Lexra 6-Stage Products User's Guide

February 1, 2002

Revision 5.0

LEXRA, Inc.
2 University Park, 51 Sawyer Road
Waltham, MA 02154
Tel: 781-899 5799
Fax: 781-899 5769

Table of Contents

Chapter
1

Lexra Development Environment

1.1 Overview

To assist in the development of hardware and software using Lexra processors, the following tools are available within the RTL (register transfer language) design database including:

- Database configuration tool
- Verilog RTL
- Verification environment
- Synthesis scripts
- Lexra Software Development Kit
- Lexra Bus Transaction Models
- Instruction Set simulator
- H/W & S/W Development Board (Altera FPGA Based)

The RTL Design Database contains all the files you need to configure, simulate, synthesize and test your Lexra processor. Starting with a text based configuration form known as the `lconfig` form, users are able to easily configure and adapt the RTL database to accommodate their ASIC methodology as well as their system design requirements. From reset and clock methodology to memory

footprint and system bus interface, the `lconfig` tool reads the `lconfig` form and properly adjusts the RTL database to conform to the system requirements. In doing this, an RTL database is created that reflects your customized processor. No hand modifications need to be made other than the editing of the `lconfig` form. The `lconfig` tool verifies the validity of your selections, creates a customized regression suite and generates synthesis scripts specific to your methodology and processor configuration options.

To assist in the validation of your customized configuration, Lexra supplies a verification environment called `rundvt`. Supporting industry standard simulators, `rundvt` acts as an easy to use command line interface allowing you to run your own customized software tests or the Lexra supplied regression suite on your configured processor verilog RTL or gate level netlist.

Needed for the running of verification tests, Lexra ships with their database a GNU based software developer's kit called the LSDK. This collection of files compiles, assembles and links the C and assembly files that make up part of the testbed environment. From boot vectors, exception routines, targeted C and assembly tests to customer written software routines, the LSDK is used in conjunction with `rundvt` to process high level code so that it can be run on the verilog RTL or gate level netlist.

Lexra provides bus transaction models that support their PCI-like system bus called LBUS. The models can be used to help users understand the system bus topography and to act as a task driven replacement for the entire verilog processor in the customer's own simulation environment.

The instruction set simulator comes in both instruction accurate and cycle accurate version. This C-model of the processor is intended to enable software development prior to first silicon. The `ISS` accurately models the pipeline and memory subsystem of the processor, which can be interfaced with an easy to use command line interface (`CLUE`) or with the Green Hill's `MULTI` integrated development environment. Fine tuning critical code loops and optimizing software performance can efficiently be done utilizing this type of tool. The `ISS` also supports a co-verification/co-simulation environment to allow ASIC design and software teams to work together in the validation of system requirements and functionality prior to tapeout. The `ISS` has additional support for the creation of custom instructions and the use of coprocessors.

The Altera FPGA based development board is usable by both the hardware and software teams to get a head start in the development of their system on a chip

(SOC). Able to boot third party RTOS like VxWorks, ThreadX, Nucleus and Linux, the development board gives software teams hardware to work with prior to tapeout improving their efficiency in creating application code. As a hardware development platform, the development board supports a dedicated FPGA for the sole purpose of hardware integration. This FPGA is able to communicate to the Lexra processor held in an additional FPGA on the board. This functionality allows hardware engineers the ability to prototype system peripherals, custom instructions, and even coprocessors. Software debug on the development board can be achieved through the use of an EJTAG probe.

1.2 RTL Design Database Overview

The Lexra processor is packaged in a standalone development directory. Do not integrate this directory directly into your design environment. Instead, use it to configure and simulate the Lexra processor as well as to generate the design objects you will need in your design environment, specifically:

- simulation RTL
- synthesized gate level netlists

1.3 Requirements for the RTL Design Database

The RTL design database runs on Sun Solaris platform version 2.6 and higher. It works with the following design tools:

- Synopsys VCS or Cadence Verilog-XL & NC-Verilog for verilog simulation
- Synopsys Design Compiler for logic synthesis and optimization
- Synopsys TetraMAX ATPG for test vector generation

The design environment also requires the following UNIX environment:

- C shell (/bin/csh)
- /usr/ccs/lib/cpp (Solaris C preprocessor for vpp script)
- The LSDK tools (specifically lxgcc for compiling new tests)
- PERL 5.0xx

Check the README file referred to in Section 1.4, Installing the RTL Design Database to see the version numbers of the tools used in testing the current version of the Lexra processor. If you are using a tool with a version different then the one listed in the README file, please contact a Lexra Application Engineer to check for compatibility.

1.4 Installing the RTL Design Database

The RTL database distribution contains multiple compressed tar files that can be found on the Lexra FTP site at `ftp.lexra.com`. See your local Lexra Application Engineer for your login account information.

Once logged into the FTP server, proceed to the `/releases/RTL/<rtl_version>` directory to find the files associated with the release to be installed. Below are examples of what you might find in the `<rtl_version>` directory:

`README`	Release Notes
`doc\*`	Documentation directory (Errata, datasheet, ...)
`lx4189-rtl-1.10.tar.Z.crypt`	RTL source files (includes `lconfig`, `rundvt`, synthesis scripts)
`lx4189-ejtag-1.10.tar.Z.crypt`	EJTAG source files (license option)
`lx4189-mac-1.10.tar.Z.crypt`	MAC source files (license option)
`lx4189-devboard-1.10.tar.Z.crypt`	Development board source files

Before downloading the compressed TAR files, download, open and read both the README and the Errata file. The README file contains the most up-to-date information regarding the release. For example:

- LSDK version requirements
- Verilog simulator version numbers used in release testing
- Design Compiler version numbers used in release testing
- UNIX decryption instructions
- Description of database and RTL changes from previous release

The Errata file is used to track issues regarding the various releases of the processor design. This might include:

- Documentation discrepancies
- Synthesis issues & incompatibilities
- Configuration dependent issues & possible work-arounds

Make sure that you understand the contents of the Errata. This one document can save you valuable time in debugging issues that may already be known. Also, feel free to ask you Lexra Application Engineer for an up-to-date errata showing all of the currently known issues effecting this version of the RTL.

After reading both the README and Errata file, continue the download process. Download only the files pertaining to your license agreement since some files may be quite large and time consuming to download. Only those who has purchased the Lexra Development Board and plan to perform hardware prototyping with the board need to download the devboard compressed tar file.

Once the download procedure is done, decrypt the tar files as follows:

crypt < tarfile.tar.Z.crypt > tarfile.tar.Z

When prompted, enter the crypt key (the crypt key can be obtained from your Lexra Application Engineer)

Determine the location to install the RTL design database (<mydir>).

cd <mydir>

You must have read and write permissions to properly configure the database in this directory. When you untar the database a top level directory will be created with the product name of the processor (i.e. `<mydir>/lx4189` - for the rest of this document we will refer to this directory as `$LX_HOME`).

zcat <archive path>/tarfile.tar.Z | tar xvf -

Start with the compressed RTL file and follow with subsequent files (i.e. ejtag, mac, ..) After untarring all of the files, `lconfig` must be run with either one of the

default `lconfig` forms or your own `lconfig` form to properly setup the database. However, before running `lconfig` make sure that PERL is installed properly. See Chapter 2, Using Lconfig and Section 1.5, Configuring for PERL..

1.5 Configuring for PERL

Due to it's scripting ability, many of the Lexra supplied tools require the installation of PERL. To check if you have PERL installed, enter the following at the UNIX command prompt:

which perl

If PERL is not found ask your system administrator whether or not it is installed and make the changes so that it can be found in your path. PERL can be downloaded from the internet at `www.perl.com`.

Many of the Lexra provided scripts hardcode the location of PERL to `/usr/local/bin/perl`. If your PERL location is different, please run the following utility to modify the hardcoded location:

cd $LX_HOME

bin/setup_scripts

Specify the installation path you want to use. This will traverse the RTL design database and change the hardcoded paths for PERL to point to the path you have specified. Once PERL is successfully installed, run `lconfig` to verify the setup. See Section 1.6, Running Lconfig.

1.6 Running `Lconfig`

`Lconfig` is the configuration utility that is used to properly configure the RTL design database for a given processor configuration. If this is the first time to run `lconfig` for a new installation, it is good measure to use one of the `lconfig` forms provided with the release to configure and test the design database for the first time. The provided `lconfig` forms can be found at `$LX_HOME/user`. To run `lconfig` perform the following steps:

cd $LX_HOME/regression

../bin/lconfig -help

This will show the available `lconfig` commands. More on this will discussed in Chapter 2, Using Lconfig.

To run `lconfig` with a customer specific form or with one of the provided forms, run the following:

../bin/lconfig ../user/<myform.form>

As `lconfig` processes the `lconfig` form and configures the database, it will print various messages on the screen. As long as no ERROR messages occurred, the `lconfig` process was a success. After configuring the design database, use `rundvt` to run one of the provided tests to verify database integrity. See Chapter 10, Using the Rundvt Regression Environment.

1.7 Installing the LSDK

In order to run simulations on the RTL design database using `rundvt`, the Lexra software developers kit (LSDK) must be installed. This software kit provides the necessary tools to compile, assemble and link software tests for running on the verilog RTL. To check what LSDK version is required for the given version of the RTL design database, see the README file described in Section 1.4, Installing the RTL Design Database.

To get the LSDK compressed tar file, log on to the Lexra FTP server and proceed to the directory `/releases/LSDK/<lsdk_version>`. Below are examples of what you might find in the `<lsdk_version>` directory:

`README.txt`	Release notes
`lsdk-i586-Linux-2.3.3.tar.bz2`	Linux version of LSDK (bzip format)
`lsdk-i586-Linux-2.3.3.tar.Z`	Linux version of LSDK
`lsdk-sparc-solaris2-2.3.3.tar.bz2`	Solaris version of LSDK (bzip format)
`lsdk-sparc-solaris2-2.3.3.tar.Z`	Solaris version of LSDK

Read the contents of the README file to find out any special requirements for this version of the `LSDK`. Then download the appropriate file. Decide where you want the lsdk installed (`<mydir>`) and run the following command:

zcat <archive>/lsdk-platform-x.x.x.tar.Z | tar xvf -

Substitute the operating system that you are using with platform and change x.x.x to reflect the real version number, i.e. sparc-solaris2. This will create a directory called lsdk-platform-x.x.x. After untarring the compressed file, add the following to your C shell `.cshrc` file:

setenv LSDKDIR <mydir>/lsdk-platform-x.x.x

setenv LSDKHOST sparc-solaris2

setenv MANPATH $LSDKDIR/man:$MANPATH

set path = ($LSDKDIR/$LSDKHOST/bin $path)

After adding the above to your `.cshrc` file, source it and verify that the setup was correct by running the following commands:

which make

You should see that the make utility found is the one in the LSDK release (`$LSDKDIR/$LSDKHOST/bin`). If the **make** found in your path is not the one in the LSDK installation directory, `rundvt` will fail when it tries to compile code. Correct the path so that everything looks good, then go to the Section 10.1, Rundvt Simulators to check that all tools are able to work with each other.

1.8 Running `Rundvt` for the First Time

For a final verification that the database is correctly installed, run the program `hello.c` on the verilog RTL of the processor. As a note, `hello.c` is one of the c tests that can be found in the `$LX_HOME/tests` directory.

cd $LX_HOME/regression

rundvt -help

This will show you all of the `rundvt` options available to you. See Section 10.3, Using the Command-line Options for more information on the use of these options.

Continue by running one of the following commands:

rundvt hello (If VCS is the default simulator)

rundvt -sim ncv hello (if NC-Verilog is the default simulator)

rundvt -sim vxl hello (if Verilog-XL is the default simulator)

If the simulation runs successfully, you will see output similar to the following when the simulation finishes:

```
INFO: rundvt simv -l vrun.log
Notice: timing checks disabled with +notimingcheck at compile-time
Chronologic VCS simulator copyright 1991-2000
Contains Synopsys proprietary information.
Compiler version VCSi 5.2R9; Runtime version VCSi 5.2R9; Jun 6 18:30 2001
Starting to read tests
Hello Lexra!
   49277500 M000>>>        ../tests/obj/hello.00400000.bin (0 NOPS) PASSES
$finish at simulation time             49277600
          V C S   S i m u l a t i o n   R e p o r t
Time: 492776000 ps
CPU Time: 21.210 seconds; Data structure size: 28.6Mb
Wed Jun 6 18:31:22 2001
INFO: rundvt Results of simulation
INFO: rundvt   PASS: 1; FAIL: 0
INFO: rundvt Execution Complete
INFO: rundvt   Total PASS 1
INFO: rundvt   Total FAIL 0
```

If the simulation stops before the test has completed or the test runs but fails, look at the following files found in the regression directory for help in debugging the problem with the installation.

rundvt.log
vcompile.log

1.9 Understanding the RTL Design Database File Organization

The top level directory `$LX_HOME` in the distribution contains the following files:

`TAG`	Lexra internal database version number
`release_summary_xxx_x.x`	a list of each file and version in the release

and the following directories:

`atpg`	scripts and templates for ATPG
`bin`	scripts used to build Makefiles and symbolic links, Verilog preprocessor, and RTL configuration tool
`chip`	Verilog RTL code for chip-level modules (for example, lx_base used in regression tests, behavioral RAM models)
`cpu`	Verilog RTL code for processor module and submodules
`ejtag`	Verilog RTL code for EJTAG modules and submodules (optionally licensed)
`include`	Verilog include files
`lbc`	Verilog RTL code for LBC module and submodules
`lmi`	Verilog RTL code for LMI modules and submodules
`lx`	Verilog RTL code for the lx0, lx1, and lx2 processor layers
`macs`	Verilog RTL code for MAC modules and submodules (optionally licensed, required for RISC-DSP)
`regression`	`rundvt` script and associated files
`syn`	individual subdirectories for synthesized blocks with links to source code, generic synthesis scripts, Synopsys synthesis constraints, and Makefile for each block
`system`	Verilog RTL code for LX modules (includes behavioral models for RAM, CLK & reset buffers)
`testbed`	Verilog RTL code for associated test bench code
`tests`	tests to be run on the Lexra processor verilog model
`tlb`	Verilog RTL code for TLB MMU (optionally licensed)
`tm_lbus`	Verilog RTL code for LBC bus functional models
`user`	files that need to be modified by the user to describe the local tool and library environment and locations (includes `lconfig` form)

Each Verilog RTL file includes only one module. The filename for module `<module_name>` is `<module_name>.v`.

1.10 Customer Configurations

Once you have verified proper installation, the environment is ready to generate design objects for the configuration and technology you have specified. Subsequent chapters describe each step below in more detail.

- generate a configuration using `lconfig` Chapter 2
- use the lx0,1,2 design hierarchies Chapter 3
- interface the RAMs Chapter 4
- use a bus controller Chapter 5
- add a custom engine interface Chapter 6
- add a coprocessor Chapter 7
- use the EJTAG option Chapter 8
- add testability Chapter 9
- run regressions (RTL & gates) Chapter 10
- synthesize Chapter 11
- simulate Chapter 12

Chapter
2

Using `Lconfig`

2.1 Overview

The `lconfig` utility supplied by Lexra configures the RTL design database. This includes: RTL code, simulation environment, and synthesis scripts.

This chapter describes how to use `lconfig`. It does not describe specific processor configuration options. `Lconfig` generates blank forms, which document these options.

`Lconfig` is a PERL 5 script providing an easy to use form based method for configuring the RTL design database. You fill out a form and process it with `lconfig` to accomplish the tasks listed below. Forms include documentation that describes all available options. The option defaults may meet your needs.

`Lconfig` lets you do the following:

- Configure features such as cache sizes, optional coprocessor interfaces, custom engines, and system bus attributes
- Generate an include file that tailors the RTL code and top level integration. Lexra's RTL code, which uses the symbol definitions to manage its configuration-dependent features, includes this file. Lexra's `rundvt` script also examines this file to determine what simulation tests must be executed to verify the configuration.
- Create Verilog module lists for RTL simulation
- Generate makefiles to control synthesis

The `lconfig` PERL script is located in the $LX_HOME/bin directory. To use it, you must have PERL installed on your system. See Section 1.5, Configuring for PERL.

2.2 When to Execute `Lconfig`

`Lconfig` creates files that tailor the simulation and synthesis environments to your configuration. You must run `lconfig` at least once before performing simulation and synthesis tasks. Because it creates files that are directly used for these tasks, do not execute `lconfig` when simulation or synthesis is in progress.

After you have settled upon a configuration and started synthesis, do not run `lconfig` to change the configuration unless you are willing to resynthesize. The makefiles generated by `lconfig` include dependencies that rebuild the required modules the next time you synthesize.

2.3 Running `Lconfig`

The execution of `lconfig` should be done from the $LX_HOME/regression directory. You will find preconfigured `lconfig` forms supplied by Lexra in the $LX_HOME/user directory. The user directory is a good place to keep application specific forms.

`Lconfig` has many command line options. For example: generating a blank form, copying an old form to a new one and building clean RTL files.

A blank form is the starting point for your custom configuration of the Lexra processor. It includes detailed documentation of all configurable features with working default values assigned to each feature.

To get a blank configuration form, type:

cd $LX_HOME/regression

../bin/lconfig -blank_form ../user/<my.form>

where `my.form` is the name of the output file you wish `lconfig` to produce.

`Lconfig` does not overwrite an existing file when it creates blank forms. Instead it reports an error.

After creating a blank form, review it and make changes from the default values to satisfy your particular needs.

Next, process the configuration form with the command

../bin/lconfig ../user/<my.form>

where my.form is the name of your form.

When creating forms, `lconfig` always writes the output form in the current working directory unless given a path to another subdirectory. When processing forms, `lconfig` writes the required output files into the appropriate project subdirectories. See Section 1.6, Running Lconfig.

New releases of the RTL design database can include configuration options not available in prior releases. When new options are added, `lconfig` assigns an appropriate default. This allows new releases of the RTL design database to process old user forms.

`Lconfig` always issues a warning when assigning a default value. To avoid these warnings add the default assignment or some other assignment to the input form. `Lconfig` facilitates this by including a command that copies an old form to a new form (-copy_form), preserving assigned values in the original form and assigning default values to unspecified options. By doing this, you will be able to view the documentation that comes with the new configuration option.

../bin/lconfig -copy_form ../user/<input_form_name> ../user/<output_form_name>

Always compare the input form and output form to ensure that `lconfig` has kept old values and that the assignments for new options are acceptable.

2.4 `Lconfig` Forms

The blank forms produced by `lconfig` are not actually blank, they are just a default starting point for system configuration. `Lconfig` specifies each configurable feature in its own section. Here is a sample section.

```
////////////////////////////////////////////////////
//
// DCACHE -- data cache size
//
// configuration choices: NONE 64K_1 32K_1 16K_1 8K_1 4K_1 2K_1 1K_1
//
//   "NONE"  -- no data cache
//   "64K_1" -- 64K byte direct mapped data cache
//   "32K_1" -- 32K byte direct mapped data cache
//   "16K_1" -- 16K byte direct mapped data cache
//    "8K_1" -- 8K byte direct mapped data cache
//    "4K_1" -- 4K byte direct mapped data cache
//    "2K_1" -- 2K byte direct mapped data cache
//    "1K_1" -- 1K byte direct mapped data cache
//
// The following settings are required when DCACHE = NONE:
//  MEM_GRANULARITY = BYTE
//
// default: DCACHE = "2K_1";
//
////////////////////////////////////////////////////
DCACHE = "2K_1";
```

Comment lines begin with // characters. A blank form includes comments describing, at a minimum, the legal settings for an option and the default value. For more complex options, `lconfig` supplies additional documentation. When `lconfig` processes a form, it does not interpret the comments. A default assignment follows the comment block for a given setting. You may edit the form to change these values.

Carefully review all of the documentation provided in a blank form to determine the configuration settings your application requires.

Throughout the documentation supplied by `lconfig`, the word module means a configurable feature. This is more general than the Verilog module keyword. For `lconfig`, a module is any configurable feature that is conveniently named and given a list of possible choices. More than one processor Verilog module may be affected by the choice for a single `lconfig` setting.

2.5 Forms Supplied by Lexra

Lexra supplies several pre-configured forms in the RTL design database. These are in the `$LX_HOME/user` directory. Do not use them as a starting point for configuring your version of the processor. Generate a blank form for that purpose. Rather, consult these files for useful regression test configurations and examples of how to configure the Lexra processor in a full system environment. Also use these files to verify that the initial installation of the RTL design database was successful. Here is a list of forms you might find in this directory:

`lx4x80.form`	a basic LX4x80 configuration w/ no MAC, no EJTAG, in an environment with memory models to test it
`lx4x80_mac.form`	a basic LX4x80 configuration MAC support, no EJTAG, in an environment with memory models to test it
`lx4x80_ej.form`	a basic LX4x80 configuration with no MAC, EJTAG support, in an environment with memory models to test it

The reference configurations above highlight the minimal stand-alone chip level simulation testbed provided with the Lexra processor, which employs a stand-alone chip and minimal memory system model. Take the time to look through a blank `lconfig` form so that you get a chance to read the documentation that explains each available processor option.

In any of the reference configurations above, you can fully synthesize all the design modules at or below the level of the lx1 module supplied by Lexra. See Chapter 11, Synthesizing the Lexra CPU,. Lexra supplies modules outside of this hierarchy for simulation purposes only.

2.6 Files Produced by `Lconfig`

When processing a form, `lconfig` creates the following files in the RTL design database.

include/lxr_symbols.vh	include file with define symbols
chip/sram_<type>_<data type>_<depth>x<width>.v	behavioral RAM models
regression/*.inpfiles	list of RTL & testbed source files to include in simulation
syn/syntrol/Makefile	synthesis makefile

See more detailed descriptions of these files in the sections below.

Never directly modify the contents of any these files. To ensure consistent simulation and synthesis with the RTL code, `lconfig` must produce these files.

2.6.1 lxr_symbols.vh

This file contains many defines. They control aspects of the Lexra processor RTL configuration and simulation testbed setup. The file has the following sections:

- summary of the configuration choices
- summary of RAM requirements
- configuration independent symbols
- configuration dependent symbols

Here is an example RAM summary provided by `lconfig`:

```
// RAM requirement summary
//
// MODULE  CONFIG  DEPTH  WIDTH PORTS RAM                   QTY  USED FOR
// ======  ======  =====  ===== ===== ===                   ===  ========
// ICACHE    1K_2    128 x   32   1   sram_ic_data0_128x32   1   data store set 0
// ICACHE    1K_2    128 x   32   1   sran_uc_data1_128x32   1   data store set 1
// ICACHE    1K_2     32 x   24   1   sram_ic_tag0_32x24     1   tag store set 0
// ICACHE    1K_2     32 x   26   1   sram_ic_tag0_32x26     1   tag store set 1
//                                                              and LOCK/LRU flags
// DCACHE    2K_1    512 x   32   1   sram_dc_data_512x32    1   data store
// DCACHE    2K_1    128 x   22   1   sram_dc_tag_128x22     1   tag store
```

Following the documentation section in the `lxr_symbols.vh` file, there are some symbol definitions that are independent of the configuration. These are the same for every `lconfig` execution.

Following these are a set of sections, one for each configurable item in the processed form. These can be very simple, as in this example.

```
// SEN_BUFFERS = "NONE" -- Do Not Buffer Scan Enable

  `define VPP_POP_NO_SENBUFS
```

Other items result in several to many symbol definitions for tailoring the RTL code.

After all of the sections for the configurable features, there are some more pre-determined definitions. Some of these may rely on configuration dependent definitions in the middle section of the `lxr_symbols.vh` file.

2.6.2 chip/sram_<type>_<data_type>_<depth>x<width>.v

`Lconfig` automatically generates behavioral ram files. They are listed in the `lxr_symbols.vh` file, with the specific sizes required for the chosen configuration. `Rundvt` uses them for RTL simulation. `Lconfig` puts the models in the chip project sub-directory, The `rundvt` script uses them when it starts the RTL simulation via the file `regression/lx2.inpfiles`.

For synthesis, you must supply Verilog wrappers with the names of the modules generated by `lconfig` and put them in the `chip/<technology>` directory. You specify `<technology>` through the `lconfig TECHNOLOGY` variable. You must supply the wrappers with the same port list as the behavioral models generated by `lconfig`, but internally the wrappers instance an application-specific RAM. To help you write wrappers, we provide an example wrapper in the `chip/custom/tsyncram_example` file.

2.6.3 regression/*.inpfiles

The *.inpfiles `lconfig` produces contain lists of Verilog files needed for simulation with the `rundvt` script. The `rundvt` script, which is also in the regression directory, passes these lists to Verilog. The different inpfiles separate the source files into groups according to basic code category. `Lconfig` lists only the files required for the specific configuration. Below is a list of some of the inpfiles created by `lconfig`:

`lx0.inpfiles`	Files that make up the processor, local bus memory controllers, co-processor interface, and custom engine interface. All of the modules in this list are ultimately instanced as submodules in a module named lx0.
`lx1.inpfiles`	system bus interface, MAC, EJTAG, the integration layer connecting these modules to the lx0 module
`lx2.inpfiles`	RAM models, the integration layer connecting RAMs to the lx1 module
`testbed.inpfiles`	Simulation testbed support: system memory transactors and bus monitors, for example

You can specify additional application specific files on the `rundvt` command line to make them available to the simulation.

