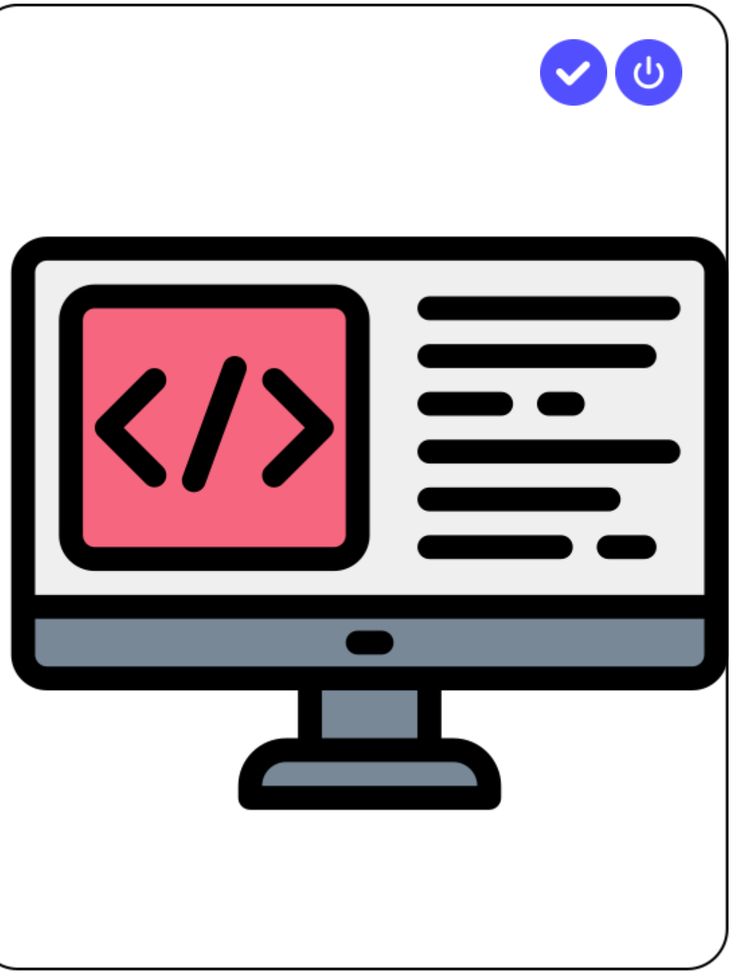
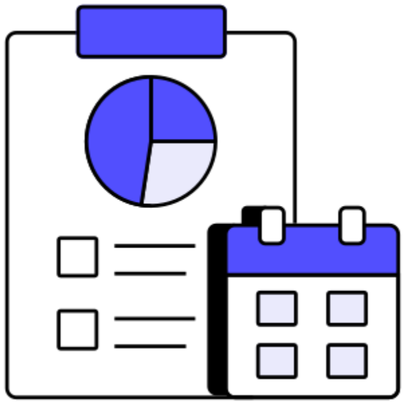
유지훈

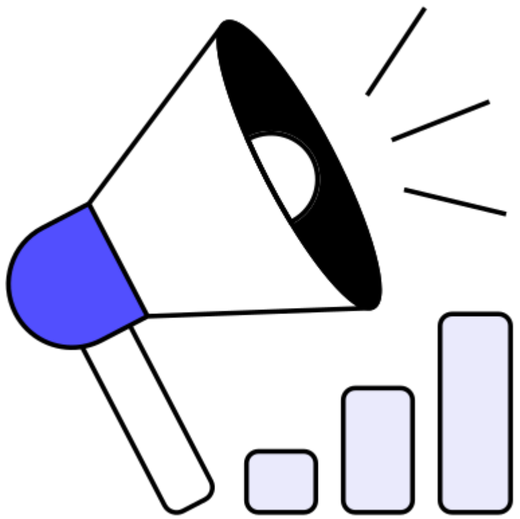
UART/FI-FO



목차



01 Introduction



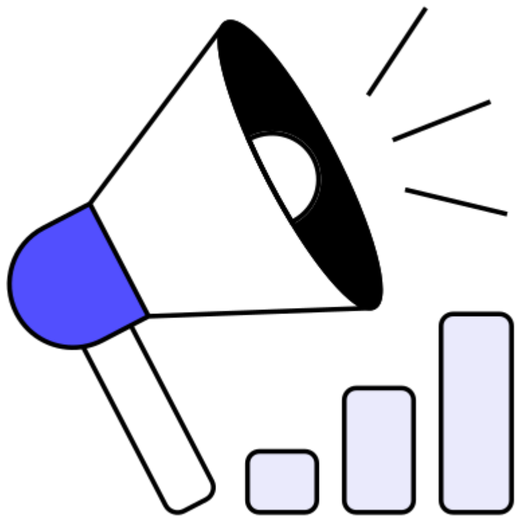
전체 기능

Basys3 FPGA 보드의 스위치(sw)와 버튼(Btn_L, Btn_R, Btn_U, Btn_D) 및 UART 입력을 이용해 동작 제어
mode[1:0] 값에 따라 Watch/Stopwatch 모드를 선택하고, 선택된 결과를 7세그먼트(FND)에 표시
우선순위 제어: 스위치에 따라 보드 버튼과 UART 명령을 동시에 하거나, UART만 제어 가능

목표

FPGA 기반으로 동작하는 이중 모드(Watch / Stopwatch) 시간 표시기 구현

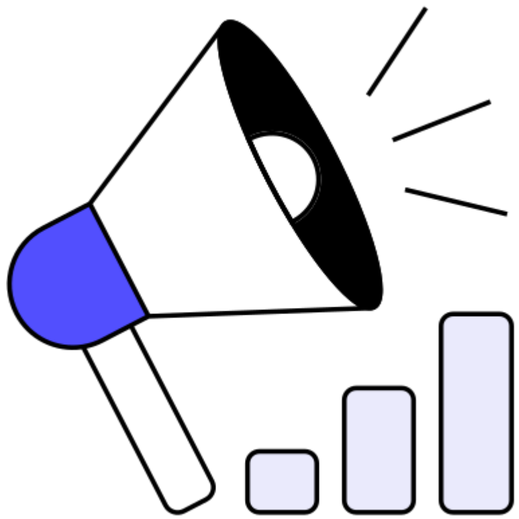
01 Introduction



요약

FPGA 보드를 이용하여 스톱워치 및 시계 기능을 통합한
디지털 시간 표시기
Stopwatch와 Watch는 각각 독립적으로 동작하며,
Mode 선택을 통해 어느 값을 표시할지 결정
버튼 및 UART 입력을 통해 시간 증가, 실행/정지, 초기화 등의
동작을 제어할 수 있으며,
최종 시간 값은 FND Controller를 통해 7-Segment 에 출력

01 Introduction



핵심 기능

Stopwatch 기능

Btn_R: Start/Stop 제어, Btn_L: Clear (00:00초기화)

Watch 기능

Btn_U: 분(Minute) +1, Btn_D: 시(Hour) +1

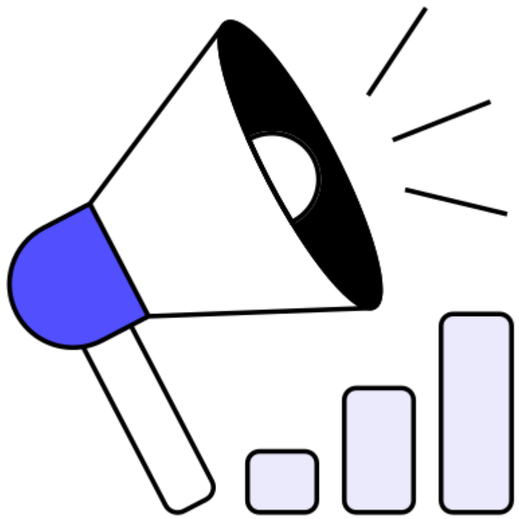
rst_watch: 12:00으로 초기화

Mode 선택 기능

mode == 2'b10 → Watch 출력 표시

mode == 2'b00 / 01 → Stopwatch 출력 표시

01 Introduction



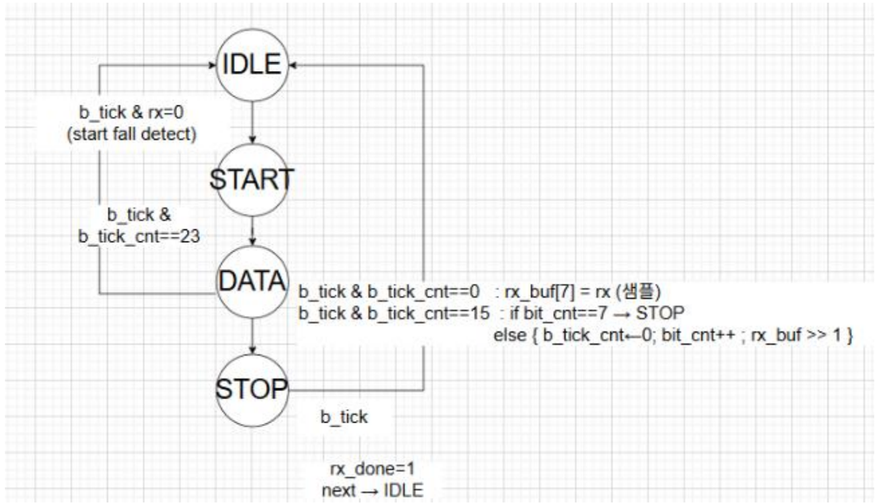
핵심 기능

UART 제어 기능
"0", "1", "2" → 모드 전환
"S" → Start/Stop "C" → Clear "M" → Min +1
문자 "H" → Hour +1 "R" → Watch 리셋 (12:00)

