Multi-Cycle RISC-V 기반

APB Peripheral 설계

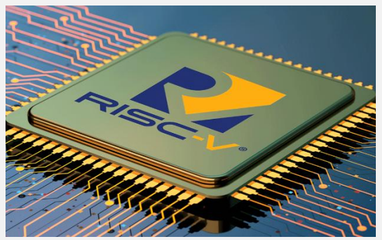
박찬호

목차

개요

•SoC의 핵심 구성 요소인 **RISC-V 코어**, 표준 버스인 **APB**, 그리고 PeriPheral를 직접 설계하고 연동한 후 c 코드를 이용한 하드웨어 제어를 목표로 합니다.

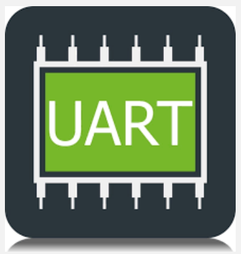
Core



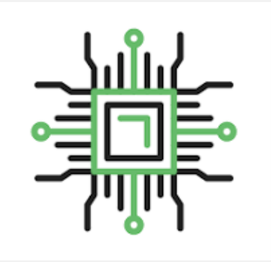
BUS

ARM AMBA
Interconnect Standards

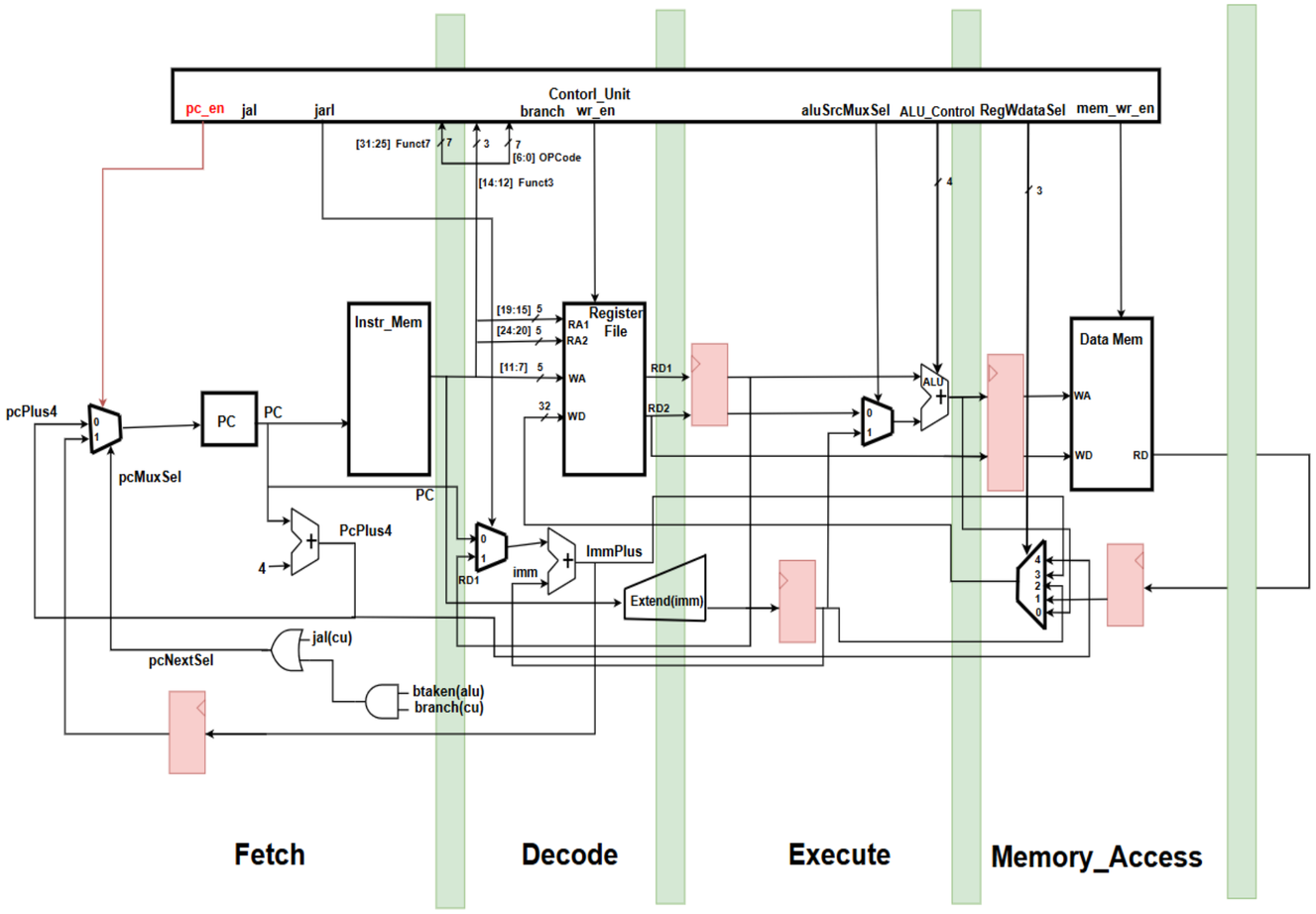
Periperal



C Application



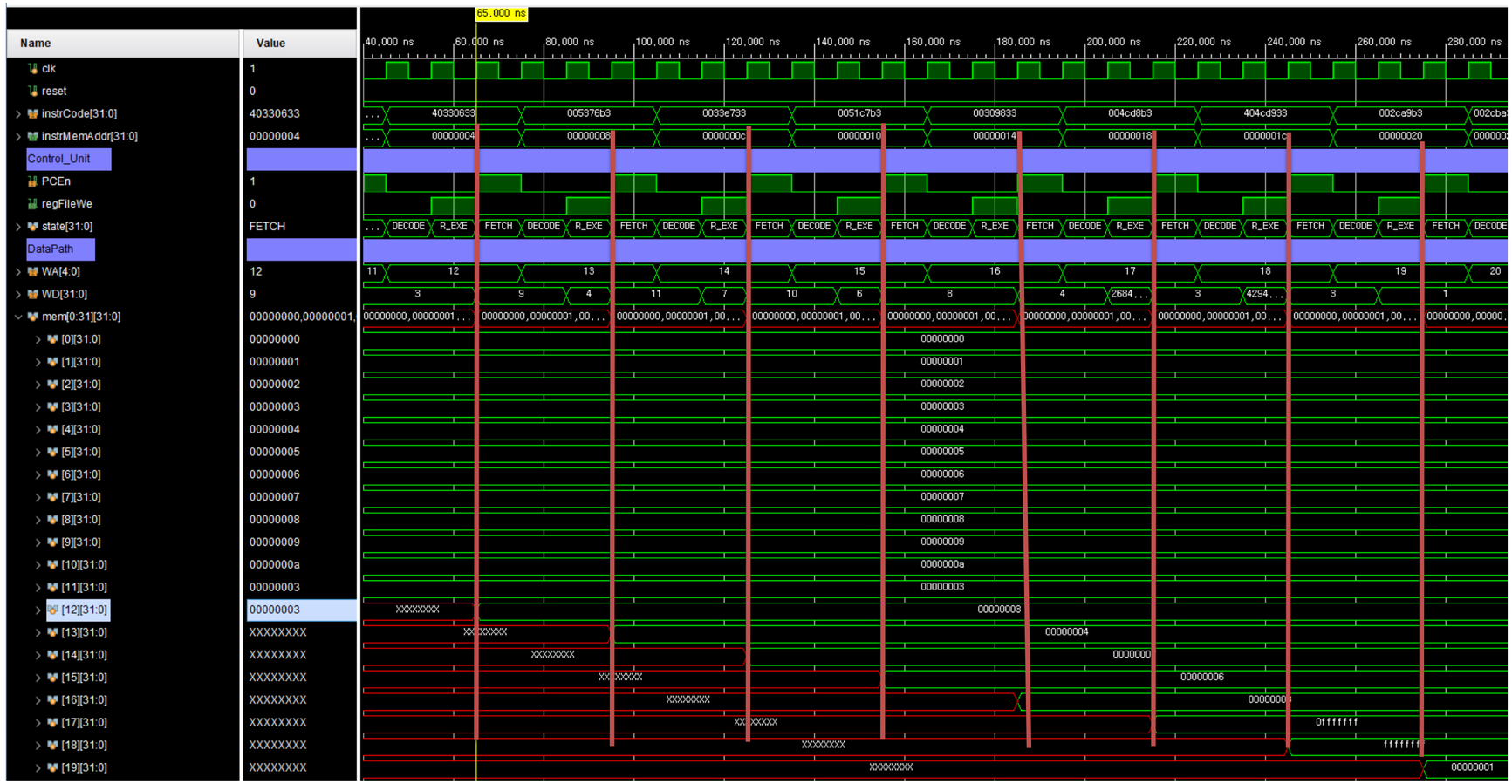
Multi-Cycle RISC-V



- Single-Cycle RISC-V에서는 모든 명령어가 1 사이클에 처리되기 때문에 모든 명령어가 가장 느린 명령어의 실행 속도에 맞춰야 한다.

- 이러한 비효율을 해결하기 위해 더 짧은 스테이지로 분리하여 처리하는 방식

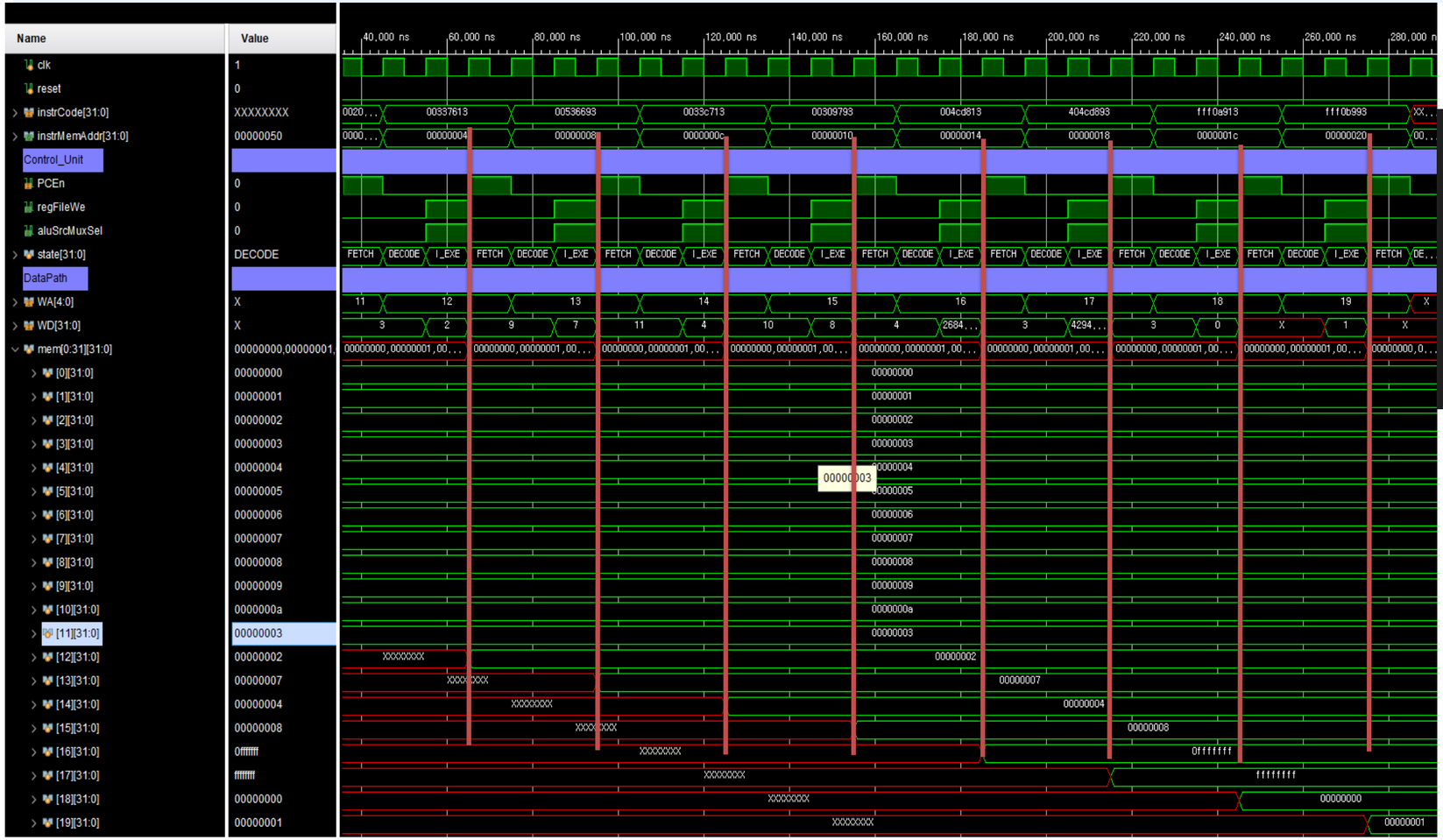
R-type Simulation




```
// R-type
rom[0] = 32'h002085B3;  // add x11, x1, x2 = 3
rom[1] = 32'h40330633;  // sub x12, x6, x3 = 3
rom[2] = 32'h005376B3;  // and x13, x6, x5 = 4
rom[3] = 32'h0033E733;  // or x14, x7, x3  = 7
rom[4] = 32'h0051C7B3;  // xor x15, x3, x5 = 6
rom[5] = 32'h00309833;  // sll x16, x1, x3 = 8
rom[6] = 32'h004CD8B3;  // srl x17, x25, x4 = 0fff_ffff
rom[7] = 32'h404CD933;  // sra x18, x25, x4 = ffff_ffff
rom[8] = 32'h002CA9B3;  // slt x19, x25, x2 = 1
rom[9] = 32'h002CBA33;  // sltu x20, x25, x2 = 0
```

- Fetch : 명령어 인출
- Decode : 명령어에 맞는 제어 신호 생성
- Execute : 명령어에 맞는 연산 수행
- PCEn : Fetch 단계에서만 1로 세팅

