

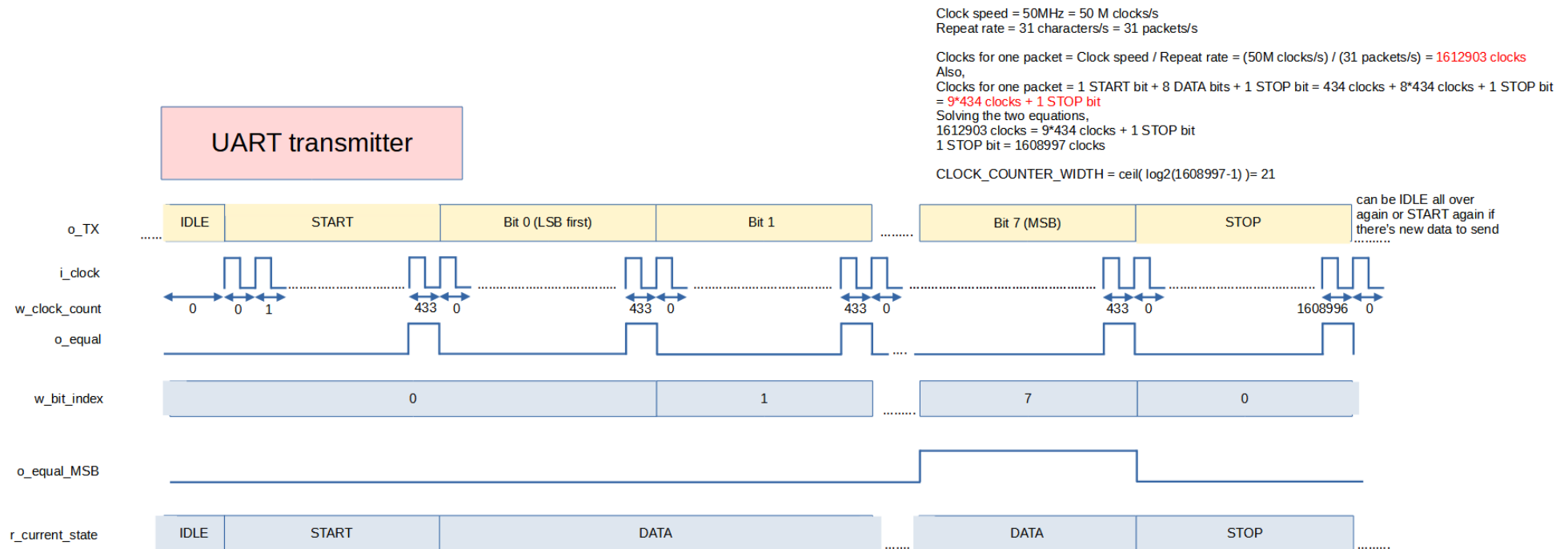
# UART Transmitter Design Ideas

This section is to understand how to formulate the design intuitively.

Take note that this project works for clock speed other than 50MHz and bit rate other than 115200 bits/s too (simply change the appropriate parameters in verilog files). In this section, we will see how the parameters are calculated.

Let's say we are using a CPLD with clock speed 50MHz, and we are interested in designing an UART receiver with bit rate 115200 bits/s. Then,  $\text{CLOCKS\_PER\_BIT} = \text{clock speed} / \text{bit rate} = (50 \text{ M clocks/s}) / (115200 \text{ bits/s}) \approx 434 \text{ clocks per bit}$ . Besides, we set the data width (number of data bits in a packet) to 8 bits (1 byte).

Now, let's discuss the timing diagram below. Do not worry about the initial underscore like "i\_", "w\_", "o\_", "r\_". They are only used in Verilog coding later on (to represent input, wire, output, register respectively). They are not important for the design formulation.



Initially, when there is nothing to transmitt, `r_current_state` of FSM is IDLE and `o_TX` is high. When there is something to send (e.g. it might be some event that triggers sending, like in this project, the trigger is when FSM sees that we are pressing push buttons), `r_current_state` transitions into START state and the value to be sent can be instructed to be loaded into transmitter shift register by the START state.

In START state, the clock counter will start counting, and take note that the clock count is `w_clock_count`. Since `CLOCKS_PER_BIT` is 434, the clock count will be reset after `w_clock_count=434-1=433`. To tell the FSM to transitions from START state into DATA state, we can make use of the `o_equal` pulse generated when `w_clock_count=433`. The counting continues until `w_clock_count=433` again, then transmitter shift register shift by one bit (so that `o_TX` changes), and `w_bit_index` is incremented by one from 0 to 1. This process of clock counting, shifting, and incrementing `w_bit_index` is continued for all data bits. When sending MSB bit, `o_equal_MSB=1`, and this can be used to tell FSM to transition into STOP state. After sending the MSB bit, `w_bit_index` is reset back to zero and FSM transitions into STOP state.

For the STOP bit, the `w_clock_count` reaches 1608996 instead of 433 before resetting. Basically the STOP bit is to limit how fast can data be sent out. For example, in this project, when we press and hold a push button, the key value of the push button is sent at repeat rate of 31 times per second. To know the length of STOP bit (`CLOCKS_FOR_STOP`) to limit repeat rate to 31 times per second, the calculation is as follows.

Clock speed = 50MHz = 50 M clocks/s

Repeat rate = 31 characters/s = 31 packets/s

Clocks for one packet = Clock speed / Repeat rate = (50M clocks/s) / (31 packets/s) = 1612903 clocks

Also,

Clocks for one packet = 1 START bit + 8 DATA bits + 1 STOP bit = 434 clocks + 8\*434 clocks + 1 STOP bit = 9\*434 clocks + 1 STOP bit

Solving the two equations,

1612903 clocks = 9\*434 clocks + 1 STOP bit

`CLOCKS_FOR_STOP` = 1 STOP bit = 1608997 clocks

`CLOCK_COUNTER_WIDTH` = `ceil( log2(1608997-1) )` = 21

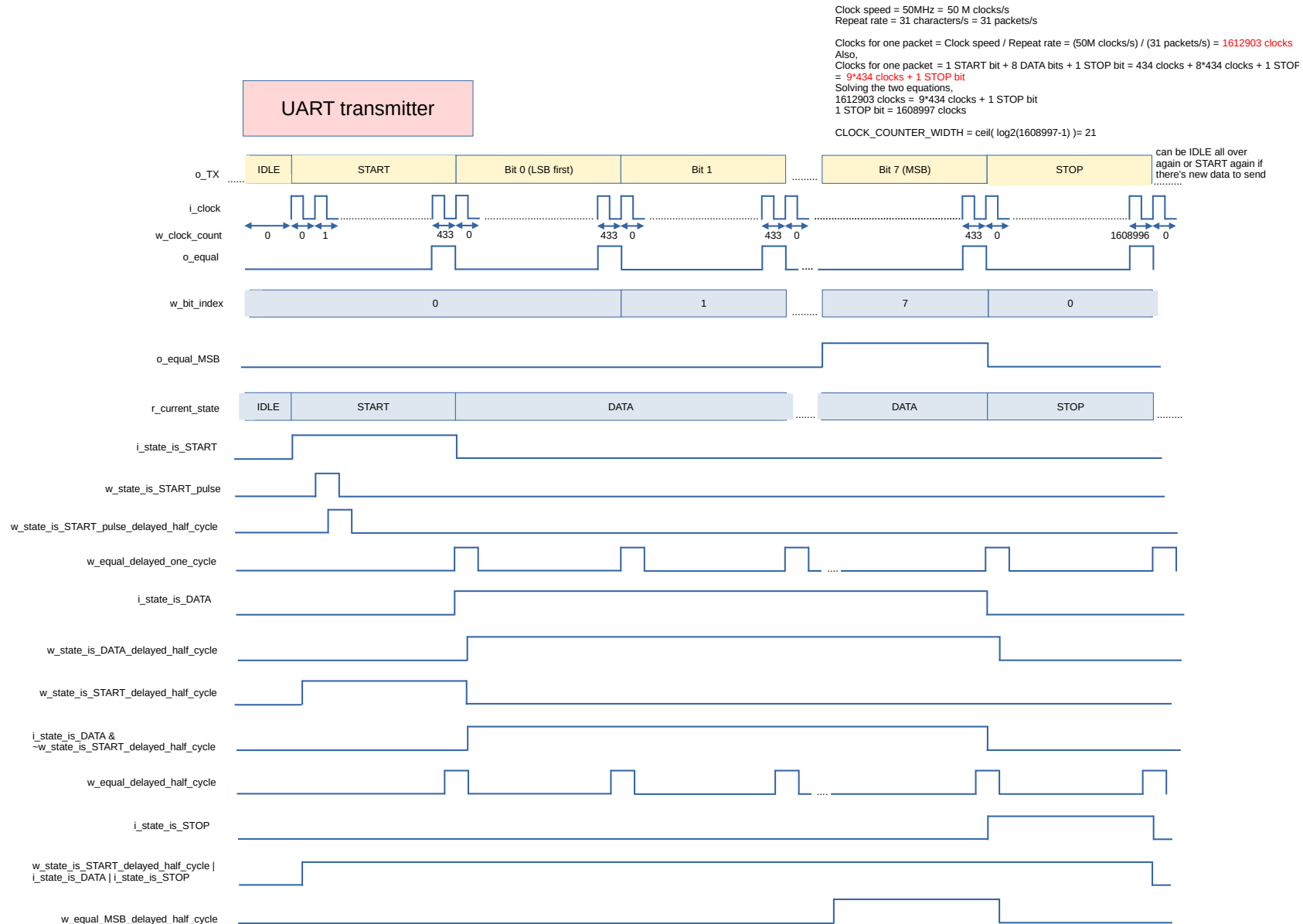
After the STOP bit is complete, the transmitter can send START bit again if there is still data to send, or back to IDLE state, sit back and relax.

