Reminder:
With the execution of this training you declare that you understand and accept the regulations about using CAE/CAD software installations at the ETH Zurich. These regulations can be read anytime at http://eda.ee.ethz.ch/index.php/Regulations
1 Introduction

Design for testability (DFT) is an underestimated part of the design process and should be addressed at a very early stage of your design. As you will realize in this exercise, you cannot just build your chip, insert a simple scan chain and expect to have a design with perfect testability. In many cases, solutions that improve testability will require significant changes to the way your circuit works. Therefore it is important to consider testability at an early stage, and make sure that your final design does not have weak testability.

If you are interested in selling your chips, testing is actually the most important part of the design flow. Simply put, you have to test every chip that you sell, and if you can not be sure that the chips you sell function properly, you can not run a successful business. We have already covered the theory related to testing in Exercise 0.

Before we start, let us make one thing clear. There is a difference between functional verification (the circuit does what you want), and testing (the netlist has been manufactured properly). In this exercise we will deal with automatic test pattern generation for standard digital circuits, learn how to add scan chains to a digital circuit, see practical problems, analyze test coverage, and investigate methods that will improve the test coverage.

1.1 Common methods to achieve testability

Dedicated test circuits need to be inserted into your design to add testability. These test circuits are inserted right at the end of the front end design, i.e., just before you start with the back-end design flow that will be covered in the following exercises. The most common test circuits are scan chains, test points and dedicated built-in-self-test (BIST) circuits.

Given a combinational circuit, and a fault model, test vectors can automatically be generated to test whether or not a fault is present in the circuit. This is very practical, as vectors can be automatically generated regardless of the functionality of the circuit. The same tools can also grade the efficiency of the generated vectors by determining what percentage of faults can be detected. The percentage of faults that can be detected depends mainly on the amount of inputs that can be controlled (controllability) and the number of outputs that can be observed (observability). The goal is to generate a set of vectors that achieves a high (more than 99%) test coverage for the circuit. While there are some very efficient algorithms that can solve this problem for combinational circuits, sequential circuits are more challenging, as the number of internal states of the circuit can be unmanageably high. One solution is to convert the internal state holding elements into scan chains. This essentially allows the output of every flip-flop to be controllable and the input of every flip-flop to be observable, effectively turning your sequential circuit into a combinational circuit with many controllable inputs and observable outputs. Test points are non-functional flip-flops that are inserted into the circuit to allow additional controllability and/or observability.

However, sometimes it is not possible to use those tools for certain circuit parts. This is often the case when IP-cores are used, such as memories, which do not provide appropriate models for the automated tools. In these cases, we need to isolate those circuit parts from the rest of the circuit such that we can still use the automated tools. But this is not sufficient, we need to come up with a strategy to test the circuits we have isolated as well.

Another term that is frequently used is built-in-self-test (BIST). This is a generic name for all circuits that are designed to automatically test parts of the circuit, reducing the overhead of testing. These can for example be designed to generate a specific test pattern for memories and automatically verifying that all memory locations can be written to and read from without errors. However, designing a good BIST is not trivial, and a BIST often requires a dedicated operation mode which must be supported by the overall circuit control unit.

In this exercise, we will first start with a simple toy example to learn the principles. Then we will move to the familiar FIR filter from previous VLSI exercises for more advanced topics. You will learn:

- How to insert scan chains
- How to insert test structures
- How to use Synopsys TetraMAX ATPG to measure test coverage and to find weaknesses in your design that degrade its testability.
- How to implement a BIST
- How to generate test vectors

Most importantly, you should realize that there exists no single solution for DFT that fits all circuit designs. For every design you need to check again what makes sense and what is the best strategy. Please do not just blindly copy the solutions in this exercise for your own projects.
1.2 About the Style

We will try to use different styles to identify different types of actions.

**Student Task:** Parts of the text that have a gray background, like the current paragraph, indicate steps required to complete the exercise.

Throughout the exercise you will be asked to enter certain commands using the command-line. The following is an example of the Linux command line.

```bash
sh> command to be entered on the Linux command line
```

Some of the commands will be entered on the command line of a specific tool:

```bash
dcs> this command is an Synopsys Design Compiler command
tmax> this command is an Synopsys TetraMAX command
```

2 Getting Started

You will need a terminal program to type in commands throughout this exercise. You can get a terminal by accessing the menu on the top left corner and selecting **Applications→Accessories→Terminal**.

**Student Task 1:**
- Change to your home directory and install the training files with the script provided:
  ```bash
  sh> cd ~
  sh> /home/vlsi2/ex01/install_ex01
  ```
- Change to the design directory
  ```bash
  sh> cd ex01
  ```

3 Writing Scripts

In this exercise you will have to execute **A LOT** of commands multiple times. If you do not write your own scripts on the go, this exercise will be tedious! We suggest to create a script for every larger set of commands. Scripts are to be placed in the scripts folder of the tools main folder:

**Student Task 2:** How to create and execute scripts:
- Create a empty script file for the Synopsis Design Compiler commands for the tasks in the next section and open it with your preferred editor. You can use gedit:
  ```bash
  sh> gedit ./synopsys/scripts/sc_run_all.tcl &
  ```
  Now you can collect all `dcs` commands in this file.
- You can execute the script from Synopsis Design Compiler with
  ```bash
dcs> source scripts/sc_run_all.tcl
  ```
  The same you can do for TetraMAX. We suggest to make a new script for every exercise section.
4 Scan Chain Insertion

In this first part of the exercise you will learn how to insert a scan chain.

