

# UART/FIFO 발표

이승후

# 목차

1. 프로젝트의 목표
2. 프로젝트에 사용한 기능 설명
3. Schematic & Block diagram
4. Simulation
5. 동작 영상
6. Troubleshooting

# 프로젝트의 목표

이 프로젝트는 FPGA 보드인 Basys3를 활용하여, UART 통신과 FIFO 구조를 기반으로 PC와 실시간으로 상호작용하는 Stopwatch/Watch를 제어하는 것을 목표

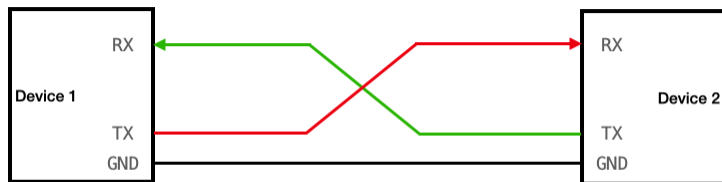
# 프로젝트에 사용한 기능 설명



**Basys3**는 미국 Digilent에서 제작한 FPGA 학습용 보드로, Xilinx Artix-7 FPGA를 탑재하고 있다.

대학생, 연구자, 개발자가 디지털 논리 회로, Verilog/VHDL 하드웨어 설계, 임베디드 시스템 개발을 학습하는 데 널리 사용된다.

# 프로젝트에 사용한 기능 설명



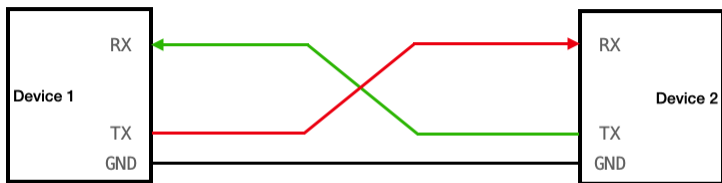
## UART (Universal Asynchronous Receiver/Transmitter)

비동기식 직렬 통신 방식 중 하나로, 데이터를 한 비트씩 순서대로 전송하는 방식.

송신기와 수신기가 별도의 클럭 신호를 공유하지 않는

대신, **Start bit / Data bits / Parity bit / Stop bit**에 동기화

# 프로젝트에 사용한 기능 설명



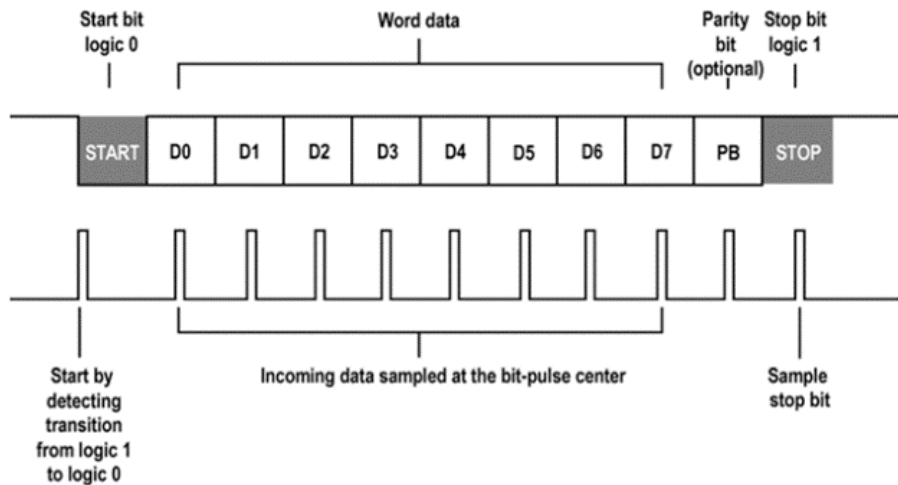
uart통신의 속도 = Baud Rate란?

직렬 통신에서 초당 신호 변화 횟수를 의미하며  
Baud Rate = 전송 속도(bps)로 생각할수있다

이번 프로젝트에서는 Baud Rate = 9600bps으로  
설정

9600bps는 초당 9600비트를 내보내며 이는 1비  
트를 보내는데  $1/9600 == 104166\text{ns}$ 가 된다

# 프로젝트에 사용한 기능 설명

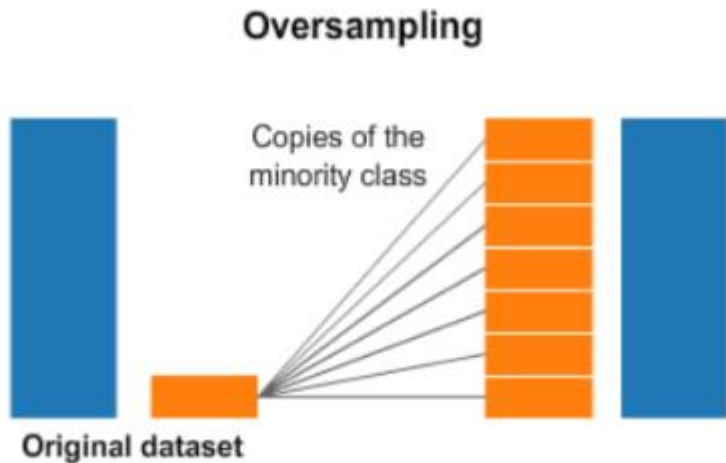


## Parity bit란?

UART 같은 비동기 직렬 통신에서 데이터가 손상되었는지 간단히 확인하기 위해 붙이는 보조 비트.

데이터 비트(예: 8비트)를 전송할 때, 그 뒤에 1비트를 추가해서 전체 비트의 "1의 개수"가 짝수 또는 홀수가 되도록 만든다.

# 프로젝트에 사용한 기능 설명



## Over Sampling이란?

오버샘플링은 신호를 필요한 샘플링 속도보다 훨씬 높은 속도로 샘플링하는 기술

주로 아날로그 → 디지털 변환(ADC), UART 통신, 센서 데이터 처리 등에서 사용됩니다.

UART 같은 디지털 통신에서는 데이터 비트의 중앙을 정확히 읽기 위해 사용

# 프로젝트에 사용한 기능 설명



shutterstock.com · 2494722845

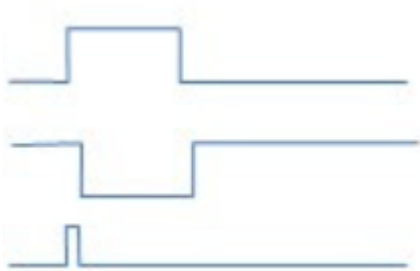
**FIFO (First In, First Out)**는 먼저 들어온 데이터가 먼저 나가는 방식을 의미하는 데이터 처리 규칙입니다

## FIFO의 특징

**순서 보장** - 입력된 순서대로 출력됨 → 데이터 무결성 보장

**버퍼(Buffer) 역할** - 데이터를 임시 저장하여 송수신 장치 간 속도 차이를 완화 (예: UART, 네트워크, 스트리밍 데이터)

# 프로젝트에 사용한 기능 설명



## 엣지 디텍트(Edge Detection)란?

엣지 디텍트는 디지털 신호에서 변화 순간(엣지)을 감지하는 방식 즉, 신호가  $0 \rightarrow 1$ 로 바뀌거나  $1 \rightarrow 0$ 으로 바뀌는 순간을 포착

