ARM-Cadence Encounter™ Reference Methodology User Guide

CORTEXM0

Release RM7.1USR2

00rel1

Confidential
Cadence Design Systems Inc

MARCH 2009

TABLE OF CONTENTS

REVISION HISTORY

No.	Description	Date
00rel0	RM7.1USR1 production release	March 2008
00rel1	RM7.1USR2 production release	August 2008

REFERENCED DOCUMENTS

No.	Description
1	Encounter User Guide, Product Version 7.1.2

INTRODUCTION

This document describes an ARM-Cadence Encounter Implementation Reference Methodology for implementing a microprocessor core from ARM.

This methodology should allow users to quickly set up and take a core from RTL to GDSII without a detailed knowledge of all the tools used. For expert users, this document describes the steps in sufficient detail to allow users to make adjustments to the flow.

While every effort has been made to make this flow as real as possible, with a tape out quality output, it should be used as a reference only and all signoff criteria should be reviewed prior to hardening for real. Some compromises on Power, Performance and Area may also have been made to ensure a single pass out-of-box replay. With some extra effort on the part of the user this basic flow could be used to achieve better results, but some minor iteration may be required. See the section of flow tuning for ideas.

Much of the flow described is generic and is used in multiple iRM's. This document contains both generic flow descriptions and information specific to the particular target core and library/process used.

iRM Availability

Cadence iRMs for every ARM core are available to download directly from ARM. The flow described in this document is the most up to date of all the iRM flows and any cores not using this flow will be ported across at some point. Some cores will be ported as a matter of course and some based on customer demand. If you want an iRM targeted at a specific core ported you should contact ARM who will work with Cadence to ensure this happens.

OS and Tool Environment for RM

It is important that the correct tool versions are installed prior to running the RM flow, but also that the Operating System (OS) and patches are up to date as defined by the tool release from Cadence.

The user can run ***checkSysConf***, a Cadence utility that ships with all software releases to ensure that the host OS and patches are up to date. If it detects a problem it will generate a report that should be echoed to your system administrator in order to get the OS patches up to date.

Flow configuration

A file called ***cortexm0_config.tcl*** exists in the scripts directory of the installation. This contains references to a number of design and technology specific variables that are used by the majority of the scripts throughout the flow. It is therefore very important to spend time in getting these variables defined correctly prior to running the flow. The installation contains an example ***cortexm0_config.tcl*** file that has been configured for the technology this iRM has been tested on. This shows the user how a correct completed ***cortexm0_config.tcl*** file should look.

Directory Structure

The Reference Methodology directory structure provides a concise and intuitive way of managing the ARM hardening flow. (See Figure 'Directory Structure' Below).

The iRM directories have the following functions:

Data: used to store all design data during the flow.

Reports: all reports are placed here, in the correct sub directory.

Work: the directory where tools run, may contain temp files.

Scripts: all the flow scripts are in here.

Logs: tool logs are written to this directory.

Floorplan: Contains the floorplan of the design.

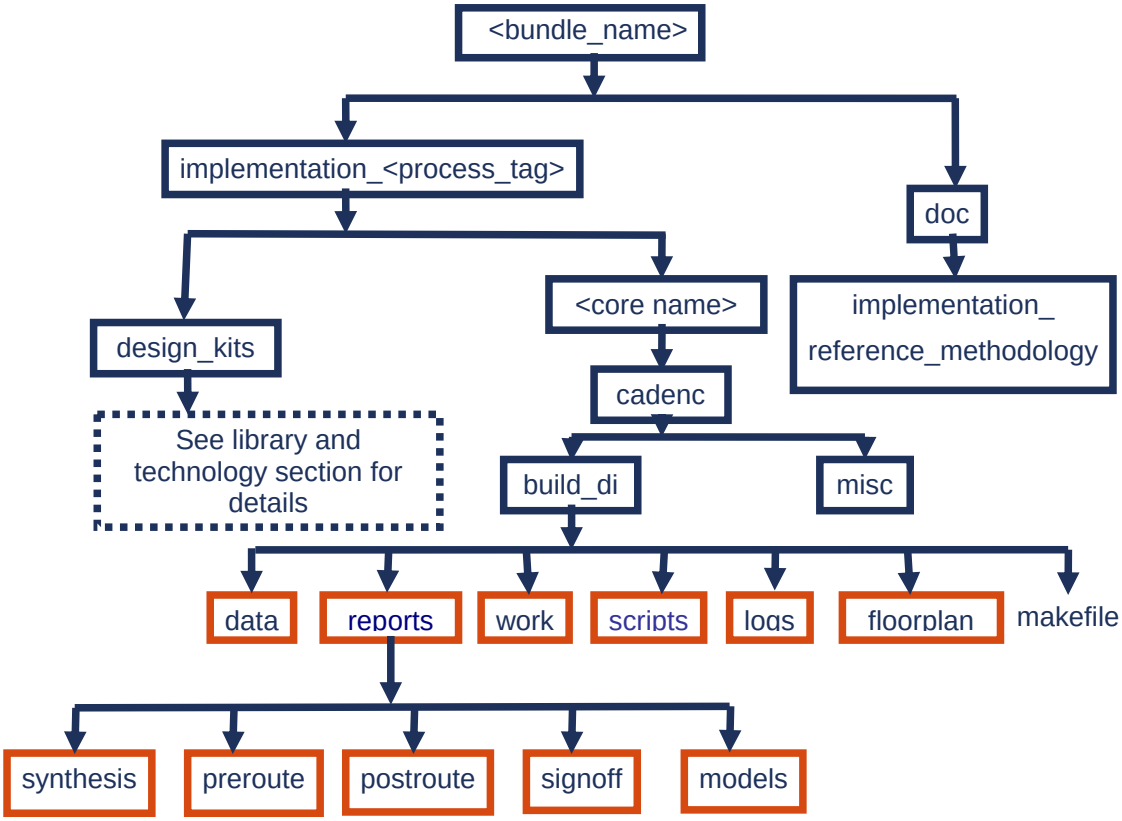


Figure: Directory Structure

To generate a directory in which to run the flow, recursively copy the *core run directory* area (core name) to a run specific title. For example copy **build_dir** to **cortexm0_run1**. The user can now run the flow in the new directory and retain the original directory for reference.

Reference Methodology Overview

The Implementation Reference Methodology (iRM) is a set of scripts that automate the synthesis, implementation, signoff and ATPG of an ARM core. The iRM does not contain the RTL for the core, the Cadence tools or the proprietary process information required to run and 'out of the box' recreation of the flow as tested by ARM and Cadence. The user must ensure that these components are available and correctly installed before running the iRM.

RTL Configuration

Before running the Reference Methodology the user should configure the RTL from ARM to suit their requirements. Some cores require memory configuration and clock gating configuration. Details of how to do this can be found in the Core Implementation Guide. Once the RTL is successfully configured and validated the RM flow can begin. The user should point to the configured RTL in the ***cortexm0_config.tcl*** file via the **rm_rtl_inst_dir** variable. For a true out of box recreation of the flow, refer to the core specific section of this document for the exact core configuration used by ARM and Cadence for their testing.

Tool Installations

Before running the Reference Methodology the user should ensure they have the correct versions of the Cadence tools (see below) and the correct licences for those tools in place. Contact your local cadence representative if you have any issues or questions with this.

This version of the reference flow has been developed and tested with the tools shown in Table 1.

Tool	Version
SOC Encounter	7.10-USR2-s140_1
RTL Compiler	7.2
Cadence Conformal	7.2
Fire & Ice QRC	EXT 7.1
VoltageStorm	ANLS 7.11-s014_1
Encounter Test	6.2.5
Encounter Timing System	7.1
IUS	6.2usr3

Table 1 : Supported Tool Versions

Process Data

The iRM does not ship with any restricted access 3rd party IP. This includes the process data – most often TSMC. This data does not even move between ARM and Cadence! However the user should be able to install exactly the same data the flow was developed with and perform and out-of-box replay. See the process section for details.

Library Preparation

The iRM does come with some ARM library data. However where a library view contains proprietary 3rd party IP that is not obfuscated the view is not supplied. Also certain tasks, especially

in the signoff flow, can be made more accurate if the full front-to-back library drop is available, separately, to the user. See the Library section for how to combine the delivered views with the process data to produce the cadence compiled views. This section also covers some of the ways in which production library data can be used.

Running the flow

Once the above steps are completed the flow itself can be run. The iRM flow is controlled by a makefile located in build_dir directory. All parts of the flow can be initiated from this directory using a 'make' command together with a Makefile target. For example:

make synthesis executes the synthesis task, the command itself changes the working area to work directory and executes the required command for the synthesis.

make full command can be used to execute the whole RM as out of box, it will execute all the tasks from start to finish.

make implementation command can be used to execute the encounter implementation tasks from start to finish.

make delete can be used to delete all the reports, logs and files written during the hardening process

The make command can also be submitted to load sharing systems such as LSF e.g.

bsub make full

See the comments in the scripts and the design flow section for more information on what the flow is doing.

CORTEXM0

This particular iRM is targeted at the CORTEXM0 processor. The CORTEXM0 can have multiple RTL configurations depending on how we configure parameters in the top level RTL file. To allow for the libraries and floorplan provided in this delivery it is necessary to have the same RTL configuration and memory instantiation for an "out of box" experience

CORTEXM0 Configuration

In order to allow configuration of the CORTEXM0, or addition of user I/O to the top level, implementation wrapper is supplied for CORTEXM0 level of hierarchy which is named CORTEXM0IMP and is the name of the top level that is implemented in the iRM. Please see the Configuration and Sign-off Guide for detailed information on configuration of the CORTEXM0.

In order to replay this iRM as tested in Cadence and ARM, customer will need to configure the RTL file CORTEXM0IMP.v to match what we used. Here is the core configuration options used.

Top level: ***CORTEXM0IMP***

RTL configuration options (Defined in CORTEXM0IMP.v)***:***

parameter ACG = 1, // Architectural clock gating

parameter AHBSLV = 1, // SLV port AHB-Lite compliance

parameter BE = 0, // Data transfer endianess

parameter BKPT = 4, // Number of breakpoint comparators

parameter DBG = 1, // Debug configuration

parameter NUMIRQ = 32, // Functional IRQ lines

parameter OS = 1, // Operating system extensions

parameter SMUL = 0, // Multiplier configuration

parameter WIC = 1, // Wake-up interrupt controller support

parameter WICLINES = 34, // Supported WIC lines

parameter WPT = 2, // Number of DWT comparators

Misc Verilog Files

The misc directory in the iRM directory structure contains verilog files that are required to replay the iRM but are either not provided or would require editing in the delivered RTL.

This iRM includes:

Cells- Architectural clock gating module with the clock gate inserted – these would not synthesize in exactly this manner.

LIBRARY AND PROCESS

This section describes the Library and Process specific data structure requirements to make the ARM-Cadence Reference Methodology replay out of the box.

TSMC process data directory structure

The Spice models and QRC Extraction Techfiles should be obtained from the TSMC foundry for cworst, cbest and typical process corners for the TSMC 90nm process.

The TSMC Online website names the zipfiles that are supplied in a unique format which includes the process node name, the spice models referenced and finally a datestamp. In order to obtain similar timing results to those quoted by ARM, it is important that the same versions of the spice models files and QRC extraction techfiles are downloaded from the TSMC website. These files are:

t-n90-lo-sp-002-f1_1_6a_20060914.zip

The Spice models files are:

t-n90-lo-sp-002_1_6_20060807.zip

The names of the zipfiles are used as the sub-directory names under the tsmc directory. The zipfiles will extract to produce multiple tarfiles that will then be further extracted into the metal scheme tarfiles and then the 6X2Z metal scheme tarfiles extracted into the following directory structure so that the Reference Methodology scripts work correctly with minimal editing.