The basic idea behind scan chains, as illustrated in Figure 1, is the following. State holding elements (flip-flops) are replaced with a version that has an additional input. Each flip-flop has therefore two inputs, the standard input and the Scan Data input. A control signal (Scan Enable) is used to select the standard input or the scan data input. A chain is formed by connecting the output of one state holding element to the scan data input of the following state holding element. The ends of this chain are then made available to the test equipment. For each clock cycle, data enters the scan chain at DataIn and exits the scan chain at DataOut, and the scan behavior is controlled by the ScanEn signal.

In the normal mode, the circuit works just as it did before scan chain insertion. However, by activating the scan enable, the scan chain can be turned into a shift register. In this mode, the content of all the state holding registers can be changed to an arbitrary pattern (controllability) by serially shifting in the desired pattern. After this, the circuit can be turned back to normal mode by deasserting scan enable. The output corresponding to the vector can be sampled by running the circuit for one (or more) clock cycle. The content of all the state holding elements in the scan chain can then be read out serially by activating the scan chain again.

If all state holding elements of a circuit are included in the scan chain, it is known as a full-scan design. If, for some reason, some state holding elements are not included in the scan chain, it is called a partial-scan design. The distinction is important, as full-scan designs effectively turn the entire circuit into a combinational circuit making it easy to test, whereas partial-scan circuits still have a sequential characteristic (you need more than one cycle to make sure that the observable outputs are changed directly by the controllable inputs) and are more difficult to test.

In principle, it is possible to add the scan functionality to the description of the original circuit and insert a scan chain manually. However, most synthesis tools support automatic scan chain insertion and are very efficient at it. In this exercise we will use SYNOPSYS DESIGN COMPILER to insert a scan chain into a standard design. Note that this exercise uses only the basic scan insertion functionality of the SYNOPSYS DESIGN COMPILER. For more information on extended capabilities refer to the Synopsys on-line documentation (SOLD) by typing

```
sh> synopsys-2016.03 sold &
```

on the command line. Select DFT Compiler to see a list of manuals related to testing. The functionality of the DFT compiler is accessible using standard SYNOPSYS DESIGN COMPILER interfaces (design_analyzer, dc_shell, design_vision). SYNOPSYS TETRAMAX ATPG is a separate program used to generate test vectors and to assess the test coverage. We will be using SYNOPSYS TETRAMAX ATPG at a later stage of the exercise.

For our simple scan chain, we need three additional connections:

- **Scan enable signal**: ScanEn_TI
- **Scan data input**: ScanIn_TI
- **Scan data output**: ScanOut_TO
Depending on the design, we may want to share part or all of these connections with existing ports of our design. Sharing ScanIn_TI with a regular input is usually very convenient, as sharing inputs has relatively low overhead. ScanOut_TO can share an output signal as well. If the output is directly taken from a flip-flop this does not involve any overhead. Otherwise, sharing the output would introduce a multiplexer into the signal path and increase the corresponding propagation delay by a small amount. ScanEn_TI can not directly be shared with normal functional inputs of the design, since it is used to control the operation mode of the entire system.

The first design that we will use in this exercise has been specifically designed for this exercise and is a simple pipeline with 3 expansion/reduction stages as seen in the Fig.2.

In this exercise we will share the ScanIn_TI with one of the DataIn_DI pins and ScanOut_TO with one of the DataOut_DO pins. However, we need to modify the top level of our design sc_combination.vhd to include the additional port for the ScanEn_TI signal.

Student Task 3:

- Modify the top level file sc_combination.vhd in the sourcecode sub directory and add a new input port definition for ScanEn_TI.
- Start SYNOPSYSS DESIGN COMPILER from the folder ex01. The most convenient way is to launch it with the cockpit using

```
sh> icdesign umcL65
```

and push the appropriate button. We will need to read in design files, analyze and elaborate them. There is a sample script that will do that for you (do not forget to check the output for errors):

```
dcs> source ./scripts/sc_elaborate.tcl
```

This script will also set simple constraints for the compilation that will be performed later.

As you may recall from the exercises in VLSI I, SYNOPSYSS DESIGN COMPILER uses two steps (‘analyze’ and ‘elaborate’) in order to read in behavioral VHDL descriptions. The ‘elaborate’ phase will list all memory devices inferred in the source code with a message like the following:

```
Inferred memory devices in process
   in routine sc_combination line 105 in file
      ../sourcecode/sc_combination.vhd'.
```

```
<table>
<thead>
<tr>
<th>Register Name</th>
<th>Type</th>
<th>Width</th>
<th>Bus</th>
<th>MB</th>
<th>AR</th>
<th>AS</th>
<th>SR</th>
<th>SS</th>
<th>ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>InputReg_DP_reg</td>
<td>Flip-flop</td>
<td>8</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>
```

In this example, the process starting in line 105 of the indicated file has resulted in 8 flip-flops. These flip-flops will be automatically named InputReg_DP_reg by SYNOPSYSS DESIGN COMPILER.

Note that our current design does not contain a reset synchronizer. If it would, we need to take certain measures to assure that the synchronizer flip-flop will not be integrated into the scan-chain by the automated scan-chain insertion.

1 In some systems, multi-bit code-words (instructions) are used. For such systems it is possible to “map” the ScanEn_TI signal to a combination of bits that is not used by the functional part.
2 It is possible to change the default naming convention of SYNOPSYSS DESIGN COMPILER.
At this moment we are ready to map the design into gates of the standard cell library of our 65 nm CMOS technology. Before we start with that we have to decide on how we will insert the scan chain in the architecture. There are two basic approaches for scan insertion:

1. The circuit is synthesized using the conventional design flow. Scan insertion is performed after design optimization. The netlist is modified, and all flip-flops are replaced with scanable variants.

