Suggestions for Lab Success (v.3)
With System Verilog, Vivado, and BASYS3

Design Flow

The correct design flow for developing digital circuits is shown from top to bottom on the left-hand side of the Vivado main window. Students especially should follow the steps in this order, to avoid getting into trouble: write a source module in System Verilog, then check its syntax. Then write a System Verilog testbench for it, then do a simulation. When results are correct, repeat again for the next source module. When all source modules are known to work all the way up to the top level one, then write a constraint file for the top level module in the design, and synthesize. When that works, download to the BASYS3 board and do physical testing.

System Verilog

System Verilog is a parallel modeling language; it is not a sequential programming language like C++ or Java. It tells what should be, not what should happen. Don't try to write System Verilog codes with the same mindset that you write C++ or Java codes.

System Verilog files are filetype .sv Therefore, if you create files of type .v it will understand that these are supposed to contain Verilog code; if you put System Verilog code in them, there will be many syntax errors!

To write code correctly in System Verilog, follow the patterns ("idioms") given in Chapter 4 of our textbook (DD & CA, 2nd ed). These are very good examples of System Verilog.

If X is defined as an 8-bit signal, then X[5:0] = blah blah will only assign 6 bits, leaving X[7] and X[6] unassigned. A number of problems can arise from this, including unwanted latches, cryptic error messages, etc. The number of bits assigned should match the number bits defined

If X[7:0] is defined as an 8-bit signal, trying to assign X[8] and X[9] etc will result in an error. Trying to assign non-existent signals (like X[8]) in the constraints file will result in very cryptic error: "WARNING:[Labtools 27-3123] The debug hub core was not detected at User Scan Chain 1 or 3. Resolution: 1. Make sure the clock …."

When you use code modules you didn't write, be sure to study them thoroughly until you understand them. If you need help, ask questions. DON'T use something if you don't understand how it works!

Source modules in Vivado:

System Verilog module names must not begin with a digit. As an example, instead of "8to1mux", use _8to1mux, or mux8to1 as names. If you make this syntax error, Vivado causes the module to be put into a folder with the word "Error" in the name, visible in the view which shows modules in the tree-like hierarchy. Because it is not in the correct place, the error message says something like "missing source" or "no source". In fact, there is a source, but it will not be compiled since it is incorrect.

In System Verilog, instantiations of user-defined modules require an instance name. For example, instantiating mux1 requires the statement mux1 m1(...) . A common syntax error is to instantiate modules without giving them an instance name, i.e. mux1 (…).

When sources are Created, or Added, be sure to choose "System Verilog" as their language. This includes even testbench modules used for simulation. [All files except the constraint file will be

type .sv . The constraint file, of type .xdc, is the only exception.] It is very important that the source codes inside your .sv files be System Verilog. This applies to all modules you will create, and also to modules you didn't create but will use in the Project. All must be System Verilog, not the earlier Verilog. Otherwise, many syntax errors will result, and cause you many problems.

Testbench modules:

Testbench modules are separate and independent. They are not part of any other module, nor included inside any other module. If a testbench file is put inside a normal module modeling a component, it will create many syntax errors!

Testbench modules in System Verilog must contain exactly 1 instantiation of the uut ("unit under test") module. This uut is a regular System Verilog module of source code for the system or part of the system which will be tested in simulation. Besides the one instantiation of the uut, the testbench module should contain System Verilog *initial* procedures which stimulate the uut with test vectors, which are input patterns designed to prove if it works or not.

Chapter 4 and Chapter 7 of our DD & CA textbook contain examples of testbench modules, with correct syntax and good design.

Vivado

Project creation:

When Creating a new Project, Vivado must be instructed what the FPGA chip is that will be the eventual target for the hardware implementation. You must select the correct target FPGA—in our case , the chip used on the BASYS3 card is xc7a35t-1cpg236c. Even when Opening an existing Project, it is good to check to ensure that the correct FPGA is selected.

When the wrong FPGA chip is selected, several problems will result. The most serious is that the .xdc constraint file will be incorrect, referring to pins that don't exist on the wrongly-selected FPGA that the Vivado Project is targeting. Error messages will say such-and-such a pin named in the Constraint file doesn't exist on the chip.

