ECE275: Sequential Logic Circuits
Lab 6

Pascal Francis-Mezger

October 19, 2020

Contents

0 Lab Overview **2**
0.1 Important New Topic: Concatenation 2

1 Part 1: Convert Binary to BCD and Display on Seven Segement LED **2**
1.1 Create the Double Dabble/Shift-and-Add-3 Algorithm 2
1.1.1 Shift-Add-3 Module 3
1.1.2 Implement the Double Dabble Algorithm 4
1.2 Test Your Results 5

2 Part 2: Create a 10 Second Timer **6**
2.1 Provided Modules 6
2.2 Completing Part 2 8

0 Lab Overview

This lab will cover creating a binary to BCD converter in combinational logic. You will then use this converter to display the decimal equivalent of a binary value that that is input on the switches using the 7 segement LEDs.

The second part of the lab will cover displaying the current timing value on the 7 segment LEDs using a sequential module that is provided for you. You will then be able to compare the combinational implementation to the sequential implementation for the BCD converter.

0.1 Important New Topic: Concatenation

A valuable tool you can use when implementing today's lab is concatenation. It makes it so you can combine multiple individual values into a single array. For example, in the part of this lab where you are utilizing the Shift Add 3 module, you will need to pass in A as an array of 4 bits. You should be familiar with passing in an array of 4 bits that are contiguous, for example switches 0-3 as SW[3:0]. What if instead you wanted to pass in switches 6, 3, 2, and 0? Concatenation allows you to do this. If you place multiple values in curly braces and separate them with commas, Verilog will treat them as a single array. For example, "assign LEDG[3:0] = SW[6],SW[3],SW[2],SW[0]" would treat the switches as a contiguous array and assign the values to the LEDs.

1 Part 1: Convert Binary to BCD and Display on Seven Segement LED

For this first section you will write your own routine to display an 8 bit value on the seven segment LEDs. To accomplish this you will need to create a module to allow you to convert binary inputs from your switches into BCD values. You will be accomplishing this by creating a module to implement the "Double Dabble" algorithm.

1.1 Create the Double Dabble/Shift-and-Add-3 Algorithm

If you would like more information about the Double Dabble Algorithm, you can read the Wikipedia page here: `https://en.wikipedia.org/wiki/Double_dabble`. This page has Verilog code, but you will **NOT** be using their implementation. For this section of the lab you will be writing a combinational implementation, while the implementation shown in the Wikipedia article is a sequential implementation. The difference is use of registers and an always loop, which you will not be using either of for your implementation.

1.1.1 Shift-Add-3 Module

Table 1 is the truth table for the Shift-Add-3 Module. As you have done before, create your POS/SOP, use a Karnaugh map, or whatever method you would like to create your reduced combinational equations from the truth table. Use those to create the Shift-Add-3 Module, giving it whatever relevant name you would like. The module would represent the digital component shown in Figure 1.

$A[3]$	$A[2]$	$A[1]$	$A[0]$	$S[3]$	$S[2]$	$S[1]$	$S[0]$
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	1
0	1	0	0	0	1	0	0
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
1	1	0	0	X	X	X	X
1	1	0	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X

Table 1: Shift Add 3 Truth Table

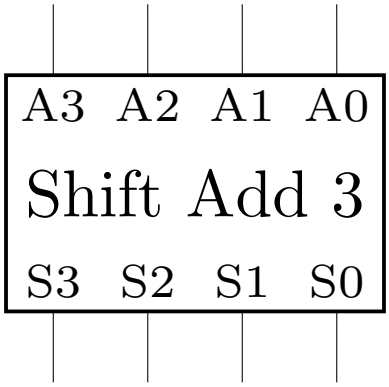

Figure 1: Shift Add 3 Module Digital Representation

Make sure to test your results from the Shift-Add3-Module using switches and LEDs to make sure your results match the truth table before moving to the next section. If your module is not correct and you move on, the troubleshooting will be significantly more difficult later.

1.1.2 Implement the Double Dabble Algorithm

The Double Dabble algorithm can be implemented on an arbitrary amount of input binary bits. For this lab we will be creating an 8 bit BCD converter. The maximum decimal value would be $2^8 - 1 = 255$ so you will need to output 3 BCD digits, which is 12 bits of BCD. The implementation to convert 8 binary bits to 3 digits of BCD (12 bits) is shown in Figure 2.

Write another module that converts 8 binary bits to BCD based on Figure 2. Name the module something relevant, but likely a good name would be along the lines of Binary_to_BCD_8Bit. The module should utilize the Shift-Add-3 module several times, and will need you to use wires for interconnections. If you are fuzzy on how the wires would be used, review Lab 4.

