

ECE4530 Digital Hardware Design

Project 01

ALU PROJECT TIPS AND DESIGN GUIDE

1. Design using Parameters and component instantiation

Design reuse is one of the major goals of designing using Verilog modeling. Therefore, it is desirable that a module can be customized to some degree to meet specific needs. This design practice is called “*parameterized design*”. The most important parameter to specify is the “width” of the module which can be done using the keyword “*parameter*”.

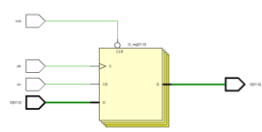
- Parameters are defined in the module port declarations (before the ports) and are treated as constants in the architecture
- Parameters must have a default value that can be changed during instantiation using “parameter map”
- Example 1 – design a generic register with a 32-bit bus width

```

module dff #(parameter WIDTH = 32) (    // define a default bus width
    input [WIDTH-1:0] D,
    input clk, en, n_rst,
    output reg [WIDTH-1:0] Q
);

always @(posedge clk or negedge n_rst)
begin
    if (!n_rst) begin // Low-active reset signal
        Q <= 0;
    end
    else if (en) begin // high-active enable signal
        Q <= D;
    end
end
endmodule

```



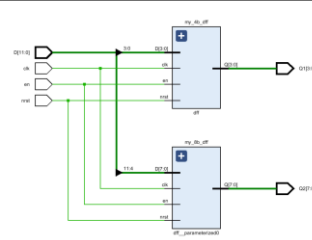
- Example 2 – generate two registers of different widths a 4-bit and an 8-bit from a generic 32-bit register using instantiation

```

module Two_dff(
    input [11:0] D,
    input clk, n_rst, en,
    output [3:0] Q1,
    output [7:0] Q2
);

    dff #(4) my_4b_dff(.D(D[3:0]), .clk(clk), .en(en), .n_rst(n_rst), .Q(Q1));
    dff #(8) my_8b_dff(.D(D[11:4]), .clk(clk), .en(en), .n_rst(n_rst), .Q(Q2));
endmodule

```



2. Design using “parameters” and “generate ... for” Loop

- Design the component you want to replicate a design unit using parameters
- Use a *generate ... for* statement to replicate the desired component any number of times defined as *genvar* value.
- Example 3 – Generate nbit ripple adder from a 1-bit full adder. Notice that the instantiation of a single full adder is used here, where all of the connections are made based on the *genvar* with the parameter WIDTH being used as the controller for size. The label used ‘adder_gen_block’ will then appear once the design is elaborated where each adder will be labeled as such, with a given identification number, indexed from 0. The generated schematic is shown afterwards:

```

module nbit_adder #(parameter WIDTH = 4) (A, B, Cin, Sum, Ov_sgn);
    // the keyword parameter is used with a default value of 4.
    input  [WIDTH-1:0] A, B;
    input  Cin;
    output [WIDTH-1:0] Sum;
    output Ov_sgn;
    wire [WIDTH:0] carry;

    genvar i; /* the for index variable is declared as a genvar to
               be able to elaborate a for loop */
    generate // generate ... for statement
        for (i = 0; i < WIDTH; i = i + 1)
            begin: adder_gen_block
                full_adder FA
                    (.a(A[i]), .b(B[i]), .cin(carry[i]),
                     .sum(Sum[i]), .cout(carry[i+1]));

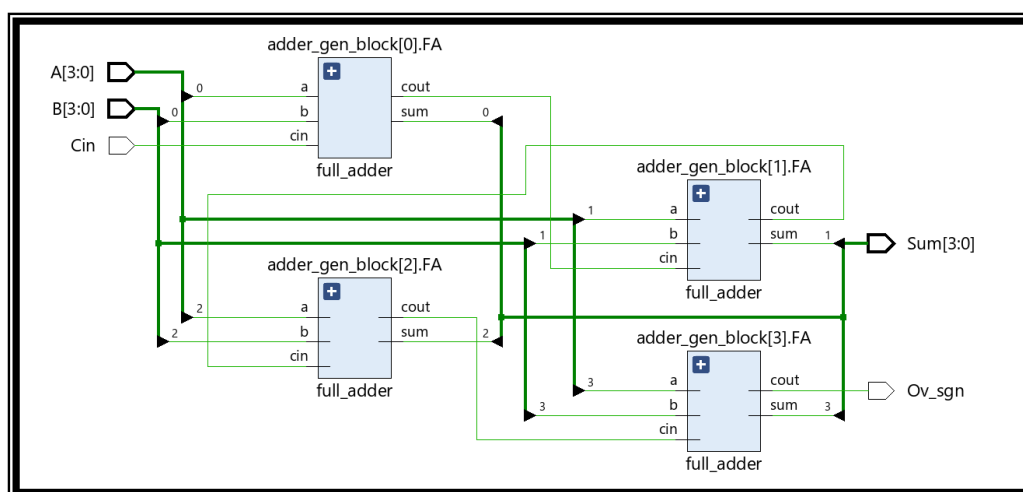
            end
    endgenerate

    /* the following two continuous statements are for the
       initial carry input and the output overflow */

    assign carry[0] = Cin;
    assign Ov_sgn = carry[WIDTH];

endmodule

