

```

module both( //AND, Meagan
    input [3:0] num1,
    input [3:0] num2,
    output reg result);

    //assign result = num1&num2;

    always @ (*)
        begin
            result = num1&num2;
        end

endmodule

```

```

module either( //OR, Meagan
    input [3:0] num1,
    input [3:0] num2,
    output reg result);

    //assign result = num1 | num2;

    always @ (*)
        begin
            result = num1 | num2;
        end

endmodule

```

```

module adder( //Meagan
    input num1,
    input num2,
    input cin,
    output reg result,
    output reg cout);

    //wire p, g;

    reg p, g;

    //assign p = num1^num2;
    //assign g = num1&num2;

    //assign result = p^cin;
    //assign cout = g | (p&cin);

    always @ (*)
        begin
            p = num1^num2;
            g = num1&num2;

            result = p^cin;
            cout = g | (p&cin);
        end

end

```

```
endmodule
```

```
module multibitAdder(  
    input [3:0] number1, number2,  
    output [4:0] sum,  
    output cout);  
  
    wire adder1_o;  
    wire adder2_o;  
    wire adder3_o;  
  
    adder Adder1(  
        .num1(number1[0]),  
        .num2(number2[0]),  
        .cin(1'b0),  
        .result(sum[0]),  
        .cout(adder1_o));  
  
    adder Adder2(  
        .num1(number1[1]),  
        .num2(number2[1]),  
        .cin(adder1_o),  
        .result(sum[1]),  
        .cout(adder2_o));  
  
    adder Adder3(  
        .num1(number1[2]),  
        .num2(number2[2]),  
        .cin(adder2_o),  
        .result(sum[2]),  
        .cout(adder3_o));  
  
    adder Adder4(  
        .num1(number1[3]),  
        .num2(number2[3]),  
        .cin(adder3_o),  
        .result(sum[3]),  
        .cout(sum[4]));  
  
endmodule
```

```
endmodule
```

```
module subtractor( //Jeff  
    input [3:0] num1,  
    input [3:0] num2,  
    output reg [3:0] result);  
  
    always @ (*)  
        begin  
            if(num1 < num2)  
                result = num2 - num1;  
            else if(num1 > num2)  
                result = num1 - num2;  
        end  
  
endmodule
```

end

endmodule

```
module multiplier( //Rhea Mae
    input [3:0] num1,
    input [3:0] num2,
    output reg [6:0] result); //2N Result

    always @ (*)
    begin
        result = num1 * num2;
    end
end
```

endmodule

```
module divider( //Rhea Mae
    input [3:0] num1,
    input [3:0] num2,
    output reg [7:0] result); //2^N-1 Result

    always @ (*)
    begin
        if(num2 == 0)
            result = 0;
        else
            result = num1/num2;
    end
end
```

endmodule

```
module comparator( //Rhea Mae
    input [3:0] num1,
    input [3:0] num2,
    output reg [2:0] result);

    always @ (*)
    begin
        if(num1 == num2)
            result = result + 1;
        if(num1 < num2)
            result = result + 2;
        if(num1 > num2)
            result = result + 4;
    end
end
```

endmodule

```
module mux( //Jeff
    input result1,
    input result2,
    input [4:0] result3,
```

```

input [3:0] result4,
input [6:0] result5,
input [7:0] result6,
input [2:0] result7,
input sel,
output reg [7:0] result_s;

```

```

always @ (*)
    begin
        case(sel)
            0 : result_s = result1;
            1 : result_s = result2;
            2 : result_s = result3;
            3 : result_s = result4;
            4 : result_s = result5;
            5 : result_s = result6;
            6 : result_s = result7;
            default : result_s = 0;
        endcase
    end

```

```

endmodule

```

```

module digit_parser(
    input [7:0] number,
    output reg [3:0] ones,
    output reg [3:0] tens,
    output reg [3:0] hunds,
    output reg [3:0] thous );

    always @ (*)
        begin
            ones = number % 10;
            tens = ((number * 13)/128) % 10;
            hunds = (((number * 13)/128) * 13)/128 % 10;
            thous = ((((((number * 13)/128) * 13)/128) * 13)/128) % 10;
        end

```

```

endmodule

```

```

module output_select(
    input [2:0] select,
    input [3:0] ones,
    input [3:0] tens,
    input [3:0] hunds,
    input [3:0] thous,
    output reg [3:0] data );

    always @ (*)
        begin
            case (select)
                0: data = ones;

```

```

        1: data = tens;
        3: data = hunds;
        4: data = thous;
    endcase
end

endmodule

module state_machine( //example of a Moore type state machine
    input clk_i,
    input reset_n,

    output reg [2:0] select );

    //state and next state registers
    reg [1:0] state;
    reg [1:0] state_n;

    //each possible value of the state register is given a unique name for easier use later
    parameter S0 = 2'b00;
    parameter S1 = 2'b01;
    parameter S2 = 2'b10;
    parameter S3 = 2'b11;

    //asynchronous reset will set the state to the start, S0, otherwise, the state is changed on the positive
    edge of the clock signal
    always @ (posedge clk_i, negedge reset_n)
        begin
            if(!reset_n)
                state = S0;
            else
                state = state_n;

        end

    //this section defines what the next state should be for each possible state. in this implementation, it
    simply rotates through each state automatically
    always @ (*)
        begin
            case(state)
                S0: state_n = S1;
                S1: state_n = S2;
                S2: state_n = S3;
                S3: state_n = S0;

                default: state_n = S0;
            endcase
        end

    //this is the output definition section of the state machine. outputs are based on which state the state
    machine is currently on
    //hex values are used for the LEDs since there are 8 of them and hex values are an effective way of
    represiting this size

