Laboratory 6

Calculator With Friendly Output

6.1 Outcomes and Objectives

The outcome of this lab is to instantiate a calculator with signed decimal output making it easy for anyone to use the circuit. Through this process you will achieve the following learning objectives.

- Wire Logic
- Designing glue logic to interface building blocks
- Analyzing a circuit with a combination of building blocks
- Writing a Verilog statement using an Always/CaseZ statement
- Synthesizing a module on the FPGA development board

6.2 Calculator with Friendly Output

This week you are going to build a calculator that can add or subtract 4-bit values using the input and output shown in Figure 6.1 and display the results as decimal, base-10, values, not as hexadecimal 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 6.1. 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 four 7-segment displays on board, the same 7-segment displays of operand Y will be used to show the operation result (yellow) when the **yOrResult** button is pressed.

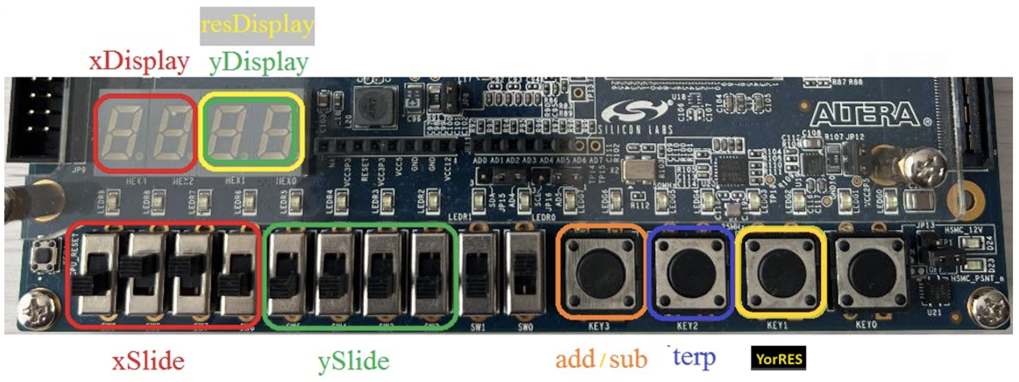


Figure 6.1: The input and output of the calculator digital circuit.

6.3 System Architecture

The system architecture shown in Figure 6.2 shows the adder subtractor processing the **xSlide** and **ySlide** inputs. The 4-bit x, y and result values are processed by the `sigUnsig` module before being displayed on the 7-segment displays. It is now time to turn our attention to this module.

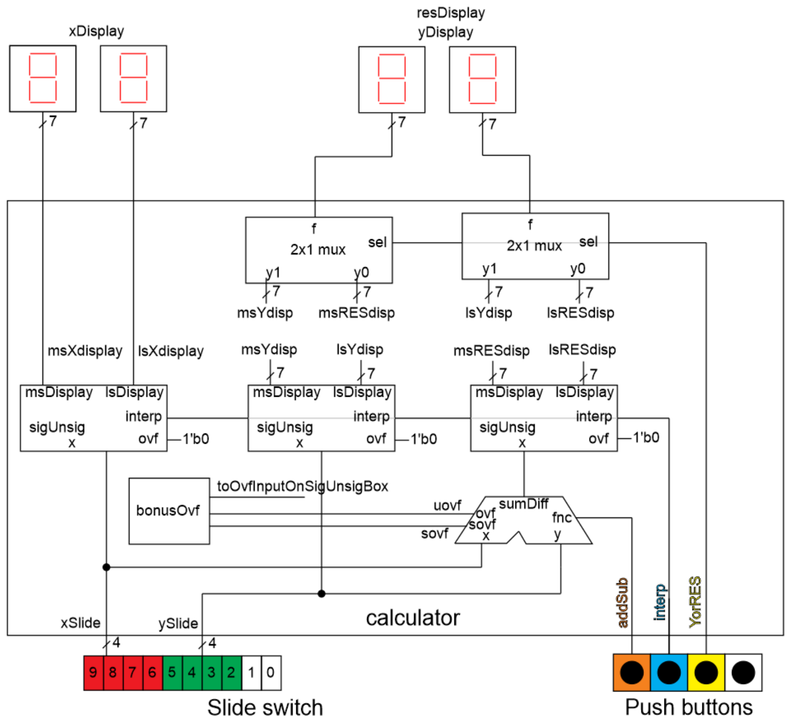


Figure 6.2: The system architecture of the calculator.

6.4 Module: sigUnsign

The significant design problem in today's lab comes in this section, building the `sigUnsign` module that shows up three times in Figure 6.2. This module takes in a 4-bit value and displays a 2-digit signed or unsigned representation on a pair of 7-segment displays. The `sigUnsign` module declaration is shown in Listing 6.1.