우선순위/통합 제어
sw[2]=1 → UART만 단독 제어
sw[2]=0 → UART와 보드 버튼 입력을 OR 연산으로 병합

02 BLOCK Diagram

uart_rx



IDLE

rx==0 감지 시 START 진입

rx_done=0, 카운터 유지

START

b_tick_cnt 증가

23 도달 시 DATA 진입

DATA

rx_buf에 비트 저장

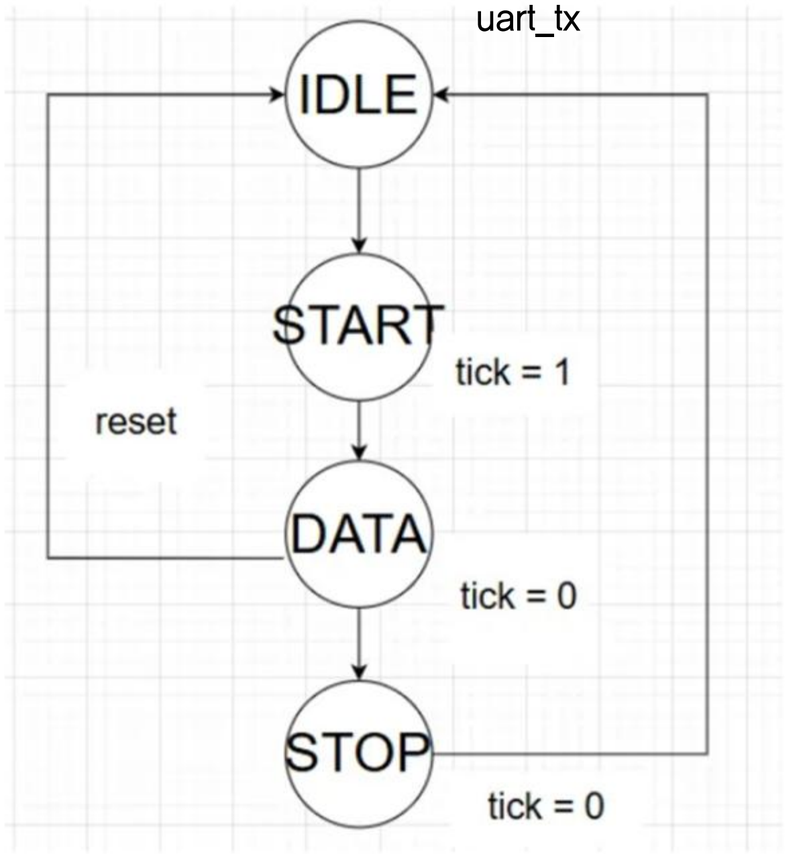
8비트 수신 완료 시 STOP 이동

STOP

rx_done=1 펄스 발생

다시 IDLE 복귀

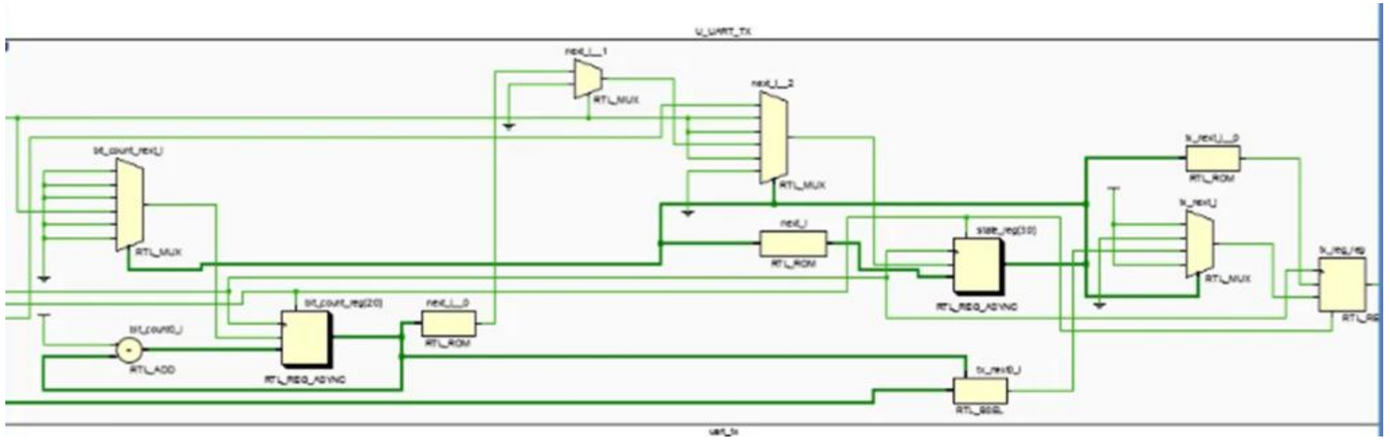
02 BLOCK Diagram(FSM)



case문을 사용해 FSM 코드 작성
코드 간편화를 위해 count를 사용해
bit_count =7이 될때까지 증가 시킴

```
BIT : begin
    tx_next = tx_data[bit_count];

    if (b_tick) begin
        bit_count_next = bit_count+ 1;
        if (bit_count == 7)


            next = STOP;
    end
end
```

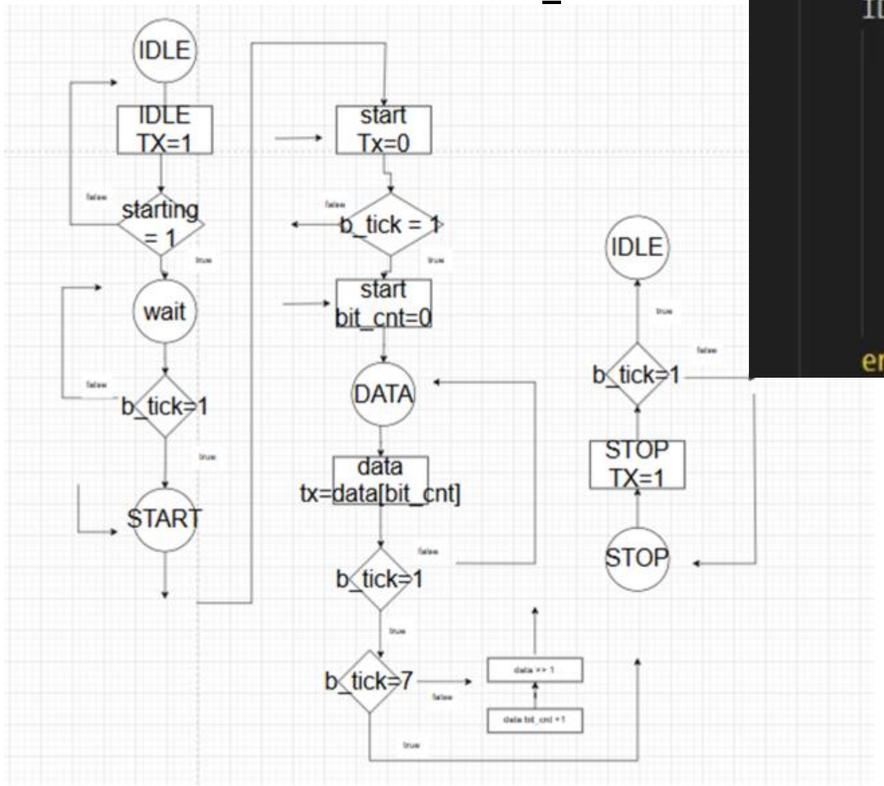
03 Schematic, 시뮬레이션 결과



UART_tx -> UART_rx 에서
전송시 방해 받지 않기 위해
ASM으로 수정

02 BLOCK Diagram(ASM)

uart_tx



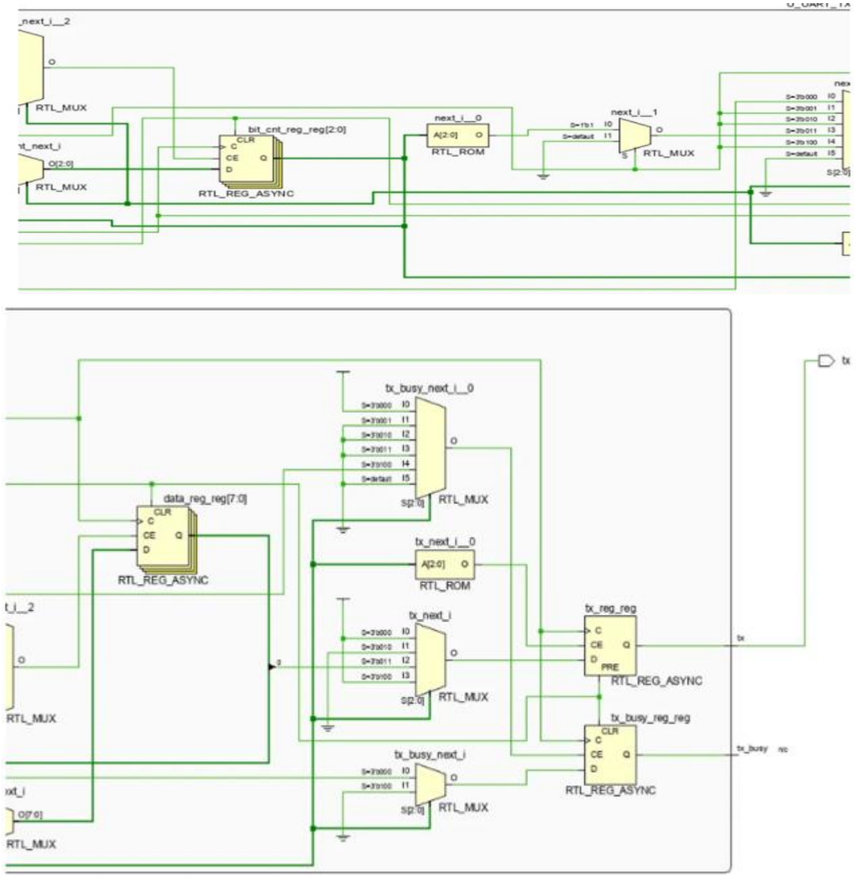

```
IDLE: begin
    //output tx
    tx_next = 1'b1;
    tx_busy_next = 1'b0;
    if (start_trigger == 1'b1) begin
        tx_busy_next = 1'b1;
        next = WAIT;
    end
end
```

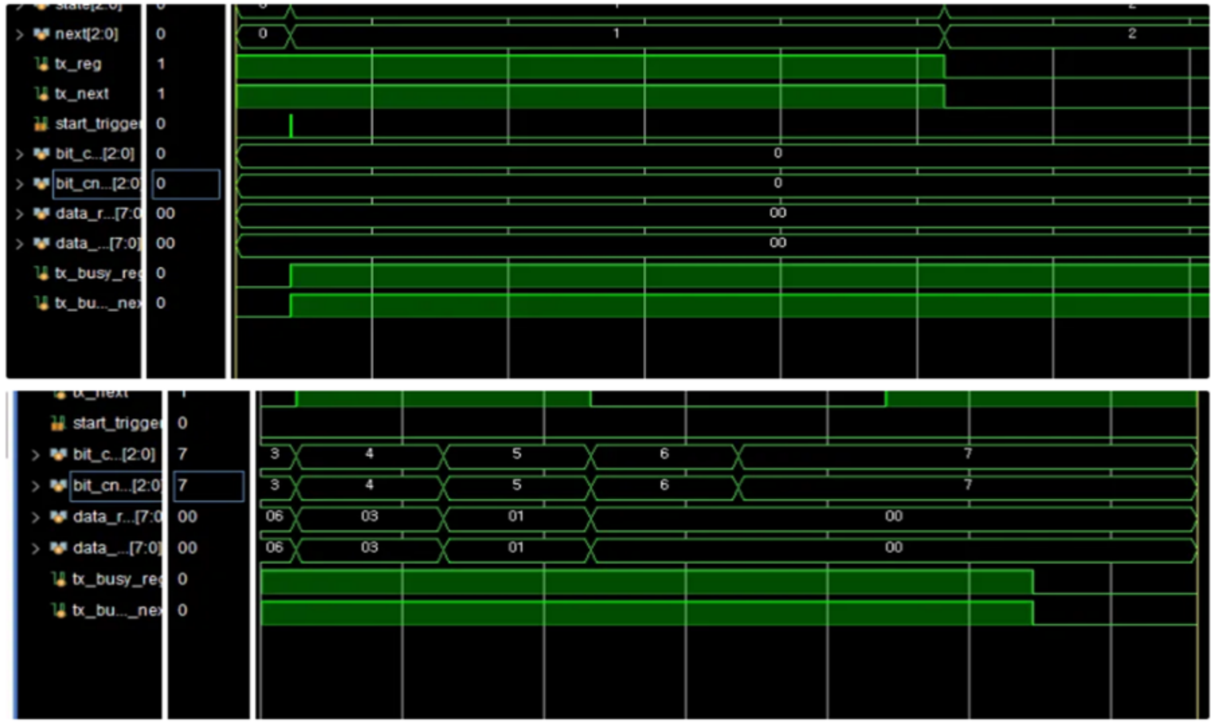
```
START: begin
    //output
    tx_next = 1'b0;
    if (b_tick) begin
        bit_cnt_next = 0;
        next = DATA;
    end
end

DATA : begin
    // output
    tx_next = tx_data[bit_cnt_reg];
    if (b_tick) begin
        if (bit_cnt_reg == 7) begin
            next = STOP;
        end else begin
            bit_cnt_next = bit_cnt_reg + 1;
        end
    end
end
```