```

```

value //numbers are represented by the number of bits (not digits!), an apostrophe, the base system, then the
//common bases are binary, b, octal, o, decimal, d, hexadecimal, h
always @ (*)
begin
    if(state == S0)
        select = 3'b000;
    else if(state == S1)
        select = 3'b001;
    else if(state == S2)
        select = 3'b011;
    else
        select = 3'b100;
end

endmodule

module clock_counter(
    input clk_i,           //often, "tags" are added to variables to denote what they do for the user
    input reset_n,         //here, 'i' is used for input and 'o' for the output, while 'n' specifies an active low signal
    ("not")
    output reg clk_o );

    reg [18:0] count = 0; //register stores the
counter value so that it can be modified on a clock edge. register size needs to store as large of a number as the counter
reaches //for
this implementation, count must reach 415999, so  $2^n \geq 415999$ ,  $n = 19$ 

    always @ (posedge clk_i)
    begin
        count <= count + 1; //at every positive
edge, the counter is increased by 1
        if(!reset_n)
        begin
            clk_o <= 0;
            count <= 0; //if reset
(active low) is pushed, the counter is reset
        end
        else
            if(count >= 5000) //count value of greater
than or equal to this value causes the output clock to be inverted. the resulting frequency will be
input_frequency/(1+count_value)
            begin //for this
implementation, a frequency of 5 Hz was desired, so  $2.08e6/5 - 1 = 415999$ 
            clk_o <= ~clk_o;
            count <= 0; //resets the
counter after the output clock has been inverted
        end
    end
end

```

```
endmodule
```

```
module seven_segment(  
    input [3:0] data,  
    output reg [6:0] segments );  
  
    always @(*)  
        case( data ) // 7'bABCDEFGH  
            0 : segments = 7'b00000001;  
            1 : segments = 7'b10011111;  
            2 : segments = 7'b00100101;  
            3 : segments = 7'b00001110;  
            4 : segments = 7'b10011100;  
            5 : segments = 7'b01001001;  
            6 : segments = 7'b01000000;  
            7 : segments = 7'b00011111;  
            8 : segments = 7'b00000000;  
            9 : segments = 7'b00001110;  
            default : segments = 7'b11111111;  
        endcase  
endmodule
```

```
endmodule
```

```
module fpga(  
    input [3:0] num1,  
    input [3:0] num2,  
    input sel,  
    input reset_n,  
    output [2:0] select,  
    output [6:0] segments );  
  
    wire clock;  
    wire clk;  
    wire [3:0] ones;  
    wire [3:0] tens;  
    wire [3:0] hunds;  
    wire [3:0] thous;  
    wire [3:0] data;  
    wire result1;  
    wire result2;  
    wire [4:0] result3; //wire [5:0] result3  
    wire [3:0] result4;  
    wire [6:0] result5;  
    wire [7:0] result6;  
    wire [2:0] result7;  
    wire [3:0] number1;  
    wire [3:0] number2;  
    wire cout;  
    wire [7:0] number;  
  
    both both(  
        .num1(num1),
```

```
.num2(num2),  
.result(result1));
```

```
either either(  
    .num1(num1),  
    .num2(num2),  
    .result(result2));
```

```
multibitAdder add(  
    .number1(num1),  
    .number2(num2),  
    .sum(result3),  
    .cout(cout));
```

```
subtractor sub(  
    .num1(num1),  
    .num2(num2),  
    .result(result4));
```

```
multiplier multi(  
    .num1(num1),  
    .num2(num2),  
    .result(result5));
```

```
divider div(  
    .num1(num1),  
    .num2(num2),  
    .result(result6));
```

```
comparator com(  
    .num1(num1),  
    .num2(num2),  
    .result(result7));
```

```
mux m(  
    .result1(result1),  
    .result2(result2),  
    .result3(result3),  
    .result4(result4),  
    .result5(result5),  
    .result6(result6),  
    .result7(result7),  
    .sel(sel),  
    .result_s(number));
```

//Untouched Lab 4 Below

```
digit_parser dp(  
    .number(number),  
    .ones(ones),  
    .tens(tens),  
    .hunds(hunds),  
    .thous(thous));
```



```

output_select os(
    .select(select),
    .ones(ones),
    .tens(tens),
    .hunds(hunds),
    .thous(thous),
    .data(data));

//This module is instantiated from another file, 'State_Machine.v'
//It contains a Moore state machine that will take a clock and reset, and output LED combinations
state_machine sm(
    .clk_i(clock),
    .reset_n(reset_n),
    .select(select));

//This module is instantiated from another file, 'Clock_Counter.v'
//It will take an input clock, slow it down based on parameters set inside of the module, and output the new
clock. Reset functionality is also built-in
clock_counter cc(
    .clk_i(clk),
    .reset_n(reset_n),
    .clk_o(clock));

seven_segment ss(
    .data(data),
    .segments(segments));

    OSCH #("2.08") osc_int ( // "2.03" specifies the operating frequency, 2.03
MHz. Other clock frequencies can be found in the MachX02's documentation
    .STDBY(1'b0), // Specifies active state
    .OSC(clk), // Outputs clock signal to 'clk'
net
    .SEDSTDBY());

endmodule

```