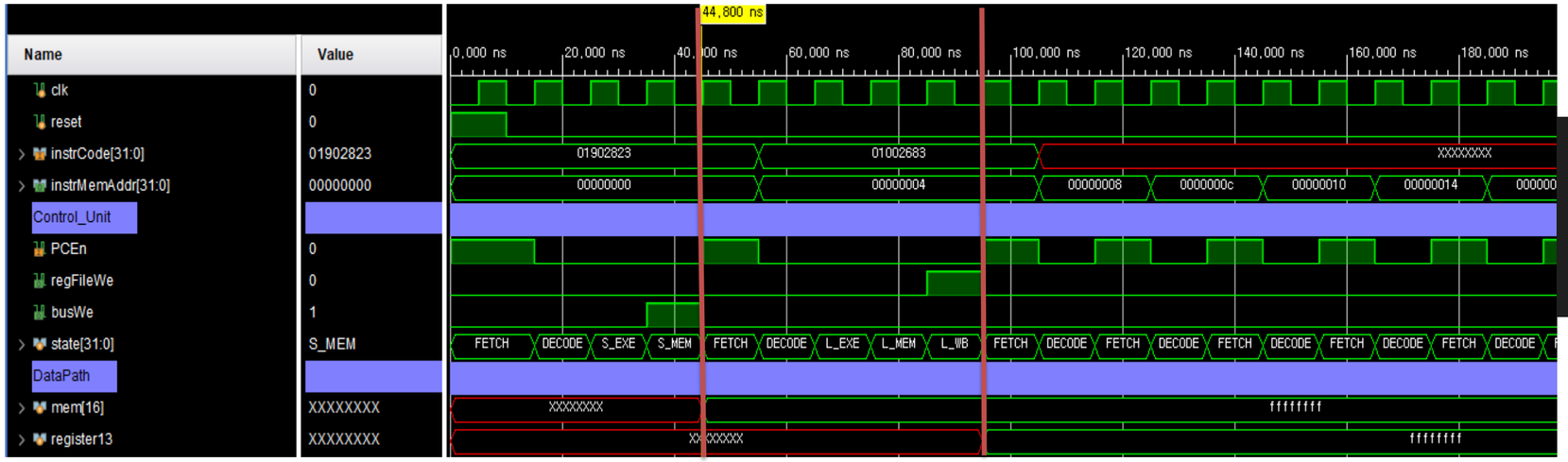
I-type Simulation




```
// I-type
rom[0] = 32'h00208593;  // addi  x11, x1, 2 = 3
rom[1] = 32'h00337613;  // andi  x12, x6, 3 = 3
rom[2] = 32'h00536693;  // ori   x13, x6, 5 = 4
rom[3] = 32'h0033C713;  // xori  x14, x7, 3 = 7
rom[4] = 32'h00309793;  // slli  x15, x1, 3 = 8
rom[5] = 32'h004CD813;  // srli  x16, x25, 4 = 0fff_ffff
rom[6] = 32'h404CD893;  // srai  x17, x25, 4 = ffff_ffff
rom[7] = 32'hFFF0A913;  // slti  x18, x1, -1 = 0
rom[8] = 32'hFFF0B993;  // sltiu x19, x1, -1 = 1
```

- aluSrcMuxSel이 1로 세팅되어 imm 값과 rs1이 연산

S, L-type Simulation

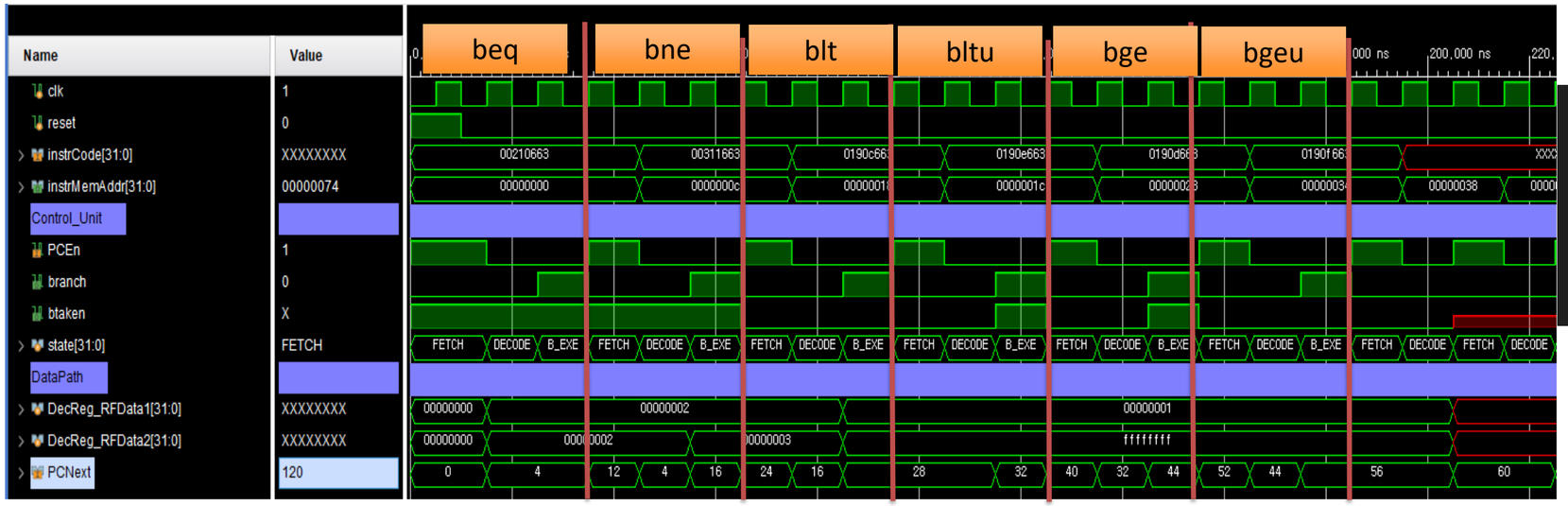



```
// S, L-type
rom[0] = 32'h01902823;  //sw x25, 16(x0)
rom[1] = 32'h01002683;  //lw x13, 16(x0)
```

- S-type
  - Mem_access 단계에서만 write 신호를 1로 세팅
- L-type
  - RV32I 명령어 중 실행 시간이 가장 긴 명령어
  - WriteBack 단계에서 레지스터에 값을 쓴다.

- core는 PeriPheral 을 메모리로 생각하기 때문에 PeriPheral을 제어할 때 S, L 타입 사용된다.

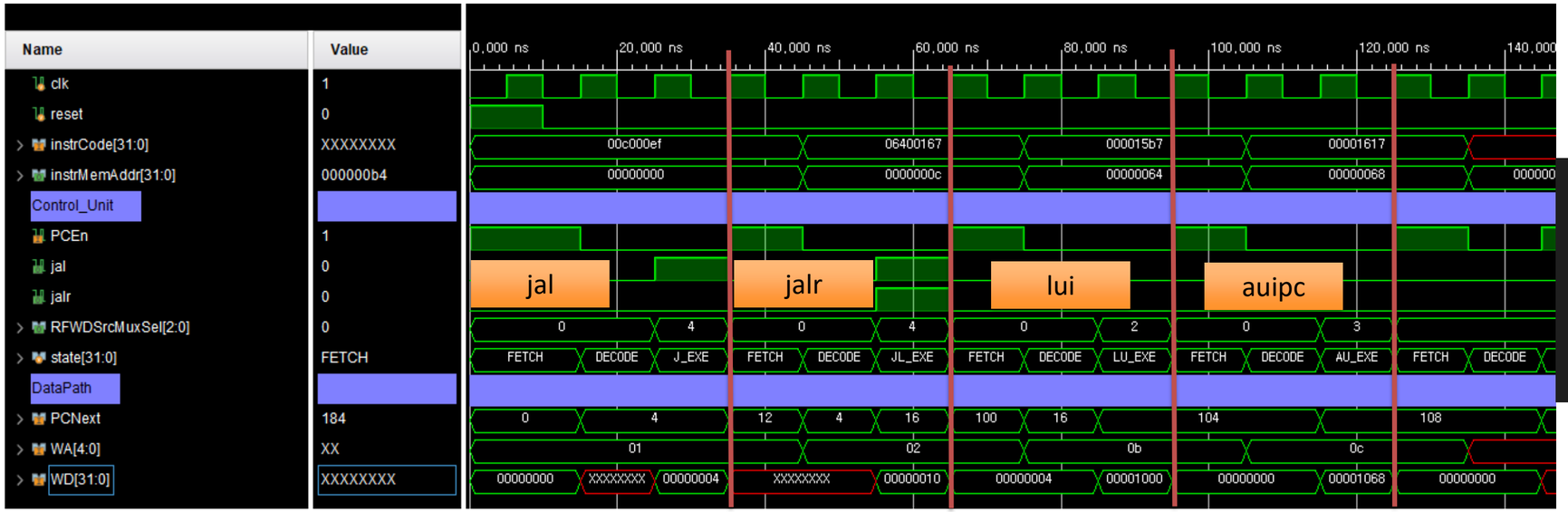
B-type Simulation




```
// B-type
rom[0]  = 32'h00210663;  // beq x2, x2, 12
rom[3]  = 32'h00311663;  // bne x2, x3, 12
rom[6]  = 32'h0190C663;  //blt x1(1) x25(-1) 12
rom[7]  = 32'h0190E663;  //bltu x1(1) x25(-1) 12
rom[10] = 32'h0190D663;  //bge x1 x25 12
rom[13] = 32'h0190F663;  //bgeu x1 x25 12
```

