



Universität Bremen

FACULTY OF PHYSICS AND ELECTRICAL ENGINEERING

Institute for Theoretical Electrical Engineering and Microelectronics

Digital System Laboratory

Lab Report 3: Exercise on DFT

Submitted By

Md Shazzad Hossain

5228229

Supervised by
Prof. Dr. Alberto Garcia-Ortiz

Assisted by
Dr. -Ing. Amir Najafi

Contents

| | |
|---|----|
| Chapter 1: Introduction | 2 |
| 1.1 Goals | 2 |
| Chapter 2: RTL Design | 2 |
| Chapter 3: Launching Design Compiler | 5 |
| Chapter 4: DFT configuration and synthesis | 6 |
| 4.1 Importing Design | 6 |
| 4.2 First DFT Synthesis | 6 |
| 4.3 Second DFT synthesis..... | 7 |
| Chapter 5: Automatic Test Pattern Generation..... | 9 |
| 5.1 Lunching TetraMax..... | 9 |
| 5.2 Reading Netlist and Library Files | 9 |
| 5.3 Building the Design | 10 |
| 5.4. DRC check..... | 12 |
| 5.5. Generate the Test Pattern | 13 |

Chapter 1: Introduction

Using Design for Testability (DFT) methodologies, we will improve the IC design in this exercise to make post-fabrication testing easier and more effective. A collection of design approaches and practices called Design for Testability (DFT) aims to make integrated circuits more testable both during production and throughout their operational life. The design will also incorporate Automatic Test Pattern Generation and Scan architecture integration in addition to DFT adjustment. We improve the testability of the IC design by applying these DFT approaches. Post-fabrication testing is made easier, more extensive, and more effective, which enables the early identification of flaws in the design. DFT gives us the potential to enhance the integrated circuit's quality and dependability, providing a more durable and reliable final product.

1.1 Goals

The primary goals of our design are as follows: -

- Gated clock and storage elements are the hardest to test, our goal is to modify our design in such a way we can deal with these elements.
- We must integrate a scan architecture into our design.
- At the end of our design, we need to generate an automatic test pattern (ATPG) for testing our design.

Chapter 2: RTL Design

We must modify our rtl design for testing by adding some scan input and output pins. We also need to consider the different design styles such as gated clocks.

2.1 Modifying RTL Design

In the subfolder rtl, we modified the waveform_gen.vhd file. We added the following element to our top RTL design.

- SERIAL IN, TESTMODE, and SCAN EN as input ports
- SERIAL OUT as output port of the std logic
- Memory bypassing, as the state of the memory is unknown, it's better to bypass.

2.2 RTL Code

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.parameter.all;

entity waveform_gen is
  port(
    clk,rst,start,mode,data_in : in std_logic;
    sync_rst_ram_addr : in std_logic;
```

```

period_multiplier:in std_logic_vector(counter_sleep_width-1 downto 0);
data_out: out std_logic_vector(bit_width_data-1 downto 0);
sleep_reg: out std_logic_vector(counter_sleep_width-1 downto 0);
-----
SERIAL_IN, SCAN_EN, TESTMODE: in std_logic;           -- Modified for TST
SERIAL_OUT: out std_logic
);
end entity;

architecture behaviour of waveform_gen is
signal gated_clk,sleep_inv: std_logic;
signal gated_clk2,sleep2_inv: std_logic;
signal re_ram,we_ram,oe_ram,enable_reg_file,sleep,sleep2,enable_shift_reg: std_logic;
signal addr_ram: std_logic_vector(size_ram_addr-1 downto 0);
signal data_shift_reg,ram_out,ram_out_dft:std_logic_vector(bit_width_data-1 downto 0);
-----
signal RAMBO, RAMBO_out : std_logic_vector(bit_width_data-1 downto 0); -- Modified for TST
signal intEnRAM : std_logic;

begin

sleep_inv <= not sleep;
sleep2_inv <= not sleep2;

clk_gate1:CLKGATE_X1
port map(
    CK => clk,
    E => sleep_inv,
    GCK => gated_clk
);

clk_gate2:CLKGATE_X1
port map(
    CK => clk,
    E => sleep2_inv,
    GCK => gated_clk2
);

controller_main:fsm
port map(
    clk=> gated_clk,
    rst=>rst,
    start=>start,
    mode=>mode,
    re_ram=>re_ram,
    we_ram=>we_ram,
    oe_ram=>oe_ram,
    enable_shift_reg => enable_shift_reg
);

controller_sleep0:controller_sleep
port map(
    clk=>clk,
    rst=>rst,
    re_ram => re_ram,
    we_ram => we_ram,

```

