
Contents

Contents	1
1 Calculator With Friendly Output	3
1.1 Objective	3
1.2 System Architecture	3
1.3 sigUnsign Module	4
1.4 Bonus Ovf Logic	10
1.5 Pin Assignment	10
1.6 Turn in	10

When clicking on a link in Adobe use alt+arrow left to return to where you started.

Laboratory 1

Calculator With Friendly Output

1.1 Objective

The objective of this lab is to modify existing code to add increased functionality. The design requires utilization of basic building block and custom combinational logic blocks to realize a complex digital circuit.

Basic Calculator

This week you are going to build a very basic calculator that can add or subtract 4-bit values. On the surface, this should require nothing more than connecting some slide switches to the x and y inputs of an adder/subtractor which sends its output to a 7-segment display. And for the most part this is correct. However, instead of displaying the input and output of the adder as hexadecimal values, you will display them as 2-digit decimal values. The user input and output are shown in Figure 1.1. The user enters a pair of 4-bit operands using the left-most slide switches, **xSlide** and **ySlide**. The value entered for **xSlide** is displayed on the two (red) **xDisplay** 7-segment displays. The value entered for **ySlide** is displayed on the two (green) **yDisplay** 7-segment displays. The leftmost the **addSub** buttons specify the operation performed on **xSlide** and **ySlide**. The result is **xSlide + ySlide** or **xSlide - ySlide**. The **interp** button determines how the values are displayed on the 7-segment display. When unpressed, the 7-segment displays show the decimal value, when pressed, the 7-segment displays show 2's complement. This will be explained in the next section. As we have only 4 7SDs on board, the same 7SD of operand Y will be used to show the operation result (yellow) when the **yOrResult** button is pressed.

1.2 System Architecture

The system architecture shown in Figure 1.2 shows the adder subtractor processing the xSlide and ySlide inputs. The 4-bit x, y and result values are processed by the sigUnsig box before being displayed on the 7-segment displays. It is now time to turn our attention to this module.

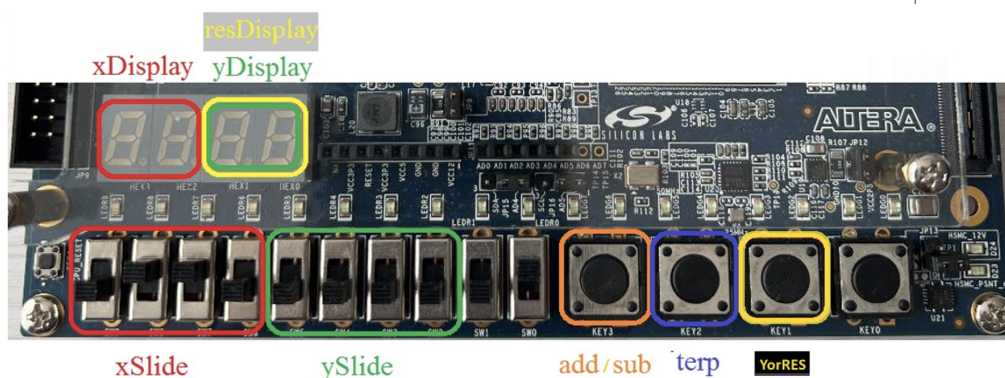


Figure 1.1: The input and output of the calculator digital circuit.

1.3 sigUnsign Module

The significant design problem in today's lab comes in this section, building the sigUnsig module. This module takes in a 4-bit value and displays a 2-digit signed or unsigned representation on a pair of 7-segment displays. Before we go into the internal organization of this module, look at its module declaration in Listing 1.

The 4-bit input x is interpreted as either signed (2's complement value) when $interp = 1$ or unsigned (regular binary number) when $interp = 0$. The interpreted value is displayed on the pair of 7-segment display with the tens-digit, blank, or minus sign being displayed by *msDisplay* and the units digit being displayed by *lsDisplay*. If the *ovf* input equals 1, the conversion is overruled and both displays show "X" (which looks a lot like a capital letter "H"). Because we will need it in the next section, Figure 1.3 is the logical arrangements of segments in a 7-segment display. Remember that the segments are active low, meaning a logic 0 illuminates a segment. Thus, the 7-bit code 7'b0100100 illuminates the pattern "2".

