IDESA

Implementation of widespread IC design skills
in advanced deep submicron technologies
at European Academia

Advanced Digital Physical Implementation Flow
Lecture 3b: RTL Synthesis – RTL Compiler
27/10/2008

Mark Wilmott - STFC

Science & Technology Facilities Council
Rutherford Appleton Laboratory

IDESA

Copyright Statement

Outline

- Flows
- Tool Setup
- Importing The Design
- Timing Constraints
- Physical Constraints
- Optimization
- Design Output
- Analysis and Debugging
- Fixing Problems
- Power Optimisation
- Scan Test
- Compressed Scan Test

Flows

Synthesis Flow from RC

- Pro's
  - Enables customised synthesis script development
  - Supports advanced test and power flows
  - Enables interactive constraint development
  - Enables PnR correlation through report_qor flow
- Con's
  - No access to physical information

Synthesis Flow from SOCE

- Pro's
  - Basic synthesis for early floorplan prototyping
- Con's
  - No interactive development of scripts or constraints
  - Constraints must be developed in standard RC synthesis flow
  - Fiddly to debug
  - Tedious to synthesise complex designs with large numbers of files

Flows

- Initial PLE Synthesis (RC)
  - SDC Timing Constraints development
  - Synthesis flow script development
- Floorplan Development (SOCE)
  - DEF floorplan output
- Refined PLE Synthesis (Using DEF) (RC)
  - Enables RC SOCE placement correlation (report_qor)
  - From RC, running report_qor transfers the design to SOCE, performs placement, and annotates parasitics and layout back into RC for debug.

Tool Setup - Starting RTL Compiler

RTL compiler command line shell	`rc` (Can run faster – more important for bigger designs)

RTL compiler debug gui	`rc -gui`
	Can also start and stop the GUI
	`gui_show`
	`gui_hide`

Tool Setup – Initialization files

- Most configuration settings can be stored in a number of different ".synth_init" configuration files.
- These are TCL script files, and they are located in:
  - Tool installation directory (Tool version specific defaults)

    `$SOCE/etc/synth/master.synth_init`
  - In your home area in the .cadence directory:

    `~/.cadence/.synth_init`
  - Current Directory (when tool was started)

    `./.synth_init`
- RTL Compiler loads each file in the sequence above.
  - E.g. The settings in the latter files take precedence over those in the earlier files.

Tool Setup – File Hierachy

- RC maintains a "virtual file hierarchy"
  - Contains data about the
    - Synthesis Libraries → /libraries
    - HDL Code and Libraries → /hdl_libraries
    - Design → /designs
  - The virtual hierarchy can be access with the "ls", "cd" and "pwd" commands much like normal Unix hierarchy
- The standard unix hierachy can still be accessed:
  - Using special commands such as "lls", "lcd" and "lpwd" commands.

Tool Setup - General Settings

- The default behaviour of key commands give you minimal feedback.
  - You can find out more by increasing the information level:

```
set_attribute information_level 7
```

- It is always useful to be able to analyze the inference of sequential elements, and datapath components
  - We can enable line monitoring to keep track of the inference:

```
set_attribute hdl_track_filename_row_col 1
```

    - NOTE: This must be done prior to reading in the HDL code.
  - This then enables HDL line information for the "report datapath", "report sequential" and "report power –rtl" commands

Tool Setup - General Settings

- Preventing missing instances in your design hierarchy
  - RC by default just warns about missing instances, and it will let you continue with an incomplete design.
  - You can tell RC to fail the elaborate stage is the design is incomplete:

```
set_attribute hdl_error_on_blackbox 1
```

- Preventing the inference of latches
  - RC by default just warns about latch inference
  - You can make RC issue an error if your code infers a latch:

```
set_attribute hdl_error_on_latch 1
```

Tool Setup - Libraries

- Library Search path
  - A list specifying all the locations for the design kit library files (in .lib format)

```
set_attribute lib_search_path "./lib /lib_path"
```

- Target library
  - A list of the standard cell libraries (these are the library .lib files)

```
set_attribute library "core.lib io.lib"
```

- Specify the Worst Case Operating Condition
  - You need to specify the operating condition in RC, especially when you are using older timing libraries, which have both BC and WC operating conditions in the same library.
  - You can find out the available operating conditions in the timing libraries with the following command:

```
find /lib* -operating_condition *
```

  - Then specify the desired operating condition:

```
set_attribute operating_conditions worst_case /
```

Tool Setup – Physical Libraries

- Physical Layout Estimation (PLE) Setup
  - To Enable PLE
    ```
    set_attribute interconnect_mode ple
    ```
  - Load the LEF layout abstracts:
    ```
    set_attribute lef_library "tech.lef core.lef"
    ```
  - Load the interconnect RC extraction models
    ```
    set_attribute cap_table_file "core.captable"
    ```
  - Checking the PLE setup once the libraries have loaded (after design elaboration):
    ```
    report ple
    ```
- Additional Settings:
  - Use Area units from the lef files (will only make a difference if the lef and liberty area units differ.)
    ```
    set_attribute use_area_from_lef true
    ```
  - Check the libraries being used
    ```
    get_attribute lef_library /
    ```

Importing The Design - Analysis

- HDL Search Path
  - Search path for the design files (in VHDL / Verilog / SystemVerilog)

```
set_att hdl_search_path "./src ../src"
```

- Analyze the RTL code into the tool

```
read_hdl file.v
read_hdl -vhdl file.vhd
read_hdl -library my_lib -vhdl file.vhd
```

- Alternate revisions of verilog are also supported:

```
read_hdl -v2001 file.v
read_hdl -sv file.sv
read_hdl -netlist netlist.v
```

- At this point the design has been loaded into the tool, but it doesn't yet exist as a netlist, since it has not been elaborated.

Importing The Design - Analysis

- The read_hdl command doesn't give any output unless you have problems with your code.
  - Either it will analyze correctly, or the tool will give errors and will reject your code.
  - This is much like the elaboration stage in a simulator.
- If there are problems with your code then you will get syntax errors
  - These errors tend to be somewhat terse
  - You are usually better off debugging these sorts of errors in your simulation tool.
- Note: RC only supports one unique namespace for entities and modules.
  - This means that defining multiple architectures is not supported.

