

55:131 Introduction to VLSI Design
Project 3: SPI Master Code Coverage and Synthesis
Due Date: Friday, December 13, 2013

This is the last of three projects developed for our class. This project has four parts:

- Measure the code coverage for an SPI master.
- Synthesize, place, and route the SPI master. Measure its area and timing.
- Create a bit file for the SPI master and try it in a Xilinx development board.
- Data to turn in and questions

It may be useful to review the questions before proceeding through either Part 1 or Part 2.

Part 1 -- Code Coverage

I. What is Code Coverage?

Code coverage provides a way to measure how much of the design has been exercised by the test bench. The intent of coverage is simple. Coverage metrics should help answer questions such as:

- Have I done enough testing?
- Are my test cases complete?
- Have I exercised all my design's code?
- What have I verified?

Code coverage alone cannot fully answer these questions, but it can help. The nice thing about code coverage is that it can provide metrics with no change to the design flow. Nothing special needs to be done – just turn it on, run the simulation, and analyze the results. If 100% code coverage is not achieved, this reveals one (or more) of the following:

- The test suite is incomplete, in which case more tests are needed.
- There is unreachable but necessary code in the RTL.
- There is unused and unneeded RTL code, whether accidentally or deliberately created (also called “dead code”)

II. Installing Questasim

Questasim is installed on the Linux-based computers in the Elder and Herring labs. After logging in, you'll need to source the Mentor setup file by typing the line below at the Linux prompt (assuming you are using the *bash* shell):

```
source /usr/css/etc/mentor_setup.sh
```

To avoid having to source this file every time you start a session, you can add the source command line in your *.bashrc* file (which is in your home directory):

- open your *.bashrc* file with a text editor
- add the following two lines at the end of the file (the first line is only comment):

```
#source of setup file for Mentor Tools  
source /usr/css/etc/mentor_setup.sh
```

III. Coverage Basics

Normally the goal is to enhance the test bench to achieve 100% statement, branch, condition, and expression coverage for all design units. Note that a few other types of coverage are available too. If there are statements, branches, conditions, or exceptions that are not reachable (for example, “when others” in some state machines), it may be necessary to understand and explain as to why they are not reachable. Note that “not reachable” means that no possible test bench stimuli could hit them -- they are truly unreachable. Coverage metrics are only useful on the modules in the SPI master, not on the test bench itself.

Statement coverage is the simplest form of code coverage. It checks that the test bench touched every executable statement. A statement is a line of code that ends in a semicolon. Most statements are assignments or calls to tasks, procedures, or functions. Statement coverage provides a very basic view of how well the test bench is covering the design.

If a statement contains conditions or branches, such as "if" or "case" statements, *branch coverage* tells us if the test bench exercised all the possible branch directions in the design.

Condition coverage is an extension of branch coverage. Branch coverage looks to see that all branches are exercised, while condition coverage looks to see that all reasons (conditions) for taking a branch are exercised. Questasim has two methods of measuring condition coverage, "UDP" and "FEC". "FEC" is more thorough.

Expression coverage is like condition coverage, but applies to continuous assignment statements. Expression coverage looks to see that all reasons for the evaluation of a continuous assignment statement (e.g. "when" statement) are exercised. Like condition coverage, Questasim has two methods of measuring expression coverage, "UDP" and "FEC", and "FEC" is more thorough.

IV. Code Coverage Limitations

100% code coverage does not guarantee a functionally correct design. Reaching 100% code coverage will not guarantee that the code is exercised enough, or in interesting ways. It cannot tell if concurrent design behaviors or interesting corner cases have been stimulated. In complex designs, functional bugs are typically caused by a combination of corner cases occurring simultaneously. That is, several different areas of the RTL must be simultaneously activated in order to cause the erroneous behavior. Code coverage only tells the user that the code was exercised in isolation, and does not provide any method to observe concurrent behavior.

And what if the code being exercised is actually wrong? You might stimulate the line in your testing (and therefore achieve coverage), but the test may not propagate a faulty value to an output for observation, or may fail to detect the faulty behavior.

Thus, in summary, code coverage is one of several metrics needed to determine verification completeness.

V. Compiling and Running Simulations with Code Coverage Enabled

Questasim supports an integrated approach to code coverage. In other words, coverage is collected during the simulation process itself. The process follows this flow:

- Specify the statistics to collect (e.g. "vlog -sv -cover bces PmodOLEDCtrl.v ")
where s=statement, b=branch, c=condition, e=expression
- Enable the collection of data (e.g. "vsim -novopt -coverage work.PmodOLEDCtrl_tb ")
- Save data (e.g. coverage save "my_coverage.ucdb")
- View or analyze the data (e.g. "vsim -viewcov"), more typically done in the GUI or a report file

Before running code coverage, change the text to be displayed so that it contains a personalized message with your initials in it. This change is made in a parameter array in the "OledEX.v" file.

All of the code is provided on ICON. When compiling the design for the simulator, use the "do" file provided with the RTL. Simply type "do oled.do" at the Questasim prompt to compile the design with coverage enabled. Note the "Delay.v" file has been changed to reduce simulation time. It will need to be changed before synthesis.

