Adder + Fnd Controller 설계

김민기

목차

1 | 설계 소개, 목적, 예상 simulation/동작

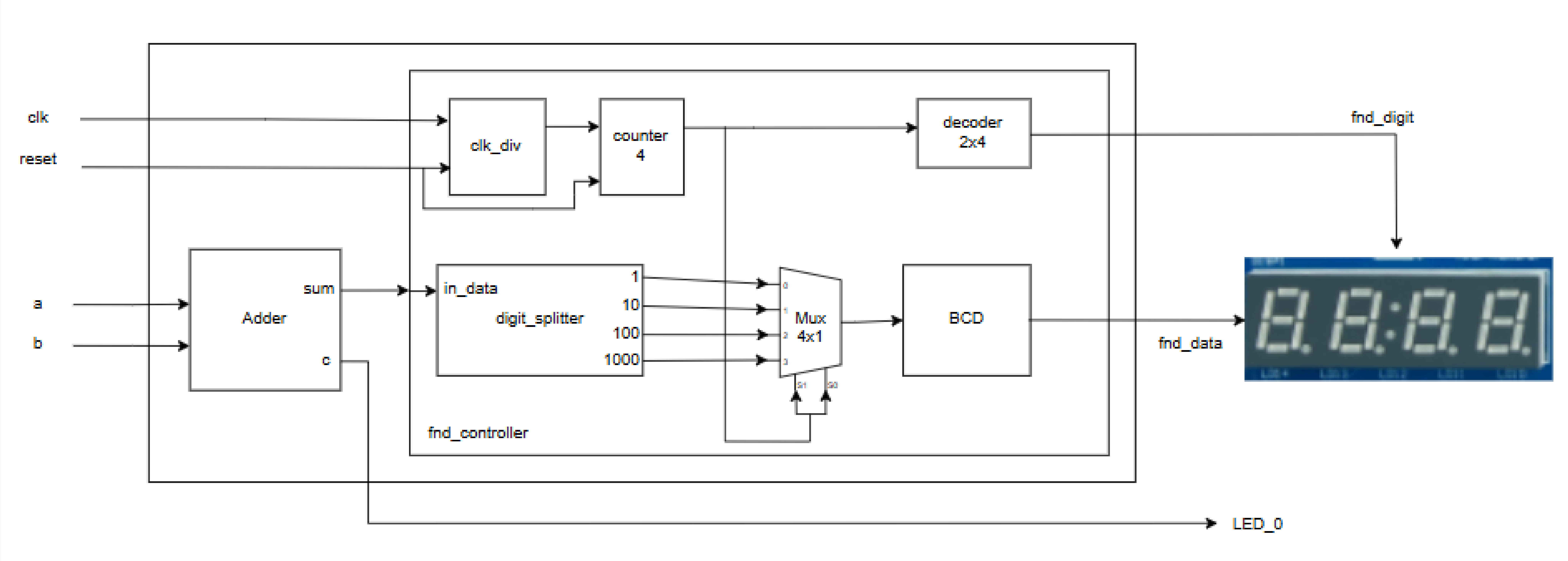
- 이번 설계는 8bit adder 결과 값을 fnd_controller 내부의 clk_div, digit_splitter, counter, bcd 등 다양한 module을 통해 7-segment display에 시각화 하는 설계입니다.

- 단순 가산기를 넘어서 100Mhz인 system clock을 clk_divider를 사용하여 1khz로 낮추어 사람의 눈으로도 볼 수 있도록 설계하였습니다.

- testbench를 통해 예상되는 adder의 결과와 fnd_data의 값을 simulation 파형으로 관찰하였습니다.

- 실제 HW board를 통해 동작을 구현하였습니다.