Importing The Design - Elaboration

- Elaboration converts the analyzed design from the work library into an in-memory netlist.
- This is the "Synthesis Process"
  - Converts the analyzed RTL into a generic netlist
  - Links the elaborated netlist to find all of the instances.

```
elaborate
```

- or

```
elaborate my_entity
```

- If your design elaborates correctly then you will get a simple "Done elaborating …" message.
  - Ensure that you check the final output of this command, to verify that the design linked correctly.
  - E.g. That all design instances were found.
- You can now interact with and view (in the GUI) the generic netlist at this stage

Importing The Design – Design Checking

- Elaborate doesn't give us much information about the elaborated design
- The first time you elaborate your code you will want to check for the correct interpretation:
  - Correct register inference

    ```
    report sequential -hier
    ```
  - Correct datapath inference

    ```
    report datapath -all
    ```
  - Latch inference
    - From the elaboration report
  - Note: These commands rely upon the following variable:

    ```
    set_attribute hdl_track_filename_row_col 1
    ```

Importing The Design – Design Checking

- Use the check_design command to verify the design integrity:

```
check_design -all
```

- Checking for functional problems.
  - Pins not connected
  - Feedthroughs
  - Reporting inputs and outputs
  - Missing instances

Importing The Design – Unloading The Design

- It is possible to unload an elaborated design (and all sub-designs) from memory using the following command:

```
rm /designs/*
```

- Once you have removed the design from memory, you will need to re-read the HDL files if you want to elaborate the design again.
- If you only want to reset the design constraints you can do this separately:

```
reset_design
```

Timing Constraints - Loading Timing Constraints

- Loading SDC Timing Constraints:

```
read_sdc design.sdc
```

- All SDC units are specified in terms of library units
  - Usually the library time units are specified in nanoseconds
- However all timing units used directly in RC are specified in absolute values:

Time	- picoseconds
Capacitance	- femtoFarads

- Can directly enter the SDC commands into RC by prefixing all SDC command with “dc::” e.g.

```
dc::set_input_delay …
```

Timing Constraints - Constraints Checking

- Ensure that the minimum required constraints have been applied
  - Clocks, input (delays and drives), output (delays and loads)
- Checking that the constraints have been applied correctly

```
report timing -lint
report timing -lint -verbose
```

  Note: This will also report on any timing exceptions
- Check that your clocks have been constrained and modelled:

```
report clocks
```

- Check that your Input/Output ports are constrained for delays, drives and loads:

```
report port *
```

- Any path exceptions that are reported, then be debugged with:

```
report timing -exceptions /des/*/tim*/excep*/…
```

- Checking for broken timing loops:

```
find / -instance cdn_loop_breaker*
report_cdn_loop_breaker
```

  - Note: This command comes from the load_etc.tcl scripts

Timing Constraints - Multi-Mode Timing Constraints

- Loading Mult-Mode timing constraints
  - Define the timing modes:

```
create_mode -name {functional test bist}
```

  - Read at least one timing constraints file for each mode:

```
read_sdc -mode test core_test.sdc io_test.sdc
read_sdc -mode bist core_bist.sdc io_bist.sdc
read_sdc -mode functional core_func.sdc io_func.sdc
```

- Subsequent commands will analyze and optimise over the multiple operational modes.
  - Analysis can be performed with the –mode switch:

```
report timing -mode test
```

  - SDC files for a particular mode can be saved out:

```
write_sdc -mode test > timing.test.sdc
write_sdc -mode functional > timing.func.sdc
```

Physical Constraints – Basic Constraints

- **Basic Physical Constraints**
  - Check and set the design aspect ratio:

```
get_attribute aspect_ratio /designs/*
set_attribute aspect_ratio 1.5 /designs/*
```

  - Checking/Setting the upper limit on metal layers for routing:

```
get_attribute number_of_routing_layer /designs/*
set_attribute number_of_routing_layers 8 /designs/*
```

- **DEF Physical Constraints**
  - Relies upon quality floorplanning within SOCE

```
read_def floorplan.def
```

  - Typically:
    - Floorplan Initialisation
    - Pin/IO Assignment/Placement
    - Macro Placement
    - No power structures

Optimisation – Generic Optimisation

- Once the design has been elaborated, it exists as a netlist of generic logic cells.
- Generic optimisation improves the quality of the generic implementation:

```
synthesize -to_generic
```

  – Note: If the design has just been elaborated, then running synthesize without the –to_generic switch just performs the generic optimisation

- You can also improve results by turning carry save arithmetic optimisation and high effort optimisation:

```
synthesize -csa_effort high -effort high
```

Optimisation – Mapping and Optimisation

- The technology mapping and optimisation stages are integrated as one stage:

```
synthesize -to_mapped
```

- This maps the design into your technology specific libraries, and optimises the design to meet timing.
  - Note: If the design has been optimised (generic) or already been mapped, then running the "synthesize" command without the –to_mapped switch performs mapping and optimisation
- You can also enable a high effort compile:

```
synthesize -to_mapped -effort high
```

- Finally if all else fails it may be useful to check constraints, and perform and incremental optimisation

```
synthesize -incremental
```

Optimisation – Physical Timing Correlation Flow

- To run the Timing Correlation flow you need to have loaded a DEF floorplan into RC.

```
read_def design.def
```

- Then you can run SOCE to prototype place, and route the design (after mapping and optimisation)

```
predict_qos
```

- Interpreting the Results from predict_qos
  - Congestion/Timing Report at the end of the command
- Timing Analysis

```
report timing -physical
```

Optimisation - Manual Timing Correlation Flow

- In RTL Compiler:
  - Synthesise the design, then:

    ```
    write_encounter *
    ```
- In SOCE:
  - Load the design, written out by RC

    ```
    loadConfig rc_enc_des/rc.conf 0; comitConfig
    ```
  - Load an IO file/Floorplan/Place/Opt the design

    ```
    loadIoFile …
    floorplan …
    placeDesign -noPrePlaceOpt
    timeDesign -preCTS
    ```
  - Write out the DEF floorplan + stanard cell placement

    ```
    defOut -floorplan design.def
    ```
  - Write out the estimated Parasitics:

    ```
    rcOut -spef design.spef
    ```
  - Write out the complete design:

    ```
    saveDesign soce_enc_des
    ```

Optimisation - Manual Timing Correlation Flows

...Continued

- In RC:
  - Re-load the design:

    ```
    read_encounter soce_enc_des/<design>.conf
    read_def design.def
    read_spef design.spef
    ```
  - Analyse the design

    ```
    report timing -physical
    ```

- Warning: RTL Compiler only loads the worst case libraries.
  - If you are using the output of write_encounter, it is important that you load all of the requisite best case timing libraries and captable's when you take your design into SOCE for final Place and Route

