Lab 3
Multiplexer Logic Fun in Verilog!

Gavin Chen

4/25/18

CE 100

Description:

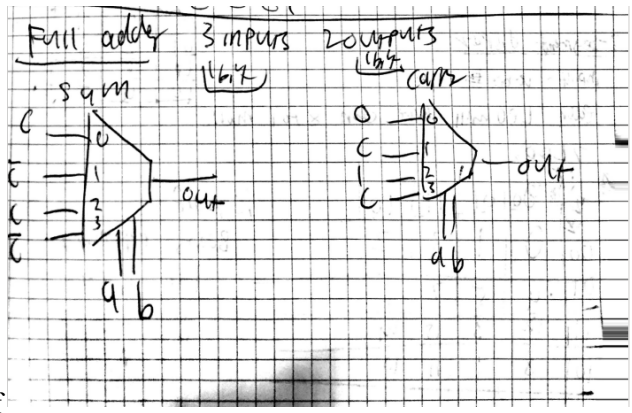
In this lab we implemented a full 8 bit adder and 7 segment display interpreter using multiplexers in place of the normal logic gate design. As such we began by implementing the logic for the multiplexers using gates, creating the building blocks of the lab. From there we then created Karnaugh maps that output the desired bits according to an input and mapped them to the multiplexers. Finally, with our adder and segment display converter created we interfaced them in our top module to interact with each other and the hardware on the board. Due to the nature of the 7 segment display, we had to implement a Schalg-provided clock signal to toggle input and output and cycle the display.

Methods:

Overall there were 3 types of multiplexers to be implemented. A 4 to 1, 8 to 1, and 2 to 1 x 8 bus mux. We begin with the 4 to 1 mux. We chose to implement the circuit by taking the sum of product formula for the 2 bit selector input, and ANDing the desired output line with the corresponding SOP formula. Through doing so we were able to implement a simple 4 to 1 bit selector using 10 gates.

Next we focused on the 8 to 1 selector. For this task we choose to implement mux logic as opposed to gates, and utilized two 4 to 1 multiplexers and several AND and OR gates to get the desired output. Using Shannon's expansion theorem allowed us to extract the most significant bit of our 8 to 1 multiplexer and create 2 sub functions with 4 inputs each that corresponded to the multiplexers. We first laid down the mux's and fed in the 2 least significant bits of our 8 to 1 multiplexers as common input. The resulting 2 outputs were then chosen using AND gates and the normal or inverted form of the selector bit, similar to the 4 to 1 multiplexer. The final output was then ANDed with the enable input to create an efficient enabling mechanism.

Finally we created the 2 to 1 bus mux using a simple SOP formula refitted to accommodate an 8 bit wide bus. Both inputs were connected to the output, and a formula was calculated to zero out one of the inputs given the selector bit. To do this the selector bit was extended into an 8 bit wide bus and ANDed with the 2 input buses. This effectively created a bit mask that allowed the input from the selected input through, while zeroing out the other.



With our 3 mux's created we then set about designing a single bit adder implementing 4 to 1 mux's using Karnaugh maps. We designated one bit to be used as input, and 2 bits to be used as selectors. Using the 2 bits we selected out output to be either 1, 0, or some form of our input data bit. In the case of our sum we chose the input number bits to be the selector and the carry in bit to be part of the data stream. The process for determining the arrangement is as follows: we created 3 input K-maps for the sum and carry out data. The selector bits corresponded to the columns, and the input/carry bit the rows. Each corresponding output of the mux was then some column of the K-map that corresponded to 1, 0, C or ~C and was wired into the mux accordingly. This created a module that implemented the addition logic using mux's which were then cascaded together to create a full 8 bit adder. Twos compliment and overflow logic was then implemented in the adder module. The first was done by XORing an 8 bit extension of the subtraction option bit with the B input bus, and simultaneously feeding that bit into the first carry in bit of the adder. This inverted B on high and added one into the first carry in, simulating subtraction. Overflow logic was performed by analyzing the MSB of the input numbers and the output, as well as the carry out bit of the most significant adder. If the signs of the input numbers matched each other but did not match the sum, overflow occurred and the line was driven high. Similarly for the case where the input numbers were positive and a carry out bit occurred from the MSB of the adder.

Our next task was to implement the 7-segment display interpreter using mux's. We began with the seven 4 input truth tables which corresponded to each cathode of the segment display. From there K-maps were generated and 3 bits were chosen to become selector bits and one an input value. This reduced the table from 16 individual blocks into eight 2-block groups, each either 0, 1, or some form of the input bit. Creating the correct output for the mux was then a matter of connecting the correct input for each mux port. The diagrams and data are presented in the lab notes in the Appendix.

The final step was to organize all of the modules together in the top level to produce a visible result on the Bayes board. The top level of our module had one instance of the full 8 bit adder which produced two 4 bit hexadecimal outputs with input taken in from switches 0-15 as two 8 bit numbers. The overflow bit was fed into the decimal line of the segment display, while each half of the 1 byte number was fed into an individual instance of the segment display converter each of which output an 8 bit bus. The two bus outputs were fed into an instance of our 2 to 1 bus mux, and an oscillating digital signal was used to toggle the bus selection connected to the segment display cathode and to switch on the corresponding LED digit.

Results:

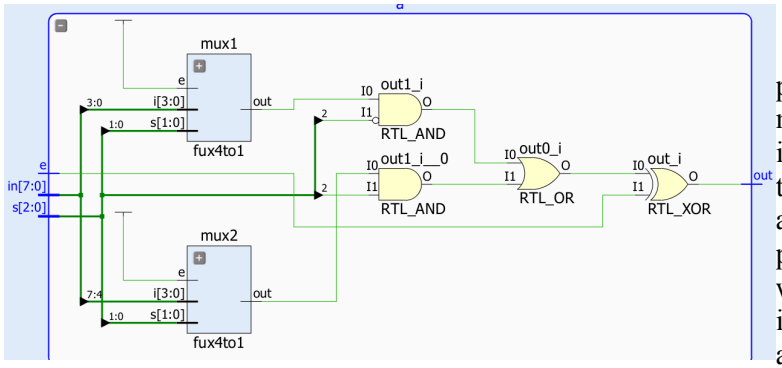
4 to 1 mux:

The implemented mux's and design logic produced a working circuit that calculated the sum of two 1 byte numbers and output the result onto 2 digits of the segment display. A truth table of the 4 to 1 mux is shown to the right. The 4 In columns represent which row of selection bits connect it to the output. For example, the 0^{th} input is connected to output on Select 00, corresponding to a sum of products partial formula

(~S1 & ~S2 & In0)

Which connected the value of the In0 line to the output. The corresponding formulas were then all ORed together to create the final output. A schematic of the design in presented in the appendix.

Select 2	Select 1	In 0	In 1	In 2	In 3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

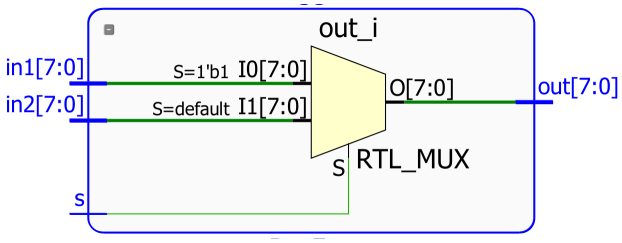


8 to 1 mux:

A schematic of the 8 to 1 mux is presented to the left. It consists of two 4 to 1 mux's that act as an initial selector of the inputs. The desired mux is then connected to the output through the use of an AND gate, and the final output is ANDed with the enable pin to produce the final output. The mux worked as specified and we were able to implement the 7 segment converter logic accordingly.

2 to 1 bus mux:

The final mux implemented in the lab is the 8 bit 2 to 1 bus mux. As stated before, the mux was created by first extending the selector bit into an 8 bit bus, then using it as a bit mask on the incoming data. The bitmask results were ORed together to create an 8 bit bus output controlled by a single bit.



7-Segment Display:

The most difficult/tedious section of the lab was in the implementation of the 7-segment display logic. Due to the fact that there was no real relation between segment cathode connections and therefore could not be reduced, creating the logic for the converter involved computing the logic for each connection individually. As such 8 connections were made, along with mistakes. A major issue was with the inverted nature of the display, and the first iteration of the design resulted in the number appearing in negative. This was remedied with a simple inversion of the output. Again, the use of modules made the implementation of this logic vastly simpler. Due to this, it was simpler to implement the logic with mux's rather than gates, as the same template could be used for each output with the only difference between them being the order of the input. While the calculations were more involved, this made error shooting much simpler. The final result was a working converter abstracted into a module design, with an efficient enabling mechanism. A schematic is presented in the appendix.

Add/subtract logic:

Implementing the add/subtract logic using mux's gave a strong insight into how Shannon's expansion theorem is able to generate any output as a function of selecting a certain input. The Karnaugh maps for the implementations are presented to the right, where AB represents the input number bits, C the carry bit, the bold number the output for the circuit given those inputs, and the formula the selected input for a specific sequence of AB. In the summation circuit design, it was noticed that all input selections were some form of the carry in bit. This is opposed to the carry circuit which had 2 constant inputs of 0 and 1for the cases where AB are 00 and 11 respectively. A schematic of the full bit adder design is included in the appendix, as well as in the preceding section.

Sum					
	AB	00	01	11	10
C	0	**0**	**0**	**1**	**0**
	1	**0**	**1**	**1**	**1**
Formula		**C**	**~C**	**C**	**~C**

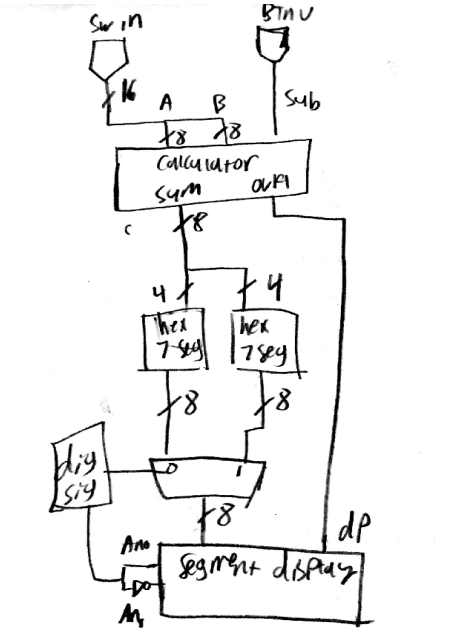
Carry					
	AB	00	01	11	10
C	0	**0**	**0**	**1**	**0**
	1	**0**	**1**	**1**	**1**
Formula		**0**	**C**	**1**	**C**

In implementing the 8 bit adder the correct inputs and outputs from the module header were wired to the respective components i.e. full bit adder modules. 2's compliment logic and overflow detection were carried out within the wrapper module as opposed to the individual adder modules to facilitate modularity.

Because we performed subtraction through the inversion of the secondary number, i.e. B, the operation of the calculator was such that input numbers for B appeared negative. The calculation was therefore interpreted as A-B as opposed to taking the absolute difference between 2 numbers. Moreover, only general cases with the overflow logic were covered, and there were certain cases involving the inversion of a number that produced an overflow when none should have occurred.

Top module:

Combining all of the modules together in the top module, we were able to create a calculator that took as input 16 switches as two 8 bit binary numbers, calculating the summation or difference depending on an input from a button. The general flow of the module is presented in the figure to the right. The switch and button input passed through the calculator logic first, which output an 8 bit number and high on ovfl if an overflow occurred. The number is then input into 7 segment converters to create a corresponding ground connection to display a digit on the segment display.



The outputs from the segment converters are sent to a bus mux to toggle which connection is made, and the output of the mux is selected in synchronous with the power connection of the segment display. By creating modules to implement the logic, the creation of the top module dealt only with creating connections and alleviated the need to implement logic. The result function according to the specs, outputting 2 hexadecimal digits on the segment display as a function of the 16 input switches.

Testing and simulation:

To test the design there were 2 main components we focused on. The first was the calculator where we tested various inputs of 8 bit numbers to see if the correct output was achieved. As testing all combinations of 8 bit numbers would have resulted in 256 * 256 = 65,536 inputs, we focused on 6 cases that encompassed the overall behavior. The calculator output as expected for all inputs, pushing the overflow output high for tests 5 and 6.

