction

APB

- **A**dvanced **P**eripheral **B**us
- ARM **AMBA** (Advanced Microcontroller Bus Architecture) Protocol 중 하나
- 간단한 인터페이스 및 **Non-pipeline** 구조

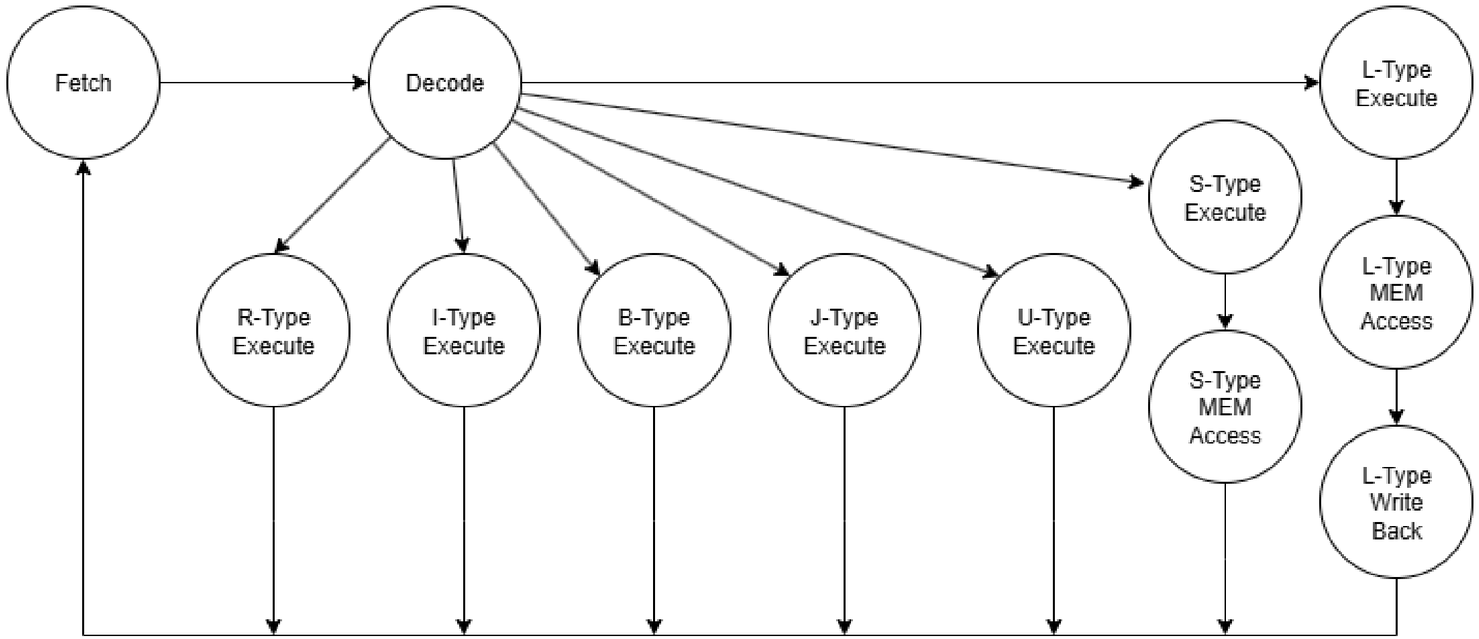




RISC-V RV32I Multi-Cycle

RISC-V RV32I Multi-Cycle Design

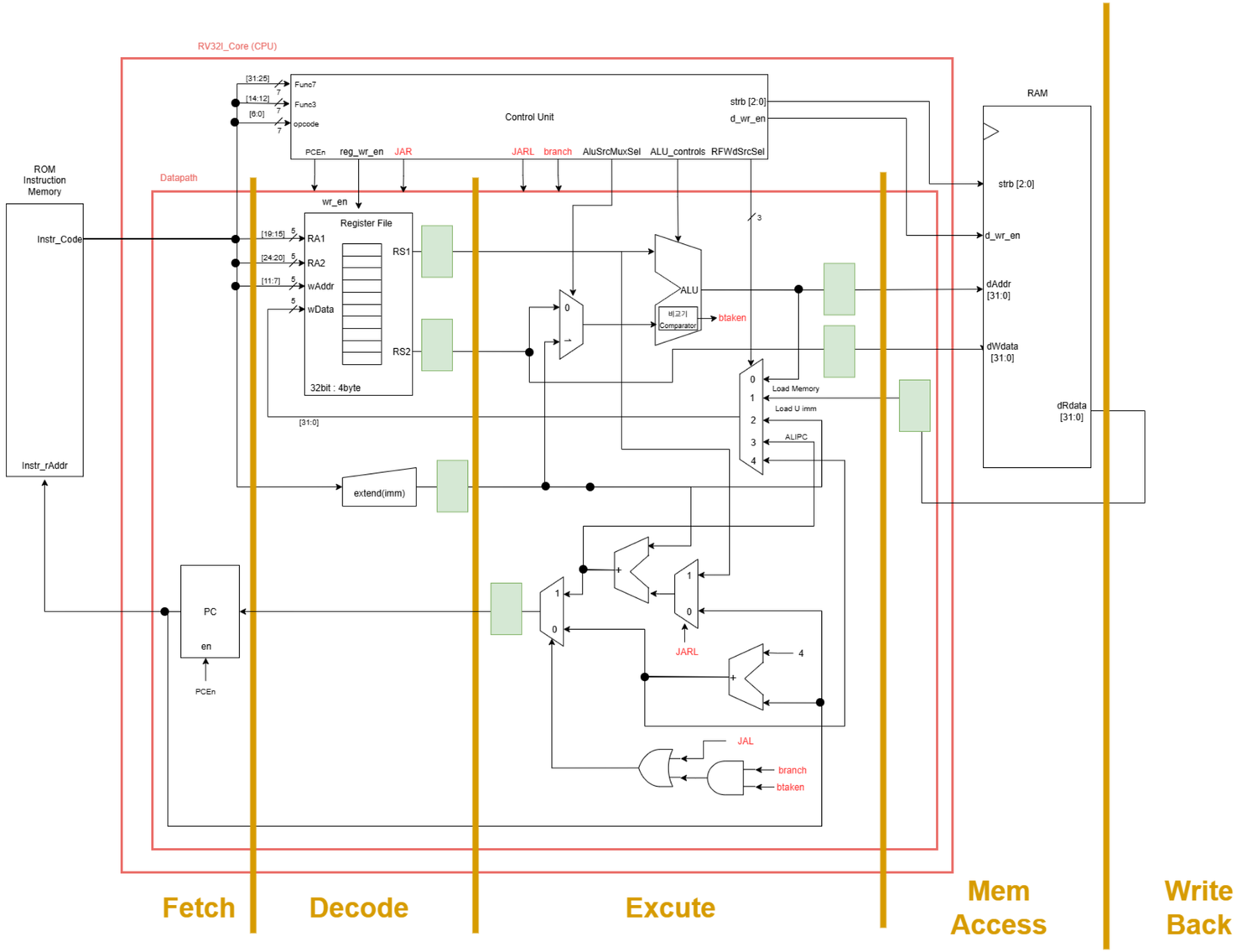
Control Unit FSM



- **Fetch**: 메모리(ROM)에서 PC가 가리키는 주소의 명령어 가져오기
- **Decode**: 가져온 명령어를 해석하고, 레지스터 값을 미리 읽어 실행 준비하기
- **(R/I/B/J/U-Type) Execute**: ALU를 사용하여 명령어 타입에 맞는 연산(산술, 논리, 비교, 주소 계산) 수행하기
- **(L/S-Type) Execute**: ALU를 사용하여 데이터를 읽거나 쓸 메모리 주소(Base + Offset) 계산하기
- **(S-Type) MEM Access**: 계산된 메모리 주소에 레지스터(rs2)의 데이터를 저장하기 (Store)
- **(L-Type) MEM Access**: 계산된 메모리 주소에서 데이터를 읽어오기 (Load)
- **(L-Type) Write Back**: 메모리에서 읽어온 데이터를 목적지 레지스터(rd)에 최종 저장하기

RISC-V RV32I Multi-Cycle Design

Block Diagram



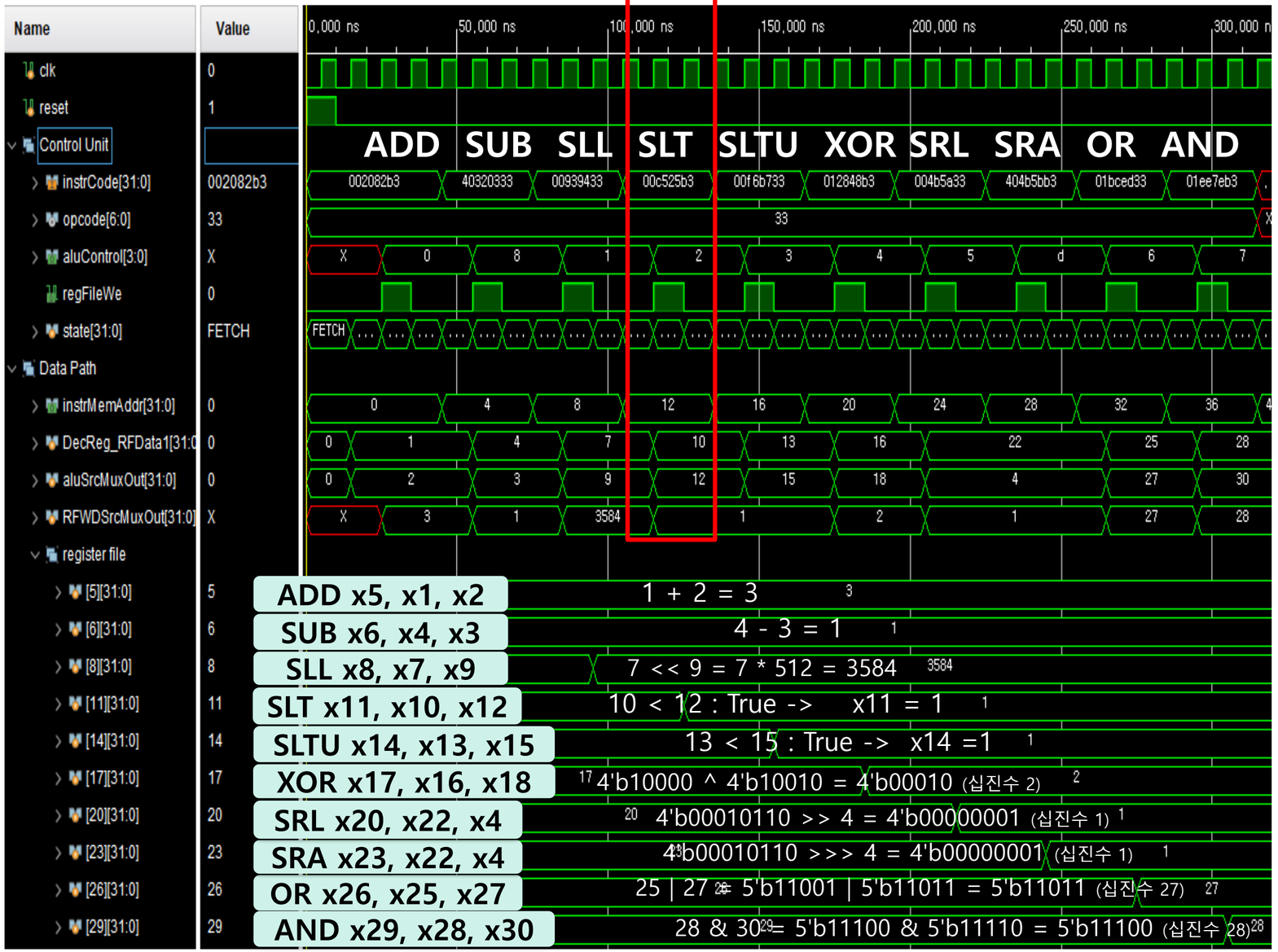
R – Type Simulation

2개의 source register(rs1, rs2) 값을
사용하여 연산
-> 연산 결과를 목적지 register (rd)에 저장

Type	Mnemonic	Descript	Extension
R	ADD	rd = rs1 + rs2	
	SUB	rd = rs1 - rs2	
	SLL	rd = rs1 << rs2	
	SLT	rd = (rs1 < rs2)?1:0	
	SLTU	rd = (rs1 < rs2)?1:0	zero-extends
	XOR	rd = rs1 ^ rs2	
	SRL	rd = rs1 >> rs2	
	SRA	rd = rs1 >> rs2	msb-extends
	OR	rd = rs1 | rs2	
	AND	rd = rs1 & rs2	

