CoreSight™ Components

Implementation Guide

Confidential

ARM®

ARM DII 0143E

CoreSight Components
Implementation Guide

Release Information

The *Change history* table shows the release state and change history of this document.

Change history

Date	Issue	Confidentiality	Change
6 January 2006	A	Confidential	First release
31 July 2006	B	Confidential	Updated to include Serial Wire blocks
20 December 2006	C	Confidential	Amended directory name for Serial Wire JTAG Debug Port on page 1-4
11 June 2007	D	Confidential	Corrections and enhancements
20 July 2007	E	Confidential	Corrections and enhancements

Proprietary Notice

Confidentiality Status

This document is Confidential. This document may only be used and distributed in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Product Status

The information in this document is final, that is for a developed product.

Web Address

http://www.arm.com

Contents
CoreSight Components Implementation Guide

List of Tables
CoreSight Components Implementation Guide

List of Figures
CoreSight Components Implementation Guide

Preface

This preface introduces the *CoreSight Components Implementation Guide*. It contains the following sections:

About this guide

This guide provides a description of the implementation of the CoreSight *Design Kit* (DK) components.

Product revision status

The r*n*p*n* identifier indicates the revision status of the product described in this guide, where:

r*n* Identifies the major revision of the product.

p*n* Identifies the minor revision or modification status of the product.

Intended audience

This guide is written for ASIC implementation engineers. It assumes some experience of the complete design cycle for hardware generated using *Register Transfer Level* (RTL) source, from RTL verification, through synthesis, to post place and route verification of the design. It also assumes that you have some knowledge of Verilog and scripting languages such as *Tool Command Language* (TCL).

Using this guide

This guide is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an introduction to the CoreSight components, their development platform and tools, the supported design flow, and the design hierarchy.

Chapter 2 *RAM Integration*

Read this chapter for information on the organization of the CoreSight ETB RAM block, and how to integrate your own RAM blocks into the ETB.

Chapter 3 *RTL Verification*

Read this chapter for information on how to use the test vectors.

Chapter 4 *Synthesis*

Read this chapter for information on how to perform synthesis. This chapter describes the supported design flow, the synthesis scripts, and how to set up and run synthesis. This chapter also contains a detailed description of the synthesis process and how to save your design.

Chapter 5 *Functional Verification of Netlist*

Read this chapter for information on how to perform functional verification of the design netlist. This chapter introduces functional verification and gives methods for achieving verification using equivalence checking, using test vectors and using the CoreSight DK model.

Chapter 6 *Production Test*

Read this chapter for a description of the supported flow for production test, and information on the test wrapper and scan test ports. This chapter also contains information on using Synopsys TetraMAX for *Automatic Test Pattern Generation* (ATPG).

Chapter 7 *Example CoreSight Subsystem*

Read this chapter for information on the example CoreSight DK subsystem provided as an example of how to connect the CoreSight components together, and how to configure, modify, and implement the example subsystem.

Chapter 8 *Sign-Off*

Read this chapter for information on how to satisfy the requirements for signing off the design. This chapter covers RTL verification, post synthesis functional and formal verification, and timing sign-off.

Conventions

Conventions that this manual can use are described in:

- *Typographical*
- *Signals* on page xiv
- *Numbering* on page xv.

Typographical

The typographical conventions are:

italic Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.

bold Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`mono`space	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`monospace italic`	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
`monospace bold`	Denotes language keywords when used outside example code.
< and >	Angle brackets enclose replaceable terms for assembler syntax where they appear in code or code fragments. They appear in normal font in running text. For example: • `MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>` • The Opcode_2 value selects which register is accessed.

Signals

The signal conventions are:

Signal level	The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means HIGH for active-HIGH signals and LOW for active-LOW signals.
Lower-case n	Denotes an active-LOW signal.
Prefix A	Denotes global *Advanced eXtensible Interface* (AXI) signals:
Prefix AR	Denotes AXI read address channel signals.
Prefix AW	Denotes AXI write address channel signals.
Prefix B	Denotes AXI write response channel signals.
Prefix C	Denotes AXI low-power interface signals.
Prefix H	Denotes *Advanced High-performance Bus* (AHB) signals.
Prefix P	Denotes Advanced Peripheral Bus (APB) signals.
Prefix R	Denotes AXI read data channel signals.
Prefix W	Denotes AXI write data channel signals.

Numbering

The numbering convention is:

<size in bits>'<base><number>

This is a Verilog method of abbreviating constant numbers. For example:

- 'h7B4 is an unsized hexadecimal value.
- 'o7654 is an unsized octal value.
- 8'd9 is an eight-bit wide decimal value of 9.
- 8'h3F is an eight-bit wide hexadecimal value of `0x3F`. This is equivalent to b00111111.
- 8'b1111 is an eight-bit wide binary value of b00001111.

Further reading

This section lists publications from ARM and third parties that provide additional information on developing code for the ARM family of processors.

ARM periodically provides updates and corrections to its documentation. See `http://www.arm.com` for current errata sheets, addenda, and the ARM Frequently Asked Questions list.

ARM publications

This guide contains information that is specific to the CoreSight components. See the following documents for other relevant information:

- *CoreSight Components Technical Reference Manual*, ARM DDI 0314
- *AMBA® AHB Trace Macrocell (HTM) Technical Reference Manual*, ARM DDI 0328
- *AMBA AHB Specification (Rev 2.0)*, ARM IHI 0011
- *ETM Architecture Specification*, ARM IHI 0014
- *AMBA 3 APB Protocol v1.0 Specification*, ARM IHI 0024
- *CoreSight Architecture Specification*, ARM IHI 0029
- *CoreSight Technology System Design Guide*, ARM DGI 0012
- *Systems IP ARM11AMBA (Rev 2.0) AHB Extensions*, ARM IHI 0023.

Feedback

ARM welcomes feedback on the CoreSight components and their documentation.

Feedback on the CoreSight components

If you have any comments or suggestions about these products, contact your supplier giving:

- the product name
- a concise explanation of your comments.

Feedback on this guide

If you have any comments on this guide, send e-mail to `errata@arm.com` giving:

- the document title
- the document number
- the page number(s) to which your comments apply
- a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Chapter 1
Introduction

This chapter introduces the CoreSight components. It contains the following sections:

1.1 Development HDL, platform, and tools

This section provides details of the platform and tools used in the development of the CoreSight components:

- *Development HDL*
- *Development platform*
- *Development tools*.

1.1.1 Development HDL

The CoreSight components were developed using the Verilog language. This is the only supported source language.

1.1.2 Development platform

The CoreSight components were developed and tested on Red Hat Linux.

1.1.3 Development tools

The CoreSight Release Notes list the tools used to develop the CoreSight components.

Note

The Release Notes list the tools used to verify the CoreSight components. ARM cannot guarantee that the deliverables are compatible with other versions of these development tools.

1.2 Supported design flow

Figure 1-1 shows the supported design flow.

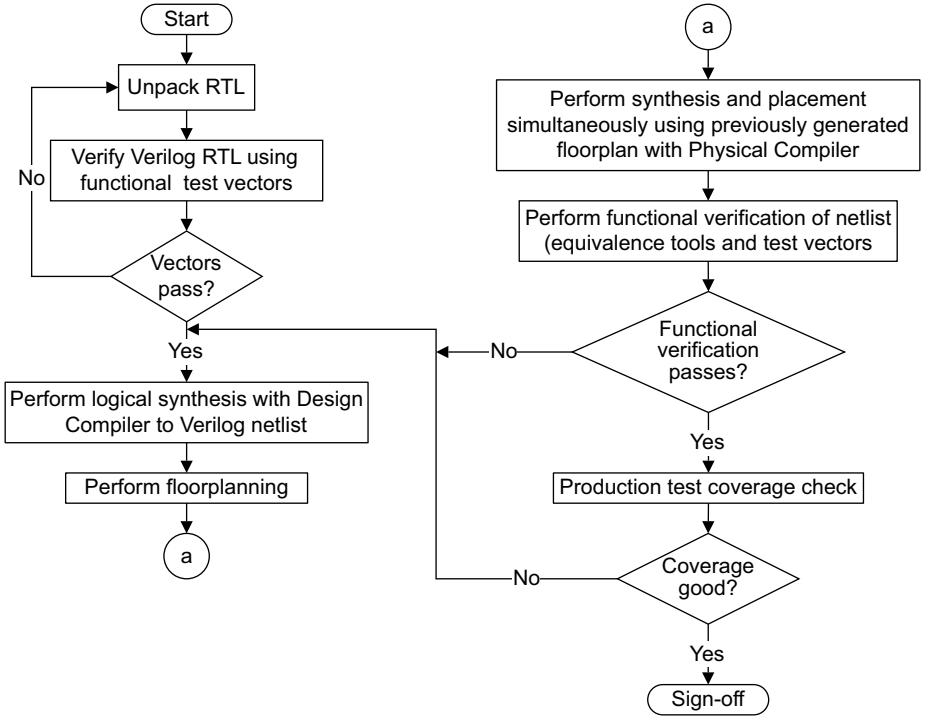

Figure 1-1 CoreSight design flow overview

Note

In addition to your normal SoC flow sign-off checks, you must satisfy certain verification criteria before you sign off your CoreSight design. See Chapter 8 *Sign-Off*.

1.3 CoreSight block summary

The CoreSight blocks are:

BRIDGESYNC1T1
: Synchronous 1:1 ATB Bridge.

CSATBUPSIZER
: ATB Upsizer.

CSBRIDGEASYNC
: ATB Asynchronous Bridge.

CSCTI
: Cross Trigger Interface.

CSCTM
: Cross Trigger Matrix.

CSETB
: Embedded Trace Buffer.

CSHTM
: AHB Trace Macrocell.

CSITM
: Instrumentation Trace Macrocell.

CSREPLICATOR
: ATB Replicator.

CSSWO
: Serial Wire Output.

CSSWV
: Serial Wire Viewer.

CSTFUNNEL
: Trace Funnel.

CSTPIU
: Trace Port Interface Unit.

CSTPIULITE
: Trace Port Interface Unit Lite.

DAP
: Debug Access Port.

  Although synthesis is only performed at the top level for the DAP block, this block contains additional directories:

  `DAPAHBAP` AHB Access Port.

  `DAPAPBAP` APB Access Port.

  `DAPAPBMUX` APB Multiplexor.

  `DAPJTAGAP` JTAG Access Port.

  `DAPSWJDP` Serial Wire JTAG Debug Port.

  `DAPROM` ROM Table, referred to as DAPROM in this guide.

DAPLITE
: Debug Access Port Lite.

Although synthesis is only performed at the top level for the DAPLITE block, this block contains additional directories:

`DAPAPBAP` APB Access Port.

`DAPAPBMUX` APB Multiplexor.

`DAPSWJDP` Serial Wire JTAG Debug Port.

`DAPROM` ROM Table, referred to as DAPROM in this guide.

In this chapter, replace `<CSBLOCK>` with the name of the appropriate block.

Note

- Not all of the blocks listed are shipped with all versions of the product. See the Release Notes for a list of the blocks supplied with the version of the product you have received.
- Where the block name is upper case, it is indicated as `<CSBLOCK>`, and where it is lower case, it is indicated as `<csblock>`.
- The example CoreSight *Design Kit* (DK) subsystem described in Chapter 7 *Example CoreSight Subsystem* follows the same directory structure as the CoreSight blocks.

1.4 Directory structure

Figure 1-2 on page 1-7 shows the principal directory structure of the CoreSight deliverables as they appear before overlaying with the ARM CPU deliverables.

You can overlay the directory structures for the CoreSight components and the ARM CPU. All the files are unique, therefore no files are lost during this process.

Table 1-1 shows the top-level directories required for each stage of implementation.

