대한상공회의소
서울기술교육센터

2025.10.27

RISC-V 기반 AMBA Peripheral 설계

하만 세미콘 아카데미 변지인

목차

01

프로젝트 개요 & 목표

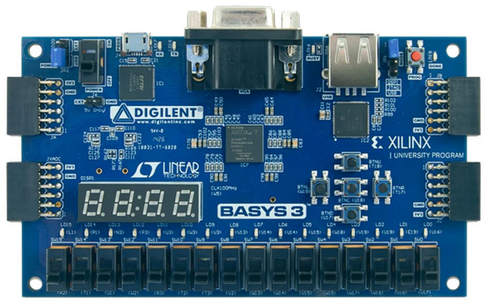
01 프로젝트 개요 & 목표

1. 프로젝트 개요

- 32-bit RISC-V 기반 **Multi Cycle CPU Core**를 설계하고, AMBA **APB Protocol**을 통신 규격으로 사용하여 CPU와 다양한 **Peripheral을 통합**한 시스템을 설계 및 구현

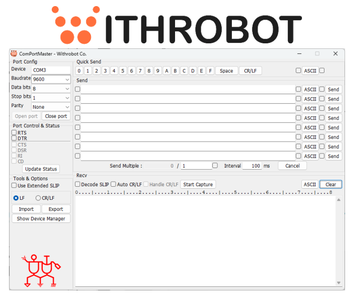
2. 프로젝트 목표

- CPU Core 설계: RISC-V RV32I CPU Core를 Multi-Cycle로 설계
- 시스템 통합: AMBA APB Protocol 기반의 **Master-Slave Interface를 구현**하여 **각종 Peripheral 통합**
- 기능 검증: System Verilog Testbench를 활용하여 UART **Peripheral의 기능 검증**
- 최종 시연: **C Application code**를 ROM에 로드하여 CPU가 이를 실행하는 전체 시스템 동작 시연



RISC-V

ARM AMBA
Interconnect Standards



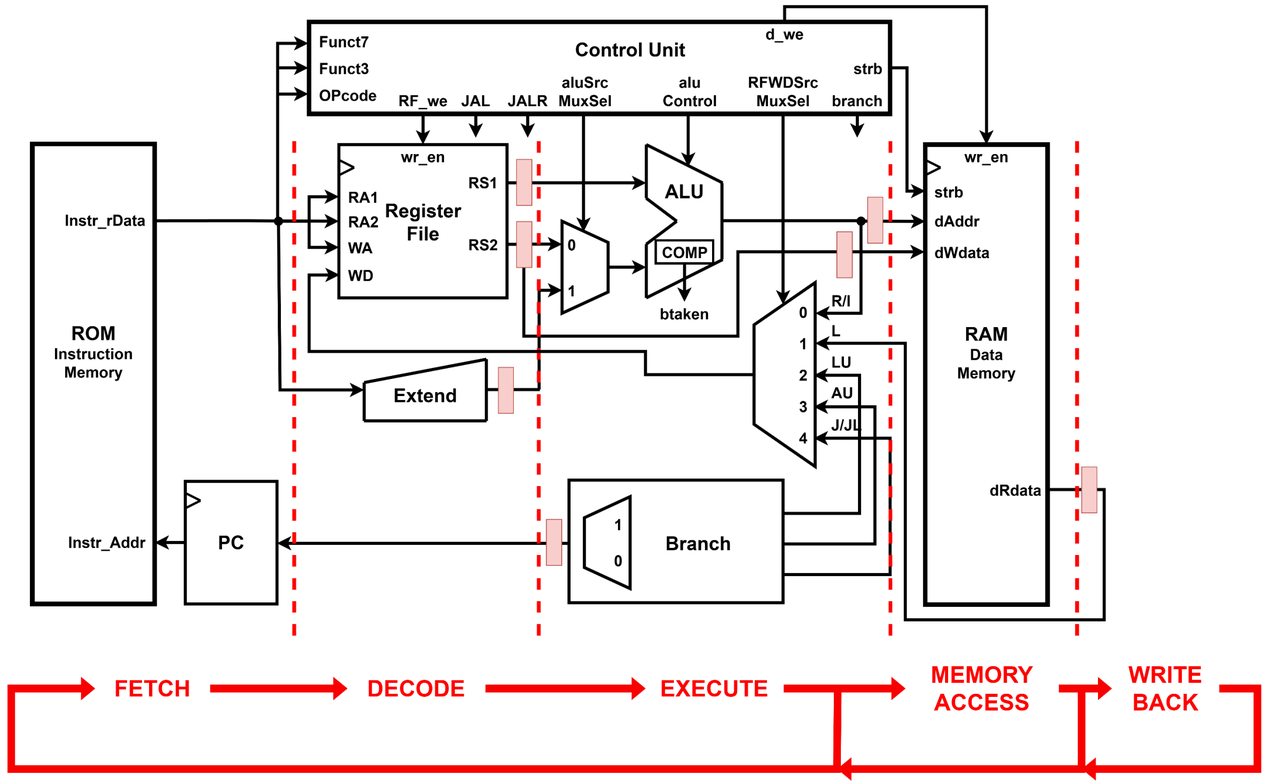
02

Multi-Cycle RISC-V CPU Core 설계

02 Multi-Cycle RISC-V CPU Core 설계

기존 Single-Cycle CPU의 한계

- 모든 명령어를 가장 긴 명령어 기준으로 실행 → 클럭 주기가 묶여 비효율적
- 성능 및 효율 향상을 위해 Multi-Cycle 구조로 설계



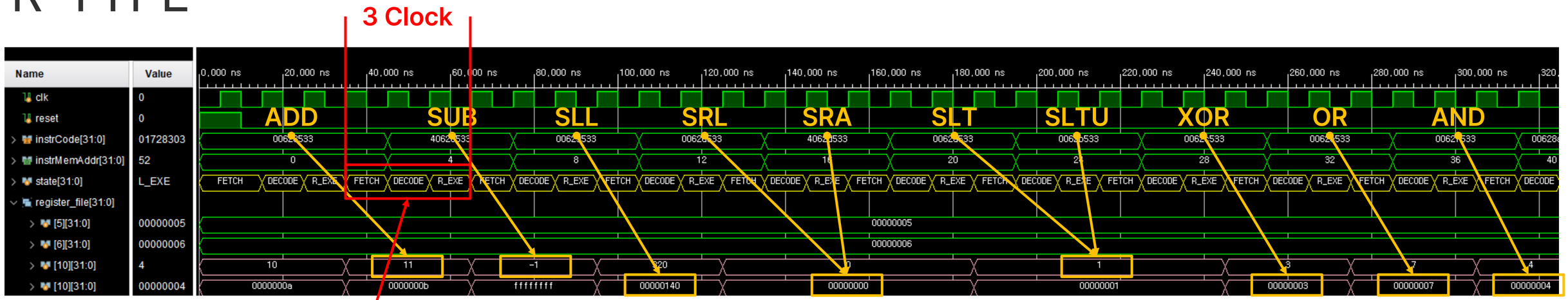
명령어 처리 5단계 (FSM State)

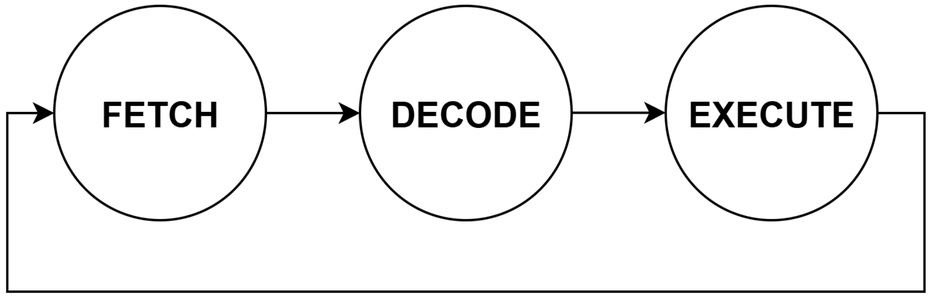
FETCH	명령어 가져오기
DECODE	명령어 해석 및 레지스터 읽기
EXECUTE	ALU 연산 수행
MEMORY	메모리 읽기/쓰기
WRITE BACK	레지스터에 결과 쓰기