Optimisation – Hierarchy

- Ungrouping / Flattening is performed automatically according to area and timing constraints as part of the synthesize command.
- Ungrouping of cells can also be performed manually:

```
ungroup <hierarchy_path>
ungroup [find / - subdesign my_alu]
```

- Hierarchical Ungrouping of cells (flattening) can also be performed manually:

```
ungroup -flatten <hierarchy_path>
```

- Ungrouping of cells by cell count threshold can also be done:

```
ungroup -threshold 100 <hierarchy_path>
```

Optimisation – Hierarchy

- Uniquification
  - This will be done automatically as part of the compile process.
  - However it is also possible to do this manually using the following command:

```
edit_netlist uniquify *
```

- Grouping cells into new hierarchy:
  - For design management

```
edit_netlist group [find / -hdl_inst *add*]
edit_netlist group -group_name add \
                                   [find / -inst *add* ]
```

Optimisation – Advanced optimisation

- Synthesize will automatically perform boundary optimisation by default
  - This will give you the best overall results.
- However you may want to disable this:

```
set_attribute boundary_opto false \
                                    [find /des* -subdesign name]
```

- Synthesize will automatically remove sequential cells and preceding combinational logic if they do not drive any outputs, or are at a constant value
  - You can also disable this if required:

```
set_attribute delete_unloaded_seqs false /
set_attribute optimize_constant_0_flops false /
set_attribute optimize_constant_1_flops false /
```

Optimisation – Advanced optimisation

- Optimize "Total Negative Slack" rather than "Worst Negative Slack"

```
set_attribute tns_opto true /
```

- Force RC to fix DRC errors before timing:

```
set_attribute drc_first true /
```

- Apply custom RC scale factors to improve timing correlation in PLE mode

```
set_attr scale_of_cap_per_unit_len 1.1
set_attr scale_of_res_per_unit_len 0.9
```

Optimisation – Preventing problems with ports

- By default RC will not fix certain issues with the ports in your design:
  - A feed-through port is a port that connects from an input to an output port.
  - A constant output port is a port driven by a constant value, these cause a problem if the same constant drives multiple ports
  - Multi-driver problems occur when a cell pin drives multiple ports without buffering.
- These need to be fixed prior to Place and Route
  - This can be done by specifying:

```
remove_assigns
```

    - This must be run after running the mapping and optimisation process.

Design Output – Writing out the design

- Names of items within the design must be legalized before writing out, so you need to legalize the names to verilog naming rules:

```
change_names -verilog
```

- Write out a verilog netlist

```
write_hdl > file.v
```

- Write out a design database :

```
write_db -to_file file.db
```

- Write constraints out

```
write_sdc > file.sdc
```

Design Output - Writing out the design

- Write out SOCE files and .conf file:

```
write_encounter *
```

  - Creates a directory called "rc_enc_des", in which you will find a rc.conf, rc.v and rc.sdc

- Note: The rc.conf file only specifies the worst case timing libraries, so this will not be good enough for full physical implementation.

Design Output – Formal Equivalence

- Comparison of two designs to determine if they are functionally equivalent by examining the logic structure of the designs.
  - Indicates whether two different versions of a design implement the same functionality.
  - Gives you a pass / fail result
  - Conformal is the Formal Equivalence checker from Cadence.
- You can get RC to write out your netlist, and then generate a script to run conformal on your verilog netlist:

```
write_hdl > a.v
write_do_lec -verbose -revised a.v > a.do
```

- This following command runs Conformal to compare your netlist `a.v` with the original RTL code files that you read into RC.

```
exec lec -LPGXL -nogui -do a.do
```

- Note: You must have setup Conformal prior to running RC

Analysis and Debugging - Design Checking

- Quick design checking
  - Get a summary of all of the main design metrics:
    ```
    report qor
    ```
  - Obtain a summary of any timing/area/DRC issues:
    ```
    report summary
    ```
- Detailed design checking
  - Report on all violating design rules
    ```
    report_design_rules
    ```
  - Report on the worst timing path:
    ```
    report timing
    ```

Analysis and Debugging – Analysis Techniques

- Reporting path slacks (setup only)
- Report the worst path for each path group:

```
report timing
```

- Report the n worst endpoint path for the group myclock:

```
report timing -num_paths n -cost_group myclock
```

- Report the path slacks only:

```
report_timing -num_paths 10 -endpoints
```

- Report the path slacks only, with up to two paths per endpoint:

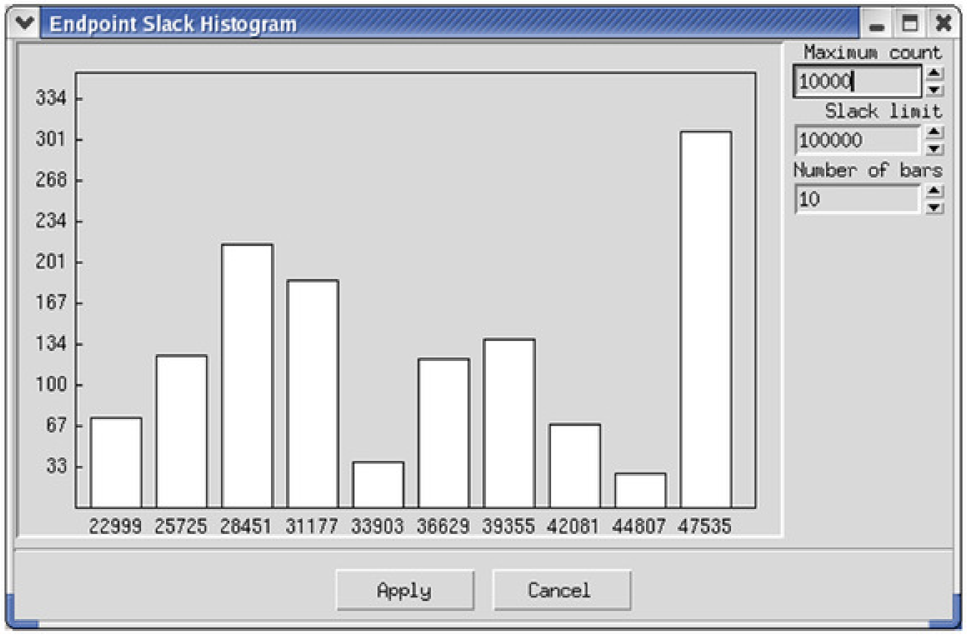
```
report_timing -num_paths 10 -worst 2 -endpoints
```

- Note: Hold timing analysis is not possible in RTL Compiler.

