# 第 2 回輪講資料

4321　野秋 琳太郎

2025 年 6 月 27 日

指導教員　宮田 尚起

## 1.　はじめに

LED で光通信をして,「糸なし糸電話」をやりたい.

## 2.　班の進捗

全体の進捗は図 1 のとおりである. 今週で, デジタル信号による通信で必要な要素が一通り揃ったことになる.

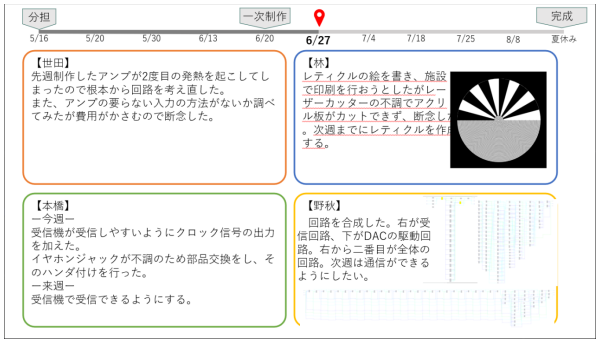


図 1. 全体の進捗

## 3.　野秋の進捗

受信側で使用する FPGA の回路をすべて書いた. 回路は図 2 に示すような構成になっている.

Uart モジュールが受け取る通信のプロトコルは図 3 のとおりである. これは uart を 16bit に拡張したものであり, データは最上位ビットから順に送られる.

論理合成の結果を表 1 に示す. 表より,fpga の容量には余裕があることが分かる.

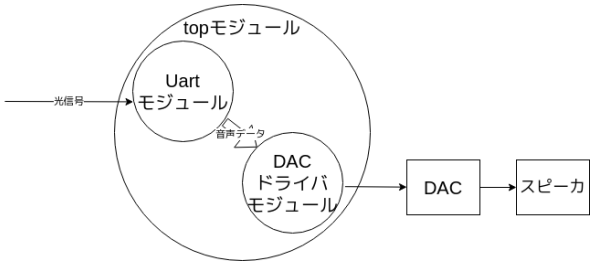


図 2. 回路のブロック図

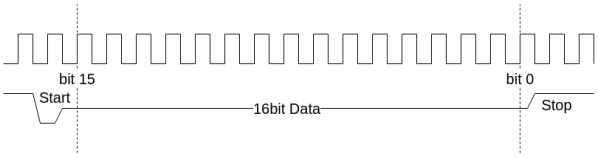


図 3. 使用する通信のプロトコル

## 4.　今後の予定

現状, 送信したデータを図 4 のように反転して受信していた. これは FPGA で受信したデータを反転させることで対応したい.

表 1. 合成のレポート

| Resource Utilization Summary | | |
|---|---|---|
| Resource | Usage | Utilization |
| Logic | 97(93 LUT, 4 ALU) / 20736 | <1% |
| Register | 97 / 16173 | <1% |
| –Register as Latch | 0 / 16173 | 0% |
| –Register as FF | 97 / 16173 | <1% |
| BSRAM | 0 / 46 | 0% |