02 Multi-Cycle RISC-V CPU Core 설계

TYPE 별 동작 확인

R-TYPE



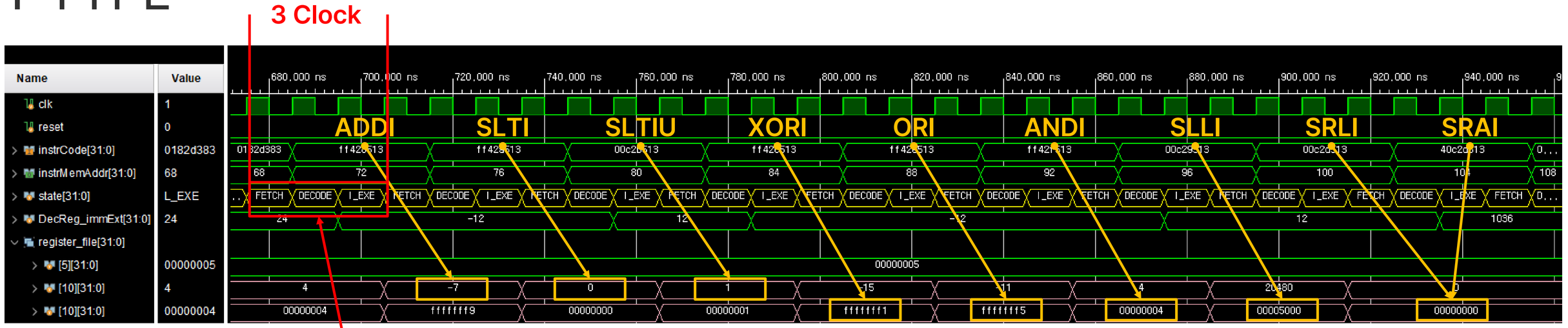


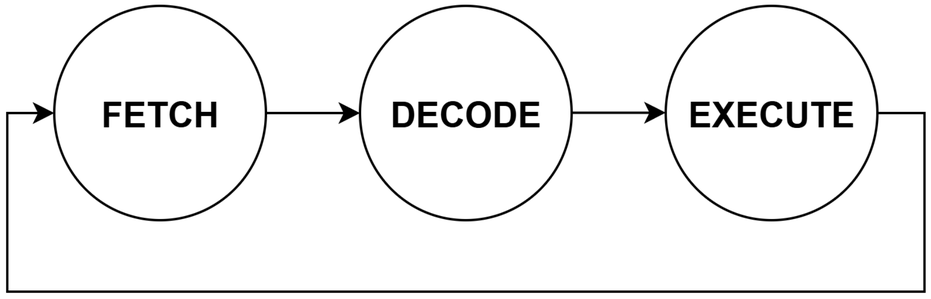
TYPE	MNEMONIC	NAME	Assembly	Description	Expected Result
R-TYPE	ADD	ADD	add x10, x5, x6	x10 = x5 + x6	11
	SUB	SUB	sub x10, x5, x6	x10 = x5 - x6	-1
	SLL	Shift Left Logical	sll x10, x5, x6	x10 = x5 << x6[0:4]	32'h0000_0140
	SRL	Shift Right Logical	srl x10, x5, x6	x10 = x5 >> x6[0:4]	32'h0000_0000
	SRA	Shift Right Arith	sra x10, x5, x6	x10 = x5 >>> x6[0:4]	32'h0000_0000
	SLT	Set Less Than	slt x10, x5, x6	x10 = (x5 < x6) ? 1 : 0	1
	SLTU	Set Less Than (U)	sltu x10, x5, x6	x10 = (x5 < x6) ? 1 : 0	1
	XOR	XOR	xor x10, x5, x6	x10 = x5 ^ x6	32'h0000_0003
	OR	OR	or x10, x5, x6	x10 = x5 | x6	32'h0000_0007
	AND	AND	and x10, x5, x6	x10 = x5 & x6	32'h0000_0004

02 Multi-Cycle RISC-V CPU Core 설계

TYPE 별 동작 확인

I-TYPE



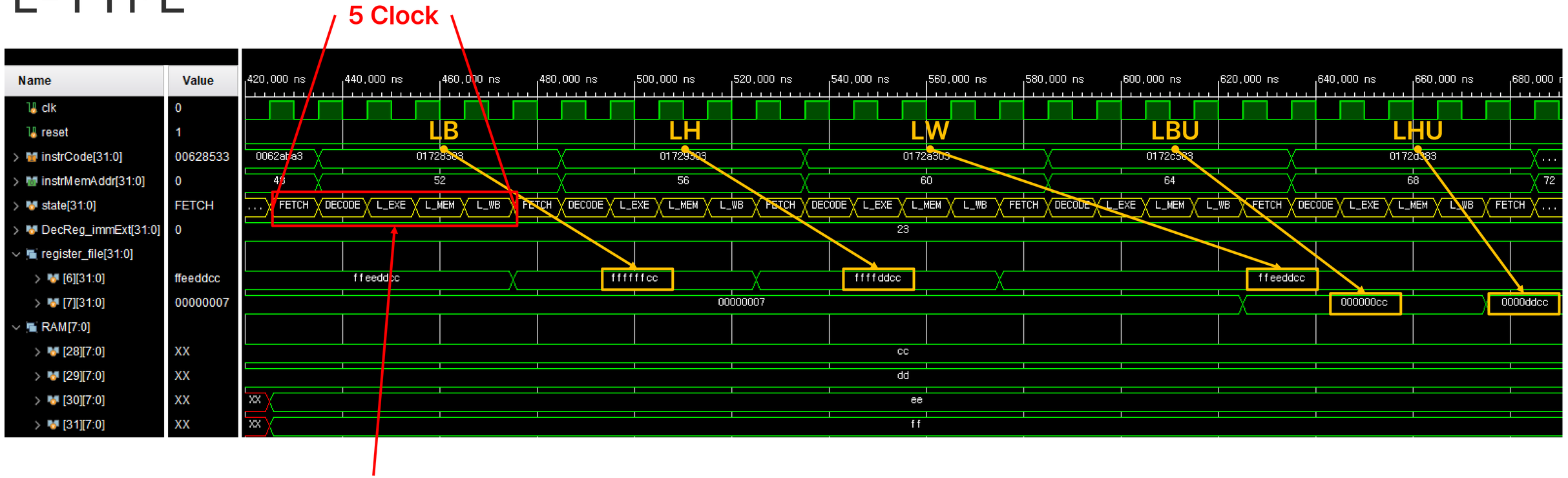


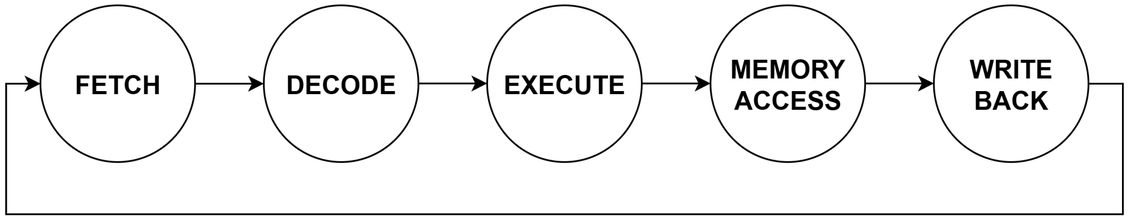
TYPE	MNEMONIC	NAME	Assembly	Description	Expected Result
I-TYPE	ADDI	ADD Immediate	addi x10, x5, 10	x10 = x5 + 10	-7
	SLTI	Set Less Than Imm	slti x10, x5, 10	x10 = (x5 < 10) ? 1 : 0	0
	SLTIU	Set Less Than Imm (U)	sltiu x10, x5, 10	x10 = (x5 < 10) ? 1 : 0	1
	XORI	XOR Immediate	xori x10, x5, 10	x10 = x5 ^ 10	32'hffff_fff1
	ORI	OR Immediate	ori x10, x5, 10	x10 = x5 | 10	32'hffff_fff5
	ANDI	AND Immediate	andi x10, x5, 10	x10 = x5 & 10	32'h0000_0004
	SLLI	Shift Left Logical Imm	slli x10, x5, 10	x10 = x5 << 10[0:4]	32'h0000_5000
	SRLI	Shift Right Logical Imm	srli x10, x5, 10	x10 = x5 >> 10[0:4]	32'h0000_0000
	SRAI	Shift Right Arith Imm	srai x10, x5, 10	x10 = x5 >>> 10[0:4]	32'h0000_0000