With the thinking process so far, we can see that our UART receiver design should consist of four things

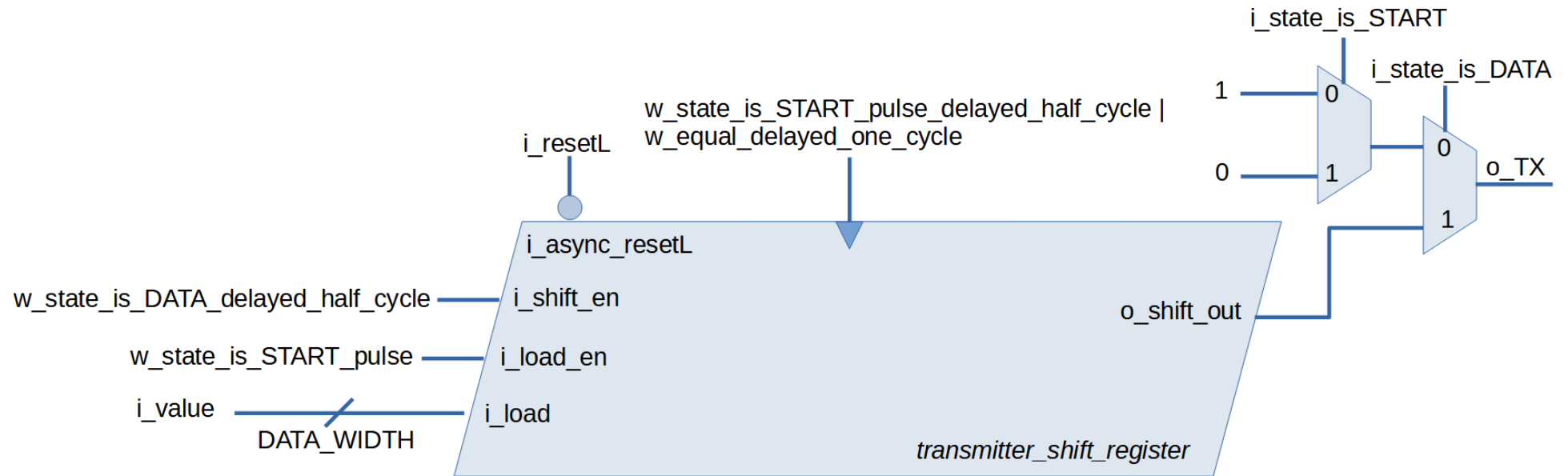
- 1) `transmitter_shifter.v`: To shift out serially the parallel load to `o_TX`.
- 2) `transmitter_bit_counter.v`: To keep track of `w_bit_index` and output `o_equal_MSB`. `o_equal_MSB` is used to tell `r_current_state` when to transition from DATA to STOP state.
- 3) `transmitter_clock_counter.v`: To keep track of `w_clock_count` and output `o_equal`. `o_equal` is useful to reset clock counter itself, tell FSM when to transition, tell when `transmitter_shifter` should shift, tell when bit counter should increment etc.
- 4) `transmitter_control.v`: The FSM that will tell `r_current_state`. `r_current` state can be used to disable shifting of `transmitter_shifter`, tell when `bit_counter` and `clock_counter` should not be counting etc.

# UART Transmitter Datapath Design

When discussing datapath and control unit design, this timing diagram will be referred extensively.



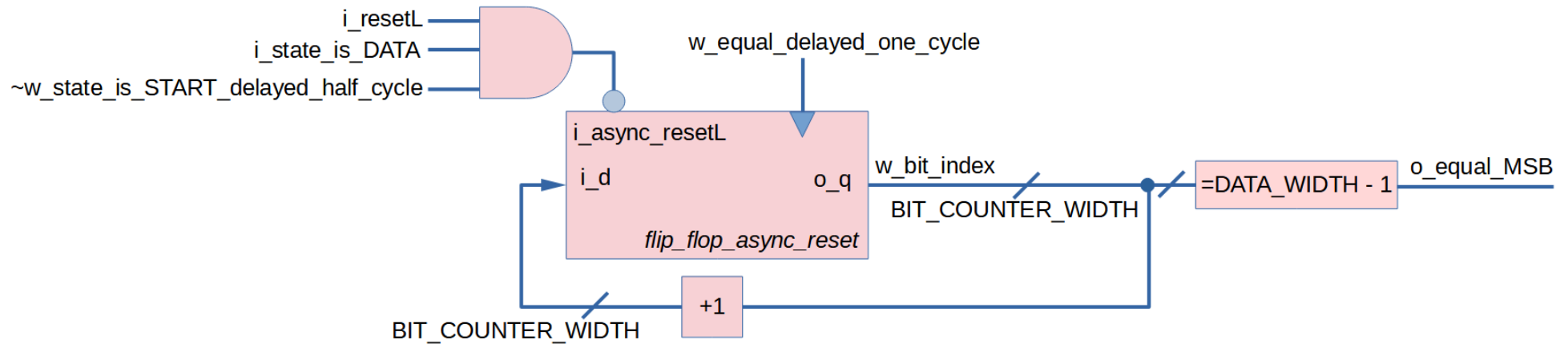
## transmitter\_shifter.v



Transmitter shifter consists of `transmitter_shift_register` and two multiplexers. The `transmitter_shift_register` is a parallel-in-serial-out (PISO) shift register. Internally the implementation is based on rotation register, but, whether we use rotation register or right shift register internally, it doesn't matter since on the outside we are using it as shift register. During IDLE state, the multiplexer selects logic 0 for the `o_TX`. During START state, a pulse (`w_state_is_START_pulse`) is generated from `i_state_is_START`, and this pulse is used to load data to be sent into the shift register at posedge of the pulse delayed half cycle (`w_state_is_START_pulse_delayed_half_cycle`). Also, during START state, the multiplexer selects logic 0 for the `o_TX`.

During DATA state, the multiplexer selects `o_shift_out` of the shift register. Also, the shift register is allowed to shift, with the posedge trigger being `w_equal_delayed_one_cycle`. At first thought, it seems like `i_shift_en` should be `i_state_is_DATA`, but to prevent unwanted shifting due to posedge of `w_equal_delayed_one_cycle` right after the START bit, we delay `i_state_is_DATA` by half cycle to get `w_state_is_DATA_delayed_half_cycle`. It turns out delaying by half cycle has an additional benefit of providing an extra shift right after sending MSB bit, such that the internal rotation register in `transmitter_shift_register` completes one cycle of rotation and is back to the data it loaded during START state. This is not an important point, just an extra for debugging if wanted.