Table 1-1 Implementation stages and directories required

Implementation stage	Directories required
Functional verification of RTL	`logical`, `implementation/<csblock>/vectors`
Synthesis	`logical`, `implementation/<csblock>/synopsys`
Functional verification of netlist	`logical`, `implementation/<csblock>/cadence`, `implementation/<csblock>/synopsys`, `implementation/<csblock>/vectors`
Production coverage test	`implementation/<csblock>/synopsys`

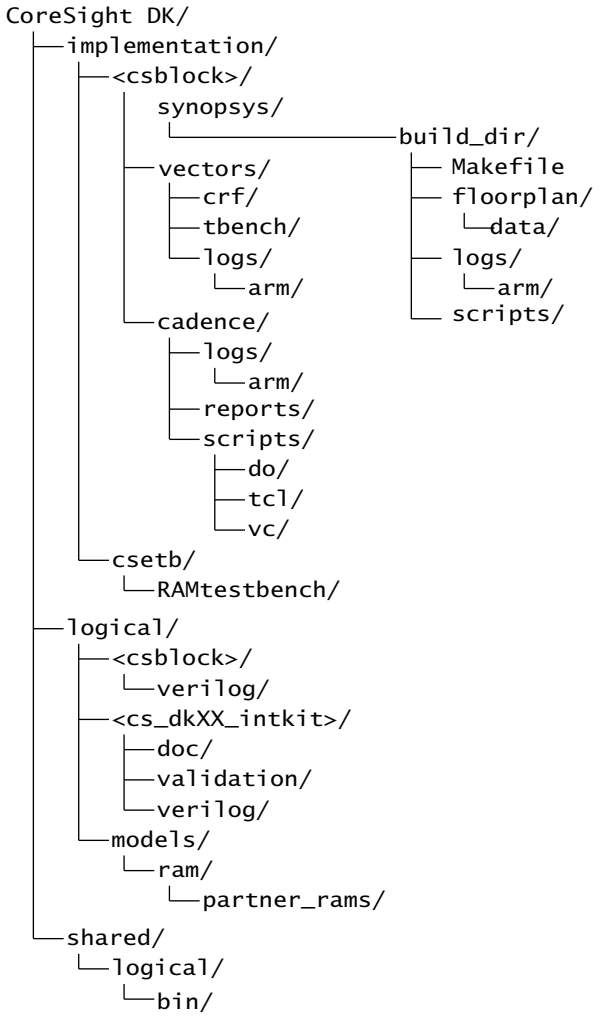

Figure 1-2 Release structure

Note

In Figure 1-2, `<cs_dkXX_intkit>` represents the applicable CoreSight integration kit.

1.5 Before you begin

Before you begin implementation, you must read all of the `README` files that are released with the CoreSight deliverables. See the Release Note for a list of locations where `README` files are located.

1.6 RTL configuration parameters

The RTL deliverables contain Verilog `defines for amendment by partners. These are described in:

- *ROM table Verilog `defines*
- *Cross Trigger Interface Verilog `define*
- *CoreSight ETB RAM size* on page 1-10
- *CoreSight subsystem* on page 1-10.

1.6.1 ROM table Verilog `defines

When implementing CoreSight components in your system, you must amend various parameters. The Verilog `defines that you can specify in daprom/verilog/DAPRomDefs.v are:

- ROM table entries:
  - ENABLE_xx where xx = {16, 24, 32}
  - ROMENTRYxx where xx = {00 to 31}.
- ROM table Peripheral IDs:
  - ROM_PERIPHIDx_VAL where x = {0 to 4}.

ENABLE_xx reflects the number of ROMENTRYxx entries that are used. For example, to use ROMENTRY00 to ROMENTRY15, specify ENABLE_16.

To define entries in the ROM table, starting at ROMENTRY00 and continuing sequentially for each component:

- Replace ROM_TERMOFFSET with the base address of the component. This is 20 bits.
- Change ROM_TERMFORMAT to ROM_FORMAT. Specifies 32-bit entry type.
- Change ROM_TERMNOENTRY to ROM_ENTRY. Indicates an entry is present.

The ROM_PERIPHID{0-4}_VAL `defines correspond to the PeripheralID{0-4} register locations whose values must be set to reflect the part number of the ASIC, the designer of the ASIC (JEP106 identity), and appropriate revision of the ASIC.

For more information on the ROM table, see the *CoreSight Components Technical Reference Manual*.

1.6.2 Cross Trigger Interface Verilog `define

In cscti/verilog/CSCtiDefs.v, EXTMUXNUM is a user-definable parameter that you can amend. See the Verilog file and the *CoreSight Components Technical Reference Manual* for more information on how to use this parameter.

1.6.3 CoreSight ETB RAM size

csetb/verilog/CSEtbDefs.v contains the Verilog `define CSETB_RAM_ADRW that you can amend. See the comments in the ETB RAM testbench file for details of how to use this parameter.

1.6.4 CoreSight subsystem

cssys/verilog/CSSysDefs.v contains the following Verilog `defines that can be amended:

- CSSYS_BRIDGE_PORT
- CSETB
- CSTPIU.

Note

CSSYS_BRIDGE_PORT exists as CSSYS_BRIDGE_PORT0 to CSSYS_BRIDGE_PORT7 inclusive.

1.6.5 HTM configuration

Two procedures are required before the HTM deliverable is used:

- Because you can configure the Verilog file into HTM64 (HTM for 64-bit AHB), HTM32 (HTM for 32-bit AHB), or HTM32L (HTM for 32-bit AHB with a 64 byte FIFO), you must configure the Verilog code. By default the HTM64 configuration is used.
- For the same reason, you must set up a system variable called CSHTM_MODE so that various scripts can determine if the design flow is for 64-bit version or 32-bit version (normal or large FIFO).

Both steps can be performed by running the cshtm_cfg.csh C-shell script.

```
>cshtm_cfg.csh HTM64          # Setup the files and environment for HTM64
>cshtm_cfg.csh HTM32          # Setup the files and environment for HTM32
>cshtm_cfg.csh HTM32_FIFO64   # Setup the files and environment for HTM32L
```

Chapter 2
RAM Integration

This chapter describes the CoreSight ETB RAM organization, and how to integrate your RAM block into the processor. It contains the following sections:

2.1 About RAM integration

The CoreSight ETB contains one RAM block that you must instantiate at the CSETB level of hierarchy.

Note

The RAM block example shown in this chapter is for a 4KB RAM size. If your implementation requires a different RAM size, before performing RAM integration, ensure that the implementation flow is validated for a 4KB ETB using the replay vectors supplied, and continue with synthesis in Chapter 4.

2.1.1 Requirements

You must ensure your library RAM satisfies the following requirements:

- all timings must be synchronous to the rising clock edge
- single-cycle access
- chip select (RAM enable) control required
- RAM outputs must always be valid, do not tristate.

2.2 Supported RAM integration flow

Figure 2-1 shows the supported RTL validation flow that enables you to:

- identify the RAM organization required for each RAM block, see *RAM organization* on page 2-4
- integrate your library RAM, see *Integrating your RAM block* on page 2-8
- test that RAM integration has been successful, see *RAM integration testbench* on page 2-10.

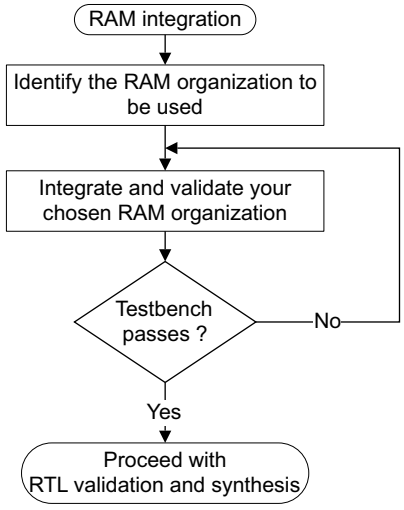

Figure 2-1 RAM integration flow

2.3 RAM organization

Table 2-1 shows the instantiated RAM block in the ETB RAM module.

Table 2-1 Instantiated RAM blocks

RAM block	Instance name	Description
CSEtbRam	uCSEtbRam	Trace Buffer RAM

Table 2-2 shows the RAM size required for the ETB RAM module. Although the table shows RAM sizes up to 32KB, the RAM size can be up to 1024KB. The ETB RAM supports word write only.

Table 2-2 RAM module RAM size

RAM block	Write enable	RAM sizes					
		1KB	2KB	4KB	8KB	16KB	32KB
CSEtbRam	Word	256x32	512x32	1024x32	2048x32	4096x32	8192KB

Note

If you instantiate a larger RAM than required, for example if your RAM generator cannot produce a RAM of the required size, you must tie off the redundant upper address bits to 1'b0.

In the ideal case, you can produce a single block of compiled RAM for each block of RAM. This might not always be possible if your compiler cannot produce a single RAM block that is the required size, or a single RAM block might not meet the timing requirements. In this case, you must produce the RAM out of two or more blocks of smaller RAM. See *Producing a large memory from smaller RAM blocks* on page 2-6.

2.3.1 CSEtbRam RAM organization

Figure 2-2 shows the required RAM organization for `CSEtbRam`. The `CSETB_RAM_ADRW` `` `define `` is set in the `CSEtbDefs.v` file and defines the address bus width of the ETB RAM. As the example shows, a 4KB RAM, that is, 1024 words x 32 bits, requires `CSETB_RAM_ADRW` to be set to 10.

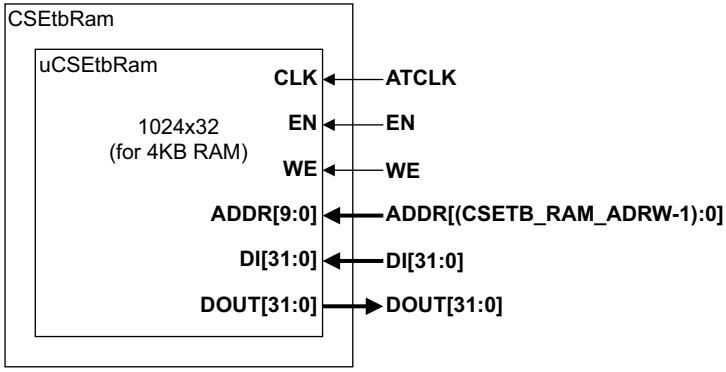

Figure 2-2 CSEtbRam organization

Table 2-3 shows the address bus **ADDR** bits that connect to the RAM for each size.

Table 2-3 CSEtbRam address bus connections

RAM size	Address connection
1KB	**ADDR[7:0]**
2KB	**ADDR[8:0]**
4KB	**ADDR[9:0]**
8KB	**ADDR[10:0]**
16KB	**ADDR[11:0]**
32KB	**ADDR[12:0]**
64KB	**ADDR[13:0]**
128KB	**ADDR[14:0]**
256KB	**ADDR[15:0]**
512KB	**ADDR[16:0]**
1024KB	**ADDR[17:0]**

2.3.2 Producing a large memory from smaller RAM blocks

You might have to create a large memory out of smaller RAM blocks, for one of the following reasons:

- your RAM compiler cannot produce a RAM of the required size
- a single large RAM is too slow for your performance requirements
- a single large RAM does not fit into your floorplan.

The rules for producing a memory out of smaller RAM blocks are:

- The number of RAM blocks must be a power of 2 (`b` = 2, 4, 8, for example).
- If the address width of the required memory size is `n` bits, the address port of the smaller RAM blocks is `m = [n - (log b/log2)]` bits wide. For example, if you create a RAM out of two smaller RAM blocks (`b`=2) and the required address width for that memory size is 10 bits (`n`=10), then the address width, `m`, of the two smaller RAM blocks is 9 bits.
- Address bits `[m-1:0]` are applied to all the RAM blocks.
- You must ensure that only the addressed RAM is enabled, by performing a `b` bit decode of the `[n-1:m]` address bits and ANDing these with the RAM enable control signal. In the example, address bits [8:0] are applied to the two RAM blocks and a 2-bit decode of address bit [9] are ANDed with the RAM enable to create two RAM enable signals, that is:

  ```
  assign RAMEnable_0 = ~Addr[9] & RAMEnable;
  assign RAMEnable_1 =  Addr[9] & RAMEnable;
  ```

  You must connect `RAMEnable_0` to the RAM enable port of the first RAM block, and `RAMEnable_1` to the RAM enable port of the second RAM block.

