ARM® Cortex®-A57 MPCore™ Processor

Revision: r0p1

Configuration and Sign-off Guide

Confidential

ARM DII 0279B (ID100113)

ARM Cortex-A57 MPCore Processor
Configuration and Sign-off Guide

Release Information

The following changes have been made to this book.

Change history

Date	Issue	Confidentiality	Change
04 June 2013	A	Confidential	First release for r0p0
01 October 2013	B	Confidential	First release for r0p1

Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM in the EU and other countries, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Where the term ARM is used it means “ARM or any of its subsidiaries as appropriate”.

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

ARM Cortex-A57 MPCore Processor Configuration and Sign-off Guide

Preface

This preface introduces the *ARM® Cortex®-A57 MPCore™ Processor Configuration and Sign-off Guide*. It contains the following sections:

- *About this book* on page vi.
- *Feedback* on page x.

About this book

This book is for the Cortex-A57 MPCore multiprocessor.

Note

This guide is complemented by the supplied reference methodology documentation. You must read the CSG because it describes how to configure your device and the steps you must take to sign-off your design.

Implementation obligations

This book is designed to help you implement an ARM product. The extent to which the deliverables can be modified or disclosed is governed by the contract between ARM and the Licensee. There might be validation requirements which, if applicable, are detailed in the contract between ARM and the Licensee and which, if present, must be complied with prior to the distribution of any devices incorporating the technology described in this document. Reproduction of this document is only permitted in accordance with the licenses granted to the Licensee.

ARM assumes no liability for your overall system design and performance. Verification procedures defined by ARM are only intended to verify the correct implementation of the technology licensed by ARM, and are not intended to test the functionality or performance of the overall system. You or the Licensee are responsible for performing system level tests.

You are responsible for applications that are used in conjunction with the ARM technology described in this document, and to minimize risks, adequate design and operating safeguards must be provided for by you. Publishing information by ARM in this book of information regarding third party products or services is not an express or implied approval or endorsement of the use thereof.

Product revision status

The r*n*p*n* identifier indicates the revision status of the product described in this book, 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 book is written for hardware engineers who have experience writing Verilog and performing synthesis, and who plan to implement a Cortex-A57 MPCore multiprocessor design. It is not required that engineers have experience with ARM products.

Using this book

This book is organized into the following chapters:

Chapter 1 *Introduction*

Read this for a description of the multiprocessor design platforms and tools, including the supported design flow, directory structure, and design hierarchy.

Chapter 2 *Configuration Guidelines*

Read this for a description of the configuration options available for your multiprocessor, how to configure them, and how to check the results.

Chapter 3 *Memory Integration*

Read this for a description of memory integration, associated scripts, and how to check the results.

Chapter 4 *Checking your RTL*

Read this for a description of how to check the RTL for your multiprocessor.

Chapter 5 *Floorplan Guidelines*

Read this for a general description of the layout recommendations for floorplanning the multiprocessor.

Chapter 6 *Design For Test*

Read this for a description of the Design for Test features for the multiprocessor.

Chapter 7 *Dynamic Verification*

Read this for a description of the dynamic verification requirements for the multiprocessor.

Chapter 8 *Power Intent Specification* Read this for a description of the Power Intent Unified Power Format (UPF) files and the Power Domain Clamp values.

Chapter 9 *Sign-off*

Read this for a description of the ARM verification criteria, and how to sign-off your design.

Appendix A *Tarmac Trace Description*

Read this for a description of the text file that contains information about executed instructions, register file updates, memory accesses, and exception events.

Appendix B *Revisions*

Read this for a list of the technical changes between released issues of this book.

Glossary

The *ARM® Glossary* is a list of terms used in ARM documentation, together with definitions for those terms. The *ARM® Glossary* does not contain terms that are industry standard unless the ARM meaning differs from the generally accepted meaning.

See *ARM® Glossary*, http://infocenter.arm.com/help/topic/com.arm.doc.aeg0014-/index.html.

Conventions

This book uses the conventions that are described in:

- *Typographical conventions* on page viii.
- *Timing diagrams* on page viii.
- *Signals* on page viii.

Typographical conventions

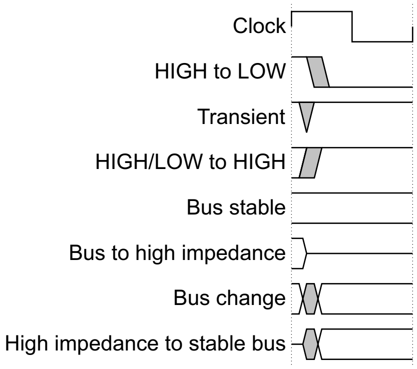
Typographical conventions

italic	Introduces special terminology, denotes 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.
monospace	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>	Enclose replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>`
SMALL CAPITALS	Used in body text for a few terms that have specific technical meanings, that are defined in the *ARM® Glossary*. For example, IMPLEMENTATION DEFINED, UNDEFINED, and UNKNOWN.

Timing diagrams

The figure named *Key to timing diagram conventions* explains the components used in timing diagrams. Variations, when they occur, have clear labels. You must not assume any timing information that is not explicit in the diagrams.

Shaded bus and signal areas are undefined, so the bus or signal can assume any value within the shaded area at that time. The actual level is unimportant and does not affect normal operation.



Key to timing diagram conventions

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.
- LOW for active-LOW signals.

Lower-case n At the start or end of a signal name denotes an active-LOW signal.

Additional reading

This section lists publications by ARM and by third parties.

See Infocenter, http://infocenter.arm.com, for access to ARM documentation.

ARM publications

This book contains information that is specific to this product. See the following documents for other relevant information:

- *ARM® Architecture Reference Manual ARMv8, for ARMv8-A architecture profile* (ARM DDI 0487).

The following confidential books are only available to licensees:

- *ARM® CoreSight™ Architecture Specification* (ARM IHI 0029).
- *ARM® Cortex®-A57 MPCore™ Processor Technical Reference Manual* (ARM DDI 0488).
- *ARM® Cortex®-A57 MPCore™ Processor Cryptography Extension Technical Reference Manual* (ARM DDI 0514).
- *ARM® Cortex®-A57 MPCore™ Processor Integration Manual* (ARM DII 0280).
- *ARM® Cortex®-A57 MPCore™ Processor Release Note*.
- *ARM® Tarmac Specification* (ARM-EPM-041435).

Other publications

This section lists relevant documents published by third parties:

- *IEEE Standard for Design and Verification of Low Power Integrated Circuits, IEEE Standard 1801-2009*.

Feedback

ARM welcomes feedback on this product and its documentation.

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to `errata@arm.com`. Give:

- The title, ARM Cortex-A57 MPCore Processor Configuration and Sign-off Guide.
- The number, ARM DII 0279B.
- The page numbers to which your comments apply.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Note

ARM tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Chapter 1
Introduction

This chapter introduces the implementation, configuration, and sign-off tasks for the Cortex-A57 MPCore multiprocessor. It contains the following sections:

1.1 About implementation

Figure 1-1 shows the implementation process, including the top-level inputs, resources, outputs, controls and constraints for implementation.

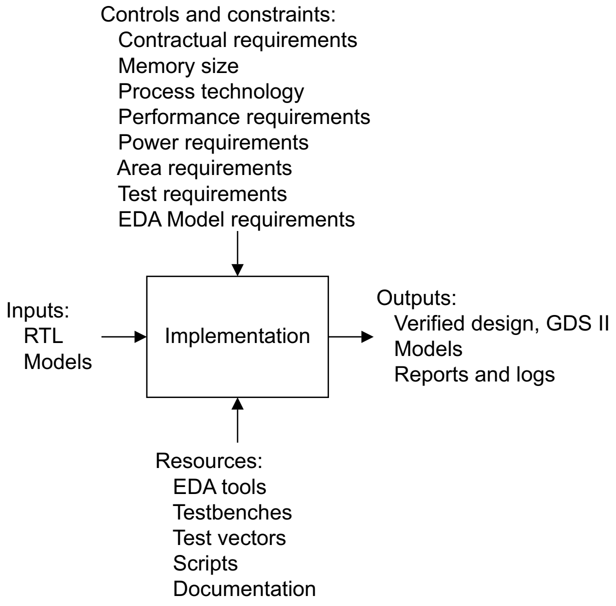


Figure 1-1 Implementation process

1.2 Implementation resources

This guide assumes that you have suitable EDA tools and compute resources for implementation. See the *ARM® Cortex®-A57 MPCore™ Processor Release Note* for:

- The list of deliverables.
- The specific tool revisions required.

Table 1-1 shows the tools used to develop the multiprocessor.

Table 1-1 Development tools

Purpose	Vendor	Tool
HDL simulator	Cadence	*Incisive Enterprise Simulator* (IES)
	Mentor Graphics	QuestaSim
	Synopsys	VCS
RTL checking	Accellera	*Universal Verification Methodology* (UVM)
	Google	Protocol Buffers
Software development	ARM	*RealView® Core Tools* (RVCT) armasm, armlink
	GNU Project	*GNU Compiler Collection* (GCC)

Note

- The *ARM® Cortex®-A57 MPCore™ Processor Release Note* describes any special requirements that might affect the flow, such as information about any special tool requirements that enable optional flows within the implementation.
- See the supplied reference methodology documentation for details of other tools used. For EDA tool support, contact your EDA vendor.

1.3 Implementation controls and constraints

Figure 1-1 on page 1-2 shows the general controls and constraints that apply to the implementation. You must implement the device in accordance with your contract, see *Implementation obligations* on page vi.

1.4 Implementation inputs

The *ARM® Cortex®-A57 MPCore™ Processor Release Note* describes deliverables that are inputs to the implementation flow. These deliverables include:

- *Register Transfer Language* (RTL) description of the multiprocessor.
- RAM integration testbench.
- Execution testbench.
- Test vectors.
- Documentation.

Additionally, the following are required for implementation but are not deliverables:

- EDA tools.
- Standard cell libraries.
- Memory or custom cell libraries.

1.5 Implementation flow

This section lists the main points you must consider when you implement the macrocell. You must read this section in conjunction with the rest of this guide and the information provided in the documents listed in *Additional reading* on page ix.

Figure 1-2 shows the implementation flow for the Cortex-A57 MPCore multiprocessor.

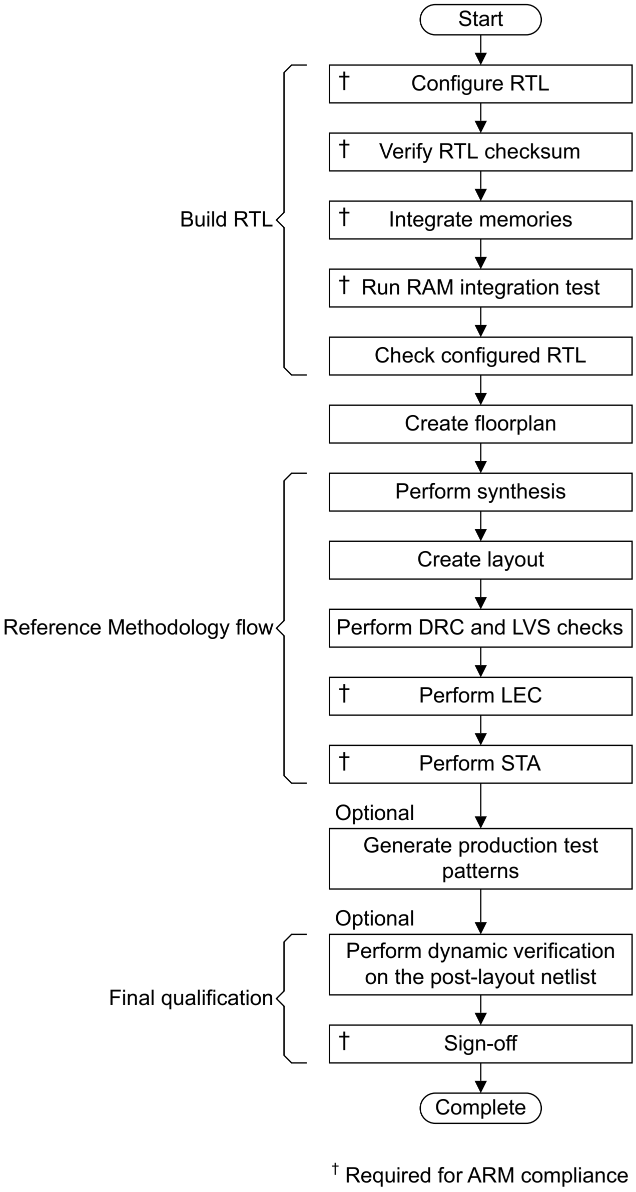


Figure 1-2 Implementation flow

Table 1-2 lists the key tasks for implementation.

Table 1-2 Implementation tasks

Implementation task	Description
1. Configure RTL	How to configure the supplied RTL with the configuration script. See Chapter 2 *Configuration Guidelines*.
2. Perform RTL checksum verification	How to verify the configuration with the checking script. See Chapter 2 *Configuration Guidelines*.
3. Integrate memories	How to integrate memory using the supported memory integration flow. See Chapter 3 *Memory Integration*.
4. Run RAM integration testbench	How to verify the memory integration using the RAM integration testbench. See Chapter 3 *Memory Integration*.
5. Check configured RTL	How to check your configured RTL with the supplied test vectors using the execution testbench. See Chapter 4 *Checking your RTL*.
6. Create a floorplan	How to create a floorplan for implementation of the Cortex-A57 MPCore multiprocessor. See Chapter 5 *Floorplan Guidelines*.
7. Perform synthesis	For information about these tasks, see the supplied reference methodology documentation. For EDA tool support, contact your EDA vendor.
8. Create layout	
9. Perform *Design Rule Checking* (DRC) and *Layout Versus Schematic* (LVS) checks	
10. Perform *Logical Equivalence Checking* (LEC)	
11. Perform *Static Timing Analysis* (STA)	
12. Perform dynamic verification	How to verify post-layout netlist with supplied test vectors. See Chapter 7 *Dynamic Verification*.
13. Sign-off	How to satisfy required sign-off criteria. See Chapter 9 *Sign-off*. **Note** Your contract requires you to complete sign-off as part of the completed flow.

You must complete all the tasks in Table 1-2, to produce complete and verified deliverables. See *Obligations for sign-off* on page 9-3.

1.6 Implementation outputs

The outputs from the implementation flow are:

- Logs and reports, including:
  - Verifying configured RTL by running configuration checking script.
  - RAM integration testbench after integrating memories.
  - Running the supplied tests on the execution testbench.
  - Synthesizing the configured RTL.
  - Performing *Design Rule Checking* (DRC) and *Layout Versus Schematic* (LVS) checking.
  - *Logical Equivalence Checking* (LEC) between configured RTL and post-layout netlist.
  - Performing *Static Timing Analysis* (STA) on post-layout netlist.
  - Running test vectors on post-layout netlist.
- Components
  - Post-layout netlist.
  - GDSII.
  - *Standard Delay Format* (SDF) output.
- Test vectors:
  - ATPG vectors.

1.7 Reference data for implementation

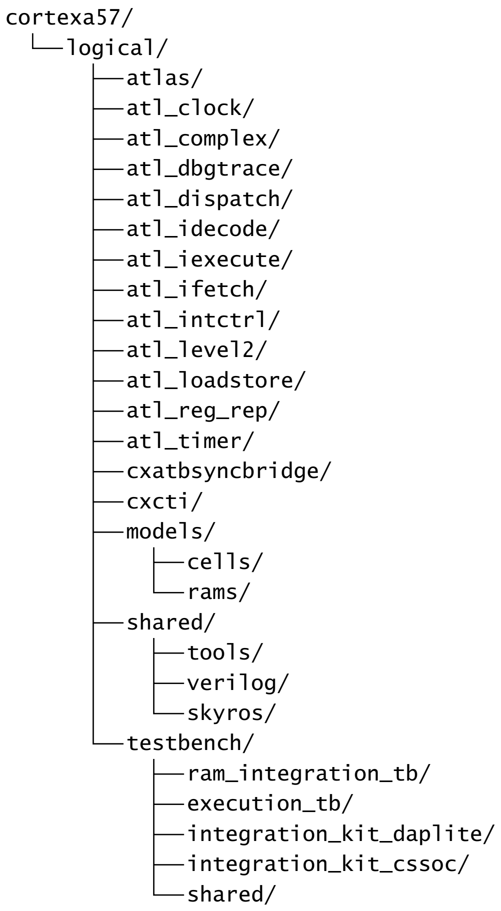
This section provides reference data for your implementation.

1.7.1 Release directory structure

Figure 1-3 shows the principal directory structure of the Cortex-A57 MPCore multiprocessor as it appears when you have unpacked the deliverables.

Note

The root_directory is of the form `MP019-BU-00000-r0p0-00rel0`. This is the bundle number of the deliverables and is used as an example in this document. Your bundle number can be different.



```
cortexa57/
  └─logical/
       ├─atlas/
       ├─atl_clock/
       ├─atl_complex/
       ├─atl_dbgtrace/
       ├─atl_dispatch/
       ├─atl_idecode/
       ├─atl_iexecute/
       ├─atl_ifetch/
       ├─atl_intctrl/
       ├─atl_level2/
       ├─atl_loadstore/
       ├─atl_reg_rep/
       ├─atl_timer/
       ├─cxatbsyncbridge/
       ├─cxcti/
       ├─models/
       │    ├─cells/
       │    └─rams/
       ├─shared/
       │    ├─tools/
       │    ├─verilog/
       │    └─skyros/
       └─testbench/
            ├─ram_integration_tb/
            ├─execution_tb/
            ├─integration_kit_daplite/
            ├─integration_kit_cssoc/
            └─shared/
```

Figure 1-3 Release directory structure

Chapter 2
Configuration Guidelines

This chapter describes the multiprocessor configuration script and the associated checking script. It contains the following sections:

2.1 About configuration guidelines

You must configure the supplied RTL for your specific implementation. After configuring the RTL, you must verify that the configuration completed correctly.

The `rtl_configure.pl` script automates the process of modifying the RTL files based on a set of configuration options and generates a configured Cortex-A57 MPCore multiprocessor. The `rtl_checking.pl` script verifies the configuration and generates a checksum value based on the configured RTL.

Caution

For successful configuration of the RTL you must:

- Set up the configurable options.
- Integrate the memory, see Chapter 3 *Memory Integration*.
- Check the configured RTL, see Chapter 4 *Checking your RTL*.

Failure to complete all the necessary configuration can result in malfunction.

2.2 Configuration options

This section describes the scripts that you must run to configure the delivered RTL for your specific implementation, and to verify your configuration. It contains the following sections:

- *Configuration script*.
- *Checking script* on page 2-8.
- *IP-XACT configuration script* on page 2-9.

2.2.1 Configuration script

This section describes the script to configure your RTL. It contains the following sections:

- *Script functionality*.
- *Script synopsis* on page 2-4.
- *Script examples* on page 2-6.

Script functionality

The configuration script is `rtl_configure.pl` and is located in the `logical/shared/tools/bin` directory.

The configuration script requires that Perl is installed in the execution path.

The configuration script enables you to configure the following parameters:

- The number of processors in the Cortex-A57 MPCore multiprocessor.
- The L2 cache size.
- The L2 Tag and Data RAM register slices.
- The L2 arbitration logic register slice.
- The L1 ECC/Parity support.
- Whether regional clock gating is included.
- Whether the external memory system bus interface is ACE or CHI.
- Whether the Cryptography engine is included for the processors.

Each processor is configured with the same Cryptography and L1 ECC/parity parameters.

The configuration script:

- Copies the entire source `logical` directory tree to the specified output directory.
- Preserves the structure within the output directory so the files are always located in the same relative path as with the source directory.
- Resizes the L2 cache RAM macro instantiations to the correct size for the given L2 cache size configuration.
- Removes the L1 Data ECC RAM instances and disables L1 instruction cache parity if ECC/parity is excluded.
- Replaces removed blocks with dummy empty blocks with tie-offs if you exclude the Cryptography engine.
- Modifies the top-level Cortex-A57 MPCore multiprocessor module to remove the extra pins associated with removed blocks.
- Generates a `README.txt` file in the specified output directory that includes the:
  - Complete configuration options.
  - List of modified files.

— List of files that require additional modification in the RAM integration step.

— List of files with dummy empty blocks with tie-offs.

Script synopsis

The format of the script is:

```
%> rtl_configure.pl -srcdir <dir> -outdir <dir> -cpu <number> -L2size <size>
{-L2tagslice <number>} {-L2datslice <number>} {-L2arbslice <present>}
{-rcg <present>} {-ecc <options>} {-bus <options>} {-crypto <present>}
```

The following command displays the script functionality and options:

```
%> rtl_configure.pl -help
```

The configuration script can take the following options:

`-srcdir <dir>`

Path to the source or install `logical` RTL directory. For example:

`MP019-BU-00000-r0p0-00rel0/cortexa57/logical`

——— **Note** ———

This directory must not be modified prior to running this script.