Rising edge (상승 엣지) :  $0 \rightarrow 1$  전환 순간

Falling edge (하강 엣지) :  $1 \rightarrow 0$  전환 순간

# 프로젝트에 사용한 기능 설명



Stopwatch/Watch에 포함되어 있는 기능

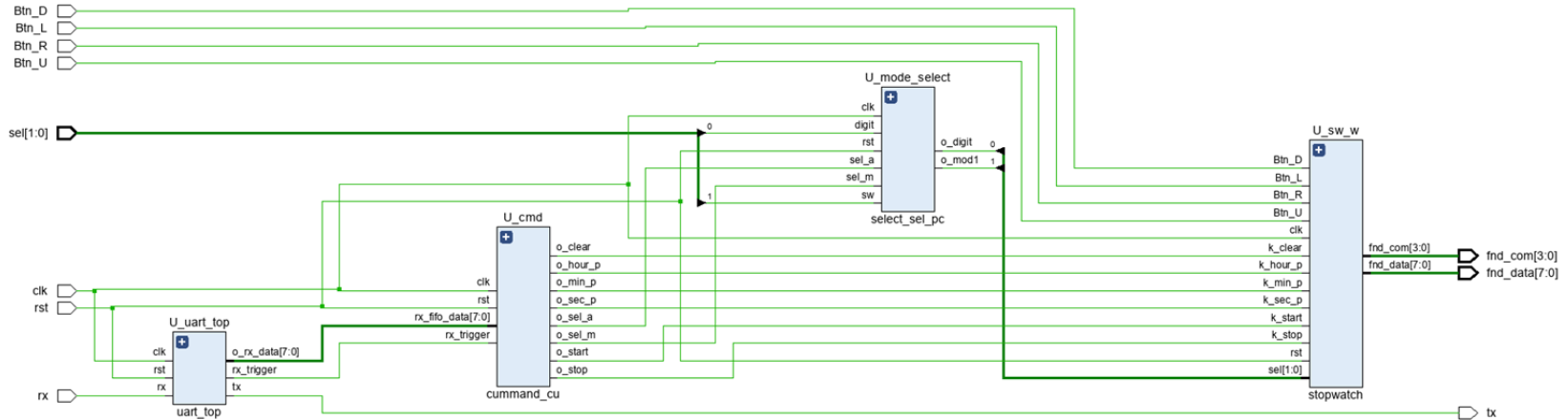
Stopwatch - start / stop / clear

Watch - 시 추가 / 분 추가 / 초 추가

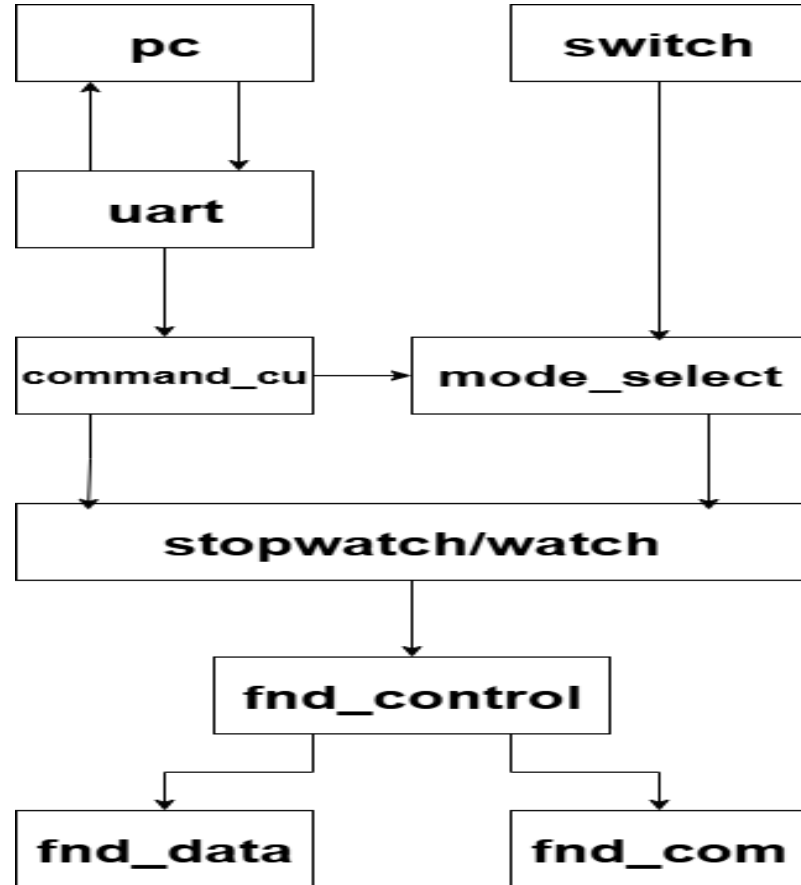
sw[0] - sec&msec/hour&min

sw[1] - Stopwatch/Watch

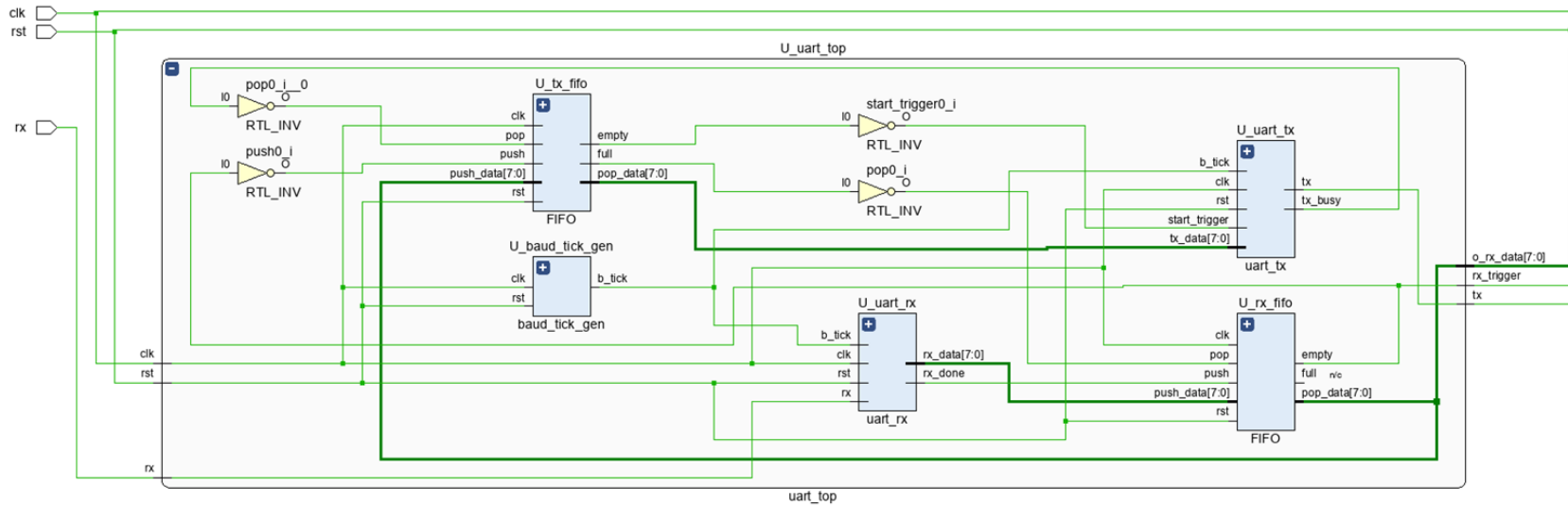
# Schematic



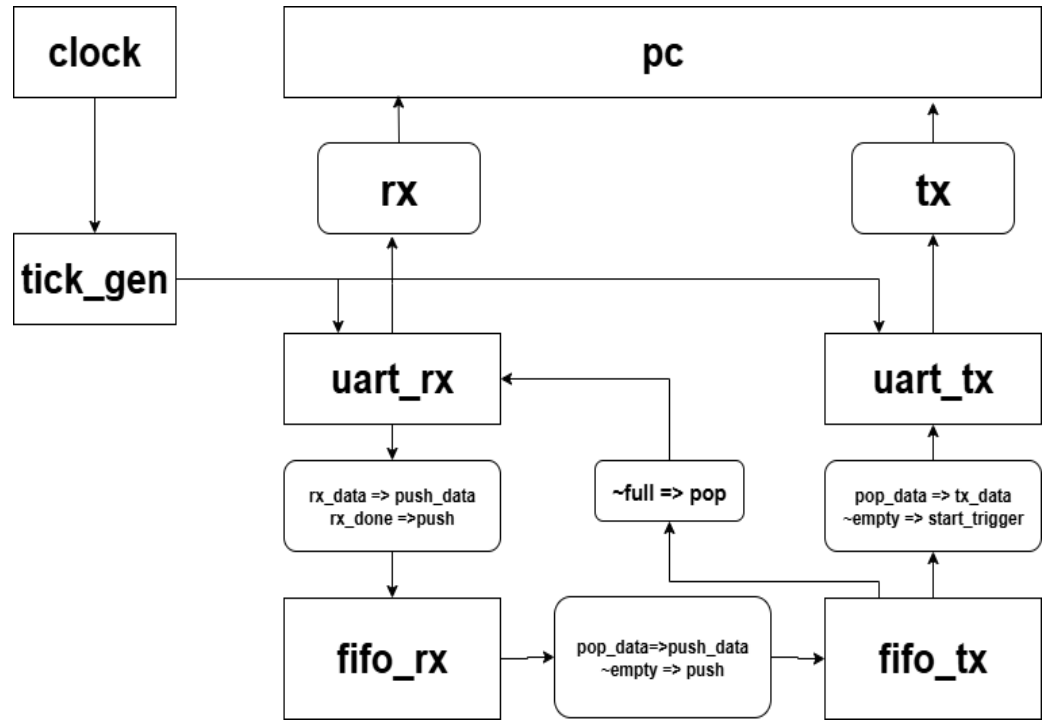
# Block diagram



# Schematic



## Block diagram

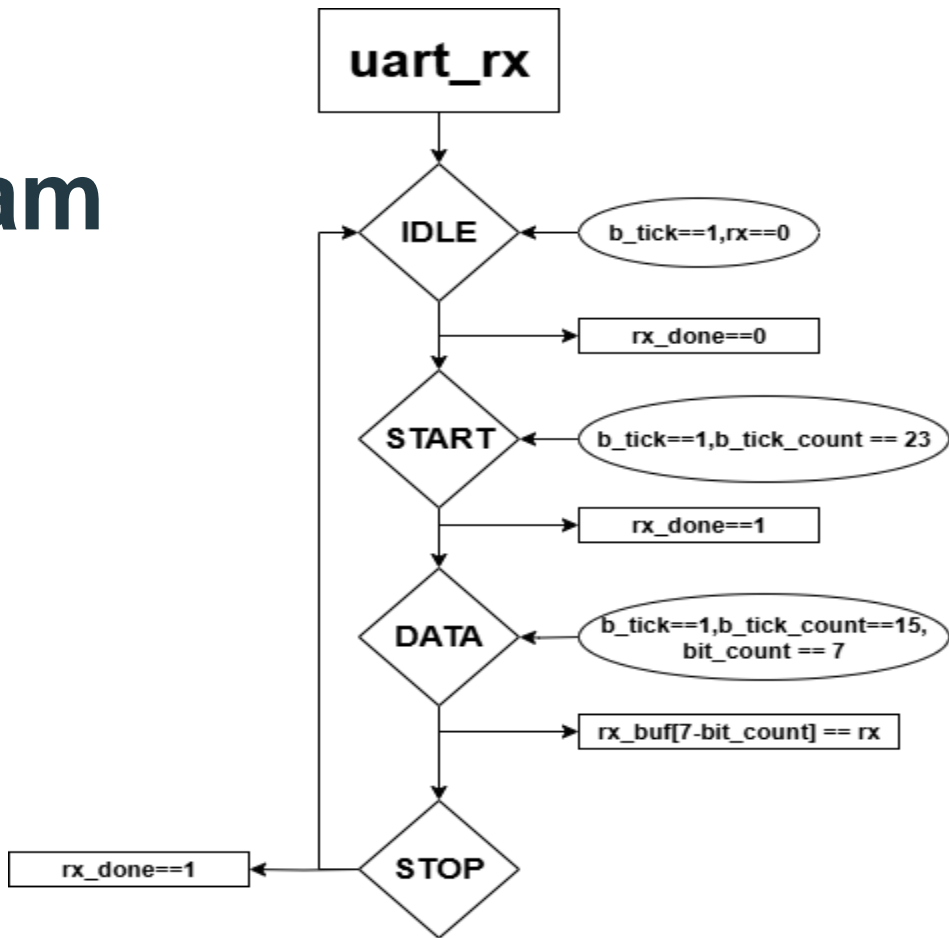


# Block diagram

```

always @(*) begin
    next = state;
    b_tick_count_next = b_tick_count_reg;
    bit_count_next = bit_count_reg;
    rx_done_next = rx_done_reg;
    rx_buf_next = rx_buf_reg;
    case (state)
    IDLE: begin
        rx_done_next = 1'b0;
        if (b_tick) begin
            if (rx == 1'b0) begin
                b_tick_count_next = 0;
                bit_count_next = 0;
                next = START;
            end
        end
    end
    START: begin
        if (b_tick) begin
            if (b_tick_count_reg == 23) begin
                b_tick_count_next = 0;
                next = DATA;
            end else begin
                b_tick_count_next = b_tick_count_reg + 1;
            end
        end
    end
    DATA: begin
        if (b_tick) begin
            if (b_tick_count_reg == 0) begin
                rx_buf_next[7] = rx;
            end
            if (b_tick_count_reg == 15) begin
                if (bit_count_reg == 7) begin
                    next = STOP;
                end else begin
                    b_tick_count_next = 0;
                    bit_count_next = bit_count_reg + 1;
                    rx_buf_next = rx_buf_reg >> 1;
                end
            end else begin
                b_tick_count_next = b_tick_count_reg + 1;
            end
        end
    end
    STOP: begin
        if (b_tick) begin
            rx_done_next = 1'b1;
            next = IDLE;
        end
        default: next = IDLE;
    end
endcase
end
endmodule

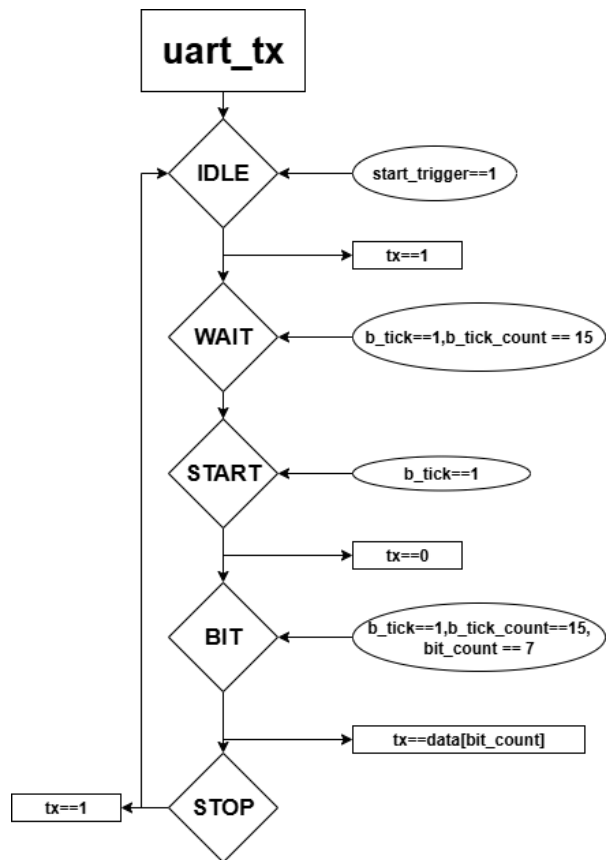
```



# Block diagram

```
always @(*) begin
    bit_count_next = bit_count_reg;
    next = state;
    tx_next = tx_reg;
    data_next = data_reg;
    tx_busy_next = tx_busy_reg;
    b_tick_count_next = b_tick_count_reg;
    case (state)
        IDLE: begin
            tx_next = 1'b1;
            tx_busy_next = tx_busy_reg;
            if (start_trigger == 1'b1) begin
                data_next = tx_data;
                tx_busy_next = 1'b1;
                next = WAIT;
            end
        end
        WAIT: begin
            if (b_tick == 1'b1) begin
                b_tick_count_next = 0;
                next = START;
            end
        end
        START: begin
            tx_next = 1'b0;
            if (b_tick == 1'b1) begin
                if (b_tick_count_reg == 15) begin