The approach is exactly the same for any memory that has to be constructed from smaller RAM blocks.

Figure 2-3 on page 2-7 shows the creation of a 4KB RAM out of two 2KB RAM blocks.

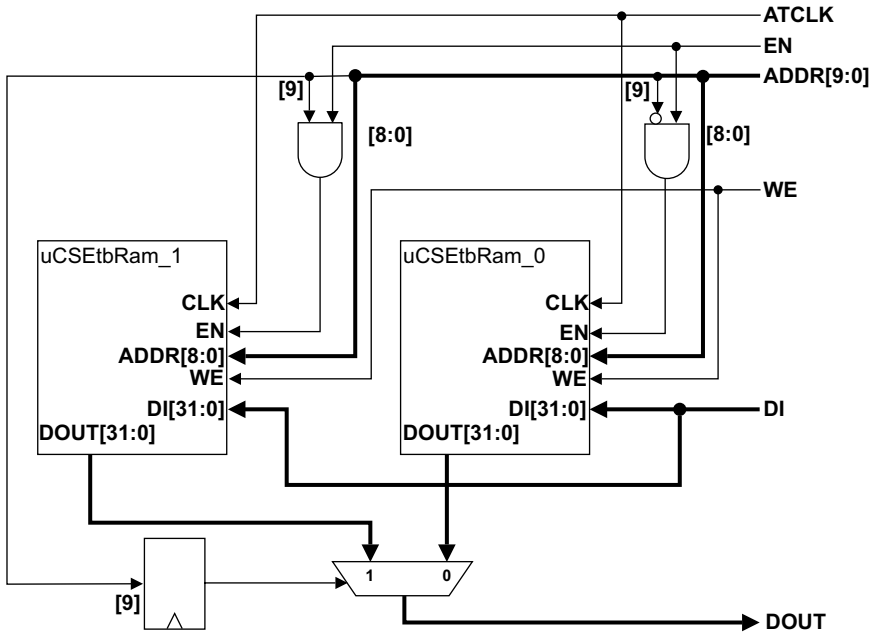

Figure 2-3 Creating large memory from smaller RAM blocks

2.4 Integrating your RAM block

This section describes how to integrate your RAM block. RAM integration is performed at the CSETB level. All the files you must edit are in the logical/models/ram/partner_rams directory.

To perform RAM integration:

1. Go to the logical/models/ram/partner_rams directory:

   ```
   cd logical/models/ram/partner_rams
   ```

2. Create a directory where you are to integrate your RAM, and go to this directory:

   ```
   mkdir <my_ram_dir>
   cd <my_ram_dir>
   ```

3. *RAM organization* on page 2-4 explains how to connect the RAM. Use this to identify the RAM blocks that you require then generate them using your library RAM generator.

4. Copy the CSEtbRam module into the directory you created in step 2. This is the module that you integrate your library RAM blocks into the Verilog RAM module from logical/models/ram/partner_rams/template:

   ```
   cp ../template/CSEtbRam.v.
   ```

5. Integrate your RAM blocks into the CSEtbRam module, adhering to the organization described in *RAM organization* on page 2-4. All RAM control signals are driven as active HIGH. If your RAM blocks have active LOW control inputs, you must invert the RAM write enable pin, **WE**. Write the port declaration of CSEtbRam as:

   ```
   .WE  (~YourWE),
   ```

6. Go to the logical/csetb/verilog directory:

   ```
   cd ../../../../csetb/verilog
   ```

7. Edit the CSEtbDefs.v file to specify the address bus width of the CSEtbRam as shown in Table 2-4.

Table 2-4 Address bus width encoding

RAM size	CSETB_RAM_ADRW
1KB	8
2KB	9
4KB	10

Table 2-4 Address bus width encoding (continued)

RAM size	CSETB_RAM_ADRW
8KB	11
16KB	12
32KB	13
64KB	14
128KB	15
256KB	16
512KB	17
1024KB	18

8. Check that the RAM integration is correct by running the *RAM integration testbench* on page 2-10.

2.5 RAM integration testbench

A testbench checks that your RAM blocks are correctly integrated.

Before running the testbench you must complete all steps in *Integrating your RAM block* on page 2-8.

To run the RAM integration testbench:

1. Go to the `/implementation/csetb/RAMtestbench` directory where the RAM integration testbench is run from:

   ```
   cd /implementation/csetb/RAMtestbench
   ```

2. Edit the `CSETB_ram_testbench.vc` and change:

   - the following lines to the directory where RAM integration was performed:

     ```
     // Edit the following lines and change 'template' to the directory into
     // which you performed RAM integration.
     +v ../../../logical/models/ram/partner_rams/template/CSEtbRam.v
     ```

   - the following `-y` line to the directory where the models for your library RAM blocks exist:

     ```
     // Edit the following line to point to the directory in which the verilog
     // models for your library RAMs exist.
     -y ../../../logical/models/ram/
     ```

3. Run the RAM integration testbench:

   - using VCS, for example:

     ```
     vcs -f CSETB_ram_testbench.vcs CSETB_ram_testbench.v
     simv
     ```

   - using ModelSim, for example:

     ```
     vlib work
     vlog -f CSETB_ram_testbench.vc CSETB_ram_testbench.v
     vsim -c CSETB_ram_testbench -do "run -all"
     ```

   If your RAM model performs setup and hold timing checks, you must disable the checks when running the testbench by adding the `+notimingchecks` option to the command line.

   The testbench is self-running. If your RAM integration has been successful, the simulation completes with the following message:

   ```
   RAM Integration Tests
   ----------------------
   CSETB RAM     word test passed
   CSETB RAM     bit test passed

   RAM integration passed. Test completed with no errors
   ```

If your RAM integration has been unsuccessful, the simulation is completed and a report is made that the RAM block failed integration, for example:

```
RAM Integration Tests
---------------------
CSETB RAM     word test failed
CSETB RAM     bit test failed

!!! RAM integration FAILED. Test completed with errors !!!
```

To assist you in debugging any errors the simulation also reports expected and actual RAM read data for failing RAM integration.

Note

The default directory setting in steps 2 and 3 simulate the example ARM RAM blocks. You might want to simulate the ARM RAM models as a golden reference if you have to debug any errors with your RAM integration.

4. If RAM integration is successful:
   - If your RAM size is 4KB, proceed with Chapter 3 *RTL Verification*, followed by Chapter 4 *Synthesis*.
   - If your RAM size is not 4KB, proceed directly with synthesis, because the RTL verification stage does not support other RAM sizes. You must perform RTL verification before RAM integration.
5. When RAM integration is complete, exit the simulator.

2.6 Memory BIST support

The CoreSight ETB has a memory BIST interface that you must connect up to your system BIST controller. See the *CoreSight Components Technical Reference Manual* for more information on the BIST interface.

Chapter 3
RTL Verification

This chapter describes how to use functional test vectors to verify that the RTL of each CoreSight component is delivered and unpacked correctly. You can use the same procedure to replay vectors on a netlist. This chapter contains the following sections:

Note

All CoreSight blocks are verified in the same way. In this chapter, replace <CSBLOCK> with the name of the appropriate block. See *CoreSight block summary* on page 1-4 for details on the blocks used in CoreSight.

3.1 Verification using test vectors

The test vectors enable you to check that all your <CSBLOCK> files compile and simulate correctly. This section provides information on how to replay functional test vectors using Verilog simulation. It contains the following sections:

- *Test vectors directory structure*
- *Test vector files*.

3.1.1 Test vectors directory structure

Figure 3-1 shows the test vectors directory structure.

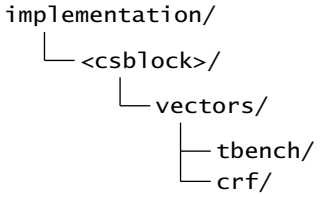

Figure 3-1 Test vectors directory structure

The purpose of each subdirectory is as follows:

`tbench` Contains the testbench and compile scripts for vector replay.

`crf` Contains the test vector files in *Condensed Reference Format* (CRF).

Note

The `CSHTM` block contains an additional directory under the `vectors` directory:

`vrf` Contains the test vector files in *Vector Replay Format* (VRF) at run-time.

Note

For the `CSETB` block, the `verilog.vc` file in the `tbench` directory must contain the path to the file holding the Verilog RAM model.

3.1.2 Test vector files

All test vector files have been compressed using the UNIX compression tool. The build scripts provided automatically decompress these files. A list of vectors is provided and a summary of their functionality is located in `<csblock>_test.list` in the `tbench` directory.

3.2 Building the simulation world and replaying vectors on RTL

To build the vector replay testbench for Verilog RTL simulation and replay vectors:

1. From your top-level CoreSight directory go to the vector build and replay area:
   `cd implementation/<csblock>/vectors/tbench`
   where `<csblock>` refers to the appropriate CoreSight block.
2. Ensure that your simulator is on the path. The scripts provided for vector replay on RTL are:

   BuildVerilogMTI
   Builds testbench and replays vectors using the ModelSim tools.

   BuildVerilogVCS
   Builds testbench and replays vectors using the VCS tools.

   BuildVerilogNC
   Builds testbench and replays vectors using the NC tools.

3.3 Analyzing vector simulation logs

A log file is created for each vector that is replayed. This file is located in `tbench/<sim_name>` where `<sim_name>` is:

- `VCS` if the vector was replayed using the VCS tools
- `NC` if the vector was replayed using the NC tools
- `MTI` if the vector was replayed using the ModelSim tools.

The log file is `vector_name_<sim_name>_RTL_Replay.log`, where `<sim_name>` is `VCS`, `NC`, or `MTI`.

The reference log files are located in the `tbench/logs/arm` directory.

When a test vector replays successfully, a message similar to the following appears at the end of the log file:

```
**Simulation end        16908 vector(s) applied        0 errors found**
```

Check the log files for all the replayed vectors to ensure no errors are found.

Chapter 4
Synthesis

This chapter describes synthesis of the CoreSight components. It contains the following sections:

Note

All CoreSight blocks are synthesized in the same way. In this chapter, replace <CSBLOCK> with the name of the appropriate block. See *CoreSight block summary* on page 1-4 for details on the blocks used in CoreSight.

4.1 About synthesis

This chapter provides information that enables you to get synthesis up and running as quickly as possible. It contains information on the files and scripts supplied, and how the synthesis flow uses them. You can examine the scripts if you require more detailed information on how they work.

Note

There is no restriction if you want to:

- modify the scripts to make optimizations for your technology
- update the scripts for any synthesis enhancements not available when the scripts were released.

The synthesis strategy is:

- to produce a complete netlist in a target technology that is functionally equivalent to the RTL description
- to satisfy all the imposed performance, area, and power goals.

This section provides information on the:

- *Synthesis directory structure*
- *Synopsys tool requirements* on page 4-4.

4.1.1 Synthesis directory structure

There is a pre-defined directory structure called build directory, `build_dir`, in which the `<CSBLOCK>` is implemented. This build directory is found in the synopsys directory.

You can have as many parallel build directories in the synopsys directory as you require to support concurrent implementations of a given `<CSBLOCK>`. You can name these build directories as you choose.

Figure 4-1 on page 4-3 shows the structure of the build directory.

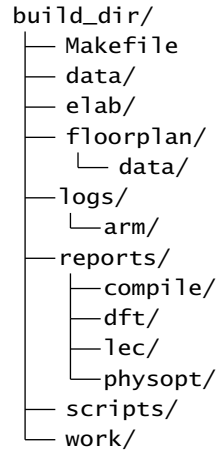

Figure 4-1 Synthesis directory structure

This directory structure is fixed, and you cannot change it without significant script modifications. All tool execution steps are performed in the `work` directory. Data generated by any step in the flow is stored in the `data` directory. The `elab`, `logs`, and `scripts` directories hold the intermediate file, logs files, and scripts respectively. Reports are written into the `reports` directory. A Makefile defines and exercises the various steps in the flow, and illustrates how to invoke the tools. You can add as much control and dependency as you require to the Makefile. Run `make dirs` to create the directory structure under `build_dir`.