```

    addr_ram => addr_ram,
    sync_rst_ram_counter=> sync_rst_ram_addr,
    sleep_time=>period_multiplier,
    sleep=>sleep,
    sleep2=> sleep2,
    enable_reg_file => enable_reg_file,
    sleep_reg => sleep_reg
);

shift_reg_data_in:shift_reg
port map(
    clk=>gated_clk,
    rst=>rst,
    data_in=>data_in,
    ena=>enable_shift_reg,
    data_out=>data_shift_reg
);

ram_block:SRAM64x8_1rw
port map(
    CE=>gated_clk,
    WEB=>we_ram,
    REB=>re_ram,
    OEB=>intEnRAM,          -- Modified for TST
    A=>addr_ram,
    I=> data_shift_reg,
    O=> ram_out
);

output_register:reg_file
port map(
    clk=>gated_clk2,
    rst => rst,
    ena => enable_reg_file,
    data_in => RAMBO_out,    -- Modified for TST
    data_out => data_out
);

-----
-- Added for TST
-----

xor_reg:process(clk,rst) is
begin
    if rst='0' then
        RAMBO <= (others => '0');
    elsif clk='1' and clk'event then
        RAMBO <= ((addr_ram & we_ram & re_ram) xor data_shift_reg);
    end if;
end process xor_reg;

mux2:process(RAMBO ,ram_out, TESTMODE) is
begin
    if TESTMODE='1' then
        RAMBO_out <= RAMBO;
    else

```

```

    RAMBO_out <= ram_out;
end if;
end process mux2;

mux1:process(oe_ram, TESTMODE) is
begin
    if TESTMODE='1' then
        intEnRAM <= '0';
    else
        intEnRAM <= oe_ram;
    end if;
end process mux1;
end architecture;

```

Chapter 3: Launching Design Compiler

To use the Design compiler, we created a [sourceme.sh](#) file with the following content in the do_synthesis folder.

```

export SNPSLMD_LICENSE_FILE=28231@item0096
export PATH=/usrf01/prog/synopsys/syn/R-2020.09-SP4/bin:${PATH}

```

Now we can source the file by writing [source.sourceme.sh](#)

To use the standard cell libraries and to instruct the tool to save the log files into the directories that we have previously defined we created a file named [.synopsys_dc.setup](#) with the following contents.

```

define_design_lib work -path ./tool/work

set_app_var view_log_file    ./log/synth_view.log
set_app_var sh_command_log_file ./log/synth_sh.log
set_app_var filename_log_file ./log/synth_file.log

set_app_var search_path      [concat ./cmd/ [get_app_var search_path] ]

set library_path "..././0_FreePDK45/LIB/"
set library_name "NangateOpenCellLibrary_typical_ccs_scan.db"
set library_name2 "SRAM64x8_1rw.db"

set_app_var target_library    $library_name
set_app_var link_library      [concat $library_name $library_name2 dw_foundation.sldb "*"]
set_app_var search_path       [concat $library_path [get_app_var search_path] ]
set_app_var synthetic_library [list dw_foundation.sldb]
set_app_var symbol_library     [list class.sdb]