output data 를 shift register로 바꿈

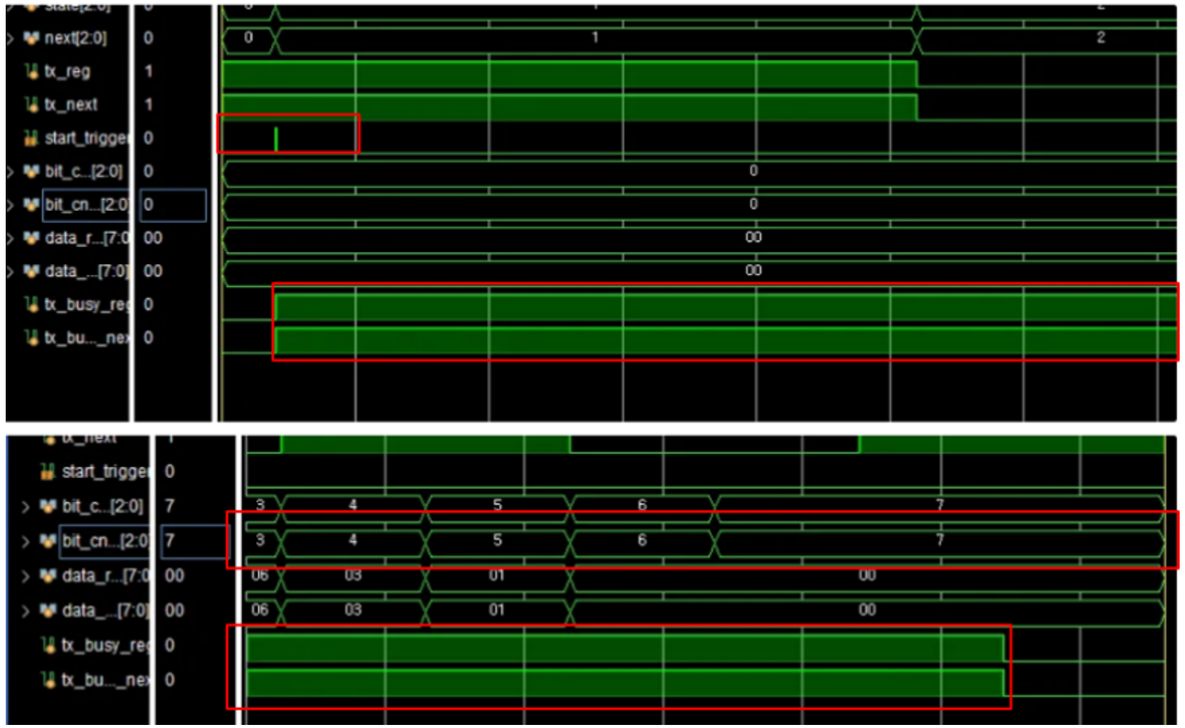
신호 전체를 하나로 보내서
tx_busy를 start에서stop까지 묶어서
한번에 보냄

03 Schematic, 시뮬레이션





04 시뮬레이션 결과

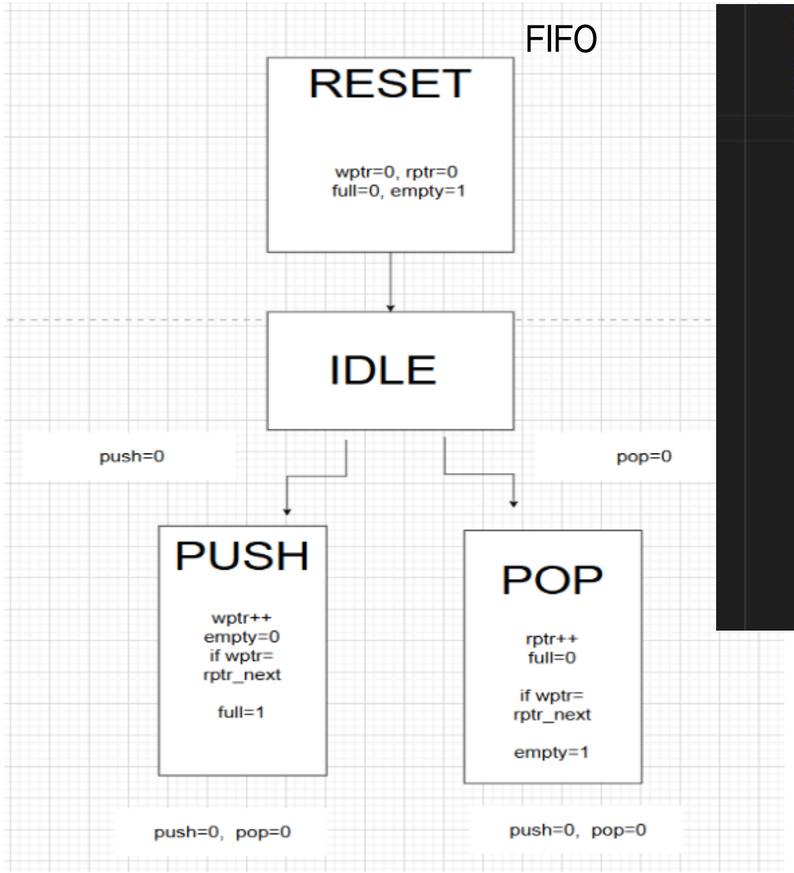


start_trigger가 짧게 1펄스 → 그 직후 tx_busy_reg가 High 유지

bit_cnt가 0→1→…→7까지 증가한 뒤 종료 → 8비트(8N1)의 데이터 전송

start_trigger와 bit_cnt 변화 타이밍과 잘 맞음 → FSM 타이밍 정상.

02 BLOCK Diagram

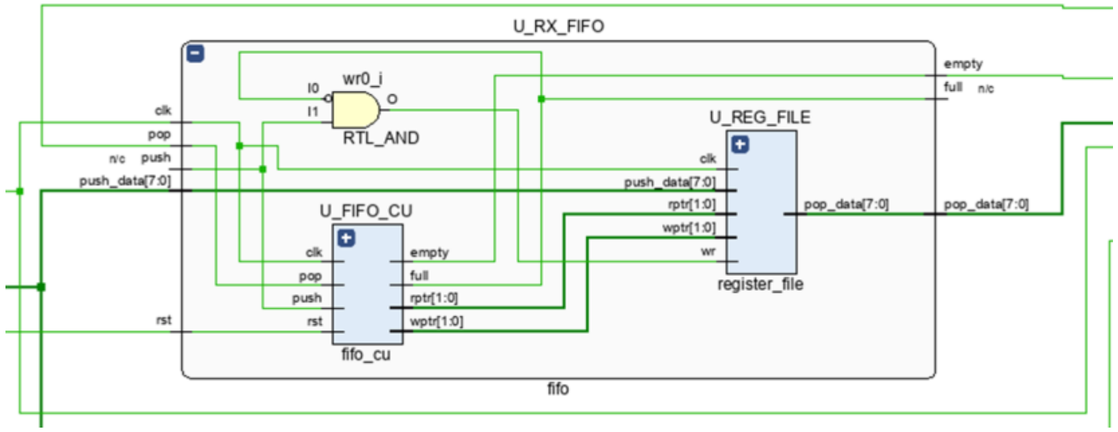



```
case ({
    push, pop
})
    2'b01: begin
        //pop
        full_next = 1'b0;
        if (!empty_reg) begin
            rptr_next = rptr_reg + 1;
            if (wptr_reg == rptr_next) begin
                empty_next = 1'b1;
            end
        end
    end
    2'b10: begin
        // push
        empty_next = 1'b0;
        if (!full_reg) begin
            wptr_next = wptr_reg + 1;
            if (wptr_next == rptr_reg) begin
                full_next = 1'b1;
            end
        end
    end
    2'b11: begin
        // push&pop
        if (empty_reg == 1) begin
            wptr_next  = wptr_reg + 1;
            empty_next = 1'b0;
        end else if (full_reg == 1'b1) begin
            rptr_next = rptr_reg + 1;
            full_next = 1'b0;
        end else begin

            wptr_next = wptr_reg + 1;
            rptr_next = rptr_reg + 1;
        end
    end
endcase
```

push, pop에 따라 동작 (PUSH / POP / PUSH&POP)결정되는 구조.

03 Schematic

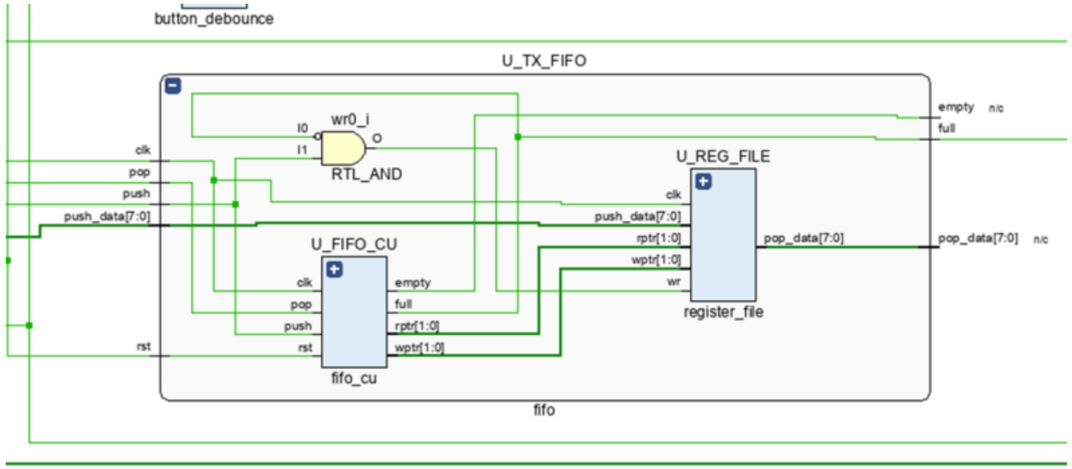
RX_FIFO 수신한 데이터(rx_data)를 저장해두는 버퍼



UART 수신기에서 들어온 데이터를 임시로
저장해두는 버퍼 역할
FIFO는 제어 유닛과 레지스터 파일로 구성되며,
제어 유닛은 push와 pop에 따라 write pointer와
read pointer를 제어하고,
레지스터 파일은 실제 데이터를 저장
UART에서 수신된 데이터는 push 시점마다 FIFO에
쌓이고,
상위 로직이 필요할 때 pop으로 꺼내 사용

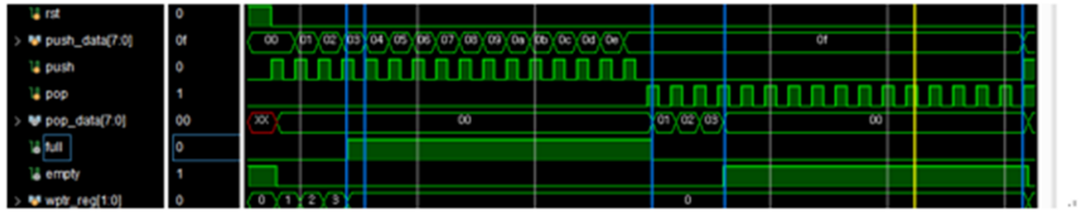
03 Schematic

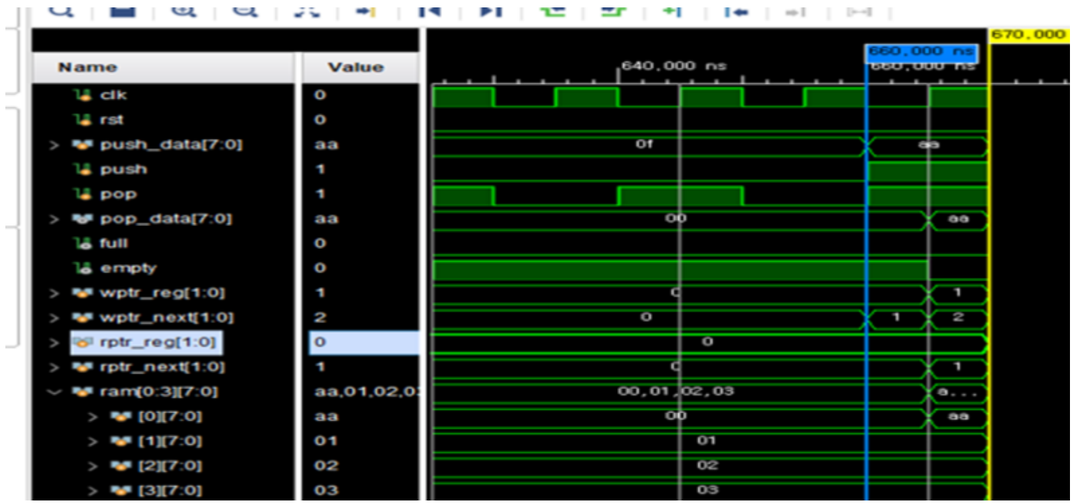
TX_FIFO 송신할 데이터를 저장해두는 버퍼



TX_FIFO는 상위 로직에서 보낸 송신 데이터를 임시 저장해 두었다가, UART 송신기가 준비되면 꺼내서 전송하는 버퍼
제어 유닛이 push/pop 동작을 관리하고, 레지스터 파일이 데이터를 저장
송신 데이터가 순서대로 쌓이고, 필요할 때 차례로 꺼내 전송할 수 있도록 보장하는 구조