1. Adding 2 positive numbers no overflow
2. Subtracting 2 positive numbers
3. Adding 2 negative numbers no overflow
4. Subtracting 2 negative numbers
5. Adding 2 positive numbers with overflow
6. Adding 2 negative numbers with overflow

Next we tested the segment display to ensure it output the correct display for both hex digits. Since the same hex converter was used for both digit displays we needed only to test the full range of values for one digit, and then to test that distinct digits could be represented simultaneously on individual displays. We input a waveform that corresponded to all 16 values for hex display 1, and performed 4 tests for hex display 1 and 2, cycling through different digits for each display on each test. After fixing several errors involving the ^ sign representing XOR as opposed to AND, the hex converter and bus mux performed as expected in both the simulations and the physical testing. The waveform for all of the tests including the calculator is included in the appendix.

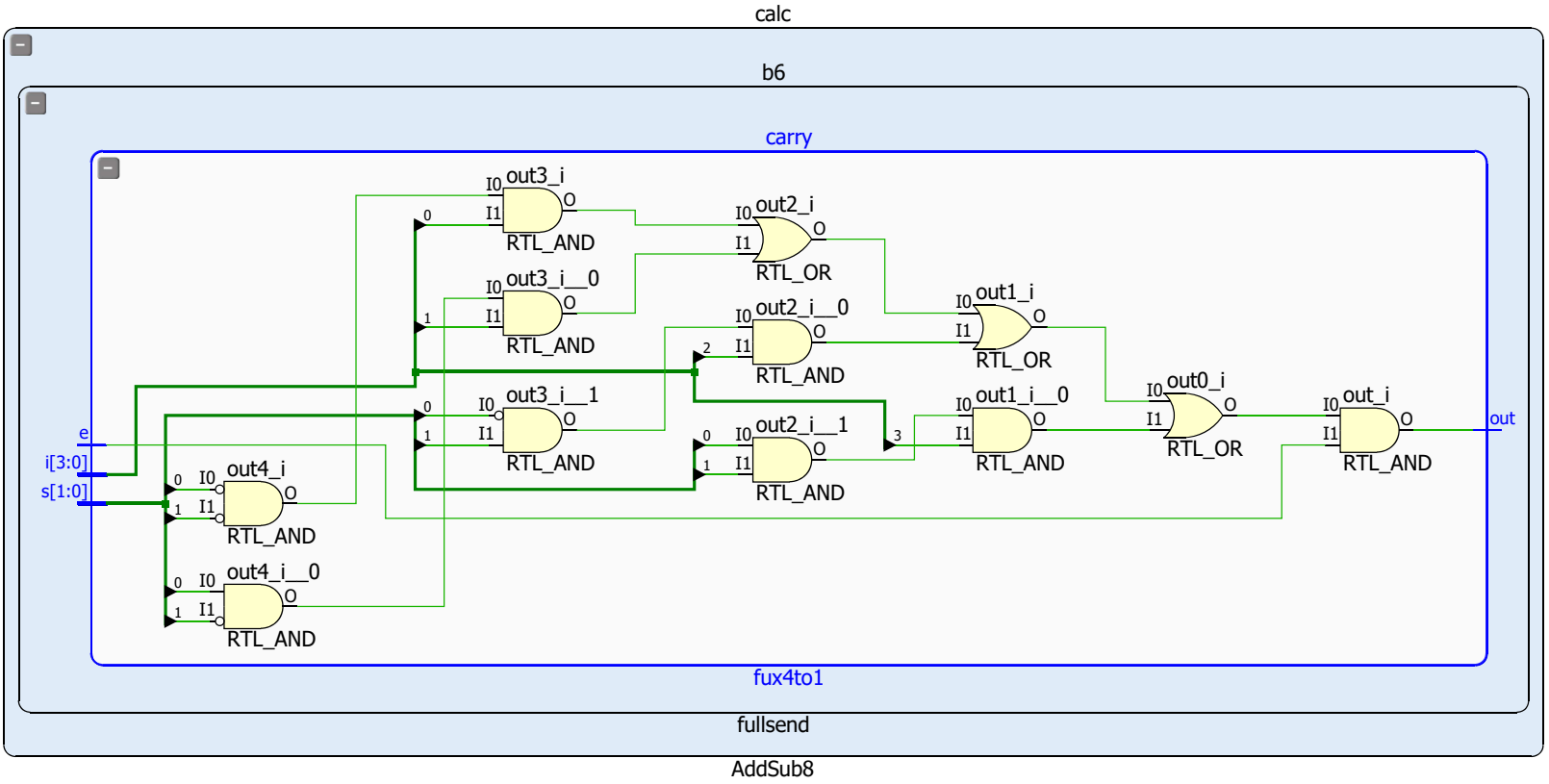
Discussion:

Having created and tested the circuit, we performed some analysis on the unknown oscillating signal dig_sel provided by Headmaster Schlag to control the LED cycling. Using the analog oscilloscope and organic light detecting sensors we observed the frequency of the oscillation. Quantitative analysis with the oscilloscope yielded an on/off period of 1.5 us, for a total period of 3 us. This corresponded to a digital oscillation frequency of approximately 333 kHZ. Qualitatively analyzing the segment display with the aforementioned organic sensors yielded no sign of flickering on the LED's. As such it was determined that the oscillation frequency was high enough to toggle connections with no perceptible lag by humans (at least this one).

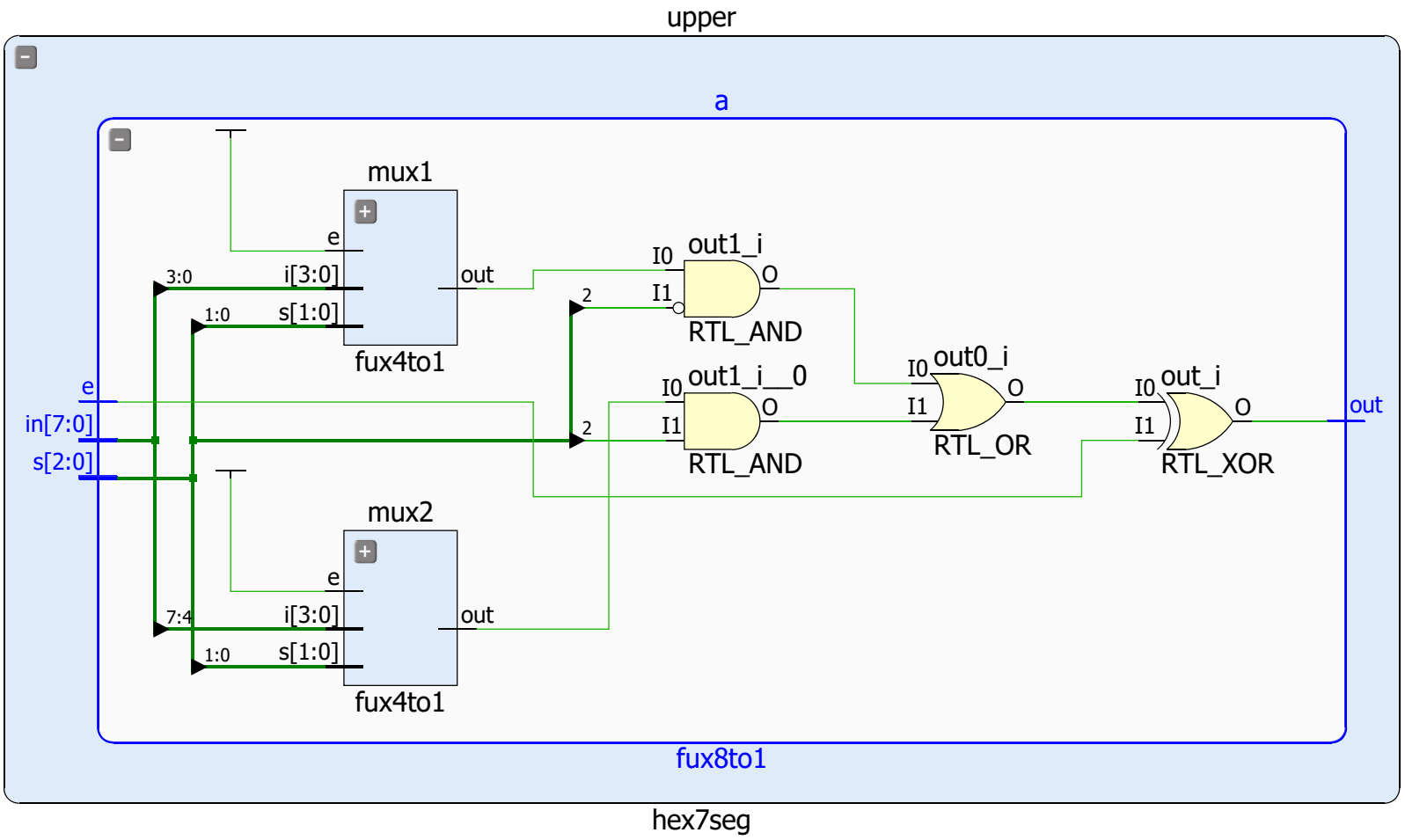
Conclusion:

The lab was difficult but interesting. Having already applied a full bit adder using gates, the logic and operation of the adder was much more familiar and simpler to implement. The largest difficulty was in the creation of the segment converter. There was some difficulty in assigning the correct inputs and sequence of selector bits to each 8 to 1 mux, and several errors and design flaws were found during the course of testing. From the safe distance of retrospection if this lab were to be repeated the experimenter would have begun earlier so as to have more time for trouble shooting. In addition, careful enforcement of bit ordering and input/output matching would have avoided some of the issues with the segment converter.

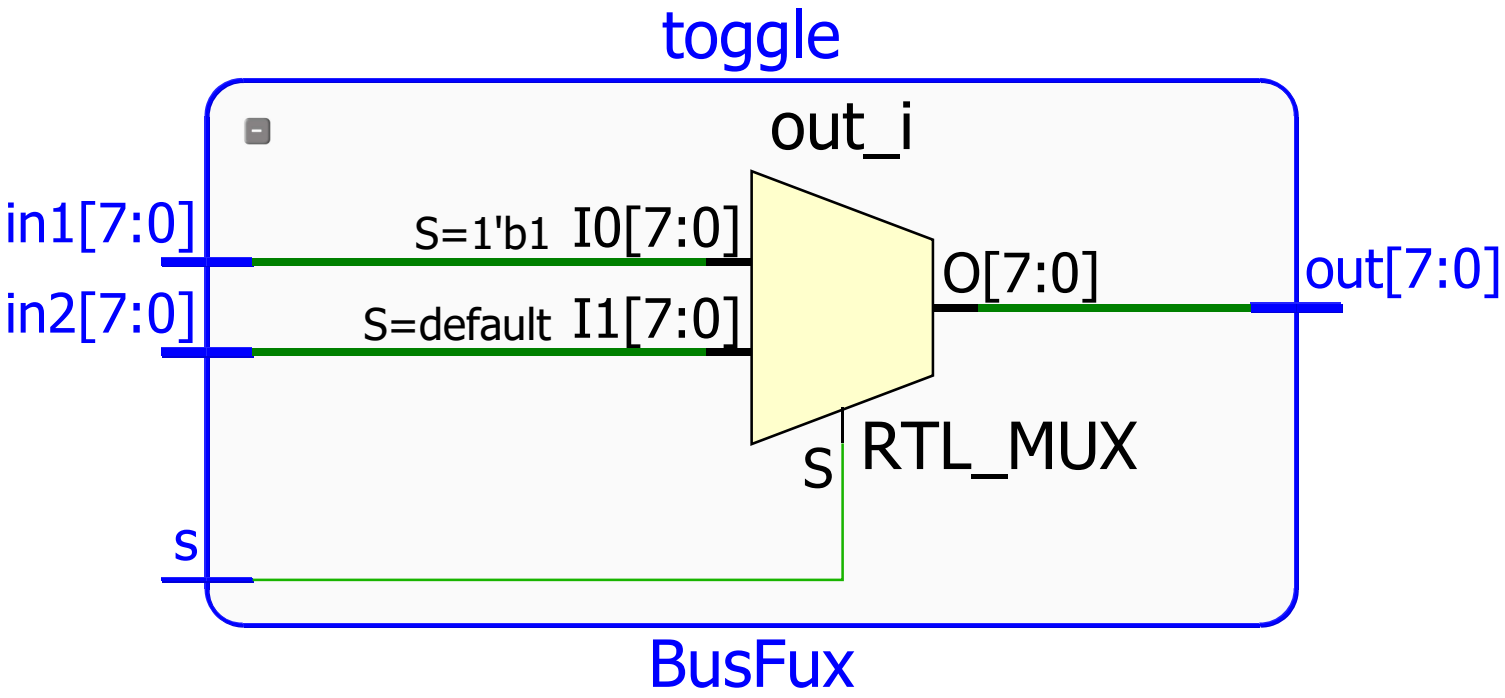
Design wise, choosing different input and selector bits for the mux logic would have made a slight optimization in the form of fewer inverter gates. However the largest optimization would have been in the segment display converter. Instead of creating 2 separate converters it would have been more efficient to use one and to toggle the input value and output destination in order to toggle the display. Overall the lab was interesting, and applying the logic for the mux's was tedious but understandable.