———————————

`-outdir <dir>`

Path to the output directory created by the script, that includes the configured RTL files. The entire source `logical` directory tree is copied to the output directory and the RTL files are modified in this output directory tree.

——— **Note** ———

This directory must not already exist prior to running this script and must be outside the source `logical` directory tree.

———————————

The configuration script cannot create more than one new level of hierarchy. For example, if you specify an output directory of `../cortexa57_configured/a57_cpu2_l21M_ace/logical`, you must create all upper directories down to `a57_cpu2_l21M_ace`. The configuration script creates the `logical` directory and all subdirectories within it.

`-cpu <number>`

The number of processors in the Cortex-A57 MPCore multiprocessor. You can configure the multiprocessor to have 1, 2, 3, or 4 processors.

——— **Note** ———

You must specify this option. There is no default for the number of processors.

———————————

`-L2size <size>`

The L2 cache size, in KB. You can enter 512, 1024, or 2048 as the value of this option.

——— **Note** ———

You must specify this option. There is no default L2 cache size.

———————————

`{-L2tagslice <number>}`

Specifies the L2 Tag, Snoop Tag, Dirty, and *Prefetch Stride Queue* (PSQ) RAM register slices. The value for this option can be 0 or 1. If you do not include this option, L2 Tag, Snoop Tag, Dirty, and PSQ RAM register slices are not added.

This option adds register slices to the L2 Tag, Snoop Tag, Dirty, and PSQ RAMs.

Table 2-1 shows the valid combinations for the `L2tagslice` and `L2datslice` options.

`{-L2datslice <number>}`

Specifies the L2 Data RAM register slices. The value for this option can be 0, 1, or 2. If you do not include this option, L2 Data RAM register slices are not added.

This option adds register slices to the L2 Data RAM.

Table 2-1 shows the valid combinations for the `L2tagslice` and `L2datslice` options.

Table 2-1 Valid options for L2tagslice and L2datslice

`L2tagslice`	`L2datslice`
0	0
0	1
0	2
1	1
1	2

`{-L2arbslice <present>}`

Specifies whether the L2 arbitration logic register slice is present. The value for this option can be:

0 L2 arbitration logic register slice is not present.

1 L2 arbitration logic register slice is present. This adds:

- An additional pipeline stage for the processor-L2 arbitration logic interface, to the L2 arbitration logic.
- A flop stage to the L2 Tag bank clock enable.

If you do not include this option, the L2 arbitration logic register slice is not present.

`{-rcg <present>}`

Specifies whether regional clock gating is included. The value for this option can be:

0 Regional clock gating is not present.

1 Regional clock gating is present. Adds an additional level of clock gating to several logic blocks. Regional clock gating can provide lower power dissipation, but at the cost of a more complex clock tree implementation.

If you do not include this option, regional clock gating is present.

`{-ecc <options>}`

Specifies whether the L1 cache provides ECC/Parity support. The value for this option can be:

`L1andL2` Both L1 and L2 have ECC/Parity support.

`L2only` Only L2 has ECC/Parity support. L1 does not have ECC/Parity support.

If you do not include this option, both L1 and L2 have ECC/Parity support.

Note

ECC in the L2 cache RAMs is not a configurable option. The L2 cache RAMs always include ECC protection.

`{-bus <options>}`

Specifies the external memory interface support. The values for this option can be:

`ACE` The ACE interface is implemented.

`Skyros` The CHI interface is implemented.

If you do not include this option, the external memory interface is ACE.

`{-crypto <present>}`

Specifies whether the Cryptography Extension is included. The value for this option can be:

0 The Cryptography Extension is not present.

1 The Cryptography Extension is present.

If you do not include this option, the Cryptography Extension is present.

Note

The `-crypto` option only applies to the Cortex-A57 MPCore multiprocessor Cryptography Extension product.

Script examples

Example 2-1 shows an example invocation of the configuration script and the screen output from the script.

Note

The `-crypto` option only applies to the Cortex-A57 MPCore multiprocessor Cryptography Extension product.

Example 2-1 Configuration script invocation example

```
%> ./MP019-BU-00000-r0p0-00rel0/cortexa57/logical/shared/tools/bin/rtl_configure.pl -srcdir
MP019-BU-00000-r0p0-00rel0/cortexa57/logical -outdir a57_cpu2_l21M_ace -cpu 2 -L2size 1024 -bus ACE

NOTE: -L2tagslice option not specified. L2 Tag RAMs will have 0 slice.
NOTE: -L2datslice option not specified. L2 Data RAMs will have 0 slice.
NOTE: -L2arbslice option not specified. L2 arbitration slice will not be present.
NOTE: -rcg option not specified. Regional gated clocks will be present.
NOTE: -ecc option not specified. Both L1 and L2 will have ECC/Parity support.
NOTE: -crypto option specified. Cryptography Extension will be present.
```

```
STATUS: Copying entire design tree from MP019-BU-00000-r0p0-00rel0/cortexa57/logical to a57_cpu2_l21M_ace
Configuring RTL.......done.

IMPORTANT! Please run  the rtl_checking.pl script and ensure you log the script output into your
           'Verification Confirmation' summary, and for release to support-cores@arm.com on request.
%>
```

See the `a57_cpu2_l21M_ace/README.txt` file for additional information about the results of the configuration.

Example 2-2 shows the `README.txt` file produced by the configuration script.

Example 2-2 `README.txt` **example**

```
This directory tree contains the Cortex-A57 MPCore RTL files configured by
rtl_configure.pl with the following parameters:

  CPU number                : 2
  L2 Cache Size             : 1024 KB
  L2 Tag RAM Slice          : 0
  L2 Data RAM Slice         : 0
  L2 Arbitration Slice      : No
  Regional gated clocks     : Yes
  ECC/Parity                : L1 and L2
  Bus master interface      : ACE
  Cryptography Extension    : Yes


The following files have been changed from the original RTL.The changes are complete,
and no further modifications are required:

  [List of RTL files]


The following L2 RAM files have been changed from the original RTL.They require
modification to integrate customer-specific RAM models:

  [List of RTL files]


The following L1 and L2 RAM files have not changed from the original RTL. They require
modification to integrate customer-specific RAM models:

  [List of RTL files]


The following files are new to the original RTL. The changes are complete, and no
further modifications are required:

  [List of RTL files]


IMPORTANT! Please run  the rtl_checking.pl script and ensure you log the script output into your
           'Verification Confirmation' summary, and for release to support-cores@arm.com on request.
```

2.2.2 Checking script

This section describes the checking script that you must run after you configure the RTL. It contains the following sections:

- *Script functionality*.
- *Script synopsis*.
- *Script examples*.

Script functionality

You must run the `rtl_checking.pl` script after the `rtl_configure.pl` script completes to verify the success of the configuration script.

The checking script requires that Perl and `sum`, the GNU core utility, are installed in the execution path.

See the *ARM® Cortex®-A57 MPCore™ Processor Release Note* for information about the versions of software required to run this script.

The `rtl_checking.pl` script extracts and reports the following information from the configured RTL:

- A checksum value based on the configured RTL.
- Cortex-A57 MPCore multiprocessor revision number.
- Number of processors.
- L2 cache size.
- L2 Tag and Data RAMs register slices.
- L2 arbitration register slice.
- Regional gated clock existence.
- L1 ECC/Parity support.
- Bus master interface type.
- Cryptography engine existence.

The checking script generates a `checksum.txt` file in the specified output directory. This file contains the exact output displayed to the screen.

You must submit the output from the checking script to your Cortex-A57 MPCore multiprocessor technical representative for checksum value verification before you proceed with the implementation of the configured RTL into your products. See *Requirements for sign-off* on page 9-3.

Script synopsis

The checking script takes the output directory specified in the configuration script as a single parameter.

The format of the script is:

```
%> rtl_checking.pl -outdir <dir>
```

The following command displays the script functionality and options:

```
%> rtl_checking.pl -help
```

Script examples

Example 2-3 on page 2-9 shows an example invocation of the checking script.

Example 2-3 Checking script invocation example

```
%> ./MP019-BU-00000-r0p0-00rel0/cortexa57/logical/shared/tools/bin/rtl_checking.pl -outdir a57_cpu2_l21M_ace
```

Example 2-4 shows the `checksum.txt` file produced by the checking script.

Example 2-4 checksum.txt example

```
Cortex-A57 MPCore RTL configuration check script output

        checksum                 : 00083ff1b8

        Revision                 : r2p0

        CPU number               : 2
        L2 Cache Size            : 1M
        L2 Tag RAM Slice         : 0
        L2 Data RAM Slice        : 0
        L2 Arbitration Slice     : No
        Regional gated clocks    : Yes
        ECC/Parity               : L1 and L2
        Bus master interface     : ACE
        Cryptography Extension   : Yes

IMPORTANT! Please ensure you log the Cortex-A57 RTL configuration check script output into your
           'Verification Confirmation' summary, and for release to support-cores@arm.com on request.
```

2.2.3 IP-XACT configuration script

This section describes how to render the Cortex-A57 MPCore multiprocessor IP-XACT using the `gen_ipxact.pl` script located in:

`outdir``/atlas/ipxact/tools/`

Note

- *`outdir`* is the output directory specified by the `-outdir <dir>` option in the `rtl_configure.pl` configuration script.
- The `gen_ipxact.pl` script uses Perl and the XML::Writer module. Therefore, the Perl installation must have the XML::Writer module installed. Consult with your system administrator if you are unsure whether this module is installed.

  It is possible to install the XML::Writer module in your local directory without root privilege, using the bootstrapping technique that the Perl CPAN documentation describes at `http://search.cpan.org/dist/local-lib/lib/local/lib.pm#The_bootstrapping_technique`
- For more information about the script, type `gen_ipxact.pl -h`.

Execute the following command from the *`outdir`*`/atlas/ipxact/tools` directory to generate an IP-XACT file, `CORTEXA57.xml`, in the *`outdir`*`/atlas/verilog` directory:

```
./gen_ipxact.pl -p gen_ipx.pin_db -r gen_ipx.regs_db -f gen_ipx.file_db \
 -o ../../verilog/CORTEXA57.xml -v ../../verilog/CORTEXA57.v
```

Note

You require Perl in your execution path to execute the command.

The script uses the configured top-level verilog file, `CORTEXA57.v`, to create an IP-XACT file that matches the configuration of the top-level RTL file.

Chapter 3
Memory Integration

This chapter describes memory integration for the Cortex-A57 MPCore multiprocessor. It contains the following sections:

3.1 About memory integration

You must perform memory integration and you must verify the integration. See *Resource requirements for memory integration* on page 3-4 for integration information. See *Confirmation of memory integration* on page 3-27 for information about how to verify RAM integration.

Caution

For successful configuration of the RTL you must:

- Set up the configurable options.
- Integrate the memory.

Failure to complete all the necessary configuration can result in malfunction.

Figure 3-1 shows the memory integration process.

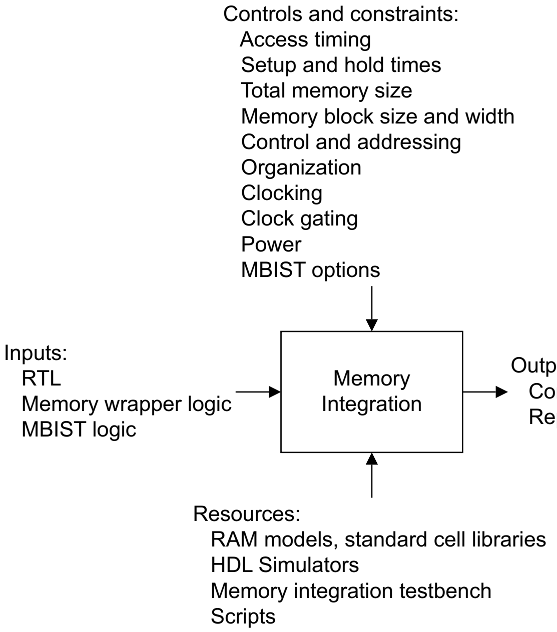


Figure 3-1 Memory integration process

When your RTL configuration is complete, the RAM wrapper blocks of the RTL model contain correctly sized generic RAM models. Although these RAM models are suitable for logical simulation of the Cortex-A57 MPCore multiprocessor implementation, they cannot be implemented without change. You must substitute your physical RAMs for the generic RAM models to correctly floorplan, synthesize, and time the design.

The hierarchy is organized so the only requirement is to modify the RAM wrapper blocks. These blocks consist of a single instance of the generic `atl_sram` model. They often also include minor clock gating logic. The RAM integration process substitutes the physical RAM for this generic model. You must not make changes to the logic block that instantiates the RAM wrapper blocks. ARM provides a RAM integration testbench to verify the result of the RAM integration process.

The `atl_cpu` RAMs are located in the following units:

***Instruction Fetch* (IF)**

The L1 instruction cache and branch prediction hardware.

***Load/Store* (LS)** The L1 data cache.

***Level 2* (L2) processor slave unit**

The L2 *Translation Lookaside Buffer* (TLB).

Note

All the RAMs in these three units are duplicated for each processor instance. This is important for area planning in a multiprocessor implementation.

The `atl_noncpu` RAMs implement the L2 cache that is shared between the processors. Many RTL configuration options affect the implementation of the shared L2 memory. You must consider implementation constraints when configuring the RTL. During L2 integration you must consider timing requirements because some of the L2 RAMs might not complete accesses in a single clock cycle.

Note

Sign-off requires successful completion of RAM integration and verification. See *Requirements for sign-off* on page 9-3.

3.2 Resource requirements for memory integration

To perform memory integration, you require a target library RAM to substitute your physical RAMs for the generic RAM models. The physical RAMs must meet the constraints listed in *Controls and constraints for memory integration* on page 3-5. You also require the supplied memory integration testbench to validate your memory integration. See *Confirmation of memory integration* on page 3-27.

3.3 Controls and constraints for memory integration

You must ensure that your library RAM satisfies the requirements in:

- *atl_cpu RAM organization* on page 3-6.
- *atl_noncpu, shared L2, RAM organization* on page 3-10.

3.4 atl_cpu RAM organization

Each processor contains eight unique RAM modules that require integration. Each RAM module has a unique size and write granularity. These unique RAM types can be instantiated multiple times. Each unique RAM in the Cortex-A57 MPCore multiprocessor is contained inside a single Verilog wrapper file. Only modify the files specified in the tables in this section. Any other edits can cause the design to fail RTL validation.

3.4.1 RAM requirements

You must ensure that the physical RAMs incorporated in the Cortex-A57 MPCore multiprocessor substituted for the generic RAMs included in the `atl_cpu` block satisfy these requirements:

- All timings must be synchronous to the master rising clock edge.
- There is single-cycle access at the main clock frequency.
- RAM accesses must be suppressed during production test, preferably by inhibiting chip select.
- The clock can be held static during inactive cycles because there are no minimum frequency requirements.
- Because RAM outputs must always be driven, you must not use tristate logic. However, if the RAM is disabled for a given cycle:
  - The outputs are not required to retain the value read in the last legal access.
  - The RAM outputs are not required to be meaningful during a write cycle.
- Variable bit-write control is required for some RAM blocks.
- Verilog simulation models are required to run the RAM integration testbench.

These first two requirements mean that control, address, and write data inputs are set up before the rising clock edge.

This setup time must be of some percentage value that is less than 100% of the period of the master clock. There can be significant logic external to the RAM within the setup cycle. To maximize frequency, you must select RAM architectures with minimal setup time.

The output read data is launched from the same rising clock edge, and the clock-to-output delay must be of some percentage value that is less than 100% of the period of the main clock.

Figure 3-2 shows the timing of L1 RAM cache access.

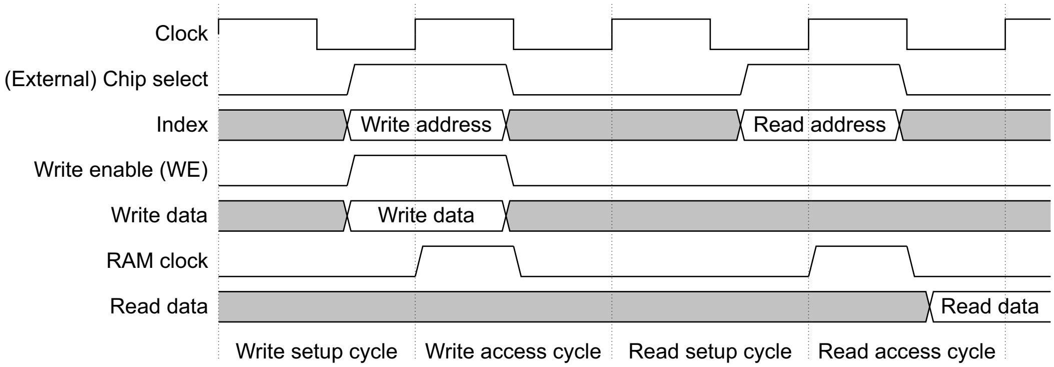


Figure 3-2 L1 RAM cache access timing

Figure 3-2 on page 3-6 shows the timing requirements in waveform view. The waveform shows four sequential clock cycles. The time order of the four cycles correspond to the setup cycle for a write cycle, idle, the setup cycle for a read cycle then idle. The positive clock edge between a setup cycle and an access cycle is the master rising clock edge. To save power, the rising clock to the RAM is suppressed for idle cycles, see *Flow for memory integration* on page 3-21 for more information.

Control and data inputs, that is, Index, Write enable, and Write data, must meet setup and hold requirements against the RAM clock. These requirements are determined by the RAM characteristics, buffering, and wire delay to the RAM pin.

The Read data output is launched some time after the rising RAM clock edge. The Read data launch time must be early enough to meet the setup requirements of receiving flops. These requirements are determined by any optional post-RAM logic, buffering, and wire delay.

The absence or presence of L1 ECC affects the RAM sizes in the Cortex-A57 MPCore multiprocessor. This affects the data cache Tag and Data RAM module name and RAM size.

Note

Each processor in the Cortex-A57 MPCore multiprocessor has the same ECC characteristics, which are specified by a single option during RTL configuration. See *Configuration script* on page 2-3 for more information.

3.4.2 Instruction fetch unit RAMs

The instruction fetch unit contains five unique RAM modules. These RAMs comprise the instruction cache and branch prediction hardware.

Table 3-1 shows the five RAM models and their associated modules and file names.

Table 3-1 Instruction fetch unit RAM model

RAM name	RAM module name	Wrapper file name
BTB[a]	`atl_if_btb_ram`	`models/rams/generic/atl_if_btb_ram.v`
GHB[b]	`atl_if_ghb_ram`	`models/rams/generic/atl_if_ghb_ram.v`
IP[c]	`atl_if_ip_ram`	`models/rams/generic/atl_if_ip_ram.v`
I-cache Data array	`atl_if_data_ram`	`models/rams/generic/atl_if_data_ram.v`
I-cache Tag array	`atl_if_tag_ram`	`models/rams/generic/atl_if_tag_ram.v`

[a] *Branch Target Buffer* (BTB).

[b] *Global History Buffer* (GHB).

[c] *Indirect Predictor* (IP).

Table 3-2 shows the required sizes and write enable types for the RAMs in the instruction fetch unit.

Table 3-2 Instruction fetch RAM sizes and write enable types

RAM name	Write enable type	RAM size	Instances
`atl_if_btb_ram`	Word[a]	512×86	4
`atl_if_ghb_ram`	1-bit	512×8	12
`atl_if_ip_ram`	Word	256×64	2
`atl_if_data_ram`	Word	256×72	24
`atl_if_tag_ram`	36-bit	128×108	2

[a] Word write enable indicates that the entire row is written. No bit write capability is required for this RAM.

Note

RAM sizes are given in the form ROWS×COLS, where only a single row can be accessed per cycle. COLS bits are available in the row accessed.

Configuring the L1 to exclude ECC inhibits parity checking in the L1 instruction cache. The RTL configuration accomplishes this by forcing signals to turn off the parity checking logic. The instruction fetch unit RAM sizes and names are not affected by the inclusion or exclusion of parity.

3.4.3 Load/Store Unit RAMs

The Load/Store Unit contains two unique RAM modules. These module names and sizes differ depending on whether your implementation includes L1 ECC or not. This is determined at RTL configuration time based on the value of the `-ecc` option. See *Configuration script* on page 2-3 for more information. If you specify `-ecc L1andL2`, indicating that both L1 and L2 ECC are present, then the RAM module names and sizes in Table 3-3 must be used.

Table 3-3 shows the Load/Store Unit RAM module names with ECC.