The memory file to be used for simulation is memory.v and the memory file to be used for synthesis is charLib.v. memory.v is behavioral and charLib.v was created by the Xilinx "core gen" tool. Both contain a charLib module. The charLib.mif is the memory information file for simulation. It contains the mapping of characters to "pixels" for the display. Without it, the SPI data sent to the display is incorrect. For synthesis, this information was provided to core gen in a "coe" file.

VI. Code Coverage Reporting

Figures 1 and 2 show what a typical display of code coverage reporting looks like in the Questasim GUI. To generate a coverage report, select Tools -> Coverage Report -> HTML or Text. Report selections for this project should be: Condition/Expression Tables, Source Annotation, FEC Analysis, Code Coverage, and All code coverage. Save the report for use in the questions.

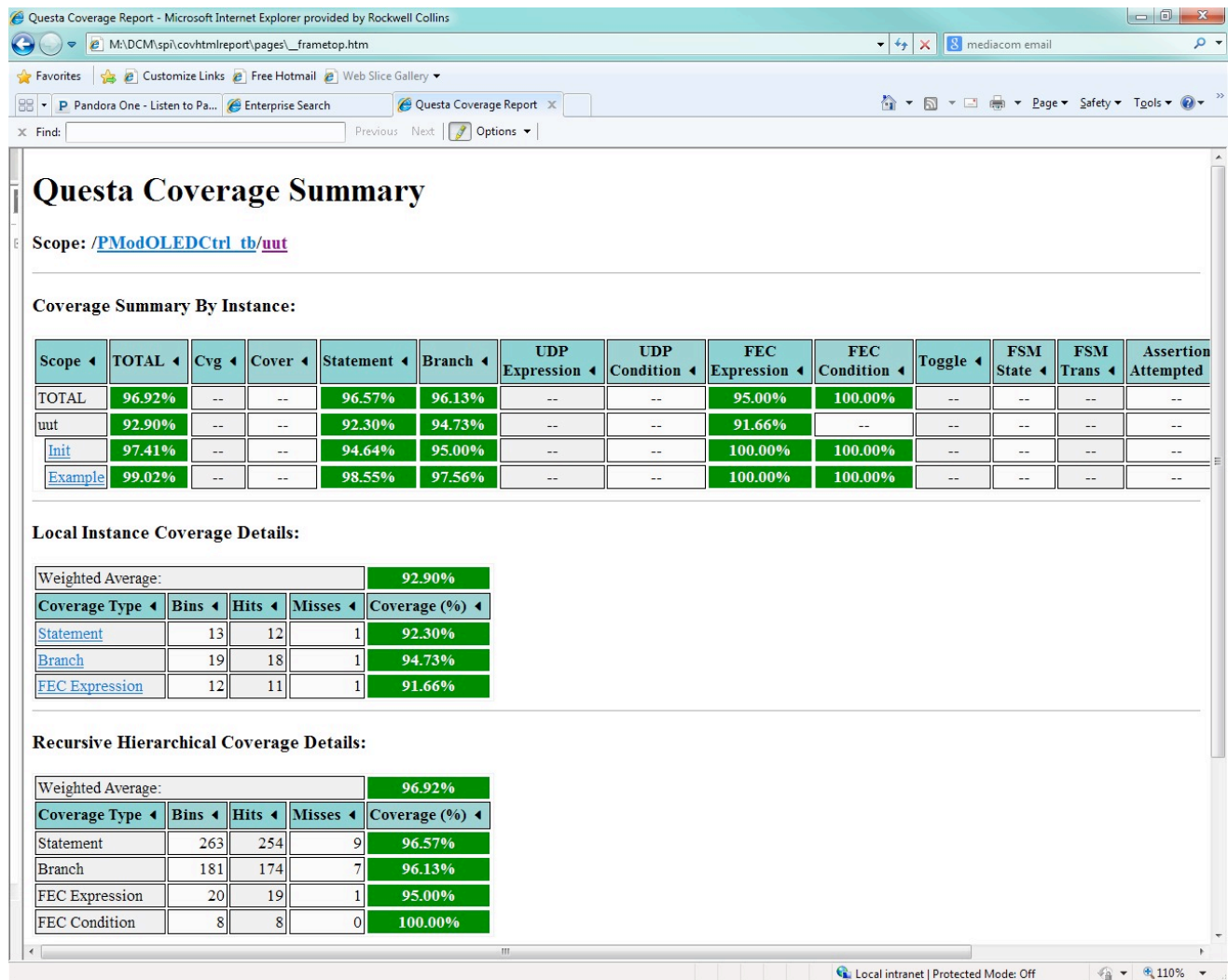


Figure 1 Code Coverage Reporting

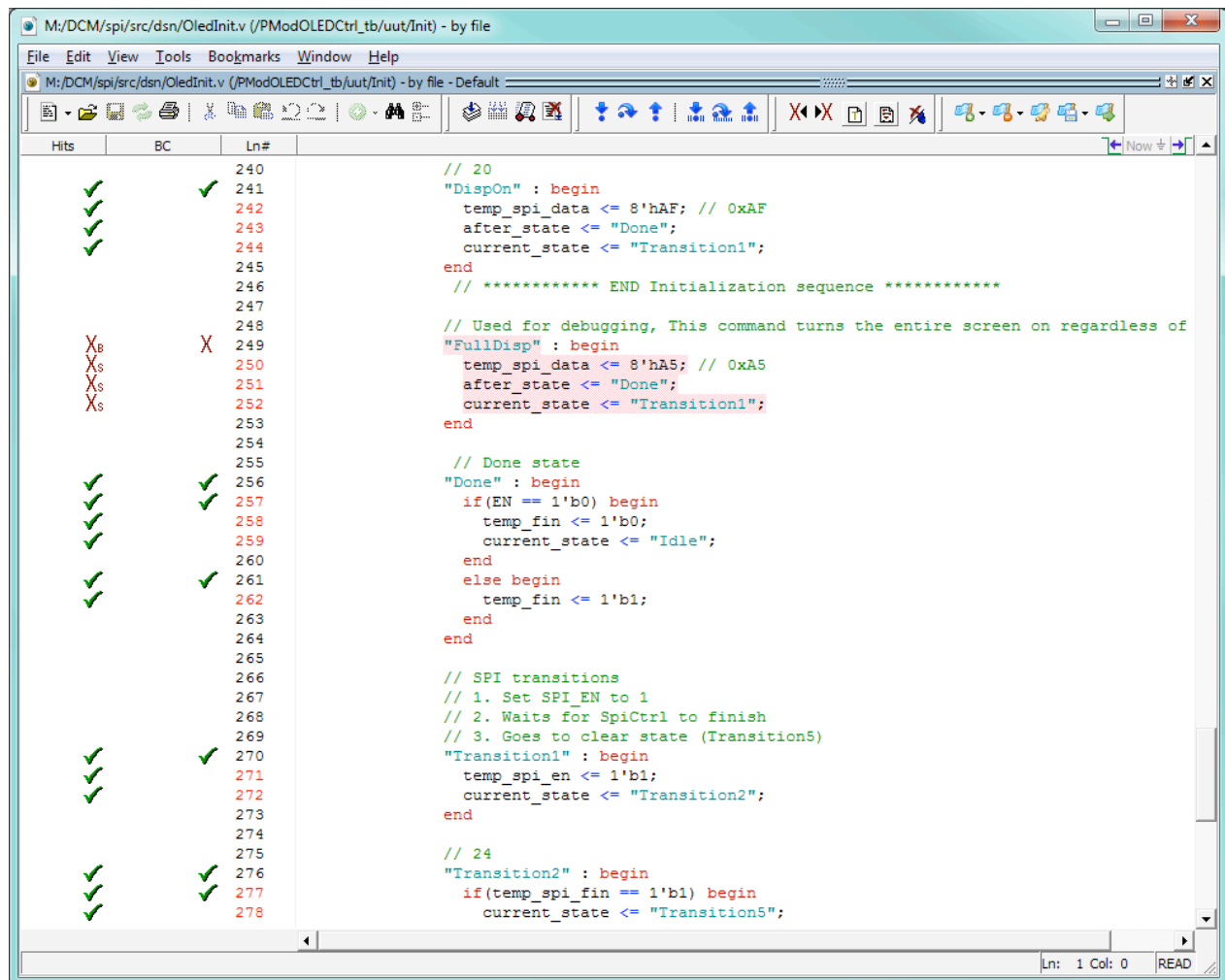


Figure 2 Questasim Code Coverage on Verilog File

Each instance in each design unit can be viewed to see coverage statistics for each of the various types of coverage metrics that have been turned on. This view also links into both the test environment and code, so you can debug your tests, code, etc. from this window.

VII. Questasim Help