calc
b6
carry
out3_i
RTL_AND
out2_i
RTL_OR
out3_i__0
RTL_AND
out2_i__0
RTL_AND
out1_i
RTL_OR
out3_i__1
RTL_AND
out2_i__1
RTL_AND
out1_i__0
RTL_AND
out0_i
RTL_OR
out_i
RTL_AND
out
e
i[3:0]
s[1:0]
out4_i
RTL_AND
out4_i__0
RTL_AND
fux4to1
fullsend
AddSub8

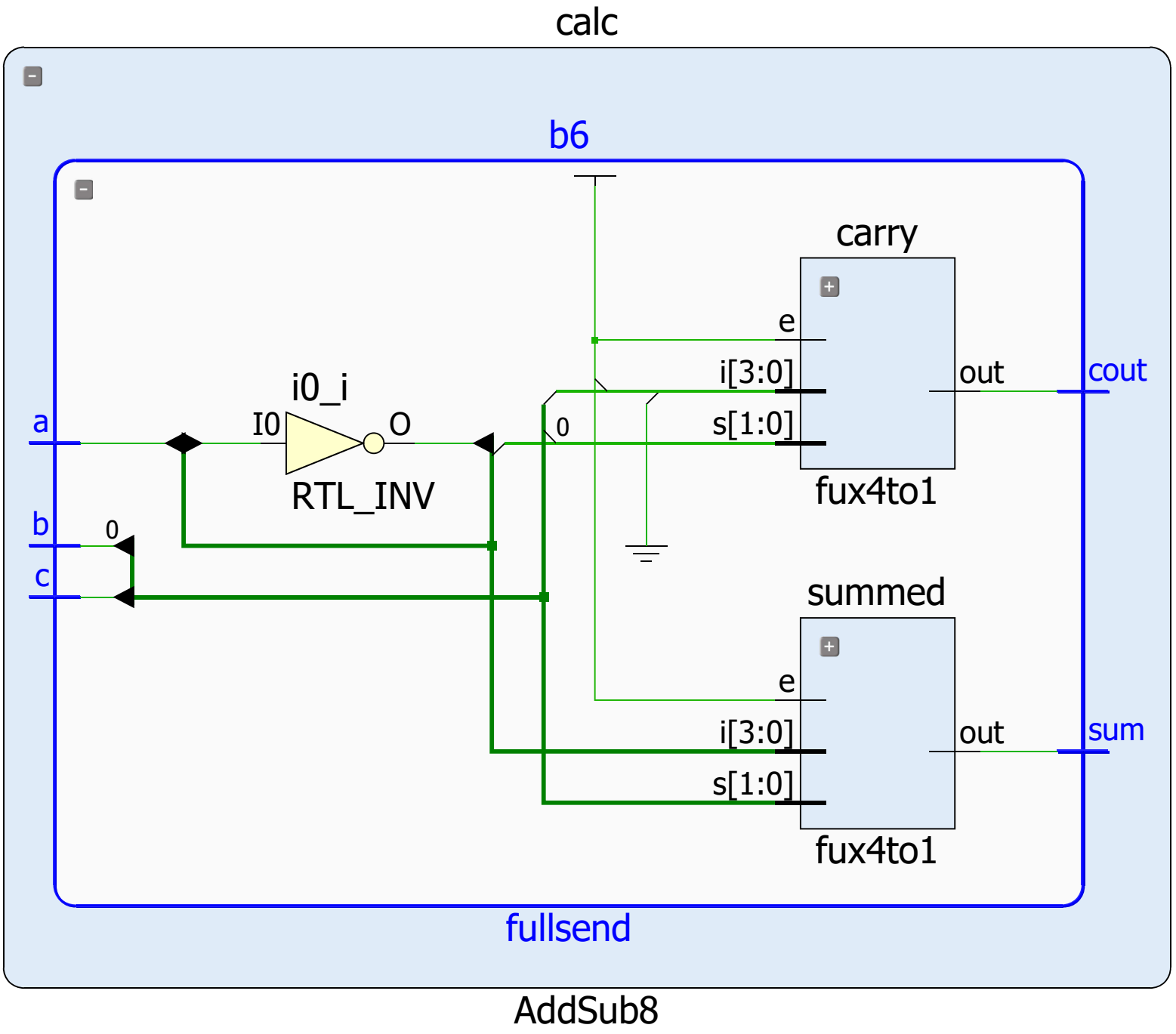
```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 04/17/2018 11:55:15 AM
// Design Name:
// Module Name: fux4to1
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


//4 to 1 mux
//input 1 bit enable, 2 bit selector and 4 bit input
//output 1 bit according to input and selector bit

//input = single selector bit, 2 bit selector bus, and 4 bit input bus
//output = single bit

module fux4to1(
    input [1:0] s,
    input [3:0] i,
    input e,
    output out
    );

    assign out = (~s[0]&~s[1]&i[0] | s[0]&~s[1]&i[1]
         | ~s[0]&s[1]&i[2] | s[0]&s[1]&i[3]) & e;

endmodule
```


upper
a
mux1
e
3:0
i[3:0]
1:0
s[1:0]
out
fux4to1
out1_i
I0
I1
O
RTL_AND
out1_i__0
out0_i
RTL_OR
out_i
RTL_XOR
2
e
in[7:0]
s[2:0]
out
mux2
7:4
fux4to1
fux8to1
hex7seg

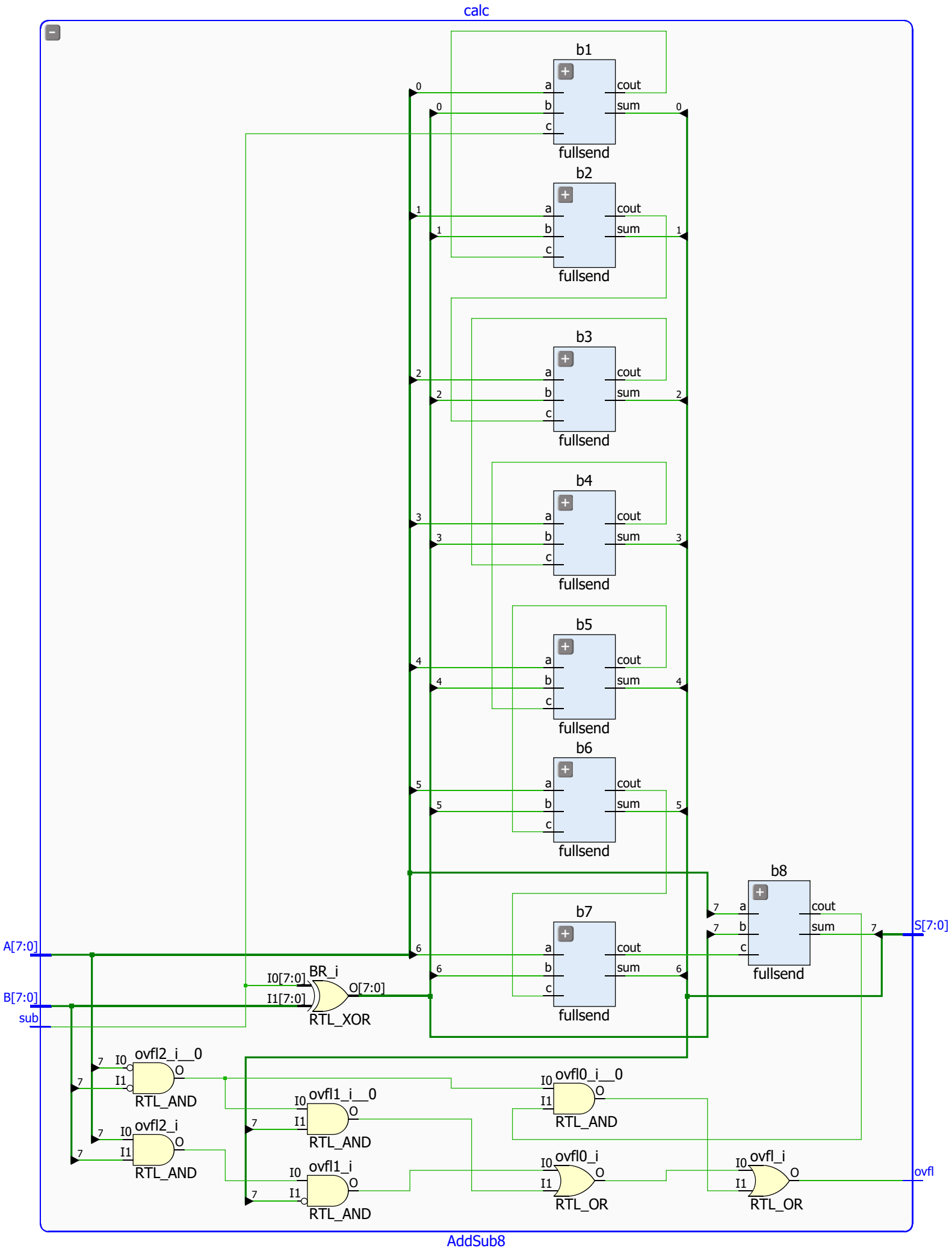
```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
////////////
// Company:
// Engineer:
//
// Create Date: 04/17/2018 12:02:06 PM
// Design Name:
// Module Name: fux8to1
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
////////////

//8 bit mux
//input 1 bit enable,
//1 bus of 8 bit width input,
//and 1 bus of 3 bit width selector
//output is one bit

module fux8to1(
    input [7:0] in,
    input [2:0] s,
    input e,
    output out
    );

    wire m1, m2;

    fux4to1 mux1 (.s(s[1:0]), .i(in[3:0]), .e(1), .out(m1));
    fux4to1 mux2 (.s(s[1:0]), .i(in[7:4]), .e(1), .out(m2));
    assign out = (m1&~s[2] | m2&s[2])^e;

endmodule
```


toggle
out_i
in1[7:0]
S=1'b1
I0[7:0]
O[7:0]
out[7:0]
in2[7:0]
S=default
I1[7:0]
RTL_MUX
S
s
BusFux

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 04/17/2018 12:13:08 PM
// Design Name:
// Module Name: BusFux
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

//bussed mux
//takes in 2 busses in1 and in2
//outputes either entire bus 1 or bus 2