Table 3-3 Load/Store Unit RAM models for L1 ECC enabled configuration

RAM name	Wrapper module name	Wrapper file name
D-cache Data array	`atl_ls_data_ecc_ram`	`models/rams/generic/atl_ls_data_ecc_ram.v`
D-cache Tag array	`atl_ls_tag_ecc_ram`	`models/rams/generic/atl_ls_tag_ecc_ram.v`

Table 3-4 shows the Load/Store Unit RAM module names without ECC.

Table 3-4 Load/Store Unit RAM models for L1 non-ECC configuration

RAM name	Wrapper module name	Wrapper file name
D-cache Data array	`atl_ls_data_ram`	`models/rams/generic/atl_ls_data_ram.v`
D-cache Tag array	`atl_ls_tag_ram`	`models/rams/generic/atl_ls_tag_ram.v`

Table 3-5 shows the required sizes and write enable types for the RAMs in the Load/Store Unit with ECC configuration.

Table 3-5 Load/Store Unit RAM sizes and write enable types for L1 with ECC

RAM name	Write enable type	RAM size	Instances
atl_ls_data_ecc_ram	39-bit	256×312	4
atl_ls_tag_ecc_ram	40-bit	64×80	4

Table 3-6 shows the required sizes and write enable types for the RAMs in the Load/Store Unit without ECC configuration.

Table 3-6 Load/Store Unit RAM sizes and write enable types for L1 without ECC

RAM name	Write enable type	RAM size	Instances
atl_ls_data_ram	32-bit	256×256	4
atl_ls_tag_ram	33-bit	64×66	4

3.4.4 L2 processor slave unit RAMs

A portion of the L2 cache that exists within each processor is called the L2 processor slave. This unit contains a single RAM module that implements the L2 TLB.

Table 3-7 shows the L2 processor slave RAM module name.

Table 3-7 L2 processor slave RAM module

RAM name	Wrapper module name	Wrapper file name
L2 TLB	atl_l2_tlb_ram	models/rams/generic/atl_l2_tlb_ram.v

Table 3-8 shows the required size and write enable for the L2 processor slave RAM.

Table 3-8 L2 processor slave RAM size and write enable types

RAM name	Write enable type	RAM size	Instances
atl_l2_tlb_ram	1-bit	256×130	4

3.5 atl_noncpu, shared L2, RAM organization

The non-processor or shared L2 cache unit contains multiple RAMs that implement the L2 cache array. Figure 3-3 shows the RAMs are split into two groups that exist in different `atl_noncpu` level blocks.

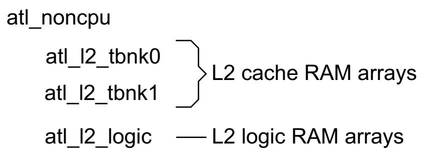


Figure 3-3 Shared L2 cache RAM organization

The L2 cache consists of two banks. Each bank consists of Data, Tag, and Dirty RAMs and is instantiated twice at the `atl_noncpu` hierarchy. The number of banks is not configurable.

Only the RAMs comprising the L2 arrays in the banks are resized depending on the chosen L2 cache size. The L2 cache RAM arrays and the L2 logic RAM arrays have different timing requirements, so they are described separately.

3.5.1 L2 logic RAMs

The L2 logic RAMs contains two unique RAM modules. The timing requirements for these RAMs are similar to the processor RAMs. These RAMs are expected to operate in a single cycle and must have the same constraints as those for the `atl_cpu` RAMs listed in *RAM requirements* on page 3-6.

Table 3-9 shows the L2 logic RAM module names.

Table 3-9 L2 logic RAMs

RAM name	RAM module name	Wrapper file name
L2 Snoop Tag	`atl_l2_snp_tag_ram`	`models/rams/generic/atl_l2_snp_tag_ram.v`
L2 Prefetch Stride Queue	`atl_l2_psq_ram`	`models/rams/generic/atl_l2_psq_ram.v`

Table 3-10 shows the L2 logic RAM sizes and write enable types.

Table 3-10 L2 Logic RAM sizes and write enable types

RAM name	Write enable type	RAM size	Instances
`atl_l2_snp_tag_ram`	Bit (40-bit)	128×80	2× the number of processors
`atl_l2_psq_ram`	Bit (52-bit)[a]	32×52	1× the number of processors

[a] All 52 bits are written together.

The current `atl_sram` model chosen for these RAMs indicates a bit-write requirement. The logical requirements for write granularity are shown in parentheses in Table 3-10. You can remove the duplicate write enables and choose a RAM with larger write granularity.

The number of Snoop Tag and *Prefetch Stride Queue* (PSQ) RAMs depends on the number of processors implemented in the Cortex-A57 MPCore multiprocessor. The RTL configuration script removes the RAMs associated with non-existent processors if the number of implemented processors is less than four.

To accommodate the larger L2 floorplan, you can add two additional pipeline stages on the input and output sides of the Snoop Tag and PSQ RAMs. See *L2 RAM pipeline timing* on page 3-13 for more information.

3.5.2 L2 cache RAMs

The configuration of the L2 cache size specification directly affects the size of the arrays. For example, doubling the L2 size doubles the size of the arrays.

Table 3-11 shows the L2 single cycle cache RAM module name.

Table 3-11 L2 single cycle cache RAM

RAM name	RAM module name	Wrapper file name
L2 cache Dirty RAM	`atl_l2_dirty_ram`	models/rams/generic/atl_l2_dirty_ram.v

Table 3-12 shows the L2 multi-cycle cache RAM module names.

Table 3-12 L2 multi-cycle cache RAMs

RAM Name	Ram Module Name	Wrapper file name
L2 cache Data RAM	`atl_l2_data_ram`	`models/rams/generic/atl_l2_data_ram.v`
L2 cache Tag RAM	`atl_l2_tag_ram`	`models/rams/generic/atl_l2_tag_ram.v`

Table 3-13 shows the RAM sizes for each supported L2 cache size. The number of instances does not depend on memory size. The instance number in the table is the total number of each RAM instantiated for both banks, not for each bank.

Table 3-13 L2 cache RAM sizes and write enable type

RAM name	Write enable type	RAM size for L2 cache size			Instances
		512KB	1024KB	2048KB	2 tag banks
`atl_l2_data_ram`	Bit (8-bit)	4096×144	8192×144	16384×144	8
`atl_l2_tag_ram`	Bit (41-bit)	256×82	512×82	1024×82	16
`atl_l2_dirty_ram`	Bit (Alternating 8-bit and 4-bit)	256×192	512×192	1024×192	2

The current `atl_sram` model chosen for these RAMs indicates a bit-write requirement. The logical requirements for write granularity are shown in parenthesis in Table 3-13. You can remove the duplicate write enables and choose a RAM with larger write granularity.

L2 RAMs are always configured with ECC.

The L2 RAMs that can be configured with parity are:

- L2 PSQ RAM.
- L2 slave TLB RAM.

L2 RAM read and write path

This section describes the various categories of L2 RAM timing paths in the multiprocessor. For this section, routing delay refers to both propagation delays because of resistive-capacitive effects and buffering delays.

Tag RAM timing paths

Figure 3-4 shows the timing paths for the Tag RAMs:

- RAM input path.
- RAM read path.

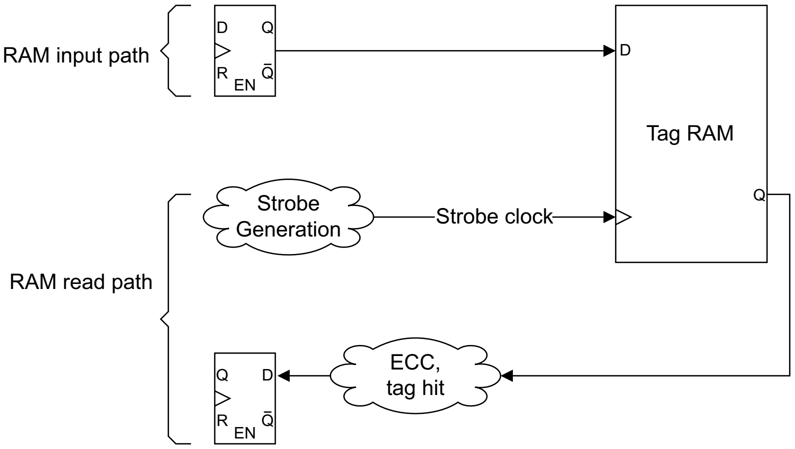


Figure 3-4 Tag RAM timing paths

The Tag RAM input paths, by default, are single cycle paths for setup and 2-cycle multi-cycle paths for hold. However, software can change the setup paths to be a 2-cycle multi-cycle path by programming L2CTLR[9]=1. The input signals must meet the Tag RAM setup and hold requirements against the strobe clock. The input signal delay is the sum of these component delays:

- Source register.
- Input routing.

The strobe clock delay depends on the configuration, see *L2 strobe configuration* on page 3-18.

The RAM read path is a multi-cycle setup path. This path begins with the strobe clock. To determine how many cycles a Tag RAM read requires, you must calculate the total read path delay. The total read path delay is the sum of these component delays:

- Strobe clock delay.
- Tag RAM.
- RAM output logic.
- RAM output routing.

The RAM read path must meet the setup and hold requirements of the read data output register controlled by the main system clock **CLK**.

Data RAM timing paths

Figure 3-5 on page 3-13 shows the timing paths for the Data RAMs:

- RAM input path.
- RAM read path.

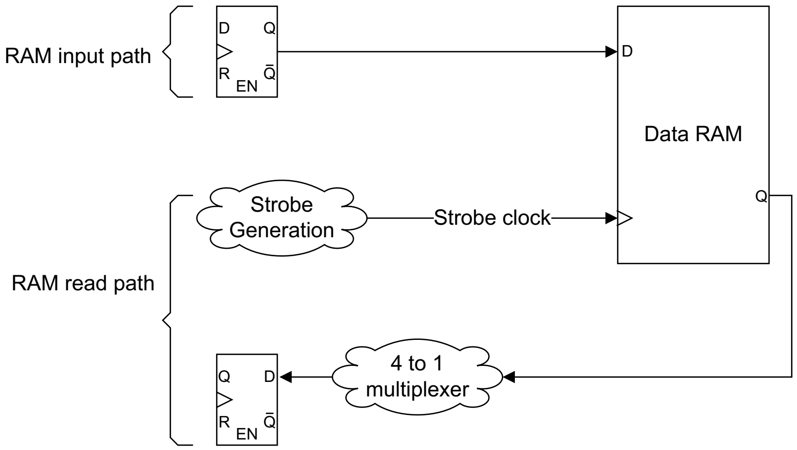


Figure 3-5 Data RAM timing paths

The Data RAM input paths, by default, are single cycle paths for setup and 2-cycle multi-cycle paths for hold. However, software can change the setup paths to be a 2-cycle multi-cycle path by programming L2CTLR[5]=1. The input signals must meet the Data RAM setup and hold requirements against the strobe clock. The input signal delay is the sum of these component delays:

- Source register.
- Input routing.

The strobe clock delay depends on the configuration, see *L2 strobe configuration* on page 3-18.

The RAM read path is a multi-cycle setup path. This path begins with the strobe clock. To determine how many cycles a Data RAM read requires, you must calculate the total read path delay.

The total read path delay is the sum of these component delays:

- Strobe clock delay.
- Data RAM.
- RAM output logic.
- RAM output routing.

The RAM read path must meet the setup and hold requirements of the read data output register controlled by the main system clock **CLK**.

3.5.3 L2 RAM pipeline timing

This section describes the waveforms for the L2 RAM accesses with various numbers of access cycles and slice configurations. It contains the following sections:

- *Tag array accesses without slices* on page 3-14.
- *Tag array accesses with slices* on page 3-15.
- *Data array accesses without slices* on page 3-16.
- *Data array accesses with slices* on page 3-17.

Tag array accesses without slices

All inputs to the Tag array are set up in the L2 stage and are stable until the next L2 stage that targets the same Tag bank. The strobe clock is formulated by the RAM control logic, based on the chip select and raised during the following cycle. See *L2 strobe configuration* on page 3-18 for more information about the strobe clock generation.The Tag array is accessed during the L3x through L3 stages as shown in Figure 3-6 through Figure 3-10 on page 3-16.

Output sampling from the Tag array occurs in the L3 stage.

Figure 3-6 shows the back-to-back Tag array accesses for a 2-cycle tag configuration.

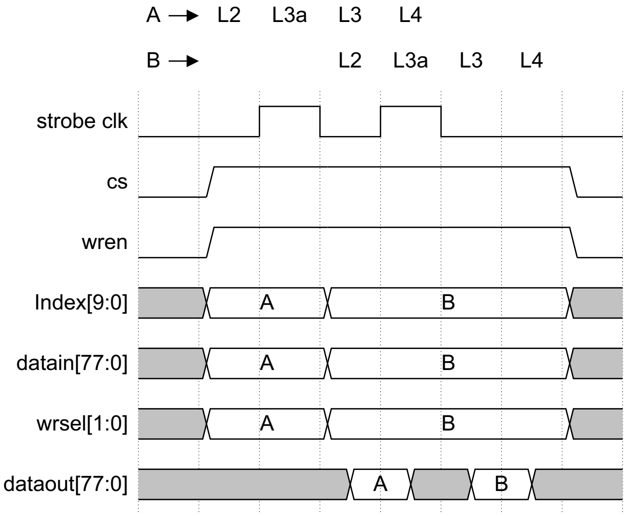


Figure 3-6 Back-to-back Tag array accesses for 2-cycle tag configuration

Figure 3-7 shows the back-to-back Tag array accesses for a 3-cycle tag configuration.

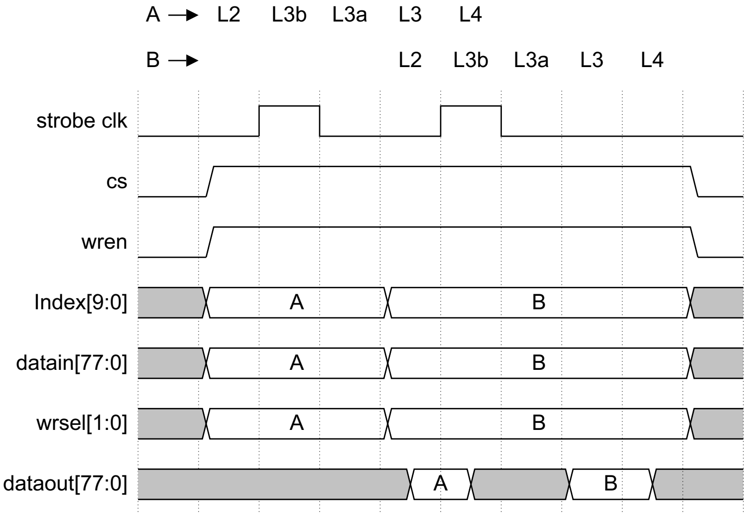


Figure 3-7 Back-to-back Tag array accesses for 3-cycle tag configuration

Figure 3-8 on page 3-15 shows the back-to-back Tag array accesses for a 4-cycle tag configuration.

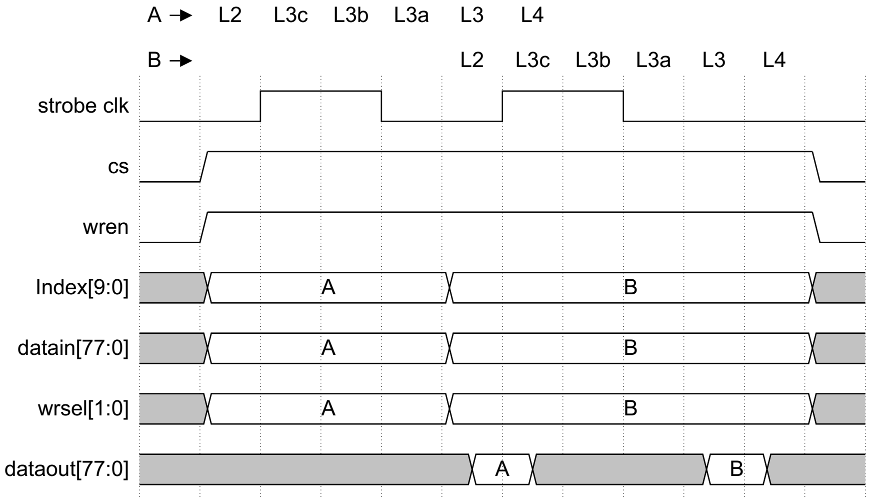


Figure 3-8 Back-to-back Tag array accesses for 4-cycle tag configuration

Tag array accesses with slices

An implementation can insert slices around the RAMs to accommodate increases in wire delay associated with larger L2 cache sizes. In this case the pipeline is adjusted for the extra stages. The actual RAM access time can remain the same as without the slices, that enables the design to maintain a high throughput. However, the latency of the request is extended by the number of slices inserted.

Figure 3-9 shows the back-to-back Tag array accesses for a 2-cycle tag configuration with the additional stages added for the slice implementation. In this figure L3c and L3 are slice stages.

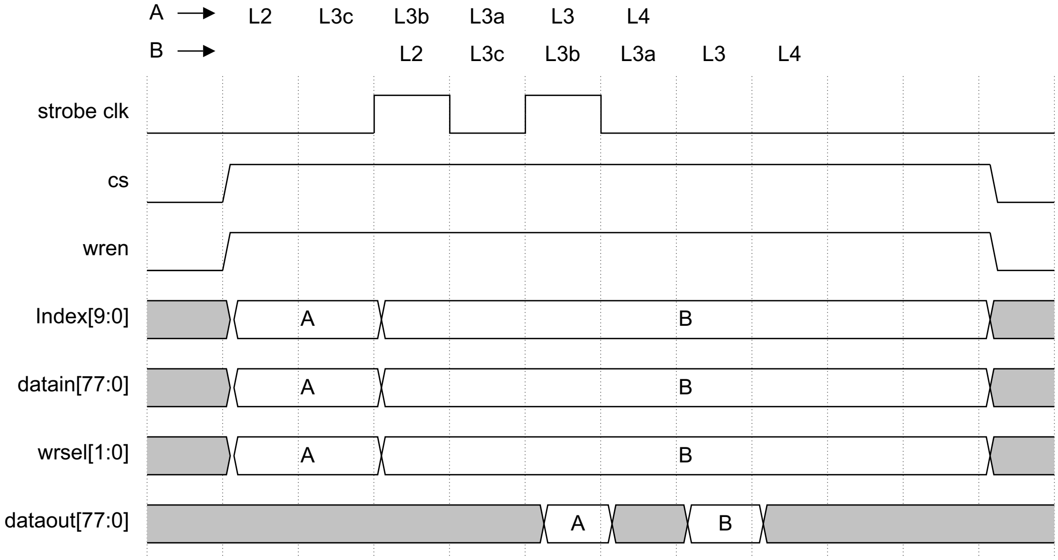


Figure 3-9 Back-to-back Tag array accesses for 2-cycle tag configuration with additional stages

Figure 3-10 on page 3-16 shows the back-to-back Tag array accesses for a 3-cycle tag configuration with the additional stages added for the slice implementation. In this figure L3d and L3 are slice stages.

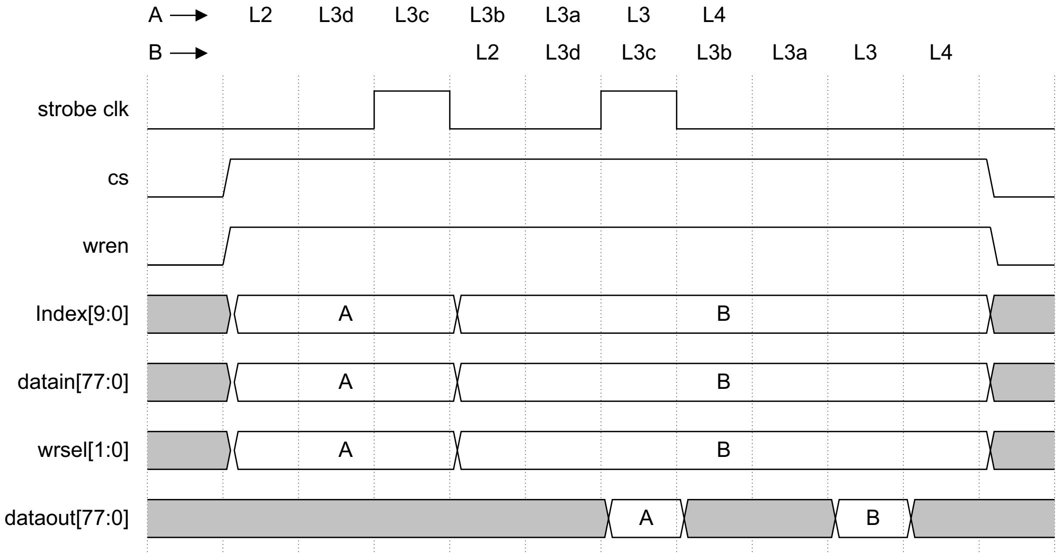


Figure 3-10 Back-to-back Tag array accesses for 3-cycle tag configuration with additional stages

Data array accesses without slices

All inputs to the Data array are set up in the L4 stage and are stable until the next L4 stage that targets the same data bank. The strobe clock is formulated by the RAM control logic, based on the chip select and raised during the following cycle. See *L2 strobe configuration* on page 3-18 for more information about the strobe clock generation. Figure 3-11 through Figure 3-14 on page 3-18 show the Data array accesses during the L5x through L5 stages.