Let's start the design of the sigUnsig module by looking at the high-level input/output of the module by completing Table 1. Do this by filling in the segments of the 7-segment displays that are illuminated for each of the inputs. Then write the binary and hexadecimal value to illuminate those patterns.

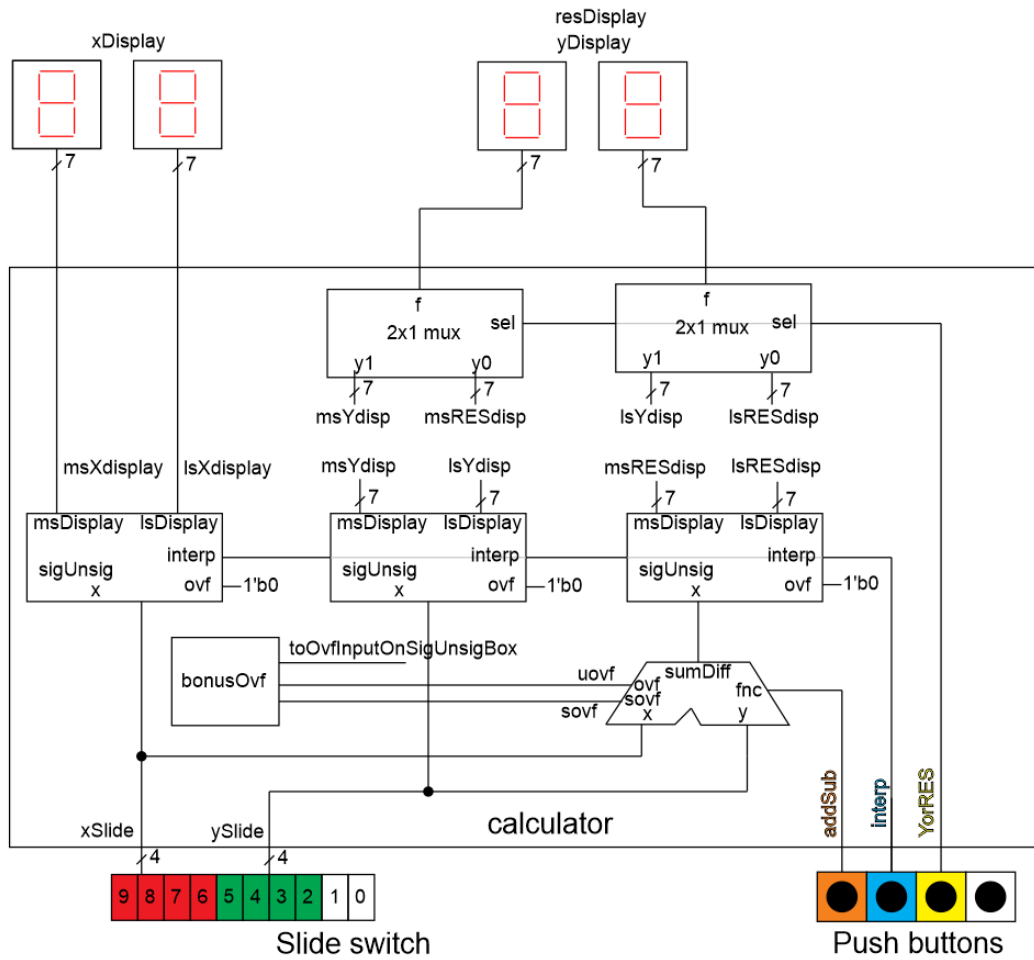
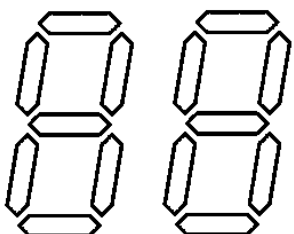
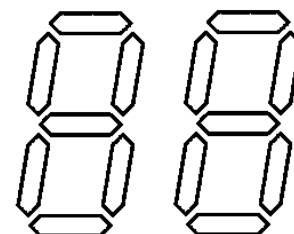
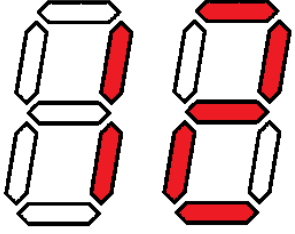
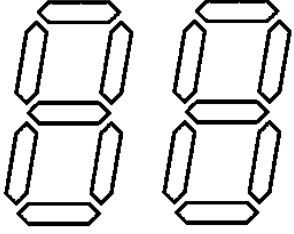
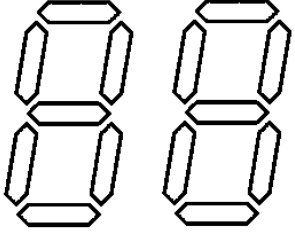
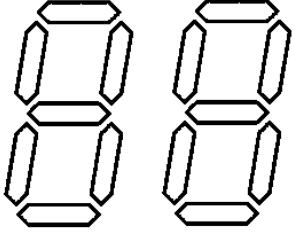


