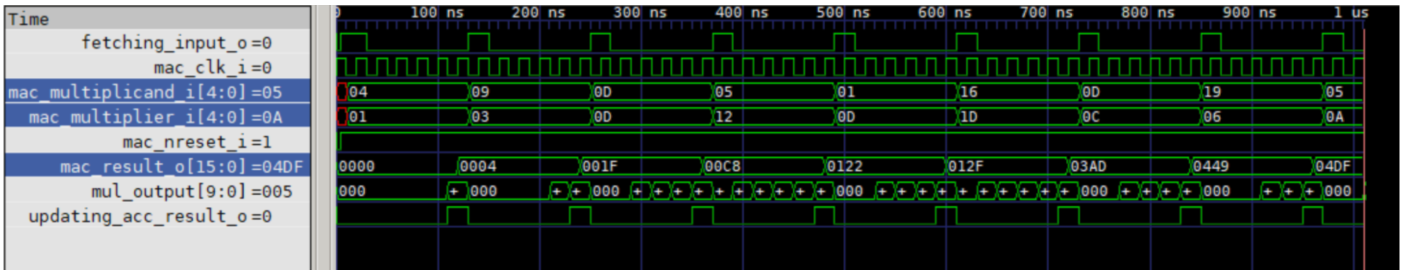
2차 과제 – MAC

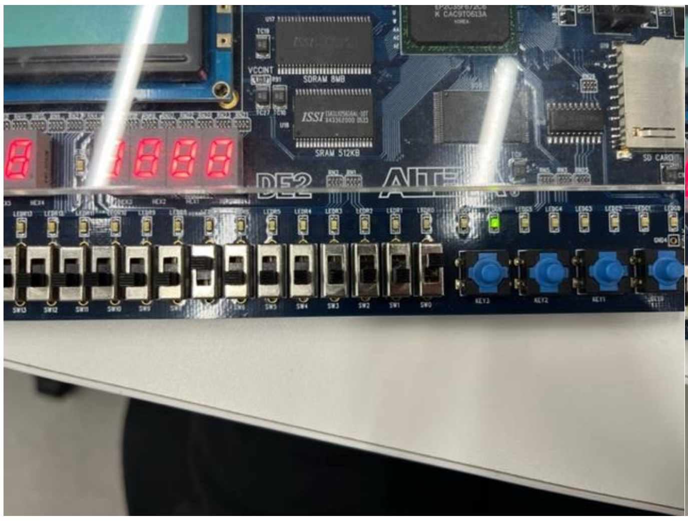
2022104346 정지헌

Simulation Results



Order	Input 1			Input 2			Accumulated Value		
	Bin	Hex	Dec	Bin	Hex	Dec	Bin	Hex	Dec
1	00100	4	4	00001	1	1	100	4	4
2	01001	9	9	00011	3	3	11111	1f	31
3	01101	d	13	01101	d	13	11001000	c8	200
4	00101	5	5	10010	12	18	100100010	122	290
5	00001	1	1	01101	d	13	100101111	12f	303
6	10110	16	22	11101	1d	29	1110101101	3ad	941
7	01101	d	13	01100	c	12	10001001001	449	1097
8	11001	19	25	00110	6	6	10011011111	4df	1247