The purpose of each component shown in Figure 4-1 is:

`Makefile`	Controls tool flow.
`data`	Stores output data generated during the flow.
`elab`	Stores intermediate files.
`floorplan`	Manages data used in the floorplanning stage.
`logs`	Maintains the transcripts for all tool execution.
`reports`	Stores reports generated during tool sessions.
`scripts`	Contains all flow and tool scripts.
`work`	Performs all tool execution steps.

4.1.2 Synopsys tool requirements

This section describes Synopsys tool requirements for synthesis:

The methodology and scripts are developed using the Synopsys tools specified in the CoreSight Release Notes. The methodology is portable across all operating systems supported by that release.

The following licenses are used to implement the <CSBLOCK>:

- HDL-Compiler for Verilog
- DC Ultra
- Design Compiler
- DesignWare Foundation
- DFT Compiler or Test Compiler
- Power Compiler
- Physical Compiler.

4.2 Supported synthesis flow

This section provides information on the synthesis flow for the example CoreSight block, <CSBLOCK>. The synthesis flow enables you to:

1. Set up the environment for the compiler.
2. Control the analysis and elaboration of the <CSBLOCK>.
3. Identify the operating environment for the <CSBLOCK>, including port characteristics, operating conditions, and interconnect models.
4. Establish specific performance goals for the design. Define clocks and interface timing. Define all required design rules that the resulting implementation must satisfy.
5. Map the design.
6. Synthesize the design.
7. Insert scan chains into the <CSBLOCK>.
8. Save the design and all associated derivative files including *Synopsys Design Constraints* (SDC) and testability information.

The synthesis flow described does not include details on multiple power domains or *Intelligent Energy Management* (IEM).

Note

This guide does not describe IEM implementation. Contact ARM Limited for details.

For information on implementing multiple power domains, see the *CoreSight Technology System Design Guide*.

Figure 4-2 on page 4-6 shows the <CSBLOCK> synthesis flow.

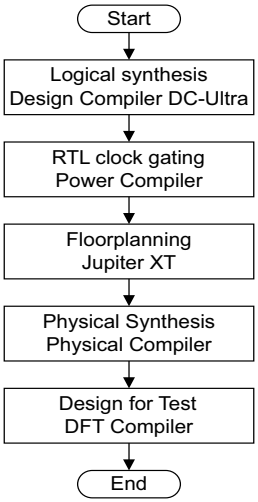

Figure 4-2 Supported synthesis flow

4.3 Flow setup

This section describes the synthesis of the <CSBLOCK> with Design Compiler and Physical Compiler to create a placed gates implementation that is functionally equivalent to the configured compliant RTL image. The synthesis must also meet the required performance, area, and power requirements. To satisfy strict time constraints, synthesis with DC-Topographical enabled Design Compiler followed by detailed placement with Physical Compiler is used.

4.3.1 Initial setup

You must use the configuration scripts in the scripts directory to control the overall flow through the tool steps. The two main configuration files contain:

- <CSBLOCK> specific parameters. The simple configuration, <csblock>_specific.tcl contains block specific variables used throughout the tool flow. This is sourced at the beginning of script execution.
- Flow control parameters. The flow configuration file, <csblock>_config.tcl, contains TCL variables that are used in the synthesis scripts to control the tool flow, and to define specific features of the flow.
- Technology-specific parameters and library setup. A separate file, <csblock>_tech.tcl defines these parameters. This file is sourced after <csblock>_config.tcl

Specific control parameters

The <csblock>_specific.tcl available in scripts contains variables which configure the synthesis scripts to the specific <CSBLOCK>. This includes block name, clock and reset grouping, and scan enable pin name.

Note

The CSHTM block has a configuration control rm_cshtm64 which, by default, is set to 1 for the HTM64, for a 64-bit AHB system. When synthesizing the HTM32 or HTM32L for a 32-bit AHB system, change rm_cshtm64 to 0.

Flow control parameters

The flow control options are minimized to limit the level of configurability and so ensure a predictable and effective flow.

Table 4-1 shows the flow parameters and default values available during synthesis. This section of the flow configuration script defines variables to describe elements and features of the flow.

Table 4-2 shows the DFT parameters and default values available during synthesis. This section of the flow configuration script defines variables used to control the DFT setup for the design.

Table 4-1 Synthesis flow control parameters

Option	Default	Function
rm_use_ungroup	1	When set, hierarchy is ungrouped before the compile step using the routine in <csblock>_ungroup.tcl.
rm_fp_exist	1	When set, the floorplanning step using Jupiter-XT runs using pre-existing replay files. The default is set.

Table 4-2 Synthesis DFT control parameters

Option	Default	Function
rm_num_scan_chains	-	Defines the number of scan chains in the design. This is based on the number of checks with <csblock>.
rm_use_scan_comp	0	When set, directs DFT-Compiler to perform Scan Data Compression.
rm_comp_ratio	10	Default value to use for pattern compression ratio.
rm_scan_data_in	SI	Defines name of internal scan-in ports added by the scan insertion process.
rm_scan_data_out	SO	Defines name of internal scan-out ports added by the scan insertion process.
rm_comp_enable	COMP_EN	Defines name of the port added to control scan compression, if enabled by rm_use_scan_comp.

You can use the options in Table 4-1 and Table 4-2 to control:

- how the synthesis takes place
- how scan structures are inserted into the design
- whether clock gating is used or not.

Technology-specific parameters

Technology-specific parameters are defined in the technology setup file, <csblock>_tech.tcl. These define the parameters associated with the clocks and input/output in the design. All the technology-specific information is found in this single script. This enables you to quickly port IP from one technology to another.

Optimization and constraint parameters

This section of the technology script defines variables that constrain the design and that set up the environmental parameters. The suggested values are for a typical Artisan 130nm implementation. You must ensure that you set them appropriately for your own implementation. Table 4-3 shows the optimization and constraint parameters.

Table 4-3 Optimization and constraint parameters

Option	Default	Function
rm_load_value	0.05	Specifies the capacitive load value for all design outputs (pF)
rm_driving_cell	BUFX4HS	Defines the cell type used as the driving cell for all design inputs
rm_driving_pin	Y	Defines the output pin name of the driving cell. The driving cell characteristics are applied to the design inputs
rm_clock_driving_cell	CLKBUFX16HS	Defines the cell type used as the driving cell for clock ports in the design
rm_clock_driving_pin	Y	Defines the output pin name of the driving cell. The driving cell characteristics are applied to the design clock ports
rm_target_setup_slack	0.1	Target slack for setup fixing (ns)
rm_target_hold_slack	0.0	Target slack for hold fixing (ns)
rm_max_transition	0.3	Maximum transition constraint for design nets
rm_max_capacitance	2.0	Maximum capacitance constraint (pF)
rm_target_utilization	70	Target utilization for area recovery
rm_max_utilization	95	Maximum utilization for area recovery
rm_max_fanout	32	Maximum fanout constraint
rm_latency	-	Predicted clock tree transition delay of [clockname]
rm_icg_latency	-	Latency of integrated clock gating
rm_[clockname]_period	-	Target clock period of [clockname]. If a block has multiple clocks, there are multiple entries.
rm_atclk_pclkdbg_dir	2	Only present in components with **PCLKDBG** and **ATCLK**. Used to define the synchronous relationship between **ATCLK** and **PCLKDBG**.
rm_clock_uncertainty	0.2	Specifies the uncertainty in the arrival time of every clock (skew) (ns)
rm_clock_transition	0.0	Maximum target clock transition for design clocks (ns)

Table 4-3 Optimization and constraint parameters (continued)

Option	Default	Function
rm_critical_range	0.2	Specifies the critical range, for example 10% of the rm_clock_period
rm_icg_name	TLATNTSCAX8HS	Defines the clock gating cell used by Power Compiler
rm_min_icg_fanout	4	Lower fanout limit for ICG cell
rm_max_icg_fanout	16	Upper fanout limit for ICG cell
rm_operating_condition	ss_1v08_125c	Defines the operation condition used during synthesis
rm_tie_high_cell	TIEHIHS	Cell used to tie signal high
rm_tie_low_cell	TIELOHS	Cell used to tie signal low
rm_process_corner	rcbest	Default process corner for use in STA

DesignWare setup for Formality

Table 4-4 shows the DesignWare setup parameters.

Table 4-4 DesignWare setup parameters

Option	Default	Function
rm_hdlin_dwroot	-	Defines location of the DesignWare installation. Formality uses this to resolve DesignWare instances. Usually set to the top of the tool install directory tree.

TetraMAX library settings

Table 4-5 shows the TetraMAX library settings.

Table 4-5 TetraMAX library settings

Option	Default	Function
rm_max_verilog_lib	-	Defines location of the TetraMAX model views for standard cells, including high-threshold library.

Floorplanning

The rm_aspect_ratio and rm_utilization parameters set the shape and size of the design.

The rm_core_offset parameter sets the space between the PR boundary and cell placeable area.

The rm_irdrop_target parameter sets the maximum permitted ir drop for power and ground wire creation. This parameter is used when rm_fp_exist is set to 0.

Design compiler and physical compiler

The rm_lib_dirs and rm_tech_dirs parameters are pointers to the root of the library and technology directories.

The rm_sc_reflib, rm_corner_core_lib, and rm_corner_lib_name parameters set the path to the standard cell library, the DB that is contained in each Milkyway LM view, and the name of the DB respectively.

The rm_target_library and rm_target_lib_name parameters set the target DB and lib name.

The rm_search_path parameter sets the search path to the Milkyway DB standard cell lib.

Milkyway

The mw_power_net, mw_ground_net, mw_power_port, and mw_ground_port parameters set the power and ground nets and ports.

The mw_reference_library parameter sets the path to the Milkyway standard cell reference library.

You can control the routability by setting the following parameters.

rm_max_layer sets the maximum permitted metal layer.

rm_size_rt_width sets the minimum width for clock nets.

rm_size_multiple_space sets the minimum spacing for clock nets.

Note

The wire sizes must match what is in the Astro tech file (.tf).

Standard cell lists

The rm_dont_use parameter sets a don't use attribute on the list of cells provided. This is used by DC, PC, and Astro.

The rm_clock_cell_list and rm_delay_cell_list parameters set the names of cells to be used during CTS.

The rm_filler_cells parameter sets the name of the cells to be used as core filler. The largest is listed first.

Astro rail

The rm_supplies_file and rm_activity_file parameters set the voltage of the power supplies and the default switching activity rate. The rm_activity_file parameter is used if a VCD file is not present.

4.3.2 Makefile

A Makefile located in the build_dir controls the flow. All parts of the flow can be initiated from this directory using a make command with a Makefile target. To create multiple parallel build directories, you copy the entire, clean structure to a new area, including the Makefile.

To start a new build, you must first create all the working and data directories using the make dirs command. You must also use the make clean command to clean up a working area that contains a partial or completed build that you no longer require.

Assuming that all required tools are in the Users $path and that all appropriate licences are available, then the <CSBLOCK> can be built:

- In individual stages. This is recommended for the first build run.
- In parts. In this case, use make front for the synthesis parts, and use make analyze for TetraMAX and Formality.
- Whole flow together. In this case, use make build.

4.4 Running synthesis

There is a Makefile provided to control synthesis. To run <CSBLOCK>_compile.tcl use:

```
cd synopsys/build_dir
make compile
```

You can add any additional control and configuration here as required.

Note

To use the Makefile provided you must update it to reflect the correct paths and versions of your EDA tools. See *Makefile* on page 4-12.

4.5 Compile process

This section provides a breakdown of the operations performed within the synthesis steps as defined by the supplied Makefile:

- compile - *Basic compile* describes the synthesis of the design without physical information.
- jxt - *Floorplanning* on page 4-15
- physopt - *Physical synthesis* on page 4-16 using information in *Floorplanning* on page 4-15.