set_app_var vhdlout_use_packages { ieee.std_logic_1164.all NangateOpenCellLibrary.Components.all }
set_app_var vhdlout_write_components FALSE

```

Afterward, we launch the design compiler with the command [design_vision | tee log/synthesis.log](#), [tee log/synthesis.log](#) command is used to make sure that a log file is saved in the file synthesis.log in the log directory.

Chapter 4: DFT configuration and synthesis

4.1 Importing Design

Before DFT synthesis, we write some TCL commands to set different options in the tool. For a more complex design, it is more useful to write the TCL commands in a separate file and then later source that file for use. But for our purpose, we can source them separately. The TCL codes have the following functions: -

- *config_synth.tcl*: all .vhd files, top-level rtl files, names, and variables are defined in this file.
- *read_design.tcl*: used to read the RTL code.
- *global_constraints.tcl*: used to set simple design constraints.

We can source the TCL command by writing *source ./cmd/config_synth.tcl*, *source ./cmd/read_design.tcl*, *source ./cmd/global_constraints.tcl*.

4.2 First DFT Synthesis

To synthesis the DFT, first, we're going to configure the DFT. Several settings must be set, to configure the DFT. For example, we choose the clock, test duration, kind of scan cells, reset, scan input and output, scan enable, and test mode from all the possibilities. For scan cells, there are three widely used scan cells Muxed-D, Clocked, and LSSD scan cells. For our design, we used Muxed-D scan cells.

We created a file called *dft_config.tcl* to add all the dftsynthesis commands by writing *emacs cmd/dft_config.tcl*.

Into that file, we added the following DFT configuration commands:

```
set_scan_configuration -style multiplexed_flip_flop
set_test_default_period 100
set_dft_signal -view existing_dft -type ScanClock -timing {45 55} -port clk
set_dft_signal -view existing_dft -type Reset -active_state 0 -port rst
set_dft_signal -view spec -type ScanDataIn -port SERIAL_IN
set_dft_signal -view spec -type ScanDataOut -port SERIAL_OUT
set_dft_signal -view spec -type ScanEnable -port SCAN_EN -active_state 1
set_dft_signal -view existing_dft -type Constant -port TESTMODE -active_state 1
create_test_protocol
```

After that we started DFT synthesis by executing the command: *compile -scan*, with this we compiled our system using the scan option. This tool replaces all the sequential elements with scan equivalent.

Now, we can check the design for DFT violations using the command: *dft_drc*.

After that to specify the scan chain we wrote the following commands:

- *Set_scan_configuration -chain_count 1*, to select the number of the scan chain.
- *Set_scan_configuration -clock_mixing no_mix*, to define that no different clock edges may occur in the scan chain.
- *set_scan_path chain1 -scan_data_in SERIAL_IN -scan_data_out SERIAL_OUT*, to define the input and output of the scan chain.
- *Insert_dft*, to insert the scan chain into the design.
- *Set_scan_state scan_existing*, indicates if the scan chain is fully implemented or not.

Now to see the report of the scan chain

```
report_scan_path -view existing_dft -chain all > reports/chain1.rep  
report_scan_path -view existing_dft -cell all > reports/cell1.rep
```

4.3 Second DFT synthesis

From the RTL of our design, we can see that, we have a gated clock in our design, to improve the testability of our design we need to do further modifications. So, we replaced our gated clock architecture with its test equivalent architecture.

The modified RTL code looks like:

```
clk_gate1:CLKGATETST_X1          -- TEST CLOCK GATE  
port map(  
    CK => clk,  
    E => sleep_inv,  
    GCK => gated_clk,  
    SE => TESTMODE  
);  
  
clk_gate2:CLKGATETST_X1          -- TEST CLOCK GATE  
port map(  
    CK => clk,  
    E => sleep2_inv,  
    GCK => gated_clk2,  
    SE => TESTMODE  
);
```

With this standard cell, we use an extra OR gate, which forces CEN to 1 using either the TM or SE signals, which solves our gated clock problem for testability.

Now we need to do the DFT configurations again for our new RTL code. We can source our *dft_config.tcl* file by writing source *./cmd/dft_config.tcl*

After the configuration, we can start our DFT synthesis by running compile -scan, and check the DFT violation by writing *dft_drc*, same as before.

We received no design violations, as a result, which represents our design is good.

Now we can specify the scan chain, same as before by writing the following codes.