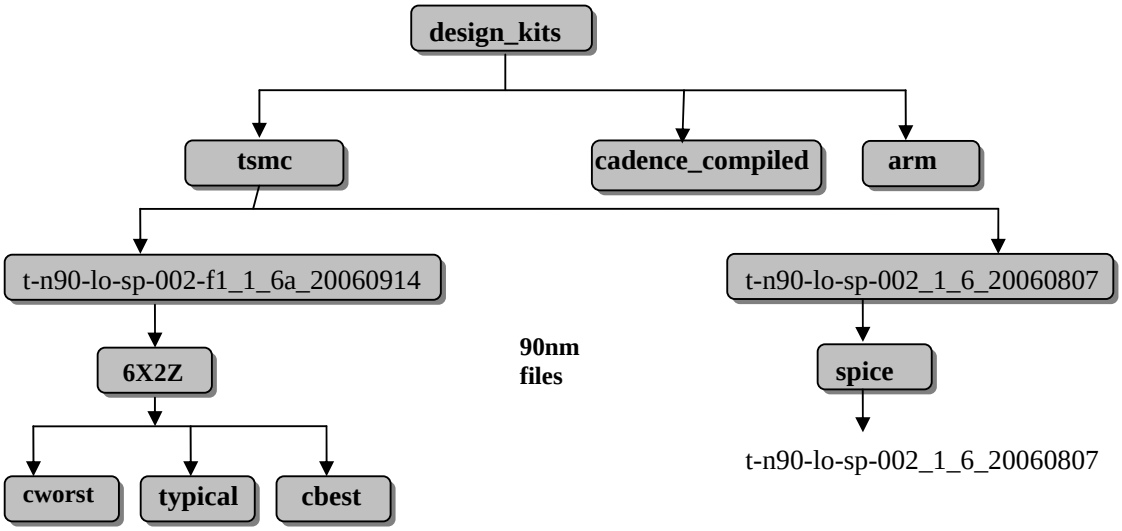


Figure: TSMC directory structure.

The t-n90-lo-sp-002-v1_1_6a_20060914 directory holds the TSMC90G process data. The 6X2Z sub-directory is named after the metal scheme being used for the reference methodology and this will appear in the TSMC tarfile name for the QRC Techfiles. The date stamp at the end of the TSMC zipfiles indicates the release date of the data. The Spice models directory name is similar to the extraction corner files, but has a different release date.

The three process corners used are cworst, typical and cbest. It is required that the user creates a file in the 6X2Z directories called corner.defs that contains the following lines for the TSMC90G process.

```
DEFINE tsmc90g .
DEFINE cworst   ./cworst
DEFINE cbest    ./cbest
DEFINE typical  ./typical
```

These file allows QRC to find the three corner extraction techfile directories and the scripts reference them.

Encounter CapTable creation

Before the reference methodology is started Encounter Cap Tables are required for the three process corners being used. The command generateCapTbl is an Encounter executable and will be found in the SOC Encounter tool install path.

For example, to run the CapTable generation for the 90g cworst process corner the command is:

```
<soce_install_dir>/lnx86/tools/bin/generateCapTbl -64 -ict \
<path_to_design_kits>/tsmc/t-n90-lo-sp-002-v1_1_6a_20060914/6X2Y/cworst/
n90clsp_1p09m_+alrdl_cworst.ict \
-solverExe <soce_install_path>/lnx86/tools/bin/coyote \
-lef   <path_to_design_kits>/arm/fe   fe_tsmc090g-hvt_sc-adv_v10_2007q4v1/
aci/sc-ad/lef/tsmc090adghvt_9lm_2thick_tech.lef \
-output n90clsp_1p09m_+alrdl_cworst.CapTbl
```

(Substitute the actual <soce_install_dir> and <path_to_design_kits> from the users own installation).

This command requires the TSMC 90G process cbest, cworst and typical .ict files, found in the QRC extraction corner directories, and the tech LEF delivered by ARM. This command should be run in each of the QRC techfile process corner directories and it will save the resulting CapTables into the same directory where the scripts are expecting them to exist. The corner name in the .ict files and output files should be edited to match the corner directory where the command is currently being executed. The runtime for each corner may be as long as 10 to 12 hours on an 64bit Opteron machine.

Library data directory structure

The design_kits directory that holds the ARM Standard cells and Macro library data, the TSMC process data and the directory containing the Cadence compiled views, should be arranged in the directory structure detailed below. The cadence_compiled sub-directory names should mirror the arm sub-directory names.

ARM supply CeltIC Noise Libraries and VoltageStorm power views for the PIPD standard cell libraries.

The Noise models are generated using the make_cdb command which requires the TSMC 90G Spice model files and the Liberty timing files. The make_cdb command will be found in the ETS install path.

The VoltageStorm port power views of the generation is done using libgen and requires the LEF files, the rcgenTechfile in the QRC Techfiles corner directories and the LEFDEF layermap that maps the LEF layer names to the rcgenTechfile layer names. While the rcgenTechfile is a binary file, the .ict file can be used to check the metal and via layer names match the Tech LEF file.

The port power views created are useful to show an example power analysis flow in this reference methodology but the results will not be of the highest signoff accuracy that could be obtained if detailed power views were used the libraries. Detailed power views can only be generated when the

GDS data is available for the libraries. To obtain the highest signoff accuracy in the power analysis step using VoltageStorm it is strongly advised to use detailed power views of all the cells.

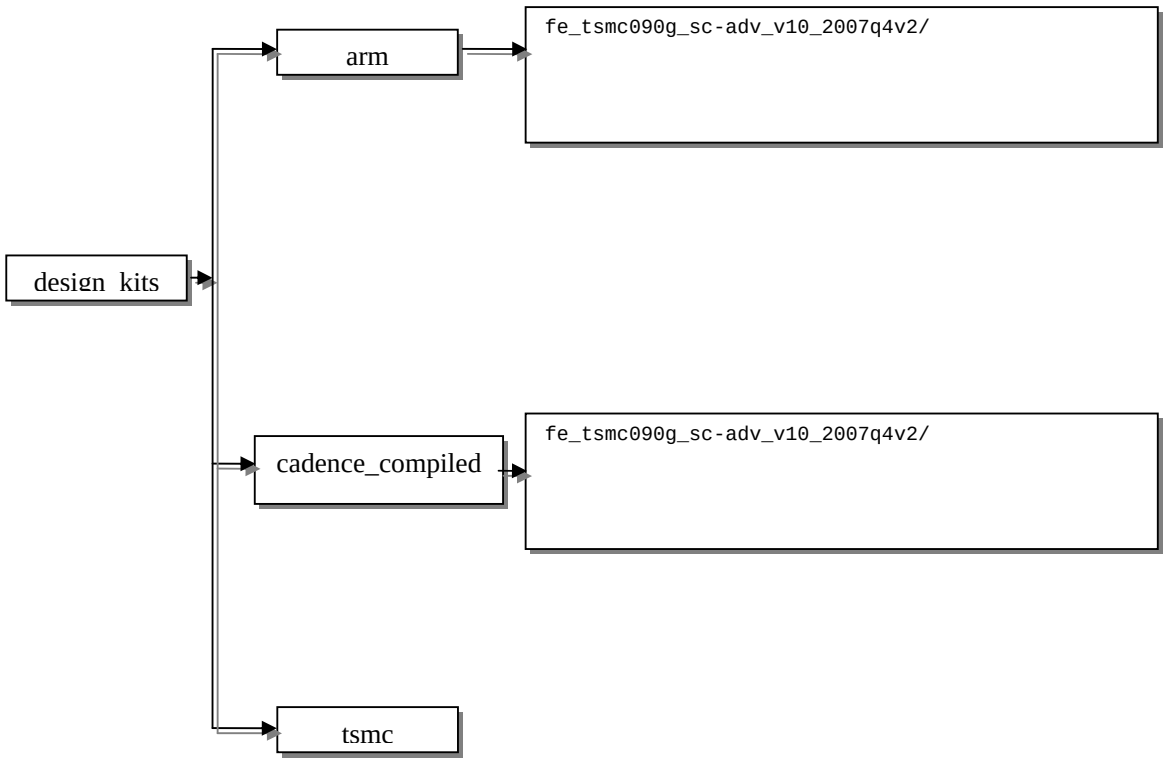


Figure: design_kits directory structure

DESIGN FLOW

RTL Compiler Synthesis

Overview

The first stage in the hardening flow is to take the configured RTL and to synthesise to technology specific gates in order to perform place and route. The initial synthesis of the RTL to gates is performed by Cadence RTL Compiler (RC) using the following script:

scripts/cortexm0_synthesis.tcl

The script performs some or all of the steps in Figure 4, depending on the options selected in the **cortexm0_*config.tcl*** file.

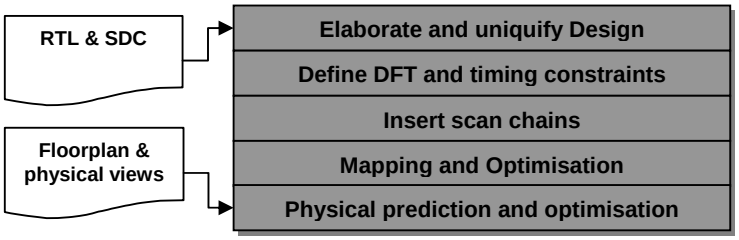


Figure: RTL Compiler Synthesis Flow Diagram

The synthesis script produces a synthesised Verilog netlist and new SDC constraints to use in floorplanning. A formal verification command script (**cortexm0_rtl_vs_netlist.do**) and a setup file for use in the ATPG step (**cortexm0_assign**) are also generated. All of these files are output in the **data** directory.

Initialisation

Reducing runtime by using additional processors

By default the synthesis job runs on a single processor. If the user has a multi-processor machine and suitable tool license, they can reduce runtime by setting the *rm_parallel_cpus* variable in **cortexm0_*config.tcl***. To enable this feature the variable should be set to a value greater than 1 and no higher than the number of processors available on the host machine.

The user will need to check that they can run the UNIX "rsh" command on the host machine without error and without the need to enter their password. RTL Compiler will perform the same checks during synthesis and will revert to single processor mode if either check fails.

RTL Compiler can also be configured to use additional hosts during synthesis if the user does not have access to a suitable multi-processor host or wishes to further decrease runtime. Different operating systems and hardware architectures can be mixed without impacting the quality of the netlist produced. The user should consult the tool documentation for further details and examples of how to use these features

Timing performed using PLEs

The libraries are read and operating conditions are set before the RTL files are read. RTL Compiler will use the LEF files and capacitance table to incorporate physical information into synthesis. It will use Physical Layout Estimators (PLEs) instead of wire-load models, enabling it to produce a better netlist than traditional wire-load models allow.

Specifically, PLEs

- Use bounding boxes and fanout to dynamically derive wire length.
- Calculate load and delay using resistance and capacitance values derived from the process technology information.

Avoiding certain library cells

It is possible to define a set of library cells which are not to be used in synthesis. The environmental variables *rm_dont_use_cells* and *rm_delaycells* define technology-specific lists of cells that are to be excluded.

At the end of the synthesis stage new SDC constraints are written, with these cells marked with the *set_dont_use* construct. The new SDC constraints are read into First Encounter, preventing these cells being used during the implementation flow. The cells in the *rm_delaycells* list might be used during the hold fixing step of the implementation flow.

