Prof. Jerry Wu (Rev. RE)

ECE 4150/6250 – Synopsys Tutorial: Using DFT & TetraMax

Objectives:

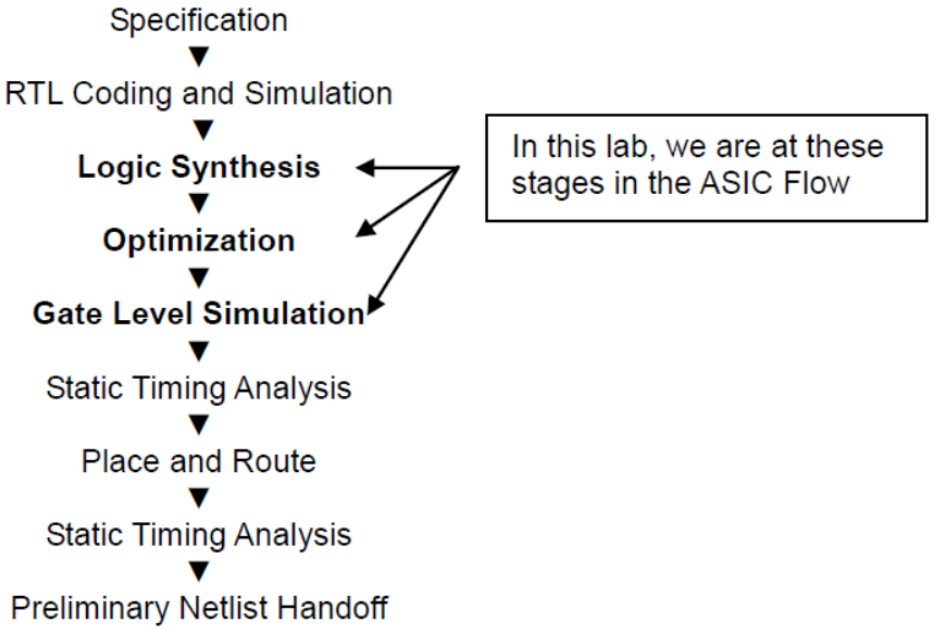
- Use Synopsys Design Compiler's to automatically synthesize designs via scripts.
- Use Synopsys TetraMax tool, to generate test patterns (ATPG) for the synthesized verilog from the Design Compiler's output.
- Use Verilog to test the output of the TetraMax tool's patterns against the synthesized synopsys code

Assumptions:

- Student has completed the lab: “Synthesizing full adder” and has a working structural full adder

Introduction:

The ASIC design flow is as follows:



In the “synthesizing full adder” lab 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:

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.

2) ***Transition Fault Model***: The transition delay fault model is used to generate test patterns to detect singlenode slow-to-rise and slow-to-fall faults.
3) ***Path Delay Fault Model***: The path delay fault model tests and characterizes critical timing paths in a design.
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.
5) ***Bridging Fault Model***: The bridging fault model tests for shorts between two normally unconnected instance pins or net names.

In this lab, we will use Synopsys tools to implement the *Stuck-At Fault model* with your Verilog code.

Note: "ece128" is an example; feel free to modify to your own directory name in the following example

Part II: File Setup for This Lab

Any time you wish to "synthesize" some verilog code, create a directory in your ece[your course number] 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:

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

```
cd ece128       //(change to your own name)
mkdir lab4
cd lab4
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/lab4/src
```

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

- In this lab, we want to use the fulladder & halfadder you created in lab1:

- 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 /home/seas/vlsi/course_ece128/lab_files/lab4/src/fulladder.v ~/ece128/lab4/src
cp /home/seas/vlsi/course_ece128/lab_files/lab4/src/halfadder.v ~/ece128/lab4/src
cp /home/seas/vlsi/course_ece128/lab_files/lab4/src/fulladder_tb.v ~/ece128/lab4/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.