Analysis and Debugging - GUI Analysis

- Endpoint slack histogram
  - A histogram of the worst endpoint slacks
  - E.g. Only the worst slack is considered for each endpoint.
- This can be analyze in the GUI:

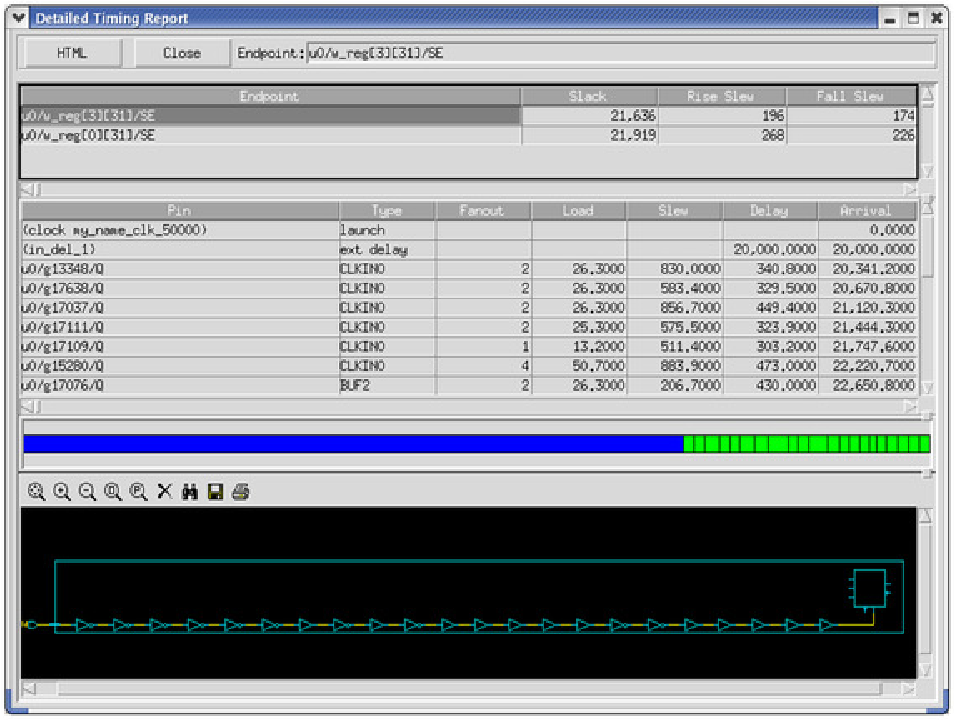
Report → Timing → Endpoint Histogram



Maximum count: 10000
Slack limit: 100000
Number of bars: 10

Analysis and Debugging – GUI Path Inspection

- Path schematic and Path delay profile
  - Visualisation of the path schematic and the delay components of the path



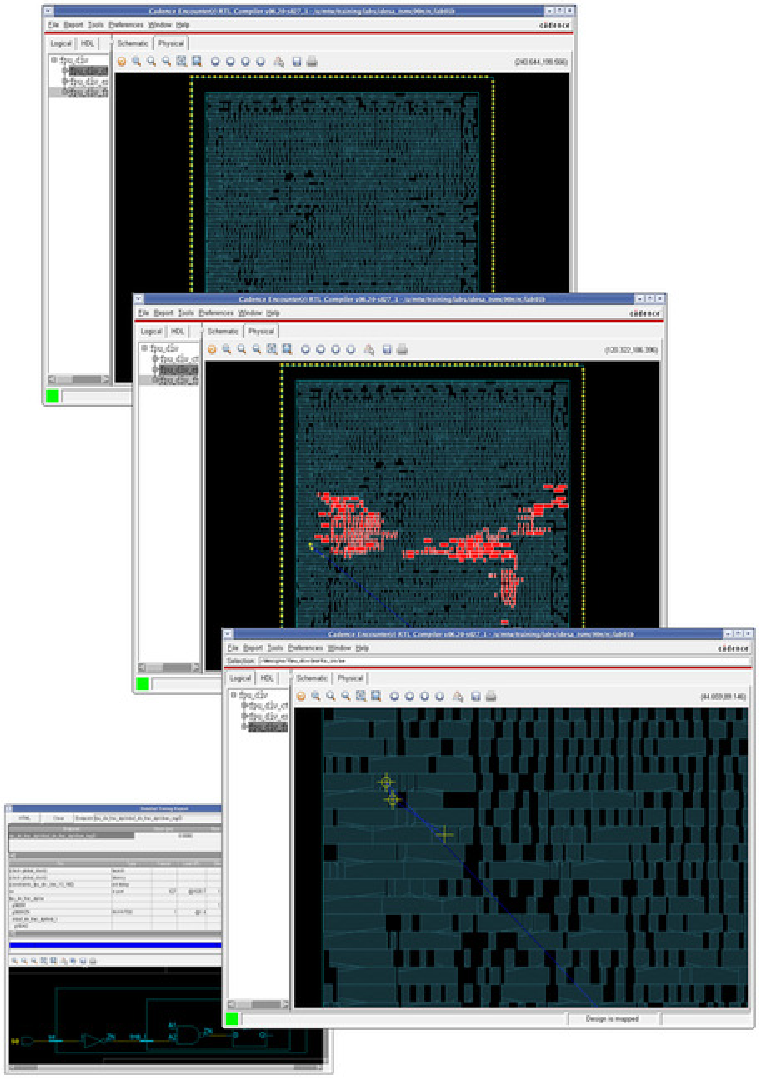
Endpoint: u0/w_reg[3][31]/SE

Endpoint	Slack	Rise Slew	Fall Slew
u0/w_reg[3][31]/SE	21,636	196	174
u0/w_reg[0][31]/SE	21,919	268	226

Pin	Type	Fanout	Load	Slew	Delay	Arrival
(clock my_name_clk_50000)	launch					0,0000
(in_del_1)	ext delay				20,000,0000	20,000,0000
u0/g13348/Q	CLKINO	2	26,3000	830,0000	340,8000	20,341,2000
u0/g17638/Q	CLKINO	2	26,3000	583,4000	329,5000	20,670,8000
u0/g17037/Q	CLKINO	2	26,3000	856,7000	449,4000	21,120,3000
u0/g17111/Q	CLKINO	2	25,3000	575,5000	323,9000	21,444,3000
u0/g17109/Q	CLKINO	1	13,2000	511,4000	303,2000	21,747,6000
u0/g15280/Q	CLKINO	4	50,7000	883,9000	473,0000	22,220,7000
u0/g17076/Q	BUF2	2	26,3000	206,7000	430,0000	22,650,8000

