VERILOG COMINATIONAL LOGIC INTRO

This lab has you implement an adder that is capable of normal addition, unsigned saturating addition, and signed saturating addition. It also introduces the Xilinx Coregen tool used for generating IP cores. This lab does not require use of the Basys 2 board.

BACKGROUND

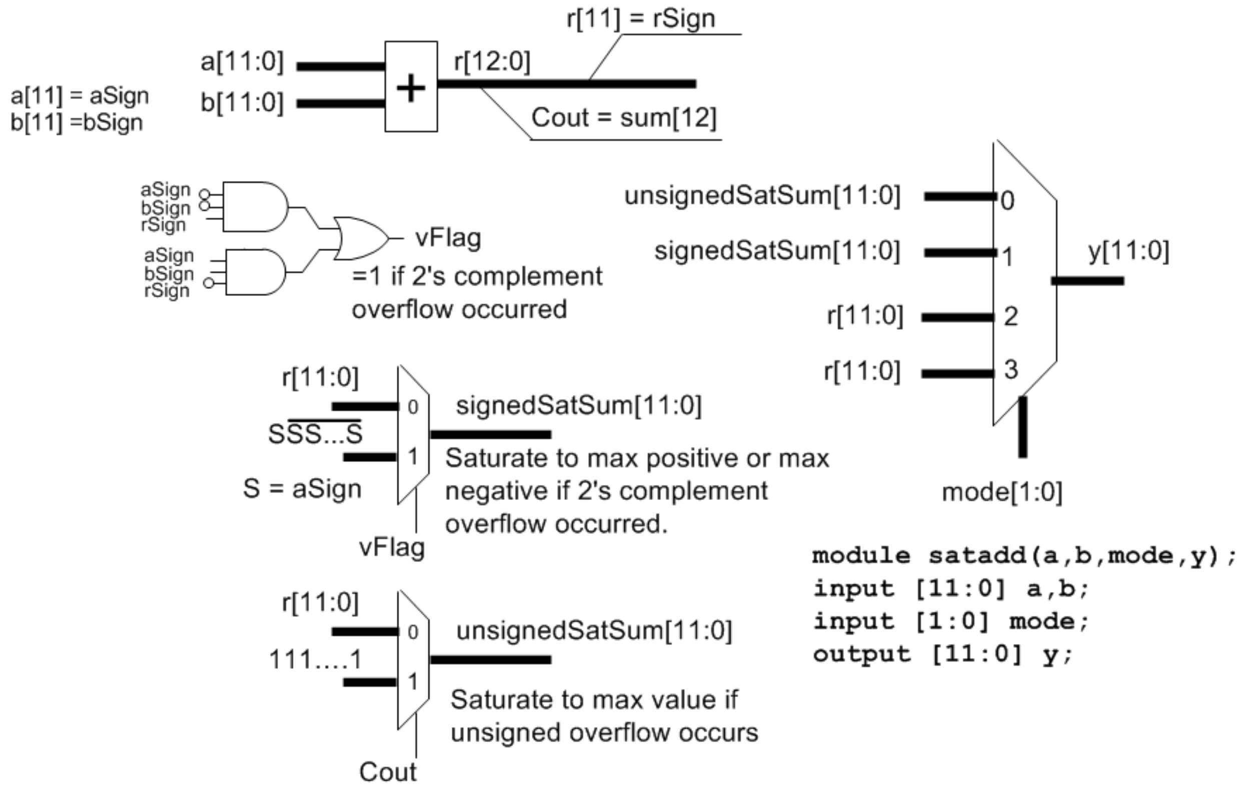
Review the Power Point notes on the Class website titled "fixed_point.ppx". This reviews the definition of signed integers using 2's complement encoding, and also the definitions of unsigned saturating addition and signed saturating addition. You need to understand this material in order to complete the lab successfully.