3 Cycle

Register : Fetch – Decode – R_EXE



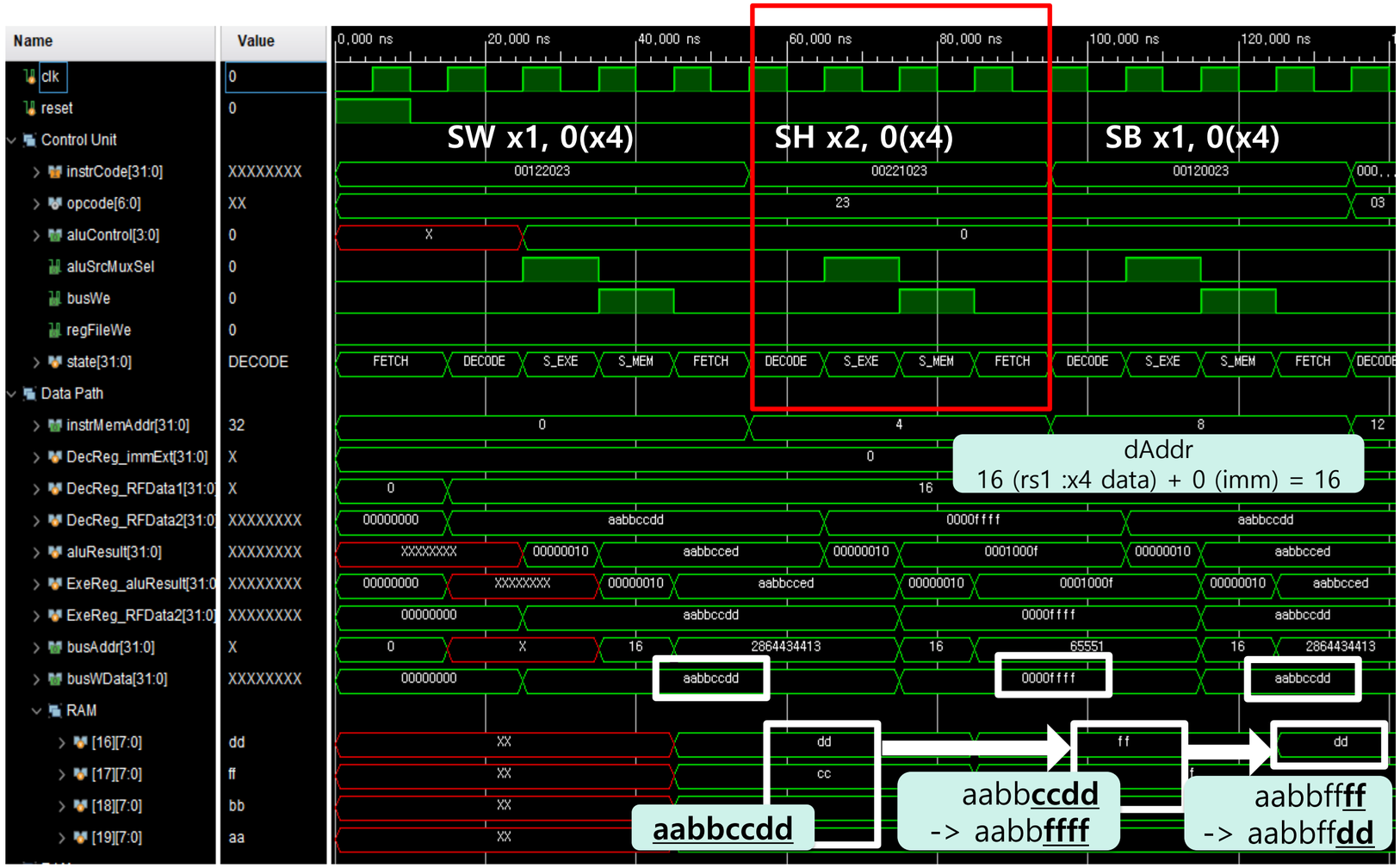
S – Type Simulation

STORE : Fetch – Decode – S_EXE – S_MEM

4 Cycle

특정 레지스터(rs2)에 있는 데이터를
메모리에 저장(Store)

Type	Mnemonic	Descript
S	SB	M[rs1+imm][0:7] = rs2[0:7]
	SH	M[rs1+imm][0:15] = rs2[0:15]
	SW	M[rs1+imm][0:31] = rs2[0:31]



Store Word

RAM에
Wdata 저장

Store Half-Word

RAM에
Wdata 16bit저장

Store Byte

RAM에
Wdata 8bit저장

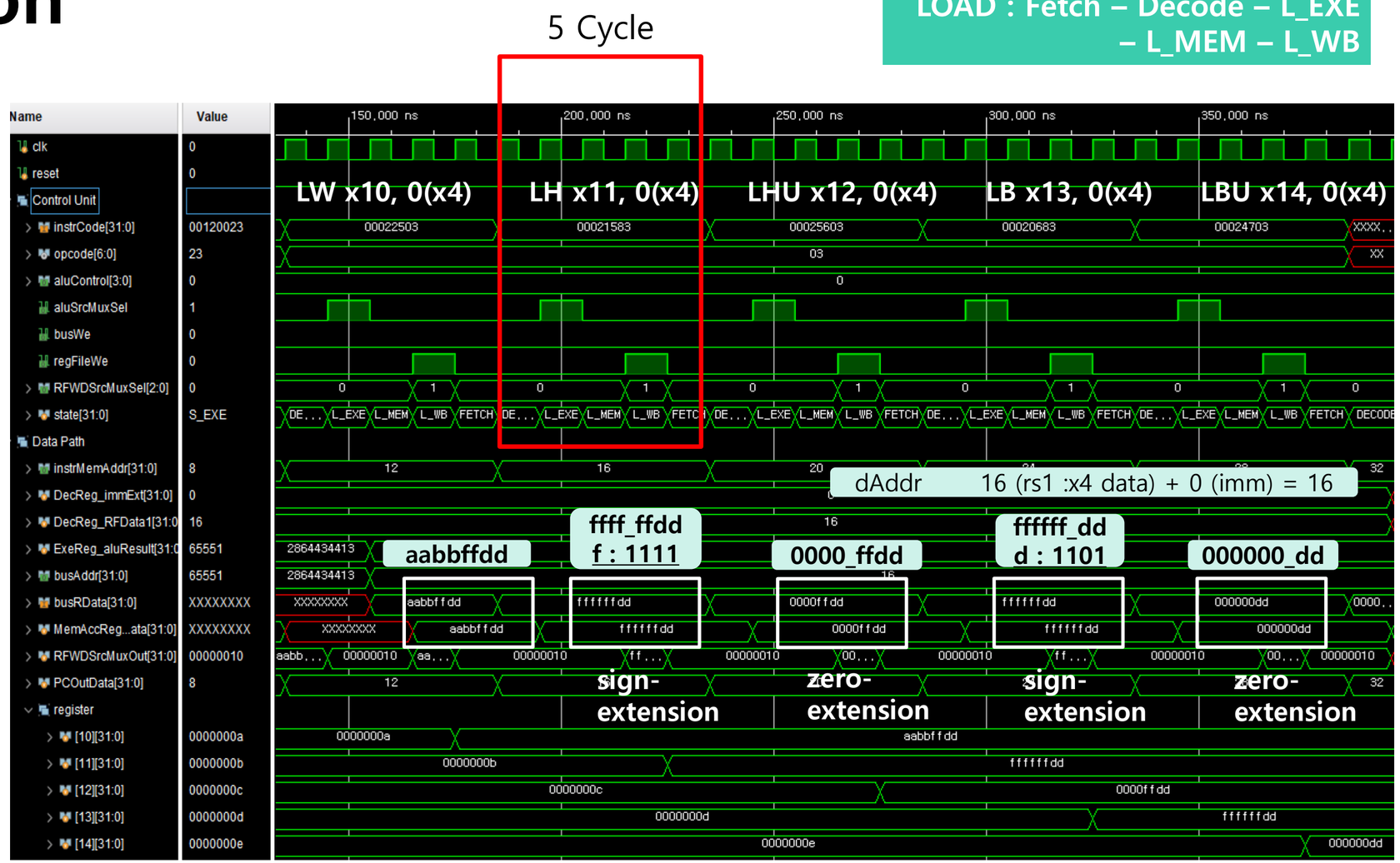
L – Type Simulation

LOAD : Fetch – Decode – L_EXE – L_MEM – L_WB

베이스 주소(rs1)에 12비트 offset을 더하여 최종 메모리 주소를 계산

-> 해당 주소에서 데이터를 읽어와서 목적지 레지스터(rd)에 로드(Load)

Type	Mnemonic	Descript	Extension
IL	LB	rd = M[rs1+imm][0:7]	
	LH	rd = M[rs1+imm][0:15]	
	LW	rd = M[rs1+imm][0:31]	
	LBU	rd = M[rs1+imm][0:7]	zero-extends
	LHU	rd = M[rs1+imm][0:15]	zero-extends



