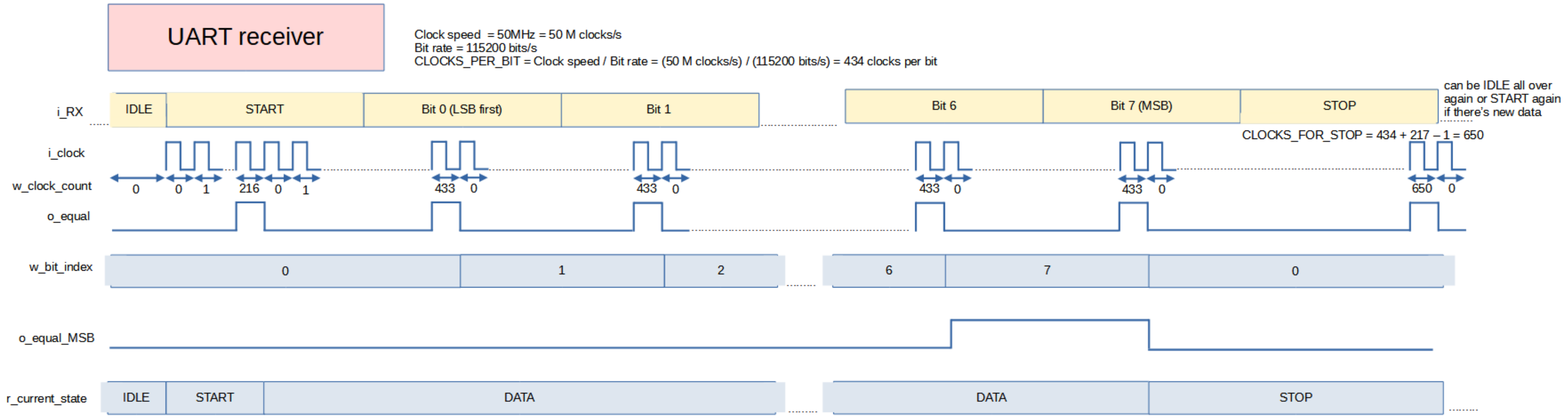
UART Receiver Design Ideas

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, CLOCKS_PER_BIT = clock speed / bit rate = (50 M clocks/s) / (115200 bits/s) ≈ 434 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.


UART receiver
Clock speed = 50MHz = 50 M clocks/s
Bit rate = 115200 bits/s
CLOCKS_PER_BIT = Clock speed / Bit rate = (50 M clocks/s) / (115200 bits/s) = 434 clocks per bit
i_RX
IDLE
START
Bit 0 (LSB first)
Bit 1
Bit 6
Bit 7 (MSB)
STOP
can be IDLE all over again or START again if there's new data
CLOCKS_FOR_STOP = 434 + 217 – 1 = 650
i_clock
w_clock_count
0
0
1
216
0
1
433
0
433
0
433
0
433
0
650
0
o_equal
w_bit_index
0
1
2
6
7
0
o_equal_MSB
r_current_state
IDLE
START
DATA
DATA
STOP

Initially, i_RX = 1 during IDLE. When i_RX = 0, we know that it can be a START bit, so, for finite state machine (FSM), r_current_state is changed to START. At this time, we activate clock counter to start counting (labelled w_clock_count in the diagram). Take note that w_clock_count is zero indexed, so, later when you see some values that are minus one, do not freak out, if you are familiar with programming language that use zero-indexed style (sorry to MATLAB users!), you should already know why we need minus one.

Anyway let’s get back to the discussion. When w_clock_count reaches CLOCK_PER_BIT/2 – 1 = 434/2 – 1 = 216, we can output a pulse (o_equal) to signify that half of the START bit is about to be elapsed (the next posedge of clock is where we are at half of the START bit). This pulse is very useful, we can delayed it by half cycle to reset clock count back to zero at next posedge, we can delayed it by one cycle as a posedge to instruct shift register to shift in the i_RX bit and increment bit index (like what happens at middle of Bit 0 of i_RX, where the w_bit_index is incremented by 1), more on these in the next paragraph.

Let’s get the discussion back to the middle of START bit. After the middle of START bit, r_current_state of FSM is set to DATA, and we continue counting for a length of CLOCKS_PER_BIT (and not CLOCK_PER_BIT/2 – 1 like during r_current_state = START). So, we see that r_current state can be used to select how long to elapse before outputting o_equal. When w_clock_count reaches CLOCKS_PER_BIT – 1 = 434-1 = 433, we again output a pulse (o_equal). The posedge right after w_clock_count = 433 is when we are at halfway of Bit 0. Therefore, we delay the pulse by one cycle as a posedge for shift register to shift in i_RX (i.e. to sample Bit 0) and increment w_bit_index by 1. w_clock_count is again reset back to zero, and the process continues.

After sampling Bit 6, w_bit_index is incremented to 7, which is equal to DATA_WIDTH – 1 = 8-1. Since this is the last data bit of the packet (Bit 7, the MSB), we can output o_equal_MSB = 1 to tell that we are at the last data bit. This o_equal_MSB is useful to tell finite state machine to change r_current_state from DATA to STOP later on.

