Lab 2: Arithmetic Logic Unit

EE478 Fall 2019

Objective

In this lab, you will implement a complex combinational digital system for an Arithmetic Logic Unit using Behavioral VHDL. You will first design the individual modules of the system on paper based on digital design principles. Then you will develop a block diagram of the system. Each module will be implemented in VHDL, then using components the top level design will be completed. You will create a testbench to test several input patterns in the design. Then you will implement the design on the Zybo board and again use the slider switches and LEDs for IO.

Deliverables

Lab start: Beginning of lab session during the week of 9/16

Demo: Working ALU in a testbench demonstrating all five functions output correctly on at least two different input patterns each. Working design on Zybo board, allowing input of operands using slider switches, selection of function using momentary pushbuttons, and output displayed on LEDs.

Demo due: End of lab session during the week of 9/23

Design Document due:

- On UBLearns by 11:59PM on 9/30. *Submit one per team*.

Lab Contents

ALU Review

An Arithmetic Logic Unit (ALU) is a digital system that accepts binary numerical inputs and computes one out of several possible outputs based on an arithmetic or logical function of the inputs. ALUs generally have two wide data input operands (32 or 64 bits each in modern processors, for example) and one or two wide data outputs (multiplication instructions double the number of bits of the inputs.) ALUs generally operate on integers; special floating point units are used for calculations involving floating point numbers.

The ALU also accepts an opcode, which is a binary value indicating which of the supported functions should be computed. In reality, ALUs typically compute several of the possible functions at once, and the opcode is used to select from the different results using a multiplexer.

Part A: ALU system requirements

In this lab, you will develop an ALU system to be implemented on the Zybo board. This lab description will not cover Vivado usage; see Lab 1 for assistance with project setup, etc. The following requirements must be satisfied by your demo or design document.

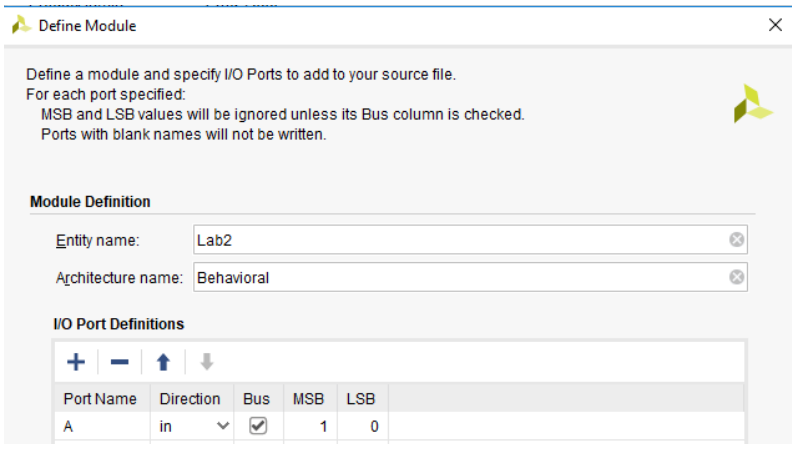
1. The ALU shall accept two 2-bit input operands and produce one 4-bit output.
    a. The first 2-bit input shall be connected to the first two slider switches on the Zybo board, and the second input shall be connected to the last two switches.
    b. The 4-bit output shall be displayed using the four LEDs on the board.
2. The ALU shall support five functions
    a. The first four functions shall be selectable by pressing one of four momentary push buttons on the board, labeled BTN0, BTN1, BTN2, and BTN3. The fifth function will be selected when no buttons are pressed i.e. when S = “0000”.
    b. The first function shall be $Y = A + B$ where A and B represent the two input operands and are **signed** binary numbers.
    c. The second function shall be $Y = A \text{ ASR } B$, namely, the arithmetic right shift of A by B positions.
    d. The third function shall be $Y = A * B$, the 4-bit product of A and B, in which both inputs are considered as **unsigned** binary numbers.
    e. The fourth function shall be $Y = A \text{ XOR } B$, the bitwise exclusive OR of A and B.
    f. The fifth function shall be $A > B$, namely, the LSb of Y should be ‘1’ if $A > B$ and ‘0’ otherwise.
3. You may assume that only one of the select signals will be ‘1’ at any given time.

