

# CE-355

## Lab 2: Calculator

### Objective:

In this lab you will design a calculator that is capable of performing

1. addition
2. subtraction
3. multiplication

of two **signed** numbers. This calculator will be implemented on the Altera DE2-115 development kits.

### Lab Description:

#### I. Create new project:

1) Open a new blank project in Quartus II. The top level entity should be named something like “display\_calc” because this project will involve both a calculator entity and an LED display module as sub-modules.

2) Assemble the LED decoder design from Lab 1 into a package called “decoder”.

#### How to create a package:

A package is essentially VHDL code saved as a .vhd file containing the definitions of user defined data types, constants, functions, and component declarations of design modules, which you have designed earlier and would like to re-use in future designs. For example we will reuse the LED decoder that was designed in Lab1 as a component in Lab2 as part of the calculator.

The template for a package is as follows:

```
library IEEE;
use IEEE.std_logic_1164.all;
--Additional standard or custom libraries go here if needed

package decoder is

    COMPONENT leddcd is
        PORT ( .... ) -- enter the port declaration of your led decoder here
    end COMPONENT;

    -- For each module, which you want to add to this package, you will
    -- place their COMPONENT declarations here, in this case we just have one

end package decoder;

package body decoder is

    -- Subroutine declarations (if there are any such as functions and procedures)

end package body decoder;
```

Double check the lecture notes or the book for complete correct syntax for component declaration, which was discussed in structural vhd modeling.

You will also have a standalone .vhd, (e.g., *leddcd.vhd*) which describes the entity and architecture of the same led decoder. Remember that when a package includes a VHDL component declaration, you must include both the .vhd file for the package and the .vhd file for the original entity describing that component in your project.

3) Add the package file (and the original decoder entity “leddcd.vhd”) into the new project, by clicking on “Project” > “Add/Remove Files In Project” and then following the prompts.

4) Remember that the compilation tools in Quartus II will always return an error if there is no entity description for the top level entity (in this case “display\_calc”) that you specified in the setup wizard. Although you will not complete this entity until you are done with all the lower level entities, you must include the basic information about your top-level design in order to continue. In Quartus II, select “File” > “New” > “VHDL file”.

5) Copy this entity description into the new VHDL file:

```
-----  
library IEEE;  
  
use IEEE.std_logic_1164.all;  
--Additional standard or custom libraries go here  
  
entity display_calc is  
    port(  
        --You will replace these with your actual inputs and outputs  
        inputs          : in std_logic;  
        outputs         : out std_logic  
    );  
end entity display_calc;  
  
architecture structural of display_calc is  
  
    --Signals and components go here  
  
begin  
  
    --Structural design goes here  
  
end architecture structural;  
-----
```

6) In the File menu, click “Save as”. Save the file as “display\_calc”. The file type should be “VHDL File (\*.vhd,\*.vhdI)” and the box next to “Add file to current project” should be checked.

7) In Quartus II, double click “Analysis & Synthesis” under the “Compile Design” heading in the Tasks window. You may get some warnings, but there should be no errors.

At the end of this phase, you should have a new project in Quartus II. So far, this project should have three files in it: the package containing the LED decoder, the entity for the LED decoder, and the new top-level entity file for the calculator. You can verify which files are in a project by clicking on the “Files” tab at the bottom of the Project Navigator pane on the top left of the Quartus II environment window.

## II. Create calculator entity and a package for constants:

1) Open a new VHDL file and copy this package description into it:

```
-----  
library IEEE;  
  
use IEEE.std_logic_1164.all;  
--Additional standard or custom libraries go here  
  
package calc_const is  
  
    constant DIN1_WIDTH : natural := 8;  
    constant DIN2_WIDTH : natural := 4;  
    constant OP_WIDTH   : natural := 2;  
    constant DOUT_WIDTH : natural := 12;  
  
    --Other constants, types, subroutines, components go here  
  
end package calc_const;  
  
package body calc_const is  
  
    --Subroutine declarations go here  
    -- you will not have any need for it now, this package is only for defining -  
    -- some useful constants  
  
end package body calc_const;  
-----
```

2) Save as “calc\_const” and make sure that the file is added to the project.