2.6.4 syn/syntrol/Makefile

This is the main `Makefile` that controls the synthesis process. For every synthesized block in the RTL design database, there is a subdirectory in the syn directory used for synthesis. `Lconfig` writes a `Makefile` in each sub-directory, tailored as needed to support the specific configuration. This `Makefile` will work with the `syn/syntrol/Makefile` to perform the bottom-up synthesis of the Lexra processor.

The syn/syntrol/Makefile contains commands to synthesize the various blocks in the processor. It also contains hierarchical **make** commands to build sub-blocks. You can synthesize the entire design by executing **make** from within the `syn/lx2` directory.

2.7 Diagnostic Messages

Before processing your request, `lconfig` validates its command line to ensure it is properly formed. It also cross-checks information supplied in your input form with its internal database and performs purely internal consistency checks as it runs.

This section contains a complete listing of the diagnostic messages displayed by `lconfig`:

2.7.1 Notice Messages

These are messages containing useful information.

reading <form_name>

Specifies the file being read by `lconfig`

writing <dir>/<filename>

Specifies the file being written by `lconfig`

generating behavioral RAM model <dir><model_filename>

`Lconfig` has created a behavioral RAM model based on configuration requirements

generating <file>

Creating files used by Lexra software tools:

The `cvtlconf` utility is used to parse the `lconfig`.form to generate the `.asym_config` file used by the ASYM version of the instruction set simulator (`ISS`). If not using the ASYM ISS, ignore the WARNING that the `cvtlconf` utility cannot be found.

The `bin/common_symbols` script is used to generate common symbol files used by Lexra's simulation testbed, PERL scripts, and regression tests. This ensures that all configuration dependencies needed for the regression suite are consistently represented in the include files used by these programs.

making technology link <link>

Creating needed links for proper database setup. For example, when a vendor specific ram model is placed into the `chip/<technology>` directory, a link will be created to the `syn/lx2` directory.

using technology specific file <dir>/<technology>/<filename> for synthesis

`Lconfig` has selected a technology specific file for synthesis in place of the normal RTL behavioral file. `Lconfig` searches the directory `<dir>/<technology>` to find replacements for behavioral verilog models found in the `<dir>` directory.

preparing atpg related files

`Lconfig` modifies files related to atpg for the specific processor configuration as specified by the `lconfig` form.

preparing synthesis <dir><Makefile>
calling make to generate symbol reference files in <dir>
building per-block include files in <dir>

`Lconfig` generates `makefiles`, symbol reference files and include files to be used in the synthesis process.

lconfig finished

`Lconfig` has completed configuration of the RTL design database.

2.7.2 Warning Messages

These messages indicate problems you may need to correct. `Lconfig` still produces useful output files, however.

no configuration value for <module>, using default: <config>

The input form did not specify a required configuration. `Lconfig` will use the default value.

default values were used for <list> options

`Lconfig` used one or more default values

CE0=<config0> AND CE1=<config1> supported for RTL testing only

Lexra does not support the chosen CE0/CE1 configuration for synthesis. We allow the configuration because it is useful for RTL testing.

CE0=NONE AND CE1=EXPORT requires HI/LO to be implemented in CE1

No module is declared for CE port 0, and a module you have supplied is declared for CE port 1. Therefore, you must design your CE module to respond to the MFHI, MFLO MTHI and MTLO instructions. See Section 6.2.1, Instancing Custom Engines.

CE0=<config0> AND CE1=EXPORT requires HI/LO to NOT be implemented in CE1

A Lexra module responding to the MFHI, MFLO MTHI and MTLO instructions is declared for CE port 0, and a module you have supplied is declared for CE port 1. Therefore, you must design your CE module not to respond to the MFHI, MFLO MTHI and MTLO instructions. See Section 6.2.1, Instancing Custom Engines.

when LBC_SYNC_MODE is SYNCHRONOUS, LBC_RBUF need not be greater than 2

The combination of LBC_SYNC_MODE set to "SYNCHRONOUS" and LBC_RBUF set greater than 2 results in rarely used read buffer entries, wasting chip area. Set LBC_RBUF to 2 for this case.

LBC_RBUF is set to more than twice the line size and thus will waste area

Setting LBC_RBUF to more than twice LINE_SIZE results in unused read buffer entries, wasting chip area. Decrease the read buffer size to save area.

<ram_model> requires technology specific wrapper <dir>/<technology>/ <ram_model> for synthesis

A technology specific RAM wrapper of the specified width and depth is required for synthesis, but was not found in the expected location

<dir>/<filename> requires technology specific version <dir>/<technology>/ <filename> for synthesis

A technology specific file is required for synthesis, but was not found in the expected location. See the RTL code in `<dir>/<filename>` for the required module function. Module functions are usually simple one-gate functions that you must choose manually to ensure proper implementation. The missing files do not prevent RTL only simulation, as the RTL code always provides a generic functional equivalent in the `<dir>/<filename>` file.

missing technology specific files will cause synthesis problems.

Previous error messages identified one or more required technology specific files as missing. Absence of these files will result in synthesis errors. The RTL simulation is still valid, however, because `rundvt` uses generic RTL versions of the required technology specific files for RTL simulation.

2.7.3 Error Messages

These messages indicate problems that you need to correct before `lconfig` can produce useful output files.

<module> is not a configurable feature

An unknown configurable feature appears on the left side of "=" in the form. Check the feature name.

<config> is not a valid configuration option for <module>

An unknown value appears on the right side of "=" in the form. Check the list of legal values in the form.

more than one configuration option specified for <module>

A configurable feature appears on the left side of more than one "=" in the form. Delete the unnecessary feature name.

<module1> = <config1> requires <module2> = <config2>

The configuration value you chose for `<module1>` requires a specific value for the configuration of `<module2>`, but the form specifies some other value. Select the value specified for `<config2>`.

<config> is less than allowed minimum <min> for <module>
<config> is greater than allowed maximum <max> for <module>

A numerical configuration value is outside the allowed range

BASE[9:0] must be zero (BASE=<value>)
TOP[3:0] must be 1111 (TOP=<value>)
BASE[31:16] and TOP[31:16] must have same value (BASE=<value1>, TOP=<value2>)
BASE[15:4] must be less than or equal to TOP[15:4] (BASE=<value1>, TOP=<value2>)

The address range specification for IMEM, IROM or DMEM does not conform to configuration rules in the form. Review the rules and correct the range.

range for <module1> overlaps with range for <module2>

The address ranges for local RAMS (IMEM, IROM, DMEM) overlap in the form. Correct the ranges so they don't overlap.

required RTL source file <filename> does not exist

A Lexra supplied source file required by the configuration is missing. This could indicate improper installation or accidental file deletion.

2.7.4 Abort Messages

`Lconfig` is unable to execute the requested command.

file name required for blank form output

The `-blank_form` command-line option does not specify an output file name. Specify an appropriate output file.

invalid option <-option>

The command line has an unknown -option

too many arguments

The command line has extraneous arguments

output file <formname> already exists

The `-blank_form` command line option specifies an output filename already in use.

input file <formname> does not exist

The input form specified on the command line does not exist

cannot find include and regression directories

`Lconfig` ran from an improper location. Always run `lconfig` from the `regression` directory

arguments required

`Lconfig` ran without arguments

input file name required for source form
output file name required for destination form

The `-copy_form` option lacks the proper command line arguments. The syntax is

lconfig -copy_form <input_form_name> <output_form_name>

2.7.5 Internal Messages

If `lconfig` halts with an INTERNAL error message, it has detected an internal inconsistency. Lexra tests `lconfig` with randomly generated input forms to ensure that you do not encounter these messages. If you do see such a message, please contact your local Lexra Application Engineer.

Chapter
3

RTL Organization

3.1 Building Blocks

The Lexra processor includes the processor core as well as a number of interface modules. Connect your surrounding logic to the processor through these interface blocks to architect your system.

The block diagram below shows an example of the processor complex. The major functional blocks include:

- processor core
- local memory interfaces (LMIs)
- coprocessor interface(*s*) (CIs)
- custom engine interface (CEI)
- Lexra bus controller (LBC)

In addition, you can specify and configure optional, product dependent, modules that are placed in the `lx1` level of the processor hierarchy (i.e. EJTAG, MAC, TLB, LBC.) Connect the processor complex to the rest of your design through ports associated with the various interface blocks in the RTL. Successive chapters of this guide explain how to use these interface modules.

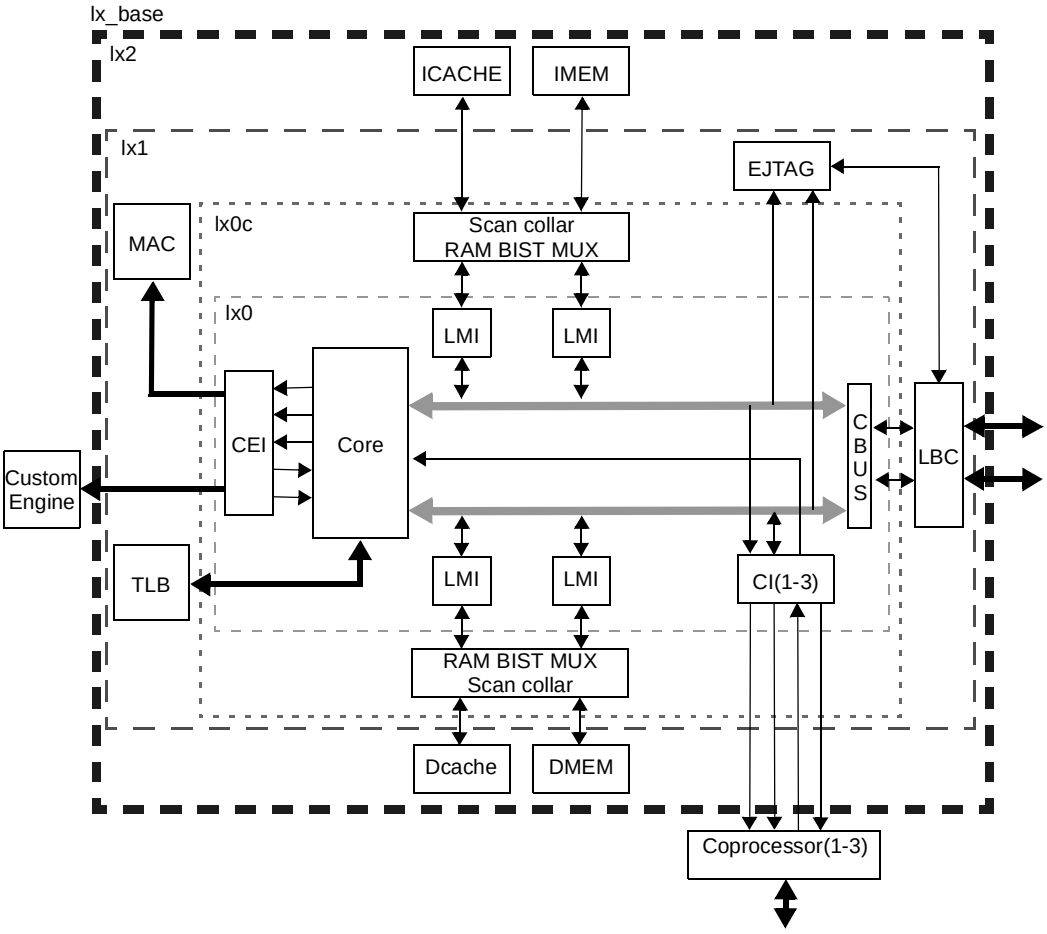


Figure 3-1. Processor Block Diagram

3.2 Using the Lx0/Lx0c/Lx1/Lx2 Design Hierarchies

To help you integrate Lexra processors into your design, we have organized the hierarchy into a set of module layers. The organization provides an efficient means to incorporate the processor into an application specific design.

The block diagram above highlights the four module layers in the processor hierarchy. The innermost layer, `lx0`, contains the core processor complex. You

should never need to touch it. Layers `lx0c` and `lx1` let you select Lexra-supplied product options. The hierarchical topmost layer `lx2` lets you connect standard SRAMs to the processor memory interfaces. Anything above the `lx2` layer is design specific.

You never need to change the RTL source file used by the `lx0`, `lx0c`, or `lx1` layers. `Lconfig` configures and manages them. Further, if you follow the default RAM integration methods described below (see), you won't need to edit the `lx2.v` RTL source file either. See Chapter 4, Local Memory.

3.2.1 Module Definitions

The `lx0` module contains the core of the processor. The register file, interrupt logic, pipeline and decode logic all reside here.

The optional `lx0c` module supplies scan collar isolation of the `lx0` module. This layer is present if you set SCAN_SCL = YES or RAM_BIST_MUX = YES in your lconfig form. It contains an instance of the `lx0` module, scan isolation logic as well as RAM test muxes. The test muxes can be used for BIST testing or as a point of entry into the RAMs by a DMA engine.

The `lx1` module has instances of `lx0` (or `lx0c`) and optional modules we supply in RTL form, such as the MAC and EJTAG. It also passes the RAM, LBC, Custom Instruction and Coprocessor ports up to the next layer as needed.

The `lx2` module has instances of `lx1` and all RAMs required to support the LMIs. As long as no customization of the interfaces to the RAMs is required (i.e. custom DMA ports or third-party RAM BIST), then this layer uses standard interconnect and RAM wrappers. It also passes the LBC, Custom Instruction and Coprocessor ports up to the next layer as needed.

We supply a module `lx_base`. It instances `lx2` and other testbed modules like `copstub` (an example for the coprocessor interface) and `ce_dvt` (example custom instructions). This level of the hierarchy may be a useful bare-bones starting point for your design, in that it includes wire declarations and an instance of the `lx2` module. The port list of this module is designed to work with Lexra's simulation environment and is probably not directly useful for most applications.

Chapter
4

Local Memory

This chapter outlines different features and options for Lexra's RISC & RISC-DSP internal memory system. The main memory system (R3000 4GB memory space) on the external Lexra bus is part of the chip architecture and is not covered here.

Lexra processor's internal bus is based on the Harvard bus architecture. This specifies that there is one instruction bus and one data bus. The architecture allows Lexra to support a flexible internal memory system with many different memory configurations. These include different memory types, namely cache memories, RAMs and ROMs, as well as different memory sizes. In addition, Lexra processors can support both byte writable as well as word writable RAMs. When using word writable RAMs, byte and half-word stores will cause the processor to perform read-modify-writes.

Using the `lconfig` tool and your `lconfig` form, you can specify the exact configuration of your processor. Once you've specified the type, size and address location of the caches, RAMs and ROM in your design, `lconfig` will generate generic memory models to allow you to simulate your configured processor.

The generic memory models serve two purposes. First, `rundvt` can use them for system simulation before you decide on a specific memory technology to use. Second, when you do decide on your memory technology, you can replace the generic simulation models with a memory wrapper that will allow you to instance your own technology specific model.

Lexra processor's memory architecture also supports a number of runtime features such as flexible cache flushing options, DMA to internal memories, cache locking and support for memory testing.

The first four sections below outline how to configure the memory architecture and how to use the different memory models, while the following four sections discuss the runtime features of the memory architecture. The last section discuss product specific memory topics.

4.1 Memory Architecture and Configurability

4.1.1 Available Options

Memory Sizes and Memory Types

The two tables below show possible memory configurations.

ICACHE	IMEM	IROM
no	no	no
no	no	yes
no	yes	no
no	yes	yes
yes	no	no
yes	no	yes
yes	yes	no
yes	yes	yes

You can configure the ICACHE to be direct or two-way set associative.

DCACHE	DMEM	DROM
yes	no	no
yes	yes	no
yes	no	yes
no	yes	no
no	no	yes
no	no	no

The DCACHE is direct mapped.

See the product datasheet for supported memory sizes.

Cache Line Size

You can also configure the cache line size via the `lconfig` form. The cache line size is same for both the ICACHE and the DCACHE.

Changing the cache line size has an impact on system level performance and characteristics. If you change the size, consider such things as interrupt latency time, main memory characteristics, bus utilization and the like.

Latency of serving interrupts - The processor stalls while the memory system is serving a cache miss and, as a result, the processor can not service an interrupt in the period. Thus, the longer cache lines are, the longer it takes the processor to service a cache miss. Therefore, the processor takes longer to service an interrupt.

NOTE: If using MEM_FIRST_WORD = DESIRED, once the desired data word is received by the processor, the processor will continue to execute. Only if another access occurs on the data bus while the line is still being fetched, will the processor stall.

Main memory characteristics - Some memories have long setup times but once set up, they burst data very efficiently. Such memories are better suited for long cache lines.

Bus utilization - Long cache lines force the processor to occupy the bus for longer times for every cache miss. This may cause longer wait times for other devices trying to access the bus. On the other hand, memory systems transfer more data when there are longer cache lines. Depending on such software characteristics as locality of code and data, this may reduce the number of cache misses.

Memory Granularity

Lexra processors have two options relating to memory granularity. These are labeled: MEM_GRANULARITY and LMI_DATA_GRANULARITY. Both of these are configured via the lconfig form. MEM_GRANULARITY refers to the ability of the system bus to perform word or byte accesses and LMI_DATA_GRANULARITY refers to the local RAMs ability to perform the same functions.

When MEM_GRANULARITY = WORD, byte and half-word store instructions cause a read-modify-write transaction to occur on the system bus. A store half-word instruction to byte address 0x02, for example, will cause the processor to read the word at location 0x00, modify the upper half of the word, and write the entire word back across the bus to address 0x00. In this manner, the processor is

ensuring that the processor's system bus controller only performs word accesses on the system bus.

When MEM_GRANULARITY = BYTE, both byte and half-word store instructions will cause the system bus controller to perform byte and half-word accesses on the system bus. Thus, a store half-word instruction to byte address 0x02, for example, will cause the processor to issue a half-word write access on the system bus to address 0x02. In this sort of system, bus peripherals must be designed in a way to understand and accept byte and half-word access.

NOTE: Even if MEM_GRANULARITY=WORD, byte and half-word accesses to uncacheable space will cause byte and half-word transactions to occur on the system bus. Uncacheable address space (KSEG1) is commonly mapped to hardware control registers such that unsolicited reads (from a read-modify-write) could cause problems with hardware control.

When LMI_DATA_GRANULARITY = WORD the local SRAMs (DCACHE and DMEM) are specified as being word accessible. In this case, byte and half-word writes to the data memories cause a read-modify-write to occur. This is a two cycle process.

When LMI_DATA_GRANULARITY = BYTE the local data SRAMs are specified as being byte accessible. In this case the RAMs must have write enables for each of the byte lanes. In this configuration, byte and half-word writes are written to the RAMs in a single cycle.

The following table shows the possible combinations of the granularity settings

MEM_GRANULARITY	**LMI_DATA_GRANULARITY**
word	word
byte	word
byte	byte

For the best performance, set both MEM_GRANULARITY and LMI_DATA_GRANULARITY to BYTE. Otherwise, the following talks about the trade-offs:

MEM_GRANULARITY = WORD, LMI_DATA_GRANULARITY=WORD

Byte and half-word accesses to the local data memories require two cycles to complete (read-modify-write). Byte and half-word accesses to memory situated on the system bus require the processor to first read the word, modify it, then write it back. While the initial read occurs, the processor is stalled. For systems

with heavy bus usage, this may significantly impact performance. Once the word has been read and modified, the processor will continue to operate as soon as the word to be written is accepted by the system bus.

MEM_GRANULARITY = BYTE, LMI_DATA_GRANULARITY=WORD

Byte and half-word accesses to the local data memories require two cycles to complete (read-modify-write). Byte and half-word accesses to memory situated on the system bus are finished by the processor as soon as the system bus can accept the data to be written. This frees the processor to continue processing while the system bus handles the transfer.

MEM_GRANULARITY = BYTE, LMI_DATA_GRANULARITY=BYTE

Byte and half-word accesses to the local data memories require a single cycle to complete. Byte and half-word accesses to memory situated on the system bus are finished by the processor as soon as the system bus accepts the data. This frees the processor to continue processing while the system bus handles the transfer. In a system with a need for many byte and half-word accesses, this will be the highest performing solution.

4.1.2 IMEM and DMEM Controllers

Optional IMEM and DMEM allow efficient storage of frequently used or timing critical code and data (code for exception handlers for example). You configure the address ranges of the RAMs using the `lconfig` form. `Lconfig` then sets the address range to be hardwired internal to the processor or exported to the `LX2` port list at synthesis time. These RAMs need no tag storage.

The RAM controllers work together with the cache controllers so that an address claimed by the RAM controller does not trigger the cache controller. Consequently, the RAM, like a scratch pad RAM, works much like an add-on cache with fixed address range and single cycle access times.

The DMEM controller handles an address within its range as an uninitialized data section. Thus, the only time the DMEM controller initiates a write to the DMEM is when explicitly told to do so by the program, that is, when the processor executes a store instruction within its configured range. Furthermore, the DMEM controller does not examine any valid bits. Therefore a read from a DMEM address not written to previously returns undefined data, just like any uninitialized data section.

You can configure the address range of the IMEM and DMEM to be within all segments, that is, within, kuseg, kseg0, kseg1 or kseg2 using the physical addresses that these ranges define.

For more information on how the cache and RAM controllers operate, please read the product datasheet.

4.1.3 Configuring the Memory Architecture

The exact RAMs required for a processor application depend on how you configure optional features with Lexra's `lconfig` utility. Here is a sample `lconfig` form entry for configuring ICACHE.

```
////////////////////////////////////////////////////
//
// ICACHE -- instruction cache size
//
// configuration choices:  NONE 64K_2 32K_2 16K_2 8K_2 4K_2 2K_2 1K_2
//                         64K_1 32K_1 16K_1 8K_1 4K_1 2K_1 1K_1
//
//       "NONE"  --  no instruction cache
//      "64K_2"  --  64K byte 2-way set associative instruction cache
//      "32K_2"  --  32K byte 2-way set associative instruction cache
//      "16K_2"  --  16K byte 2-way set associative instruction cache
//       "8K_2"  --  8K byte 2-way set associative instruction cache
//       "4K_2"  --  4K byte 2-way set associative instruction cache
//       "2K_2"  --  2K byte 2-way set associative instruction cache
//       "1K_2"  --  1K byte 2-way set associative instruction cache
//      "64K_1"  --  64K byte direct mapped instruction cache
//      "32K_1"  --  32K byte direct mapped instruction cache
//      "16K_1"  --  16K byte direct mapped instruction cache
//       "8K_1"  --  8K byte direct mapped instruction cache
//       "4K_1"  --  4K byte direct mapped instruction cache
//       "2K_1"  --  2K byte direct mapped instruction cache
//       "1K_1"  --  1K byte direct mapped instruction cache
//
// default: ICACHE = "2K_1";
//
////////////////////////////////////////////////////


ICACHE = "1K_2";
```

After filling out and processing a configuration form with `lconfig`, you may examine the RAM documentation section in the `include/lxr_symbols.vh` file created by `lconfig`.

Here is an example of this documentation:

```
// RAM requirement summary
//
//  MODULE  CONFIG  DEPTH  WIDTH  PORTS  RAM                  QTY   USED FOR
//  ======  ======  =====  =====  =====  ===                  ===   ========
//  ICACHE  1K_2      128 x   32    1    sram_ic_data_128x32   2    data store
//  ICACHE  1K_2       32 x   24    1    sram_ic_tag0_32x24    1    tag store
//  ICACHE  1K_2       32 x   26    1    sram_ic_tag1_32x26    1    tag store and LOCK/LRU flags
//  DCACHE  2K_1      512 x   32    1    sram_dc_data_512x32   1    data store
//  DCACHE  2K_1      128 x   22    1    sram_dc_tag_128x22    1    tag store
```

This shows that the configuration requires six RAM instances. The names of the Verilog modules that define these RAMs have the format sram_<type>_<data type>_<depth>x<width>. `Lconfig` automatically writes behavioral models for these RAMs, and places them in the `$LX_HOME/chip` subdirectory of the processor.

NOTE: If using LMI_DATA_GRANULARITY=BYTE the names for the RAM wrappers and behavioral RAM models will include a trailing _Xwe, where X is the number of write enable lines. For example, sram_dc_data_4096x32_4we.v.

NOTE: If using the LX8000, the RAM for the DMEM has two ports. This is to allow fast DMA of packet data into the local RAM (DMEM). In this case, the RAM will be labeled something like sram_2rw_dw_data_1024x64.

For synthesis, you must use application-specific RAMs that meet these requirements. You may use larger RAMs if the required size is not available. The synthesis wrapper for each RAM must, however, present the required depth and width at its interface, as described in Section 4.4, Using Library Vendors' RAM Models below.

4.2 Memory Requirements

As described in previous sections, the processor uses memories to implement instruction and data cache storage, instruction and data tag storage, instruction and data RAM and ROM storage. The processor can interface with very simple memories that requires a minimum of memory functionality. Section 4.2.1, RAM Function below outlines functional requirements. Section 4.2.2, RAM Timing below discusses timing requirements for optimal performance.

4.2.1 RAM Function

The Lexra processor interfaces to synchronous RAMs. The only signals it requires are address bus, write data bus, read data bus, write enable and clock. Optional signals are read enable and chip select. Depending on the setting of LMI_DATA_GRANULARITY the RAMs may or may not require byte access capability (i.e. write enables for each byte lane).

The standard RAM models and wrappers include active high and low control lines. The memory controllers drive these redundant control pins to let you configure wrappers for synchronous RAMs without adding logic in the wrapper.

If your RAM is asynchronous, you can easily make it synchronous by creating a wrapper that latches all of the input signals on the positive edge of the clock.