Output sampling from the Data array occurs in the L5 stage.

Figure 3-11 shows the back-to-back Data array accesses for a 3-cycle data configuration.

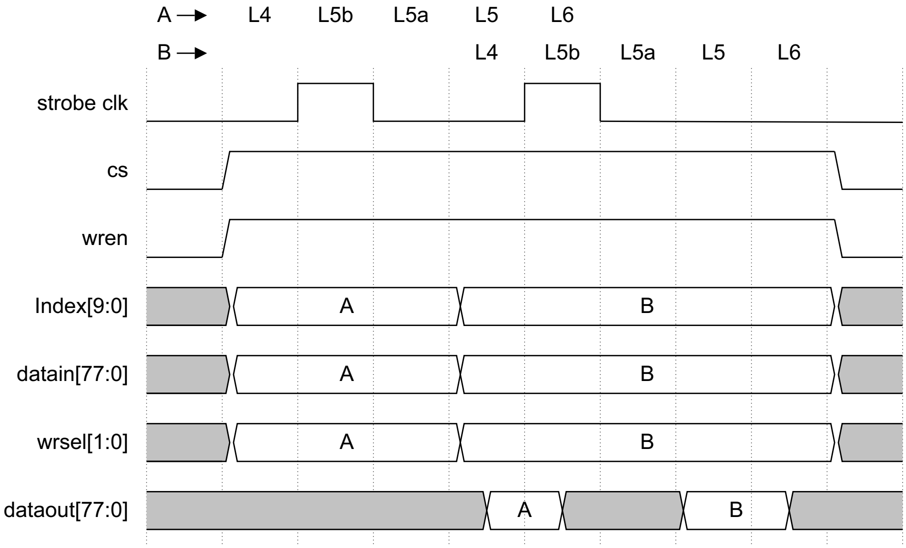


Figure 3-11 Back-to-back Data array accesses for 3-cycle data configuration

Figure 3-12 on page 3-17 shows the back-to-back Data array accesses for a 4-cycle data configuration.

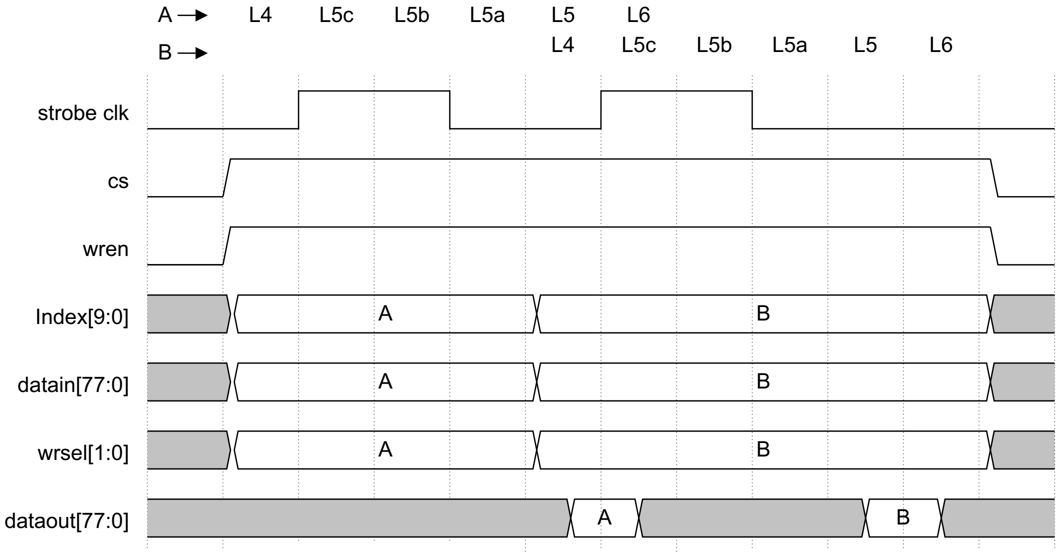


Figure 3-12 Back-to-back Data array accesses for 4-cycle data configuration

Data array accesses with slices

Similar to the Tag array, an implementation can insert slices around the RAMs to accommodate increases in wire delay associated with larger L2 cache sizes. If an implementation selects to insert slices then the pipeline is adjusted for the extra stages. The actual RAM access time can remain the same as without the slices, that enables the design to maintain a high throughput. However, the latency of the request is extended by the number of slices inserted.

Figure 3-13 shows back-to-back Data array accesses for a 3-cycle data configuration with additional stages added for a single slice implementation. In this figure, L5d and L5 are slice stages.

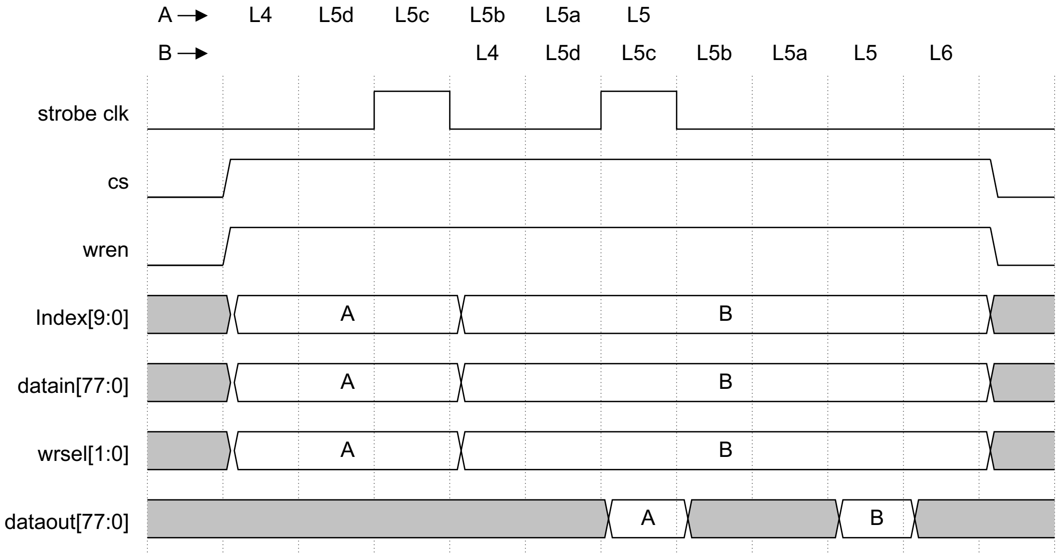


Figure 3-13 Back-to-back Data array accesses for 3-cycle data configuration with additional stages

Figure 3-14 on page 3-18 shows back-to-back Data array accesses for a 4-cycle data configuration with additional stages added for a 2-slice implementation. In this figure, L5g, L5f, L5a, and L5 are slice stages.

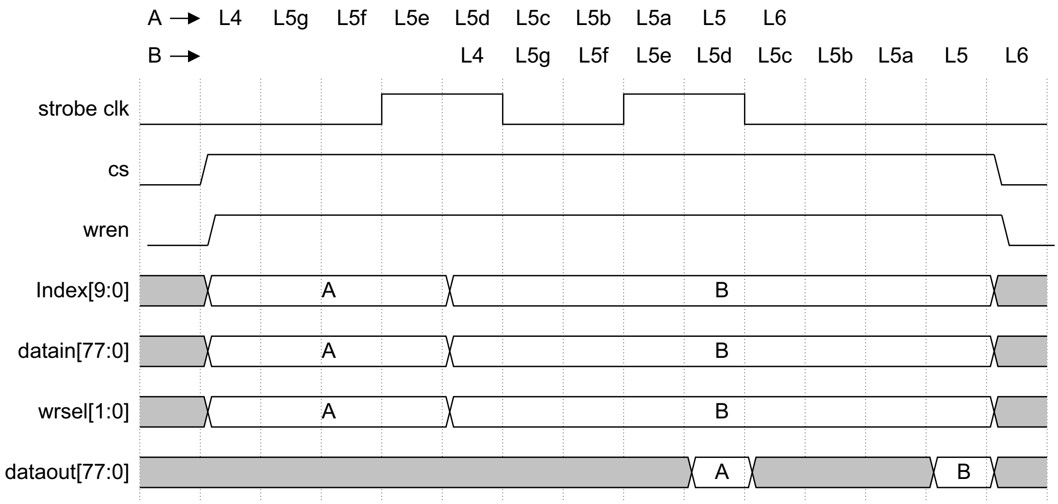


Figure 3-14 Back-to-back Data array accesses for 4-cycle data configuration with additional stages

3.5.4 L2 strobe configuration

The `` `ATL_STRETCH_L2RAMCLK ` ``define in the `atl_level2/verilog/atl_l2_clk_ram.v` file configures the L2 RAM strobe clock logic to generate either a system **CLK** frequency pulse or a ~50% duty cycle clock to drive the L2 RAMs, depending on your RAM clock requirements. Defining `` `ATL_STRETCH_L2RAMCLK `` results in the ~50% duty cycle waveform, and not defining it results in a system **CLK** phase pulse generated by a clock gate, ICG. Figure 3-15 shows the strobe clock generation logic contained in the `atl_level2/verilog/atl_l2_clk_ram.v` file.

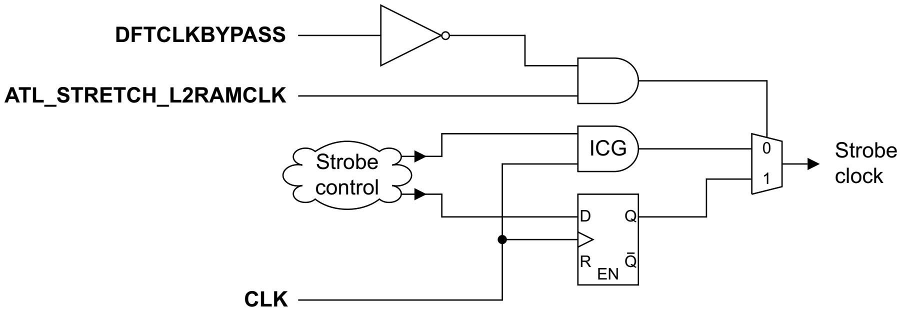


Figure 3-15 L2 RAM strobe clock generation

Another requirement of the RAM strobe clock is the ability to be controlled directly by the system **CLK** input for DFT purposes. This is done through the ICG path, that is forced by the **DFTCLKBYPASS** signal as Figure 3-15 shows.

When `` `ATL_STRETCH_L2RAMCLK `` is not defined, the strobe clock is generated from the ICG clocked by **CLK** resulting in a **CLK** phase pulse. When `` `ATL_STRETCH_L2RAMCLK `` is defined, the strobe clock is generated by the flop output and stretches the strobe clock to 1 or more system **CLK** periods, depending on the latency programming.

You can choose to define or not define `` `ATL_STRETCH_L2RAMCLK `` in the `atl_level2/verilog/atl_l2_clk_ram.v` file, depending on your RAM requirements. By default, `` `ATL_STRETCH_L2RAMCLK `` is not defined.

Note

If `ATL_STRETCH_L2RAMCLK is defined, a 50% duty cycle clock is generated only if the L2 RAM latency is an even number. If the L2 RAM latency is an odd number, for example 3, then the duty cycle of the clock is not symmetric. See Figure 3-16 and Figure 3-17 on page 3-20.

Table 3-14 shows that although the strobe clock target duty cycle is 50%, actual cycles for Tag array accesses with odd numbered latency configurations are less than 50%.

Table 3-14 Tag array access cycles

Tag latency cycles	Cycles CLK active
2	1
3	1
4	2
5	2

Table 3-15 shows that although the strobe clock target duty cycle is 50%, actual cycles for Data array accesses with odd numbered latency configurations are less than 50%.

Table 3-15 Data array access cycles

Data latency cycles	Cycles CLK active
2	1
3	1
4	2
5	2
6	3
7	3
8	4

Figure 3-16 shows the strobe clock behavior for both configurations of `ATL_STRETCH_L2RAMCLK, with a latency of 4.

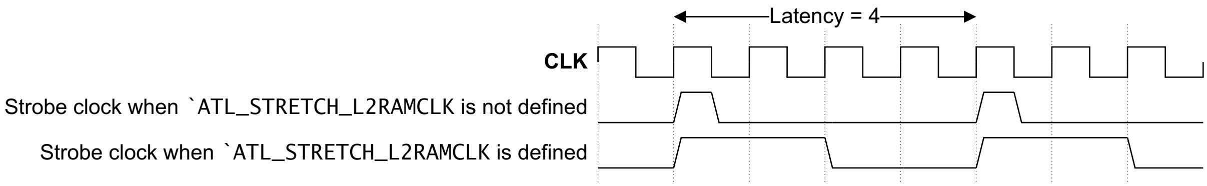

Latency = 4
CLK
Strobe clock when `ATL_STRETCH_L2RAMCLK is not defined
Strobe clock when `ATL_STRETCH_L2RAMCLK is defined

Figure 3-16 L2 RAM strobe clock configurations

Figure 3-17 on page 3-20 shows the strobe clock behavior for both configurations of `ATL_STRETCH_L2RAMCLK, with a latency of 3.

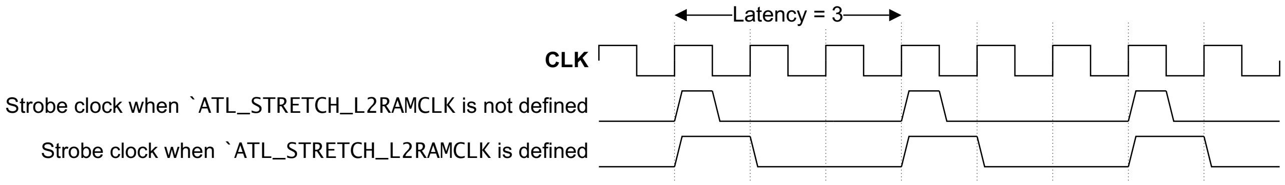

Latency = 3
CLK
Strobe clock when `ATL_STRETCH_L2RAMCLK is not defined
Strobe clock when `ATL_STRETCH_L2RAMCLK is defined

Figure 3-17 L2 RAM strobe clock with latency of 3

3.6 Flow for memory integration

Figure 3-18 shows the supported RAM integration flow for the Cortex-A57 MPCore multiprocessor. This integration flow enables you to:

- Identify the RAM organization required for each RAM block, see *atl_cpu RAM organization* on page 3-6 and *atl_noncpu, shared L2, RAM organization* on page 3-10.
- Integrate your library RAM, see *Flow for memory integration*.
- Test that RAM integration is successful, see *Confirmation of memory integration* on page 3-27.

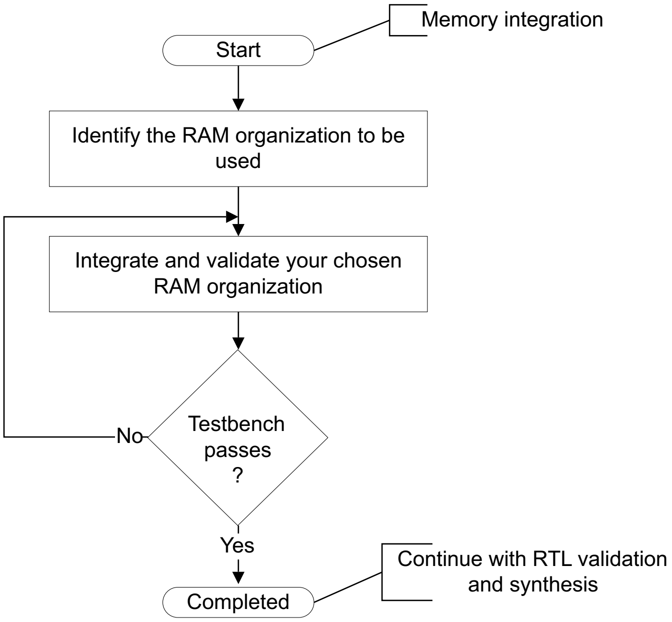

Memory integration
Start
Identify the RAM organization to be used
Integrate and validate your chosen RAM organization
Testbench passes ?
No
Yes
Completed
Continue with RTL validation and synthesis

Figure 3-18 RAM integration flow

RAM integration is the process of substituting a physical RAM for the existing generic `atl_sram` model. You must edit the RAM wrapper module to accommodate your RAM instance, and add any necessary glue logic to support interface differences between the generic `atl_sram` model and your physical RAM.

3.6.1 RAM wrapper module

All RAM wrapper modules are located in the same directory, `models/rams/generic`. After RTL configuration, the RAM wrapper modules are sized and named appropriately in a single directory. Each RAM wrapper model contains other minimal logic in addition to the RAM instantiation. To save power, the clock to an inactive RAM is gated. Each RAM module contains an instantiation of either an `atl_ck_gate` or an `atl_l2_clk_ram` module that serves as the clock gate for the RAM. There is no requirement to modify this clock gate instantiation. Standard synthesis maps this into an ICG cell in identical fashion to standard logic clock gating. The RAM chip select is used to inhibit RAM accesses during production scan testing. During normal operation it is held enabled with the clock gating serving as the RAM access control. ARM recommends that you do not modify the clock gate or chip select. The RAM wrapper module also contains *SystemVerilog Assertions* (SVA) that check for unknown (X) values on the inputs to the RAMs. ARM recommends that you do not modify these assertions.

For more information see Chapter 6 *Design For Test*.

3.6.2 Legal RAM organization modifications

Your physical RAM library might contain RAMs that exactly match the row and column organization or write granularity you require. More often, the physical RAMs available do not match the floorplan you require, or do not meet the timing specifications. You can substitute alternate RAMs or split a single logical RAM into multiple physical RAM instances, as long as the resultant aggregate RAM module behaves identically in both normal operation and production test modes. You can use the following potential modifications:

Row splitting

You can split a RAM with a large number of rows into multiple RAMs if the number of rows cannot be supported. You are responsible for adding the proper multiplexers to combine the data outputs of the multiple RAMs. Be aware that the RAM data outputs are delayed one cycle from the RAM control inputs. If timing permits, you can duplicate the clock gate and qualify the chip select input to each clock gate with upper address bits to minimize power consumption.

Column splitting

You can split a RAM with a large number of columns into multiple RAMs if the memory width cannot be supported.

Bit enable splitting

You can implement a RAM with a write granularity that is not supported by your physical RAM with a bit-writable RAM. You must implement the write enable and mask correctly.

The Load/Store Data RAM, `atl_if_data_ram`, is the only RAM type that explicitly contains logic to support column splitting with unique chip selects. The 256-bit wide RAM can be split into four 64-bit wide RAMs. Only the `atl_if_data_ram` contains four sub-chip selects to individually generate clock enables to each split RAM instance. Because most load and store operations are 64-bit wide, this permits only one of the four potentially split RAMs to be accessed for 64-bit, or smaller, load or store operations. See *Sub-chip selects* on page 3-25 for more information.

3.6.3 Generic `atl_sram` model

A single `atl_sram` model can represent all of the RAM sizes and write enable requirements. The `atl_sram` model uses five parameters that are used in the following order:

`ROWS` This is the number of rows, or uniquely addressable words, in the RAM. This number is always a power of 2.

`ADDR_WIDTH` This is the number of bits required to address the rows. This number is always equal to log2 `ROWS`.

`COLS` This is the number of columns, or bits read out per row, of the RAM.

`WR_MASK_TYPE` This is the write granularity, or the number of bits within a row that can be written independently.

If `WR_MASK_TYPE=0`, the RAM is word-writable, the entire row is written.

If `WR_MASK_TYPE=1`, the RAM is bit-writable, this means you can write each bit independently.

Any other value of `WR_MASK_TYPE` indicates that the RAM has a write granularity equal to this value. That is, you can control the writing of each contiguous group of `WR_MASK_TYPE` bits independently.

WR_MASK_WIDTH

This is the width of the write control mask that controls the bytes or bits within the RAM the system can write to. If WR_MASK_TYPE is not 0, this is always equal to COLS/WR_MASK_TYPE. If WR_MASK_TYPE is 0, this is undefined, and no **write_mask** signal is present on the RAM.

Example 3-1 shows an instantiation for the D-cache Data array.