Listing 6.1: Module declaration for the sigUnsig module.

```
module sigUnsig(x, interp, ovf, msDisplay, lsDisplay);
    input   wire  [3:0]   x;
    input   wire          interp;
    input   wire          ovf;
    output  wire  [6:0]   msDisplay, lsDisplay;
```

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 **msDisplay** is the most significant (ms) symbol being displayed and **lsDisplay** is the least significant (ls) symbol being display. The term "symbol" is used because more than one type of information can be displayed depending on the values of the inputs. Let's explore this.

For example, let `x = 4'b1100`.

- If `interp = 1'b0` then `x` is interpreted as unsigned and its value is 12. Then the **msDisplay** should show "1" and **lsDisplay** "2".
- If `interp = 1'b1` then `x` is interpreted as 2's complement and its value is -4. Then the **msDisplay** should show "-" and **lsDisplay** "4".
- In the previous two cases we assumed, without stating it, that `ovf = 1'b0`. If `ovf = 1'b1` then the operation which generated `x` overflowed and the value of `x` is invalid. In this case both display's should show "X". Since we are working with 7-segments, our "X" looks much more like "H" :(

Not complete Table 6.1 by filling in the values of `msDisplay` and `lsDisplay` for a signed and unsigned interpretation, assuming *ovf*=0. If the interpreted value is positive and a single digit then assign *msDisplay* blank. If the interpreted value is negative then assign *msDisplay* "-". If the interpreted value is greater than 10, assign *msDisplay* "1".

Table 6.1: The output of the `sigUnsig` module when `ovf=0`.

4-bit input x	`interp = 0` Unsigned		`interp = 1` Signed	
	`msDisplay`	`lsDisplay`	`msDisplay`	`lsDisplay`
4'b0000	blank	0	blank	0
4'b0001				
4'b0010				
4'b0011				
4'b0100				
4'b0101				
4'b0110				
4'b0111				
4'b1000				
4'b1001				
4'b1010				

4-bit input x	`interp = 0` Unsigned		`interp = 1` Signed	
	`msDisplay`	`lsDisplay`	`msDisplay`	`lsDisplay`
4'b1011				
4'b1100	1	2	-	4
4'b1101				
4'b1110				
4'b1111				

Take a moment and look at the patterns in Table 6.1. You should make the following important observations.

- `msDisplay` is assigned one of four values
  - `blank` when the interpretation of `x` is an unsigned or signed value less than 10.
  - `1` when the interpretation of `x` is an unsigned value greater than 10.
  - `-` when the interpretation of `x` is a **signed value less than 0**.
  - `X` (the invalid character)when the `ovf = 1`.
- `lsDisplay` is assigned one of four values,
  - `x` (the value of the `x` input) when the interpretation of `x` is a unsigned or signed value less than 10.
  - `x-10` when the interpretation of `x` is an unsigned value greater than 10.
  - `0-x` when the interpretation of `x` is a **signed value less than 0**.
  - `X` (the invalid character) when the `ovf = 1`.

Why are we taking the 2's complement of `x`?

Please take a moment and reflect on pair of rows where you are asked to interpret `x` as a **signed value less than 0**. Under a signed (2's complement) interpretation, if the most significant bit of `x` is 1 then the value of `x` is less than 0. In this case the **msDisplay** 7-segment display should be illuminated with a "-" to indicate negative. The **lsDisplay** needs to show the negation of `x` because the negation of a negative number is a positive number and we can use a `hexToSevenSeg` module to display positive numbers. This is a complex but important observation.

If you follow the above reasoning, there is a need to form the 2's complement of `x` in certain input situations. You will form the negation of `x` by subtracting `x` from 0, that is compute `0-x`. You will do this by putting `4'b0000` on the `x` input of an `addSub`, put the sigUnsign input `x` on the `y` input of an `addSub`, and hardwire the `fnc` input to `1'b1` so that the `addSub` is hardwired to always subtract.

Formalized the observations in a more algorithmic syntax by completing Listing 6.2. Do this by filling the values of `msDisplay` and `lsDisplay` for the different input conditions. The values for these two signals are given in the two lists above. Note that this code is NOT to be used in your actual code for this lab.

