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 IP Catalog tool used for generating IP (intellectual property) cores. This lab does not require use of the Basys 3 board.

BACKGROUND

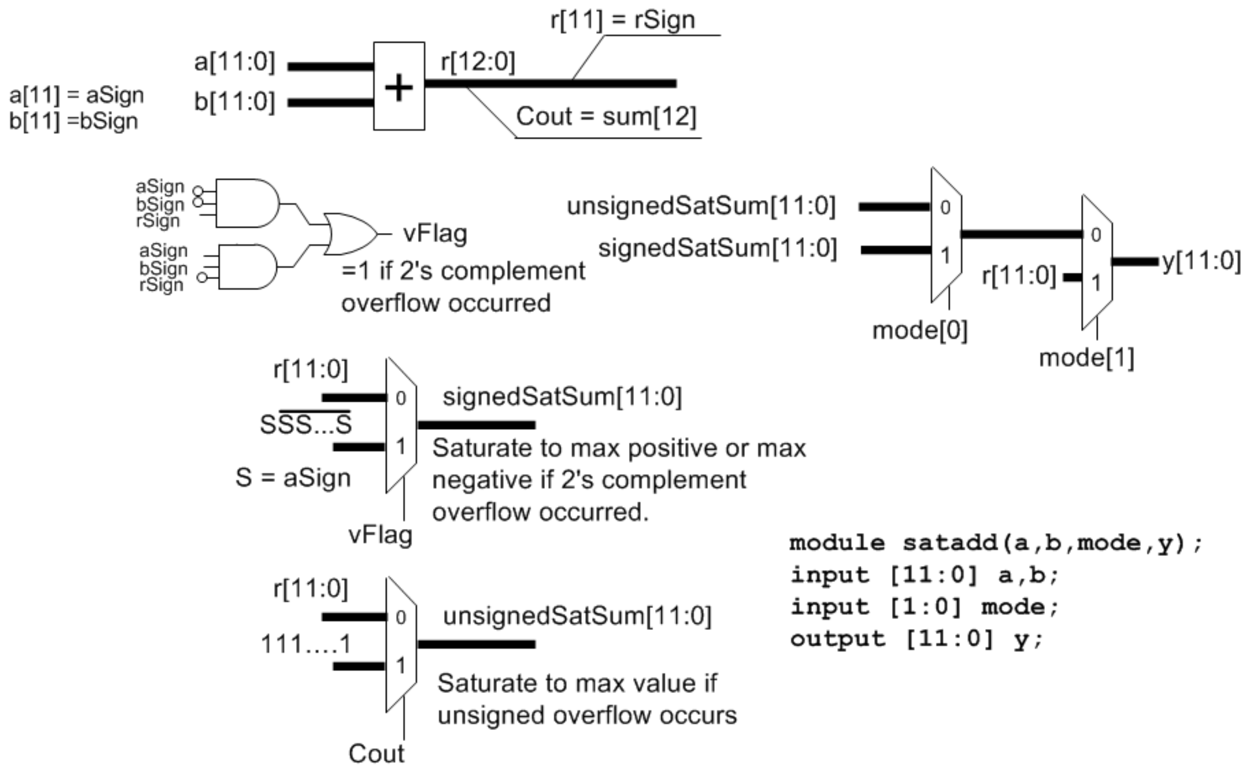
Review the Power Point notes on the Class website titled "Fixed-Point Numbers". This reviews the definition of signed integers using 2's complement encoding, and also the definitions of unsigned saturating addition and signed saturating addition. You need to understand this material in order to complete 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 be kept as specified.



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

Implementation Guidance

1. All of this logic must be implemented using *assign* statements; you may not use *always* blocks/sequential statements.
2. 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.
3. 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 *lab3_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.