3. Copy synthesis scripts into your lab 4 directory:

```
cp /home/seas/vlsi/course_ece128/lab_files/lab4/dc_syn.tcl ~/ece128/lab4
```

Part III: Creating Test Patterns for Purely Combinational Code:

Overview of Part III:

We will use synopsys design compiler, synopsys tetramax, and verilog to do the following:

1. Use Synopsys Design Compiler to synthesize the full adder to AMI .5 technology
2. Use Synopsys TetraMAX"s ATPG program to create „test patterns" to determine all „stuck-at-faults" in our synthesized AMI .5 full adder circuit
3. Use Verilog to run the 'test patterns' against our synthesized AMI .5 full adder circuit

Note: Before ever attempting to "synthesize" verilog code, you must ensure that it compiles properly (using the verilog simulator) and its waveforms are as you expect (using simvision). Only after the code passes the simulation phase can you move on to the synthesis phase of the ASIC design flow.

Synthesizing the Full Adder using Synopsys Design Compiler using a SCRIPT:

1. Change to your working directory

   ```
   cd ~/ece128/lab4
   ```

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

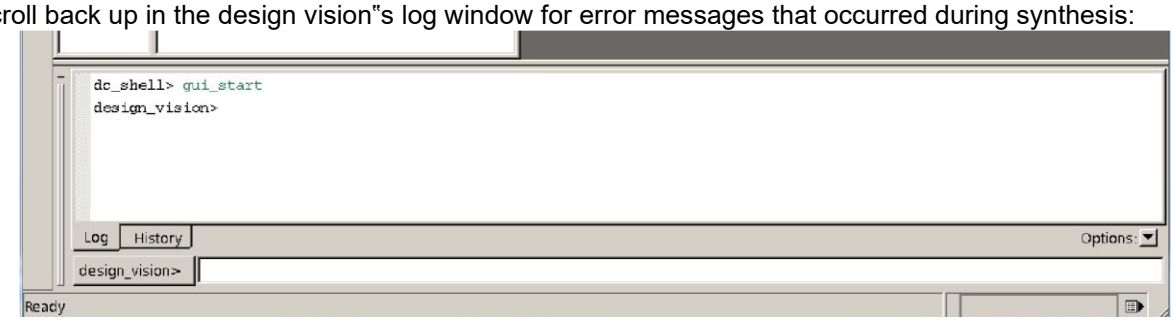
   Browse for the file named: **dc_syn.tcl** *(it should be in the lab4 directory)*

   What this script does is all of the things we did in lab2 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/lab4

   ```
   dc_shell -f dc_syn.tcl
   ```

4. Scroll back up in the design vision"s log window for error messages that occurred during synthesis:

   - If any errors occurred, you must go back and solve the problems before continuing to the next step *(perhaps it couldn"t find your verilog file, things like this will cause synthesis to fail)*

5. Once you have a successful synthesis, exit the Design Vision GUI

6. Inspect the „reports" generated during the synthesis

   - Go into the lab4/reports sub-directory and you will find a list of reports like this:

```
$ cd ../reports
$ ls -al
total 512
drwx------   2 wgibb    user        2.0K Feb 21 15:21 .
drwx------   6 wgibb    user       2.0K Feb 21  2010 ..
-rw-------   1 wgibb    user       2.0K Feb 21  2010 fulladder_syn.area
-rw-------   1 wgibb    user       1.4K Feb 21  2010 fulladder_syn.design
-rw-------   1 wgibb    user       1.7K Feb 21  2010 fulladder_syn.net
-rw-------   1 wgibb    user       3.5K Feb 21  2010 fulladder_syn.ports
-rw-------   1 wgibb    user       1.1K Feb 21  2010 fulladder_syn.pow
-rw-------   1 wgibb    user        16K Feb 21  2010 fulladder_syn.timing
```

   - Use a text editor (like gedit) to open up each report. As an example, this is what the „report.area" file should contain:

```
****************************************
Report : area
Design : fulladder
Version: B-2008.09-SP2
Date   : Mon Feb 21st 11:27:12 2010
****************************************

Library(s) Used:

    osu05_stdcells (File: /apps/design_kits/osu_stdcells_v2p7/synopsys/lib/ami05/osu05_stdcells.db)

Number of ports:                5
Number of nets:                14
Number of cells:                9
Number of references:           4

Combinational area:       2736.000000
Noncombinational area:       0.000000
Net Interconnect area:      undefined  (No wire load specified)

Total cell area:          2736.000000
Total area:                 undefined
```