3) Open another new VHDL file and copy this entity description below into it:

```
-----  
library IEEE;  
  
use IEEE.std_logic_1164.all;  
--Additional standard or custom libraries go here  
  
entity calculator is  
    port(  
        --Inputs  
        DIN1      : in std_logic_vector (DIN1_WIDTH - 1 downto 0);  
        DIN2      : in std_logic_vector (DIN2_WIDTH - 1 downto 0);  
    );  
end entity;
```

```

        operation      : in std_logic_vector (OP_WIDTH - 1 downto 0);

        --Outputs
        DOUT            : out std_logic_vector (DOUT_WIDTH - 1 downto 0);
        sign            : out std_logic
    );
end entity calculator;

architecture behavioral of calculator is

    --Signals and components go here

begin

    --Behavioral design goes here

end architecture behavioral;
-----

```

4) Save as “calculator” and make sure that the file is added to the project.

This describes an entity that takes three inputs (DIN1, DIN2, operation). The calculator performs the arithmetic operation specified on the two inputs and saves the result to DOUT. The operations are always performed with DIN1 on the left and DIN2 on the right. For example,  $DOUT = DIN1 * DIN2$ .

***Additionally, there is a status flag that must be set in your design.*** Because you are working with signed numbers, sign is ‘1’ when DOUT is a negative number and is ‘0’ when DOUT is 0 or a positive number.

On the DE2-115 board, slide switches will be used for DIN1, DIN2, and the operation, seven-segment displays will be used for DOUT, and the sign.

Finally, it is acceptable for your output display (DOUT) to be in hexadecimal or decimal. It is much easier, however, to use hexadecimal because the values stored in the registers will be interpreted as binary rather than decimal by default.

### III. Design calculator entity

1) In the previous phases you have created two custom packages that must be included in the calculator entity. One was for the led decoder and the other for useful constants to be used in your calculator design. Now, you will include those in your calculator entity. The syntax for including a custom package is:

```
use WORK.custom_package.all;
```

or in our case:

```
use WORK.decoder.all;
use WORK.calc_const.all;
```

This is placed in the section right after the line “use IEEE.std\_logic\_1164.all” at the top of the VHDL file. Remember that packages must be added to the project directory before they can be included in any other entity descriptions. Verify that “decoder.vhd” and “calc\_const.vhd” are present in the files tab in the project navigator.

The arithmetic operations necessary in the design of the calculator **are not defined for the std\_logic or std\_logic\_vector data types**. Instead, **you will need to use the signed/unsigned and integer data types** provided in the standard package “numeric\_std”. Be sure to declare this package in the entity by including the line “use IEEE.numeric\_std.all;”.

I have provided more information and links about important packages from the ieee and std libraries on Canvas under this **Assignment**.

The following functions are relevant for conversion between std\_logic type and integer type. You can use these functions in your vhdl code:

```
-- Conversion from std_logic_vector to integer:  
to_integer(signed(ARG:std_logic_vector))  
-- Conversion from integer to unsigned std_logic_vector of size N:  
std_logic_vector(to_signed(ARG:integer, ARG:N))  
-- Conversion from std_logic_vector to an unsigned bit_vector of same size:  
unsigned(ARG: std_logic_vector)  
-- Conversion from integer to an unsigned bit_vector of size N:  
to_unsigned(ARG: integer, ARG:N)
```

As you will notice in the example above, the conversion function between std\_logic and integer first uses the conversion function **signed(ARG:std\_logic\_vector)** to go from std\_logic\_vector to a signed bit vector. On the other hand, the conversion function from integer to std\_logic\_vector uses the **to\_signed(ARG:integer, ARG:N)** function to go from an integer to a signed vector of bits first. Different options can be combined and you can experiment with them to observe the outcome.

The **signed** data type uses Two’s Complement to represent negative numbers. This will result in DOUT being displayed incorrectly if it is negative. ***To correct this, you should first raise the sign flag if DOUT is negative, and then calculate the absolute value of DOUT before continuing. The absolute value can be calculated using the appropriate function defined in numeric\_std:***

To compute the absolute value of an integer:  
**abs(integer)**

To compute the absolute value of a signed number in the form of a bit vector of size N:  
**abs(ARG:vector, ARG:N)**

2) Complete the calculator entity to function as outlined in the project description.

In Quartus II, double click “Analysis & Synthesis” to check your syntax. If Analysis & Synthesis runs without error when you are done with your design, then you are ready to move onto the next phase, which is testing.