02 Multi-Cycle RISC-V CPU Core 설계

TYPE 별 동작 확인

L-TYPE



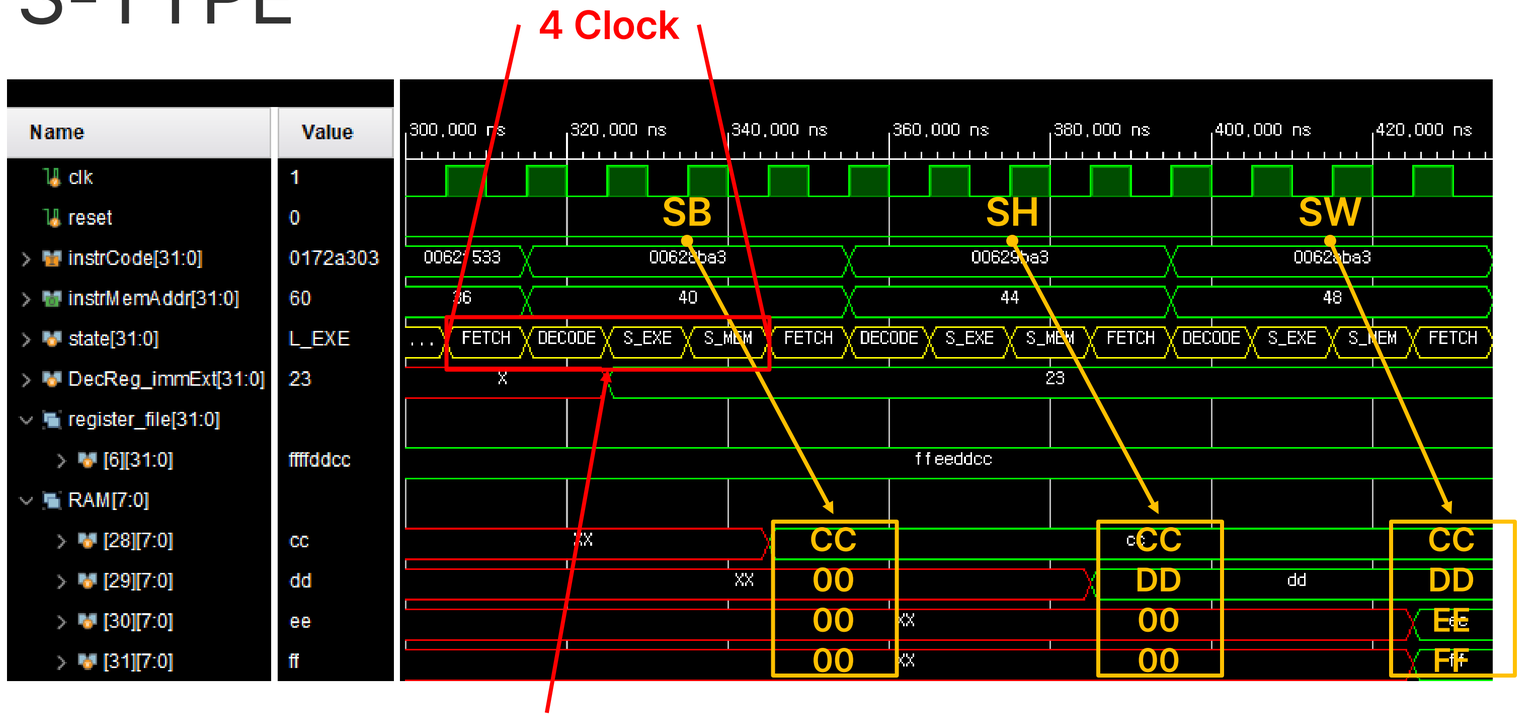


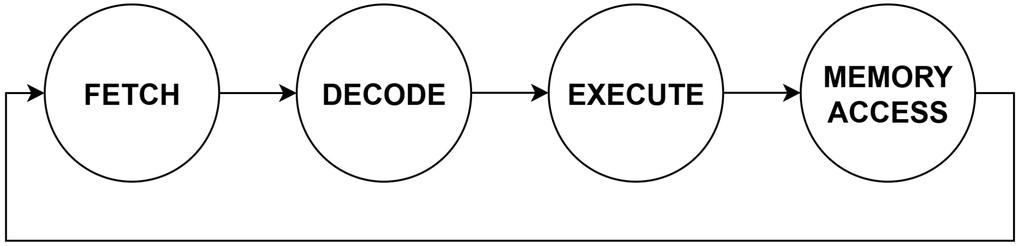
TYPE	MNEMONIC	NAME	Assembly	Description	Expected Result
L-TYPE	LB	Load Byte	lb x6, 23(x5)	x6 = Mem[x5+23][0:7]	x6 = 32'hFFFF_FFCC
	LH	Load Half	lh x6, 23(x5)	x6 = Mem[x5+23][0:15]	x6 = 32'hFFFF_DDCC
	LW	Load Word	lw x6, 23(x5)	x6 = Mem[x5+23][0:31]	x6 = 32'hFFEE_DDCC
	LBU	Load Byte (U)	lbu x7 23(x5)	x7 = Mem[x5+23][0:7]	x7 = 32'h0000_00CC
	LHU	Load Half (U)	lhu x7 23(x5)	x7 = Mem[x5+23][0:15]	x7 = 32'h0000_DDCC

02 Multi-Cycle RISC-V CPU Core 설계

TYPE 별 동작 확인

S-TYPE



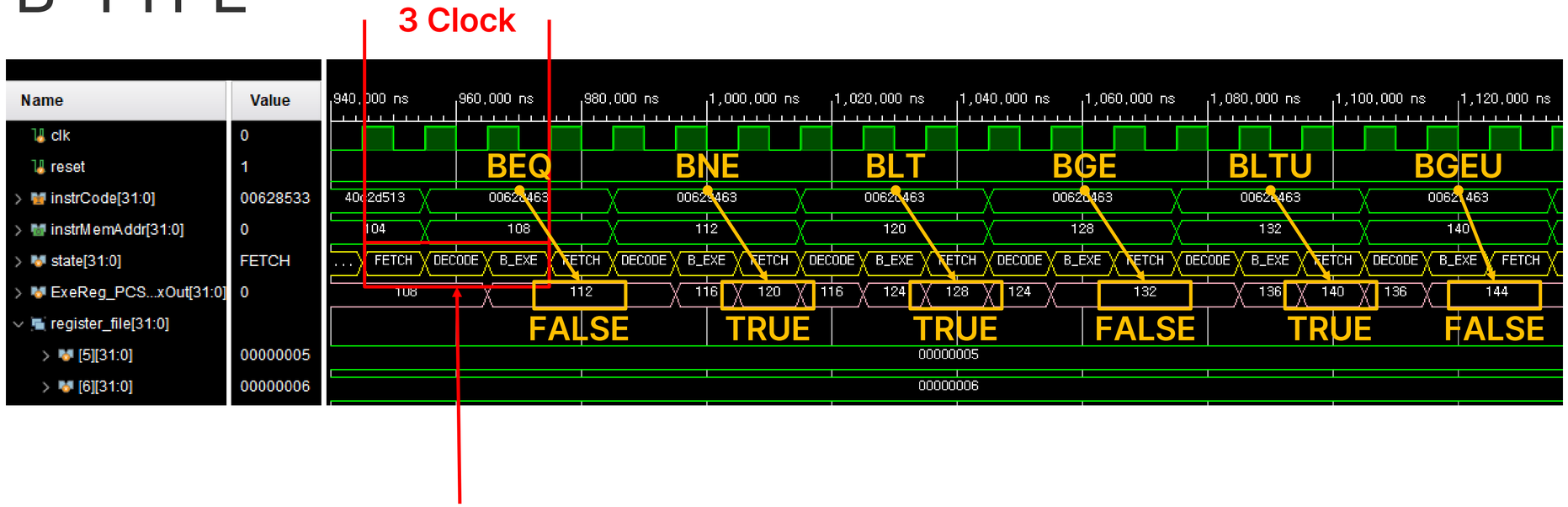


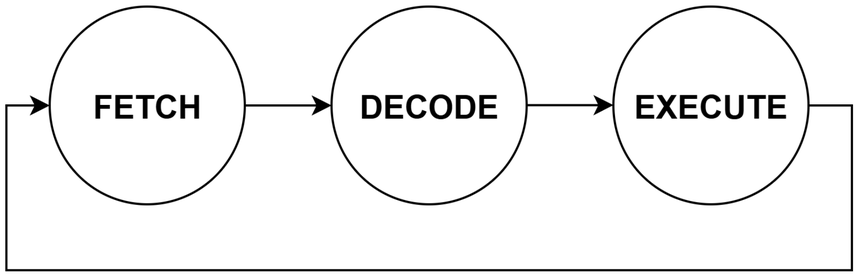
TYPE	MNEMONIC	NAME	Assembly	Description	Expected Result
S-TYPE	SB	Store Byte	sb x6, 23(x5)	Mem[x5+23] = x6[0:7]	Mem[28] = 8'hCC
	SH	Store Half	sh x6, 23(x5)	Mem[x5+23] = x6[0:15]	Mem[28] = 16'hDDCC
	SW	Store Word	sw x6, 23(x5)	Mem[x5+23] = x6[0:31]	Mem[28] = 32'hFFEE_DDEE