Load Word

32bit data load

Load Half-Word

하위 16bit load + sign-extension

Load Half-Word Unsigned

하위 16bit load + zero-extension

Load Byte

하위 8bit load + sign-extension

Load Byte Unsigned

하위 8bit load + zero-extension

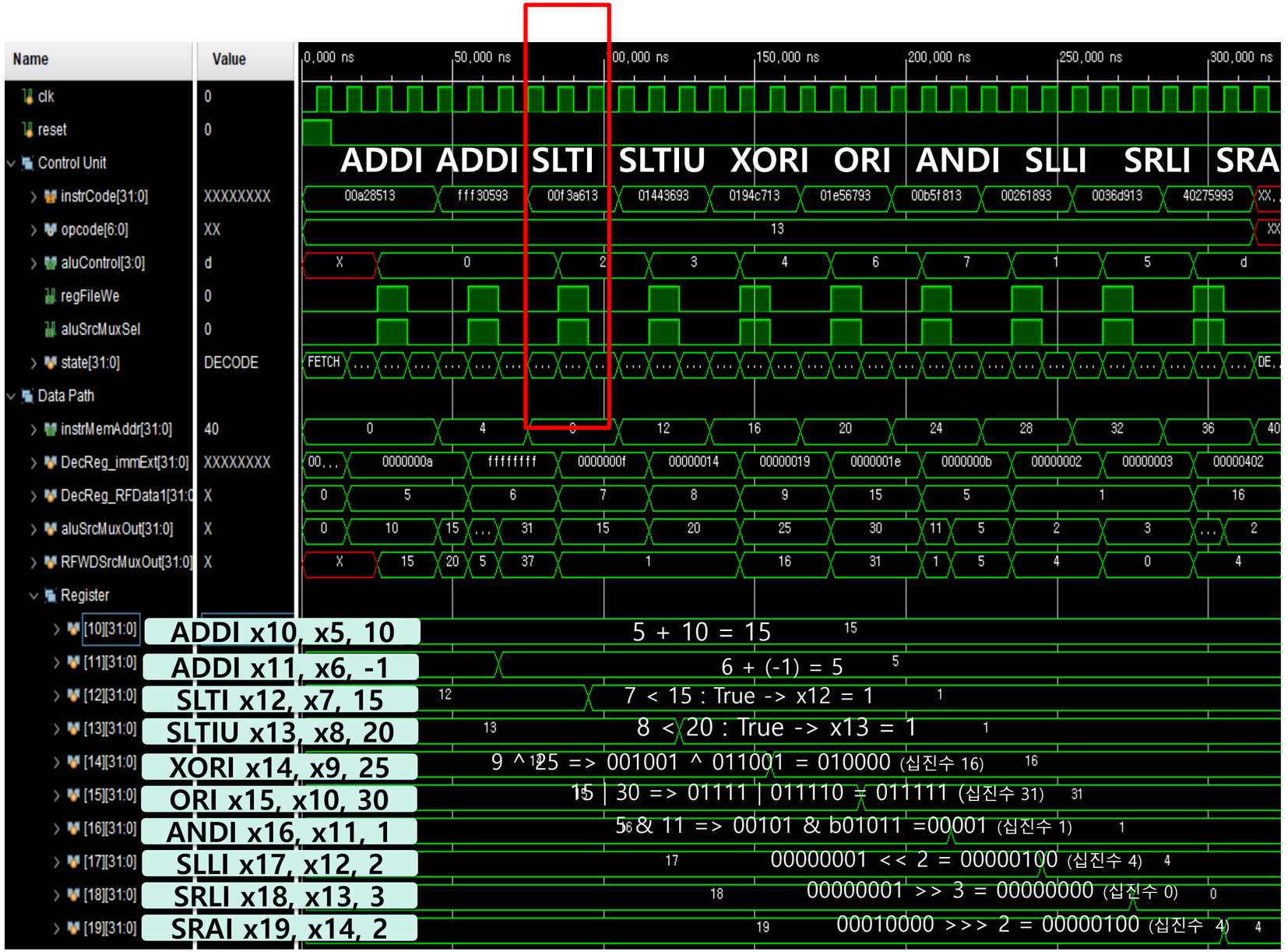
I – Type Simulation

3 Cycle

Immediate : Fetch – Decode – I_EXE

source register(rs1)와
12비트 immediate를 사용하여
산술/논리 연산을 수행
-> 결과를 목적지 register (rd)에 저장

Type	Mnemonic	Descript	Extension
I	ADDI	rd = rs1 + imm	
	SLTI	rd = (rs1 < imm)?1:0	
	SLTIU	rd = (rs1 < imm)?1:0	zero-extends
	XORI	rd = rs1 ^ imm	
	ORI	rd = rs1 | imm	
	ANDI	rd = rs1 & imm	
	SLLI	rd = rs1 << imm[0:4]	
	SRLI	rd = rs1 >> imm[0:4]	
	SRAI	rd = rs1 >> imm[0:4]	msb-extends

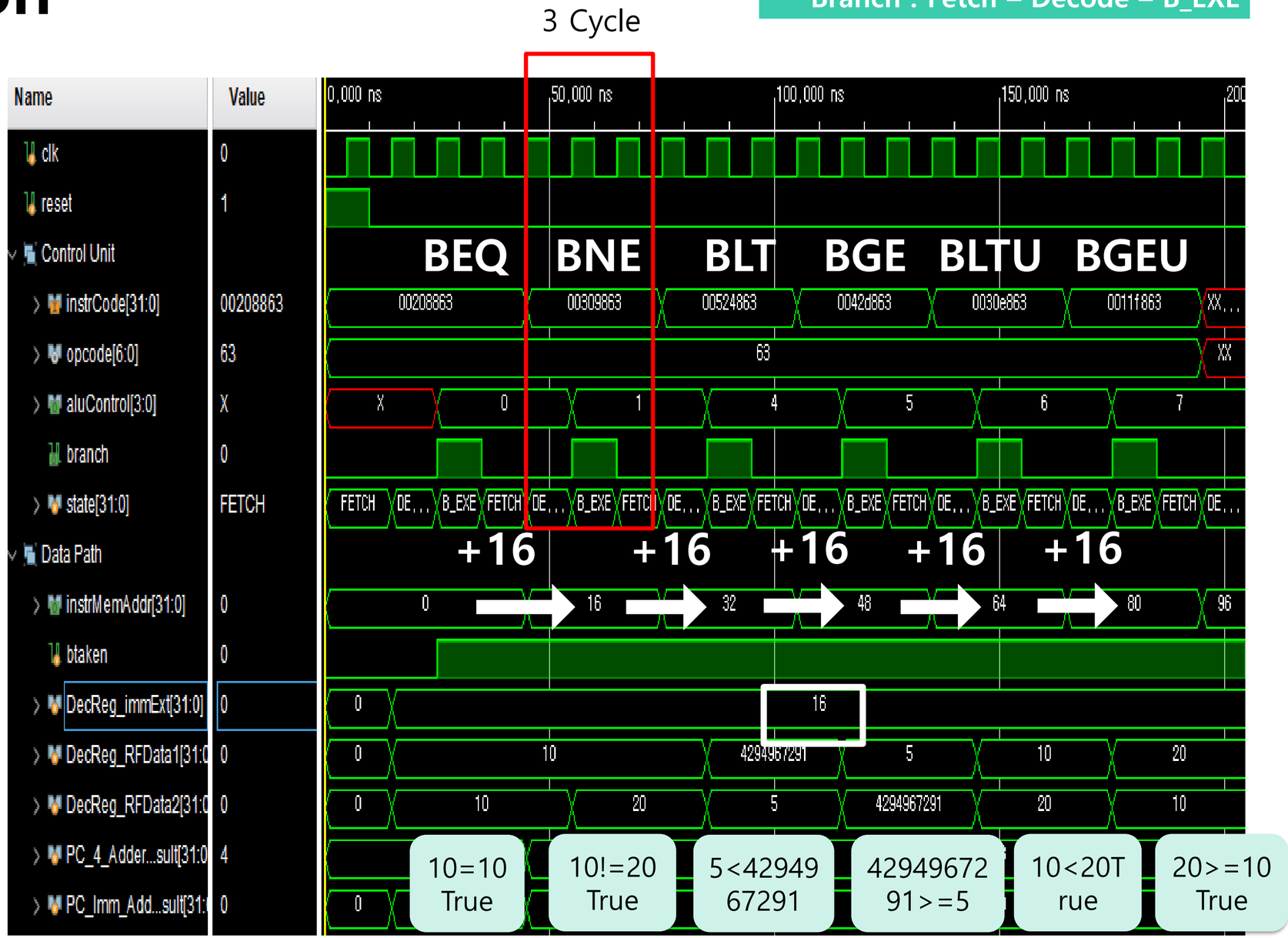


B – Type Simulation

Branch : Fetch – Decode – B_EXE

2개의 source register(rs1, rs2)를 비교하여 특정 조건이 만족되면 프로그램 카운터(PC)를 변경해 특정 주소로 분기

Type	Mnemonic	Descript	Extension
B	BEQ	if(rs1 == rs2) PC += imm	
	BNE	if(rs1 != rs2) PC += imm	
	BLT	if(rs1 < rs2) PC += imm	
	BGE	if(rs1 >= rs2) PC += imm	
	BLTU	if(rs1 < rs2) PC += imm	zero-extends
	BGEU	if(rs1 >= rs2) PC += imm	zero-extends



LU/AU – Type Simulation

LUI : Fetch – Decode – LU_EXE
AUIPC : Fetch – Decode – AU_EXE

LUI(Load Upper Immediate)

20비트의 immediate를 register의 상위 비트(Upper bits)에 로드, 나머지 하위 12비트는 0으로 채우기

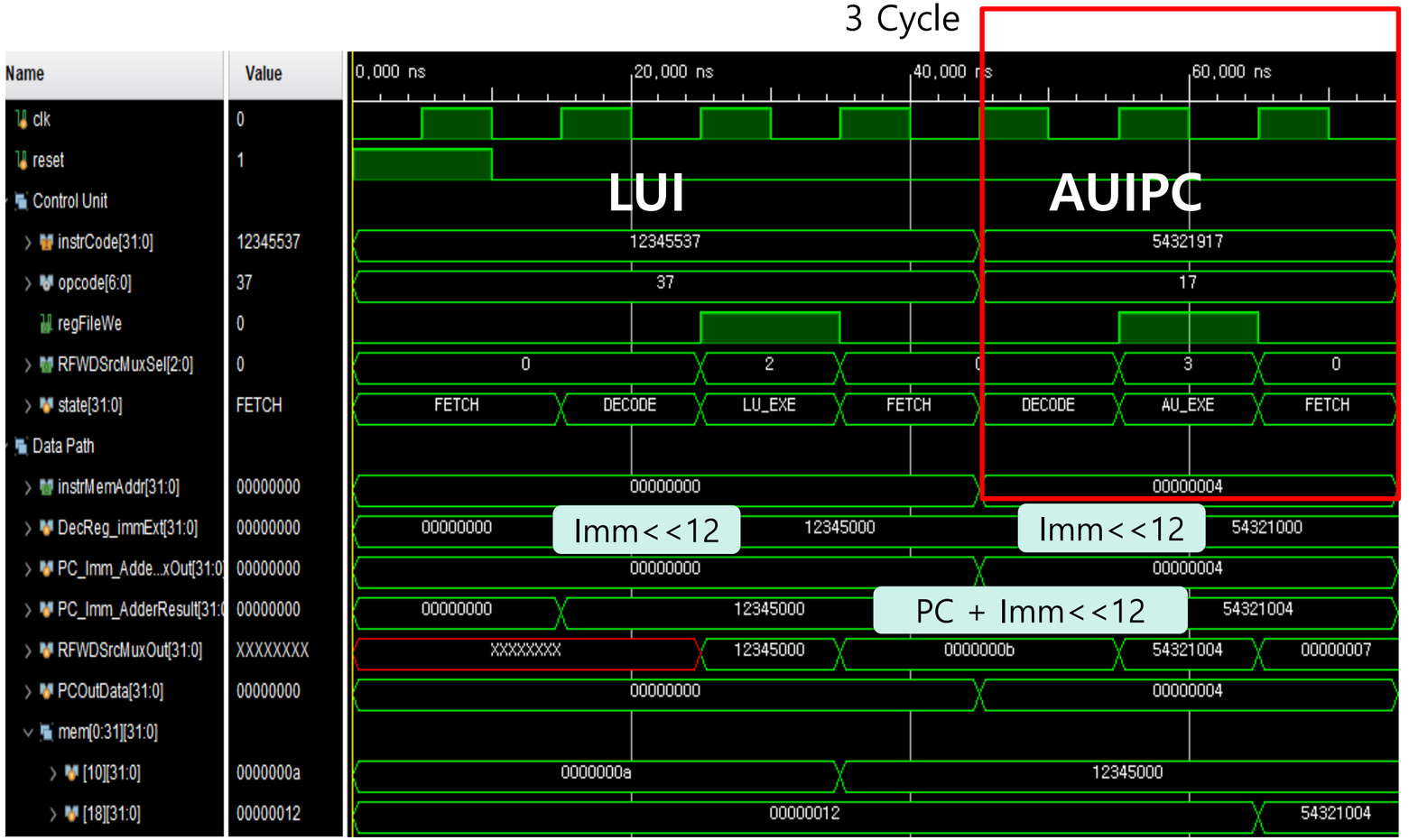
AUIPC(Add Upper Immediate to PC)

immediate를 12비트 왼쪽 시프트 후 그 값을 PC 값에 더하기

-> 그 결과를 목적지 register에 저장

Type	Mnemonic	Descript
U	LUI	rd = imm << 12
	AUIPC	rd = PC + (imm<<12)

3 Cycle



lui x10, 0x12345

32'h12345537
-> 32'h12345000

auipc x19, 0x54321

PC + (imm << 12)
= 0x4 + (0X54321 << 12)
= 0x4 + 0x54321000
= 0x54321004

J/JL – Type Simulation

JAL (Jump and Link)