2 | Block Diagram, RTL Schematic

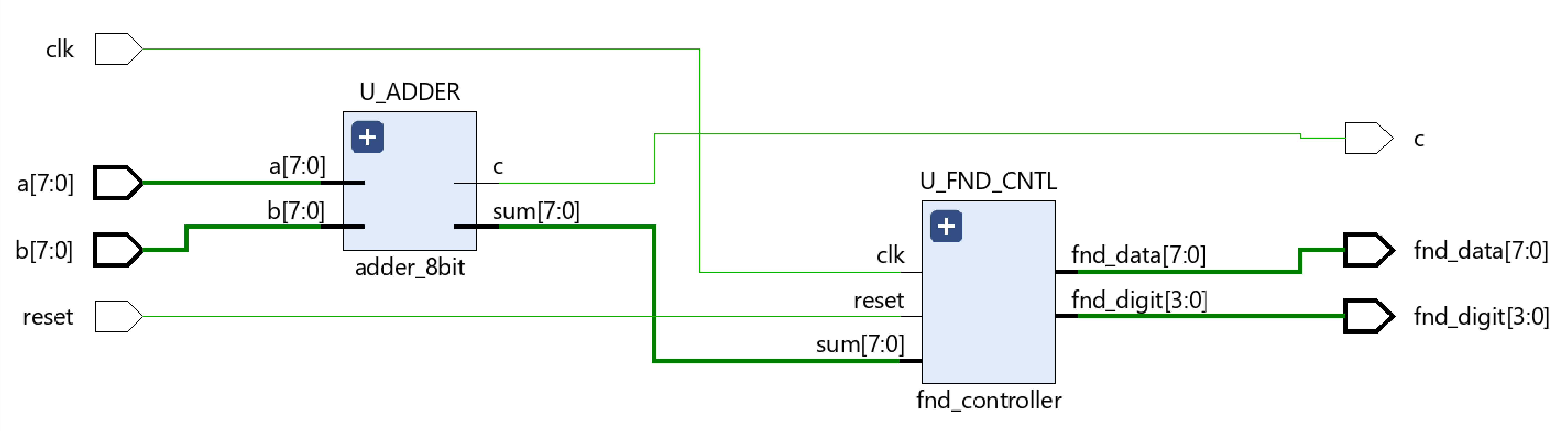


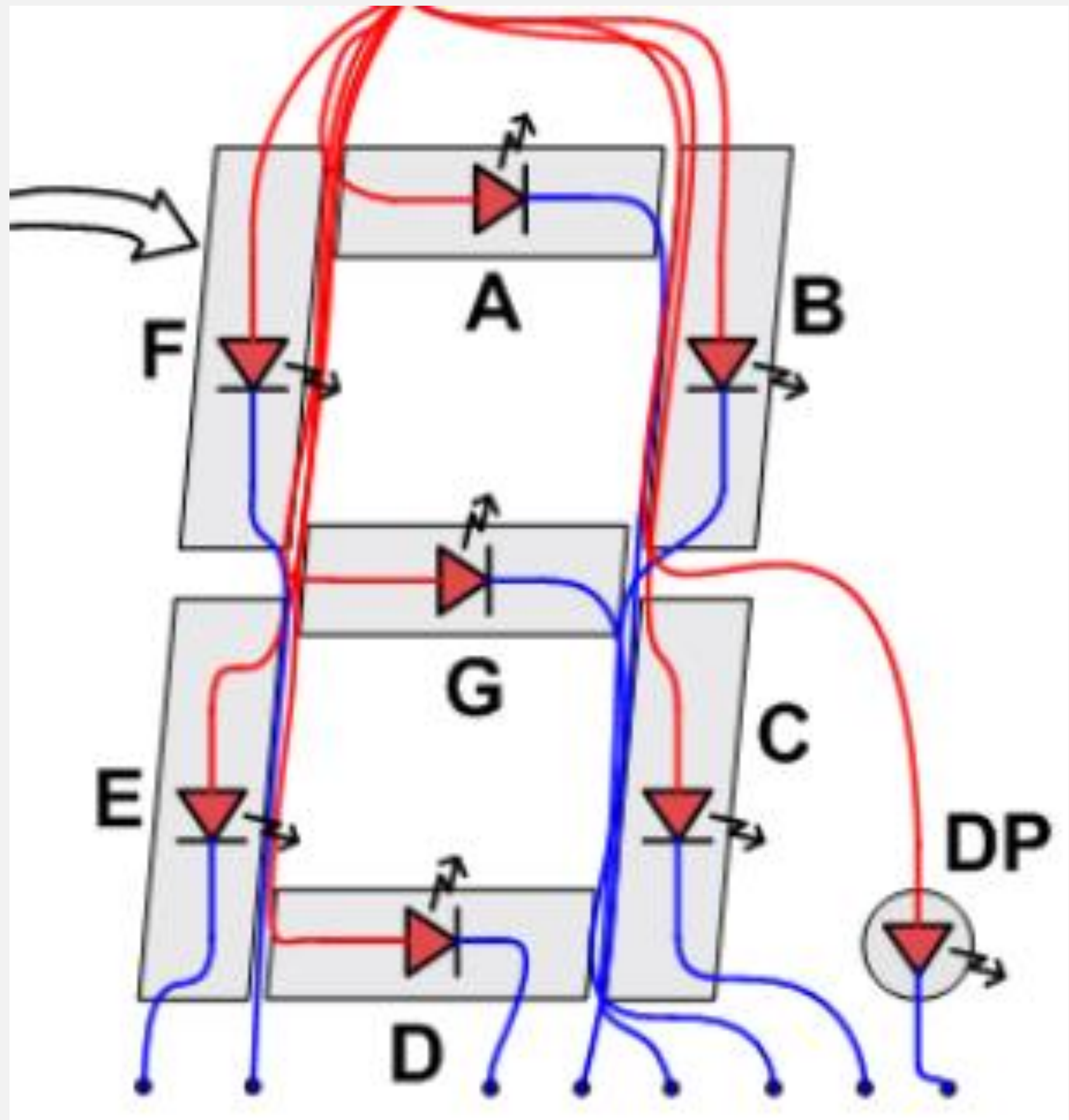
이번 설계의 전체적인 block diagram입니다.