Low Power Implementation

The synthesis script has the option to read a common power format power intent file and run a low power flow. This flow is provided for information only in general iRMs, however the same script will be used for all future low power iRM's.

Elaborate and uniquify the design

The RTL is read using the *read_hdl* command and a list of files and search paths that are set in ***cortexm0_verilog.tcl***. The user should set the *rm_rtl_inst_dir* variable in **cortexm0*_config.tcl*** to point to the correct top-level directory where the configured RTL is installed.

RTL Compiler uses its own library of synthesisable code to substitute any DesignWare™ used in the RTL.

Once all the verilog RTL files have been read, the *elaborate* command is issued to build the design and ensure all components are defined. The database is then uniquified to ensure best synthesis results are achieved.

A report of any unresolved modules, undriven nets or nets with multiple drivers is generated. The user should check this report for errors as this would indicate a problem with the RTL configuration or the integration of technology specific gates (e.g. RAMs).

Define DFT and timing constraints

The name of the scan enable signal is set in **cortexm0*_config.tcl***, using the *rm_scan_enable* variable. By default the synthesis script also uses this signal to bypass the clock gating latches during test mode. If the core requires other signals to be held constant during test mode, these are set using the *rm_test_high_ports* and *rm_test_low_ports* variables, set in **cortexm0*_config.tcl***.

RTL Compiler checks the design against a set of DFT rules. If a flip flop fails the DFT rules it will not be placed on a scan chain. The user should check the dft report (**cortexm0_preinsert.dft**) any fix any errors reported.

The *rm_no_of_chains* variable, set in **cortexm0*_config.tcl***, controls the number of scan chains created. By default the synthesis script will insert 30 scan chains, without mixing clock edges or clock domains. If the user wants to use mixed edges or different clocks on the same scan chain they should consult the tool documentation for the required settings.

Define timing constraints

Timing constraints are applied as follows:

- Operating conditions are defined immediately after reading the libraries.

- Output pin capacitances are applied by the synthesis script
- Input signal drivers are defined by the synthesis script
- IO timing and loading, maximum slew, any multicycle and false paths and clocks definitions are read from **scripts/cortexm0_<modename>_constraints.sdc**

The capacitive load applied to output pins is calculated using the *rm_driving_cell* and *rm_driving_cell_input_pin* variables, set in **cortexm0_*config.tcl***. The same cell is used to model the external logic driving the inputs pins, this time using the *rm_driving_cell* and *rm_driving_cell_output_pin* variables.

Note that the input and output signals are constrained using virtual clocks. This allows the I/O timings to be adjusted after the clock tree is inserted, to correctly account for the delay of the internal clock tree. All clocks (real and virtual) are defined with uncertainties defined for setup and hold and have a period and uncertainty set to the final target values. Additional sdc files with clock latency definitions are use to overconstraint the flow as required.

Define physical constraints

The synthesis script uses the same variables as the implementation flow to control the number of metal layers used for signal routing and the resistance and capacitance scaling factors applied during delay calculation. This ensures that the synthesis netlist is optimised with the same physical constraints as the implementation flow.

If the *rm_predict_qos* variable is set to 1, the synthesis script will also read a floorplan DEF file. The tool uses the floorplan to further improve the accuracy of the PLEs that are used during timing driven mapping and optimisation. The floorplan is also used during the physical prediction and optimisation stage of the flow.

Cost group definitions and timing driving optimisation

The synthesis script performs timing-driven optimisation of the design before mapping to technology specific gates. During this step, the tool calculates a “target slack” based on an estimate of the timing critical path. The target slack is used to control the netlist structuring optimisations and determines how the design is optimised for power, area and timing.

The synthesis script defines a number of “cost groups”. RTL Compiler will calculate a target slack for each cost group, allowing each to be optimised independantly. By default four cost groups are defined, so that the internal timing paths are not adversely affected by the I/O timing constraints.

If the user sets the *rm_latch_path_group* variable to 1, the synthesis script will define additional cost groups for paths that start or end with a latch. These cost groups will allow the user to see the amount of time-borrowing being used during the initial optimisation and mapping.

Optimization of clock gating latches

By default the synthesis script will automatically insert clock gating latches during synthesis. These cells are only created during the mapping stage, so they are not included in the cost groups defined earlier. An additional cost group is created to target the logic that drives the enable input of the clock gating latches that are inserted during synthesis.

If the user has set the *rm_pre_cts_clock_gating_target_slack* variable, the synthesis script will use a “path_adjust” to tighten the timing on the enable inputs to the clock gating latches.

Note that the final timing reports generated by the synthesis script do not include the effect of the *rm_pre_cts_clock_gating_target_slack* variable. The path_adjust on the clock gating latches is removed before the final timing reports are generated.

The synthesis script runs a pass of incremental optimisation when the enable paths are tightened.

Additional optimisation steps

If the user has set the *rm_synth_extra_opt* variable to 1, the synthesis script will run additional passes of incremental optimisation during the main synthesis and also during physical prediction (if the *rm_predict_qos* variable is set to 1).

These additional optimisation steps are intended to allow the tool to further reduce the area of the design and improve the quality of the final design.

Some users may want to turn off this feature during their initial development to reduce the tool runtime.

Physical prediction and optimisation

If the *rm_predict_qos* variable is set to 1, the synthesis script will run the physical prediction and optimisation flow. These commands are available in the 7.2 release of RTL Compiler.

During the physical prediction flow, RTL Compiler will use First Encounter to generate a Silicon Virtual Prototype (SVP). The detailed placement information, wire resistance and capacitance from the SVP are used for delay calculation and annotation of physical delays.

If the *rm_synth_extra_opt* variable is set to 1, the synthesis script will perform incremental optimisation using the placed database. Logic that is identified as being sensitive to placement will be preserved to maintain the accuracy of the SVP.

Reports generated

The synthesis script generates a number of reports, which are placed in the **reports/synthesis** directory. In addition to checking the final timing, power and area reports, the user should also review the clock gating report, scan chain and DFT reports.

The synthesis script generates a timing "lint" report and a summary of all messages issued during the synthesis run. These should be used to check that the all inputs and outputs have been constrained. The user should review the messages summary to check for unexpected warnings or errors.

Note that when RTL Compiler is run using PLEs, the area report is based on the information read from the physical libraries, not the timing models.

Files generated.

The synthesis script generates a number of outputs to be used by other tools. These are all written to the **data** directory.

The **cortexm0_rtl2gates.opt.v** and **cortexm0_scan_chains.def** are used by First Encounter in the implementation stage of the flow. The DEF file defines the scan chains that were inserted during synthesis, removing the need for First Encounter to trace the chains before running scan reorder.

The **cortexm0_rtl2gates.mapped.v** file is written part way through the synthesis script. This is generated to give the user the ability to begin floorplan development while the remaining synthesis steps are being executed.

The **cortexm0_assign** file defines the test related I/O used on the core. This is used by Encounter Test during the ATPG flow.

Formal verification is run to compare the RTL and the final layout Verilog netlist, using the **cortexm0_rtl_vs_netlist.do** file written by the synthesis script. By default the synthesis script will include the commands to run the verification in hierarchical mode. If the user wants to run a flat formal verification the "-flat" option should added to the synthesis script.

Implementation with SoC Encounter

Overview

Cadence SoC Encounter is used to Place, Optimize and Route the design. The implementation portion of the flow is split into various individual encounter runs. If the implementation step is rerun any of the individual portions that has completed and written out a design will be skipped.

This section of the flow takes the placed design and performs the functions in the figure below to reach a fully routed and timing analysed design.

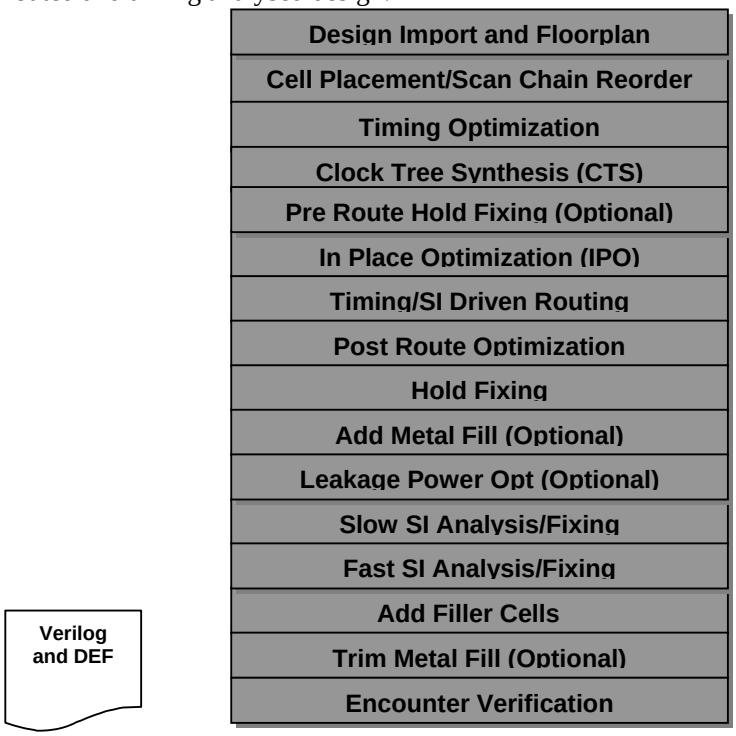


Figure: Place and Route Flow Diagram

Each of these steps is elaborated in the following sections.

Low Power and MMMC

In this release the flow scripts have the option of running in low power mode, where a common power format power intent file is read in and the additional low power functionality implemented. In normal iRM's these are provided for information only abd the alternative MMMC commands are used to do normal implementation. In low power iRM's the same flow scripts will be used.

Design Import/Floorplan

SOC Encounter is the primary interface tool for this section of the ARM methodology flow. This section of the flow uses the script **"cortexm0*_floorplan.tcl*".**

In order to load the design into Encounter, a configuration file is used. The file is called **"cortexm0*_fe.conf*"** and can be found in the scripts directory. It references the following design and technology specific data which is setup by the **"cortexm0*_config.tcl*"** script:-

- Design netlist/timing constraints
- Technology, standard cell and macro LEF
- Standard cell and macro .lib timing models (max and min)

- SI library files (.cdb's)

The file **"cortexm0_*fe.conf*"** points to the netlist output from the synthesis stage of the flow which will be in the data directory.

Once the design is imported the script goes on to load the floorplan. By default the script will load the floorplan file ***"cortexm0.fp"*** from the floorplan directory**.** If this does not exist it will then look for a floorplan def ***"cortexm0.def"*** in the floorplan directory**.** If that does not exist then the flow runs plan design to generate a quick floorplan.

Now the power and ground nets are created and instance power and ground pins are connected to those nets. The names of the power and ground nets and which pins are added to them is controlled by the **"cortexm0_*config.tcl*"** script. Once the nets are created the script builds the power and ground grids. The settings for build the grids are process and library specific and should be revisited if retargeting the flow.

The final physical floorplan objects to be inserted at this stage are well tie cells. These tie the wells to the correct power and ground nets. They are placed in a regular pattern across the entire standard cell area.

Information on the scan chains, inserted by RTL Compiler, for use in scan re-ordering is read from a scan def file.

Information on how to construct the clock tree is also imported or created at this point in the flow. If there is a cts technology file called **"cortexm0_*cts.ctstch*"** in the scripts directory this file will be copied to the data directory and also read in. If the file is not there a cts technology file will be created by the tool automatically. The auto created cts tech represents a 'best guess' of the criteria for building the clock trees, it is recommended to tune this file based on actual requirements if a cts tech has not been provided.