Questasim help can be found under the help menu. A User's Manual and Reference Manual are both available.

Part 2 -- Synthesis, Place and Route, Area, and Timing

I. What is ISE?

ISE is a synthesis and place & route tool from Xilinx. It can be used to generate a netlist, timing information, support files and programming files for Xilinx CPLDs or FPGA from Verilog or VHDL RTL code. Several types of constraints, including timing, pin, mapping, and path, can be applied during synthesis and place & route.

Like Questasim, the ISE command line will interpret Tool Command Language (TCL) commands. TCL can also be used to run ISE in batch mode.

II. Installing ISE

ISE is installed on the PCs in the Elder and Herring labs. Start -> All Programs -> Xilinx Design Tools -> ISE Design Suite 14.1 -> ISE Design Tools -> 64-bit Project Navigator (or 32-bit Project Navigator, if you're running on a 32-bit machine).

III. Getting Started

After choosing File -> New Project, enter "proj_3_xx" as your project name, where "xx" are your initials, select a path where the project will reside, and select "Next". In the project settings, select "Spartan6" for the family, "XC6SLX45" for the device, "CSG324" for the package, "-2" for the speed grade, and select "Next". After reviewing the summary for accuracy, select "Finish". The design properties should resemble those shown in Figure 1. A file named "proj3_xx.xise" will be created in your design directory.