Synthesis and running on FPGA

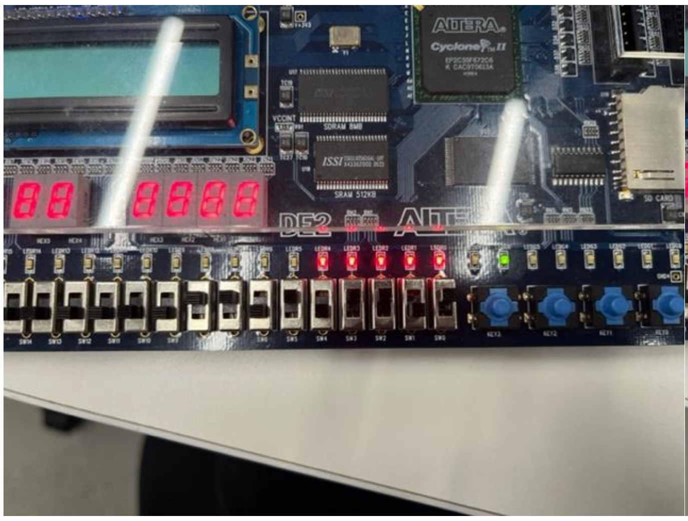


Output: 0

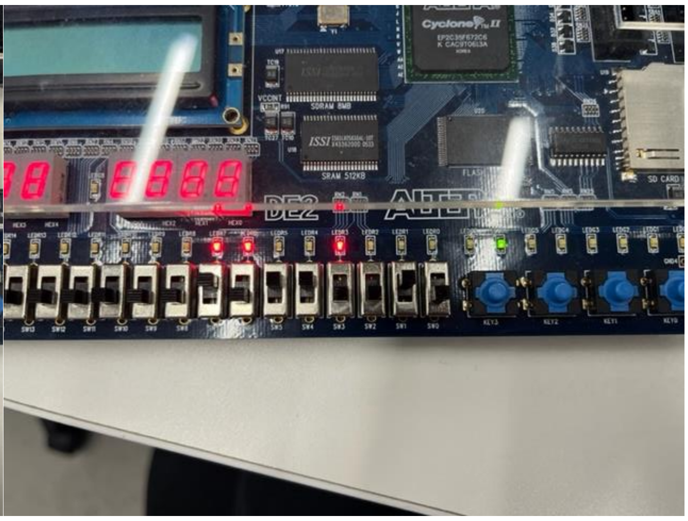
Initial input: 00100, 00001

Output: 100

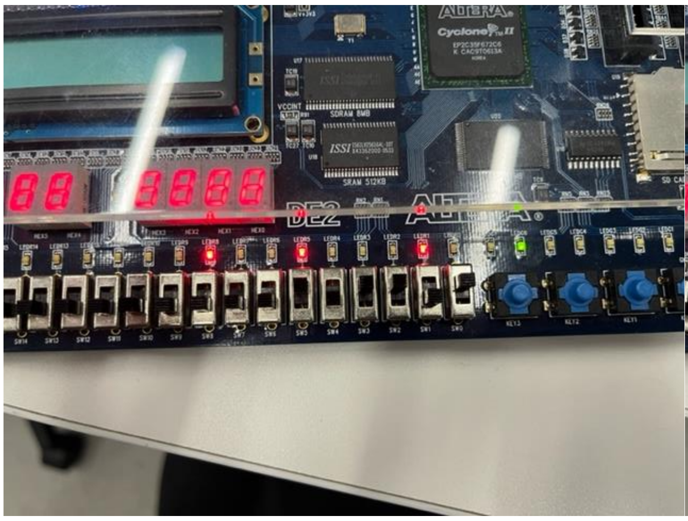
Next input: 01001, 00011


Output: 11111

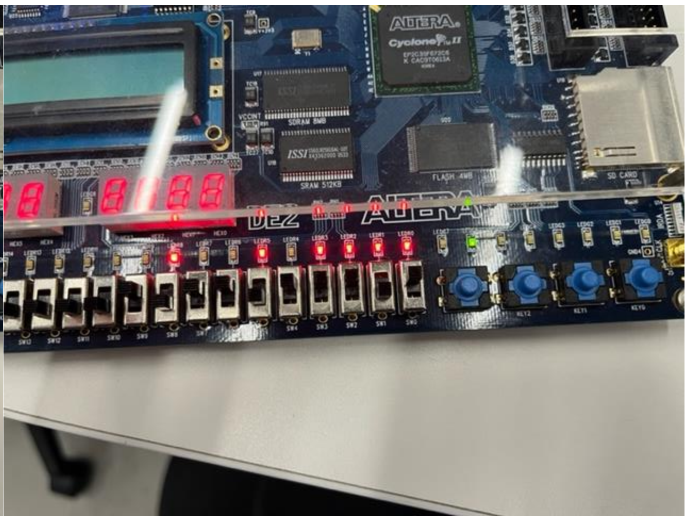
Next input: 01101, 01101


Output: 11001000

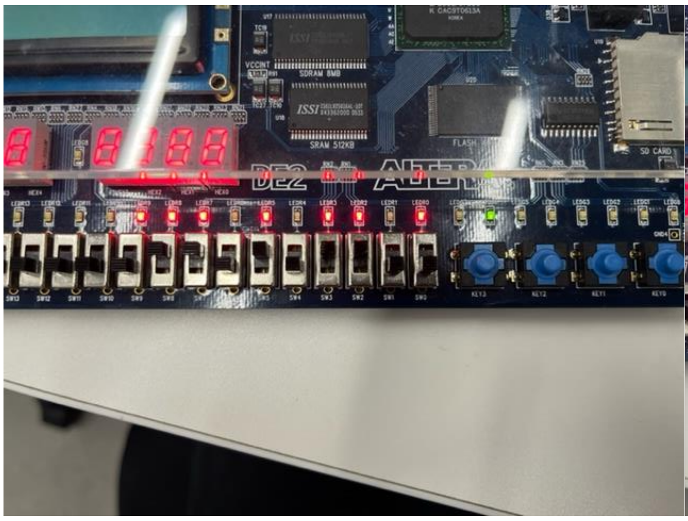
Next input: 00101, 10010


Output: 100100010

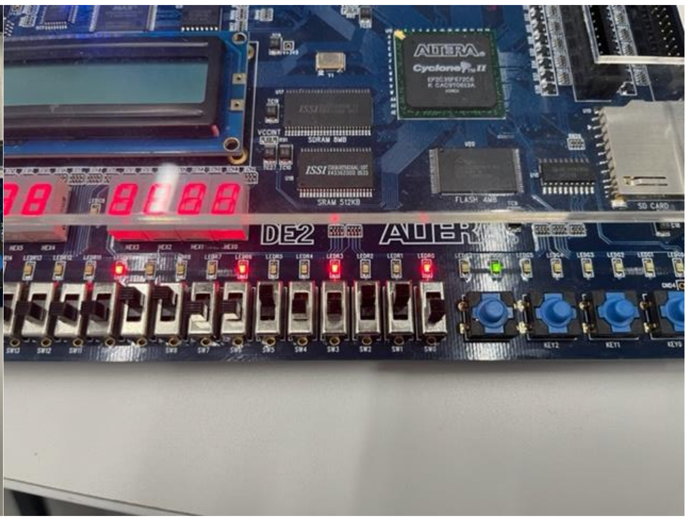
Next input: 00001, 01101


Output: 100101111

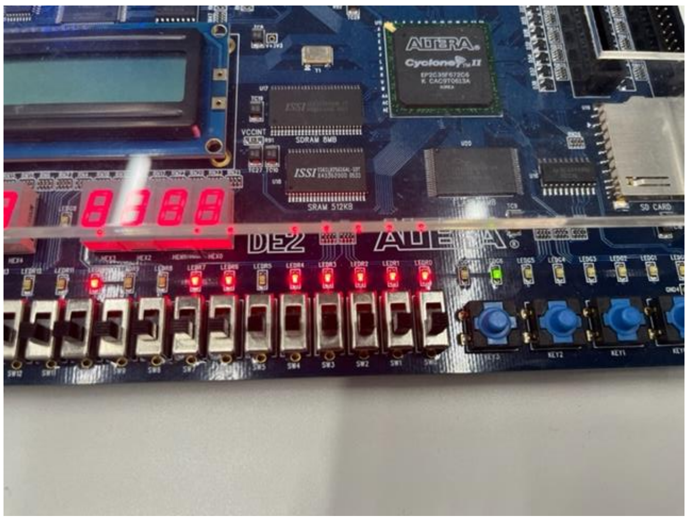
Next input: 10110, 11101


Output: 1110101101

Next input: 01101, 01100


Output: 10001001001

Next input: 11001, 00110



Output: 10011011111

Next input: X, X

Source Code:

https://github.com/wjdwlgjs/Hong_Verilog/tree/main/MAC

MulAndAcc.v

```verilog
`include "MAC/SubModules/SequentialMultiplier.v"
`include "MAC/SubModules/Accumulator16Bit.v"

module MulAndAcc(
    input [4:0] mac_multiplicand_i,
    input [4:0] mac_multiplier_i,
    input mac_clk_i,
    input mac_nreset_i,

    output [15:0] mac_result_o,
    output updating_acc_result_o, // when this turns into 1 at a clk falling edge, the accumulator's output will be updated at the following rising edge
    output fetching_input_o // input values will be read at a clk rising edge when this is 1. It is OK to change the inputs when this is 0
    );


    // Multiplier that, has one 10-bit adder circuit, and sequentially accumulates its partial sums into an array of D flip flops (a.k.a. a register)
    wire [9:0] mul_output;

    SequentialMultiplier main_multiplier_module( // Fetch input at rising edge, let out output at falling edge
        .multiplicand_i(mac_multiplicand_i),
        .multiplier_i(mac_multiplier_i),
        .mul_clk_i(mac_clk_i),
        .mul_nreset_i(mac_nreset_i),
        .mul_result_o(mul_output),
        .is_result_o(updating_acc_result_o),
        .mul_fetching_input_o(fetching_input_o)
    );
    Accumulator16Bit main_accumulator( // update final result when multiplier is done at rising edge
        .new_in_i({6'b000000, mul_output}),
        .acc_clk_i(mac_clk_i),
        .acc_nreset_i(mac_nreset_i),
        .sel_i(updating_acc_result_o),
        .acc_result_o(mac_result_o)
    );

endmodule
```