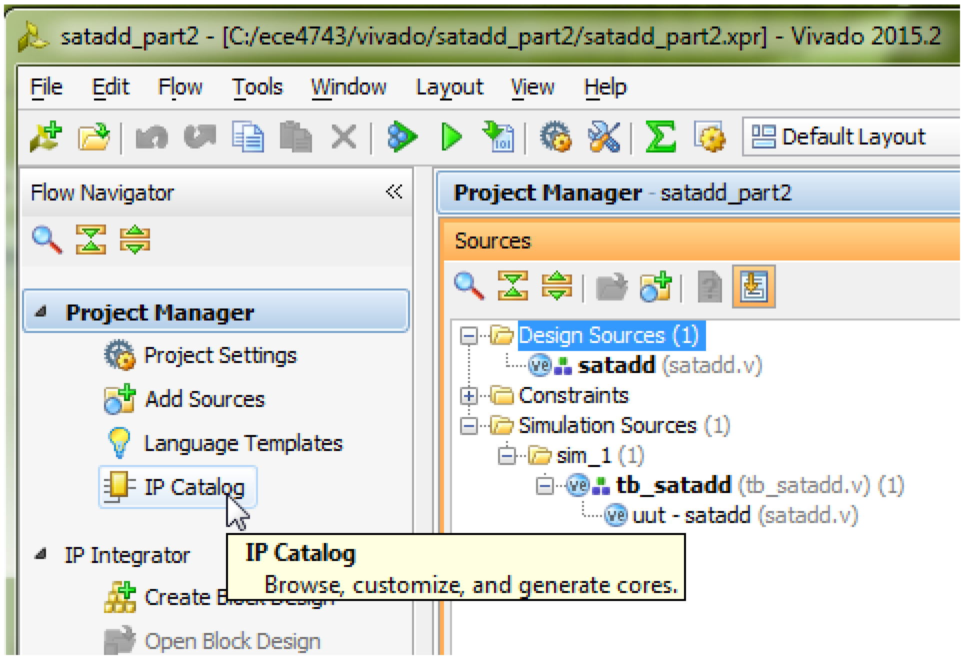
5. Implement your Verilog code in the *satadd.v* file. Before running the simulation for the first time, use the 'Run Synthesis' command to run the synthesis tool to verify that the design has no syntax errors and can be synthesized . Look at any warning messages generated by the synthesis tool and fix your code to address these (look at the slides titled '*Common Verilog Mistakes*' for hints on what is wrong if you get errors/warning messages from the synthesis tool). After your design is passing synthesis, test it with the *tb_satadd.v* using the 'Behavioral Simulation'. If you have vectors that fail, use the Debugging methodology from the previous lab exercise to debug your design until all vectors have passed.
6. Used the 'Run Implementation' command to implement your design.
7. Re-simulate using the 'Post-Implementation Timing Simulation' and verify that all vectors pass.
8. From the 'Reports' Tab, open the 'Place Design/Utilization Report'. Copy the tables that give 'Slice Logic' and 'Summary of Registers by Type' into your report. This design should have no registers reported.

WARNING: You will get NO CREDIT if your design fails any test vector or if it does not pass the synthesis tool!!! There is no such thing as a 'partially working' Digital System!!!!