현재 PC에 immediate 오프셋을 더한 주소로 점프하며, 복귀 주소(PC+4)를 rd에 저장

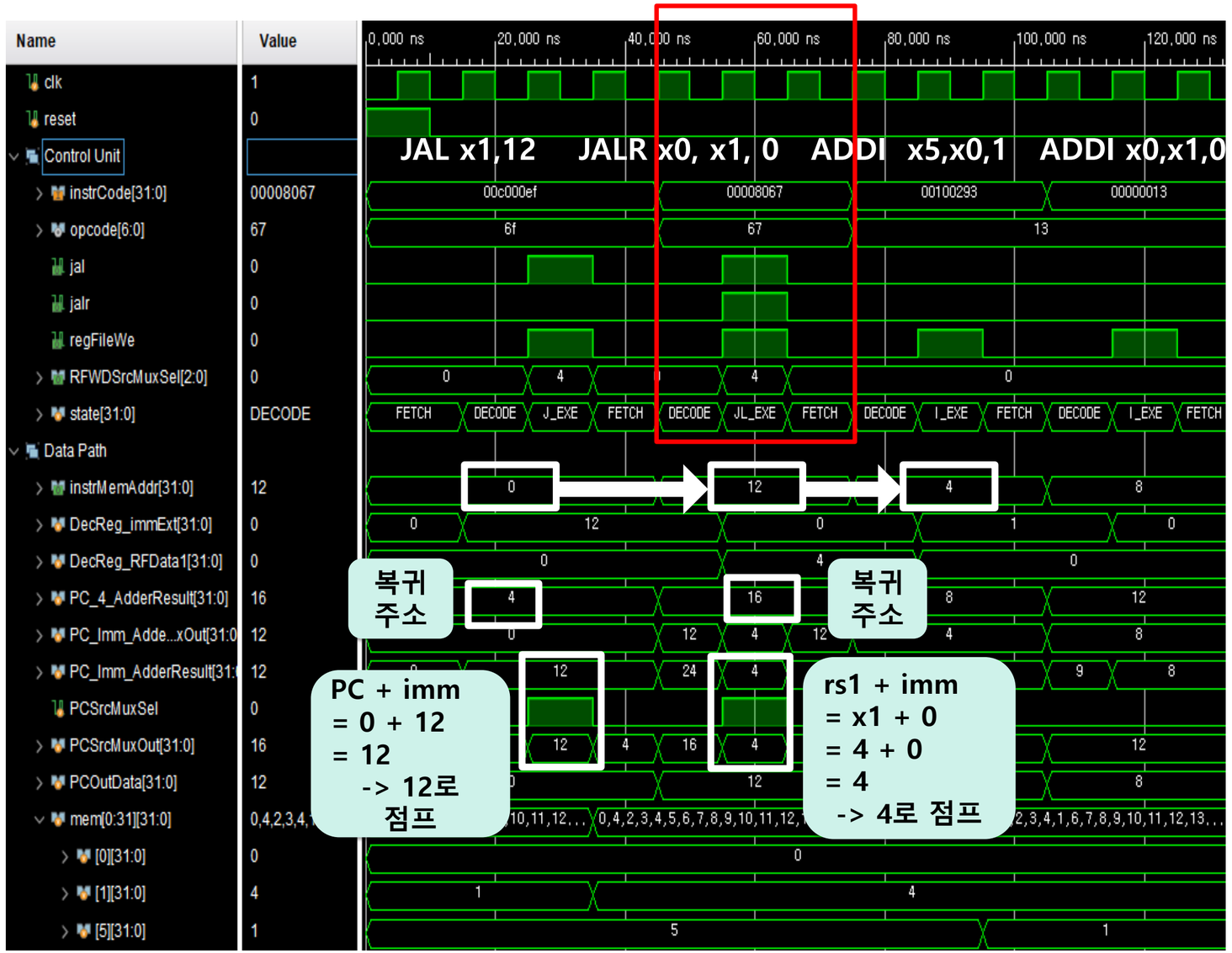
JALR (Jump and Link Register)

rs1 레지스터 값에 immediate 오프셋을 더한 주소로 점프하며, 복귀 주소(PC+4)를 rd에 저장

Type	Mnemonic	Descript
J/JI	JAL	rd = PC+4; PC += imm
	JALR	rd = PC+4; PC = rs1 + imm

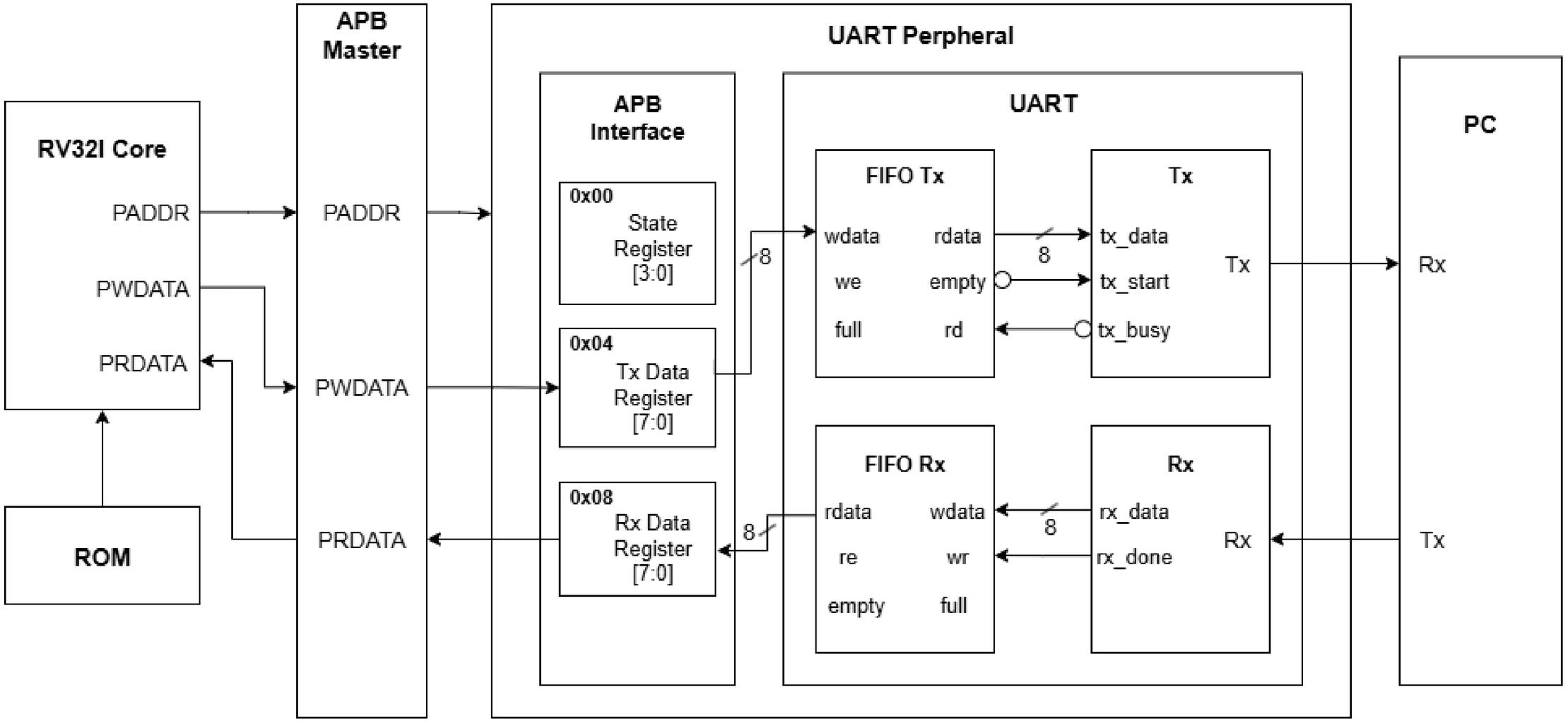
JAL : Fetch – Decode – J_EXE
JALR : Fetch – Decode – JL_EXE

3 Cycle

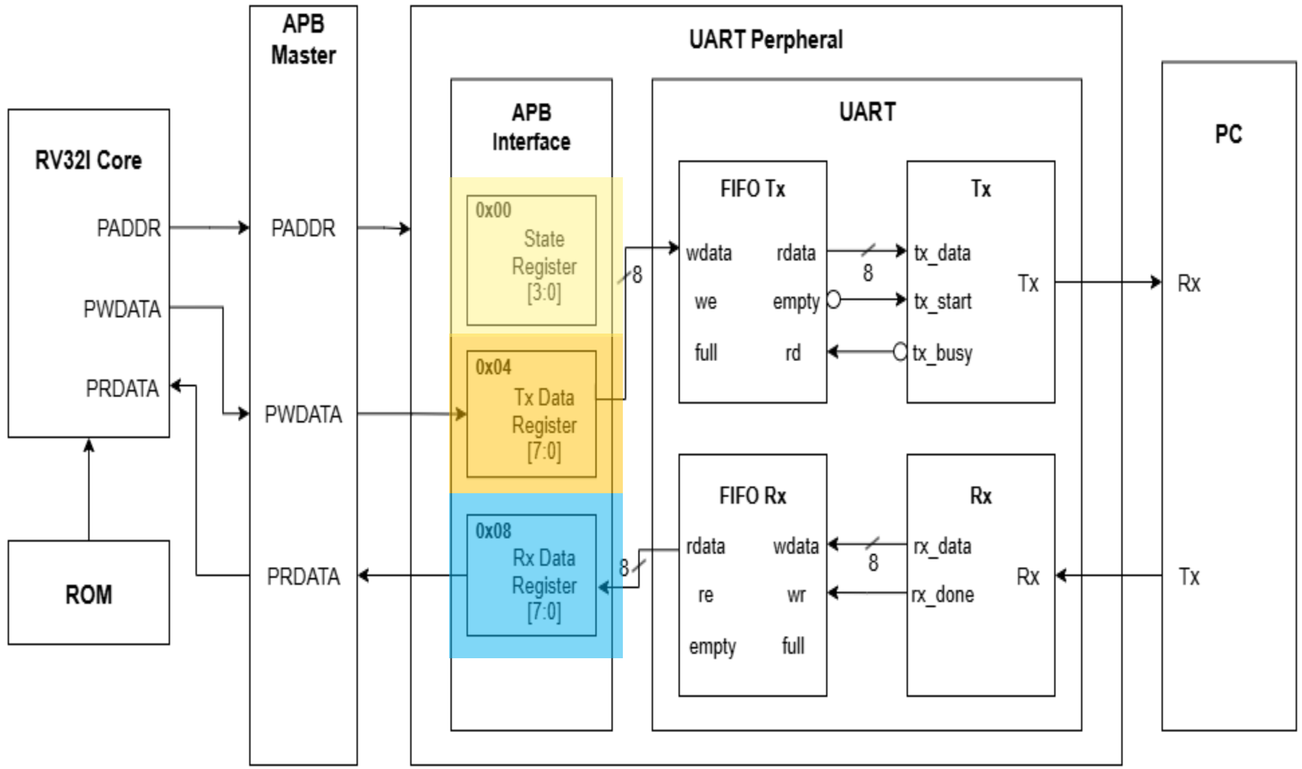


UART Peripheral Design

UART Peripheral Block Diagram



UART Peripheral Block Diagram




```
assign USTATEREG = {
    w_rx_fifo_full, w_tx_fifo_empty, w_tx_fifo_full, w_rx_fifo_empty
};
assign w_tx_wdata = UWDATA;
assign URDATA = w_rx_rdata;
```