2. All flip-flops of the design can be synthesized to have scan capability from the start.

The second option is more feasible, as the resulting netlist will also include the overhead of the scan chain (i.e., the flip-flops might become larger, might have different timings, etc.). Note that individual flip-flops are not yet connected to form a chain at this stage.

**Student Task 4:**

- Let us compile the design with the second option. Simply add the `-scan` parameter to the `compile_ultra` command. Please also add the `-no_autoungroup` flag to prevent the hierarchy from being dissolved:

  ```
  dcs> compile_ultra -scan -no_autoungroup
  ```

- Now let us find a particular flip-flop, e.g., the `InputReg_DP` register mentioned earlier. Using Design Vision, generate a schematic for the compiled circuit and then look for the flip-flop. It should be within the top-level entity `sc_combination`. You can also use `Select → By Name` to find the flip-flop faster.

- Examine the flip-flops corresponding to `InputReg_DP`. Can you identify the additional pins and their connections? If you are interested, you can search for the specific standard cell used to implement this flip-flop in the standard cell databook (`./docs/stdcell_low_vt_b03.databook.pdf`).

**Design Vision can be started from SYNOPSIS DESIGN COMPILER with the start_gui command.**

Note that all flip-flops have additional inputs for scan testing, however, they are not yet connected to form a scan chain. The design is `scan ready` but not yet `scan testable`.

Now that the design has been compiled, it is time to insert the scan chain.

**Student Task 5:**

- First, we need to make sure that clock and reset pins are properly identified for scan purposes.

  ```
  dcs> set_dft_signal -view existing_dft -type ScanClock -port Clk_CI -timing {50 80}
  dcs> set_dft_signal -view existing_dft -type Reset -port Reset_RBI -active_state 0
  ```

- Now the next point is to specify where the scan points will be connected to:

  ```
  dcs> set_dft_signal -view spec -type ScanEnable -port ScanEn_TI -active_state 1
  dcs> set_dft_signal -view spec -type ScanDataIn -port DataIn_DI[0]
  dcs> set_dft_signal -view spec -type ScanDataOut -port DataOut_DO[0]
  ```

  This time we use the `spec` view for the signals that are at the time not yet in use.

- We can use the command `report_dft_signal` to make sure that the configuration that we have entered matches what we expect.

- Now we need to create a test protocol. The following command will basically tell SYNOPSYS DESIGN COMPILER to analyze the design and prepare for scan insertion:

  ```
  dcs> create_test_protocol
  ```

- We need to execute the following command to see whether or not there are any violations:

  ```
  dcs> dft_drc
  ```

  If `dft_drc` reports problems, we must understand the problem and if necessary find a solution. For this very simple example there should not be any problems.
Student Task 6:

- We will now attempt to insert the scan chain:
  
  ```
  dcs > insert_dft
  ```

  This command will also perform again design rule fixings if required to meet timing constraints.

- Locate the same flip-flop in SYNOPSYS DESIGN COMPILER as you did earlier. The scan-chain should now be formed. Follow the input of the scan chain to the first few flip-flops. You may have to expand the busses Right-click on bus (dark blue line) → Expand → All Busses/Pins for better illustration. You can use report_scan_path command to see how the scan chain is actually connected. It is a good idea to save the output of this command in the ./synopsys/reports directory
  
  ```
  dcs > report_scan_path > reports/sc_scan_path.txt
  ```

We are now finished with scan insertion. We will now move to another software (SYNOPSYS TETRAMAX ATPG) to analyze the test coverage 3 Before we leave SYNOPSYS DESIGN COMPILER, let us make sure that we get a report for the area and the register to register timing. Then we can export the gate-level netlist.

Student Task 7:

- Run the following script to generate area and timing reports under ./synopsys/reports and the gate level netlist under ./synopsys/netlists/sc_combination.v:
  
  ```
  dcs > source scripts/sc_report_export.tcl
  ```

- By investigating the reports determine the Area: ____________________________

  Register to register timing: ____________________________

5 Analyzing Test Coverage

SYNOPSYS DESIGN COMPILER can be used to generate netlists that have scan chains, but is not designed to create test vectors directly. We will use a separate program called SYNOPSYS TETRAMAX ATPG for this purpose. What we want to know is what kind of test coverage we can obtain with our current circuit.

This part of the exercise is a simplified walk-through for the SYNOPSYS TETRAMAX ATPG tool.

5.1 Build-Phase

There are three phases of design for SYNOPSYS TETRAMAX ATPG. The first phase is called Build. In this phase, we load the standard cell libraries and the Verilog netlist that contains our circuit.

Student Task 8:

- Start SYNOPSYS TETRAMAX ATPG from the cockpit

- First we need to load the standard cell libraries (and if necessary macro block definitions). Cockpit automatically generates a script ./tetramax/scripts/read_lib.tcl that will read in these commands
  
  ```
  tmax > source scripts/read_lib.tcl
  ```

- Now we read in the netlist we have generated in SYNOPSYS DESIGN COMPILER
  
  ```
  tmax > read_netlist ../synopsys/netlists/sc_combination.v
  ```
• We are now ready to move to the next stage, we can tell SYNOPSYS TETRAMAX ATPG to build a test model for our top level entity

```
tmax > run_build_model sc_combination
```

5.2 DRC-Phase

If everything works fine, you will be in the second phase which is called DRC in SYNOPSYS TETRAMAX ATPG. In this phase the test signals are defined and a sanity check is performed to see if SYNOPSYS TETRAMAX ATPG can go to the third phase Test.