**At the end of this phase, you should have a project in Quartus II with five files in it: the package for the LED decoder (decoder), the entity for the LED decoder (leddcd), the top-level entity file (display\_calc), the calculator constants package (calc\_const), and the calculator entity file (calculator).**

#### **IV. Simulation with ModelSim Software**

1) Open ModelSim. Create a new project by clicking “File” > “New” > “Project”. You will use this project to test both the calculator by itself, and the “display\_calc” entity with the LED decoders, so name the project something similar to your top-level entity. Create a new folder named “simulation” inside your Quartus II project directory and save your ModelSim project into it. Click OK to continue.

2) Click “Add Existing File” to add the VHDL files from the previous phases. Click browse and navigate to your Quartus II project directory. Shift-click to select all of your VHDL files. At this point, you have a choice about how to add the files to the ModelSim project.

“Reference from current location” will link your new project to the existing VHDL files. That means that if you change them in ModelSim the original files will be changed. “Copy to project directory” will not affect the original files if you make any changes in ModelSim. In general, it is easier to choose the reference option.

3) Close the “Add items to project” window. ModelSim should display a window showing all of the files in your project, and they will all have a question mark icon next to them under “Status”. This is because they have not been compiled into the project library. To compile all of the files in the project, click “Compile” > “Compile All”.

Something important to notice is that the compile process may fail even though it compiled in Quartus II. Usually, this has to do with the order in which files are compiled. Because “calculator” references the “calc\_const” package, ModelSim will give an error if it tries to compile calculator first. To ensure that your included packages are always compiled first, click on “Compile” > “Compile Order”. Move “calc\_const” to the top of the order. Remember to do this with all custom packages you create. Click compile again and this time the compile should succeed.

**At the end of this phase, you should have a project in Quartus II with five files in it: the package for the LED decoder (decoder), the entity for the LED decoder (leddcd), the top-level entity file (display\_calc), the calculator constants package file (calc\_const), and the calculator entity file (calculator). You should also have a ModelSim project that compiles successfully with all of your VHDL files.**

4) Write a testbench, (call it “calculator\_tb.vhd” for example) for the calculator that can handle the test-case files. The data for the test-cases are in text files named “cal8.in” and “cal16.in”. These files must be downloaded to your work directory (the directory with the ModelSim project file). Start this testing process with the 8-bit versions of the input and output files. Later, switch to the 16-bit versions and repeat the test.

The textio and std\_logic\_textio packages contain the functions necessary to handle input and output files as well as text processing. When including these files, note that the library declaration:

**library STD;**

must be included in addition to **library IEEE;**

There is also an example vhdl file in the Assignment folder about reading and writing to and from files.

The format of the test-case data files are

```
Data1
Data2
Operation
Data1
Data2
Operation
```

Send the test data into your component and output the data (either to standard output or to a file named something like “cal8.out” or “cal16.out”). The output should be formatted like this (copy the output to your report):

```
12 + 4 = 16
12 * -4 = -48
```

In addition to the numerical value of the answer, make sure that the status flags (sign, overflow) are being set and cleared properly.

2) Add your testbench VHDL file to your ModelSim project by clicking “Project” > “Add to project” > “Existing file”. Compile your project again to verify that the testbench compiles.

3) Click “Start simulation” from the “Simulate” menu. Expand the “work” library by clicking on the “+” button next to it. Select the testbench file from the library and click OK.

4) In the simulation window that appears, right click on the top-level item with the name of your testbench. Click on “Add items to wave” > “All items in region and below”. Run the simulation by clicking the “Run” button in the wave diagram window.

**Include the wave and the output file (cal8.out) in your report. Make sure that the all relevant names, values, and transitions are easy to read in the report. Include multiple pictures if all of this information will not fit in one.**

**In order to extract the waveforms of your simulations in Modelsim: choose File->Export->Image, it will save the waveform as a BMP file. Then, you can insert these images into a Word document.**

5) Continue editing your calculator entity until it can correctly perform all of the required operations and can handle all of the 8-bit test cases. When you are finished, modify the calc\_const package to handle 16-bit inputs.

```
-----  
library IEEE;  
  
use IEEE.std_logic_1164.all;  
--Additional standard or custom libraries go here  
  
package calc_const is  
  
    constant DIN1_WIDTH : natural := 16;  
    constant DIN2_WIDTH : natural := 8;  
    constant OP_WIDTH   : natural := 2;  
    constant DOUT_WIDTH : natural := 24  
  
    --Other constants, types, subroutines, components go here  
  
end package calc_const;  
  
package body calc_const is  
  
    --Subroutine declarations go here  
  
end package body calc_const;  
-----
```

Recompile the project, and repeat the simulation using the same testbench (remembering to change the file names to match the 16-bit versions of the inputs and outputs). This illustrates the advantage of using separate files to handle input and output and constants to represent data widths, as they allows you to use the same testbench and design files by only changing a few constants.