입력 a, b가 adder를 통해 sum, carry를 발생시키고, carry가 발생하면 LED0이 on이 됩니다. 그리고 sum은 fnd_controller로 들어가 받은 data 값을 1의 자리부터 1000의 자리까지 나누어서 mux를 통해 bcd module에게 전달하고, bcd 내부 input 값에 따라 fnd_data가 0~9의 숫자 모양으로 display에 표시가 되는 동작을 보여줍니다.

그리고 clk과 counter를 통해 해당 자리 수에 불이 켜지도록 설계하였고, clk_divider를 사용하여 사람의 눈에 fnd_data가 보이도록 설계하였습니다.

Block Diagram, RTL Schematic



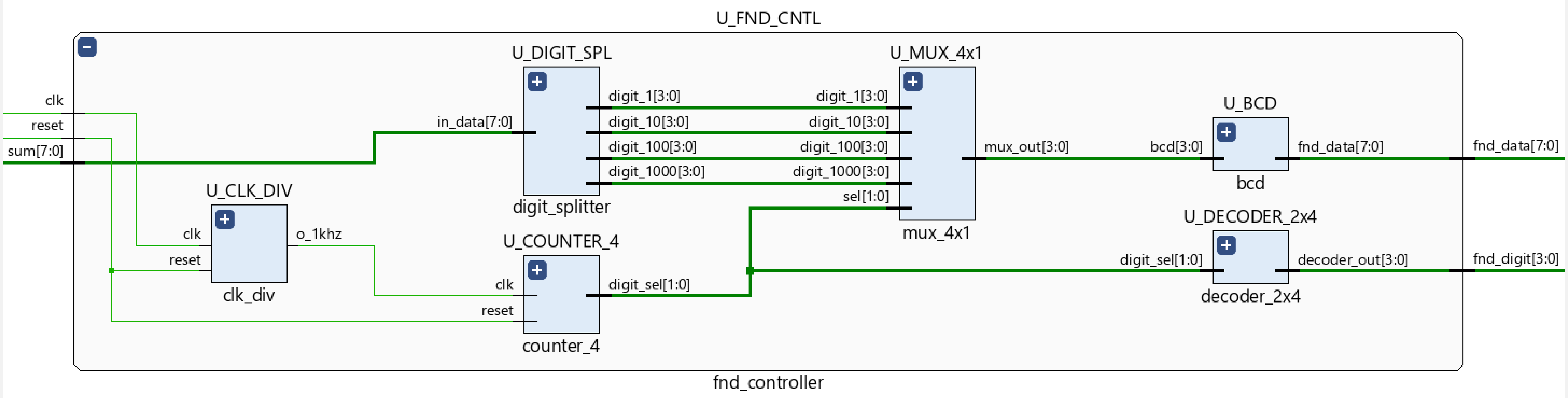


이미지 출처: Basys 3 Reference Manual
https://digilent.com/reference/programmable-logic/basys-3/reference-manual

이번 설계의 전체적인 RTL schematic입니다.

입력 a와 b는 각각 8bit로 4bit adder 2개로 구성된 8bit adder에서 c와 8bit sum값을 받습니다.

fnd_controller의 input 값 역시 8bit로 받아오고, 1의 자리부터 1000의 자리까지의 수를 표현하기 위해 digit를 4bit로 사용하였고,
오른쪽 이미지의 A, B, C, D, E, F, G, DP의 값으로 7-segment 값을 표현하기 위해 8bit의 fnd_data를 사용하였습니다.



fnd_controller unit을 좀 더 자세히 보겠습니다.

우선 기본적인 동작은 adder에서 받아 온 data를 각 자릿수 마다 0에서 9까지 표현하기 위해 digit_splitter로 4bit씩 나눠서 mux의 sel 입력을 통해 bcd가 fnd_data를 처리하는 동작과 같은 sel 입력으로 2x4 decoder를 통해 각 자릿수의 불이 켜지도록 하는 4bit fnd_digit 동작입니다.