Finally IO buffers defined in the config file are added to every Input and Output. The accuracy of any timing model generated for the core is affected by the nets which are routed into it at the top level. Anything which has not been characterised and tested by extraction and STA within the block can potentially affect the block when it is connected into an unknown top level. Therefore, it is important to minimise the impact the top level can have on the block. This is achieved by adding an isolation buffer onto every boundary net, as close as possible to the pin to which it is connected. By minimising the net length inside the block, the slew and load of the net outside the block has less of an impact on the timing internal to the block. As a result the timing model of the core (TLF or .lib) will be more accurate across a range of slews and loads. A variable is set in the file config.tcl to specify which library cell (e.g. BUFX8) is used as the isolation buffer; ***rm_boundary_net_buffer*** is specified in the **cortexm0_*config.tcl*** file. Also, it may be required to exclude certain nets from this step. In this instance the input clock net is excluded as this will be handled separately during Clock Tree Synthesis. The clock net is, thus, listed in the file **"cortexm0_*excl.file"*** which is in the ***scripts*** directory. Finally, the newly added buffers are fixed so that subsequent optimizations do not affect them.

Placement & Scan Chain Re-Ordering

Placement of standard cells is performed by the script **scripts/cortexm0_placement.tcl**.

The first step in the placement script is to restore the design saved at the end of the floorplan step. Once this is in place the script creates path groups. Up to thirteen path groups are created. Path groups are used for two reasons:

1) Many ARM cores have critical clock gating enable paths, driven by both internal and external logic. By default only register-to-register paths are optimized with the highest priority and in some versions of the tools the clock gates do not get added to the reg-to-reg group.

2) Many ARM cores also have negative edge triggered memory elements (flops, rams or latches). These create 'half cycle' paths. If the design is being optimised targeting a higher frequency that is being attained (which is standard practice) these half cycles need to be treated as a special case. A negative slack on a half cycle path is equivalent to twice the negative slack of a normal path. By creating separate path groups for these paths they do not get 'masked' by the full cycle path during optimization.

This gives thirteen possible path groups, which may not all apply to every core:

In2regr – input to rising edge triggered register

In2regf – input to falling edge triggered register

In2clkg – input to clock gate

In2out – input to output path

Regr2regr – rising edge triggered register to rising edge triggered register.

Regr2regf – rising edge triggered register to falling edge triggered register

Regf2reg2 – falling edge triggered register to rising edge triggered register

Regf2regf - falling edge triggered register to falling edge triggered register

Regr2clkg - rising edge triggered register to clock gate

Regf2clkg - falling edge triggered register to clock gate

Regr2out - rising edge triggered register to output path

Regf2out - falling edge triggered register to output path

Default – path group for any other paths

Before any placement is carried out timing analysis is performed with zero wire load model. This can highlight any problems with the synthesized netlist:

Next various options are set. See the cadence tool documentation for details on what these options do, but they can be summarised as setup a max routing layer so the tools know not to use all layers defined in the technology file. Run placement in a slightly more runtime intensive manner that improves final results and switch on useful skew in its more aggressive mode.

The design is now placed using the place design command. This will perform placement and scan chain re-ordering. This command is run with the option –inPlaceOpt which means after the first placement is done a short optimization is run and an incremental placement is then performed. This has been shown to improve overall results.

After initial placement and prior to Clock Tree Insertion, pre-CTS timing optimization is performed. The tool does global optimization before concentrating on the critical paths. The optimization step will work to improve the timing of all 'high effort' path groups stand alone. 'low effort' path groups are only worked on as part of the overall path list.

If the option 'rm_extra_opt' is set a second optDesign is run. Some cores also benefit from further optDesigns at this stage.

All outputs from the **cortexm0*_placement.tcl*** script are written to the ***./data*** directory.

Clock Tree Synthesis (CTS)

Clock Tree Synthesis is run from within Encounter in the CTS directory. If the placement step has finished successfully the CTS script (**cortexm0*_cts.tcl***) will be sourced with make full usage.

During CTS, ***Nanoroute*** will be called to route the clock nets. Therefore, the required options for ***Nanoroute*** are set at this stage.

As stated during the Placement section, the tool can automatically generate clock tree constraints based on the timing constraints. Alternatively, the user may wish to edit the clock tree constraints and create their own specification file instead in order to have more control over how the clock tree is generated. Some options in the CTS specification file which are worth noting are shown here, these are some of the only technology specific data not controlled by the config file and must be edited to re-target the flow:

```
Buffer          CLKBUFX2AD CLKBUFX4AD CLKBUFX8AD CLKBUFX12AD CLKINVX1AD CLKINVX2AD
                CLKINVX4AD CLKINVX8AD CLKINVX12AD
AddDriverCell   CLKBUFX8AD

RouteClkNet     YES
```

The full list of Buffers and Inverters allowed during CTS can be defined. In order to ensure a specific cell is placed at the beginning of the clock tree, the AddDriverCell option can be used. Finally, to call Nanoroute to route the clock nets, the RouteClkNet option must be set.

Tuning cts is generally done by getting this list correct, giving the tool just enough clock insertion delay to achieve its skew target and being quite aggressive on the slew target. If clock tree power is a concern consider increasing the skew target and keep an eye of pre-route timing to see how much de-tuning of the tree can be done before timing is effected.

An OCV Clock Tree Report file is generated. The early and late derating factors are set, this allows the tool to scale Setup and Hold paths correctly. The report file shows the skew of the clock tree with and without OCV derating. This allows the user to see how much worse the results will get when using OCV.

On Chip Variation Support (OCV)

On Chip Variation is supported throughout the flow (from Clock Tree Insertion onwards). OCV is a way of modelling different operating conditions that can be present simultaneously on a die. These variations can be due to voltage differences, temperature differences or process differences.

The OCV support in this flow uses 1 library with 2 derating factors specified by the user. E.g. Early Derate “0.95”, Late Derate “1.05”. If the user does NOT wish to use OCV then these factors should be set to “1”. The scaling factors affect the delay values generated in the timing reports. Setup paths are slowed down and Hold paths are speeded up, therefore increasing timing pessimism. This additional pessimism gives extra margin to compensate for OCV effects on the timing.

OCV is used in conjunction with Clock Re-convergence Pessimism Removal (CRPR); this allows the tool to remove any additional pessimism when timing common clock paths. This is required for hold fixing when using OCV. When timing reports are generated by SOC Encounter these will include a pessimism figure and also the early or late derating factor.

In Place Optimization (IPO)

When Clock Tree Synthesis is finished the **cortexm0*_update_clock_insertion.tcl*** script is sourced and run. This creates new clock latency sdc files in the ../data directory. These constraints now have the clock insertion delays included, which will be used for all Post CTS stages of the flow. It also includes any additional constraints, which may be optionally specified in the ***cortexm0_.additional.constraints.sdc*** file, to be used for post CTS stagesIf the option ‘rm_balance_clocks_externally’ is set the command will compute clock latencies for virtual clocks and source latencies for clocks such that the design is constrained on the assumption that all clocks will be balanced at the top level. This allows the top level integrator more flexibility. If this is not required and clocks need to be balanced internally the cts tech file needs the clocks ‘grouped’.

If the option ‘rm_skew_io’ is set the flow now does a quick optimisation and then calls **cortexm0*_update_clock_insertion.tcl*** again, but this time the script will look at the overall worst input and output paths (it does not account for clock domains, which is a limitation) and will setup the clock latencies to model the top level integrator doing useful skew to minimise the overall

timing violations to and from the core rather than perfectly balancing the core with the top level clock tree.

Finally is pre-route margin is defined the script will set up the latency sdc's to include the margin.

Up until this point, all timing analysis and fixing has been based on an ideal clock (zero insertion, zero skew). Now that we have inserted a clock tree we perform additional timing optimizations based on a propagated clock. The constraints generated at the end of the CTS stage are now loaded into memory.

The **cortexm0_*cts.tcl* script** also performs optDesign within Encounter. OptDesign uses the very fast transforms available to Encounter, which quickly get the timing to a reasonable level. The optimizations include upsizing/downsizing of cells and logic restructuring. If the option 'rm_preroute_holdfix' the flow will run a hold fix prior to the main optDesign

If the option 'rm_extra_opt' is set a second optDesign is run. Some cores also benefit from further optDesigns at this stage

Timing/SI Driven Routing

After optimization, **Nanoroute** carries out routing on the remaining (non-clock) nets. The script that performs the signal routing is called **cortexm0_*route.tcl***.

If they are available in the technology library Tie high and Tie low cells are inserted before routing. If these cells are not inserted tie high/low signals will be automatically hooked-up by **Nanoroute** and connected to the power/ground structure.

Next the multi-cut via effort is set to the setting defined for route. This may well be lower than that defined for clocks earlier in the flow. Ensure this setting correlates with you signoff criteria for multi-cut vias. The design is now routed.

The clock nets are unfixed at this stage. The previous route may have had conflicts between data routes trying to access instance pins and clock routes that must be resolved. To identify the clock nets, the script uses the CTS specification file is in. There are a number of **Nanoroute** options that are setup during the CTS step and they are saved in every subsequent database.

The design is now re-routed with post-route wire spreading turned on. This route will fix the clock net violations and will spread the wires to improve yield.

Now the design is routed, timing analysis is performed. This uses detailed RC-extraction and is more accurate than the pre-route RC estimation. A further optimization is called to resolve any new issues caused by routing. This is followed by hold-fixing and then a final optimization.

At this stage, a custom wire load model is generated for use by synthesis tools that use wireloads. By default RTL Compiler uses PLEs as they give better results than wireload models.

Metal Fill

Metal fill might optionally be added to the design. Due to the planar process techniques employed to fabricate silicon devices, there are a number of polishing steps as each layer is put down. If there are areas on the silicon that are low in metal density then damage can be caused to the design so metal fill is added to these areas to help avoid these situations. The metal is tied off to the power rails. Adding metal fill prior to extraction will give a much more accurate description of degradation of timing than the traditional chip finishing approach.

This is an optional stage: in order to run it, the rm_*metalfill_insertion* variable should be set to "1" in the **cortexm0_*config.tcl*** file.

Since the interpretation of fill rules may vary slightly according to the design rule check (DRC) verification decks, the user should ensure that the default values used are correct for the process being used. *setMetalFill* command may be used to change the parameters that the *addMetalFill*, *trimMetalFill*, and *verifyMetalDensity* commands use for metal fill size, spacing, verification, and

metal density. These parameters could also need some tuning in the case some connectivity violations are generated because of missing connections of metal fills to the corresponding special nets.

The metal fill has to be added prior to final extraction so that the capacitive coupling of signals to fill metal can be taken into account during extraction:-

Multi Voltage Threshold Optimization

At 0.13 micron and below leakage power dramatically increases. Leakage power increases as the threshold voltage (Vth) of the transistor decreases, as a result a trade-off between delay and power is required at smaller process geometries.

Fully balancing leakage power and timing requires the use of 2 libraries with cells that operate at different threshold voltages. One library has high Vth cells and the other Low Vth cells. The high Vth cells have a lower leakage power but are slower. The tool replaces fast cells with slower cells if the paths have positive timing slack, hence reducing the leakage power. To run this step correctly the user must define the 2 libraries in the **cortexm0*_config.tcl*** file, however this command may well still reduce leakage power even if only one library is present, by swapping to lower leakage cells in the same library.

This is an optional stage and in order to run this stage the *rm_mvt_run_opt* variable should be set to "1" in the **cortexm0*_config.tcl*** file.