                    b_tick_count_next = 0;
                    bit_count_next = 3'b000;
                    next = BIT;
                end else begin
                    b_tick_count_next = b_tick_count_reg + 1;
                end
            end
        end
    end
end
```

```
BIT: begin
    tx_next = data_reg[0];
    if (b_tick == 1'b1) begin
        if (b_tick_count_reg == 15) begin
            b_tick_count_next = 0;
            if (bit_count_next == SEVEN) begin
                bit_count_next = 0;
                next = STOP;
            end else begin
                b_tick_count_next = 0;
                bit_count_next = bit_count_reg + 1;
                data_next = data_reg >> 1;
            end
        end else begin
            b_tick_count_next = b_tick_count_reg + 1;
        end
    end
end
end
STOP: begin
    tx_next = 1'b1;
    if (b_tick == 1'b1) begin
        if (b_tick_count_reg == 15) begin
            tx_busy_next = 1'b0;
            next = IDLE;
        end else begin
            b_tick_count_next = b_tick_count_reg + 1;
        end
    end
end
default: next = IDLE;
endcase
end
endmodule
```



# Schematic

```

always @(*) begin
    wptr_next = wptr_reg;
    rptr_next = rptr_reg;
    full_next = full_reg;
    empty_next = empty_reg;
    case ((
        push, pop
    ))
        2'b01: begin //pop
            full_next = 0;
            if (!empty_reg) begin
                rptr_next = rptr_reg + 1;
                if (wptr_reg == rptr_reg) begin
                    empty_next = 1'b1;
                end
            end
        end
        2'b10: begin //push
            empty_next = 0;
            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'b1) 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
end
endmodule

```

```

FIFO U_rx_fifo (
    .clk      (clk),
    .rst      (rst),
    .push_data(w_rx_data),
    .push      (rx_done),
    .pop      (~w_tx_fifo_full),
    .pop_data  (w_rx_fifo_pop),
    .full      (),
    .empty     (w_rx_empty)
);

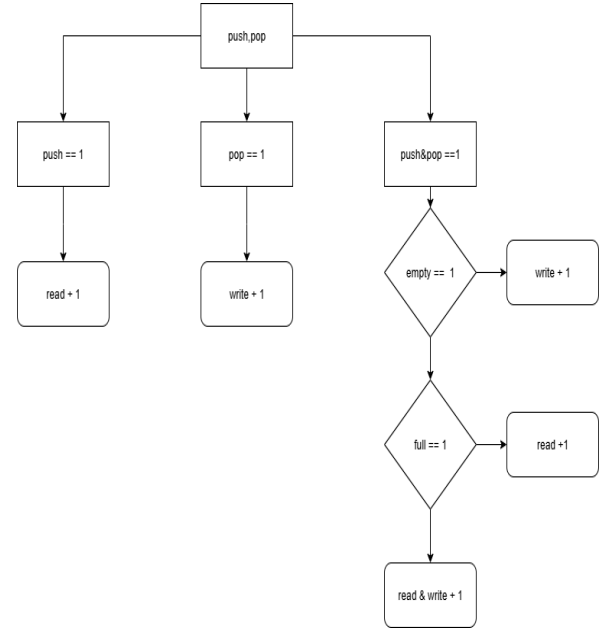
FIFO U_tx_fifo (
    .clk      (clk),
    .rst      (rst),
    .push_data(w_rx_fifo_pop),
    .push      (~w_rx_empty),
    .pop      (~w_tx_busy),
    .pop_data  (w_tx_fifo_pop_data),
    .full      (w_tx_fifo_full),
    .empty     (w_tx_fifo_empty)
);

baud_tick_gen U_baud_tick_gen (
    .clk      (clk),
    .rst      (rst),
    .b_tick   (w_b_tick)
);

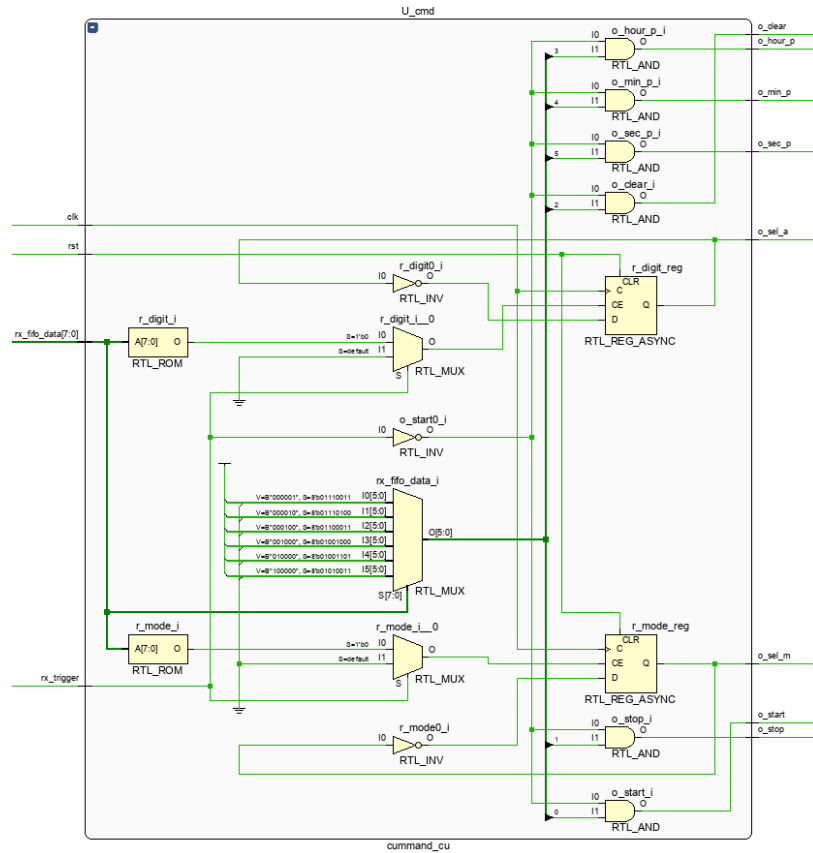
uart_tx U_uart_tx (
    .clk      (clk),
    .rst      (rst),
    .start_trigger(~w_tx_fifo_empty),
    .tx_data   (w_tx_fifo_pop_data),
    .b_tick    (w_b_tick),
    .tx        (tx),
    .tx_busy   (w_tx_busy)
);

uart_rx U_uart_rx (
    .clk      (clk),
    .rst      (rst),
    .rx        (rx),
    .b_tick    (w_b_tick),
    .rx_data   (w_rx_data),
    .rx_done   (rx_done)
);