TASK 2

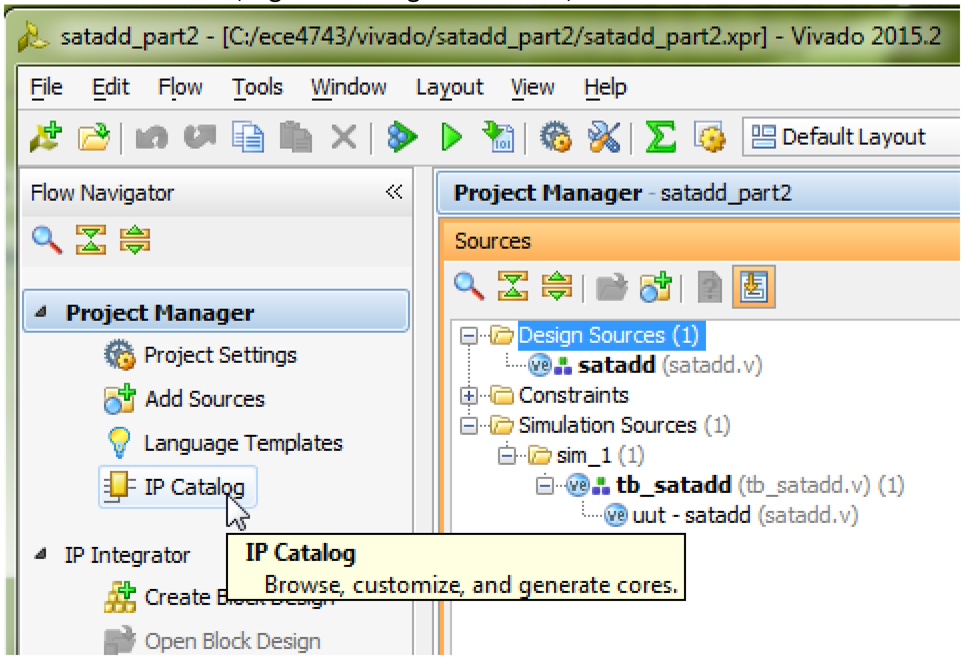
This task introduces you to the Xilinx IP Catalog. Task 1 must be working before you can do Task 2. The IP Catalog tool creates optimized versions of many common building blocks used in Digital Systems Design. These IP cores range from the relatively simple (adders) to very