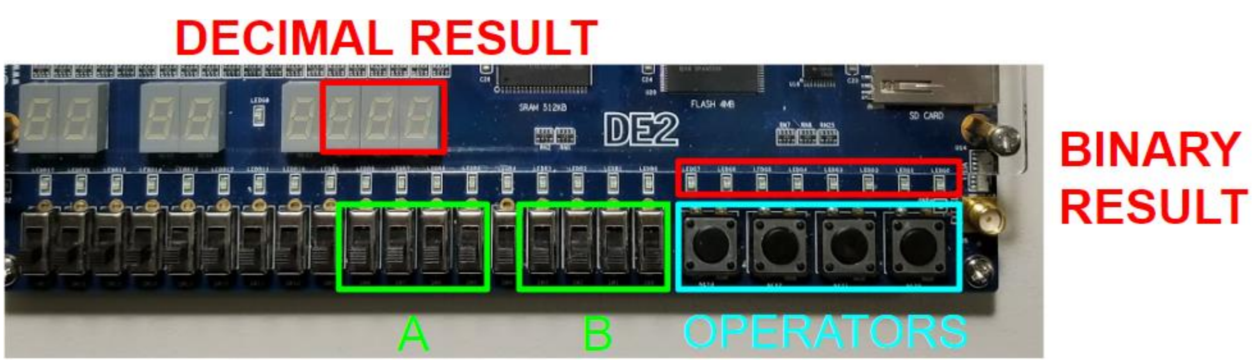
Class:	CPE300L - 1001		Semester:	Fall 2018
Points		Document author:	Chris Barr	
		Author's email:	Barrc1@unlv.nevada.edu	
		Document topic:	Final Project	
Instructor's comments:				

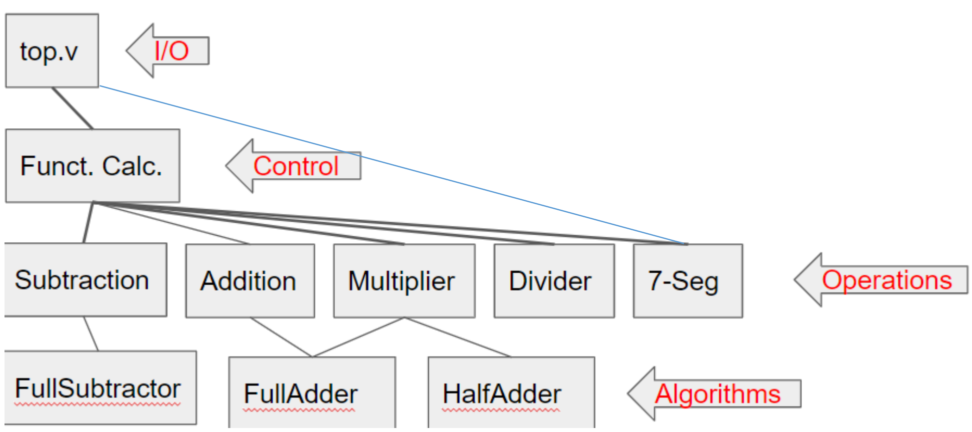
1. Introduction / Theory of Operation

To use knowledge learned from lab 8 and create a functioning calculator with four operators - addition, subtraction, multiplication and division. Aside from the operations, the device also became a binary to binary and decimal converter. We implemented Verilog code in order complete the design.