```
Set_scan_configuration -chain_count 1  
Set_scan_configuration -clock_mixing no_mix  
Set_scan_path chain1 -scan_data_in SERIAL_IN -scan_data_out SERIAL_OUT  
Insert_dft  
Set_scan_state scan_existing
```

Now to see the report of the scan chain

```
Report_scan_path -view existing_dft -chain all > reports/chain1.rep  
Report_scan_path -view existing_dft -cell all > reports/cell1.rep
```

Now we save our files for test pattern generation with TetraMax, using the following tcl commands.

Change_names -hierarchy -rule verilog

```
Write -format verilog -hierarchy -out results/waveform_gen.vg  
Write -format ddc -hierarchy -output results/waveform_gen.ddc  
Write_scan_def -output results/waveform_gen.def  
Set test_stil_netlist_format verilog  
Write_test_protocol -output results/waveform_gen.stil
```

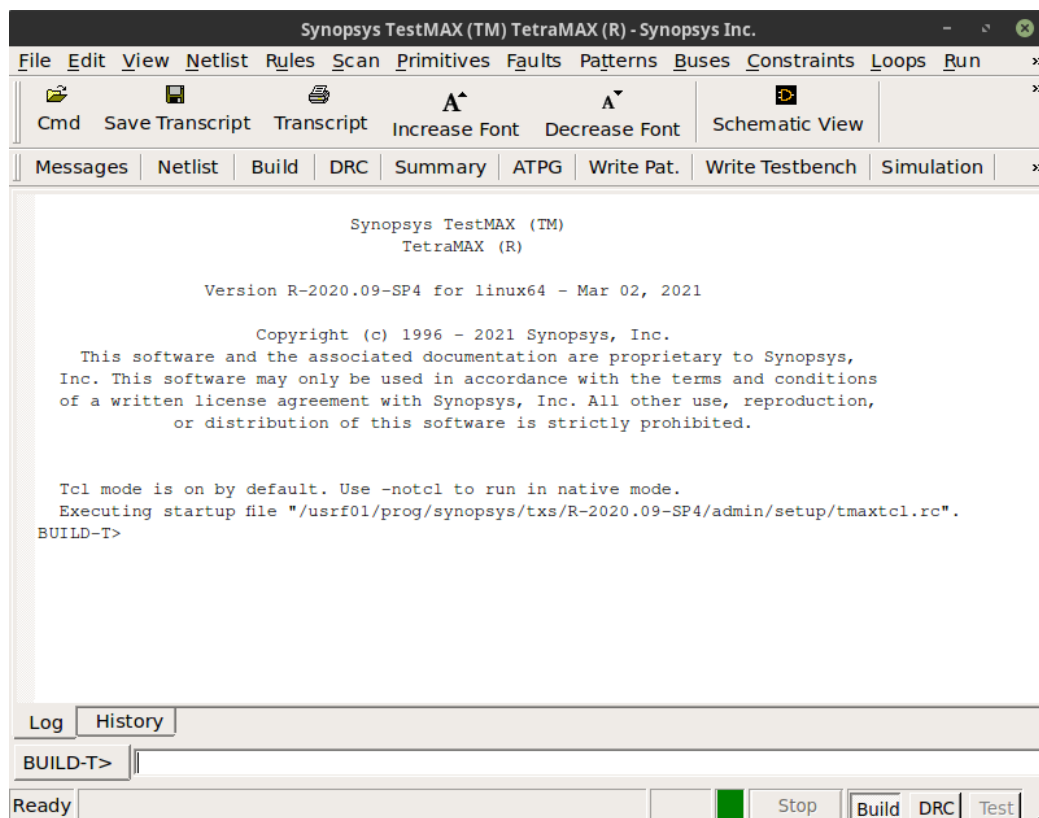
After that, we used *remove_design* command to remove the design.

Chapter 5: Automatic Test Pattern Generation

5.1 Lunching TetraMax

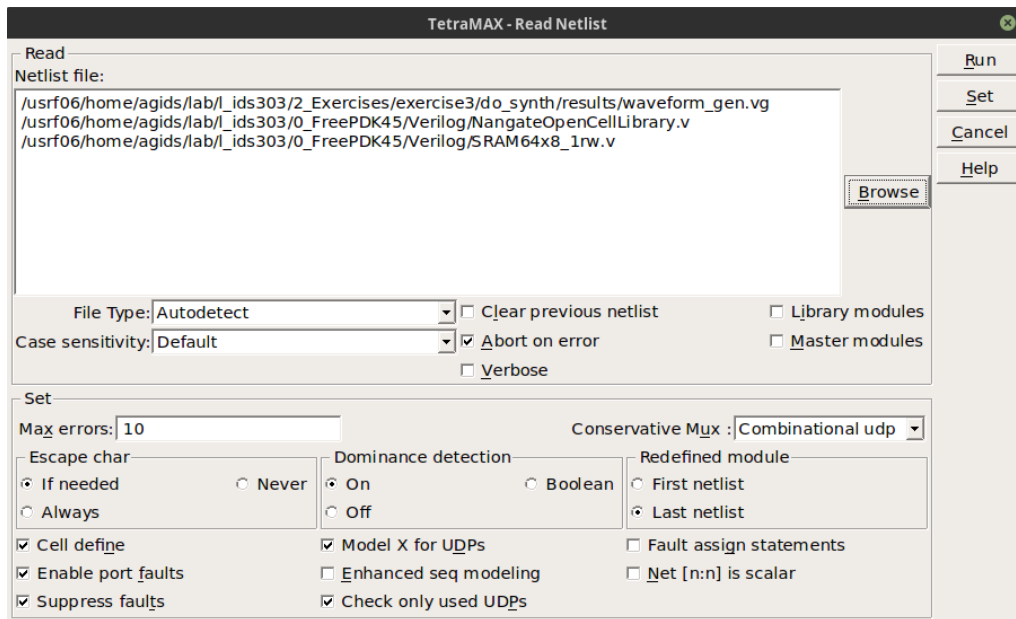
To create a test pattern, we need to use TetraMax, after going to the TetraMax subfolder, we can launch TetraMax by using The following command.