```

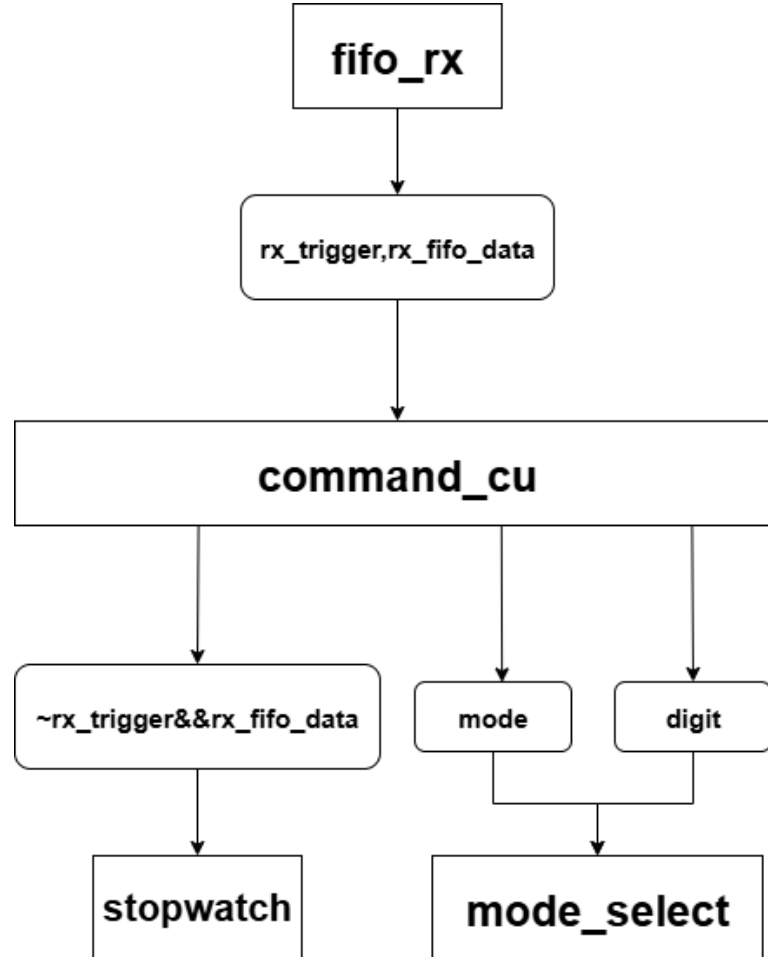


# Schematic

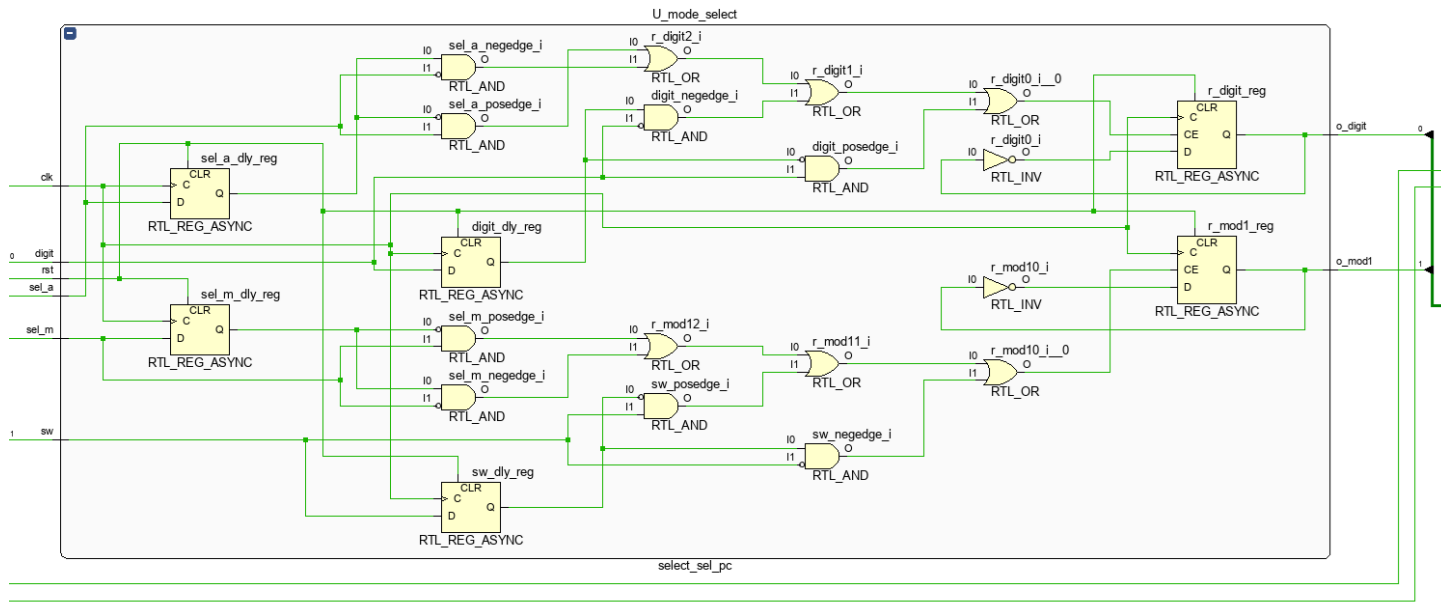


# Schematic

```
module command_cu (  
    input      clk,  
    input      rst,  
    input      rx_trigger,  
    input [7:0] rx_fifo_data,  
    output     o_start,  
    output     o_stop,  
    output     o_clear,  
    output     o_hour_p,  
    output     o_min_p,  
    output     o_sec_p,  
    output     o_sel_m,  
    output     o_sel_a  
);  
  
    reg r_mode,r_digit;  
  
    assign o_start = ~rx_trigger && (rx_fifo_data == 8'h73); // 's'  
    assign o_stop  = ~rx_trigger && (rx_fifo_data == 8'h74); // 't'  
    assign o_clear = ~rx_trigger && (rx_fifo_data == 8'h63); // 'c'  
    assign o_hour_p = ~rx_trigger && (rx_fifo_data == 8'h48); // 'H'  
    assign o_min_p  = ~rx_trigger && (rx_fifo_data == 8'h4D); // 'M'  
    assign o_sec_p  = ~rx_trigger && (rx_fifo_data == 8'h53); // 'S'  
  
    assign o_sel_m = r_mode;  
    assign o_sel_a = r_digit;  
  
    always @(posedge clk or posedge rst) begin  
        if (rst) begin  
            r_mode <= 1'b0;  
            r_digit <= 1'b0;  
        end else begin  
            if (~rx_trigger) begin  
                case (rx_fifo_data)  
                    8'h60: r_mode <= ~r_mode; // 'm'  
                    8'h61: r_digit <= ~r_digit; // 'a'  
                endcase  
            end  
        end  
    end  
endmodule
```



