ECE275: Sequential Logic Circuits
Lab 6

Pascal Francis-Mezger

October 25, 2021

Contents

0 Lab Overview

The goal of this lab is to create a module to convert binary to BCD using two different methods in Verilog. The first is to utilize your traditional combinational techniques, and the second is to utilize the Verilog procedural operators.

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

For this section you will write a Verilog module based on combinational logic to convert an 8-bit binary value into BCD. 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`.

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. You are welcome to use Verilog operators such as shifting and addition, you just cannot use procedural operators such as the always block.

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

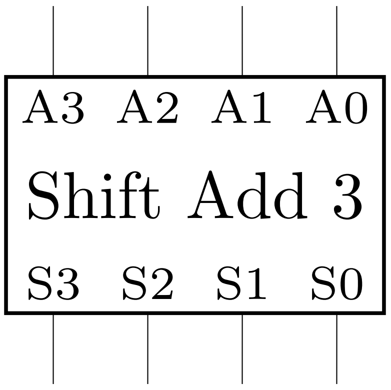

Figure 1: Shift Add 3 Module Digital Representation

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.

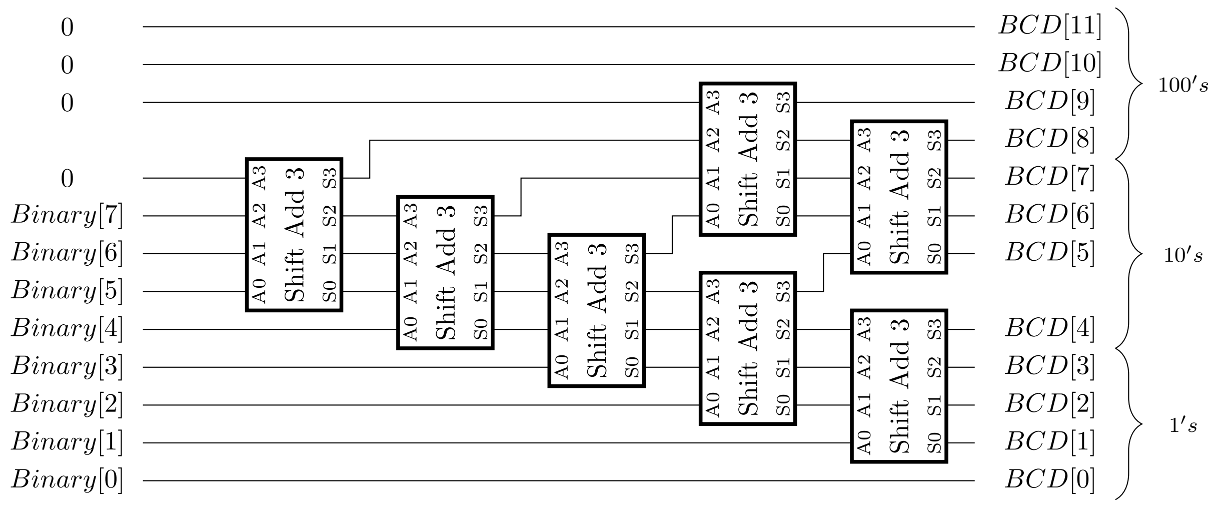

Figure 2: Shift Add 3 Module Digital Representation

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: Implement the Double Dabble Algorithm Using Verilog Procedural Operators

The procedural implementation follows the same idea as the combinational, but you can break it out into a step by step process. The following are the sequential steps for the double dabble algorithm.

1. Shift the most significant binary bit into the least significant bit of the BCD value
2. If any of the BCD digit values are 5_{10} or larger, add 3_{10} to that digit
3. Repeat until there is only one bit left in the binary value. The last binary bit becomes the least significant bit of the BCD value.

2.1 Verilog Procedural Operations

There are several Verilog operators you will need to familiarize yourself with.