//input = 8bit bus
//output = 8bit bus


module BusFux(
    input [7:0] in1,
    input [7:0] in2,
    input s,
    output [7:0] out
    );

    assign out = ({8{s}} & in1) | ({8{~s}} & in2);
endmodule
```

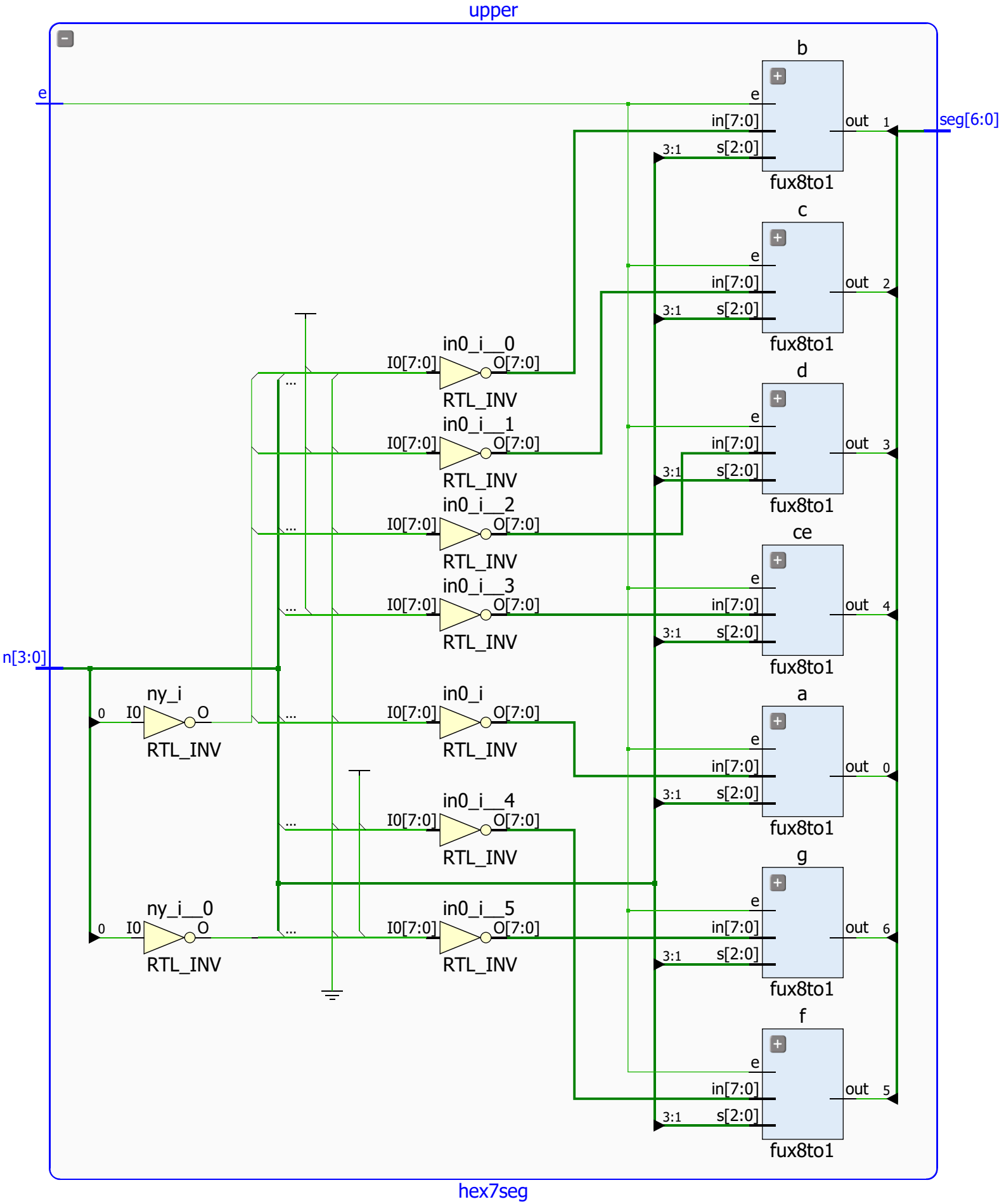

calc
b6
carry
e
i[3:0]
s[1:0]
out
cout
fux4to1
i0_i
I0
O
RTL_INV
a
b
c
0
0
summed
e
i[3:0]
s[1:0]
out
sum
fux4to1
fullsend
AddSub8

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 04/17/2018 12:30:15 PM
// Design Name:
// Module Name: fullsend
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

//full bit adder using muxes
//input single bits
//3 inputs a, b, carryin
//2 outputs sum and carry bit

module fullsend(
    input a,
    input b,
    input c,
    output sum,
    output cout
    );
    fux4to1 summed (.s({c,b}), .i({a, ~a, ~a,
a}), .e(1), .out(sum)); //sum mux
    fux4to1 carry (.s({a,b}), .i({1'b1,c,c,
1'b0}), .e(1), .out(cout)); //carry mux
endmodule
```


calc
b1
b2
b3
b4
b5
b6
b7
b8
fullsend
a
b
c
cout
sum
A[7:0]
B[7:0]
sub
S[7:0]
BR_i
I0[7:0]
I1[7:0]
O[7:0]
RTL_XOR
ovfl2_i__0
ovfl2_i
ovfl1_i__0
ovfl1_i
ovfl0_i__0
ovfl0_i
ovfl_i
RTL_AND
RTL_OR
I0
I1
O
ovfl
AddSub8

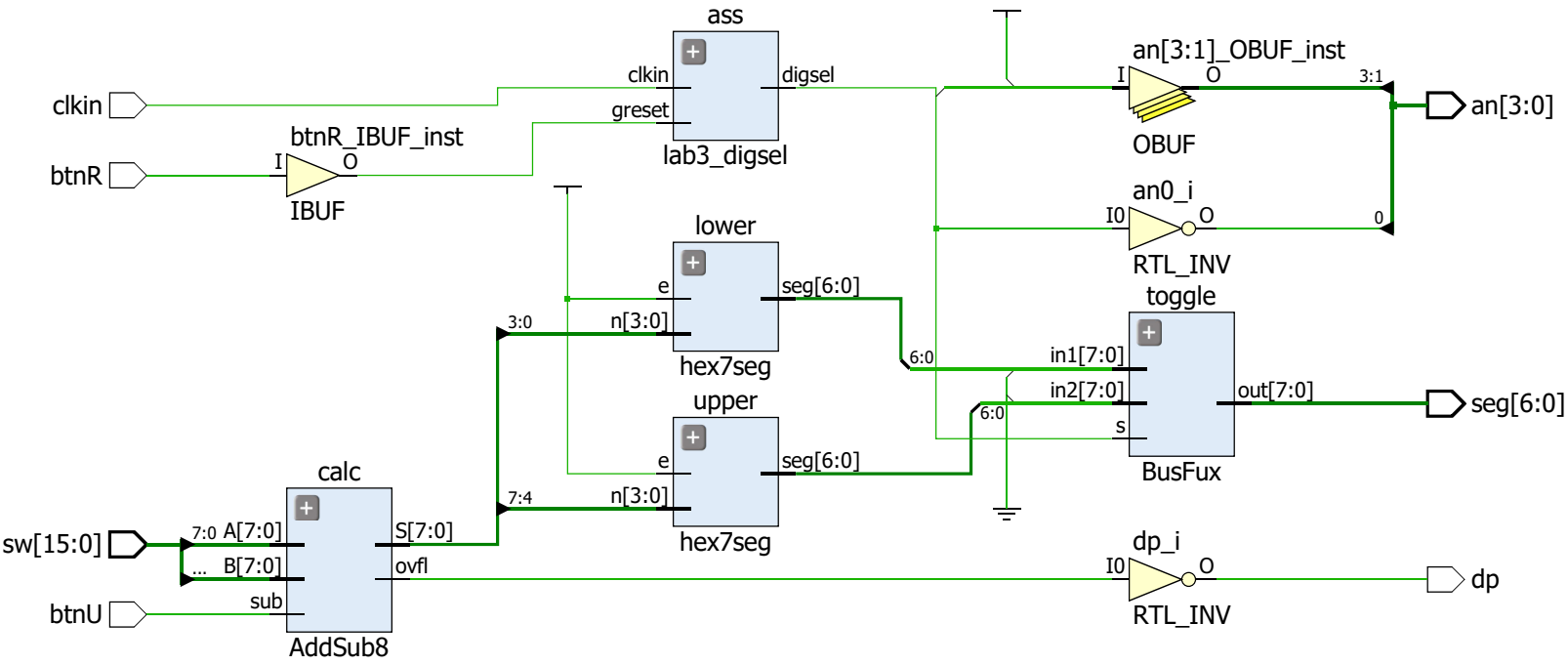
```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
////////////
// Company:
// Engineer:
//
// Create Date: 04/17/2018 01:01:28 PM
// Design Name:
// Module Name: addShit
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
////////////

//8 bit full adder
//input 2 8 bit bus numbers and subtraction option
//output 1 8 bit bus number and overflow bit
module AddSub8(
    input [7:0] A, //number 1
    input [7:0] B, //number 2
    input sub, //sub function
    output [7:0] S, //sum number
    output ovfl //overflow bit
    );

    wire [7:0] BR; //correctly inverted or not B input
    wire c1, c2, c3, c4, c5, c6, c7, c8; //carry wires

    //the inverse of B if needed to do 2's compliment
    //xor is inverting if needed
    assign BR = ({8{sub}} ^ B);

    fullsend b1(.a(A[0]), .b(BR[0]), .c(sub), .sum(S[0]), .cout(c1));
    fullsend b2(.a(A[1]), .b(BR[1]), .c(c1), .sum(S[1]), .cout(c2));
    fullsend b3(.a(A[2]), .b(BR[2]), .c(c2), .sum(S[2]), .cout(c3));
    fullsend b4(.a(A[3]), .b(BR[3]), .c(c3), .sum(S[3]), .cout(c4));
    fullsend b5(.a(A[4]), .b(BR[4]), .c(c4), .sum(S[4]), .cout(c5));
    fullsend b6(.a(A[5]), .b(BR[5]), .c(c5), .sum(S[5]), .cout(c6));
    fullsend b7(.a(A[6]), .b(BR[6]), .c(c6), .sum(S[6]), .cout(c7));
    fullsend b8(.a(A[7]), .b(BR[7]), .c(c7), .sum(S[7]), .cout(c8));
    assign ovfl = (A[7] & B[7]&~S[7]) | (~A[7]&~B[7]&S[7]) | (~A[7] &
```

```
~B[7] & c8);

endmodule
```


upper
e
n[3:0]
seg[6:0]
b
c
d
ce
a
g
f
fux8to1
e
in[7:0]
s[2:0]
3:1
out
in0_i__0
in0_i__1
in0_i__2
in0_i__3
in0_i__4
in0_i__5
in0_i
I0[7:0]
O[7:0]
RTL_INV
ny_i
ny_i__0
I0
O
0
hex7seg