To get started, create a new project and add a source file with two 2-bit vector inputs A and B, one 5-bit select input S, and one 4-bit output Y.

Appendix A: Signed/Unsigned types and Vector ports

Remember that the only types allowed for input and outputs at the top level (signals connected to actual pins) are *std_logic* and *std_logic_vector*. Therefore, you will need to perform type conversions as needed to get your signals into the *signed* or *unsigned* data types.

When creating a new source file for your module, Vivado can automatically set up the std_logic_vector ports. Check the ‘bus’ checkbox next to the signal name, and enter the signal numbers for the MSb and LSb. For example, a 2-bit std_logic_vector named A would be entered as shown below:


Define Module

Define a module and specify I/O Ports to add to your source file.
For each port specified:
MSB and LSB values will be ignored unless its Bus column is checked.
Ports with blank names will not be written.

Module Definition

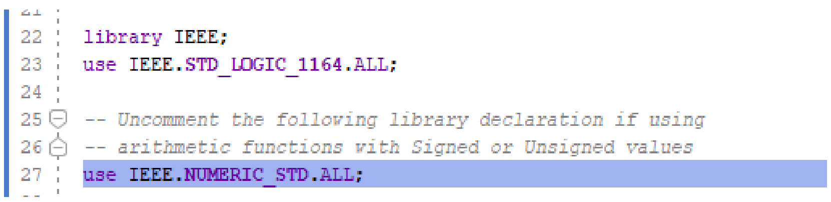
Entity name: Lab2

Architecture name: Behavioral

I/O Port Definitions

Port Name	Direction	Bus	MSB	LSB
A	in	✓	1	0

In order to make use of the *signed* and *unsigned* types, you should uncomment line 27 in the auto-generated VHDL template in Vivado:


```
22  library IEEE;
23  use IEEE.STD_LOGIC_1164.ALL;
24
25  -- Uncomment the following library declaration if using
26  -- arithmetic functions with Signed or Unsigned values
27  use IEEE.NUMERIC_STD.ALL;
```

In order to use arithmetic operators like ‘+’ or ‘*’, you need to convert the signal to *signed* or *unsigned*. This can be done easily using a *type cast*:

```
signal A_signed : signed(1 downto 0);

begin

A_signed <= signed(A);
```

Be sure that you also type cast back to *std_logic_vector* for the outputs.

You may need to combine signals together using the concatenation operator ‘&’. For instance, the sum of two 2-bit signed signals is another 2-bit signed signal. Because we have 4 LED outputs, we will need to pad with zeros, which can be done as follows:

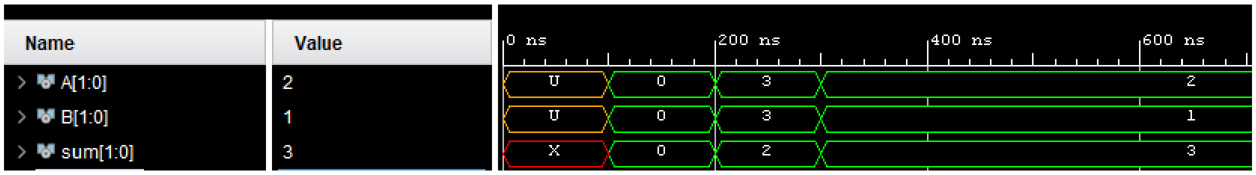
```
 Y <= "00" & std_logic_vector(sum) when S(0) = '1' else
```

Appendix B: Simulating with Vector Types

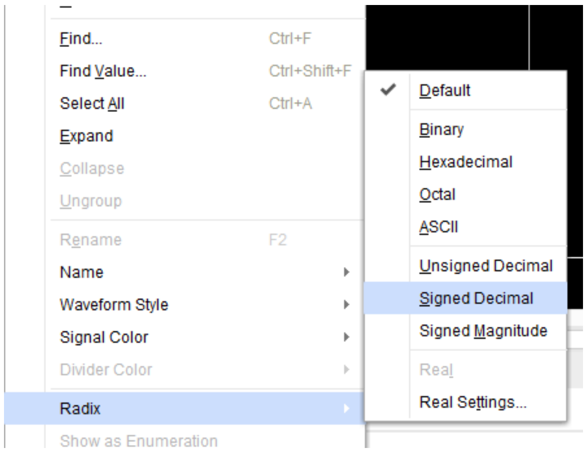
In your testbench, you can drive inputs to signals of vector types by simply using double quotes. For example, here is a test sequence for a 2-bit signed adder:

```
sim_proc : process
begin
    wait for 100 ns;
    A <= "00";
    B <= "00";
    wait for 100 ns;
    A <= "11";
    B <= "11";
    wait for 100 ns;
    A <= "10";
    B <= "01";
    wait;
end process sim_proc;
```