Figure 1.2: The system architecture of the calculator.

Table 1.1: Table 1: For each set of inputs to the signUnsig module, determine the 7-segment display pattern.

Input	7-segment pattern	Input	7-segment pattern
4'b0010 interp = 1 ovf = 0	 msDisplay = 7'b lsDisplay = 7'b	4'b0111 interp = 0 ovf = 0	 msDisplay = 7'b lsDisplay = 7'b

Input	7-segment pattern		Input	7-segment pattern
4'b1100 interp = 0 ovf = 0	 $msDisplay = 7'b1111001$ $= 7'h79$ $lsDisplay = 7'b0100100 = 7'h24$		4'b1000 interp = 1 ovf = 0	 $msDisplay = 7'b$ $lsDisplay = 7'b$
4'b1100 interp = 1 ovf = 0	 $msDisplay = 7'b$ $lsDisplay = 7'b$		4'b1010 interp = 1 ovf = 1	 $msDisplay = 7'b$ $lsDisplay = 7'b$

In order to better understand the output from the `sigUnsig` box, complete Table 2 by filling in the values of *msDisplay* and *lsDisplay* for a signed and unsigned interpretation – assume that *ovf*=0 while completing this table. If the interpreted value is positive and a single digit then leave *msDisplay* blank. If the interpreted value is negative then assign *msDisplay* “-“. If the interpreted value is greater than 10, assign *msDisplay* “1”.

For example, let the 4-bit input *x* equal to 4'b1100. If *x* is interpreted as unsigned then its value is 12. In this case your hardware should assign *msDisplay* “1” and *lsDisplay* “2”. If *x* is interpreted as a signed value, the 7-segment displays should show -4, by assigning *msDisplay* “-” and *lsDisplay* “4”. Complete the remaining rows of the table.

Table 1.2: Table 2: The output of the `sigUnsig` module when *ovf*=0.

4-bit input <i>x</i>	<i>interp</i> = 0 Unsigned		<i>interp</i> = 1 Signed	
	<i>msDisplay</i>	<i>lsDisplay</i>	<i>msDisplay</i>	<i>lsDisplay</i>
4'b0000	blank	0	blank	0
4'b0001				
4'b0010				
4'b0011				
4'b0100				
4'b0101				
4'b0110				
4'b0111				
4'b1000				

4-bit input x	<i>interp</i> = 0 Unsigned		<i>interp</i> = 1 Signed	
	msDisplay	lsDisplay	msDisplay	lsDisplay
4'b1001				
4'b1010				
4'b1011				
4'b1100	1	2	-	4
4'b1101				
4'b1110				
4'b1111				

Note, when there is overflow, you should assign both the *msDisplay* and *lsDisplay* “X”.