- B-type
  - Execute 단계에서 분기 조건 판단
  - C언어에서 if, for, while 조건문에서 사용

J, U-type Simulation

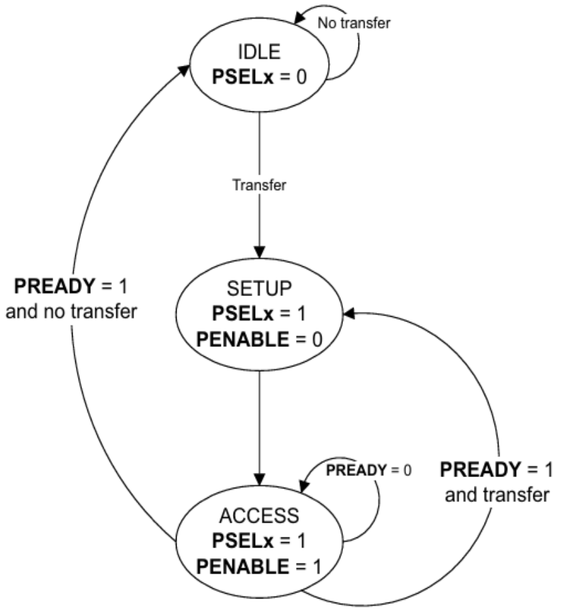



```
// J-type
rom[0] = 32'h00C000EF;  // jal x1, 12
rom[3] = 32'h06400167;  // jalr x2, x0, 100
// U-type
rom[25] = 32'h000015B7; // lui   x11 1  x11 = 0x1000
rom[26] = 32'h00001617; // auipc x12 1  x12 = 0x1068
```

- J-type
  - jal : 함수 호출에 사용
  - jalr : 함수 종료후 리턴할 때 사용
- U-type
  - 큰 값을 load할 때 사용

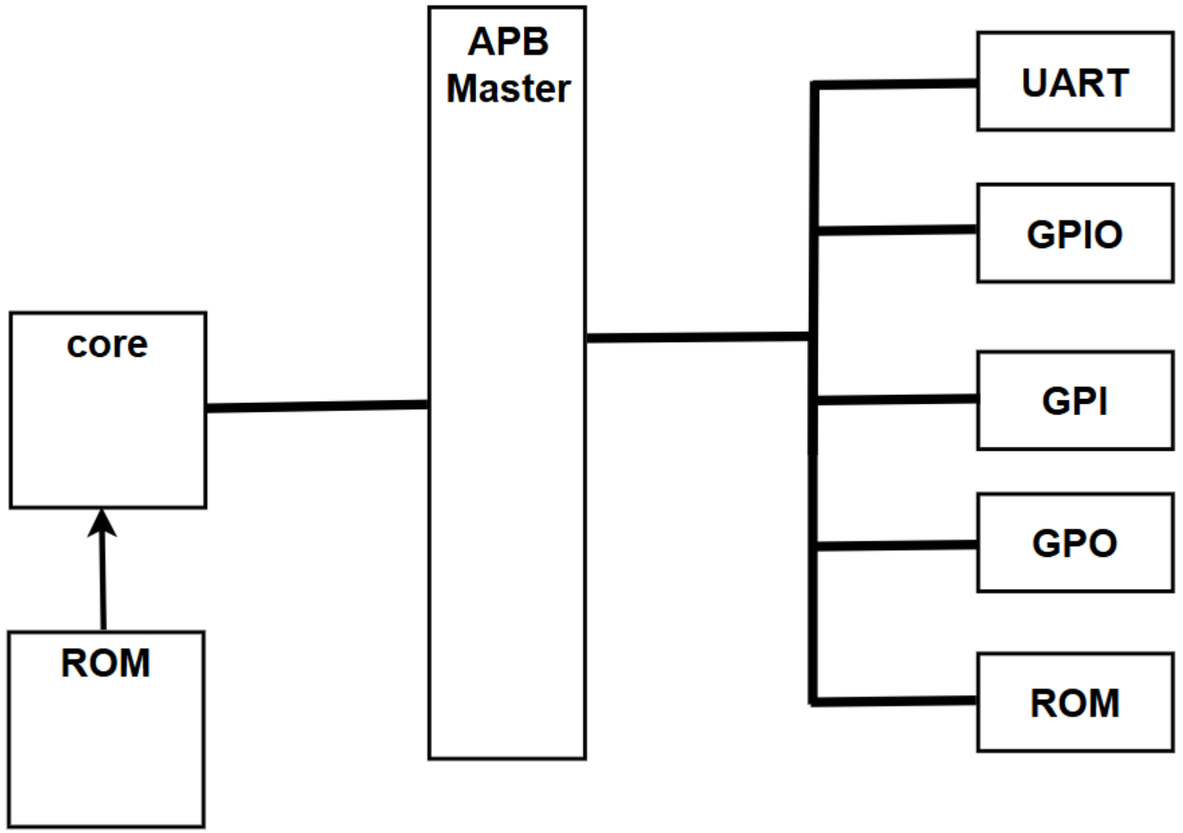
APB Protocol

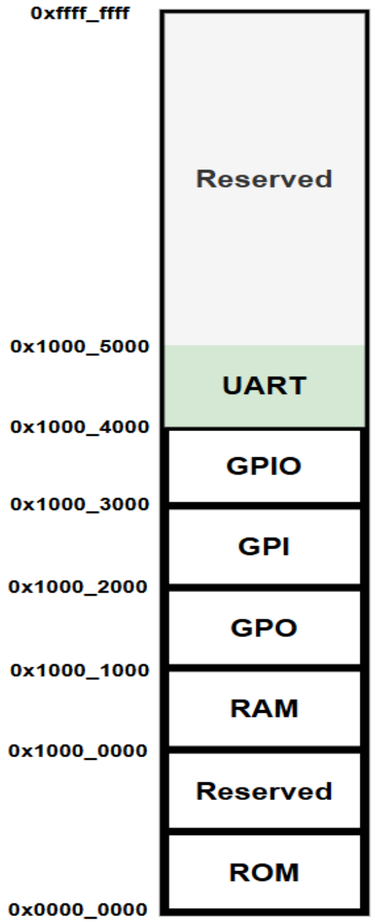
- AMBA Bus 란?
  - SoC 내부의 core, memory, peripheral 사이의 통신을 위한 표준 인터페이스이다.
  - AMBA 규격에 맞는 IP를 설계했다면 RISC-V 프로세서뿐만 아니라 ARM 프로세서를 사용하는 다른 SoC 시스템에도 수정없이 바로 연결하여 사용할 수 있기 때문에 재사용성이 높다
- APB는 가장 단순하며 전력 소모가 적으며, UART, GPIO, TIMER 저속으로 동작하는 peripheral 연결에 최적화 되어 있다.