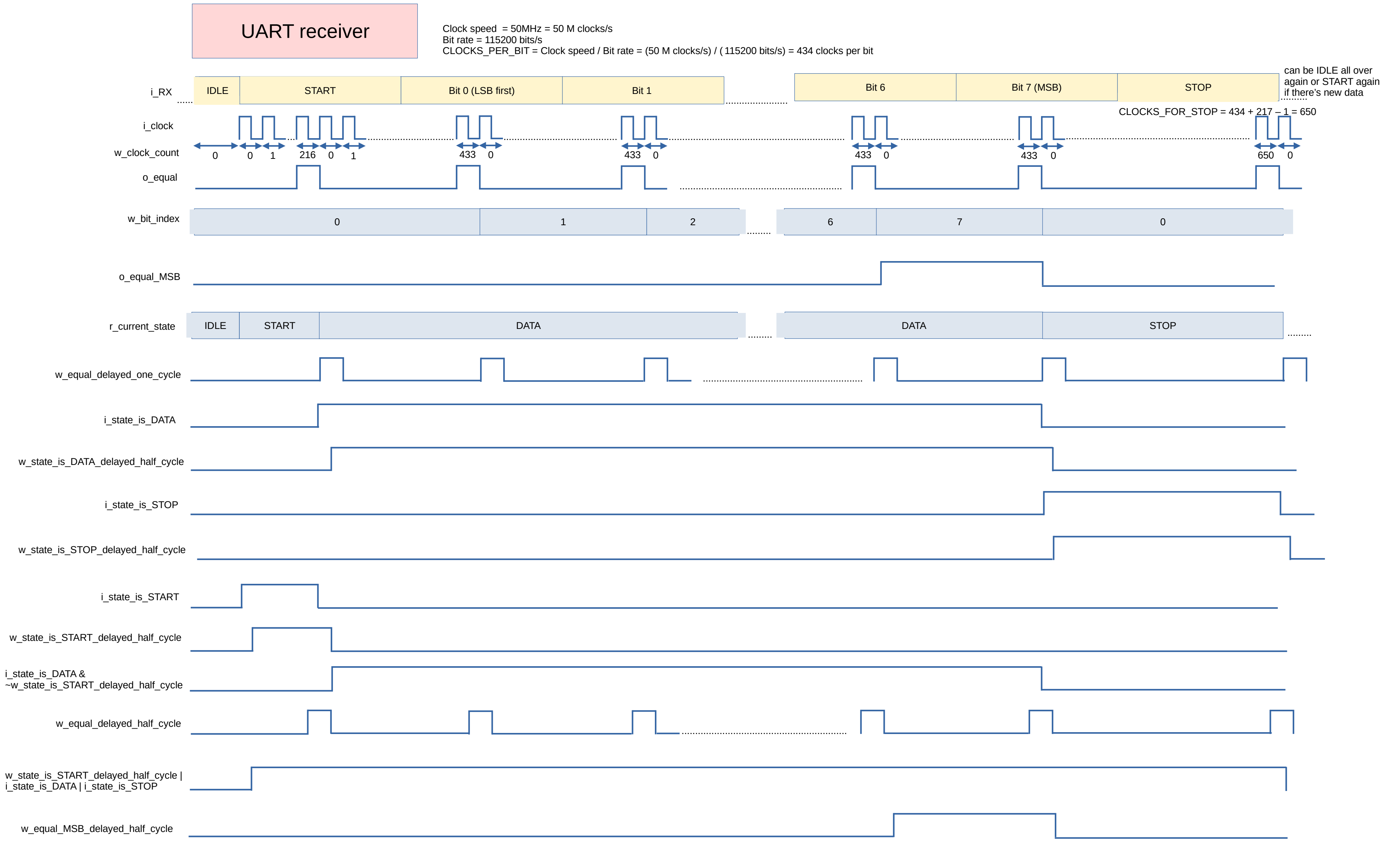
When Bit 7 has been sampled at middle of Bit 7, r_current_state transitions into STOP state. As we discussed just now, r_current state can be used to select how long to elapse before outputting o_equal. When we are at middle of Bit 7, we need one and a half of CLOCKS_PER_BIT to complete the packet, therefore, we can use r_current_state=STOP to tell the clock counter to keep counting for CLOCKS_FOR_STOP = 1.5*CLOCKS_PER_BIT = 1.5*434 = 651. Since we are using zero-indexed counting, the w_clock_count will reach maximum value of CLOCKS_FOR_STOP – 1 = 651-1=650.

After w_clock_count reaches 650, it will again be reset back to zero, we have just completed receiving one packet! After this, there are two possibilities: 1) In case the sender sends another packet , then we will have START bit again and our receiver will continue the counting and sampling process. 2) In case the sender doesn’t send another packet, then receiver is at rest at IDLE state again. During IDLE state, there is no counting and sampling the bits, the clock counter and bit index counter just relax and wait for any future START bit.

With the thinking process so far, we can see that our UART receiver design should consist of four things
1) receiver_shifter.v: To store the sampled bits (received data) from i_RX. In this project, the received data will be displayed on 2 7segment displays.
2) receiver_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) receiver_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 receiver_shifter should sample the bit data, tell when bit counter should increment etc.
4) receiver_control.v: The FSM that will tell r_current_state. r_current state can be used to disable shifting (and hence sampling) of receiver_shifter, tell when bit_counter and clock_counter should not be counting etc.

UART Receiver Datapath Design

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



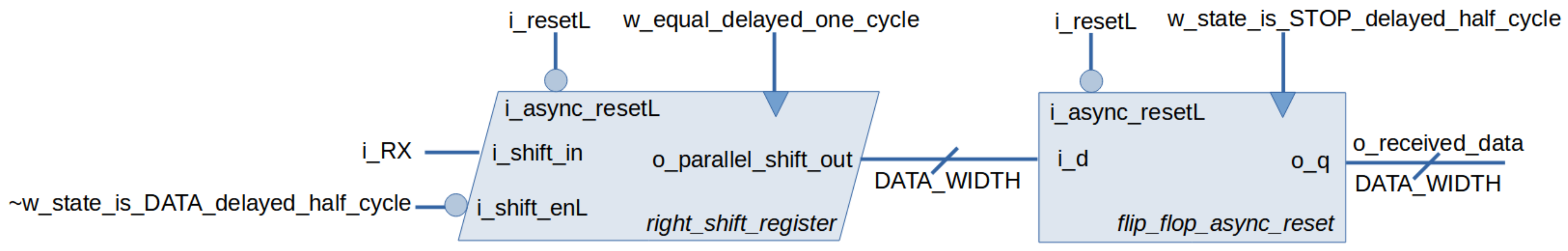
Clock speed = 50MHz = 50 M clocks/s
Bit rate = 115200 bits/s
CLOCKS_PER_BIT = Clock speed / Bit rate = (50 M clocks/s) / (115200 bits/s) = 434 clocks per bit

CLOCKS_FOR_STOP = 434 + 217 – 1 = 650

can be IDLE all over again or START again if there's new data

UART receiver
i_RX
IDLE
START
Bit 0 (LSB first)
Bit 1
Bit 6
Bit 7 (MSB)
STOP
i_clock
w_clock_count
0
0
1
216
0
1
433
0
433
0
433
0
433
0
650
0
o_equal
w_bit_index
0
1
2
6
7
0
o_equal_MSB
r_current_state
IDLE
START
DATA
DATA
STOP
w_equal_delayed_one_cycle
i_state_is_DATA
w_state_is_DATA_delayed_half_cycle
i_state_is_STOP
w_state_is_STOP_delayed_half_cycle
i_state_is_START
w_state_is_START_delayed_half_cycle
i_state_is_DATA & ~w_state_is_START_delayed_half_cycle
w_equal_delayed_half_cycle
w_state_is_START_delayed_half_cycle | i_state_is_DATA | i_state_is_STOP
w_equal_MSB_delayed_half_cycle