2. Description of Project



Board implementation



Code Directory of the module instantiations

Because our calculator deals with 4-bit inputs, the green box indicates the four switches used. So, our program will run based off of these operator equations:

A + B = RESULT
A – B = RESULT
A * B = RESULT
A / B = RESULT

The image below this text shows the Verilog code associated to grabbing the result from these specific operands:

```
module Calculator (input [3:0] a,
                   input [3:0] b,
                   input [3:0] butt,
                   input CLK,
                   input start,
                   input rst,
                   output [7:0] out);

                   reg [7:0] y;

                   assign out = y;

                   wire [7:0] sum, diff, prod, quot;
                   wire done;
                   wire [7:0] rem;
                   wire [3:0] hun, ten, one;

Addition A1 (a[3:0], b[3:0], sum[7:0]);
Subtractor S1 (a[3:0], b[3:0], diff[7:0]);
Multiplier M1 (a[3:0], b[3:0], prod[7:0]);
Divider D1 (CLK, rst, start, a[3:0], b[3:0], quot[7:0], rem[7:0], done);

BCD SD (.binary(out));

always @(*)
begin
case(butt)
   4'b1110 : y[7:0] <= sum[7:0];
   4'b1101 : y[7:0] <= diff[7:0];
   4'b1011 : y[7:0] <= prod[7:0];
   4'b0111 : y[7:0] <= quot[7:0];
   default : y[7:0] <= 8'b00000000;
endcase
end
endmodule
```

There are four module instantiations, *Addition*, *Subtractor*, *Multiplier*, and *Divider*. Each of those four modules are imperative to grabbing the result of the operand and sending that signal to the board.

This module acts similarly to a MUX. Dependent on what *butt* (pushbutton) is selected, it will grab that specific result and send it out as the output. We call this module the ‘control’ portion of our code.

```verilog
module Multiplier (X, Y, P);
   input[3:0] X;
   input[3:0] Y;
   output[7:0] P;
   wire[3:0] C1,  C2, C3, S1, S2, S3, XY0, XY1, XY2, XY3;
   assign XY0[0] = X[0] & Y[0];
   assign XY1[0] = X[0] & Y[1];
   assign XY0[1] = X[1] & Y[0];
   assign XY1[1] = X[1] & Y[1];
   assign XY0[2] = X[2] & Y[0];
   assign XY1[2] = X[2] & Y[1];
   assign XY0[3] = X[3] & Y[0];
   assign XY1[3] = X[3] & Y[1];
   assign XY2[0] = X[0] & Y[2];
   assign XY3[0] = X[0] & Y[3];
   assign XY2[1] = X[1] & Y[2];
   assign XY3[1] = X[1] & Y[3];
   assign XY2[2] = X[2] & Y[2];
   assign XY3[2] = X[2] & Y[3];
   assign XY2[3] = X[3] & Y[2];
   assign XY3[3] = X[3] & Y[3];

   FullAdder FA1 (XY0[2], XY1[1], C1[0], C1[1], S1[1]);
   FullAdder FA2 (XY0[3], XY1[2], C1[1], C1[2], S1[2]);
   FullAdder FA3 (S1[2], XY2[1], C2[0], C2[1], S2[1]);
   FullAdder FA4 (S1[3], XY2[2], C2[1], C2[2], S2[2]);
   FullAdder FA5 (C1[3], XY2[3], C2[2], C2[3], S2[3]);
   FullAdder FA6 (S2[2], XY3[1], C3[0], C3[1], S3[1]);
   FullAdder FA7 (S2[3], XY3[2], C3[1], C3[2], S3[2]);
   FullAdder FA8 (C2[3], XY3[3], C3[2], C3[3], S3[3]);
```