0x00 Status Register

UART FIFO full/empty 정보 담고 있는 Register

0x04 Tx Data Register

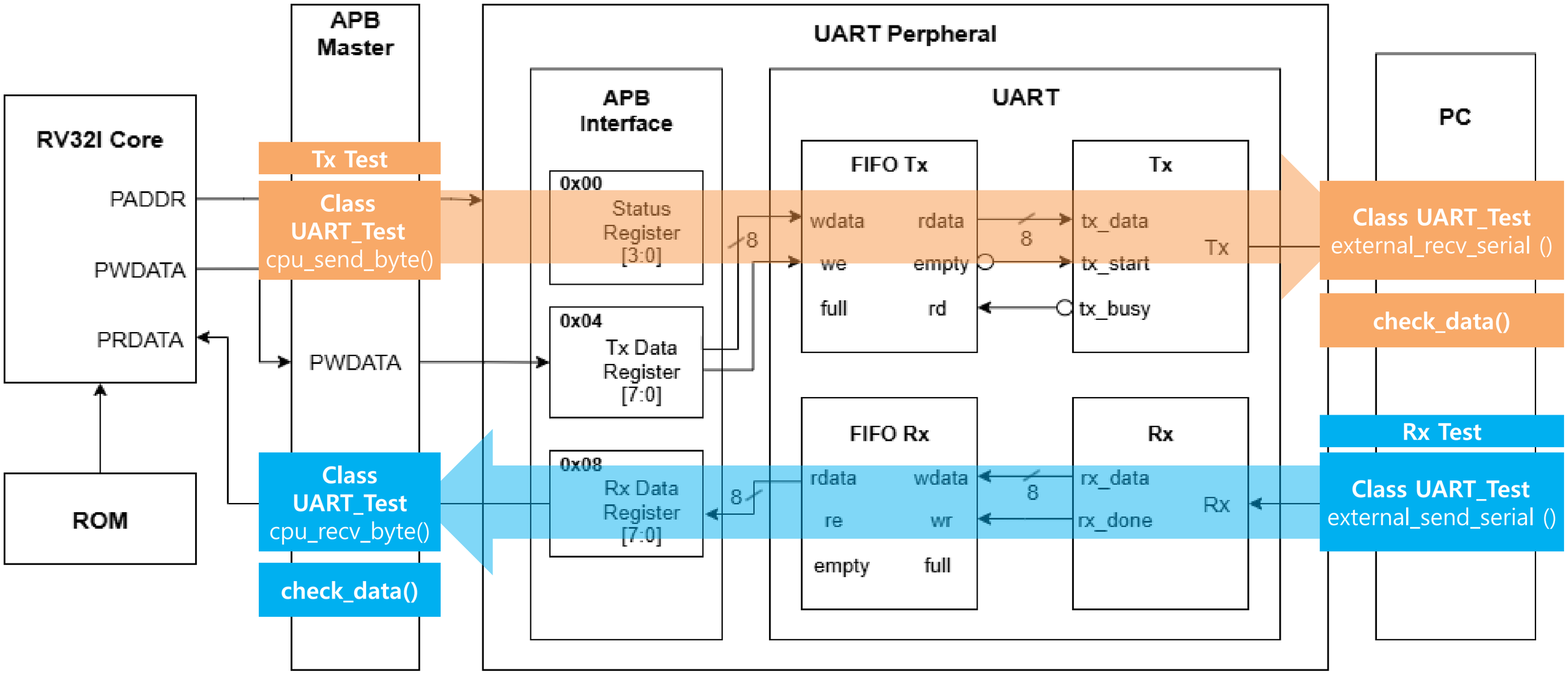
CPU가 송신(TX)하는 8비트 데이터를
APB 버스로부터 받아서
UART의 TX FIFO로 전달하는 쓰기 전용 Register

0x08 Rx Data Register

외부로부터 수신(RX)한 8비트 데이터를
CPU가 읽어갈 수 있도록
APB 버스로 전달해주는 읽기 전용 Register

UART Peripheral Verification

Class-based Verification

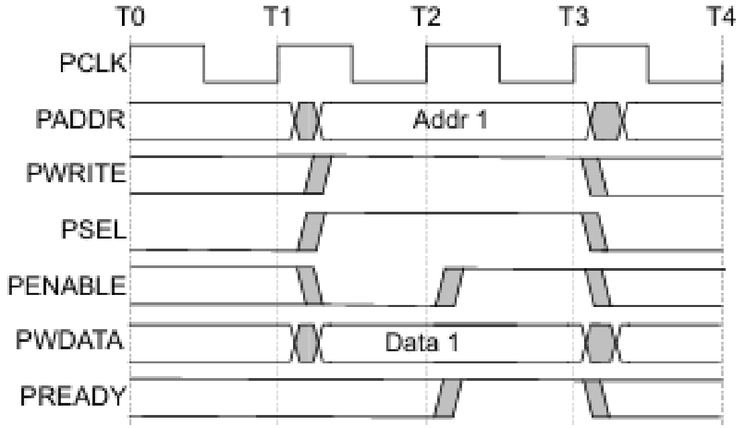


Class-based Verification

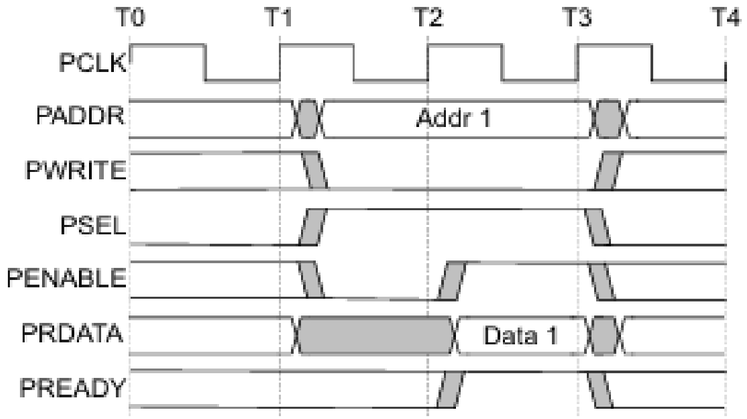
– APB Timing Diagram based Write / Read Task

```
task automatic apb_write_reg(input logic [3:0] addr, input logic [31:0] data);
    tb_if.PSEL    <= 1'b1;
    tb_if.PADDR   <= addr;
    tb_if.PWRITE  <= 1'b1;
    tb_if.PWDATA  <= data;
    tb_if.PENABLE <= 1'b0; // SETUP
    @(posedge tb_if.PCLK);
    tb_if.PENABLE <= 1'b1; // ACCESS
    wait (tb_if.PREADY == 1'b1);
    @(posedge tb_if.PCLK);
    tb_if.PSEL    <= 1'b0;
    tb_if.PENABLE <= 1'b0;
endtask

task automatic apb_read_reg(input logic [3:0] addr, output logic [31:0] rdata);
    tb_if.PSEL    <= 1'b1;
    tb_if.PADDR   <= addr;
    tb_if.PWRITE  <= 1'b0;
    tb_if.PENABLE <= 1'b0; // SETUP
    @(posedge tb_if.PCLK);
    tb_if.PENABLE <= 1'b1; // ACCESS
    wait (tb_if.PREADY == 1'b1);
    rdata = tb_if.PRDATA;
    @(posedge tb_if.PCLK);
    tb_if.PSEL    <= 1'b0;
    tb_if.PENABLE <= 1'b0;
endtask
```

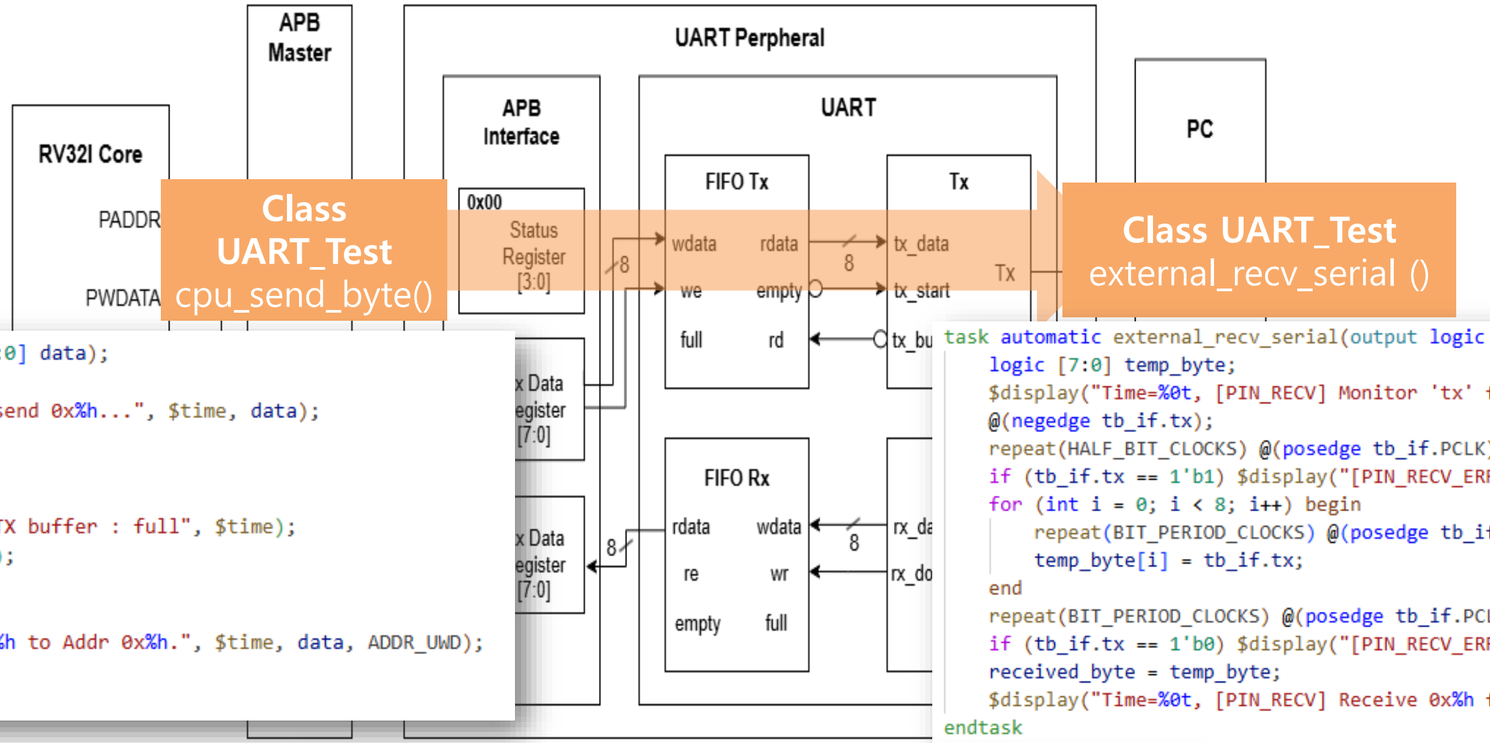


Write protocol



Read protocol

Class-based Verification – Tx Test (APB → Tx)



