ECE 128 – Synopsys Tutorial: Using DFT Compiler & TetraMax
Created at GWU by Thomas Farmer
Updated at GWU by William Gibb, Spring 2011

Objectives:

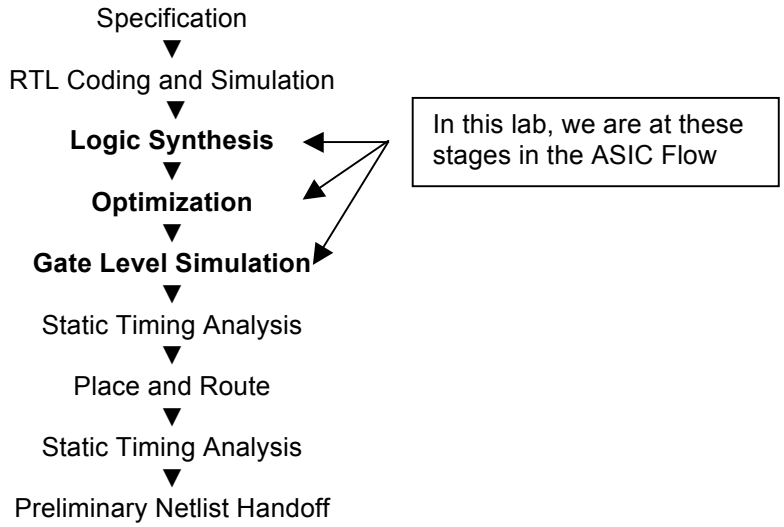
- Use Synopsys Design Compiler's DFT, synthesize 'scan-cell' test structures into verilog code
- Use Synopsys TetraMax tool, to generate test patterns (ATPG) for the 'test structures' from the Design Compiler's output
- Use Verilog to test the output of the TetraMax tool's patterns against the synthesized synopsys code

Assumptions:

- Student has completed lab 4
- Student a basic understanding of scan-chain theory.

Introduction:

The ASIC design flow is as follows:



In lab 3 we introduced the Synopsys Logic Synthesis tool called the Design Compiler. In this lab we will use the Design Compiler to insert test structures into our synthesized verilog code. We will then use the Synopsys TetraMAX® tool to exercise these test structures with a pattern of 1's and 0's. Prior to using these tools, you should be familiar with the basics of common testing methodologies used throughout the industry:

From lecture, you've learned that circuit testing falls into two main categories: functional testing & manufacturing testing:

1. ***Functional testing*** verifies that a circuit performs as it was intended to perform. For example, assume you have designed an adder circuit. Functional testing verifies that this circuit performs the addition function and computes the correct results over the range of values tested. However, exhaustive testing of all possible input combinations grows exponentially as the number of inputs increases. To maintain a reasonable test time, you must focus functional test patterns on the general function and corner cases.

2. ***Manufacturing testing*** verifies that the circuit does not have manufacturing defects by focusing on circuit structure rather than functional behavior. Manufacturing defects include problems such as Power or ground shorts, Open interconnect on the die due to dust particles, Short-circuited source or drain on the transistor due to metal spike-through, etc. Manufacturing defects might remain undetected by functional testing yet cause undesirable behavior during circuit operation.

To test for possible Manufacturing Faults, we use "fault models" to organize ways of detecting these defects:

Fault Models: When a ***manufacturing defect*** occurs, the *physical defect* has *a logical effect* on the circuit behavior. An open connection can appear to float either high or low, depending on the technology. A signal shorted to power appears to be permanently high. A signal shorted to ground appears to be permanently low. Many manufacturing defects can be represented using the stuck-at fault model. Synopsys TetraMAX® actually provides more fault models than the traditional stuck-at fault. Those fault models are transition, path delay, IDDQ, and bridging.

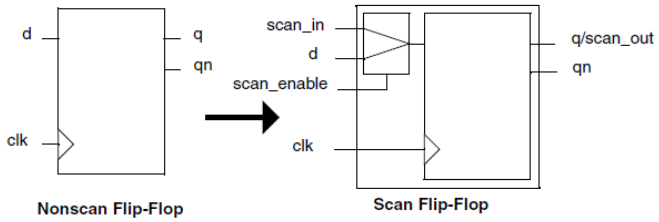
1) ***Stuck-at Fault Models***: The stuck-at-0 model represents a signal that is permanently low regardless of the other signals that normally control the node. The stuck-at-1 model represents a signal that is permanently high regardless of the other signals that normally control the node. For example, assume that you have a two-input AND gate that has a stuck-at-0 fault on the output pin. Regardless of the logic level of the two inputs, the output is always 0.
   a) Preconditions for detecting Stuck-at Faults: The node of a stuck-at fault must be **controllable** and **observable** for the fault to be detected.
      i) A node is **controllable** if you can drive it to a specified logic value by setting the primary inputs to specific values. A primary input is an input that can be directly controlled in the test environment.
      ii) A node is **observable** if you can predict the response on it and propagate the fault effect to the primary outputs where you can measure the response. A primary output is an output that can be directly observed in the test environment.
   b) How to detect a stuck-at fault on a target node:
      i) Control the target node to the opposite of the stuck-at value by applying data at the primary inputs.
      ii) Make the node's fault effect observable by controlling the value at all other nodes affecting the output response, so the targeted node is the active (controlling) node. This is known as path sensitization.
      iii) The set of logic 0s and 1s applied to the primary inputs of a design is called the input stimulus. The resulting values at the primary outputs, assuming a fault-free design, are called the expected response. The actual values measured at the primary outputs are called the output response. If the output response does not match the expected response for a given input stimulus, the input stimulus has detected the fault.
         (1) For example, if the faulty node is stuck-at-0, you need to apply input stimulus that forces that node to 1. For a two-input AND gate whose output node is stuck-at-0, apply a logic 1 at both inputs. The expected response for this input stimulus is logic 1, but the output response is logic 0. This input stimulus detects the stuck-at-0 fault.
         (2) This method of determining the input stimulus to detect a fault uses the single stuck-at fault model. The single stuck-at fault model assumes that only one node is faulty and that all other nodes in the circuit are good. The single stuck-at fault model greatly reduces the complexity of fault modeling and is technology independent, enabling the use of algorithmic pattern generation techniques.
2) ***Transition Fault Model***: The transition delay fault model is used to generate test patterns to detect single-node slow-to-rise and slow-to-fall faults. For this model, TetraMAX launches a logical transition upon completion of a scan load operation and uses a capture clock procedure to observe the transition results.
3) ***Path Delay Fault Model***: The path delay fault model tests and characterizes critical timing paths in a design. Path delay fault tests exercise the critical paths at-speed (the full operating speed of the chip) to detect whether the path is too slow because of manufacturing defects or variations.
4) ***IDDQ Fault Model***: The IDDQ fault model assumes that a circuit defect will cause excessive current drain due to an internal short circuit from a node to ground or to a power supply. For this model, TetraMAX does not attempt to observe the logical results at the device outputs. Instead, it tries to toggle as many nodes as possible into both states while avoiding conditions that violate quiescence, so that defects can be detected by the excessive current drain that they cause.
5) ***Bridging Fault Model***: The bridging fault model tests for shorts between two normally unconnected instance pins or net names. This model reports candidate defects as types bridging fault at 0 (ba0) or bridging fault at 1 (ba1), and victim and aggressor node sets.