4.5.1 Basic compile

The <csblock>_compile.tcl script performs an initial synthesis of the <CSBLOCK> using simple assumptions about physical placement. *Physical synthesis* on page 4-16 provides a more accurate estimation for physical placement. This section describes the operation steps performed within the supplied TCL compile script.

1. Load component specific variables defined within <csblock>_specific.tcl. This defines:
   - Top level block name, <CSBLOCK>
   - DFT enable port
   - Clock and reset ports
   - High fan-out nets.
2. Setup the configuration defined within <csblock>_config.tcl for:
   - Script options
   - DFT parameters
   - DFT port names.
3. Setup the Target Technology with parameters specific for the intended library and synthesis constraints. For details of <csblock>_tech.tcl, see *Initial setup* on page 4-7.
4. Setup libraries adding DesignWare to the link path.
5. Create the MilkyWay database based using the target technology, removing any cells specified as 'dont_use' and configuring tie-off cells.
6. Setup for verification / equivalence checking.
7. Setup clock gating style.
8. Read in the RTL from <csblock>_verilog.tcl, analyze the design, and perform a design check.

9. Configure Define Design For Test Environment, adding scan ports where appropriate.
10. Define the Clocks in the <CSBLOCK>, including virtual clocks for use with input and output ports.
11. For each clock domain in the <CSBLOCK>, search for registers labeled with `clk_gate` and replace with the clock gating cell.
12. Define the Design Environment by setting up input and output port constraints using `<csblock>_constraints.tcl`. See *Constraining the design* on page 4-18.
13. Identify high fanout nets.
14. Setup the compile options.
15. Isolate the ports for accurate timing model creation before saving the GTECH netlist.
16. Group interface paths separately from other paths.
17. Compile the <CSBLOCK> using Design Compiler.
18. Perform DFT design rule checking on the prescan netlist.
19. Report on the results of the synthesis operation and save the compiled <CSBLOCK> design.

4.5.2 Floorplanning

A basic file is provided with each <CSBLOCK> that details pin and RAM location. The JupiterXT script, `<csblock>_jxt.tcl`, is a set of Astro operations for importing /cts/placement optimizations.

Note

Before you can run the JupiterXT script, `<csblock>_jxt.tcl`, you must run a preparation shell script, `<csblock>_prep.csh`, to generate a subset of the constraints `<csblock>-jxt.sdc` within the *Synopsys Design Constraints* (SDC) file created during `<csblock>_compile.tcl`.

The steps are:

1. Load in <CSBLOCK> specific variables, configuration options, and target technology setup. These are in `<csblock>_specific.tcl`, `<csblock>_config.tcl` and `<csblock>_tech.tcl` respectively.
2. Copy the previously saved cell to working cell.

3. Read the pin placements file, <csblock>_pins.tdf, and auto-generated timing constraints, <csblock>-jxt.sdc.
4. Use floorplan information in the <csblock>_tech.tcl file to define the core area.
5. Set constraints on max layer.
6. Setup the timing parameters.
7. Define core and load macro placement. You can do this using the supplied <csblock>_floorplan.scm file that is supplied blank. You can also do this interactively using the tool (optionally using a virtual flat placement mpc constraints, <csblock>_mpc_constraint.tcl).
8. Add blockage around macros.
9. If a quick floorplan has not been used, then perform a placement, congestion, and pin optimization on the supplied floorplan information.
10. Connect the power and ground ports.
11. If a power floorplan is not provided, then a power network is synthesized and pads placed. Alternatively, power supplies are generated according to <csblock>_VDD_VSS_create_pns_pg_upper.cmd which is supplied blank.
12. Connect standard cell rails.
13. Perform a clean up and final save.

4.5.3 Physical synthesis

You must only perform the physical synthesis flow after Floorplanning as described in *Floorplanning* on page 4-15. The operation is similar to the Basic compile as described in *Basic compile* on page 4-14, however it uses the floorplanned design and implements DFT features.

1. Load in the <CSBLOCK> specific variables, configuration options, and target technology setup. These are in <csblock>_specific.tcl, <csblock>_config.tcl and <csblock>_tech.tcl respectively.
2. Set up the libraries adding DesignWare to the link path.
3. The floorplanned version of the design is then read in along with the same dont_use cells as for the basic synthesis.
4. Create and optimize placement for a netlist out of design compiler (main physopt run).

5. Enable PC to establish move bounds for clock-gating cells.
6. Setup DFT by defining variables, identifying ports, defining scan configuration, enabling adaptive scan if required, and creating test clocks.
7. A DRC is performed before the reorganization of the scan change, and again afterwards along with the creation of test protocol files with another physopt run.
8. Create a buffer tree and perform a final physopt run for each net specified in the high fanout list.
9. Create reports of the physically optimized design, and save the design results.

4.6 Constraining the design

Constraints for the <CSBLOCK> are defined within the <csblock>_constraints.tcl and are based on the performance targets specified within the target technology file.

The constraints file begins by defining a set of cycle times based on percentage expressions from the clock domains present within the design.

Where multiple clock domains are present within the <CSBLOCK>, there might be a series of false paths between asynchronous domains and multicycle paths where appropriate. In general:

- **PCLKDBG** and **ATCLK** are designed to be operated synchronously with **PCLKDBG** at the same speed or slower than **ATCLK**. Because some non-control **PCLKDBG** input signals might be sampled directly by **ATCLK** registers, there are multicycle paths defined based on the **PCLKDBG** to **ATCLK** relationship.
- All other clocks are typically designed to operate asynchronously, having suitable logic where appropriate to handle the clock domain crosses and be defined within the constraints file as false paths.
- Because the scan enable port operates at a lower clock frequency, this is also defined as a multi-cycle path.

For each interface in the different clock domains, the script constrains the input, output, and any static ports based against the respective clocks, as follows:

- Virtual clocks based on the corresponding register clock constrain the appropriate ports.
- DFT ports are based on the fastest clock for that <CSBLOCK>.

4.7 README files

In the synopsys directory, there is a README_<CSBLOCK> file. This file lists all the files involved in the synopsys flow, and any notes relevant to the process, including any known violating paths because of overconstraint of timing requirements.

4.8 False paths between clock domains

A number of <CSBLOCK> components contain multiple clock domains that are asynchronous to each other. Because of the complexity of correctly constraining the paths between the clock domains, they are set as false paths. This is safe to do because the <CSBLOCK> is designed to overcome the problem by means of synchronized control signals that operate in parallel to any data signals.

On some blocks, where false paths have been used, they can also relate to interface ports and typically result in partially constrained paths. You can define timing requirements on those interfaces. This is not done by default because it depends on the clock frequencies and ratios used in your implementation.

See Figure 4-3 for reference. A port is part of an interface based in a single clock domain (A) that has an external delay based on the timing of Clock A. Within <CSBLOCK>, any paths to Clock A registers are correctly constrained. Where any paths go to a clock domain that has a false path, for example Clock B, a section of the path to the Clock B registers is not constrained. The timing path for clock B must not be greater than that for clock A. However, because clock B is asynchronous to the clock used to define the external delay (clock A), it is possible that the external delay might become greater than the clock period of B resulting in a significantly violating timing path.

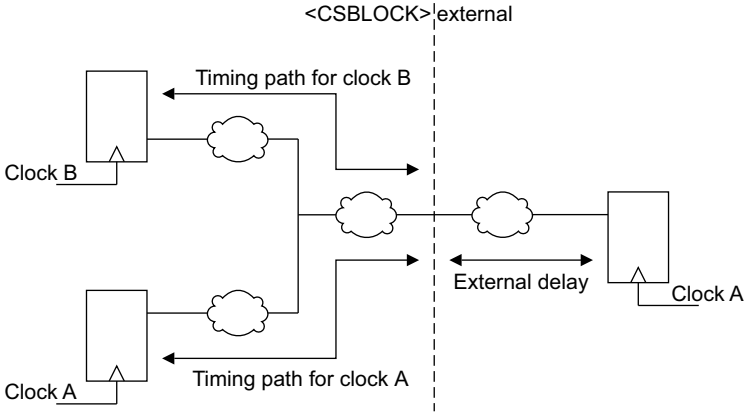

Figure 4-3 <CSBLOCK> timing paths

4.9 Hold time violations

The synthesis scripts provided do not optimize the netlist for hold time violations. Hold time violations are normally fixed during the place and route stage of the synthesis process. However, for timing-annotated netlist simulation it is preferable to have a netlist free of these violations. To do this, add the following command in your script before compiling the design:

```
set_fix_hold {CLK1 CLK2 CLK3}
```

where CLK1, CLK2, and CLK3 are all the clocks in the <CSBLOCK>.

Chapter 5
Functional Verification of Netlist

This chapter describes guidelines for functional verification of the netlist. It contains the following sections:

Note

All CoreSight blocks are verified in the same way. In this chapter, replace `<CSBLOCK>` with the name of the appropriate block. See *CoreSight block summary* on page 1-4 for details on the blocks used in CoreSight.

5.1 About functional verification

This section provides an overview of a flow that you can use to verify your post-synthesis netlists:

- *Verification using equivalence checking tools* on page 5-3
- *Verification using test vectors* on page 5-11.

Note

ARM Limited requires that you use equivalence tools for verifying your netlist. Equivalence checking provides a complete method for verifying your netlist and does not require the extensive simulation compute resources that other methods require.

You must run test vectors in addition to equivalence checking for sign-off, see Chapter 8 *Sign-Off*.

5.2 Verification using equivalence checking tools

This section provides information on how to perform equivalence checking of the <CSBLOCK> RTL and Verilog gate-level netlist. It contains:

- *Equivalence checking overview*
- *Equivalence checking using Formality* on page 5-6
- *Equivalence checking using Conformal* on page 5-7.

5.2.1 Equivalence checking overview

Equivalence checkers use formal mathematical techniques to verify logic functions between two implementations of a design. Using equivalence checkers you can determine if the design functionality is consistent between the two implementations. You must maintain the functionality of the configured <CSBLOCK> at each stage of the design process. You can use equivalence checkers to verify that the functionality of the RTL is maintained, through successive iterations of the Verilog netlist, by a process of:

- *Building the design* on page 5-4
- *Mapping the design* on page 5-4
- *Comparing the design* on page 5-5.

You must also consider design hierarchy, scan chains, and clock gating when equivalence checking the <CSBLOCK>. See *Other considerations* on page 5-5.

Figure 5-1 on page 5-4 shows the supported equivalence checking flow.

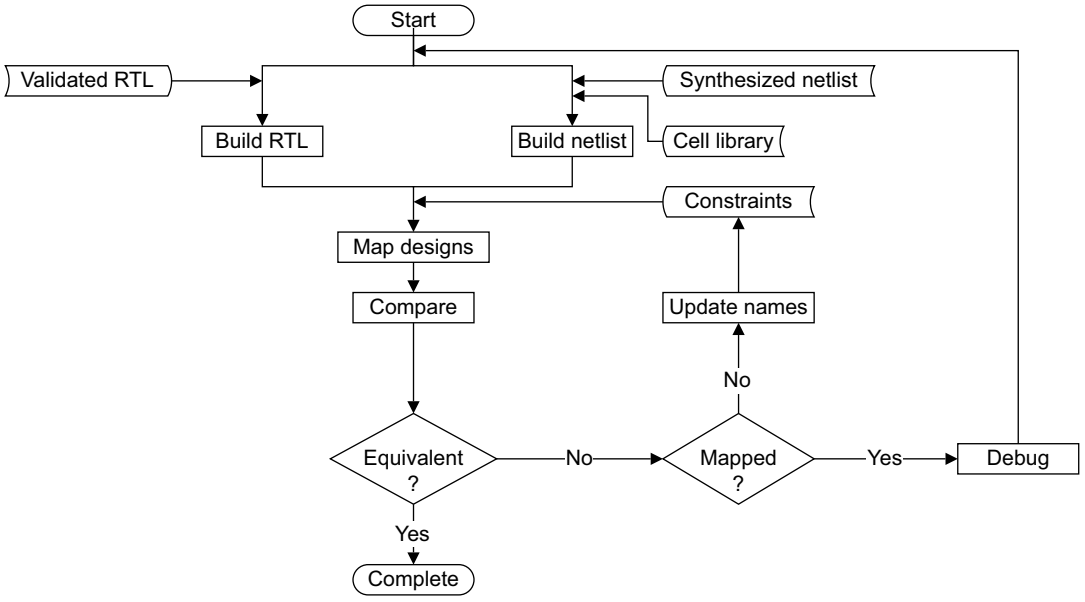

Figure 5-1 Equivalence checking flow

Building the design

The equivalence checker must read in the:

- configured and validated RTL
- synthesized Verilog netlist and cell library.