04 시물레이션 결과





FIFO에 데이터가 들어갈 때 full 상태가 되는지 확인

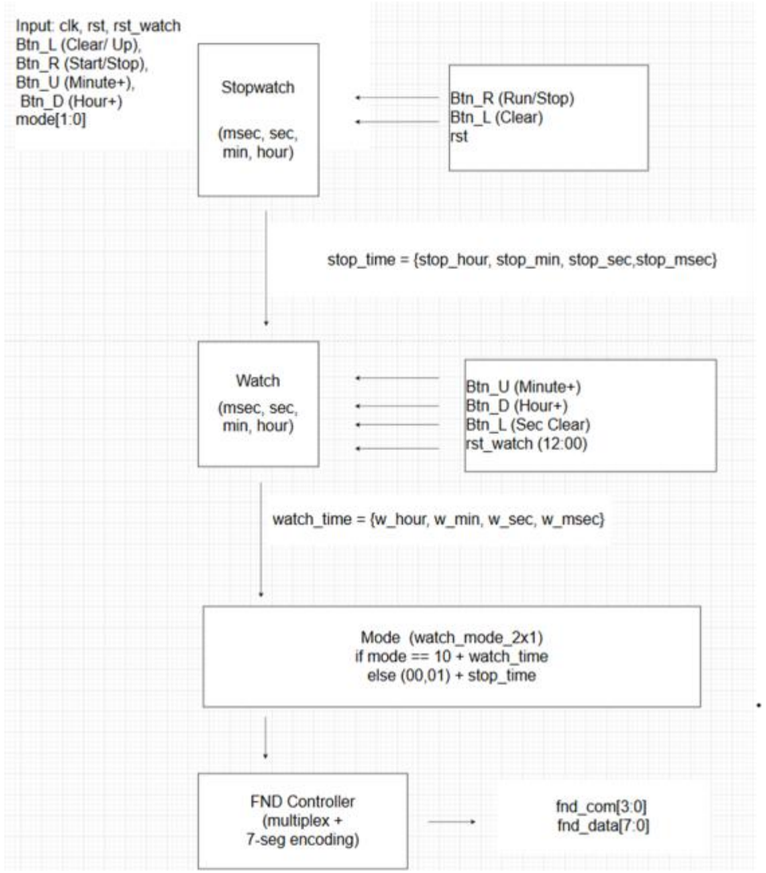
push 신호가 올라갈 때마다 데이터 하나씩 들어감
push_data가 0,1,2,3 들어갈 때까지는 full=0
pop이 비워졌을 때 empty=1로 뜬다

push와 Pop을 동시에 했을 때 FIFO가 유지되고 동작이 정상인지 확인

파형 중간에 push=1과 pop=1이 겹치는 순간 있음
wptr와 rptr 모두 +1 증가 → FIFO에 쌓인 데이터는 유지

02 BLOCK Diagram

STOPWATCH_WATCH

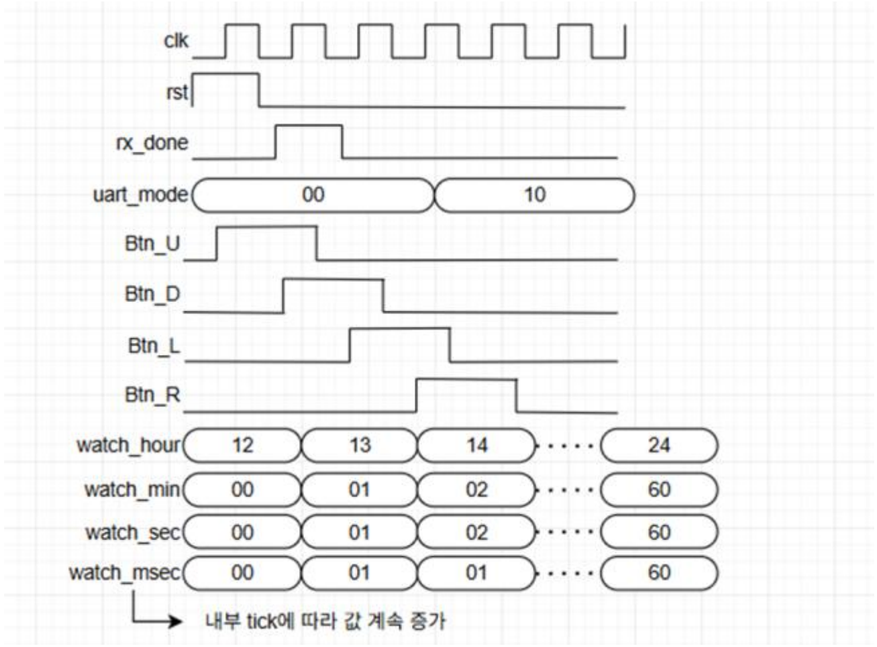


“Stopwatch 블록은 카운터 기반으로 시간 측정을 하고,
Watch 블록은 시·분을 수동으로 조정

Mode 블록에서 선택된 데이터가 FND Controller를 통해
최종적으로 7세그먼트에 출력되는 구조

02 BLOCK Diagram

STOPWATCH_WATCH

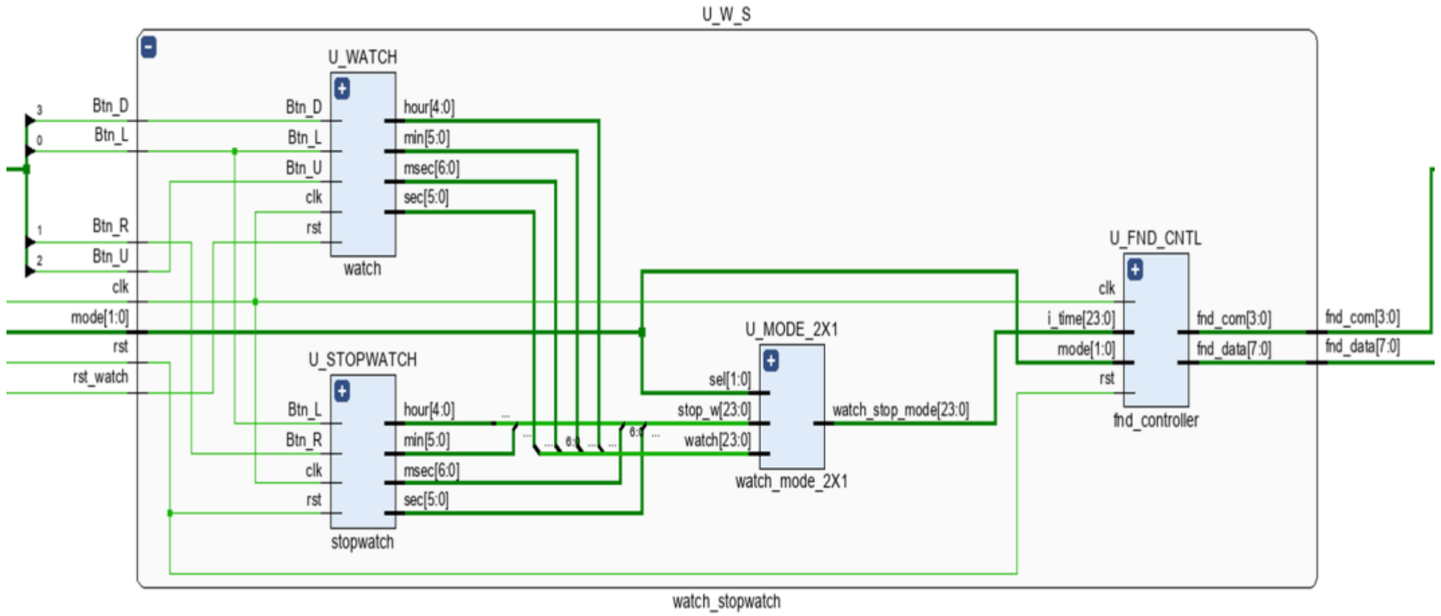


Stopwatch 모드 (00)
Btn_R 입력(시작/정지)에 따라 msec, sec, min 카운트가 동작.
Clear(Btn_L)를 누르면 모두 00으로 초기화.

Watch 모드 (10)
Btn_D → 시 증가, Btn_U → 분 증가.
rst_watch가 들어오면 12:00으로 초기화.
내부 tick에 따라 초와 밀리초가 계속 증가.