TASK 1

You are to build an adder module that implements the following:

Mode Input	Output function
`mode = 2'b00`	`y = a + b   (unsigned saturation)`
`mode = 2'b01`	`y = a + b   (signed saturation)`
`mode = 2'b10`	`y = a + b   (normal addition)`
`mode = 2'b11`	`y = a + b   (normal addition)`

The schematic below shows the design and that you are to implement in one Verilog module. It also defines the Verilog port declarations and module name. In your Verilog module, you can use whatever internal signal names make the most sense to you – only the external port names must kept as specified.



```
module satadd(a,b,mode,y);
input [11:0] a,b;
input [1:0] mode;
output [11:0] y;
```

Implementation Hints

1. Just about all of this logic can be implemented using *assign* statements. You will need to use temporary signals (*wire* type for *assign* statements, *reg* type if driven by an *always* block).
2. The 4-to-1 mux that produces the final *y* output is probably best implemented by an *always* block.
3. Note the inputs of the adder are 12-bits wide, while the output is 13-bits wide. Bit r[11] of the adder output is the sign of the output value, while bit r[12] is the carry out (Cout). If the carry out is '1', then unsigned overflow occurred.
4. The vFlag internal signal is '1' if 2's complement overflow occurs. This is true if the sum of two positive numbers produces a negative number or if the sum of two negative numbers produces a positive number. The signs of the two inputs (a[11], b[11]) represented by the signals *aSign*, *bSign* and the sign of the result (r[11]) by *rSign*.

Procedure

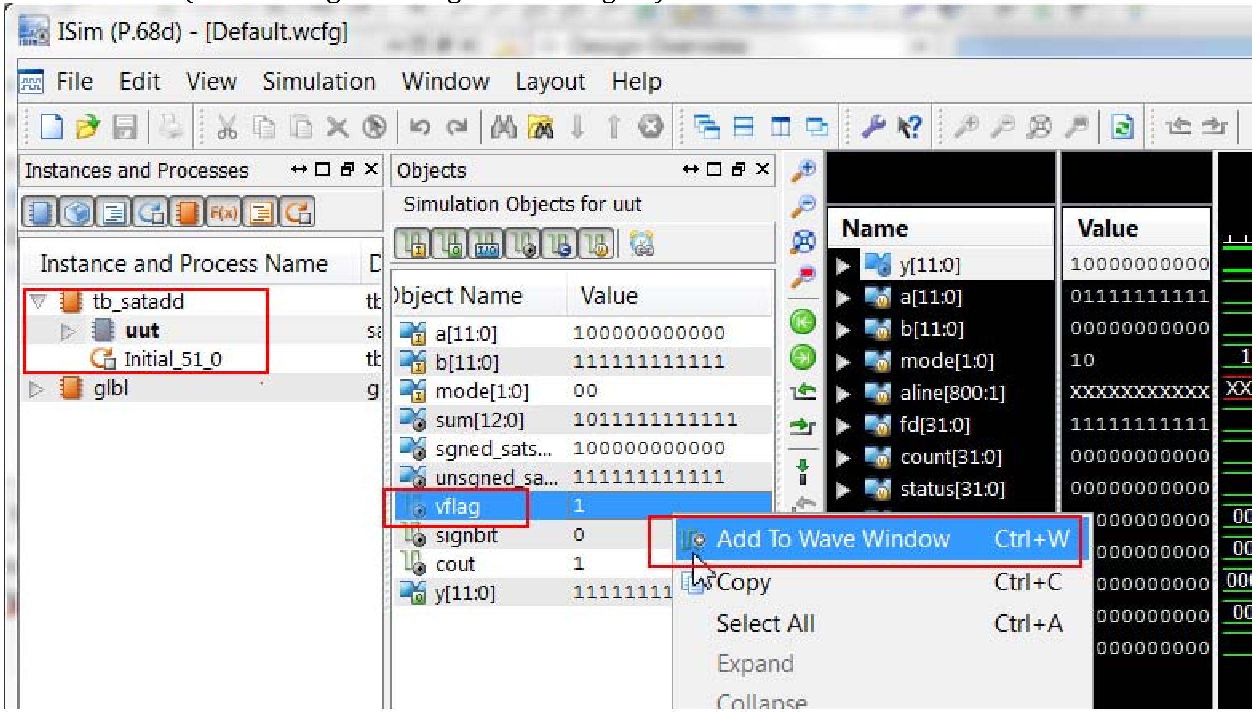
1. Download the zip archive associated with the lab.
2. This contains four files: *satadd.v*, *tb_satadd.v*, *satadd_vectors.txt* and *report.doc* (report file). The *satadd.v* file contains the empty Verilog module that you are to complete. The *tb_satadd.v* contains the testbench, and it reads the test vectors from the *satadd_vectors.txt* file.
3. Create a project named *lab2_part1* and add the *satadd.v* file to it as the top module. Then, generate a test fixture for this file named *tb_satadd.v*, and copy the contents of the original *tb_satadd.v* into it.