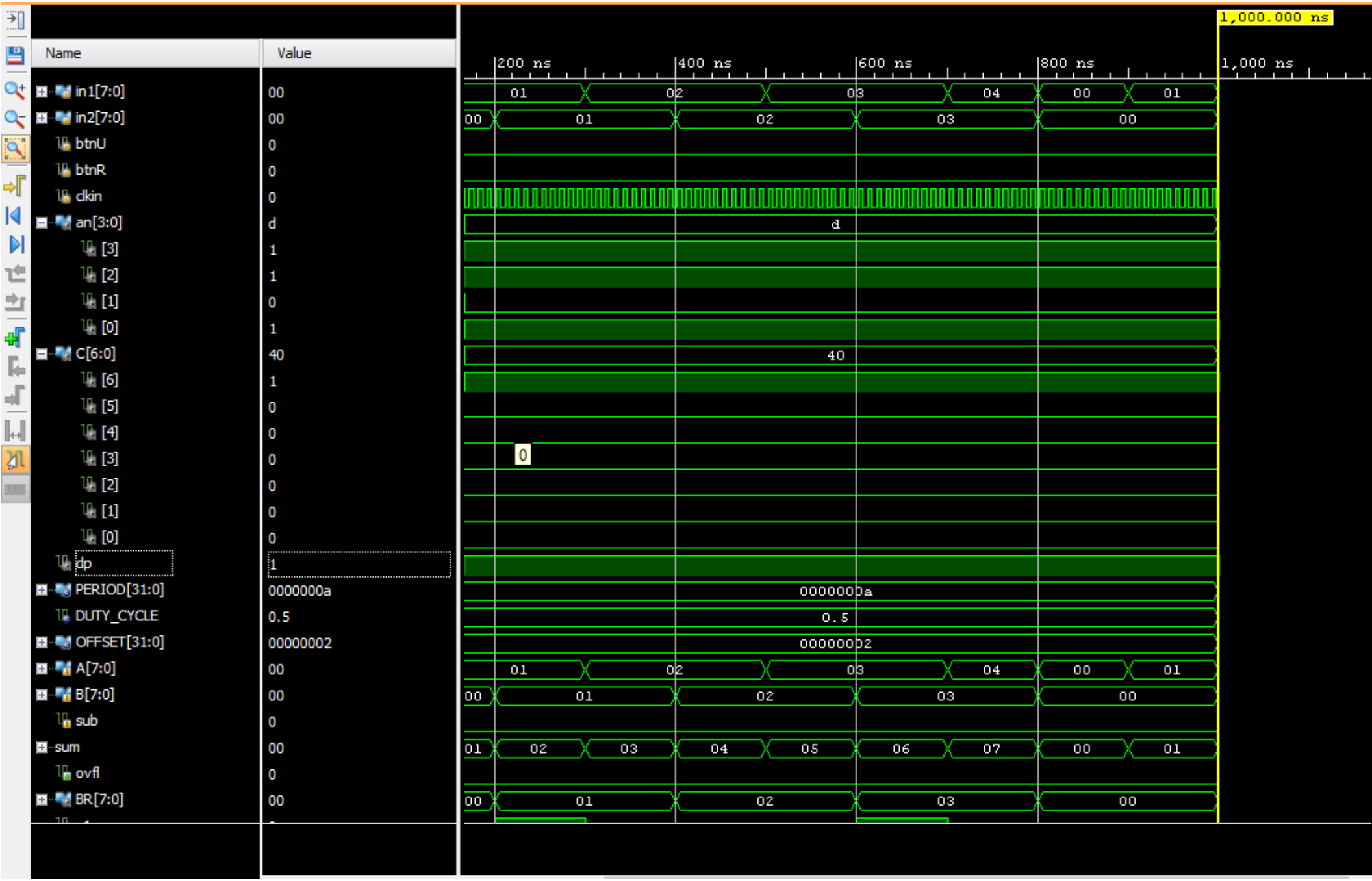
```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
////////////
// Company:
// Engineer:
//
// Create Date: 04/19/2018 01:35:26 AM
// Design Name:
// Module Name: hex7seg
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
////////////


//interface with 7 segment display

module hex7seg(
    input [3:0] n,//4 bit input number
    input e, //enable pin
    output [6:0] seg // segment output

    );

    wire y, ny, x, z, w;
    assign y = n[0];
    assign ny = ~n[0];
    assign x = n[1];
    assign w = n[2];
    assign z = n[3];

    //a led,
    //min terms: 1, 4, 11, 13
    fux8to1 a (.in(~{1'b0,y,y,1'b0,1'b0,ny,
1'b0,y}),.s({z,w,x}), .e(e), .out(seg[0]));

    //b led
    //min terms: 5, 6, 11, 12, 14, 15
    fux8to1 b (.in(~{1'b1,ny,y,1'b0,ny,y,
1'b0,1'b0}),.s({z,w,x}), .e(e), .out(seg[1]));
```

```
    //c led
    //min terms: 2, 12, 14, 15
    fux8to1 c (.in(~{1'b1,ny,1'b0,1'b0,1'b0,1'b0,ny,
1'b0}),.s({z,w,x}), .e(e), .out(seg[2]));

    //d led
    //min terms: 1, 4, 7, 10, 15
    fux8to1 d (.in(~{y,1'b0,ny,1'b0,y,ny,
1'b0,y}),.s({z,w,x}), .e(e), .out(seg[3]));

    //e led
    //min terms: 1, 3, 4, 5, 7, 9
    fux8to1 ce (.in(~{1'b0,1'b0,1'b0,y,y,
1'b1,y,y}),.s({z,w,x}), .e(e), .out(seg[4]));

    //f led
    //min terms: 1, 2, 3, 7, 13
    fux8to1 f (.in(~{1'b0,y,1'b0,1'b0,y,
1'b0,1'b1,y}),.s({z,w,x}), .e(e), .out(seg[5]));

    //g led
    //min terms: 0, 1, 7, 12
    fux8to1 g (.in(~{1'b0,ny,1'b0,1'b0,y,
1'b0,1'b0,1'b1}),.s({z,w,x}), .e(e), .out(seg[6]));


endmodule
```


ass
clkin
digsel
greset
lab3_digsel
clkin
btnR_IBUF_inst
I
O
IBUF
btnR
an[3:1]_OBUF_inst
I
O
3:1
OBUF
an[3:0]
an0_i
I0
O
0
RTL_INV
lower
e
seg[6:0]
n[3:0]
hex7seg
3:0
6:0
toggle
in1[7:0]
in2[7:0]
out[7:0]
s
BusFux
seg[6:0]
upper
e
seg[6:0]
n[3:0]
hex7seg
6:0
7:4
calc
7:0 A[7:0]
... B[7:0]
sub
S[7:0]
ovfl
AddSub8
sw[15:0]
btnU
dp_i
I0
O
RTL_INV
dp

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
////////////
// Company:
// Engineer:
//
// Create Date: 04/19/2018 01:52:42 AM
// Design Name:
// Module Name: TopGun
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
////////////


module TopGun(
    input [15:0] sw, //input switch a = lower bits, b = upper
    input btnU, //subtraction function
    input btnR, //greset
    input clkin, //input clock
    output [6:0] seg, //segment display
    output dp, //decimal point
    output [3:0] an //annode


    );
    wire [7:0] sum;
    wire [7:0] low;
    wire [7:0] up;
    wire over, dig_sel;

    //lab3 header provided and shit
    lab3_digsel ass (.clkin(clkin), .greset(btnR), .digsel(dig_sel));

    //calculator using switches and up button as input
    //switches are number in binary
    //up button is subtraction option
    //sum is wired to 8 bit bus
    AddSub8 calc
(.A(sw[7:0]), .B(sw[15:8]), .sub(btnU), .S(sum), .ovfl(over));
```

```
    //upper 4 bit led converter
    //converts upper 4 bits of sum into led display
    //always enabled
    //output on 8 bit bus up
    hex7seg upper(.n(sum[7:4]), .e(1), .seg(up));
    //same as previous but with lower bits
    //output bussed to low
    hex7seg lower(.n(sum[3:0]), .e(1), .seg(low));

    //chooses whitch segment display to wire
    //both lower and upper bits fed as input
    //choose connection with dig_sel wire
    //output = 8 bit bus to segment display
    assign an = {1'b1, 1'b1, dig_sel, ~dig_sel};
    assign dp = ~over;
    //assign seg = {4{1'b0}};
    BusFux toggle(.in1(low), .in2(up), .s(dig_sel), .out(seg));

    //assign JA[0] = dig_sel;

endmodule
```


Name
Value
in1[7:0]
00
in2[7:0]
00
btnU
0
btnR
0
clkin
0
an[3:0]
d
[3]
1
[2]
1
[1]
0
[0]
1
C[6:0]
40
[6]
1
[5]
0
[4]
0
[3]
0
[2]
0
[1]
0
[0]
0
dp
1
PERIOD[31:0]
0000000a
DUTY_CYCLE
0.5
OFFSET[31:0]
00000002
A[7:0]
00
B[7:0]
00
sub
0
sum
00
ovfl
0
BR[7:0]
00
200 ns
400 ns
600 ns
800 ns
1,000 ns
1,000.000 ns
01
02
03
04
00
01
00
01
02
03
00
d
40
0
0000000a
0.5
00000002
01
02
03
04
00
01
00
01
02
03
00
01
02
03
04
05
06
07
00
01
00
01
02
03
00

```
// CMPE 100
// This is a testbench for Lab 2.
// If the top level module in your Lab 2 project is named "top_lab2"
// and you used the suggested names for its inputs/outputs then
// then it will run without modification.  Otherwise follow the
instructions
// in the comments marked "TODO" to modify this testbench to conform
to your project.
`timescale 1ns/1ps

module lab2_tests();

   reg [7:0] in1; //first input number a
   reg [7:0] in2; //second input number b
   reg btnU; //sub function
   reg btnR; //global reset
   reg clkin;//switch input and subtract func
   wire [3:0] an; //anode switch
   wire [6:0] C;
   wire dp; //output and decimal point

// TODO: replace "top_lab2" with the name of your top level Lab 2
module.
   TopGun UUT (
        .sw({in2,in1}), .btnU(btnU), .btnR(btnR), .clkin(clkin), .seg(
C), .dp(dp), .an(an)
        );
// TODO: In the three lines above, make sure the pin names match the
names
// used for the inputs/outputs of your top level module.   For
example, if you
// used "cin" rather than "sw0", then replace ".sw0(sw0)" with
".cin(sw0)"

    initial
    begin

        btnU=1'b0;
        btnR=1'b0;
        in1 = {8{1'b0}}; //initialize numbers to 0
        in2 = {8{1'b0}}; //inititialize numbers to 0
        // sum is 0
        //-------------  Current Time:  0ns
        #100;  //This advances time by 100 units (ns in this case)
        in1 = {8'b00000001};
        in2 = {8'b00000000};
        // sum is 1
        // -------------  Current Time:  100ns

        #100;  //This advances time by 100 units (ns in this case)
```

```
 in1 = 8'b00000001;
 in2 = 8'b00000001;
// sum is 2
// -------------  Current Time:  100ns

 #100;  //This advances time by 100 units (ns in this case)
 in1 = 8'b00000010;
 in2 = 8'b00000001;
// sum is 3
// -------------  Current Time:  100ns

 #100;  //This advances time by 100 units (ns in this case)
 in1 = 8'b00000010;
 in2 = 8'b00000010;
// sum is 4
// -------------  Current Time:  100ns

 #100;  //This advances time by 100 units (ns in this case)
 in1 = 8'b00000011;
 in2 = 8'b00000010;
// sum is 5
// -------------  Current Time:  100ns

 #100;  //This advances time by 100 units (ns in this case)
 in1 = 8'b00000011;
 in2 = 8'b00000011;
// sum is 6
// -------------  Current Time:  100ns

 #100;  //This advances time by 100 units (ns in this case)
 in1 = 8'b00000100;
 in2 = 8'b00000011;
// sum is 7
// -------------  Current Time:  100ns

 #100;  //This advances time by 100 units (ns in this case)
 in1 = 1'h8;
 in2 = 8'b00000000;
// sum is 8
// -------------  Current Time:  100ns

 #100;  //This advances time by 100 units (ns in this case)
 in1 = 1'h9;
 in2 = 8'b00000000;
// sum is 9
// -------------  Current Time:  100ns

 #100;  //This advances time by 100 units (ns in this case)
 in1 = 1'hA;
 in2 = 8'b00000000;
```

```
// sum is A
// -------------  Current Time:  100ns

 #100;  //This advances time by 100 units (ns in this case)
 in1 = 1'hB;
 in2 = 8'b00000000;
// sum is B
// -------------  Current Time:  100ns

 #100;  //This advances time by 100 units (ns in this case)
 in1 = 1'hC;
 in2 = 8'b00000000;
// sum is C
// -------------  Current Time:  100ns

 #100;  //This advances time by 100 units (ns in this case)
 in1 = 1'hD;
 in2 = 8'b00000000;
// sum is D
// -------------  Current Time:  100ns

 #100;  //This advances time by 100 units (ns in this case)
 in1 = 1'hE;
 in2 = 8'b00000000;
// sum is E
// -------------  Current Time:  100ns

 #100;  //This advances time by 100 units (ns in this case)
 in1 = 1'hF;
 in2 = 8'b00000000;
// sum is F
// -------------  Current Time:  100ns

 #100;  //This advances time by 100 units (ns in this case)
 in1 = 2'H1F;
 in2 = 8'b00000000;
// sum is 1F
// -------------  Current Time:  100ns

 #100;  //This advances time by 100 units (ns in this case)
 in1 = 2'h2F;
 in2 = 8'b00000000;
// sum is 2F
// -------------  Current Time:  100ns

 #100;  //This advances time by 100 units (ns in this case)
 in1 = 2'h3F;
 in2 = 8'b00000000;
// sum is 3F
// -------------  Current Time:  100ns
```

```
 #100;  //This advances time by 100 units (ns in this case)
 in1 = 2'h4F;
 in2 = 8'b00000000;
// sum is 4F
// -------------  Current Time:  100ns

 #100;  //This advances time by 100 units (ns in this case)
 in1 = 2'h5F;
 in2 = 8'b00000000;
// sum is 5F
// -------------  Current Time:  100ns

 #100;  //This advances time by 100 units (ns in this case)
 in1 = 2'h6F;
 in2 = 8'b00000000;
// sum is 6F
// -------------  Current Time:  100ns