```
Export PATH="/usrf01/prog/synopsys/txs/R-2020.09-SP4/bin:${PATH}"
Tmax
```



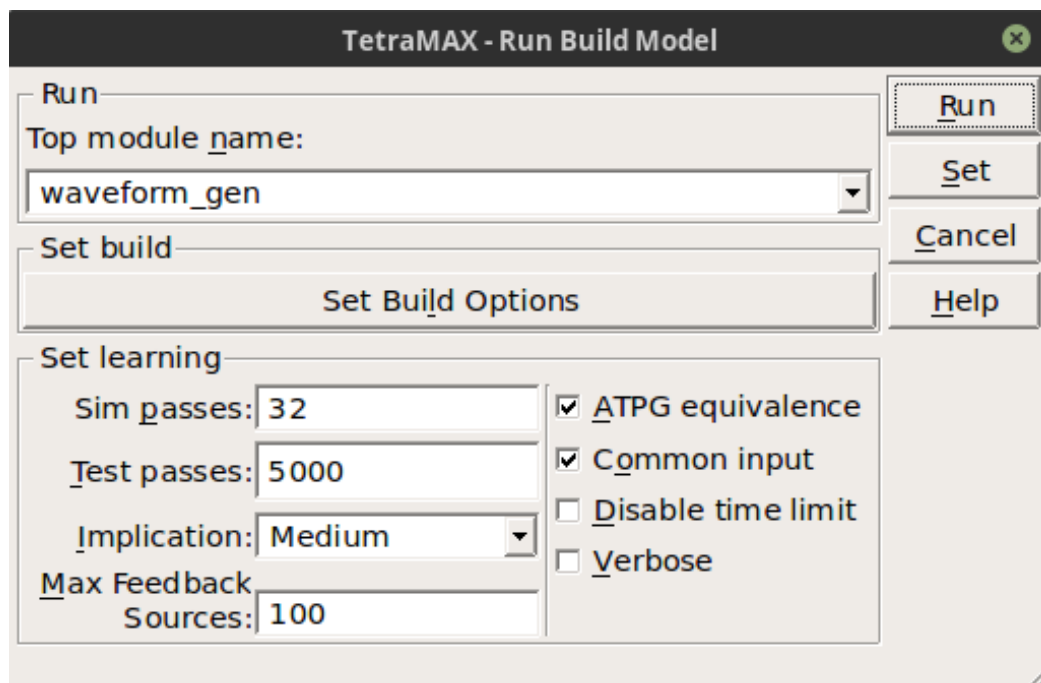
5.2 Reading Netlist and Library Files

Now we have to read the netlist and the library file into the read Netlist window. We found the netlist in do_synth/results sub-directory. We also had to add the nangate library file.



5.3 Building the Design

To build our design we need to set relevant parameters. First, we select build from the menu bar and select our top model. We used the default options of the Set learning box.



After selecting the top module, we selected Set Build Options and added SRAM as the black box.

TetraMAX - Set Build

Hierarchical: /

☒ Add buffers