In this lab, we will use Synopsys tools to implement the *Stuck-At Fault model* with Verilog code. But if you'd like to understand in greater detail about the concepts discussed above, read pages: "A-1 through A-14" in the TetraMax User Guide posted on the VLSI website.

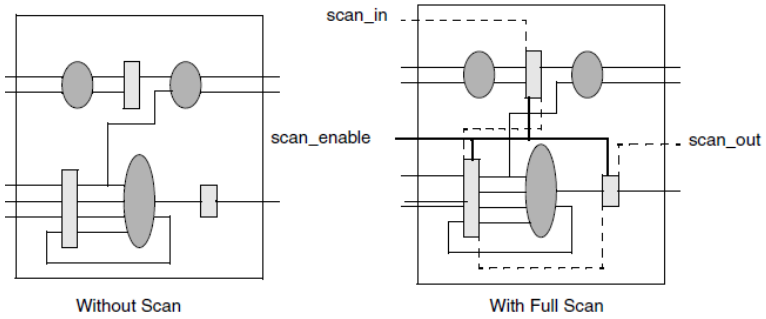
What is design-for-test (DFT) and what is a scan-chain?

In ATPG tutorial we previously ran, our full adder had no sequential components (no flip/flops or latches, i.e. –no state) and was a very small design in terms of gate count. Once a design becomes extremely large (many thousands of gates), gates may no longer be 'observable' or 'controllable' from a primary input and output pins. So the "stuck-at" model will not work. A technique known as 'design for test' has been developed in the industry to work around this problem and extend the "stuck-at" fault model.

If a design contains sequential components (like flip/flops), the design compiler can be used to replace these components with what is known as 'scan-cells.' A common example is replacing a d-flop-flop with a multiplexed-flip-flop (a flip flop with a MUX in front of it). A multiplexed input allows us to load test data into a flip/flop as opposed the regular data, simply by using the 'select line' of the mux. When the select is low, the normal data that was meant to go into the flip-flop passes through to Q, but when the select is high, test data propagates through the flip-flop to the Q output.



By replacing all of the flip-flops in a large design (assuming there are flip-flops throughout the design) with multiplexed flip-flops (scan-cells), we can increase the observability/controllability of all the non-sequential logic throughout the chip. The 'test data-input' and the 'select' lines of the scan-cells are wired to input pins; this creates what is known as a scan-chain. Now the user can put the chip into 'scan-mode' and load data into all the scan-cells. One can think of all the flip-flops in the design as having been stitched together into a giant 'shift-register' allowing data to be inputted in at 1 pin (serially) and being able to move test data all over the chip, making the non-sequential portions of the design more observable/controllable then they were before.



Overview of this Design For Test Tutorial:

We will use Synopsys Design Vision to automatically insert multiplexed flip/flops into our synthesized design and to interconnect them into scan-chains. Then we will use TetraMax to generate the patterns that will test the scan-chains in our design. Finally we will export the patterns to a verilog test bench and use it to test the scan-chains inserted into our synthesized design. If you wish to understand in greater depth the idea of a scan-chain (including multiple scan-chains), see the TetraMax user's guide, pages A7-A11.

Part I: File Setup for This Lab

Any time you wish to "synthesize" some verilog code, create a directory in your ece128 folder to house all of the files that will be created during the synthesis process. Note: the contents of all the files discussed below are listing in the appendix (for non-GW students viewing this on the web):

1. Login to a workstation, open up a terminal window and type:

```
cd ece128
mkdir lab5
cd lab5

mkdir reports
mkdir work
mkdir src
mkdir db
```

Always create the "*reports*" "*work*" "*src*" and "*db*" directories in whatever directory you decide to work under. In our case, our 'working' directory will be 'lab4'

- Copy the AMI 0.5 "standard cell library's verilog code into your "src" directory:

```
cp /apps/design_kits/osu_stdcells_v2p7/cadence/lib/ami05/lib/osu05_stdcells.v  ~/ece128/lab5/src
```

- Copy the OSU scan cell library's verilog code into your "src" directory:

```
cp ~vlsi/course_ece128/lab_files/lab5/src/osu_scan.v  ~/ece128/lab5/src
```

- Copy the OSU scan cell library's Synopsys database into you "db" directory:

```
cp ~vlsi/course_ece128/lab_files/lab5/db/osu_scan.db  ~/ece128/lab5/db
```

2. Copy the verilog code you wish to synthesize into the "**src**" subdirectory:

- In this lab, we want to use a modified ripplecarry adder from the ATPG tutorial:

- We will use a 'master' copy of this code, so that the instructions and names for modules in this lab match up. So you will copy the 'fulladder' from the VLSI account, instead of using your own for this lab.

```
cp ~vlsi/course_ece128/lab_files/lab5/src/fulladder.v ~/ece128/lab5/src
cp ~vlsi/course_ece128/lab_files/lab5/src/halfadder.v ~/ece128/lab5/src
cp ~vlsi/course_ece128/lab_files/lab5/src/fulladder_tb.v ~/ece128/lab5/src
```