For write operations, all signals, address, write data, read enable, write enable and chip select are clocked on at the same positive edge of the clock.

For read operations: address, write enable, read enable and chip select are clocked into the RAM at the same positive edge of the clock. Read data is returned in the following cycle after a combinatorial read access delay.

See also the waveform in Section 4.2.2, RAM Timing below for memory access.

See the product datasheet for a list of the RAM interface signals.

The Lexra RTL does not support CLKN but supplies it in the wrapper for cases that need your intervention, for example if your RAM only supports negative edge clocking.

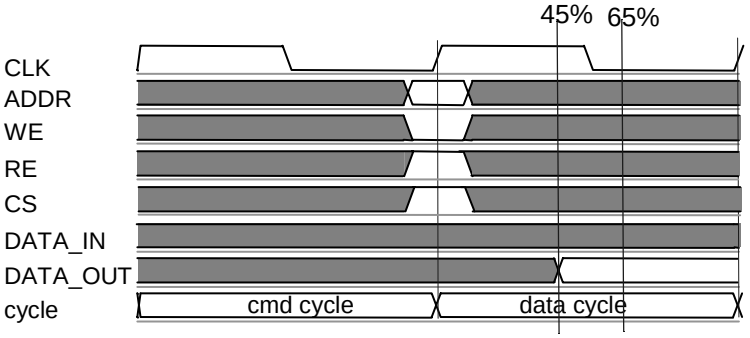
4.2.2 RAM Timing

A RAM access takes two cycles to complete, but the RAM may start a new access every cycle. The two cycles are the command cycle, and the data cycle.

In the command cycle, the processor supplies ADDR, WE, RE, CS and, if applicable, DATA_IN. The RAM device register these at the rising edge of CLK.

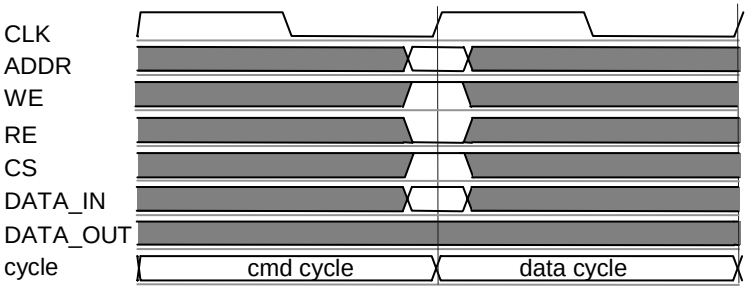
In the data cycle, the RAM performs a read or write operation, as indicated by the WE control it receives in the command cycle. For a read operation, read data flows out of the RAM during the data cycle without any intervening data registers. For a write operation, the RAM writes the RAM array. The processor does not observe DATA_OUT during the data cycle of a write operation.

The diagram below shows a RAM read access.



To achieve timing balance with the rest of the processor design, the RAM read data must be stable by the 45% to 65% mark in the processor cycle. For RAMs used as instruction or data stores, the processor uses the rest of the cycle to transfer the information from the caches to the processor core. For RAMs used as tag stores, the processor uses the rest of the cycle for tag comparison. Read access times longer than 45% to 65% of the processor cycle time may well require a decrease in the processor operating frequency.

The diagram below shows a RAM write access. The processor does not use DATA_OUT during the data cycle of a write operation.



4.2.3 Critical Paths Involving RAMs

In most cases, the RAM performance dictates the performance of the processor. This section contains guidelines on memory performance.

For most silicon technologies (groups of files for modeling foundry, process and library information), the most critical paths in the processor are the following four.

- Address to instruction RAMs: the path providing the next instruction address from the processor to the ICACHE or IMEM. The RAM's address register input setup time is in the critical path.
- Instruction to processor: the path providing the instruction from the ICACHE or IMEM back to the processor. The RAM's clock to data out latency is in the critical path.
- Address to data RAMs: the path providing the data address from the processor to the DCACHE or DMEM. The RAM's address register input setup time is in the critical path.
- Data to processor: the path providing the data from the DCACHE or DMEM back to the processor. The RAM's clock to data out latency is in the critical path.

4.3 Using Lexra's Generic RAM Models

`Lconfig` produces generic RAM models based on your selected memory architecture and memory sizes. It generates all simulation models necessary (I-cache, ICACHE tags, DMEM etc.) based on the settings in the lconfig form file.

The generated simulation models are in `$LX_HOME/chip` and the names of the generated models are `sram_<type>_<data type>_<depth>x<width>.v` (or with the trailing `_Xwe` if using byte accessible memories). These modules are connected to the processor, thus, no intervention is necessary. The models are instantiated in the `lx2` hierarchy. For further details, see Section 3.2, Using the Lx0/Lx0c/Lx1/Lx2 Design Hierarchies.

The generated models are intended for RTL simulation only. Do not use them for synthesis or sign-off simulation

4.4 Using Library Vendors' RAM Models

While the Lexra generic RAM models are well suited for early architecture exploration when your own RAM models might not even be available, you should not use them exclusively throughout the design cycle. At some point, use your library vendor's RAM simulation models instead.

To use vendor RAM models, write RAM wrappers with port lists that are compatible with `lconfig's` corresponding generic RAM models and name the files the same as they are named for the generic RAM models. Put the wrappers in the `chip/<technology>` subdirectory and rerun `lconfig`. This will allow `lconfig` to create links from the `syn/*` directories to the location of the RAM wrappers where the synthesis scripts will look for them. Your wrapper must provide all the required and optional signals for interfacing to the RAMs. If the RAM doesn't use the optional signals, simply leave them unconnected inside the wrapper.

See the `chip/<technology>` directory for an example RAM wrapper.

When simulating, ensure that the vendor RAM models instanced in the RAM wrappers are accessible. You do this, typically, by adding library entries to the `user/<technology>/gate.f` file. Also, in the file `regression/lx2.inpfiles`, be sure that all the RAM wrappers are listed so that the proper verilog files are read into the verilog simulation.

RTL simulation of the processor interfacing to vendor RAM models with timing requirements might result in simulation problems (i.e. timing checks must be disabled). See Chapter 12, Simulation Guidelines for further details.

Vendor Memory Initialization

The vendor RAMs that get connected to the ICACHE, DCACHE and IMEM interfaces must all be initialized to a non-X value. If they are not the uninitialized contents can cause problems in simulation. Lexra provides in their RTL environment a debug monitor (LMON) that will look for this condition. However, if the monitor in LMON is not being instanced, as in the case for gate level simulation, simulation may become corrupt. If this monitor warns that the RAM interface has gone X, usually this comes from either the RAM content being uninitialized or a system bus read that came back X.

To remedy the situation, the local RAMs connected to these interfaces must all be initialized to a non-X value. For the IMEM and ICACHE, it is best to initialize the RAM contents to an invalid opcode. If the processor unexpectedly executed one of these opcodes, a bad opcode exception trap would occur and simulation would fail.

For the DCACHE, initializing the RAMs to random data should suffice, for this may very well be what the real silicon will do. DMEM contents should be initialized by an application (i.e. via store instructions) before any data is read from it. If it is not done, X's may enter the processor data paths and cause simulation problems.

Uninitialized Memory Impact on Cache Controllers

The X's in the RAMs are a problem because the caches are permitted to return invalid instructions or data to the processor. If a miss occurs, the cache logic sends a signal to the processor in the next cycle to nullify the effect of having sent the wrong information. An X in the tag stores can cause the cache controllers to not even be able to determine if a miss occurred or not. Although the hardware contains state machines to write 0's to all tag stores upon reset, there is no guarantee that the processor state will not be corrupted when the tag RAMs are allowed to contain X before reset.

An X should not be in the instruction store because the processor performs a speculative decode of the instruction that it is sent. The nullification signal from the cache controller will make the logic do the right thing, but only if it is at some binary value. For the logic to support an X in an invalid instruction, the logic would have to be structured in a way to ensure that the X is gated out before it is sensed by the decode logic. In general this is possible for RTL, but this property is not always preserved once the design is mapped to gates by synthesis. And either way, the structure would have a very adverse impact on critical path timing.

An X in the data cache's data store is not wanted for a similar reason as the instruction store. The data that is read from the data cache might be needed to resolve a branch condition that is present in the D stage. The X in the logic can cause an X on the branch outcome. Even though the pipeline is being stalled, the X can still leak through the branch logic and corrupt the pipeline. So to ensure valid simulation, initialize the ICACHE, IMEM and DCACHE rams to values that are non-X.

4.5 Direct Memory Access to Internal RAMs

Devices on the external Lexra bus cannot snoop the internal instruction or data bus in order to access the internal memories. Instead, the Lexra processor provides two methods for direct access to internal memories. First, a request/ grant method has been created for an external device to get exclusive access to the internal memories using the same memory port that the processor uses. Second, any of the internal memories can be made with dual-ported memories so that an external agent can view & modify the memory contents without impacting the performance of the processor.

4.5.1 Using Request/Grant

You can use the DMA access method outlined below to DMA data or instructions into a local memory as well as allow an external agent to force cache invalidation upon the processor. Thus, you can use it to control data coherency in a multiprocessor environment.

Each ICACHE and DCACHE controller as well as each RAM controller provides a request signal that an external agent may use to signal that it wants access to the internal RAMs. The memory controller responds to a REQ signal asserted by the external agent by asserting a grant (GNT) signal. When the external agent detects the asserted GNT signal, it has exclusive access rights to the memory devices that the memory controller normally uses.

Note: Before the memory controller asserts grant, it may have to execute a transaction on the system bus. Make sure that the system bus is not held indefinitely or the memory controller may not assert grant.

The external agent may access the ICACHE store and tag, IMEM, DCACHE store and tag, DMEM, or any combination of these depending on which memory controller's REQ and GNT signals you have connected.

The processor stalls while the grant is asserted. As a result, the length of the DMA operation may affect system characteristics like maximum interrupt response time.

In order to give an external agent access to internal memory using this method, you must insert a 2-1 mux into the appropriate critical path. You can insert this mux, called the RAM BIST MUX, using the interface we describe in Section 9.3, RAM Testing.

DMA Priority

The external DMA engine is guaranteed to get first priority (i.e. first access) immediately after the processor comes out of reset. This enables the DMA to write to any memory before the processor starts executing. The DMA REQuest line must be asserted before the processor comes out of reset in order to guarantee this behavior.

4.5.2 Using Dual Ported Memories

Dual ported memories give you the best performance when doing DMA to the internal processor memories. For area conscious design, the area impact of a dual ported memory may make this feature unfeasible. Also, when using dual ported memories, care must be taken to ensure that the processor does not process data that is currently being modified by the external agent.

4.6 Invalidating a Cache

Lexra processors provides several different methods by which you can invalidate the caches or portion of the caches. Since different applications have different requirements, no method is better or worse than another. The sections below describe the different cache invalidation schemes and their limitations and the last section summarizes them.

4.6.1 Invalidating a Cache Completely

The coprocessor 0 has a control register called CCTL, general register address=20. Some bits in this register control the complete invalidation of the caches, one bit for the ICACHE and one bit for the DCACHE. Thus, you can control the instruction and data caches separately.

An assembler program, easily created, can set these bits to desired values. Once the processor detects that cache invalidation bits have been set, the processor stalls. The cache controller(s) invalidate(s) the ICACHE tag and/or the DCACHE tag, depending on the CCTL setting. The processor resumes once the caches have been invalidated.

The cache invalidation sequence takes one cycle per cache line. You can do instruction and data cache invalidation in parallel.

4.6.2 Invalidating a Cache Line with an Aliased Approach

You can also invalidate the DCACHE with one line granularity. Use the aliased approach to replace the cache line you want to invalidate with another cache line.

Since the DCACHE is directly mapped, you can invalidate the cache line of a particular address by loading an address with the same 10 lower address bits (for a 1Kbyte cache, for example). You can find such an address by adding the cache size to the address whose cache line you want to invalidate.

A disadvantage of this method is that it displaces the cache line at the indexed location even if it is not the target address.

4.6.3 Invalidating a Cache Line by Uncached Reference

If the address whose cache line you want to invalidate is within the kseg0 segment, you can invalidate it by making an uncached reference to the same physical address.

Lexra's simple memory manage unit (SMMU) provides a fixed memory mapping scheme. The processor maps the two virtual memory segments kseg0 (cached) and kseg1 (uncached) to the same physical memory. Furthermore, even for uncached references, the Lexra cache controller translates the uncached virtual address to a physical address and looks up the tag memory. If the address matches the tag, it invalidates the cache line.

Thus, if an address in kseg0 should be invalidated in the cache, it is enough to do a load operation from the corresponding address in kseg1.

The advantage of this method compared to the previous method is that it invalidates the line only if it is resident.

4.6.4 Invalidating a Cache by Using DMA

You can use the DMA access method described in Section 4.5, Direct Memory Access to Internal RAMs to invalidate the caches since the Lexra processor can grant access not only to the cache store, but also, to the cache tag. Using the DMA, an external agent can force cache invalidation upon the processor instruction and/or data caches.

A cache invalidation sequence looks like this:

1. The external agent asserts REQ to the appropriate cache.

2. Once granted access to the cache (by receiving an asserted GNT), the external agent reads some position in the cache tag memory. To ensure data coherency it determines if there is any data in the cache whose cache line you want to invalidate.

3. If there is, the external agent invalidates the appropriate cache line by resetting the valid bit for these lines.

4.6.5 Invalidating a Cache Using Multi-port Memories

This method requires that the cache tag memories be implemented with multi-port memories, or more precisely, two read ports and one write port.

This cache invalidation method is very similar to the one described in the previous section. However, having access to the second read memory port, the external agent does not need to use the DMA request in order to examine (read) the tag memory. It need only use the DMA channel when finding data in the cache whose cache line needs to be invalidated.

A cache invalidation sequence looks like this:

1. The external agent reads the cache tags when needed, using the second read port on the tag memory.

2. If it finds data in the cache that needs to be invalidated, it issues the REQ and invalidates the appropriate cache lines when receiving an asserted GNT.

4.6.6 Conclusion

The methods in the first three sections of 4.6 Invalidating a Cache differ from those the last two sections in that the first set of methods requires that the application running on the processor itself know that is has to invalidate the cache. It initiates the commands that invalidate the entire cache or portions of the cache. For some systems, this is not a limitation for others, it is unusable.

The last set of methods use an external agent to force cache invalidation upon the processor. The last method is more efficient in terms of system performance but it may require multi-port RAMs.

4.7 ICACHE Locking

If you configure the instruction cache for two-way set associativity, the Lexra processor lets you lock instructions into the ICACHE. You can force all fetched instructions to occupy set 1 of the ICACHE during execution of a critical code segment. Subsequently you can lock the instructions held in set 1 so that misses don't displace them. In this mode, the processor uses set 0 to service ICACHE misses.

The coprocessor 0 has a control register CCTL, general register address = 20. The CCTL[3:2] controls the ICACHE locking.

CCTL[3:2]	Functionality
00	No effect, normal operation
01	No effect, normal operation
10	forces all cache misses to occupy set 1
11	replaces cache lines in set 1 and locks it

To ensure that a critical piece of code fetched by the processor be stored and locked into set 1, modify your function with a small wrapper as follows:

```
Custom_function_wrapper
        Assembler code to set CCTL[3:2] = 10
        Custom_function( .....) ;
        Assembler code to set CCTL[3:2] = 11
End custom_function_wrapper
```

4.8 RAM Manufacturing or BIST Testing

The Lexra processor provides a mechanism for testing RAM at manufacturing time or using BIST machines. See Section 9.3, RAM Testing for more details.

4.9 LX4280 Memory Specifics

The LX4280 has a 64-bit instruction bus to accommodate the dual-issue pipeline. For consistency sake, the LX4280 also has a 64-bit interface to the data caches and DMEM. Thus all the memory interfaces/wrappers must be 64 bits. However, the memories themselves may be 32 bits (as in the LX4189) or 64 bits. In any case, there must be a separate WE signal (write enable) for the upper 32 bit memory (bit 63-32) and another WE for the lower 32 bit memory (bit 31-0).

Example:

A memory configured to be "8K_1" in the `lconfig` form (8 Kbytes direct mapped) will have the following depth and width:

1024x64	for LX4280
2048x32	for LX4189

4.10 LX5280 Memory Specifics

The LX5280 has a 64-bit instruction bus to accommodate the dual-issue pipeline. It also has a 64-bit interface to the data caches and DMEM to allow 64 bit accesses in a single cycle. Thus all the memory interfaces/wrappers must be 64 bits. However, the memories themselves may be 32 bits (as in the LX5180) or 64 bits. In any case, there must be a separate WE signal (write enable) for the upper 32 bit memory (bit 63-32) and another WE for the lower 32 bit memory (bit 31-0).

Example:

A memory configured to be "8K_1" in the `lconfig` form (8 Kbytes direct mapped) will have the following depth and width:

1024x64	for LX5280
2048x32	for LX4189

4.11 LX8000 Memory Specifics

The LX8000 has a 64-bit system bus interface. To take advantage of this the ICACHE, IMEM and DCACHE are 64 bits wide. All the memory interfaces/ wrappers for these RAMs must be 64 bits. However, the memories themselves may be 32 bits (as in the LX4189) or 64 bits. In any case, there must be a separate WE signal (write enable) for the upper 32 bit memory (bit 63-32) and another WE for the lower 32 bit memory (bit 31-0).

The DMEM on the LX8000 has it's width specified via the lconfig form option (DMEM_WIDTH). A write enable bit is added for each 32-bit word per DMEM address. Therefore, a 32-bit wide DMEM will require one write enable bit. A 64-bit DMEM will require two write-enable bits and the 128-bit wide DMEM requires four write enable bits.

NOTE: If using LMI_DATA_GRANULARITY=BYTE there will be write-enables for every byte. Thus, if the DMEM interface is 128 bits wide there will be 16 write-enables.

Chapter
5

Using the LBC Interface

The Lexra Bus Controller (LBC) is the interface between the internal instruction and data busses of the core and the external bus, which is used to connect peripherals such as main memory or I/O devices. The LBC implements Lexra's LBUS protocol, a full-featured bus protocol that supports multiple bus masters through a request/grant arbitration.

A detailed description of the LBC and LBUS protocol, as well as the pin description is given in the product datasheet.

The product datasheet also describes an alternative bus structure available to the Lexra cores, the CBUS. The CBUS is an appropriate alternative when a simple point-to-point bus protocol is desired.

5.1 Configuring the LBC with `Lconfig`

5.1.1 Configuring a Synchronous/Asynchronous Interface

The LBC's bus clock can operate either synchronous or asynchronous to the Lexra core clock, as the LBC contains built-in asynchronous handshaking logic. This handshaking logic does, however, significantly increase the latency of bus accesses. Therefore, the LBC should be operated in synchronous mode if bus latency is a concern.

The LBC is configured to run either synchronous or asynchronous using the LBC_SYNC_MODE `lconfig` option. Please refer to the `lconfig` form and the product datasheet for the available settings.

The latency for LBUS accesses depends upon several variables. For a synchronous bus clock, the latency will depend upon the following factors:

- Wait states of target device;

- LBUS arbitration delay;

- Type of bus operation (e.g., single word read, cache line read);

-Cache line size;

- For data cache miss operations, the location of the critical word within the retrieved cache line (the data cache controller releases the pipeline stall when the desired word has been retrieved over the Lexra bus);

- LBUS width (32 bits or 64 bits. Refer to product datasheet for your specific product configuration).

When an asynchronous bus clock is used, the latency will increase due to the asynchronous handshaking logic, which adds approximately 4 cycles per word of round trip latency. However, the actual latency will depend greatly upon the ratio of the bus clock to the core clock.

The table below gives the latency assuming a synchronous bus clock for a 4-word cache line read. The best case assumes the critical word is the first word retrieved in the cache line. The worst case assumes the critical word is the last word retrieved, which will always be true for instruction accesses, as the instruction cache does not implement critical word return capability. Both columns assume no wait states on the target device and that the LBC has been pre-granted the Lexra bus through the use of a parked master arbitration scheme.

Mode	Best-case Latency (cycles)	Worst-case latency (cycles)
synchronous	7	12

5.1.2 Configuring Cache Policies

When the LBC initiates a bus transaction for a cache line read, it puts one and only one address on the address bus for the whole cache line transaction. System memory controllers may respond to cache line requests in various ways.

Therefore, it is important that the LBC be properly configured to accept the words in the cache line in the order in which they will be read by the system memory controller. The memory controller may respond in one of the following ways;

- by transmitting the desired word (the word that actually caused the miss) first
- by transmitting the zero word (the first word in the cache line independent of which word actually caused the miss) first

When transmitting the second and subsequent words, the memory controller may increment the word position within the cache line in a linear or interleaved manner. Please refer to the product datasheet and your lconfig form for details and the appropriate lconfig settings.

5.1.3 Configuring Read and Write Buffer Sizes

The LBC implements a write buffer to accept write requests from the processor. Details of operation and configuration options are explained in the datasheet and lconfig form. Customers who wish to save area may use a reduced write buffer size at the expense of performance. Selecting a larger write buffer size may improve performance slightly for those applications requiring a lot of writes to memory devices at the expense of area.

The LBC also implements a read buffer. The read buffer stores data or instructions temporarily before transmitting them to the internal instruction or data busses. The read buffer allows the LBC to accept read data as it is provided by the target device. If the LBC is not able to accept read data as fast as it is provided by the target device, its read buffer may become full. The LBC will de-assert the IRDY signal during this condition, requiring the target device to stall the LBUS during the transaction. The optimal read buffer size is typically the minimum amount required to avoid the deassertion of IRDY.

If an asynchronous LBC is used, the optimal read buffer size is twice the cache line size, or eight entries assuming a 4-word cache line size. In the case of simultaneous instruction and data cache misses, the LBC will perform successive ICACHE and DCACHE refills. A read buffer of eight entries will allow both requests to be serviced and posted to the read buffer without the deassertion of IRDY during the transaction. Setting the read buffer size to a smaller value will reduce area, but may result in the de-assertion of IRDY during a transaction, thereby reducing overall LBUS bandwidth.

If you configure the LBC with a synchronous interface, the cache can accept the data as fast as the LBC can read it. Therefore, there is no need for a large read buffer. In this case, you may reduce the size of the read buffer size to two entries, the minimum allowable.

For more information on the read buffer and the associated configuration options, please refer to the datasheet and the lconfig form.

5.2 Lbus Device Design Rules

The LBUS can accommodate both master and target devices. Such devices must be designed to accommodate the LBUS protocol as described in the product datasheet. Please refer to the datasheet for detailed signalling descriptions.

5.2.1 LBUS Arbiters

If the LBUS contains additional bus masters besides the Lexra core, or connects multiple master cores, you must design a bus arbiter to arbitrate requests from the various bus masters. Detailed arbitration rules are provided in the product datasheet. Below are some guidelines on bus arbiter design:

- The choice of prioritization (fixed priority, round robin, etc.) is up to the customer. It is the designer's responsibility to ensure that the prioritization scheme used will allot sufficient bus bandwidth to each bus master.

- Only one GNT signal can be asserted at any one time.

- You can design an arbiter that pre-grants a bus master while a previous transaction is in progress. A pre-granted master is one that has received a GNT signal even though it has not yet requested the bus by asserting REQ. This technique will save 2 cycles when the next master requests the bus.

- An arbiter that pre-grants a master may take away the GNT and give it to another master at any time without regard for the state of the transaction in progress (within the limits of the master device's GNT setup time, of course). If the first master has already sampled GNT, it starts the transaction on the next cycle. The new master samples GNT, but also sees that the bus is now being used and does not try to drive the bus. There is no chance of two masters simultaneously driving the bus if they both sample GNT&~FRAME&~IRDY on the

cycle before taking ownership of the bus.

- An arbiter may park a master by continuously asserting GNT, thus giving it instant access to the Lbus if it needs it. If another device needs the bus, the arbiter may take away GNT to the parked master and give it to the new master. The new master still must sample GNT&~FRAME&~IRDY to make sure the parked master has not started a transaction.

5.3 Device Interconnections

On any bus based system, you need to connect devices such that every slave device sees all the master devices' signals, and all the master devices see all the slave devices' signals. In a traditional board level design, you connect them this way with a combination of tri-state busses and pull-up busses. You typically use the pull-up busses for handshake signals that need to be continuously sampled. You use the tri-state busses for wide busses that can be enabled by the device about to drive them.

In an application specific integrated circuit (ASIC), pull-up busses are not possible. Tri-state busses are difficult because ASICs cannot tolerate an un-driven tri-state bus and some ASIC vendors do not permit tri-states. Therefore, you have to modify the traditional model somewhat to build a system on a chip.

There are two basic models you can use for device interconnect: tri-state and multiplexed. The tri-state model resembles the traditional board level tri-state model, except that the tri-state busses must be driven at all times.

The multiplexed model uses multiplexers to create point-to-point connections, either by distributing the multiplexers such that there is one set for each input to each master and each target; or by cascading the signals through two-to-one muxes at each input. Alternatively, you can supply a central set of muxes to route the bus.

For a more detailed discussion of the tri-state model, see Section 5.3.2, Connecting the Address, Data, and Command Busses below.

There are two rules for tri-state busses in most ASIC technologies:

- You must not sample, that is, register a signal in the high impedance state.