The leakage power is reported before optimization, optimization is carried out by swapping cells from the other library and then the leakage power is reported again. Finally, the DEF, Netlist and design are saved.

SI Closure

As the best timing has now been achieved for the given floorplan and technology, it is time to perform an SI analysis of the design and fix any nets that are deemed to have the potential to cause both glitch and delay violations due to proximity to other nets in conjunction with fast or slow switching signals. The analysis is performed by ***CeltIC*** and the fixing by Encounter. The script that performs these functions is entitled **cortexm0*_si_fixing.tcl*** located in the **scripts** directory. The analysis and fixing loop is invoked by a single command as shown in the next few sections.

To perform SI analysis and fixing some setup steps are required. The technology process must be specified and this can be done using the ***setSIMode*** option. In this case the process is a variable, which is defined in the setup**cortexm0*_config.tcl***. It is also possible to run the SI analysis based on QRC extraction, but the recommendation is to use the detailed RC-extraction within Encounter. For the analysis, Celtic is called and automatically references the worst case ***cdb*** library and ***UDN/ECHO*** models defined in the environment.

Encounter performs the following as part of the ***optDesign –si*** flow:-

- Timing Windows generated by Encounter timing engine
- ***CeltIC*** launched reading in extracted parasitic data and timing windows, generating an incremental SDF
- Encounter reads incremental SDF and determines which paths have worsened, marking nets that are deemed to have changed significantly as requiring fixing
- *fixNoiseDelay* command is run
- *fixGlitchViolation* command is run
- ECO routing is performed after each pass of the flow in order to route the ripped up nets in SI aware mode, and route the new nets for buffers/inverters that were added.

This flow is then iterated to ensure that no more SI problems have been introduced. If they have, then another pass is performed and this loop is repeated until the design is deemed to have no more outstanding SI problems at this particular corner.

Encounter now performs SI analysis in best case. By default Celtic references the fast library and models, which have been defined in the initial configuration file. For the hold SI analysis only glitches will be flagged and fixed.

Adding Filler Cells and Trimming Metal Fill

After routing the design, filler cells must be added to the gaps in the standard cells in order to maintain the connectivity of the diffusion layers in the standard cells. The type of cells will be determined by the technology used. These cells are listed (with the largest first) in the variable rm_fillercelllist in the config file **cortexm0*_config.tcl***. Subsequent optimization steps will remove and add fillers as needed based on the ***setFillerMode*** options.

After all the steps that may require ECO routing, it is necessary to legalize metal fill if it was previously inserted. This is done as using the trimMetalFIll command.

Encounter Verify Design Commands

The following Encounter commands are now run. The physical checks are run at cell level within Encounter, for gate level checks DRC must be run using the GDS2 generated from the flow. Report files are generated for each of these commands. These give the user confidence that at this stage of the flow there are no major problems with the floor planning or setup.

verifyProcessAntenna – Checks for process antenna violations and outputs an Antenna LEF

verifyConnectivity – checks for shorts, opens, unconnected pins, connectivity loops

verifyGeomerty – Checks metal spacing rules, metal widths, minimum area, shorts, cell overlaps, checks to ensure cells are on manufacturing grid.

verifyMetalDensity - Checks the metal density for each routing layer and the density of large macros. To run this command, density values have to be specified either by the LEF file or by *setMetalFill* command.

As well as saving the verilog/def at this stage, a ***LEF*** file for the core is generated. Also an IO file is created. These will be required during the Model generation/validation.

Signoff Parasitic Extraction

Signoff parasitic extraction is done using QRC from the EXT install path. It will use the three process corner extraction QRC Techfiles in the tsmc directory structure detailed in section 1.1. QRC reads the corner.defs file to locate the cworst, cbest and typical qrcTechfiles in the corner sub-directories and will generate the spef parasitics for all three corners in the same run. While this may increase the runtime for one corner, it saves time when running all three corners simultaneously. This step can be multi-threaded onto a server farm if required. Please refer to the QRC User guide for more information.

The command to run this step is:

make extraction

which will cd to the work directory and start QRC in 64bit mode reading the layout file named cortexm0_si_fixing.def and will use the cortexm0_qrc.tcl command file:

```
qrc -64 -cmd $(scripts)/cortexm0_qrc.tcl ../data/cortexm0_si_fixing.def
```

The resulting output SPEF files will be written to the data directory and are named:

cortexm0_cworst.spef.gz

cortexm0_cbest.spef.gz

cortexm0_typical.spef.gz

Signoff STA

The Signoff STA is done using the **Encounter Timing System**, (ETS), which is invoked after the parasitic extraction step. ETS will perform setup and hold timing analysis on the final netlist from the SI_FIXING stage for the required process corner and report the static timing results and will then run SI analysis and finally report the static timing including the SI effects.

The three makefile steps that are used to run Signoff STA are:

```
signoffwc:
        cd work; \
        ets -init $(scripts)/cortexm0_signoff_wc.tcl -nowin \
        -log $(log)/cortexm0.signoff_wc.log

signoffbc:
        cd work; \
        ets -init $(scripts)/cortexm0_signoff_bc.tcl -nowin \
        -log $(log)/cortexm0.signoff_bc.log
signofftyp:
        cd work; \
        ets -init $(scripts)/cortexm0_signoff_typ.tcl -nowin \
        -log $(log)/cortexm0.signoff_typ.log
```

Each signoff STA step runs in ETS for a specific process corner.

Overview

Encounter Timing System (ETS) is a stand-alone timing and SI signoff tool. It is based on the same timing engine (CTE) as SOC Encounter but uses SignalStorm as the delay calculator. SignalStorm has the capability to read libraries which include ECSM models in order to perform more accurate delay calculation. ETS includes Celtic for Signal Integrity analysis.

Static Timing Analysis.

The steps for running ETS are outlined here.

First the relevant design information must be read in. This includes library models for timing and SI, the final verilog netlist, design constraints and spef parasitics, (from signoff extraction). The read_spef command writes out the missing nets from the spef to a report file in the reports/signoff directory named signoff_<corner>_spef_missing_nets.rpt. This report file should be checked to make sure that none of the routed nets are missing from the spef parasitic files as this will impact the accuracy of the timing reports.

Various checks are performed on the data which provides information about the design. This includes the following checks:

- Timing information check
- Top level netlist check
- Instance Pin check
- Multiple driven netlist check
- Sub Module Port definition check
- Don’t use cells in design
- I/O Pin check
- Cell Capacitance load check

- Cell transition violation check
- Cell fanout design rule violation check

The check_design command creates a sub directory named signoff_<corner>_check_design

that contains html format files that show the results. The file named cortexm0.main.htm should be opened in an Internet browser. This has a summary table where hypertext links to the other files appear highlighted in blue text.

The following report files are generated by the other design checks and are written to the reports/signoff directory.

```
signoff_<corner>_check_library.rpt
signoff_<corner>_check_timing.rpt
signoff_<corner>_report_ports.rpt
signoff_<corner>_spef_annotation.rpt
signoff_<corner>_analysis_coverage.rpt
signoff_<corner>_constraint.rpt
signoff_<corner>_check_cap_violations_drv.rpt
signoff_<corner>_check_tran_violations_drv.rpt
signoff_<corner>_check_fanout_violations_drv.rpt
```

Then the OCV settings and propagated analysis mode are set up and the required operating conditions from the timing libraries are defined.

The default OCV settings used are 10% for cworst and cbest corners. The typical corner does not have any derating applied. If different values of derating for OCV are required then the two values of the rm variables $rm_derate_max_early and $rm_derate_max_late can be changed in the cortexm0_config.tcl file.

Next the timing report format is defined. This defines the order of the columns in the reports. If another reporting format is required the order of the keywords between the { } characters on the following line should be changed.

```
set_global report_timing_format \
          {instance cell arc delay slew load arrival fanout instance_location net}
```

There is another report_timing_format command later in the ETS scripts that adds the delay pushout due to SI effects as a new column into the reports.

The instance location will be reported as 0,0 due to the scripts not reading in the physical data. However this will be fixed in a future release of the RM.

Next the four major path groups are defined as reg2reg, in2reg, reg2out and in2out.

Then the TWF and SDF files are generated. The twf file is used in the power analysis step to define the timing windows for the signals changing state. The SDF file can be used for example during the testbench simulation using dhrystone loop power vectors in a simulation tool like NCVerilog, when the real design timing is required to make the delays more accurate.

The timing reports are generated next and written to the ../reports/signoff directory. The slack paths are reported first into a file named signoff_<corner>_ocv_setup.slk and then the first 1000 setup violations are reported to four files for each of the path groups.

```
signoff_<corner>_ocv_reg2reg_setup.rpt
signoff_<corner>_ocv_in2reg_setup.rpt
signoff_<corner>_ocv_reg2out_setup.rpt
signoff_<corner>_ocv_in2out_setup.rpt
```

There are two additional report files that are written out to highlight the half cycle paths from the leading edge of the clock to the trailing edge of the clock and from the trailing edge to the leading edge. These paths were found to be critical during implementation and so have been put into two files on their own to make it easier to check for them. These reports are named

```
signoff_<corner>_ocv_trail_to_lead_setup.rpt
signoff_<corner>_ocv_lead_to_trail_setup.rpt
```

Then the hold slacks are written to a file named signoff_<corner>_ocv_hold.slk. Then the first 1000 hold violations for each of the four path groups are written out to files named

```
signoff_<corner>_ocv_reg2reg_hold.rpt
signoff_<corner>_ocv_in2reg_hold.rpt
signoff_<corner>_ocv_reg2out_hold.rpt
signoff_<corner>_ocv_in2out_hold.rpt
```

Finally a critical instance report is written out to a file named

```
signoff_<corner>_critical_instance.rpt
```

This file contains the top 20 critical instances that cause the worse timing paths during a bottleneck analysis of the design. The paths are sorted by cost value which is the Total Negative Slack, (TNS), timing violation value associated with the instance. This may help to determine which instances have loading or fanout issues that need to be investigated first and would fix the most timing issues.

SI Analysis

ETS runs SI Analysis using the noise models files read in at the start of the script. The set_noise_run_mode command defines the process to be 90nm and the mode is defined as signoff for best accuracy.

Then the set_noise_thresh command is used with the –delta_absolute parameter set to be the rm_delay_threshold value converted to nanoseconds.

```
set rm_delay_value [expr 1.0e-9 * $rm_delay_threshold]
set_noise_thresh -delta_absolute $rm_delay_value
```

This parameter defines the delay threshold for ETS when it analyzes the SI delay pushout values in the design. If this value is smaller then more nets will be analyzed. If it is bigger, then fewer nets will be analyzed for delay pushout during SI analysis. Consequently there is a runtime trade off in ETS if this value is smaller. It is therefore left to the user to decide what the optimum value should be for the technology being used for the design.

When the SI analysis has completed, the noise results are reported to the ../reports/signoff/< corner>_noise directory. This directory will contain various html format files. Using a web browser to open the index.htm file which will load a summary with hypertext links to the other files that hold details of the violations. Alternatively, the file named noise_results.txt has the violations listed in a text format.

Noise induced doubled clocking issues are checked in the design and the results are written to a report file called signoff_<corner>_noise_double_clocking_check.rpt

Then the SOC Encounter format eco commands required to fix the SI violations are written out. If this file contains violations it should be read back to SOC Encounter with the si_fixing design database loaded and run to fix the SI violations on the various nets.

Next the timing reports for setup and hold are generated which will include the SI effects with the filename format: signoff_<corner>_inc_si_setup.rpt and signoff_<corner>_inc_si_hold.rpt. The half cycle path reports are now generated which will include the SI effects.