# Schematic



# Schematic

```

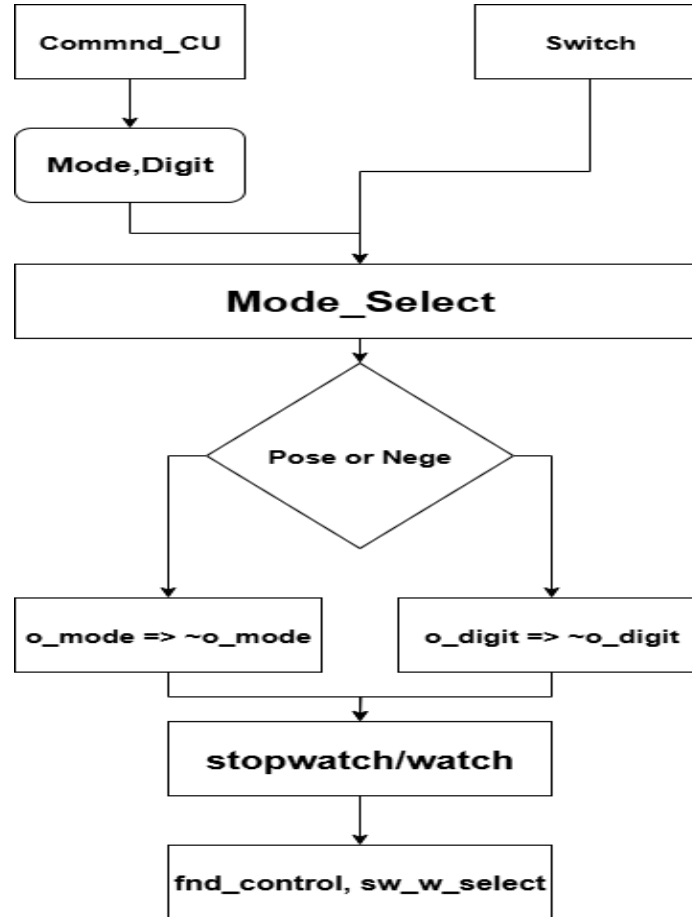
module select_sel_pc (
    input clk,
    input rst,
    input digit,
    input sel_a,
    input sel_m,
    input sw,
    output o_modi,
    output o_digit
);
    reg sel_m_dly, sel_a_dly, sw_dly, digit_dly;
    reg r_modi, r_digit;

    wire sel_a_posedge = ~sel_a_dly & sel_a;
    wire sel_a_negedge = sel_a_dly & ~sel_a;
    wire sel_m_posedge = ~sel_m_dly & sel_m;
    wire sel_m_negedge = sel_m_dly & ~sel_m;
    wire sw_posedge = ~sw_dly & sw;
    wire sw_negedge = sw_dly & ~sw;
    wire digit_posedge = ~digit_dly & digit;
    wire digit_negedge = digit_dly & ~digit;

    assign o_modi = r_modi;
    assign o_digit = r_digit;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            sel_m_dly <= 1'b0;
            sel_a_dly <= 1'b0;
            sw_dly <= 1'b0;
            digit_dly <= 1'b0;
            r_modi <= 1'b0;
            r_digit <= 1'b0;
        end else begin
            sel_m_dly <= sel_m;
            sel_a_dly <= sel_a;
            sw_dly <= sw;
            digit_dly <= digit;
            if (sel_m_posedge || sel_m_negedge || sw_posedge || sw_negedge) begin
                r_modi <= ~r_modi;
            end
            if (sel_a_posedge || sel_a_negedge || digit_posedge || digit_negedge) begin
                r_digit <= ~r_digit;
            end
        end
    end
end
endmodule

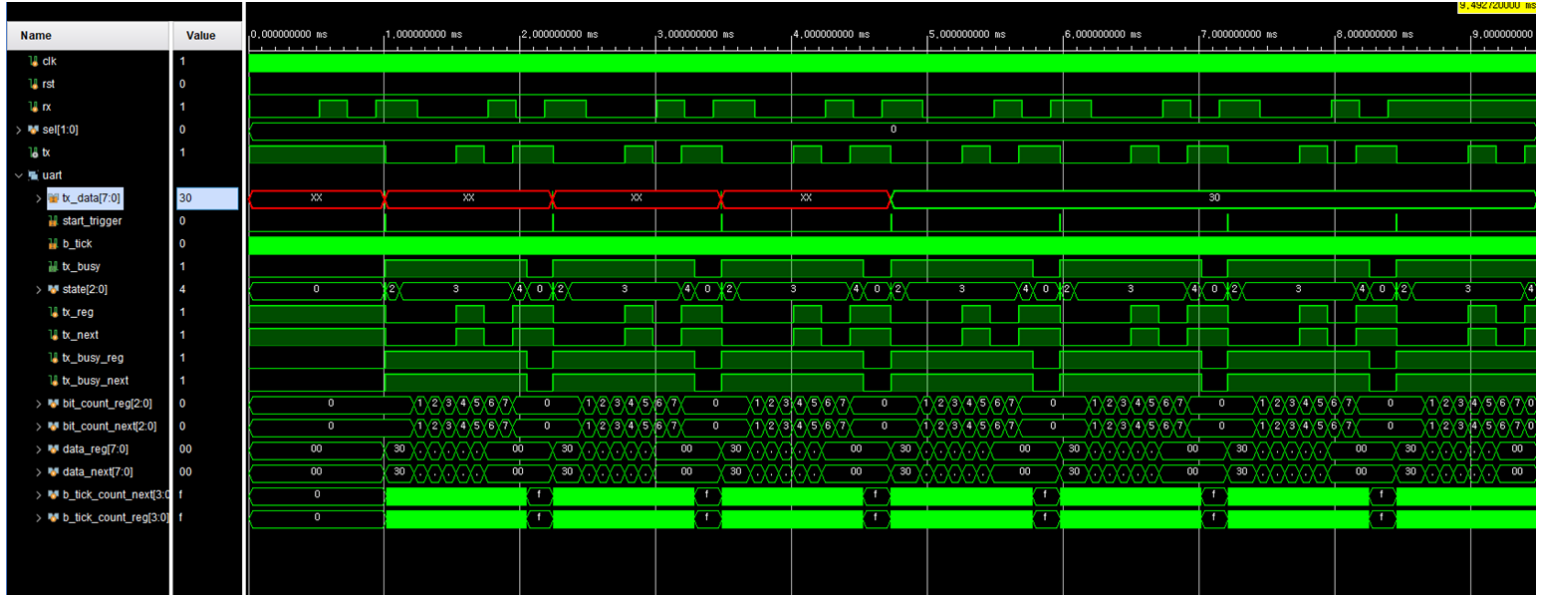
```



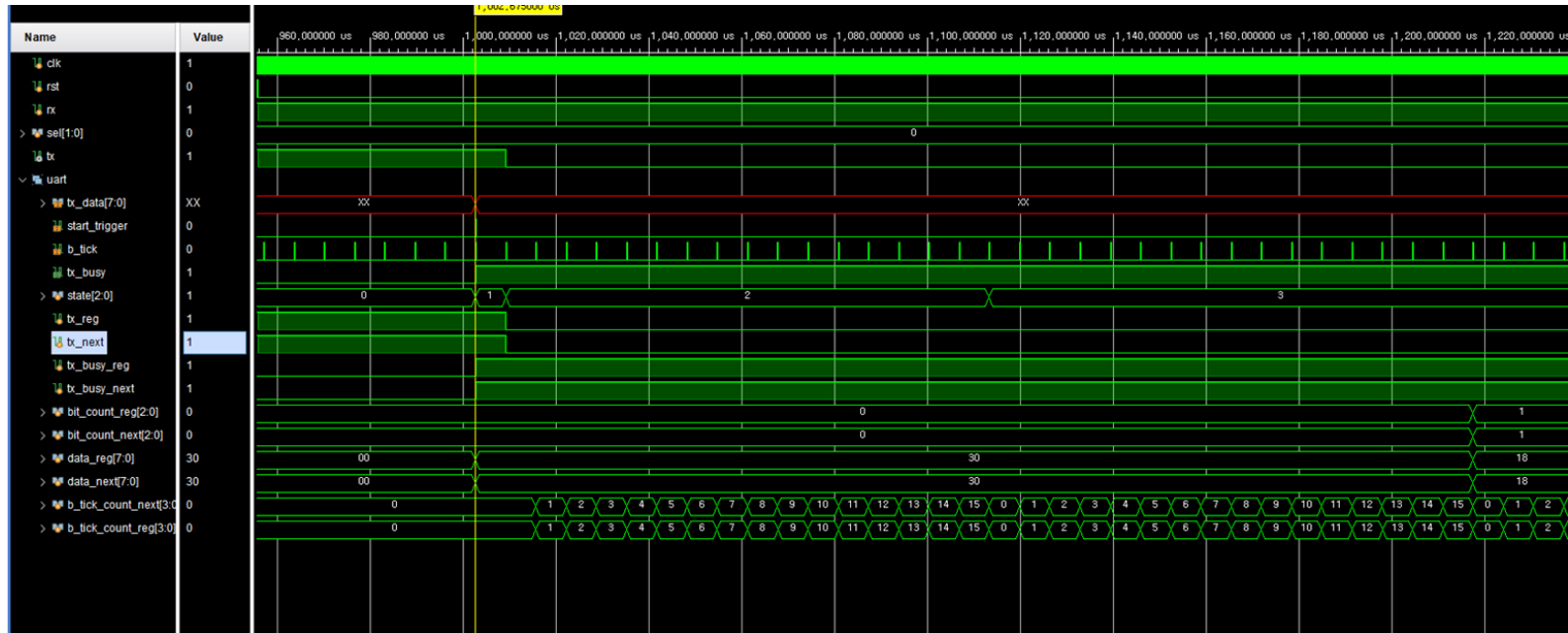
# Simulation

[illegible]