   - This indicates the total area required to synthesize the full adder in AMI .5 is: 2736 μm^2
   - Look through each report (design, nets, ports, power, timing) to ensure that your design was properly synthesized and met the „constraints" (e.g. – like the clock speed) you setup during synthesis
   - If the design did not properly synthesize, you must fix these problems before moving onto the next step

7. To show that you"ve completed this section of the lab, from the reports, fill in the answers below regarding the full adder, print this page, and **post it to the BB (lecture lection) before next lab**:

   - How much area did each HALF ADDER consume? _____
   - Does the full adder"s total area (2736 μm^2) include the wiring ? Y/N
   - What temperature and voltage are used during the synthesis? _____
   - How many NETs are in the design? _____
   - What was the fanout load set on the cout port? _____
   - What was the total dynamic power estimated for the fulladder? _____
   - What was the actual time (not required) for data to pass from port A to port COUT? _____ *(hint, time cannot be „negative")*

The remainder of this page is intentionally left black.

Creating Test Patterns for COMBINATIONAL LOGIC circuits using Synopsys TetraMAX:

1. After a successful synthesis, the „dc_syn.tcl" script that you used in design vision, will have produced a synthesized version of your fulladder under the "src" directory, verify this file exists before beginning this section:

```
$ cd src
$ ls -al
total 640
-rw-------   1 wgibb    user         16K Feb 21 15:03 class.v
-rw-------   1 wgibb    user        1.1K Feb 21 15:23 fulladder_syn.sdc
-rw-------   1 wgibb    user        2.9K Feb 21 15:23 fulladder_syn.sdf
-rw-------   1 wgibb    user         800 Feb 21 15:23 fulladder_syn.v
-rwx------   1 wgibb    user         412 Feb 21 15:22 fulladder_tb.v
-rwx------   1 wgibb    user         243 Feb 21 15:22 fulladder.v
-rwx------   1 wgibb    user         100 Feb 21 15:22 halfadder.v
-rwx------   1 wgibb    user         23K Feb 21 15:02 osu05_stdcells.v
```

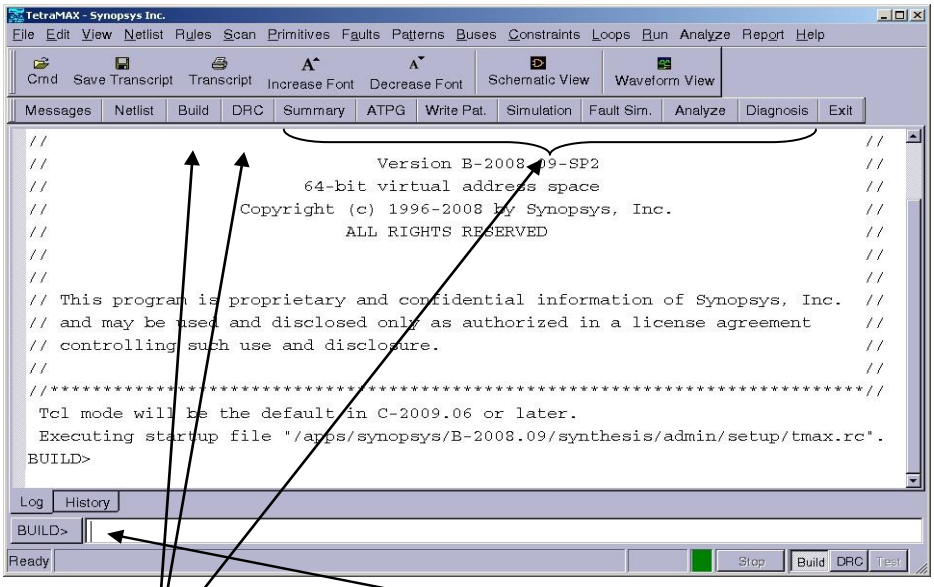
 You should see a file called: "fulladder_syn.v" under the "src" directory. If you do not see this file, your synthesis is incomplete. Verify that you did not make any mistakes or have any errors in the steps (1-7) above.