**Student Task 9:** In this phase we define the clocks and the scan chains:

- Define the clock with the following command. The `-shift` parameter tells SYNOPSYS TETRAMAX ATPG that this clock will be used as the scan clock.

```
tmax > add_clocks 0 Clk_CI -shift
```

- SYNOPSYS TETRAMAX ATPG defines clocks as “Top-level ports that can change the stored state of sequential devices”. By definition the reset signal is also a clock as far as SYNOPSYS TETRAMAX ATPG is concerned. The ’1’ specifies the off-state of the clock, in our case (active low) this is ’1’.

```
tmax > add_clocks 1 Reset_RBI
```

- Now we need to define at which port the scan chain starts and where it ends, and on which port the scan mode is enabled.

```
tmax > add_scan_chains chain1 DataIn_DI[0] DataOut_DO[0]
tmax > add_scan_enables 1 ScanEn_TI
```

Furthermore if certain ports need to be set to 0 or 1 such that the scan chain works, these need to be constrained with `add_pi_constraints` command at this point. For this circuit we do not need this.

- Perform a design rule check (DRC) that performs a variety of checks to ensure that the scan structure is correct and to determine how to use the scan structure for test generation and fault simulation.

```
tmax > run_drc
```

This test must be run successfully before you can continue with the next phase.

5.3 Test-Phase

Now we are in the final phase called Test. The first task is to generate a fault dictionary. We will use a simple stuck-at fault model.

**Student Task 10:**

- In order to generate the fault dictionary, type:

```
tmax > add_faults -all
```

- SYNOPSYS TETRAMAX ATPG reports the total number of faults in the system. To see a short summary of the current status of the faults, type:

```
tmax > report_summaries
```

4 You will see a couple of warnings, (Rule B9, B10, N23) for this design, these are OK
Notice that we have so far not generated any ATPG vectors. Still more than 650 faults were "Detected", these faults are detectable by implication, and they involve faults associated with the scan chain test circuitry. We will investigate all fault types shortly, for the time being let us generate ATPG test patterns.

5.3.1 Different approaches to reduce number of vectors

The number of test cycles affect both the speed of the test, and the cost of test equipment, as the memory for such equipment is extremely expensive. Perhaps for this exercise, this is not so critical, but if you are not careful, the number of clock cycles can increase very quickly.

Increasing the number of parallel scan chains and adding Built-in Self Test can both significantly reduce the test time. However, there are also a couple of tricks you can apply to reduce the number of test vectors in SYNOPSYS TETRAMAX ATPG without changing the circuit at all.

As you can see, with a couple of simple tricks you can significantly reduce the test time.

In our experience the -auto_compression option works the best, so try using that one in your scripts.

---

For example if you have 10'000 FFs, and need 4'000 test vectors, you will need roughly 40 million clock cycles. even when running at 100 MHz this is not negligible.
5.4 Analysis of faults

Consider once again the output of the `report_summaries` command which should be similar to the output below:

```
Uncollapsed Stuck Fault Summary Report
-----------------------------------------------
fault class code #faults
------------------------------ ---- ---------
Detected DT 19384
Possibly detected PT 0
Undetectable UD 14
ATPG untestable AU 0
Not detected ND 0
-----------------------------------------------
total faults 19398
test coverage 100.00%
```

The summary includes the following fault classes:

**DT** (Detected) Faults that can be detected.

**PT** (Possibly detected) These are faults for which the output of the circuit would produce an unknown (X) rather than a 0 or 1 for which the fault would have been detected. If the actual output of the circuit (which we can not determine at the moment) exhibiting this fault differs from that of a correct circuit, the fault would be detected. This is why it is listed as a possible detected fault.

**UD** (Undetectable) Undetectable faults can not be tested using any test vectors applied to the pins. The logic could be redundant, the circuit may have no connections to observable outputs, or its inputs may be tied to a fixed level. If an input is tied to logic-1, it is not possible to test the same input for a stuck-at-1 fault.

**AU** (ATPG untestable) Although these faults could theoretically be tested using ATPG algorithms, constraints of the circuit do not allow test patterns to be generated to detect these faults. A common example are test mode enable signals used to turn on block isolation, which are tied at logic-1 for pattern generation. You will later learn more about those signals. Another common problem is that logic is attached to sequential cells that are not part of the scan path.

**ND** (Not detected) Faults that cannot be detected.

Some of these faults also have subclasses, you can issue the following command to make the reports more verbose and to display sub groups.

```
tmax> set_faults -summary verbose
tmax> report_summaries
```

If you are careful, you will notice that the test coverage is listed as 100% even though there are clearly 14 undetectable faults as seen in the reports. This is due to the definition of test coverage by SYNOPSYS TETRAMAX ATPG. By default SYNOPSYS TETRAMAX ATPG calculates:

\[
\text{test coverage} = \frac{DT + PT}{All - UD - AU} \times 100
\]  

(1)

The better option would be to use the fault coverage of SYNOPSYS TETRAMAX ATPG which calculates:

\[
\text{fault coverage} = \frac{DT + PT}{All} \times 100
\]  

(2)

To change the reporting to this more relevant mode, you can use the following command:

```
tmax> set_faults -fault_coverage
tmax> report_summaries
```

It is important to understand why SYNOPSYS TETRAMAX ATPG cannot detect certain faults, as it will allows us to find methods to solve (at least part of) the problems. SYNOPSYS TETRAMAX ATPG can generate textual list files with a selected group of faults for further processing.