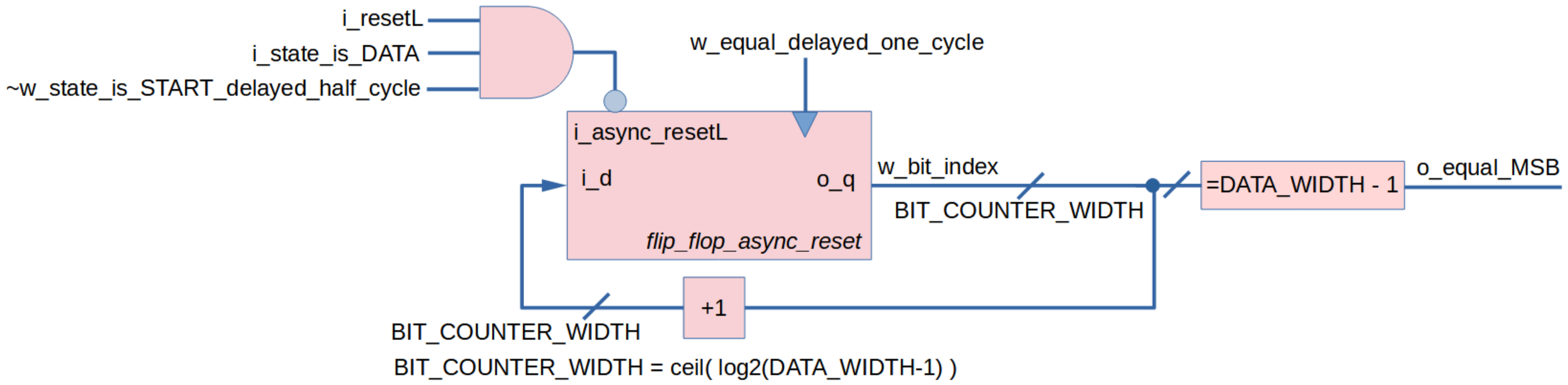
receiver_shifter.v



The shifter consists of a right shift register and a flip flop with asynchronouns reset. Let’s first look at the right shift register. It is a generic serial-in-parallel-out (SIPO) right shift register with active low asynchronous reset and active low shift enable. The serial shift in (i_shift_in) is fed with i_RX, so that when the right shift register shifts, we are sampling i_RX. The posedge to instruct it to shift is obtained from w_equal_delayed_one_cycle (that is, o_equal delayed one cycle, as discussed in the timing diagram just now). On first thought, it seems like we should disable shift when the current state is not DATA. But on closer inspection on timing diagram, we need to delayed by half cycle, this is to prevent w_equal_delayed_one_cycle from causing unwanted shifting. Also, this ensures no hold time violation when sampling Bit 7. So, i_shift_enL = ~w_state_is_DATA_delayed_half_cycle.

Lastly, since o_paralleL_shift_out keeps changing when shifting, we need another flip flop to store the final received data. In other words, o_received_data changes only when data is received completely. The right time to store o_parallel_shift_out into the flip flop is at posedge of w_state_is_STOP_delayed_half_cycle (at posedge of i_state_is_STOP, the right_shift_register is sampling MSB data bit, so we have to delay the posedge by half cycle to prevent storing data when right_shifting_register is shifting).

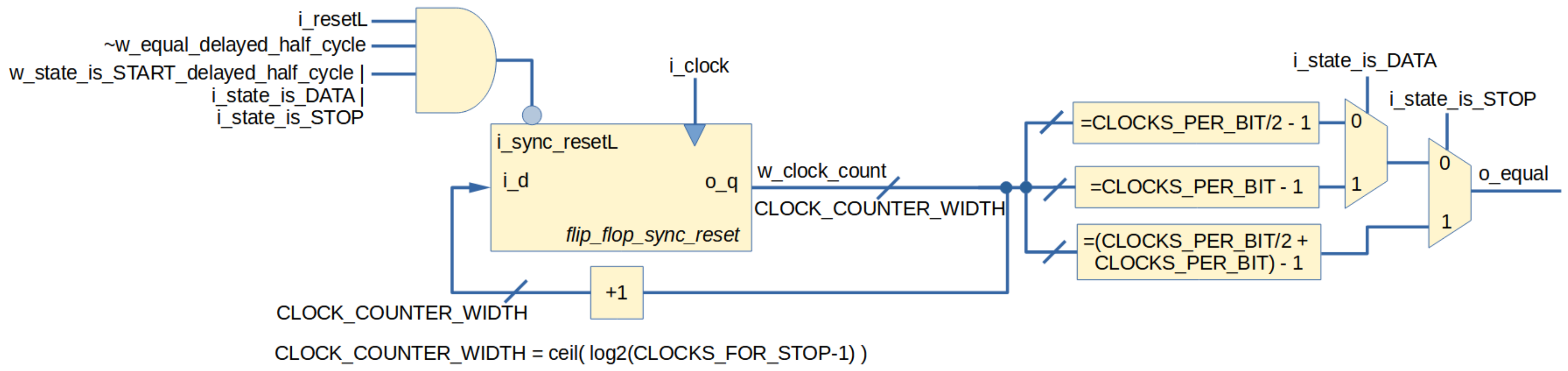
receiver_bit_counter.v



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, BIT_COUNTER_WIDTH = ceil(log2(DATA_WIDTH-1)) = ceil(log2(8-1)) = 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 middle of 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.

receiver_clock_counter.v



The clock counter consists of a flip flop with active low synchronous reset, adder to increment w_clock_count by 1, three comparators and two multiplexers. Notice that the bit counter we discussed previously used asynchronous reset because the clock fed into bit counter is slow, so asynchronous reset is more appropriate. Anyway, let's get our focus back on clock counter. The maximum value that w_clock_count can be is CLOCKS_FOR_STOP – 1 = 651-1 = 650, so CLOCK_COUNTER_WIDTH = ceil(log2(CLOCKS_FOR_STOP-1)) = ceil(log2(651-1)) = 10.