Mapping the design

When the RTL and netlist have been built the equivalence checker must map the key points in each design. Key points can be:

- primary inputs
- primary outputs
- D flip-flops
- D latches
- black boxes.

Comparing the design

After mapping the design the equivalence checker verifies that the logic driving each compare point is functionally equivalent between the two designs. A compare point can be:

- primary outputs
- D input of D flip-flops or D latches
- black box inputs.

To prove equivalence between the two designs, the equivalence checker drives fan-in logic to each compare point to determine whether they are equivalent or non-equivalent.

Other considerations

When equivalence checking the <CSBLOCK> you must consider:

Hierarchical compare versus flat compare

Hierarchical compare is the process of taking two hierarchical designs and comparing them at the module level from the bottom up. When a low-level design module has been successfully compared, that particular design can be black-boxed. The compare process can then continue with the next level up.

A hierarchical compare requires both designs to have matching hierarchy. This approach can significantly reduce the time required to verify large designs, and it is recommended for RTL to gate-level comparisons.

A flat compare simply performs the compare process at the top-level of the two designs.

You can use flat comparisons when comparing two gate-level designs where the differences between the two design hierarchies are nonfunctional and typically include:

- insertion of scan chains
- insertion of buffer trees for high fanout nets
- re-ordering of the scan chains
- post-layout cell resizing.

The scripts provided do a flat compare because the <CSBLOCK> is not large enough for a hierarchical compare.

Scan chains

When scan chains are inserted into the design new input and output pins can be added, depending on the scan methodology. The additional pins are typically the scan input, output, and enable

pins. Because these additional pins can cause problems with the equivalence checker when mapping the design, you must instruct the equivalence checker to ignore these compare points.

You must also ensure that the scan chains and scan enable signal **SE** are disabled.

Clock gating When clock gating is used in the synthesis flow, you must ensure that the appropriate mode is set in the equivalence checker.

5.2.2 Equivalence checking using Formality

This section provides information on:

- *Setting up the script*
- *Running Formality equivalence checking*
- *Checking results* on page 5-7.

Setting up the script

Use the `<CSBLOCK>_fm.tcl` script, located in the `synopsys/build_dir/scripts` directory, to run the Formality equivalence checking flow. The Synopsys tools read the script on start-up.

Note

You must ensure that the `hdlin_dwroot` variable references your Synopsys site installation and that you read in your technology library. If you are using RAMs (CSETB), you must also read in your RAMs as black boxes.

The component name is set at the top of the script, for example, `set cs_block CSETB`.

The `read_verilog` command reads the RTL into the RTL container:

```
read_verilog -container r -libname WORK $rtl_image
```

The `read_verilog` command again reads the netlist into the GATE container:

```
read_verilog -libname WORK -netlist -c i ./data/${rm_core_top}-physopt.v
```

Running Formality equivalence checking

There is a makefile provided to control equivalence checking using Formality:

- To run `<CSBLOCK>_fm.tcl` use:

```
cd synopsys/build_dir
make fm
```

Note

To use the makefile provided you must update it to reflect the correct paths and versions of your EDA tools.

It is recommended that you invoke Formality using the makefile and the transcript saved for analysis. However, you can run Formality equivalence checking in graphical mode by sourcing the script from within the *Graphical User Interface* (GUI).

Checking results

To check the results:

1. Check the log file `<CSBLOCK>_fm` for errors. If errors exist, you must correct them then rerun the script

   Equivalence checking passes if the log shows `Verification SUCCEEDED` at the CoreSight component level.

2. Various report files appear in the `./reports/lec` directory giving details of why the equivalence failed. Use this information to solve debug problems.

3. Check the results against the golden logs located in the `./logs/arm` directory.

5.2.3 Equivalence checking using Conformal

This section provides information on:

- *Setting up the scripts* on page 5-8
- *Running Conformal equivalence checking* on page 5-9
- *Checking results and troubleshooting* on page 5-9.

Setting up the scripts

Table 5-1 shows the files required for Conformal equivalence checking of the <CSBLOCK>, and their location.

Table 5-1 Conformal equivalence checking files

File	Description	Location
uniquifyallmodules.tcl	This is the Conformal-sourced procedure that performs uniquifying on all the modules in the CoreSight design.	cadence/scripts/tcl
read_designs.tcl	This is the Conformal-sourced procedure that reads in the implementation and reference design.	cadence/scripts/tcl
<CSBLOCK>-rtl-vs-tsmc.do	This is the Conformal.do file that performs equivalence checking. You can configure it for flat or hierarchical comparisons. The default configuration is flat.	cadence/scripts/do
<CSBLOCK>-tsmc.vc	This is the Verilog command file that reads in the technology libraries and Verilog netlist.	cadence/scripts/vc
<CSBLOCK>-rtl.vc	This is the Verilog command file that reads in the Verilog RTL.	cadence/scripts/vc

To set up the Conformal equivalence checking flow, you must set up the following files:

- *<CSBLOCK>-rtl-vs-tsmc.do*
- *<CSBLOCK>-rtl.vc* on page 5-9
- *<CSBLOCK>-tsmc.vc* on page 5-9.

<CSBLOCK>-rtl-vs-tsmc.do

Table 5-2 shows the variables to set in the <CSBLOCK>-rtl-vs-tsmc-FLAT.do script. You must set these variables to match your requirements.

Table 5-2 <CSBLOCK>-rtl-vs-tsmc.do

Variable	Default	Description
ROOT_MODULE	<CSBLOCK>	This variable defines the module to be compared. Do not change.
RAM_ID	SRAM	This variable defines the start of the RAM names, for black boxing. Change this to match the start of your instantiated RAM names.
GOLDEN	rtl	This variable defines the golden design to be RTL. Do not change.

Table 5-2 <CSBLOCK>-rtl-vs-tsmc.do (continued)

Variable	Default	Description
REVISED	tsmc	This variable defines the revised design to be netlist.
R_DES_TYPE	Verilog	This variable defines the language of your revised design. Do not change. Only a Verilog netlist is supported.
G_LIBERTY_LIB	None	Set to library if .lib libraries are used.
R_LIBERTY_LIB	None	Set to library if .lib libraries are used.
PROOF_MODE	FLAT	Set to FLAT for flat comparison, or HIER for hierarchical comparison.

<CSBLOCK>-rtl.vc

Update the <CSBLOCK>-rtl.vc file to read in your Verilog RTL, changing the paths to match those of your memory models.

<CSBLOCK>-tsmc.vc

Update the <CSBLOCK>-tsmc.vc file to read in your Verilog netlist, changing the paths to match those of your memory models.

Running Conformal equivalence checking

The Conformal equivalence checker runs from the ./cadence directory. To run the equivalence checker, type:

```
lec -dofile scripts/do/<CSBLOCK>-rtl-vs-tsmc.do -nogui
```

To monitor the Conformal equivalence checking process, type:

```
tail -f log/<CSBLOCK>-rtl-vs-tsmc.log
```

Checking results and troubleshooting

The results of the Conformal equivalence checking process are located in:

```
cadence/log/<CSBLOCK>-rtl-vs-tsmc-FLAT.log
```

If there are errors in the <CSBLOCK>-rtl-vs-tsmc-FLAT.log, you must fix them before rerunning the tool:

1. Equivalence checking passes if everything is proven equivalent. Check the end of the log file for any reported Non-equivalent points.

2. If there are non-equivalent, aborted, or uncompared modules, you must check that the design maps correctly. If the following message appears in the `<CSBLOCK>-rtl-vs-tsmc-FLAT.log`:

   ```
   Warning: More than 1/3 of the key points have mis-matched names
   ```

   Then this message can indicate that the design does not map correctly.

   Examine the key point names in the RTL and the netlist and, if required, update the naming rules in the `<CSBLOCK>-rtl-vs-tsmc-FLAT.do` file.

3. Even if you get equivalence from Conformal, you must qualify the result if there are any lint warnings. A number of useful lint checks are provided in the scripts. You can check the results by searching the log file for:

   ```
   // Command: report
   ```

   The checks include undriven signals and summary or verbose reports on rule checking.

4. Check the results against the reference log file located in the `cadence/log/arm` directory.

   If you have performed a hierarchical comparison, the number of equivalent modules in the log file might vary, depending on how the comparison was performed. There must be no non-equivalent modules in the log file.

   For example, if you have run a hierarchical comparison, and there are non-equivalent, aborted, or uncompared modules, and boundary optimization is switched on, you might have to apply the `add no black box` command to the modules that fail equivalence checking. See the script for examples of this command.

5.3 Verification using test vectors

The test vectors enable you to perform quick netlist checks in addition to performing formal or functional verification. The vectors were captured using the source RTL as a reference, so replaying the vectors checks that your netlist matches the cycle-by-cycle behavior of the RTL reference. The test vectors do not completely cover the <CSBLOCK> functionality and are therefore not suitable for design sign-off alone. Verification is described in:

- *Test vectors directory structure*
- *Test vector files* on page 5-12.

Note

The BuildNetlist<sim> scripts support back-annotated timing, where <sim> is VCS, NC, or MTI.

5.3.1 Test vectors directory structure

Figure 5-2 shows the test vectors directory structure.

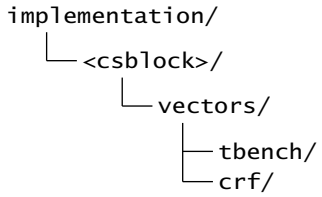

Figure 5-2 Test vectors directory structure

The purpose of each subdirectory is as follows:

`tbench` Contains the testbench and compile scripts for vector replay.

`crf` Contains the test vector files as *Condensed Reference Format* (CRF) files.

The `netlist.vc` file in the `tbench` directory must contain the paths of the Verilog netlist and cell library.

Note

For the `CSETB` block, this file must also contain the RAM model for the corresponding synthesis library.

5.3.2 Test vector files

All test vector files have been compressed using the UNIX compression tool. The build scripts provided automatically decompress these files. A list of vectors is provided and a summary of their functionality is located in `<csblock>_test.list` in the `tbench` directory.

5.4 Building the simulation world and replaying vectors on Netlist

To build the vector replay testbench for Verilog netlist simulation and replay vectors:

1. From your top-level CoreSight component directory go to the vector build and replay area:

   `cd implementation/<csblock>/vectors/tbench`

   where `<csblock>` refers to the appropriate CoreSight block.

2. Ensure that your simulator is on the correct path. The scripts provided for vector replay on a netlist are:

   BuildNetlistMTI

   Builds testbench and replays vectors using the ModelSim tools.

   BuildNetlistVCS

   Builds testbench and replays vectors using the VCS tools.

   BuildNetlistNC

   Builds testbench and replays vectors using the NC tools.

5.5 Analyzing vector simulation logs

A log file is created for each vector that is replayed. This file is located in `tbench/<sim_name>` where `<sim_name>` is:

- `VCS` if the vector was replayed using the VCS tools
- `NC` if the vector was replayed using the NC tools
- `MTI` if the vector was replayed using the ModelSim tools.

The log file is `vector_name_<sim_name>_NET_Replay.log`, where `<sim_name>` is `VCS`, `NC`, or `MTI`.

The reference log files are located in the `tbench/log/arm` directory.