Vivado creates many dozens of files in the course of making and managing the Project. Filepaths are critically important, in order to be able to locate the necessary files. Therefore, Projects created elsewhere will not easily move from one computer to another. It is best, to avoid creating errors, to Create new Projects on the computers in the lab, rather than try to transport files and directories created at home. Use the sources (System Verilog modules and constraint files) that you have created elsewhere, but Add them into a new Project you make in lab.

To avoid unnecessary errors, separate designs should be developed in separate Projects. For example, many students encountered errors in Lab 2 when the MUX parts and the decoder parts were together in the same Vivado Project. To avoid mistakes in which files relate to which (such as testbenches and constraints), it is strongly suggested that students keep things as separate as possible.

Syntax checking:

Since syntax errors are easy to find and fix, it is very good development strategy to find and fix these first, using the syntax checking feature of Vivado. After syntax is perfect, then the next step is to find any logical errors, using simulation.

Simulation:

Any source module can be simulated except testbench modules. Simulation requires 1 testbench module and 1 top-level source module which is instantiated in the testbench. The source module being simulated is the uut (unit under test).

The uut module being tested in simulation and the testbench module must be matched to each

The uut module being tested in simulation and the testbench module must be matched to each other for simulation to work correctly. Trying to simulate an 8-to-1 MUX with a testbench designed for simulating a 2-to-1 MUX will lead to strange and confusing error messages.

For simulation to work in Vivado, the testbench must "see" and be associated with the source module it is testing, and vice versa. Many Vivado errors with cryptic error messages will result if they do not correctly associate with each other. For this reason, it is recommended that for simulation, the Project only contain the testbench module that will be used in the simulation, plus the top-level source module being simulated (the uut) plus any modules needed at lower levels by the uut. Doing this (by saving files then closing the Project, creating a new Project with a different name, creating New source files with new names, and pasting in the testbench code and the uut's code) has been seen to allow simulation when other fixes didn't.

Several times in the lab, removing all other testbenches except one has resulted in simulation working. But with several testbenches present in the Project, Vivado only gave cryptic error messages and didn't simulate, or else the simulation was entirely flat, with no change in the waveform values.

If simulation happens, but waveforms are flat (no changes in inputs, outputs, or either) the testbench may have syntax errors. If it does, then it is moved to an "Error" folder, and the simulator program (Xsim) cannot see the test vectors. Therefore the uut will not receive any changing inputs and all waveforms will be constant.

Synthesis:

The constraint file (filetype.xdc in Vivado) must exist and be part of the project, located in the correct place in the tree-like hierarchy, in order for Vivado to be able to implement the design. It must have the same exact filename as the module it is associated with for implementation. Usually this is the top-level System Verilog module, so MyProj.xdc is the correct name for the constraint file which is for MyProj.sv Lack of a constraint file, or having it with the wrong name, or having it attached to the wrong place in the tree-like hierarchy, will result in many hard-to-understand errors!

There should only be one .xdc file in the project, and it should be associated with (i.e have the same name as) the System Verilog module that will be synthesized. Vivado problems may occur if more than 1 constraint file is in the Project.

Your .xdc constraint file used in synthesis must contain constraints for every pin of the FPGA that will have a signal on it (including clock, control signals, data inputs, or data outputs). Only +V and GND are automatically connected. Constraints tell which I/O pin to connect to which internal logic signal, and what voltage levels to use (the I/O Block for each pin is programmable, and can operate at different voltage levels). Rather than list physical pin numbers on the FPGA chip (like 137 or 216), the inputs and outputs are given names like T2 and U1, corresponding to the BASYS3 board places they are connected to. These names must be used in the .xdc constraint file when you implement.

The full listing of I/O constraints for every pin on the xc7a35t-1cpg236c FPGA that is connectable is are given in the basys3xdc.docx file posted on Unilica. Because these are written in correct .xdc syntax, students should copy-paste from this file, to avoid typos and resultant syntax errors. Hand-typing the constraints is likely to create a lot of syntax errors! One that can occur is not having a space between the get_ports and the curly bracket { After copy-pasting the lines you need into your .xdc source file, be sure to eliminate the # characters from the copied constraint code.

"Object not created" error will occur when synthesizing if the wrong FPGA chip is selected for the Project. Check Project > Properties to see if the xc7a35t-1cpg236c is the target FPGA for your Project. If not, the constraints are wrong for the chip selected, and this error will be given.