2. Change to your working directory:

```
cd ~/ece128/lab4
```

3. Start the Synopsys TetraMAX GUI by typing:

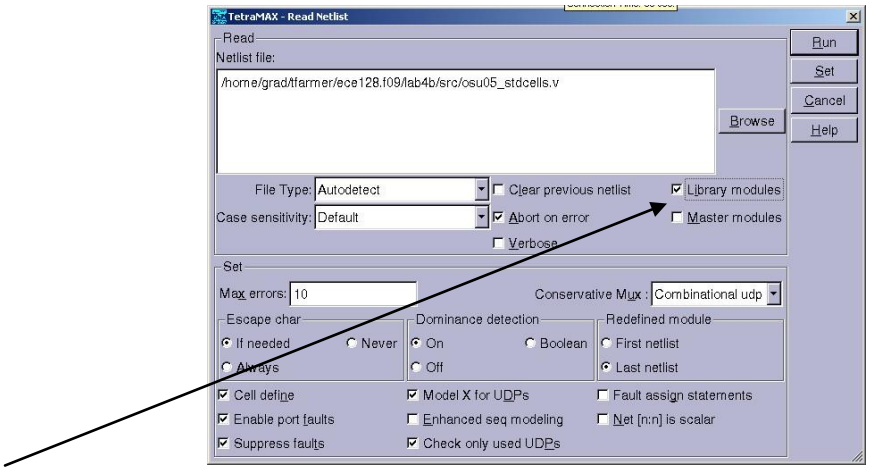
```
tmax &
```



4. Once the GUI opens, notice that you are in the "BUILD" mode. TetraMax has 3 modes: BUILD, DRC, and TEST. Notice that the buttons along the top of the GUI, follow this order. You cannot go from BUILD to TEST, you must go in the BUILD->DRC->TEST order as you will see in the next few steps.

5. Load the AMI .5 standard cell verilog library:

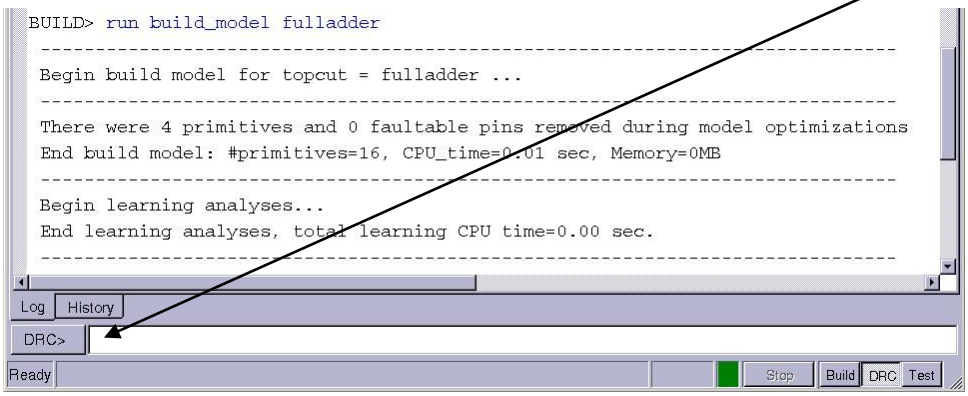
- Click on the "**NETLIST**" button at the top of the screen
- Browse for file: osu05_stdcells.v (in your src directory):



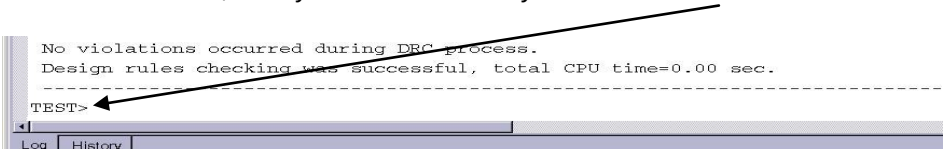
- Click on the “Library Modules” check box, to indicate that this is a „library"
- Click on RUN

6. Load your Synthesized full adder:
- Again click on the “**NETLIST**” button at the top of the screen
- Browse for the file: fulladder_syn.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

7. **BUILD** the Full Adder:
- Click on the “**BUILD**” button at the top of the screen
- Ensure that the „top module" in the dialog box, matches your top module name: fulladder
- Once you have a successful „build" notice that the prompt at the bottom switches to **DRC** mode:



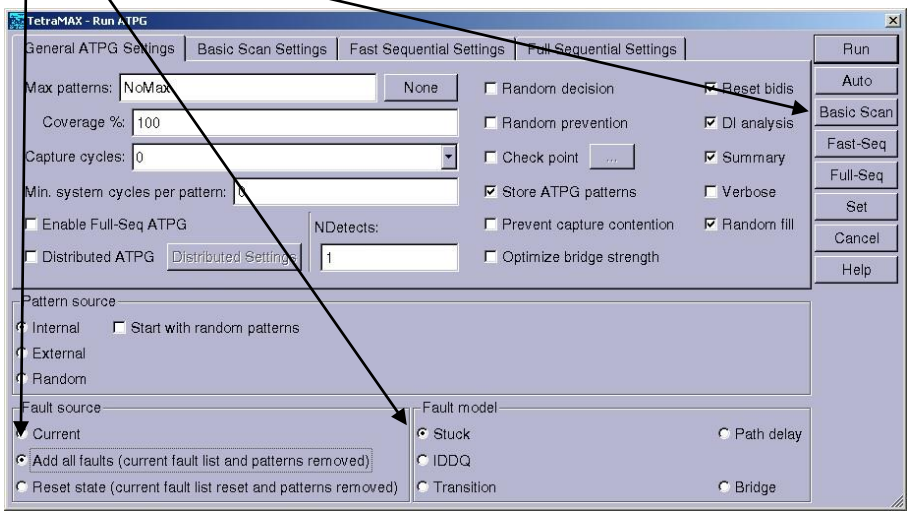
8. **DRC** the Full Adder:
- Click on the “**DRC**” button at the top of the screen
- Accept the defaults an click on the “RUN” button
- This Design Rule Check (DRC) is not the same as a „DRC" in cadence. It is checking to see if your design is in fact valid “Testable” code
- If there are NO DRC errors, then you will successfully be moved into **TEST** mode.



9. 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")

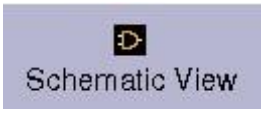
- Check the „add all faults" check box at the bottom of the window
- Ensure the „stuck" at fault model is checked
- Click the „Basic scan" button



- Read the generated report & notice three things:
  - There are **64** possible stuck-at-faults
  - **100 %** of the stuck-at-faults can be tested using the ATPG patterns TetraMax has made
  - TetraMax has generated **6** test patterns to test all of the possible stuck-at faults in your full adder

```
     Uncollapsed Stuck Fault Summary Report   --
-----------------------------------------------
fault class                     code    #faults
------------------------------  ----  ---------
 Detected                        DT          64
 Possibly detected               PT           0
 Undetectable                    UD           0
 ATPG untestable                 AU           0
 Not detected                    ND           0
-----------------------------------------------
total faults                                 64
test coverage                            100.00%
-----------------------------------------------
Pattern Summary Report
 ---------------------------------------------
 #internal patterns                           6
      #basic_scan patterns                    6
```

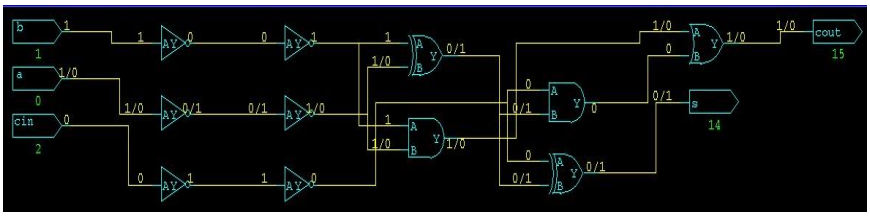
10. View the 6 patterns TetraMax has generated (with the ATPG basic scan algorithm)