Example 3-1 Example instantiation

```
atl_sram #(256, 8, 312, 39, 8) usram (
    .data_out(data_out[311:0]),
    .addr(addr[7:0]),
    .data_in(data_in[311:0]),
    .chip_select(cs),
    .write_enable(write_enable),
    .write_mask(write_mask[7:0]),
    .clk(clk_sram),
    .global_reset(1'b0)
);
```

In addition to parameters, the atl_sram model contains various input and output signals. Although these signals are named in a generic fashion, they usually map to a physical RAM signal. Table 3-16 shows the signals on the generic atl_sram model.

Table 3-16 Generic atl_sram model input and output signals

Name	I/O	Width	Description
data_out	O	COLS	Read data output.
addr	I	ADDR_WIDTH	Address for reads and writes.
data_in	I	COLS	Write data input.
chip_select	I	1	Master chip select: • If LOW, neither a read nor write operation occurs. • If HIGH, either read or write is completed depending on **write_enable**.
write_enable	I	1	Master write enable: If HIGH, and **chip_select** is HIGH, writes to the RAM are enabled.
write_mask	I	WR_MASK_WIDTH	This signal controls the granules that are written to. Each bit i enables writing of the bit range [WR_MASK_WIDTH×(i+1)–1, WR_MASK_WIDTH×i]. Ignored if **chip_select** and **write_enable** are not both active. If WR_MASK_WIDTH=1, this bit is set to a constant 1.
clk	I	1	Clock for synchronous RAM.
global_reset	I	1	Only used for formal verification, has no functional use.

All instantiated RAMs must be synchronous with all inputs set up to the rising edge of the clock, and **data_out** driven following the rising edge of the clock.

The **write_mask** input is not sufficient for full write control. It is only valid when **write_enable** is active. For those RAMs that do not have a master write enable, but have a single per-bit or per-partial word write enable, you must AND together the **write_mask** bus and **write_enable** signals:

```
assign write_enable_mask[31:0] = write_mask[31:0] & {32{write_enable}};
```

If the RAM is word-writable, the **write_mask** is redundant because only **write_enable** is required. The RAM wrappers tie **write_mask** HIGH. If you substitute a physical RAM, this signal is normally deleted.

The **global_reset** signal is only for use in formal verification of custom implementations. This signal is not part of the functionality of the RAM.

RAM models are not required to implement any clearing or preset function. The **global_reset** signal does not clear the RAM to a 0, but to an unknown value.

RAM integration example

This is an example of a RAM integration. It shows how the `atl_sram` instantiation is modified in RAM integration.

In Example 3-2 on page 3-25, the `atl_sram` instance is substituted with a physical RAM. The physical RAM integrated in this example is a commercially available, single-port memory.

The RAM differs from the generic `atl_sram` model in that **chip_select** and **write_enable** are active LOW.

This example assumes that the RAM cannot implement a 312 column wide RAM. In this example world, the RAM compiler only generates up to 128-bit wide RAMs. Therefore, you must split the columns of the logical RAM into multiple smaller physical RAMs. One method of splitting is to subdivide the 312 columns into multiples of the write granularity, that is 39 bits. The 256×312 RAM can be split up into four identical 256×78-bit RAMs.

To split the RAM by columns, the RAM write data and read data buses must be split. Because the entire logical RAM is accessed as one, the chip select is identical for all four RAMs, as is the index. This is the ordinary case. However, the Load/Store Data RAM contains sub-chip selects that allow the logical 256×312 RAM, or the corresponding 256×256 non-ECC version, to be split into 4 physical RAM instantiations with unique clock enables. See *Sub-chip selects* on page 3-25 for more information.

The RAM differs from the `atl_sram` model in a few ways that require the addition of glue logic. The first difference is the RAM module has an active-LOW chip select pin. The `atl_sram` model chip select pin is active-HIGH, and must be inverted.

The RAM chip select pin is only used during production test, and the external chip select pin that enables accesses is only used during production test. The internal chip select signal **cs** must be inverted, and the inverted signal connected to the substituted RAMs. No modification to the gated clock is required.

```
assign cs = ~ DFTRAMHOLD;
assign chip_select_n = ~ cs;
```

The second difference is the RAM only has a single active-LOW write enable bus. This requires combination of the **write_mask** bus and **write_enable** signals, as *Generic atl_sram model* on page 3-22 describes. For this example, the **write_enable_mask_n[7:0]** is:

```
assign write_enable_mask_n[7:0] = ~ (write_mask[7:0] & {8{write_enable}});
```

You must substitute your physical RAM pins and instance for the generic pins and instance to complete the RAM integration.

Example 3-2 on page 3-25 shows the four RAMs that you must substitute for the `atl_sram` instance.

Example 3-2 Substitution for atl_sram instance

```
SRAM256x78WE39 usram0 (
   .CLK(clk_sram),
   .Q(data_out[77:0]),
   .D(data_in[77:0]),
   .WEN(write_enable_mask_n[1:0]),
   .CEN(chip_select_n),
   .A(addr[7:0]));

SRAM256x78WE39 usram1(
   .CLK(clk_sram),
   .Q(data_out[155:78]),
   .D(data_in[155:78]),
   .WEN(write_enable_mask_n[3:2]),
   .CEN(chip_select_n),
   .A(addr[7:0]));

SRAM256x78WE39 usram2 (
   .CLK(clk_sram),
   .Q(data_out[233:156]),
   .D(data_in[233:156]),
   .WEN(write_enable_mask_n[5:4]),
   .CEN(chip_select_n),
   .A(addr[7:0]));

SRAM256x78WE39 usram3(
   .CLK(clk_sram),
   .Q(data_out[311:234]),
   .D(data_in[311:234]),
   .WEN(write_enable_mask_n[7:6]),
   .CEN(chip_select_n),
   .A(addr[7:0]));
```

Sub-chip selects

In the Load/Store Unit, the data cache Data array contains placeholder logic that provides the option to split the 256/312-bit wide logical RAM instance into four 64/78-bit wide instances. The majority of load and store operations are 64-bit or less, so this can result in power savings because there are fewer total RAM accesses.

```
assign chip_select = |dw_select[3:0];
assign en = chip_select;
atl_ck_gate usram_clk(.gated_clk(clk_sram), .clk(clk), .clk_enable(en), .se(dftse_cpu));
```

The OR gate can be removed, and the clock gate instantiated four times to provide four unique gated clocks:

```
atl_ck_gate usram_clk0(.gated_clk(clk_sram_0), .clk(clk), .clk_enable(dw_select[0]), .se(dftse_cpu));
atl_ck_gate usram_clk1(.gated_clk(clk_sram_1), .clk(clk), .clk_enable(dw_select[1]), .se(dftse_cpu));
atl_ck_gate usram_clk2(.gated_clk(clk_sram_2), .clk(clk), .clk_enable(dw_select[2]), .se(dftse_cpu));
atl_ck_gate usram_clk3(.gated_clk(clk_sram_3), .clk(clk), .clk_enable(dw_select[3]), .se(dftse_cpu));
```

Each of the four instances created by the column split of the logical RAM can then have their own gated clock, as Example 3-3 on page 3-26 shows.

Example 3-3 Substitution for Data array `atl_sram` instance with sub-chip selects

```
SRAM256x78WE39 usram0 (
.CLK(clk_sram_0),
.Q(data_out[77:0]),
.D(data_in[77:0]),
.WEN(write_enable_mask_n[1:0]),
.CEN(chip_select_n),
.A(addr[7:0]));

SRAM256x78WE39 usram1(
.CLK(clk_sram_1),
.Q(data_out[155:78]),
.D(data_in[155:78]),
.WEN(write_enable_mask_n[3:2]),
.CEN(chip_select_n),
.A(addr[7:0]));

SRAM256x78WE39 usram2 (
.CLK(clk_sram_2),
.Q(data_out[233:156]),
.D(data_in[233:156]),
.WEN(write_enable_mask_n[5:4]),
.CEN(chip_select_n),
.A(addr[7:0]));

SRAM256x78WE39 usram3(
.CLK(clk_sram_3),
.Q(data_out[311:234]),
.D(data_in[311:234]),
.WEN(write_enable_mask_n[7:6]),
.CEN(chip_select_n);
.A(addr[7:0]));
```

3.7 Confirmation of memory integration

This section describes how to validate your memory integration.

Note

You must run the RAM validation testbench described in *RAM integration testbench* as part of the sign-off criteria.

3.7.1 RAM integration testbench

ARM provides a testbench that checks that your RAM blocks are correctly integrated. Before running the testbench you must have successfully configured your RTL with the `rtl_configure.pl` script and integrated your RAM models.

The RAM integration testbench script is `config_ram_tb.pl` and is located in the *`outdir`*`/shared/tools/bin` directory. *`outdir`* is the output directory specified by the `-outdir <dir>` option in the `rtl_configure.pl` configuration script. See *Configuration script* on page 2-3.

The RAM integration testbench script requires that Perl is installed in the execution path.

3.7.2 Running the RAM integration testbench script

To run the RAM integration testbench script:

1. Navigate to the *`outdir`*`/testbench/ram_integration_tb/sim` directory.

   Note

   `outdir` is the output directory specified by the `-outdir <dir>` option in the `rtl_configure.pl` configuration script.

2. Edit the `ram_tb.vc` file and change the `//PATH TO YOUR RAMS` lines to point to the location of your RAMs. If no single directory contains all the RAMs and modules that are specified in the `` `include or explicitly call out `` section, you must add additional lines to include all the directories for your RAMs and any modules inside those RAMs.

3. Format the script options that match your configured RTL.

4. Run the RAM integration testbench script, `config_ram_tb.pl`. The script is in the *`outdir`*`/shared/tools/bin` directory.

5. Check the simulation log file for error messages. The log file is located in the *`outdir`*`/testbench/ram_integration_tb/sim` directory.

3.7.3 Script synopsis

The format of the script is:

```
%> config_ram_tb.pl -dir <dir> -l2size <size> {-ecc <option>}
{-l2tagslice <option>} {-l2datslice <option>} {-l2tagsetup <option>}
{-l2datsetup <option>} {-l2taglatency <number>} {-l2datlatency <number>}
{-stretch_l2ramclk < option>} {-rams <options>} {-sim <option>}
{-ignore <"text_string">} {-plusargs <“string_to_enable_vcd_generation”>}
```

The following command displays the script functionality and options:

```
%> config_ram_tb.pl -help
```

The RAM integration testbench script can take the following options:

`-dir <dir>` The absolute path to the directory of the configured RTL with the integrated RAMs. This is usually the output directory specified by the `-outdir <dir>` option in the `rtl_configure.pl` configuration script. See *Script synopsis* on page 2-4.

Note

This parameter must be specified.

`-l2size <size>`

Specifies the size of the L2 cache. The values for this option can be 512, 1024, or 2048.

Note

This parameter must be specified and must match your configured RTL.

`{-ecc <option>}`

Specifies the ECC/Parity support in the L1 caches. The values for this option can be:

`L1andL2` Both L1 and L2 have ECC/Parity support.

`L2only` Only L2 has ECC/Parity support and the L1 does not have ECC/Parity support.

This option must match your configured RTL. If you do not specify this option, both L1 and L2 have ECC/Parity support.

`{-l2tagslice <option>}`

Permits the simulator to adjust the timing for a register slice in the L2 Tag, Snoop Tag, Dirty, and PSQ RAMs. The value of this option can be:

0 A register slice is not implemented in the L2 Tag, Snoop Tag, Dirty, and PSQ RAMs.

1 One register slice is implemented in the L2 Tag, Snoop Tag, Dirty, and PSQ RAMs.

This option must match your configured RTL and the value in L2CTLR[12]. If you do not specify this option, the simulator does not expect a register slice in the L2 Tag RAMs.

`{-l2datslice <option>}`

Permits the simulator to adjust the timing for a register slices in the L2 Data RAMs. The value of this option can be:

0 Register slices are not implemented in the L2 Data RAMs.

1 One register slice is implemented in the L2 Data RAMs.

2 Two register slices are implemented in the L2 Data RAMs.

This option must match your configured RTL and the value in L2CTLR[11:10]. If you do not specify this option, the simulator does not expect register slices in the L2 Data RAMs.

`{-l2tagsetup <option>}`

Specifies whether the L2 Tag RAM inputs are held for one additional clock cycle. The value of this option can be:

0 The L2 Tag RAM inputs are not held for one additional clock cycle.

1 The L2 Tag RAM inputs are held for one additional clock cycle.

This option must match the value in L2CTLR[9]. If you do not specify this option, the L2 Tag RAM setup is 0.

`{-l2datsetup <option>}`

Specifies whether the L2 Data RAM inputs are held for one additional clock cycle. The value of this option can be:

0 The L2 Data RAM inputs are not held for one additional clock cycle.

1 The L2 Data RAM inputs are held for one additional clock cycle.

This option must match the value in L2CTLR[5]. If you do not specify this option, the L2 Data RAM setup is 0.

`{-l2taglatency <option>}`

Specifies the latency for the L2 Tag RAMs. The latency is the number of clock cycles that non-control inputs are held stable. The values for this option are 2, 3, 4, or 5.

This option must match the value in L2CTLR[8:6]. If you do not specify this option, the L2 Tag RAM latency is 2 clock cycles.

`{-l2datlatency <option>}`

Specifies the latency for the L2 Data RAMs. The latency is the number of clock cycles that non-control inputs are held stable. The values for this option are 2, 3, 4, 5, 6, 7, or 8.

This option must match the value in L2CTLR[2:0]. If you do not specify this option, the L2 Data RAM latency is 2 clock cycles.

`{-stretch_l2ramclk <option>}`

Specifies whether the stretch clock for L2 Tag and Data RAMs is used. The values for this option can be:

0 The stretch clock is not used for the L2 Tag and Data RAMs.

1 The stretch clock is used for the L2 Tag and Data RAMs.

This option must match your configured RTL. If you do not specify this option, the stretch clock is not used for the L2 Tag and Data RAMs.

`{-rams <options>}` Specifies the RAMs to test. The values for this option can be:

`L2` Tests all L2 RAMs.

`L2_TLB` Test only the TLB RAM.

`L2_DATA` Test only the L2 Data RAM.

`L2_TAG` Test only the L2 Tag RAM.

`L2_PSQ` Test only the L2 Prefetch Stride Queue RAM.

`L2_DIRTY` Test only the Dirty RAM.

`L2_SNP_TAG` Test only the L2 Snoop Tag RAM.

`LS` Tests all Load/Store RAMs.

`LS_DATA` Test only the Load/Store Data RAM.

`LS_TAG` Test only the Load/Store Tag RAM.

`IF` Tests all Instruction Fetch RAMs.

`IF_IP` Test only the Instruction Fetch Indirect Predictor RAM.

`IF_BTB` Test only the Instruction Fetch Branch Target Buffer RAM.

`IF_GHB` Test only the Instruction Fetch Global History Buffer RAM.

`IF_TAG` Test only the Instruction Fetch Tag RAM.

`IF_DATA` Test only the Instruction Fetch Data RAM.

If you do not specify this option, all RAMs are tested.

`{-sim <option>}`

Specifies the simulator to run the RAM integration testbench script. The values for this option can be:

`mti` Selects the Mentor QuestaSim simulator.

`ius` Selects the Cadence Incisive simulator.

`vcs` Selects the Synopsys VCS simulator.

If you do not specify this option, the simulator is `mti`.

`{-ignore <“text_string”>}`

An optional parameter that specifies the text string to ignore when parsing the log file for errors. This is useful to prevent invalid error entries in the log file.

For example, the option `-ignore “error_injection”` prevents writing any line containing that syntax to the log file.

Multiple text strings can be passed by delimiting them with commas. For example `-ignore “error_injection,parity_error_check”`.

`{-plusargs <“string_to_enable_vcd_generation”>}`

An optional parameter that specifies additional arguments to pass to the following commands:

- `vlog` or `vsim`, for the Mentor simulator.
- `irun`, for the Cadence simulator.
- `vcs`, for the Synopsys simulator.

This argument is typically used to enable VCD generation. For example:

mti
```
config_ram_tb.pl -dir <full_path_to_configured_rtl> \
-l2size 512 -sim mti -plusargs "' +acc +vcd_on'"
```

ius
```
config_ram_tb.pl -dir <full_path_to_configured_rtl> \
-l2size 512 -sim ius -plusargs "' +access+r +vcd_on'"
```

vcs
```
config_ram_tb.pl -dir <full_path_to_configured_rtl> \
-l2size 512 -sim vcs -plusargs "' +vcd_on'"
```

3.7.4 Script examples

Example 3-4 shows that a successful RAM testbench execution prints all RAMs have passed.

Example 3-4 Successful RAM integration testbench execution

```
===========================================
TEST SUMMARY FOR:
L2 SIZE:            2048
ECC:                L1andL2
L2 DATA SLICE:      0
L2 DATA SETUP:      1
L2 DATA LATENCY:    4
L2 TAG SLICE:       0
L2 TAG SETUP:       0
L2 TAG LATENCY:     2
L2 STRETCH RAM CLK: 0
SIMULATOR:          mti
===========================================
L2_TLB_RAM                    :NO MISMATCHES
```

```
IF_IP_RAM                     :NO MISMATCHES
IF_BTB_RAM                    :NO MISMATCHES
IF_GHB_RAM                    :NO MISMATCHES
IF_TAG_RAM                    :NO MISMATCHES
IF_DATA_RAM                   :NO MISMATCHES
LS_TAG_ECC_RAM                :NO MISMATCHES
LS_DATA_ECC_RAM               :NO MISMATCHES
L2_DATA_RAM                   :NO MISMATCHES
L2_TAG_RAM                    :NO MISMATCHES
L2_DIRTY_RAM                  :NO MISMATCHES
L2_SNP_TAG_RAM                :NO MISMATCHES
L2_PSQ_RAM                    :NO MISMATCHES
TESTS PASSED: 13, FAILED: 0, WARNINGS: 0
```

Example 3-5 shows that an unsuccessful RAM testbench execution stops at the first RAM that fails.

Example 3-5 Unsuccessful RAM integration testbench execution

```
TEST FAILED FOR L2_DIRTY_RAM; parsed 55 lines

===========================================
TEST SUMMARY FOR:
L2 SIZE:            2048
ECC:                L1andL2
L2 DATA SLICE:      0
L2 DATA SETUP:      1
L2 DATA LATENCY:    4
L2 TAG SLICE:       0
L2 TAG SETUP:       0
L2 TAG LATENCY:     2
L2 STRETCH RAM CLK: 0
SIMULATOR:          mti
===========================================
L2_TLB_RAM                    :NO MISMATCHES
IF_IP_RAM                     :NO MISMATCHES
IF_BTB_RAM                    :NO MISMATCHES
IF_GHB_RAM                    :NO MISMATCHES
IF_TAG_RAM                    :NO MISMATCHES
IF_DATA_RAM                   :NO MISMATCHES
LS_TAG_ECC_RAM                :NO MISMATCHES
LS_DATA_ECC_RAM               :NO MISMATCHES
L2_DATA_RAM                   :NO MISMATCHES
L2_TAG_RAM                    :NO MISMATCHES
L2_DIRTY_RAM                  :# ** Error: At time 4162000 for read to addr 9
expected data 1032dcfe98ba1032dcfe98ba1032dcfe98ba1032dcfe98ba but got data
9032dcfe98ba1032dcfe98ba1032dcfe98ba1032dcfe98ba data_mask is
ffffffffffffffffffffffffffffffffffffffffffffffff
TESTS PASSED: 10, FAILED: 1, WARNINGS: 0
```

3.8 Outputs from memory integration

The main output from memory integration is the configured RTL and the RAM integration testbench results required for sign-off. See *Requirements for sign-off* on page 9-3.

3.9 Solving memory integration problems

If the outputs from the memory integration testbench indicate a failure, you have not integrated the memory correctly. If you cannot identify the reason for the failure, contact ARM for more information.

Chapter 4
Checking your RTL

This chapter describes how to check the delivered Cortex-A57 MPCore multiprocessor RTL. Using the execution testbench and the supplied test vectors enables you to determine whether the deliverables are set up correctly. You can also use these tests to check your RTL configuration. It contains the following sections:

4.1 About checking your RTL

This chapter describes using the execution testbench and the supplied test vectors to provide limited validation of your RTL. ARM recommends that you use the execution testbench and the supplied test vectors to validate your configured RTL.

Note

This chapter describes how to check your configuration of the RTL. It does not validate your synthesized RTL which must pass:

- Logical equivalence verification.
- Timing verification.

See the implementation methodology documents for information about these processes.

Using the supplied simulation environment, you can compile and run code on your configured RTL, while capturing the vectors, or VCDs. ARM supplies tests, as *Executable and Linkable Format* (ELF) files, source files and Makefiles, that you can run in the simulation environment. You can also write your own tests.

If the source code of a test is modified, you must rebuild and install them so the new test can be run. To build new ELF files from source, and install them in the run directory, see *Rebuild the tests* on page 4-7.