 #100;  //This advances time by 100 units (ns in this case)
 in1 = 2'h7F;
 in2 = 8'b00000000;
// sum is 7F
// -------------  Current Time:  100ns

 #100;  //This advances time by 100 units (ns in this case)
 in1 = 2'h8F;
 in2 = 8'b00000000;
// sum is 8F
// -------------  Current Time:  100ns

 #100;  //This advances time by 100 units (ns in this case)
 in1 = 2'h9F;
 in2 = 8'b00000000;
// sum is 9F
// -------------  Current Time:  100ns

 #100;  //This advances time by 100 units (ns in this case)
 in1 = 2'hAF;
 in2 = 8'b00000000;
// sum is AF
// -------------  Current Time:  100ns

 #100;  //This advances time by 100 units (ns in this case)
 in1 = 2'hBF;
 in2 = 8'b00000000;
// sum is BF
// -------------  Current Time:  100ns

 #100;  //This advances time by 100 units (ns in this case)
```

```
     in1 = 2'hCF;
     in2 = 8'b00000000;
    // sum is CF
    // -------------  Current Time:  100ns

     #100;  //This advances time by 100 units (ns in this case)
     in1 = 2'hDF;
     in2 = 8'b00000000;
    // sum is DF
    // -------------  Current Time:  100ns

     #100;  //This advances time by 100 units (ns in this case)
     in1 = 2'hEF;
     in2 = 8'b00000000;
    // sum is EF
    // -------------  Current Time:  100ns

     #100;  //This advances time by 100 units (ns in this case)
     in1 = 2'hFF;
     in2 = 8'b00000000;
    // sum is FF
    // -------------  Current Time:  100ns



//ALTERNATING SIGN TESTS

     #100;  //This advances time by 100 units (ns in this case)
in1 = 8'b00000010;
in2 = 8'b00000001;
        // sum is 3
        // -------------  Current Time:  100ns

     #100;  //This advances time by 100 units (ns in this case)
     in1 = 8'b10000010;
     in2 = 8'b00000001;
    // sum is 3
    // -------------  Current Time:  100ns


     #100;  //This advances time by 100 units (ns in this case)
     in1 = 8'b00000010;
     in2 = 8'b10000001;
    // sum is 3
    // -------------  Current Time:  100ns


     #100;  //This advances time by 100 units (ns in this case)
     in1 = 8'b10000010;
     in2 = 8'b10000001;
```

```
    // sum is 3
    // -------------  Current Time:  100ns


     #100;  //This advances time by 100 units (ns in this case)
     in1 = 8'b01000010;
     in2 = 8'b01000001;
    // sum is 3
    // -------------  Current Time:  100ns


     #100;  //This advances time by 100 units (ns in this case)
     in1 = 8'b11000010;
     in2 = 8'b01000001;
    // sum is 3
    // -------------  Current Time:  100ns

     #100;  //This advances time by 100 units (ns in this case)
     in1 = 8'b01000010;
     in2 = 8'b11000001;
    // sum is 3
    // -------------  Current Time:  100ns

     #100;  //This advances time by 100 units (ns in this case)
     in1 = 8'b11000010;
     in2 = 8'b11000001;
    // sum is 3
    // -------------  Current Time:  100ns


//SAME TESTS BUT NOW SHIT HAS THE SUB BUTTON FUNCTION ACTIVATED

btnU = 1'b1;
#100;  //This advances time by 100 units (ns in this case)
in1 = 8'b00000010;
in2 = 8'b00000001;
        // sum is 3
        // -------------  Current Time:  100ns

#100;  //This advances time by 100 units (ns in this case)
in1 = 8'b10000010;
in2 = 8'b00000001;
    // sum is 3
    // -------------  Current Time:  100ns


#100;  //This advances time by 100 units (ns in this case)
in1 = 8'b00000010;
in2 = 8'b10000001;
    // sum is 3
```

```
    // -------------  Current Time:  100ns


#100;  //This advances time by 100 units (ns in this case)
in1 = 8'b10000010;
in2 = 8'b10000001;
    // sum is 3
    // -------------  Current Time:  100ns


#100;  //This advances time by 100 units (ns in this case)
in1 = 8'b01000010;
in2 = 8'b01000001;
    // sum is 3
    // -------------  Current Time:  100ns


#100;  //This advances time by 100 units (ns in this case)
in1 = 8'b11000010;
in2 = 8'b01000001;
    // sum is 3
    // -------------  Current Time:  100ns

#100;  //This advances time by 100 units (ns in this case)
in1 = 8'b01000010;
in2 = 8'b11000001;
    // sum is 3
    // -------------  Current Time:  100ns

#100;  //This advances time by 100 units (ns in this case)
in1 = 8'b11000010;
in2 = 8'b11000001;
    // sum is 3
    // -------------  Current Time:  100ns


end

parameter PERIOD = 10;
 parameter real DUTY_CYCLE = 0.5;
 parameter OFFSET = 2;

 initial    // Clock process for clkin
 begin
     #OFFSET
       clkin = 1'b1;
     forever
     begin
         #(PERIOD-(PERIOD*DUTY_CYCLE)) clkin = ~clkin;
     end
```