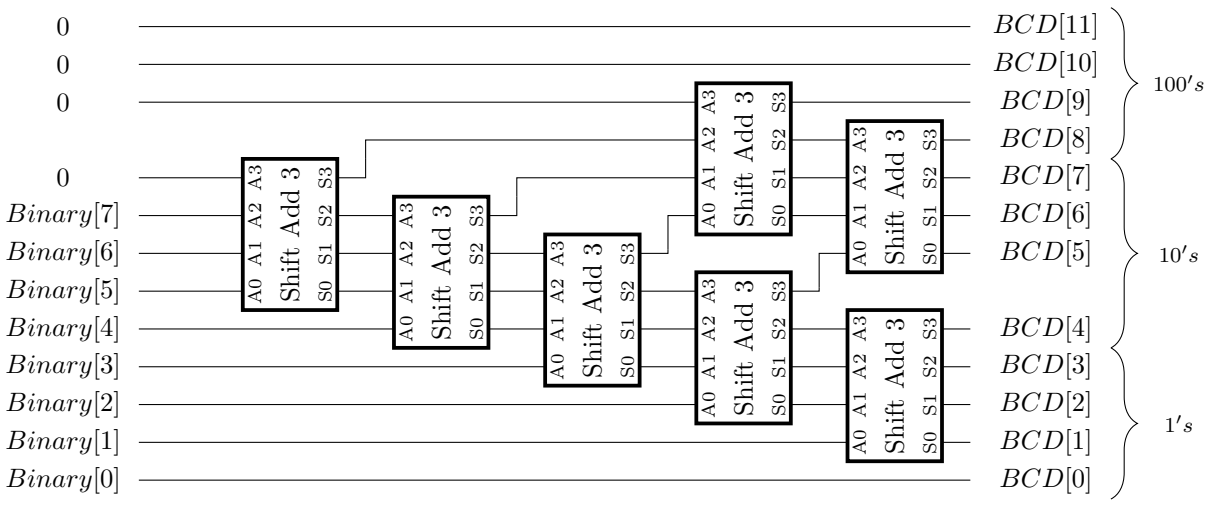

Figure 2: Shift Add 3 Module Digital Representation

1.2 Test Your Results

In your top level, use switches 7-0 to represent your 8 bits to pass into your Double Dabble 8 bit BCD converter. Also, copy and paste in your 7 Segment BCD display module from Lab 3. Use this module to display the decimal equivalent of the binary value of the 8 switches.

2 Part 2: Create a 10 Second Timer

2.1 Provided Modules

In this section you will be provided with 2 modules. One is the timer module which provides an output of the current amount of milliseconds, requires a 50MHz clock input, and has an input for a pause and reset switch. The other is a 16 bit BCD converter, that implements a sequential version of the Double Dabble algorithm. This requires a 16 bit binary input, and outputs 4 digits of BCD (16 bits).

As you have not learned sequential logic yet, it is not too important for you to understand exactly what is going on inside the modules. One thing that you should note, is the code complexity of the 16-bit converter. Does this seem like it is lower code complexity than implementing a 16 bit combinational BCD converter? (Keep in mind you only implemented an 8 bit converter for part 1, you can refer to the Double Dabble Algorithm Wikipedia page to see what a 16 bit version would look like for Shift-Add-3 Modules). You can find the modules below:

```
module timer_module(
        output reg [13:0] milliseconds,
        input Clock_50MHz,
        input pause,
        input reset
);
reg [15:0] difference;
reg [15:0] timer_value;

always @ (posedge Clock_50MHz) begin
        if((reset == 0) && (pause == 0)) begin
                difference = milliseconds - 16'd9999;
                milliseconds = difference?milliseconds:16'b0;
                timer_value = timer_value + 1'b1;
                if(timer_value == 16'b1100001101010000) begin
                        milliseconds = milliseconds+1;
                        timer_value = 16'b0;
                end
        end
        else if (reset == 1)
                milliseconds = 0;
end
endmodule

module BCD_16_Bit_Converter(
        input [15:0] input_binary,
        output reg [15:0] output_BCD,
        input Clock_50MHz
);
        reg [4:0] i;

        always @(Clock_50MHz) begin
                output_BCD = 16'b0;
                for(i=0;i<16;i=i+1) begin
                        output_BCD = {output_BCD[14:0],input_binary[15-i]};
                        if(i<15 && output_BCD[3:0] > 4)
                                output_BCD[3:0] = output_BCD[3:0] + 3;
                        if(i<15 && output_BCD[7:4] > 4)
                                output_BCD[7:4] = output_BCD[7:4] + 3;
                        if(i<15 && output_BCD[11:8] > 4)
                                output_BCD[11:8] = output_BCD[11:8] + 3;
                        if(i<15 && output_BCD[15:12]>4)
                                output_BCD[15:12] = output_BCD[15:12] + 3;
                end
        end
endmodule
```

2.2 Completing Part 2

Instantiate these modules from your main routine in a manner than displays the current time value on the 7 Segement LEDs, and allows you to pause and reset the current time value. You will just need to pass a switch to each the reset and pause inputs to accomplish this. You will be finished with this lab when you can show your TA a timer that can be paused and reset that displays properly on the 7 segments.