Parallel Array Multiplier

The parallel array multiplier was instantiated from the *Calculator* module. First, it will multiply each bit in parallel. Then, it will call the *FullAdder* and *HalfAdder* module to add the resulted multiplication of each bit. The result will be sent back through the instantiation seen from the *Calculator* module.

```verilog
module Divider(
    input clk,
    input rst,
    input start,
    input  [3:0] num,
    input  [3:0] den,
    output [7:0] res,
    output [7:0] rem,
    output reg done
);

reg [3:0] num_r;
reg [3:0] den_r;
reg [7:0] result_integer;
reg working;

always @(posedge clk) begin
    if(rst == 1'b1)begin
        num_r <= 4'b0;
        den_r <= 4'b0;
        working <= 1'b0;
        result_integer <= 'b0;
        done <= 'b0;
    end else if(start == 1'b1) begin
        num_r <= num;
        den_r <= den;
        working <= 1'b1;
        done <= 1'b0;
    end
    // Algorithm
    if (working == 1'b1 && start == 1'b0)begin
        if(num_r >= den_r) begin
            num_r <= num_r - den_r;
            result_integer <= result_integer + 8'b1;
        end else begin
            working <= 'b0;
            done <= 1'b1;
        end
    end
end

assign rem = num_r;
assign res = result_integer;
```

Parallel Array Divider

The parallel array divider was instantiated from the *Calculator* module. This module works based off the clock because it needs to determine when to restart and start the module. Afterwards, the algorithmic portion of the code will begin and computer the quotient. Thus, sending back the result to be outputted onto the DE2 Board.

Note that this operator module is the only one that carries the actual algorithm within its code. Every other operator calls their algorithm module to do the actual adding, subtracting, etc.

```
module Addition (input [3:0] X,
                 input [3:0] Y,
                 output [7:0] S);

                 wire cin = 1'b0;
                 wire [3:0] cout, sum;

   FullAdder FA1 (X[0], Y[0], cin, cout[0], sum[0]);
   FullAdder FA2 (X[1], Y[1], cout[0], cout[1], sum[1]);
   FullAdder FA3 (X[2], Y[2], cout[1], cout[2], sum[2]);
   FullAdder FA4 (X[3], Y[3], cout[2], cout[3], sum[3]);

   assign S[0] = sum[0];
   assign S[1] = sum[1];
   assign S[2] = sum[2];
   assign S[3] = sum[3];
   assign S[4] = cout[3];

endmodule
```

Single bit Addition

The single bit addition module was instantiated from the *Calculator* module. How this works is that the module will take in both the input that are to be added, X and Y. Then, it will add the least significant bit and carry over any overflow to the next significant bit. The process will repeat until the sum of the two numbers are calculated.

```
module Subtractor (input [3:0] X,
                   input [3:0] Y,
                   output [7:0] D);

                 wire cin = 1'b0;
                 wire [3:0] borr, diff;

   FullSubtractor FS1 (X[0], Y[0], cin, borr[0], diff[0]);
   FullSubtractor FS2 (X[1], Y[1], borr[0], borr[1], diff[1]);
   FullSubtractor FS3 (X[2], Y[2], borr[1], borr[2], diff[2]);
   FullSubtractor FS4 (X[3], Y[3], borr[2], borr[3], diff[3]);

   assign D[0] = diff[0];
   assign D[1] = diff[1];
   assign D[2] = diff[2];
   assign D[3] = diff[3];


endmodule
```

Single bit Subtractor

The single bit subtractor module was instantiated from the *Calculator* module. This module, similar to the *Addition* module, will start from the least significant bit and start calculating bit-by-bit. However, this module will borrow from the most significant bit when needed.
This module can also account for 2's compliment.

The code below is considered our algorithmic portion of the code. Meaning, every operator, excluding *Division*, used either the *FullAdder*, *FullSubtractor*, *HalfAdder*, or multiple of the three. Here are those three modules:

```
module FullAdder (X, Y, Cin, Cout, Sum);

   input X;
   input Y;
   input Cin;
   output Cout;
   output Sum;

   assign Sum = X^Y^Cin;
   assign Cout = (X & Y) | (X & Cin) | (Y & Cin);
endmodule
```

FullAdder

```
module FullSubtractor ( a ,b ,c , borrow, diff );

output diff ;
output borrow ;

input a ;
input b ;
input c ;

assign diff = a ^ b ^ c;
assign borrow = ((~a) & b) | (b & c) | (c & (~a));

endmodule
```