이러한 sel 입력은 clk 입력을 받는 counter_4로 만들었고, system clk에 맞춰서 00, 01, 10, 11순으로 count를 하며 이에 따라 fnd_data, digit의 값이 다르게 반복되는 방식입니다.
하지만 이러한 방식은 너무 빠른 system clk의 주파수로 인해 사람의 눈으로는 board의 모든 불이 다 켜진 것 처럼 보이는 현상이 발생하였고, 이를 해결하기 위해 100Mhz인 clk을 1khz로 낮추는 clk_div를 사용해 1khz에 맞춰서 count되어 사람의 눈으로도 정상적인 보드동작을 확인할 수 있었습니다.

3 | RTL Code Review

```verilog
`timescale 1ns / 1ps

module top_adder (
    input         clk,
    input         reset,
    input   [7:0] a,
    input   [7:0] b,
    output [3:0] fnd_digit,
    output [7:0] fnd_data,
    output        c
);

    wire [7:0] w_sum;

    fnd_controller U_FND_CNTL (
        .clk       (clk),
        .reset     (reset),
        .sum       (w_sum),
        .fnd_digit(fnd_digit),
        .fnd_data (fnd_data)
    );

    adder_8bit U_ADDER (
        .a   (a),
        .b   (b),
        .sum(w_sum),
        .c   (c)
    );

endmodule
```

8bit adder와 fnd_controller를 통합하는 top_adder module을 보겠습니다.

입력으로는 clk, reset과 adder의 a, b가 있고,
출력으로는 adder의 결과 carry와 display에 표시하기 위한 fnd_data, fnd_digit가 있습니다.

중점으로 봐야 할 부분은
wire [7:0] w_sum; 이고,

그 이유는 adder의 sum이 fnd_controller의 입력 sum으로 들어오기 때문에
8bit adder의 연산 값인 8bit sum을 fnd_controller의 8bit 입력 sum으로 받아야 하고,
그래서 그 둘을 연결해 줄 wire type의 w_sum도 같이 8bit로 연결해준 모습입니다.

```verilog
`timescale 1ns / 1ps

module fnd_controller (
    input        clk,
    input        reset,
    input  [7:0] sum,
    output [3:0] fnd_digit,
    output [7:0] fnd_data
);

    wire [3:0] w_digit_1, w_digit_10, w_digit_100, w_digit_1000, w_mux_4x1_out;
    wire [1:0] w_digit_sel;
    wire w_1khz;

    digit_splitter U_DIGIT_SPL (
        .in_data   (sum),
        .digit_1   (w_digit_1),
        .digit_10  (w_digit_10),
        .digit_100 (w_digit_100),
        .digit_1000(w_digit_1000)
    );

    clk_div U_CLK_DIV (
        .clk   (clk),
        .reset (reset),
        .o_1khz(w_1khz)
    );
    counter_4 U_COUNTER_4 (
        .clk      (w_1khz),
        .reset    (reset),
        .digit_sel(w_digit_sel)
    );

    decoder_2x4 U_DECODER_2x4 (
        .digit_sel  (w_digit_sel),
        .decoder_out(fnd_digit)
    );

    mux_4x1 U_MUX_4x1 (
        .sel       (w_digit_sel),
        .digit_1   (w_digit_1),
        .digit_10  (w_digit_10),
        .digit_100 (w_digit_100),
        .digit_1000(w_digit_1000),
        .mux_out   (w_mux_4x1_out)
    );

    bcd U_BCD (
        .bcd     (w_mux_4x1_out),
        .fnd_data(fnd_data)
    );

endmodule
```

fnd_controller module을 보겠습니다.

이 module은 digit_splitter, clk_div, counter_4, decoder_2x4, mux_4x1, bcd 같은 많은 module들을 instance화 하고 있고,

각각의 module 끼리의 연결을 위한 wire도 bit에 맞게 알맞게 설정되어 있는 모습을 볼 수 있습니다.

```verilog
module clk_div (
    input       clk,
    input       reset,
    output reg o_1khz
);

    //reg [16:0] counter_r;
    reg [$clog2(100_000)-1:0] counter_r;

    always @(posedge clk, posedge reset) begin
        if (reset) begin
            counter_r <= 0;
            o_1khz    <= 1'b0;
        end else begin
            if (counter_r == 99999) begin
                counter_r <= 0;
                o_1khz    <= 1'b1;
            end else begin
                counter_r <= counter_r + 1;
                o_1khz    <= 1'b0;
            end
        end
    end

endmodule
```

clk_div module을 보겠습니다.

이 module은 system clk인 100Mhz clk을 1khz clk로 바꿔주는 역할을 합니다.

세부 동작은 100Mhz clk이 발생할 때 마다 내부 counter인 17bit counter_r가 올라갑니다.
1khz를 표현하기 위해선 100_000번에 1번 output이 나와서 이를 clk처럼 사용하게끔 해야 합니다.