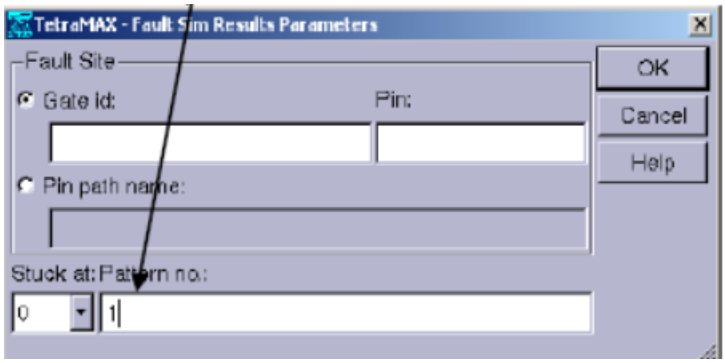
- Click on the „**Schematic View'** icon at the top of the screen
- Click on the „**Show'** icon that appears in the schematic view window, scroll down and select: **ALL**
- Click on the „**Setup'** ○ Set the Pin Data Type to: "PATTERN" ○ Set the Pattern No. to: "0", click OK
- Click on the „Zoom Full" button and see the schematic:
- You will see a schematic of your full-adder with Pattern #0 (of the 6), showing you what input is at the input PINS and what „good-non-faulted" data should be at the output PINS.
- You can change the „**Setup'** to show the 5 other patterns and their expected data

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

- In the schematic view, click on the „Setup" button again
  - Set the Pin Data Type to: "Fault Sim Results", click OK
  - Your schematic will update and show the stuck-at faults captured by pattern #0, as shown here:



  - Zoom around the schematic and see if you can determine which gate is being tested with pattern 0
- Click on the „Setup" button again
  - Set the Pin Data Type to: "Fault Sim Results"
  - Before pressing OK, click on the „set parameters" button
  - Set the "stuck-at" fault to be 0
  - Set the "pattern no" to be 1, click OK



  - Your schematic will update and show the stuck-at-0 faults captured by pattern #1
- To see what else is possible to display on the schematic window, using the „setup" icon, look on page 7-19 of the tetraMax User Guide (which is posted under lab4 on the VLSI-128 website)
  You can now close the schematic view

12. 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/fulladder_tb_patterns.v`
- For the **File Format**, select: Verilog-Single File
- Press OK

13. Test the patterns out against your Synthesized full adder
    - Close TetraMax
    - Type:

```
cd ~/ece128/lab4/src
sim-nc osu05_stdcells.v fulladder_tb_patterns.v fulladder_syn.v
```

- If you have a successful run, you should see:

```
//            0.00 ns : Begin patterns, first pattern = 0
//         1200.00 ns : Simulation of 6 patterns completed with 0 errors
```

- This indicates that all 6 patterns that TetraMAX created were tested against the FullAdder you synthesized to AMI .5 technology in the Design Compiler
- Since your verilog has NO manufacturing defects, you will always get 0 errors!

- The idea is to use these patterns against a chip that you some day fabricate, and see if you still get 0 errors

14. Viewing the patterns in Simvision

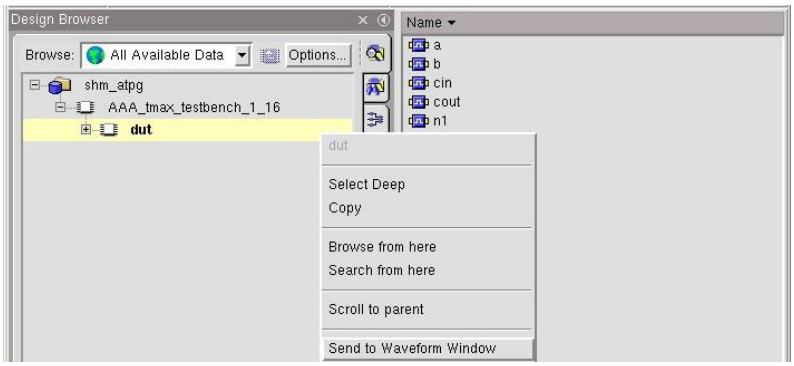
- Open up an editor (like gedit) and edit the file: `fulladder_tb_patterns.v`
- Notice the 6 patterns are at the bottom of the verilog code
- Go to the very bottom of the code
- Right before the “end module” statement, add the following lines of verilog:

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

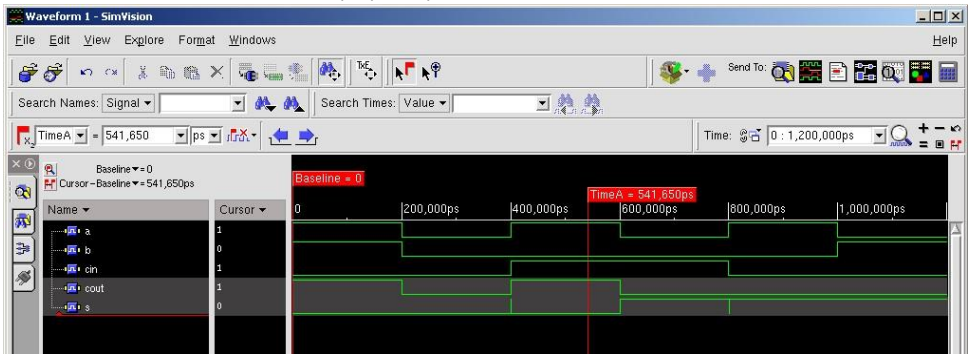
- This code makes it so the TetraMax test bench will generate simvision waveforms
- Once you"ve save the file, type the following:

```
sim-nc osu05_stdcells.v fulladder_tb_patterns.v fulladder_syn.v
```

- After the run, open up simvision and view the waveforms in the „*shm_atpg.db*“ directory
- Send the waveforms from the “DUT” sub-module to the waveform viewer



- Send the waveforms from the “DUT” sub-module to the waveform viewer
- Once the waveform view opens, delete the n1-n6 signals, and the wire1-3 signals
- You should see 6 different values for a, b, cin , does cout & s look correct for the full adder?



15. To show that you"ve completed this section of the lab, fill in the answers below regarding the full adder, print this page, and **post it to the BB (lecture lection) before next lab**:

- How many inputs combinations are possible for a fulladder? _____
- How many input combinations did TetraMAX create? _____
- Why do you think TetraMAX didn"t create all possible input combos? _____
  (*Hint: think of what these patterns are meant to test)*
- List all the patterns and their expected outputs that TetraMax generated:
  _____
- Draw the schematic from TetraMax, for the stuck-at-1 fault model for pattern #4, for the fulladder; what gate is pattern #4 actually testing?

Part IV: Generating Test Patterns for a COMBINATIONAL LOGIC circuit (without a scan-chain):

1. Setup files for this lab:

   - Copy a demonstration ripple counter from the VLSI account into you lab directory, type:

   ```
   cp /home/seas/vlsi/course_ece128/lab_files/lab4/src/ripplecarry4_clk.v ~/ece128/lab4/src
   ```

   - This ripplecarry adder is 4-bits. It uses 4 fulladder"s chained together. The only difference is that inbetween 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.
   - Note: we will discuss what a “scan-chain” is in the next lab, but it is important to note that this design does not have a scan-chain inserted into it.

2. Edit the synthesis script:

   - Open up an editor (gedit), and edit the file: lab4/**dc_syn.tcl**
   - Change the top of the file to make the synthesis script synthesize the ripple carry adder. Make the changes highlighted in yellow:

   ```
   ###################################
   # 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)
   ```

   - You can see how easy it would be to use this script for your ECE 126 synthesis, or any Verilog project for that matter!
   - Save and exit the editor.

3. Change to your working directory

   ```
   cd ~/ece128/lab4
   ```

4. 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)*

5. 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 lab4 directory)*

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

7. 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 in-between each fulladder circuit

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

1. Change to your working directory:

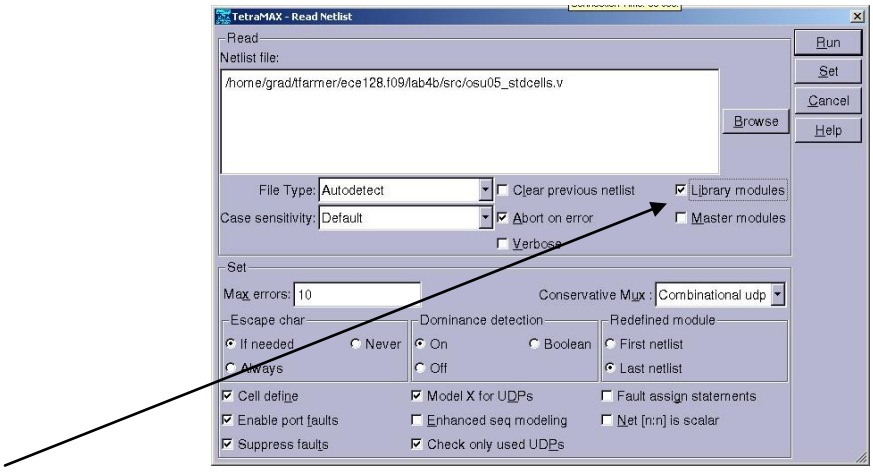
```
cd ~/ece128/lab4
```

2. Start the Synopsys TetraMAX GUI by typing:

```
tmax &
```

3. Load the AMI .5 standard cell verilog library

   - Click on the "**NETLIST**" button at the top of the screen
   - Browse for file: **osu05_stdcells.v** (in your src directory):