Then ETS will then exit the step.

If timing violations are found in the timing reports at any stage, then the design can be loaded into ETS and the timing interactively debugged using the Timing Debug window in the tool. Please refer to the ETS User Guide for information on the features available for interactive timing debug. If the physical location of the violating cells and the routing is required to be inspected interactively in the GUI then a similar interactive timing debug window exists in SOC Encounter.

Static Power Analysis

When the Signoff STA flow steps have completed the Static Power Analysis step can be run.

```
#-----------------------------------------------------------------------
# Run the VSTORM stage of the flow
#-----------------------------------------------------------------------
vstorm:
        cd work; \
        encounter -64 -nowin -init ../scripts/cortexm0_vstorm_analysis.tcl \
        -log  $(log)/vstorm_analysis.log; \
        vstorm2 -64 -cmd ../scripts/cortexm0_vs_static.cmd; \
        cp powermeter.summary.log ../logs/cortexm0_powermeter_static.summary.log; \
        cp vstorm2.log ../logs/cortexm0_vstorm2_static.log
```

Overview

VoltageStorm is used to perform static power analysis of the core. All VoltageStorm work is performed in the *work* directory.

The analysis and power grid model generation is invoked by the ***make vstorm*** command. The VoltageStorm command files and voltage source location files are generated from the Encounter interface and then they are used by VoltageStorm for the static power analysis. Finally the logfiles from Powermeter and VoltageStorm are copied to the logs directory. The initial script that is executed is named cortexm0_vstorm_analysis.tcl.

There are a number of pre-requisites that must be met prior to running the flow:-

- Complete power grid inserted in design
    - Including FOLLOWPINS connections
    - All memories hooked into power grid
- Design must be PLACED and ideally optimised for most accurate results
    - Each PLACED cell appears as a tap current to VoltageStorm on the power grid
- Design must be routed and extracted
    - In order to perform accurate power analysis, exact net loading data is required
- Signoff STA has been run in the typical process corner to generate the typical case Timing Window File, (TWF).

The data used is the final database, which is fully routed and extracted with QRC. This can be changed at the user's discretion as long as all data (verilog, DEF and SPEF) matches.

Figure below shows the flow for VoltageStorm analysis and model generation.

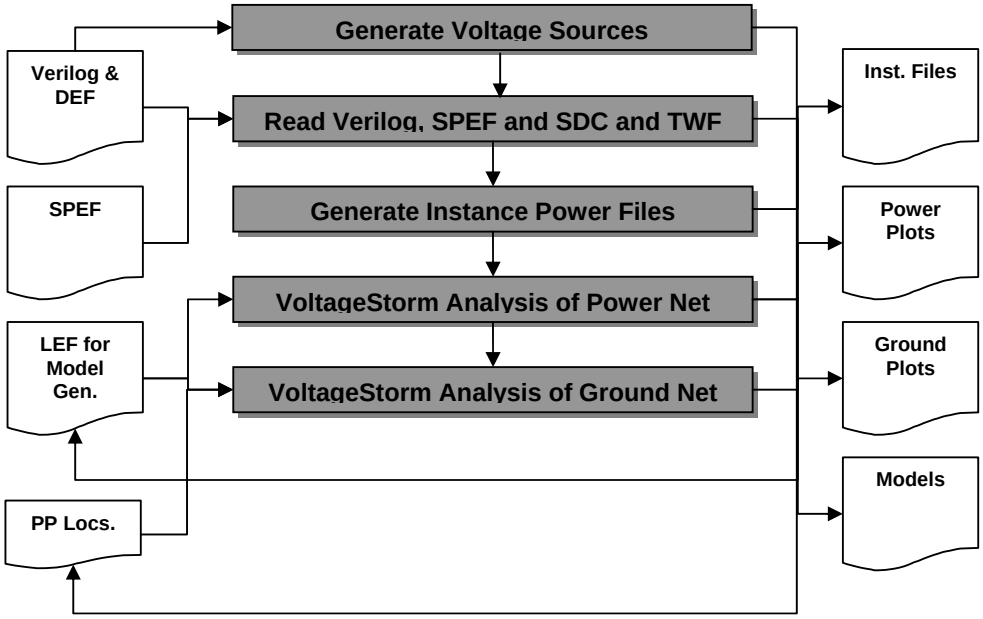


Figure: VoltageStorm Static Power Analysis Flow Diagram

Generate Voltage Sources

VoltageStorm requires a list of voltage sources in order to perform the analysis. Ideally, the def would have power pins defined (PIN with +USE POWER/GROUND) as well as the power grid. If this is the case, an Encounter command may be used to automatically generate a file with the coordinates of the voltage sources:

```
autoFetchDCSources VDD
savepadLocation –file ../data/cortexm0.VDD.pp
autoFetchDCSources VSS
savepadLocation –file ../data/cortexm0.VSS.pp
```

In this RM, the power supply pins are defined as being at the end of all the power and ground stripes all around the core. If this is not the case, the following files for VDD and VSS have to be manually created and saved at the following location: ***../data*** directory; files: ***cortexm0.VDD.pp***, ***cortexm0.VSS.pp***.

A perl script named cortexm0_gen_vsrc_from_encounter.pl is then used to convert these files into VoltageStorm format which writes out cortexm0_VDD.vsrc and cortexm0_VSS.vsrc files to the data directory.

Generate LEF with Power Pins

A pre-requisite for correct model generation is that a LEF exists for the core that has the power grid abstracted in it as pins such that VoltageStorm can attach the voltage sources. Encounter can generate LEFs with this information in it, but will only abstract sections of the power grid those are on the upper 2 layers of the process. In the case where the power pins are not on the top-layers, an option can be added to the ***lefOut*** command to identify which layers the power pins are on ***(-PgpinLayers*** MH MV). Before the ***lef*** is created, all metal fill is removed. This is necessary as power analysis currently does not support metal fill connections to internal power pins (excluding power rails) of standard cells. The SPEF which is used already has considered the metal fill so the accuracy is still assured.

```
#-------------------------------------------------------------------------
# Generate a LEF for PGV model generation
#-------------------------------------------------------------------------
editDelete -shapes FILLWIRE

lefOut ../data/cortexm0.pg.lef -5.5 -stripePin -PGpinLayers $rm_layer_v $rm_layer_h
```

Instance Power Files

VoltageStorm static power analysis runs Powermeter to generate the instance power files. Powermeter performs an analysis of each instance of the design based on a default input switching activity of 20%, (0.2), a default input duty cycle of 50%, (0.5) and the frequencies of the clocks used in the core. The default input switching activity value of 0.2 equates to the signal inputs at the core toplevel that do not have clocks or fixed values defined on them in the design constraints file are toggled once every fifth clock cycle with a 50% duty cycle. The clock information is taken from the design constraints and the Timing window files. The static power analysis step reads a design constraints file, created by the implementation step, that keeps the Scan Enable and Reset pins inactive to prevent unrealistic power results due to either the scan chains being active or the reset signal toggling across the whole design.

Each instance is then assigned a power consumption value and based on the switching and loading of its pins and the resultant power calculation from the tables in the input Liberty files. These values are output to results files used by VoltageStorm, which then applies them to the extracted physical layout grid in the analysis stage to calculate the currents and voltages in the layout. Powermeter creates a text file called powermeter.pwr, stored in the vstorm_tc/pmtmp directory of the work area. This file lists the internal power, switching power, total power and leakage power for all the design instances. At the end of the file are the totals for each type of power for the whole design.

VoltageStorm Analysis of Power Net

All relevant data to perform the analysis and model generation is now available and is outlined below:-

- Voltage source location file
- Instance power file
- LEF containing power nets to attach sources to
- EM model file – this is a requirement for VoltageStorm. This file (***scripts/vstorm_models.em***) describes each metal layer's maximum current density and thickness**. The supplied file is a template only. The dummy values are for example only. The real values for your target technology should be calculated from the foundry process Design Rule Manual and the model command lines uncommented. See the VoltageStorm User Guide for more information on how to calculate the actual values for the process.**

  There are several different types of analysis that can be performed by VoltageStorm. The analyses performed by this flow are:-

- IR Drop (IR)
- Instance Voltage (IV)
- Tap Current (TC)
- Resistor Current (RC)
- Current Density (RJ)
- Via Current (VC)

- Via Voltage (VV)

Static power analysis and power grid model generation are performed by VoltageStorm using this command:-

```
vstorm2 -64 -cmd ../scripts/cortexm0_ vs_static.cmd
```

The results for the run will be stored in the data directory in a sub-directory named VDD_static_tc_1 for a power net called VDD. In this directory there are a number of results file. For a quick analysis of the results of each analysis run refer to the GIF files e.g. ***ir_linear.gif*** shows the IR drop graphically. A text file that lists the results of the analysis is available in a file called ***results*.** This file lists the worst and average ir drop values of the supply rail grid.

Finally, a VoltageStorm library is generated containing the abstract power view of the core in this location: ../data/cortexm0_Power_VDD.cl

The libraries are used in the same way that any hierarchical VoltageStorm library is used. i.e. it is pointed to as a cell library for the design and picked up by *VoltageStorm* when performing any static power analysis measurements.

VoltageStorm Analysis of Ground Net

Analysis of the ground net is performed in exactly the same way as analysis of the power net. The results can be found in a similar named directory as outlined previously, e.g. VSS_static_tc_1. The VoltageStorm library containing the abstract power view is stored here

../data/cortexm0_Power_VSS.cl

If the vstorm step is run again, the results directory names will be named VDD_static_tc_2 and VSS_static_tc_2. However the power grid model generation commands are looking for VDD_static_tc_1 and VSS_static_tc_1 and so will fail on the second and subsequent runs. The solution is to rename the VDD_static_tc_1 and VSS_static_tc_1 directories to new names if you need to save the data or delete them before re-running the vstorm step.

Model Generation And Validation

Overview

In order to integrate the hardened core into a SoC design, abstract models of the core must be generated. The table below outlines the type of model generated, the relevant tool used and a brief description of the model usage.

Model Type	Tool	Flow Section	Description of Usage

Functional	***ampkg/ncsim***	Functional Model Generation	Compiled binary with annotated timing used for functional verification.
Timing	***Encounter***	Model Generation	.lib models describing the core's Interface timing for place and route tools.
Physical Abstract	***Encounter***	SI Closure	LEF model describing physical pin placement and macro size along with antenna information.
Signal Integrity	***ETS***	Signoff timing	Noise model describing noise susceptibility of the hardened core.
Static Power	***VoltageStorm***	Verification / Power Analysis	VoltageStorm power grid model describing current taps and IR drop for the hardened core.

Table: Models Overview

Model Generation Script

When the Signoff STA steps have been successfully completed, models of the hardened core can be generated. The step is run using the command:

make generate

Cadence Encounter is used to generate the timing models and the ***Ampkg*** Model Packager tool is used to generate the functional model.

Functional Model

The IP Model Packager tool, supplied as part of the Cadence LDV (Logic Design and Verification) deliverable, is used in this flow to generate a compiled functional model of the hardened core. This model can then be used in functional simulations of the system hosting the ARM, while preventing visibility of the netlist itself, and hence maintaining the security of the core architecture.