- The above three lines copy the "fulladder" "halfadder" and the test bench we created in lab1 into 'src' directory underneath the lab4 directory you created in step1.

```
cp ~vlsi/course_ece128/lab_files/lab5/src/ripplecarry4_clk.v ~/ece128/lab5/src
```

- This ripplecarry adder is 4-bits. It uses 4 fulladder's chained together. The only difference is that in-between the fulladders, there is a clocked d-flip/flop. While this has no practical application, it is a quick example of a circuit with 'sequential' logic, which we need for this tutorial.
    - This ripplecarry adder is **different** from the rippler carry adder used in the ATPG tutorial. It has an active high reset added to it. **This design flow will not work with designs that have asynchronous sets or resets.**

3. Copy synthesis scripts into your lab 4 directory:

```
cp ~vlsi/course_ece128/lab_files/lab5/dc_syn.tcl ~/ece128/lab5
cp ~vlsi/course_ece128/lab_files/lab5/dc_test.tcl ~/ece128/lab5
cp ~vlsi/course_ece128/lab_files/lab5/tmax_atpg.tcl ~/ece128/lab5
```

Part II Synthesizing the Ripple Counter (with registers) using Synopsys Design Vision:

1. Change to your working directory

   ```
   cd ~/ece128/lab5
   ```

2. Start the design compiler's GUI by typing

   ```
   design_vision
   ```
   (note: do NOT put an "&" after this command, it needs to run in the foreground)

3. Run the design compiler **script** to synthesize your code automatically into AMI .5 technology:

   From the menu, choose: File->Execute Script

   Browse for the file named: **dc_syn.tcl** *(it should be in the lab5 directory)*

   What this script does is all of the things we did in lab3 automatically:
   - It analyzes and elaborates your verilog code
   - It sets up a 'reference clock of 25MHz' and sets up *constraints* on the input and output pins
   - It then compiles the design with the *constraints* and synthesizes your design using AMI .5 gates
   - It saves the synthesized design in verilog code format
   - It writes out reports discussing the progress of the synthesis

   Design Compiler can also be run via the command line. To execute the script via the command line, execute the following command from ~/ece128/lab5

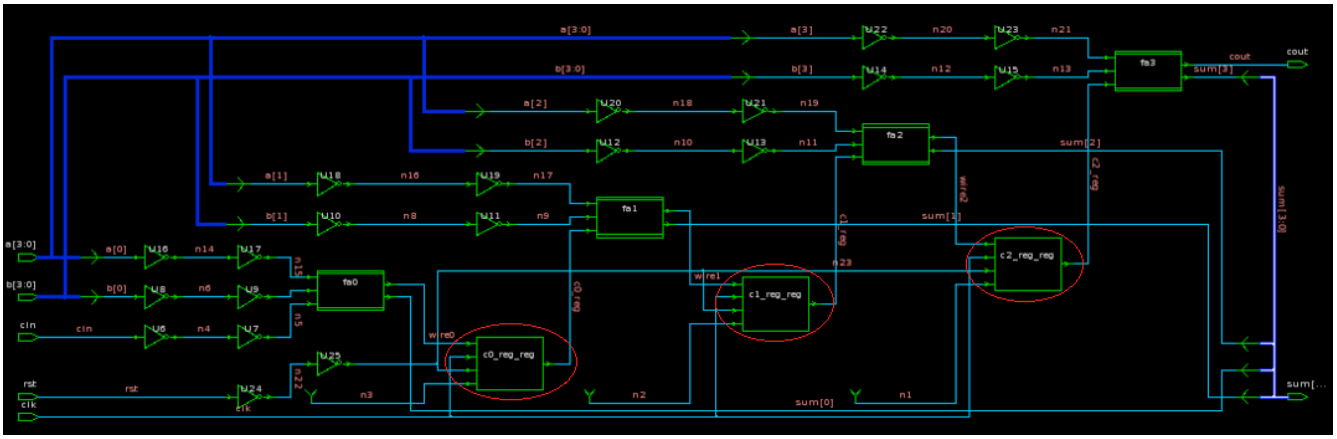
   ```
   Dc_shell-t -f dc_syn.tcl
   ```

   To start Design Vision and have it automatically load the script, execute the following command from ~/ece128/lab5

   ```
   Design_vision -f dc_syn.tcl
   ```

4. Check the error log, ensure that no errors are present before continuing

5. Open up the schematic window, the 4-bit ripple carry adder should be synthesized using AMI .5 standard gates. Verify that there is a DFF inbetween each fulladder circuit. **Do not exist Design Vision.**

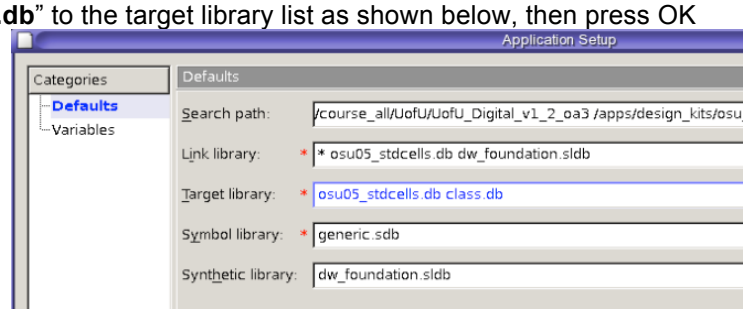


6. To show that you've completed this section of the lab, fill in the answers below regarding the ripplecarry adder, print this page, and hand it in to your GTA before next lab:

   - How many inputs combinations are possible for our ripplecarry adder? _____
   - How many internal states can our ripplecarry adder have? _____
   - What was the total dynamic power estimated for the ripplecarry adder? _____

PART III Inserting the Test Structures:

1. Change your setup:
   - From the menu, choose: File->Setup
   - Add "**osu_scan.db**" to the target library list as shown below, then press OK



2. Copy & Paste the following commands into the Design_Vision Prompt in order to setup variables for scan chain insertion:

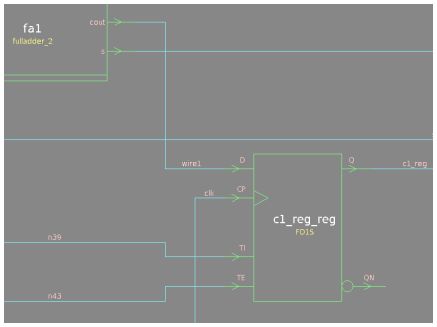
```
# Setup variables for testing
set dft_runname scan             ; # name appended to output files
set scan_library [list osu_scan.db] ; # Library with scan chain cells
set scancell    DFFPOSX1_SCAN ; # Name of ScanFF Cell

# Setup timing variables for dft_drc command

set test_default_delay 0      ; # define time when values are applied to input ports
set test_default_bidir_delay 0 ; # Defines  the  default switching time of bidirectional
                                 # ports in a tester cycle.
set test_default_strobe 40    ; # default strobe time in a test cycle for output ports
                                  # and bidirectional ports in output mode
set test_default_period 100   ; # Defines the default length of a test vector cycle

# Setup scan chain for insert_dft

set test_default_scan_style multiplexed_flip_flop;
# Defines  the  default scan style for the insert_dft command.
# type "man test_default_scan_style" for more information
```

   - The commands above setup some timing information for the design for test (DFT)

3. Copy & Paste the following commands into the Design_Vision Prompt in order to insert test structures into your design:

```
# Update filebase
set filebase [format "%s%s" [format "%s%s" $basename "_"] $dft_runname]
# Update target library
set target_library [list $target_library $scan_library]
# Set the scan cells to use in the design
#set_scan_register_type -type {DFFPOSX1_SCAN} ;
set_scan_register_type -type ${scancell} ;
# Make sure to add a test_out port
set_scan_configuration -create_dedicated_scan_out_ports true
# Infer clock and reset lines
create_test_protocol -infer_async -infer_clock
dft_drc -verbose
# Replace flip flops with multiplexed flipflops
compile -scan
# Check for constraint violations
report_constraint -all_violators
```

   - If the 'design violation browser' appears, simply close that window.
   - The 'compile -scan' command replaced all the DFF's in your ripple-carry adder with Multiplexed DFF's.
   - Open up the schematic window and see what has happened to all of your DFFs, notice the extra ports on all the DFFs:



4. Copy & Paste the following commands into the Design_Vision Prompt in order to Build a Scan Chains

   - At this point, we have all of the scan-cells inserted into our design, but they are not wired together into a scan-chain. You can see that the “TE” or ‘test-enable’ pin and ‘TI” or ‘test-input’ pins are grounded. Use the following commands to build the scan chain.

```
# connects all scan-enabled ff's together into scan-chain
# note, it creates two new ports: test_si & test_se
insert_dft
# set drive strength of the test ports to 2 (so it isn't assumed to be infinite)
set_drive 2 test_si
set_drive 2 test_se
# since you've already inserted scan-ff's, we don't want that to happen again,
# when we run insert_dft
set_scan_configuration -replace false
# run insert_scan again to set drive-strength constraints
insert_dft
# report any constraints that may have been violated by inserting the test
# structures
report_constraint -all_violators
dft_drc -verbose -coverage_estimate
report_scan_path -view existing -chain all
report_cell
```

5. Copy & Paste the following commands into Design Vision Prompt in order to save reports, test protocol and DFT verilog design

```
# report dft_drc
set filename [format "%s%s%s" ./reports/ $filebase ".violators"]
redirect $filename { report_constraint -all_violators }
# report dft_drc
set filename [format "%s%s%s" ./reports/ $filebase ".dft_drc"]
redirect $filename { dft_drc -verbose -coverage_estimate }
# report scan path
set filename [format "%s%s%s" ./reports/ $filebase ".scan_path"]
redirect $filename { report_scan_path -view existing -chain all }
# report cells
set filename [format "%s%s%s" ./reports/ $filebase ".cell"]
redirect $filename { report_cell }
# Write out protocol
set filename [format "%s%s%s" ./src/ $filebase ".spf"]
write_test_protocol -output $filename
# Write out scan chain design
set filename [format "%s%s%s" ./src/ $filebase ".v"]
redirect change_names { change_names -rules verilog -hierarchy -verbose }
write -format verilog -hierarchy -output $filename
```

- close the violation browser
- check for errors, if none, close Design Vision

6. Inspect the reports

   - In the 'reports' directory, four new reports: ripplecarry4_clk_scan.violators, ripplecarry4_clk_scan.dft_drc, ripplecarry4_clk_scan.scan_path, and ripplecarry4_clk_scan.cell have been created. Check them to see if any errors have occurred.
   - A new file, ripplecarry4_clk_scan.spf in the src directory has been created. This is a scan chain test protocol file, generated by DFT Compiler. We will use this with Tetramax.
   - A new verilog file has been created in the "src" directory, called: "ripplecarry4_clk_scan.v"
   - It contains our ripple_carry_adder synthesized into Generic gates, but with a scan-chain inserted into it
   - We will now use this synthesized code in TetraMax to generate patterns for it

7. To show that you've completed this section of the lab, fill in the answers below regarding the ripplecarry adder, print this page, and hand it in to your GTA before next lab:

   - What command is used to replace your DFF's with Scan-enabled DFF's? _____
   - What command is used to connected your scan-DFF's into a scan chain? _____
   - What different types of DFT strategies can DFT_Compiler insert? What are the differences between them? Hint: Look at the DC_Test.tcl script as a starting point for information.

     _____

     _____

     _____

     _____

     _____

     _____

   - What is the estimated test coverage for this design? _____
   - Can you simulate this design with the OSU Standard cell library now? Why or why not?