## transmitter\_bit\_counter.v

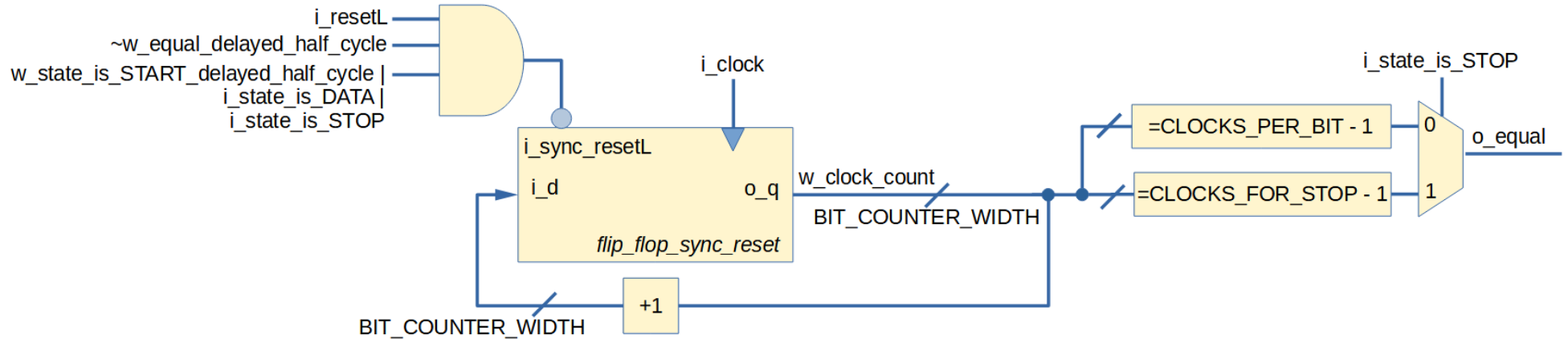


This transmitter\_bit\_counter.v is actually the same as receiver\_bit\_counter.v, anyway, let's discuss the same thing again here.

The bit counter consists of a flip flop with asynchronous reset, a comparator to tell `o_equal_MSB` and an adder to increment `w_bit_index` by one. Since we are sending 8 data bits per packet,  $\text{BIT\_COUNTER\_WIDTH} = \lceil \log_2(\text{DATA\_WIDTH}-1) \rceil = \lceil \log_2(8-1) \rceil = 3$ . The clock for bit counter is `w_equal_delayed_one_cycle` (i.e. `o_equal` delayed by one cycle as discussed before in timing diagram).

There are three conditions when we should not reset the bit counter. Firstly, the most obvious condition is when `i_resetL` (the global active low reset) is high. Secondly, we don't want reset when `i_state_is_DATA`. However, on a closer inspection on the timing diagram, we notice that having the second condition is still not enough, because the `w_equal_delayed_one_cycle` right after the START bit may cause unwanted increment of `w_bit_index`. So, the third condition is to make sure `resetL` is zero during the first half cycle of the first clock during `i_state_is_DATA`, and this is done by & with `~w_state_is_START_delayed_half_cycle`. All together, we feed the `i_async_resetL` with the expression `i_resetL & i_state_is_DATA & ~w_state_is_START_delayed_half_cycle`.

## transmitter\_clock\_counter.v



The clock counter consists of a flip flop with active low synchronous reset, adder to increment `w_clock_count` by 1, two comparators and one multiplexers. The maximum value that `w_clock_count` can be is  $\text{CLOCKS\_FOR\_STOP} - 1 = 1608997 - 1 = 1608996$ , so  $\text{CLOCK\_COUNTER\_WIDTH} = \text{ceil}(\log_2(\text{CLOCKS\_FOR\_STOP} - 1)) = \text{ceil}(\log_2(1608996)) = 21$ .

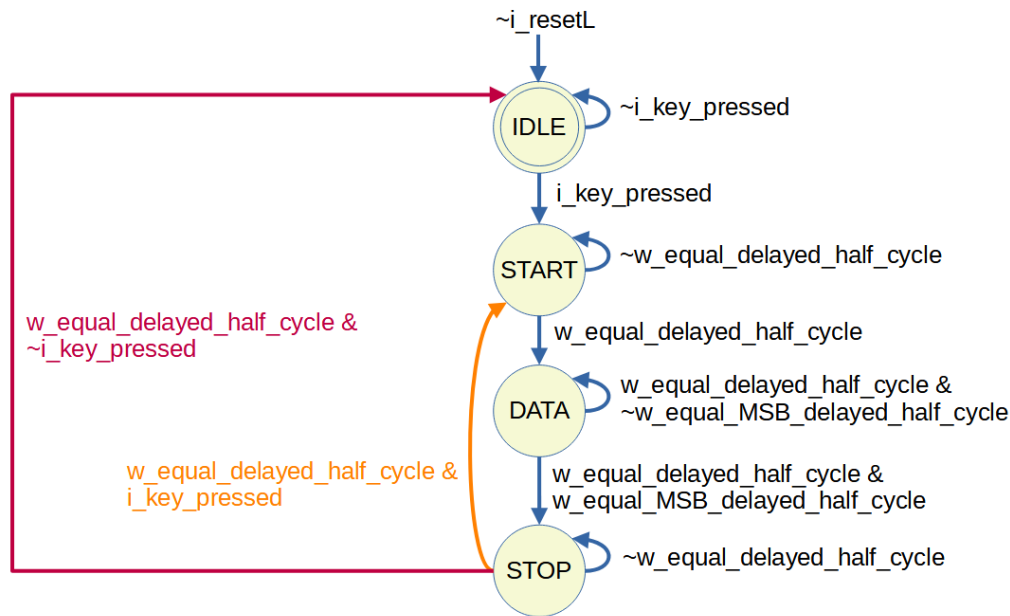
The clock for the flip flop is `i_clock`. Using states from FSM as selector for the multiplexers, we can select at what value should the clock counter reset. When state is STOP, the number of clocks to elapse is `CLOCKS_FOR_STOP`. Else (during START bit and each DATA bit), we only need `CLOCKS_PER_BIT`.

Now let's consider when we should not reset the clock counter. First condition is obviously when the global active-low reset is high (`i_resetL = 1`). Secondly, we need reset when `w_equal_delayed_half_cycle` (i.e. `o_equal` delayed by half cycle), and because reset is active low, we negate it to  $\sim w\_equal\_delayed\_half\_cycle$ . Third condition where we don't want reset is when current state is START or DATA or STOP. However, taking a closer look at timing diagram, to prevent unwanted `w_clock_count` increment right after the middle of the START bit, we use `w_state_is_START_delayed_half_cycle` (i.e. `i_state_is_START` delayed half cycle). All together, we feed `i_sync_resetL` with  $i\_resetL \& \sim w\_equal\_delayed\_half\_cycle \& (w\_state\_is\_START\_delayed\_half\_cycle \mid i\_state\_is\_DATA \mid i\_state\_is\_STOP)$ .

# UART Transmitter Control Unit Design

The control unit is nothing but an FSM.

## transmitter\_control.v



The logic `w_equal_delayed_half_cycle` is `o_equal` delayed half cycle, while `w_equal_MSB_delayed_half_cycle` is `o_equal` delayed half cycle. Delaying by half cycle is done intentionally to prevent hold time violation. Although synthesizer may insert delay to ensure zero hold time requirement, we would rather be safe and do the delay ourselves.

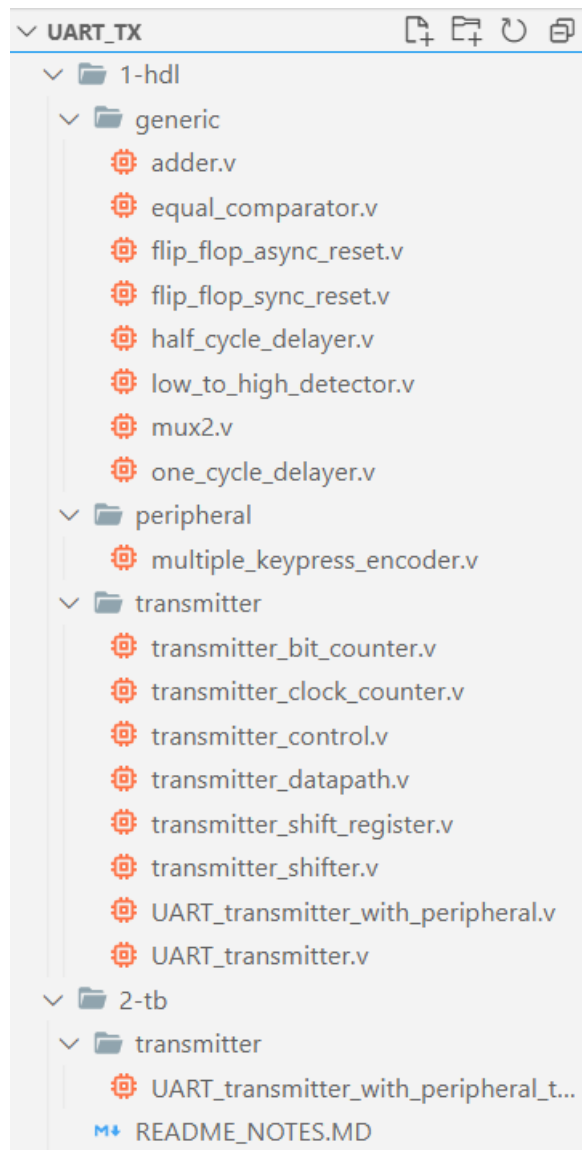
Starting from IDLE state, as long as there is no trigger (i.e. `i_key_pressed` in this project), the state stays. Else, it's time to start sending the data, so the state transitions to START state. As long as clock counter hasn't issued pulse (`w_equal_delayed_half_cycle`) to transition to DATA state, the state stays. Else when there is pulse issued (due to `CLOCKS_PER_BIT` has elapsed), state transitions to DATA.