4. Copy the *satadd_vectors.txt* file into the project directory. This is needed before you simulate your design.
5. Implement your Verilog code in the *satadd.v* file and test it with the *tb_satadd.v* . You are finished when all of the vectors pass as indicated by messages printed to the simulation console pane.
6. Once the simulation is working, use the 'Implement Top Module' command to map this design to the FPGA. You do not have to download it into your board.
7. Record the number of 4-input LUTs that it took to implement this design (see the 'Design Summary' page).

Debugging Hints

So, your design is not passing all of the vectors, what do you?

1. Since this is hardware, you cannot 'single step through code' as you would in a computer program.
2. However, what *you can do*, is examine all of the signals in your design at any given time, which is just as good. This gives you COMPLETE OBSERVABILITY and with this, you can debug any problem.
3. First, identify the vector that is failing. Is it producing the wrong answer for signed saturation? Then it might be your mux logic for the signed saturation value. Is it producing the wrong answer for unsigned saturation? Then it might by the mux logic for he unsigned saturation value.
4. To determine what is wrong, you will have look at the internal signals in your design, and not just the *a*, *b*, *y*, *mode* externals signals.
5. In the simulator, you can add internal signals to the waveform window by expanding the instance hierarchy in the 'Instance and Process Name' pane, and selecting the 'uut' instance (this is your module). This causes all of your internal signals to appear in the 'Objects' pane. You can then add any internal signals that you wish to the list of displayed signals as shown below (I am adding the 'vflag' internal signal).


ISim (P.68d) - [Default.wcfg]
File Edit View Simulation Window Layout Help
Instances and Processes
Instance and Process Name
tb_satadd
uut
Initial_51_0
glbl
Objects
Simulation Objects for uut
Add To Wave Window Ctrl+W
Copy Ctrl+C
Select All Ctrl+A
Expand
Collapse

Object Name	Value
a[11:0]	100000000000
b[11:0]	111111111111
mode[1:0]	00
sum[12:0]	1011111111111
sgned_sats...	100000000000
unsgned_sa...	111111111111
vflag	1
signbit	0
cout	1
y[11:0]	1111111