     _____

     _____

     _____

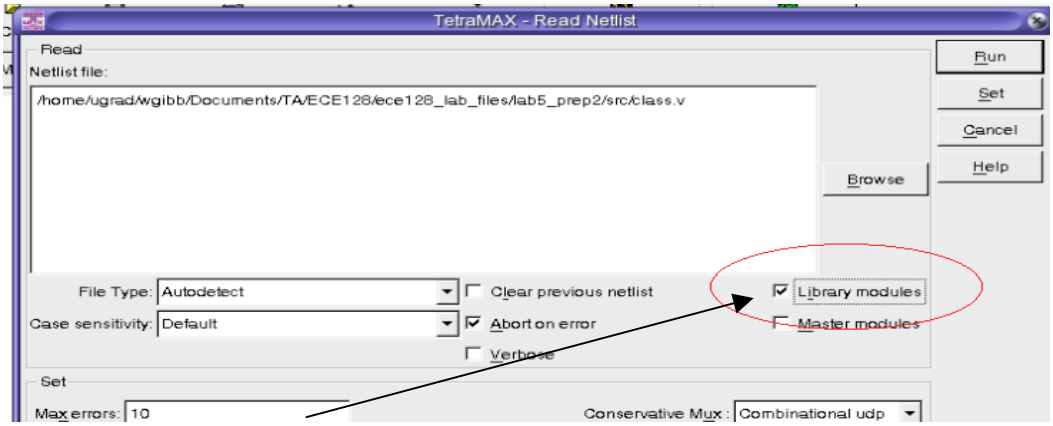
Creating Test Patterns for the Ripple Carry Adder using Synopsys TetraMAX:

1. Change to your working directory:
   `cd ~/ece128/lab5`

2. Start the Synopsys TetraMAX GUI by typing "
   `tmax64`
   `gui_start`

3. **Load the OSU standard cell library & Scan Cell library**

   - Click on the "**NETLIST**" button at the top of the screen
   - Browse for file: **osu05_stdcells.v** and **osu_scan.v** (in your src directory). **The osu_scan.v contains the muxed-FF's Verilog models.**

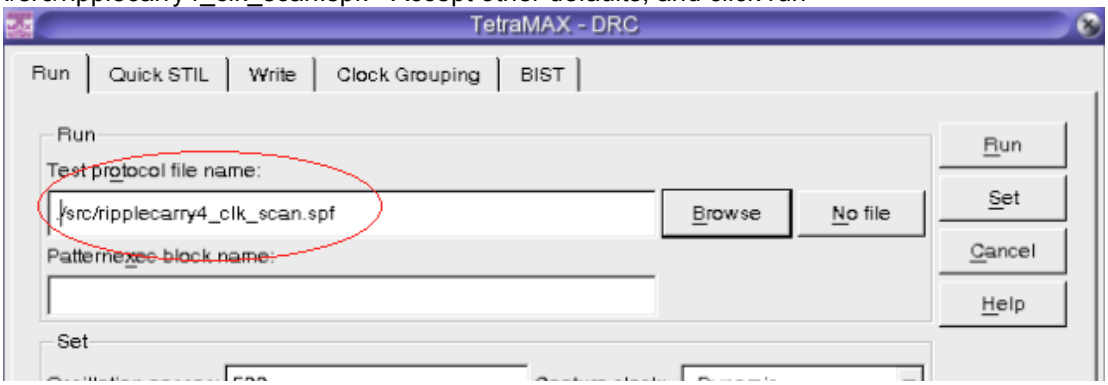


   - Click on the "Library Modules" check box, to indicate that these are libraries
   - Click on RUN

4. Load your Synthesized full adder:
   - Again click on the "**NETLIST**" button at the top of the screen
   - Browse for the file: ripplecarry4_clk_**scan**.v (in your src directory)
   - ***UNCHECK*** the "Library modules" check box, as this is not a library file, it is your design
   - Click on RUN

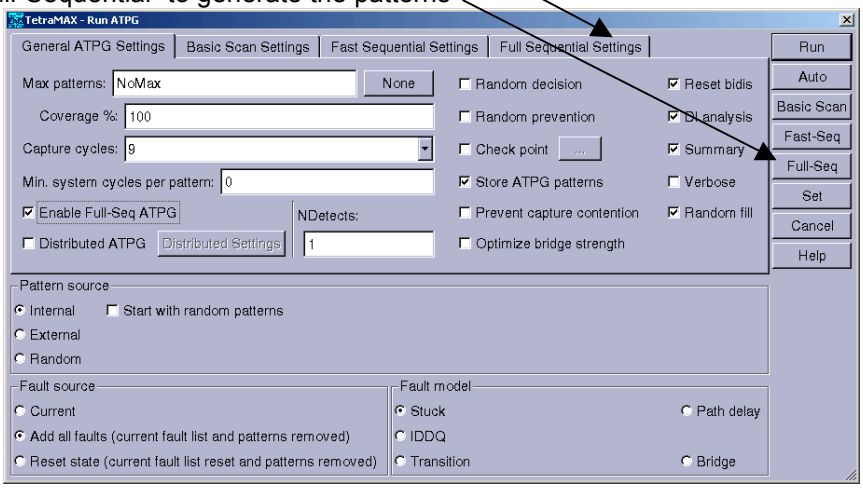
5. **BUILD** the Ripple Carry Adder
   - Click on the "**BUILD**" button at the top of the screen, accept defaults, and click run

6. **DRC** the Full Adder:
   - Click on the "**DRC**" button at the top of the screen
   - You'll need to add the Test Protocol file here. Under "Test protocol file name" enter: ./src/ripplecarry4_clk_scan.spf. Accept other defaults, and click run



Generate the Test Patterns for the Full Adder:

- Click on the “ATPG” button at the top of the screen (stands for ‘automatic test pattern generation’)
- Check the ‘add all faults’ check box at the bottom of the window
- Ensure the ‘stuck’ at fault model is checked
- Click on “Enable Full Seq ATPG”
- Change the capture cycles to 9
- Next, click on the Full Sequential Settings Tab
  - Set the merge effort = low
- Click on “Full-Sequential” to generate the patterns



- Read the generated report & notice the # of faults possible, % of coverage:

7. View the patterns TetraMax has generated (with the ATPG sequential scan algorithm)
   - Use the same technique as you did in the last tutorial to show the faults