Figure 4-1 shows the overview of the RTL validation process.

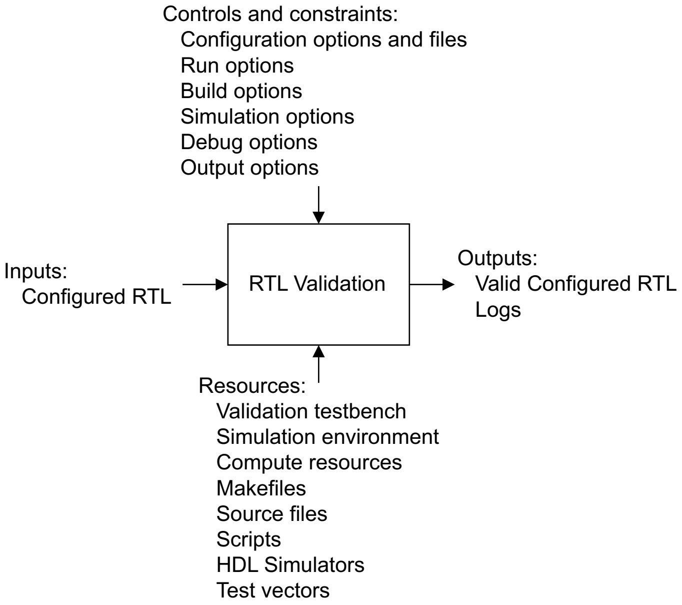


Figure 4-1 RTL validation process

4.2 Resource requirements for checking your RTL

You require the execution testbench to check your RTL. Depending on your implementation and Tarmac requirements you might also require the Universal Verification Methodology and the Protocol Buffers resources.

The following sections describe the resources for checking your RTL:

- *Execution testbench*.
- *Protocol Buffers* on page 4-4.
- *Universal Verification Methodology* on page 4-5.

4.2.1 Execution testbench

ARM provides an execution testbench that checks your RTL. Before running the testbench you must have successfully configured your RTL with the `rtl_configure.pl` script, integrated your RAM models, and verified the memory integration.

The execution testbench adds the following memory-mapped devices on the multiprocessor AXI bus:

TUBE The tube is located at address `0x13000000`.

Characters written to this address are echoed to the screen during simulation. Characters are buffered with one buffer per processor, and displayed when a CR or LF is received. Even if characters are written by different processors interleaved in time, the messages themselves are displayed coherently.

If a ctrl-d (`0x4`) is written to this address, the simulation terminates.

FIQ counter

A write to address `0x13000008` loads the **FIQ** counter.

After loading, the counter decrements on every **CLK** cycle. When the counter reaches 0, the logic asserts **nFIQ**.

nFIQ remains asserted until the processor writes to address `0x13000010`. **nFIQ** deasserts when the write occurs. The value written is not important.

Included with the execution testbench are the source code and test vector files for the functional, and power tests, and a simple `hello_world` program. For more information about the checking tests, see *Inputs for checking your RTL* on page 4-7.

You must use the execution testbench compile command for the bus protocol in your design:

ACE The testbench file is `execution_tb.vc`.

CHI The testbench file is `execution_tb_skyros.vc`.

The execution testbench files are located in the *`outdir`*`/testbench/execution_tb/` directory and must be present to check the configured RTL.

Note

`outdir` is the output directory specified by the `-outdir <dir>` option in the `rtl_configure.pl` configuration script. See *Script synopsis* on page 2-4.

4.2.2 Protocol Buffers

The Protocol Buffers are only required if you require the model to support Tarmac trace.

Protocol Buffers encode structured data in an efficient yet extensible format for use with the Tarmac instruction trace. Protocol compilers for C++, Java, and Python are available to the public under a free software, open source license.

The Protocol Buffers are available to download at Google,
https://developers.google.com/protocol-buffers/

Table 4-1 shows the Protocol Buffers requirement for your simulator type and machine operating system.

Table 4-1 Simulator, OS, and Protocol Buffers requirements

Simulator	32-bit OS	64-bit OS
32-bit	32-bit Protocol Buffers	32-bit Protocol Buffers
64-bit	-	64-bit Protocol Buffers

To install Protocol Buffers in your simulation environment:

1. Enter the commands:

```
> mkdir protobuf
> cd protobuf
> mv ../protobuf-version.tar.gz .
> tar -xzvf protobuf-version.tar.gz .
> cd protobuf-version
```

Note

Replace `version` with the Protocol Buffers version number that the *ARM® Cortex®-A57 MPCore™ Processor Release Note* recommends to use.

2. Depending on whether your simulator is 32-bit or 64-bit, you configure `protobuf` using the command:

32-bit
```
> ./configure CXX=”g++ -m32” \
{--prefix=<path_to_protobuf_installation_directory>}
```

64-bit
```
> ./configure CXX=”g++ -m64” \
{--prefix=<path_to_protobuf_installation_directory>}
```

Note

`--prefix=<path_to_protobuf_installation_directory>` is enclosed by curly braces because it is optional. If not specified, the prepend path is `/usr/local/`.

3. Enter the commands:

```
> make
> make check
> make install
> setenv LD_LIBRARY_PATH \
<path_to_prepend_to_the_makefile_commands>/lib:${LD_LIBRARY_PATH}
```

Note

Installing Protocol Buffers might require assistance from your system administrator.

4.2.3 Universal Verification Methodology

If your multiprocessor implements a CHI interface then you require the *Universal Verification Methodology* (UVM) Class Library Code, to run the execution testbench. UVM is available to download from Accellera Systems Initiative, `http://www.accellera.org`.

The location of the UVM Class Library Code is expected to be:

```
> +incdir+testbench/shared/globalshared/uvm-version/src
> -y      testbench/shared/globalshared/uvm-version/src/uvm.sv
```

Note

`version` is the UVM version number that the *ARM® Cortex®-A57 MPCore™ Processor Release Note* recommends to use.

You can download UVM anywhere in your directory structure, but if you download it to a directory different from `testbench/shared/globalshared/uvm-`*`version`*, you must edit the UVM path so that the location of the UVM library is correct in:

- The `testbench/execution_tb/sim/execution_tb_skyros.vc` file.
- The simulator compile commands in *Build commands to use when the UVM install differs from the default location* on page 4-12.

4.3 Controls and constraints for checking your RTL

During RTL validation you must:

- Ensure your simulator is in the execution path.
- Ensure that the ARM RVCT tools are in the execution path if you rebuild the ELF files for A32 tests.
- Ensure that the GCC compiler is in the execution path if you rebuild the ELF files for A64 tests.

4.4 Inputs for checking your RTL

Table 4-2 describes the tests for checking your RTL with A32 instructions.

Table 4-2 RTL-checking tests with A32 instructions

File	Description
`arm_ok`	Simple data processing test
`cache_miss`	Processor stalled waiting for data to be returned from memory
`dhrystone`	Exercises processor integer pipeline[a]
`hello_world`	Displays the text `hello, world`
`post_config_check`	Displays the configuration of the RTL
`wfi`	Minimal low-power state

[a] Everything is cached in L1.

Table 4-3 describes the tests for checking your RTL with A64 instructions.

Table 4-3 RTL-checking tests with A64 instructions

File	Description
`crypto64`	Exercises the Cryptography engine
`dhrystone64`	Exercises processor integer pipeline[a]
`hello_world64`	Displays the text `Hello, 64-bit World!`
`saxpy64`	Exercises processor floating-point pipeline[a]
`sustained_max_power64`	Maximum sustained power consumption
`sustained_max_power64_mp`	Maximum sustained power consumption for multiprocessor implementations

[a] Everything is cached in L1.

4.4.1 Rebuild the tests

The tests vector files are supplied with the testbench in the appropriate ELF format so you are not required to rebuild them from source. If the source code is modified, you must rebuild and install them so the new test can be run. To build new ELF test vector files from source, and install them in the run directory use these commands:

```
> setenv PROJ_LOGICAL {full_path_to_configured_rtl}
> cd ${PROJ_LOGICAL}/testbench/execution_tb/tests
> make
```

Individual ELF test vector files can also be built and cleaned using the following commands:

```
> make <testname>
> make clean_<testname>
```

These are the commands for all the tests defined in the top of the master `Makefile`.

If your configuration includes the Cryptography engine and you want to re-compile the `crypto64` test then you must edit the `Makefile` and change `crypto64 = no` to `crypto64 = yes`.

4.4.2 Power test measurement windows

Table 4-4 describes the measurement windows for the power tests.

Table 4-4 Power test measurement windows

Test name	Loop number	Cycle (start)	Cycle (stop)	Number of cycles	Instruction
cache_miss[a]	3	16691	16847	156	LDR r1,[r0,#0] to LDR r2,[r1,#0]
dhrystone	11	11441	11578	137	MUL r0,r7,r0
dhrystone64	11	14862	14992	130	SDIV w19,w27,w26
saxpy64	9	10490	10693	203	BL {pc}+{offset};0x800186C0[b]
wfi[c]	-	6875	7175	300	WFI
sustained_max_power64	5	19060	19260	200	SUBS x5,x5,#2 to MOV x6,x11
sustained_max_power64 LH[d]	-	16694	16703	9	-
sustained_max_power64 HL[e]	-	20033	20041	8	-
sustained_max_power64_mp	5	23636	23867	231	SUBS x5,x5,#2 to MOV x6,x11

a. Timing measured with testbench option +memory_latency=50 to simulate very slow external memory.

b. The offset changes per loop in the Tarmac file but the BL instruction and the 0x800186C0 are constant.

c. Calculated from cycle number for WFI instruction + 300 cycles. The number of cycles value is chosen to be inside the WFI low-power state window.

d. Maximum $^{di}/_{dt}$ for LOW to HIGH transitions. The value is observed at the start of the second iteration through the main loop.

e. Maximum $^{di}/_{dt}$ for HIGH to LOW transitions. The value is observed when exiting the main loop, at the end of the pattern.

Note

The information in Table 4-4 is valid for:

- The ELF files included in the deliverables. The information might not be valid if you recompile the ELF files.
- The 2-processor, 2MB L2, CHI implementation. For other configurations, the cycle counts are not valid. You can use the loop number and instruction data to find the appropriate loop times from the Tarmac file.

4.4.3 Runtime options for the execution testbench

A plusarg is a runtime option to the simulator. The execution testbench simulation has one required plusarg, and several optional plusargs. The required plusarg specifies the ELF test vector file to load and run. The optional plusargs control the dumping of signal values to a .vcd file and the display of a Tarmac instruction trace. Table 4-5 on page 4-9 shows the execution testbench argument descriptions.

Note

The TESTARGS command line argument enables you to pass plusargs to the Makefile. By default, the Makefile includes some of the plusargs that Table 4-5 on page 4-9 shows. Use the TESTARGS argument if you require any changes to the plusargs.

Table 4-5 Execution testbench options

plusarg	Required	Description
`+elfname=<testname>`	Yes	This plusarg specifies the test to load and run. This argument is required. If you do not supply it, the simulation halts with an error message. The load file must be an `elf` file. The `.axf` file produced by the ARM development tools is acceptable. The `testname` parameter can include a path component to run a test not located in the current directory.
`+refclk_period=<time>`	No	Sets the clock frequency of the simulation. The minimum clock period is 2ns, that is 500MHz.
`+pclk_enable=<value>`	No	When `<value>` is: **0** Disables **PCLKENDBG**. **1** Enables **PCLKENDBG**.
`+memory_latency=<value>`	No	Inserts the specified number, `<value>`, of wait states into every read transaction.

`+dumpon=<time>`
`+dumpoff=<time>`
	No	These plusargs create a `.vcd` file of the signals at the top three levels of the Cortex-A57 MPCore multiprocessor hierarchy.
		These arguments specify the time in nanoseconds to start and stop dumping signal data. If `dumpoff` is given, the simulation halts when the time is reached.
		When you use these plusargs, the file `dump.vcd` is created by the testbench and written to the current directory.

`+dumpall=<time>`
`+dumpalloff=<time>`
	No	These plusargs create a `.vcd` file of every signal in the simulation. These include the testbench and the entire Cortex-A57 MPCore multiprocessor.
		These arguments specify the time in nanoseconds to start and stop dumping signal data. If `dumpalloff` is given, the simulation halts when the time is reached.
		When you use these plusargs, the file `dumpall.vcd` is created by the testbench and written to the current directory.

`+mem_verbose`	No	This plusarg enables verbose logging from the memory model. Every byte read or written, including loading the `elf` file at the beginning of the simulation, is logged. This can generate a large log file.
`+arch64`	No[a]	Use this plusarg when you use 64-bit ELFs for ACE and CHI.
`+rvbarN=00080000000`	No[a]	This plusarg is the reset vector base address for processor *N*, where *N* is 0, 1, 2, or 3. A reset vector base address plusarg, `rvbarN=0x00080000000`, is required for each processor in your configuration when executing 64-bit ELFs.[b]
`+tarmacALL`	No	Enables Tarmac trace for instructions, memory accesses, events and the register file.[c] If you require Tarmac support then the `make` command must include `ENABLE_TARMAC=1`.
`+tarmac_prefix=<string>`	No	Prefix `<string>` to Tarmac filenames. Prefix a path to the Tarmac file.[c]

[a] This plusarg is required for the 64-bit ELFs.

[b] The reset vector base address plusarg value, `0x0008000000` is required for all processors in the A64 checking tests ARM supplies. If you create your own tests, you can use this plusarg to relocate the reset vector base address to the location your test requires.

[c] The Tarmac trace is a text file that contains information about executed instructions, register file updates, memory accesses, and exception events. See Appendix A *Tarmac Trace Description* for more information.

4.5 Flow for checking your RTL

The RTL validation methodology consists of running ELF test vector files. ARM supplies a series of tests with the Cortex-A57 MPCore multiprocessor deliverables. These tests demonstrate certain behaviors of the multiprocessor.

Figure 4-2 shows the RTL build and validate flow for a test.

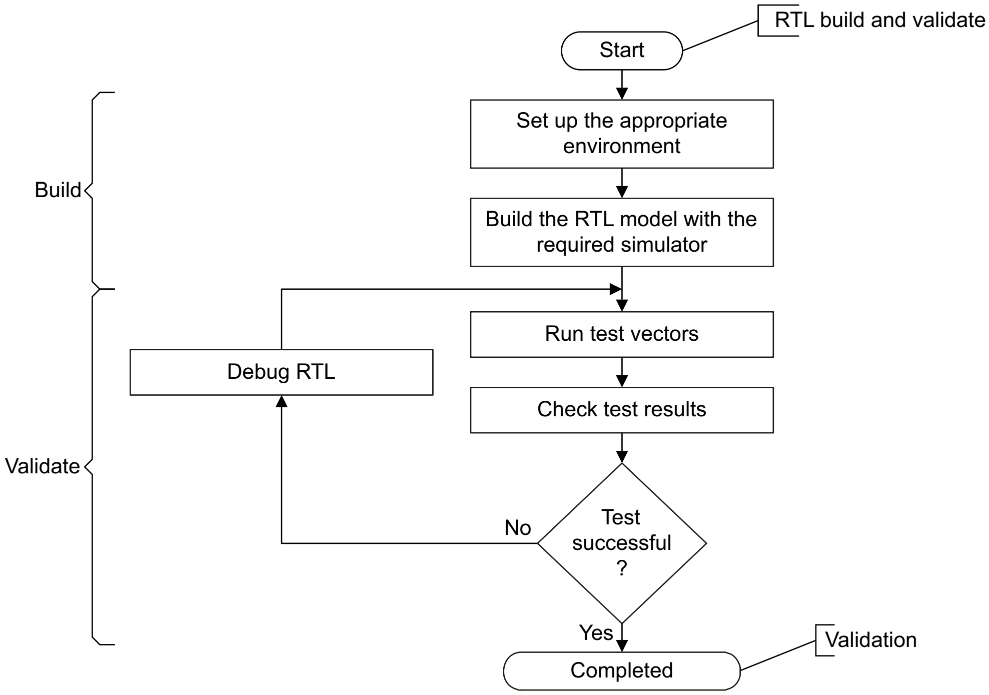


Figure 4-2 RTL build and validate flow

4.5.1 Compile and run examples for Verilog simulators

This section describes the process to build the model and run tests.

To check your RTL with the Verilog simulators you use the `Makefile`:

- Enter `make help` to display the command line arguments and their default values.

  Note

  — To run a test using a 64-bit ELF you must append `ELF64=1` to the `make` command.

  — The `Makefile` updates the `LD_LIBRARY_PATH` so that the trace shared library is found at simulation time.

- Table 4-5 on page 4-9 shows the runtime options you can use.

- Table 4-6 shows the {test_name} and {test_directory} parameters that you can use, and the expected test results.

Table 4-6 Test and directory names

Test name	Test directory	Test result
hello_world	hello_world	*hello_word test result* on page 4-14
hello_world64		*hello_word64 test result* on page 4-14
post_config_check	configuration_check/post_config_check	*post_config_check test result* on page 4-14
arm_ok	instruction_execution/arm_ok	*arm_ok test result* on page 4-15
cache_miss	power_indicative/cache_miss	*cache_miss test result* on page 4-15
crypto64	crypto64	*crypto64 test result* on page 4-15
dhrystone	power_indicative/dhrystone	*dhrystone test result* on page 4-15
dhrystone64	power_indicative/dhrystone	*dhrystone64 test result* on page 4-15
saxpy64	power_indicative/saxpy64	*saxpy64 test result* on page 4-15
wfi	power_indicative/wfi	*wfi test result* on page 4-15
sustained_max_power64	power_max/sustained_max_power64	*sustained_max_power64 test result* on page 4-15
sustained_max_power64_mp	power_max/sustained_max_power64_mp	*sustained_max_power64_mp test result* on page 4-16

The tests directory path is *outdir*/logical/testbench/execution_tb/tests/.

Note

outdir is the output directory specified by the -outdir <dir> option in the rtl_configure.pl configuration script. See *Script synopsis* on page 2-4.

4.5.2 Checking your RTL with a simulator

This section describes the commands to build the model, run tests, and create waveforms, on the Verilog simulators. It contains the following sections:

- *How to compile a build* on page 4-12.
- *How to run a test* on page 4-13.
- *How to dump waves from a time window* on page 4-14.

Note

You must run the commands from the *outdir*/logical/testbench/execution_tb/sim directory. *outdir* is the output directory specified by the -outdir <dir> option in the rtl_configure.pl configuration script. See *Script synopsis* on page 2-4.

How to compile a build

Table 4-7 lists the compile command to build a model that does not support Tarmac trace. The command you use depends on the simulator and whether the multiprocessor implements an ACE or CHI memory interface.

Table 4-7 Command to compile a build

Simulator		ACE memory interface	CHI memory interface
IUS	32-bit	`make SIM=ius BUS=ACE build`	`make SIM=ius BUS=Skyros build`[a]
	64-bit	`make SIM=ius BUS=ACE OS64=1 build`	`make SIM=ius BUS=Skyros OS64=1 build`[a]
QuestaSim	32-bit	`make SIM=mti BUS=ACE build`	`make SIM=mti BUS=Skyros build`[a]
	64-bit	`make SIM=mti BUS=ACE OS64=1 build`	`make SIM=mti BUS=Skyros OS64=1 build`[a]
VCS	32-bit	`make SIM=vcs BUS=ACE build`	`make SIM=vcs BUS=Skyros build`[a]
	64-bit	`make SIM=vcs BUS=ACE OS64=1 build`	`make SIM=vcs BUS=Skyros OS64=1 build`[a]

[a] Use this command if the UVM library install is in the default location otherwise see *Build commands to use when the UVM install differs from the default location*.

If your simulation environment includes support for the Protocol Buffers then you can generate a model that supports Tarmac trace by including the `ENABLE_TARMAC=1` option. For example:

- `make SIM=ius BUS=ACE ENABLE_TARMAC=1 build`
- `make SIM=vcs BUS=Skyros OS64=1 ENABLE_TARMAC=1 build`

Build commands to use when the UVM install differs from the default location

For a CHI implementation, the `Makefile` requires the UVM library and expects it to exist at `testbench/shared/globalshared/uvm-`*`version`*.

Note

`version` is the UVM version number that the *ARM® Cortex®-A57 MPCore™ Processor Release Note* recommends to use.

If your UVM install exists in a different folder then to compile a build for each simulator use the command:

IUS

```
make SIM=ius BUS=Skyros build \
UVM_PATH={absolute path to, and including, the uvm-version directory}
```

QuestaSim

```
make SIM=mti BUS=Skyros build \
UVM_PATH={absolute path to, and including, the uvm-version directory}
```

VCS

```
make SIM=vcs BUS=Skyros build \
UVM_PATH={absolute path to, and including, the uvm-version directory}
```

Note

- To compile a 64-bit build you must append `OS64=1` to the `make` command.
- To display the default values of `UVM_PATH` you can use the `make` or `make help` commands.
- If your simulation environment includes support for the Protocol Buffers and you require Tarmac trace support then append `ENABLE_TARMAC=1` to the `make` command.
- If you only want to echo, not execute, any of the make commands, append `--just-print` to the `make` command.