Testbench

```
`timescale 1ns/1ns
`include "MAC/MulAndAcc.v"

module tb_MulAndAcc();
    reg [4:0] tb_multiplicand;
    reg [4:0] tb_multiplier;
    reg tb_clk;
    reg tb_nreset;

    wire [15:0] tb_result;
    wire tb_updating_acc_result;
    wire tb_fetching_input;

    MulAndAcc TestMAC(
        .mac_multiplicand_i(tb_multiplicand),
        .mac_multiplier_i(tb_multiplier),
        .mac_clk_i(tb_clk),
        .mac_nreset_i(tb_nreset),

        .mac_result_o(tb_result),
        .updating_acc_result_o(tb_updating_acc_result),
        .fetching_input_o(tb_fetching_input)
    );

    always @(negedge tb_clk) #1 begin
        if (tb_fetching_input) begin
            tb_multiplicand = $random;
            tb_multiplier = $random;
        end
    end


    initial begin
        $dumpfile("MAC/BuildFiles/tb_MulAndAcc.vcd");
        $dumpvars(0, tb_MulAndAcc);

        tb_clk = 1;
        tb_nreset = 0;

        #5
        tb_nreset = 1;

        #5
        tb_clk = 0;


        repeat (100) begin
            #10 tb_clk = ~tb_clk;
        end

    end
endmodule
```

SequentialMultiplier.v

```verilog
`include "MAC/SubModules/Adders.v"
`include "MAC/SubModules/Registers.v"
`include "MAC/SubModules/Counter3Bit.v"

module SequentialMultiplier(
    input [4:0] multiplicand_i,
    input [4:0] multiplier_i,
    input mul_clk_i,
    input mul_nreset_i, // nreset signal should only rise when clk is at 0

    output [9:0] mul_result_o,
    output is_result_o,
    output mul_fetching_input_o
    );
    // {input fetching, bitwise ANDing multiplier digit and multiplicand, and addition} is done on rising edge clk
    // partial sum register update is done on clk falling edge

    wire input_fetched; // supresses counter (keeps count at 0, as input registers will read inputs at rising edge when count = 0) until a rising edge is met while nreset = 1
    // without this, if the nreset input rises while clk is at 1, multiplication sequence will begin without fetching the inputs
    PosedgeDFF fetching_complete_checker(
        .d_i(1'b1),
        .enable_i(mul_clk_i),
        .nreset_i(mul_nreset_i),
        .q_o(input_fetched)
    );

    // counter
    wire [2:0] count;
    wire modulo_not_reached;
    wire count_iszero;

    assign modulo_not_reached = ~(count[2] & count[1] & ~count[0]); // create a short falling pulse signal when count == 6, resetting everything including the counter
    assign count_isnotzero = count[2] | count[1] | count[0];

    Counter3Bit main_counter( // consists of falling edge triggered DFFs
        .pulse_i(mul_clk_i),
        .count_nreset_i(input_fetched & modulo_not_reached),
        .count_o(count)
    );

    // PI-SO register made up of D FFs to fetch the multiplier
    wire current_multiplier_digit;

    RegisterPISO5Bit multiplier_fetch( // consists of rising edge triggered DFFs
        .pdi_i(multiplier_i),
        .reg_clk_i(mul_clk_i),
        .reg_nreset_i(mul_nreset_i),
        .shift_sel_i(count_isnotzero), // shift right if count !=0 , fetch parellel input
(multiplier_i) if count == 0
        .sdo_o(current_multiplier_digit)
    );

    // PI-PO register made up of D FFs to fetch the multiplicand

    wire [4:0] current_multiplicand;
    RegisterPIPO5Bit multiplicand_fetch( // consists of rising edge triggered DFFs
        .pdi_i(multiplicand_i),
        .reg_clk_i(mul_clk_i),
        .reg_nreset_i(mul_nreset_i),
        .pdi_sel_i(~count_isnotzero), // keep the current thing if count != 0. I made
this port to avoid clk gating
        .pdo_o(current_multiplicand)
    );
    /* could have just used somthing like

    reg [4:0] current_multiplicand;
    always @(posedge mul_clk_i)
        current_multiplicand = multiplicand_i;

    for all the registers, but I assumed the point of the assignment as to make use of
flip-flops so.. */

    // main adder
    wire [9:0] adder_operand_a;
    wire [9:0] adder_operand_b;
    wire [9:0] adder_output;

    assign adder_operand_a = {5'b00000, {5{current_multiplier_digit}} &
current_multiplicand};

    Adder10Bit main_adder(
        .a10_i(adder_operand_a),
        .b10_i(adder_operand_b),
        .cin10_i(1'b0),
        .sum10_o(adder_output),
        .cout10_o()
    );

    // PI-PO register made up of D FFs for partial sum

    RegisterPIPO10BitNeg partial_sum_reg( // consists of falling edge triggered DFFs
        .pdi_i(adder_output),
        .reg_clk_i(mul_clk_i),
        .reg_nreset_i(mul_nreset_i & modulo_not_reached),
        .pdi_sel_i(1'b1),
        .pdo_o(mul_result_o)
    );

    // shift and feed back
    assign adder_operand_b = {mul_result_o[8:0], 1'b0};

    // a signal to indicate that the output is indeed the final product
    assign is_result_o = count[2] & ~count[1] & count[0];
    // a signal to tell the outside world that input reading is in progress
    assign mul_fetching_input_o = mul_nreset_i & ~count_isnotzero;

endmodule
```