complex (square root, digital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

1. Create a project named *lab3_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 IP Catalog tool.
3. In the left hand window, select 'Ip Catalog'



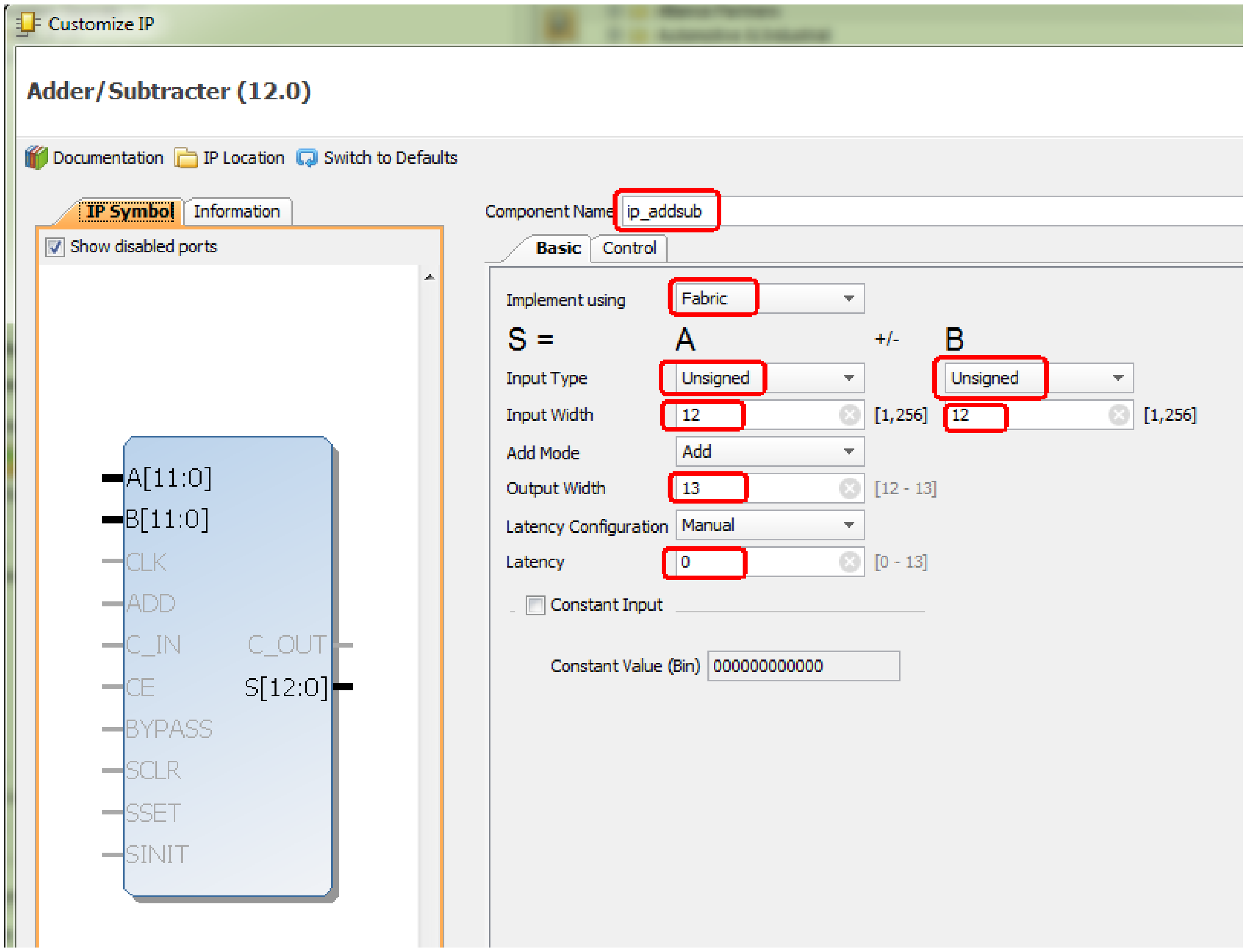
IP Catalog
Browse, customize, and generate cores.