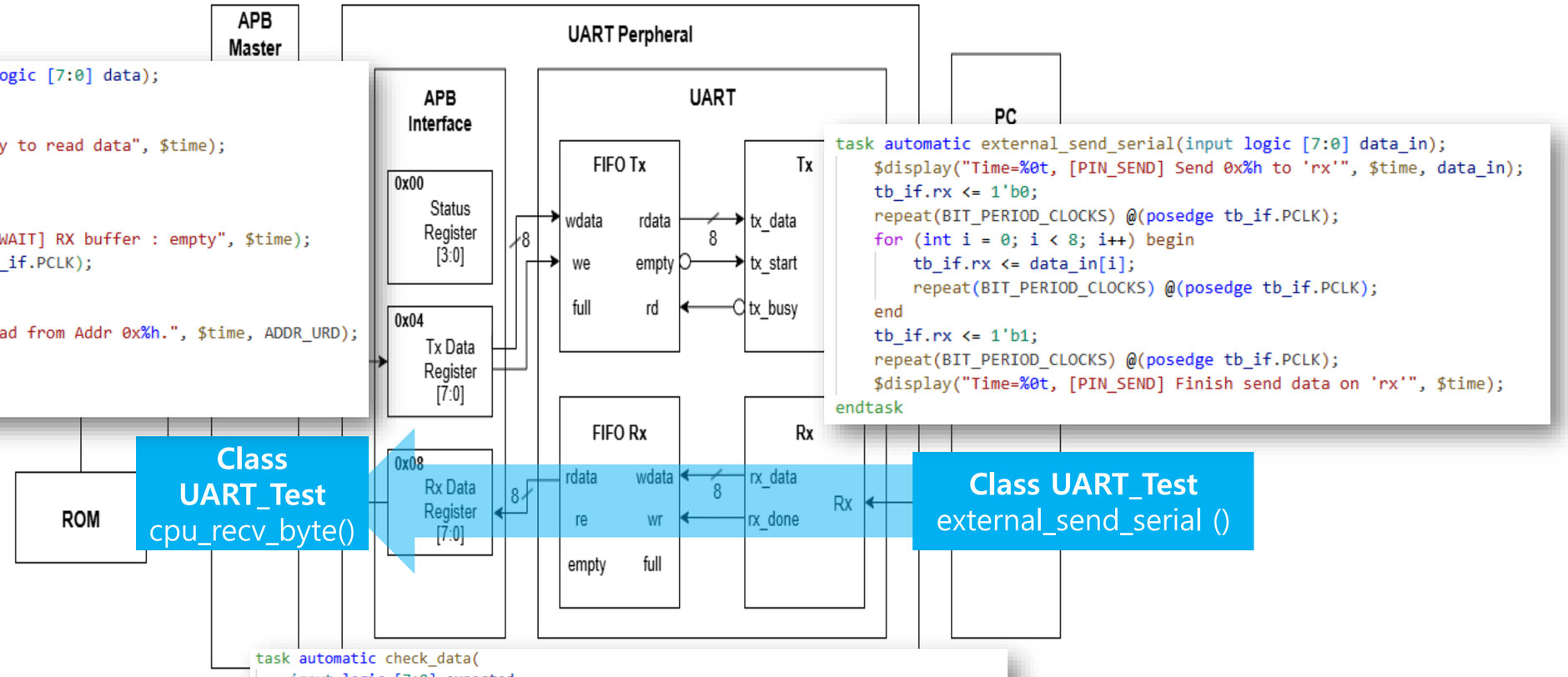

```
task automatic cpu_send_byte(input logic [7:0] data);
    logic [31:0] usr;
    $display("Time=%0t, [CPU_WRITE] Try to send 0x%h...", $time, data);
    do begin
        apb_read_reg(ADDR_USR, usr);
        if (usr[1]) begin
            $display("Time=%0t, [CPU_WAIT] TX buffer : full", $time);
            repeat(10) @(posedge tb_if.PCLK);
        end
    end while (usr[1] == 1'b1);
    $display("Time=%0t, [CPU_WRITE] Send 0x%h to Addr 0x%h.", $time, data, ADDR_UWD);
    apb_write_reg(ADDR_UWD, data);
endtask
```

```
task automatic external_recv_serial(output logic [7:0] received_byte);
    logic [7:0] temp_byte;
    $display("Time=%0t, [PIN_RECV] Monitor 'tx' for data", $time);
    @(negedge tb_if.tx);
    repeat(HALF_BIT_CLOCKS) @(posedge tb_if.PCLK);
    if (tb_if.tx == 1'b1) $display("[PIN_RECV_ERROR]");
    for (int i = 0; i < 8; i++) begin
        repeat(BIT_PERIOD_CLOCKS) @(posedge tb_if.PCLK);
        temp_byte[i] = tb_if.tx;
    end
    repeat(BIT_PERIOD_CLOCKS) @(posedge tb_if.PCLK);
    if (tb_if.tx == 1'b0) $display("[PIN_RECV_ERROR]");
    received_byte = temp_byte;
    $display("Time=%0t, [PIN_RECV] Receive 0x%h from 'tx'", $time, received_byte);
endtask
```

check_data()

```
task automatic check_data(
    input logic [7:0] expected,
    input logic [7:0] actual,
    input string      test_name
);
    $display("----------------------------------------");
    this.test_total_count++;
    if (expected == actual) begin
        $display("Time=%0t, [PASS] %s", $time, test_name);
        $display("                  (Expect: 0x%h, Actual: 0x%h)", expected, actual);
        this.test_pass_count++;
    end else begin
        $display("Time=%0t, [FAIL] %s", $time, test_name);
        $display("                  (Expect: 0x%h, Actual: 0x%h)", expected, actual);
        this.test_fail_count++;
    end
endtask
```

Class-based Verification – Rx Test (Rx → APB)




```
task automatic cpu_recv_byte(output logic [7:0] data);
    logic [31:0] usr;
    logic [31:0] rdata;
    $display("Time=%0t, [CPU_READ] Try to read data", $time);
    do begin
        apb_read_reg(ADDR_USR, usr);
        if (usr[0]) begin
            $display("Time=%0t, [CPU_WAIT] RX buffer : empty", $time);
            repeat(2000) @(posedge tb_if.PCLK);
        end
    end while (usr[0] == 1'b1);
    $display("Time=%0t, [CPU_READ] Read from Addr 0x%h.", $time, ADDR_URD);
    apb_read_reg(ADDR_URD, rdata);
    data = rdata[7:0];
endtask
```

```
task automatic external_send_serial(input logic [7:0] data_in);
    $display("Time=%0t, [PIN_SEND] Send 0x%h to 'rx'", $time, data_in);
    tb_if.rx <= 1'b0;
    repeat(BIT_PERIOD_CLOCKS) @(posedge tb_if.PCLK);
    for (int i = 0; i < 8; i++) begin
        tb_if.rx <= data_in[i];
        repeat(BIT_PERIOD_CLOCKS) @(posedge tb_if.PCLK);
    end
    tb_if.rx <= 1'b1;
    repeat(BIT_PERIOD_CLOCKS) @(posedge tb_if.PCLK);
    $display("Time=%0t, [PIN_SEND] Finish send data on 'rx'", $time);
endtask
```

check_data()

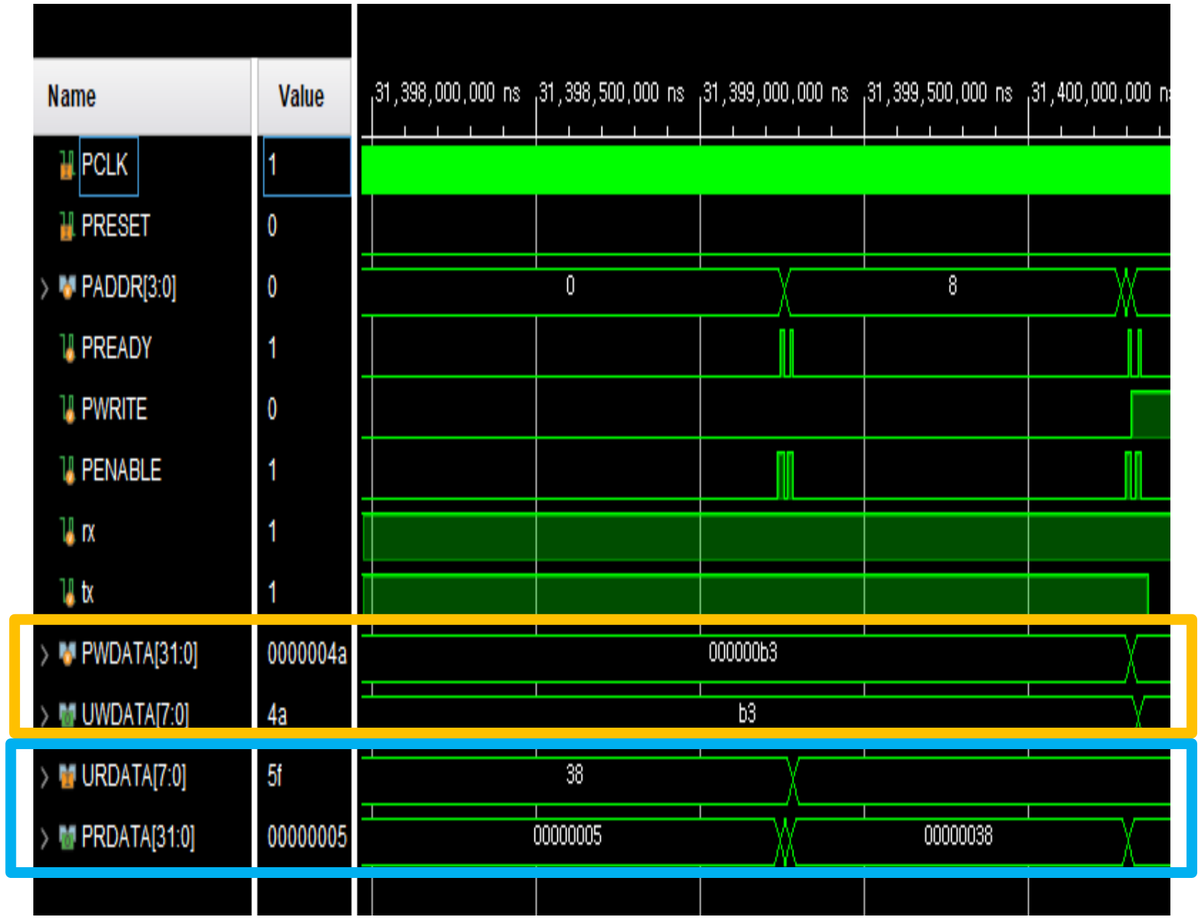
```
task automatic check_data(
    input logic [7:0] expected,
    input logic [7:0] actual,
    input string      test_name
);
    $display("----------------------------------------");
    this.test_total_count++;
    if (expected == actual) begin
        $display("Time=%0t, [PASS] %s", $time, test_name);
        $display("                  (Expect: 0x%h, Actual: 0x%h)", expected, actual);
        this.test_pass_count++;
    end else begin
        $display("Time=%0t, [FAIL] %s", $time, test_name);
        $display("                  (Expect: 0x%h, Actual: 0x%h)", expected, actual);
        this.test_fail_count++;
    end
endtask
```

Class-based Verification Result

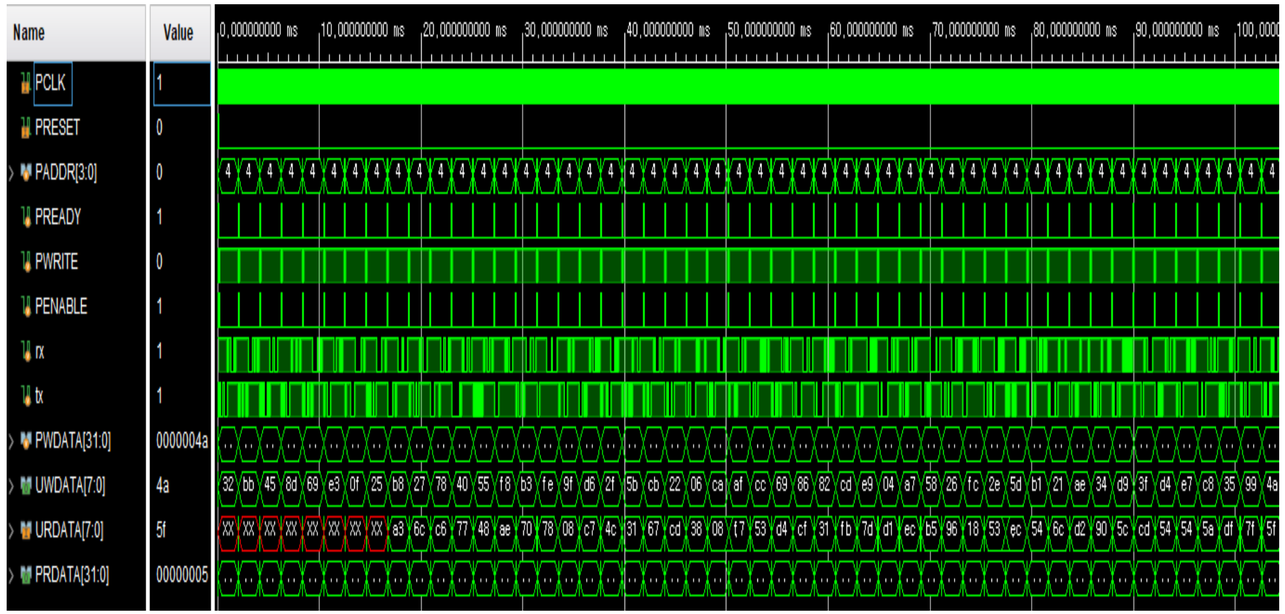
```
*******************************************
--- [Test TX] Send random data 0xb3 ---
Time=29306935000, [CPU_WRITE] Try to send 0xb3...
Time=29306965000, [CPU_WRITE] Send 0xb3 to Addr 0x4.
Time=29306995000, [PIN_RECV] Monitor 'tx' for data
Time=30296535000, [PIN_RECV] Receive 0xb3 from 'tx'
-----------------------------------------
Time=30296535000, [PASS] Random (CPU Write -> Pin Read)
                  (Expect: 0xb3, Actual: 0xb3)
--- [Test RX] Send random data 0x38 ---
Time=30297535000, [PIN_SEND] Send 0x38 to 'rx'
Time=31339135000, [PIN_SEND] Finish send data on 'rx'
Time=31339135000, [CPU_READ] Try to read data
Time=31339165000, [CPU_WAIT] RX buffer : empty
Time=31359195000, [CPU_WAIT] RX buffer : empty
Time=31379225000, [CPU_WAIT] RX buffer : empty
Time=31399255000, [CPU_READ] Read from Addr 0x8.
-----------------------------------------
Time=31399285000, [PASS] Random (Pin Send -> CPU Read)
                  (Expect: 0x38, Actual: 0x38)
*******************************************
```