IV. Add Source Files and Constraints

Before adding the source code to the synthesis tool, change the delay time in "Delay.v" to the right value for synthesis (i.e. for real hardware). Open the file in a text editor, uncomment line 91 and comment out line 92. Also, in the OledEX.v file, comment out the "modelsim" parameters and uncomment the "xilinx" parameters. Make sure your customized message is in line 65.

Choose Project -> Add Source to add input files. Navigate to the RTL files (charLib.v, Delay.v, .OledInit.v, OledEX.v, SpiCtrl.v, and PModOLEDCtrl.v. A file to define the contents of the character RAM, charLib.ngc, is also needed during synthesis so that the Xilinx tool can load the RAM during configuration of the FPGA on the Atlys board. Add it the same way.

IO pins are often assigned using a constraints file. IO and timing constraints give the synthesizer and P&R tools information that enable the logical design (described by the RTL) to function as desired in the physical device. Constraint examples include IO pin assignments, IO signal type (such as LVCMOS, or SSTL_2), output drive strength, device input setup delay requirements, device output propagation delay requirements, and internal clock cycle requirements. In the case of the SPI master for this project, only a few timing and pin constraints are needed.

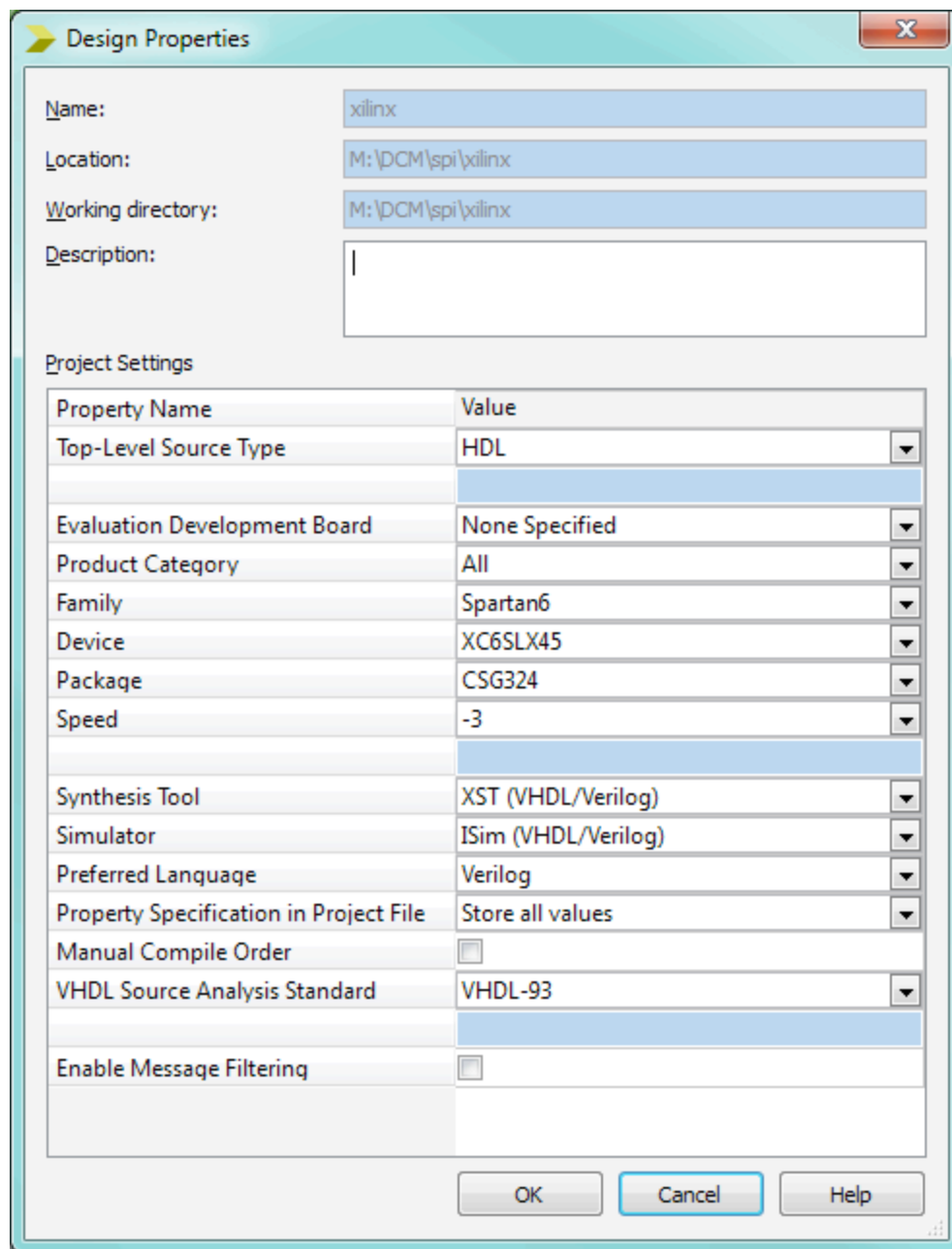
The IO pin assignments come from the Atlys board and OLED board schematics, which can be found here (for reference only):

<http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,400,836&Prod=ATLYS>

<http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,401,963&Prod=PMOD-OLED>

The pin numbers for the clock and reset inputs and the SPI outputs were found from these schematics and added to the oled.ucf file.