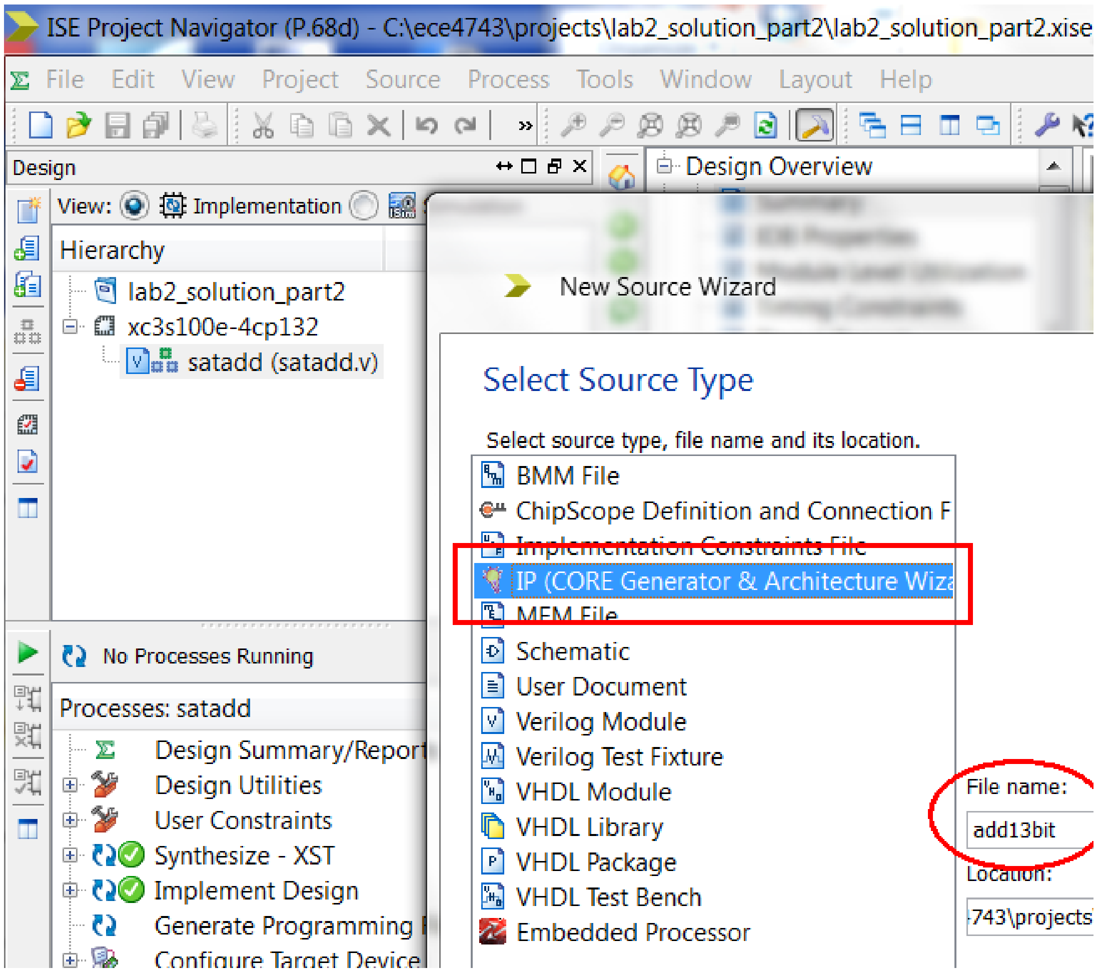
Name	Value
y[11:0]	10000000000
a[11:0]	01111111111
b[11:0]	00000000000
mode[1:0]	10
aline[800:1]	XXXXXXXXXXX
fd[31:0]	11111111111
count[31:0]	00000000000
status[31:0]	00000000000

6. After adding internal signals, use the 'Re-launch' button in the right hand corner of the simulator to re-run the simulation. You will be asked to save the new signal configuration list to a file, just call it 'Default'.
7. In the new simulation, zoom in to the test vector that is failing, and see if the internal signals match what you expect them to be. You need to be able to predict what the internal signals should be, so that you determine the problem. This means that you have to understand how saturating addition (signed and unsigned) actually works ☺.
8. Following this procedure, you should be able to find where the actual internal signal values do not match your expected values, and this should indicate the problem source.

TASK 2

This task introduces you to the Xilinx Coregen tool. Task 1 must be working before you can do Task 2. The Coregen tool creates optimized versions of many common building blocks used in Digital Systems Design. These IP cores range from the relatively simple (adders) to very complex (square root, digital filters, etc).

1. Create a project named *lab2_part2* and copy your *satadd.v* , *tb_satadd.v*, and *satadd_vectors.txt* files to it. Add the *satadd.v* file as the top module. Add the *tb_satadd.v* functionality as you did in the previous task. Verify that you can successfully simulate this design in the new project.
2. You are to replace the 12-bit adder logic that you have written (probably by an assign statement) with an adder module generated with the Xilinx Coregen tool.
3. In the Implementation pane, select 'satadd'. Then use Project | New Source – in the New Source Wizard, select the 'IP (Core Generator & Architecture Wizard)' and type in 'add13bit' as the new file name. The click 'Next'.