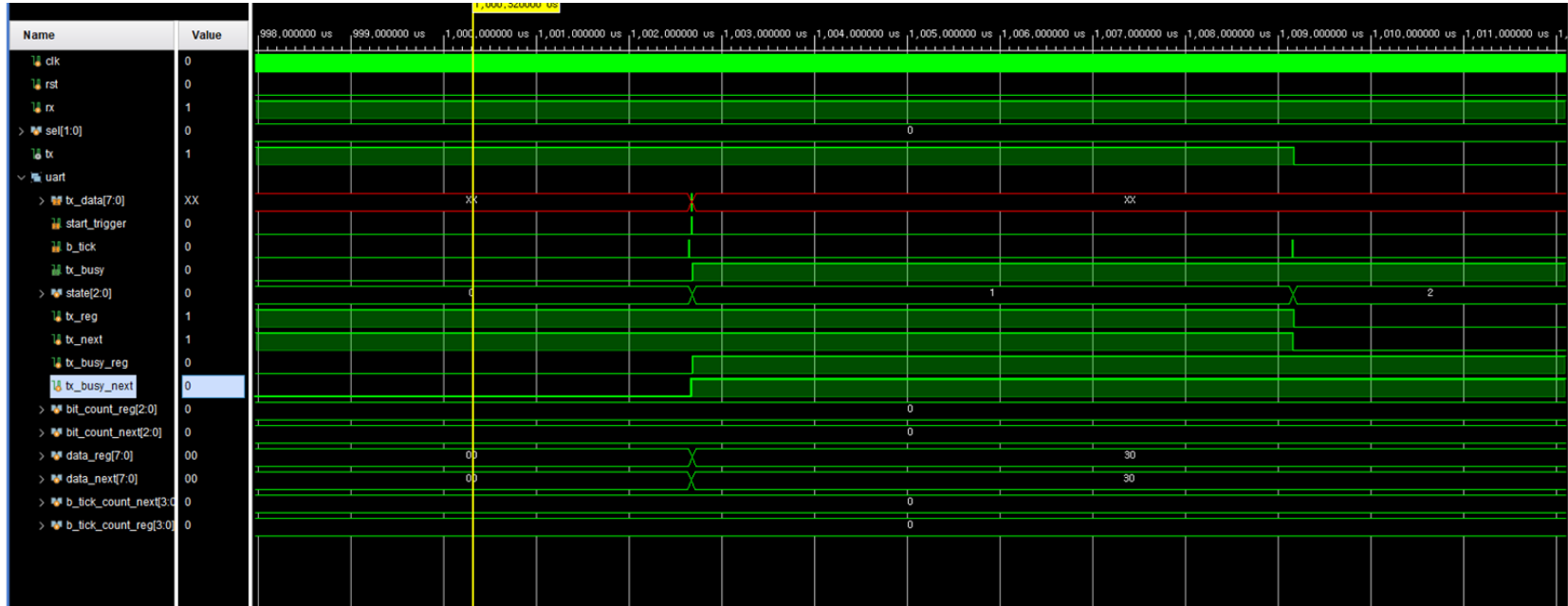
# Simulation - UART\_RX



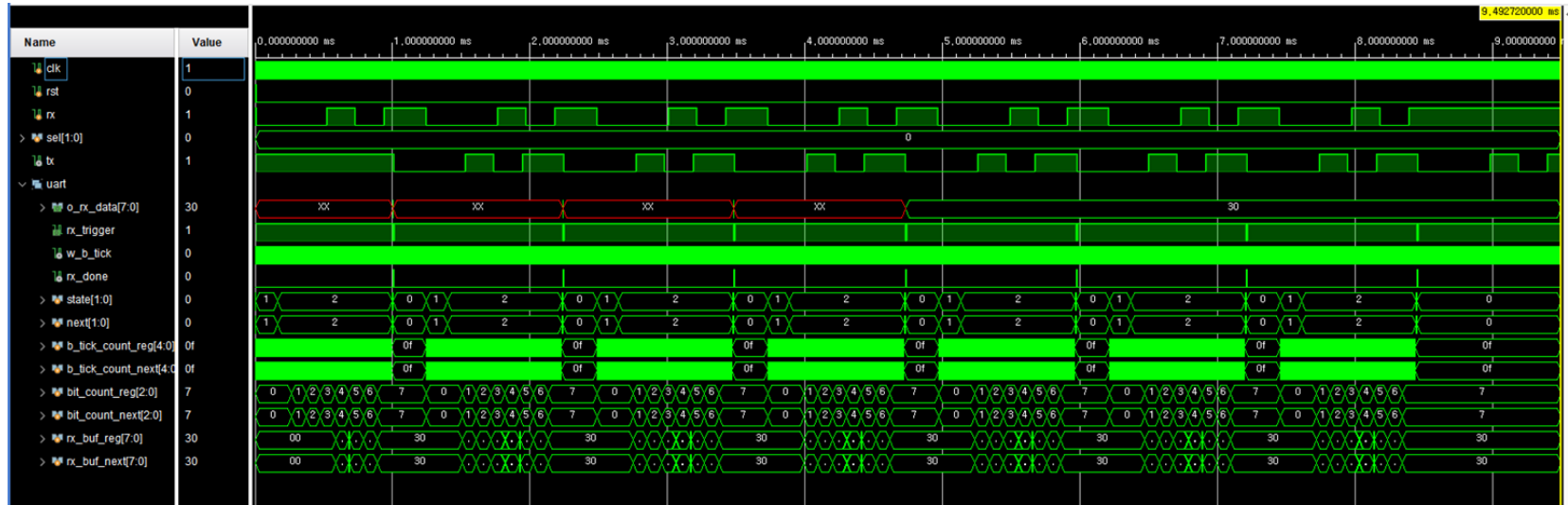
# Simulation - UART\_RX



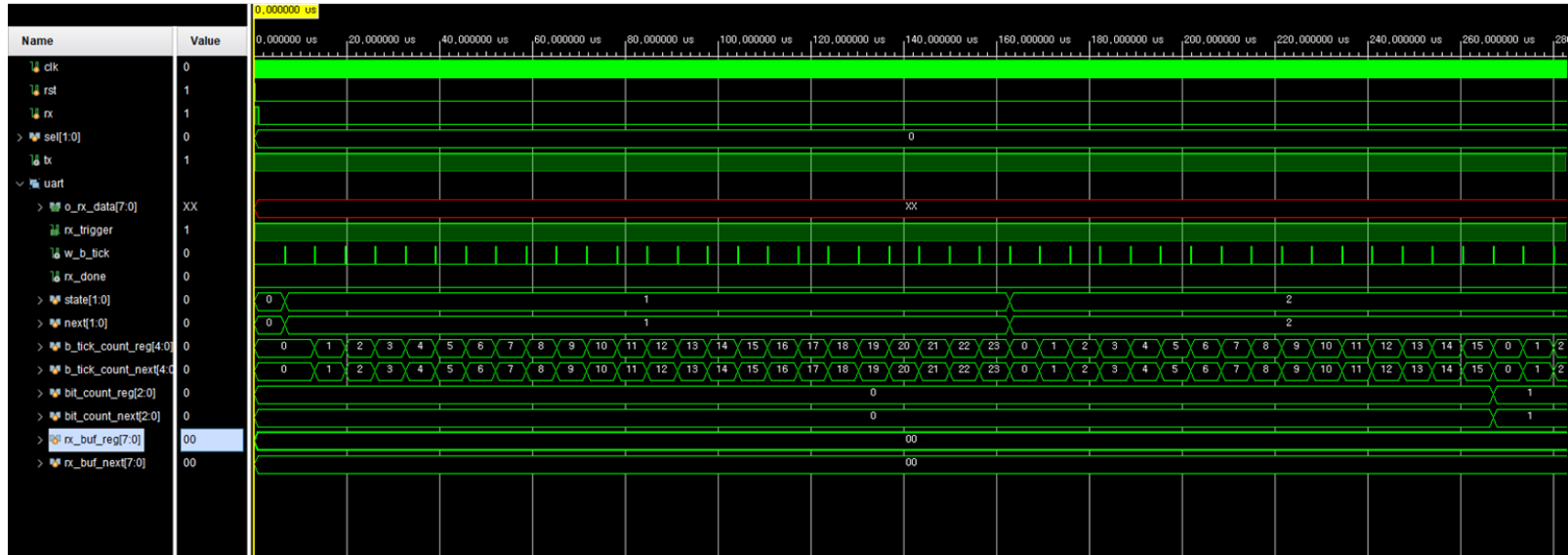
# Simulation - UART\_RX



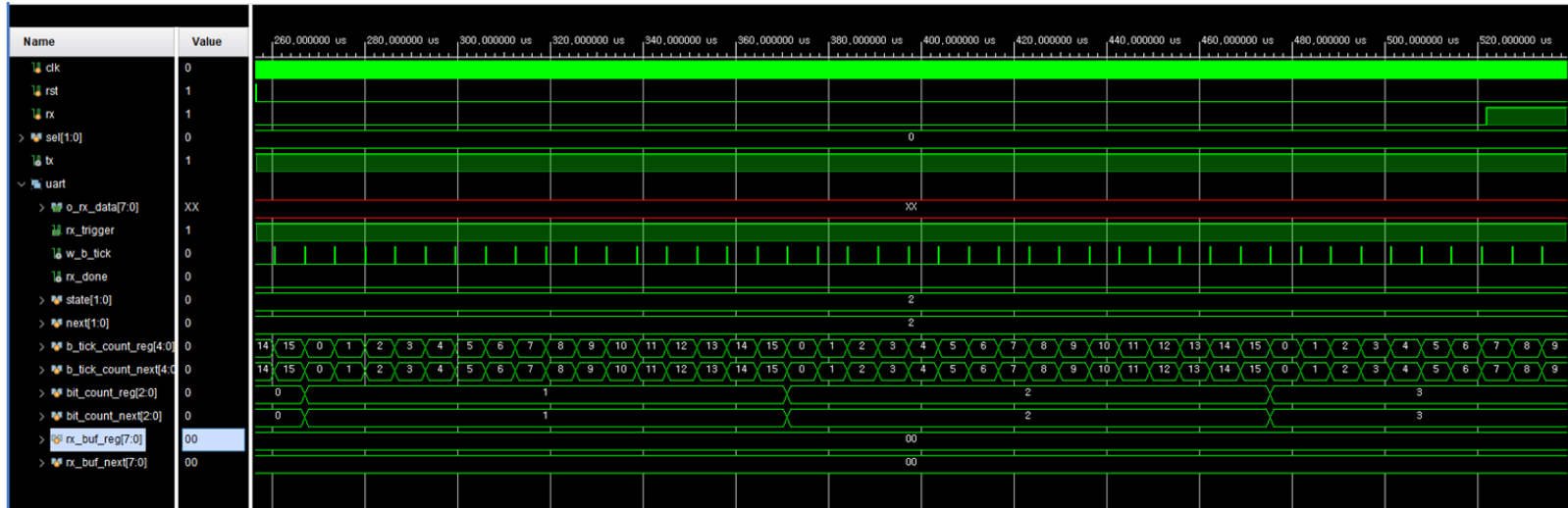
# Simulation - UART\_TX



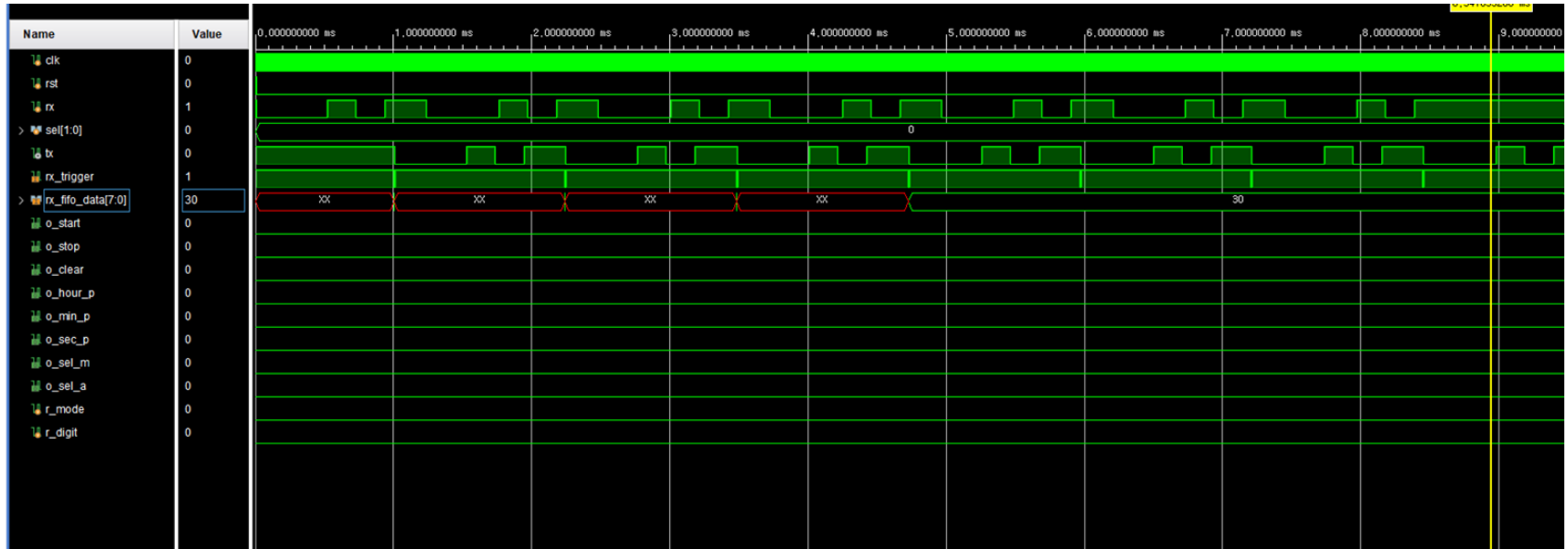
# Simulation - UART\_TX



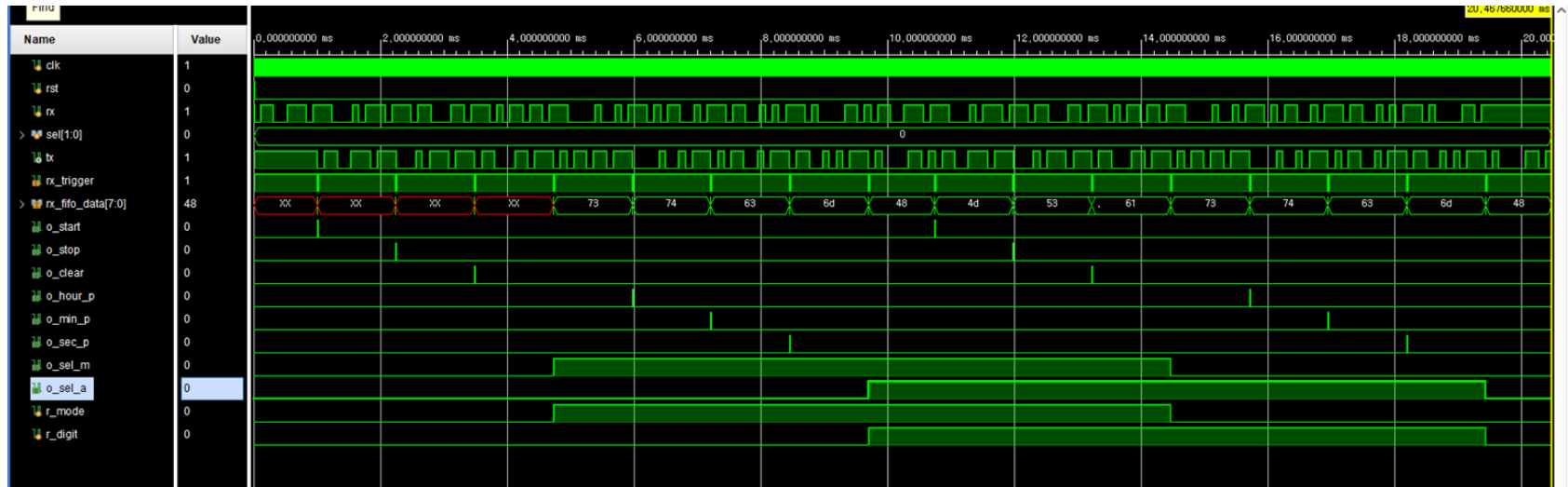
# Simulation - UART\_TX



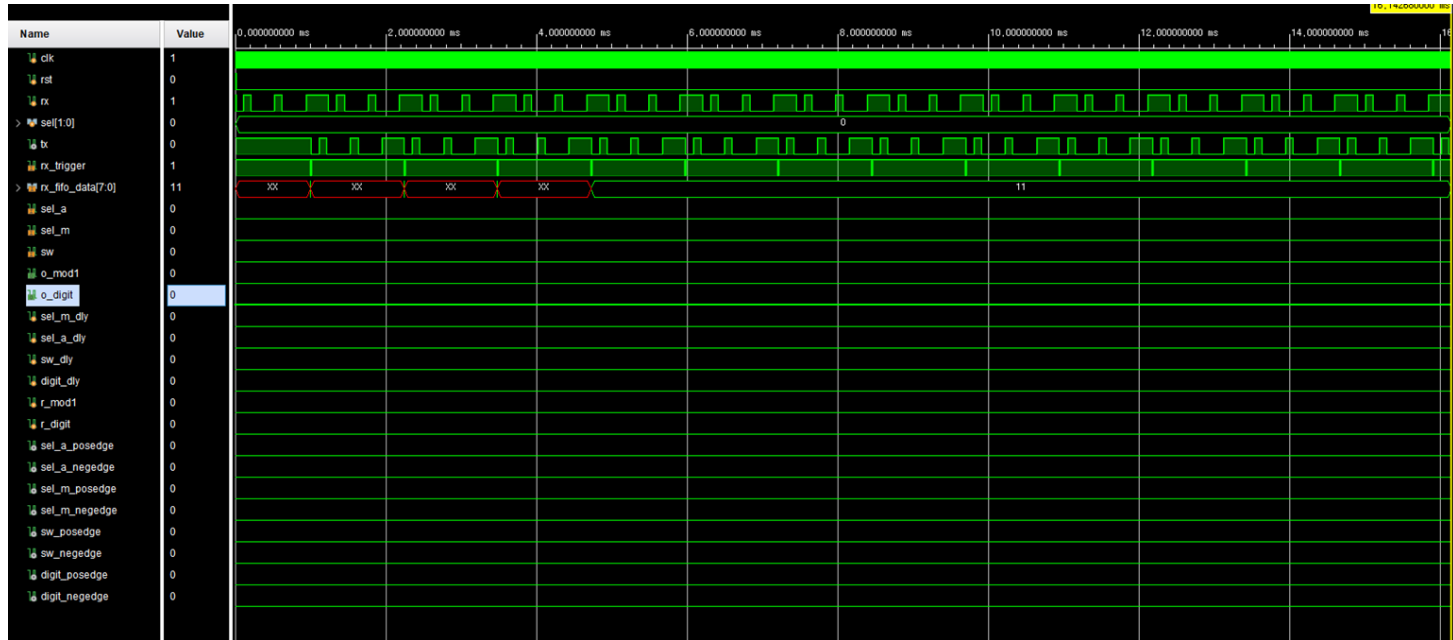
# Simulation - CMD\_CU



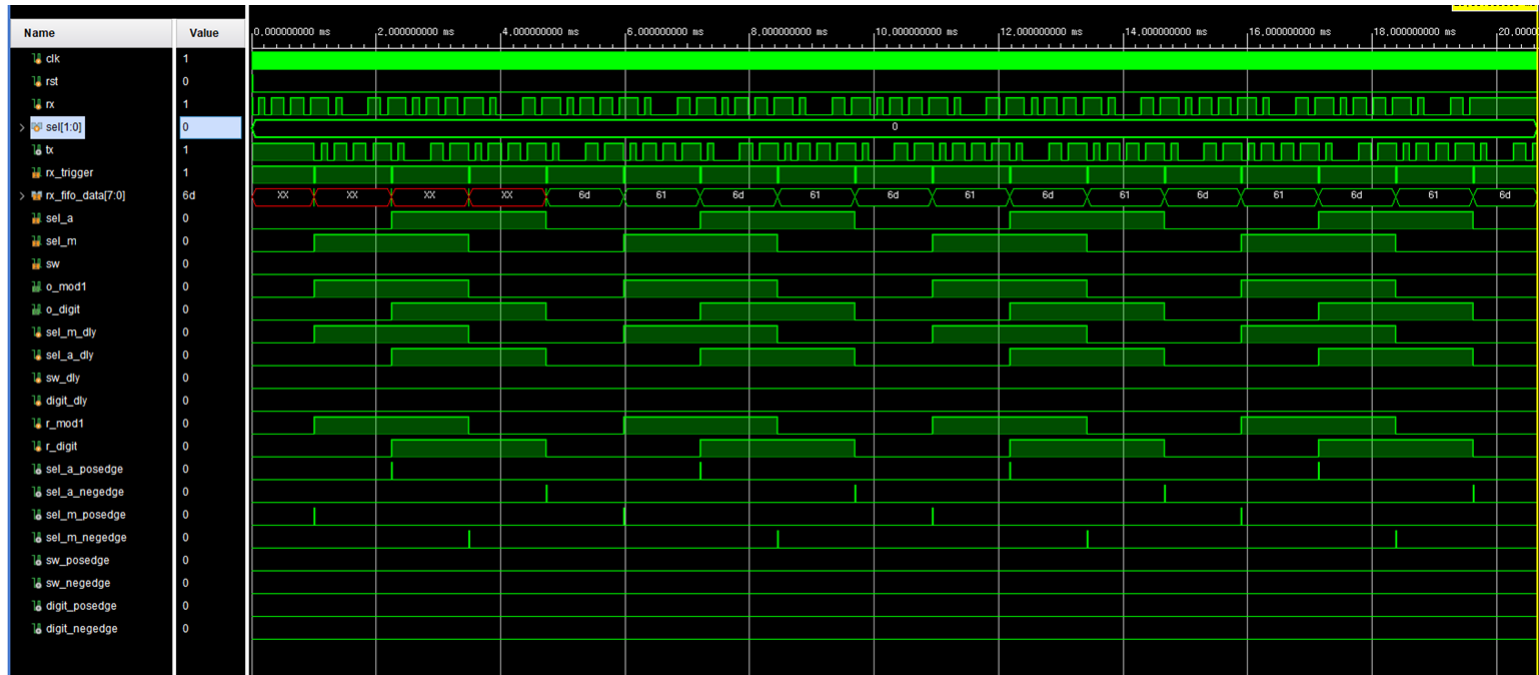
# Simulation - CMD\_CU



# Simulation - MODE\_SEL



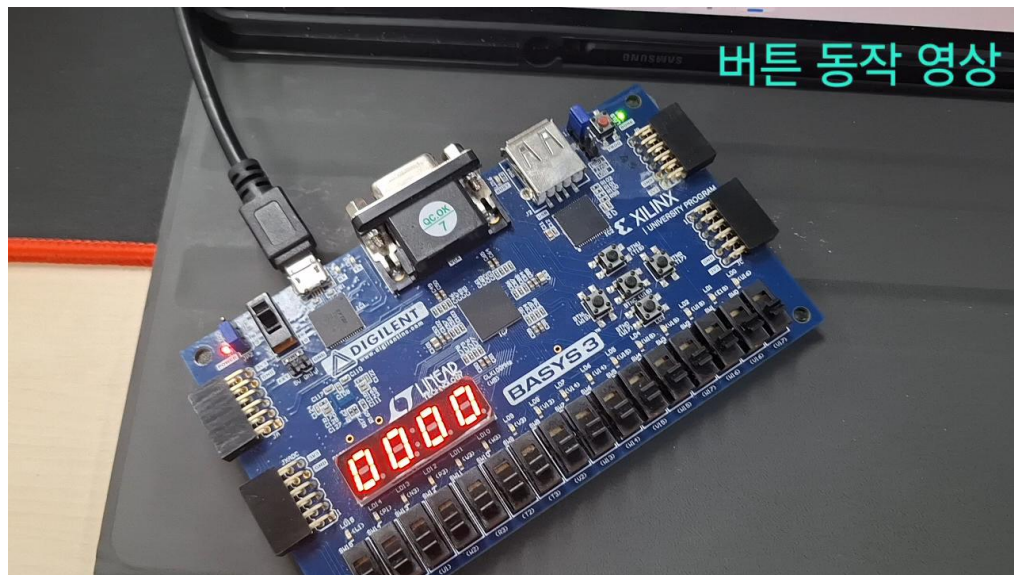
# Simulation - MODE\_SEL



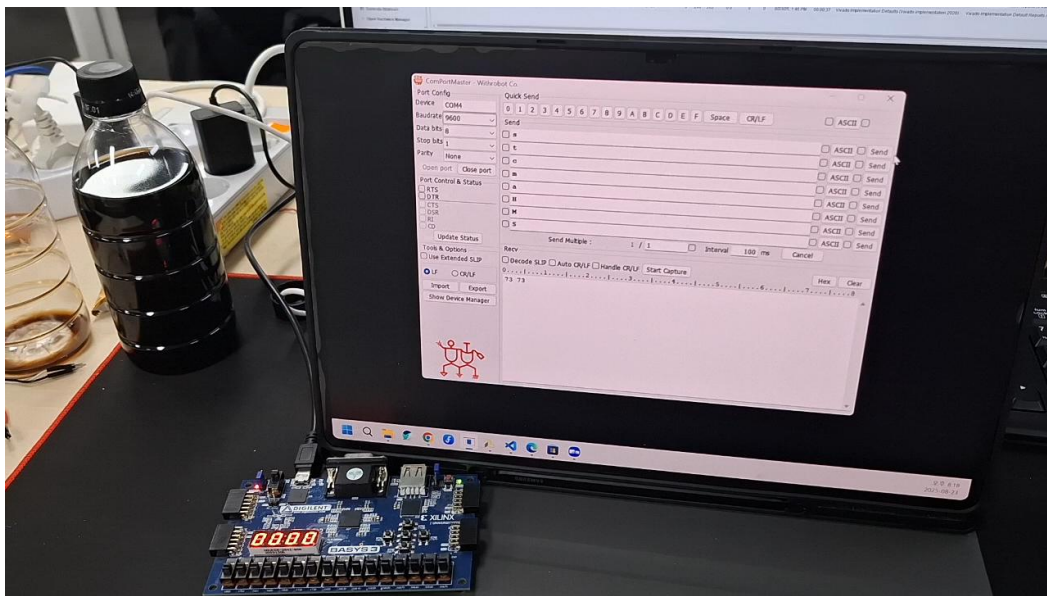
# Simulation - MODE\_SEL



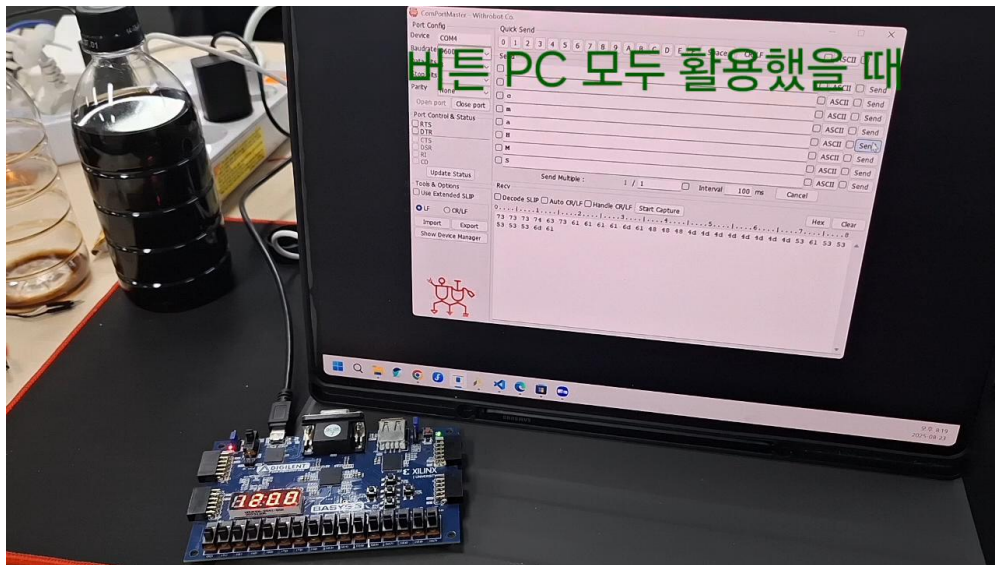
# 동작 영상



## 동작 영상



# 동작 영상



# Troubleshooting

PC에서 값을 한번 보내면 동작을 하지 않고 3~4번 보내야 동작 함

=> 처음에는 rx\_trigger가 1이 되었을 때 동작을 시켰으나 디버깅 과정에서 rx\_trigger 값이 1을 유지 하는 것으로 확인되어 rx\_trigger가 1이되는 시점이 아닌 0이 되는 시점으로 변경

```
module cummand_cu (  
    input      clk,  
    input      rst,  
    input      rx_trigger,  
    input [7:0] rx_fifo_data,  
    output     o_start,  
    output     o_stop,  
    output     o_clear,  
    output     o_hour_p,  
    output     o_min_p,  
    output     o_sec_p,  
    output     o_sel_m,  
    output     o_sel_a  
);  
  