8. View the results of a possible stuck-at-fault:

9. Write out the test patterns to a verilog file
   - Click on the “Write Pat.” Button on the top of the TetraMAX window
   - For the **Pattern File Name** type: `./src/ripplecarry4_clk_tb_patterns.v`
   - For the **File Format**, select: Verilog-Single File
   - Press OK

10. Viewing the Patterns in Simvision
    - Open up an editor (like gedit) and edit the file: `ripplecarry4_clk_tb_patterns.v`
    - Go to the very bottom of the code
    - Right before the “end module” statement, add the following lines of verilog:

```
initial begin
   $shm_open ("ripplecarry4_clk_scan_patterns.db") ;
   $shm_probe("AS") ;
end
```

11. Test the patterns out against your Synthesized full adder
    - Close TetraMax
    - Now, on a local workstation type:

```
cd ~/ece128/lab5/src
sim-nc ripplecarry4_clk_tb_patterns.v ripplecarry4_clk_scan.v osu_scan.v osu05_stdcells.v
```

12. To show that you've completed this section of the lab, fill in the answers below regarding the ripple carry adder, print this page, and hand it in to your GTA before next lab:
    - How many inputs combinations are possible for a ripple carry adder? _____
    - How many input combinations did TetraMAX create? _____
    - What % coverage did TetraMax obtain with the generated patterns? _____
        - Was this coverage the same coverage DFT Compiler estimated? _____

You can now use the dc_scan.tcl script to automatically generate scan chain inserted designs. Like the dc_syn.tcl script, it does require parameters to be set prior to using it. Likewise, tmax_atpg.tcl can be used to automatically generate ATPG vectors and testbenches for a design. The tmax_atpg.tcl script will also write out reports to your /reports directory, for your reference.

References

- University of Minnesota ECE Department Web Publication: http://mountains.ece.umn.edu/~sobelman/courses/ee5327/Synlab6_S08.pdf
- Synopsys TetraMAX® ATPG User Guide, Version B-2008.09-SP2, December 2008
- Synopsys DFT Compiler User Guide: Scan, Version C-2009.06-SP4, December 2009

Appendix:

For students viewing this tutorial who do not have access to our server, this is a listing of the contents of the files reference in Part II of this tutorial:

- **dc_syn.tcl** - Synthesis script for Design Compiler. Omitted from this tutorial. See ATPG tutorial for dc_syn.tcl.
- **dc_test.tcl** - Synthesis script for DFT Compiler. Goes through the regular synthesis process and then performs DFT Scan-chain insertion.
  - Note the additional section in the settings, DFT Switches. These are the commands used to drive DFT Compiler.
- **Tmax_atpg.tcl** - TCL script to drive Tetramax with tutorial commands.
- **ripplecarry4_clk.v** - Example design used in tutorial.

The Synopsys generic standard cell library can be used in place of the osu_scan library; however the OSU_scan library will be made available from the GWU VLSI site in the future.