When a test vector replays successfully, a message similar to the following appears at the end of the log file:

```
**Simulation end        16908 vector(s) applied        0 errors found**
```

Chapter 6
Production Test

This chapter describes production testing for the CoreSight components. It contains the following sections:

Note

All CoreSight blocks are tested in the same way. In this chapter, replace <CSBLOCK> with the name of the appropriate block. See *CoreSight block summary* on page 1-4 for details on the blocks used in CoreSight.

6.1 About production test

This section gives an overview of the tests, including features, and how to apply the tests to the <CSBLOCK> when in production. It describes scan insertion and how to manage clock gating when scan enable is asserted.

6.2 DFT

This section describes how to manage:

- clock gating when scan enable is asserted
- system clocks.

6.2.1 Clock gating

To ensure that the scan chains operate correctly, without affecting the fault coverage, you must ensure that the clock gating cells are forced on when scan enable is asserted. This applies to:

- clock gates that are inferred in the RTL and inserted by Power Compiler
- architectural clock gates.

6.3 Scan chain insertion

Table 6-1 shows the scan test ports.

Table 6-1 Scan test ports

Name	Direction	Description
SE	Input	Enable scan chains. High fanout of this signal means this must be a false path, and cannot be switched at-speed. This signal must be tied LOW during functional mode.
SIn[a]	Input	Inputs to scan chains. Each input must be identified individually as a bit instead of in a bus to prevent routing problems in downstream tools.[b]
SOn[a]	Output	Outputs from scan chains. Each input must be identified individually as a bit instead of in a bus to prevent routing problems in downstream tools.[b]

a. n represents the number of scan chains set.
b. The scan I/Os are created during scan insertion.

6.4 Fault models

The *Automatic Test Pattern Generation* (ATPG) script produces ATPG vectors for the following fault models:

- stuck at
- transition delay
- IDDQ.

The ATPG script writes vectors into separate files for each of these fault models.

6.5 Automatic Test Pattern Generation

This section describes how TetraMAX uses the <CSBLOCK>_tmax.tcl script and how you can control it for ATPG for the macrocell.

Figure 6-1 shows the supported flow for ATPG of the macrocell for production test.

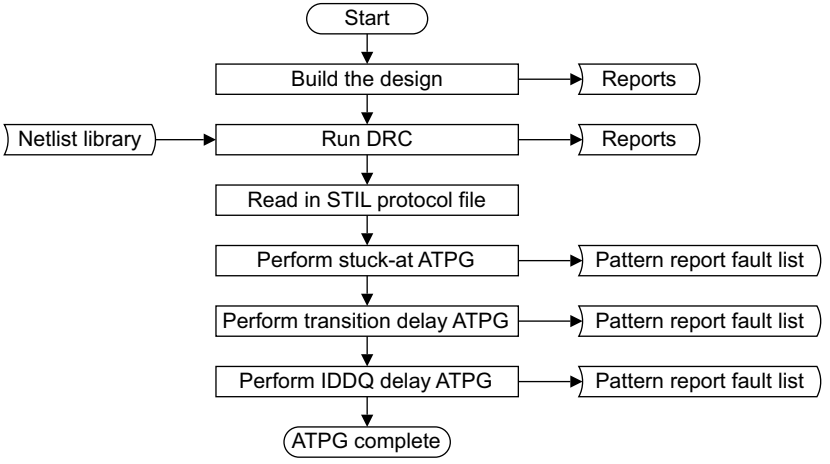

Figure 6-1 Macrocell ATPG supported flow

6.5.1 Building the design

To build the design, the script:

1. Sets up a log file and turns on verbose messages.

   ```
   set_messages -replace log ../logs/tmax.logset messages -level expert
   ```

   If any errors are encountered the script exits from ATPG. You might want to remove this line if you want to run TetraMAX interactively.

   ```
   set_commands exit
   ```

2. Reads in the cell library and TetraMAX RAM models. You must modify these commands to read your cell library and your RAM models:

   ```
   read_netlist -f verilog ${rm_tmax_verilog_lib}
   ```

Note

You must have TetraMAX-compatible RAM models to build the design, therefore you must either:

- use a RAM compiler that generates TetraMAX RAM models
- use the TetraMAX RAM models supplied by your RAM library vendor
- create your own TetraMAX models, see the *TetraMAX ATPG User Guide*.

3. Reads in the Verilog netlist.

```
read_netlist -f verilog ../data/${rm_core_tops}_physopt.v
```

4. Reports any errors encountered in reading in the design:

```
report_modules -summaryreport modules -error
```

5. Prevents unconnected logic from being removed and so avoids giving optimistic results:

```
set_build -nodelete_unused_gates
```

6. Builds the ATPG model:

```
run_build ${rm_core_top}
```

6.5.2 Run DRC

To run DRC the script:

1. Turns on dynamic clock grouping to reduce the pattern count as much as possible:

```
set_drc -clock -dynamic -serial_clock_grouping -disturb_clock_grouping
```

2. Runs DRC:

```
run_drc ${protocol_file}
```

This reads in the STIL protocol file that describes:

- the scan chains in the design
- any pin constraints
- the load and unload procedures for the scan chains
- the capture procedures
- timing that your ATE must use.

During synthesis the DFT Compiler writes out a STIL protocol file. For more details see *Basic compile* on page 4-14. You must ensure the contents of this file meet the requirements of your ATE. You might want to modify the file created by DFT Compiler, or write your own.

To check that the scan chains operate correctly DRC applies the procedures that the STIL file describes.

3. Writes any rule violations that it finds to a report file in the `/reports/dft` directory.

   If a report file is created you must check it and understand the violation codes that occur. Table 6-2 shows the expected violation codes that occur.

Table 6-2 Violation codes

Violation code	Description
`B7`, `B8`, `B9`, and `B10`	These mean that there are unconnected internal signals in the design. This is because of the optimizations made during synthesis.

4. After step 3, the script produces:
   - *Stuck-at ATPG patterns*
   - *Transition delay ATPG patterns* on page 6-10
   - *IDDQ ATPG patterns* on page 6-10.

6.5.3 Stuck-at ATPG patterns

To produce stuck-at ATPG patterns the script:

1. Ensures TetraMAX reports fault coverage in addition to test coverage:
   ```
   set_faults -fault_coverage
   ```
2. Adds all faults to the design:
   ```
   add_faults -all
   ```
3. Sets the flops in your cell library to output an x if the clock input is x
   ```
   set_simulation -XClock_gives_xout
   ```
   Note

   You must remove this line if your cell library does not have this behavior.
4. Runs ATPG. Untestable faults are identified to prevent TetraMAX from continuing to consider them:
   ```
   set_atpg -analyze_untestable
   set_atpg -abort 10 -merge high -resim_basic_scan_patterns auto
   run_atpg -auto
   set_atpg abort 100
   ```
5. Produces basic scan patterns for faults that are not complex.
   ```
   set_atpg -analyze_untestable
   ```

```
set_atpg -abort 10 -merge high -resim_basic_scan_patterns auto
run_atpg -auto_compression
set_atpg -abort 100
run_atpg -auto_compression
```

6. Produces uncompressed fast sequential patterns in several stages. with increasing capture depth. For example:

```
set_atpg -capture 2 -fastseq_merge
run_atpg -auto_compression
set_atpg -capture 4
run_atpg -auto_compression
set_atpg -capture 6
run_atpg -auto_compression
set_atpg -capture 8
run_atpg -auto_compression
```

7. Compresses the patterns from step 6 so that step 8 can produce full sequential patterns:

```
run_pattern_compression 10 -min 0 -max_useless 2 -reset_au_faults -verbose
```

8. Produces full sequential patterns:

```
set_atpg -full_seq_merge high -full_seq_atpg -full_seq_time 10
run_atpg -auto_compression full_sequential_only
```

9. Writes several reports to the `reports` directory using report fault commands. These give the status of the faults in the design that follow ATPG. You can use these reports to debug low fault coverage if required.

   Reports are also created giving details of the patterns produced.
   See `./reports/dft`

10. Writes out the patterns in WGL format and writes out patterns for both parallel and serial replay. You must change the format of the patterns to suit your requirements:

```
write_patterns ../data/${rm_core_top}-sa_parallel.bin.gz -replace -format binary      -compress gzip
write_patterns ../data/${rm_core_top}-sa_parallel.wgl.gz -replace -format wgl         -compress gzip
write_patterns ../data/${rm_core_top}-sa_serial.wgl.gz   -replace -format wgl -serial -compress gzip
write_patterns ../data/${rm_core_top}-sa_parallel.v.gz   -replace -format verilog_single_file
-compress gzip
write_patterns ../data/${rm_core_top}-sa_serial.v.gz     -replace -format verilog_single_file -serial
-compress gzip
```

11. Writes out fault data so it can be reloaded into TetraMAX if later required and removes all faults before moving on to the next fault model:

```
write_fault ../data/${rm_core_top}-sa_all.dat -all -replace
```

6.5.4 Transition delay ATPG patterns

The process for generating transition fault patterns is very similar to the process for generating stuck-at patterns. The fault model is set to transition faults, and the transition faults are added to the design:

```
set_faults -model transition -fault_coverage
set_delay -launch_cycle system_clock -two_clock
add_faults -all
```

Only fast sequential patterns are generated for transition delay faults.

```
set_atpg -capture 2
set_atpg -abort 10
set_atpg -merge high
set_atpg -fastseq_merge
set_atpg -nofull_seq_atpg
set_atpg -capture 4
run_atpg -auto_compression
set_atpg -capture 6
run_atpg -auto_compression
set_atpg -capture 8
run_atpg -auto_compression
```

Reports, patterns, and faults are then written in the same way as for stuck-at fault ATPG.

6.5.5 IDDQ ATPG patterns

The fault model is set to IDDQ faults, and the IDDQ faults are added to the design.

```
set_faults -model iddq
set_iddq float strong weak write -atpg -notoggle
add_faults -all
```

The required number of IDDQ patterns is set, before they are produced.

```
set_atpg -merge highset atpg -analyze_untestableset atpg -patterns 12 run atpg -auto_compression
```

Reports, patterns, and faults are then written in the same way as for stuck-at fault ATPG.

6.5.6 Run ATPG

There is a makefile provided to control ATPG:

- To run <CSBLOCK>_tmax.tcl use:

  ```
  cd synopsys/build_dir
  make tmax
  ```

Note

To use the makefile provided you must update it to reflect the correct paths and versions of your EDA tools.

Chapter 7
Example CoreSight Subsystem

This chapter describes the example CoreSight DK subsystem. It contains the following sections:

7.1 About the example CoreSight DK subsystem

ARM Limited provides a fully integrated and tested CoreSight DK subsystem as an example of how to connect the CoreSight components together. The Verilog source code for this is provided in:

`logical/cssys/verilog/CSSYS.v`

Figure 7-1 shows the components included in the example subsystem.