Analysis and Reporting - PLE

- GUI Debug Analysis
  - If you have loaded a DEF flooplan and run the `predict_qos` command you have a number of additional GUI debug options:
  - Physical Layout view in the GUI
    - Note: This only display cells once `predict_qos` has generated an updated cell placement, not from the RC PLE placement!
  - Physical Hierarchy analysis with GUI highlighting
  - Timing analysis with GUI path highlighting



Analysis and Debugging – Reporting Area

- Looking at area statisitics

```
report area -summary
```

- Or a hierarchical area report:

```
report area
```

- It does have some important limitations
  - The areas it provides are assuming 100% utilization
    - It is usual to expect 75% utilization of core cells in the core area
  - Total chip area can be vastly different, due to core utilization values, spacing for power structures, and depending upon whether the design is significantly core or pad limited.

Analysis and Debugging – Resources

- It is often useful to look at the hardware that has been created by the synthesis tool
  - Visual inspection is often unsuitable for this purpose
- Want to report the resources (arithmetic or comparison operators) and data-path operators (grouped resources) that have been used in a design:
  - Data-paths
  - Muxes

```
report datapath
```

- This is the same command as used after elaboration, however mapping/optimisation may change these datapath implementations.

Analysis and Debugging – General

- Other design analysis commands:
- List the ports and their attributes

```
report port
```

- Report the clock specification

```
report clocks
```

- List the design hierachy

```
report hierarchy
```

- Report on all of the instances used in the design, and their associated area, and counts.

```
report gates
```

- Report the design power consumption

```
report power
```

Fixing Problems - Checklist

- Things to check first:
  - Check for structural problems with your design
  - Check for problems with your timing constraints
    - Have you over-constrained your design?
  - Check that your IO paths aren't over-constrained
    - Check that in to out paths, do not have impossible constraints.
  - Visually analyze the failing paths
    - Are they particularly long?
    - What type of paths are they?
      - e.g. Reg to Reg, Reg to Output, Input to Reg, or Input to Output
- Rule of Thumb:
  - After this if you design is still missing your timing constraints by more than 5% then you will definitely have to go back and re-work the RTL code.

Fixing Problems – Check Your Design and Constraints

- Understand the messages
  - Numerous problems can result if you ignore warnings.
- After reading in the RTL
  - Check for problems with the design, including missing instances

    ```
    check_design -all
    ```
- After constraining the design
  - Check for any unconstrained paths, or incorrect timing exceptions:

    ```
    report timing -lint
    ```
  - Check on port and clocks constraints

    ```
    report port *
    report clocks
    ```

Fixing Problems – Check Your Design and Constraints

- After mapping and optimisation
  - Check for problems with the design

    ```
    check_design -all
    report timing -lint
    ```
  - Check and remove any timing loops:

    ```
    find / -instance cdn_loop_breaker*
    ```
- At any time
  - Check the information/warning/error messages

    ```
    report messages
    ```
  - Check any timing exceptions reported by the "`report timing -lint`" command

    ```
    find / -exception …
    ```

Fixing Problems – Improving Timing

- Optimization
  - If you are only failing to meet timing by small margins, then you can try and incremental optimisation:
    `synthesize -incremental -effort high`
  - Note: You May get better results using `synthesize -effort high` from scratch if you didn't use high effort first time.
- Design
  - If you need to change your timing constraints, it is probably a good idea to restart synthesis from scratch.
  - It isn't uncommon to need to modify RTL code if there are long timing paths, or a few poorly designed paths failing timing.

Power Optimisation – Gate Level Optimisations

- Gate level Power Optimisation
  - Reduce dynamic power through gate level transformations:
    ```
    set_attribute max_dynamic_power 0 /designs/*
    ```
  - Reduce leakage power through gate level transformations:
    ```
    set_attribute max_leakage_power 0 /designs/*
    ```
- Increasing the accuracy of gate level power analysis
  - Re-simulate using the gate level netlist
  - Increase the analysis effort level (from the default of medium):
    ```
    set_attribute lp_power_analysis_effort high /
    ```
  - Setting this to “low” effort will prevent any switching activity propagation. E.g. RC will not propagate switching probabilities for un-annotated nets, but just uses default switching probability annotations.
  - It is generally only useful to set this to high if you have design specific switching activity information derived from meaningful simulation

Power Optimisation – Switching Activity Information

- Switching activity Information
  - Required in order to get useful gate level dynamic or leakage power optimisation
  - Generated during "representative" simulation
- Statistical Information
  - Two formats, SAIF (Synopsys "Switching Activity Information Format") and TCF (Cadence "Toggle Count File"):

```
read_tcf -inst /tb/8051_inst 8051.tcf
read_saif -inst /tb/8051_inst 8051.saif
```

  - NOTE: The default behaviour is to overwrite previous annotations.
  - Additional TCF/SAIF files can be loaded by using the `–update` switch on all subsequent `read_tcf` and `read_saif` commands.

Power Optimisation – Switching Probabilities

- **Manual specification of switching probabilities**
  - Switching probability specified WRT to the associated clock:

```
set_attribute lp_asserted_probability 0.5 \
                        /designs/design/*/nets/net
```

  - Toggle rate specified in units of "toggles per nanosecond":

```
set_attribute lp_asserted_toggle_rate 0.02 \
                        /designs/design/*/nets/net
```

- **Full switching information**
  - Using static (statistical analysis) analysis:

```
read_vcd -static -vcd_module \
        /tb/8051_inst 8051.vcd
```

  - Or activity profile analysis:

```
read_vcd -activity -vcd_module \
        /tb/8051_inst 8051.vcd -simvision
```

Power Optimisation – Debugging Switching Information

- The annotation coverage can be checked to ensure that it is correctly annotating into the design
  - Report on the nets which have had switching information annotated:

```
report power -tcf_summary
```

- Checking Switching Activity Annotation
  - By dumping out a TCF file, and then viewing it in a text editor, or comparing it to the origional source TCF files.

```
write_tcf > design.tcf
```

  - By dumping out a TCF file with just the boundary conditions (PI's and Sequential outputs):

```
write_tcf -boundary_only > design.tcf
```

  - To check the default computed probabilities:

```
write_tcf -computed > design.tcf
```