**Include the wave picture(s) along with the cal16.out file in your report.**

**Change back to the 8-bit version of the constants file before continuing with the implementation on the DE2-115 Board!!!**



## VI. Top-level design

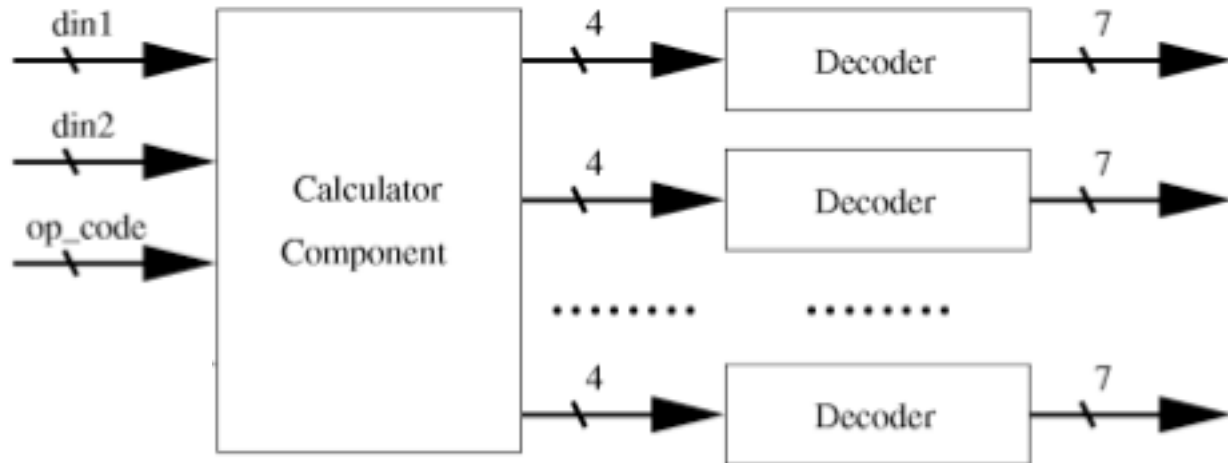


Figure 6.1: Block diagram for top-level entity

- 1) Return to the TOP LEVEL “display\_calc” entity that you created at the beginning. Include the LED decoder and the calculator entities as components.
- 2) Complete your TOP LEVEL entity so that all of the appropriate inputs are passed to the calculator entity and so that the outputs from the calculator entity are passed to the LED decoders to be displayed. Use one LED to display the sign of the result (the LED should display “-“ if the result is negative and be blank if positive). Use as many additional LEDs to display the result.

Plan connecting these components carefully.


### **DO NOT HARDWIRE A FIXED NUMBER OF LED DECODERS!**

**In order to determine how many decoders you will need to define the number of LED decoders as a function of DOUT WIDTH and then instantiate the proper number of components using the “GENERATE” statement. If I were to go in and change the values of the D IN widths in the calc\_const package the design should automatically synthesize correctly, without needing to change any other part of your code!**

- 3) To begin the compilation process, click “Analysis & Synthesis” in the tasks window. Once the synthesis process completes successfully, you are ready to set your pin assignments. This is an important process that you will need to be able to perform in all future assignments. **Only begin assigning pins to your ports after you have successfully synthesized your design.**

Continue the pin assignment until all input and output ports are assigned to pins.

4) To fully compile your project, double click on “Compile Design” in the tasks window. This will run several processes in a row including “Analysis & Synthesis”, “Fitter (Place & Route)”, “Assembler (Generate programming files)”, and “Classic Timing Analysis”. Once this process is complete, you are ready to program the FPGA.

5) Once your design has been compiled, it can be loaded onto the FPGA using the programming utility in Quartus II. To open the programmer, click on the  button in the toolbar, or double click “Program Device (Open Programmer)” in the “Tasks” pane. Make sure that the “USB-Blaster” is selected next to the “Hardware Setup” button in the top left of the programming window. Click on “Start” to program the device.