dc_test.tcl:

```
#################################################################
#### Design Compiler Script for ECE 128
#### Performs Synthesis only to AMI .5 technology
#### author: wgibb
#### note: this is a TCL script
#### modified from work done by tjf and eb
#################################################################

###################################
# ITEMS YOU WILL NEED TO SET FOR EACH DESIGN
# 1) myFiles - LIST OF YOUR FILES TO SYNTHESIZE
# 2) basename - TOP LEVEL MODULE IN YOUR DESIGN
# 3) myClk - NAME OF YOUR CLOCK SIGNAL
# 4) virtual - USE A REAL CLOCK (SEQUENTIAL DESIGNS) OR A VIRTUAL
#              CLOCK (COMBINATORIAL DESIGNS)
# 5) myPeriod - SETS THE CLOCK SPEED, THUS DEFINING THE SYNTHESIS SPEED GOAL
###################################

# list of all HDL files in the design
set myFiles [list ./src/ripplecarry4_clk.v ./src/fulladder.v ./src/halfadder.v] ;
set basename ripplecarry4_clk       ;# Top-level module name
set myClk clk                     ;# The name of your clock
set virtual 0                       ;# 1 if virtual clock, 0 if real clock
set myPeriod_ns 40                ;# desired clock period (in ns) (sets speed goal)

###################################
# Some runtime options, change only if needed
set runname syn                 ;# Name appended to output files
set exit_dc 0                  ;# 1 to exit DC after running, 0 to keep DC running
# set the target library
set target_library [list osu05_stdcells.db] ;
###################################

###################################
# Control the printing of result files
###################################
set verbose 0                  ;# 1 Write reports to screen, 0 do not write reports to screen
set verbose_dft 0              ;# 1 Write reports to screen, 0 do not write reports to screen
###################################
# Timing and loading information
###################################
```

```
set myClkLatency_ns 0.3              ; # clock network latency
set myInDelay_ns 2.0               ;# delay from clock to inputs valid
set myOutDelay_ns 1.65              ;# delay from clock to output valid
set myInputBuf INVX1           ;# name of cell driving the inputs
set myLoadLibrary [file rootname $target_library]    ;# name of library the cell comes from
set myLoadPin A             ;# name of pin that the outputs drive
set myMaxFanout       1      ;# max fanout load for input pins
set myOutputLoad 0.1  ;# output pin loading

##################
# compiler switches...
##################
set optimizeArea 1                ;# 1 for area, 0 for speed
set useUltra 0                      ;# 1 for compile_ultra, 0 for compile
                                     # mapEffort, useUngroup are for
                                     # non-ultra compile...
set useUngroup 0                    ;# 0 if no flatten, 1 if flatten


###################################
# DFT Switches                    #
###################################
set dft_runname scan             ; # name appended to output files
set scan_library [list osu_scan.db] ; # Library with scan chain cells
set scancell   DFFPOSX1_SCAN ; # Name of ScanFF Cell

# Setup timing variables for dft_drc command

set test_default_delay 0     ; # define time when values are applied to input ports
set test_default_bidir_delay 0       ; # Defines  the  default switching time of bidirectional
                                     # ports in a tester cycle.
set test_default_strobe 40    ; # default strobe time in a test cycle for output ports
                                     # and bidirectional ports in output mode
set test_default_period 100   ; # Defines the default length of a test vector cycle

# Setup scan chain for insert_dft

set test_default_scan_style multiplexed_flip_flop;
# Defines  the  default scan style for the insert_dft command.
# type "man test_default_scan_style" for more information

###################################
# Set some system-level things that RARELY change...
###################################
# synthetic_library is set in .synopsys_dc.setup to be
# the dw_foundation library.
set link_library [concat  [concat  "*" $target_library] $synthetic_library]
###################################
set fileFormat verilog              ;# verilog or VHDL

#################################################################
#################################################################
### YOU SHOULD NOT NEED TO CHANGE ANYTHING BELOW THIS LINE ###
#################################################################
#################################################################

#################################################################
#### read in, link to standard cells, and uniquify design ####
#################################################################

###################################
# remove any other designs from design compiler's memory
###################################
remove_design -all

echo IMPORTING DESIGN
###################################
# analyzer & elaborate verilog source files
###################################
analyze -format $fileFormat -lib WORK $myFiles
elaborate $basename -lib WORK -update

###################################
# set design to 'highest' module level
###################################
```

```
current_design $basename

###################################
# link to standard cell libraries and uniquify
###################################
link
uniquify

#####################################################
#### setup clock & all input/output constraints ####
#####################################################
echo SETTING CONSTRAINTS

###################################
# now you can create clocks for the design
# and set other constraints
###################################
if {  $virtual == 0 } {
   create_clock -period $myPeriod_ns $myClk
} else {
   create_clock -period $myPeriod_ns -name $myClk
}
set_clock_latency $myClkLatency_ns $myClk
###################################
# set delays on all inputs & outputs with respect to the clock (in ns)
# set the input and output delay relative to myClk
###################################
if {  $virtual == 0 } {
    set_input_delay $myInDelay_ns -clock $myClk [all_inputs]
} else {
    set_input_delay $myInDelay_ns -clock $myClk [remove_from_collection [all_inputs] $myClk]
}
set_output_delay $myOutDelay_ns -clock $myClk [all_outputs]
###################################
# Set the driving cell for all inputs except the clock
# The clock has infinite drive by default. This is usually
# what you want for synthesis because you will use other
# tools (like SOC Encounter) to build the clock tree
# (or define it by hand).
###################################
if {  $virtual == 0 } {
    set_driving_cell  -library $myLoadLibrary -lib_cell $myInputBuf [all_inputs]
} else {
   set_driving_cell  -library $myLoadLibrary -lib_cell $myInputBuf [remove_from_collection [all_inputs]
$myClk]
}
###################################
# set load/fanin/fanout for all inputs/outputs
###################################
set_load $myOutputLoad [all_outputs]

###################################
# check value of fanout
###################################
set_max_fanout $myMaxFanout [all_inputs]
set_fanout_load 8 [all_outputs]

echo DONE SETTING CONSTRAINTS

###################################
# This command will fix the problem of having
# assign statements left in your structural file.
# But, it will insert pairs of inverters for feedthroughs!
set_fix_multiple_port_nets -all -buffer_constants
###################################

echo BEGIN COMPILING DESIGN
###################################
# optimize for area
###################################
if { $optimizeArea == 1} {
       set_max_area 0
}
###################################
# now compile the design with given mapping effort
# and do a second compile with incremental mapping
```

```
# or use the compile_ultra meta-command
####################################
if {  $useUltra == 1 } {
   compile_ultra
} else {
   if {  $useUngroup == 1 } {
        compile -ungroup_all -map_effort medium
  } else {
        compile -map_effort medium -exact_map
  }
}
check_design
echo VIOLATIONS
report_constraint -all_violators

#####################################################
#### generate verilog code for synthesized module ###
#### sdc files, sdf files, design compiler project###
#### and write out reports                      ###
#####################################################
echo OUTPUT FILES AND REPORTS
set filebase [format "%s%s" [format "%s%s" $basename "_"] $runname]

####################################
# structural (synthesized) file as verilog
####################################
set filename [format "%s%s%s" ./src/ $filebase ".v"]
redirect change_names { change_names -rules verilog -hierarchy -verbose }
write -format verilog -hierarchy -output $filename

####################################
# write out the sdf file for back-annotated verilog sim
# This file can be large!
####################################
set filename [format "%s%s%s" ./src/ $filebase ".sdf"]
write_sdf -version 1.0 $filename

####################################
# this is the timing constraints file generated from the
# conditions above - used in the place and route program
####################################
set filename [format "%s%s%s" ./src/ $filebase ".sdc"]
write_sdc $filename

####################################
# generate reports for user to view
####################################

if { $verbose == 1 } {
        report_design
        report_hierarchy
        report_timing -path full -delay max -nworst 3 -significant_digits 2 -sort_by group
        report_timing -path full -delay min -nworst 3 -significant_digits 2 -sort_by group
        report_area
        report_cell
        report_net
        report_port -v
        report_power -analysis_effort low
}

# Design and Hierarchy reports
set filename [format "%s%s%s" ./reports/ $filebase ".design"]
redirect $filename { report_design }
set filename [format "%s%s%s" ./reports/ $filebase ".design"]
redirect -append $filename { report_hierarchy }

# Timing reports
set filename [format "%s%s%s" ./reports/ $filebase ".timing"]
redirect $filename { report_timing -path full -delay max -nworst 5 -significant_digits 2 -sort_by group
}
set filename [format "%s%s%s" ./reports/ $filebase ".timing"]
redirect -append $filename { report_timing -path full -delay min -nworst 5 -significant_digits 2 -
sort_by group }

# Report_cell and report_area
set filename [format "%s%s%s" ./reports/ $filebase ".area"]
```

```
redirect $filename { report_area }
set filename [format "%s%s%s" ./reports/ $filebase ".area"]
redirect -append $filename { report_cell }

# Report port
set filename [format "%s%s%s" ./reports/ $filebase ".ports"]
redirect $filename { report_port -v}

#report net
set filename [format "%s%s%s" ./reports/ $filebase ".net"]
redirect $filename { report_net }

# report power
set filename [format "%s%s%s" ./reports/ $filebase ".pow"]
redirect $filename { report_power -analysis_effort low }

########################################
#### Insert Test Structures ###
########################################
# Update filebase
set filebase [format "%s%s" [format "%s%s" $basename "_"] $dft_runname]
# Update target library
set target_library [list $target_library $scan_library]
# Set the scan cells to use in the design
#set_scan_register_type -type {DFFPOSX1_SCAN} ;
set_scan_register_type -type ${scancell} ;

# Make sure to add a test_out port
set_scan_configuration -create_dedicated_scan_out_ports true

# Infer clock and reset lines
create_test_protocol -infer_async -infer_clock

dft_drc -verbose

# Replace flip flops with multiplexed flipflops
compile -scan

# Check for constraint violations
report_constraint -all_violators

########################################
### Building Scan Chains              ###
########################################

# connects all scan-enabled ff's together into scan-chain
# note, it creates two new ports: test_si & test_se
insert_dft

# set drive strength of the test ports to 2 (so it isn't assumed to be infinite)
set_drive 2 test_si
set_drive 2 test_se

# since you've already inserted scan-ff's, we don't want that to happen again,
# when we run insert_dft
set_scan_configuration -replace false

# run insert_scan again to set drive-strength constraints
insert_dft

# report any constraints that may have been violated by inserting the test
# structures

if { $verbose_dft == 1 } {
       report_constraint -all_violators
       dft_drc -verbose -coverage_estimate
       report_scan_path -view existing -chain all
       report_cell
}

# report dft_drc
set filename [format "%s%s%s" ./reports/ $filebase ".violators"]
redirect $filename { report_constraint -all_violators }

# report dft_drc
set filename [format "%s%s%s" ./reports/ $filebase ".dft_drc"]
```

```
redirect $filename { dft_drc -verbose -coverage_estimate }

# report scan path
set filename [format "%s%s%s" ./reports/ $filebase ".scan_path"]
redirect $filename { report_scan_path -view existing -chain all }

# report cells
set filename [format "%s%s%s" ./reports/ $filebase ".cell"]
redirect $filename { report_cell }

# Write out protocol
set filename [format "%s%s%s" ./src/ $filebase ".spf"]
write_test_protocol -output $filename

# Write out scan chain design
set filename [format "%s%s%s" ./src/ $filebase ".v"]
redirect change_names { change_names -rules verilog -hierarchy -verbose }
write -format verilog -hierarchy -output $filename

###################################
# this is the timing constraints file generated from the
# conditions above - used in the place and route program
###################################
set filename [format "%s%s%s" ./src/ $filebase ".sdc"]
write_sdc $filename

###################################
# quit dc
###################################
if { $exit_dc == 1} {
       exit
}
```