Listing 6.2: Logic that determines the output of the 4:1 muxes in Figure 6.4.

```
if        ( (interp == 0) && (x < 10) ) {     // y0 input
    msDisplay =                  lsDisplay =

} else if ( (interp == 0) && (x >= 10) ) {    // y1 input
    msDisplay =          lsDisplay =

} else if ( (interp == 1) && (x >= 0) ) {     // y0 input
    msDisplay =          lsDisplay =

} else if ( (interp == 1) && (x < 0) ) {      // y2 input
    msDisplay =                  lsDisplay =

}
```

Now that you know what should be displayed on **msDisplay** and **lsDisplay**, let's look at how we an form these symbols on the 7-segment displays. In order to do this, you need Figure 6.3, the bit-order of the segments controlling the illumination o the segments. 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".

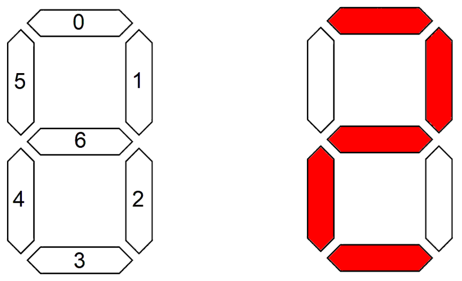
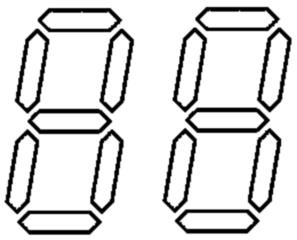
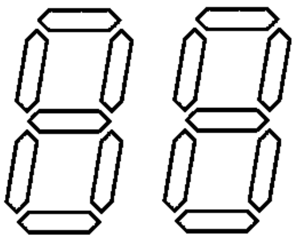


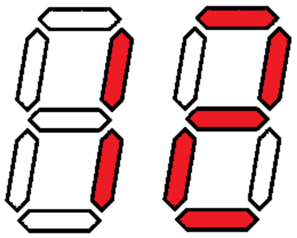
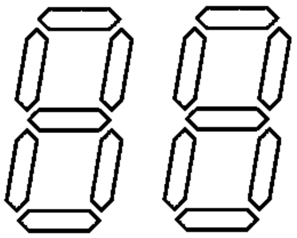
Figure 6.3: The logical arrangements of the segments in a 7-segment display.

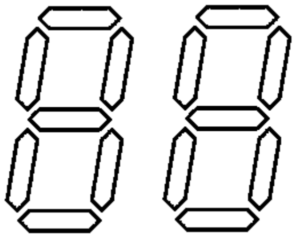
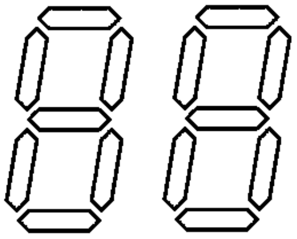
Test your understanding o the `signUnsign` output by completing Table 6.2. Do this by coloring in the segments of the 7-segment displays that are illuminated for each of the inputs. Then write the binary and hexadecimal value to illuminate those patterns to the right of `msDisplay=7'b` and `lsDisplay=7'b`.

Table 6.2: 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

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

Now we are ready to put the pieces of the sigUnsig module together. The building blocks in Figure 6.4 are captured in the organization described by Listing 6.2, along with some extra hardware.

Complete Figure 6.4 by adding the following:

- Connect the inputs of the 4:1 mux using the logic described in Listing 6.2.

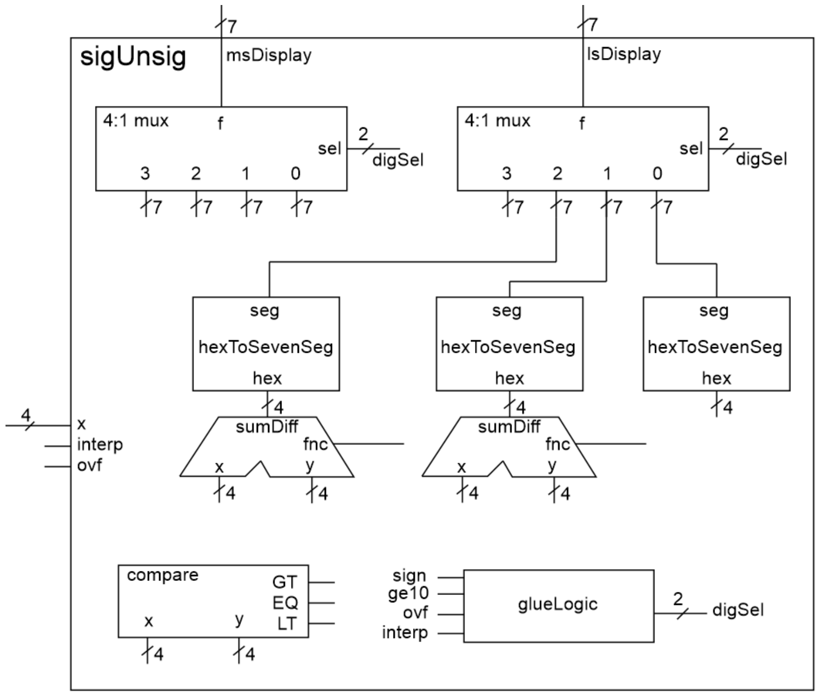


Figure 6.4: The internal architecture of the signUnsig module.