02 Multi-Cycle RISC-V CPU Core 설계

TYPE 별 동작 확인

B-TYPE



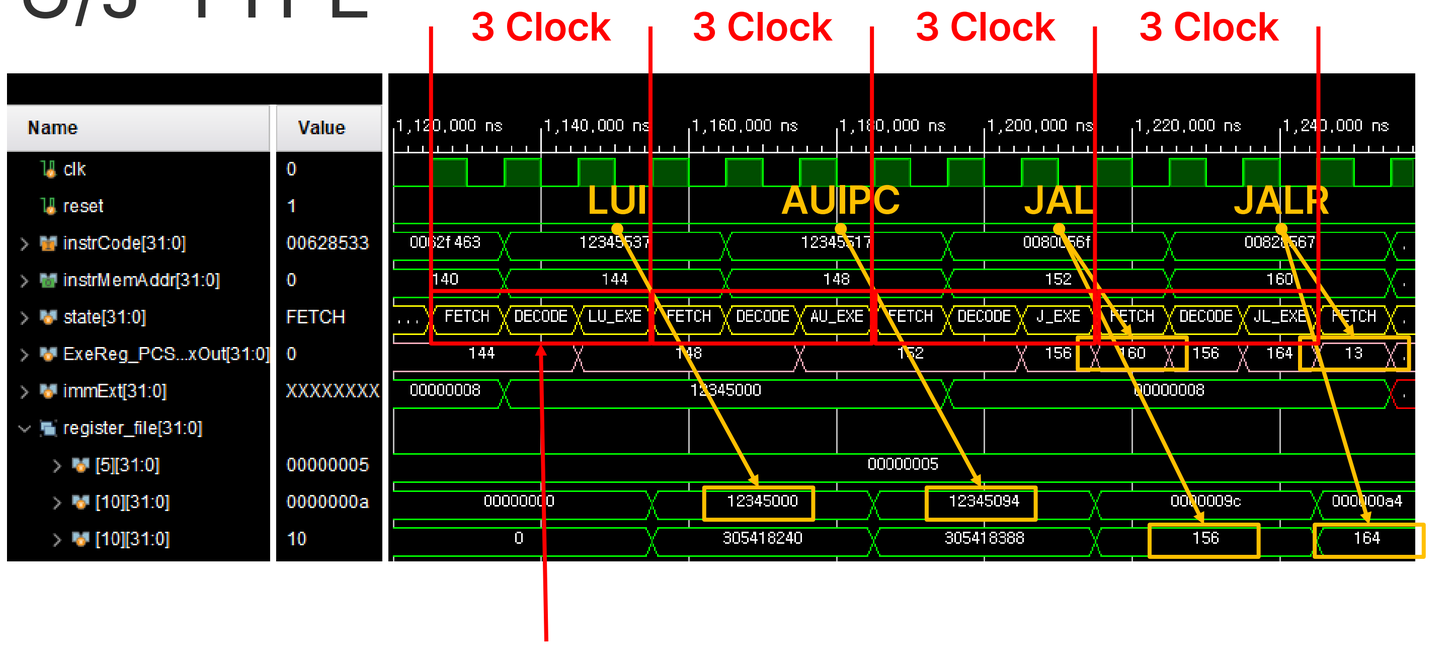


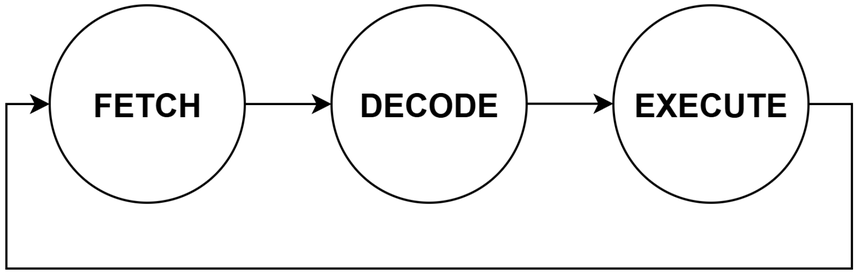
TYPE	MNEMONIC	NAME	Assembly	Description	Expected Result
B-TYPE	BEQ	Branch ==	beq x5, x6, 8	if(x5 == x6) PC += 8	FLASE, PC += 4
	BNE	Branch !=	bne x5, x6, 8	if(x5 != x6) PC += 8	TRUE, PC += 8
	BLT	Branch <	blt x5, x6, 8	if(x5 < x6) PC += 8	TRUE, PC += 8
	BGE	Branch >=	bge x5, x6, 8	if(x5 >= x6) PC += 8	FLASE, PC += 4
	BLTU	Branch < (U)	bltu x5, x6, 8	if(x5 < x6) PC += 8	TRUE, PC += 8
	BGEU	Branch >= (U)	bgeu x5, x6, 8	if(x5 >= x6) PC += 8	FLASE, PC += 4

02 Multi-Cycle RISC-V CPU Core 설계

TYPE 별 동작 확인

U/J-TYPE





TYPE	MNEMONIC	NAME	Assembly	Description
LU-TYPE	LUI	Load Upper Imm	lui x10, 0x12345	x10 = 0x12345000
AU-TYPE	AUIPC	Add Upper Imm to PC	auipc x10, 0x12345	x10 = PC + 0x12345000
J-TYPE	JAL	Jump And Link	jal x10, 8	x10 = PC+4; PC += 8
JL-TYPE	JALR	Jumpp And Link Reg	jalr x10, 8(x5)	x10 = PC+4; PC = x5 + 8

03

AMBA APB Protocol 설계

03 AMBA APB Protocol 설계

AMBA APB

- 저전력, 저속 주변 장치를 연결을 위한 단순한 인터페이스 버스로, Non-pipeline 구조
- CPU와 다양한 저속 주변 장치(UART, 센서 등) 간의 표준화된 데이터 교환 방식 제공
- 간단한 Read/Write 방식으로 장치들을 연결하여 IP 재사용성 및 확장성을 높임