For example, to echo the exact vcs command that is used to build a 32-bit binary for the ACE configured model, use the following command:

```
make SIM=vcs BUS=ACE build --just-print
```

For example, if your UVM install exists at `/usr/lib/uvm-`*`version`* then the build command for each build on the IUS simulator is:

32-bit build

```
make SIM=ius BUS=Skyros build \
UVM_PATH=/usr/lib/uvm-version
```

64-bit build

```
make SIM=ius BUS=Skyros OS64=1 build \
UVM_PATH=/usr/lib/uvm-version
```

How to run a test

If the multiprocessor implements an ACE interface then Table 4-8 lists the command you use to run a test, for each simulator and build.

Table 4-8 Command to run a test on a multiprocessor that implements an ACE interface

Simulator		Command to run the `{test_name}`[a] test[b]
IUS	32-bit	`make SIM=ius BUS=ACE TEST={test_name} CPU={N} run`
	64-bit	`make SIM=ius BUS=ACE OS64=1 TEST={test_name} CPU={N} run`
QuestaSim	32-bit	`make SIM=mti BUS=ACE TEST={test_name} CPU={N} run`
	64-bit	`make SIM=mti BUS=ACE TEST={test_name} OS64=1 CPU={N} run`
VCS	32-bit	`make SIM=vcs BUS=ACE TEST={test_name} CPU={N} run`
	64-bit	`make SIM=vcs BUS=ACE TEST={test_name} OS64=1 CPU={N} run`

[a] See Table 4-6 on page 4-11 for a list of tests you can assign to `{test_name}`.

[b] {N} = the number of processors in the multiprocessor.

If your simulation environment includes support for the Protocol Buffers then you can run a test that supports Tarmac trace by including the `ENABLE_TARMAC=1` and `TESTARGS="+tarmacALL"` options. For example:

- `make SIM=ius BUS=ACE TEST={test_name} CPU={N} ENABLE_TARMAC=1 TESTARGS="+tarmacALL" run`
- `make SIM=vcs BUS=ACE TEST={test_name} OS64=1 CPU={N} ENABLE_TARMAC=1 TESTARGS="+tarmacALL" run`

If the multiprocessor implements a CHI interface then Table 4-9 lists the command you use to run a test, for each simulator and build.

Table 4-9 Commands to run a test on a multiprocessor that implements a CHI interface

Simulator		Command to run the `{test_name}`[a] test[b]
IUS	32-bit	`make SIM=ius BUS=Skyros TEST={test_name} CPU={N} run`
	64-bit	`make SIM=ius BUS=Skyros OS64=1 TEST={test_name} CPU={N} run`
QuestaSim	32-bit	`make SIM=mti BUS=Skyros TEST={test_name} CPU={N} run`
	64-bit	`make SIM=mti BUS=Skyros TEST={test_name} OS64=1 CPU={N} run`
VCS	32-bit	`make SIM=vcs BUS=Skyros TEST={test_name} CPU={N} run`
	64-bit	`make SIM=vcs BUS=Skyros TEST={test_name} OS64=1 CPU={N} run`

a. See Table 4-6 on page 4-11 for a list of tests you can assign to `{test_name}`.

b. {N} = the number of processors in the multiprocessor.

If your simulation environment includes support for the Protocol Buffers then you can run a test that supports Tarmac trace by including the `ENABLE_TARMAC=1` and `TESTARGS="+tarmacALL"` options. For example:

```
•make SIM=ius BUS=Skyros TEST={test_name} CPU={N} ENABLE_TARMAC=1 TESTARGS="+tarmacALL" run
•make SIM=vcs BUS=Skyros TEST={test_name} OS64=1 CPU={N} ENABLE_TARMAC=1 TESTARGS="+tarmacALL" run
```

How to dump waves from a time window

You use the plusargs `+dumpon=<time>` and `+dumpoff=<time>` to dump waves from a time window.

The following commands are examples that dump waves from time 10000 - 50000, for the different simulators:

IUS

```
make SIM=ius BUS=ACE TEST={test_name} TESTARGS="+dumpon=10000 \
+dumpoff=50000 +tarmacALL" CPU={N} ENABLE_TARMAC=1 run
```

QuestaSim

```
make SIM=mti BUS=ACE TEST={test_name} TESTARGS="+dumpon=10000 \
+dumpoff=50000 +tarmacALL" CPU={N} ENABLE_TARMAC=1 run
```

VCS

```
make SIM=vcs BUS=ACE TEST={test_name} TESTARGS="+dumpon=10000 \
+dumpoff=50000 +tarmacALL" CPU={N} ENABLE_TARMAC=1 run
```

4.5.3 `hello_word` test result

The `hello_world` test displays the text `hello, world` and terminates.

4.5.4 `hello_word64` test result

The `hello_world64` test displays the text `Hello, 64-bit World!` and terminates.

4.5.5 `post_config_check` test result

The `post_config_check` test displays the configuration of the RTL and then terminates.

An example output is:

```
CPU0: Cortex-A57 pre/post configuration check
CPU0: CPU number               : 4
CPU0: L2 Cache Size            : 2048 KB
CPU0: L2 Tag RAM Slice         : 0
CPU0: L2 Data RAM Slice        : 0
CPU0: L2 Arbitration Slice     : No
CPU0: ECC/Parity               : L2 only
CPU0: Bus Master Interface     : ACE
CPU0: Cryptography Extension   : Yes
CPU0: ** TEST PASSED OK **
Simulation complete in      13483 cycles
```

Note

The `post_config_check` test does not:

- Perform any checking of the expected configuration. You must check that the configured design and the test output are consistent.

- Check the *Regional clock gating* (RCG) configuration option because of a lack of software visibility.

4.5.6 `arm_ok` test result

The `arm_ok` test is a simple data-processing test. If the test completes successfully, it prints `CPU0: ** TEST PASSED OK **` and terminates.

4.5.7 `cache_miss` test result

If the test completes successfully, it prints `CPU0: ** TEST PASSED OK **` and terminates.

Table 4-4 on page 4-8 shows the measurement window for this test.

4.5.8 `crypto64` test result

If the test completes successfully, it prints `CPU0: ** TEST PASSED OK **` and terminates.

Note

To perform this test, the multiprocessor must implement the Cryptography Extension.

4.5.9 `dhrystone` test result

If the test completes successfully, it prints `CPU0: ** TEST PASSED OK **` and terminates.

Table 4-4 on page 4-8 shows the measurement window for this test.

4.5.10 `dhrystone64` test result

If the test completes successfully it prints `CPU0: ** TEST PASSED OK **` and terminates.

Table 4-4 on page 4-8 shows the measurement window for this test.

4.5.11 `saxpy64` test result

If the test completes successfully, it prints `CPU0: ** TEST PASSED OK **` and terminates.

Table 4-4 on page 4-8 shows the measurement window for this test.

4.5.12 `wfi` test result

If the test completes successfully, it prints `CPU0: ** TEST PASSED OK **` and terminates.

Table 4-4 on page 4-8 shows the measurement window for this test.

4.5.13 `sustained_max_power64` test result

If the test completes successfully, it prints `CPU0: ** TEST PASSED OK **` and terminates.

Table 4-4 on page 4-8 shows the measurement window for this test.

4.5.14 `sustained_max_power64_mp` test result

If the multiprocessor implements two or more processors and the test completes successfully then it:

1. Prints:

   ```
   CPU0: ** TEST PASSED OK **
   CPU1: ** TEST PASSED OK **
   ```

2. If the multiprocessor implements three processors it also prints:

   ```
   CPU2: ** TEST PASSED OK **
   ```

3. If the multiprocessor implements four processors it also prints:

   ```
   CPU3: ** TEST PASSED OK **
   ```

4. Terminates the test.

Table 4-4 on page 4-8 shows the measurement window for this test.

4.6 Outputs from checking your RTL

The log files output are specific to your verification tool. By default, the results of running tests on the RAM integration testbench are output to the screen. You can redirect this output into a log file if required.

4.7 Solving any checking your RTL problems

If any of the tests do not complete with `** TEST PASSED OK **` and you cannot identify the reason for the failure, contact ARM for assistance.

Chapter 5
Floorplan Guidelines

This chapter describes the floorplan you can use as a starting point for your design. It contains the following sections:

5.1 About floorplanning

Figure 5-1 shows the top-level inputs, resources, outputs, and controls and constraints for floorplanning.

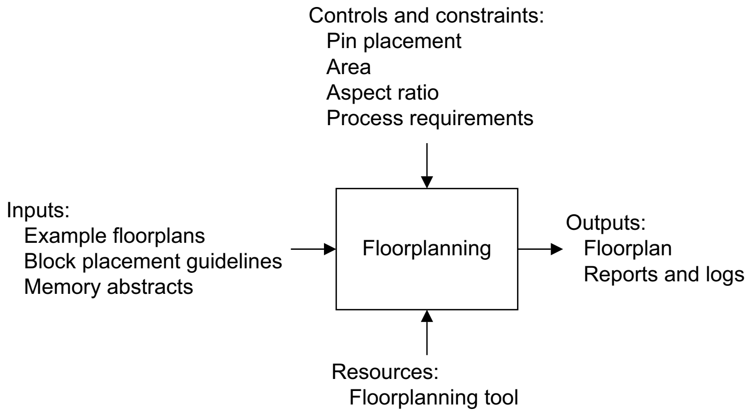


Figure 5-1 Floorplanning process

5.2 Resource requirements for floorplanning

This guide assumes that you have suitable EDA tools and compute resources for floorplanning.

5.3 Controls and constraints for floorplanning

The following are controls and constraints that can influence floorplanning:

- Pin placement.
- Area.
- Aspect ratio.
- Process requirements.

Documentation for your chosen implementation flow might contain a sample floorplan with these constraints.

5.4 Inputs for floorplanning

Inputs are specific to your floorplanning tool, and can include the following:

- Example floorplans.
- Block placement guidelines.
- Pin placement.
- Placement blockages.

Documentation for your chosen implementation flow might contain a sample floorplan with these inputs.

5.5 Considerations for floorplanning

Your floorplan can implement the `atl_cpu` and `atl_noncpu` as separate macros. This is a proven method but does not imply that this is the only option for implementing the Cortex-A57 MPCore multiprocessor. The complexity of the Cortex-A57 MPCore multiprocessor permits other methods, including a hierarchical approach and different flat approaches.

For optimum performance, ARM recommends that you:

- Group RAM instances together for a single function to permit functional and power-related cells, or both, to be placed between instances.
- Make the standard cell placement area as close to a 1:1 ratio as possible.
- Make the size of each `atl_cpu` and `atl_noncpu` dependent on Cortex-A57 MPCore multiprocessor macro size requirements, RAM sizes and placement, and pin placement.

See the documentation for your chosen implementation flow for:

- Aspect ratios for memories that are specific to your implementation flow. These might be specific to the memory compilers available to you.
- Required floorplanning of different hierarchical levels.
- Clock modules that have placement constraints to optimize clock latency matching.

RAM placement and pin placement have a direct effect on the results of floorplanning.

5.6 Outputs from floorplanning

Although output files are specific to your floorplanning, these can be:

- Fixed pin locations.
- Route guides or blockages.
- Placement blockages.
- Initial cell placement or macro placement.
- Bounds, whether soft or hard, inclusive or exclusive.

5.7 Reference data for floorplanning

The following sections provide reference information for floorplanning:

- *Standard cell floorplanning*.
- *Memory block floorplanning*.
- *Interface floorplanning*.
- *Power grid design*.

5.7.1 Standard cell floorplanning

Floorplanning does not place the standard cells. ARM expects the synthesis tool to place the standard cells using a physical constrained placement methodology.

In most cases, architectural clock gates must be fixed in place to control clock latencies.

5.7.2 Memory block floorplanning

The reference information for memory block placement are:

`atl_cpu` RAMs must be placed with instances of the same function near each other. In some cases, such as data cache and instruction cache, they can be placed facing each other so that multiplexing can take advantage of pin placement.

`atl_noncpu` Many of the RAMs, primarily the L2 cache RAMs in the tag banks, require specific placements to optimize multiplexing.

5.7.3 Interface floorplanning

The following sections provide reference information for interface floorplanning:

- *Reference multiprocessor interface planning*.
- *Cortex-A57 MPCore multiprocessor interface planning*.

Reference multiprocessor interface planning

If you use the reference flow for floorplanning then each processor has an interface. This interface is unique to each processor and in the reference flow it is flipped to use one integrated processor in each instantiation. Care is given to the pin placement on the `atl_cpu` and `atl_noncpu` macros to optimize both in terms of interface.

Cortex-A57 MPCore multiprocessor interface planning

The Cortex-A57 MPCore multiprocessor pins are optimized in the microarchitecture to permit one and only one interface. The design includes appropriate levels of pipelining so that during integration there is no requirement for multiple connection to pins.

The `atl_cpu` and `atl_noncpu` pins are specific to the processors in the system. There are no specific requirements for their placement except that they must meet the timing constraints and follow power domain rules.

It is best to locate the debug block in the `atl_noncpu` on an edge of the design. Interface pins for this block must be located directly adjacent to this block.

5.7.4 Power grid design

ARM recommends adding the power grid before physical synthesis so that the tool has a more accurate representation of the available routing resource.

You must design the grid to meet the requirements of your library. However, ARM recommends a grid that satisfies an IR drop of 2%, V_{DD} and V_{SS} combined.

ARM also recommends that your metal stack selection includes sufficient horizontal metal resources to permit signals to go over your RAMs.

Chapter 6
Design For Test

This chapter describes the Design for Test features of the Cortex-A57 MPCore multiprocessor. It contains the following section:

- *About Design for Test (DFT) features* on page 6-2.

6.1 About *Design for Test* (DFT) features

The DFT features of the Cortex-A57 MPCore multiprocessor enable you to perform production testing of your design.

The multiprocessor uses *Automatic Test Pattern Generation* (ATPG) test vectors for production test. To accomplish this, scan chains are inserted during synthesis, see the *ARM® Cortex®-A57 MPCore™ Processor Integration Manual* for more information.

See the documentation from your EDA vendor for information about:

- Requirements for DFT.
- Controls and constraints.
- DFT features.
- Confirmation of DFT feature operation and test coverage.
- Solving DFT feature problems.

The *ARM® Cortex®-A57 MPCore™ Processor Integration Manual* describes how to use the *Memory Built-In Self Test* (MBIST) interface to access the memories in the multiprocessor. Use this information to connect an MBIST controller to the multiprocessor.

Note

ARM does not supply an MBIST controller with the multiprocessor.

Figure 6-1 shows the top-level inputs, resources, outputs, controls, and constraints for DFT.

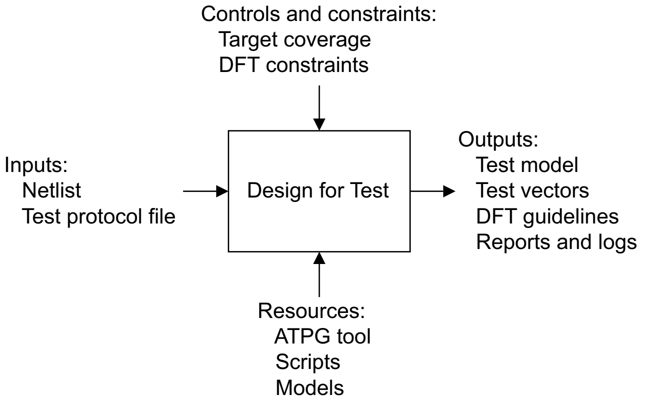


Figure 6-1 DFT process

Chapter 7
Dynamic Verification

This chapter describes how to use the supplied test vectors to verify your implementation. It contains the following sections:

7.1 About dynamic verification

Figure 7-1 shows the top-level inputs, resources, outputs, and controls and constraints for dynamic verification.

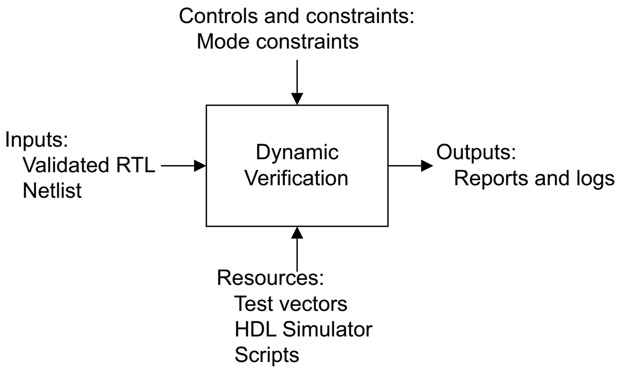


Figure 7-1 Dynamic verification process

This chapter describes replaying the vectors on the netlist. There are two methods to accomplish replaying the test vectors:

- Direct netlist replay.
- Extracting and replaying I/O VCD.

You decide the method to use.

Note

- ARM recommends running the test vector simulations on the final netlist for all tests. See *Additional recommendations for sign-off* on page 9-4.
- Dynamic verification does not completely verify your implementation. It must also pass logical equivalence verification and timing verification.

7.2 Resource requirements for dynamic verification

Table 7-1 shows the resource requirements for dynamic verification.

Table 7-1 Resource requirements for dynamic verification

Requirement	Description
Testbench	See *Execution testbench* on page 4-3
ELF test vector files	See *Inputs for dynamic verification* on page 7-4

7.3 Inputs for dynamic verification

The inputs for dynamic verification are:

- *ELF test vector files*.
- *ARM RealView Compilation Tools*.
- *GNU Compiler Collection*.

7.3.1 ELF test vector files

Table 7-2 shows the ELF test vector files that are supplied for dynamic verification.

Table 7-2 ELF test vector files

File	Description
`hello_world`	Displays the text `hello, world`
`hello_world64`	Displays the text `Hello, 64-bit World!`
`post_config_check`	Displays the configuration of the RTL
`arm_ok`	Simple data processing test
`cache_miss`	Power indicative test. Processor stalled waiting for data to be returned from memory
`crypto64`	Exercises the Cryptography engine
`dhrystone`	Exercises processor integer pipeline[a]
`dhrystone64`	Exercises processor integer pipeline with A64 instructions[a]
`saxpy64`	Exercises processor floating-point pipeline with A64 instructions[a]
`wfi`	Minimal low-power state
`sustained_max_power64`	Maximum sustained power consumption with A64 instructions
`sustained_max_power64_mp`	Maximum sustained power consumption with A64 instructions for multiprocessor implementations

[a] Everything is cached in L1.

7.3.2 ARM RealView Compilation Tools

You can use the ARM *RealView Compilation Tools* (RVCT) to recompile the ELF files for the A32 tests. This is optional, and is only required if you recompile the ELF files. See *Rebuild the tests* on page 4-7 for the commands to build new ELF files from source, and install them in the run directory.

7.3.3 GNU Compiler Collection

You can use the *GNU Compiler Collection* (GCC) to recompile the ELF files for the A64 tests. This is optional, and is only required if you recompile the ELF files. See *Rebuild the tests* on page 4-7 for the commands to build new ELF files from source, and install them in the run directory.

7.4 Flow for dynamic verification

There are two methods for dynamic verification:

- *Direct netlist replay*.
- *Extracting and replaying I/O VCD*.

7.4.1 Direct netlist replay

With this method, the netlist is instantiated into the `execution_tb` environment and the test is run as normal. From this, VCDs can be generated as required for downstream tools. You must create a new Verilog command file (`.vc`) that includes references to:

- The netlist.
- The relevant library RAM models.
- The relevant library cell models.

You can copy the supplied `execution_tb.vc` and replace RTL references with references to your netlist and libraries. The following is an example of how your `execution_tb.vc` might appear:

```
+libext+.v+.vlib+.sv+.svh

// Testbench files
<All testbench files from original execution_tb.vc>

    // NETLIST
-v <your_design_netlist>

// RAMS
-y <your_ram_models>

// STD CELLS
-y <your_std_library_cells>
```

You must remove the EIS section in the `execution_tb.vc`, to allow the netlist simulation to compile. An example compile command might be:

```
vcs -full64 -sverilog -f atl_netlist.vc +define+GLS
```

(and any other netlist sim options you require)

The execution_tb requires +define+GLS when compiling to remove some RTL-specific constructs from the testbench. Simulation then runs as normal:

```
./simv +elfname=../tests/power_indicative/dhrystone/elf/dhrystone.elf
```

To run and generate a VCD file:

```
./simv +elfname=./tests/power_indicative/dhrystone/elf/dhrystone.elf +dumpon=0
```

The VCD file that is generated can then be used in downstream tools as required, for example to measure power.