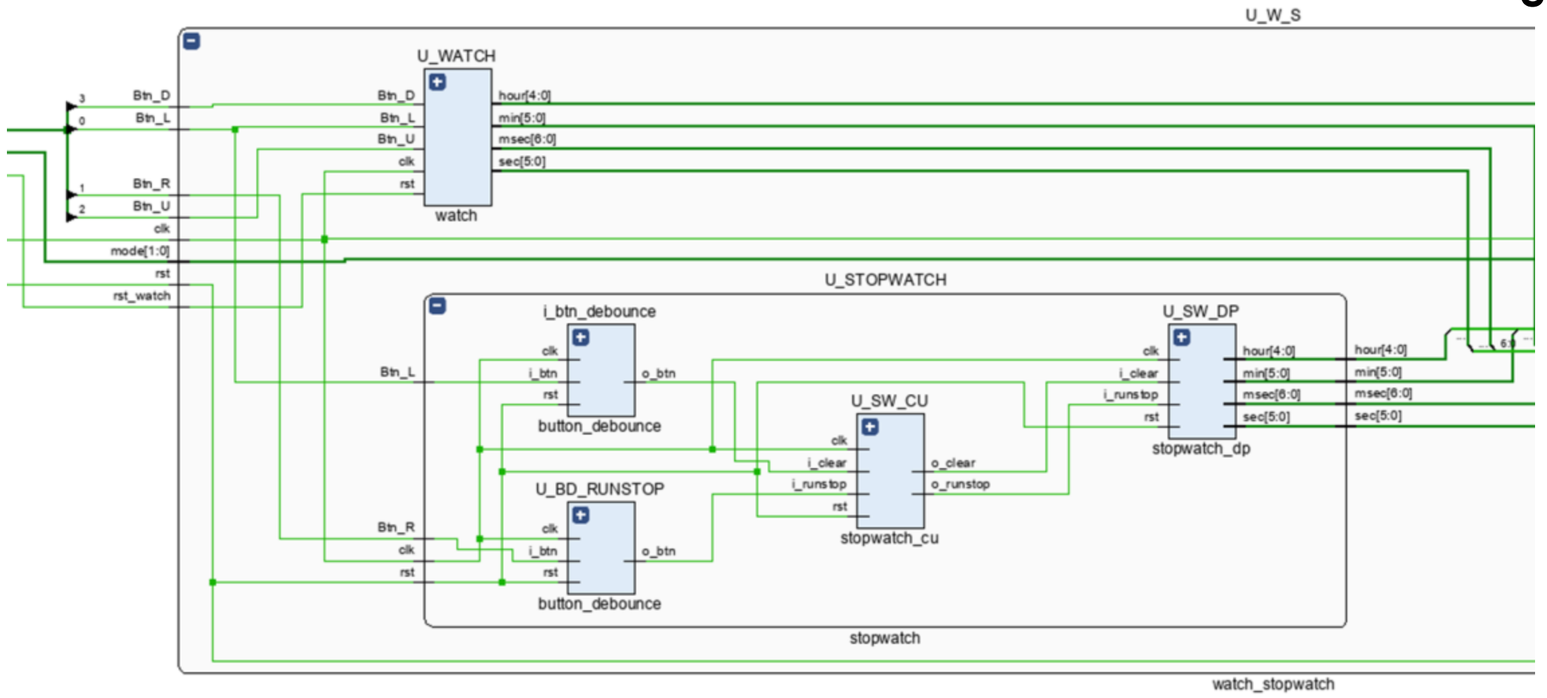
03 Schematic

stopwatch_watch

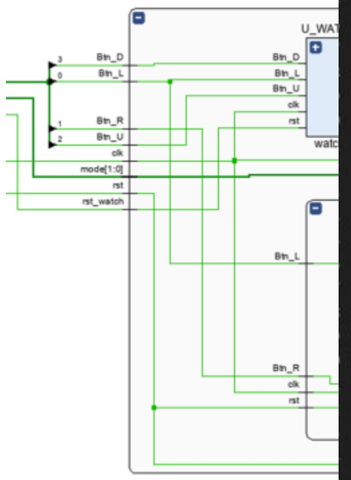


03 Schematic

stopwatch_top



03 Schematic




```
case (c_state)
    STOP: begin
        // STOP에서는 run=0, clear=0
        if (i_clear) begin
            n_state = CLEAR;
        end else if (i_runstop) begin
            n_state = RUN;
        end
    end

    RUN: begin
        runstop_next = 1'b1; // RUN상태에서는 run=1
        if (i_clear) begin
            n_state = CLEAR;  // 런 중 클리어 허용 (권장)
        end else if (i_runstop) begin
            n_state = STOP;
        end
    end

    CLEAR: begin
        // 1클럭 clear 펄스
        clear_next = 1'b1;
        // 클리어 후 즉시 STOP 복귀
        n_state = STOP;
    end

    default: begin
        n_state = STOP;
    end
endcase
end
```

stopwatch_top

기능: 스톱워치 동작 상태 제어 (RUN / STOP / CLEAR)

원리: FSM 기반 상태 전환

동작:

RUN → 카운터 증가

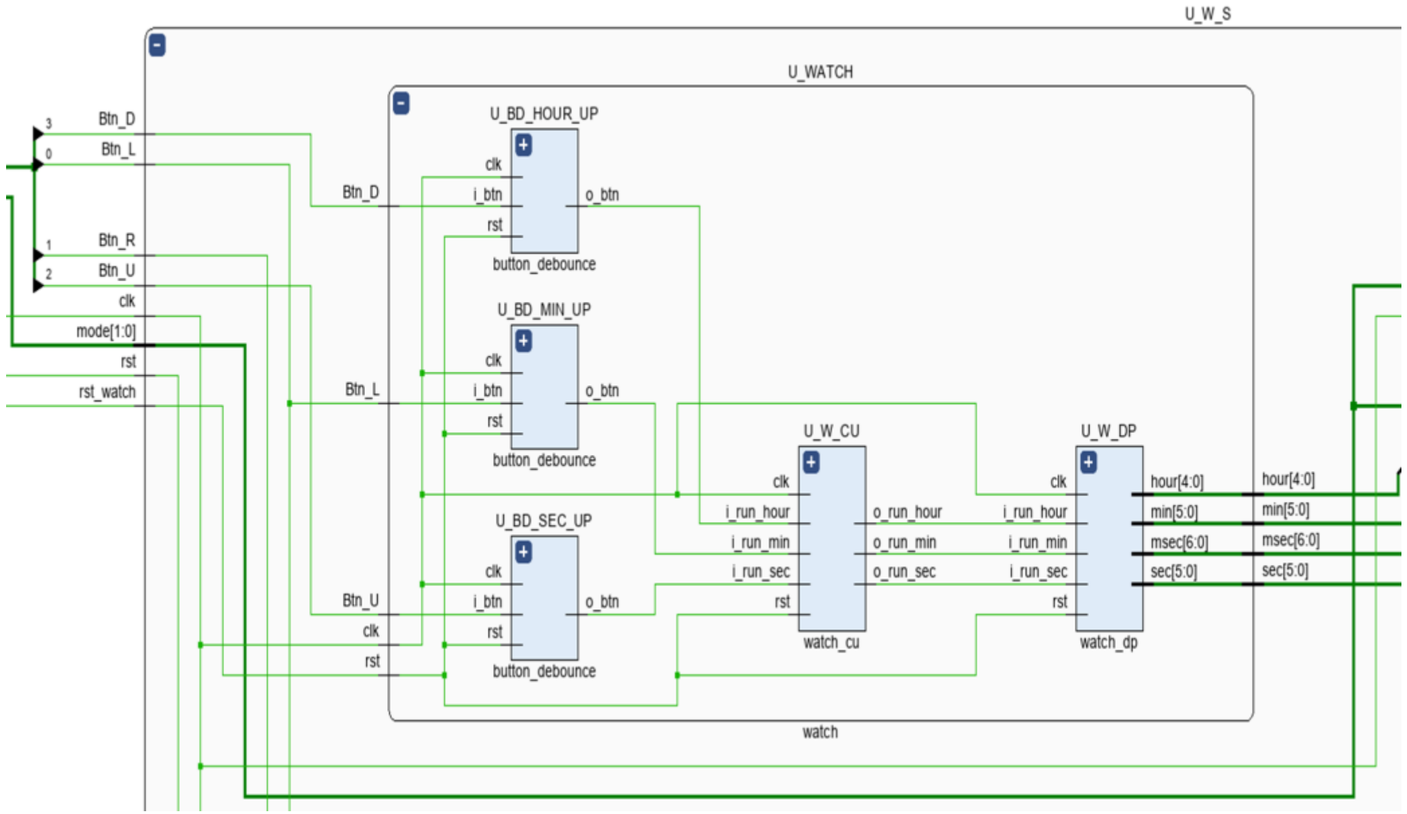
STOP → 카운터 유지

CLEAR → 카운터 리셋 후 STOP으로 복귀

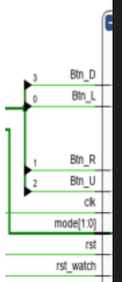
출력: o_runstop(동작 여부), o_clear(리셋 신호)

03 Schematic

watch_top



03 Schematic



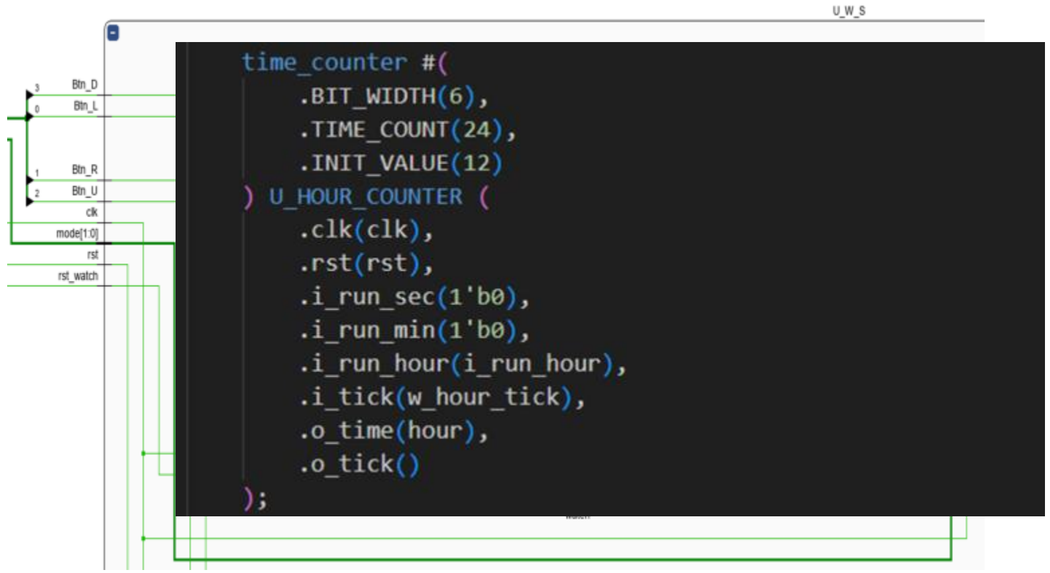

```
    case (s)
        // 대기 상태: 버튼 입력을 보고 해당 상태로 진입
        // 동시 입력 시 우선순위: hour > min > sec
        IDLE: begin
            if (i_run_hour) ns = HOUR_RUN;
            else if (i_run_min) ns = MIN_RUN;
            else if (i_run_sec) ns = SEC_RUN;
        end

        // 각 상태에서 1클럭 동안 해당 출력 펄스 내고 바로 IDLE로 복귀
        SEC_RUN: begin
            run_sec_next = 1'b1;
         ns   = IDLE;
        end

        MIN_RUN: begin
            run_min_next = 1'b1;
         ns   = IDLE;
        end

        HOUR_RUN: begin
            run_hour_next = 1'b1;
         ns    = IDLE;
        end

        default: begin
         ns = IDLE;
        end
    endcase
end
```

watch_top

기능: 버튼 입력으로 시/분 증가 제어
구성: 버튼 입력 → debounce → CU 제어 신호 생성 → DP에서 카운터 증가
동작:
Btn_L → 분 +1
Btn_D → 시 +1

rst_watch → 12:00 초기화
출력: o_run_hour, o_run_min, o_run_sec (증가 신호)

03 Schematic

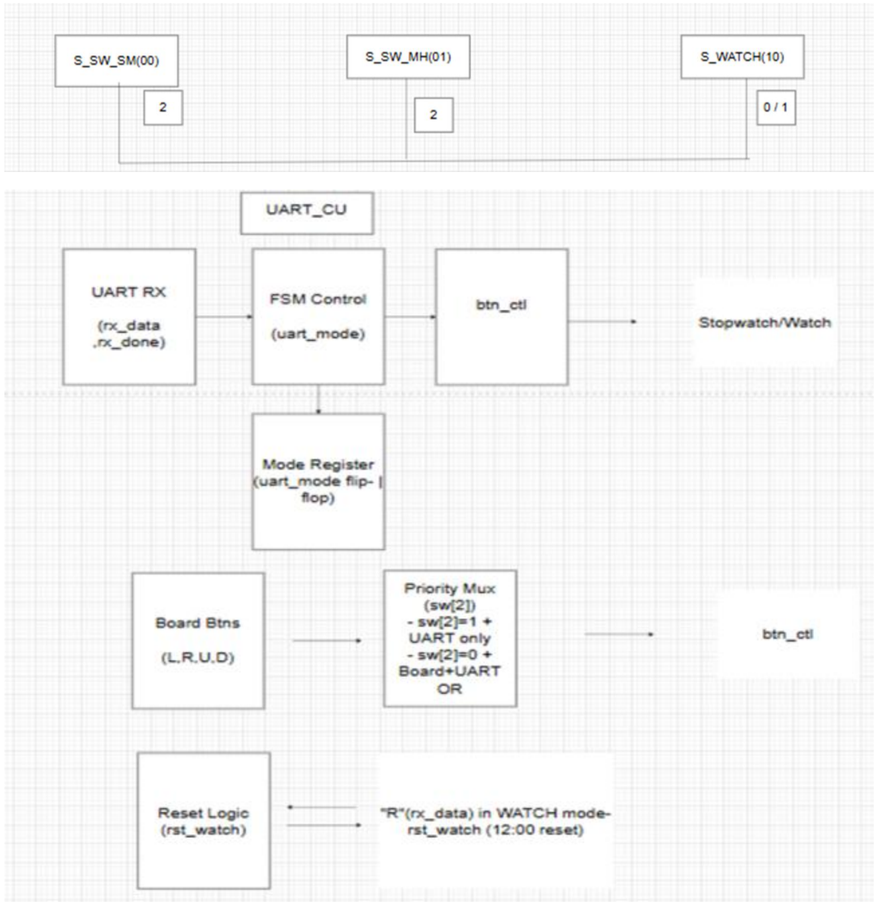


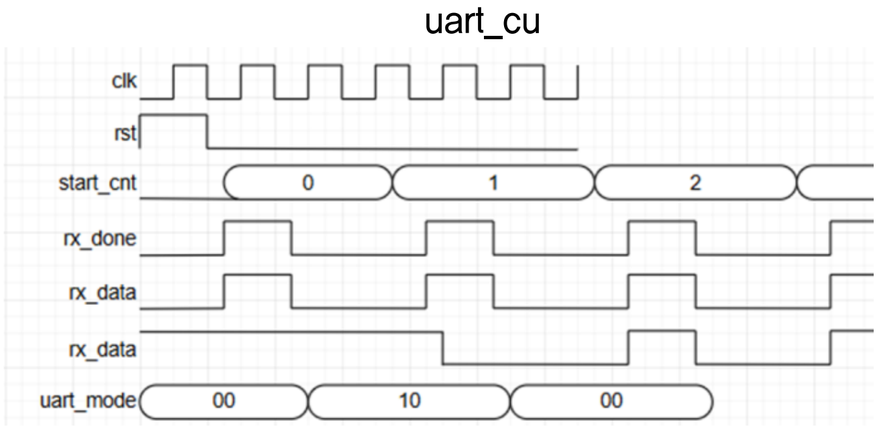
watch_top

기능: 버튼 입력으로 시/분 증가 제어
구성: 버튼 입력 → debounce → CU 제어 신호 생성 → DP에서 카운터 증가
동작:
Btn_L → 분 +1
Btn_D → 시 +1