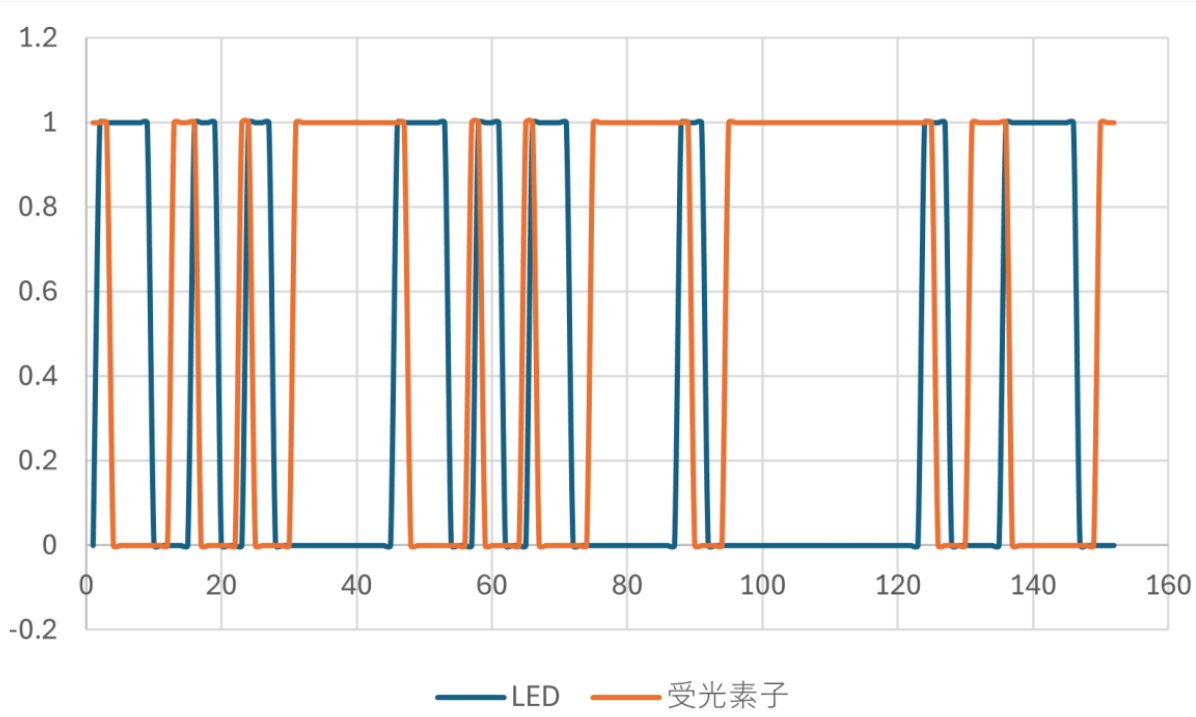


図 4. LED にかけた電圧と, 受光素子で受信した電圧

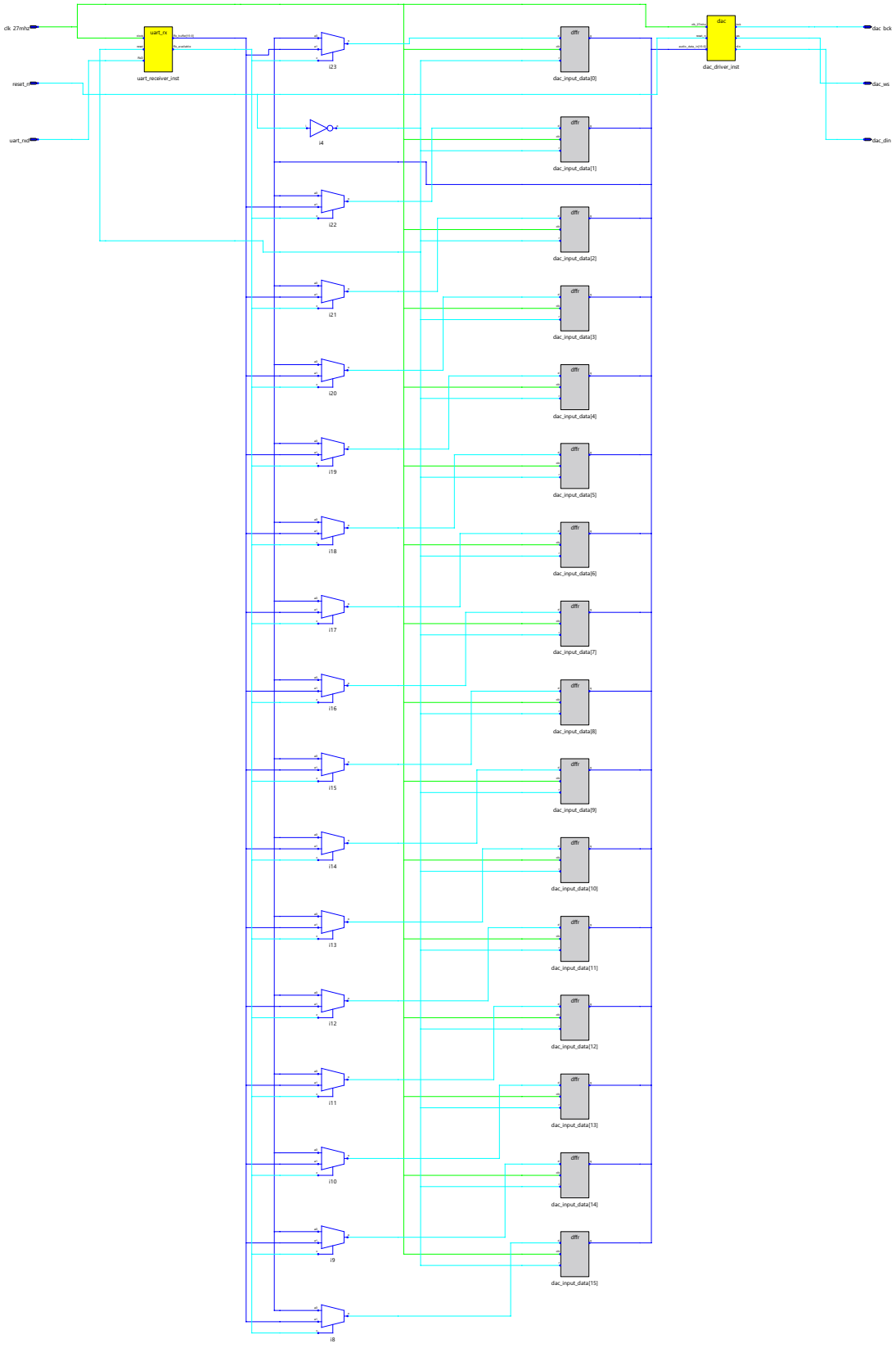


図 5. 合成した top モジュール

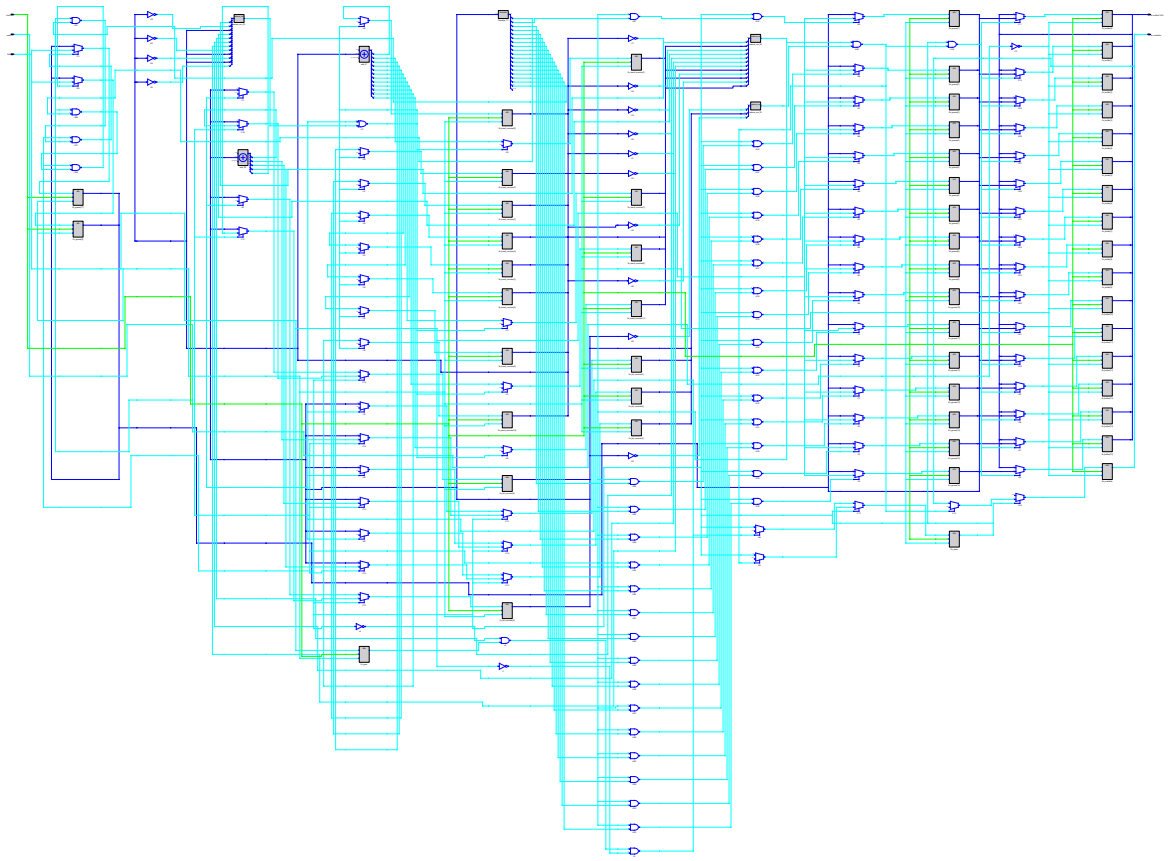

図 6. 合成した uart モジュール

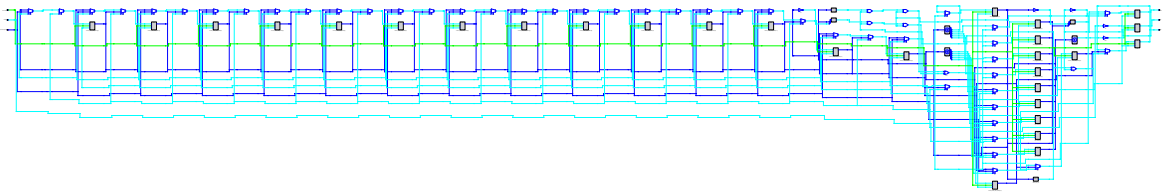

図 7. 合成した dac モジュール

## 5. 書いた回路たち

Listing 1: top モジュール

```
// top_module.sv
module top_module  (
    input wire clk_27mhz, // 27MHz FPGA Clock (e.g., from an oscillator)
    input wire reset_n,   // Active low reset
    input wire uart_rxd,  // UART Receive Data input (connect to external UART Tx
        pin)

    // PT8211 DAC outputs
    output logic dac_bck, // Bit Clock
    output logic dac_ws,  // Word Select
    output logic dac_din  // Data Input
);
    // Internal wires for connecting modules
    wire [15:0] uart_rx_data;
    wire uart_data_available;

    // Instantiate UART Receiver
    uart_rx uart_receiver_inst (
        .clock          (clk_27mhz),
        .reset          (~reset_n), // uart_rx expects active high reset
        .RxD            (uart_rxd),
        .Rx_buffer      (uart_rx_data),
        .Rx_available   (uart_data_available)
    );
    // Register to hold the last received UART data
    logic [15:0] dac_input_data;

    // Logic to latch UART data into DAC input when available
    always_ff @(posedge clk_27mhz or negedge reset_n) begin
        if (!reset_n) begin
            dac_input_data <= 16'b0;
        end else begin
            if (uart_data_available) begin
                dac_input_data <= uart_rx_data; // Latch new data from UART
            end
        end
    end
    // Instantiate DAC Driver
    dac dac_driver_inst (
        .clk_27mhz      (clk_27mhz),
        .reset_n        (reset_n),
        .audio_data_in  (dac_input_data), // Pass the latched UART data to DAC
        .bck            (dac_bck),
        .ws             (dac_ws),
        .din            (dac_din)
    );

endmodule
```