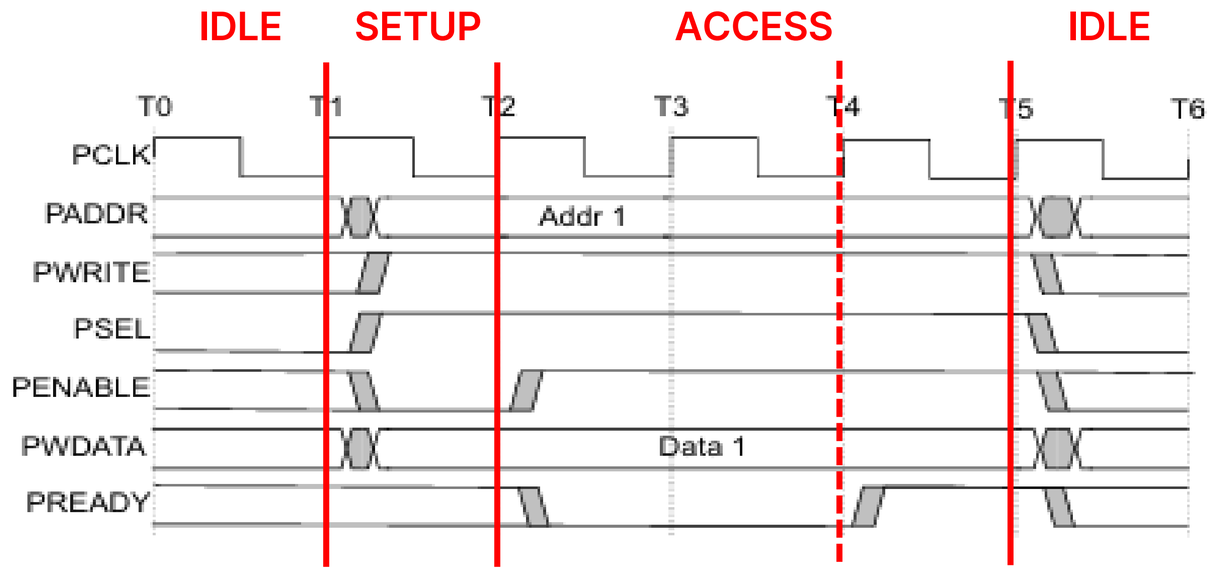


Figure 3-2 Write transfer with wait states

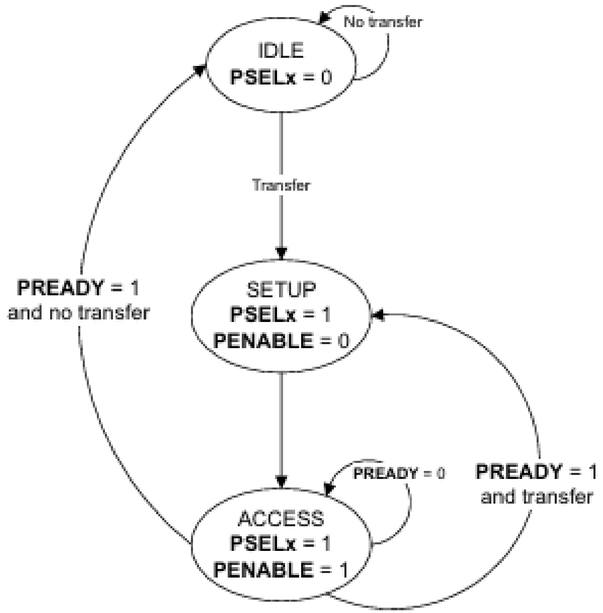
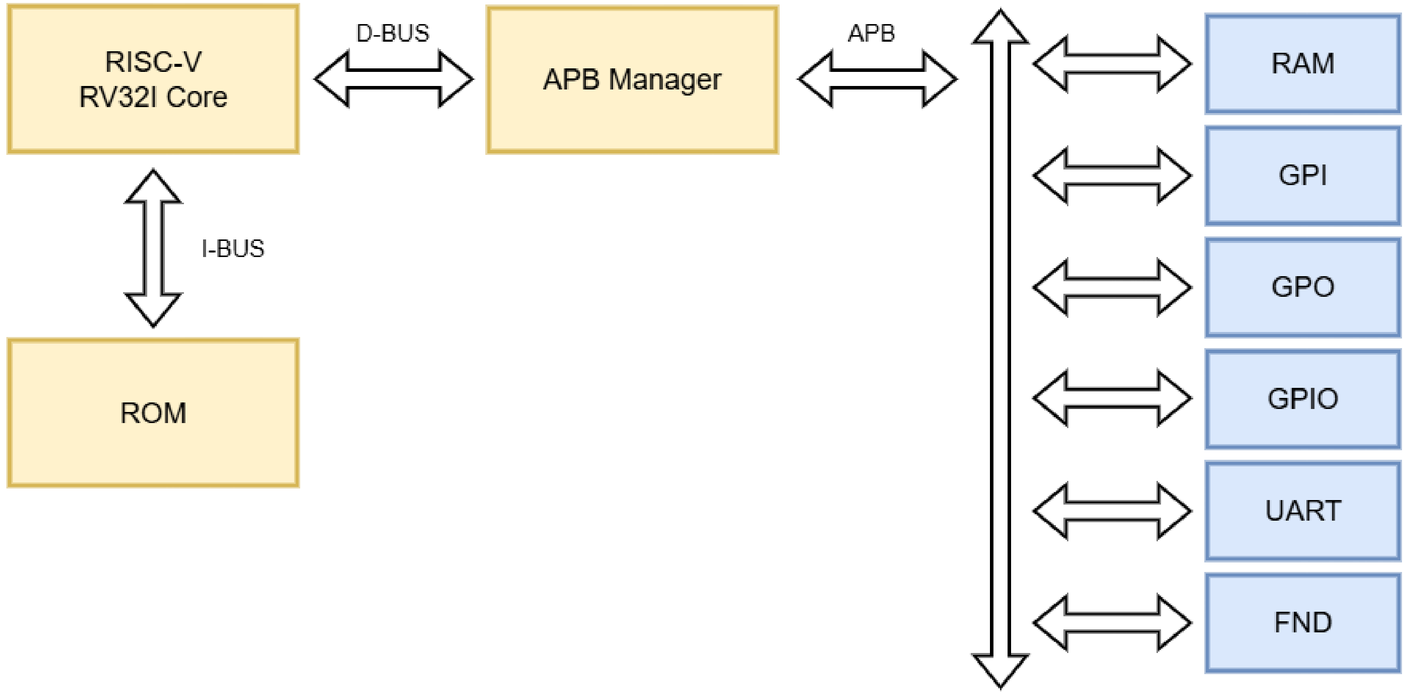