During the DATA state, there will be a few pulses of `w_equal_delayed_half_cycle`, but eventually what will cause the state to transition to STOP state is if it is accompanied with `w_equal_MSB_delayed_half_cycle` (which tells FSM that MSB bit has already completed transmitting).

At STOP state, if the clock counter hasn't issued pulse (`w_equal_delayed_half_cycle`), the state stays. Else, when there is pulse, the FSM check if key is still being pressed. If it is, then the state transitions to START to initiate transmission. Else, the state simply goes back to IDLE.

# UART Transmitter Code

Here are all the code for this UART transmitter project. The Verilog HDL files are organized in the repository like this:





## Generic

This folder contains commonly-used blocks.

generic/adder.v	generic/equal_comparator.v
<pre>module adder #(   parameter WIDTH=8 )()   input  [WIDTH-1:0] i_a, i_b,   output [WIDTH-1:0] o_y );   assign o_y = i_a + i_b; endmodule</pre>	<pre>module equal_comparator#(   parameter WIDTH=8 )()   input  [WIDTH-1:0] i_a, i_b,   output              o_c );   assign o_c = (i_a == i_b); endmodule</pre>
generic/flip_flop_async_reset.v	generic/flip_flop_sync_reset.v
<pre>module flip_flop_async_reset #(   parameter WIDTH=8 )()   input          i_clock,   input          i_async_resetL,   input  [WIDTH-1:0] i_d,   output reg [WIDTH-1:0] o_q );   always @(posedge i_clock or negedge i_async_resetL) begin     if (~i_async_resetL) o_q &lt;= 0;     else o_q &lt;= i_d;   end endmodule</pre>	<pre>module flip_flop_sync_reset #(   parameter WIDTH=8 )()   input          i_clock,   input          i_sync_resetL,   input  [WIDTH-1:0] i_d,   output reg [WIDTH-1:0] o_q );   always @(posedge i_clock) begin     if (~i_sync_resetL) o_q &lt;= 0;     else o_q &lt;= i_d;   end endmodule</pre>
generic/half_delayer.v	generic/one_cycle_delayer.v

```

module half_cycle_delayer(
    input i_clock,
    input i_async_resetL,
    input i_to_be_delayed_half_cycle,
    output reg o_delayed_half_cycle
);
    always @(negedge i_clock or negedge i_async_resetL) begin
        if (~i_async_resetL) o_delayed_half_cycle <= 0;
        else o_delayed_half_cycle <= i_to_be_delayed_half_cycle;
    end
endmodule

```

```

module one_cycle_delayer(
    input i_clock,
    input i_resetL,
    input i_to_be_delayed_one_cycle,
    output o_delayed_one_cycle
);
    wire w_delayed_half_cycle;
    half_cycle_delayer inst_half_cycle_delayer(
        .i_clock(i_clock),
        .i_async_resetL(i_resetL),
        .i_to_be_delayed_half_cycle(i_to_be_delayed_one_cycle),
        .o_delayed_half_cycle(w_delayed_half_cycle)
    );

    flip_flop_async_reset #(
        .WIDTH(1)
    ) inst_flip_flop_async_reset(
        .i_clock(i_clock),
        .i_async_resetL(i_resetL),
        .i_d(w_delayed_half_cycle),
        .o_q(o_delayed_one_cycle)
    );
endmodule

```

## generic/mux2.v

```

module mux2 #(
    parameter WIDTH=1
)(
    input [WIDTH-1:0] i_d0, i_d1,
    input i_s,
    output [WIDTH-1:0] o_y
);
    assign o_y = i_s ? i_d1 : i_d0;
endmodule

```

## generic/low\_to\_high\_detector.v

```

module low_to_high_detector(
    input i_clock,
    input i_resetL,
    input i_level,
    output o_pulse
);
    localparam [1:0]
        STATEZERO= 2'b00,
        STATEONE = 2'b01,
        STATETHREE=2'b10;

    reg[1:0] r_current_state, r_next_state;
    initial begin
        r_current_state <= STATEZERO;
        r_next_state <= STATEZERO;
    end

    always @(r_current_state, i_level) begin
        r_next_state = r_current_state;
        case(r_current_state)
            STATEZERO:
                if (i_level) r_next_state <= STATEONE;
            STATEONE:
                r_next_state <= STATETHREE;
            STATETHREE:
                if (~i_level) r_next_state <= STATEZERO;
        endcase
    end

    always @(posedge i_clock, negedge i_resetL)
        if (~i_resetL) r_current_state <= STATEZERO;
        else r_current_state <= r_next_state;

    assign o_pulse = (r_current_state == STATEONE);
endmodule

```

## Peripheral

Under this folder is code for peripheral attached to UART transmitter. In this project, four push buttons are used for key a, s, d, w respectively. In the code below, notice the workaround used to send value in case two buttons are pressed simultaneously. For example, when both w and a are pressed, we send “#”. Later on we will write Python driver that will unpack this “#” symbol back into “w” and “a”.

peripheral/multiple\_keypress\_decoder.v

```
module multiple_keypress_encoder #(
    parameter DATA_WIDTH=8
)(
    input i_key_a,
    input i_key_s,
    input i_key_d,
    input i_key_w,
    output reg [DATA_WIDTH-1:0] o_value,
    output o_key_pressed
);

localparam [DATA_WIDTH-1:0] lp_a_value = 'h61;
localparam [DATA_WIDTH-1:0] lp_s_value = 'h73;
localparam [DATA_WIDTH-1:0] lp_d_value = 'h64;
localparam [DATA_WIDTH-1:0] lp_w_value = 'h77;

localparam [DATA_WIDTH-1:0] lp_wa_value = 'h23; //#
localparam [DATA_WIDTH-1:0] lp_wd_value = 'h24; //$
localparam [DATA_WIDTH-1:0] lp_sa_value = 'h25; //%
localparam [DATA_WIDTH-1:0] lp_sd_value = 'h26; //&

always @(i_key_a, i_key_s, i_key_d, i_key_w)
    case({i_key_a, i_key_s, i_key_d, i_key_w})
        4'b0000: o_value <= 0; //null
        4'b1000: o_value <= lp_a_value;
        4'b0100: o_value <= lp_s_value;
        4'b0010: o_value <= lp_d_value;
        4'b0001: o_value <= lp_w_value;
        4'b1001: o_value <= lp_wa_value;
        4'b0011: o_value <= lp_wd_value;
```

```

    4'b1100: o_value <= lp_sa_value;
    4'b0110: o_value <= lp_sd_value;
    default: o_value <= 0; //null
endcase

assign o_key_pressed = i_key_a | i_key_s | i_key_d | i_key_w;
endmodule

```

## Transmitter

Under this folder is code for datapath and control unit of the UART transmitter.

transmitter/transmitter\_shift\_register.v

```

module transmitter_shift_register #(
    parameter DATA_WIDTH = 8
)(
    input i_clock,
    input i_async_resetL,
    input i_shift_en,
    input i_ld_en,
    input [DATA_WIDTH-1:0] i_load,
    output o_shift_out
);
    reg [DATA_WIDTH-1:0] r_internal_load = 0;

    always @(posedge i_clock or negedge i_async_resetL) begin
        r_internal_load = r_internal_load;
        if (~i_async_resetL) r_internal_load <= 0;
        else if (i_ld_en) r_internal_load <= i_load;
        else if (i_shift_en) r_internal_load <= {r_internal_load[0], r_internal_load[DATA_WIDTH-1:1]};
    end

    assign o_shift_out = r_internal_load[0];
endmodule