Power Optimisation – Clock Gating

- Enable Clock Gating:

```
set_attribute lp_insert_clock_gating true
```

  - Specify this before elaborating your design.
  - The clock gating insertion process can be controlled:

```
set_attribute lp_clock_gating_exclude true /designs/*
set_attribute lp_clock_gating_exclude false \
                                /des*/*/subdesigns/subdesign
set_attribute lp_clock_gating_exclude false \
                                /des*/design/inst*_seq/name
```

- Specify the minimum CG register count threshold (default = 3):

```
set_attribute lp_clock_gating_min_flops 4 /des*/*
```

- Specify the maximum CG register count threshold (default = 2048):

```
set_attribute lp_clock_gating_max_flops 128 /des*/*
```

- Set/Check the setting of a non-default clock gating prefix (default = none):

```
set_attribute lp_clock_gating_prefix CG_ /
get_attribute lp_clock_gating_prefix /
```

Power Optimisation – Clock Gating

- Clock Gating cell usage
  - RC prefers to automatically insert clock gating logic as per your requirements, preferably utilizing library "Integrated Clock Gating " (ICG) cells.
- This is done by interpreting the following settings:
  - Default clock gating style

    ```
    set_attr lp_clock_gating_style latch  /designs/*
    ```
  - Type of control point

    ```
    set_attr lp_clock_gating_control_point precontrol /designs/*
    ```
  - Note: these correspond with a latch_posedge_precontrol or latch_negedge_precontrol ICG cells
  - You can alternately specify a specific ICG cell to use:

    ```
    set_attribute lp_clock_gating_cell \
                                      [find / -libcell CKLNQD1] /des*/*
    ```

Power Optimisation – Operand Isolation

- Operand Isolation
  - Enable Operand Isolation

    ```
    set_attribute lp_insert_operand_isolation true
    ```
  - Specify this before elaborating your design.
- Set/Check the setting of a non-default operand isolation prefix:
  - `set_attribute lp_operand_isolation_prefix OI_ /`
  - `get_attribute lp_operand_isolation_prefix /`
  - Note: By default there is no operand isolation prefix.

Power Optimisation - Multi-V_{TH} Optimisation

- Reduce leakage power by using low V_{TH} library cells on non-critical timing paths and high V_{TH} library cells on critical timing paths.
- Load Multiple Timing Libraries into RC as you would with any other library
  - One for each V_{TH} Operating Condition
- Enable Leakage power optimisation

```
set_attribute max_leakage_power 0 /designs/*
```

- By default low effort multi-V_{TH} opt is performed, if you wish, you can increase the multi-V_{TH} synthesis effort level:

```
set_attribute lp_multi_vt_optimization_effort medium \
/designs/*
```

- When you run the synthesize command, RC will then perform the requisite multi-V_{TH} optimisation during the synthesis process

```
synthesize
```

Power Optimisation – Advanced Low Power

- Enable SRPG use prior to mapping and optimisation:

```
set_attribute lp_map_to_srpg_cells true /des*/*
set_attribute lp_srpg_pg_driver pwr_ctrl /des*/*
```

- Insert Level Shifters after Mapping
  - Define the level shifters to insert between two particular domains:

```
define_level_shifter_group -from_library_domain \
 library_domain1 -to_library_domain library_domain2 \
 -libcells cell_list
```

  - Insert the level shifters:

```
level_shifter insert
```

  - Check the level shifters:

```
report level_shifter -hier -detail
```

Power Optimisation - Analysis

- Power reporting:
  - Hierarchical power report
    ```
    report power
    ```
  - Estimate the clock tree power:
    ```
    report power -clock_tree
    ```
  - Check the power consumption on a gate by gate basis:
    ```
    report gates -power
    ```
  - Report power of all cell instances:
    ```
    report power -flat
    ```
- Investigating power consumption, by correlating power consumption numbers back to the RTL level (to trace power consumption back to specific RTL constructs):
    ```
    report power -rtl
    ```
  - Note: You need to set the following attribute prior to elaboration to enable RTL power reporting:
    ```
    set_attribute hdl_track_filename_row_col true /
    ```

Power Optimisation - Analysis

- Reporting on Clock gating
  - Basic summary report:

    ```
    report clock_gating
    ```
  - Details of all of the gated flip-flops:

    ```
    report clock_gating -gated_ff
    ```
  - Detailed clock gate report, with per-instance breakdown:

    ```
    report clock_gating -detail
    ```
  - Report on multi-stage clock gating:

    ```
    report clock_gating -multi_stage
    ```
- Reporting on Operand Isolation
  - Standard Operand Isolation report

    ```
    report operand_isolation
    ```

Scan Test – Configuration

- Setup the scan settings (after elaboration):
- Define the scan style (default is "muxed_scan")

```
set_attribute dft_scan_style muxed_scan /
```

Or

```
set_attribute dft_scan_style clocked_lssd_scan /
```

- Define the scan enable (shift enable) and test mode signals
  - Add a scan enable port, connected through an IO pad:

```
define_dft shift_enable -configure_pad port -hookup_pin pad_in se
```

Or directly to an IO port (which needs to be created)

```
define_dft shift_enable -name se -active high se -create_port
```

  - Add a test mode port , connected through an IO pad:

```
define_dft test_mode -configure_pad port
```

Or directly to an IO port (which already exists)

```
define_dft test_mode -name tm -active high tm
```

Scan Test – Configuration

- Setup the scan test settings (after elaboration):
  - Define scan constants
    - Synchronous reset (active low)

```
define_dft test_mode -active high rst_n
```

  - Define scan clocks
    - Define the test clock (period specified in ps)

```
define_dft test_clock -name rclk -period 10000 clk
```

- Check the scan test settings:

```
report dft_setup
```

Scan Test – Configuration

- Define the scan chains by one of these methods: (After Elaboration):
  - Manually defined scan chain IOs with pre-existing ports:

```
define_dft scan_chain -sdi si -sdo so \
       -domain rclk -name chain1
```

  - Manually defined scan chain IO's, creating new top level ports:

```
define_dft scan_chain -sdi si -sdo so \
       -domain rclk  -name chain1 –create_ports
```

  - By specifying the minimum number of scan chains to be created (no default):

```
set_attribute dft_min_number_of_scan_chains 2 /des*/*
```

  - By specify the maximum length of any scan chain (no default):

```
set_attribute dft_max_length_of_scan_chains 100 /des*/*
```

  - Note: If scan inputs are not defined, then they will need to be created by the `connect_scan_chains` command later

Scan Test – Configuration

- Scan Chain configuration (If defaults are not correct)
  - Specify whether to allow mixing of rising and falling edge triggered scan flip-flops from the same test-clock domain in the same scan chain (defaults to “false”):

    ```
    set_attribute dft_mix_clock_edges_in_scan_chain true /designs/*
    ```
  - Specify the type of lockup element to include in the scan chain (defaults to “edge_sensitive”):

    ```
    set_attribute dft_lockup_element_type level_sensitive /designs/*
    ```
- If ports names are not specified, then they will be created automatically.