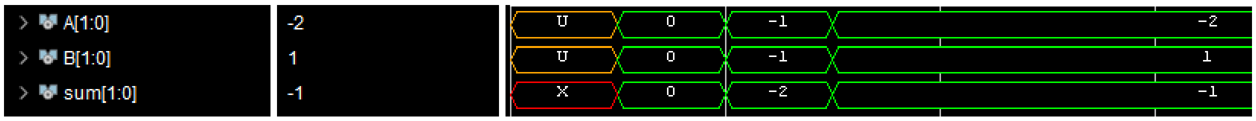
Once you are in the actual simulation waveform, note that signals that have vector types are displayed in unsigned decimal by default:



Very often, this won't be what you want. To view all values in a different radix, select the traces you want to change, right click, go to radix, and select either Signed Decimal, Unsigned Decimal, or Hexadecimal.



This will cause the waveform to display numbers in the selected radix. As you can see from this example, it is much easier to verify that the signed adder is working correctly.

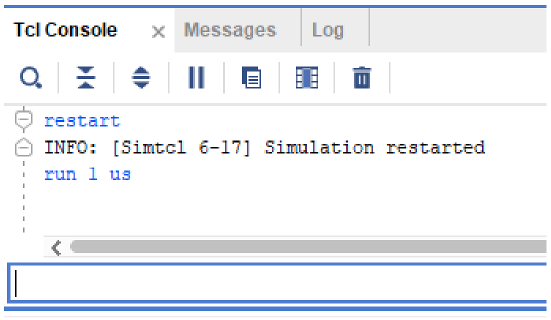


Appendix C: Simulator Commands

XSim supports many different commands to control the simulation process. By default, XSim runs for 1us, but we may want to simulate our designs for longer.

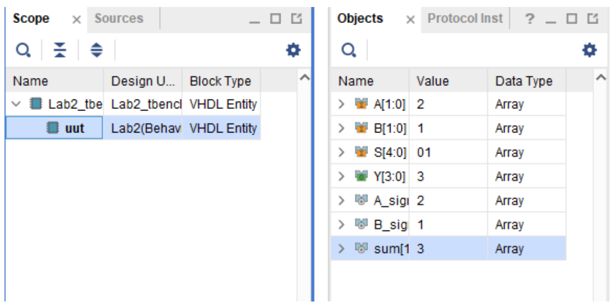
At the bottom of the XSim GUI, there is a console into which we can type commands. The console says “Type a Tcl command here”. Tcl stands for “Tool Command Language”, which is a commonly used general-purpose scripting language for simulation used across the industry. We will study more commands later; for now, here are two useful commands:

- *reset*: causes the simulator to restart with an empty waveform
- *run <time> <units>* run the simulation for a certain amount of time.
    - For example *run 1 us* or *run 100 ns*



```
restart
INFO: [Simtcl 6-17] Simulation restarted
run 1 us
```

These commands can also be useful if you want to view any internal signals in your modules during simulations (by default, only the inputs and outputs of the module are plotted.) To add an internal signal to the waveform display, highlight the name of the uut (probably just “uut”) under the testbench top level and drag the signal from the Objects panel to the waveform. Then just restart and run the sim again using the commands above.



Name	Design U...	Block Type
Lab2_tbe	Lab2_tbencl	VHDL Entity
uut	Lab2(Behav	VHDL Entity

Name	Value	Data Type
A[1:0]	2	Array
B[1:0]	1	Array
S[4:0]	01	Array
Y[3:0]	3	Array
A_sigı	2	Array
B_sig	1	Array
sum[1	3	Array