초기화를 위해 reset이 1이면 모든 값을 0으로 초기화 하고,
counter_r이 0부터 계속 올라가 99999가 되면 출력인 o_1khz가 1이 됩니다. 이를 반복하여
100_000번에 한 번 output이 1이 되고,

결과적으로 100Mhz가 1khz가 되는 clk_divider의 역할을 하게 됩니다.

- $clog2(value)를 사용하면 해당 value를 다 담을 수 있는 최소 bit 값을 계산해줍니다.
  0부터 계산하기 때문에 뒤에 -1을 붙였습니다.

```
module counter_4 (
    input        clk,
    input        reset,
    output [1:0] digit_sel
);

    reg [1:0] counter_r;

    assign digit_sel = counter_r;

    always @(posedge clk, posedge reset) begin
        if (reset) begin
            counter_r <= 0;  // init counter_r
        end else begin
            // to do
            counter_r <= counter_r + 1;
        end
    end
endmodule

module digit_splitter (
    input  [7:0] in_data,
    output [3:0] digit_1,
    output [3:0] digit_10,
    output [3:0] digit_100,
    output [3:0] digit_1000
);

    assign digit_1    = in_data % 10;
    assign digit_10   = (in_data / 10) % 10;
    assign digit_100  = (in_data / 100) % 10;
    assign digit_1000 = (in_data / 1000) % 10;

endmodule
```

counter_4와 digit_splitter module을 보겠습니다.

counter_4 module은 뒤에 나올 4x1_mux, fnd_digit를 위해 사용한 module 이고,
간단한 동작은 clk이 오면 counter 값을 하나씩 더하는 동작을 수행합니다.
2bit를 사용하였기에 0에서 3까지 count가 됩니다.

여기서 사용된 clk은 앞에서 나눴던 1khz clk을 사용합니다.

digit_splitter module은 adder에서 온 sum data를 나눗셈과 나머지 연산자(/, %)를 사용하여
각각의 자릿수의 값을 구해주는 연산기 module입니다.

0~9까지 data를 받아야 하기 때문에 각 자릿수 모두 4bit의 data값을 가집니다.

```verilog
module decoder_2x4 (
    input       [1:0] digit_sel,
    output reg [3:0] decoder_out
);

    always @(digit_sel) begin
        case (digit_sel)
            2'b00: decoder_out = 4'b1110;
            2'b01: decoder_out = 4'b1101;
            2'b10: decoder_out = 4'b1011;
            2'b11: decoder_out = 4'b0111;
        endcase
    end
endmodule

module mux_4x1 (
    input       [1:0] sel,
    input       [3:0] digit_1,
    input       [3:0] digit_10,
    input       [3:0] digit_100,
    input       [3:0] digit_1000,
    output reg [3:0] mux_out
);

    always @(*) begin
        case (sel)
            2'b00: mux_out = digit_1;
            2'b01: mux_out = digit_10;
            2'b10: mux_out = digit_100;
            2'b11: mux_out = digit_1000;
        endcase
    end
endmodule
```

```verilog
module bcd (
    input      [3:0] bcd,
    output reg [7:0] fnd_data  // always output must reg type
);

    always @(bcd) begin
        case (bcd)
            4'd0: fnd_data = 8'hC0;
            4'd1: fnd_data = 8'hF9;
            4'd2: fnd_data = 8'hA4;
            4'd3: fnd_data = 8'hB0;
            4'd4: fnd_data = 8'h99;
            4'd5: fnd_data = 8'h92;
            4'd6: fnd_data = 8'h82;
            4'd7: fnd_data = 8'hF8;
            4'd8: fnd_data = 8'h80;
            4'd9: fnd_data = 8'h90;
            default: fnd_data = 8'hFF;
        endcase
    end
endmodule
```

마지막으로 decoder_2x4, mux_4x1 그리고 bcd module을 보겠습니다.

decode와 mux 두 module 모두 앞에서 말한 counter_4의 출력 값을 입력으로 사용하고,
decode는 fnd_digit, mux는 bcd와 연결되어 fnd_data의 값을 결정하는 module입니다.
같은 digit_sel 값을 사용하여 case 문으로 해당 module의 출력 값을 제어하는 모습을 가지고 있습니다.
4자리의 digit 표시를 위해 4bit로 decoder의 출력을 결정하였습니다.