```
     end

    initial
    begin
      // add your stimuli here
      // to set signal foo to value 0 use
      // foo = 1'b0;
      // to set signal foo to value 1 use
      // foo = 1'b1;
      //always advance time my multiples of 100ns
      // to advance time by 100ns use the following line
      #100;
     end
endmodule
```

Lab 3

4/17/18

Full 8-bit Calculator w/ subtraction function

- using busses & muxes

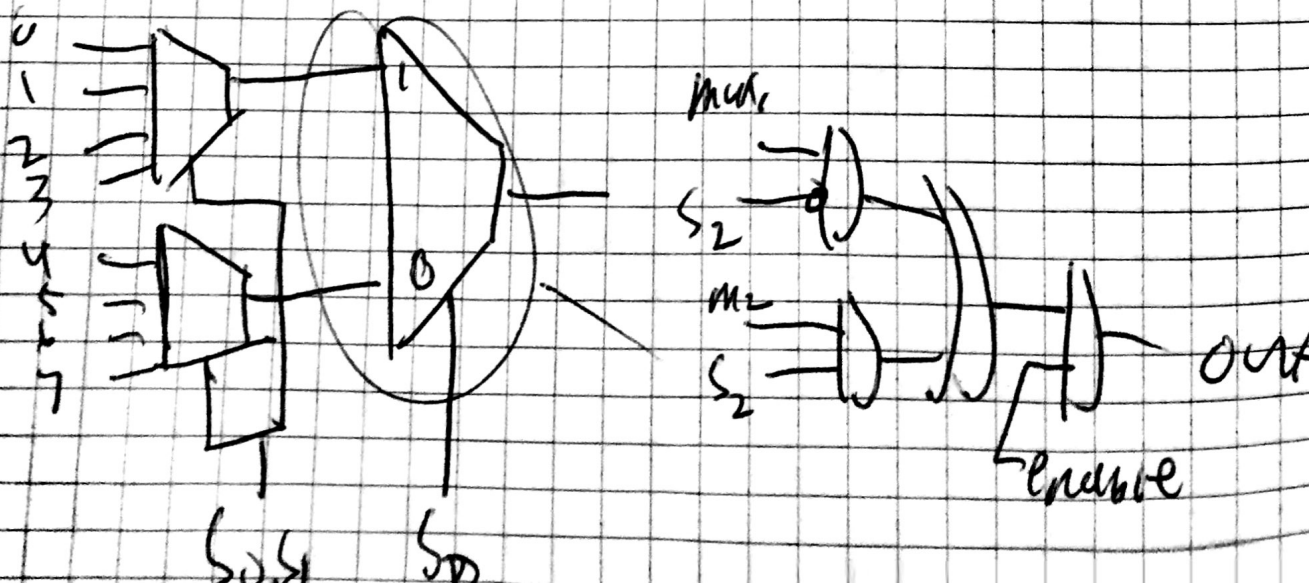
3 muxes

Mux 4:1 4 input, 2 select bits, enable bit, 1 output bit

- 0 when enable = 0
- output select bit when e = 1

Mux 8:1 8 input, 3 selector bits, enable, output

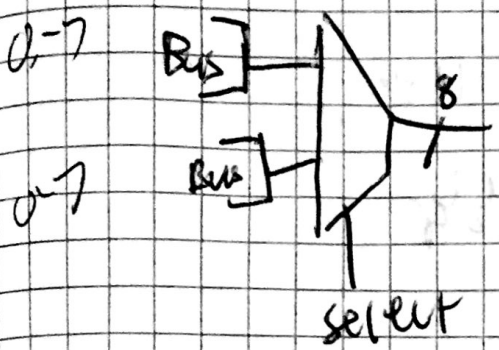
- 0 when enable = 0
- output ~~nothing~~ selector bit
- implement w/ 2 4-bit muxes



M 2 → 1 x 8 2 inputs — 8 bits each

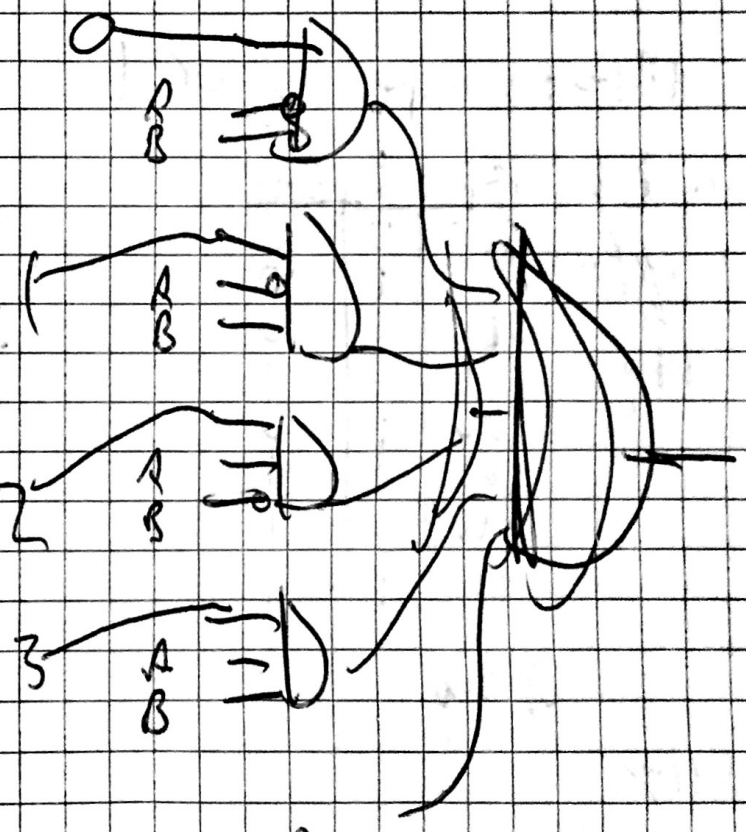
1 output — 8 bits wide

1 selector bit



4-1 mux

a	b	0	1	2	3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1



7 input, 1 out

$$\left((\bar{A}\bar{B}I_0) + (\bar{A}BI_1) + (A\bar{B}I_2) + (ABI_3)\right) \text{Enable}$$

OR

Anded w/ Enable

add/sub

Can only use muxes

4/17

- 2 4-1 muxes
- one inverter
- create full adder from muxes?
- create 8 bit adder from full adder

~~A~~ are selector bits

A-B

inputs are 0 or 1

A	B	C	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Sum

C \ ab	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$\bar{a}\bar{b}=C$ $\bar{a}b$ $a\bar{b}$ ab

$\bar{C}$ C $\bar{C}$

a and b= selector bits

$\bar{C}$, C is input

Carry

C \ ab	00	01	11	10
0	0	0	1	0
1	0	1	1	1

$\bar{a}\bar{b}$ $\bar{a}b$ $a\bar{b}$ ab

0 C 1 C

output dependent on selector bits & input, since C is input

Bus mux

$S=1 \Rightarrow$ and S w/ input 1

$S=0 \Rightarrow$ and $\bar{S}$ w/ input 2

$$(Bus1 \cdot S) + (Bus2 \cdot \bar{S})$$

$S=0$ 0 $S=1$ 0

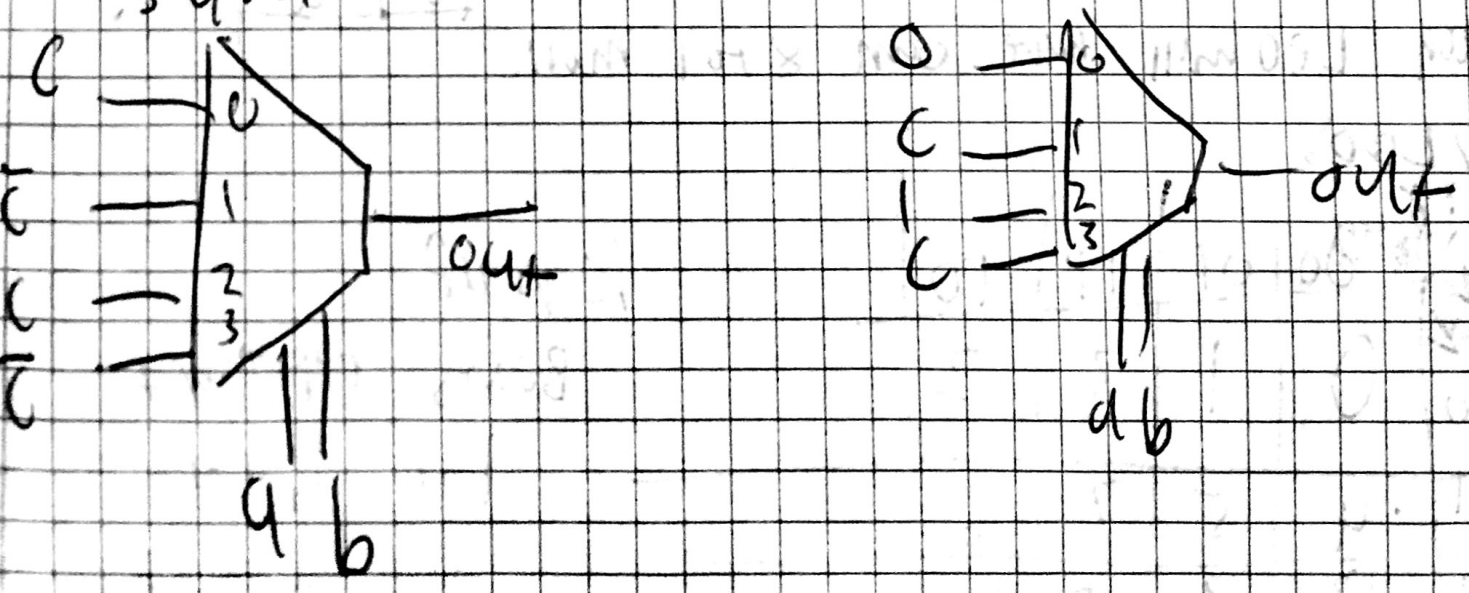
$\{S, S, S, S\}$

0 0 0 S

| | | |

0 0 0 1

Full adder 3 inputs (bit) 2 outputs (bit) carry



8-bit adder

- made of full bit adders
- bus inputs & outputs
  - 2 8bit bus inputs
  - 1 8bit bus output
- implement integer 2's complement Sub
- implement overflow option

Inputs: 8bit A,B (bus) | Sub (1bit)

Outputs: 8bit Sum (bus) | overflow (1bit)

Y = selector bit

X \ Y	0	1
00	0	1
01	2	3
11	6	7
10	4	5