Undriven bidi

☒ PIO
☐ PO
☐ PI

Fault boundary

☒ Lowest
☐ Hierarchical

OK

Cancel

Help

Optimizations

MUXes:

No Fault Site Loss

XOR/XNORs:

No Fault Site Loss

DLAT/DFFs:

Strong Equivalence

FlipFlops:

Flipflop from dlat

☒ Bus keepers

☐ Cascaded gates w/ fault site loss

☒ Delete unused gates

☒ Feedback paths

☒ Global tie propagate

☐ Master slave into DFF

☒ Tied gates with pin loss

☐ Tied gates with fault site loss

☒ Wire to buffer

Net connections

☒ Change netlist

Boxes

Module:

Box type:

BlackBox

Add

| Module | Box Type |
|--------------|----------|
| SRAM64x8_1rw | BlackBox |

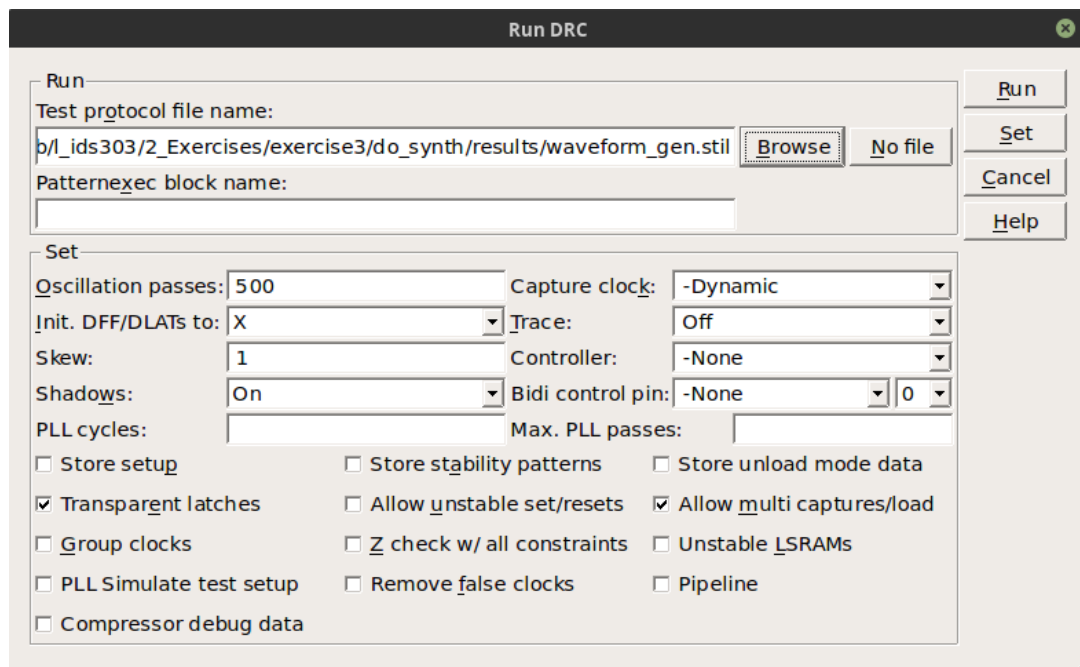
Remove

Modify

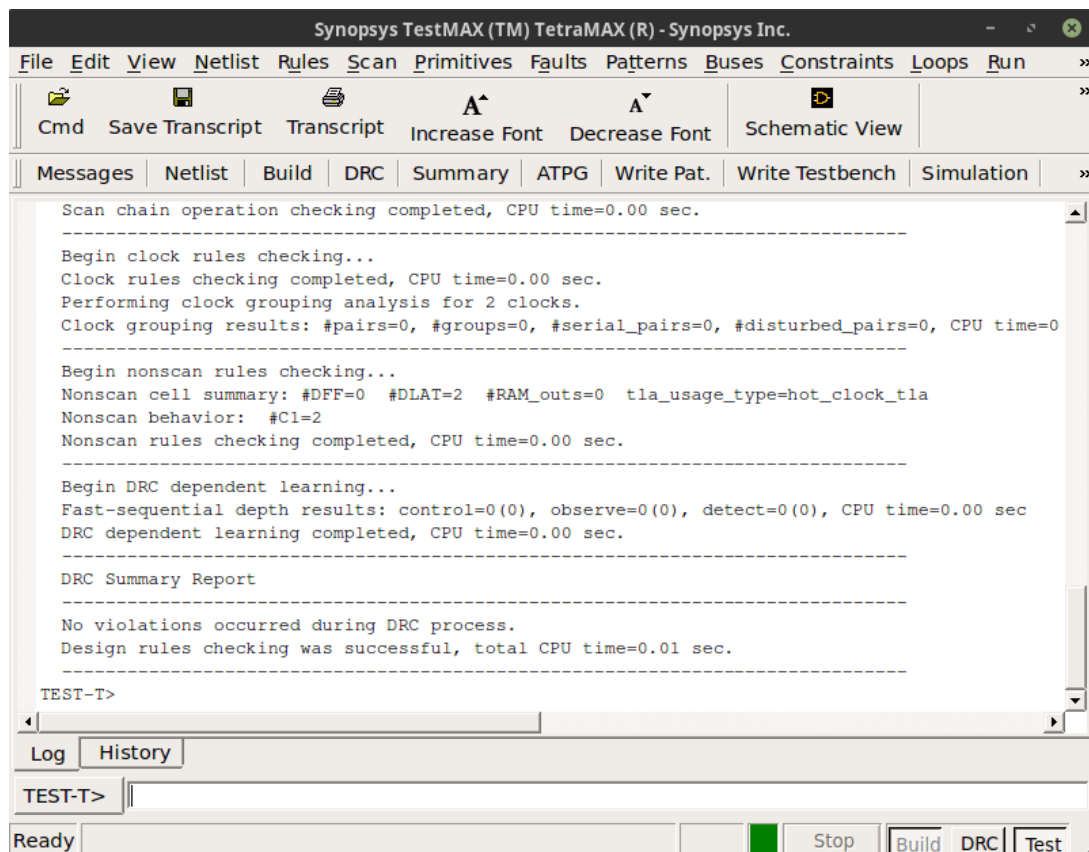