---

6. Of course in our case, the coverage is pretty good, but the methods described here will be useful in larger circuits as well.
Student Task 13:

- Write out all the faults of class UD to the text file `reports/faults−ud.rpt` by using the following command:
  ```
  tmax> report_faults -class UD > reports/faults-ud.rpt
  ```

- Repeat the same procedure with all detected faults (class DT)
  ```
  tmax> report_faults -class DT > reports/faults-dt.rpt
  ```

You can view this file using simple UNIX commands such as:

```sh
sh> less ./tetramax/reports/faults-dt.rpt
```

The file contains one line for every fault, and it will look similar to the following:

```
sa0 DI ScanEn_TI
sa0 DI i_sc_expander_1_0/OutReg_DP_reg_15_/CK
sa1 DS i_sc_expander_1_0/U300/Z
sa1 -- i_sc_expander_1_0/U250/C
```

The first column reports the fault type, the second column shows the detection status, and the last column states the fault location. The fault type can either be 'sa0' (stuck-at-zero) or 'sa1' (stuck-at-one). In the example, the first line specified that we can detect the stuck-at-0 fault at the `ScanEn_TI` by implication (if the scan chain works, this pin can not be stuck at zero).

A ‘−’ status in the second column means that the fault reported on the corresponding line is indistinguishable for the purpose of fault testing from the fault on the line immediately before that. In the specific case above, a stuck-at-one fault at the output pin `i_sc_expander_1_0/U300/Z`, and the input pin `i_sc_expander_1_0/U250/C` would have exactly the same behavior when observed from outside. The test vector that is able to detect one of this failure could just as well detect the other one.

You can make separate lists for different types of faults to see which parts of the circuit are testable to what extent. You can also list the summaries in a hierarchical manner. The following command reports the testability two levels deep and for every module that contains at least 32 faults.

```
 tmax> report_faults -level 2 32 -verbose
```

5.5 Using the graphical interface for analysis

It is possible to investigate why a certain fault is not detectable using a simple GUI in SYNOPSYS TETRAMAX ATPG. As an example let us try to understand why SYNOPSYS TETRAMAX ATPG reports the following faults (under `./tetramax/eports/faults−ud.rpt`).

```
sa1 UB InputReg_DP_reg_0_/SE
sa0 UB InputReg_DP_reg_0_/SE
```

Student Task 14:

- We want to start the GUI analysis tool. It could be that your initial window is too small to show this option on the menus (Analyze→Analyze Faults...) or the ANALYZE button on the row just above the text messages is not visible. In this case, make the window slightly larger, or use the '>>' on the right side of the window to access these options.

- Make sure that you are in the FAULTS tab of the Analyze window and press the <FILL button. This will bring up a requester called Fill Faults where you can choose which class of faults you want to select. Select UB: UNDETECTABLE BLOCKED and press OK.

- Now you should have two faults (stuck-at-one and stuck-at-zero of InputReg_DP_reg_0_/SE. Select one of the two and press OK. This will bring a small graphical window where you can see the schematic of the relevant

---

7 These pins are most probably directly connected.
components. Also note that the report window has a lengthy explanation of the problem.

- How can you solve this problem?
- You can use CLOSE GSV button to close the graphical window, if you no longer need it.

Now that we know why these faults are there, we could decide to remove them from our fault dictionary. In this case we could create a text file that contains all the faults that we could detect in a different way. Assuming that these faults are in the file `/tetramax/reports/ignore_list.txt`, you could write the following commands to: reset the state, add faults to the library, and remove the faults listed in the given file from the dictionary:

```bash
tmax> reset_state
tmax> add_faults -all
tmax> read_faults reports/ignore_list.txt -delete
```

### 6 Partial Scan Test

In this section we are going to show you how to generate the test patterns for a partial scan based design. We want you to:

- Understand what we gain or lose by using a partial scan circuit
- Learn the commands necessary to generate test patterns for a partial scan circuit.

We will once again use the same example design from before (see Figure 2), but will not connect all registers to the scan chain to get a partial scan based design. If you examine the figure you will see that the design consists of consecutive pipeline stages that are called `sc_expander` and `sc_reducer`. Each block has a pipeline register at its output. In this exercise we will tell SYNOPSYS DESIGN COMPILER not to include the flip-flops within the `sc_expander` stages.

**Student Task 15:**

- Return to SYNOPSYS DESIGN COMPILER and reload our example design using the following script:
  ```bash
dcs> source scripts/sc_elaborate.tcl
  ```

- Run the following command to exclude the flip-flops within the `sc_expander` stages:
  ```bash
dcs> set_scan_element false [find cell *i_sc_expander_1*/OutReg_DP_reg*]
  ```

- Continue the design flow just like before. `dft_drc` will report how many flip-flops are in the scanchain and how many not. Note that the script `/synopsys/scripts/sc_report_export.tcl` will overwrite the results from the previous run with full scan.

- By investigating the reports determine the
  - Area:
  - Register to register timing:

- Can you tell us how much you have gained by omitting scan capability from some flip-flops? 

Now that we have the design with partial scan chains, we can load it into SYNOPSYS TETRAMAX ATPG and perform the same analysis as before.

**Student Task 16:**

- Re-start SYNOPSYS TETRAMAX ATPG and follow the same steps as before, add all the faults, and look at the summaries before generating any tests. What category are most of the faults?

- There are only 88 faults that were not yet detected, if we run the standard ATPG algorithm, in the best case only these can be detected:
  ```bash
tmax> run_atpg -auto_compression
  ```
As you can see the test coverage is disastrous. During normal scan testing we apply one vector and operate the circuit for one cycle before capturing the result. In our present design, we have one additional register between every scanable register. This means that we will need to load in the vector and apply two cycles before sampling the output. We should better tell this SYNOPSYS TETRAMAX ATPG.