Tx Test

Rx Test



Class-based Verification Result



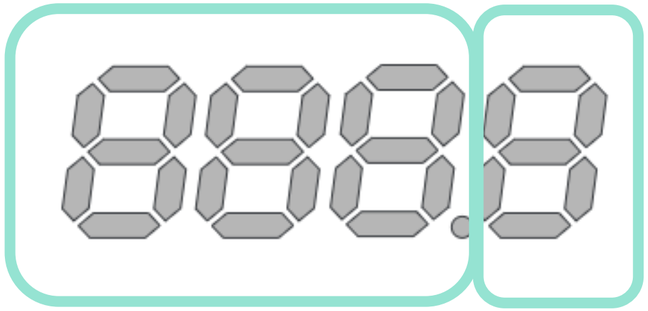

```
==========================================
          Test Result Summary
==========================================
              Total   100
              Pass    100
              Fail    0
==========================================
          >>> TEST PASS !!!! <<<
==========================================
```

C Code: Implementation & Demonstration

C Code: Implementation

C application Goal

- UART 통신 및 GPIO 제어를 활용하여 LED 동작을 제어
- FND 및 UART를 통한 상태 모니터링



LED 제어

ON/OFF

제어 모드

- C : UART Command Mode
- S : Switch Mode

- 5초마다 현재 LED 이동 상태(좌/우/정지) 안내 메시지 UART로 PC에 전송
- 제어 모드/활성화/명령에 따른 즉각적인 메시지 UART로 PC에 전송

C Code: Implementation

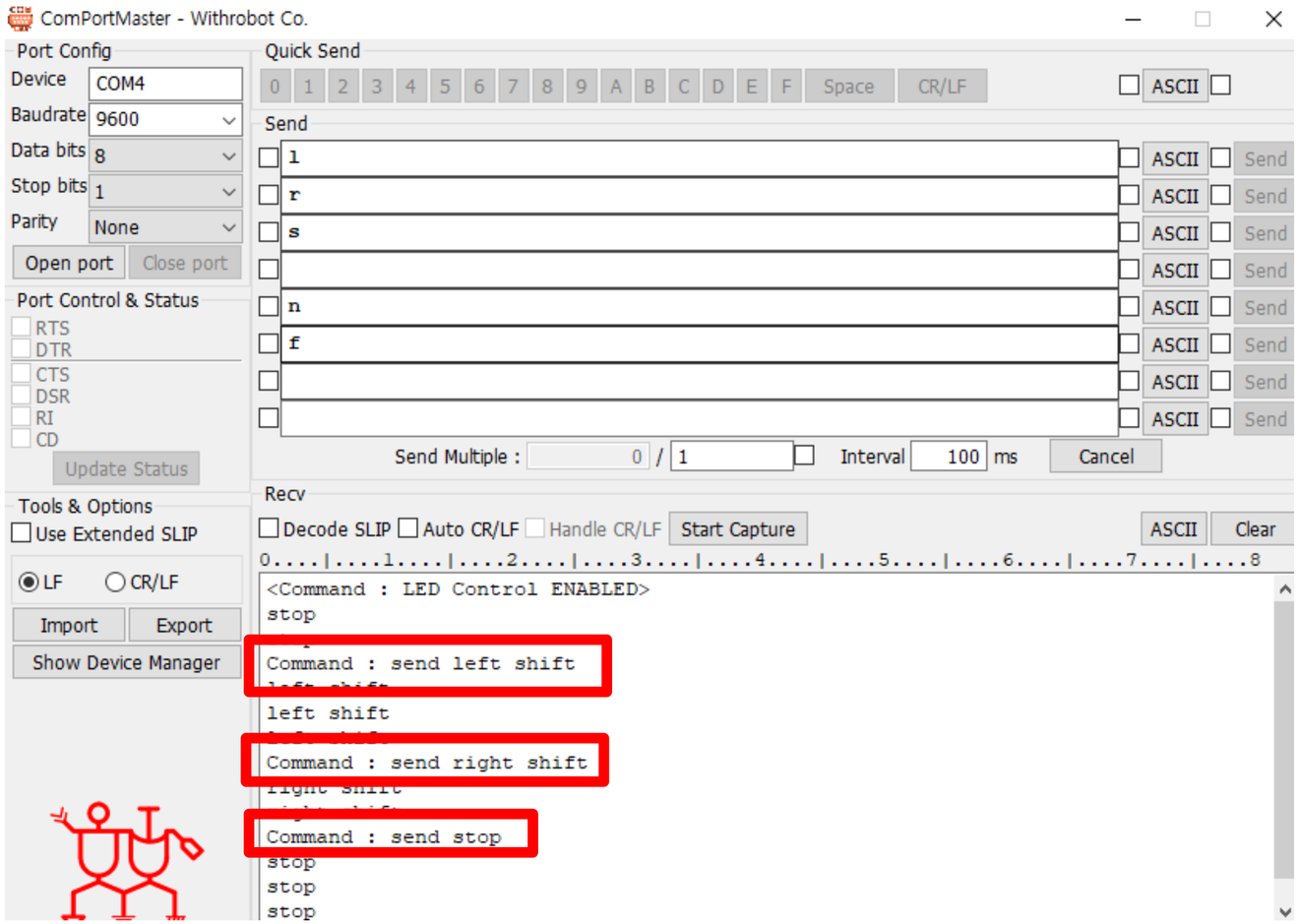
```
if (led_control_enabled) {
    if (ledState != STOP) {
        LED_write(ledData);
        delay(200);
        time_accumulator += 200;
        if (ledState == LEFT) {
            LED_leftShift(&ledData);
        } else {
            LED_rightShift(&ledData);
        }
    } else {
        LED_write(0x00);
        delay(10);
        time_accumulator += 10;
    }
```

```
void LED_write(uint32_t data) {
    GPO_ODR = data;
}

void LED_leftShift(uint32_t* pData) {
    *pData = ((*pData << 1) & 0xFF) | ((*pData >> 7) & 0x01);
}

void LED_rightShift(uint32_t* pData) {
    *pData = ((*pData >> 1) & 0xFF) | ((*pData << 7) & 0x80);
}
```

- Led 제어 명령 (UART, Switch)에 따른 led 점등



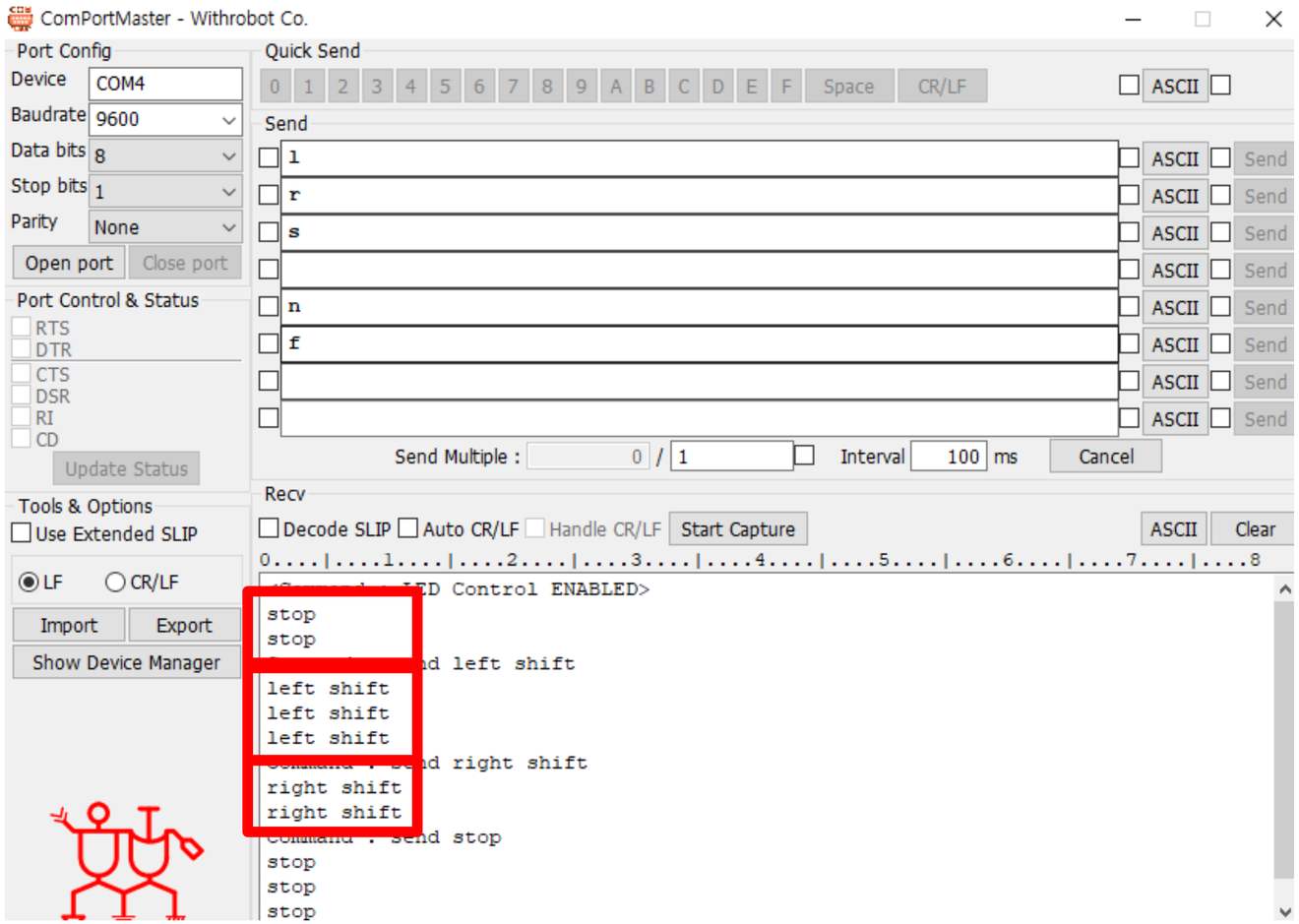
C Code: Implementation

```c
uint32_t time_accumulator = 0;
const uint32_t REPORT_INTERVAL = 5000;
```

```c
if (led_control_enabled) {
    if (ledState != STOP) {
        LED_write(ledData);
        delay(200);
        time_accumulator += 200;
        if (ledState == LEFT) {
            LED_leftShift(&ledData);
        } else {
            LED_rightShift(&ledData);
        }
    } else {
        LED_write(0x00);
        delay(10);
        time_accumulator += 10;
    }
```

```c
if (led_control_enabled && (time_accumulator >= REPORT_INTERVAL)) {
    if (ledState == LEFT) {
        uart_print_report_left();
    } else if (ledState == RIGHT) {
        uart_print_report_right();
    } else {
        uart_print_report_stop();
    }
    time_accumulator -= REPORT_INTERVAL;
}
```