```

transmitter/transmitter\_shifter.v

```
module transmitter_shifter #(
    parameter DATA_WIDTH = 8
)(
    input i_clock,
    input i_resetL,
    input [DATA_WIDTH-1:0] i_value,
    input i_state_is_START,
    input i_state_is_DATA,
    input i_equal,
    output o_TX
);
    wire w_state_is_START_delayed_half_cycle;
    half_cycle_delayer inst_half_cycle_delayer_for_START(
        .i_clock(i_clock),
        .i_async_resetL(i_resetL),
        .i_to_be_delayed_half_cycle(i_state_is_START),
        .o_delayed_half_cycle(w_state_is_START_delayed_half_cycle)
    );

    wire w_state_is_DATA_delayed_half_cycle;
    half_cycle_delayer inst_half_cycle_delayer_for_DATA(
        .i_clock(i_clock),
        .i_async_resetL(i_resetL),
        .i_to_be_delayed_half_cycle(i_state_is_DATA),
        .o_delayed_half_cycle(w_state_is_DATA_delayed_half_cycle)
    );

    wire w_equal_delayed_half_cycle;
    half_cycle_delayer inst_half_cycle_delayer_for_equal(
        .i_clock(i_clock),
        .i_async_resetL(i_resetL),
        .i_to_be_delayed_half_cycle(i_equal),
        .o_delayed_half_cycle(w_equal_delayed_half_cycle)
    );
endmodule
```

```

wire w_state_is_START_pulse;
low_to_high_detector inst_low_to_high_detector(
    .i_clock(i_clock),
    .i_resetL(i_resetL),
    .i_level(i_state_is_START),
    .o_pulse(w_state_is_START_pulse)
);

wire w_equal_delayed_one_cycle;
one_cycle_delayer inst_one_cycle_delayer_for_equal(
    .i_clock(i_clock),
    .i_resetL(i_resetL),
    .i_to_be_delayed_one_cycle(i_equal),
    .o_delayed_one_cycle(w_equal_delayed_one_cycle)
);

wire w_state_is_START_pulse_delayed_half_cycle;
half_cycle_delayer inst_half_cycle_delayer_for_START_pulse(
    .i_clock(i_clock),
    .i_async_resetL(i_resetL),
    .i_to_be_delayed_half_cycle(w_state_is_START_pulse),
    .o_delayed_half_cycle(w_state_is_START_pulse_delayed_half_cycle)
);

wire w_clock = w_state_is_START_pulse_delayed_half_cycle | w_equal_delayed_one_cycle;
wire w_shift_en = w_state_is_DATA_delayed_half_cycle;
wire w_shift_out;
transmitter_shift_register #(
    .DATA_WIDTH(DATA_WIDTH)
) inst_transmitter_shift_register(
    .i_clock(w_clock),
    .i_async_resetL(i_resetL),
    .i_shift_en(w_shift_en),
    .i_ld_en(w_state_is_START_pulse),
    .i_load(i_value),
    .o_shift_out(w_shift_out)
);

```

```

);

localparam [0:0] lp_one  = 1;
localparam [0:0] lp_zero = 0;
wire w_mux_stage1;
mux2 #(
    .WIDTH(1)
) inst_mux2_stage1(
    .i_d0(lp_one),
    .i_d1(lp_zero),
    .i_s(i_state_is_START),
    .o_y(w_mux_stage1)
);

mux2 #(
    .WIDTH(1)
) inst_mux2_stage2(
    .i_d0(w_mux_stage1),
    .i_d1(w_shift_out),
    .i_s(i_state_is_DATA),
    .o_y(o_TX)
);
endmodule

```

transmitter/transmitter\_bit\_counter.v

```

module transmitter_bit_counter #(
    parameter BIT_COUNTER_WIDTH=3,
    parameter DATA_WIDTH=8
)(
    input i_clock,
    input i_resetL,
    input i_state_is_START,
    input i_state_is_DATA,
    input i_equal,
    output o_equal_MSB

```



```

);
wire w_equal_delayed_one_cycle;
one_cycle_delayer inst_one_cycle_delayer_for_equal(
    .i_clock(i_clock),
    .i_resetL(i_resetL),
    .i_to_be_delayed_one_cycle(i_equal),
    .o_delayed_one_cycle(w_equal_delayed_one_cycle)
);

wire [BIT_COUNTER_WIDTH-1:0] w_adder;
wire [BIT_COUNTER_WIDTH-1:0] w_bit_index;
wire w_resetL;
flip_flop_async_reset #(
    .WIDTH(BIT_COUNTER_WIDTH)
) inst_flip_flop_async_reset(
    .i_clock(w_equal_delayed_one_cycle),
    .i_async_resetL(w_resetL),
    .i_d(w_adder),
    .o_q(w_bit_index)
);

localparam [BIT_COUNTER_WIDTH-1:0] lp_DATA_WIDTH_minus_one = DATA_WIDTH-1;
equal_comparator #(
    .WIDTH(BIT_COUNTER_WIDTH)
) inst_equal_comparator(
    .i_a(w_bit_index),
    .i_b(lp_DATA_WIDTH_minus_one),
    .o_c(o_equal_MSB)
);

localparam [BIT_COUNTER_WIDTH-1:0] lp_one = 1;
adder #(
    .WIDTH(BIT_COUNTER_WIDTH)
) inst_adder(
    .i_a(w_bit_index),
    .i_b(lp_one),

```

```

        .o_y(w_adder)
    );

    wire w_state_is_START_delayed_half_cycle;
    half_cycle_delayer inst_half_cycle_delayer_for_START(
        .i_clock(i_clock),
        .i_async_resetL(i_resetL),
        .i_to_be_delayed_half_cycle(i_state_is_START),
        .o_delayed_half_cycle(w_state_is_START_delayed_half_cycle)
    );
    assign w_resetL = i_resetL & (~w_state_is_START_delayed_half_cycle & i_state_is_DATA);
endmodule

```

#### transmitter/transmitter\_clock\_counter.v

```

module transmitter_clock_counter #(
    parameter CLOCK_COUNTER_WIDTH=21,
    parameter CLOCKS_PER_BIT=434,
    parameter CLOCKS_FOR_STOP=1612903
)(
    input i_clock,
    input i_resetL,
    input i_state_is_START,
    input i_state_is_DATA,
    input i_state_is_STOP,
    output o_equal
);
    wire [CLOCK_COUNTER_WIDTH-1:0] w_adder;
    wire [CLOCK_COUNTER_WIDTH-1:0] w_clock_count;
    wire w_resetL;
    flip_flop_sync_reset #(
        .WIDTH(CLOCK_COUNTER_WIDTH)
    ) inst_flip_flop_sync_reset(
        .i_clock(i_clock),
        .i_sync_resetL(w_resetL),
        .i_d(w_adder),

```

```

        .o_q(w_clock_count)
    );

    localparam [CLOCK_COUNTER_WIDTH-1:0] lp_one = 1;
    adder #(
        .WIDTH(CLOCK_COUNTER_WIDTH)
    ) inst_adder(
        .i_a(w_clock_count),
        .i_b(lp_one),
        .o_y(w_adder)
    );

    wire w_equal_CLOCKS_PER_BIT_minus_one;
    localparam [CLOCK_COUNTER_WIDTH-1:0] lp_CLOCKS_PER_BIT_minus_one = CLOCKS_PER_BIT-1;
    equal_comparator #(
        .WIDTH(CLOCK_COUNTER_WIDTH)
    ) inst_equal_comparator_for_CLOCKS_PER_BIT_minus_one(
        .i_a(w_clock_count),
        .i_b(lp_CLOCKS_PER_BIT_minus_one),
        .o_c(w_equal_CLOCKS_PER_BIT_minus_one)
    );

    wire w_equal_CLOCKS_FOR_STOP_minus_one;
    localparam [CLOCK_COUNTER_WIDTH-1:0] lp_CLOCKS_FOR_STOP_minus_one = CLOCKS_FOR_STOP-1;
    equal_comparator #(
        .WIDTH(CLOCK_COUNTER_WIDTH)
    ) inst_equal_comparator_for_CLOCKS_FOR_STOP_minus_one(
        .i_a(w_clock_count),
        .i_b(lp_CLOCKS_FOR_STOP_minus_one),
        .o_c(w_equal_CLOCKS_FOR_STOP_minus_one)
    );

    mux2 #(
        .WIDTH(1)
    ) inst_mux2_for_equal(
        .i_d0(w_equal_CLOCKS_PER_BIT_minus_one),

```

```

        .i_d1(w_equal_CLOCKS_FOR_STOP_minus_one),
        .i_s(i_state_is_STOP),
        .o_y(o_equal)
    );

    wire w_equal_delayed_half_cycle;
    half_cycle_delayer inst_half_cycle_delayer_for_equal(
        .i_clock(i_clock),
        .i_async_resetL(i_resetL),
        .i_to_be_delayed_half_cycle(o_equal),
        .o_delayed_half_cycle(w_equal_delayed_half_cycle)
    );

    wire w_state_is_START_delayed_half_cycle;
    half_cycle_delayer inst_half_cycle_delayer_for_START(
        .i_clock(i_clock),
        .i_async_resetL(i_resetL),
        .i_to_be_delayed_half_cycle(i_state_is_START),
        .o_delayed_half_cycle(w_state_is_START_delayed_half_cycle)
    );

    assign w_resetL = i_resetL &
        ~w_equal_delayed_half_cycle &
        (w_state_is_START_delayed_half_cycle | i_state_is_DATA | i_state_is_STOP);
endmodule