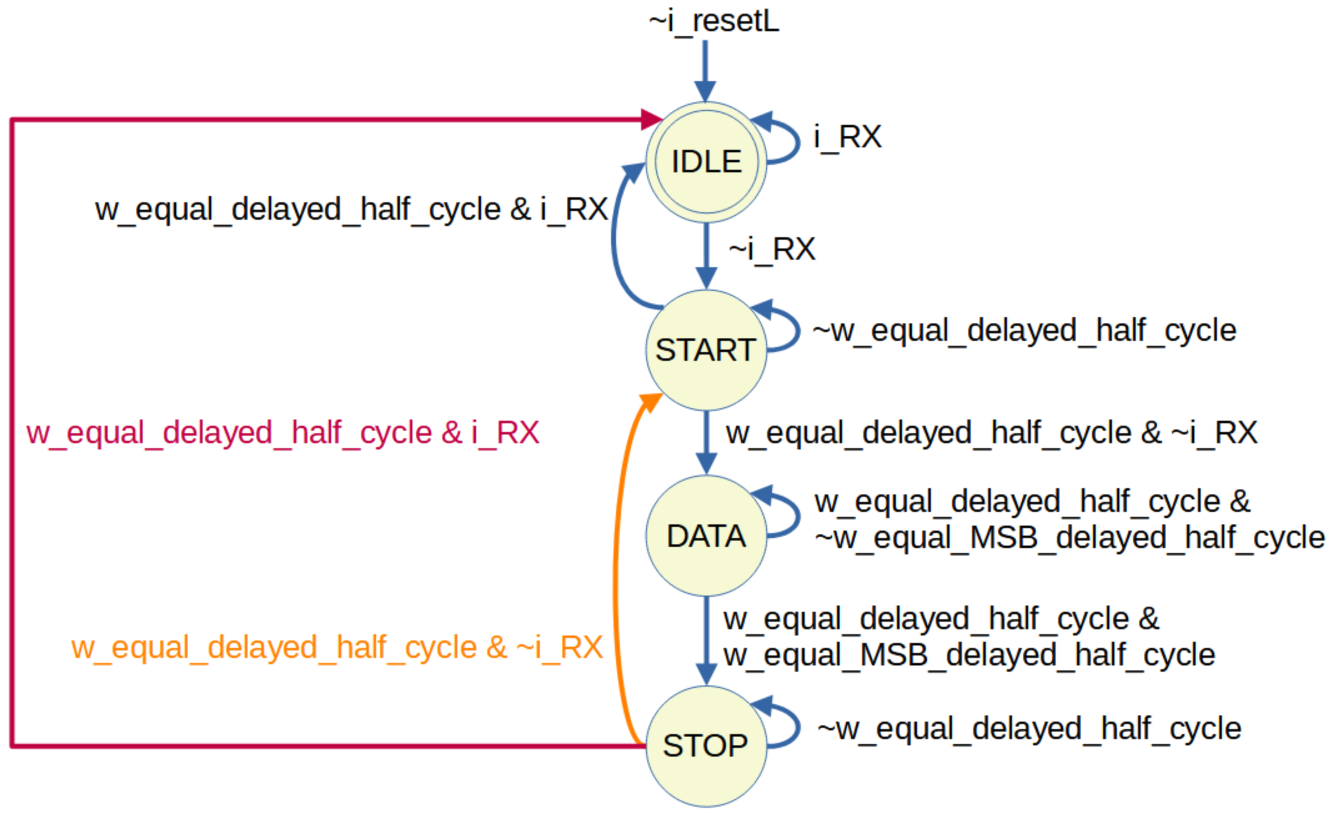
The clock for the flip flop is i_clock. Using states from FSM as selector for the multiplexers, we can select at what value should the clock counter reset. When state is STOP, the number of clocks to elapse is one and a half CLOCKS_PER_BIT. When state is DATA, the number of clocks to elapse is CLOCKS_PER_BIT. Else (when state is START), we only need half 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 ~ 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 IDLE (we are using zero-indexed counting, so the first clock count during START should be 0 and not 1), 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 & ~w_equal_delayed_half_cycle & (w_state_is_START_delayed_half_cycle | i_state_is_DATA | i_state_is_STOP).

UART Receiver Control Unit Design

The control unit is nothing but an FSM.

receiver_control.v



~i_resetL
IDLE
i_RX
w_equal_delayed_half_cycle & i_RX
~i_RX
START
~w_equal_delayed_half_cycle
w_equal_delayed_half_cycle & i_RX
w_equal_delayed_half_cycle & ~i_RX
DATA
w_equal_delayed_half_cycle &
~w_equal_MSB_delayed_half_cycle
w_equal_delayed_half_cycle & ~i_RX
w_equal_delayed_half_cycle &
w_equal_MSB_delayed_half_cycle
STOP
~w_equal_delayed_half_cycle

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 i_RX is high, we should stay at IDLE state. If i_RX, then we know it can be incoming START bit.

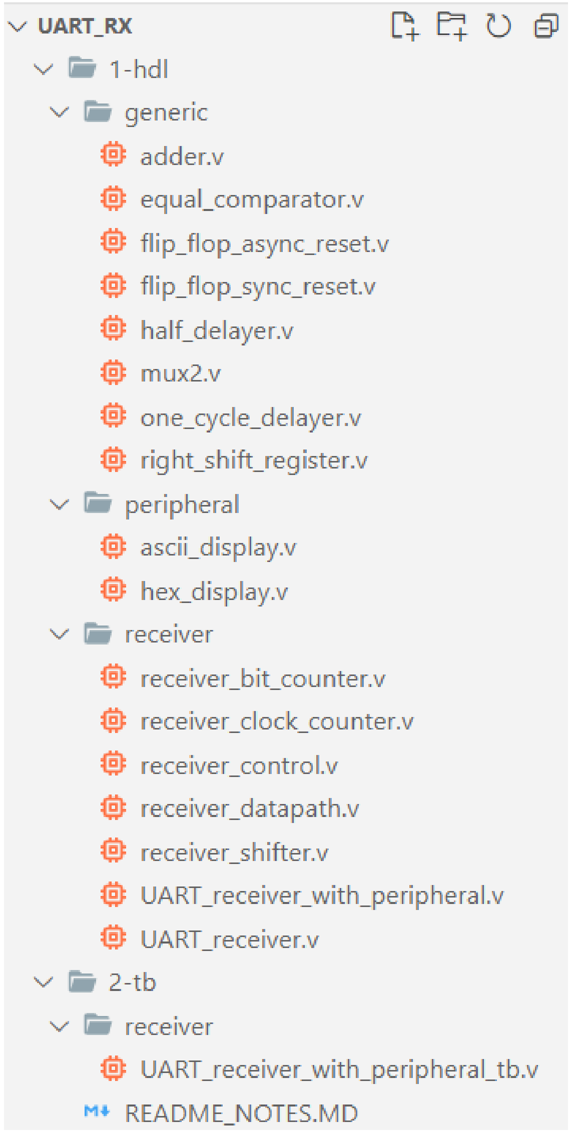
At START state, as long as the clock counter has not issued pulse (the w_equal_delayed_half_cycle), state will stay. If there is pulse from clock counter, we check whether i_RX is still low, if it’s high, then we conclude that the incoming START bit was just a glitch in system, and then state is back to IDLE. However, if i_RX is still low, state transitions to DATA.

During DATA state, there will be a few pulses from clock counter because the pulses are used to instruct clock counter itself to reset, or to sample data bit, or for bit counter to increment. However, the condition that will eventually trigger transition to STOP state is if we have already sampled the MSB.

At STOP state, as long as there is not pulse from clock counter to indicate that STOP state should ends, state will stay in STOP state. When there is pulse, we check if i_RX is 0. If it is, then we transition to START state, because next data is probably about to be received. If it is not, then the state simply go back to IDLE.

UART Receiver Code

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



```
UART_RX
  1-hdl
    generic
      adder.v
      equal_comparator.v
      flip_flop_async_reset.v
      flip_flop_sync_reset.v
      half_delayer.v
      mux2.v
      one_cycle_delayer.v
      right_shift_register.v
    peripheral
      ascii_display.v
      hex_display.v
    receiver
      receiver_bit_counter.v
      receiver_clock_counter.v
      receiver_control.v
      receiver_datapath.v
      receiver_shifter.v
      UART_receiver_with_peripheral.v
      UART_receiver.v
  2-tb
    receiver
      UART_receiver_with_peripheral_tb.v
    README_NOTES.MD
```

Generic

This folder contains commonly-used blocks.

generic/adder.v

```
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
```

generic/equal_comparator.v

```
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
```

generic/flip_flop_async_reset.v

```
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 <= 0;
    else o_q <= i_d;
  end
endmodule
```

generic/flip_flop_sync_reset.v

```
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 <= 0;
    else o_q <= i_d;
  end
endmodule
```

generic/half_delayer.v

```verilog
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
```

generic/one_cycle_delayer.v

```verilog
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