    reg r_mode,r_digit;  
  
    assign o_start  = ~rx_trigger && (rx_fifo_data == 8'h73); // 's'  
    assign o_stop   = ~rx_trigger && (rx_fifo_data == 8'h74); // 't'  
    assign o_clear  = ~rx_trigger && (rx_fifo_data == 8'h63); // 'c'  
    assign o_hour_p = ~rx_trigger && (rx_fifo_data == 8'h48); // 'H'  
    assign o_min_p  = ~rx_trigger && (rx_fifo_data == 8'h40); // 'M'  
    assign o_sec_p  = ~rx_trigger && (rx_fifo_data == 8'h53); // 'S'  
  
    assign o_sel_m = r_mode;  
    assign o_sel_a = r_digit;  
  
    always @(posedge clk or posedge rst) begin  
        if (rst) begin  
            r_mode <= 1'b0;  
            r_digit <= 1'b0;  
        end else begin  
            if (~rx_trigger) begin  
                case (rx_fifo_data)  
                    8'h6D: r_mode <= ~r_mode; // 'm'  
                    8'h61: r_digit <= ~r_digit; // 'a'  
                endcase  
            end  
        end  
    end  
endmodule
```

# Troubleshooting

sec&msec/hour&min 와 Stopwatch/Watch의 모드가  
변화하는 과정에서 pc에서 값을 두번 보내야 모드가  
변하는 현상이 생김

=> cmd\_cu에서 값을 반전시켜 Mode\_select에 들어  
오는데 이를 라이징 엣지에서만 인식을 가능하게  
만들어서 폴링엣지 즉 첫번째 다음에 오는 값은 인  
식하지 못하는 현상이 생김 이를 폴링 엣지 에서도  
인식시키게 만들면서 해결함함

```
module select_sel_pc (  
    input  clk,  
    input  rst,  
    input  digit,  
    input  sel_a,  
    input  sel_m,  