rst_watch → 12:00 초기화
출력: o_run_hour, o_run_min, o_run_sec (증가 신호)

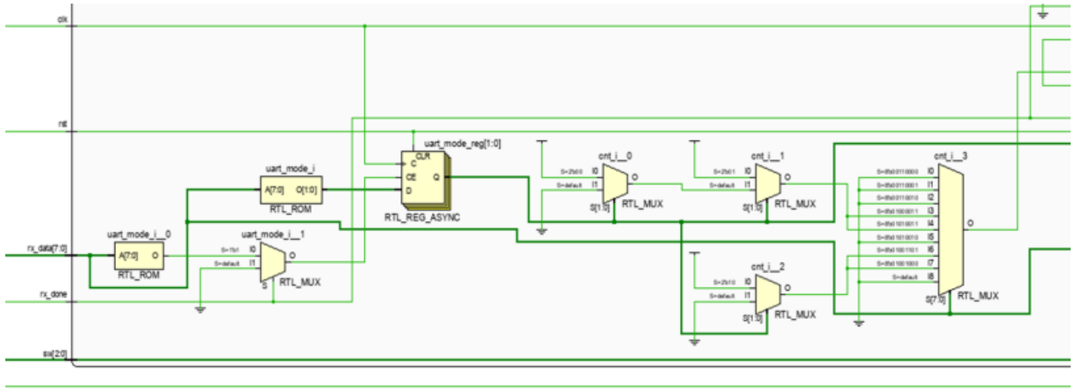
02 BLOCK Diagram





rx_done 펄스가 들어올 때마다 case문이 동작
→ rx_data에 맞춰 uart_mode 값이 바로 변경됨.
파형에서 "0" 수신 → uart_mode = 00,
다음 "1" 수신 → uart_mode = 10,
그다음 "2" 수신 → 다시 uart_mode = 00 로 전환되는 흐름을 예상
start_cnt는 "M", "H", "C", "S" 같은 제어 입력이 들어올 때 펄스 유지
시간만큼 증가했다가 자동으로 0으로 돌아가기 때문에, 이렇게
단계적으로 카운트하는 모양이 예상

03 Schematic




```
always @(posedge clk or posedge rst) begin
    if (rst) begin
        uart_mode    <= 2'b00;
        btn_ctl_uart <= 4'b0000;
        cnt          <= 0;
        rst_watch    <= 1'b0;
    end else begin
        rst_watch <= 1'b0; // 기본값

        if (rx_done) begin
            case (rx_data)
                // ---------------- 모드 전환 ----------------
                "0": uart_mode <= 2'b00; // Stopwatch sec/mc
                "1": uart_mode <= 2'b01; // Stopwatch min/hour
                "2": uart_mode <= 2'b10; // Watch
```

uart_cu

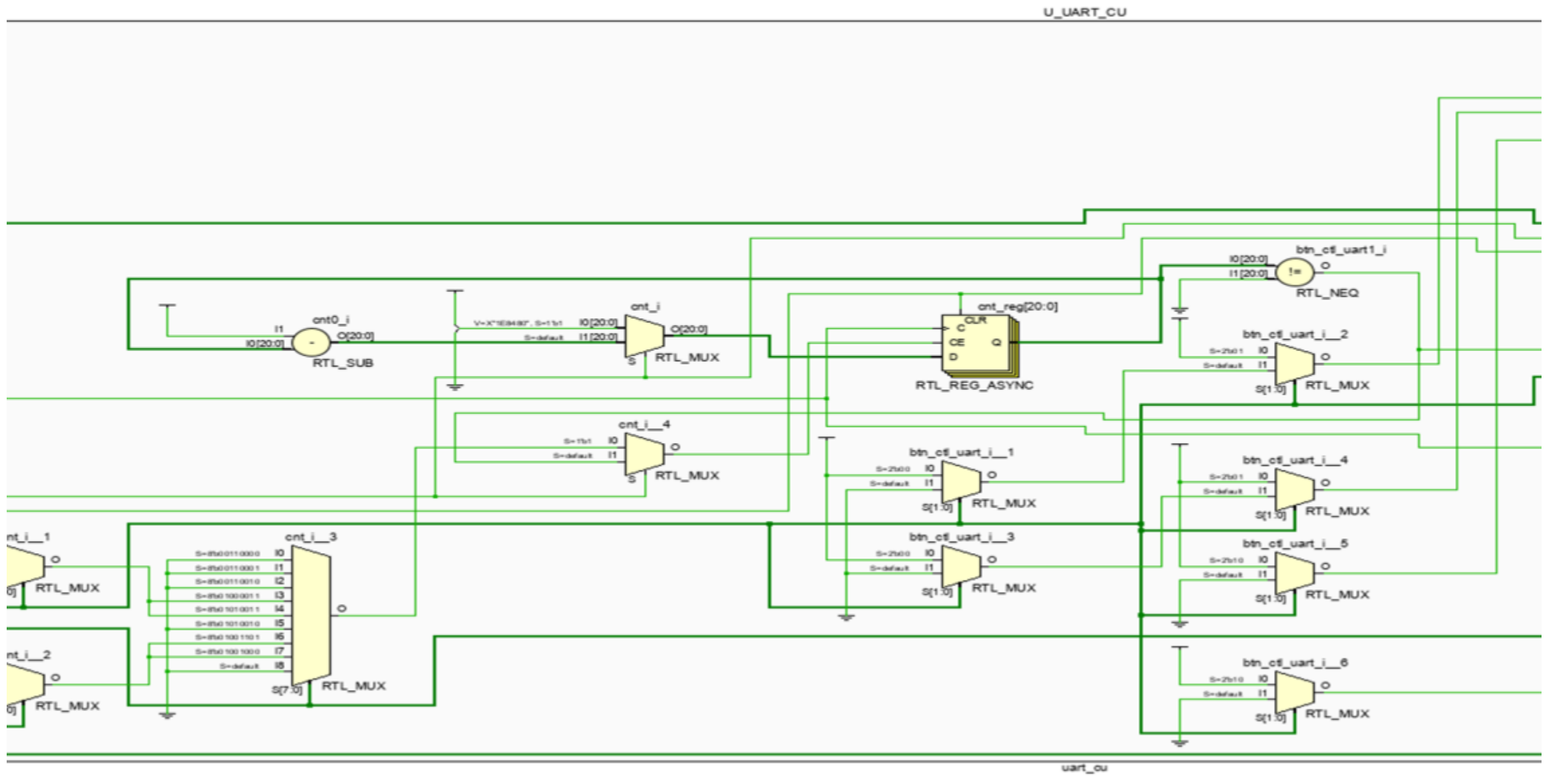
UART 수신 데이터("0","1","2") → 모드 결정

uart_mode (2비트)

00 = Stopwatch (sec/msec)
01 = Stopwatch (min/hour)