```verilog
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/right_shift_register.v

```verilog
module right_shift_register #(
  parameter WIDTH=8
)(
  input                    i_clock,
  input                    i_async_resetL,
  input                    i_shift_enL,
  input                    i_shift_in,
  output reg [WIDTH-1:0] o_parallel_shift_out
);
  always @(posedge i_clock or negedge i_async_resetL) begin
    if (~i_async_resetL) o_parallel_shift_out <= 0;
    else if (~i_shift_enL) o_parallel_shift_out <= {i_shift_in, o_p
arallel_shift_out[WIDTH-1:1]};
  end
endmodule
```

Peripheral

Under this folder is code for peripheral attached to UART receiver. In this project, two seven-segment displays will be used to display the data received in 2-digit hexadecimal value.

peripheral/hex_display.v

```
module hex_display(
  input  [3:0] i_hex,
  output reg [6:0] o_segment
  );
  //o_segment[6] is A
  //o_segment[0] is G
  always @(i_hex)
    begin
      case (i_hex)
          4'b0000 : o_segment <= 7'h7E; //0
          4'b0001 : o_segment <= 7'h30; //1
          4'b0010 : o_segment <= 7'h6D; //2
          4'b0011 : o_segment <= 7'h79; //3
          4'b0100 : o_segment <= 7'h33; //4
          4'b0101 : o_segment <= 7'h5B; //5
          4'b0110 : o_segment <= 7'h5F; //6
          4'b0111 : o_segment <= 7'h70; //7
          4'b1000 : o_segment <= 7'h7F; //8
          4'b1001 : o_segment <= 7'h7B; //9
          4'b1010 : o_segment <= 7'h77; //A
          4'b1011 : o_segment <= 7'h1F; //B
          4'b1100 : o_segment <= 7'h4E; //C
          4'b1101 : o_segment <= 7'h3D; //D
          4'b1110 : o_segment <= 7'h4F; //E
          4'b1111 : o_segment <= 7'h47; //F
      endcase
    end
endmodule
```

peripheral/ascii_display.v

```
module ascii_display(
  input [7:0] i_hex_2_digits,
  output [13:0] o_segment_2_digits
);
  hex_display inst_hex_display_digit0(
    .i_hex(i_hex_2_digits[3:0]),
    .o_segment(o_segment_2_digits[6:0])
  );

  hex_display inst_hex_display_digit1(
    .i_hex(i_hex_2_digits[7:4]),
    .o_segment(o_segment_2_digits[13:7])
  );
endmodule
```

Receiver

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

receiver/receiver_shifter.v

```
module receiver_shifter #(
  parameter CLOCK_COUNTER_WIDTH=10,
  parameter BIT_COUNTER_WIDTH=3,
  parameter DATA_WIDTH=8,
  parameter CLOCKS_PER_BIT=434
)(
  input i_clock,
  input i_resetL,
  input i_RX,
  input i_state_is_DATA,
  input i_state_is_STOP,
  input i_equal,
  output [DATA_WIDTH-1:0] o_received_data
);
  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_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 [DATA_WIDTH-1:0] w_parallel_shift_out;
  right_shift_register #(
    .WIDTH(DATA_WIDTH)
  ) inst_right_shift_register(
    .i_clock(w_equal_delayed_one_cycle),
    .i_async_resetL(i_resetL),
    .i_shift_enL(~w_state_is_DATA_delayed_half_cycle),
    .i_shift_in(i_RX),
    .o_parallel_shift_out(w_parallel_shift_out)
  );

  wire w_state_is_STOP_delayed_half_cycle;
  half_cycle_delayer inst_half_cycle_delayer_for_STOP(
    .i_clock(i_clock),
    .i_async_resetL(i_resetL),
    .i_to_be_delayed_half_cycle(i_state_is_STOP),
    .o_delayed_half_cycle(w_state_is_STOP_delayed_half_cycle)
  );

  flip_flop_async_reset #(
    .WIDTH(DATA_WIDTH)
  ) inst_register(
    .i_clock(w_state_is_STOP_delayed_half_cycle),
    .i_async_resetL(i_resetL),
    .i_d(w_parallel_shift_out),
    .o_q(o_received_data)
  );
endmodule
```

receiver/receiver_bit_counter.v

```
module receiver_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_constant_for_DATA_WIDTH_minus_one = DATA_WIDTH-1;
  equal_comparator #(
    .WIDTH(BIT_COUNTER_WIDTH)
  ) inst_equal_MSB_comparator(
    .i_a(w_bit_index),
    .i_b(lp_constant_for_DATA_WIDTH_minus_one),
    .o_c(o_equal_MSB)
  );
  localparam [BIT_COUNTER_WIDTH-1:0] lp_constant_for_one  = 1;
  adder #(
    .WIDTH(BIT_COUNTER_WIDTH)
  ) inst_adder(
    .i_a(w_bit_index),
    .i_b(lp_constant_for_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
```

receiver/receiver_clock_counter.v

```
module receiver_clock_counter #(
  parameter CLOCK_COUNTER_WIDTH=10,
  parameter CLOCKS_PER_BIT=434
)(
  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;
```

```verilog
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_constant_for_one  = 1;
adder #(
   .WIDTH(CLOCK_COUNTER_WIDTH)
) inst_adder(
   .i_a(w_clock_count),
   .i_b(lp_constant_for_one),
   .o_y(w_adder)
);

localparam [CLOCK_COUNTER_WIDTH-1:0] lp_CLOCKS_FOR_START_minus_one = CLOCKS_PER_BIT/2 - 1 ;
wire w_equal_CLOCKS_FOR_START_minus_one;
equal_comparator #(
   .WIDTH(CLOCK_COUNTER_WIDTH)
) inst_equal_comparator_for_CLOCKS_PER_BITS_minus_one(
   .i_a(w_clock_count),
   .i_b(lp_CLOCKS_FOR_START_minus_one),
   .o_c(w_equal_CLOCKS_FOR_START_minus_one)
);

localparam [CLOCK_COUNTER_WIDTH-1:0] lp_CLOCKS_FOR_DATA_minus_one  = CLOCKS_PER_BIT - 1 ;
wire w_equal_CLOCKS_FOR_DATA_minus_one;
equal_comparator #(
   .WIDTH(CLOCK_COUNTER_WIDTH)
) inst_equal_comparator_for_CLOCKS_FOR_DATA_minus_one(
    .i_a(w_clock_count),
    .i_b(lp_CLOCKS_FOR_DATA_minus_one),
    .o_c(w_equal_CLOCKS_FOR_DATA_minus_one)
);

localparam [CLOCK_COUNTER_WIDTH-1:0] lp_CLOCKS_FOR_STOP_minus_one  = (CLOCKS_PER_BIT/2 + CLOCKS_PER_BIT) - 1 ;
wire w_equal_CLOCKS_FOR_STOP_minus_one;
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)
);

wire w_equal_stage1;
mux2 #(
    .WIDTH(1)
) inst_mux2_stage1(
    .i_d0(w_equal_CLOCKS_FOR_START_minus_one),
    .i_d1(w_equal_CLOCKS_FOR_DATA_minus_one),
    .i_s(i_state_is_DATA),
    .o_y(w_equal_stage1)
);

mux2 #(
    .WIDTH(1)
) inst_mux2_stage2(
    .i_d0(w_equal_stage1),
    .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
```