<table>
<thead>
<tr>
<th>Student Task 17:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Now we should enable the sequential testing mode. We set the number of -capture_cycles to 2 (any other value would be pointless), and then activate a fast sequential test.</td>
</tr>
<tr>
<td>tmax&gt; set_atpg -capture_cycles 2</td>
</tr>
<tr>
<td>tmax&gt; run_atpg fast_sequential_only</td>
</tr>
<tr>
<td>• What is the fault coverage?</td>
</tr>
</tbody>
</table>

While this one improved the test coverage significantly, we are nowhere near to where we should be. The next step is to start the proper sequential test.

<table>
<thead>
<tr>
<th>Student Task 18:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• We use the fast sequential test to get most of the vectors out of the way quickly. The normal sequential test will take very long. In the following we use -full_seq_time as 0.02, this value is given in seconds and tells more or less how long to iterate for each step. Setting it so low will reduce the quality (default value is 10-20s) but will make the vector generation much faster.</td>
</tr>
<tr>
<td>tmax&gt; set_atpg -full_seq_time 0.02</td>
</tr>
<tr>
<td>tmax&gt; set_atpg -full_seq_atpg</td>
</tr>
<tr>
<td>tmax&gt; run_atpg full_sequential_only</td>
</tr>
<tr>
<td>• What is the fault coverage?</td>
</tr>
<tr>
<td>• If you do not want to wait you can set a target coverage rate by set_atpg -coverage 85.</td>
</tr>
</tbody>
</table>

Overall, I hope you see that there is not much point in using partial scan, unless you are somehow forced to use it. The coverage is not as good, you end up using more vectors, and the vector generation takes much much longer.
7 Issues with RAM macros

To demonstrate you the basic working of the DFT flow we have used a very simple design in the first part. In this second part we will use the familiar FIR filter seen in Fig 3 from previous VLSI exercises to demonstrate some additional practical problems.

7.1 Understanding the Problem

We have included a netlist ./synopsys/netlists/top.v generated by the script you can access under ./synopsys/scripts/top.tcl. Let us read in this netlist into SYNOPSYS TETRAMAX ATPG and try to see what the test coverage is.

Student Task 19:

- Execute the same commands as you did in the first part. The following commands need to be adapted for the design used in this task:
  
  tmax> read_netlist ./synopsys/netlists/top.v
  tmax> run_build_model top

- Generate a fault dictionary and run a standard (non sequential) atpg generation using
  
  tmax> add_faults -all
  tmax> run_atpg -auto_compression

- What is the coverage, what seems to be the problem?

Figure 3: Simplified RTL diagram of the FIR filter. The path of a single scan chain is sketched with a dashed line.
Our leading problem at the moment is the large SRAM block that acts as a large sequential block which is not part of the scan chain. In Figure 4 a simplified version of your circuit is shown to illustrate what causes the problem with the memory.

The fact that the memory is not part of the scan chain is annoying, the circuit is effectively converted to a partial-scan design, making testing more difficult. However, there is a second problem as well: although SYNOPSYS TETRAMAX ATPG is able to recognize the memory, it is unable to use it, as the write enable input is not stable during the scan phase. In other words, while a test vector is being shifted in serially, the content of the memory could change randomly, and SYNOPSYS TETRAMAX ATPG loses control of this element. This prevents it from being used in a partial-scan environment as well.

This results in two problems: We can not control the outputs of the memory which affects the post-memory stage, at the same time we can not observe the inputs of the memory which affects the pre-memory stage. This accounts for most of the ATPG untestable (AU) faults. If we do not do something about it, our test coverage will be unacceptably low.

First of all, there is no universal method to address such problems, every design has its own peculiarities. Depending on whether or not there is combinational logic before or after the black box your solutions will be different, see Fig 5. If a port of a black box is already directly connected to a register, it is already as much controllable or observable as it will be. It makes no sense to add additional test circuits in these cases. The same applies if ports are directly connected to inputs or outputs of the chip.

---

8 Part of this problem is artificial, as in the current design flow we are not yet using the capabilities of SYNOPSYS TETRAMAX ATPG properly. We are working on an approach that will alleviate this problem. As soon as we verify this approach, we will include it in the design flow.

9 There are some RAM macros which have built-in scan chains for their inputs and outputs, these can then be stitched to the rest of the circuit. Such RAM macros do not have this problem. Unfortunately, the RAM macros we are using do not have this property.

10 As you have just seen.

11 SYNOPSYS TETRAMAX ATPG reports this with Warning: Rule S30 (unstable RAM during test procedure operation) was violated 1 times during the run_drc command.

12 This is the part which we hope to fix in the near future. Unfortunately for this exercise we will also have to live with this problem.
Figure 6: The browser of the SYNOPSYS TETRAMAX ATPG is able to display graphically the distribution of faults according to the modules.

7.2 Block Isolation

If you cannot get SYNOPSYS TETRAMAX ATPG to recognize part of the circuit, you can always add circuits that will isolate the block away from the rest of the circuit. In this way, you can use SYNOPSYS TETRAMAX ATPG to test one part of the circuit, and develop an additional method to test the block you have just isolated. We will discuss solutions for both parts in this part of the exercise.

**Student Task 20:**
- Have a look at the block diagram of our circuit. Suggest a solution as to how the RAM block can be bypassed.
- What signal would you use to control this bypass solution?
- Discuss your results with an assistant.

Note that this is not the only solution that could be used (see Fig 5). We will now add the isolation that you have discussed with your assistant into the circuit at HDL level.