The only timing constraint we'll add also comes from the Atlys board schematic. A 100MHz clock is provided to the FPGA that can be used directly or sent to an internal DCM or PLL to synthesize higher frequencies for use by the FPGA. We're using the 100MHz directly in our design. Add the oled.ucf constraint file to the design by selecting Project -> Add Source



The image shows a 'Design Properties' dialog box with a teal title bar and a close button. It contains several input fields and a table of project settings.

Name: xilinx
Location: M:\DCM\spi\xilinx
Working directory: M:\DCM\spi\xilinx
Description: |

Project Settings

Property Name	Value
Top-Level Source Type	HDL
Evaluation Development Board	None Specified
Product Category	All
Family	Spartan6
Device	XC6SLX45
Package	CSG324
Speed	-3
Synthesis Tool	XST (VHDL/Verilog)
Simulator	ISim (VHDL/Verilog)
Preferred Language	Verilog
Property Specification in Project File	Store all values
Manual Compile Order	<input type="checkbox"/>
VHDL Source Analysis Standard	VHDL-93
Enable Message Filtering	<input type="checkbox"/>

Buttons: OK, Cancel, Help

Figure 3: Design Properties

V. Compile

Select Process -> Implement Top Module. This checks the syntax of the source code, and translates and optimizes the HDL code into a set of components that the synthesis tool can recognize.

At this point, design information, warnings, and errors will be reflected in the console window. Common issues include non-synthesizable code (such as inter-assignment delay), sensitivity list mistakes, and unintended latches. The state machines should be inferred by the compiler.

Make a habit of looking at the messages in the console window and learning what is normal. Warnings and errors should be taken seriously; they usually indicate real problems that will cause problems later if not corrected.

VI. Synthesize

Choose Process -> Implement Top Module to synthesize the SPI master. During synthesis, the HDL files are translated into gates and optimized for the target architecture. Again, additional information, warnings, and errors will be reflected in the console window. The timing report should be created.

VII. Explore the Design

Choose Tools -> Schematic Viewer -> RTL... or Technology to view the design in different ways. Find the CLK signal in both the RTL and Technology views. Note how it fans out in the RTL view but goes through a buffer in the Tech view. Gates in the RTL view are placed where needed to make the schematic easier to read, but in Tech view, Xilinx Look Up Tables (LUTs) are usually used. Double click on a LUT icon to see the LUT logic. Notice the canonical form used for the depiction of LUT logic. To view the critical path in the design, use Tools -> Timing Analyzer -> Post Place&Route, right click on the slowest path and select Show in Technology Viewer.

The Map and Place and Route reports show the device utilization and library elements used for the SPI master synthesis. This report would be another place to spot unintended latches which were inferred by the synthesizer. You should review these files for unintended latches.

The Report Timing icon in the Design Analysis bar shows, by default, the ten slowest paths in the design. It also shows the individual delays in the slowest (critical) path. Note that the paths will be shown regardless if the design meets timing or not. Compare the report's critical path to the Critical path schematic and the report's maximum speed – they should all agree.

A Verilog netlist can be generated by right clicking on “Generate Post-Synthesis Simulation Model” in the Design window and selecting run. The netlist will be in the netgen/synthesis subdirectory and is called PModOLEDCtrl.v. Open it with a text editor and take note of its structure. It should resemble the Tech view with LUTs shown. Note that the netlist entity's port names are the same as the RTL. It is possible to simulate the netlist model of the design (often called a structural or gate-level simulation) using the test bench used to test the RTL. After place and route, an "sdf" file, which contains timing information for each netlist element, can be included in a gate-level simulation. Since we use an event-driven simulator, a gate-level simulation can be

very slow when compared to RTL simulations. Trace the critical path from the View Critical Path schematic in the netlist.

Finally, open the constraint file (oled.ucf) and review its contents for the design speed and pads.

VIII. FPGA Architecture

The Spartan-6 architecture is an array of Configurable Logic Blocks (CLBs) connected together using programmable interconnects. Figures 2, 3, and 4, taken from the Spartan-6 CLB User Guide (UG384), show the CLB Array, interconnects, and simplified CLB. The various types of routing in the Spartan-6 architecture are primarily defined by their length. The single, double, and quad interconnects connect a CLB to every neighbor, every second neighbor, or every fourth neighbor, respectively, both vertically and horizontally.

Each CLB contains 2 slices, each of which contains four 6-input LUTs and eight flip-flops. The LUTs can implement any arbitrary 6-input combinational function or 2 arbitrary 5-input functions, assuming they share common inputs. The propagation delay through a LUT is independent of the function implemented.