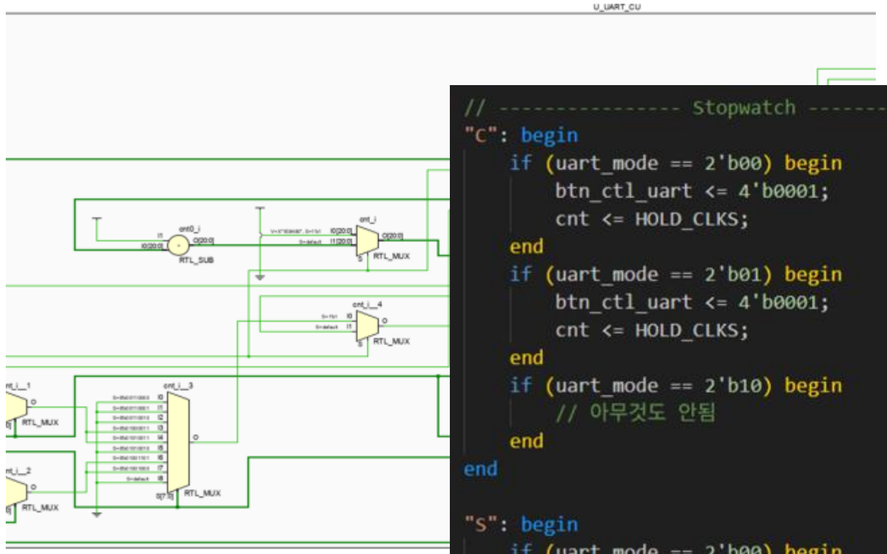
10 = Watch
case문 기반으로 데이터 해석 → 모드 전환

03 Schematic



uart_cu

03 Schematic



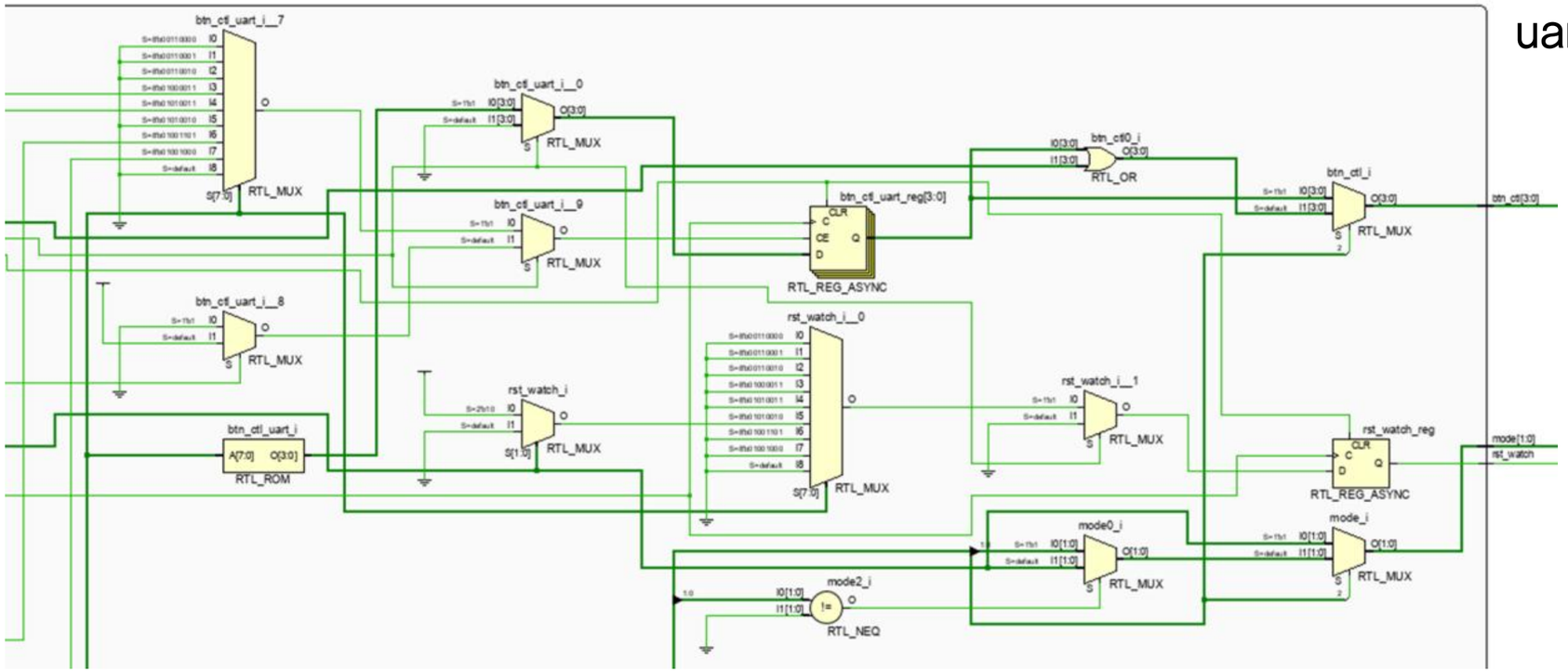

```
// ---------------- Stopwatch ----------------
"C": begin
    if (uart_mode == 2'b00) begin       // Stopwatch sec/mc
        btn_ctl_uart <= 4'b0001;        // Clear
        cnt <= HOLD_CLKS;
    end
    if (uart_mode == 2'b01) begin       // Stopwatch min/hour
        btn_ctl_uart <= 4'b0001;        // Clear
        cnt <= HOLD_CLKS;
    end
    if (uart_mode == 2'b10) begin       // Watch 모드일 때는 무시
        // 아무것도 안됨
    end
end

"S": begin
    if (uart_mode == 2'b00) begin       // Stopwatch sec/mc
        btn_ctl_uart <= 4'b0010;        // Start/Stop
        cnt <= HOLD_CLKS;
    end
    if (uart_mode == 2'b01) begin       // Stopwatch min/hour
        btn_ctl_uart <= 4'b0010;        // Start/Stop
        cnt <= HOLD_CLKS;
    end
    if (uart_mode == 2'b10) begin       // Watch 모드일 때는 무시
        // 아무것도 안됨
    end
end
```

uart_cu

UART로 들어온 "C", "S", "M", "H" 문자에 대해 버튼 신호(btn_ctl_uart)를 잠깐(펄스) High로 만들어 줌

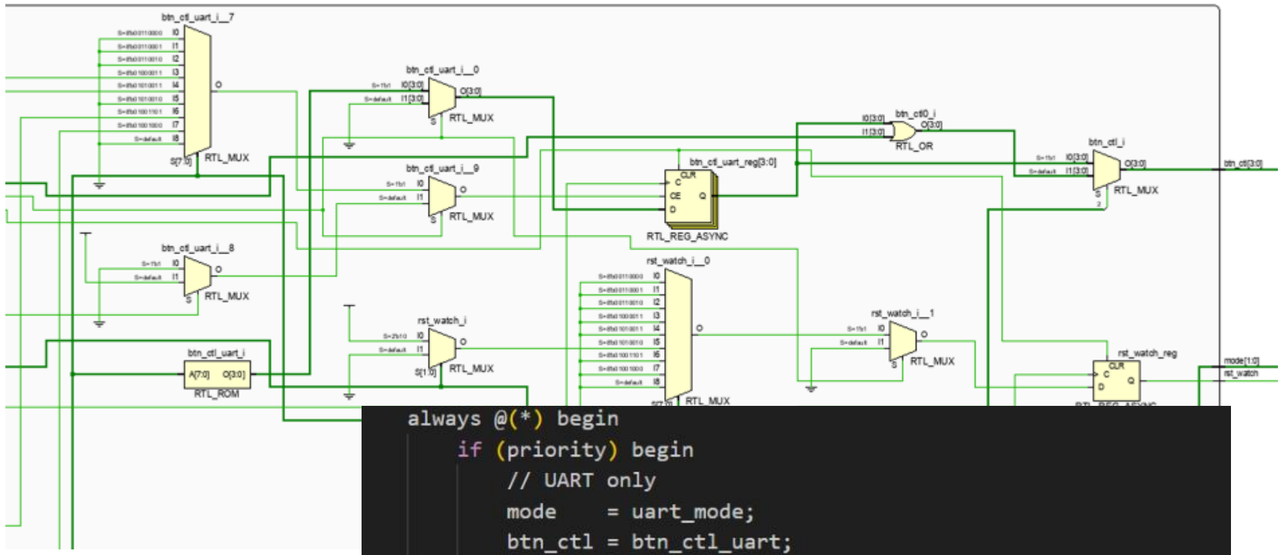
동시에 카운터(cnt)를 두어서 일정 시간(HOLD_CLKS) 유지 후 다시 0으로 리셋

03 Schematic



uart_cu

03 Schematic




```
always @(*) begin
    if (priority) begin
        // UART only
        mode    = uart_mode;
        btn_ctl = btn_ctl_uart;
    end else begin
        // 보드 + UART (debounce 적용 버튼 사용)
        mode    = (board_mode != 2'b00) ? board_mode : uart_mode;
        btn_ctl = btn_ctl_uart | {Btn_D, Btn_U, Btn_R, Btn_L};
    end
end
```

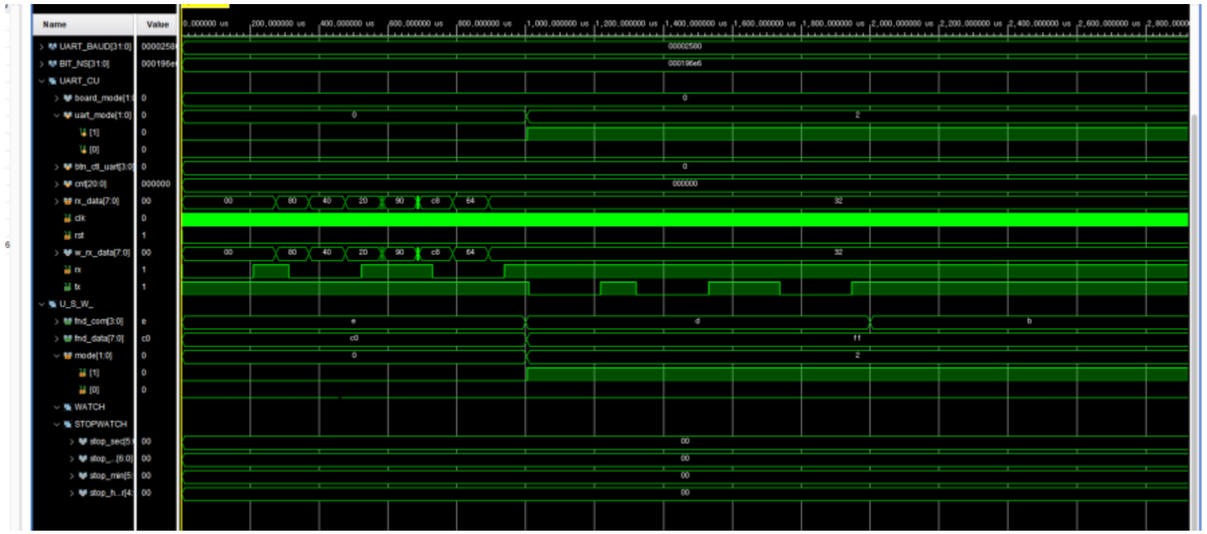
uart_cu

btn_ctl_uart와 보드 버튼(Btn_L/R/U/D)을
OR/우선순위로 합쳐서 최종 btn_ctl을 생성

sw[2] (priority)에 따라 UART 전용 제어만 쓰거나,
보드 버튼과 OR 해서 같이 동작

"R" 문자 수신 시 Watch 리셋(rst_watch) 신호 1펄스
발생 → watch 블록 12:00 초기화

04 시뮬레이션 결과

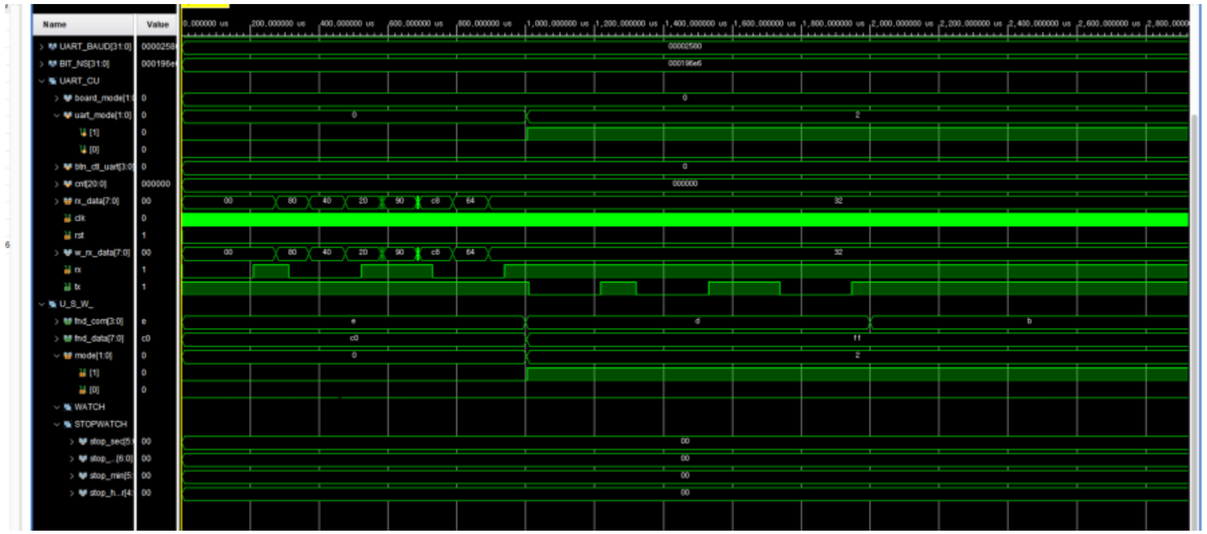


Stopwatch 동작

stop_sec, stop_min, stop_hour :
파형에서 계속 00 유지되고 있음 → 즉, Stopwatch 카운트가 증가하지 않음.