```

#### transmitter/transmitter\_datapath.v

```

module transmitter_datapath #(
    parameter DATA_WIDTH = 8,
    parameter BIT_COUNTER_WIDTH=3,
    parameter CLOCK_COUNTER_WIDTH=21,
    parameter CLOCKS_PER_BIT=434,
    parameter CLOCKS_FOR_STOP=1608997
)(
    input i_clock,

```

```

input i_resetL,
input [DATA_WIDTH-1:0] i_value,
//input i_state_is_IDLE,
input i_state_is_START,
input i_state_is_DATA,
input i_state_is_STOP,
output o_equal,
output o_equal_MSB,
output o_TX
);
transmitter_shifter #(
    .DATA_WIDTH(DATA_WIDTH)
) inst_transmitter_shifter(
    .i_clock(i_clock),
    .i_resetL(i_resetL),
    .i_value(i_value),
    .i_state_is_START(i_state_is_START),
    .i_state_is_DATA(i_state_is_DATA),
    .i_equal(o_equal),
    .o_TX(o_TX)
);

transmitter_bit_counter #(
    .BIT_COUNTER_WIDTH(BIT_COUNTER_WIDTH),
    .DATA_WIDTH(DATA_WIDTH)
) inst_transmitter_bit_counter(
    .i_clock(i_clock),
    .i_resetL(i_resetL),
    .i_state_is_START(i_state_is_START),
    .i_state_is_DATA(i_state_is_DATA),
    .i_equal(o_equal),
    .o_equal_MSB(o_equal_MSB)
);

transmitter_clock_counter #(
    .CLOCK_COUNTER_WIDTH(CLOCK_COUNTER_WIDTH),

```

```

        .CLOCKS_PER_BIT(CLOCKS_PER_BIT),
        .CLOCKS_FOR_STOP(CLOCKS_FOR_STOP)
    ) inst_transmitter_clock_counter(
        .i_clock(i_clock),
        .i_resetL(i_resetL),
        .i_state_is_START(i_state_is_START),
        .i_state_is_DATA(i_state_is_DATA),
        .i_state_is_STOP(i_state_is_STOP),
        .o_equal(o_equal)
    );
endmodule

```

#### transmitter/transmitter\_control.v

```

module transmitter_control(
    input  i_clock,
    input  i_resetL,
    input  i_key_pressed,
    input  i_equal,
    input  i_equal_MSB,
    output o_state_is_START,
    output o_state_is_DATA,
    output o_state_is_STOP
);
    localparam [1:0]
        IDLE = 2'b00,
        START = 2'b01,
        DATA = 2'b10,
        STOP = 2'b11;

    reg[1:0] r_current_state, r_next_state;

    wire w_equal_delayed_half_cycle;
    half_cycle_delayer inst_half_cycle_delayer_for_equal(
        .i_clock(i_clock),
        .i_async_resetL(i_resetL),

```

```

.i_to_be_delayed_half_cycle(i_equal),
.o_delayed_half_cycle(w_equal_delayed_half_cycle)
);
wire w_equal_MSB_delayed_half_cycle;
half_cycle_delayer inst_half_cycle_delayer_for_equal_MSB(
.i_clock(i_clock),
.i_async_resetL(i_resetL),
.i_to_be_delayed_half_cycle(i_equal_MSB),
.o_delayed_half_cycle(w_equal_MSB_delayed_half_cycle)
);

always @(r_current_state, i_key_pressed,
w_equal_delayed_half_cycle,
w_equal_MSB_delayed_half_cycle) begin
r_next_state = r_current_state;
case(r_current_state)
IDLE:
if (~i_key_pressed) r_next_state <= IDLE;
else if (i_key_pressed) r_next_state <= START;
START:
if (~w_equal_delayed_half_cycle) r_next_state <= START;
else if (w_equal_delayed_half_cycle) r_next_state <= DATA;
DATA:
if (w_equal_delayed_half_cycle & ~w_equal_MSB_delayed_half_cycle) r_next_state <= DATA;
else if (w_equal_delayed_half_cycle & w_equal_MSB_delayed_half_cycle) r_next_state <= STOP;
STOP:
if (~w_equal_delayed_half_cycle) r_next_state <= STOP;
else if (w_equal_delayed_half_cycle & i_key_pressed) r_next_state <= START;
else if (w_equal_delayed_half_cycle & ~i_key_pressed) r_next_state <= IDLE;
endcase
end

always @(posedge i_clock, negedge i_resetL)
if (~i_resetL) r_current_state <= IDLE;
else r_current_state <= r_next_state;

```

```
assign o_state_is_START = (r_current_state == START);
assign o_state_is_DATA = (r_current_state == DATA);
assign o_state_is_STOP = (r_current_state == STOP);
endmodule
```

#### transmitter/UART\_transmitter.v

```
module UART_transmitter #(
    parameter DATA_WIDTH = 8,
    parameter BIT_COUNTER_WIDTH=3,
    parameter CLOCK_COUNTER_WIDTH=21,
    parameter CLOCKS_PER_BIT=434,
    parameter CLOCKS_FOR_STOP=1608997
)(
    input i_clock,
    input i_resetL,
    input i_key_pressed,
    input [DATA_WIDTH-1:0] i_value,
    output o_TX
);
    wire w_state_is_START;
    wire w_state_is_DATA;
    wire w_state_is_STOP;
    wire w_equal;
    wire w_equal_MSB;

    transmitter_control inst_transmitter_control(
        .i_clock(i_clock),
        .i_resetL(i_resetL),
        .i_key_pressed(i_key_pressed),
        .i_equal(w_equal),
        .i_equal_MSB(w_equal_MSB),
        .o_state_is_START(w_state_is_START),
        .o_state_is_DATA(w_state_is_DATA),
        .o_state_is_STOP(w_state_is_STOP)
    );
```



```

transmitter_datapath #(
    .DATA_WIDTH(DATA_WIDTH),
    .BIT_COUNTER_WIDTH(BIT_COUNTER_WIDTH),
    .CLOCK_COUNTER_WIDTH(CLOCK_COUNTER_WIDTH),
    .CLOCKS_PER_BIT(CLOCKS_PER_BIT),
    .CLOCKS_FOR_STOP(CLOCKS_FOR_STOP)
)
inst_transmitter_datapath(
    .i_clock(i_clock),
    .i_resetL(i_resetL),
    .i_value(i_value),
    .i_state_is_START(w_state_is_START),
    .i_state_is_DATA(w_state_is_DATA),
    .i_state_is_STOP(w_state_is_STOP),
    .o_equal(w_equal),
    .o_equal_MSB(w_equal_MSB),
    .o_TX(o_TX)
);
endmodule

```

transmitter/UART\_transmitter\_with\_peripheral.v

```

module UART_transmitter_with_peripheral #(
    parameter DATA_WIDTH = 8,
    parameter BIT_COUNTER_WIDTH=3,
    parameter CLOCK_COUNTER_WIDTH=21,
    parameter CLOCKS_PER_BIT=434,
    parameter CLOCKS_FOR_STOP=1608997
)(
    input i_clock,
    input i_resetL,
    input i_key_a,
    input i_key_s,
    input i_key_d,
    input i_key_w,

```

```

    output o_TX
);
wire [DATA_WIDTH-1:0] w_value;
wire w_key_pressed;
multiple_keypress_encoder #(
    .DATA_WIDTH(DATA_WIDTH)
) inst_keypress_encoder(
    .i_key_a(i_key_a),
    .i_key_s(i_key_s),
    .i_key_d(i_key_d),
    .i_key_w(i_key_w),
    .o_value(w_value),
    .o_key_pressed(w_key_pressed)
);

UART_transmitter #(
    .CLOCK_COUNTER_WIDTH(CLOCK_COUNTER_WIDTH),
    .BIT_COUNTER_WIDTH(BIT_COUNTER_WIDTH),
    .DATA_WIDTH(DATA_WIDTH),
    .CLOCKS_PER_BIT(CLOCKS_PER_BIT),
    .CLOCKS_FOR_STOP(CLOCKS_FOR_STOP)
) inst_UART_transmitter(
    .i_clock(i_clock),
    .i_resetL(i_resetL),
    .i_key_pressed(w_key_pressed),
    .i_value(w_value),
    .o_TX(o_TX)
);
endmodule

```

# UART Transmitter Game Controller Driver

Before we can use the project to play racing game in computer, we need to write a driver that understands the meaning of the data sent from our UART transmitter and perform actions like keydown and keyup accordingly.