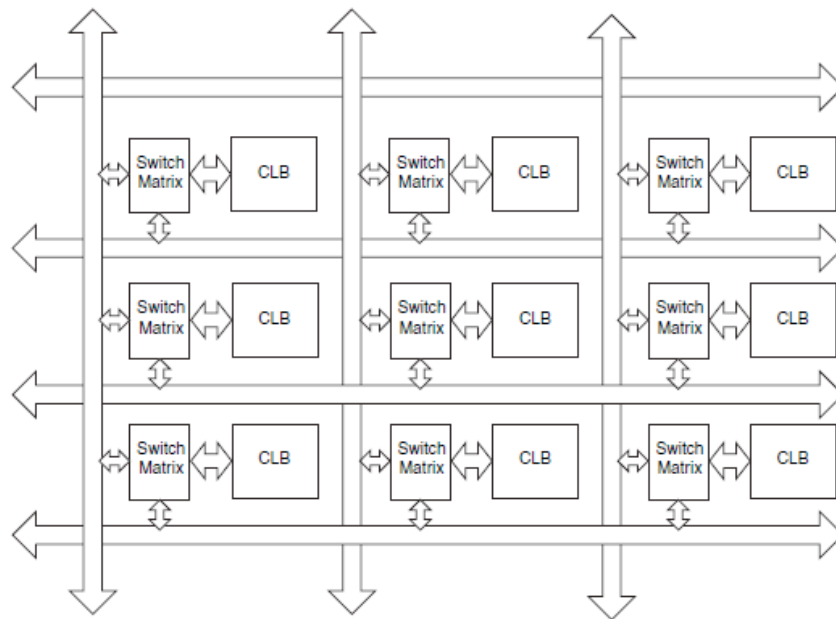


Figure 4: CLB Array and Interconnect Channels

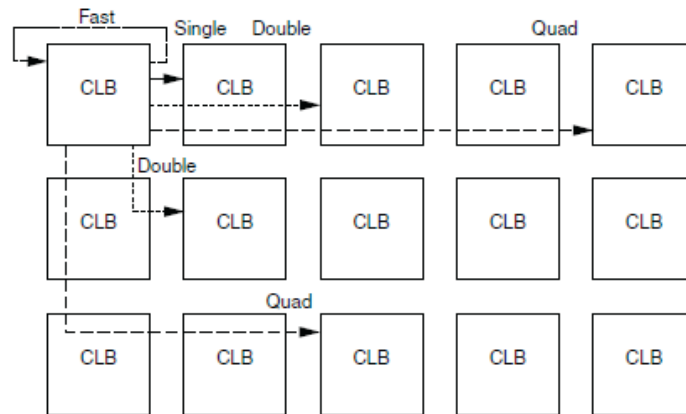


Figure 5: Examples of Interconnect Types

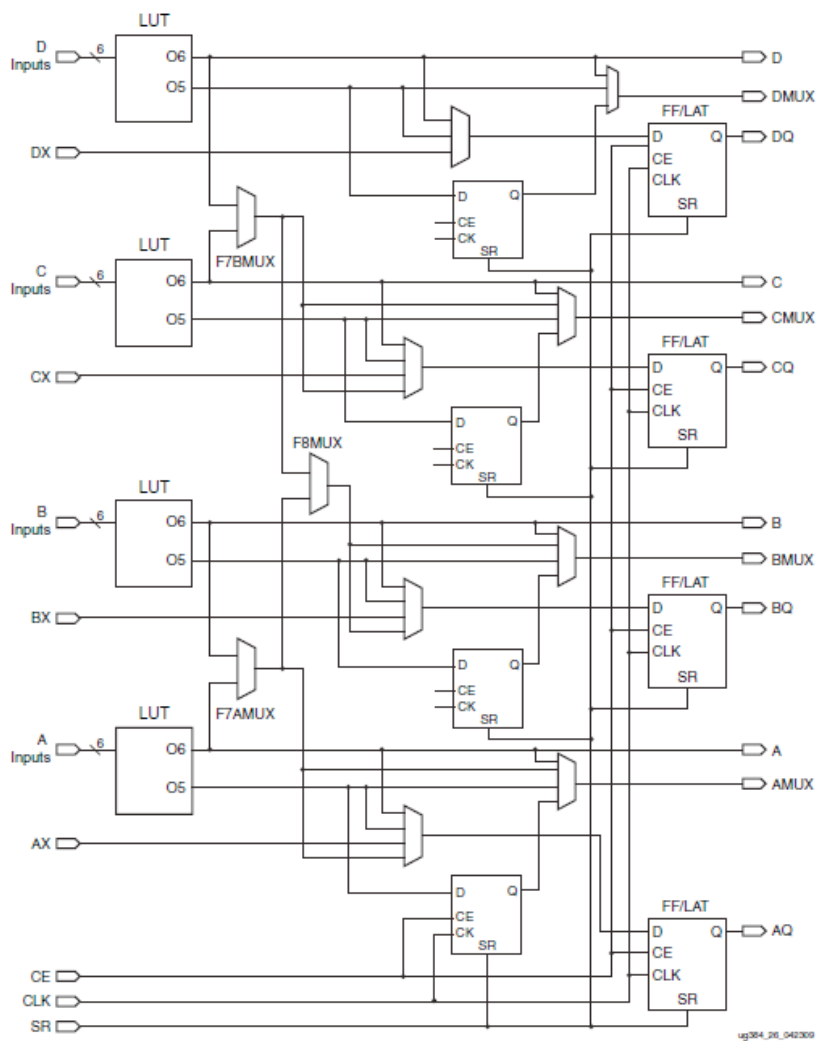


Figure 6: Simplified Spartan-6 FPGA Slice

Spartan-6 devices contain many other resources, including block RAM, Digital Clock Managers (DCMs), and high-speed bus PHYs (physical layers). The device we're using for this project, the XC6SLX45, is a mid-range device in the family. It contains 6822 slices. The Spartan-6 family is a smaller, cost-reduced cousin of Xilinx's Virtex-6 family of devices.