Select Source Type

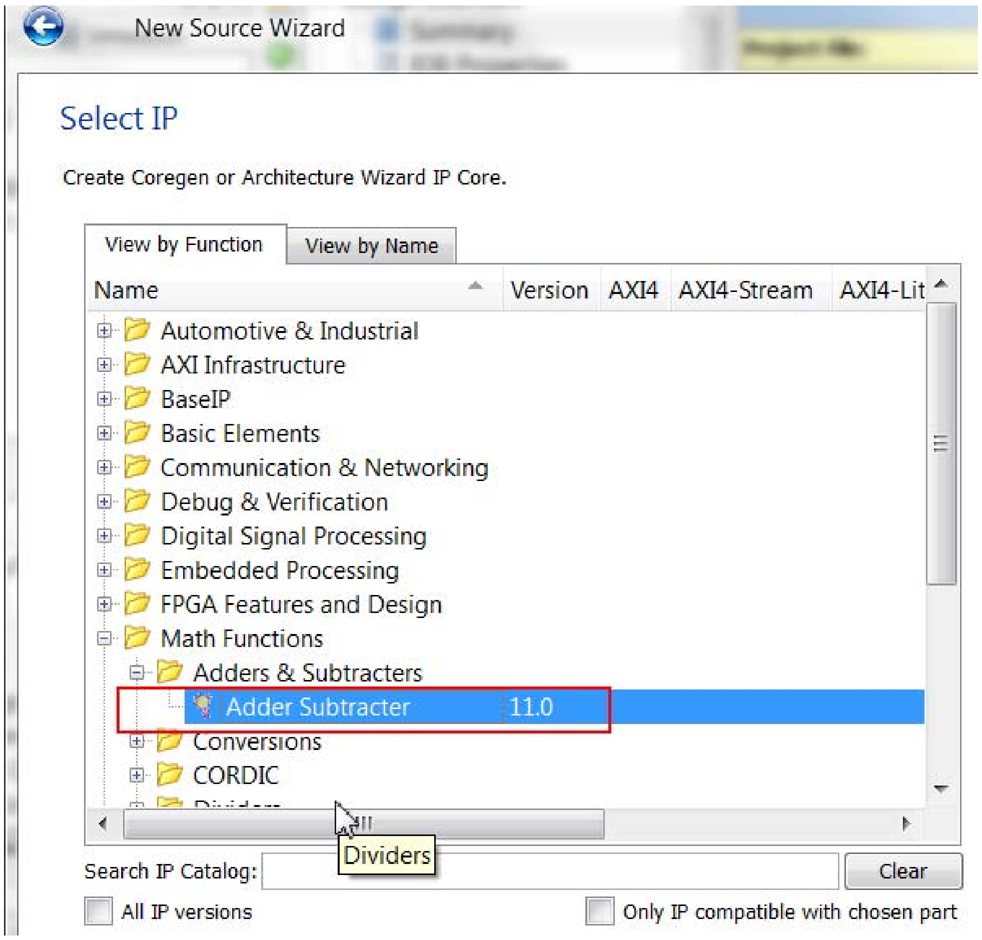
Select source type, file name and its location.

BMM File
ChipScope Definition and Connection F
Implementation Constraints File
IP (CORE Generator & Architecture Wiz
MEM File
Schematic
User Document
Verilog Module
Verilog Test Fixture
VHDL Module
VHDL Library
VHDL Package
VHDL Test Bench
Embedded Processor