4. In the 'IP Catalog' window, select 'Adder/Subtractor' under Math Functions | Adders & Subtractors, right-click to get the menu, and then select 'Customize IP…'



IP Catalog
Browse, customize, and generate cores.

5. In the Adder/Subtractor Window, select/edit the options as shown below. The final component should only have ports labeled as 'A', 'B', and 'S'. The click the 'OK' button.


Customize IP

Adder/Subtracter (12.0)

Show disabled ports: ✓

Component Name: ip_addsub

Implement using: Fabric

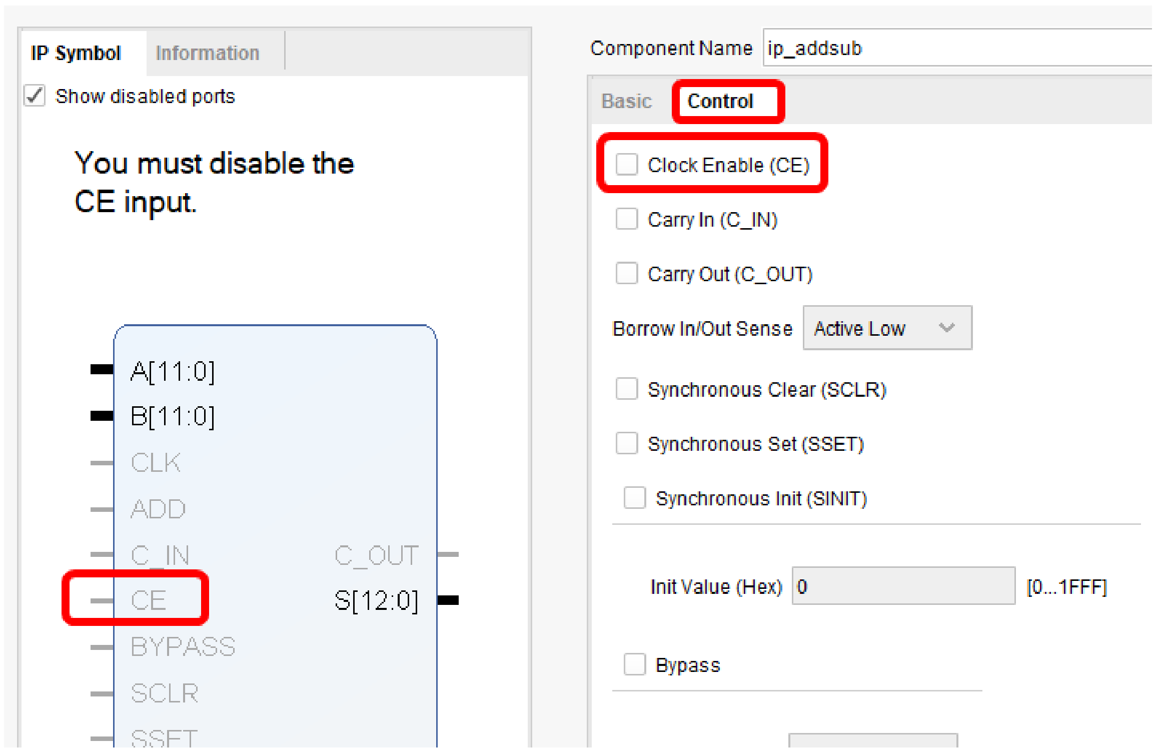
S = A +/- B

	A		B	
Input Type	Unsigned		Unsigned	
Input Width	12	[1,256]	12	[1,256]
Add Mode	Add			
Output Width	13	[12 - 13]		
Latency Configuration	Manual			
Latency	0	[0 - 13]		

Constant Input: ☐

Constant Value (Bin): 000000000000

If the CE input is not disabled, the click on the CONTROL tab and disable it. It will be grayed-out when disabled.



☑ Show disabled ports

You must disable the CE input.

Component Name: ip_addsub

☐ Clock Enable (CE)

☐ Carry In (C_IN)

☐ Carry Out (C_OUT)

Borrow In/Out Sense: Active Low

☐ Synchronous Clear (SCLR)

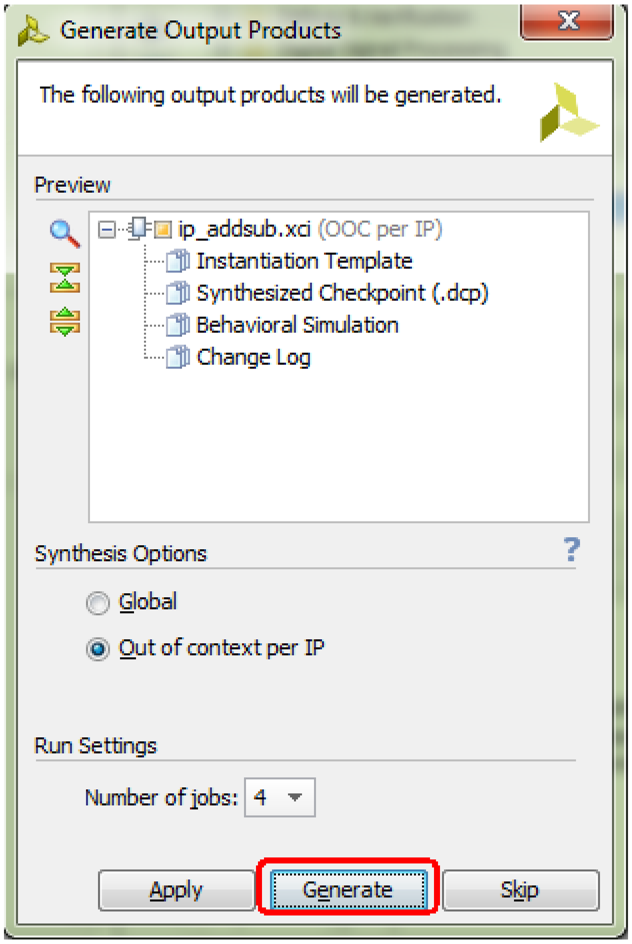
☐ Synchronous Set (SSET)

☐ Synchronous Init (SINIT)

Init Value (Hex): 0 [0...1FFF]

☐ Bypass

6. In the 'Generate 'Output Products', click the 'Generate' button:



Generate Output Products

The following output products will be generated.