Listing 2: uart モジュール

```
module uart_rx(
    input clock,
    input reset,
    input RxD,
    output logic [15:0] Rx_buffer,  // Changed to 16-bit
    output logic Rx_available
);
parameter baud_rate=9600;

// FPGAのクロック周波数/ボーレイト
parameter baud_div=(27000000/baud_rate)-1;

// 信号のハラのぶぶん (center of the bit period)
parameter valid=baud_div/2;

// マシンのステート
typedef enum logic {
    idle=0,
    trns=1
} uart_state;

logic [11:0] Rx_baud_counter; // Counter for baud rate division
logic [4:0] Rx_bit_counter;  // Changed to accommodate up to 18 bits (1 start + 16 data + 1 stop)
logic Rx_prev;               // Previous state of RxD for edge detection
wire Rx_strt;               // Detects start bit
wire Rx_trig;               // Triggers at the center of each bit
logic [17:0] Rx_queue;       // Changed to 16 data bits + 1 start bit + 1 stop bit
uart_state Rx_state;

// Rx立下り検知 (Detect falling edge on RxD for start bit)
assign Rx_strt = (Rx_state==idle && Rx_prev==1 && RxD==0) ? 1'b1 : 1'b0;

always@(posedge clock or negedge reset)begin
    if(!reset) Rx_prev <= 1'b1;
    else Rx_prev <= RxD;
end

always @(posedge clock or negedge reset) begin
    if(!reset) Rx_baud_counter <= 12'b0;
    else begin
        // Reset counter on start bit detection or when baud_div is reached
        if (Rx_strt || Rx_baud_counter == baud_div) begin
            Rx_baud_counter <= 12'b0;
        end else begin
            Rx_baud_counter <= Rx_baud_counter + 1'b1;
        end
    end
end

// Rx_trig: Trigger when Rx_baud_counter reaches the 'valid' point (center of the bit)
```

```
assign Rx_trig = (valid == Rx_baud_counter) ? 1'b1 : 1'b0;

always_ff@(posedge clock or negedge reset)begin
    if(!reset)begin
        Rx_bit_counter <= 5'b0;
        Rx_state <= idle;
        Rx_available <= 1'b0;
        Rx_buffer <= 16'b0; // Initialize Rx_buffer
        Rx_queue <= 18'b0;  // Initialize Rx_queue
    end
    else begin
        case(Rx_state)
            idle: begin
                Rx_available <= 1'b0; // Clear Rx_available when idle
                if(Rx_strt)begin
                    Rx_state <= trns;
                    Rx_bit_counter <= 5'b0; // Reset bit counter
                end
            end
            trns: begin
                if(Rx_trig)begin
                    // Store the received bit into the queue
                    Rx_queue[Rx_bit_counter] <= RxD;
                    Rx_bit_counter <= Rx_bit_counter + 1'b1; // Increment bit
                        counter

                    // Check if all 16 data bits + 1 start bit + 1 stop bit have
                        been received
                    // Total bits: 1 (start) + 16 (data) + 1 (stop) = 18 bits.
                    // Rx_bit_counter will go from 0 to 17.
                    if(Rx_bit_counter == 17) begin // After receiving the 18th bit
                        (the stop bit)
                        Rx_state <= idle; // Go back to idle state
                        // Data bits are from index 1 to 16 in Rx_queue
                            (Rx_queue[16:1])
                        // Assuming LSB first, Rx_queue[1] is the first data bit,
                            Rx_queue[16] is the last.
                        // If MSB first, you would need to reverse the order.
                        // Standard UART is LSB first.
                        Rx_buffer <= Rx_queue[16:1]; // Extract 16 data bits
                        Rx_available <= 1'b1; // Indicate new data is available
                    end
                end
            end
            default: Rx_state <= idle; // Should not happen, but for completeness
        endcase
    end
end
endmodule
```