**Student Task 21:**
- Make the necessary changes to the ./sourcecode/top.vhd to add block isolation.
- Compile the new design using the SYNOPSYS DESIGN COMPILER script ./synopsys/scripts/top.tcl, (you can make changes if necessary).
- Read in the new netlist that you have just created into SYNOPSYS TETRAMAX ATPG, complete the setup and run a standard (non sequential) atpg generation using run_atpg -auto_compression. You may need to add an additional constraint:

```
tmax > add_pi_constraints 1 signal_that_should_not_change_for_test
```

- What is the fault coverage?
- Which faults are not yet detected?

To browse the fault coverage graphically, you can also use the HIERARCHY BROWSER button on the GUI (see Fig 6). This is very convenient when trying to locate circuit parts with poor test coverage.

These tools are very useful for finding problem areas and can help you pinpoint local problems easily.
7.3 Inserting Test Points

We can now add targeted test points into the circuit to improve the test coverage even further. From our previous analysis we know that we have problems with the RAM address bits. We will now pursue option SO1 from Fig 5.

There are two ways to insert test structures: One way is to add them manually by modifying the VHDL source code. However, in that case one needs to make sure that the test structures get not optimized away during compilation. This is especially the case if test points are added as shown in SO1 and SC1 in Fig 5. These registers have no functionality if they are not used as part of a scan chain. If those registers are inserted manually in the VHDL code, you need to add compiler directives into the compilation script to tell the compiler not to optimize those registers away.

The other way to insert test structures is by using Synopsys Design Compiler directives. You can specify at which nodes you like to add which kind of test structure and they will be inserted automatically along with the scan chain (for solutions SO1 and SC1).

This exercise provides just a short introduction to this topic, we suggest that you read the DFT Compiler User Guide which is part of the synopsys online documentation. The section of interest is Inserting Test Points from page 7-4 to 7-9 which discusses different test points.

You will realize that we need an additional signal to activate the multiplexers in our test structures. We will share the signal that we used for the block isolation for this purpose.

Student Task 22:

- We suggest to close both the Synopsys Design Compiler and Synopsys Tetramax ATPG to have a clean start.
- Start Synopsys Design Compiler and follow the same steps as before until just before you run the create_test_protocol command.
- Next we need to specify the TestMode pin (assuming it is called TestModeEn_TI):

  \[
  \text{dcs} > \text{set_dft_signal} \ -\text{view} \ spec \ -\text{type} \ TestMode \ -\text{port} \ TestModeEn_TI \ \text{active_state} \ 1
  \]

- Now insert an observe test point on the address input of the memory:

  \[
  \text{dcs} > \text{set_test_point_element} \ \text{dcs} > \ -\text{clock_signal} \ \text{Clk_CI} \ \text{dcs} > \ -\text{control_signal} \ \text{TestModeEn_TI} \ \text{dcs} > \ -\text{type} \ \text{observe} \ \text{dcs} > \ [\text{get_object_name} \ [\text{find pin} \ -\text{hierarchy} \ *i\_SYKA65\_512X26X1CM4\_1/A*]]
  \]

  The command requires a list of pins, which we get with a nested find command:

  \[
  \text{dcs} > \text{find pin} \ -\text{hierarchy} \ *i\_SYKA65\_512X26X1CM4\_1/A*
  \]

- Now you can complete the design flow and write out a new netlist. If you are interested you can investigate how the test point was inserted using the Design Vision.
- Start Synopsys Tetramax ATPG, read in the new netlist and analyze the test coverage.
- Were you able to remove the errors you have identified?

This was just an example, of how you can target specific fault coverage problems and add sufficient control and observe points to detect them. Obviously every test point will come at a price, and you should consider the cost vs benefit with such approaches.

---

13 If you are working on a product, economics of test would dictate a certain test coverage that reduces the probability of delivering defect units to the customer to an acceptable level. In this exercise we just want to show you different possibilities.

14 We could have also used a different combination of solutions from the start, i.e. we could have just well added only SO1 and SC1.

15 A brute-force method is to tell Synopsys Design Compiler to not remove registers that have constant values. This can be done before starting the compile process by the following setting: set compile_seqmap_propagate_constants false, however this may stop optimizations in other parts of the circuit.
7.4 Built-in Self Test

So far in this exercise we have improved our test coverage by isolating the SRAM block. However, so far we have no solution how to test the memory itself. We will first see why this is important, by making a simple defect level calculation. From the class and exercise 1 we know that the portion of defect chips sold $D_L$ is given by

$$D_L = 1 - y_f^{1-F_C}$$

(3)

where $y_f$ is the fabrication yield and $F_C$ is the fault coverage.

Student Task 23:

- Assume that you have 100% fault coverage reported by SYNOPSYS TETRAMAX ATPG, and that your fabrication yield is 80%.
- By examining the area reports under the ./synopsys/reports directory, determine the area of the macro and the area of the rest of the circuit.
- Determine the aggregate fault coverage considering that you have 0% coverage for the memory and 100% for the rest of the circuit.
- Report the defect level $D_L$.

* Assume that the memory has approximately the same number of faults per area as the remaining circuit.

We hope it is clear why we need to come up with a solution to test the isolated blocks (in our case the SRAM) as well. We will investigate a BIST solution in this exercise.

To develop a BIST the following steps are required:

- Definition of a fault model for the design under test
- Development of a test strategy.
- Implementation of the BIST circuit.
- Functional verification of the BIST: Guarantee correct detection of both erroneous and error free circuits, according to the fault model.

Note that BISTs come with a significant control overhead: Often a special BIST operation mode is need for your circuit and you need a way to access the BIST results.

You will now develop a BIST for the single port memory in our circuit. We have prepared a skeleton for your BIST entity (sourcecode/BIST_single_port.vhd) which you should use as a base for your implementation. The BIST entity is embedded into our circuit as shown in Figure 7.