"Object not created" error may occur for other reasons, but those have to do with testbenches, naming, position in the hierarchy, etc. Before synthesizing, get those things fixed first in the

naming, position in the hierarchy, etc. Before synthesizing, get those things fixed first in the simulation step. When simulation works (and all modules are fine), but adding the .xdc constraints file for simulation causes this error, try removing all the testbench files from the project, Saving, and re-implementing.

Saving files, then closing the Project, creating a new Project with a different name, creating New source files with new names and a new constraint file with the correct new name, and then pasting in the code for the modules to be implemented and the .xdc constraint lines may allow you to get past hard-to-solve errors and be able to synthesize when other fixes don't work.

Dealing with errors:

When an error message says "source is empty" or "design empty", be sure to check that:

--the source actually exists (Saving may cause it to become visible when it wasn't before)
--it is correct error-free System Verilog (remember to check the syntax first before anything else, for every System Verilog module you write)
--the source is not in an "Error" folder (it has syntax errors)
--the source is located in the correct position in the tree-like hierarchy (does it need to be the top-level module, but isn't ?)
-- the source is selected for the operation you are trying to do

Sometimes when copying the Project from a place to another, the old directories are still used by the Project file. Close the Project and delete the contents of "cache" folder and it will now be forced to use the files from the new location.

"Please check that the file has the correct 'read/write/execute' permissions and the Tcl console output for any other possible errors or warnings." This error (and other of this kind) are still not well-understood. What follows is a collection of observations and suggestions from TAs and Tutors:

--possibly caused by wrong file paths or restrictions on the lab PCs (like not having full permission on the D:/ drive). Try using your own laptop, or a USB flash memory stick to locate all the project files on.
--possibly caused by syntax errors, such as beginning a module name with a digit, or in the testbench module
--closing Vivado and opening it and the Project solved this once
--changes in the Temp folder fixed this once
--maybe deleting the compile.bat file can fix this, assuming its contents are corrupt

"There is a mistake close to this character ';' " is a very unhelpful error message, since the error actually might be anywhere in the module, not necessarily in the line indicated (or even near it).

BASYS3

The FPGA chip on the BASYS3 board is a configurable array of gates. It is not a pre-defined microprocessor (nor a pre-defined anything). It will only be what the downloaded bit file programs it to be.

The 236 pins of the xc7a35t-1cpg236c FPGA chip on the BASYS3 contain many I/O pins. But all of these have been connected, via conducting traces on the board, to clocks, LEDs, switches, Pmod headers (connectors JA, JB, JC, etc), VGA connector, 7-segment display, and other physical places. The "pins" on the FPGA chip have names like T2 and U1, rather than numbers like 137 and 216. These are the names that must be included in the .xdc constraint file used when you implement.

Downloading a bit-file to the FPGA chip to program it requires connecting the USB cable from the computer to the BASYS3 board, and having the JP1 jumper on the BASYS3 in the JTAG position. Then use the steps in the tutorial to select the bitstream programming file and download it.

In trying to download the bitstream file, if the BASYS3 board is not recognized by Vivado, it could be that the USB connection isn't working. Try connecting to another USB port on the computer, and make sure the JP1 jumper is in JTAG mode. [Please report any non-working USB ports to the TAs, to be repaired later.]

General advice

Understand what you are doing, and why, before you try to do it. This means reading the Lab Assignment, and asking questions until you know what you are going to do, and why.

Do your own work, don't or borrow or copy someone else's code or design or PDR. How are you going to learn if you don't do the work yourself?

Coming in with a fully finished, correct Preliminary Design Report helps increase the chances of success in lab A LOT! But just having it is not enough; you must refer to it as you are doing the lab, at the appropriate points, to guide you.

Practice using Vivado, running simulations, synthesizing and downloading the bitfile, etc until the development platform and tools and tasks become familiar. This will allow you to concentrate on the design tasks of each Lab.

Simulate first, and get that working. That will find and expose both syntax errors and logical errors in your System Verilog modules. Once the simulation of the circuit/system is working well, you can have confidence in your model of the system. Now you are ready to synthesize (i.e. implement, then generate the bitfile, download it, and program the FPGA, and test the circuit/system in hardware.

Experience is the key to debugging. The more experienced you are, the more you will be able to pick up the clues, get to the problem, and fix it quickly. [Having said that, we hope that this document will speed you up the learning curve, by giving you the benefit of others' experiences.]

Searching with Google about the error message can yield valuable information online to help you understand the problem, and quickly fix it. Learn to find and fix your own errors, only calling for help when you have tried your best, having searched online.