```



3. Subtractor Design using Two's complement

For the subtract function, you will need to first create a subtractor circuit as shown **Error! Reference source not found.**. In order to do that, you will need to use previously designed components such that full adder, 2's complement circuit, and a Mux. Follow the diagram to understand the operation.

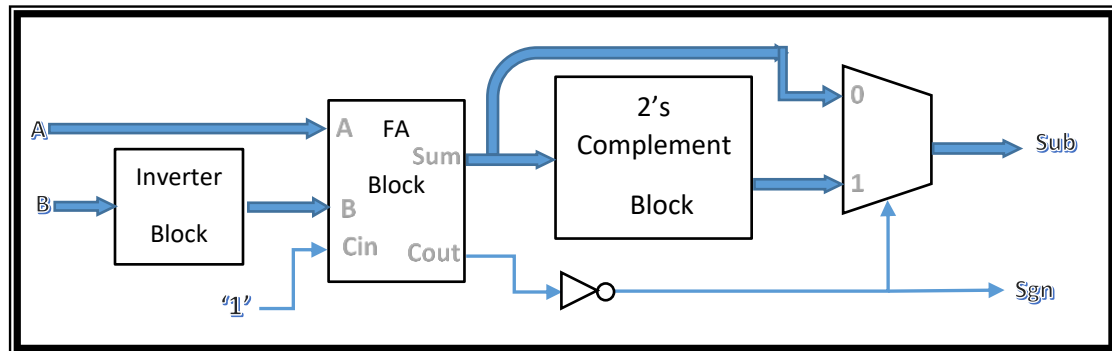


Figure 1 – Full Subtractor Schematic

Note: You should notice after simulating the *Subtractor*, that the output of negative numbers will produce the binary representation of the value as (Sub) and its sign as (Sgn). For example, $0010 - 0011$ should produce 0001 with sign out equal to 1. Review your knowledge of 2's complement addition.

4. Simple 5-bit Two's complement design

A *Two's Complement* function circuit diagram using XOR/OR gate network that outputs the two's complement of a 5-bit number is shown in **Error! Reference source not found.**. The resulting logic is simple, the first bit is always the same, then the following bits will either remain unchanged or be inverted, depending on if any of the bits preceding it are 1's. This design follows the two's complement conversion method and implements it using a cascaded XOR/OR gate chain. This method is defined as: ***“starting from the least significant bit, going towards the most significant bit: copy each 0 until you get to the first 1, copy the first 1, and then invert all other bits”***.

As it can be seen from **Error! Reference source not found.**, the implementation of this function results in a repeatable hardware structure starting from the output of the second bit. Each following bit uses exactly one more OR and one more XOR gate. use the *generate ... for* design for the dotted box shown in figure below to design an n-bit 2's complement circuit with a default value of 32.