FullSubtractor

```
module HalfAdder (X, Y, Cout, Sum);

   input X;
   input Y;
   output Cout;
   output Sum;

   assign Sum = X^Y;
   assign Cout = X & Y;
endmodule
```

HalfAdder

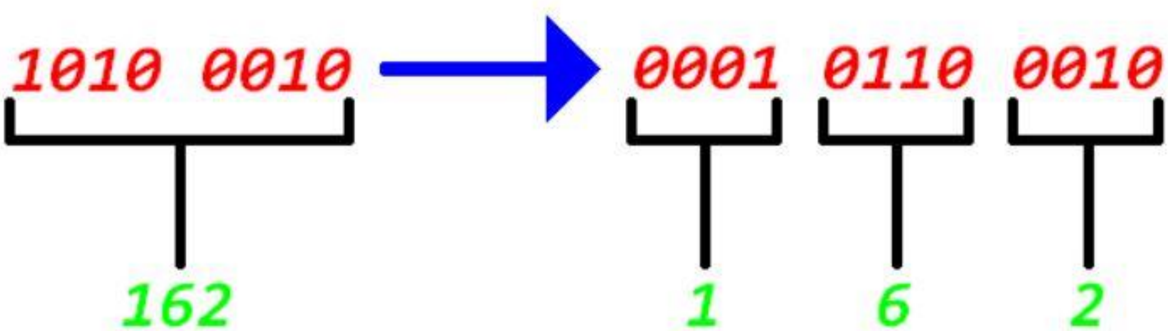
```
module BCD (
    input [7:0] binary,
    output [3:0] hex0,
    output [3:0] hex1,
    output [3:0] hex2
    );

    sevSeg pls_work2 (.Ones(hex0), .Tens(hex1), .Hundreds(hex2));

    reg [3:0] Ones, Tens, Hundreds;

    assign hex0 = Ones;
    assign hex1 = Tens;
    assign hex2 = Hundreds;

    integer i;
    always @ (binary)
    //initial
    begin
        Hundreds = 4'd0;
        Tens = 4'd0;
        Ones = 4'd0;

        for (i=7; i>=0; i=i-1)
        begin
            if (Hundreds >= 5)
                Hundreds = Hundreds + 3;
            if (Tens >= 5)
                Tens = Tens + 3;
            if (Ones >= 5)
                Ones = Ones + 3;

            Hundreds = Hundreds << 1;
            Hundreds[0] = Tens[3];
            Tens = Tens << 1;
            Tens[0] = Ones[3];
            Ones = Ones << 1;
            Ones[0] = binary[i];
        end
    end
endmodule
```

BCD

This is our BCD (Binary Coded Decimal), which is used to convert our binary to decimal for the seven segment display. Here's how the program works:

100's	10's	1's	Binary	Operation
			1010 0010	
		1	010 0010	<< #1
		10	10 0010	<< #2
		101	0 0010	<< #3
		1000		add 3
	1	0000	0010	<< #4
	10	0000	010	<< #5
	100	0000	10	<< #6
	1000	0001	0	<< #7
	1011			add 3
1	0110	0010		<< #8

In this example, our 8-bit binary code is valued at 162. We will shift the value by 1 for every bit until the value detected in the one's, ten's, or hundred's spot reaches the value of 101 (5), once that happens we add the value 11 (3). The result of these calculations will make our 1010 0010 (162) value look like this:

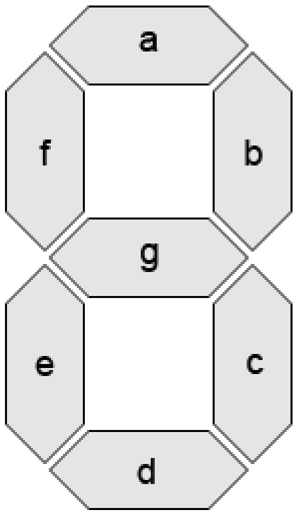


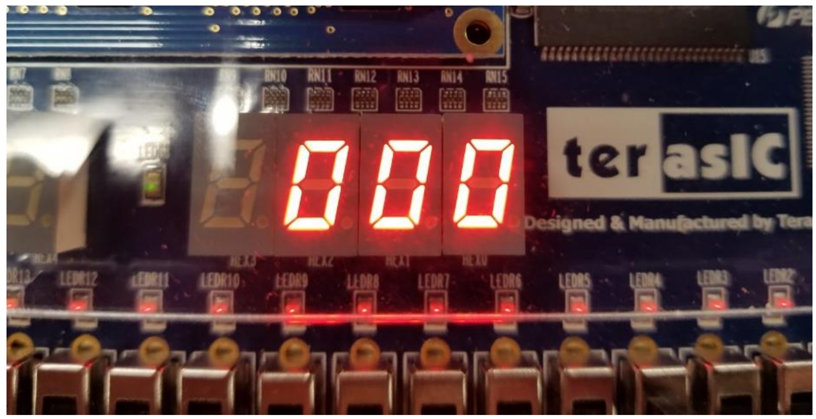
The value of the right-hand side will then be transferred to the seven segment decoder.

```verilog
module sevSeg (input [3:0] Ones,
               input [3:0] Tens,
               input [3:0] Hundreds,
               output [6:0] hex_0,
               output [6:0] hex_1,
               output [6:0] hex_2);

    reg [6:0] temp1, temp2, temp3;

    assign hex_0 = temp1;
    assign hex_1 = temp2;
    assign hex_2 = temp3;

    always @(Ones)
    begin
        case (Ones)
            0 : temp1 = 7'b1000000;
            1 : temp1 = 7'b1111001;
            2 : temp1 = 7'b0100100;
            3 : temp1 = 7'b0110000;
            4 : temp1 = 7'b0011001;
            5 : temp1 = 7'b0010010;
            6 : temp1 = 7'b0000010;
            7 : temp1 = 7'b1111000;
            8 : temp1 = 7'b0000000;
            9 : temp1 = 7'b0010000;
            default : temp1 = 7'b1111111;
        endcase
    end

    always @(Tens)
    begin
        case (Tens)
            0 : temp2 = 7'b1000000;
            1 : temp2 = 7'b1111001;
            2 : temp2 = 7'b0100100;
            3 : temp2 = 7'b0110000;
            4 : temp2 = 7'b0011001;
            5 : temp2 = 7'b0010010;
            6 : temp2 = 7'b0000010;
            7 : temp2 = 7'b1111000;
            8 : temp2 = 7'b0000000;
            9 : temp2 = 7'b0010000;
            default : temp2 = 7'b1111111;
        endcase
    end

     always @(Hundreds)
    begin
        case (Hundreds)
            0 : temp3 = 7'b1000000;
            1 : temp3 = 7'b1111001;
            2 : temp3 = 7'b0100100;
            3 : temp3 = 7'b0110000;
            4 : temp3 = 7'b0011001;
            5 : temp3 = 7'b0010010;
            6 : temp3 = 7'b0000010;
            7 : temp3 = 7'b1111000;
            8 : temp3 = 7'b0000000;
            9 : temp3 = 7'b0010000;
            default : temp3 = 7'b1111111;
        endcase
    end

endmodule
```

Seven Segment Decoder

This Verilog code will receive the binary coded decimal via module instantiation. Then, dependent on the value given, it will determine what the number will be for the three seven segment displays we used on our DE2 board.