receiver/receiver_datapath.v

```
module receiver_datapath #(
  parameter CLOCK_COUNTER_WIDTH=10,
  parameter BIT_COUNTER_WIDTH=3,
  parameter DATA_WIDTH=8,
  parameter CLOCKS_PER_BIT=434
)(
  input                   i_clock,
  input                   i_resetL,
  input                   i_RX,
  input                   i_state_is_START,
  input                   i_state_is_DATA,
  input                   i_state_is_STOP,
  output                  o_equal,
  output                  o_equal_MSB,
  output [DATA_WIDTH-1:0] o_received_data
);
  receiver_shifter #(
    .CLOCK_COUNTER_WIDTH(CLOCK_COUNTER_WIDTH),
    .BIT_COUNTER_WIDTH(BIT_COUNTER_WIDTH),
    .DATA_WIDTH(DATA_WIDTH),
    .CLOCKS_PER_BIT(CLOCKS_PER_BIT)
  ) inst_receiver_shifter(
    .i_clock(i_clock),
    .i_resetL(i_resetL),
    .i_RX(i_RX),
    .i_state_is_DATA(i_state_is_DATA),
    .i_state_is_STOP(i_state_is_STOP),
    .i_equal(o_equal),
    .o_received_data(o_received_data)
  );

  receiver_bit_counter #(
    .BIT_COUNTER_WIDTH(BIT_COUNTER_WIDTH),
    .DATA_WIDTH(DATA_WIDTH)
  ) inst_receiver_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)
  );

  receiver_clock_counter #(
    .CLOCK_COUNTER_WIDTH(CLOCK_COUNTER_WIDTH),
    .CLOCKS_PER_BIT(CLOCKS_PER_BIT)
  ) inst_receiver_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
```

receiver/receiver_control.v

```
module receiver_control(
  input  i_clock,
  input  i_resetL,
  input  i_RX,
  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)
```

```verilog
);
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_RX, 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_RX) r_next_state <= IDLE;
      else if (~i_RX) r_next_state <= START;
    START:
      if (~w_equal_delayed_half_cycle) r_next_state <= START;
      else if (w_equal_delayed_half_cycle & i_RX) r_next_state <= IDLE;
      else if (w_equal_delayed_half_cycle & ~i_RX) 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_RX) r_next_state <= IDLE;
      else if (w_equal_delayed_half_cycle & ~i_RX) r_next_state <= START;
  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
```

receiver/UART_receiver.v

```
module UART_receiver #(
  parameter CLOCK_COUNTER_WIDTH=10,
  parameter BIT_COUNTER_WIDTH=3,
  parameter DATA_WIDTH=8,
  parameter CLOCKS_PER_BIT=434
)(
  input                  i_clock,
  input                  i_resetL,
  input                  i_RX,
  output [DATA_WIDTH-1:0] o_received_data
);
  wire w_equal;
  wire w_equal_MSB;
  wire w_state_is_START;
  wire w_state_is_DATA;
  wire w_state_is_STOP;

  receiver_control inst_receiver_control(
    .i_clock(i_clock),
    .i_resetL(i_resetL),
    .i_RX(i_RX),
    .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)
  );

  receiver_datapath #(
    .CLOCK_COUNTER_WIDTH(CLOCK_COUNTER_WIDTH),
    .BIT_COUNTER_WIDTH(BIT_COUNTER_WIDTH),
    .DATA_WIDTH(DATA_WIDTH),
    .CLOCKS_PER_BIT(CLOCKS_PER_BIT)
  ) inst_receiver_datapath(
    .i_clock(i_clock),
    .i_resetL(i_resetL),
    .i_RX(i_RX),
    .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_received_data(o_received_data)
  );
endmodule
```

receiver/UART_receiver_with_peripheral.v

```
module UART_receiver_with_peripheral #(
  parameter CLOCK_COUNTER_WIDTH=10,
  parameter BIT_COUNTER_WIDTH=3,
  parameter DATA_WIDTH=8,
  parameter CLOCKS_PER_BIT=434
)(
  input                   i_clock,
  input                   i_resetL,
  input                   i_RX,
  output [13:0] o_segment_2_digits
);
  wire [DATA_WIDTH-1:0] w_received_data;
  UART_receiver #(
    .CLOCK_COUNTER_WIDTH(CLOCK_COUNTER_WIDTH),
    .BIT_COUNTER_WIDTH(BIT_COUNTER_WIDTH),
    .DATA_WIDTH(DATA_WIDTH),
    .CLOCKS_PER_BIT(CLOCKS_PER_BIT)
  ) inst_UART_receiver(
    .i_clock(i_clock),
    .i_resetL(i_resetL),
    .i_RX(i_RX),
    .o_received_data(w_received_data)
  );

  ascii_display inst_ascii_display(
    .i_hex_2_digits(w_received_data),
    .o_segment_2_digits(o_segment_2_digits)
  );
endmodule
```

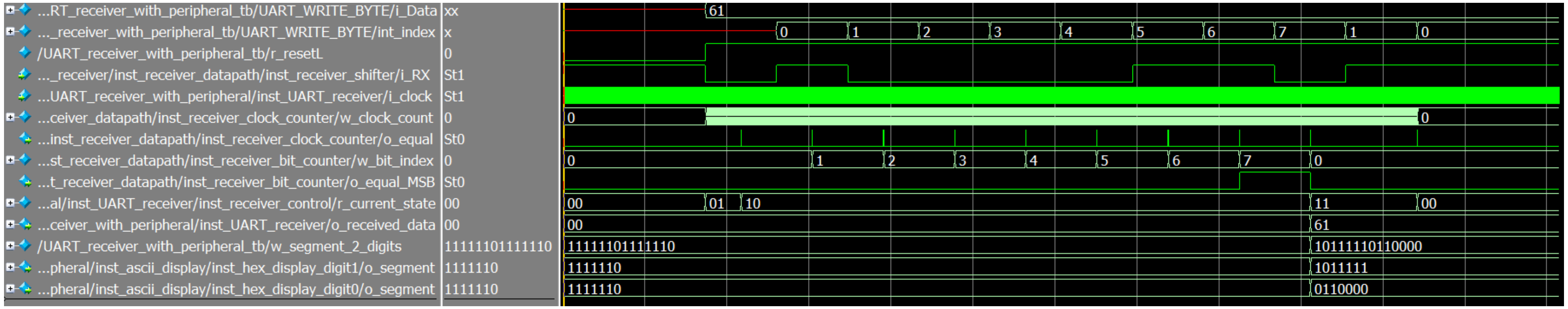
UART Receiver Testbench and Simulation

The testbench basically sends serial data 8’h61 using UART protocol to the test the UART_receiver_with_peripheral.v module.