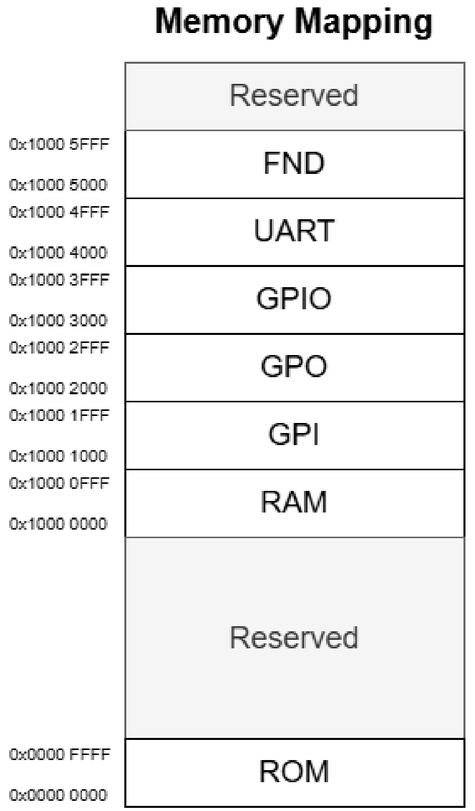
Figure 4-1 State diagram

04

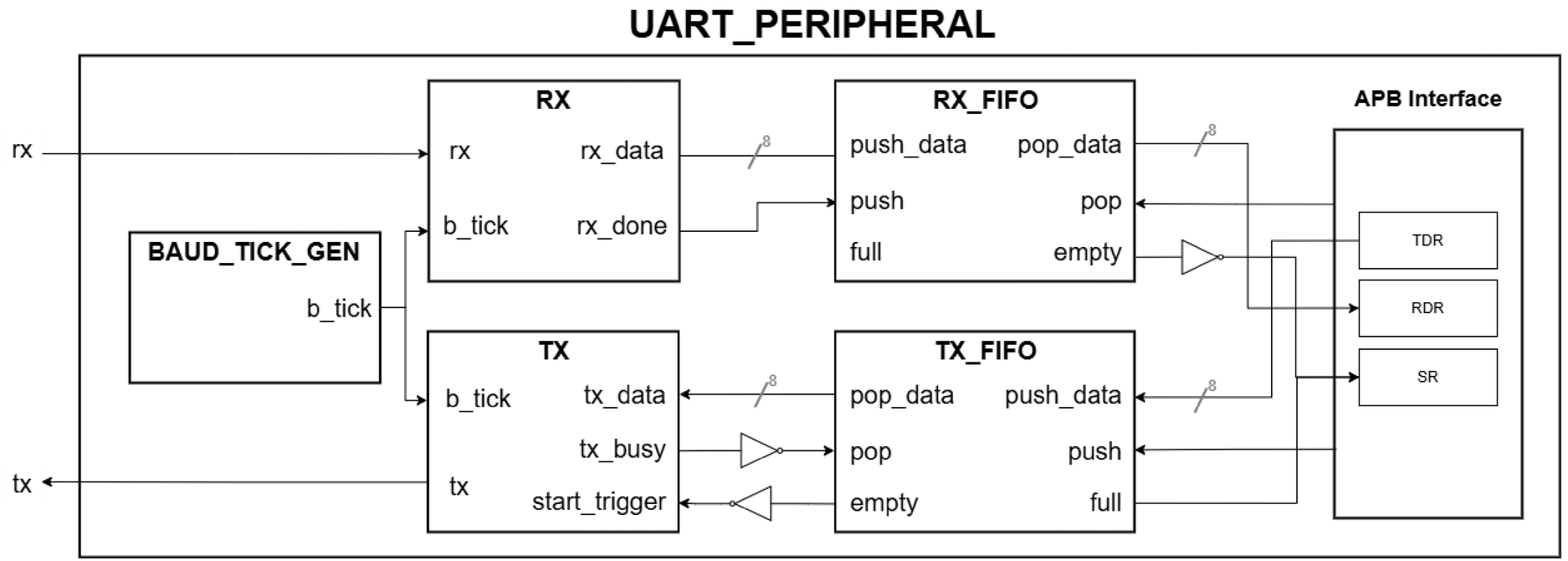
Peripheral 설계 & UART 검증

04 Peripheral 설계 & UART 검증

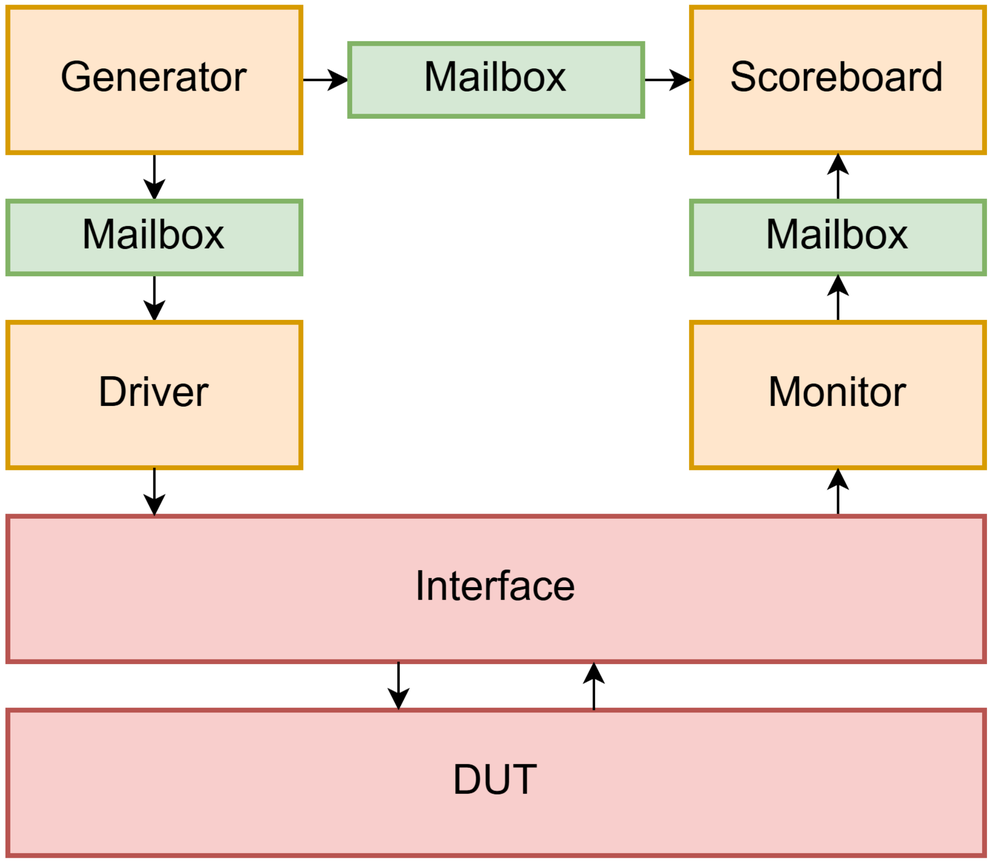




UART Peripheral 설계



offest	Register	31…8	7	6	5	4	3	2	1	0
0000	TDR	Reserved	TDR[7:0]							
0004	RDR	Reserved	RDR[7:0]							
0008	SR	Reserved							TX_full	RX_empty


Generator
Mailbox
Scoreboard
Mailbox
Mailbox
Driver
Monitor
Interface
DUT

Transaction

```
class transaction;
    // random signals
    rand logic [7:0] send_data;
    rand logic [7:0] PWDATA;
    rand logic PWRITE;
    // input signals
    logic [3:0] PADDR;
    logic PENABLE;
    logic PSEL;
    // output signals
    logic [7:0] PRDATA;
    logic PREADY;
    logic [7:0] receive_data;

    function void post_randomize();
        if (PWRITE) PADDR = 4'h0;
        else PADDR = 4'h4;
    endfunction
```

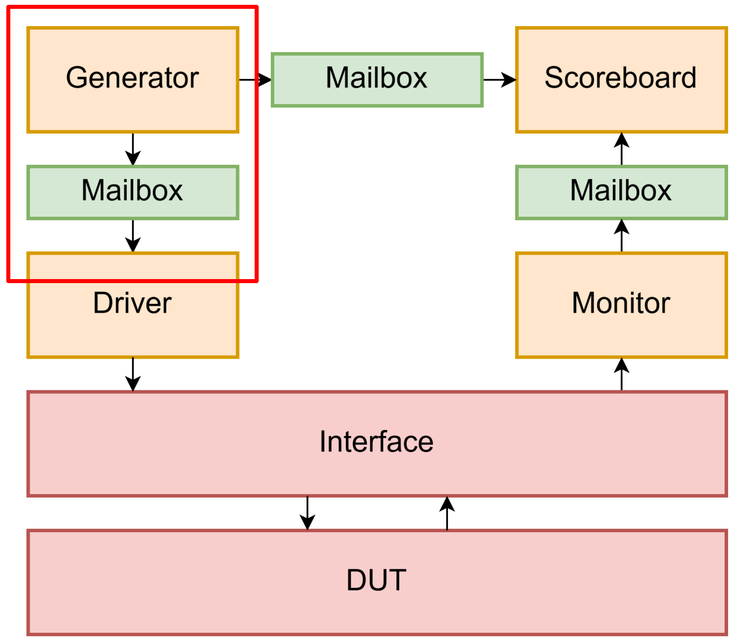
send_data : for rx (PC -> DUT)

receive_data : for tx (DUT -> PC)

Generator

Transaction의 랜덤 신호를 생성

Mailbox를 통해 Driver로 Transaction의 신호를 인가



Driver

```
task run();
    forever begin
        @(posedge UART_Periph_if.PCLK);
        gen2drv_mbox.get(tr);
        tr.display("DRV");

        if (!tr.PWRITE) begin
            send(tr.send_data);
            ->mon_next_event;
            repeat (10) @(posedge UART_Periph_if.PCLK);
        end else begin
            ->mon_next_event;
            repeat (10) @(posedge UART_Periph_if.PCLK);
        end
        UART_Periph_if.PENABLE <= 1'b0;
        UART_Periph_if.PSEL    <= 1'b1;
        UART_Periph_if.PADDR   <= tr.PADDR;
        UART_Periph_if.PWDATA  <= tr.PWDATA;
        UART_Periph_if.PWRITE  <= tr.PWRITE;
        @(posedge UART_Periph_if.PCLK);
        UART_Periph_if.PENABLE <= 1'b1;
        do
        @(posedge UART_Periph_if.PCLK);
        while (UART_Periph_if.PREADY !== 1);
        @(posedge UART_Periph_if.PCLK);
        UART_Periph_if.PENABLE <= 1'b0;
        UART_Periph_if.PSEL    <= 1'b0;
    end
endtask
```

랜덤 생성된 PWRITE 신호가 0이라면

send_data를 1bit씩 rx에 전달

Monitor

```
task run();
    forever begin
        @(mon_next_event);

        do
        @(posedge UART_Periph_if.PCLK);
        while (!(UART_Periph_if.PSEL && UART_Periph_if.PENABLE &&
                 UART_Periph_if.PREADY));

        tr = new();
        tr.PADDR = UART_Periph_if.PADDR;
        tr.PWDATA = UART_Periph_if.PWDATA;
        tr.PWRITE = UART_Periph_if.PWRITE;
        tr.PENABLE = UART_Periph_if.PENABLE;
        tr.PSEL = UART_Periph_if.PSEL;
        tr.PRDATA = UART_Periph_if.PRDATA;
        tr.PREADY = UART_Periph_if.PREADY;

        if (UART_Periph_if.PWRITE) begin
            // @(negedge UART_Periph_if.tx);
            repeat (2) @(posedge UART_Periph_if.PCLK);

            repeat (CLOCKS_PER_BIT) @(posedge UART_Periph_if.PCLK);
            tr.receive_data = 8'b0;
            for (i = 0; i < 8; i++) begin
                tr.receive_data[i] = UART_Periph_if.tx;
                repeat (CLOCKS_PER_BIT) @(posedge UART_Periph_if.PCLK);
            end

            repeat (CLOCKS_PER_BIT) @(posedge UART_Periph_if.PCLK);
        end

        mon2scb_mbox.put(tr);
        tr.display("MON");

    end
endtask
```