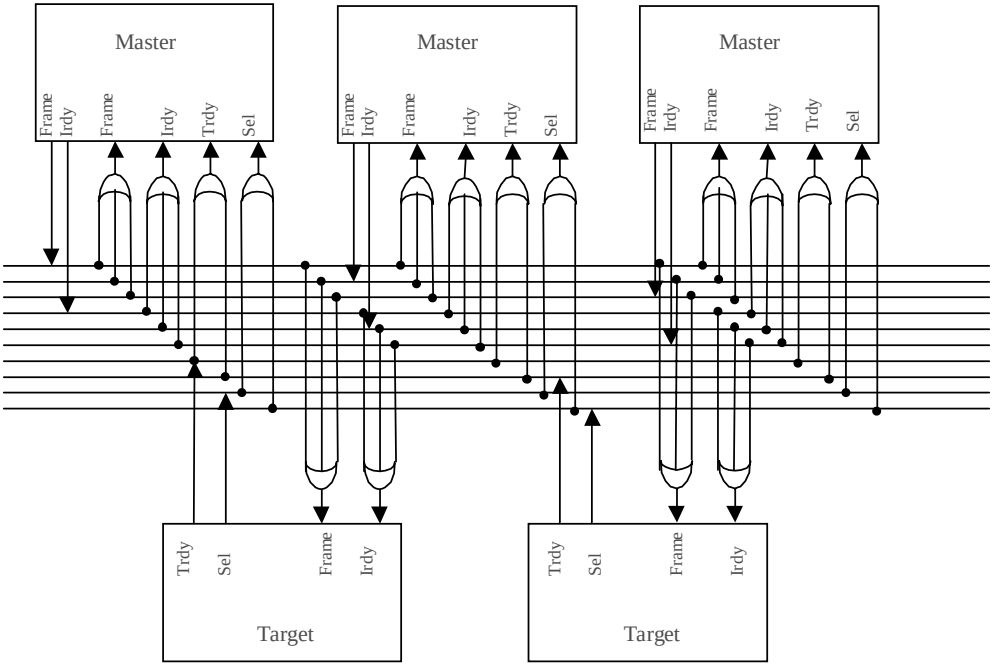
- You can not have two or more drivers to the same signal at the same time. Doing so causes shorts and reliability problems.

We treat protocol signals (FRAME, IRDY, TRDY, and SEL) and busses (ADDR, CMD, and DATA) separately.

5.3.1 Connecting the Protocol Signals Using OR Gates

The easiest way for you to route the protocol (FRAME, IRDY, SEL, TRDY) signals is to route all possible outputs to all possible inputs and logically OR them before presenting them to the input pin. In other words, route all masters' FRAME and IRDY signals to all masters and all slaves and then OR them in front of each FRAME and IRDY input pin. Likewise, route all slaves' TRDY and SEL to all masters, and logically OR them in front of each FRAME and IRDY input pin.

This works because Lexra defines all four protocol signals to be valid at all times and your protocol must guarantee that no two devices ever drive the same signal at the same time. Furthermore, your protocol must guarantee that this is also true of XOE, DOE, and COE. The routing overheard is usually low because there are only four signals.



5.3.2 Connecting the Address, Data, and Command Busses

You need to treat the ADDR, DATA, and CMD busses differently from the protocol signals for three reasons. One is that they are wider busses with more signals and are therefore, more difficult to route. Unlike the protocol signals, which are always defined to be valid, these signals have an undefined value when not valid. Therefore, you can not logically OR them with busses from other sources.

You can connect these busses using either multiplexers or tri-state buffers. Most ASIC vendors require that you use multiplexers to connect the busses, as tri-states do require special handling and can cause testability problems.

The LBC sends two sets of output enable signals to control tri-state buffers.

- COE (command output enable) controls the command and address busses.

- DOE (data output enable) controls the data bus.

Note that there is always an inactive cycle when any of these three busses change ownership. If you are using tri-state busses, this inactive cycle will require special handling to avoid un-driven tri-state busses. It is considered poor design practice to have undriven tri-state busses. Most ASIC/COT libraries that support tri-state drivers include bus holder devices. Bus holders are two cascaded buffers or NOT gates, of which the second is a weak driver. Both the input and the output of the bus holder connect to a bus signal. Because of the weak driver, the bus holder can drive the bus to the most recently driven level without risk of current problems or shorts.

5.4 Using CBUS

The CBUS is a simplified bus protocol optimized for point-to-point operation. It lacks several features of the LBUS as shown in the table below. However, it can be useful in applications in which area is critical and a multiple-master bus protocol is not needed. Also, the simpler protocol of the CBUS makes it easier to interface to a third party or proprietary bus structure.

The table below shows some of the principal differences between the CBUS and the LBUS. For complete information on the CBUS signalling protocol, please refer to the product datasheet.

Feature	**LBUS**	**CBUS**
asynchronous interface	Yes (optional)	no
write buffer	yes	no
read buffer	yes	no
multi-master bus	yes	no
EJTAG support	yes	no

Chapter
6

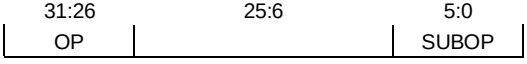
Adding Instructions Using the Custom Engine Interface (CEI)

6.1 Introduction

The Custom Engine Interface (CEI) allows you to implement custom instructions as extensions to the MIPS-1 ISA. The Lexra core contains two CEIs: CEI0 and CEI1. CEI0 is reserved for the optional multiply-accumulate (MAC) module. CEI1 is available for customer defined instructions. Up to 10 custom opcodes may be defined.

6.2 Operation

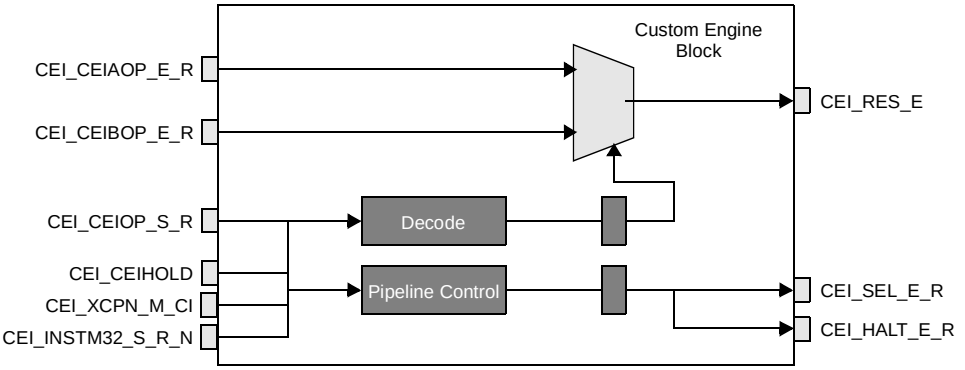
The custom engine receives 12 opcode bits: the 6-bit op field and the 6-bit subop field. The diagram below shows the location of these two fields in the MIPS-1 R-type opcode format.



The custom engine (CE) receives the opcode in the S stage. (Refer to the product datasheet for information on the pipeline stages.) If it decodes the opcode as a valid instruction, it asserts the SEL signal in the E stage. The SEL signal indicates to the processor that the custom engine has received a valid opcode, and a reserved instruction trap should not be taken. If SEL is not asserted, the processor will assume an invalid opcode has been executed and will generate a reserved instruction trap.

During the E stage, the custom engine receives the two operands. They are either two 32-bit register operands or one register operand and one 16-bit immediate operand sign extended to 32 bits. The custom engine can return the results to the processor via the RES signal to the core in the E stage. Alternatively, the custom engine can use temporary registers or the MIPS-1 HI / LO registers.

The figure below shows the block diagram of a custom engine.



6.2.1 Instancing Custom Engines

There are two `lconfig` options controlling the instancing of custom engines: CE0 and CE1. Please refer to the product datasheet and `lconfig` form for details on the `lconfig` options.

Custom engine CE0 is reserved for the Lexra MAC module. The Lexra MAC module is optional on the LX4189 and LX4280, but is required on the LX5180 and LX5280.

Note that the Lexra MAC implements the MIPS-1 HI / LO registers. If the MAC is not used, then CE0 must be set to CE_HL to instance the MIPS-1 HI / LO registers, unless these registers are implemented in CE1 by the user. Please refer to Section 6.3.4, Temporary Registers and MIPS-1 HI/LO for details.

If you want to connect a custom engine to CE1, set CE1 to "EXPORT". Setting CE1 to "CE_DVT" instances a simulation testbed `ce_dvt.v`. You can use the testbed module as a template for implementing a custom engine. The testbed itself is valid only for simulation purposes. The template is not synthesizable.

6.2.2 Interface Signals

The core exports the following interface signals to the `lx2.v` module when you set CE1 to "EXPORT". CEI inputs are outputs of your custom engine, and CEI outputs are inputs to your custom engine.

CEI inputs (outputs of your custom engine):

CE1_RES_E[31:0]. The 32-bit result of the custom engine operation. Returned to the core in the E stage.

CE1_SEL_E_R. Indicates that the custom engine has decoded a valid user defined opcode and that the core should not take the reserved instruction trap. The custom engine returns the signal to the core in the E stage.

CE1_HALT_E_R. Indicates that the custom engine is executing a multiple cycle instruction and the result of the instruction is not yet valid. Asserting this signal causes the pipeline to stall. Refer to Section 6.3.1, Pipeline Issues and Stalls for further explanation of stall conditions. The custom engine returns this signal to the core in the E stage.

CEI outputs (inputs to your custom engine):

CEI_CE1OP_S_R[11:0]. The 12-bit concatenation of the OP and SUBOP fields from the fetched instruction. Valid in the S stage.

CEI_INSTM32_S_R_N. The core asserts low if the instruction is a 32-bit MIPS-1 instruction. The core asserts high if the instruction is a 16-bit MIPS-16 instruction. The custom engine must examine this signal to avoid aliasing custom instructions with MIPS-16 instructions. Custom opcodes are not available in MIPS-16. Valid in the S stage.

CEI_CE1AOP_E_R[31:0]. The 32-bit A register operand. Valid in the E stage.

CEI_CE1BOP_E_R[31:0]. The 32-bit B register operand or the 16-bit immediate field (sign extended to 32 bits) from the instruction. Valid in the E stage.

CEI_CE1HOLD. When high, indicates a pipeline stall. During a pipeline stall, the outputs from the interface to the custom engine are not valid and should be ignored. The custom engine must stall its pipeline and retain the previous state of its outputs. In some cases, the custom engine may need to hold some of its internal state when this signal has been asserted. The core will not assert this signal if the custom engine itself causes a pipeline stall by asserting the *CE1_HALT_E_R* signal. Valid in any pipeline stage.

CE1_XCPN_M_C1. Indicates an exception has occurred. Section 6.3.2, Exceptions and Invalidation explains the significance of exceptions to custom engine instructions. Valid during the M stage.

CEI_CE1INVLD_M. Indicates that the current M stage instruction is invalid. Execution of any CE instruction in the M stage should be terminated. Refer to Section 6.3.2, Exceptions and Invalidation for details. Valid during the M stage.

CEI_CE1INVLDP_S_R. Indicates an invalid instruction in the S stage of the pipeline. Therefore, the instruction opcode present on *CEI_CE1OP_S_R* is not valid and should be ignored. Refer to Section 6.3.2, Exceptions and Invalidation for details. Valid during the S stage.

6.2.3 Available Opcodes

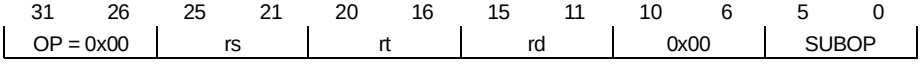
The table below shows instructions available for use by the custom engine interface in bold type. The custom engine must also decode the MIPS-1 MFHI / MFLO / MTHI / MTLO and the MIPS-16 MFHI / MFLO if the custom engine implements HI / LO registers.

Opcodes in addition to those listed below are not generally supported. Please contact Lexra if your custom engine design must implement additional instructions beyond the 10 custom opcodes allocated below.

OP[11:6] = INSTR[31:26]	OP[5:0] = INSTR[5:0]	Description	Instruction format
2'h00	2'h38	NEW_ROP0	register
2'h00	2'h3a	NEW_ROP2	register
2'h00	2'h3b	NEW_ROP3	register
2'h00	2'h3c	NEW_ROP4	register
2'h00	2'h3e	NEW_ROP6	register
2'h00	2'h3f	NEW_ROP7	register

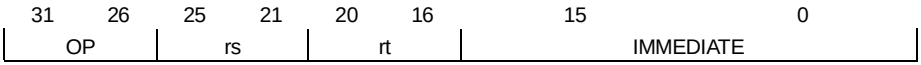
OP[11:6] = INSTR[31:26]	OP[5:0] = INSTR[5:0]	Description	Instruction format
2'h18	immediate[5:0]	NEW_IOP0	immediate
2'h19	immediate[5:0]	NEW_IOP1	immediate
2'h1a	immediate[5:0]	NEW_IOP2	immediate
2'h1b	immediate[5:0]	NEW_IOP3	immediate
2'h00	2'h10	reserved for MFHI	register
2'h00	2'h11	reserved for MTHI	register
2'h00	2'h12	reserved for MFLO	register
2'h00	2'h13	reserved for MTLO	register
2'h2d	6'bx1_0000	MIPS-16 MFHI	register
2'h2d	6'bx1_0010	MIPS-16 MFLO	register

Register format:



rd <= rs SUBOP rt

Immediate format:



rt <= rs OP Immediate

For register format instructions, the rs operand appears on *CE1AOP* and the rt operand appears on *CE1BOP*. The result is stored in register rd. To make use of internal custom engine registers (such as the MIPS-1 HI / LO registers) instead of a 32-bit destination register, the rd field must be set to 0x00.

For immediate format instructions, the rs operand appears on *CE1AOP* and the 16-bit immediate field (sign extended to 32 bits) appears on *CE1BOP*. The result is stored in rt. To make use of internal custom engine registers (such as the MIPS-1 HI / LO registers) instead of a 32-bit destination register, the rt field must be set to 0x00.

6.3 Implementation Details

6.3.1 Pipeline Issues and Stalls

There are three types of custom engine operations:

- single cycle
- multiple cycle without stalls
- multiple cycle with stalls

In single cycle operations, the custom engine decodes the instruction in the S stage and receives the operands in the E stage. It transfers the results to the core by the end of the E stage, and the core updates general registers in the W stage.

In multiple cycle custom engine operations without stalls, the custom engine decodes the instruction in the S stage, and receives the operands in the E stage. However, it does not return the result to the core in the E stage. Instead, the custom engine stores the result in temporary registers when the operation is completed. One example of internal temporary registers are the MIPS-1 HI / LO registers which are described in more detail in Section 6.3.4, Temporary Registers and MIPS-1 HI/LO.

For custom engine operations that use internal registers, the custom engine holds its own pipeline in the M stage for as many cycles as it takes the instruction to execute. The processor pipeline will continue to execute subsequent instructions. When the operation is complete, the program can retrieve the data using custom engine opcodes that read these registers. One example of these opcodes are the MIPS-1 MFHI and MFLO instructions, explained in more detail in Section 6.3.4, Temporary Registers and MIPS-1 HI/LO.

In some cases, the custom engine may receive an instruction to read the temporary registers before the execution of the multiple cycle custom engine instruction is complete. There are two approaches to solving this problem.

- Require the programmer to place the proper spacing between the custom instruction and an instruction accessing the temporary registers. Attempts to read the result earlier return invalid data. This approach simplifies the hardware design at the expense of programming complexity.

- Assert the *CE1_HALT_E_R* signal to the core until the results of the custom instruction are available in the temporary registers. This approach adds complexity to the hardware design, but avoids any programming hazards with custom engine instructions.

If you eliminate multiple cycle operations with stalls, you eliminate the requirement for temporary registers. If an instruction does not require a large number of cycles to execute, you could have the custom engine assert the *CE1_HALT_E_R* signal when the instruction enters the E stage and then deassert it when the execution is complete. The custom engine would assert the result on *CE1_RES_E* in the same cycle that it deasserts *CE1_HALT_E_R*.

This approach causes the processor pipeline to stall whenever it encounters the multiple cycle instruction, which in turn may cause performance degradation. Therefore, we recommend this approach only when the number of required instruction cycles is two or fewer. However, the reduction in design complexity that results from removing the temporary registers may make this trade-off worthwhile.

The processor can stall the custom engine by asserting the *CEI_CE1HOLD* signal. This signal indicates that an operation such as a cache miss has caused a processor stall. The processor does not assert this signal if the custom engine itself caused the stall (by asserting *CE_HALT*). This eliminates the need to check for potential deadlock conditions. If the processor asserts *CE1HOLD*, the custom engine must hold any instruction in the S or E stage. You must ignore the opcode and operand inputs and hold the *RES* output in its previous state, as the processor is not ready to access the result of the custom engine instruction. Multiple cycle instructions that are in the first M stage must be held as well for proper exception handling. Instructions that have already advanced past the first M stage may continue to execute and may update internal custom engine temporary registers when execution completes.

6.3.2 Exceptions and Invalidation

Since the custom engine is tightly coupled with the core pipeline, proper exception handling is a requirement. There are many sources of exceptions, including hardware interrupts and EJTAG breakpoints. Therefore, custom engine instructions are subject to exceptions.

Exceptions are recognized when the exception victim reaches the M stage. At that time, all subsequent instructions which are in earlier stages in the pipeline are

squashed. The exception victim and flushed instructions are re-executed when the exception routine has completed. In order to implement precise exceptions, the state of the custom engine must not be changed by the squashed instructions.

The custom engine opcodes and operands are presented to the custom engine in the S and E stages respectively. Therefore, the custom engine design must take care to avoid state changes until the CE instruction has completed the M stage without any exceptions. Exception handling is enabled through the use of the *CE1_XCPN_M_C1* and *CEI_INVLD_M* signals.

The *CEI_INVLD_M* is asserted whenever the instruction in the M stage must be invalidated. If *CEI_INVLD_M* is asserted, the custom engine must invalidate any instruction in the M stage. The invalidated instruction must not change any state internal to the custom engine. The invalidation may or may not be caused by an exception.

The *CE1_XCPN_M_C1* signal is asserted when an instruction in the M stage of the pipeline contains an exception. When *CE1_XCPN_M_C1* is asserted, the custom engine must abort any operations in the S and E stages. The aborted instructions must not proceed to the M stage, and must not change any state internal to the custom engine. Instructions already in the M stage may continue to execute, unless the *CEI_INVLD_M* signal is also asserted.

For multiple cycle instructions, the definition of the M stage may be confusing. The M stage is the first cycle following the E stage in which there are no pipeline holds (as indicated by *CE1_HALT_E_R* or *CEI_CE1HOLD*). [Note: the E stage is when the 2 operands are valid.] Subsequent cycles are not considered part of the M stage. Custom instructions executing in these cycles may continue execution. For example, consider an opcode *CE_LONG* that takes 10 cycles following its E stage and writes to HI / LO upon completion. The instruction *CE_LONG* is only subject to exceptions in the first of the 10 cycles, which is its M stage. If *CE_LONG* is executing, and has progressed beyond this first cycle, it must continue to completion, even if *CEI_INVLD_M* or *CE1_XCPN_M_C1* are asserted.

In some cases, the instruction in the S stage may be invalid. In such situations, the core asserts *CEI_CE1INVLDP_S_R*, indicating that the S stage opcode on *CEI_CE1OP_S_R* is not valid and should be ignored. Custom engine operations already in the E or M stages may continue execution. Examples of invalid opcodes in the S stage include pipeline bubbles inserted by certain instructions.

6.3.3 Dual Issue Considerations

Custom engine operations on the dual-issue processors (LX4280 and LX5280) are handled in the same manner as the single-issue processors (LX4189, LX5180, LX8000). The pipeline control signals and exception signals must be handled in the same manner as described earlier. However, there are some potentially confusing situations that warrant further explanation.

Custom engine opcodes are issued to pipeline B, also known as the MAC pipe. In some cases, an instruction may be issued only to pipeline A due to a scheduling conflict. Refer to the product datasheet for details on dual-issue scheduling conflicts. In such cases, an invalid instruction will propagate through pipeline B, causing assertion of the *CEI_CE1INVLDP_S_R* signal. Proper handling of this signal is still the same as described earlier.

If an exception occurs, there may be two instructions in the M stage of the processor pipeline. Depending upon the instruction ordering and the location of the exception, one or both instructions may be flushed. If only one instruction is flushed, the other instruction must continue to execute. The flushed instruction may be in either pipeline A or B. Despite this complexity, the processor will assert the proper combination of *CEI_INVLD_M* and *CE1_XCPN_M_C1* so that the custom engine can properly handle the exception. The definition of these signals is the same as described in Section 6.3.2, Exceptions and Invalidation.

6.3.4 Temporary Registers and MIPS-1 HI/LO

To avoid stalls on multiple cycle custom engine instructions, temporary registers must be used to store the results as described in Section 6.3.1, Pipeline Issues and Stalls. There are two approaches to implementing temporary holding registers:

- Implement registers accessed by custom engine opcodes
- Implement MIPS-1 HI and LO registers

The former approach requires at least one opcode per 32-bit register to read the result. An additional opcode per register may be required if writing to the register is required, either for initialization or for a context save and restore.

The MIPS-1 instruction set provides two 32-bit registers called HI and LO. These are accessed by the MFHI, MFLO, MTHI and MTLO instructions. In some cases, these registers may be used as temporary registers, thereby saving valuable opcodes. Some restrictions when using HI / LO are:

- The MAC uses the MIPS-1 HI and LO registers to store the results of the MIPS-1 multiply and divide instructions. Therefore, the use of HI and LO is restricted to those configurations that do not implement the MAC. This restriction prohibits the use of HI and LO on the LX5180 and LX5280 products.

- Normally HI and LO are implemented on CE0 when CE0 is set to its default value of CE_HL. If the custom engine implements HI and LO registers, the value of CE0 must be set to NONE. ***Lconfig does not check for any conflict in the use of HI and LO between custom engine modules and CE0 modules.***

- The custom engine must implement the four MIPS-1 opcodes for MTHI, MTLO, MFHI and MFLO. Additionally, if the Lexra core will be used to run MIPS-16 code, the custom engine must also implement the two MIPS-16 opcodes for MFHI and MFLO. ***This is the only situation in which a custom engine instruction will utilize MIPS-16.***

- Compilers can make use of HI and LO to implement software multiplies and divides. Therefore, the programmer should ensure that data from custom engine operations is retrieved from HI / LO as quickly as possible following execution of the CE opcode.

Regardless of the approach used to implement temporary registers, the value of these registers must not be updated until the custom engine instruction has completed its M stage without any exceptions.

6.3.5 Timing Considerations

All outputs of the custom engine (inputs to the Lexra core) should be sourced by registers to avoid any timing convergence problems.

Note that both the operands and the result are valid in the E stage. This restriction means that for a single cycle instruction, time must be allotted for the register-to-output delay for the operands, execution of the instruction, setup time for the result and wire propagation delay. The amount of time for instruction execution in

a single cycle is very limited. The exact amount is technology dependent, but Lexra recommends that the time allocated be no more than 20-25% of the total cycle time. This restriction limits single cycle opcodes to simple operations only, especially in high frequency designs.

Inputs to the custom engine (outputs of the Lexra core) are generally available early enough that they do not need to be immediately registered. Please consult the product datasheet for the approximate delay times on custom engine signals.

6.4 Waveforms

Waveforms for several common custom engine operations follow. Some general notes regarding these waveforms:

- If signals are valid only during a certain pipeline stage (e.g., *Czrd_gen_S*), the pipeline stage is appended to the end of the signal.

- The signals that start with the string "i_", such as i_CE_stage_M and i_CE_stage_W, are not actual interface signals, nor do they correspond to signals internal to the Lexra core. They are used to demonstrate the pipeline stage corresponding to the illustrated custom engine operation. For example, i_CE_stage_M "asserts" when the custom engine operation is in its M stage.