Preview

- ip_addsub.xci (OOC per IP)
  - Instantiation Template
  - Synthesized Checkpoint (.dcp)
  - Behavioral Simulation
  - Change Log

Synthesis Options

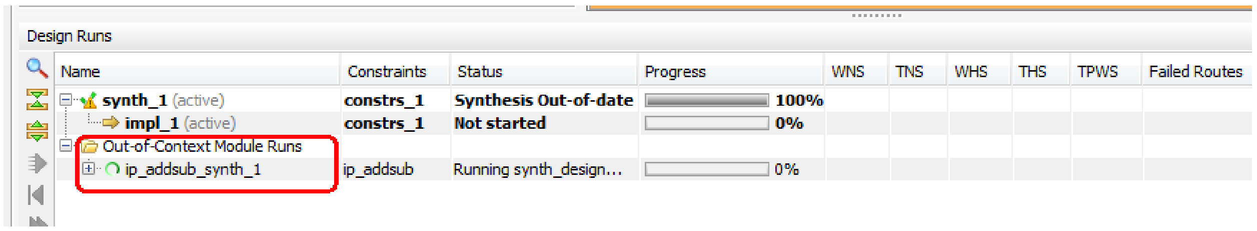
( ) Global

[✓ Out of context per IP]

Run Settings

Number of jobs: 4

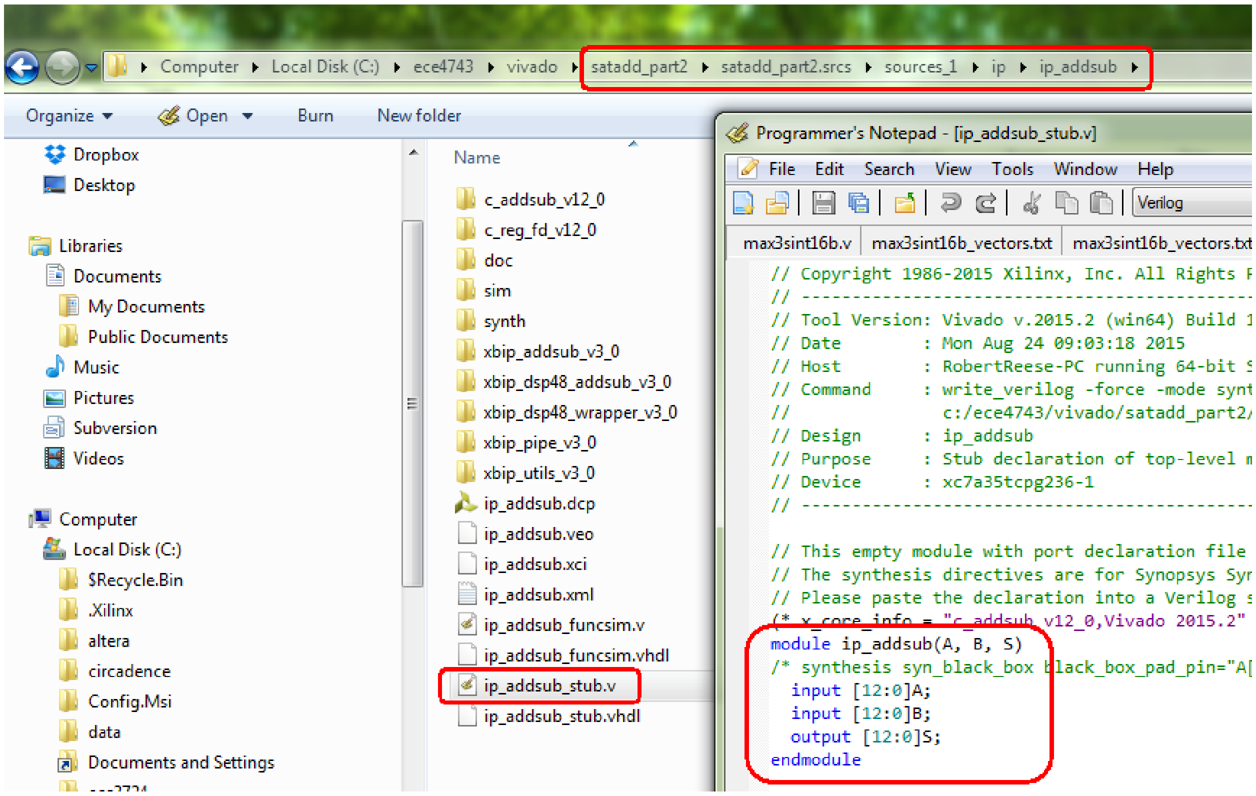
7. At this point, Vivdao begins generating the IP block:



Design Runs

Name	Constraints	Status	Progress	WNS	TNS	WHS	THS	TPWS	Failed Routes
synth_1 (active)	constrs_1	Synthesis Out-of-date	100%						
impl_1 (active)	constrs_1	Not started	0%						
Out-of-Context Module Runs									
ip_addsub_synth_1	ip_addsub	Running synth_design...	0%						

8. Once the IP block is generated, you can navigate down the '*designname*.srcs/sources_1/ip/ip_addsub folders to see the files generated. This is informational only, and is not necessary. You may find this useful in later labs to see the exact port names used in the Verilog file.



```
// Copyright 1986-2015 Xilinx, Inc. All Rights F
// ---------------------------------------------
// Tool Version: Vivado v.2015.2 (win64) Build 1
// Date        : Mon Aug 24 09:03:18 2015
// Host        : RobertReese-PC running 64-bit S
// Command     : write_verilog -force -mode synt
//               c:/ece4743/vivado/satadd_part2/
// Design      : ip_addsub
// Purpose     : Stub declaration of top-level m
// Device      : xc7a35tcpg236-1
// ---------------------------------------------

// This empty module with port declaration file
// The synthesis directives are for Synopsys Syr
// Please paste the declaration into a Verilog s
(* x_core_info = "c_addsub_v12_0,Vivado 2015.2"
module ip_addsub(A, B, S)
/* synthesis syn_black_box black_box_pad_pin="A[
  input [12:0]A;
  input [12:0]B;
  output [12:0]S;
endmodule
```

9. 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).

```
ip_addsub u1 (.A(a), .B(b), .S(sum));  //IP catalog adder
```

Replace the netname ‘sum’ with whatever net name you used in your Verilog for the adder output! The ‘a’,’b’ inputs are 12-bits, the output ‘sum’ is 13 bits.

This instantiates the IP Catalog adder in your design.

YOU MUST REMOVE THE VERILOG STATEMENT(s) THAT IMPLEMENTED THE ADDER IN THE PART 1 DESIGN!! The add operation is to be performed by the Coregen adder!

10. Verify that your behavioral simulation gives the correct result.
11. Use the ‘Run Implementation’ command to map this design.
12. Re-simulate using the ‘Post-Implementation Timing Simulation’ and verify that all vectors pass.
13. From the ‘Reports’ Tab, open the ‘Place Design/Utilization Report’. Copy the tables that give ‘Slice Logic’ and ‘Summary of Registers by Type’ into your report. This design should have no registers reported. This should be the same number of LUTs as in the first design (this design is so simple, there is no advantage to using an IP block).

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 'lab3_*netid*', i.e., (lab3_rbr5).

Copy the lab3_part1, lab3_part2 directories to this directory.
Copy your completed report.doc to this directory.

From Windows, select the 'lab3_netid' folder, and from the right-click menu, use 'Send To Compressed (zipped) Folder' command to produce a ZIP archive named 'lab3_netid.zip'.

Upload your 'lab3_*netid*.zip' file to Blackboard using the Lab3 assignment link.