Scan Test – Test DRC

- Test DRC
  - Checking for:
    - Uncontrollable clocks
    - Uncontrollable resets
  - Check test DRC rules:

    ```
    check_dft_rules
    ```
  - Report detailed DFT violations:

    ```
    report dft_violations
    ```
  - Report on the test DRC passing registers:

    ```
    report dft_registers -pass_tdrc
    ```
  - Report on the test DRC failing registers:

    ```
    report dft_registers -fail_tdrc
    ```

Scan Test – Fixing Test DRC's

- Manually in the RTL code
  - This is generally the best method, since it avoids the need to modify the function of the design unnecessarily
- Auto-Fixing the test DRC's
  - Fix test DRC's by putting additional logic to modify the function during test mode:

```
fix_dft_violations -clock -async_set -async_reset \
                   -test_mode tm -async_control tm
```

  - Preview the changes required to fix test DRC's:

```
fix_dft_violations -clock -async_set -async_reset \
                   -test_mode tm -async_control tm -preview
```

  - The fix_dtf_violations command, needs to control the additional test logic, so that it is only enabled during testing. This is usually via a test mode pin (in the above command it is called "tm").

Scan Test – Fixing Test DRC's

- If you have blocks in your design which are not testable, then you may wish to improve your test coverage by inserting testability logic "shadow logic":

```
insert_dft shadow_logic …
```

- Inserting user-defined control/observe test points, to improve your test coverage:

```
insert_dft user_test_point …
```

Scan Test – Checking

- Before Mapping
  - Check settings are correct

    ```
    report dft_setup
    ```
  - Check Test DRC's are fixed:

    ```
    check_dft_rules
    ```
  - Check the approximate ATPG coverage, using current test setup, to determine if you need to add any shadow logic or control/observe points:

    ```
    analyze_testability
    ```
  - Note: This command uses Encounter Test to run a sampled ATPG run, so you must have setup the Encounter Test search path prior to starting RC!

Scan Test – Single Pass Scan Synthesis

- Scan synthesis
  - Scan synthesis occurs during mapping if you define a scan enable (shift enable) and specify a test clock. The synthesize command will insert scan flip-flops rather than regular flip-flops:

    ```
    synthesize -to_mapped
    ```
- Scan flip-flop mapping
  - During initial synthesis, any flip-flop can be mapped to a scan flop for functional use. To prevent this:

    ```
    set_attribute use_scan_seqs_for_non_dft false /
    ```
  - By default only flops which pass Test DRC will be mapped to scan flops for DFT purposes. You can force all flip-flops to be mapped to scan-flops for early qor estimates (without scan setup) with the following command:

    ```
    set_attribute dft_scan_map_mode force_all /
    ```
    - Note: Even though all flip-flops will then be scan flops, only those passing test DRC will be connected up into the scan chains.

Scan Test – Clock Gating

- Clock Gating Hook-up
  - Clock gating can have a negative test impact unless you disable/bypass the clock gates during test mode.
    - RTL Compiler will do this automatically if you use it to insert clock gating logic and you define a test mode signal.
  - Manual clock gating:
    - If you have manual clock gating logic in your design, then you will need to manually hookup test-mode signals:
    - Define the test mode signal (select either a shift_enable or test_mode)

```
set_attr lp_clock_gating_test_signal tm /designs/*
```

    - Perform scan mode hookup to the clock gating cells:

```
clock_gating connect_test
```

    - Verify that the hookup has worked and that the associated test DRC rules are clean.

```
check_dft_rules
```

Scan Test – Testability Analysis

- After Mapping / Optimisation / CG hookup
  - Check the approximate ATPG coverage, using current test setup

```
analyze_testability -library "core_lib.v \
                              io_lib.v …."
```

  - Note: Once the design is mapped, you need to use verilog models for the test ATPG run, in order for ET to understand the function of the cells.
  - Check the test DRC, it should now indicate that the flip-flops and latches have now been mapped into the technology library:

```
check_dft_rules
```

Scan Test – Scan Chain Stitching

- Scan Chain Stitching
  - Preview the scan connection (based upon pre-defined scan chains):

    ```
    connect_scan_chains -preview
    ```

  - Preview the scan connection, based upon "min chains" or "max length" attributes:

    ```
    connect_scan_chains -preview -auto_create_chains
    ```

  - Connect the scan chains (based upon pre-defined scan chains):

    ```
    connect_scan_chains
    ```

    Or to create ports and chains based upon min_chains or max_length attributes:

    ```
    connect_scan_chains -auto_create_chains
    ```

- Note: Only ever perform scan stitching once, otherwise you may end up with unnecessary logic in your design.

Scan Test – Optimisation & Scan Test Power Analysis

- After Scan Stitching you may have timing or electrical DRC violations.
  - Fix these with:

```
synthesise -incr
```

- Scan Test Power analysis
  - Calculate the power consumption for scan test:

```
report scan_power
```

    - Note: This analysis is based upon the scan clock definition
  - Generate real test vectors for power analysis:

```
report scan_power –atpg –library "core.v io.v"
```

  - Note: This command uses Encounter Test, so you must have setup the Encounter Test search path prior to starting RC!
  - Calculate power using existing test vectors:

```
report scan_power –scan_vectors …
```

Scan Test – Checking and Test Outputs

- Scan Checking
  - Report the scan chains:

    ```
    report dft_chains
    ```
  - Report the DFT setup.

    ```
    report dft_setup
    ```
- Test Output
  - Write out a scan def file for scan re-ordering during PnR:

    ```
    write_scandef > design.scandef
    ```
  - Write out a STIL file for TetraMAX, or a Cadence test file:

    ```
    write_atpg -stil > design.stil
    write_atpg -cadence > design.test
    ```
  - Write out the design database and the verilog netlist:

    ```
    write_db -to_file design.stitched.db
    write_hdl > design.stitched.vg
    ```

Scan Test – ATPG with Encounter Test

- ATPG using Encounter Test
  - Write out a test script for Encounter Test ATPG

    ```
    write_et -atpg -library "core.v io.v" /des*/*
    ```
  - This writes out all of the files required to run Encounter Test on the final netlist out of RC.
  - Perform immediate ATPG test pattern generation if required.