tmax_atpg.tcl:

```
#####################################################################
#### TetraMax Script for ECE 128
#### Performs ATPG Pattern Generation for Synopsys Generic files
#### author: tjf
#### update: wgibb, spring 2010
#### note: this script will only run in TMAX TCL mode
#### start tmax like this:   tmax -tcl
#####################################################################


#############################################################
#### local variables, designer must change these values ####
#############################################################

set top_module ripplecarry4_clk
set synthesized_files [list ./src/ripplecarry4_clk_scan.v]
set cell_lib ./src/osu05_stdcells.v
set scan_lib ./src/osu_scan.v
set stil_file [list ./src/ripplecarry4_clk_scan.spf]


################################################
#### read in standard cells and user's design ###
################################################

# remove any other designs from design compiler's memory
read_netlist -delete

# read in standard cell library
read_netlist $cell_lib -library

# read in scan cell library
read_netlist $scan_lib -library

# read in user's synthesized verilog code
read_netlist $synthesized_files


################################################
#### BUILD and DRC test model
```

```
################################################

run_build_model $top_module
# ignoring warnings like N20 or B10

# Set STIL file from DFT Compiler
set_drc $stil_file

# run check to see if synthesized code violates any testing rules
run_drc

################################################
#### Generate ATPG (patterns)- full sequential
################################################

# capture all faults, 9 capture cycles
set_atpg -capture_cycles 9 -full_seq_atpg
remove_faults -all
add_faults -all

# run atpg in full sequential mode
run_atpg full_sequential_only

# write out patterns (overwrite old files)
write_patterns ./src/${top_module}_tb_patterns.v -replace -internal -format verilog_single_file -
parallel 0

################################################
#### Output reports
################################################

report_patterns -all >> ./reports/${top_module}.tmax.patterns
report_violations -all >> ./reports/${top_module}.tmax.violations
report_faults -summary -collapsed >> ./reports/${top_module}.tmax.coverage

################################################
#### Analyze Faults
################################################

# up to user to run these commands, they can inspect the faults and various reasons for them:
#analyze_faults -class an
#analyze_faults -class an -verbose -max 3
#analyze_faults in_a_reg_reg/p_dregscan0/q -stuck 1
```

ripplecarry4_clk.v:

```
module ripplecarry4_clk (sum, cout, a, b, cin, clk, rst) ;
   input [3:0] a, b ;
   input cin, clk, rst ;
   output [3:0] sum ;
   output cout ;

   wire wire0, wire1, wire2 ;
   reg c0_reg, c1_reg, c2_reg ;

   fulladder fa0( .s(sum[0]), .cout(wire0), .a(a[0]), .b(b[0]), .cin(cin)    );
   fulladder fa1( .s(sum[1]), .cout(wire1), .a(a[1]), .b(b[1]), .cin(c0_reg) );
   fulladder fa2( .s(sum[2]), .cout(wire2), .a(a[2]), .b(b[2]), .cin(c1_reg) );
   fulladder fa3( .s(sum[3]), .cout(cout),  .a(a[3]), .b(b[3]), .cin(c2_reg) );

   always @(posedge clk)
   begin
        if(rst)
        begin
             c0_reg <= 0 ;
             c1_reg <= 0 ;
             c2_reg <= 0 ;
        end
        else
        begin
               c0_reg <= wire0 ;
               c1_reg <= wire1 ;
               c2_reg <= wire2 ;
        end
   end
```

```
endmodule
```