Listing 3: dac モジュール

```
module dac(
    input wire clk_27mhz,          // 27MHz FPGA Clock
    input wire reset_n,            // Active low reset
    input wire [15:0] audio_data_in, // 16-bit audio data input (always valid)
    // input wire data_valid,      // Removed: assuming audio_data_in is always
        valid and updated externally

    output logic bck,              // Bit Clock to PT8211 (Pin 1)
    output logic ws,               // Word Select to PT8211 (Pin 2)
    output logic din               // Data Input to PT8211 (Pin 3)
);

    // --- Parameters ---
    localparam BCK_PERIOD_CYCLES_27MHZ = 2; // 27MHz / 2 = 13.5MHz BCK. Max BCK is
        18.4MHz.
    localparam BITS_PER_CHANNEL = 16;       // 16-bit DAC
    localparam TOTAL_BITS_PER_WS_CYCLE = BITS_PER_CHANNEL * 2; // 16 bits for Right
        + 16 bits for Left

    // --- Internal Registers ---
    logic [4:0] bck_toggle_counter; // For BCK generation
    logic [4:0] bit_counter;        // Counts bits within a WS cycle (0 to 31)
    logic [15:0] current_audio_data_shifter; // Holds data for shifting out

    // --- BCK Generation ---
    // BCK toggles at 13.5MHz
    always_ff @(posedge clk_27mhz or negedge reset_n) begin
        if (!reset_n) begin
            bck_toggle_counter <= '0;
            bck <= 0;
        end else begin
            if (bck_toggle_counter == (BCK_PERIOD_CYCLES_27MHZ - 1)) begin
                bck_toggle_counter <= '0;
                bck <= ~bck; // Toggle BCK
            end else begin
                bck_toggle_counter <= bck_toggle_counter + 1;
            end
        end
    end

    // --- WS and DIN Generation ---
    always_ff @(posedge clk_27mhz or negedge reset_n) begin
        if (!reset_n) begin
            ws <= 0; // Start with WS low for Right Channel
            bit_counter <= '0;
            current_audio_data_shifter <= '0;
            din <= 0;
        end else begin
            // Load new audio data at the beginning of each word cycle
            // This assumes audio_data_in is stable for a whole WS cycle
            // and updates synchronously or asynchronously but prior to the start
                of a new frame.
```

```
49              if (bit_counter == 0 && bck_toggle_counter == 0 && bck == 0) begin
50                  current_audio_data_shifter <= audio_data_in; // Load data for Right
                        Channel
51              end
52              // Reload for Left channel when bit_counter transitions to 16
53              else if (bit_counter == BITS_PER_CHANNEL && bck_toggle_counter == 0 &&
                    bck == 0) begin
54                  current_audio_data_shifter <= audio_data_in; // Load same data for
                        Left Channel
55              end
56
57              // Update WS based on bit_counter
58              if (bit_counter < BITS_PER_CHANNEL) begin
59                  ws <= 0; // WS low for Right Channel
60              end else begin
61                  ws <= 1; // WS high for Left Channel
62              end
63
64              // Shift DIN data on the falling edge of BCK (to ensure stability on
                    rising edge)
65              // The PT8211 shifts DIN data on the rising edge of BCK.
66              // Therefore, DIN needs to be stable *before* the rising edge.
67              // We update DIN when bck is low and bck_toggle_counter is at half the
                    period, ready for the next bck rising edge.
68              if (bck_toggle_counter == (BCK_PERIOD_CYCLES_27MHZ / 2 - 1) && bck ==
                    1) begin // This is when BCK is still high, before it goes low.
69                  din <= current_audio_data_shifter[BITS_PER_CHANNEL - 1]; // MSB
                        first
70                  current_audio_data_shifter <= current_audio_data_shifter << 1; //
                        Shift left for next bit
71                  bit_counter <= bit_counter + 1;
72              end
73
74              // Reset bit_counter for next word cycle after all bits are sent
75              if (bit_counter == TOTAL_BITS_PER_WS_CYCLE && bck_toggle_counter ==
                    (BCK_PERIOD_CYCLES_27MHZ - 1)) begin
76                  bit_counter <= 0;
77              end
78          end
79      end
80
81  endmodule
```

Listing 4: IO の割当

```
//Copyright (C)2014-2024 Gowin Semiconductor Corporation.
//All rights reserved.
//File Title: Physical Constraints file
//Tool Version: V1.9.10 (64-bit)
//Part Number: GW2A-LV18PG256C8/I7
//Device: GW2A-18
//Device Version: C
//Created Time: Fri 06 20 15:59:50 2025

IO_LOC "dac_din" P15;
IO_PORT "dac_din" IO_TYPE=LVCMOS18 PULL_MODE=UP DRIVE=8 BANK_VCCIO=1.8;
IO_LOC "dac_ws" P16;
IO_PORT "dac_ws" IO_TYPE=LVCMOS18 PULL_MODE=UP DRIVE=8 BANK_VCCIO=1.8;
IO_LOC "dac_bck" N15;
IO_PORT "dac_bck" IO_TYPE=LVCMOS18 PULL_MODE=UP DRIVE=8 BANK_VCCIO=1.8;
IO_LOC "uart_rxd" J14;
IO_PORT "uart_rxd" IO_TYPE=LVCMOS18 PULL_MODE=UP BANK_VCCIO=1.8;
IO_LOC "reset_n" T10;
IO_PORT "reset_n" IO_TYPE=LVCMOS18 PULL_MODE=UP BANK_VCCIO=1.8;
IO_LOC "clk_27mhz" H11;
IO_PORT "clk_27mhz" IO_TYPE=LVCMOS18 PULL_MODE=UP BANK_VCCIO=1.8;
```