    ```
    exec et -e ./et_scripts/runeta.atpg
    ```
    - Note: This command uses Encounter Test, so you must have setup the Encounter Test search path prior to starting RC!
- Final ATPG
  - The final netlist (from place and route) will be different from the netlist out of RC.
    - Ensure that you re-run the ATPG with this final netlist. This is critical if you perform scan chain re-ordering!

Scan Test – Test Outputs

- Test Abstracts
  - If you are following a hierarchical synthesis flow you will need to write out a number of test abstracts:
  - Test abstract (defines its scan chain architecture):

```
write_dft_abstract_model > design.dft_model.tcl
```

OR

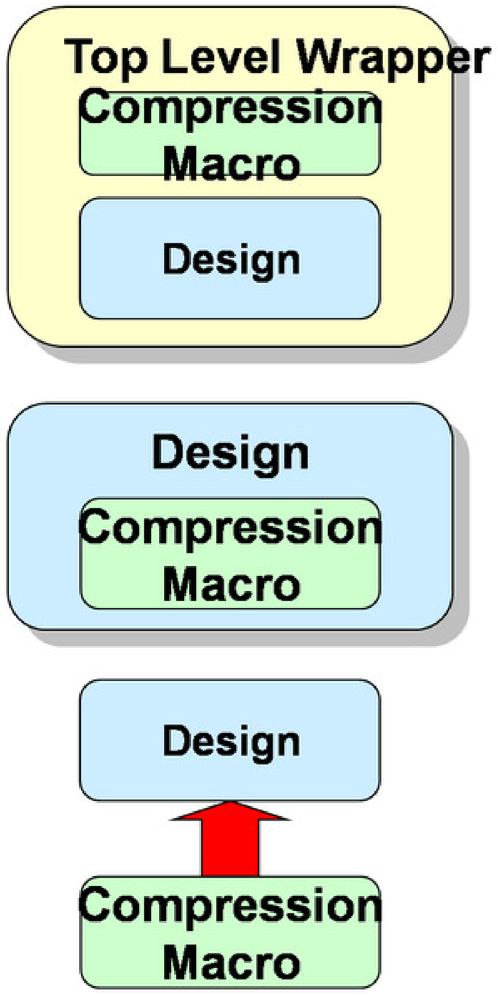
```
write_dft_abstract_model -ctl > design.dft_model.stil
```

  - Note: You can then apply apply a scan abstract model on a subdesign instantiated as a blackbox, logic abstract model, timing model reference, or a whitebox.
  - You can write out an empty test stub for the purposes of hierachical implementation with the following command:

```
write_hdl -abstract > design.stub.v
```

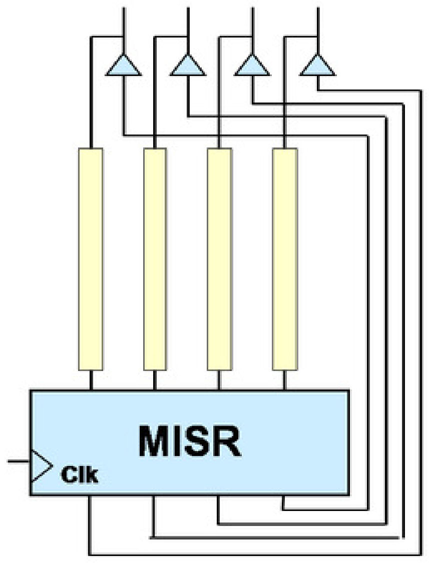
Compressed Scan Test – Design

- Scan Compression Implementation Options
  - Integration of a compression "macro" into a top-level wrapper
    - Automated generation of compression structures and top level wrapper design
    - Leaves core design structure and connectivity intact
  - Integration of the compression macro into the design top
    - Modifies the core design, to integrate a complete test solution
  - Generation of a standalone compression macro
    - Requires manual insertion into the design

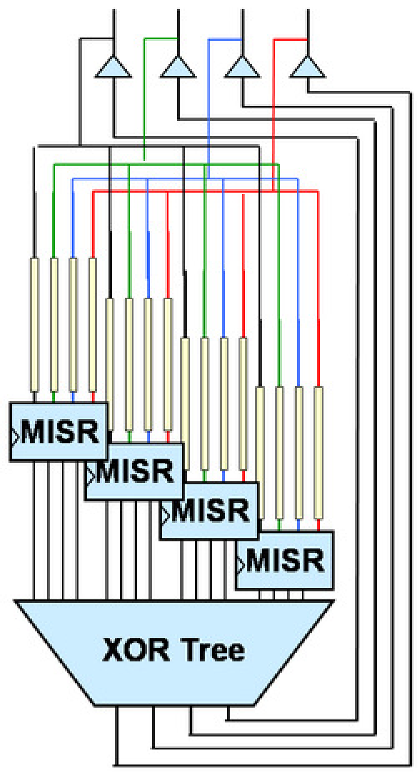


Compressed Scan Test - Architectures

- OPMISR
  - Shared scan inputs
    - Requires top level bi-dir IO pads
  - Scan data scanned in
  - MISR generates a signature
    - During "scan-out"
    - Outputs a signature in parallel through the shared IO's
- OPMISR+
  - As OPMISR, with the addition of:
    - Multiple OPMISR structures in parallel
    - Each being driven with the same inputs
    - Generating a shared signature through XOR compression



OPMISR



OPMISR+

Compressed Scan Test - Encounter Test

- Encounter Test Architect
  - Graphical Based for ease of use.
  - Supports Standard Scan, BIST and others, including Compressed Scan
  - For Compressed Scan:
    - ET generates the logic required for the compression macros and controllers
    - It then generates RC scripts to insert the compression macros
    - Runs RC on the design netlist to insert the test.
- Encounter True Time Test
  - Automatic Test Pattern Generation
  - Integrates with RC, and Encounter Test Architect

Questions / Comments / Feedback

Contact me:

Mark.Wilmott@stfc.ac.uk

Lab Example 1

- 1a) LPE Synthesis with Low power optimisations
- 1b) Placement Correlation flow
- 1c) LPE Synthesis with LP and Scan Test

IC Design Skills for Advanced DSM Technologies