Copy

5.4. DRC check

Now, we checked the design rules (DRC), and for that, we went to the DRC window and selected the test protocol file named *waveform_gen.stil*. This is the file we created and saved in the previous step in the *do_synth/result* sub-folder. Now we can initiate the DRC check by clicking run.



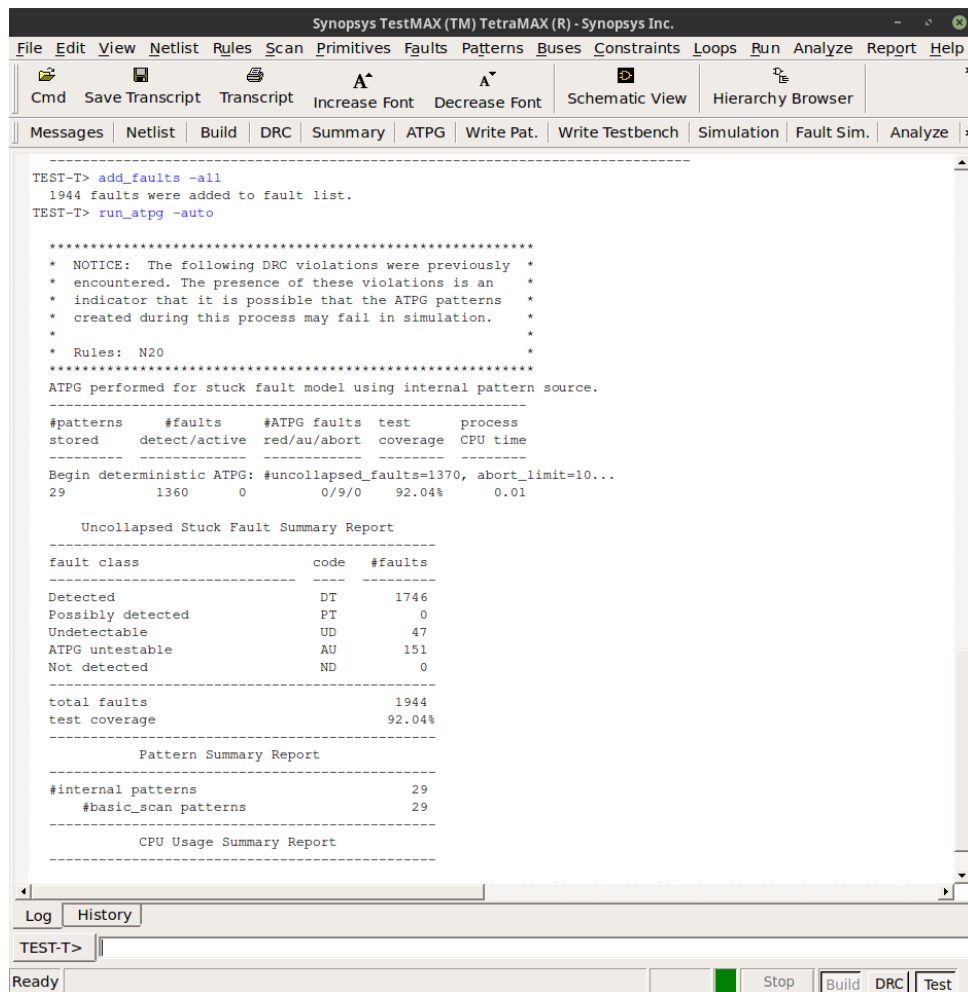
The DRC check result looks like this, which implies there were no design violations.



5.5. Generate the Test Pattern

Now after importing the design and completing the DRC check without error, we used TetraMax to create the test patterns. We used the following two commands to annotate the errors and generate the test patterns:

```
add_faults -all
run_atpg -auto
```



The screenshot shows the Synopsys TestMAX (TM) TetraMAX (R) - Synopsys Inc. interface. The main window displays the results of the ATPG process. The command prompt shows the following commands and output:

```
TEST-T> add_faults -all
1944 faults were added to fault list.
TEST-T> run_atpg -auto
```

The output includes a notice about DRC violations and a summary of the ATPG process:

```
*****
* NOTICE: The following DRC violations were previously *
* encountered. The presence of these violations is an *
* indicator that it is possible that the ATPG patterns *
* created during this process may fail in simulation. *
*
* Rules: N20
*****
ATPG performed for stuck fault model using internal pattern source.
-----
#patterns    #faults    #ATPG faults  test    process
stored      detect/active  red/au/abort  coverage  CPU time
-----
Begin deterministic ATPG: #uncollapsed_faults=1370, abort_limit=10...
29           1360      0           0/9/0    92.04%    0.01
```

The output also includes an "Uncollapsed Stuck Fault Summary Report" and a "Pattern Summary Report".

| fault class | code | #faults |
|-------------------|------|---------|
| Detected | DT | 1746 |
| Possibly detected | PT | 0 |
| Undetectable | UD | 47 |
| ATPG untestable | AU | 151 |
| Not detected | ND | 0 |
| total faults | | 1944 |
| test coverage | | 92.04% |

The "Pattern Summary Report" shows:

| #internal patterns | 29 |
|----------------------|----|
| #basic_scan patterns | 29 |

The "CPU Usage Summary Report" is also present.

The interface includes a menu bar (File, Edit, View, Netlist, Rules, Scan, Primitives, Faults, Patterns, Buses, Constraints, Loops, Run, Analyze, Report, Help) and a toolbar with icons for various functions. The status bar at the bottom shows "Ready" and "Stop" buttons.

As we can see from our result, we got 92.04% test coverage. Finally, we saved our generated test pattern in binary format with following the command:

```
write_patterns waveform_gen_pattern.v -internal -format binary
```