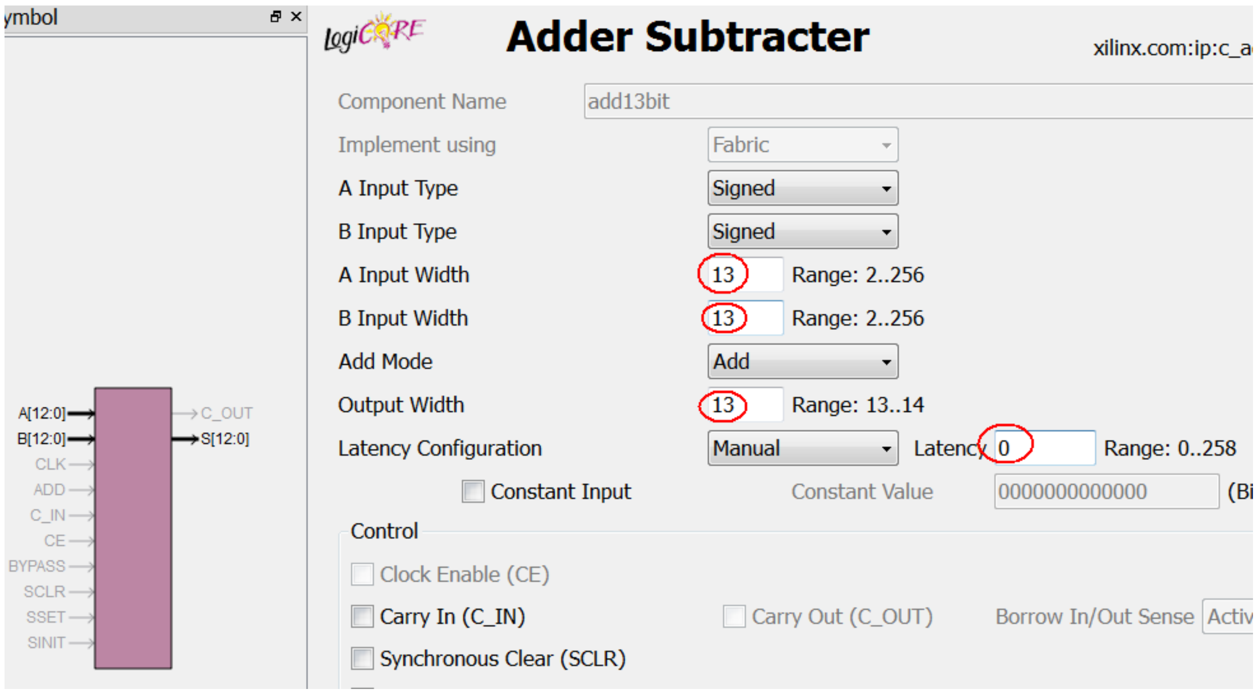
File name: add13bit

Location: 743\projects

4. In the 'Select IP' window, select 'Adder Subtractor' under Math Functions | Adders & Subtractors, then click Next.



5. In the Summary Window, click 'Finish'.
6. At this point, the generator window for the Adder/Subtracter module appears.
7. Set the input bus widths to 13 bits.
8. Set the output bus width to 13 bits.
9. Set the latency to 0 (no clock pin).
10. Then click 'Generate'. This creates a sub-directory named *ipcore_dir* and the adder as *add13bit.v*. This Verilog file is good for behavioral simulation only – other files in this directory describe the actual design in optimized Xilinx format. You need to examine this file so you can determine what port names is actually used in the adder.


ymbol

LogiCORE Adder Subtracter xilinx.com:ip:c_a

Component Name: add13bit
Implement using: Fabric
A Input Type: Signed
B Input Type: Signed
A Input Width: 13 Range: 2..256
B Input Width: 13 Range: 2..256
Add Mode: Add
Output Width: 13 Range: 13..14
Latency Configuration: Manual Latency 0 Range: 0..258
Constant Input [ ] Constant Value 0000000000000 (Bi

Control
Clock Enable (CE) [ ]
Carry In (C_IN) [ ]
Carry Out (C_OUT) [ ]
Borrow In/Out Sense Activ
Synchronous Clear (SCLR) [ ]

11. You need to incorporate this into your design. Comment out the Verilog statement(s) that you used to implement your original adder, and replace it with the statement that looks similar to the following (you will need to modify the output net name '`sum`' to match whatever temporary signal name that you used in your implementation).

```
add13bit u1 (.a({1'b0,a}), .b({1'b0,b}), .s(sum));  //coregen adder
```

This instantiates the Coregen adder in your design. The a, b inputs are padded by 1-bit since this adder is a 13-bit adder, not a 12-bit adder – however, we only did this to get access to the Carry out bit. Only the lower 12-bits of the adder result is used as the sum. For some reason, the Coregen adder behavioral model does not work correctly if you use 12-bit inputs and a 13-bit output. This is why the inputs were padded to 13-bits (this is not necessary in Part 1).

12. Verify that your simulation gives the correct result.
13. Use the Process | Implement Top Module command to map this design.
14. Record the number of 4-input LUTs that it took to implement this design (see the 'Design Summary' page).

GRADING

The grading for this lab is as follows:

a. Working Task 1: 55 pts (0 or no-credit – there is no partial credit).
b. Working Task 2: 20 pts (0 or no credit, there is no partial credit).
c. Report questions (25pts, partial credit assigned).

SUBMISSION INSTRUCTIONS

Create a directory named 'lab2_*netid*', i.e., (lab2_rbr5).

Copy the lab2_part1, lab2_part2 directories to this directory.
Copy your completed report.doc to this directory.

From Windows, select the 'lab2_netid' folder, and from the right-click menu, use 'Send To Compressed (zipped) Folder' command to produce a ZIP archive named 'lab2_netid.zip'.

Upload your 'lab2_*netid*.zip' file to Blackboard using the Lab2 assignment link.