Usage of the *ampkg* model generation and packaging tool is straightforward, and the provided ***run_ampkg*** script in the ***scripts*** directory can be used to perform the library compilation, netlist compilation and model generation. In addition to the model itself, the resultant tar file also includes the relevant user documentation, installation utilities, and the Model Manager tool that integrates the model into the simulator's environment

Packaging Control File

The control file for the *ampkg* tool, an example of which is below, is a command file that determines the model name, model revision number, and the naming for the packaged directory tree. The control files required to run ampkg are generated automatically.

```
SUPPLIER   "Cadence Design Systems"
DIRECTORY   cadence/cortexm0
ACTIVATION  inputs
MODEL      "cortexm0_model"
OUTPUT      cortexm0
ELABORATE  "-nowarn CSINFI -nowarn CUVWSP -notimingchecks"
REVISION   "1.0"
```

Functional Model Generation

The name of the model is extracted from the netlist name, which is defined as the final netlist created by the implementation script. The name is deduced by the following:-

```
# extract module_name and model_name from netlist_name
set netlist_name = "../data/cortexm0_si_fixing.v"
set module_name = “cortexm0”
set model_name  = `echo $module_name | sed 's/_[A-Z0-9]*$//'`
```

Before model generation, the netlist and any associated technology libraries must be compiled by the *ncvlog* compiler.

NOTE: The standard cell library should include a dummy functional model of an ANTENNA cell as these are added by ***NanoRoute*** in the flow and are not always included in the functional verilog library. The model should be of the form:-

```
`timescale 1ns/10ps
`celldefine
module ANTENNA (A);
inout A;

endmodule // ANTENNA
`endcelldefine
# create technology library
foreach filename ( $tech_cell_lib_ver )
  ncvlog   -work   techlib   –append_log   -logfile   ../logs/ncvlog.log      $filename
end

# compile netlist
ncvlog -work ncsimlib -append_log –logfile ..logs/ncvlog.log $netlist_name
```

The *ampkg* command line is then executed. The output file is a tar file containing the compiled model file, along with the installation and user guide and associated support files required to make use of the model in various simulation environments.

```
# create model package
exec ampkg -control ./ampkg_control.$ctl_file_ext \
   -output ../data/$model_name.$amp_os -logfile ../logs/ampkg.log \
   ncsimlib.$module_name –overlay
```

Timing Model

Black box timing models (Liberty, .lib), of the hardened core are created in the *data* directory. They are generated at the worst and best case process corners by Encounter using the ***generate_timing.tcl*** script.

As part of the model generation, a shell verilog module is written out that is used to instance the design as part of the validation flow for both timing models and abstracts. The shell module name is defined to be *cortexm0_shell*.

Physical Abstract and Antenna Model (LEF)

The SI Closure section of the implementation flow creates a LEF abstract for the hardened core in the ***data*** directory called *cortexm0.lef*. In order to include the antenna data in this LEF, it is necessary to run ***verifyProcessAntenna*** prior to writing out the ***LEF***. This is shown in the following commands:-

```
#-------------------------------------------------------------------------
# Generate LEF and IO file for LEF validation flow
#-------------------------------------------------------------------------
verifyProcessAntenna                                                -reportfile
../reports/postroute/cortexm0.verifyProcessAntenna.rpt                 –leffile
../data/cortexm0.verifyProcessAntenna.lef
lefOut     ../data/cortexm0.lef
saveIoFile ../data/cortexm0.order.io
```

By changing the –5.5 flag to –5.3 or –5.4 the tool outputs LEF in those versions. Note that it is only possible to include antenna descriptions in LEF v5.5.

The IO order file is saved for use in abstract validation.

Noise Model Generation

During the SI and static timing analysis section of the Signoff steps, ETS SI Analysis creates a noise model for the core in the three process corners. These files can be found at:

../data/<corner>_noise_models/cortexm0_ <corner>.cdb

Static Power Model Generation

Run as part of Static power analysis.

Timing Model Validation

There is a flow to help in validating the generated models to ensure that they correctly describe the final hardened core and can be used by the Cadence tool set in terms of parsing and data content.

Currently, validation is available for:-

- Timing models (Liberty)
- Physical Abstract (LEF)

Model Validation is run in the *work* directory, and is run after model generation using the ***make validate*** *command*. The following paragraphs give a description of the flow employed for both timing and abstract validation.

Timing Model Validation is run within Encounter using ***TCL*** procedures called *write_model_timing.tcl* and *compare_model_timing.tcl*, which are delivered with the Encounter installation. These procedures are referenced within the *generate_timing.tcl, and validate_timing.tcl* scripts in the *scripts* directory. The model can be validated by comparing the arcs and/or slacks in the design and in the generated model. The user can specify a tolerance for judging how accurate the model is when compared to the actual core.

The following outlines the flow used by the procedure in order to perform the validation:-

- Initialise and load libraries (`source cortexm0_config.tcl`)
- Read in source Verilog, SPEF and SDC
- Measure input and output slacks for each requested point, storing them in an array
- Remove all design components
- Read in timing shell and timing model – .lib
- Read SDC
- Measure input and output slacks for each requested point, storing them in an array
- Compare source array versus model array, outputting all IO out with both slack tolerances to a CSV file for easy analysis in Microsoft Excel

The flow generates 2 results file (for both best-case and worst-case) in the ***reports/models*** directory. These files can be easily viewed, with any large differences between the behaviour of the model and the core marked as a FAIL.

In some scenarios, a FAIL may appear in one of the comparison reports. The number of FAILs depends largely on the percentage tolerance specified during the comparison. Also, in the case of small numbers (e.g. a slightly positive or negative slack), the difference between the values for the model and the design may be quite small but represent a relatively large percentage. In these cases, the user may opt to ignore the FAIL after satisfactory analysis. The default values for percent tolerance in RM is 5% and default value for absolute tolerance is 0.001.The user can change the

percentage of tolerance and absolute difference by changing the values for –percent_tolerance and –absolute_tolerance switches in the timing model validation script

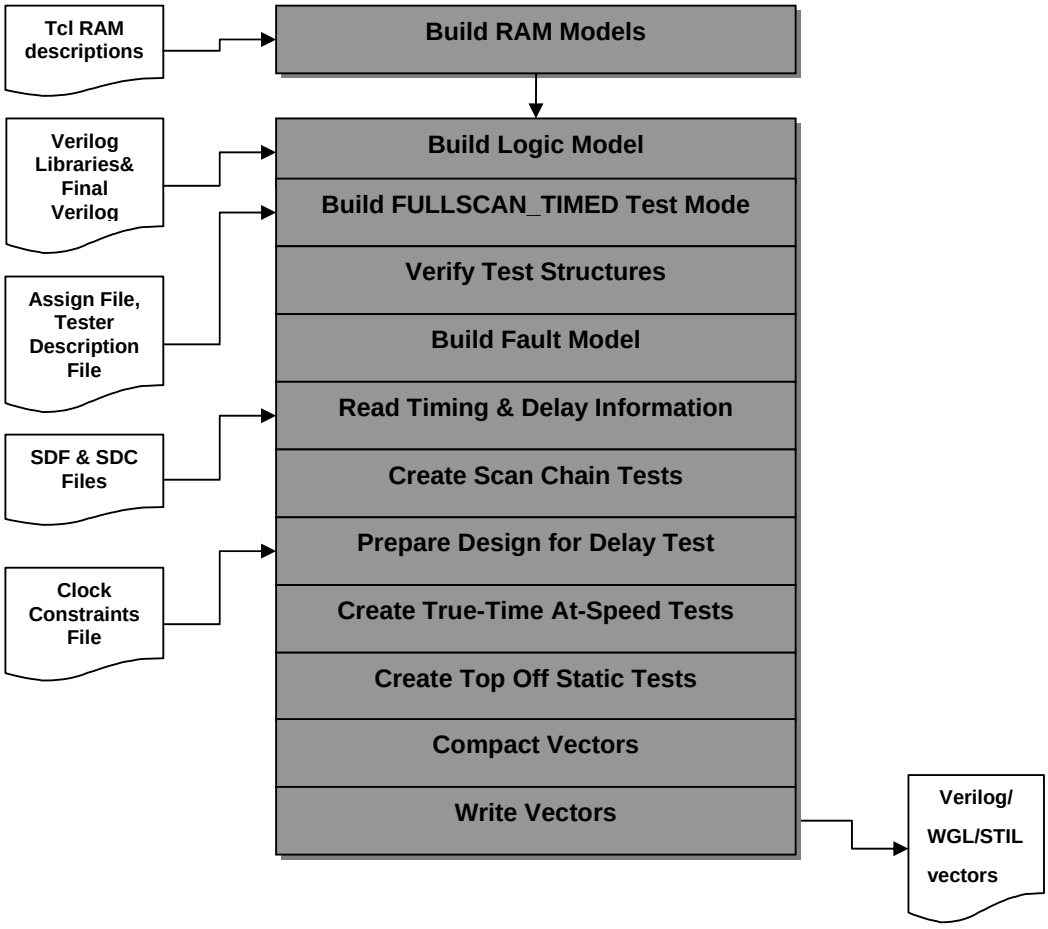
Physical Abstract (LEF) Model Validation

LEF validation ensures that the generated file can be read into Encounter without errors, and can also be used as a hierarchical element in a cell based P&R flow. The macro is placed at coordinates (50, 50) in a shell netlist to test routing to the core.

ATPG test generation and verification

The ATPG test generation and verification flow produces a set of test vectors which can be used to test the core for stuck-at and delay fault defects. Cadence Encounter Test (ET) reads the gate level netlist, SDF delay file and test pin description file, which are produced during the RTL to GDSII flow. It produces test vectors and a Verilog testbench which can be used to run gate level simulations.

ET is a standalone tool (i.e. not in the SOC Encounter suite). Users will require a separate license for this tool.



The flow is run from the **work** directory and can be started using the command **make atpg.** This runs Encounter Test using an executable script (**scripts/cortexm0_atpg.sh**) and configuration file (**scripts/cortexm0_atpg.config**). The executable script reads the configuration file and generates

the commands to be used with Encounter Test. The user should review the settings used in the configuration file before running the ATPG flow.

When the **make atpg** command is run, certain values are read from the **cortexm0_*config.tcl*** file and appended to the end of **cortexm0_atpg.config**. This ensures the same values are used for ATPG as for the rest of the flow.

This chapter explains the options used in the configuration file and gives more detail on each of the above steps. The user should consult the tool documentation for detailed information on the commands generated for each step.

Configuration file variables

The configuration file is used to define the paths and filenames of the input files, the type of ATPG to be run and the format to use when writing test patterns. There are a number of other variables which the user can use to further refine the ATPG flow if desired. The variables are commented in the configuration file and the user should refer to those comments in addition to this document.

The user may find it helpful to use the *EXECUTE*, *RESTART* and *EXITBEFORE* variables during their initial ATPG trials. These variables control which steps will be run and whether the scripts will execute the commands as they are generated or simply write a command file which can be used later.

Build RAM models

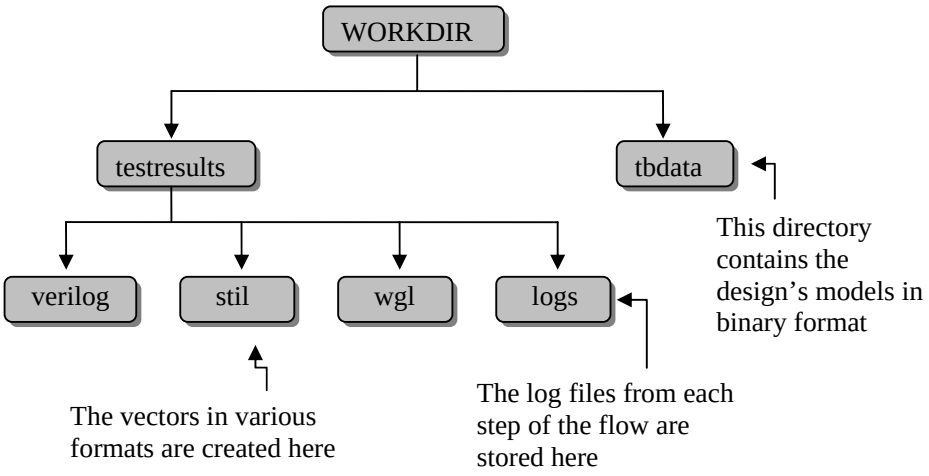
The first step in the ATPG flow is the creation and validation of RAM models using the *build_memory_model* command. With suitable RAM models, Encounter Test is able to create test patterns that perform read and write operations on the RAMs. This enables ET to improve test coverage by generating patterns to test the logic connected to RAMs, without the need for additional bypass logic.