2.1.1 Always Block and Registers

The first is the always block. You have seen this in the previous lab, where it takes the form:

```verilog
always @(posedge condition) begin

end
```

This creates a block that on the rising edge of whatever you choose for a condition, executes the statements between the begin and end sequentially. The statements inside an always block take a different form than those outside. For example, outside of an always block if we want to set variable x equal to y, we need a line such as "assign x=y" while inside an always block you do not use the assign keyword.

Another important aspect of the procedural always blocks is you would use registers on the left hand side of equations inside an always block. You would not use wires on the left hand side. So for example the following code would give an error:

```verilog
module testmodule(
    input [3:0] switch,
    input clock
);
    wire [3:0] x;
    always @(posedge clock) begin
        x=switch;
    end
endmodule
```

But this as an alternative would not give an error:

```verilog
module testmodule(
    input [3:0] switch,
    input clock
);
    reg [3:0] x;
    always @(posedge clock) begin
        x=switch;
    end
endmodule
```

Similarly assigning to registers outside of an always block will give an error:

```verilog
module testmodule(
    input [3:0] switch,
    input clock
);
    reg [3:0] x;
    assign x=switch;
endmodule
```

This is not an issue with the register/wire on the right hand side of the equation. The following example would synthesize with no errors:

```verilog
module testmodule(
    input [3:0] switch,
         input clock
);
    reg [3:0] x;
         wire [3:0] y;
         wire [3:0] z;
         assign y=switch;
         assign z=x;
    always @(posedge clock) begin
        x=y;
    end
endmodule
```

2.1.2 For Loop

```
module testmodule(
    output [3:0] led,
        input clock
);
    reg [3:0] i;
    always @(posedge clock) begin
        for(i=0;i<12i=i+1) begin
            //statements here will execute 12 times
        end
    end
endmodule
```

2.1.3 If Statement

```
module testmodule(
    input [3:0] switches,
        input clock
);
    always @(posedge clock) begin
        if (switches==4'b1001) begin
            //statement will execute if switch 0 and 3 are on
        end
    end
endmodule
```

2.1.4 Bit Shift Operator

The left bit shift operator (<<) will shift a variable by a defined amount of bits. For example, the following line would shift the 4 bit value of 1011 in the register x one bit to the left, giving x a new value of 0110. The bit that shifted off to the left is lost.

```
always @(posedge clock) begin
    x=x<<1;
end
```

2.2 Procedural Portion Hints

The following are more specific directions/hints for completing the procedural portion of the lab.

You will likely need to shift the binary value inside the always block. Verilog will give you an error if you use the input that you declared in the module inside the always block. Instead, create a new register inside the module, and then first thing in your always block assign the input binary value to that register. You can then use that register on the left hand side of equations, such as with shifts.

After you assign the input binary to a register, you will likely want to start a for loop, that will iterate over the binary bits. The for loop should run for as many iterations as we have bits to convert, minus 1, as the last bit just gets moved into the least significant point of the BCD value without needing to do an add 3 afterwards.

The first thing you would likely want to move the most significant binary bit into the least significant BCD bit position, and then shift the next most significant bit in the binary value into the most significant position.

In the for loop you can use if statements to check if any of the BCD digits are 5 or larger, and add 3 to the digit if it is.

2.3 Testing

Test your binary to BCD converter in the same manner you tested the combinational. Compare the RTL viewer between the two different implementations. Which was easier to design? Which would be easier to expand to more bits? Even though we designed the module in the second section using procedural Verilog, how much of the created hardware ended up being sequential hardware?

3 Implement the Binary to BCD Converter with a Timer

Utilize the time from the last section of lab 5. Display the value of the 8 most significant bits in decimal on the seven segment displays. You can reuse the module you created from lab 2 to display on the seven segment displays.

4 Checkoff

The TA will check you off when you demonstrate separately the working combinational and procedural versions of the binary to BCD converters by dislaying the timing value on the seven segment displays.