UART\_game\_controller\_driver.py

```
import pyautogui
import serial
ser = serial.Serial(port='COM3',baudrate=115200,timeout=0.2)

last_key = b''
last_key_combination = []
current_key = b''
current_key_combination = []

def unpack(current_key):
    if current_key == b'#':
        current_key_combination = [b'w', b'a']
    elif current_key == b'$':
        current_key_combination = [b'w', b'd']
    elif current_key == b'%':
        current_key_combination = [b's', b'a']
    elif current_key == b'&':
        current_key_combination = [b's', b'd']
    else:
        current_key_combination = [current_key]
    return current_key_combination

while True:
    try:
        current_key = ser.read()
        print(current_key)
        if current_key != last_key:
            current_key_combination = unpack(current_key)
            # First comparison:
```

```
# compare last key combination with current key combination
# if last key is null then nothing to keyUp
if last_key != b'':
    for key in last_key_combination:
        if key not in current_key_combination:
            print('key not in current_key_combination', key)
            pyautogui.keyUp(key.decode("utf-8"))
            last_key_combination.remove(key)
        else:
            # remove equal because redundant in second comparison
            current_key_combination.remove(key)
# second comparison:
# compare current key combination with last key combination
# if current key is null then nothing to keyDown
if current_key != b'':
    for key in current_key_combination:
        if key not in last_key_combination:
            pyautogui.keyDown(key.decode("utf-8"))
            last_key_combination.append(key)
# finally save current key as last key for future comparison
last_key = current_key
except:
    ser.close()
    break
```

# UART Transmitter Testbench and Simulation

UART\_transmitter\_with\_peripheral\_tb.v

```
`timescale 1ns/100ps

module UART_transmitter_with_peripheral_tb();
    parameter DATA_WIDTH=8;
    parameter BIT_COUNTER_WIDTH=3;
    parameter CLOCK_COUNTER_WIDTH=21;
    parameter CLOCKS_PER_BIT=434;
    parameter CLOCKS_FOR_STOP=1608997;

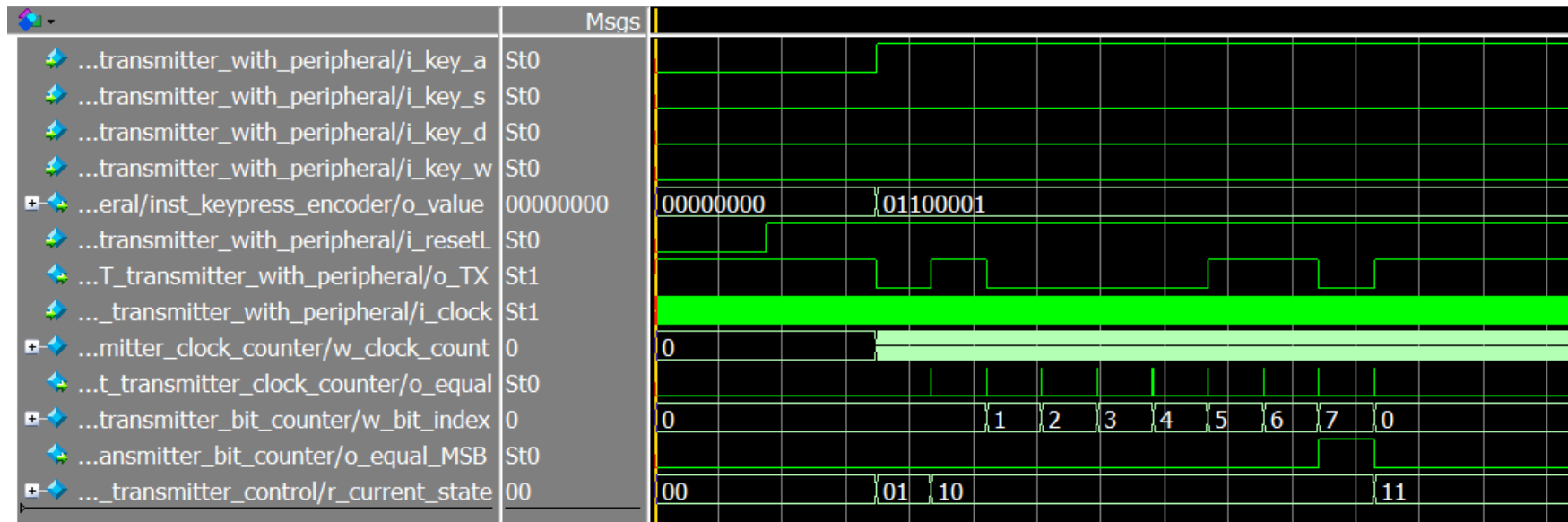
    reg r_clock;
    initial begin
        r_clock = 1;
        forever begin
            #0.5 r_clock = ~r_clock;
        end
    end

    reg r_resetL;
    reg r_key_a;
    reg r_key_s;
    reg r_key_d;
    reg r_key_w;

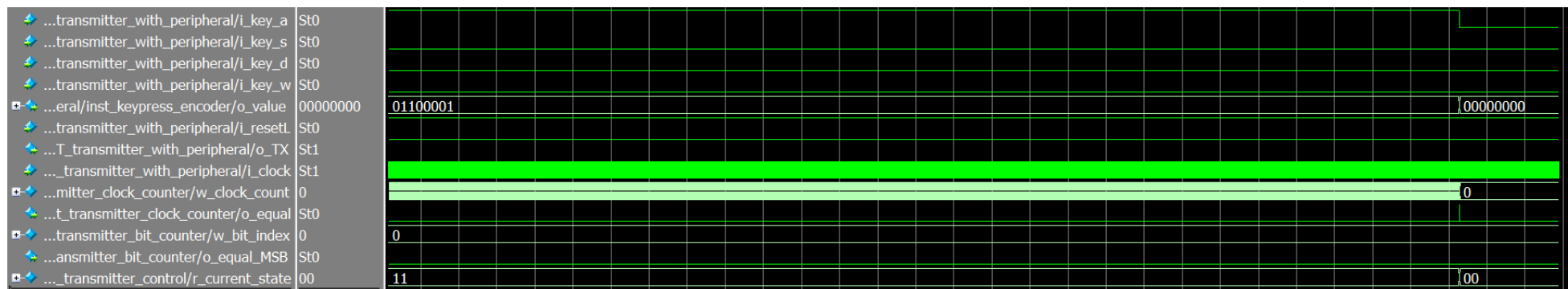
    wire w_TX;
    UART_transmitter_with_peripheral #(
        .DATA_WIDTH(DATA_WIDTH),
        .BIT_COUNTER_WIDTH(BIT_COUNTER_WIDTH),
        .CLOCK_COUNTER_WIDTH(CLOCK_COUNTER_WIDTH),
        .CLOCKS_PER_BIT(CLOCKS_PER_BIT),
        .CLOCKS_FOR_STOP(CLOCKS_FOR_STOP)
    ) inst_UART_transmitter_with_peripheral(
        .i_clock(r_clock),
        .i_resetL(r_resetL),
```

```
.i_key_a(r_key_a),  
.i_key_s(r_key_s),  
.i_key_d(r_key_d),  
.i_key_w(r_key_w),  
.o_TX(w_TX)  
);  
  
initial begin  
    r_resetL=0; r_key_a=0; r_key_s=0; r_key_d=0; r_key_w=0; #(CLOCKS_PER_BIT * 2);  
    r_resetL=1; #(CLOCKS_PER_BIT * 2);  
    r_key_a=1; #(CLOCKS_PER_BIT * (DATA_WIDTH + 1) + CLOCKS_FOR_STOP);  
    r_key_a=0; #(CLOCKS_PER_BIT * 2);  
    r_key_w=0; #(CLOCKS_PER_BIT); $stop;  
end  
endmodule
```

The simulation waveform is the same as the timing diagram we discussed earlier on except with added peripheral (4 push buttons named i\_key\_a, i\_key\_s, i\_key\_d, and i\_key\_w in the simulation waveform). When i\_key\_a is pressed, r\_current\_state becomes 01 (START) and o\_TX changes from the initial logic 1 to logic 0. After sending this START bit, r\_current\_state transition to DATA state (r\_current\_state=10). Looking when r\_current\_state=10, and looking at o\_TX, we can see the data is being sent correctly (the data to be sent is o\_value = 01100001). The o\_TX starts sending serially by sending out the LSB first and ends with MSB. After DATA state, we have quite a long STOP bit, and this is intentional as discussed before to limit repeat rate. Finally since i\_key\_a is released, at the end the r\_current\_state simply go back to IDLE state and wait for further key press.



continued...