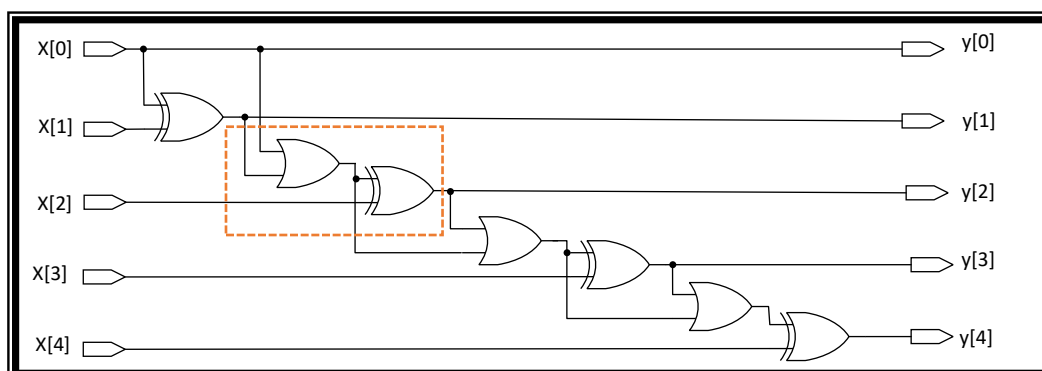


Figure 2 -- Two's Complement Implemented using a series of XOR and OR gates.

5. N-bit magnitude comparator design

An n-bit magnitude comparator is a combinational circuit that compares two unsigned integers. It has two Inputs: 1) Unsigned integer A (n -bit number), and 2) another Unsigned integer B (n -bit number). It generates three outputs: 1) $A == B$ (EQ output), 2) $A > B$ (GT output), and 3) $A < B$ (LT output). As shown in Figure 3. Exactly one of the three outputs must be equal to 1 while the remaining two outputs must be equal to 0.

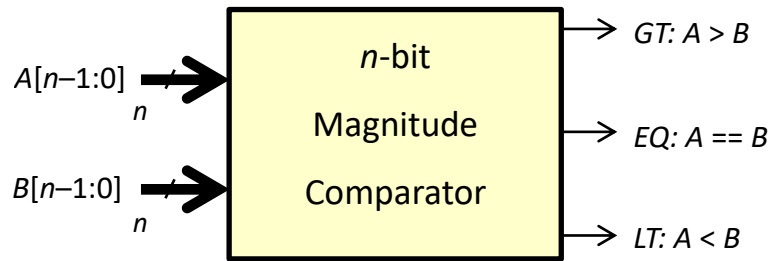


Figure 3 – n-bit magnitude Comparator

An n-bit magnitude comparator can be designed iteratively as a series of connected cells of an identical comparison cell (i) as shown in Figure 4. Each Cell i receives as inputs: Bit i of inputs A and B (A_i and B_i), GT_i , EQ_i , and LT_i from cell $(i - 1)$ and produces three outputs: GT_{i+1} , EQ_{i+1} , and LT_{i+1} .

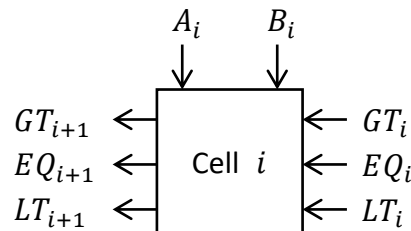


Figure 4 – Comparator cell

The logic functions of the cell can be expressed as shown below as shown below

$EQ_{i+1} = E_i EQ_i$ $GT_{i+1} = A_i B'_i + E_i GT_i$	$E_i = A'_i B'_i + A_i B_i \text{ (} A_i \text{ equals } B_i \text{)}$ $A_i B'_i \text{ (} A_i > B_i \text{)}$
--	--

Use the *generate ... for* design technique using the cell (i) to design an n-bit magnitude comparator block with a default value of 32