You will assign the *msDisplay* output one of four values (three from Table 2 and the “X” for overflow) using a 4:1 mux that is provided to you on Canvas. You will arrange the inputs to this mux using the logic that you will complete in Listing 2. Note that the four data inputs to this mux (*y0*, *y1*, *y2*, *y3*) are constants; the inputs to this mux do not depend on *x*.

You will assign the *lsDisplay* output one of four values (three from Table 2 and the “X” for overflow) using a 4:1 mux that is provided to you on Canvas. You will arrange the inputs to this mux using the logic that you will complete in Listing 2. Some of the data inputs to the mux depend on *x*. For example, if *interp* = 1 (signed) and if *x* is less than 0, then *lsDisplay* should show the 2’s complement of *x* (and *msDisplay* should display “-”). Instead of flipping the bits and adding 1, you should form the negative of *x* by subtracting *x* from 0. On the other hand, if *interp* = 0 (unsigned) and if *x* is greater than or equal to 10, then *lsDisplay* should show the units digit of *x* (and *msDisplay* should display “1”). Form this units digit by subtracting 10 from *x*.

Complete the code in Listing 2. You can assign the value blank, -, constants, *x*, or a function of *x* as needed. Note that this code is NOT to be used in your actual code for this lab.

Now we are ready to put the pieces of the sigUnsig module together. The building blocks in Figure 4 are captured in the organization described by Listing 2, along with some extra hardware.

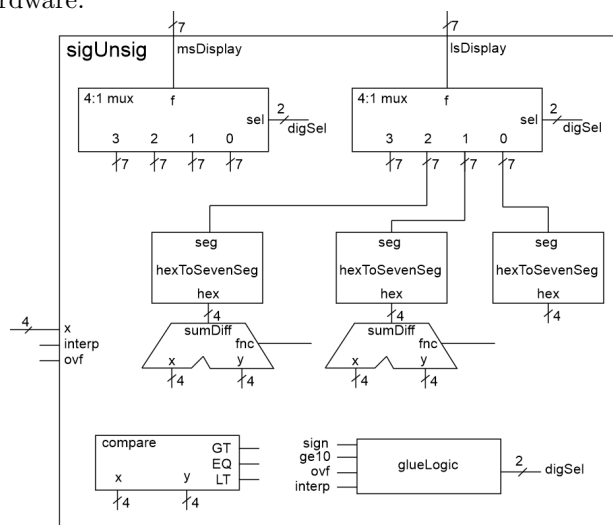


Figure 4: The internal architecture of the signUnsig module.

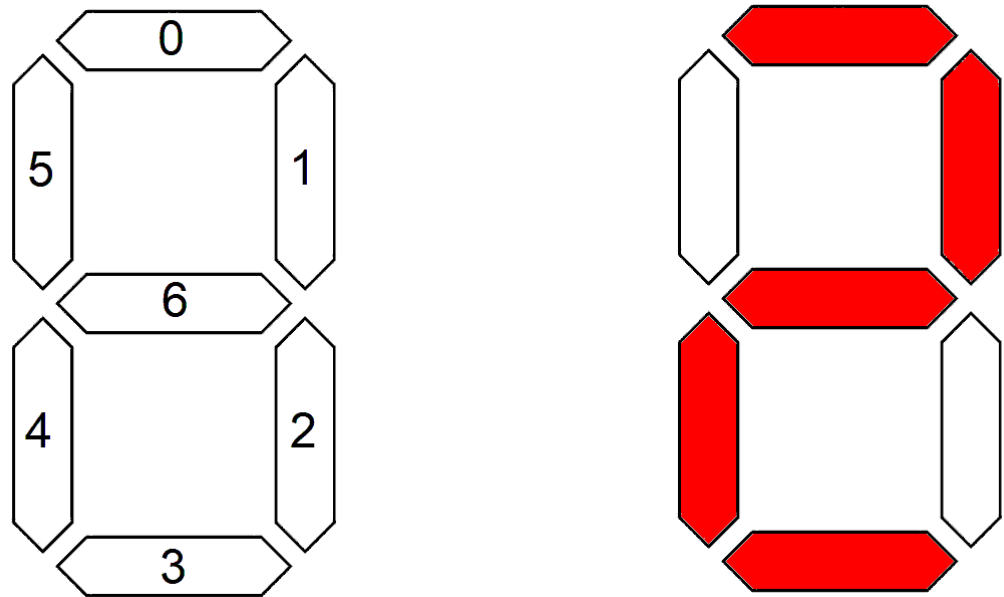


Figure 1.3: The logical arrangements of the segments in a 7-segment display.