- IDLE : core가 PWDATA, PADDR, PWRITE transfer 신호를 보내면 SETUP 상태로 천이
- Setup : PADDR을 디코딩하여 PSELx를 1로 세팅 후 1cycle 뒤에 ACCESS 상태로 천이
- ACCESS : slave로 부터 PREADY 신호가 올 때까지 대기

System Architecutre & Memory Map



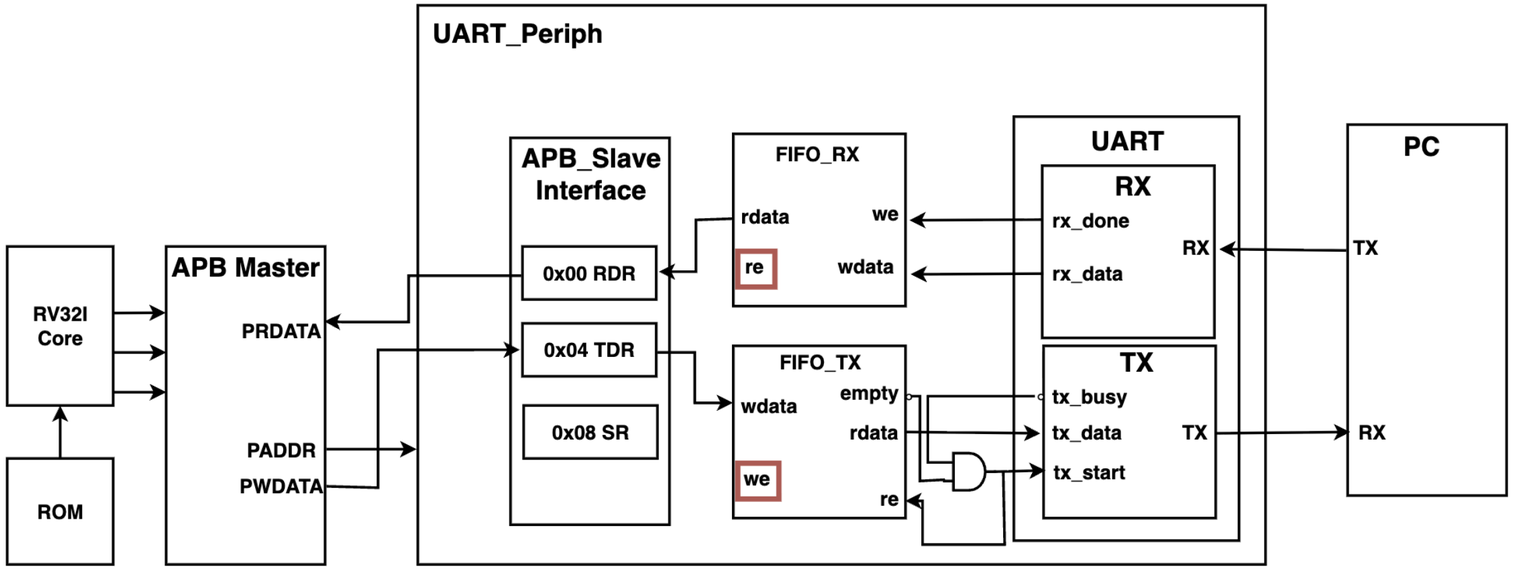


UART 설계 및 검증
Register Map

offset	register	31...8	7	6	5	4	3	2	1	0
0x00	RDR	Reserved	RDR[7:0]							
0x04	TDR	Reserved	TDR[7:0]							
0x08	SR	Reserved							RXNE	TXNF

- RDR(Rx Data Register)
  - 수신받은 데이터를 저장하고 있는 Register
- TDR(Tx Data Register)
  - 송신할 데이터를 저장하고 있는 Register
- RXNE(Rx Not Empty)
- TXNF (Tx Not Full)

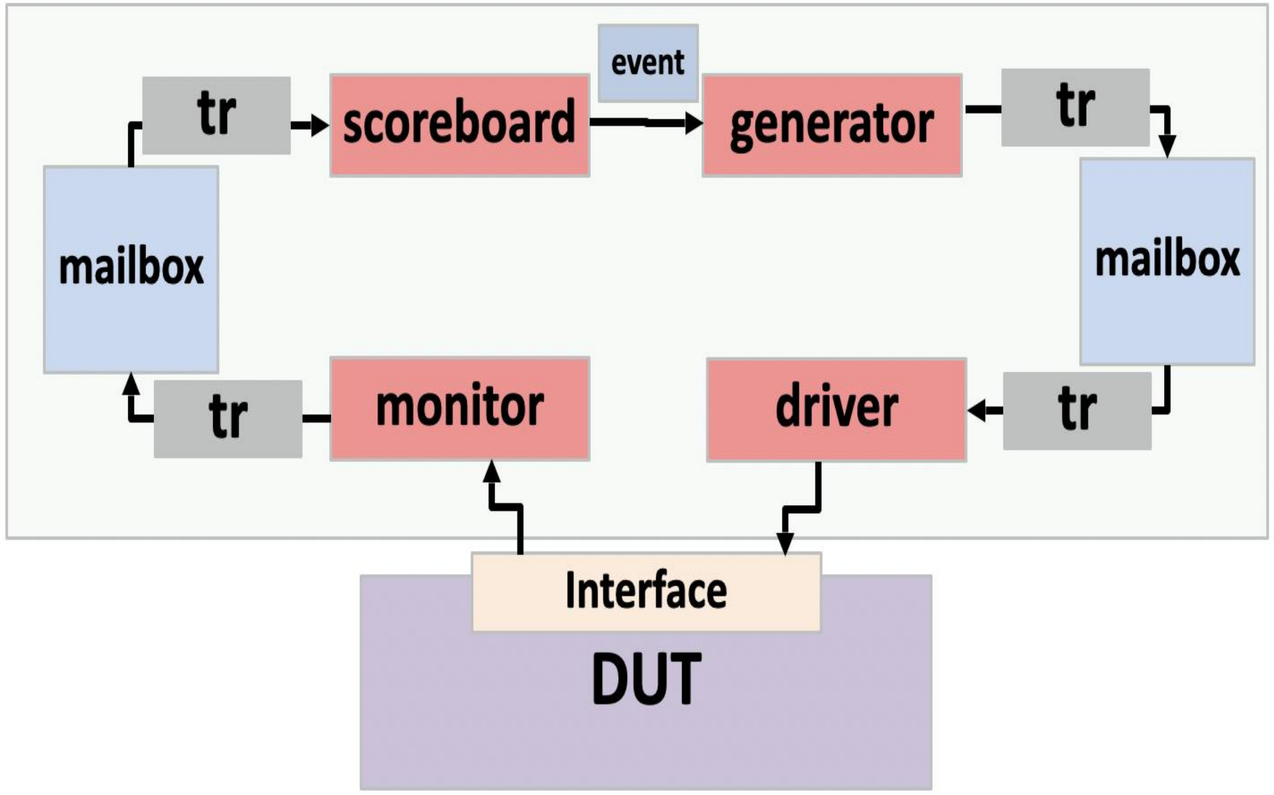
UART 설계 및 검증
Block Diagram




```
// tx_fifo_write
assign tx_fifo_wr       = (PSEL && PENABLE && PWRITE && (PADDR[3:2] == 2'd1));
assign tx_fifo_wdata    = PWDATA;
```