DUT의 PWRITE 신호가 1이라면
tx를 1bit씩 receive_data로 저장

Scoreboard

```
task run();
    forever begin
        gen2scb_mbox.get(gen_tr);
        mon2scb_mbox.get(mon_tr);
        mon_tr.display("SCB");

        if (mon_tr.PWRITE) begin
            if (gen_tr.PWDATA == mon_tr.receive_data) begin
                pass_count++;
                $display("[SCB] PASS (TX): Expected = %h, Received = %h",
                         mon_tr.receive_data, mon_tr.PWDATA);
            end else begin
                fail_count++;
                $display("[SCB] FAIL (TX): PWDATA : %h, receive_data : %h",
                         gen_tr.PWDATA, mon_tr.receive_data);
            end
            ->gen_next_event;
        end else begin
            if (gen_tr.send_data == mon_tr.PRDATA) begin
                pass_count++;
                $display("[SCB] PASS (RX): Expected = %h, Received = %h",
                         gen_tr.send_data, mon_tr.PRDATA);
            end else begin
                fail_count++;
                $display(
                    "[SCB] FAIL (RX): gen_tr.send_data : %h, PRDATA : %h",
                    gen_tr.send_data, mon_tr.PRDATA);
            end
            ->gen_next_event;
        end
    end
endtask
```

PWRITE가 1, 즉 WRITE일 때는
PWDATA와 receive_data가 같으면 PASS

PWRITE가 0, 즉 READ일 때는
send_data와 PRDATA가 같으면 PASS

Scoreboard

```
[GEN] PADDR = 4, PWDATA = 66, PWRITE = 0, PENABLE = x, PSEL = x, PRDATA = xx, PREADY = x, send_data = 62, receive_data = xx
[DRV] PADDR = 4, PWDATA = 66, PWRITE = 0, PENABLE = x, PSEL = x, PRDATA = xx, PREADY = x, send_data = 62, receive_data = xx
[MON] PADDR = 4, PWDATA = 66, PWRITE = 0, PENABLE = 1, PSEL = 1, PRDATA = 62, PREADY = 1, send_data = xx, receive_data = xx
[SCB] PADDR = 4, PWDATA = 66, PWRITE = 0, PENABLE = 1, PSEL = 1, PRDATA = 62, PREADY = 1, send_data = xx, receive_data = xx
[SCB] PASS (RX): Expected = 62, Received = 62
[GEN] PADDR = 0, PWDATA = 63, PWRITE = 1, PENABLE = x, PSEL = x, PRDATA = xx, PREADY = x, send_data = aa, receive_data = xx
[DRV] PADDR = 0, PWDATA = 63, PWRITE = 1, PENABLE = x, PSEL = x, PRDATA = xx, PREADY = x, send_data = aa, receive_data = xx
[MON] PADDR = 0, PWDATA = 63, PWRITE = 1, PENABLE = 1, PSEL = 1, PRDATA = 62, PREADY = 1, send_data = xx, receive_data = 63
[SCB] PADDR = 0, PWDATA = 63, PWRITE = 1, PENABLE = 1, PSEL = 1, PRDATA = 62, PREADY = 1, send_data = xx, receive_data = 63
[SCB] PASS (TX): Expected = 63, Received = 63
```

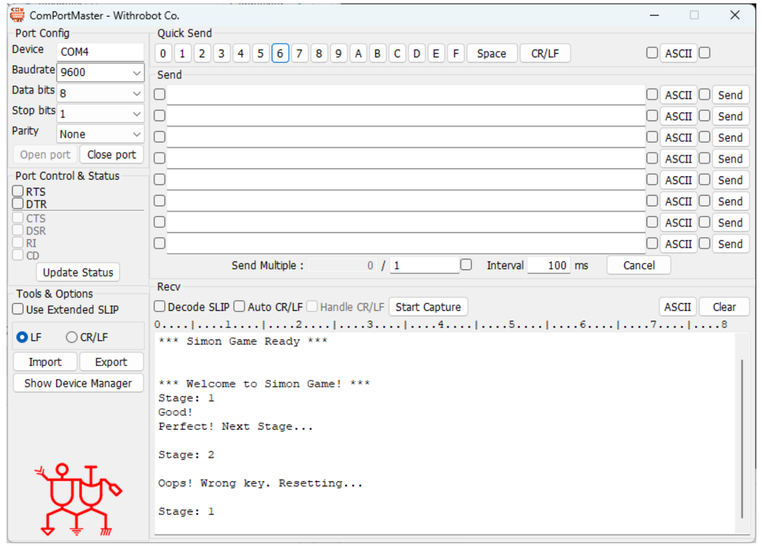
```
=======================================
============ test report ==============
=======================================
==      Total Test :         100      ==
==       Pass Test :         100      ==
==       Fail Test :           0      ==
=======================================
==        Test bench is finish        ==
=======================================
```

05

C언어 Application & 동작 영상

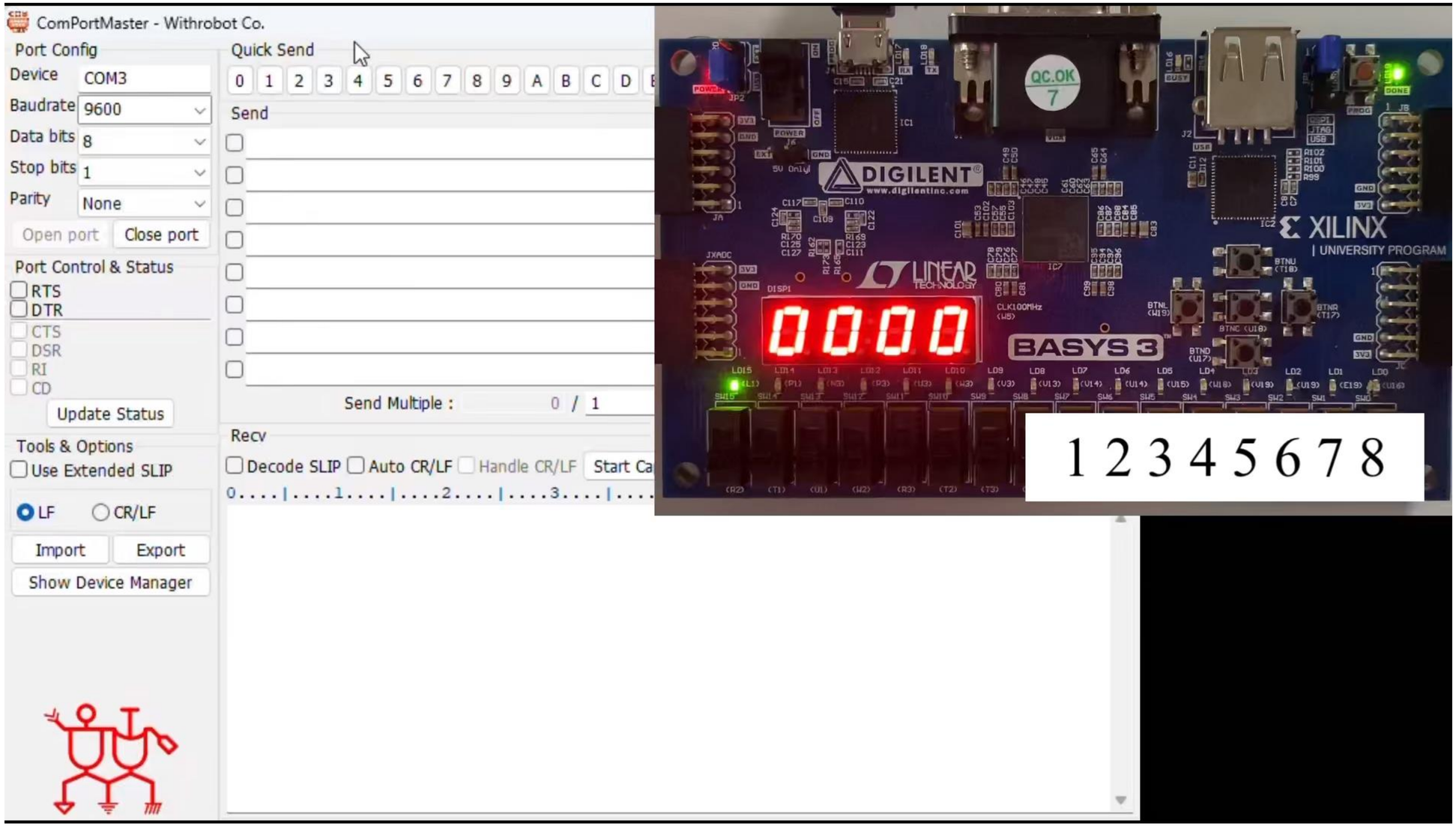
Simon Game

1. **Ready :** ready 문구 출력 & FND에 0 표시 후 sw[15]가 1이 될 때까지 대기
2. **Start :** sw[15]가 1이 되면 패턴 세트 선택 후 welcome 문구 출력한 뒤 시작
3. **Show Pattern :** 선택된 패턴으로 stage 단계에 따라 led[0] ~ [7]이 랜덤으로 점멸
4. **Get Input :** UART 입력 대기 & 정답 확인. 오답일 경우 stage 1로 돌아감
5. **Stage Clear & Win :** 모두 맞추게 되면 WIN 문구 출력 후 다음 패턴을 준비하고, stage 1로 돌아감