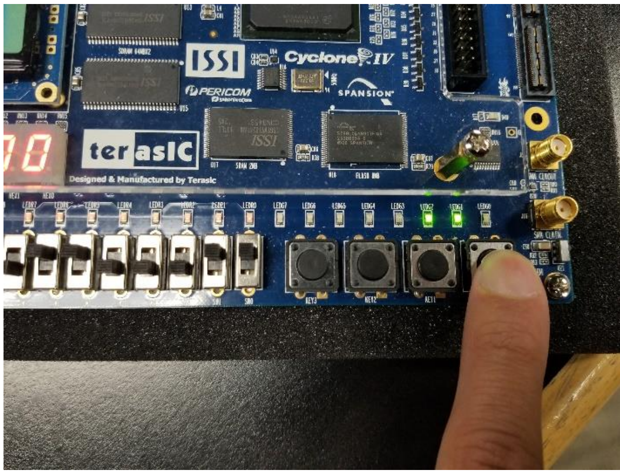


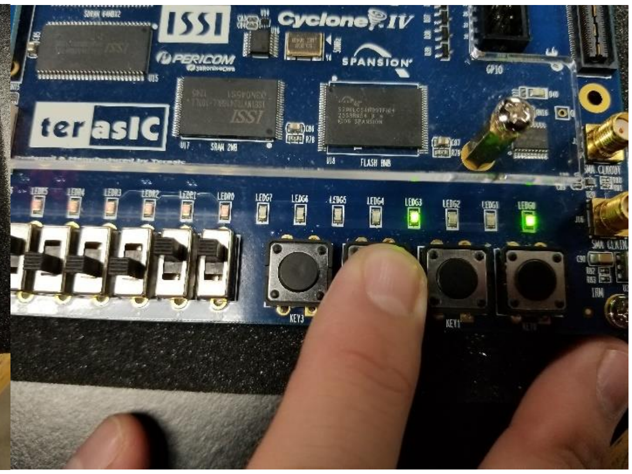
Seven segments controlled by 7 bits. When a segment needs to be turned on, a “0” is placed in that bit position. Otherwise “1” for off.
Here’s an example output of what we would get for the value 0000 0000 (0) in the picture on the right hand side.

```
module top (input [3:0] lhs, rhs,
                input [3:0] operator,
                input clk,
                input start,
                input rst,
                output [7:0] ans,
                output [6:0] Tens,
                             Hundreds,
                             Ones);

Calculator inst_start (lhs[3:0], rhs[3:0], operator[3:0], clk, start, rst, ans[7:0]);
sevSeg returnVal (.hex_1(Tens),
                  .hex_2(Hundreds),
                  .hex_0(Ones));
endmodule
```

Input and Output File (Top)

This is our *top* file, which collects all the inputs and outputs for the whole program; top-level. We used this file to clean up any messes in the module parameters. Just as seen in the Code Directory of the module instantiations. This file instantiates the modules *Calculator* and *7Seg*.





Here are two example outputs that we would get on our board for the binary portion of the calculations.

3. Encountered Problems

The Divider module was fairly difficult to deal with. Because there was a lot to it compared to the other three operands, it made it difficult to figure out how to approach it. We decided to go with a parallel array divider method. In the end, we've come to realize that we could have gone without the clock.

The seven segment display was not grabbing the output of the result. This caused the board to not output the 7 segment correctly.

4. Summary

Utilizing the boards 7 segment display, LED's, switches, and pushbuttons, we were able to simulate a binary to decimal calculator. We used the basic four operators (addition, subtraction, multiplication, and division) and calculated two 4-bit inputs to get an 8-bit result. Thereafter, we showcased the binary result on 8 green LED's and the decimal result on 3 7-segment displays.

5. Conclusions

This project was a bit more difficult than we thought. We originally had less modules and assumed we could use behavioral Verilog code for most of it, but that was not the case. We also knew the division portion of the Verilog code was going to be the most difficult, so we tried to reflect the division code from lab 8 onto our project. But, we ended up really customizing that code. The division operation probably did not need a reset and start switch; it was poorly optimized. And strangely enough, we were getting more errors in our 7-segment display code than we were in our division.

I was very happy that I could do this kind of project, because this project helped solidify my knowledge in Verilog.