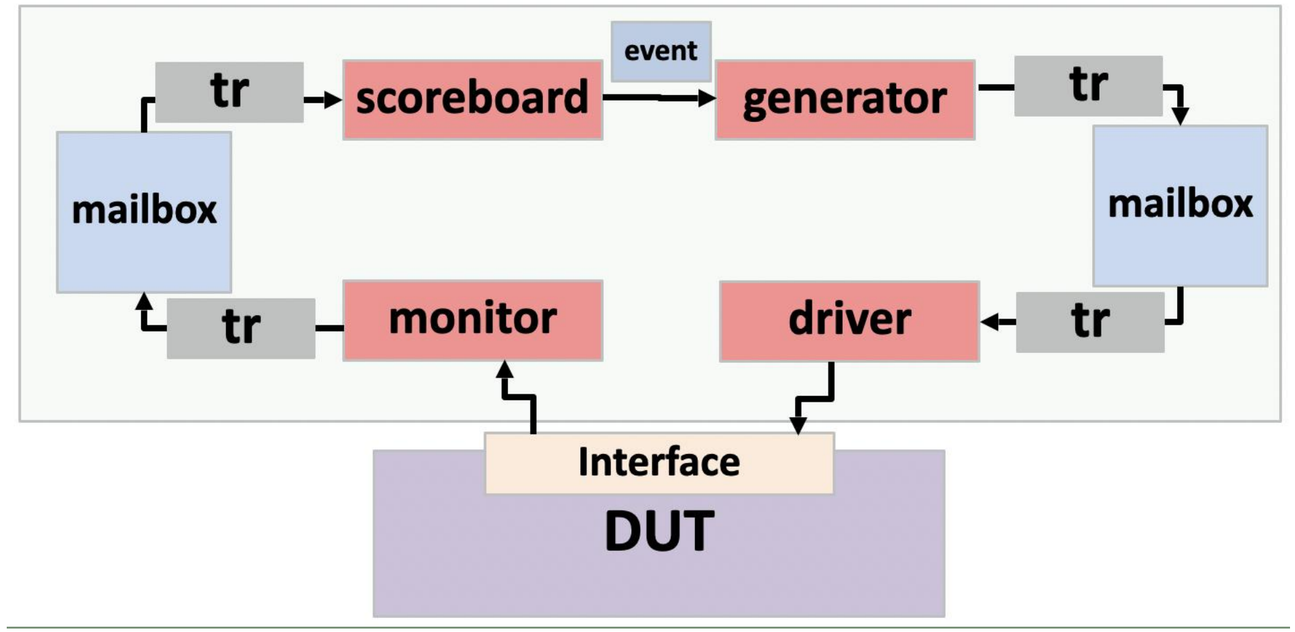
● tx_start 조건 : ~tx_busy && ~ tx_fifo_empty

UART 설계 및 검증
검증 시나리오



- Rx 검증 : random_data를 uart 통신 규약에 맞춰 rx 핀에 전송 random_data와 PRDATA가 같다면 pass
- Tx 검증 : tx 핀을 관찰하여 데이터 비트를 샘플링한 received_data가 tx_send_data와 같다면 pass

UART 설계 및 검증
Rx 검증



● generator : send_data 랜덤 생성

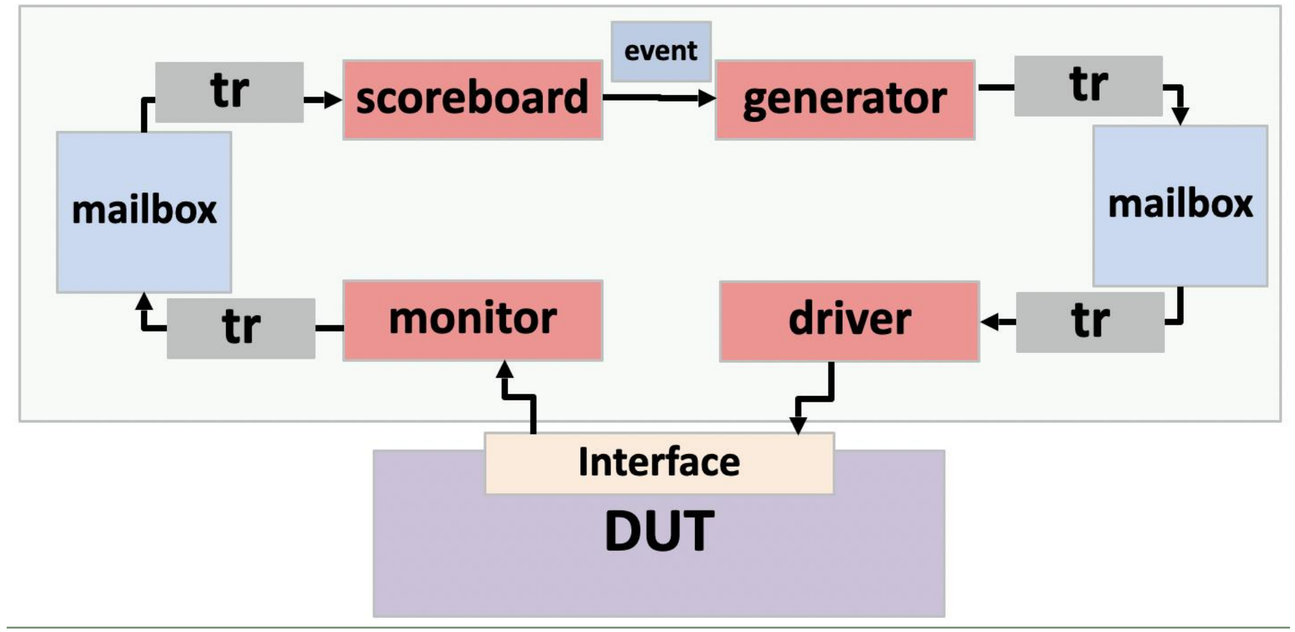
● driver

```
task automatic send_serial_to_rx(logic [7:0] rx_send_data);
    u_if.rx_expected_data <= rx_send_data;
    u_if.rx <= 0;
    #(BIT_PERIOD);
    for (int i = 0; i < 8; i++) begin
        u_if.rx <= rx_send_data[i];
        #(BIT_PERIOD);
    end
    u_if.rx = 1;
    #(BIT_PERIOD);
endtask  //automatic
```

● 랜덤 생성한 데이터를 uart 통신 규격에 맞춰 rx 핀으로 송신

● scoreboard : PRDATA와 random_data 비교

UART 설계 및 검증
tx 검증



- generator : PWDATA 랜덤 생성
- Monitor

```
task sample_uart_tx(output logic [7:0] received_data);
    @(negedge u_if.tx);
    #(BIT_PERIOD + BIT_PERIOD / 2);
    for (int i = 0; i<8; i++) begin
    received_data[i] = u_if.tx;
        #(BIT_PERIOD);
    end
    #(BIT_PERIOD);
endtask //
```