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

4. Load your Synthesized Ripple Carry adder:
   - Again click on the "**NETLIST**" button at the top of the screen
   - Browse for the file: ripplecarry4_clk_**syn**.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

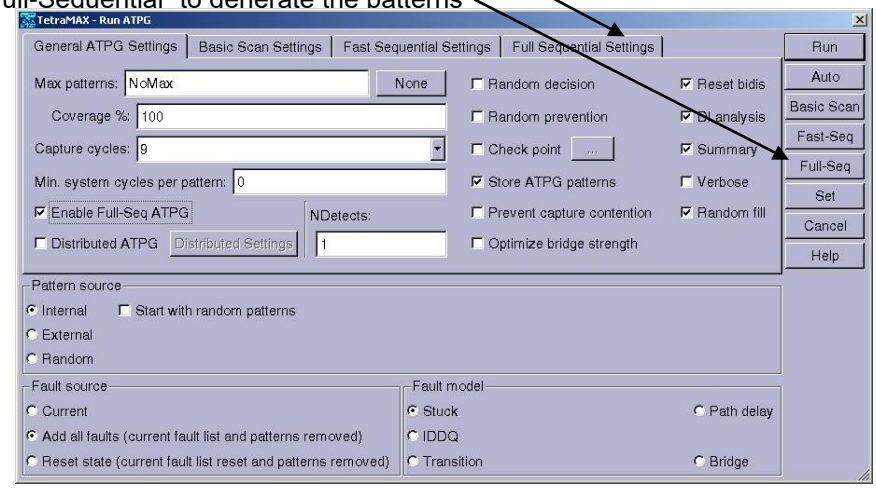
5. **BUILD** the Ripple Carry Adder
   - Click on the "**BUILD**" button at the top of the screen, accept defaults, and click run

6. **DRC** the Full Adder:
   - Click on the "**DRC**" button at the top of the screen ,accept defaults, and click run

You may ignore warnings you receive about an undefined Verilog User Defined Primitive.

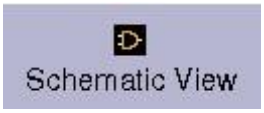
7. Generate the Test Patterns for the Full Adder:
   - Click on the "ATPG" button at the top of the screen (stands for „automatic test pattern generation")
   - Check the „add all faults" check box at the bottom of the window
   - Ensure the „stuck" at fault model is checked

- Click on "Enable Full Seq ATPG" (this is different now that our circuit is sequential in nature) 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:

8. View the patterns TetraMax has generated (with the ATPG basic scan algorithm)
   - Use the same technique as you did in the last portion of the lab to show the faults



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

10. 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

11. Test the patterns out against your Synthesized full adder
    - Close TetraMax
    - Add a $shm_open and $shm_probe command to the generated testbench like you did earlier in this tutorial.
    - Type the following commands to run the new test bench against your synthesized code

```
cd ~/ece128/lab4/src
sim-nc osu05_stdcells.v ripplecarry4_clk_tb_patterns.v ripplecarry4_clk_syn.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 **post it to the BB (lecture lection) 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? _____ Why do you think it was not 100%?
    - Is it okay to use the generated test bench instead of your own Verilog test bench to perform function testing of your design?_____ Why or why not?_____
    -

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 B-2008.09-SP2, December 2008

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:

```
#####################################################################
#### 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/fulladder.v ./src/halfadder.v] ;
set basename fulladder         ;# Top-level module name
set myClk clk                    ;# The name of your clock
set virtual 1                       ;# 1 if virtual clock, 0 if real clock
set myPeriod_ns 40                ;# desire 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 writing of result files
###################################
set verbose 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

####################################
# 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 -
ungoup_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 }

####################################
# quit dc
```

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