This step is optional and not run by default. It requires the user to describe the pin names and operation of the library specific RAMs. The user should consult the tool documentation for further details on generating and validating RAM models.

Build model

This step reads the library and gate level netlists using the *TECHLIB* and *DESIGNSOURCE* variables. Encounter Test uses these files to build a model of the design, which it stores in the database. The database is updated after each step, allowing the user to stop and restart their run without repeating the intial steps.

The database is created in the directory specified by the *WORKDIR* variable. By default the scripts will create the database in the **work** directory. The directory structure is shown below.



Build test mode

Once the model has been built, the next step is to create an ATPG testmode. The testmode defines a specific configuration of the circuit and the limitations of the test equipment to be used.

When performing ATPG for a production design the user should enter the appropriate description of the test equipment during this step using a Tester Description Rule (TDR) file. Further information on how to do this can be found in the tool documentation.

This step uses the **cortexm0_assign** file, which is created by RTL Compiler during the synthesis step of the RTL to GDSII flow. This file defines the test related IO signals, listing the scan chain input and output pins, the scan enable pin, core level reset pins and clocks. Any additional test pins defined during synthesis (using the *rm_test_high_ports* and *rm_test_low_ports* variables) will also be listed.

The user can optionally use the *SEQDEF* variable to define a custom initialization sequence to configure the design to operate in the testmode. This option is not needed when running only the processor core. It may be of use when running ATPG on a complete chip level design.

Verify test structures

Encounter Test checks the design conforms to a set of test guidelines before the ATPG pattern generation is started. These checks look for conditions that might allow invalid data to be produced or which might reduce the test coverage.

By default the scripts run these checks in two steps. The first step checks that all flip flops and latches are placed on scan chains which can be controlled and observed. The logfile will detail any flip flops or latches that do not pass these checks.

The second step runs the remaining checks for the defined testmode. By default the scripts use the FULLSCAN_TIMED testmode, which causes Encounter Test to use the “Generalize Scan Style" checks. These include:

- Three-state logic must be designed to avoid contention during test.
- Feedback loops must not exist during test.
- The clocks must be controllable during test.
- Clock gating and other clock circuits must be controllable during test.

The user should check the results of this step and investigate all warnings generated. The tool documentation lists the full list of checks performed by this step and explains the intention of each rule.

Build fault model

This step builds the fault model files in the database. These are used to record the status of each fault in the design and are updated as tests are generated. Since the status is saved in the database, each test generation step will target only the faults that were not tested by the previous steps. This reduces tool runtime and the number of patterns generated. This also enables the user to restart an ATPG run without having to regenerate all previous tests.

Create scan chain delay tests

This step generates a serial shifting pattern which is used to test the connectivity of the scan chains. Only a few patterns need to be generated, so this step will run relatively quickly. At the end of the step the tool will report the number of static and dynamic faults that were tested by the patterns.

The user should examine the logfile to check how many faults were found during this step. If the logfile reports that no faults were tested the user should review the logfile from the verify_test_structures step and review the warnings reported.

Commit tests

After the scan chain tests have been generated, the scripts commit the tests. This saves the test patterns in the ET database and updates the fault models. The faults that were detected by the scan chain tests are marked so that subsequent test generation steps do not generate additional tests to detect the same faults again.

The scripts always commit the tests generated after each test generation step.

Read SDF

This option step reads in an SDF file, which is produced during the signoff STA analysis. The SDF file must correspond to the netlist being used for ATPG. Encounter Test uses the delay information from the SDF file to build a delay model, which is used to determine which paths to stimulate and the timings to use when testing for delay faults.

If the user does not specify an SDF file to read, the delay faults will be generated without reference to the design frequency. The user will need to determine the maximum frequency the vectors can support by running gate level simulations or debugging the vectors on the test equipment.

Read SDC

This is an optional step which can read in a timing constraint file in SDC format. Encounter Test will use the false path and path disable commands to mark unconstrained paths. Multicycle path definitions will be used to mark slow logic paths, which ET will try to avoid during delay testing. If the constraints include case analysis commands these will be used to mark nets that should be held constant.

Prepare logic delay

This step runs a trial ATPG to automatically determine a set of optimum clocking sequences for use by the test generation process. By default the clocking sequences are chosen to maximise the test coverage, but do not necessarily run at the same frequency as the functional clock. They might not include the most timing critical paths. ET will ensure that 99.5% of the paths may be clocked correctly by the sequences it automatically creates.

If the user wants to ensure the tests will run at the functional clock frequency (or faster than the functional frequency) they will need to create one of more clock constraint files. These can be read by the scripts using the CLOCKCONSTRAINTS variables in the configuration file. Further

information on the syntax and operation of the clock constraint files can be found in the tool documentation.

Create logic delay tests

This step will perform delay test generation on the logic faults in the design. It will generate ATPG vectors and internally simulate them. By default this step will use the clocking sequences generated by prepare logic step. At the end of this step the tool reports the number of faults that have been tested as well as the number untested, redundant or ignored.

Compact vectors

If the SORTPATTERNS variable is set to "yes" in the configuration file, the vectors will be resimulated and compacted. This step will reduce the number of vectors needed with minimal impact on the test coverage.

Write vectors

By default the vectors are written in Verilog format. The user can also generate vectors in STIL and WGL format by changing the values of the WRITESTIL and WRITEWGL variables in the configuration file.

Logical Equivalence Checking With Conformal

Overview

Cadence Conformal is used to perform logical equivalence checking (LEC) of the final netlist against the initial configured RTL (pre-synthesis). Conformal is run using the control script that is generated by RTL Compiler during synthesis (**data/cortexm0mp_rtl_vs_netlist.do**).

The following figure gives a graphical view of the LEC flow, with its inputs and outputs.

Figure 1 : Conformal Flow Diagram

Reading Libraries and Designs

By default the control file that RTL Compiler generates (**data/cortexm0_rtl_vs_netlist.do**) will use Verilog simulation libraries to model the technology specific gates. The user can edit the control file to read the liberty timing models if required, but it is recommended to run the comparison using the Verilog models as this will expose any differences between the simulation views and the liberty models.

The RTL file list in the control file is generated from the files read during synthesis. Since RTL Compiler uses its own library of synthesisable code to substitute DesignWare™ instances, the LEC control file may also need to read models from the RTL Compiler library.

Disable Scan

The control file used during formal verification applies constraints to the scan enable pin on both the RTL design and the gate level netlist. This is needed to prevent Conformal reporting that the design is inequivalent during scan shift.

By default the scan chain output pins will be ignored during the comparison as they do not exist in the RTL files. Conformal would report these pins as unmapped points and the comparison result would be reported as "inconclusive".

Configure Conformal

There are a number of configuration options used within Conformal to help it map and reduce the logic described by the RTL and the final gate level netlist. These settings are automatically generated by RTL Compiler and the user should not need to adjust them. Further details on the effect of the settings can be found in the Conformal tool documentation.

Checking Logfile for Equivalence

Conformal does not generate a report file. The result of the comparison is recorded in the tool logfile, stored in the ***logs*** directory. By default the run will use hierarchical comparison to reduce the tool runtime. An example of successful equivalence checking in a hierarchical run is shown below.

```
================================================================================
Module Comparison Results
--------------------------------------------------------------------------------
Equivalent                    6
--------------------------------------------------------------------------------
Total                         6
--------------------------------------------------------------------------------
Hierarchical compare : Equivalent
================================================================================
```

During a hierarchical comparison it should be expected that some lower level modules will be found to be inequivalent: both RTL Compiler and First Encounter are able to optimize logic across multiple levels of hierarchy. This might introduce additional ports or logic inversions in the lower level modules.

Conformal automatically handles such lower level inequivalences when running hierarchical comparison - it will black box only the lower level modules that are found to be equivalent.

When a module is found to be inequivalent it is not black boxed. Instead it is included in the comparison of any module that references it. In this way the lower level module is rechecked in the context of the next level of logical hierarchy. Ultimately the module will either be included in a level of hierarchy that is found to be equivalent, or the inequivalence will be reported in the final comparison set, which compares the whole design.

The user should check the logfile for any inequivalence in the top level module.

A APPENDICES

A.1 Floorplanning

The floorplanning stage is concerned with devising a viable layout for the core complete with an adequate power structure.

The iRM comes with a floorplan which should be a good starting point for any further floorplanning work. Even if the iRM is being completely retargeted at another process this file should be useful.

Another way to generate a starting point for floorplanning is simple to run the cortexm0_floorplan.tcl script with no floorplan file or def file present. This will then cause the tool to run a masterplan based floorplan creation. While often not 100% final floorplan quality, this is an excellent way of generating ideas for floorplanning the cores. For example on the Cortex-A8 the masterplan derived floorplan is the best floorplan so far found for a single pass flat flow.

The planDesign/masterplan run in the flow is a very basic one – see the SoC Encounter User guide for more information about masterplan and automated floorplan generation and analysis of multiple floorplans.

Floorplan refinement begins after a prototype floorplan shows enough promise to move to a more extended analysis and refinement stage to reduce congestion and minimize wire lengths. At this stage, you determine whether the candidate floorplan can be placed and routed, and whether it meets timing constraints.

A.1.1 Floorplan Refinement

Having generated a rough idea of the floorplan, it is then time to refine it and produce a final quality floorplan that can be run through the flow.

This library and netlist information can be read in by sourcing the *fe.conf* file in the floorplan scripts directory

```
#-------------------------------------------------------------
# Load the verilog and libraries
#-------------------------------------------------------------
source ../scripts/cortexm0_fe.conf
commitConfig
```

The power grid script that comes with the iRM is process specific. The implementation generally assumes extra power is coming from above. This can be either from flip chip bumps or further dense grids on higher layers. It is important to create a power grid correct for your technology before trying out placement and congestion. Always save the floorplan before adding power though.

A.1.2 Placement

Once the floorplan is defined, Amoeba placement is run to place the standard cells in the design. The tool takes account of any pre-placed modules and places the standard cells based on the connectivity and timing of the design.

Use *Place – Place* to bring up the Place form. There are several options for placement effort – at this point, *Prototyping* is usually selected, to perform a quick placement of the design. Also, scan reordering can be switched off.

A.1.3 TrialRoute to check congestion

Running a trial route on the placed design allows you to accurately gauge routing congestion and check routing connectivity.

Route – Trial Route opens the Trial Route form. There are various options to use, depending on what stage of floorplan development you are at. *Medium Effort* is normally selected at this stage.

Here, *maxRouteLayer* should be used to tell the tool not to route above metal *$rm_top_routing_layer*. After running the trial place and route runs, analyse the results for congestion. The floorplan can then be refined based on the results and the process repeated.

A.1.4 Timing and CTS

At this stage it may be desirable to do some optimization to see if the timing constraints are realistic. Also the user may want to check the clock tree possibilities, but these are not covered here.

End