- tx 핀을 샘플링 해서 received_data로 복원
- scoreboard : received_data와 PWDATA_data 비교

UART 설계 및 검증
Report

```
[GEN] generate tx_send_data = eb
[MON], sample_data = 000000eb
[SCB] PASS! : Expected_data = eb, Received_data = 000000eb
[GEN] generate tx_send_data = e6
[MON], sample_data = 000000e6
[SCB] PASS! : Expected_data = e6, Received_data = 000000e6
[GEN] generate tx_send_data = 65
[MON], sample_data = 00000065
[SCB] PASS! : Expected_data = 65, Received_data = 00000065

========================================
            Test Report
========================================
|   Total Test   |      100            |
----------------------------------------
|   Pass Count   |      100            |
----------------------------------------
|   Fail Count   |        0            |
----------------------------------------
| Success Rate   |   100.00 %          |
========================================
          Test Finished
========================================
```

● tx_report

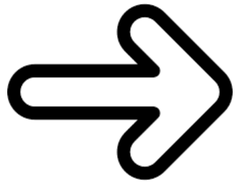
```
[GEN] generate rx_send_data = a9
[DRV], rx_send_data = a9
[SCB] PASS! : Expected_data = a9, Received_data = a9
[GEN] generate rx_send_data = 27
[DRV], rx_send_data = 27
[SCB] PASS! : Expected_data = 27, Received_data = 27
[GEN] generate rx_send_data = 81
[DRV], rx_send_data = 81
[SCB] PASS! : Expected_data = 81, Received_data = 81

========================================
            Test Report
========================================
|   Total Test   |      100            |
----------------------------------------
|   Pass Count   |      100            |
----------------------------------------
|   Fail Count   |        0            |
----------------------------------------
| Success Rate   |   100.00 %          |
========================================
          Test Finished
========================================
```

● rx_report

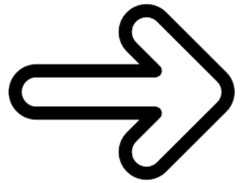
C Application

● PC로부터 받은 데이터에 따라 LED 점등 상태를 결정하고 FPGA의 상태를 주기적으로 보내는 Application 구현

1-character 전송



입력받은 데이터에 따른 LED 상태 제어



주기적으로 FPGA의 상태 전송

C Application

```c
void LED_wrtie(uint32_t data){
    GPO_ODR = data;
}

void LED_leftShift(uint32_t *pData){
    *pData = *pData <<1 | *pData >>7;
}
void LED_rightShift(uint32_t *pData){
    *pData = *pData >>1 | *pData << 7;
}
void LED_STOP(uint32_t *pData){
    *pData = *pData;
}
```

● LED 제어 함수들

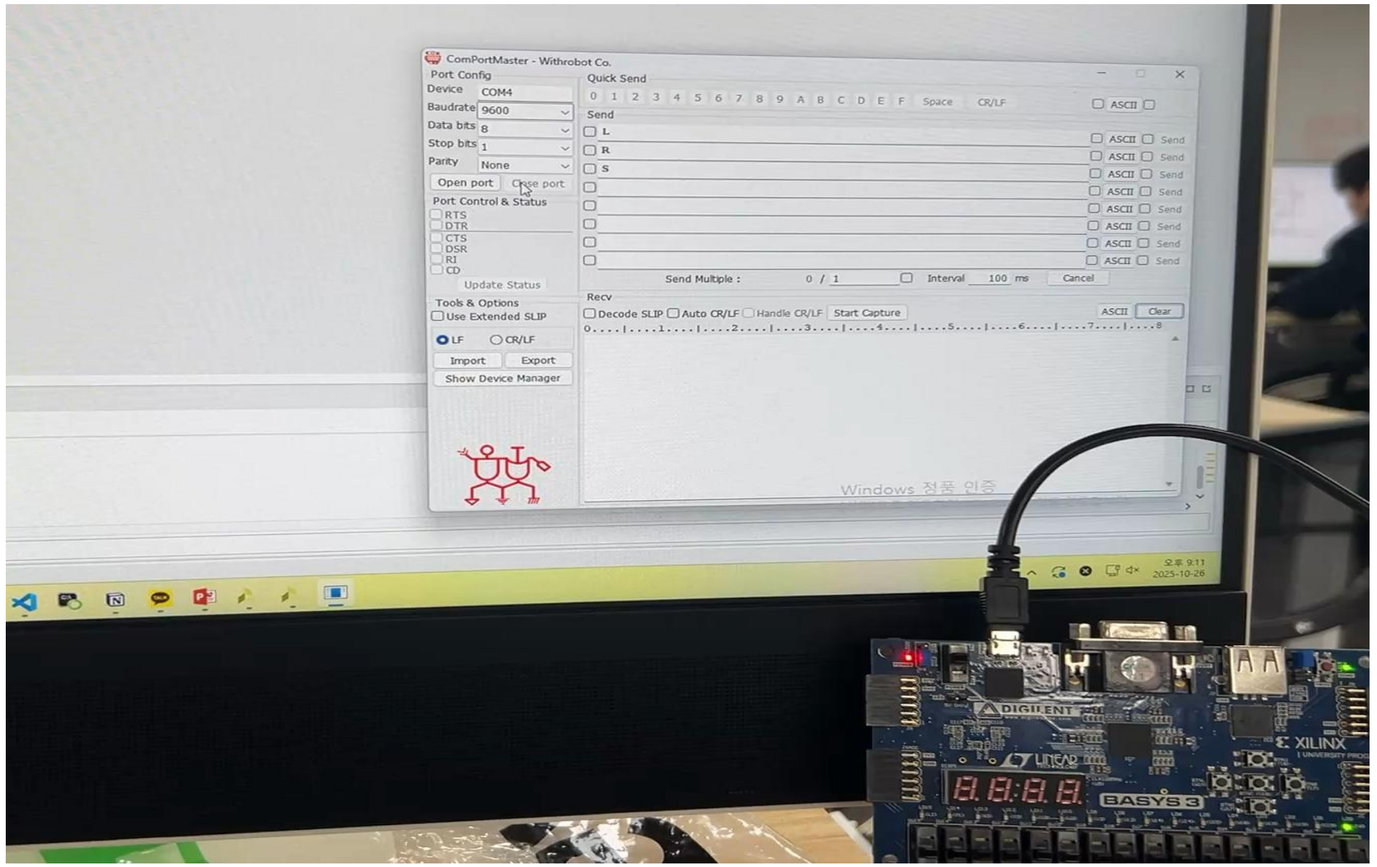
```c
void process_uart_commands(uint32_t *ledState)
{
    while(UART_FIFO_SR & RXNE_FLAG){

        uint32_t ch = UART_FIFO_RDR & 0xff;

        if(ch == 'R' || ch == 'r'){ //RIGHT
                *ledState = RIGHT;
            }
        else if(ch == 'L' || ch == 'l'){ // LEFT
            *ledState = LEFT;
        } else if(ch == 'S' || ch == 's'){
            *ledState = STOP;
        }
}
}
```

● 데이터를 송신, 수신을 처리하는 함수

```c
void UART_TX_SR(uint32_t *ledState){

    if((UART_FIFO_SR & TXNF_FLAG)){
        // TX_FIFO가 있을때만
        if(*ledState == RIGHT ){ //RIGHT
            UART_FIFO_TDR = 'r';
            UART_FIFO_TDR = 'i';
            UART_FIFO_TDR = 'g';
            UART_FIFO_TDR = 'h';
            UART_FIFO_TDR = 't';
            UART_FIFO_TDR = '\n';
            UART_FIFO_TDR = '\n';

        }
```