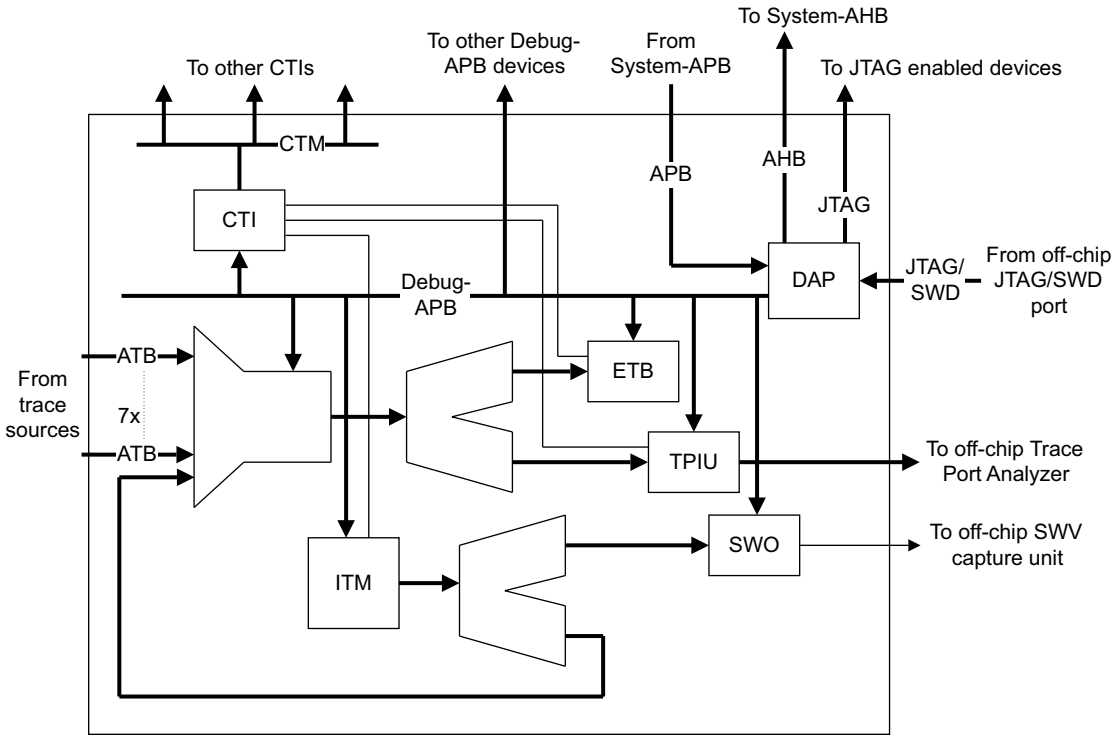

Figure 7-1 Example CoreSight subsystem

This subsystem is intended as a reference design for a typical system with multiple trace sources. It is recommended that system designers reuse and modify it, if necessary, when creating a SoC.

7.2 Example CoreSight DK subsystem implementation flow

This section describes any additional information for the implementation flow that has been developed for the example CSDK subsystem. The implementation flow is the same as that described in Chapter 4 *Synthesis* with <CSBLOCK> being that of CSSYS. Other aspects of the implementation flow are also identical to those described for CoreSight components elsewhere in this guide. CoreSight integrators are not expected to use this flow, but instead to flatten CoreSight components in place with additional SoC logic.

7.2.1 Synthesis scripts

The same set of scripts are required for the example CoreSight DK subsystem synthesis flow, with additional notes below.

CSSYS_specific.tcl

The cssys_specific.tcl contains variables which you must set to reflect the configuration of the CSSYS. Table 7-1 shows the cssys_specific.tcl variables.

Table 7-1 cssys_specific.tcl variables

Option	Default	Function
set rm_cssys_bridge	1	When set at least one synchronous bridge is instantiated
set rm_cssys_etb	1	When set CSETB block is instantiated
set rm_cssys_tpiu	1	When set CSTPIU block is instantiated
set rm_cssys_itm	1	When set CSITM or CSSWV block is instantiated
set rm_cssys_swo	1	When set CSSWV block is instantiated

CSSYS_constraints.tcl

The cssys_constraints.tcl script provides default I/O constraints.

To enable synthesis where the frequency of **PCLKSYS** and **HCLK** is different to that of **PCLKDBG**, some I/Os are constrained to **PCLKDBG** even though they might appear to be **PCLKSYS** or **HCLK** domain signals. This is because they are clocked or captured by **PCLKDBG**. To provide for this capability, these signals have been grouped as follows:

PCLKDBGFORSYS_INPUTS

PCLKSSYS domain inputs that **PCLKDBG** captures.

PCLKDBGFORSYS_OUTPUTS

PCLKSSYS domain outputs that **PCLKDBG** clocks.

PCLKDBGFORHCLK_OUTPUTS

HCLK domain outputs that **PCLKDBG** clocks.

7.2.2 Configuring the CSSYS Verilog files for synthesis

Before running synthesis, you must edit the `CSSysDefs.v` file in `logical/CSSYS/verilog` to suit your system requirements. You might have to include a synchronous bridge on a trace source if its output is not registered and/or you cannot otherwise satisfy the input timing requirements of the Trace Funnel.

To include a particular synchronous bridge, uncomment the relevant line in `CSSysDefs.v`.

For example, to include a synchronous bridge on trace source 0, uncomment the following line:

```
`define CSSYS_BRIDGE_PORT0
```

You must also include either an ETB or TPIU, or both, in your system. To do this, uncomment one or both of the following lines in `CSSysDefs.v`:

```
`define CSETB
`define CSTPIU
```

Also provided in `CSSys` are the ITM and SWO blocks. To include SWV, uncomment both of the following lines. If only ITM is required, uncomment the first line only.

```
`define CSITM
`define CSSWO
```

Note

SWO can only be used in conjunction with ITM.

To repeat the synthesis results for the standalone example CoreSight DK subsystem, you must include all eight synchronous bridges, one for each trace source, the ETB, the TPIU, the ITM, and the SWO by uncommenting the applicable lines in `CSSysDefs.v`.

Chapter 8
Sign-Off

In addition to your normal ASIC flow sign-off checks, you must satisfy certain verification criteria before you sign off the CoreSight design. This chapter describes the sign-off criteria. It contains the following section:

- *CoreSight component sign-off* on page 8-2.

Note

All CoreSight blocks are signed-off in the same way. In this chapter, replace <CSBLOCK> with the name of the appropriate block. See *CoreSight block summary* on page 1-4 for details on the blocks used in CoreSight.

Note

For details of CoreSight system-level sign-off, see the applicable CoreSight Design Kit Integration Manual.

8.1 CoreSight component sign-off

To sign off the <CSBLOCK> you must meet the criteria in each of the following stages in the design flow:

1. *RTL verification*
2. *Post-synthesis*
3. *Post place-and-route timing*.

8.1.1 RTL verification

You must verify the <CSBLOCK> RTL deliverables before you begin the synthesis stage. See the applicable README_<CSBLOCK> file for details.

8.1.2 RTL configuration

To ensure consistent understanding in tools support, it is advised that all configuration parameters are correctly set, for example ROM Table Designer and Part Number fields. For more information, see *RTL configuration parameters* on page 1-9.

8.1.3 Post-synthesis

You must verify the functionality of the final post-synthesis netlist before you sign off the <CSBLOCK> design. This verification consists of both:

1. Proving logical equivalence between the validated <CSBLOCK> RTL and the final placed-and-routed netlist using formal verification tools. See *Verification using equivalence checking tools* on page 5-3.
2. Running the supplied <CSBLOCK> vectors on the final place-and-routed netlist.

8.1.4 Post place-and-route timing

You must verify the timing of the final placed-and-routed netlist before you sign off the netlist using *Static Timing Analysis* (STA). It is also recommended that you run some or all of the supplied <CSBLOCK> vectors on a netlist with back-annotated timing as a final check.

Glossary

This glossary describes some of the terms used in this manual. Where terms can have several meanings, the meaning presented here is intended.

Advanced High-performance Bus (AHB)
The AMBA Advanced High-performance Bus system connects embedded processors such as an ARM core to high-performance peripherals, DMA controllers, on-chip memory, and interfaces. It is a high-speed, high-bandwidth bus that supports multi-master bus management to maximize system performance.

See also Advanced Microcontroller Bus Architecture and AHB-Lite.

Advanced Microcontroller Bus Architecture (AMBA)
AMBA is the ARM open standard for multi-master on-chip buses, capable of running with multiple masters and slaves. It is an on-chip bus specification that details a strategy for the interconnection and management of functional blocks that make up a *System-on-Chip* (SoC). It aids in the development of embedded processors with one or more CPUs or signal processors and multiple peripherals. AMBA complements a reusable design methodology by defining a common backbone for SoC modules. AHB conforms to this standard.

See also Advanced High-performance Bus and AHB-Lite.

AHB *See* Advanced High-performance Bus.

AHB-Lite AHB-Lite is a subset of the full AHB specification. It is intended for use in designs where only a single AHB master is used. This can be a simple single AHB master system or a multi-layer AHB system where there is only one AHB master on a layer.

AMBA *See* Advanced Microcontroller Bus Architecture.

Architecture The organization of hardware and/or software that characterizes a processor and its attached components, and enables devices with similar characteristics to be grouped together when describing their behavior, for example, Harvard architecture, instruction set architecture, ARMv6 architecture.

Central Processing Unit (CPU)
The part of a processor that contains the ALU, the registers, and the instruction decode logic and control circuitry. Also commonly known as the processor core.

Clock gating Gating a clock signal for a macrocell with a control signal, and using the modified clock that results to control the operating state of the macrocell.

Coprocessor A processor that supplements the main CPU. It carries out additional functions that the main CPU cannot perform. Usually used for floating-point math calculations, signal processing, or memory management.

Core reset *See* Warm reset.

CPU *See* Central Processing Unit.

DBGTAP *See* Debug Test Access Port.

Debug Test Access Port (DBGTAP)
The collection of four mandatory terminals and one optional terminal that form the input/output and control interface to a JTAG boundary-scan architecture. The mandatory terminals are **DBGTDI** (**TDI**), **DBGTDO** (**TDO**), **DBGTMS** (**TMS**), and **TCK**. The optional terminal is **DBGnTRST** (**TRST**).

EmbeddedICE logic An on-chip logic block that provides TAP-based debug support for ARM processor cores. It is accessed through the TAP controller on the ARM core using the JTAG interface.

EmbeddedICE-RT The JTAG-based hardware provided by debuggable ARM processors to aid debugging in real-time.

Exception An event that occurs during program operation that makes continued normal operation inadvisable or impossible, and so makes it necessary to change the flow of control in a program. Exceptions can be caused by error conditions in hardware or software. The processor can respond to exceptions by running appropriate exception handler code that attempts to remedy the error condition, and either restarts normal execution or ends the program in a controlled way.

Implementation- defined
A feature that is not architecturally defined, and which might vary between implementations. The feature is defined and documented for each individual implementation.

Load/store architecture
A processor architecture where data-processing operations only operate on register contents, not directly on memory contents.

Macrocell A complex logic block with a defined interface and behavior. A typical VLSI system comprises several macrocells (such as an ARM processor, an Embedded Trace Macrocell, and a memory block) plus application-specific logic.

Memory Management Unit (MMU)
Hardware that controls caches and access permissions to blocks of memory, and translates virtual addresses to physical addresses.

Memory Protection Unit (MPU)
Hardware that controls access permissions to blocks of memory. Unlike an MMU, an MPU does not translate virtual addresses to physical addresses.

MMU *See* Memory Management Unit.

Processor A contraction of microprocessor. A processor includes the CPU or core, plus additional components such as memory, and interfaces. These are combined as a single macrocell, that can be fabricated on an integrated circuit.

Read Reads are defined as memory operations that have the semantics of a load. That is, the ARM instructions LDM, LDRD, LDC, LDR, LDRT, LDRSH, LDRH, LDRSB, LDRB, LDRBT, LDREX, RFE, STREX, SWP, and SWPB, and the Thumb instructions LDM, LDR, LDRSH, LDRH, LDRSB, LDRB, and POP. Java instructions that are accelerated by hardware can cause a number of reads to occur, according to the state of the Java stack and the implementation of the Java hardware acceleration.

Register A temporary storage location used to hold binary data until it is ready to be used.

Scan chain A scan chain is made up of serially-connected devices that implement boundary scan technology using a standard JTAG TAP interface. Each device contains at least one TAP controller containing shift registers that form the chain connected between **TDI** and **TDO**, through which test data is shifted. Processors can contain several shift registers to enable you to access selected parts of the device.

TAP *See* Test Access Port.

Test Access Port (TAP)
The collection of four mandatory terminals and one optional terminal that form the input/output and control interface to a JTAG boundary-scan architecture. The mandatory terminals are **TDI**, **TDO**, **TMS**, and **TCK**. The optional terminal is **TRST**.

Undefined Indicates an instruction that generates an Undefined instruction trap. See the *ARM Architectural Reference Manual* for more information on ARM exceptions.

Warm reset Also known as a core reset. Initializes the majority of the processor excluding the debug controller and debug logic. This type of reset is useful if you are using the debugging features of a processor.