bcd 역시 mux에서 온 해당 자릿수의 10진수 값에 따라 fnd_data가 출력되어 display에 0~9 숫자로 표현되게끔 설계하였습니다.

```verilog
`timescale 1ns / 1ps

module tb_top_adder ();
    reg clk, reset;
    reg  [7:0] a;
    reg  [7:0] b;
    wire [3:0] fnd_digit;
    wire [7:0] fnd_data;
    wire       c;

    integer i = 0, j = 0;

    top_adder dut (
        .clk      (clk),
        .reset    (reset),
        .a        (a),
        .b        (b),
        .fnd_digit(fnd_digit),
        .fnd_data (fnd_data),
        .c        (c)
    );

    always #5 clk = ~clk;

    initial begin
        #0;
        clk   = 0;
        reset = 1;
        a     = 8'h00;
        b     = 8'h00;
        #20;
        reset = 0;
        for (i = 0; i < 256; i = i + 1) begin
            for (j = 0; j < 256; j = j + 1) begin
                a = i;
                b = j;
                #1_0;
            end
        end
        #1000;
        $stop;
    end

endmodule
```

다음은 이 모든 module을 통합한 환경인 top_adder가 서로 data를 잘 주고 받는지를 확인하고,
이를 simulation으로 확인하기 위한 testbench 파일인 tb_top_adder modul을 보겠습니다.

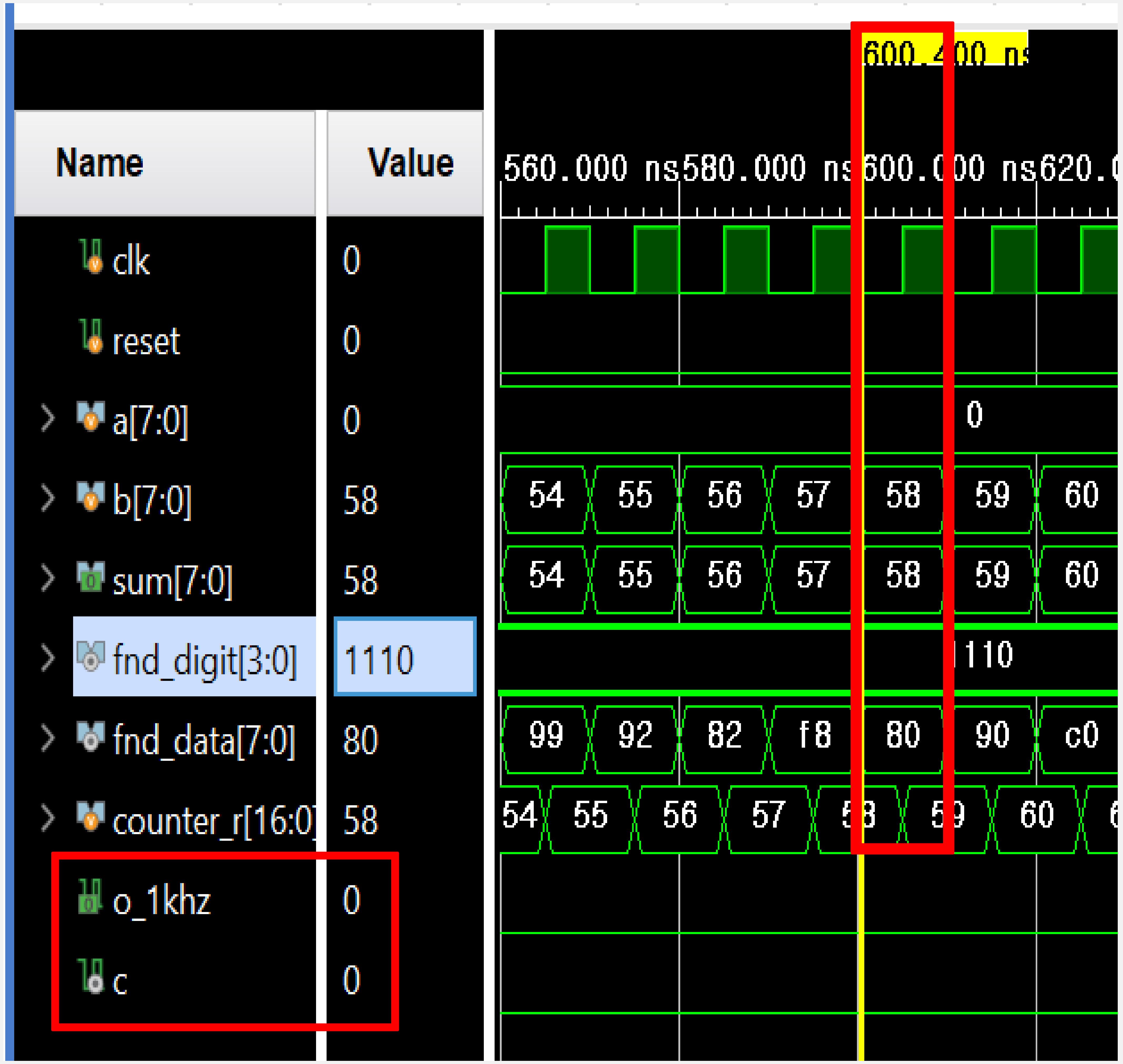
always 문으로 clk을 5ns 마다 반전시켜 총 10ns의 주기를 갖는 clk를 생성하였고,
initial 문으로 전체 simulation을 설계하였습니다.