● CPU가 주기적으로 주변 장치의 상태를 확인하여 데이터를 처리하는 방식인 폴링 방식을 사용해서 구현

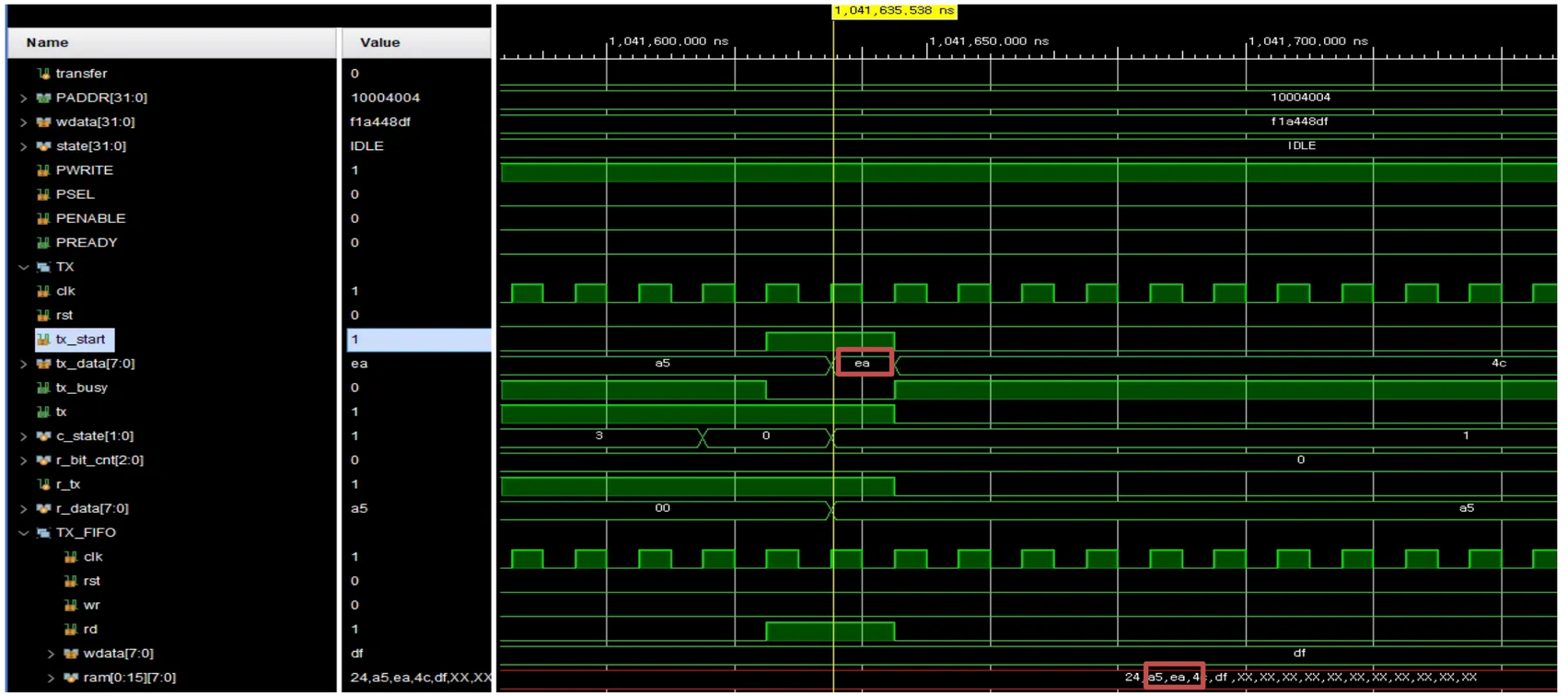
동작 영상


ComPortMaster - Withrobot Co.
Port Config
Device COM4
Baudrate 9600
Data bits 8
Stop bits 1
Parity None
Open port
Close port
Port Control & Status
RTS
DTR
CTS
DSR
RI
CD
Update Status
Tools & Options
Use Extended SLIP
LF
CR/LF
Import
Export
Show Device Manager
Quick Send
Send
Send Multiple :
Interval
Cancel
Recv
Decode SLIP
Auto CR/LF
Handle CR/LF
Start Capture
Clear
Windows 정품 인증
DIGILENT
XILINX
BASYS 3

TroubleShooting(문제 정의)

tx_fifo에 저장되어 있는 데이터 유실 발생

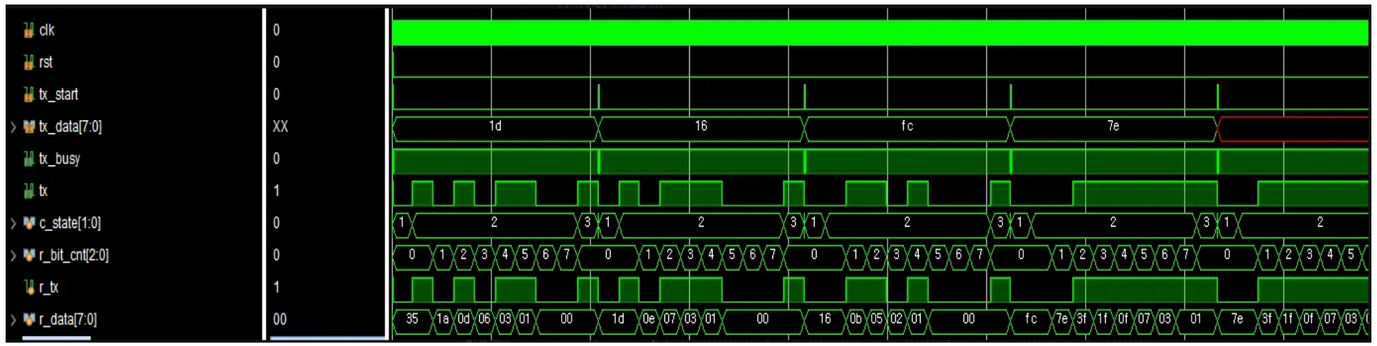
TroubleShooting(원인 분석)



tx_start가 2클럭 지속되어 데이터 하나가 유실되는 현상 발생

TroubleShooting(문제 해결)

```
// pulse tx start
logic tx_start_pulse;
logic tx_start_condition;
logic tx_start_ff;

assign tx_start_condition = (~w_tx_busy) && (~w_tx_fifo_empty);
always_ff @(posedge clk, posedge rst) begin
    if (rst) begin
        tx_start_ff <= 1'b0;
    end else begin
        tx_start_ff <= tx_start_condition;
    end
end

assign tx_start_pulse = tx_start_condition && (!tx_start_ff);
```



tx_start를 1clock 펄스 신호로 만들어 문제 해결

고찰

하드웨어 설계부터 C 코드를 이용한 펌웨어 개발 및 검증까지 **SoC 설계의 전 과정**을 경험함으로써, 하드웨어와 소프트웨어의 상호작용에 대한 깊이 있는 이해를 얻을 수 있었습니다.
그리고 컴퓨터 구조 수업에서 이론으로만 접했을 때는 폴링과 인터럽트 방식의 장단점이 크게 와닿지 않았는데, 이번 프로젝트에서 직접 폴링 방식의 비효율성을 체감할 수 있었습니다.

감사합니다