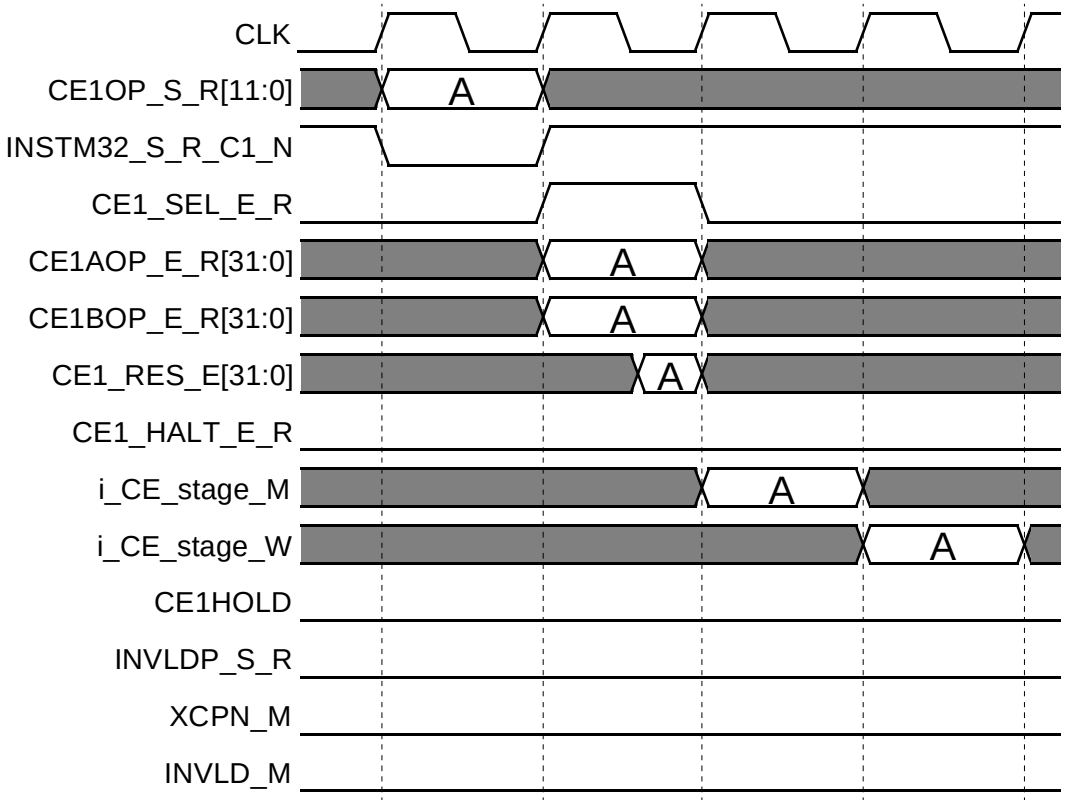


Figure 6-1. Single Cycle Custom Engine Operation

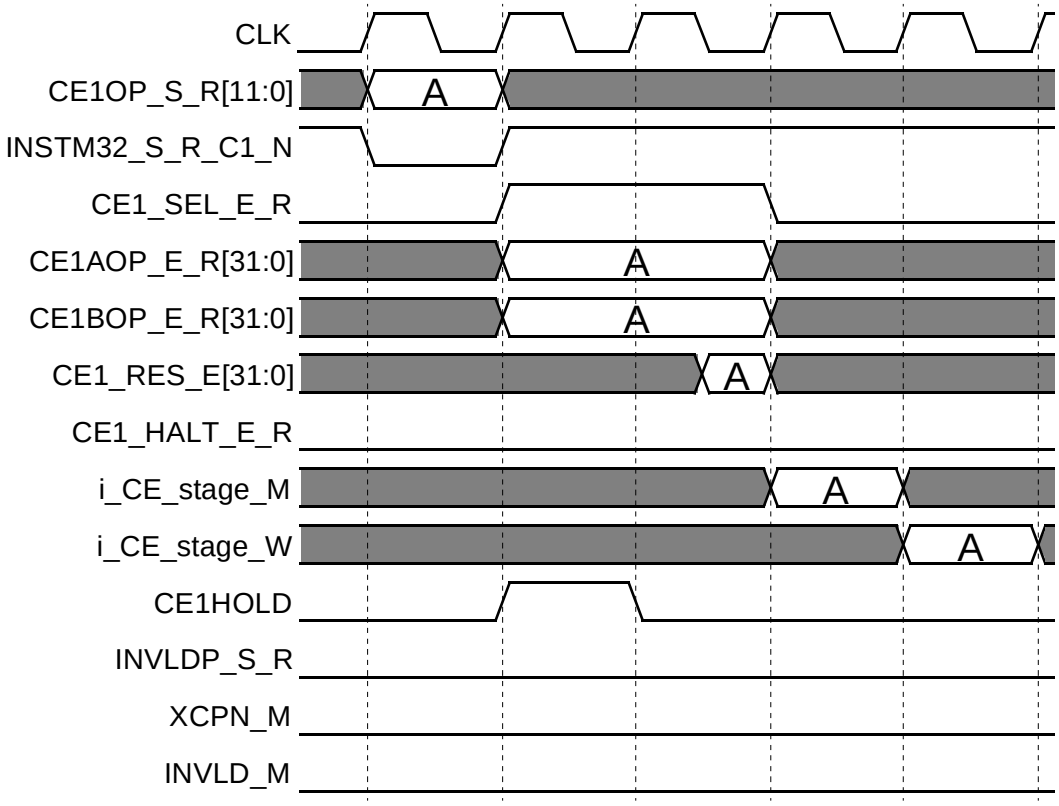


Figure 6-2. Single Cycle Custom Engine Operation with HOLD.

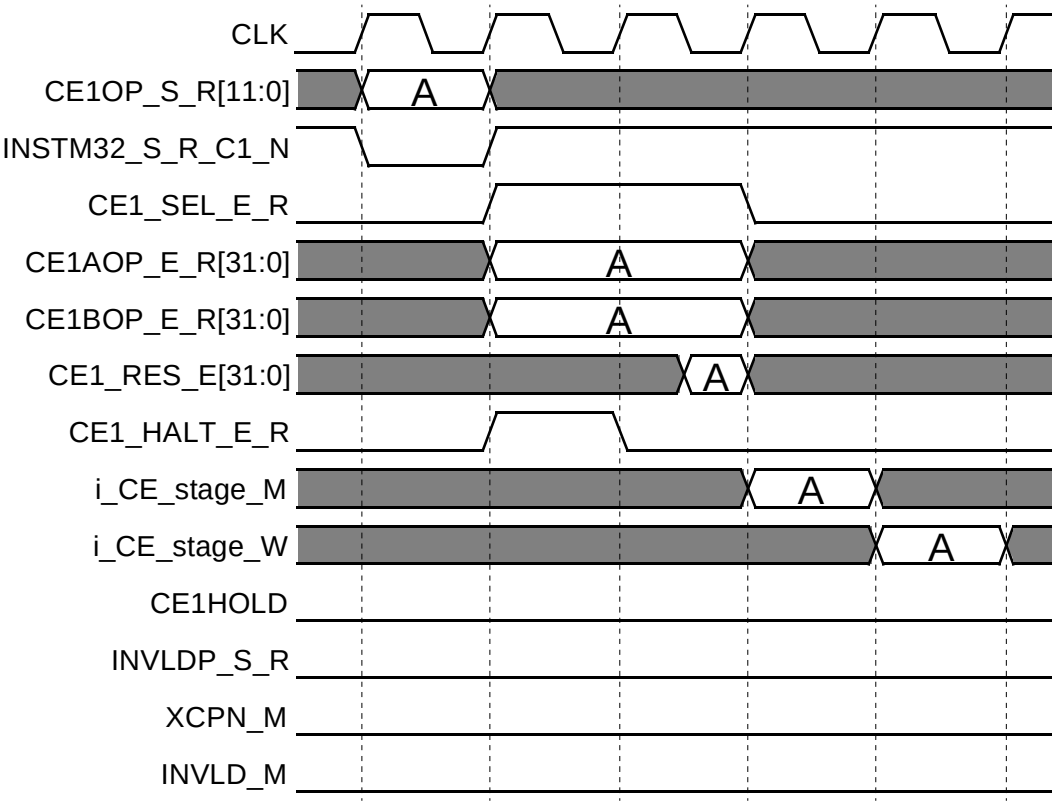


Figure 6-3. Two-cycle Custom Engine Operation with HALT.

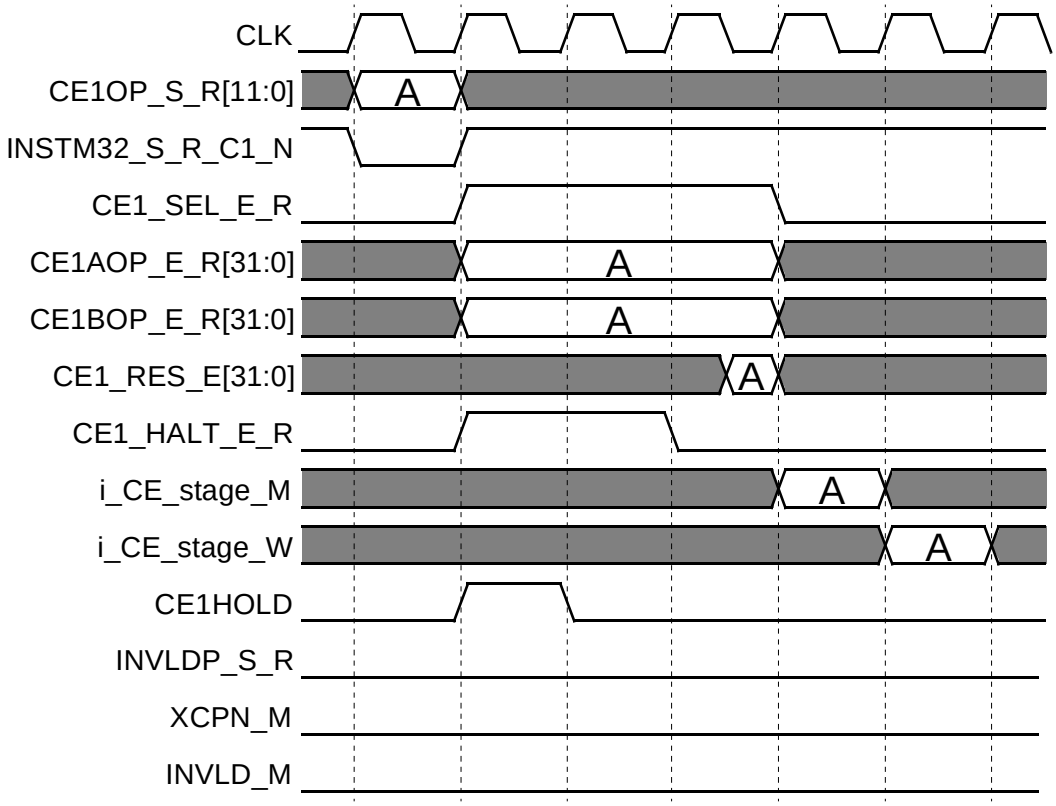


Figure 6-4. Two-cycle Custom Engine Operation with HALT and HOLD.

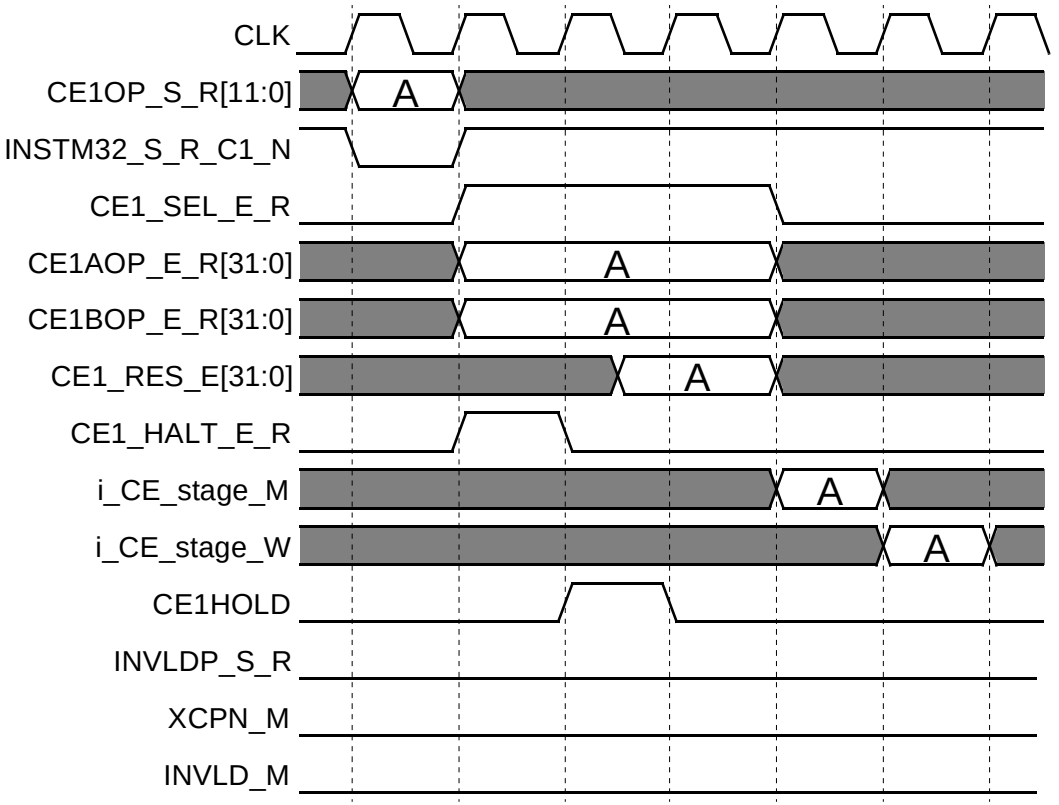


Figure 6-5. Two-cycle Custom Engine Operation with HALT and Delayed HOLD.

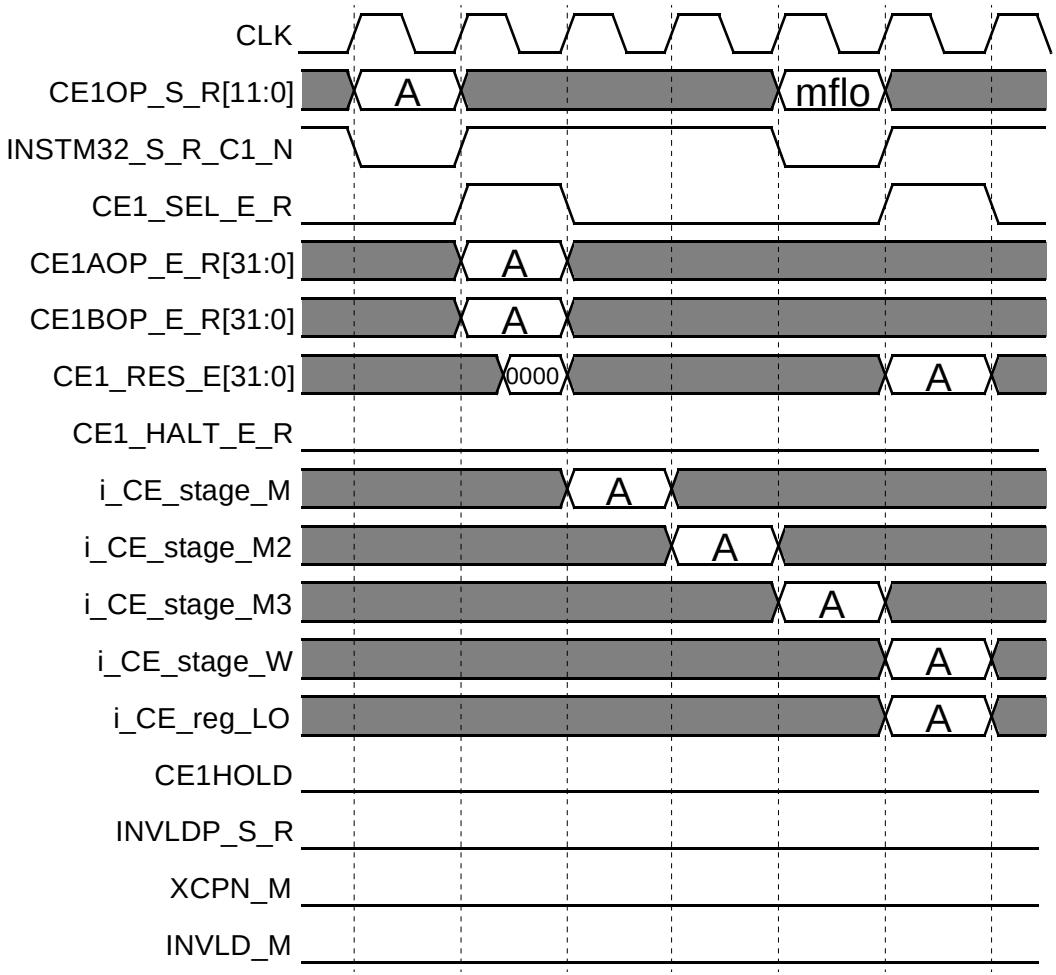


Figure 6-6. Multi-cycle Custom Engine Operation Using HI/LO. The MFLO returns the result from operation A.

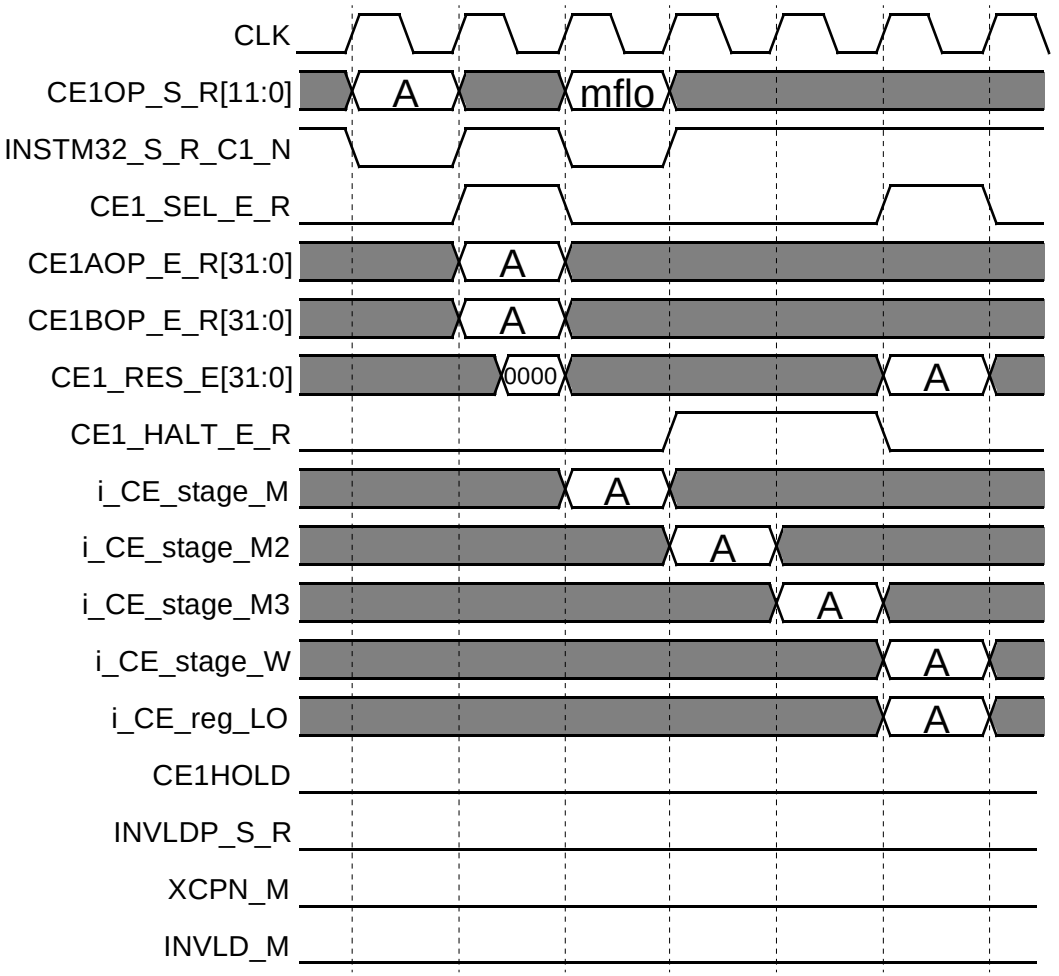


Figure 6-7. Multi-cycle Custom Engine Operation with Early MFLO Inducing HALT Condition.

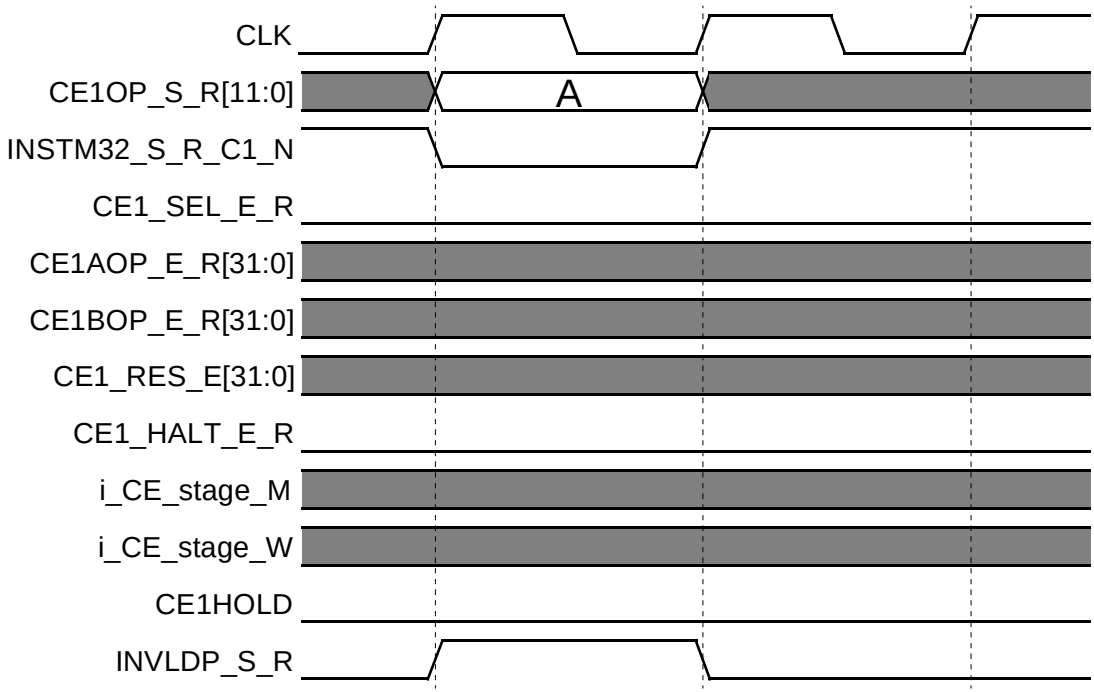


Figure 6-8. Custom Engine Operation with S-stage Invalidate.

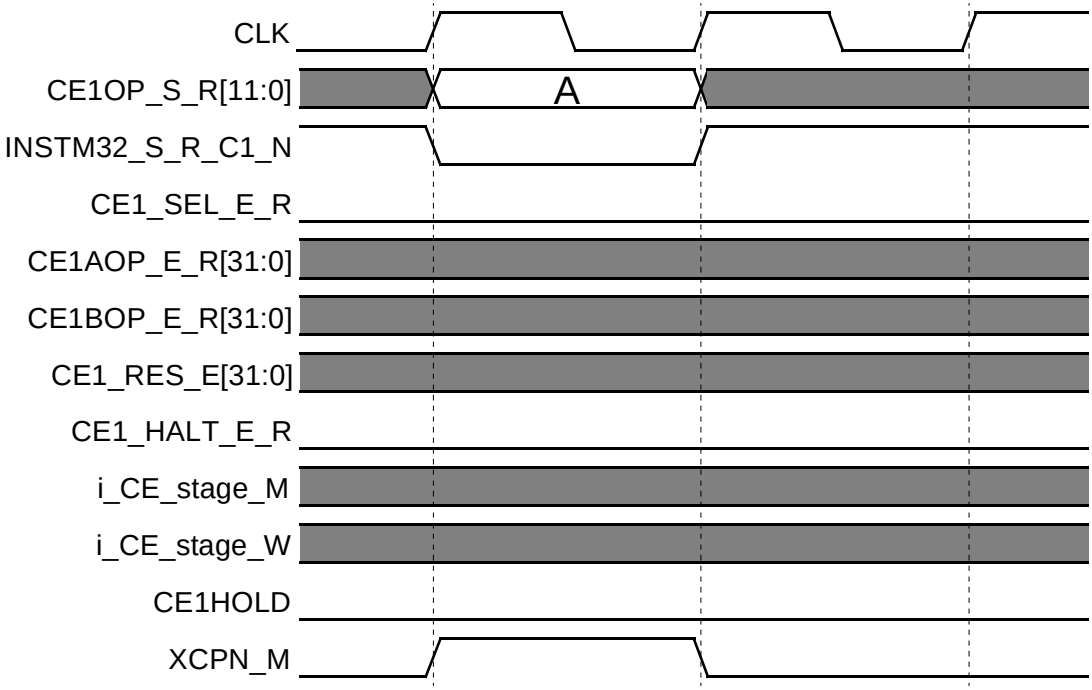


Figure 6-9. Custom Engine Operation with S-stage Exception.

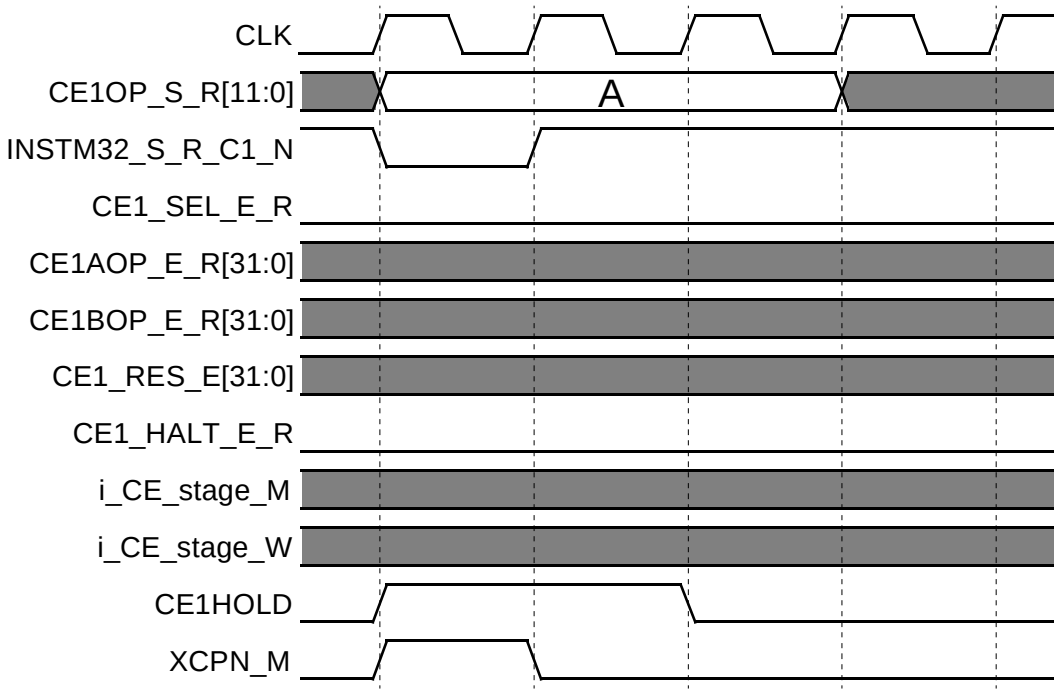


Figure 6-10. Custom Engine Operation with HOLD and S-stage Exception.