#0ns에서 모든 input 값을 초기화 시켜주고, 8bit 입력 a, b에 맞춰 full case를 확인하기 위해
for 반복문으로 65,536 가지의 case를 확인하였습니다.

다음에 확인할 simulation 예상 값은
a, b의 값에 따라 sum값이 올바르게 나오는지, count가 100,000 번 마다 o_1khz 가 발생하는지
봐야 하기 때문에 count 값이 99,999에서 0으로 넘어갈 때의 o_1khz 값을 확인할 것입니다.

4 | Simulation Waveform, 동작 구현

Simulation Waveform, 동작 구현



먼저 600ns대에서 simulation을 보겠습니다.

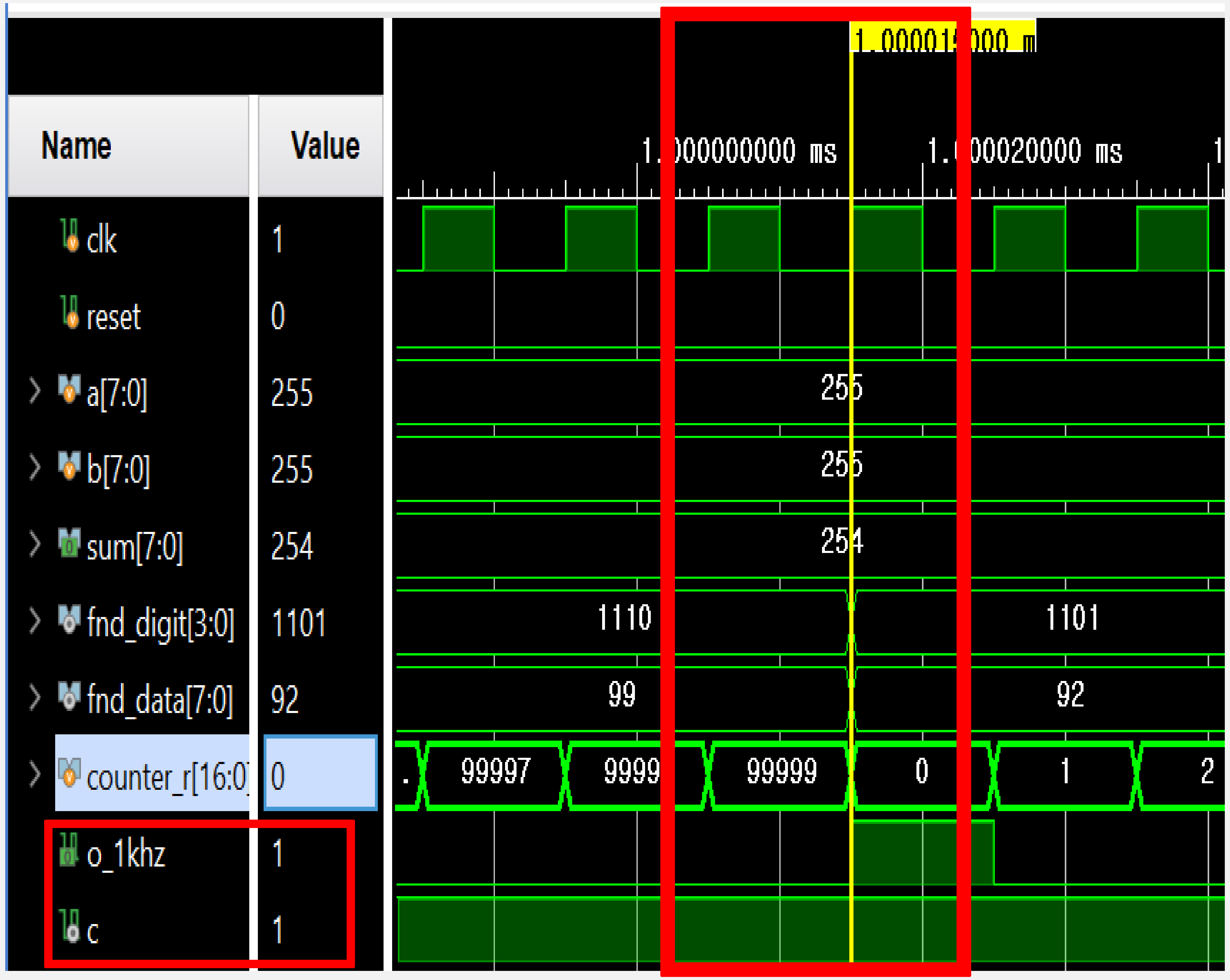
a와 b는 각각 0, 58의 값을 가지고, 둘의 합 sum 역시 58로 정상적인 simulation 값 확인을 할 수 있습니다.