You are responsible for connecting the inputs of the 4:1 mux and adder subtractors using the logic described in Listing 2. To help you do this, first complete Table 3. Take notice of the comments in the Listing 2 to determine which data inputs to associate with each of the muxes 4 data inputs. You should associate the overflow case with the y3 input.

Table 1.3: Table 3: The input values to the 4:1 muxes in Figure 4.

input	y3	y2	y1	y0
digSel	2'b11			
msDisplay	"X"		1	
lsDisplay				x

In addition, you should add the following to Figure 4:

- Wire the inputs of the comparator to determine to generate a signal `xGE10` which is logic 1 when `x` is greater than or equal to 10.
- Wire the inputs of the adder subtractor according to Table 3.
- Wire the input of the rightmost `hexToSevenSeg`.

The last step in building this module is to describe the behavior of the `glueLogic` box. This function chooses which input of the 4 mux inputs to route to the output. Before you do this, you will need to create a signal *sign* which equals 1 when `x` represents a negative value (when interpreted as a signed value) and equals to 0 when `x` represents a positive value (when

interpreted as a signed value). Logically speaking, this is a trivial operation – it does not require any logic gates.

Now, we can examine the contents of the glueLogic box. Do this by completing the truth table in Table 4.

Table 1.4: Table 4: Truth table for the glueLogic box.

ovf	interp	sign	xGE10	digSel
1	x	x	x	
0	0	x	0	
0	0	x	1	
0	1	0	x	
0	1	1	x	

It would make sense to use an always case statement to realize the logic in Listing 2. However, an always case statement requires each of the 16 difference cases to be explicitly enumerated. However, the truth table in Listing 2 is most efficiently described using don't cares in the input. Fortunately, the always/casez variation (note the “z” at the end of “case”) allows don't cares in the input in the form of “?”. For example, for the second row in Listing 2, the {ovf, interp, sign, xGE10} vector has don't cares for the *sign* value. Therefore, the case for this row is 4'b01?0. It is imperative that you include a “default” case whenever you use a always/case statement. This combination of cases is shown in Listing 3.

The Verilog code for the signUnsig module consists of 8 instantiation statements and an always/casez statement. For this module, I want you to:

- Use the module declaration given in Listing 1.
- Use the module definitions for
 - Generic Mux4x1 posted on this lab's Canvas folder
 - sevenSegment created in lab 02
 - genericAdderSubtractor posted on a previous lab's Canvas folder
 - genericComparator posted on a previous lab's Canvas folder
- Use localparam to give names to the 7-bit constant patterns (fill in the values for x).
 - localparam [6:0] displayBlank = 7'bxxxxxxx;
 - localparam [6:0] displayOne = 7'bxxxxxxx;
 - localparam [6:0] displayMinus = 7'bxxxxxxx;
 - localparam [6:0] displayX = 7'bxxxxxxx;
- Provide meaningful names to the wires in the module.
- Properly tab-indent your code
 - Single level for wire declarations
 - Single level for component instantiations
 - Two levels for casez statement
 - Three levels for casez values

1.4 Bonus Ovf Logic

The default configuration of the system architecture ignores any overflow generated by the adder subtractor. If you choose, you may implement the logic necessary to determine if overflow occurs in the selected interpretation. In order to receive credit, your circuit needs to work under all combination of addSub and interp. Overflow for unsigned subtraction will require some careful analysis.

Your solution should have 2 LEDs, one for signed and one for unsigned. The unsigned overflow LED should illuminate when overflow will occur if the numbers are interpreted as unsigned numbers. The signed overflow LED is on when an overflow will occur if the numbers are interpreted as two's complement numbers.