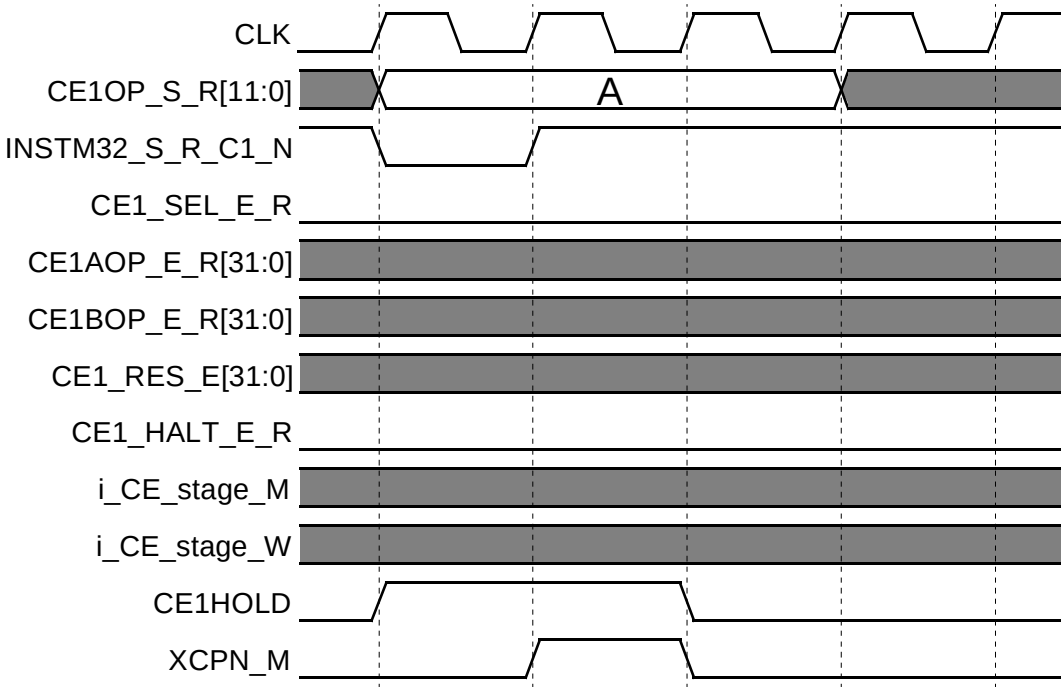


Figure 6-11. Custom Engine Operation with HOLD and Delayed S-stage Exception.

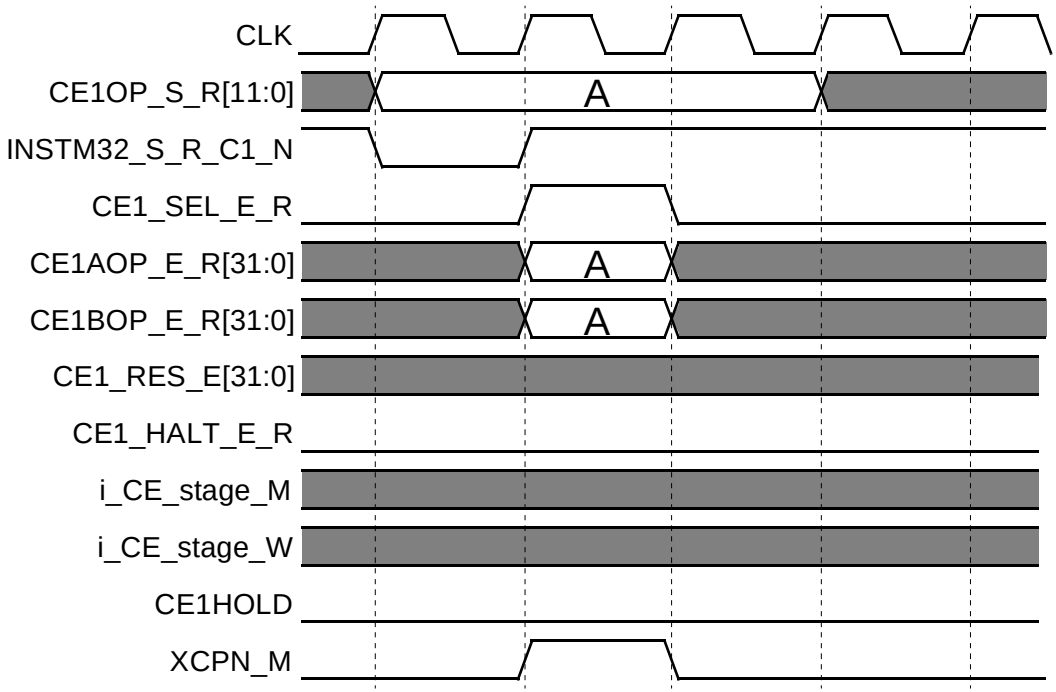


Figure 6-12. Custom Engine Operation with E-stage Exception.

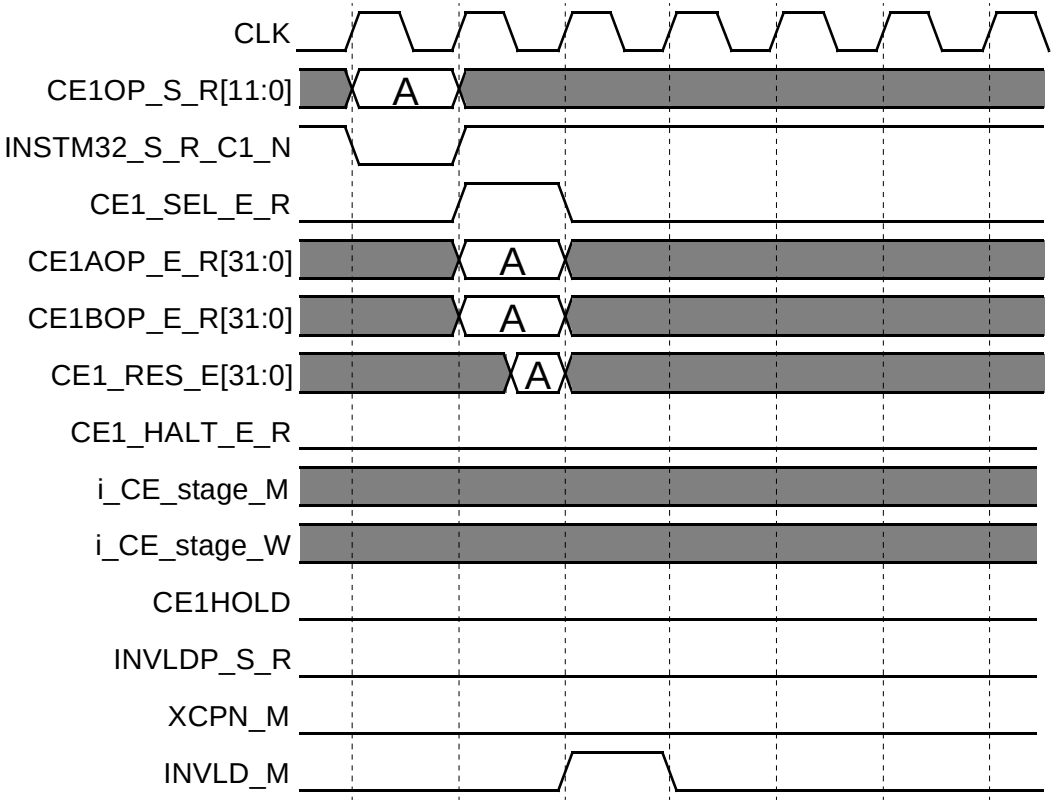


Figure 6-13. Custom Engine Operation with M-stage Invalidate.

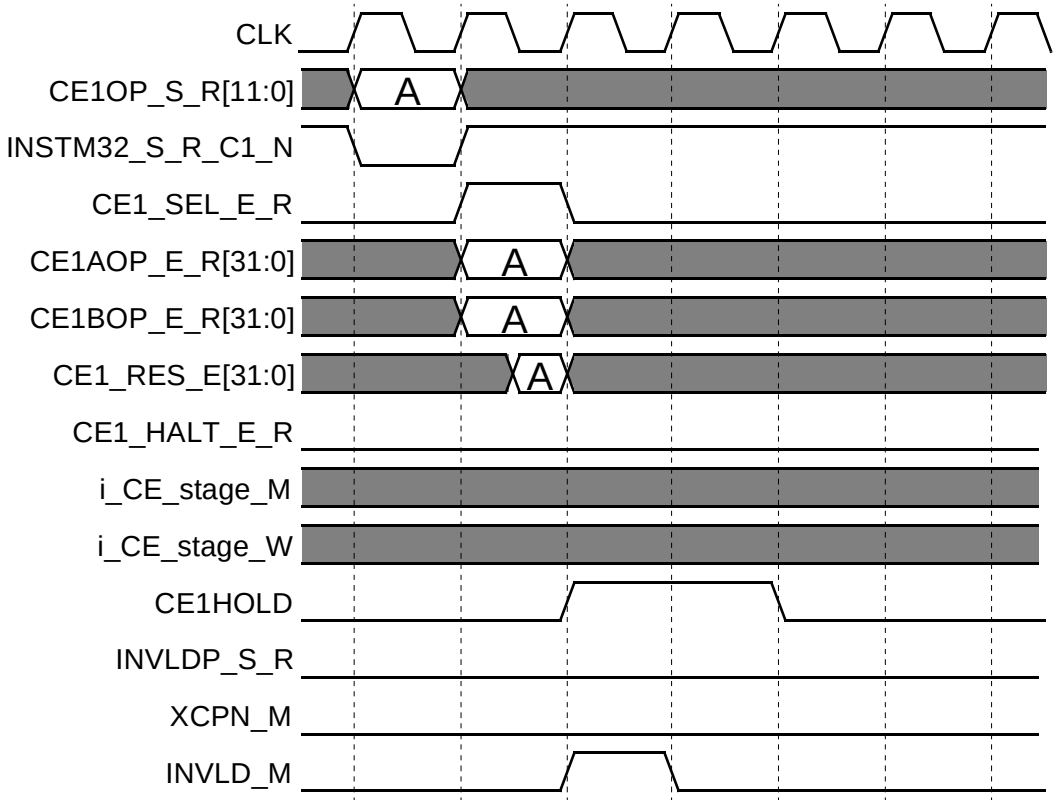


Figure 6-14. Custom Engine Operation with HOLD and M-stage Invalidate.

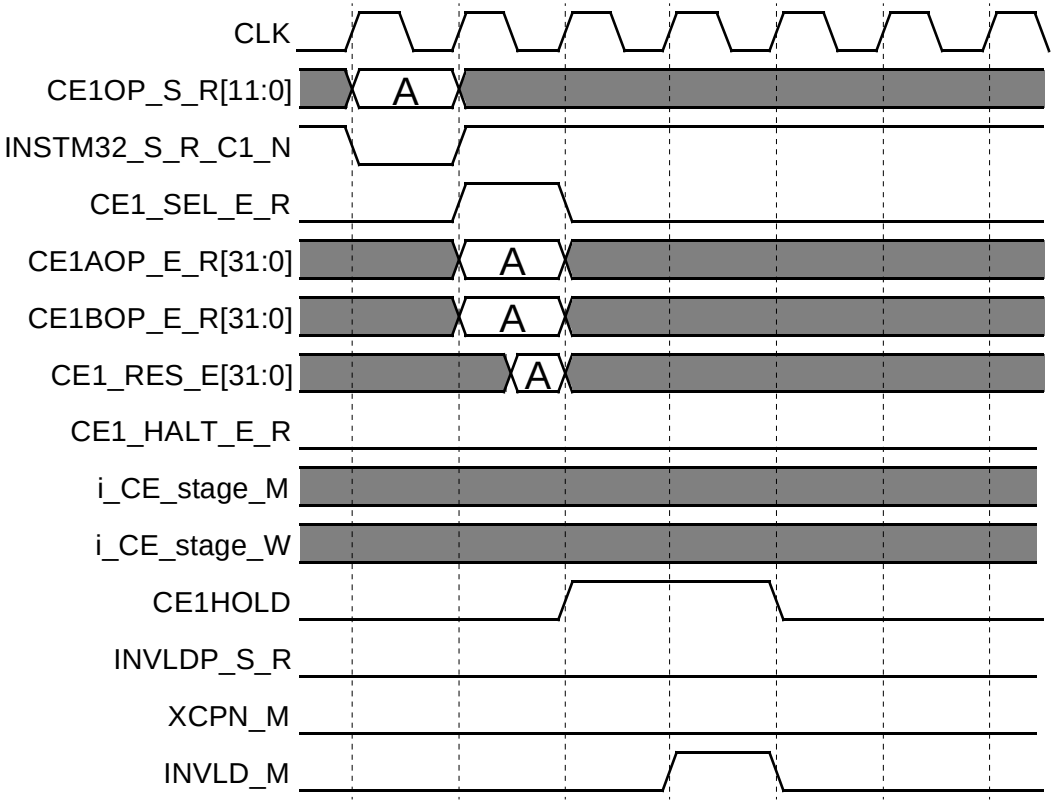


Figure 6-15. Custom Engine Operation with HOLD and Delayed M-stage Invalidate.

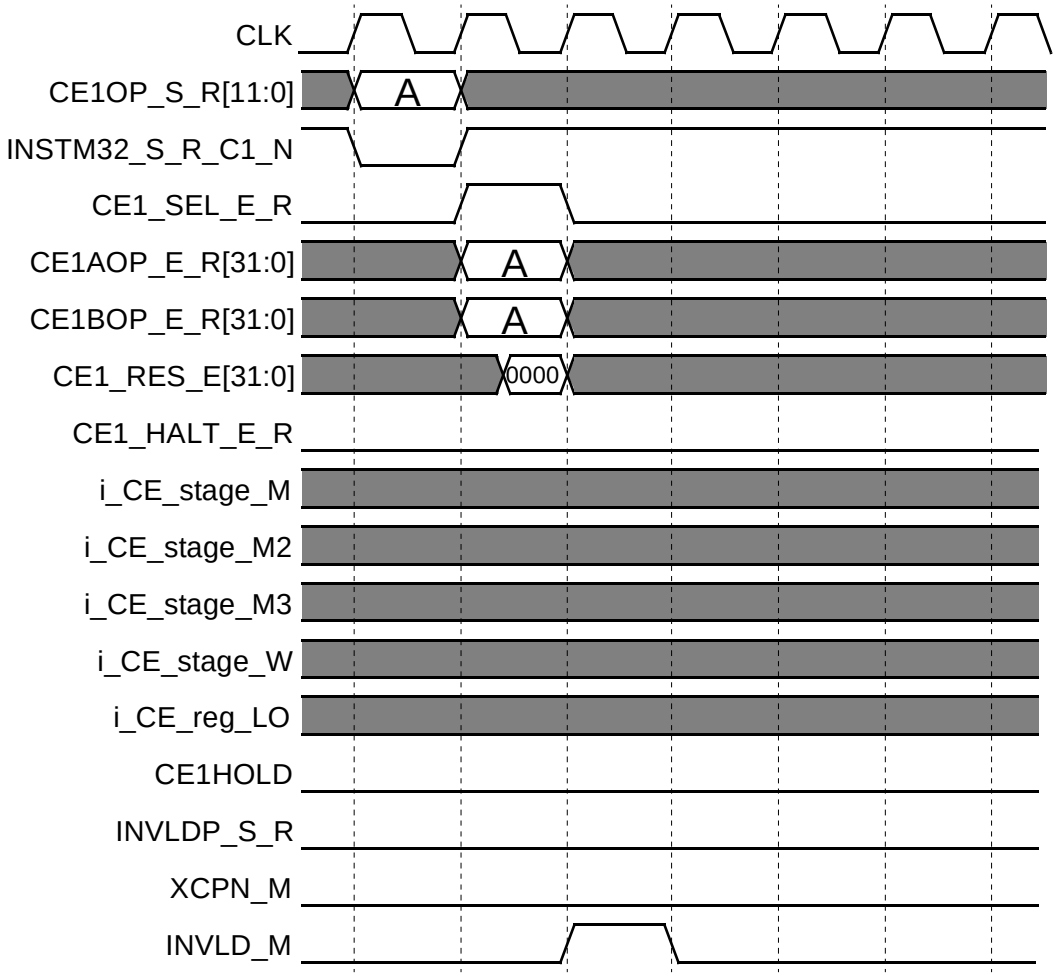


Figure 6-16. Multi-cycle Custom Engine Operation Suppressed by M-stage Invalidate.

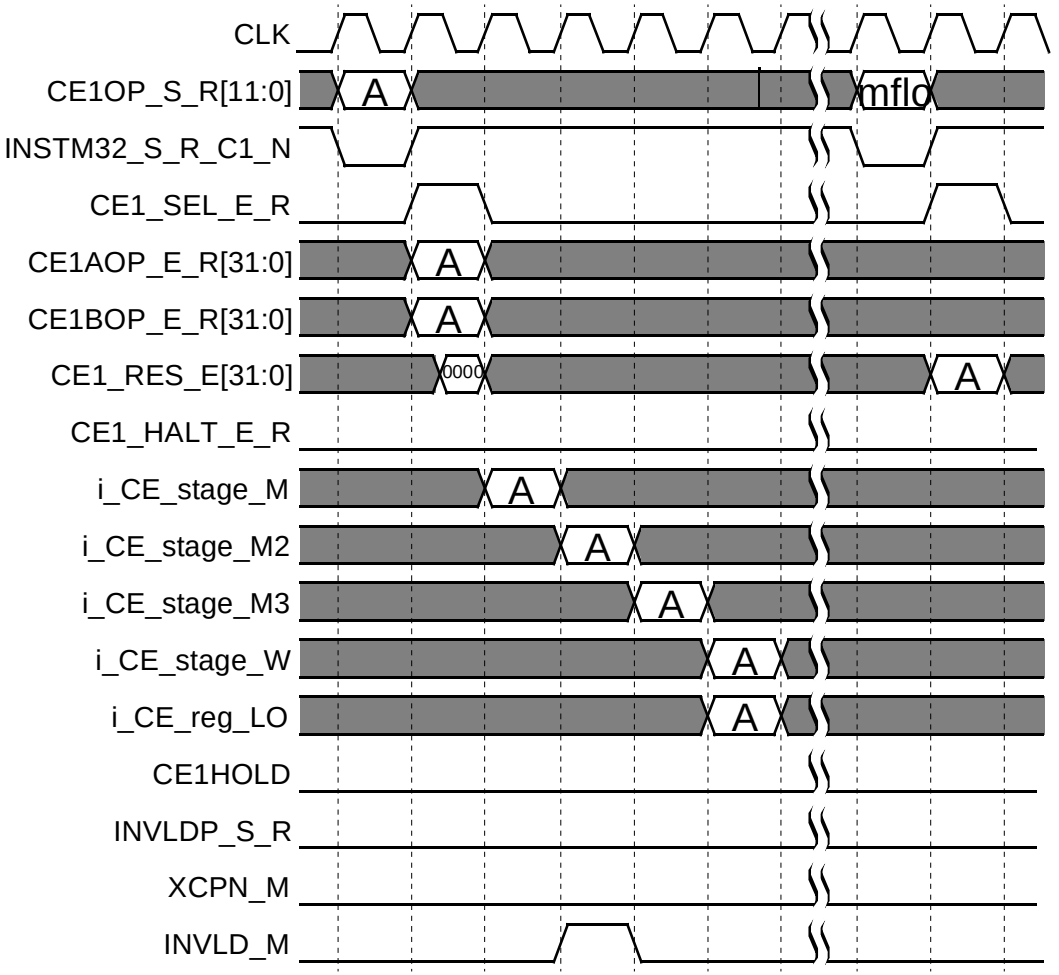


Figure 6-17. Multi-cycle Custom Engine Operation Continues if Invalidate Occurs After M-stage.

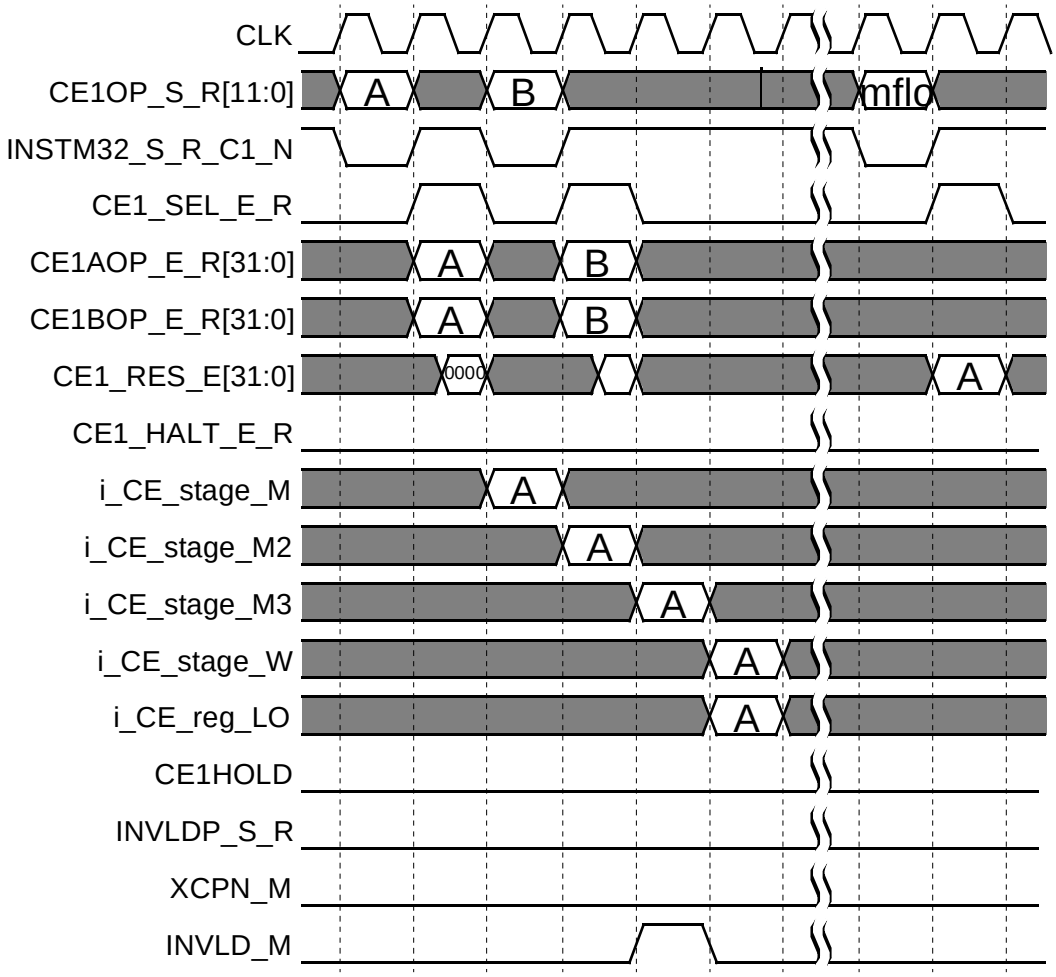


Figure 6-18. Two multiple Cycle Custom Engine Operations, with Second Operation (B) Suppressed by M-stage Invalidate.

Chapter
7

Using the Coprocessor Interface (CI)

Coprocessors can be used to implement long latency operations that are controlled directly by the processor. An example of such an operation may be a DMA transfer. Coprocessors are also useful to implement custom logic that must be directly controlled by the processor. This chapter discusses some of the design considerations when implementing a coprocessor to connect to the Lexra core. Waveforms are included to help illustrate the protocol.

A detailed description of the coprocessor interface signals, timing information, and pipeline considerations can be found in the product datasheet. The appropriate `lconfig` option settings can be found in the product datasheet and in the `lconfig` form.

7.1 Coprocessor Overview

A Lexra coprocessor can contain up to 32 processor addressable general registers and up to 32 processor addressable control registers. Each of these registers is up to 32 bits wide. Typically, you use the general registers for loading and storing data on which the coprocessor operates. Write data to the coprocessor's general registers from the core's general registers with the MTCz instruction. Read data from the coprocessor's general registers to the core's general registers with the MFCz instruction. Load and store the coprocessor's general registers directly from main memory with the LWCz and SWCz instructions.

You can load and store the coprocessor's control registers from the core's general registers with the CTCz and CFCz instructions respectively. You can not load or store the control registers directly from main memory.

The coprocessor can also provide a condition flag to the core. The condition flag can be a bit of a control register or a logical function of several control register values. Test the condition flag with the BCzT and BCzF instructions. These instructions indicate that the program should branch if the condition is true (BCzT) or false (BCzF).