UART_receiver_with_peripheral_tb.v

```
`timescale  1ns/1ps

module UART_receiver_with_peripheral_tb();
  parameter CLOCK_COUNTER_WIDTH=10;
  parameter BIT_COUNTER_WIDTH=3;
  parameter DATA_WIDTH=8;
  parameter CLOCKS_PER_BIT=434;

  reg r_clock;
  reg r_resetL;
  reg r_RX;
  wire [13:0] w_segment_2_digits;

  UART_receiver_with_peripheral #(
    .CLOCK_COUNTER_WIDTH(CLOCK_COUNTER_WIDTH),
    .BIT_COUNTER_WIDTH(BIT_COUNTER_WIDTH),
    .DATA_WIDTH(DATA_WIDTH),
    .CLOCKS_PER_BIT(CLOCKS_PER_BIT)
  ) dut_UART_receiver_with_peripheral(
    .i_clock(r_clock),
    .i_resetL(r_resetL),
    .i_RX(r_RX),
    .o_segment_2_digits(w_segment_2_digits)
  );

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

  task UART_WRITE_BYTE;
    input [7:0] i_Data;
    integer     int_index;
    begin

    // Send Start Bit
    r_RX <= 1'b0;
    #(CLOCKS_PER_BIT);

    // Send Data Byte
    for (int_index=0; int_index<8; int_index=int_index+1)
      begin
      r_RX <= i_Data[int_index];
      #(CLOCKS_PER_BIT);
      end
    int_index = 1; // not important, just a marker with duration CLOCKS_PER_BIT

    // Send Stop Bit
    r_RX <= 1'b1;
    #(CLOCKS_PER_BIT);
    int_index = 0; // not important, just a marker with duration CLOCKS_PER_BIT
    end
  endtask

 initial begin
   r_resetL = 0; r_RX = 1; #(CLOCKS_PER_BIT*2);
   r_resetL = 1; #1;
   UART_WRITE_BYTE(8'h61);
   #(CLOCKS_PER_BIT*2); $stop;
 end
endmodule
```

The output waveform is just the same as what we discussed in timing diagram earlier on. The receiver is clearly working because the data that we are sending in testbench to the receiver is 8’h61 and near r_current_state=11=STOP state, the waveform shows that o_received_data= 8’h61 as desired.



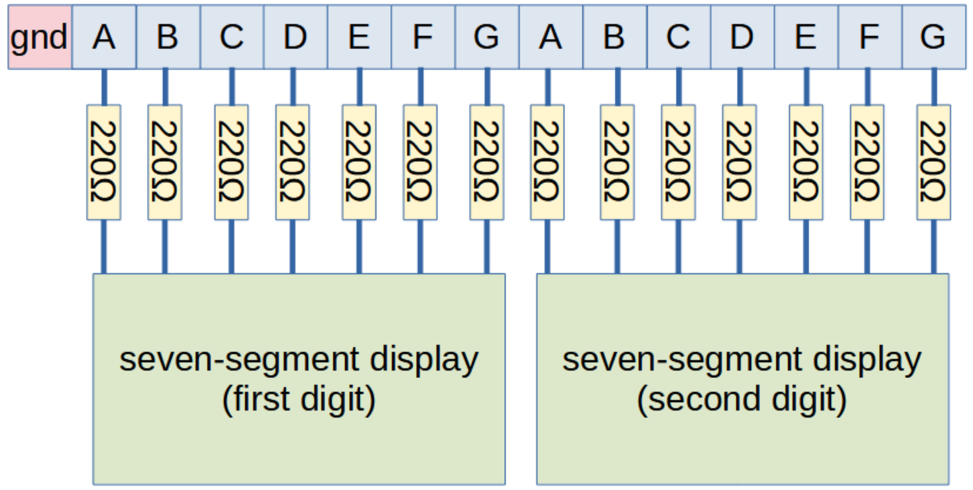
```
...RT_receiver_with_peripheral_tb/UART_WRITE_BYTE/i_Data      xx
..._receiver_with_peripheral_tb/UART_WRITE_BYTE/int_index     x
/UART_receiver_with_peripheral_tb/r_resetL                    0
..._receiver/inst_receiver_datapath/inst_receiver_shifter/i_RX St1
...UART_receiver_with_peripheral/inst_UART_receiver/i_clock   St1
...ceiver_datapath/inst_receiver_clock_counter/w_clock_count  0
...inst_receiver_datapath/inst_receiver_clock_counter/o_equal St0
...st_receiver_datapath/inst_receiver_bit_counter/w_bit_index 0
...t_receiver_datapath/inst_receiver_bit_counter/o_equal_MSB  St0
...al/inst_UART_receiver/inst_receiver_control/r_current_state 00
...ceiver_with_peripheral/inst_UART_receiver/o_received_data  00
/UART_receiver_with_peripheral_tb/w_segment_2_digits          11111101111110
...pheral/inst_ascii_display/inst_hex_display_digit1/o_segment 1111110
...pheral/inst_ascii_display/inst_hex_display_digit0/o_segment 1111110
```

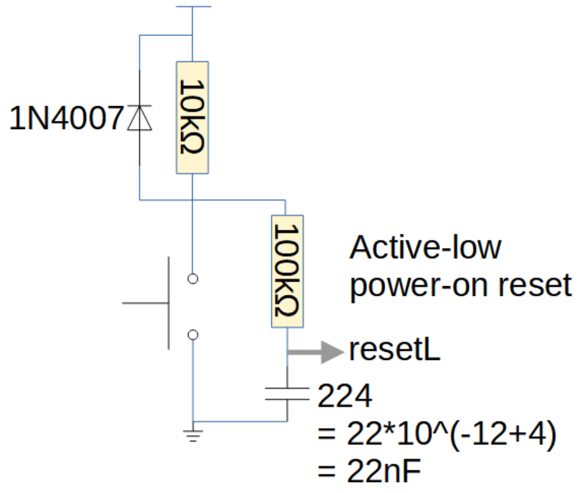
The extra things are the last three rows (the last two rows are just w_segment_2_digits separated into 1 digit respectively). Basically, the o_received_data is transformed into hexadecimal value to be displayed on 2 seven-segments dispaly.