05 C언어 Application & 동작 영상

동작영상



06

고찰

06 고찰

Trouble Shooting 1

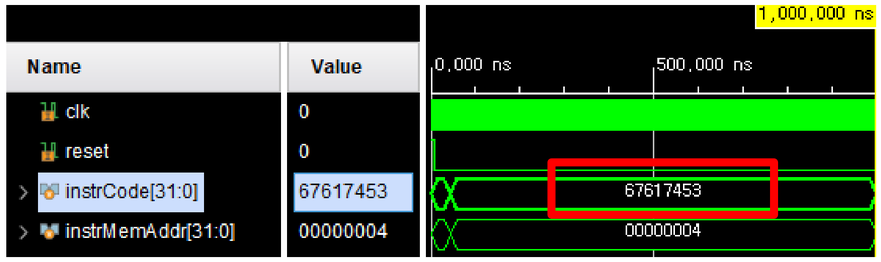
1. 문제 현상

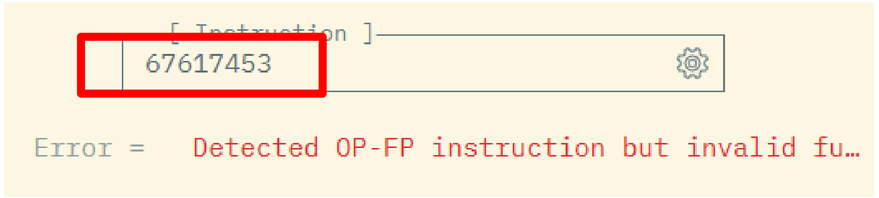
- C Application에서 초기화된 전역/정적 변수 또는 string 관련 함수 사용 시, CPU가 ROM Fetch 과정에서 멈추는 현상 발생

2. 원인

- Bare Metal 환경인 RV32I Core 특성상 Linker Script가 없기 때문에 발생
- C 컴파일러가 생성한 Data Segment가 RAM 주소에 배치되지 못하고 ROM 명령어로 합쳐 들어옴
- CPU가 유효하지 않은 Opcode를 만나 동작이 중단됨

```
1   .LC0:
2           .string "Stage: 1"
3   .LC1:
4           .string "\r\n"
```





3. 해결 방안

- 모든 상태 변수를 지역 변수 및 Hardware Register로만 관리하도록 **C코드 최적화**
- 모든 출력 문자열을 **char**로 각각 함수를 만들어 관리

```
void UART_send_char(char c) {
    UART_TDR = c;
}
```

```
void UART_send_stage(int n) {
    UART_send_char('S'); UART_send_char('t'); UART_send_char('a');
    UART_send_char('g'); UART_send_char('e'); UART_send_char(':');
    UART_send_char(' '); UART_send_char((char)(n + '0'));
    UART_send_char('\r'); UART_send_char('\n');
}
```

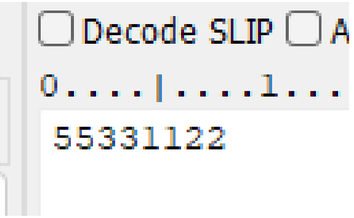
+ rand(), srand() 함수 -> RV32I ISA에서 지원되지 않아 어셈블러 되지 않음
-> 미리 10가지 **고정 패턴 배열을 생성**하여 10개가 반복 실행되도록 구현

06 고찰

Trouble Shooting 2

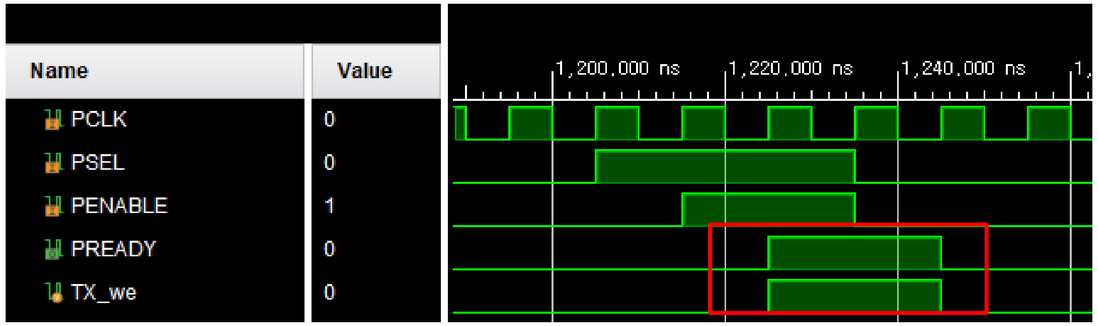
1. 문제 현상

- UART TX/RX 동작 시, 데이터가 의도치 않게 두 번 송/수신되는 오류 발생



2. 원인

- PREADY가 high일 때 PWRITE에 따라서 TX_we, RX_re가 각각 high가 되도록 Slave를 설계
- Master에서 PREADY가 1이 되면 PSEL, PENABLE이 0이 되고, 다음 clk에 PREADY가 0이 됨
- PREADY가 2 clk 동안 high를 유지하여 TX_we, RX_re 신호가 2 clk 동안 활성화됨



06 고찰

Trouble Shooting 2

3. 해결 방안

- 초기 시도 : Master의 PSEL과 PENABLE을 PREADY가 1이 되는 시점에 low가 되도록 코드 수정
  -> PREADY가 high일 때 PSEL과 PENABLE이 0이 되는 것은 표준 Protocol 위반
- 최종 방안 : SIave에 write_done, read_done 신호를 추가하여 TX_we, RX_re가 1 clk만 high가 되도록 수정

초기 시도 -> 표준 Protocol 위반

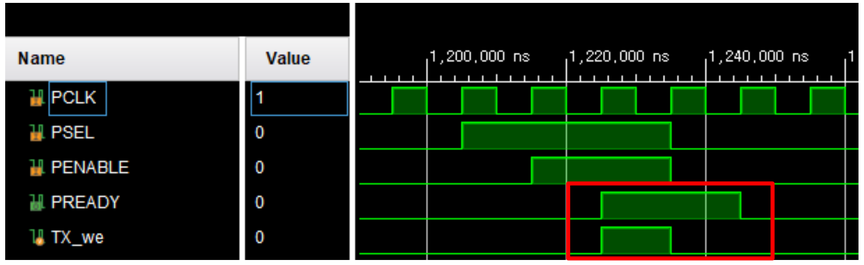
```
ACCESS: begin
    decoder_en = 1'b1;
    PENABLE    = 1'b1;
    if (ready) begin
        decoder_en = 1'b0;
        PENABLE    = 1'b0;
        state_next = IDLE;
    end
end
```

```
1'd0: begin  // TDR
    if (~TX_full) begin
        slv_reg0 <= PWDATA;
        PREADY   <= 1'b1;
        TX_we    <= 1'b1;
    end
end
```

최종 해결 방안

```
ACCESS: begin
    decoder_en = 1'b1;
    PENABLE    = 1'b1;
    if (ready) begin
        state_next = IDLE;
    end
end
```

```
1'd0: begin  // TDR
    if (~TX_full) begin
        PREADY <= 1'b1;
        if (!write_done) begin
            slv_reg0   <= PWDATA;
            TX_we      <= 1'b1;
            write_done <= 1'b1;
        end
```



06 고찰

느낀점

- Linker가 부재한 환경에서는 C 컴파일러가 생성하는 .data 및 .rodata 섹션이 ROM 코드의 실행코드(.text)와 뒤섞여 CPU 동작을 중단시킬 수 있음을 확인했습니다.
- 이를 통해 Linker Script가 코드와 데이터의 메모리 배치를 관리하는 핵심 역할을 수행함을 이해할 수 있었고, 메모리 배치에 대한 이해가 필수적임을 체감하였습니다.

- 단순한 논리 설계 뿐 아니라 시스템 수준 통합 단계에서 발생하는 타이밍, 메모리, 통신 문제를 직접 검증하고 해결하는 경험을 통해, 실제 SoC 설계 프로세스의 복잡성을 학습할 수 있었습니다.

감사합니다.