    input  sw,  
    output o_mod1,  
    output o_digit  
);  
  
    reg sel_m_dly, sel_a_dly, sw_dly, digit_dly;  
    reg r_mod1, r_digit;  
  
    wire sel_a_posedge = ~sel_a_dly & sel_a;  
    wire sel_a_negedge = sel_a_dly & ~sel_a;  
    wire sel_m_posedge = ~sel_m_dly & sel_m;  
    wire sel_m_negedge = sel_m_dly & ~sel_m;  
    wire sw_posedge = ~sw_dly & sw;  
    wire sw_negedge = sw_dly & ~sw;  
    wire digit_posedge = ~digit_dly & digit;  
    wire digit_negedge = digit_dly & ~digit;  
  
    assign o_mod1 = r_mod1;  
    assign o_digit = r_digit;  
  
    always @(posedge clk or posedge rst) begin  
        if (rst) begin  
            sel_m_dly <= 1'b0;  
            sel_a_dly <= 1'b0;  
            sw_dly <= 1'b0;  
            digit_dly <= 1'b0;  
            r_mod1 <= 1'b0;  
            r_digit <= 1'b0;  
        end else begin  
            sel_m_dly <= sel_m;  
            sel_a_dly <= sel_a;  
            sw_dly <= sw;  
            digit_dly <= digit;  
            if (sel_m_posedge || sel_m_negedge || sw_posedge || sw_negedge) begin  
                r_mod1 <= ~r_mod1;  
            end  
            if (sel_a_posedge || sel_a_negedge || digit_negedge || digit_posedge) begin  
                r_digit <= ~r_digit;  
            end  
        end  
    end  
endmodule
```

**THE END**