For example, if the x and y inputs are 1001 and the operation is addition, then both signed and unsigned LEDs will illuminate.

1.5 Pin Assignment

Use the image of the development board in Figure 1.1 in and the information in the Cyclone V GX Kit User Manual (posted on the class web page) to determine the FPGA pins associated with the input and output devices used by the calculator module

Segment	msXdisplay	lsXdisplay	msYorRESdisplay	lsYorRESdisplay
seg[6]				
seg[5]				
seg[4]				
seg[3]				
seg[2]				
seg[1]				
seg[0]				

x	y
slide[3]	
slide[2]	
slide[1]	
slide[0]	

YorRES	Key[1]
interp	Key[2]
addSub	Key[3]

1.6 Turn in

You may work in teams of at most two. Make a record of your response to the items below and turn them in a single copy as your team's solution on Canvas using the instructions posted there. Include the names of both team members at the top of your solutions. Use complete

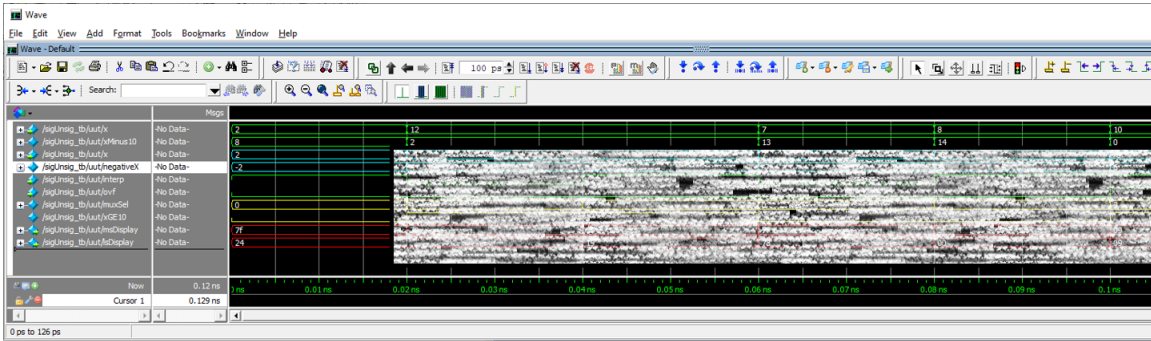
English sentences to introduce what each of the following listed items (below) is and how it was derived. In addition to this submission, you will be expected to demonstrate your circuit at the beginning of your lab section next week.

signUnsig Module

- Complete Table 1.
- Complete Table 2.
- Complete the code in Listing 2.
- Complete Figure 4, including:
 - Constant values on inputs of 4:1 mux
 - Constant value on the input of the right-most hexToSevenSeg
 - Value on the input of the adder subtractors
 - Values on the input of the comparator
- Complete Table 3.
- Complete Table 4.
- **Verilog code for the body of the sigUnsig module** (courier 8-point font single spaced), leave out header comments.
- Run the testbench for the sigUnsig module provided on Canvas. Produce a timing diagram with the following characteristics. Zoom to fill the available horizontal space with the waveform. Color inputs green and outputs red. Order the traces from top to bottom as
 - x radix unsigned Green trace
 - xMinus10 radix unsigned Green trace
 - x radix decimal Cyan trace
 - negativeX radix decimal Cyan trace
 - interp default Green trace
 - ovf default Green trace
 - digSel radix unsigned Yellow trace
 - xGE10 default Yellow trace
 - msDisplay radix hex Red trace
 - lsDisplay radix hex Red trace

I do not want the signals from the testbench, but rather the signals from inside the sigUnsig module. You can do this in sigUnsig, by expanding the sigUnsig_tb instance in the left Model-Sim pane and selecting “uut”. Since uut is an instance of the sigUnsig module, all the signals accessible in the sigUnsig module are shown in the center Object. You can add duplicates of signals by repeating the drag-and-drop operation.

Your completed timing diagram should look something like the following.



Pin Assignment:
Complete all three pin assignment tables.