7.2 Coprocessor Design Considerations

Listed below are some of the issues to consider when designing a Lexra coprocessor.

- All coprocessor instructions induce a pipeline bubble, which is effectively a NOP in the pipeline. This allows the results of coprocessor instructions to be available in the following instruction.

- A coprocessor cannot stall the processor pipeline. A coprocessor must return valid read data in the cycle following a read request. A coprocessor must accept write data when it is sent by the coprocessor interface.

- If the processor asserts *Cz_rhold*, the coprocessor must continue to hold valid data on its read data output. During a pipeline hold, the coprocessor should not sample the read or write address, or data as these signals may not necessarily be valid. The coprocessor should sample these inputs in the cycle when *Cz_rhold* deasserts.

- Coprocessor read and write operations occur at different pipeline stages of their corresponding instructions. Therefore, coprocessor reads and writes can happen simultaneously, even to the same register. The instruction sequence is described in the product datasheet.

- In some instances in which a coprocessor read immediately follows a coprocessor write operation, a forwarding path is activated in the coprocessor interface. In this instance, the coprocessor read will return the same exact data that was written in the earlier write operation. In some coprocessor designs, the returned data from the forwarding path may be different than would normally be expected had it been returned from the coprocessor itself. For example, some coprocessors may make the most significant 16 bits read only zero. If 1's are written into these bits, the coprocessor will always return 0.

If the forwarding path is activated, 1's would be returned instead. Please refer to the datasheet for the exact instruction sequence which activates the forwarding path.

- Normally, the coprocessor need not be aware of exceptions in the core. Writes occur in the W stage of the pipeline, while exceptions occur in the M stage; therefore, writes are inhibited before they become visible on the interface. Reads occur in S and E stages. If the reads are non-destructive, multiple reads are acceptable. Some coprocessors may perform destructive read operations. Precise exception handling requires that the coprocessor state may not be disturbed by an instruction that is subsequently flushed from the pipeline due to an exception. Since destructive reads cause the coprocessor to change state, the coprocessor design must take steps to restore its state if the read operation is subsequently squashed. The product decathlete gives details on the exception signals visible to the coprocessor interface (*Czxcpn_M* and *Czinvld_M*) and their proper use. Also, exceptions can occur on any instruction due to hardware interrupts or EJTAG breakpoint matches. Therefore, it is not possible to assume that coprocessor instructions themselves will not generate an exception.

- Lexra provides 3 coprocessor interfaces. Of these, Lexra recommends that coprocessor interface 2 ("COP2") be used if only one is needed. The COP1 instructions are used by R3000 class cores that have floating point units (FPU's). Lexra has reserved use of the COP3 instructions for future products.

7.3 Coprocessor Waveforms

Waveforms for several common coprocessor operations follow. Some general notes regarding these waveforms:

- The signals are listed as *Czrd_gen*, etc., where "z" is the coprocessor number (1, 2, or 3). Waveforms and signalling are identical for all 3 coprocessor interfaces.

- The signals *Czrd_cntx* and *Czwr_cntx* are available only with the LX8000 and NetVortex products.

- If signals are valid only during a certain pipeline stage (e.g., *Czrd_gen_S*), the pipeline stage is appended to the end of the signal.

- The waveforms show the signals *Czrd_gen* and *Czwr_gen*. The protocol for *Czrd_con* and *Czwr_con* is identical.

- The signals *i_Cz_stage_M* and *i_Cz_stage_W* are not actual interface signals, nor do they correspond to signals internal to the Lexra core. They are used to demonstrate the pipeline stage corresponding to the illustrated coprocessor operation. For example, *i_Cz_stage_M* "asserts" when the coprocessor operation is in its M stage.

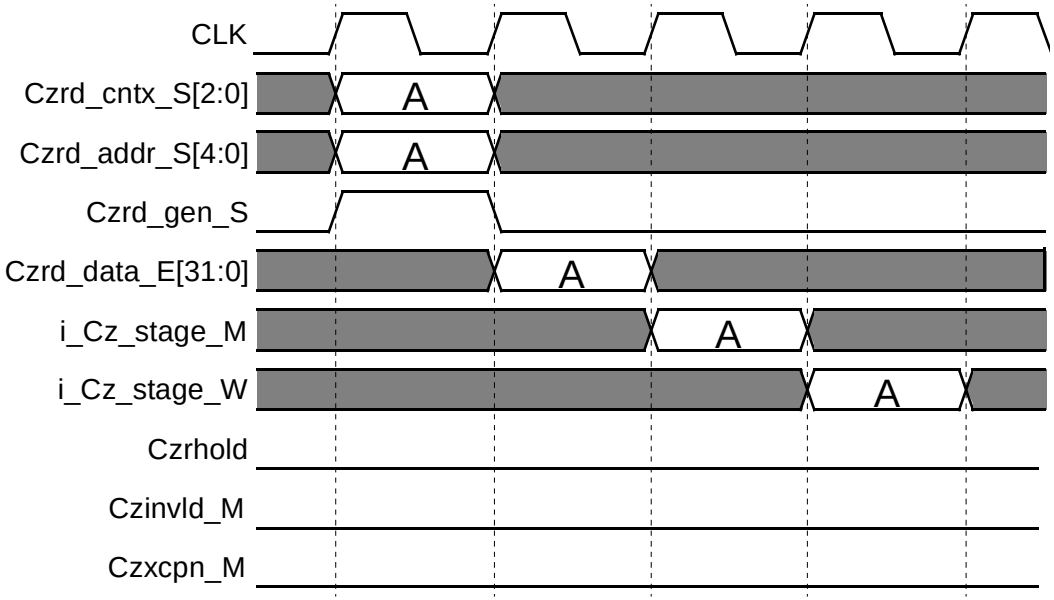


Figure 7-1. Coprocessor Read.

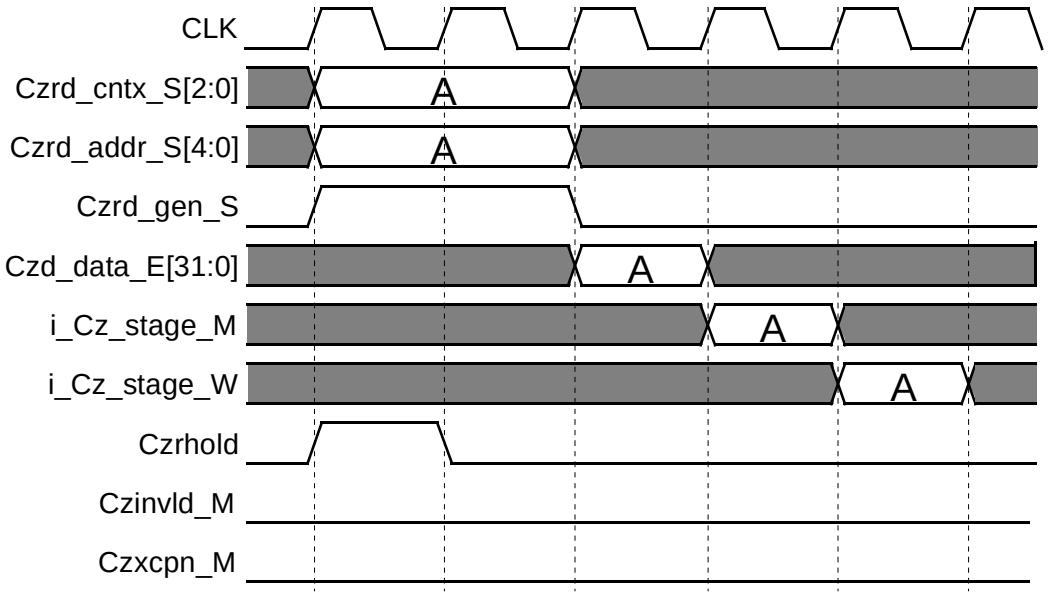


Figure 7-2. Coprocessor Read with S-stage Hold. S-stage inputs must not be sampled until Czrhold deasserts.

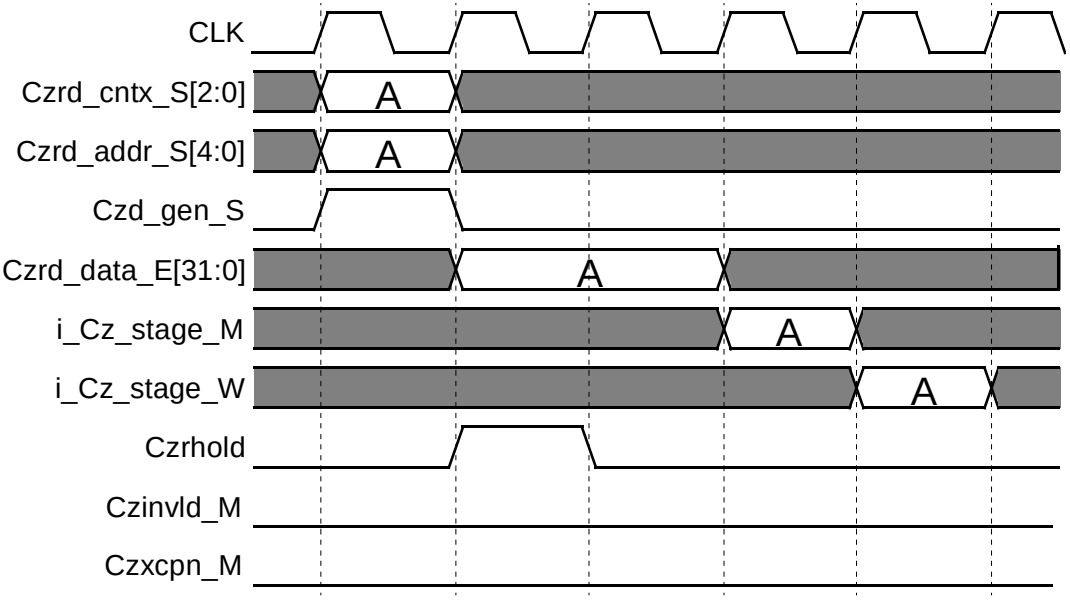


Figure 7-3. Coprocessor Read with E-stage Hold.

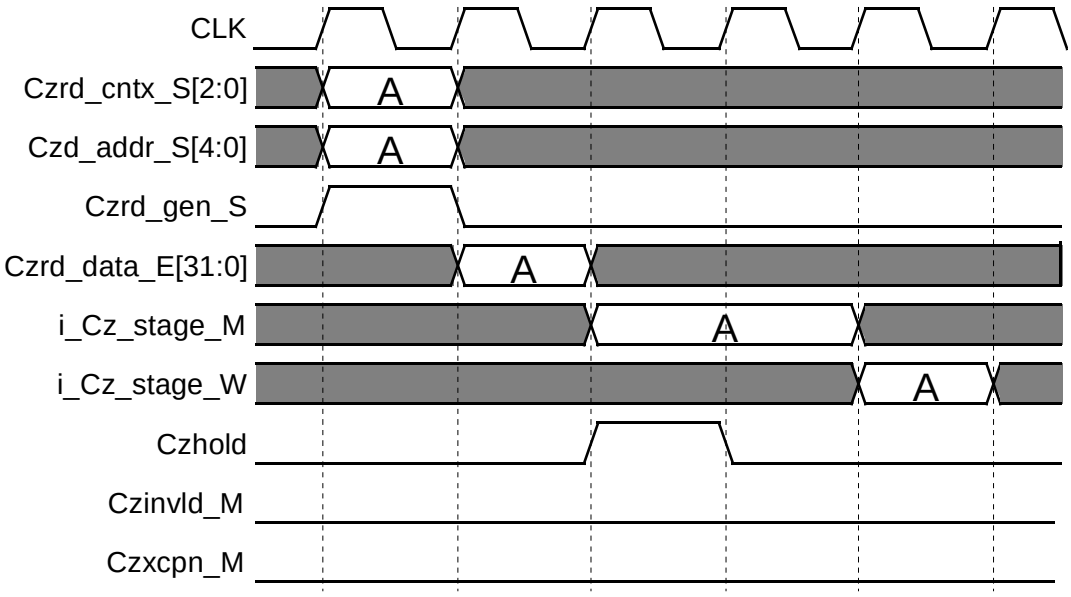


Figure 7-4. Coprocessor Read with M-stage Hold.

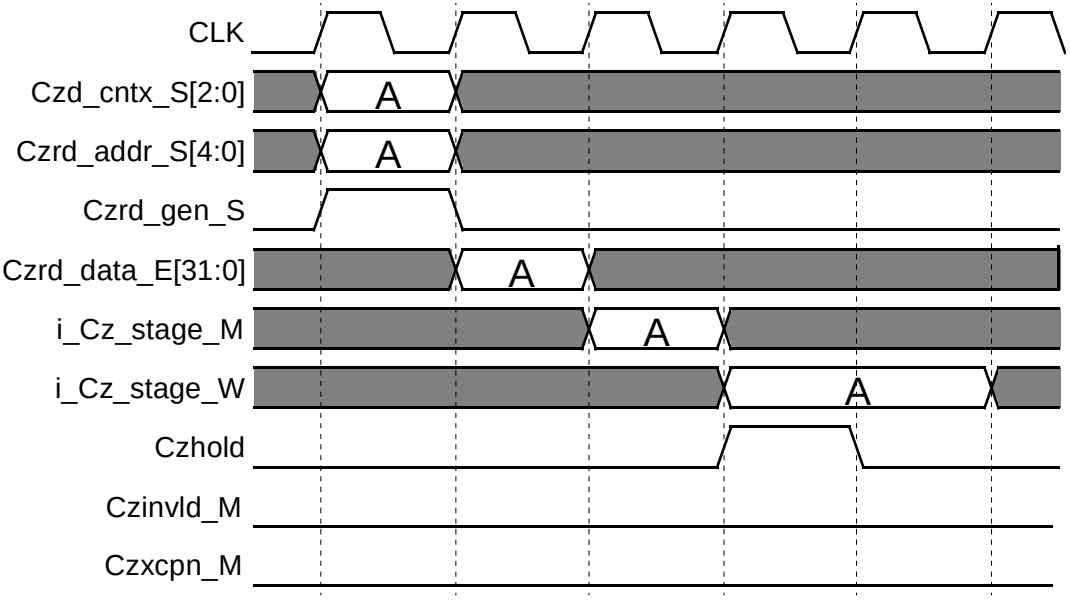


Figure 7-5. Coprocessor Read with W-stage Hold.

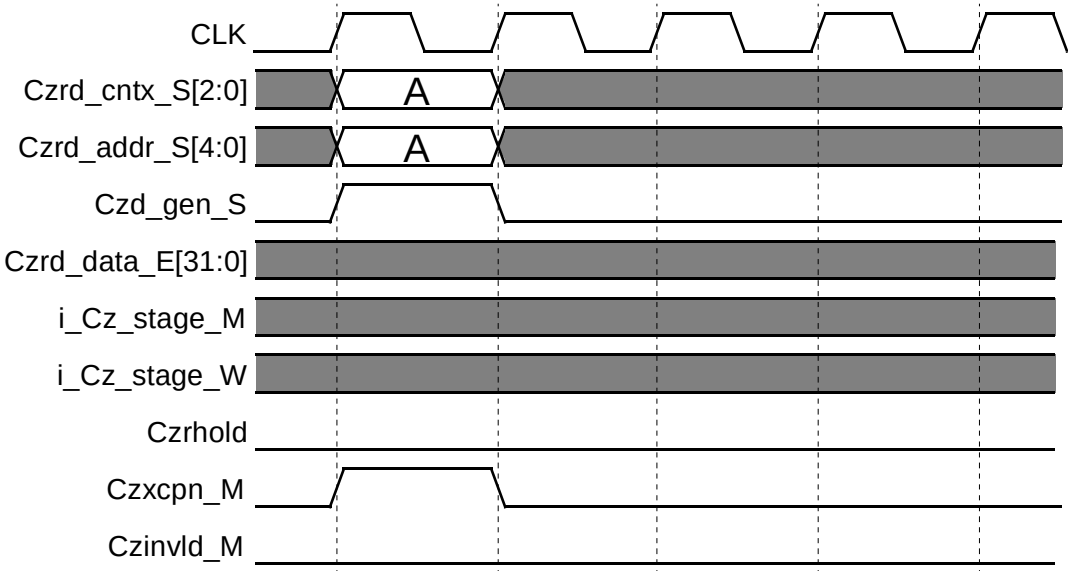


Figure 7-6. Coprocessor Read with Exception in S-stage.

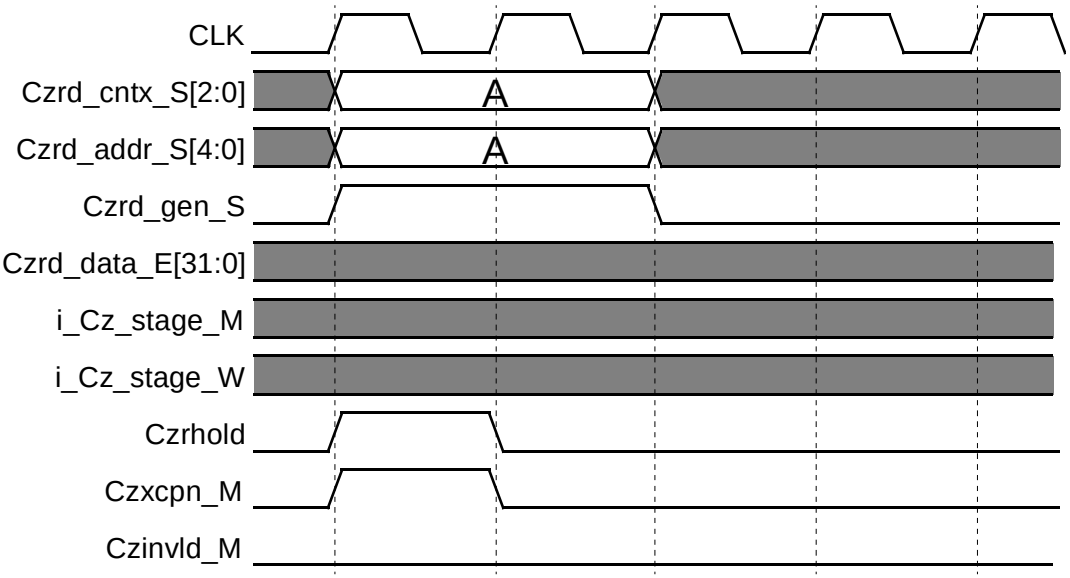


Figure 7-7. Coprocessor Read with Hold and Immediate Exception in S-stage.

Note: Exception must still be honored. This figure, along with Figure 7-8, "Coprocessor Read with Hold and Delayed Exception in S-stage." and Figure 7-9, "Coprocessor Read with Exception following Hold in S-stage." , illustrate that an exception can be raised at any time during or after a pipeline hold.

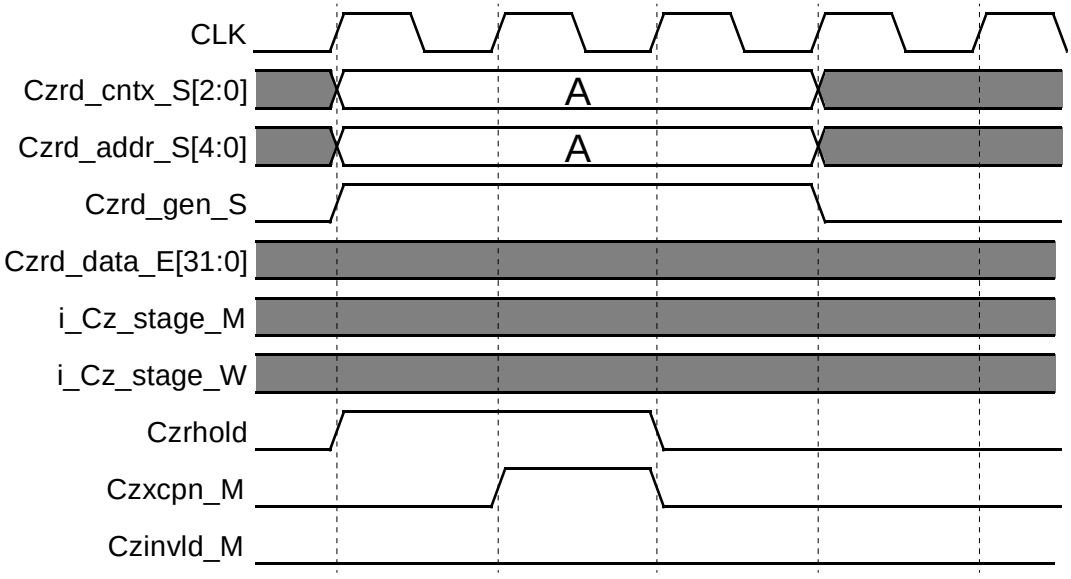


Figure 7-8. Coprocessor Read with Hold and Delayed Exception in S-stage.

This figure, along with Figure 7-7, "Coprocessor Read with Hold and Immediate Exception in S-stage." and Figure 7-9, "Coprocessor Read with Exception following Hold in S-stage." , illustrate that an exception can be raised at any time during or after a pipeline hold.

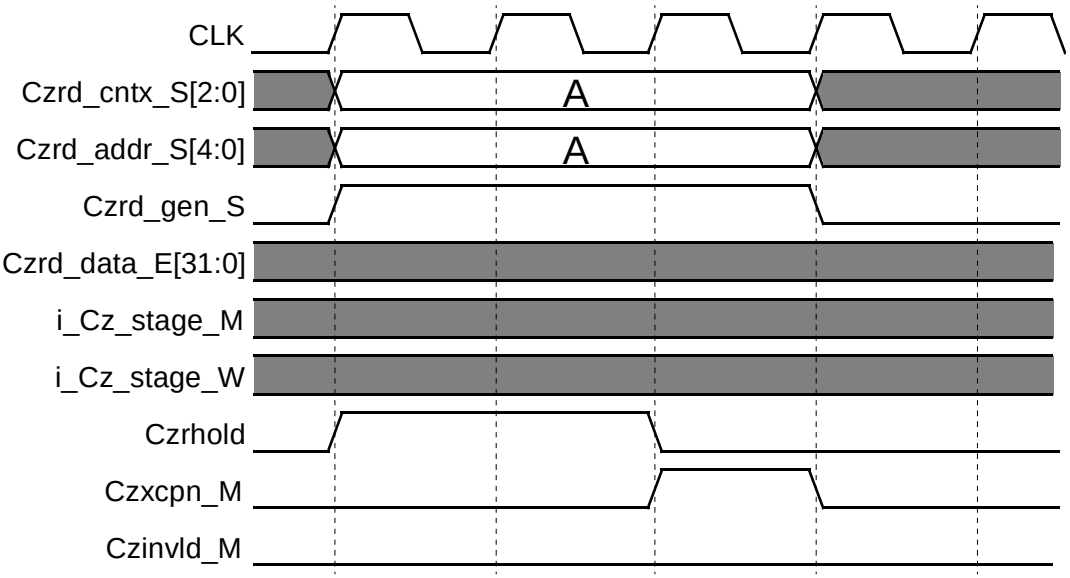


Figure 7-9. Coprocessor Read with Exception following Hold in S-stage.

This figure, along with Figure 7-7, "Coprocessor Read with Hold and Immediate Exception in S-stage." and Figure 7-8, "Coprocessor Read with Hold and Delayed Exception in S-stage." , illustrate that an exception can be raised at any time during or after a pipeline hold.

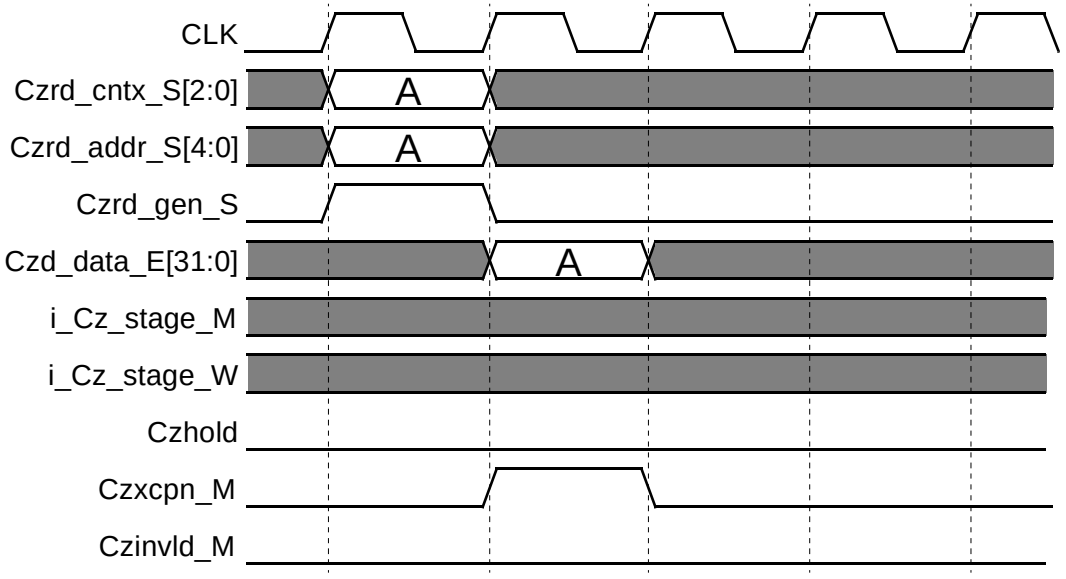


Figure 7-10. Coprocessor Read with Exception in E-stage.