Accumulator16Bit.v

```
module Accumulator16Bit(
    input [15:0] new_in_i,
    input acc_clk_i,
    input acc_nreset_i,
    input sel_i, // fetch and accumulate input if 1, hold the current thing if 0
    output [15:0] acc_result_o
    );

    wire [15:0] reg_input;

    Adder16Bit main_adder(
        .a16_i(new_in_i),
        .b16_i(acc_result_o),
        .cin16_i(1'b0),
        .sum16_o(reg_input),
        .cout16_o()
    );

    RegisterPIPO16Bit main_register(
        .pdi_i(reg_input),
        .reg_clk_i(acc_clk_i),
        .reg_nreset_i(acc_nreset_i),
        .pdi_sel_i(sel_i),
        .pdo_o(acc_result_o)
    );

endmodule
```

DFFs.v

```
module PosedgeDFF( //posedge triggered
    input d_i,
    input enable_i,
    input nreset_i, // asynchronous low level reset
    output reg q_o
    );

    always @(posedge enable_i or negedge nreset_i) begin
        if (nreset_i == 1'b0)
            q_o <= 1'b0;

        else
            q_o <= d_i;
    end
endmodule

module NegedgeDFF(
    input d_i,
    input enable_i,
    input nreset_i, // asynchronous low level reset
    output reg q_o
    );

    always @(negedge enable_i or negedge nreset_i) begin
        if (nreset_i == 1'b0)
            q_o <= 1'b0;

        else
            q_o <= d_i;
    end
endmodule
```