이유: Run/Stop 신호가 들어오지 않았거나, Clear 신호 후 Run이 눌리지 않아서 정지 상태(STOP 상태)에 머뭄

04 시뮬레이션 결과



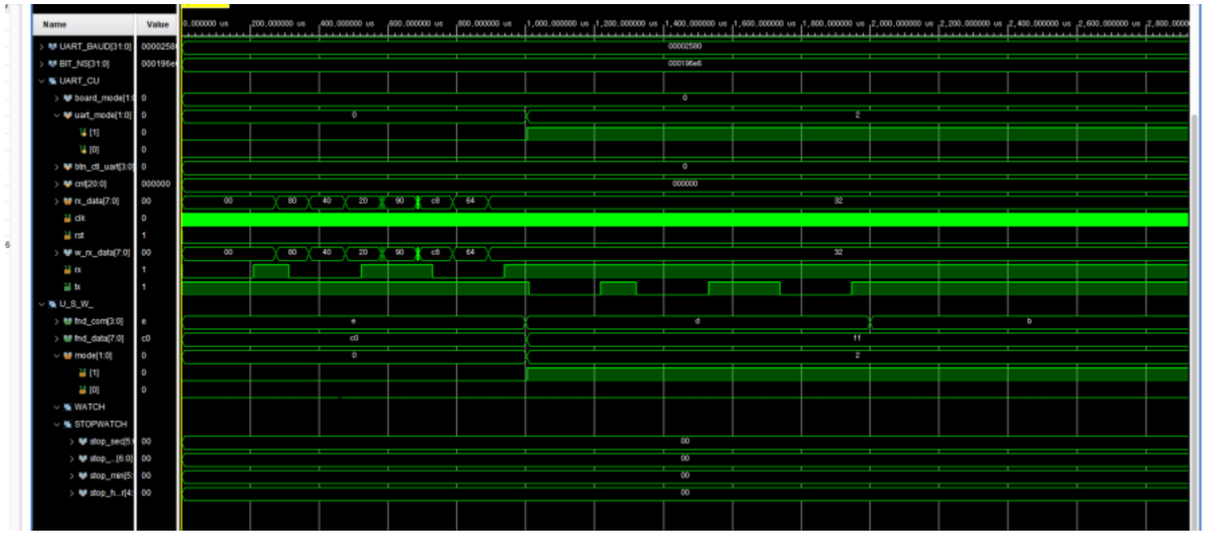
Watch 동작

mode가 2로 바뀌는 순간부터 Watch 모드가 활성화

fnd_data : 출력값이 바뀌는 것을 보면 FND에 표시되는 값이 UART 명령어(M, H)에 따라 업데이트되는 것을 확인

btn_ctl_uart에서 Min/Hour Up이 올라간 타이밍에서 fnd_data 값이 변함 → Watch 시간 데이터(w_hour, w_min, w_sec)가 갱신됨.

04 시뮬레이션 결과



UART_CU 동작
rx_data : 0x80, 0x40, 0x20, 0x90, 0xC8, 0x64 값들이 순서대로 들어온 것을 확인 가능. (즉, PC → FPGA로 UART 데이터가 전송됨)

uart_mode :

초반에는 0 (Stopwatch 모드)

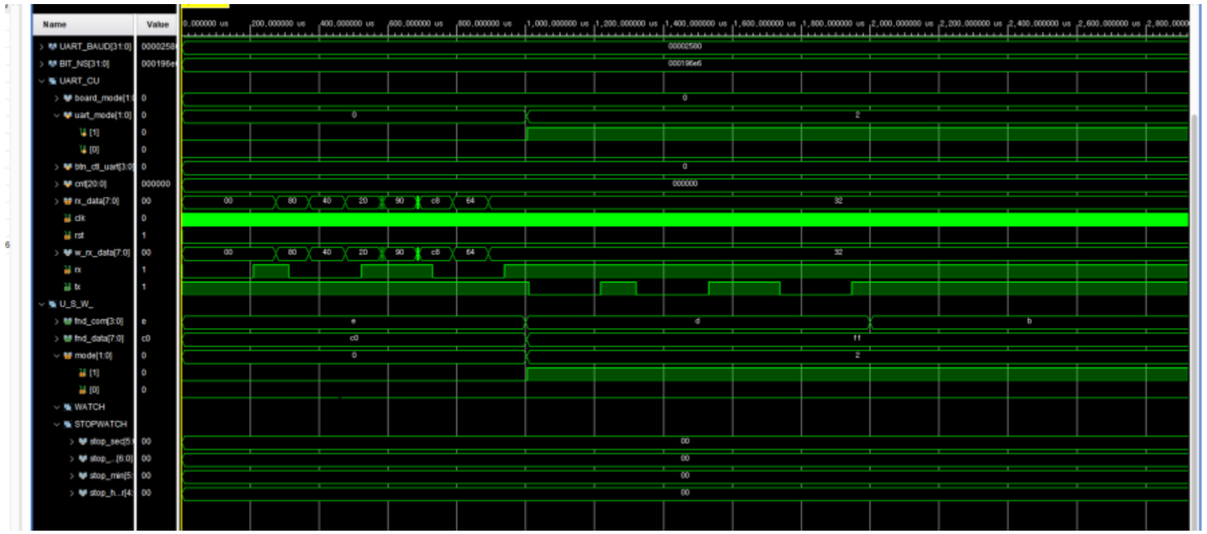
1.4ms 근처에서 2로 바뀜 (Watch 모드 전환)

btn_ctl_uart : 특정 데이터(M, H, S, C)에 따라 버튼 제어 신호가 올라갔다가 내려옴

"M" → btn_ctl_uart[2] = 1 (Min Up 동작)

"H" → btn_ctl_uart[3] = 1 (Hour Up 동작)

04 시뮬레이션 결과



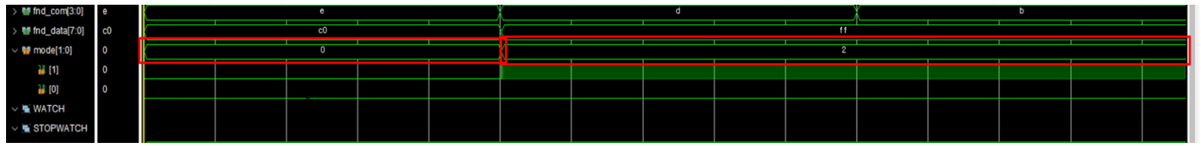
FND 출력 (fnd_com, fnd_data)

fnd_com = e, fnd_data = c0, d, b, 7, ff … 값들이 주기적으로 바뀜

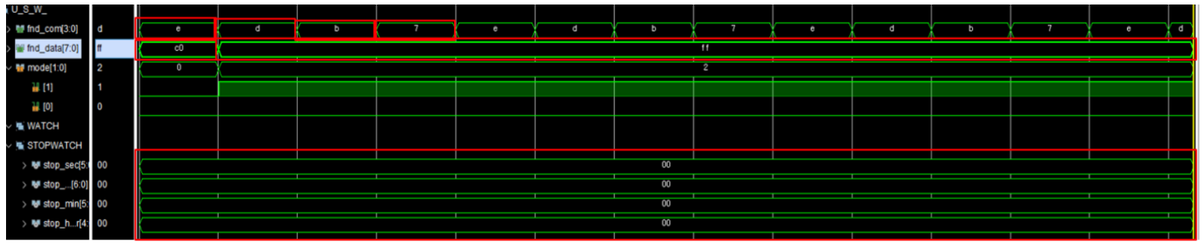
이는 다중화된 7-Segment 표시를 위한 값

UART 명령에 따라 Watch 시간이 바뀌면, 해당 값이 FND에 실시간으로 반영됨.

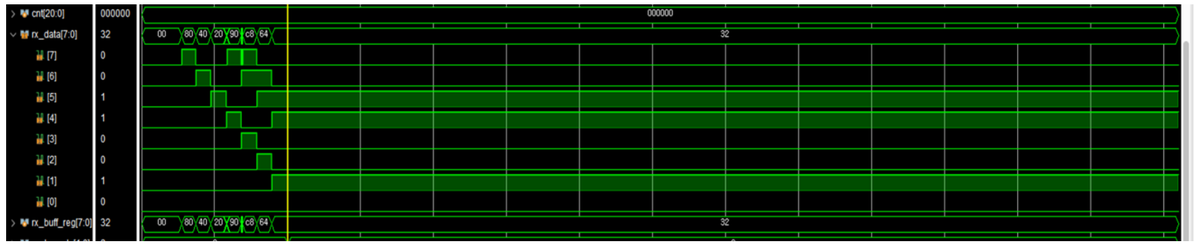
04 시뮬레이션 결과



uart_cu/uart_mode가 0 → 2로 전환

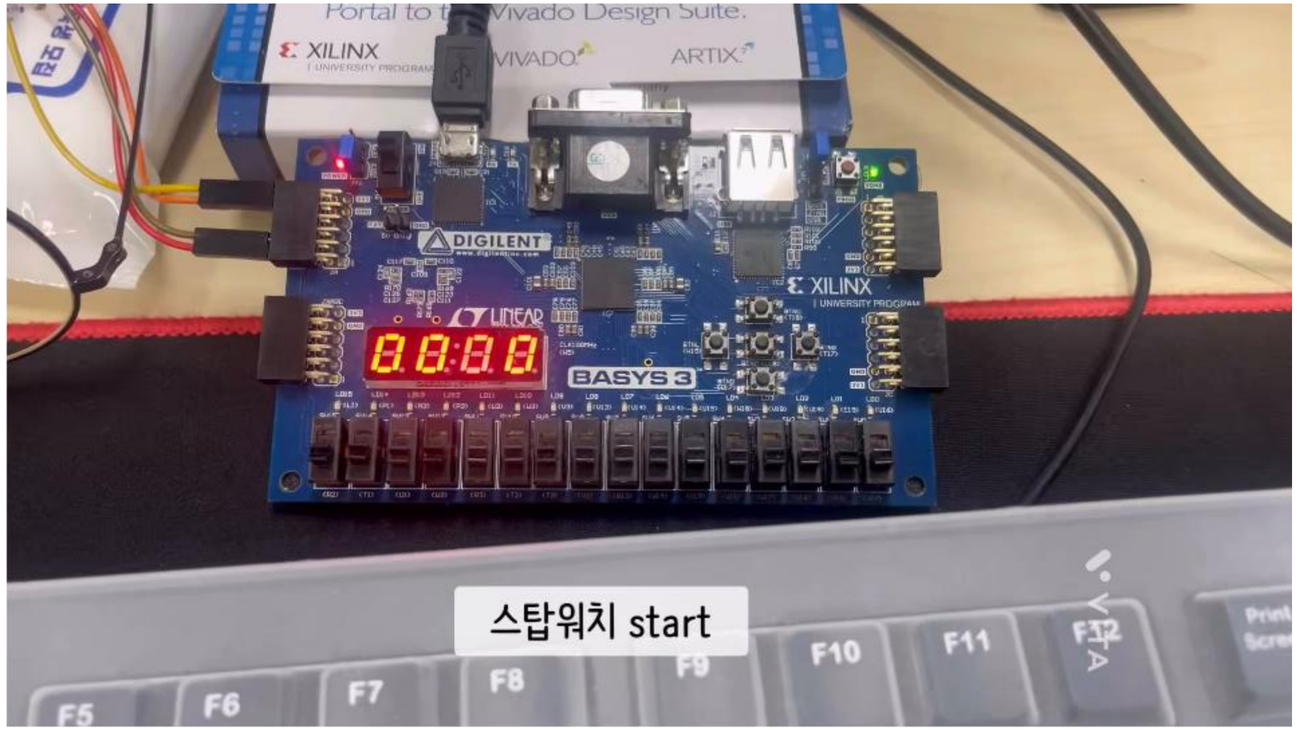


fnd_com 신호가 계속 d → e → b → 7 … 처럼 돌아가 FND 자리 선택 스톱워치는 꺼져 있고, 시계(Watch)만 켜져서 FND에 시간만 표시되는 상태
fnd_data가 c0, ff, e, d … 같은 값으로 바뀌면서 자리별 숫자



각 바이트가 uart_rx FSM을 통해 복원되고(rx_data), 그 값이 rx_fifo로 push 되며 순서대로 저장됨

05 동작영상



06 트러블슈팅&고찰(버튼/모드 분기 난잡 · 오동작)

Min Up 신호가 너무 짧게 펄스로 들어가서 watch_top 내부 카운터가 못 잡거나 debounce 처리와 충돌 → 신호가 무시되는 현상이 발생

```
//--------------------------------------
// 최종 모드/버튼 결정
//--------------------------------------
always @(*) begin
    if (priority) begin
        // UART only
        mode    = uart_mode;
        btn_ctl = btn_ctl_uart;
    end else begin
        // 보드 + UART (debounce 적용 버튼 사용)
        mode    = (board_mode != 2'b00) ? board_mode : uart_mode;
        btn_ctl = btn_ctl_uart | {Btn_D, Btn_U, Btn_R, Btn_L};
    end
end
```