The control protocol of the BIST works as follows: TriggerBist_SI is asserted for one cycle to trigger the BIST to start. This trigger signal is only sent if the BIST is ready, which is indicated by Ready_SO=1. When this signal is high, the BIST is in normal operation mode meaning it is fully transparent and no tests are running, such that the memory can be normally used by the surrounding logic. If the BIST is triggered, the Ready_SO is deasserted until the BIST is finished with testing. If the BIST is finished with testing and asserts Ready_SO, it must present the BIST result at the BISTSuccessful_S port: 1 for no error, 0 for error.

This control protocol is already implemented in the skeleton, take some time to understand it before you start with your implementation. Also read the comments for detailed understanding.

We have also prepared a testbench (sourcecode/tb_BIST.vhd) that instantiates both the BIST_single_port and a special memory entity (BIST_mem) that is used to simulate a memory with errors.

The testbench can be started by executing

```
cd modelsim
./run_BIST_test.csh
```

in the modelsim folder. This scripts also compiles the entities.

When the testbench starts a series of test cases are run: First, an error free memory is simulated (bug level 0) to test if your BIST is able to detect this case. Afterwards several different memory bugs (bug level 1-5) are simulated and the BIST result is compared against the expected result. With each step, the bug will get harder to detect. The bugs simulated at higher levels are designed to avoid detection by simple test strategies.

16 There are many other solutions. For example you could expose the inputs and outputs of the memory directly to outside, allowing arbitrary patterns to be written to and read from the memory.
Student Task 24: Implement a BIST that provides a correct result at least until bug level 4. Use the following guidelines how to approach this task:

- Come up with a test strategy
- Draw a state-diagram for your BIST
- If you are unsure, consult an assistant before starting the implementation
- Implement & Debug

Note that the BIST solution will use more area compared to the no BIST solution.

Student Task 25: Briefly compile your `BIST_single_port` entity to get an estimate of the area cost of your BIST solution. Use the `synopsis/scripts/top.tcl` script as a basis. Compare the relative increase in area with the defect level you will have if your BIST would have a test coverage of 90%. 

Figure 7: Circuit with BIST
8 For your own Designs

After the back-end design you will have to generate the final test vectors for testing after manufacturing. You can basically execute the same commands you did in this exercise with two main changes:

- You need to load the netlist you have extracted from CADENCE INNOVUS
- You have to store your test vectors in the end.

In this section we will give you some pointers about some specific problems.

**Note:** You can use the sample script "./tetramax/scripts/tmax.tcl" and adapt it for your project.

8.1 Using a Design with Pads (chip-level)

In this exercise we have used a netlist generated by SYNOPSYS DESIGN COMPILER. You will also have to go through this step for your own design to evaluate the quality of your DFT solution. However, once your design is finished, and you have your final netlist you will have to use the netlist generated by CADENCE INNOVUS for generating the actual test vectors that you will use.

The one important difference that you will have is that by default, all external connections in the pads will be of type inout, even if you have configured the pads to be used as only input or only output. This causes an issue during DRC phase. Specifically SYNOPSYS TETRAMAX ATPG wants all the tristate-able inputs to be driven to high-Z initially. This can be solved by either of the following methods:

1. Add a primary input (PI) constraint for all the scan chain outputs driving them to Z. For example:

   ```
   tmax > add_scan_chains chain1 DataIn_DI[0] DataOut_DO[0]
   tmax > add_pi_constraints Z DataOut_DO[0]
   ```

2. Alternatively, before executing DRC with run_drc write out the configuration to a STIL (Standard Test and Interface Language) file. This file will automatically contain an initialization that drives the outputs to Z by default. You can then proceed to the DRC phase by reading in the same STIL file as demonstrated below:

   ```
   tmax > write_drc_file chip.stil -replace
   tmax > run_drc chip.stil
   ```

8.2 Generating Test Vectors

So far we have always generated some test vectors, but have never written them out. At the end you will need to store the generated test vectors in a file. There are several different formats that can be used, but the one that works well in our design flow is to use the following command to write out the test vectors.

```
 tmax > write_patterns name_of_vector_file -replace -internal \
       -format wgl_flat -sorted -order_pins
```

Follow the guide in [http://eda.ee.ethz.ch/index.php/Tester_ATPG_Vectors](http://eda.ee.ethz.ch/index.php/Tester_ATPG_Vectors) to convert these vectors into a format that can be read in to the tester software.

Note that these vector files can be very large, please do not keep multiple redundant copies, and consider compressing them to save disk space.

---

17 Since these have been defined as outputs only, there would normally never be a signal driving them externally.
18 This sounds silly, but it works, since the STIL file automatically adds the Z definitions to the scan chain outputs.
9 Final Words

We have not solved all issues of the design, but we tried to show what steps have to be taken so that a workable solution can be found. Depending on your circuit you will need to spend more or less time on a testing solution. Here is a small summary that can be used as a guideline:

- **Design for Testability**
  This was a good example to demonstrate that design and testability cannot be completely separated. Some design decisions have implications on testability and test coverage can be increased by re-design. It is important to evaluate the testability of the circuit early in the design process and if necessary make functional additions to aid in testing (yes, we know that we are repeating ourselves).

- **Full-scan**
  Use full-scan whenever you can. If you opt for a partial-scan solution, make sure that you analyze the test coverage properly early in the design so that necessary modifications can be made in time.

- **Fancy additions make testing difficult**
  Reset synchronizers, macro blocks, multi-clock domains, asynchronous clock domains, tri-state signal, and bi-directional busses all bring some overhead to the testing process. Of course, you will have to use them when necessary. However, you should be aware of the implications for testing, and find solutions for test problems. You can not sell a chip which you can not reliably test.