7-segment display

input = 4-bit #

each LED will have own 8 to 1 mux

7 LEDs

	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

K map

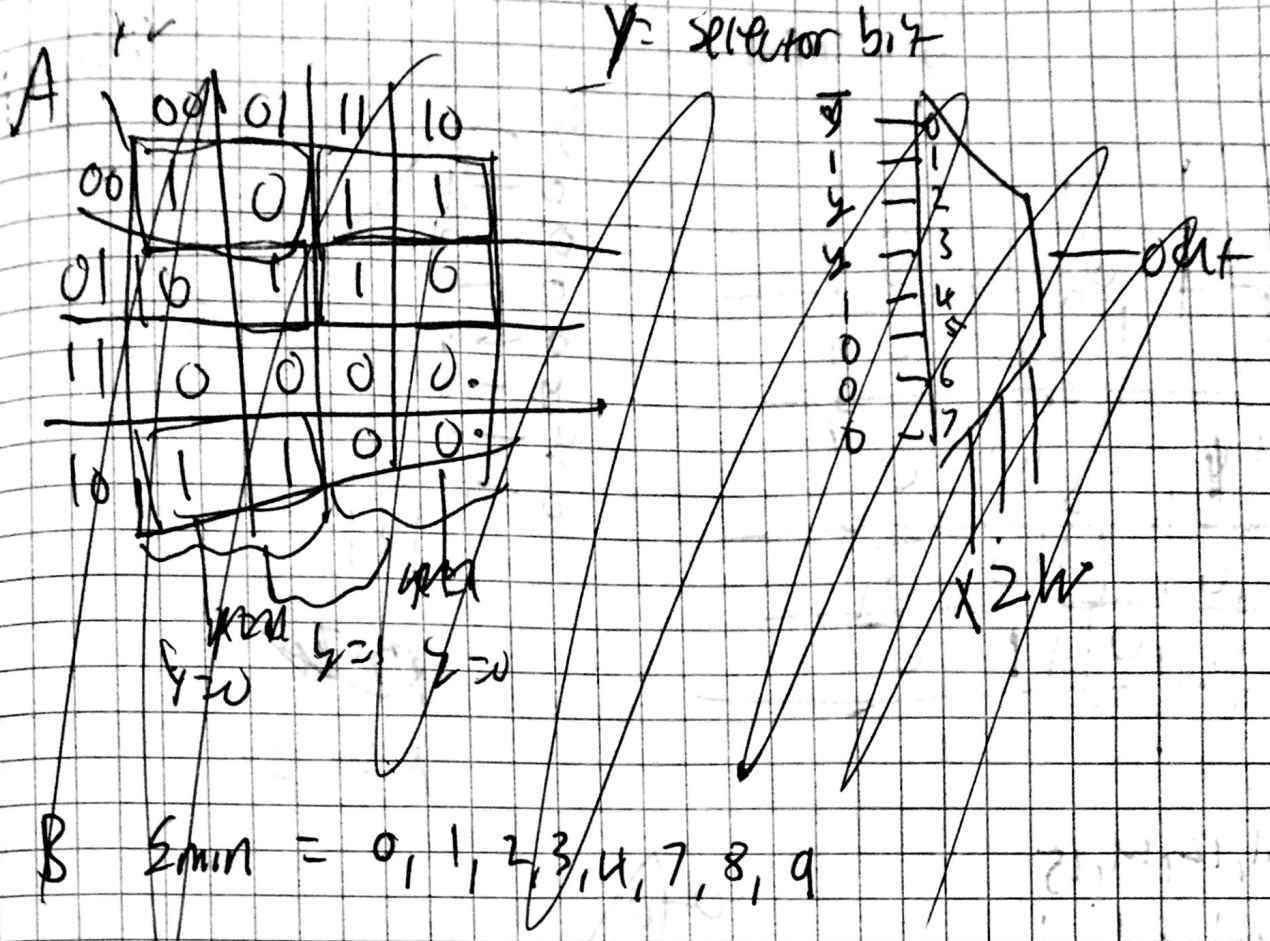
Boxes represent hex digit

By ordering

$Y = A_0$ $Z = A_3$

$X = A_1$

$W = A_2$



B $S_{min} = 0, 1, 2, 3, 4, 7, 8, 9$

#	n_0	n_1	n_2	n_3	A	B	C	D	E	F	G
0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	1	1	0	0	1	1	1	1
2	0	0	1	0	0	0	1	0	0	1	0
3	0	0	1	1	0	0	0	0	1	1	0
4	0	1	0	0	1	0	0	1	1	0	0
5	0	1	0	1	0	1	0	0	1	0	0
6	0	1	1	0	0	1	0	0	0	0	0
7	0	1	1	1	0	0	0	1	1	1	1
8	1	0	0	0	0	0	0	0	0	0	0
4	1	0	0	1	0	0	0	0	1	0	0
10 A	1	0	1	0	0	0	0	1	0	0	0
11 B	1	0	1	1	1	1	0	0	0	0	0
12 C	1	1	0	0	0	1	1	0	0	0	1
13 D	1	1	0	1	1	0	0	0	0	1	0
14 E	1	1	1	0	0	1	1	0	0	0	0
15 F	1	1	1	1	0	1	1	1	0	0	0

A = 1, 4, 11, 13 B = 5, 6, 11, 12, 14, 15 C = 2, 12, 14, 15

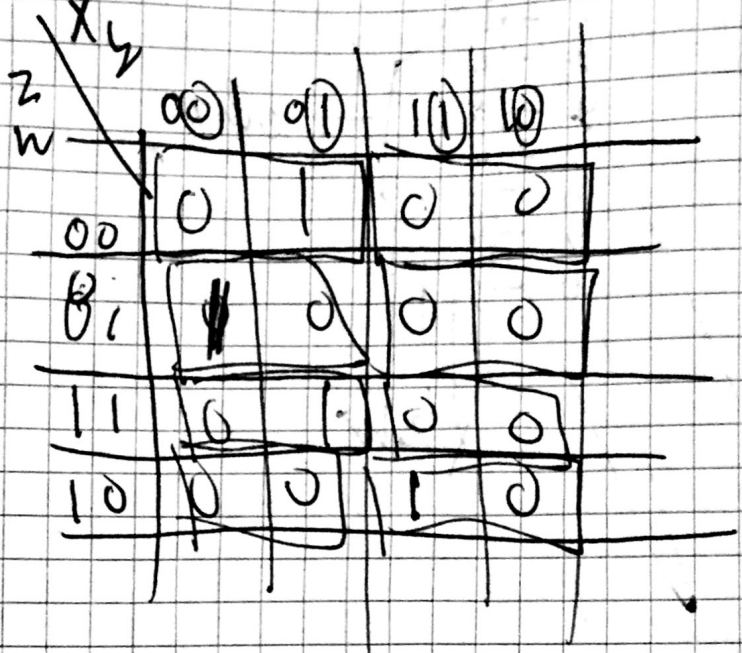
D = 1, 4, 7, 10, 15 E = 1, 3, 4, 5, 7, 9 F = 1, 2, 3, 7, 13

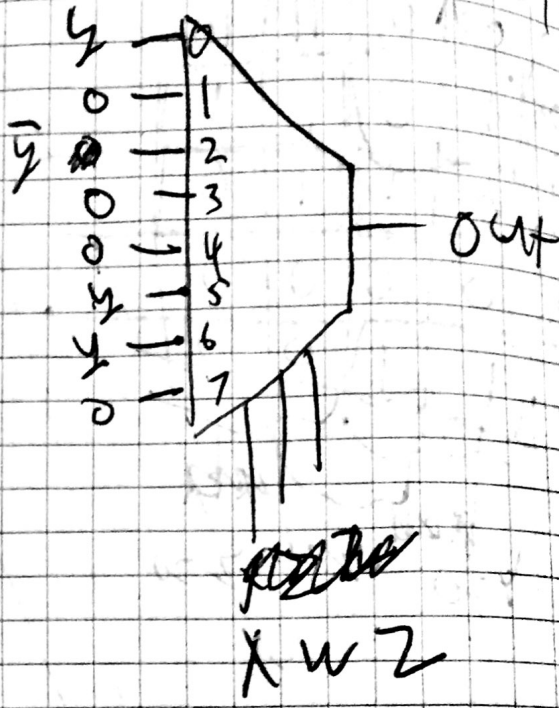
G = 0, 1, 7, 12

Min terms (remaining)

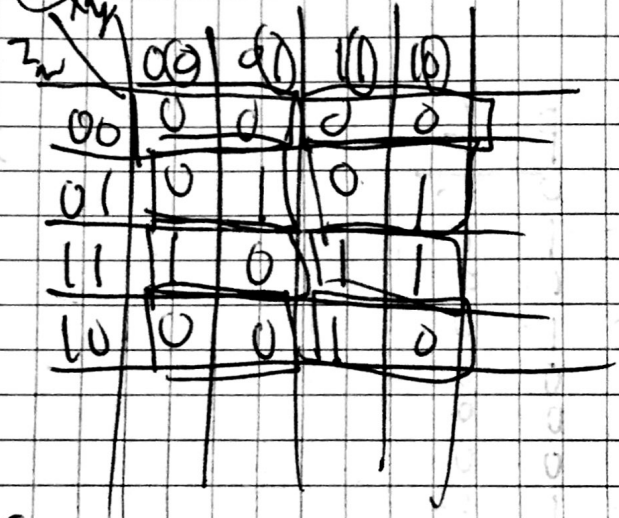
y = input

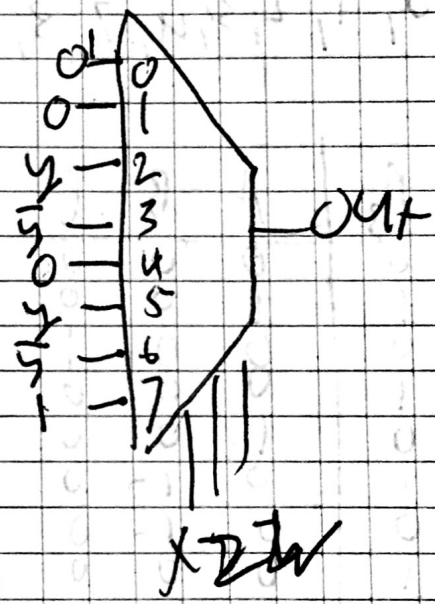
(A)



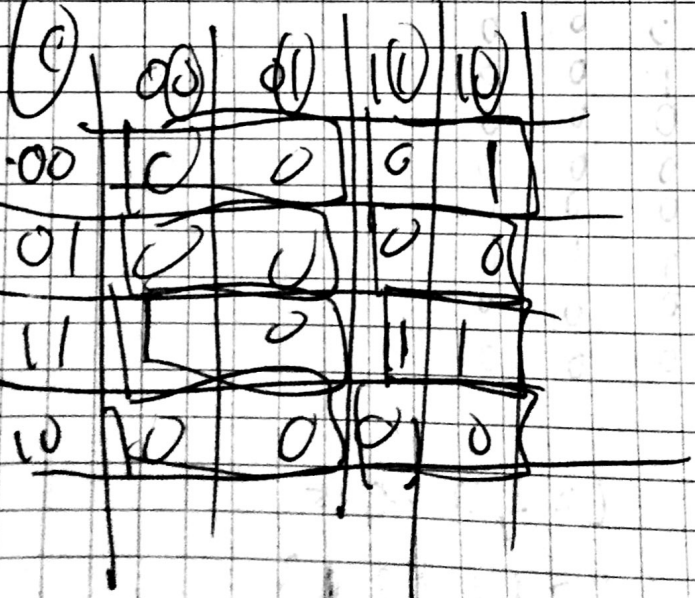


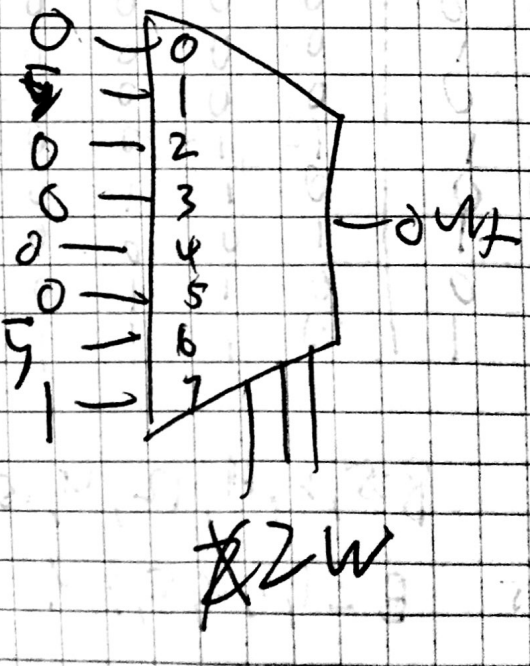
(B) 5, 6, 11, 12, 14, 15





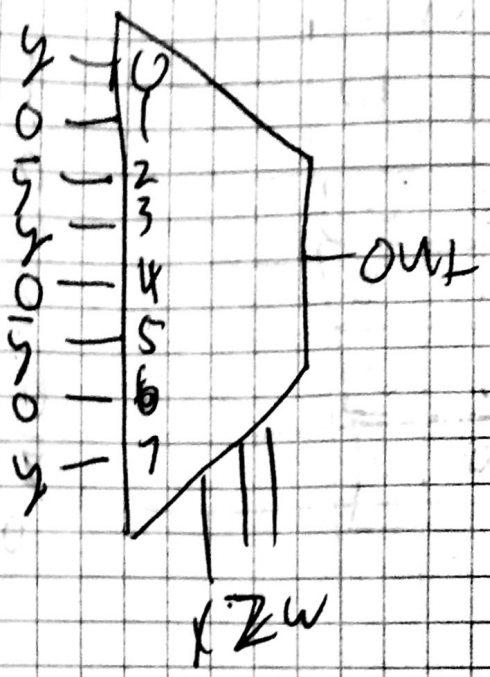
(C)





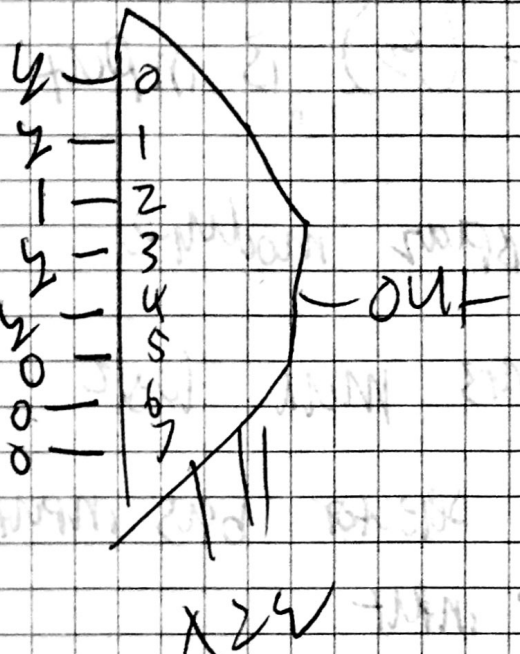
(D)

	00	01	11	10
00	0	1	0	0
01	1	0	1	0
11	0	0	1	0
10	0	0	0	1



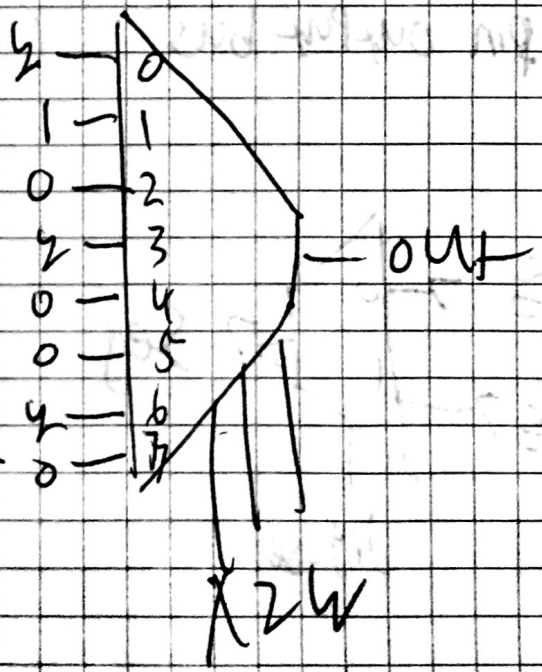
(E)

	00	01	11	10
00	0	1	1	0
01	1	1	1	0
11	0	0	0	0
10	0	1	0	0



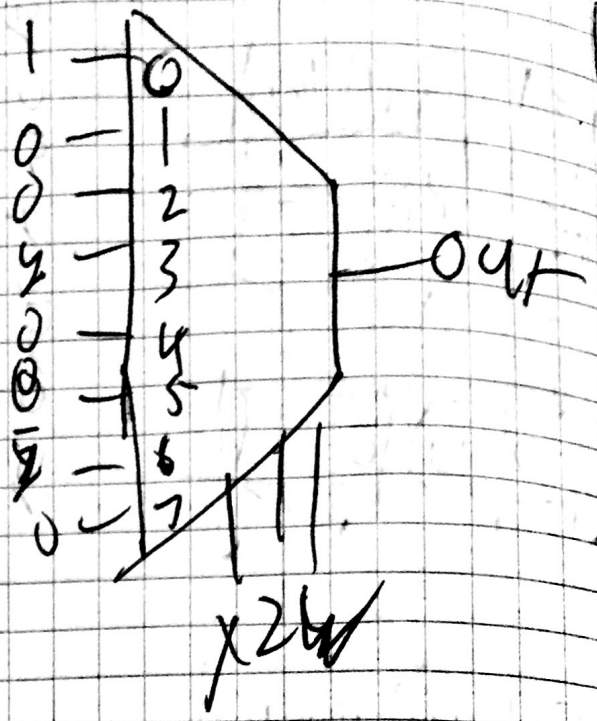
(F)

	00	01	11	10
00	0	1	1	1
01	0	0	1	0
11	0	1	0	0
10	0	0	0	0



(g)

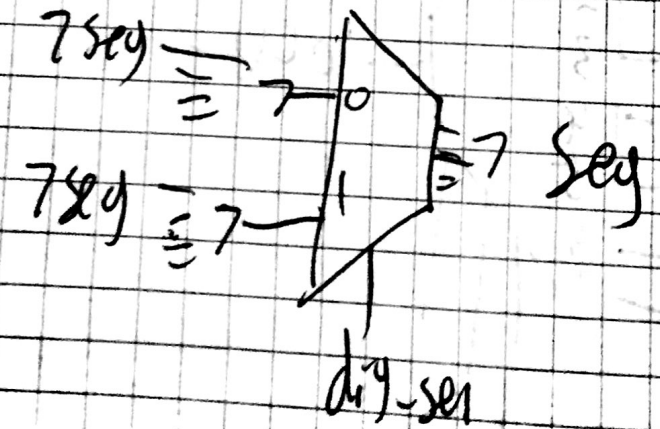
	00	01	11	10
00	1	1	0	0
01	0	0	1	0
11	1	0	0	0
10	0	0	0	0



2nd bit (2) is input

7 Seg display module

- implements mux log 2
- 4 bit selector bus input (n)
- enable input
- 7 bit pin output bus (seg)



Lab 3: 7384
24 April 2018
Turned in 20 April 2018
KL

2 8 bit #'s = 256 possible #'s

01

overflow (log2)

MSB

00 → 1 overflow

11 → 0 overflow

A	B	sum	OVFL
0	0	0	0
1	1	1	0
1	0	1	0
1	0	0	0
0	1	1	0
0	1	0	0
0	0	1	1
1	1	0	1

OVFL

MSB Changed

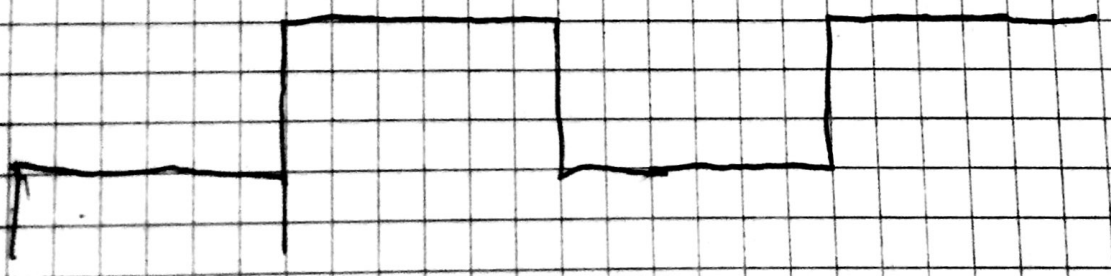
~A ∧ ~B ∧ C

A ∧ B ∧ ~C

~A ∧ ~B ∧ C

← input resets

4-sel piped to JA



.5 µs per LED

3 divisions

6 divisions

division = .5 µs

6 divisions = 3 µs Period