- 일정 주기마다 현재 LED 이동 상태(좌/우/정지) 안내 메시지 UART로 PC에 전송



C Code: Implementation

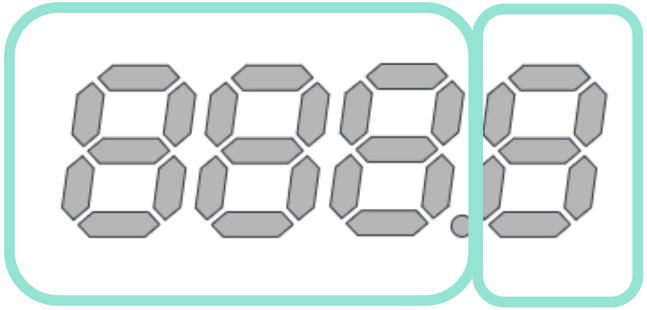
```c
while (1) {
    if (GPI_IDR & SW0_MODE) {
        control_mode = 'S';
    } else {
        control_mode = 'U';
    }
    FND_display_status(control_mode, led_control_enabled);
```

```c
void FND_display_status(char mode, int enabled) {
    uint32_t d4_bcd, d3_bcd, d2_bcd, d1_bcd;
    uint32_t dot_flags = 0;

    if (enabled) {
        d4_bcd = 0xF;
        d3_bcd = 0x0;
        d2_bcd = 0xB;
    } else {
        d4_bcd = 0x0;
        d3_bcd = 0xD;
        d2_bcd = 0xD;
    }
    if (mode == 'S') {
        d1_bcd = 0x5;
    } else {
        d1_bcd = 0xC;
    }

    dot_flags = (1 << 1);
    FND_DATA = (dot_flags << 16) | (d4_bcd << 12) | (d3_bcd << 8) | (d2_bcd << 4) | d1_bcd;
}
```

- Led 제어 기능 On/Off + 제어모드(UART/Switch)
  FND(7-Segment Display)에 구현



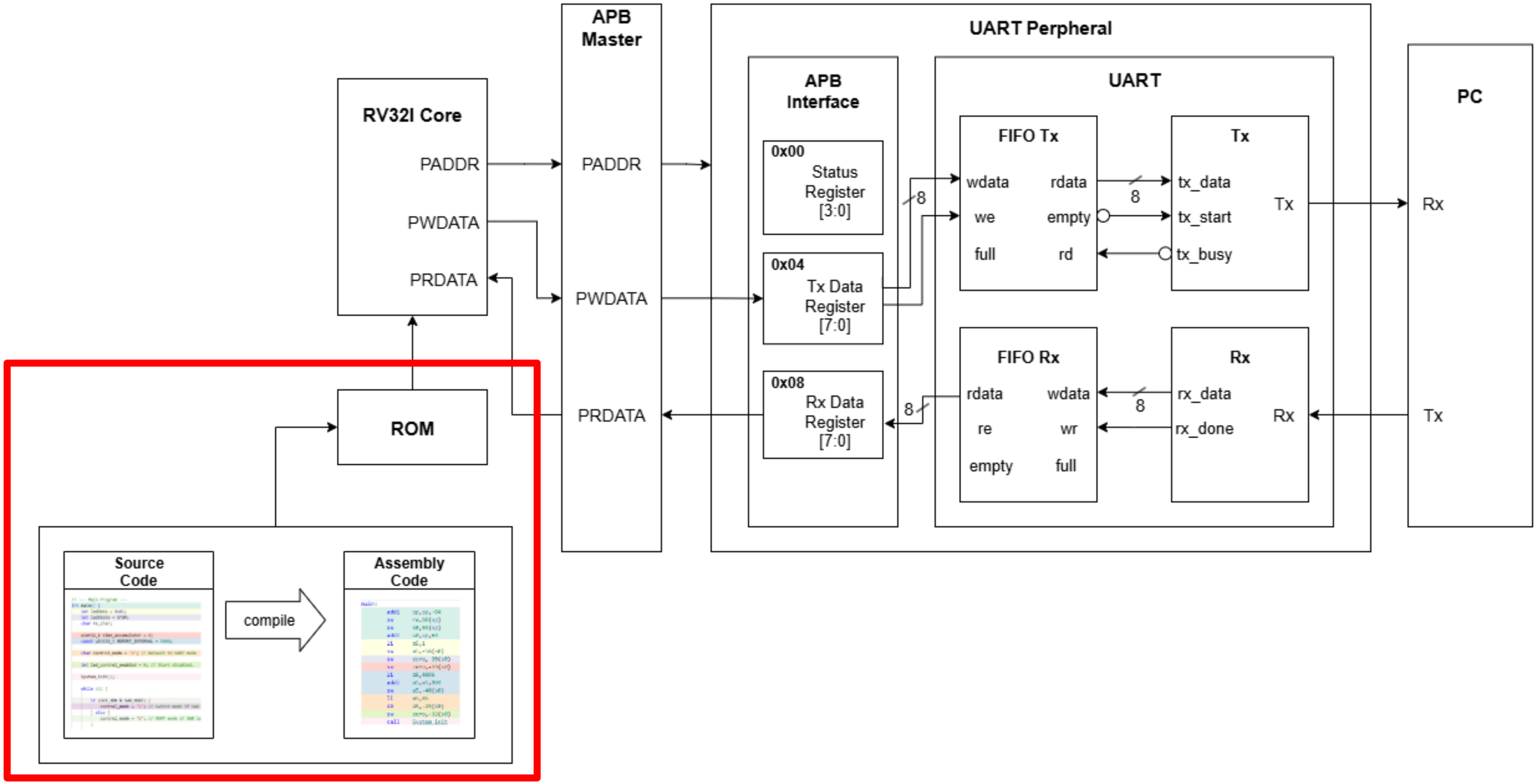
LED 제어

ON/OFF

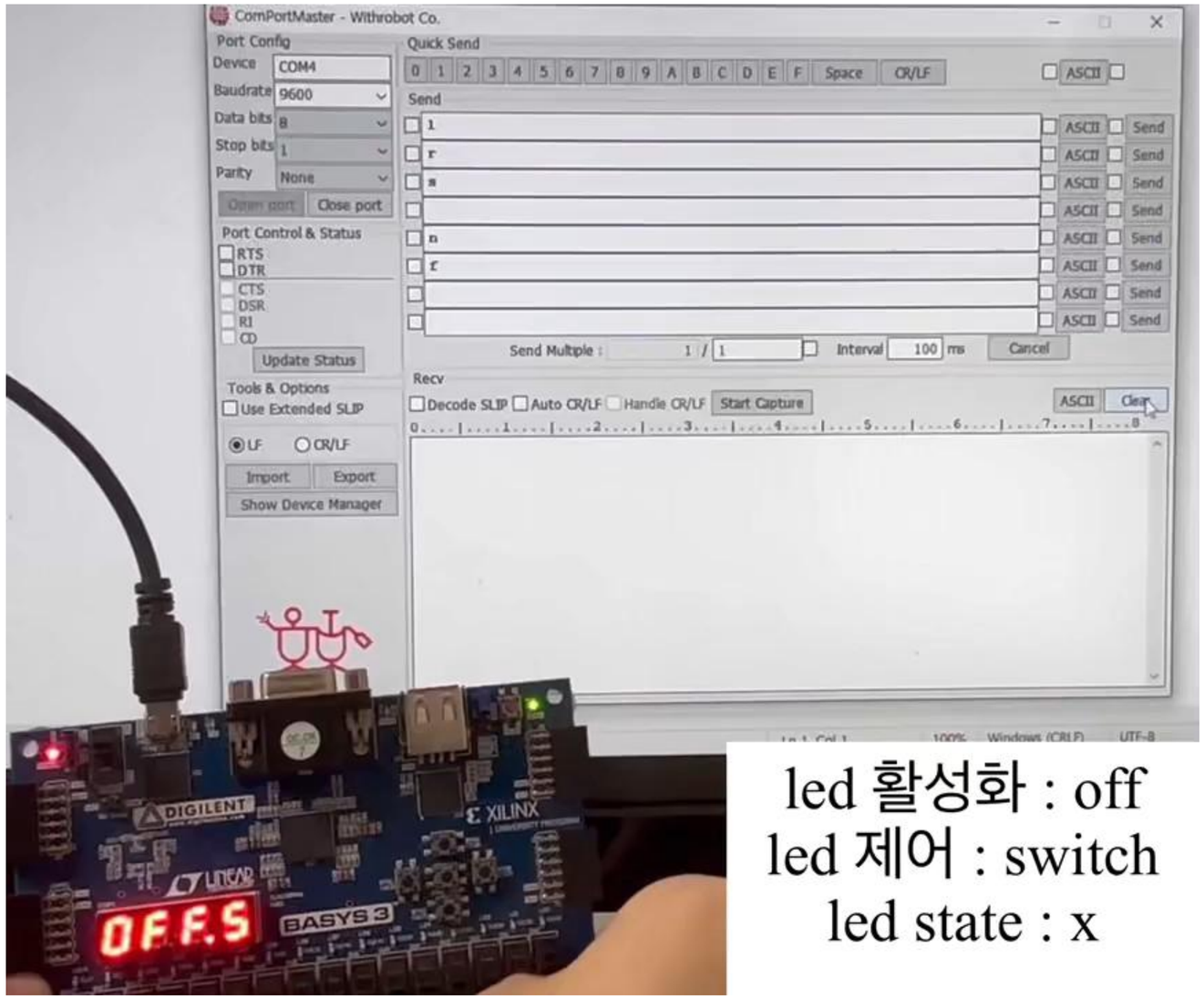
제어 모드

- C : UART Command Mode
- S : Switch Mode

C Code: Implementation



C Code: Demonstration



led 활성화 : off
led 제어 : switch
led state : x

고찰

고찰

- **프로토콜(Protocol)의 중요성**

- 이번 프로젝트는 설계와 검증 모두가 APB와 UART라는 명확한 '프로토콜'을 기반으로 만들어져야 함을 깨닫는 과정이었습니다. 설계 초기, PREADY 같은 APB 신호나 타이밍을 정확히 지키지 않았을 때, 시스템은 단순히 느려지는 것이 아니라 아예 작동하지 않는 현상이 발생하였습니다. 하드웨어 설계는 '기능'을 구현하는 동시에, 모듈 간의 '타이밍'이라는 약속이 얼마나 중요한 지 확인할 수 있었습니다.

- **C언어와 하드웨어의 간극: 문자열 전송**

- C언어(소프트웨어)와 하드웨어(레지스터)의 동작 방식 차이를 다시 한 번 확인하게 되었습니다. C언어에서는 “Stop"라는 String을 한 번에 전송하려 했으나, 하드웨어는 이를 수신하지 못했습니다. 원인은 CPU가 '문자열의 주소(Pointer)'를 보낸 반면, UART_Peripheral 모듈은 '8비트 Character' 하나만 받을 수 있었기 때문입니다. 이를 해결하기 위하여 uart_send_string 함수를 만들어 한 글자씩 순차적으로 전송하고 대기하는 로직을 구현하며, 소프트웨어의 추상화된 명령이 하드웨어에서 어떻게 low-level 동작으로 작동하는지 명확히 이해했습니다.

- **향후 계획: APB 기반의 센서 제어 시스템 확장**

- 현재 완성된 RISC-V CPU와 APB 버스 시스템을 기반으로, 더 다양한 센서(온습도 센서 등)를 제어
- 센서 통신에 필수적인 I2C 또는 SPI 프로토콜 마스터 모듈을 새로운 Peripheral로 직접 설계하여 APB 버스에 연결

Thank you