Part 3 -- Create a bit file for the SPI master and try it in a Xilinx development board

I. Programming File

In Xilinx ISE, generate a programming file by right clicking on “Generate Programming File” in the Design window and selecting run. The file created is pmodoledctrl.bit. Save this file for use with the Adept software, as described below.

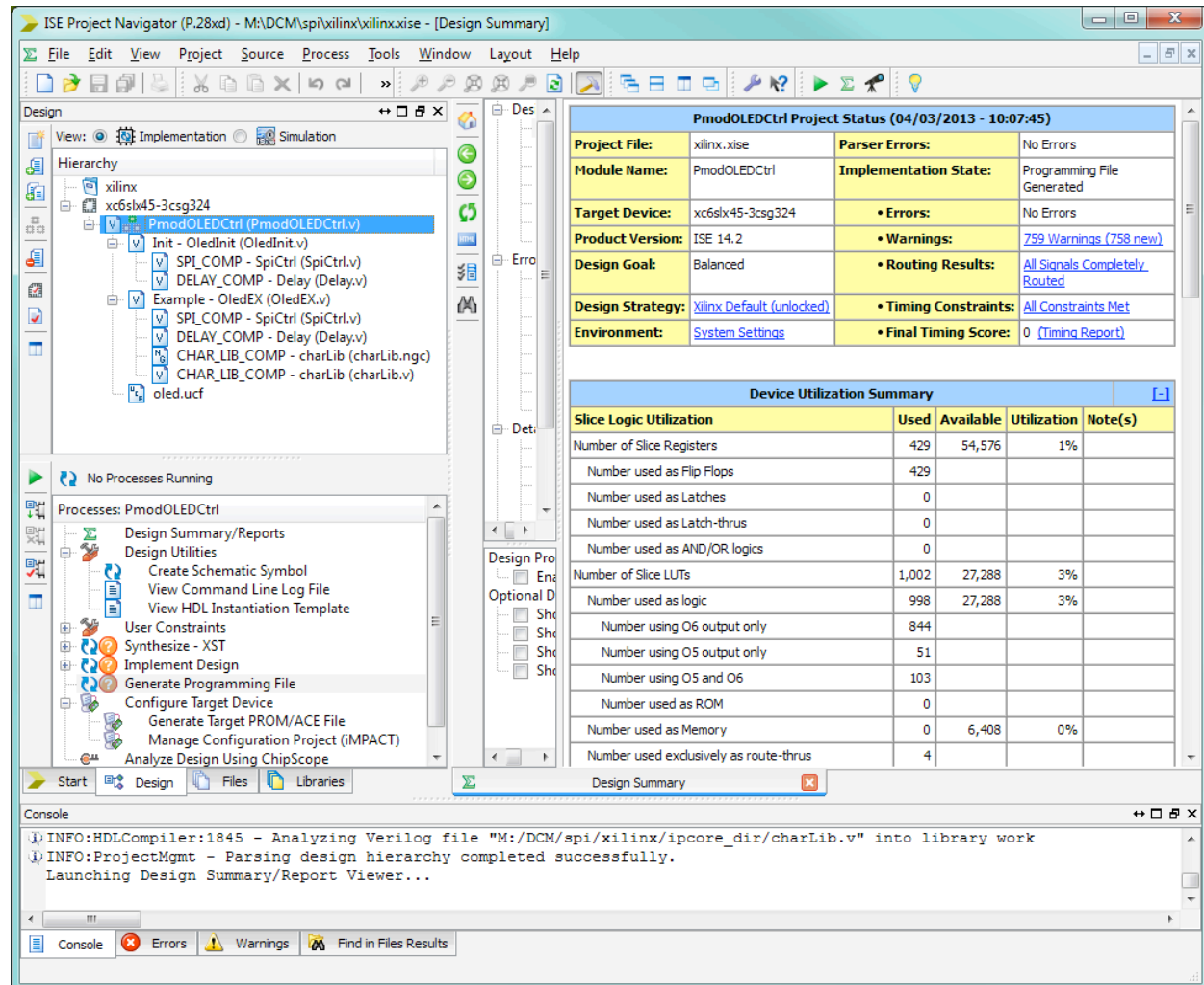


Figure 7: ISE Screenshot

This portion of the project needs to be done in the Digital Lab, 2244 SC or on your own PC with the Digilent "Adept" software installed:

<http://www.digilentinc.com/Products/Detail.cfm?Prod=ADEPT2>

You will also need an Atlys board and an OLED "PMOD" board from the instructor. Documentation for these two boards, for reference, can be found at:

<http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,400,836&Prod=ATLYS>
<http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,401,963&Prod=PMOD-OLED>

Follow these steps to download the programming file to the Atlys board:

1. Plug in the AC adapter and connect it to the Atlys board (don't turn on the Atlys board yet).
2. Connect the PMOD board to the Atlys board. The PMOD "VDD" pin should align with the Atlys connector's "VCC" pin. The "GND" pins from both boards should align.
3. Connect the Atlys board to USB programming port to a USB port on the lab PC.
4. Start the Adept software (Start -> All Programs -> Digilent -> Adept -> Adept)
5. Turn on the Atlys board (slide the switch next to the power input).
6. The Adept software should recognize the Atlys board (select the drop-down if necessary).
7. On the "Config" tab, browse to the programming file you created in the previous section.
8. Select program. A progress bar should show the status of FPGA programming.
9. Once programmed, the FPGA can be reset using the red reset button.

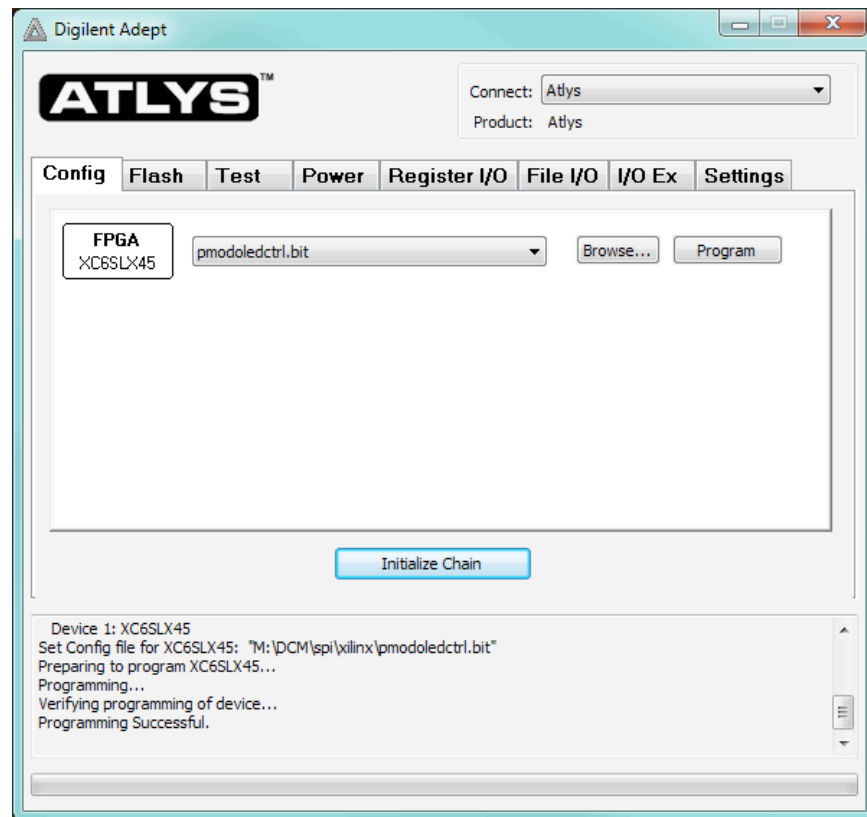


Figure 8: Adept Screenshot

II. Test and Photos

The message programmed in OledEX.v should be displayed. Take pictures of your “personalized” image on the display.

Part 4 -- Data to turn in and questions

I. What To Turn In

Please submit an electronic copy of each of the following items to ICON coverage report file, area report, timing report, constraints files, netlist, programming file, and schematic of the critical path (jpeg), and a photo of the OLED display running with your personalized image. Also submit a text file that answers the questions below.

II. Questions

1. Please indicate how many hours you spent on this project. This will not affect your grade, but will be helpful for calibrating the workload for the future
2. Generate a list of the uncovered statements, FEC expressions, and FEC conditions. Provide an explanation for why each was not covered (they can be grouped into common reasons).
3. Summarize the compilation, synthesis, and place and route warnings.
4. What type of encoding is used for each of the finite state machines?
5. How many flip-flops are implemented in the design (hint: use the area report for help)? Does this number correspond to the number of flip-flops you would expect the synthesizer to infer from the RTL (hint: search for " 'event" in the RTL)?
6. What is the maximum clock frequency of the design?
7. Describe the critical path in the Verilog RTL (identify the file and lines that are part of the path).
8. Correlate the critical path delay to the maximum clock frequency reported by the tool (i.e. add up the delay for each of the critical path elements, invert, and compare the maximum clock frequency). Account for any discrepancies.
9. Which of the following approaches will allow the design to run faster? Explain why or why not (some you can try), and the drawbacks for each approach.
 - a. Register balancing (see Design Strategy in ISE)
 - b. Change a different speed grade
 - c. Optimization Goal (see Design Strategy in ISE)