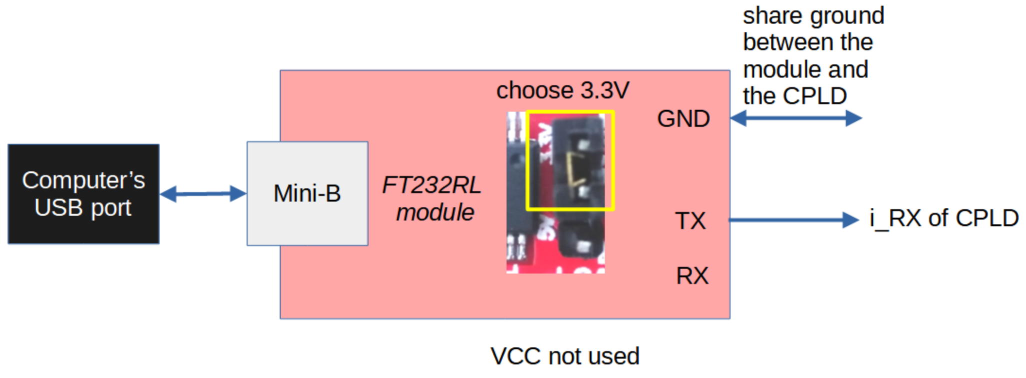
UART Receiver Hardware

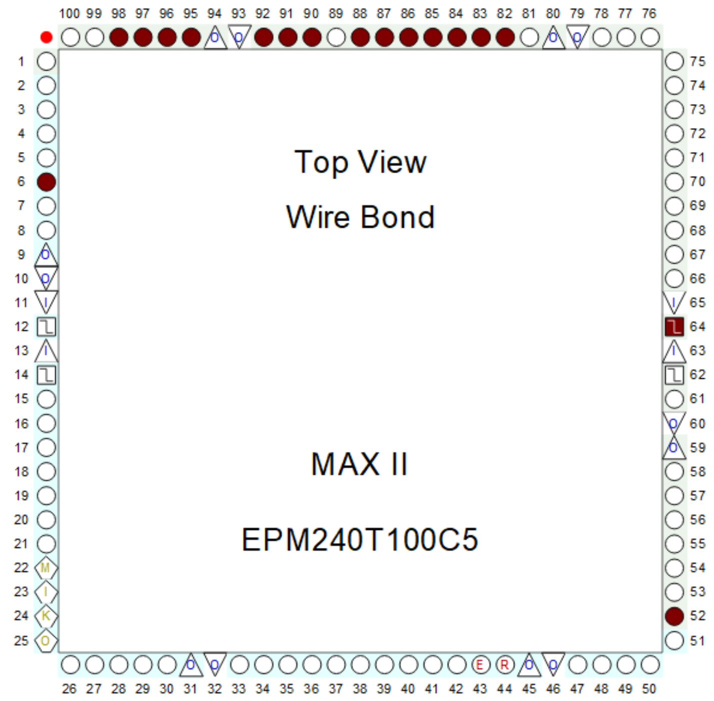
For the demo of the hardware, please watch the video on YouTube:
https://youtu.be/VMD-bAN7MmI?t=3686

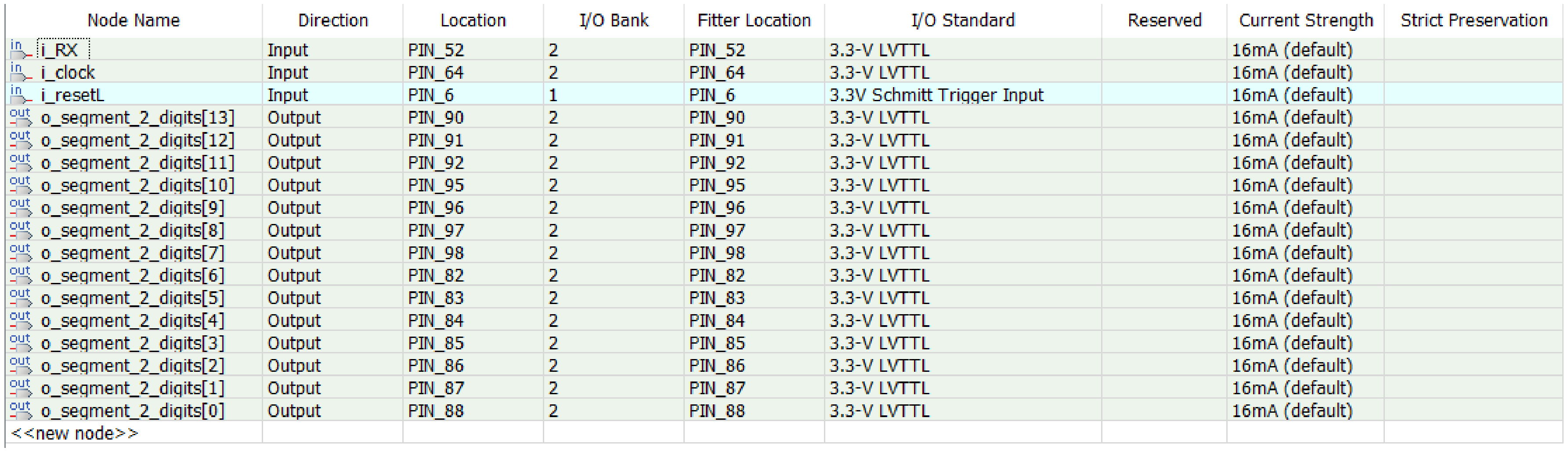
Code for download:
https://github.com/endeneer/CPLD_UART_Datapath_and_Control_Unit


gnd
A
B
C
D
E
F
G
A
B
C
D
E
F
G
220Ω
seven-segment display (first digit)
seven-segment display (second digit)









Node Name	Direction	Location	I/O Bank	Fitter Location	I/O Standard	Reserved	Current Strength	Strict Preservation
i_RX	Input	PIN_52	2	PIN_52	3.3-V LVTTL		16mA (default)	
i_clock	Input	PIN_64	2	PIN_64	3.3-V LVTTL		16mA (default)	
i_resetL	Input	PIN_6	1	PIN_6	3.3V Schmitt Trigger Input		16mA (default)	
o_segment_2_digits[13]	Output	PIN_90	2	PIN_90	3.3-V LVTTL		16mA (default)	
o_segment_2_digits[12]	Output	PIN_91	2	PIN_91	3.3-V LVTTL		16mA (default)	
o_segment_2_digits[11]	Output	PIN_92	2	PIN_92	3.3-V LVTTL		16mA (default)	
o_segment_2_digits[10]	Output	PIN_95	2	PIN_95	3.3-V LVTTL		16mA (default)	
o_segment_2_digits[9]	Output	PIN_96	2	PIN_96	3.3-V LVTTL		16mA (default)	
o_segment_2_digits[8]	Output	PIN_97	2	PIN_97	3.3-V LVTTL		16mA (default)	
o_segment_2_digits[7]	Output	PIN_98	2	PIN_98	3.3-V LVTTL		16mA (default)	
o_segment_2_digits[6]	Output	PIN_82	2	PIN_82	3.3-V LVTTL		16mA (default)	
o_segment_2_digits[5]	Output	PIN_83	2	PIN_83	3.3-V LVTTL		16mA (default)	
o_segment_2_digits[4]	Output	PIN_84	2	PIN_84	3.3-V LVTTL		16mA (default)	
o_segment_2_digits[3]	Output	PIN_85	2	PIN_85	3.3-V LVTTL		16mA (default)	
o_segment_2_digits[2]	Output	PIN_86	2	PIN_86	3.3-V LVTTL		16mA (default)	
o_segment_2_digits[1]	Output	PIN_87	2	PIN_87	3.3-V LVTTL		16mA (default)	
o_segment_2_digits[0]	Output	PIN_88	2	PIN_88	3.3-V LVTTL		16mA (default)	
<<new node>>								