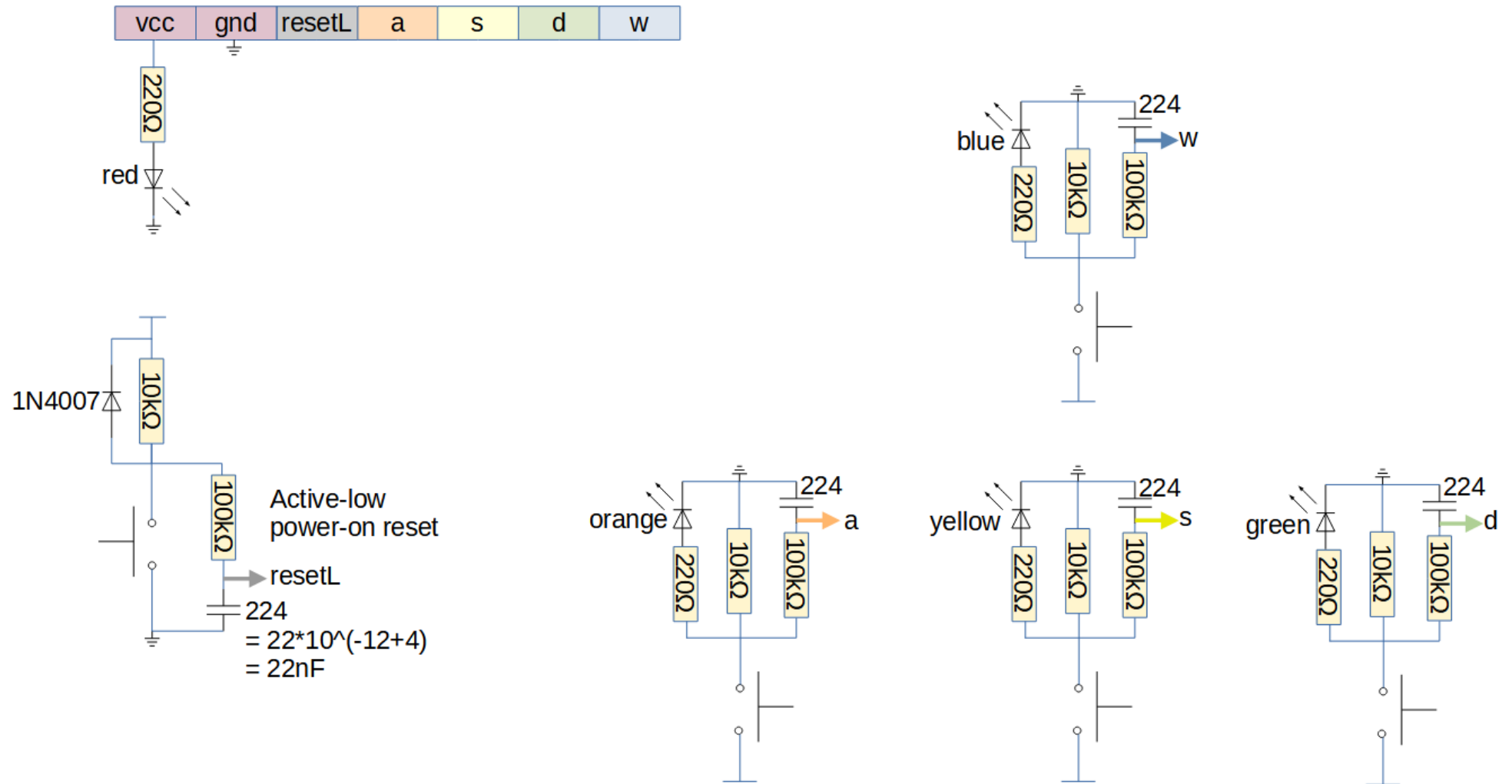
# UART Transmitter Hardware

For the demo, please watch the YouTube video:

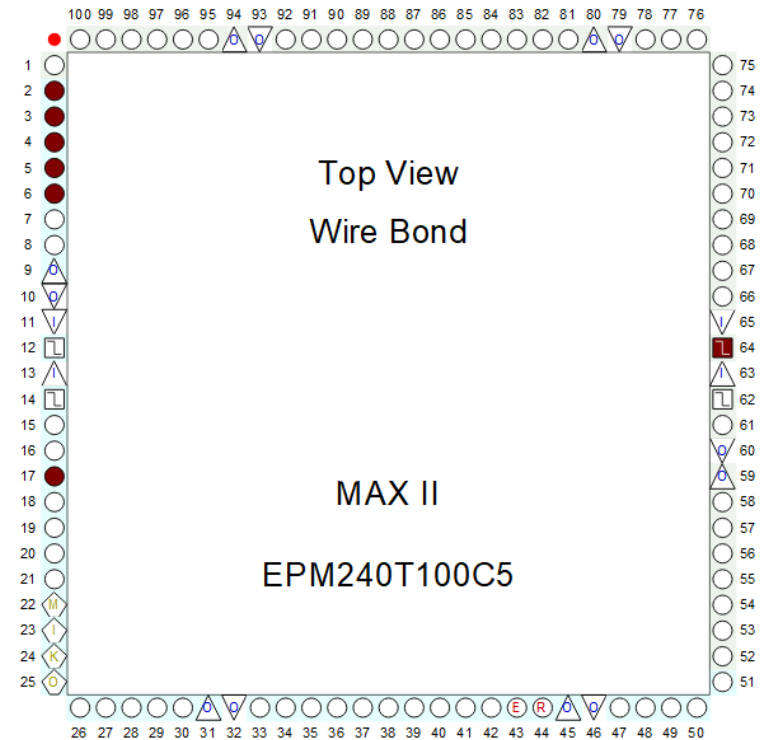
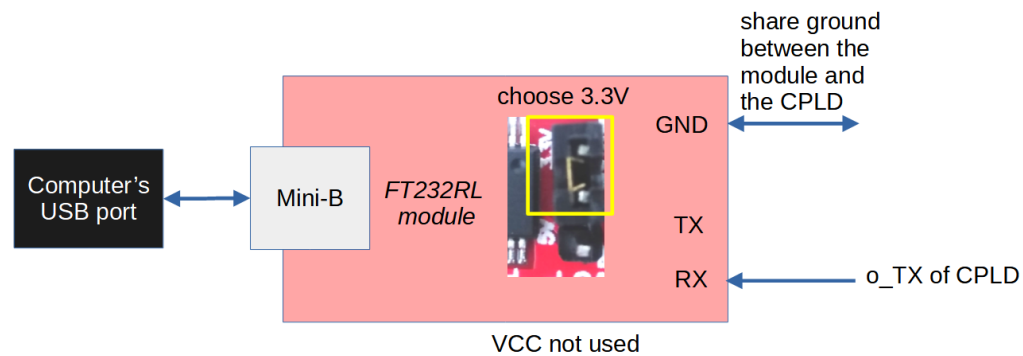
<https://youtu.be/VMD-bAN7MmI?t=3686>

Code for download:

[https://github.com/endeneer/CPLD\\_UART\\_Datapath\\_and\\_Control\\_Unit](https://github.com/endeneer/CPLD_UART_Datapath_and_Control_Unit)







Node Name	Direction	Location	I/O Bank	Fitter Location	I/O Standard	Reserved	Current Strength	Strict Preservation
<input type="checkbox"/> i_clock	Input	PIN_64	2	PIN_64	3.3-V LVTTTL		16mA (default)	
<input type="checkbox"/> i_key_a	Input	PIN_2	1	PIN_2	3.3V Schmitt Trigger Input		16mA (default)	
<input type="checkbox"/> i_key_d	Input	PIN_4	1	PIN_4	3.3V Schmitt Trigger Input		16mA (default)	
<input type="checkbox"/> i_key_s	Input	PIN_3	1	PIN_3	3.3V Schmitt Trigger Input		16mA (default)	
<input type="checkbox"/> i_key_w	Input	PIN_5	1	PIN_5	3.3V Schmitt Trigger Input		16mA (default)	
<input type="checkbox"/> i_resetL	Input	PIN_6	1	PIN_6	3.3V Schmitt Trigger Input		16mA (default)	
<input type="checkbox"/> o_TX	Output	PIN_17	1	PIN_17	3.3-V LVTTTL		16mA (default)	
<<new node>>								