- Connect the inputs of the and adder subtractors using the logic described in Listing 6.2.
- Wire the inputs of the comparator to generate the signal `xGE10` which is logic 1 when `x` is greater than or equal to 10.
- Wire the input of the rightmost hexToSevenSeg .

Al that remains is to define the contents of the `glueLogic` box in Figure 6.4.

`glueLogic` always/casez statement

the `glueLogic` box chooses which input of the 4 mux inputs to route to the output. This is the logic that you formalized in Listing 6.2. Note the signal `sign` which equals 1 when `x` represents a negative value when interpreted as a signed value.

Now complete the truth table in Table 6.3 for the `glueLogic` box in Figure 6.4.

Table 6.3: 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 6.2. However, an always case statement requires each of the 16 difference cases to be explicitly enumerated. However, the truth table in Listing 6.2 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 6.2, 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 6.3.

Listing 6.3: The always/casez statement allows don't cares in the input.

```
always @(*)
   casez ({ovf, interp, xGE10, x[3]})
        4'b01?0: digSel = 2'b00;
        default: digSel = 2'b11;
   endcase
```

`sigUnsig` Verilog code

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 6.1.
- Use the module definitions for
  - `genericMux4x1` 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 localparm 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

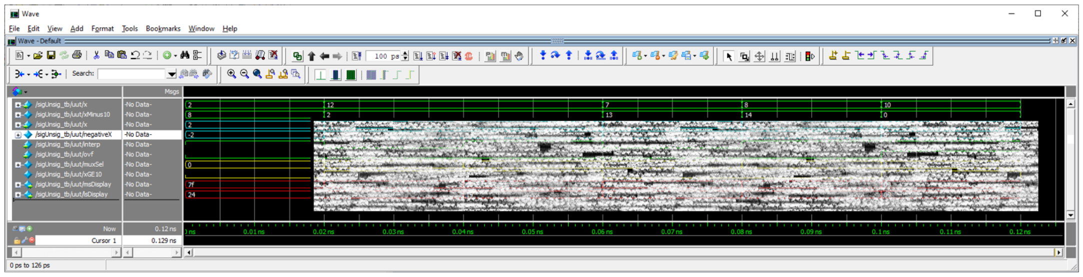
6.5 Testbench

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

signal	radix	Color for trace
x radix	unsigned	Green
xMinus10	unsigned	Green
x	decimal	Cyan
negativeX	decimal	Cyan
interp	default	Green
ovf	default	Green
digSel	unsigned	Yellow
xGE10	default	Yellow
msDisplay	hex	Red
lsDisplay	hex	Red

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 ModelSim 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.



6.6 Pin-Assignment and Synthesis

Use the image of the FPGA Development Board in Figure 6.1 and the information in the C5G User Guide to determine the FPGA pins associated with the input and output devices used by the devices used by the `calculator` module.

Table 6.4: Pin Assignment for the calculator.

Segment	msXdisplay	lsXdisplay	msYorRESdisplay	lsYorRESdisplay
seg[6]	AC22			
seg[5]		W21		
seg[4]			AE25	
seg[3]				W18
seg[2]				
seg[1]				
seg[0]				

	x	y
slide[3]	AE19	

	x	y
slide[2]		W11
slide[1]		
slide[0]		

YorRES	Key[1]	P12
interp	Key[2]	
addSub	Key[3]	

Complete the pin-assignment in Quartus, compile your design and download to the FGPA development boards. Once you get your design working, demonstrate it to a member of the lab team.

6.7 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 6.1.
- Complete Table 6.2.
- Complete the code in Listing 6.2.
- Complete Figure 6.4, including:
  - Constant values on inputs of 4:1 mux
  - Constant value on the input of the right-most hexToSeventSeg
  - Value on the input of the adder subtractors
  - Values on the input of the comparator
- Complete Table 6.2.
- Complete Table 6.3.
- Verilog code for the body of the sigUnsig module (courier 8-point font single spaced), leave out header comments.

Testbench

- Complete testbench and timing diagram from Section 6.5.

Pin-Assignment and Synthesis

- Complete the pin assignment in table 6.4.
- Demonstrate your circuit to a member of the lab team.

6.8 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.