7.4.2 Extracting and replaying I/O VCD

There are numerous ways to extract and replay the I/Os from the VCD. Follow your preferred method for this step.

The following example shows how you can use the third-party tools, Synopsys VCS utility `vcat` and Source III VTRAN, to extract and replay the vectors respectively. The following steps require that you have a VCD file produced by running test vectors on a netlist.

To replay the vectors:

1. Enter the VCS utility command:

   ```
   vcs -vcat <yourfile>.vcd -vgen io.cfg
   ```

   The `io.cfg` file requires the following information to generate a testbench:

   - A hierarchy for the *Design Under Test* (DUT).
   - The command `testbench` to generate the testbench and provide stimulus.
   - A list of the required signals to be extracted from the VCD.

To create a testbench with cycle-based vectors:

2. Enter the VTRAN tool command:

   ```
   vtran vtran.cmd <yourfile>.vcd <generated_testbench.v>
   ```

   The VTRAN tool converts an event driven VCD into cycle-based vectors. The tool also generates a testbench that includes the stimulus extracted from the VCD and the appropriate checks for the output signals.

   Example 7-1 shows an example `vtran.cmd` file.

Example 7-1 vtran.cmd file for extracting vectors

```
ovf_block
   begin
     orig_file "<yourfile>.vcd";
     script_format verilog_vcd;
     inputs <input_pin_list_for extracting>;
     outputs <output_pin_list_for extracting>;
   end;
proc_block
    begin
      cycle <your_clk_cycle_period>;
      disable_vector_filter;
      align_to_signal -warnings <your_clk_name> *->* sample=pure_inputs @ <time_to_sample_inputs>;
      align_to_signal -novector <your_clk_name> 0->1 sample=pure_outputs @ <time_to_sample_outputs>;
      mask_pins pure_outputs @ NOT CONDITION <any_conidtion_you_require_to_mask_pins>;
    end;
tvf_block
   begin
     header 50
     simulator Verilog_tb,
     -VERBOSE
     TESTBENCH_MODULE = "<testbench_name>",
     COMPONENT_MODULE = "<DUT_name>",
     INSTANCE_NAME = "<DUT_instance>",
     TIMESCALE = "<your_timescale>",
     MAX_MISMATCHES = "<max_number_of_mismatches>";
     inputs <input_pin_list_as_stimulus_vectors>;
     outputs <output_pin_list_for checking_against_DUT>;
     target_file "<generated_testbench>.v";
     command_file "<generated_testbench>.cmd";
end;
```

After generating the testbench through either your preferred way or by using steps 1 and 2, you must create a `.vc` file that instantiates the generated testbench, netlist, RAM models, and all the libraries required to run the simulation.

You also must add the required switches depending on your simulation type and your selected tool to run the simulation properly. You can generate a similar testbench for any of the vectors of your choice but as an example, see *Flow for checking your RTL* on page 4-10.

7.5 Outputs from dynamic verification

The log files that are output by tests are specific to your verification tool. By default, the results of tests that are run on the execution testbench are output to the screen. You can redirect this output to a log file if required.

Note

ARM recommends running all the test vectors on the post-layout netlist. See *Additional recommendations for sign-off* on page 9-4.

7.6 Solving dynamic verification problems

If any of the tests do not complete with `** TEST PASSED OK **` and you cannot identify the reason for the failure, contact ARM for assistance.

Chapter 8
Power Intent Specification

This chapter describes the power intent specification for the Cortex-A57 MPCore processor and the logic levels, HIGH or LOW, that the outputs of power domains must be clamped to before power is removed during a powerdown sequence. It contains the following sections:

- *Power intent specification* on page 8-2.
- *Power domain clamp values* on page 8-3.

8.1 Power intent specification

The UPF files in the `logical/atlas/power_intent/upf` directory describes the basic power intent for the Cortex-A57 MPCore processor.

The Cortex-A57 UPF files are organized hierarchically. UPF files are provided for 1 to 4 processor configurations.

Note

- The UPF files are not setup to be used in a physical implementation flow. They can be used together with the delivered RTL to validate the power intent specification of your Cortex-A57 implementation.
- The Cortex-A57 Reference Implementation deliverables include the power intent files that are appropriate for a physical implementation for that reference implementation flow.
- The UPF files are IEEE1801-2009 compliant.

8.2 Power domain clamp values

This section describes the output signals of the power domains that must be clamped to the required benign values.

Almost all the power domain output signals must be clamped LOW before power is removed during the powerdown sequence. However, there are a few exceptions. This section lists only those power domain output signals that must be clamped HIGH. All the outputs, other than those described in Table 8-1, Table 8-2, and Table 8-3 on page 8-4, must be clamped LOW.

Table 8-1 shows the processor output signals that must be clamped HIGH. All other processor output signals are clamped LOW. The value of X in the processor output signal is the RTL instance number of the processor that is powered down, where X is 0, 1, 2, or 3. This represents processor 0, processor 1, processor 2, or processor 3, respectively.

Table 8-1 Processor outputs clamped HIGH

Processor output signals
ucpuX/ds_cpuX_wfi_req
ucpuX/dt_cpuX_et_oslock_gclk
ucpuX/dt_cpuX_coredbg_in_reset_gclk
ucpuX/dt_cpuX_pmusnapshot_ack_gclk
ucpuX/npmuirq_cpuX_i
ucpuX/ncommirq_cpuX_i
ucpuX/afreadym_cpuX_i
ucpuX/commtx_cpuX_i

Table 8-2 shows the debug output signals in the **PCLKDBG** domain that must be clamped HIGH. All other debug **PCLKDBG** output signals are clamped LOW. Some of the buses include a configurable width field, **<signal>[N:0]**, where N = 0, 1, 2, or 3, to encode up to four processors.

Table 8-2 Debug PCLKDBG outputs clamped HIGH

Debug (PCLKDBG) output signals
unoncpu/udt_pclk/PMUSNAPSHOTACK[N:0]
unoncpu/udt_pclk/PREADYDBG
unoncpu/udt_pclk/ PSLVERRDBG
unoncpu/udt_pclk/CTICHINACK[3:0]

Table 8-3 shows the Cortex-A57 MPCore processor output signals that must be clamped HIGH. All other Cortex-A57 output signals are clamped LOW. Some of the buses include a configurable width field, **<signal>[N:0]**, where N = 0, 1, 2, or 3, to encode up to four processors. Some signals are specified in the form **<signal>x** where x = 0, 1, 2, or 3 refer to processor 0, processor 1, processor 2, or processor 3, respectively.

Table 8-3 Cortex-A57 outputs clamped HIGH

Cortex-A57 output signals
STANDBYWFI[N:0]
STANDBYWFIL2
nCNTPNSIRQ[N:0]
nCNTPSIRQ[N:0]
nCNTHPIRQ[N:0]
nCNTVIRQ[N:0]
nVCPUMNTIRQ[N:0]
nINTERRIRQ
nEXTERRIRQ
nPMUIRQ[N:0]
nCOMMIRQ[N:0]
COMMTX[N:0]
AFREADYMx[a]
PMUSNAPSHOTACK[N:0]
PREADYDBG
PSLVERRDBG
CTICHINACK[3:0]

[a] x is the number of the processor.

Chapter 9
Sign-off

In addition to your normal sign-off checks, there is a contractual requirement to verify that the implemented multiprocessor is ARM compliant before you sign-off the multiprocessor design. This chapter describes the sign-off criteria. It contains the following sections:

9.1 About sign-off

Figure 9-1 is a process diagram that shows the top-level inputs, resources, outputs, and controls and constraints for sign-off.

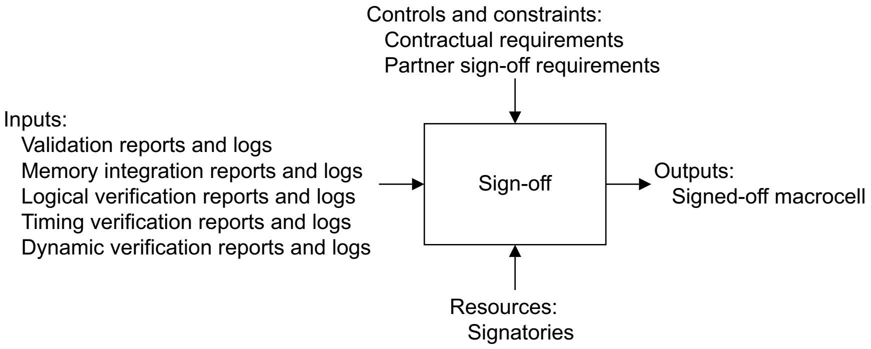

Controls and constraints:
Contractual requirements
Partner sign-off requirements
Inputs:
Validation reports and logs
Memory integration reports and logs
Logical verification reports and logs
Timing verification reports and logs
Dynamic verification reports and logs
Sign-off
Outputs:
Signed-off macrocell
Resources:
Signatories

Figure 9-1 Sign-off process

9.2 Obligations for sign-off

All ARM partners must fulfill the terms of their contract with ARM to complete sign-off.

Signatories must approve the sign-off of the design in accordance with:

- The terms of the contract with ARM.
- Any other partner sign-off requirements.

See *Implementation obligations* on page vi for more information.

9.2.1 Requirements for sign-off

Caution

When you sign-off your design you must fulfill the terms of your contract with ARM. Typically, your contract stipulates the mandatory sign-off requirements given in this section. However, you must check your contract for any variation to the mandatory requirements.

You must successfully complete the following stages for sign-off:

- RTL configuration and checksum verification. See Chapter 2 *Configuration Guidelines*.
- Memory integration and validation. See Chapter 3 *Memory Integration*.
- Check your configured RTL with the supplied tests.
- *Logical Equivalence Check* (LEC). See the supplied reference methodology documents.
- Timing verification, by *Static Timing Analysis* (STA) of the post-layout netlist. See the supplied reference methodology documents.

Note

You can change the timing constraints to suit your design provided it still meets all the mandatory requirements for sign-off.

Reports and logs from each of these stages are required for sign-off.

A certain minimum set of deliverable outputs is required at the end of the implementation. See *Completion of sign-off* on page 9-6.

All ARM partners must fulfill the terms of their contract with ARM to complete sign-off.

9.3 Additional recommendations for sign-off

ARM recommends the following stages for sign-off:

- *Design Rule Check* (DRC). See the documents supplied by your EDA tool vendor.
- *Layout Versus Schematic* (LVS). See the documents supplied by your EDA tool vendor.
- Test vector simulations on post-layout netlist. See Chapter 7 *Dynamic Verification*.
- Power analysis using the power indicative tests that Table 4-4 on page 4-8 shows. The power tests must be run on your netlist.

9.4 Steps for sign-off

To sign-off the multiprocessor, you must meet the criteria in each of the following stages in the design flow:

1. *RTL configuration*.
2. *RTL checksum verification*.
3. *Memory integration*.
4. *Verification of configured RTL*.
5. *Logical equivalence checking*.
6. *Static timing analysis*.
7. *Test vector simulations on post-layout netlist*.

9.4.1 RTL configuration

You must configure the optional parameters for the multiprocessor as required for the specific implementation. See Chapter 2 *Configuration Guidelines*.

9.4.2 RTL checksum verification

You must send RTL checksum data to your ARM representative and receive verification before proceeding. See Chapter 2 *Configuration Guidelines*.

9.4.3 Memory integration

You must successfully accomplish RAM integration. You must capture and save the RAM integration testbench results for inspection. See Chapter 3 *Memory Integration*.

9.4.4 Verification of configured RTL

You must run all the supplied tests on the configured RTL. See Chapter 4 *Checking your RTL*.

9.4.5 Logical equivalence checking

You must perform LEC between the configured RTL and the final place-and-route netlist. You must capture and save the results for inspection.

9.4.6 Static timing analysis

You must perform STA to verify timing of the final place-and-route netlist. You must capture and save the results for inspection.

9.4.7 Test vector simulations on post-layout netlist

You might run all of the supplied tests, that Chapter 4 describes, on the final place-and-route netlist before you sign-off your netlist. See Chapter 7 *Dynamic Verification*.

9.5 Completion of sign-off

For successful completion of sign-off, you must have produced or completed and verified the following ARM-related items from the implementation process:

- Configuration of the RTL and verification of the RTL checksum.
- Memory integration validation logs.
- Execution testbench validation logs.
- LEC log results.
- STA log results.
- Test vector replay logs or results on the post-layout netlist.
- All required reports and logs as specified in your contract.

Appendix A
Tarmac Trace Description

This appendix describes the Tarmac trace format for the Cortex-A57 MPCore multiprocessor execution testbench. It contains the following sections:

- *About the Tarmac trace file* on page A-2.
- *Enabling trace* on page A-3.
- *Tarmac integration guide* on page A-4.

A.1 About the Tarmac trace file

The multiprocessor generates Tarmac trace that is compliant to the *ARM® Tarmac Specification*.

A.1.1 Tarmac formats that are specific to the Cortex-A57 MPCore multiprocessor

Table A-1 lists how the Cortex-A57 MPCore multiprocessor implements the optional features and formats of the *ARM® Tarmac Specification*.

Table A-1 Tarmac feature implementation

Trace record type	Description
Header	`Tarmac Text Rev 3t`.
Timestamp unit	The multiprocessor outputs `clk` as the unit for each timestamp. One `clk` unit represents one **CLK** cycle.
Asynchronous event record	The multiprocessor does not generate Background Update events.
Memory record	The multiprocessor does not generate the Page Order field.
Branch record	The multiprocessor does not generate branch records.
Register update record	For microarchitectural reasons, at each instruction boundary, the processor does not track the value of the following bits: **In AArch32** CPSR.[T, IT, Q] and FPSCR.[QC, IDC, IXC, UFC, OFC, DZC, IOC]. **In AArch64** FPSR.[QC, IDC, IXC, UFC, OFC, DZC, IOC]. Therefore, the Tarmac is not accurate for these bits on each instruction boundary. However, the bits are accurate on any read of the CPSR, FPSCR, or FPSR.
Exception record	If a processor is using AArch32, it does not generate the following exception records: • Array Bound Check. • Null Pointer Check.

A.2 Enabling trace

There is a separate trace file for each processor. The format to enable trace is:

- `+tarmacALL`, enables Tarmac trace output to a file.

  The filename is `cl`*`x`*`.cpu`*`N`*`.tarmac`, where:

  — *x* is the cluster number, that is, 0 to 255.

  — *`N`* is the processor number in that cluster, that is, 0, 1, 2, or 3.
- `+tarmac_prefix=<string>`, appends `<string>` to the Tarmac filename. This switch must be less than 256 characters. The behavior is unpredictable if you exceed this limit.

Example A-1 shows how to create the Tarmac trace file `cl`*`x`*`_cpu`*`N`*`.tarmac`.

Example A-1 Enable trace format

The command to create the trace file `/home/user/work/logs/cl`*`x`*`_cpu`*`N`*`.tarmac` is:

```
+tarmacALL +tarmac_prefix=/home/user/work/logs/
```

See Table 4-5 on page 4-9 for more information about running tests with the Tarmac trace options.

A.3 Tarmac integration guide

The Tarmac logging system consists of:

Verilog modules These modules trace the instruction execution of the processors within a cluster and invoke DPI calls to transfer information to the C++ shared library.

`libdpi_noisscompare.so`
A C++ shared library that formats the executed instructions and prints to the Tarmac log files for each processor.

The following sections describe how to integrate the Verilog modules into your testbench:

- *Tarmac Verilog modules*.
- *How to instantiate the TraceAndCompare module for Tarmac trace support*.
- *Defines* on page A-5.
- *Limitations* on page A-5.

A.3.1 Tarmac Verilog modules

The Tarmac Verilog modules are:

`TraceAndCompare.sv` The cluster level module that instantiates trace modules. It uses the SystemVerilog `bind` method to bind to a Cortex-A57 MPCore multiprocessor cluster.

`TnC_cpu.sv` The processor trace module.

`TnC_noncpu.sv` The noncpu and L2 trace module.

A.3.2 How to instantiate the `TraceAndCompare` module for Tarmac trace support

To instantiate the `TraceAndCompare` module:

1. Define the hierarchical path, `CLUSTER0_PATH`, to the cluster 0 instance, for example, `tb.top.uatlas0`.

2. If you are building a multi-cluster testbench then define the `CLUSTER`*x*`_PATH` for each cluster, where *x* is the cluster number.

For each CLUSTER*x*_PATH:

3. Use the SystemVerilog `bind` statement to attach the `TraceAndCompare` module to each Cortex-A57 MPCore multiprocessor cluster. Each `TraceAndCompare` module represents a cluster.

   Example A-2 shows the code for a 4-cluster system testbench.

Example A-2 4-cluster system testbench

```
`ifdef CLUSTER0_PATH
bind `CLUSTER0_PATH TraceAndCompare  TraceAndCompare0();
`endif

`ifdef CLUSTER1_PATH
bind `CLUSTER1_PATH TraceAndCompare  TraceAndCompare1();
`endif

`ifdef CLUSTER2_PATH
```

```
bind `CLUSTER2_PATH TraceAndCompare  TraceAndCompare2();
`endif

`ifdef CLUSTER3_PATH
bind `CLUSTER3_PATH TraceAndCompare  TraceAndCompare3();
`endif
```

To terminate the simulation gracefully:

4. Ensure your testbench calls the `finalize_isscmp()` DPI function before it calls `$finish()`.

A.3.3 Defines

When you use the `compile` command with `+define+MC_TESTBENCH` then the `TraceAndCompare` module contains the following defines:

```
`define    EAG_L2_PATH     unoncpu          //noncpu instance

`define    EAG_CPU_PATH ucpu0               //cpu0 instance
`define    EAG_ID_PATH `EAG_CPU_PATH.uid    //cpu0 Decode instance
`define    EAG_DS_PATH `EAG_CPU_PATH.uds    //cpu0 Dispatch instance
`define    EAG_LS_PATH `EAG_CPU_PATH.uls    //cpu0 LoadStore instance

`define    EAG_CPU1_PATH  ucpu1             //cpu1 instance
`define    EAG_ID1_PATH `EAG_CPU1_PATH.uid  //cpu1 Decode instance
`define    EAG_DS1_PATH `EAG_CPU1_PATH.uds  //cpu1 Dispatch instance
`define    EAG_LS1_PATH `EAG_CPU1_PATH.uls  //cpu1 LoadStore instance

`define    EAG_CPU2_PATH  ucpu2             //cpu2 instance
`define    EAG_ID2_PATH `EAG_CPU2_PATH.uid  //cpu2 Decode instance
`define    EAG_DS2_PATH `EAG_CPU2_PATH.uds  //cpu2 Dispatch instance
`define    EAG_LS2_PATH `EAG_CPU2_PATH.uls  //cpu2 LoadStore instance

`define    EAG_CPU3_PATH  ucpu3             //cpu3 instance
`define    EAG_ID3_PATH `EAG_CPU3_PATH.uid  //cpu3 Decode instance
`define    EAG_DS3_PATH `EAG_CPU3_PATH.uds  //cpu3 Dispatch instance
`define    EAG_LS3_PATH `EAG_CPU3_PATH.uls  //cpu3 LoadStore instance
```

These defines enable a `TraceAndCompare` module to connect airwires through hierarchical paths to the `TnC_cpu` and `TnC_noncpu` instances.

Note

These defined paths require a `TraceAndCompare` module to have access to all instances under a Cortex-A57 MPCore multiprocessor. The SystemVerilog `bind` statement enables the module to access the instances because it attaches the `TraceAndCompare` module to the multiprocessor.

If you use the `compile` command without `+define+MC_TESTBENCH` then you must add suitable defines to the `TraceAndCompare` module.

A.3.4 Limitations

Some tests are designed to software reset a cluster and assign a new clusterID that is different to the initial ID.

For example, a test might assign of cluster ID of `0x5` to a cluster that initially was cluster `0x0`. However, the test does not create a separate Tarmac log file for the new cluster ID, the output Tarmac is sent to the initial Tarmac log, that is, `cl0_cpu`*`N`*`.tarmac`.

Appendix B
Revisions

This appendix describes the technical changes between released issues of this book.

Table B-1 Issue A

Change	Location	Affects
First release	-	-

Table B-2 Differences between Issue A and Issue B

Change	Location	Affects
OVL is not supported, therefore removed references to it	*RTL validation process* on page 4-2	All
Updated the values for the `sustained_max_power64` tests	*Power test measurement windows* on page 4-8	All
Updated the scope of the power test measurement windows to 2-processor, 2MB L2, CHI	*Power test measurement windows* on page 4-8	All
Added information about the echo only option	*Build commands to use when the UVM install differs from the default location* on page 4-12	All
Updated the method description and code example	*Direct netlist replay* on page 7-5	All
Added information on UPF	Chapter 8 *Power Intent Specification*	All