fnd_data는 16진수 80이 나왔는데, 이는 display 상에서 숫자 8을 표현합니다.
fnd_digit가 1110 값을 가지고, 이 의미는 1의 자리수만 불이 켜지게 하겠다는 의미를 담고 있기 때문에 정상적으로 data값이 나오는 것이 확인됩니다.

carry는 8bit의 합이 8bit로 표현되지 않을 때 발생하는데,
0과 58은 8bit로 표현이 가능하고, 그로 인해 발생하지 않았기에
0의 값을 가지는 것을 볼 수 있습니다.

그리고 counter_r은 a, b값과 마찬가지로 system clk에 맞춰서 변하는데,
파형에서 보듯이 58로 count 되는 모습을 볼 수 있습니다.
counter_r이 99,999에서 0이 될 때 o_1khz가 발생하고, 그 1khz clk에 맞춰서
fnd_digit의 값도 변하기 때문에 아직은 o_1khz는 0이고,
fnd_digit도 처음 값인 1110을 계속 유지하는 모습을 보입니다.

Simulation Waveform, 동작 구현



다음은 1ms 시간대를 자세히 보겠습니다.

testbench에서 이제 full case를 확인하여 a와 b는 모두 255의 값을 가지고, sum은 carry가 발생하여 LSB를 제외한 모든 bit의 값이 1이 되어 254의 값을 가지게 됩니다. 이에 따라 carry는 1이 발생하게 됩니다.

그리고 counter_r이 0에서부터 99,999까지 1씩 더해졌고, 그 이후에는 0으로 다시 초기화 되는 모습을 보입니다.
단순히 99,999에서 0으로 초기화 되는 것 뿐만 아니라
이 시간대를 보면 1ms -> 1khz 이고, 의도한 대로 simulation이 잘 나온 모습을 보입니다.

fnd_digit도 이제는 1khz에 맞춰 동작하기 때문에 값이
1110에서 1101으로 변한 모습을 볼 수 있습니다.
fnd_data도 이제 10의 자리 수 를 보기 때문에
숫자 4를 의미하는 16'h99가 아닌 숫자 5를 의미하는 16'h92가 되었습니다.

Simulation Waveform, 동작 구현

0	0	1	1	1	0	1	0	0	0	0	0	0	0	0	0
a[7]	a[6]	a[5]	a[4]	a[3]	a[2]	a[1]	a[0]	b[7]	b[6]	b[5]	b[4]	b[3]	b[2]	b[1]	b[0]

먼저 simulation에서 보았던 a, b의 합 58의 결과이다. sum 역시 58로 나오는 모습을 보이고, 그 결과가 display에 알맞게 나오게 된다.
여기서 carry는 발생하지 않아 LED0의 불은 켜지지 않는다.

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
a[7]	a[6]	a[5]	a[4]	a[3]	a[2]	a[1]	a[0]	b[7]	b[6]	b[5]	b[4]	b[3]	b[2]	b[1]	b[0]

다음은 simulation에서 보았던 a, b의 합 510의 결과이다. 여기서 sum은 254인데, 그 이유는 carry가 발생하여 LED0에 불이 켜지고, 나머지 bit인 1111_1110이 sum이 되어 display에는 254라는 숫자가 나오게 된다.

5 | 결론

- 이번 설계는 8bit adder 결과 값을 fnd_controller 내부의 clk_div, digit_splitter, counter, bcd 등 다양한 module을 통해 7-segment display에 시각화 하는 설계였습니다.

- 단순 가산기를 넘어서 100Mhz인 system clock을 clk_divider를 사용하여 1khz로 낮추어 사람의 눈으로도 볼 수 있도록 설계하였고, 결과는 display에 의도한 숫자가 나오게 되었습니다.

- Block diagram과 RTL schematic을 통해 전체 module과 하위 module간의 구조와 module간에 서로 데이터를 어떤 식으로 주고 받는지 등을 확인할 수 있었습니다.

- 또한 이를 testbench file을 통해 simulation으로 미리 예상 값을 확인하여 synthesis와 implementation 과정에서 소요되는 시간을 줄이고, 완벽한 상태에서 board 동작 구현을 하였습니다.

감사합니다