Registers.v

```
`include "MAC/SubModules/DFFs.v"

module RegisterPIPO5Bit( // parellel input, parellel output posedge-triggered register
    input [4:0] pdi_i,
    input reg_clk_i,
    input reg_nreset_i,
    input pdi_sel_i, // keep the current thing if 0. This is to avoid clk gating
    output [4:0] pdo_o
    );

    genvar i;

    generate
        for (i = 0; i < 5; i = i + 1) begin
            PosedgeDFF dff_inst(
                .d_i((pdi_i[i] & pdi_sel_i) | (pdo_o[i] & ~pdi_sel_i)),
                .enable_i(reg_clk_i),
                .nreset_i(reg_nreset_i),
                .q_o(pdo_o[i])
            );
        end
    endgenerate

endmodule

module RegisterPISO5Bit(
    input [4:0] pdi_i,
    input reg_clk_i,
    input reg_nreset_i,
    input shift_sel_i, // shift right if 1, fetch parellel input if 0

    output sdo_o
    );

    wire [5:0] shift_wires;
    assign sdo_o = shift_wires[5];
    assign shift_wires[0] = shift_wires[5];

    genvar i;

    generate
        for (i = 0; i < 5; i = i + 1) begin
            PosedgeDFF dff_inst(
                .d_i((shift_wires[i] & shift_sel_i) | (pdi_i[i] & ~shift_sel_i)),
                .enable_i(reg_clk_i),
                .nreset_i(reg_nreset_i),
                .q_o(shift_wires[i+1])
            );
        end
    endgenerate

endmodule
```

```
module RegisterPIPO16Bit( // parellel input, parellel output posedge-triggered register
    input [15:0] pdi_i,
    input reg_clk_i,
    input reg_nreset_i,
    input pdi_sel_i, // keep the current thing if 0. This is to avoid clk gating
    output [15:0] pdo_o
    );

    genvar i;

    generate
        for (i = 0; i < 16; i = i + 1) begin
            PosedgeDFF dff_inst(
                .d_i((pdi_i[i] & pdi_sel_i) | (pdo_o[i] & ~pdi_sel_i)),
                .enable_i(reg_clk_i),
                .nreset_i(reg_nreset_i),
                .q_o(pdo_o[i])
            );
        end
    endgenerate

endmodule

module RegisterPIPO10BitNeg( // parellel input, parellel output Negedge-triggered
register
    input [9:0] pdi_i,
    input reg_clk_i,
    input reg_nreset_i,
    input pdi_sel_i, // keep the current thing if 0. This is to avoid clk gating
    output [9:0] pdo_o
    );

    genvar i;

    generate
        for (i = 0; i < 10; i = i + 1) begin
            NegedgeDFF dff_inst(
                .d_i((pdi_i[i] & pdi_sel_i) | (pdo_o[i] & ~pdi_sel_i)),
                .enable_i(reg_clk_i),
                .nreset_i(reg_nreset_i),
                .q_o(pdo_o[i])
            );
        end
    endgenerate

endmodule
```

Adders.v

```verilog
module SingleBitFA(
    input a_i,
    input b_i,
    input cin_i,

    output sum_o,
    output cout_o
    );

    assign sum_o = a_i ^ b_i ^ cin_i;
    assign cout_o = a_i & b_i | a_i & cin_i | b_i & cin_i;

endmodule

module Adder16Bit(
    input [15:0] a16_i,
    input [15:0] b16_i,
    input cin16_i,

    output [15:0] sum16_o,
    output cout16_o
    );

    wire [16:0] carry;
    assign carry[0] = cin16_i;
    assign cout16_o = carry[16];

    genvar i;

    generate
        for (i = 0; i < 16; i = i + 1) begin
            SingleBitFA fa_inst(
                .a_i(a16_i[i]),
                .b_i(b16_i[i]),
                .cin_i(carry[i]),
                .sum_o(sum16_o[i]),
                .cout_o(carry[i+1])
            );
        end
    endgenerate

endmodule

module Adder10Bit(
    input [9:0] a10_i,
    input [9:0] b10_i,
    input cin10_i,

    output [9:0] sum10_o,
    output cout10_o
    );
    wire [10:0] carry;
    assign carry[0] = cin10_i;
    assign cout10_o = carry[10];

    genvar i;

    generate
        for (i = 0; i < 10; i = i + 1) begin
            SingleBitFA fa_inst(
                .a_i(a10_i[i]),
                .b_i(b10_i[i]),
                .cin_i(carry[i]),
                .sum_o(sum10_o[i]),
                .cout_o(carry[i+1])
            );
        end
    endgenerate

endmodule
```

Counter3Bit.v

```
// `include "MAC/SubModules/DFFs.v"

module Counter3Bit( // Negative Edge Counter
    input pulse_i,
    input count_nreset_i,
    output [2:0] count_o
    );

    NegedgeDFF dff_firstbit(
        .d_i(~count_o[0]),
        .enable_i(pulse_i),
        .nreset_i(count_nreset_i),
        .q_o(count_o[0])
    );

    NegedgeDFF dff_secbit(
        .d_i(~count_o[1]),
        .enable_i(count_o[0]),
        .nreset_i(count_nreset_i),
        .q_o(count_o[1])
    );

    NegedgeDFF dff_thirdbit(
        .d_i(~count_o[2]),
        .enable_i(count_o[1]),
        .nreset_i(count_nreset_i),
        .q_o(count_o[2])
    );

endmodule
```