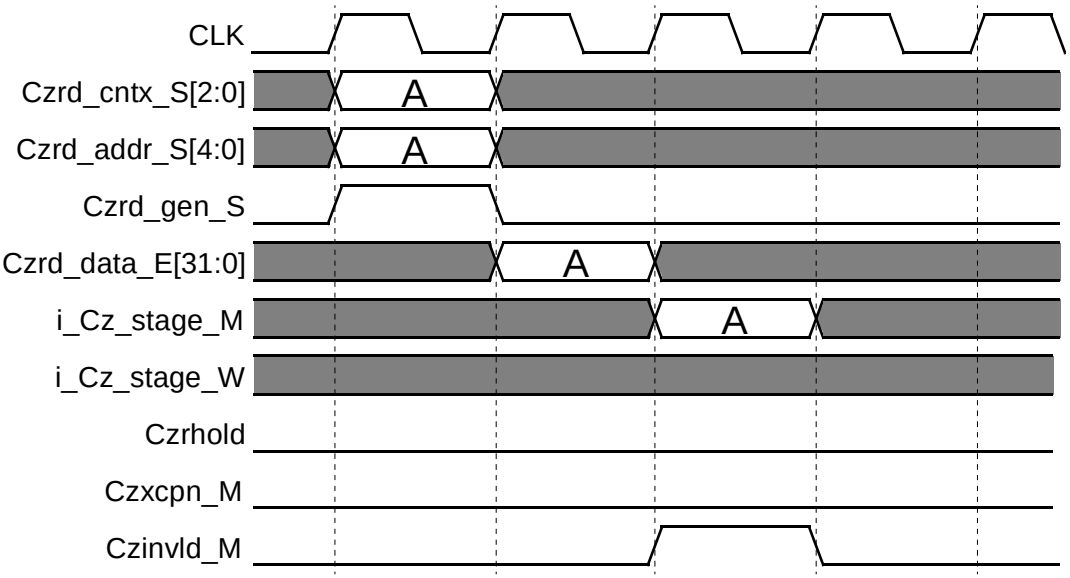


Figure 7-11. Coprocessor Read with Invalidate in M-stage.

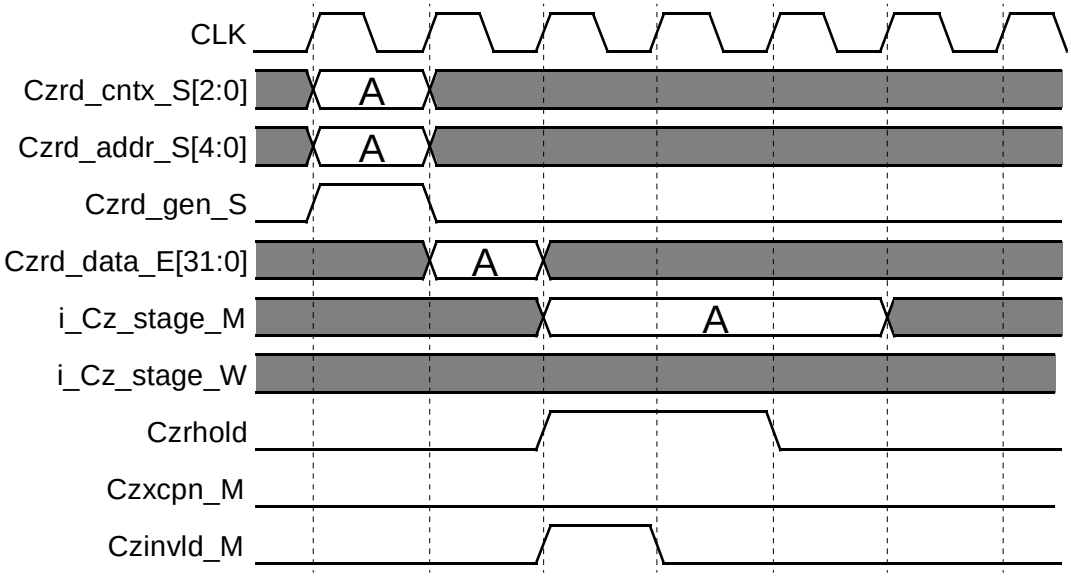


Figure 7-12. Coprocessor Read with Hold and M-stage Invalidate.

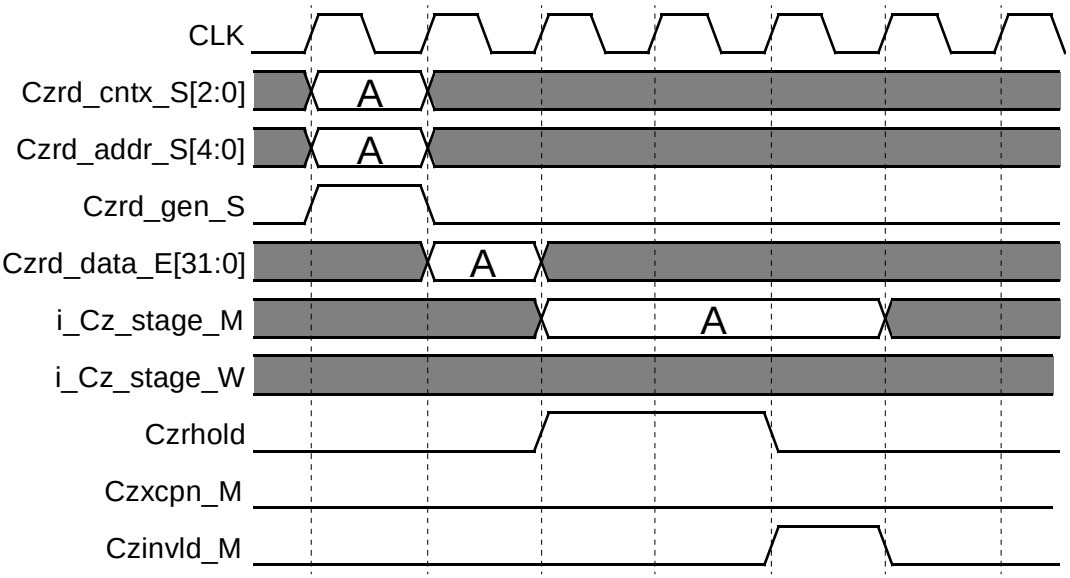


Figure 7-13. Coprocessor Read with Hold and Delayed M-stage Invalidate.

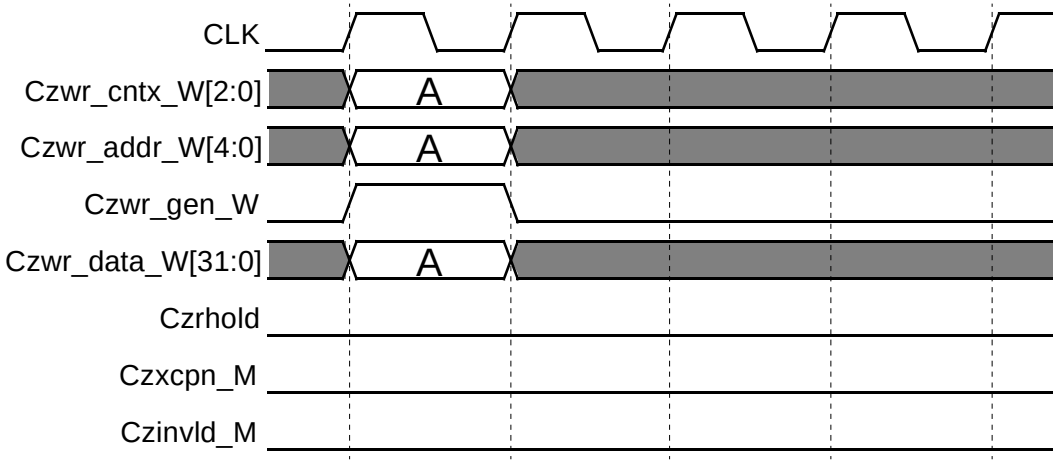


Figure 7-14. Coprocessor Write Operation.

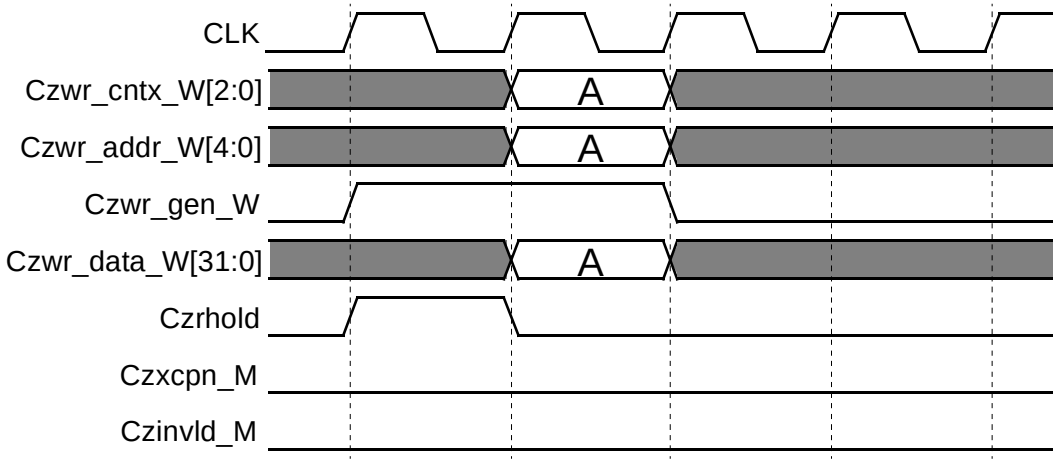


Figure 7-15. Coprocessor Write with W-stage Hold.

Chapter
8

EJTAG

EJTAG is short for MIPS EJTAG Debug Solution. EJTAG provides several hardware features that greatly facilitate debugging of embedded software code. The debug hardware itself is hidden and does not interfere with normal operation of the Lexra processor. Instead, the user accesses these features with an EJTAG compatible debugger and in-circuit emulation (ICE) probe.

The host computer communicates with the EJTAG probe through either a serial or parallel port, or an ethernet connection. The probe, in turn, communicates with the processor's EJTAG hardware through the included IEEE 1149.1 JTAG interface and TAP controller. Using the TAP controller, the probe shifts data to and from the EJTAG data and control registers where it can perform the following:

- respond to processor requests
- redirect direct memory access (DMA) into system memory
- configure the EJTAG control logic
- enable single step mode
- configure the EJTAG breakpoint registers
- enable PC Trace

EJTAG probes are not supplied by Lexra. They are provided by third party vendors specializing in embedded debug. Currently, Embedded Performance Inc. (EPI) and Green Hills Software Inc. provide EJTAG debuggers and probes that support Lexra products.

The Lexra EJTAG implementation supports all required features of the 2.0.0 EJTAG specification. To support these features Lexra has added support for a new debug exception and two new instructions including software debug exception.

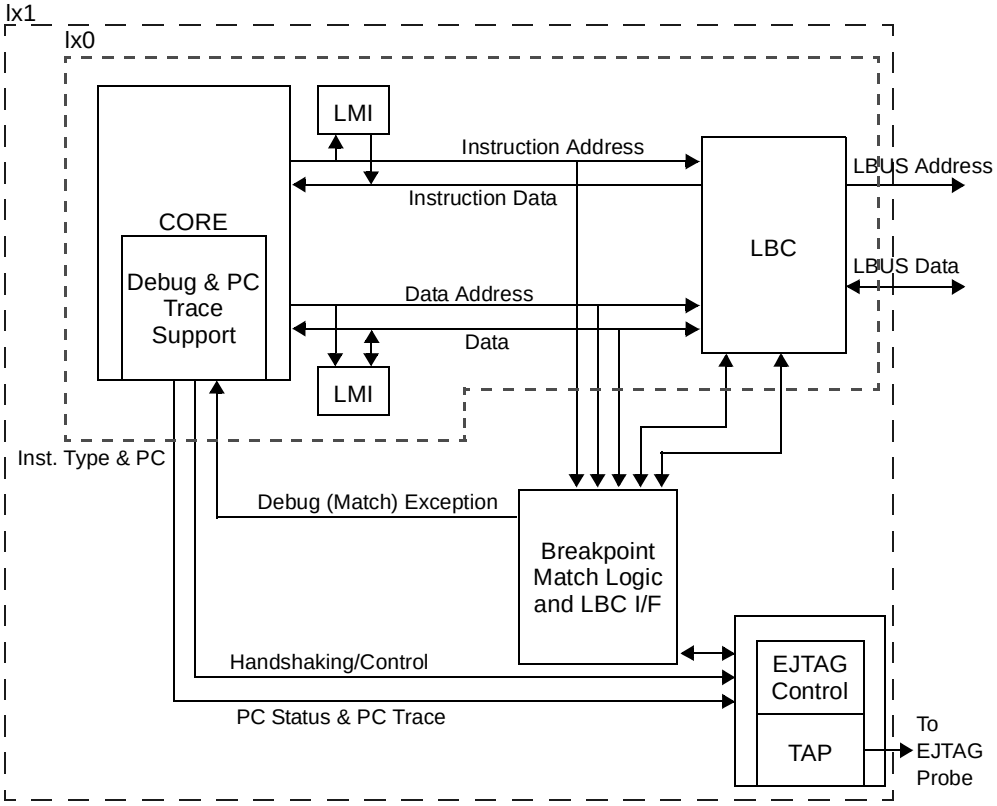
8.1 Architectural Overview: How It Works

8.1.1 Hierarchy and Block Diagram

The Lexra EJTAG implementation consists of the following hardware blocks:

- JTAG TAP controller
- LBC interface
- EJTAG control registers
- Breakpoint match logic
- Coprocessor 0 (COP0) support for new debug exception, new instructions and PC Trace

All the EJTAG modules, except the COP0 registers, reside in the `lx1` hierarchy. See the block diagram below.



8.1.2 Pinout Requirements

EJTAG needs a minimum of five[1] additional pins. These are the standard JTAG connector pins: TDI, TDO, CLK, TMS and TRST. Refer to Lexra's product datasheet for the actual RTL port names.

If the PC Trace EJTAG feature is also required, four additional pins are required for single-issue processors: DCLK and the three-bit PC status signal named PCST. For dual-issue processors, seven additional pins are required: DCLK and two sets of three-bit PC status. For both scenarios, the program counter value is output serially on the TDO pin. That is, TPC[1] is muxed with TDO and outputs

1. Due to multiple ways to reset the tap controller, TRST is an optional pin.

PC trace serially to the EJTAG probe for sampling. If desired, you can add more PC trace pins (TPC) to reduce the serial shift time of the program counter.

The maximum configuration for PC trace requires 20 additional pins: one DCLK, four 3-bit PCST outputs (12 pins) and seven TPC pins (TPC[8:2]).

For maximum flexibility, the Lexra processor provides a warm reset pin, ResetN. Connecting this pin to the EJTAG connector lets you reset the processor and boot from the probe. EJTAG control and match logic are not affected. This allows the debugging of boot code. Thus, the maximum total number of EJTAG signal pins is 26: 5 for the JTAG interface, 20 for PC trace, and 1 for warm reset.

8.1.3 Lexra JTAG TAP Controller

The EJTAG probe enables communication between the host and the Lexra processor via the TAP controller. When you choose the EJTAG option, you must use Lexra's TAP controller since it includes additional instructions for shifting data in and out of the address and data registers, and EJTAG controller.

In designing, you can use Lexra's TAP controller to control internal or boundary scan chains. You can choose to send TAP state information and the instruction register (JTAG_IR) contents from the TAP controller to the `lx2` ports where they can be connected to your own logic. The TAP controller supports the use of up to four instructions defined by you to control scan or test logic.

8.1.4 COP0 Support: Debug Exception, Instructions, Registers

The Lexra processor has COP0 support for the new debug exception, two new instructions and three new EJTAG registers.

The debug exception is the highest priority exception after Reset. Any of the following conditions can cause an exception:

- debug interrupt signal from probe via the TAP interface
- single-step
- breakpoint match
- software instruction (software debug breakpoint: SDBBP)

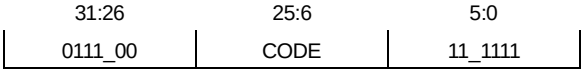
If EJTAG control logic indicates that a probe is connected, the debug exception

handler routine is at `0xFF20_0200`. Since this address is in the probe's address space, the processor downloads the exception routine from the EJTAG probe. If no probe is connected, the exception handler is at `0xBFC0_0200` (kseg1).

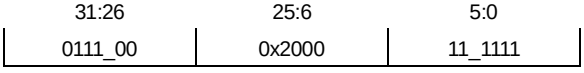
When the processor receives the debug exception, it goes into debug mode. Certain EJTAG operations such as reading and writing of probe address space are valid only while the processor is in this mode. Also, the debug exception is masked by the processor in this mode. The processor can take the debug exception while the processor is servicing a standard exception thereby, letting you debug exception handler routines.

EJTAG requires two new instructions:

- SDBBP - software debug breakpoint: Causes debug exception. Its format is:



- DERET - debug exception return: Causes the program to return to the instruction address stored in the DEPC register. This instruction incurs a delay slot.



The processor also supports three new COP0 registers:

- Debug Register (COP0 register 16): A register containing control and status information such as the cause of the debug exception, single-step enable and EJTAG reset

- DEPC Register (COP0 register 17): The debug exception program counter, containing the address to which the program returns after executing the DERET instruction.

- DESAVE register (COP0 register 18): A general purpose register for the use of the debug exception handler.

8.1.5 Hardware Breakpoints

The EJTAG block contains two sets of breakpoint registers so you can set breakpoints on the instruction and data busses internal to the processor. The EJTAG hardware raises the debug exception when a match occurs. You can set breakpoints on the instruction address, the data address and the data value. Instruction and data breakpoints are checked during reads and writes. You can set up to 15 breakpoints for each type of breakpoint.

The breakpoint registers occupy the virtual address space `0xFF3x_xxxx`. You can address them through both the EJTAG probe and the processor core.

8.1.6 Single-step Mode

Single-step mode is enabled by setting a bit in the EJTAG debug register in COP0. In single-step mode, the hardware raises a debug exception when each instruction completes.

8.1.7 DMA Capability

In debug mode, the processor can reach the probe through virtual memory space `0xFF2x_xxxx`. Similarly, it can reach the breakpoint registers through virtual memory space `0xFF3x_xxxx`. The EJTAG control, address, and data registers resides in the TAP controller and are accessible only through the EJTAG. These registers are not accessible by the processor.

The EJTAG probe uses the LBC to be able to read and write system memory. This facility allows the user to read and write devices on the system bus. The writing and reading of local memory (i.e. DCACHE & DMEM) is done through a process called “instruction jamming”. In this mode the EJTAG probe forces the processor to execute loads and stores with the source or destination of the instruction being an EJTAG register.

8.1.8 PC Trace

Lexra processors using EJTAG, if configured, can also have real-time program counter trace (PC Trace) capability. This feature allows a trace log to be created, showing the history of the instruction execution. This powerful feature can be used to find difficult to capture software problems and for code profiling.

When PC Trace is used, the processor outputs the program counter value serially whenever program flow changes due to a branch, jump or exception. This serial data is then read by an EJTAG probe which supports PC Trace and displayed in the debugger window.

The clock output, DCLK, provides synchronization of the PC Trace signals between the processor and the EJTAG probe. You can program the DCLK output frequency to be 1x, 0.5x, 0.33x or 0.25x the SYSCLK frequency for single-issue processors and 1x or 0.5x the SYSCLK for dual-issue processors. The PC status indicating whether program flow changed because of branches, jumps, exceptions or pipeline stalls is displayed with a three-bit PCST signal. The processor outputs one to four sets of PC status signals depending on the `lconfig` option EJTAG_DCLK_N. Single-issue processors output one set of PC status signals per SYSCLK while dual-issue processors output two sets of PC status per SYSCLK as shown in the table below.

EJTAG_DCLK_N	Single-issue DCLK/SYSCLK	Single-issue # of PCST sets	Dual-issue DCLK/SYSCLK	Dual-issue # of PCST sets
1	1/1	1	n/a	n/a
2	1/2	2	1/1	2
3	1/3	3	n/a	n/a
4	1/4	4	1/2	4

The 32-bit program counter value is output serially on the TDO pin. The EJTAG hardware can also be configured to use additional TPC pins to output the program counter 2, 4 or 8 bits at a time; thereby reducing the number of clock cycles required to output the full PC value. The TDO signal represents the least significant bit of the program counter (TPC[1]), while additional TPC pins (TPC[8:2]) contain the additional bits if configured in that manner. Use the `lconfig` option EJTAG_TPC_M to specify the number of pins you want to dedicate to output the program counter.

To support multi-processor debugging with daisy-chaining TAP controllers, de-muxing TPC[1] with TDO is necessary. See Section 8.2.2, Multi-processor Debugging.

After program control flow changes, it can take up to 32 clock cycles to output the new value of the program counter. Therefore, it is possible for program flow to change while EJTAG is still shifting out the previous program counter. EJTAG

has two operating modes that determine how the program counter trace behaves in these situations. In real-time mode, you truncate the old program counter and output the new program counter instead. In non-real-time mode, the processor pipeline stalls while the old program counter output completes. In either case, using more pins for the PC trace output reduces the likelihood of stalls or truncations.

If you use hardware low overhead vectored interrupts, the TPC pins output a 4-bit code with the most significant bit set to 1 to indicate which one of the eight hardware interrupt vectors, numbered 8 through 15, has been taken when the EXP code is output on the PCST pins. For other exceptions, the 4-bit code with the most significant bit set to 0 has the standard value on its three least significant bits.

If your EJTAG probe vendor does not support this Lexra specific extension to 4 bits or if you do not use hardware vectored interrupts, you may set the `lconfig` option EJTAG_XV_BITS to "3" to disable the 4-bit code. In that case, only the standard 3-bit code is used when the EXP code is output. If the 4-bit code is disabled and a hardware vectored interrupt is taken, the NMI/Reset 3-bit code is used.

If the configuration of the processor includes support for MIPS16, the `lconfig` option EJTAG_PC_ISABIT is used to determine the number of bits to be used for the PC Trace PC that is driven serially on the TPC line(s) when the JMP code outputs on the PCST pins.

The EJTAG specification states that only a 31-bit PC (bits 31:1) should be output on the TPC line(s) by systems that are capable of executing code compressed, 16-bit instructions (MIPS16 ISA mode). Systems that are not capable of executing in MIPS16 mode use a 30-bit PC (bits 31:2). Some debug software that uses PC Trace information from an EJTAG probe can make good use of the ISA mode supplied as bit 0 of the PC. This parameter supports such debug software by enabling a 32-bit PC for systems that are MIPS16 capable. For these systems, if this parameter is set to "YES", then bit 0 is the first bit to be output on the TPC line(s) and has the value "1" if the target of the JMP type instruction is in MIPS16 ISA mode, and has the value "0" otherwise. This usage of bit 0 is consistent with its use in the EPC register and as the target of a JR instruction.

Note: This parameter is ignored if the processor configuration is not capable of executing MIPS16 mode instructions. In that case, a 30-bit PC (bits 31:2) is always used for output on the TPC line(s).

The correspondence between these values and the actual width of the output PC is shown in the table below.

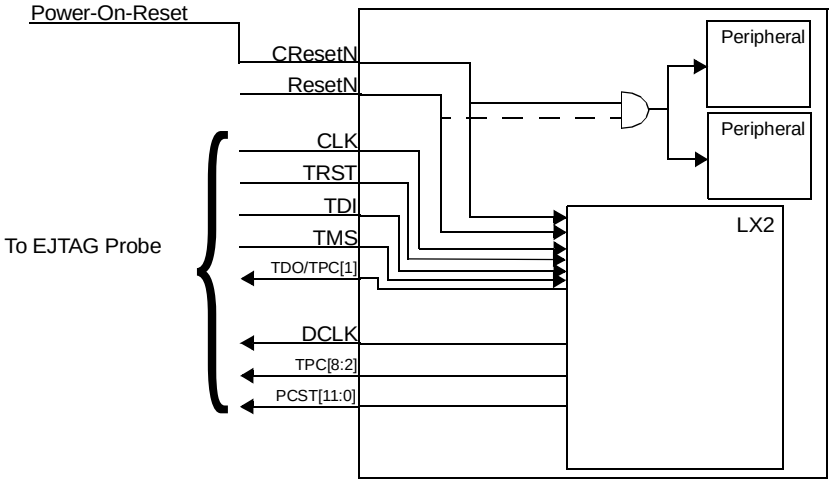
MIPS16 capable	PC Width (bits)	EJTAG_PC_ISABIT
no	30-bits (31:2)	don't care
yes	31-bits (31:1)	no
yes	32-bits (31:0)	yes

The setting of this option is based on the probe and debugger being used. Check with the probe vendor for information on how this feature is supported.

8.2 Designing with EJTAG

8.2.1 Single Processor Debugging

When designing a system with only one Lexra processor, follow the diagram below for EJTAG connections. See your EJTAG probe vendor for specific EJTAG header pin-out information.



The warm reset (ResetN) is connected to the EJTAG header to allow the processor to boot from the probe, thus allowing software developers the ability to debug their reset vector. Depending on the design of the SOC, there may also be a desire to reset peripheral logic when the warm reset is asserted. If this is the case, wire ResetN into your cold reset logic.

Depending on the setting of EJTAG_TPC_M and EJTAG_DCLK_N in the *lconfig* form, certain bits of the TPC and PCST busses will have to be connected to the EJTAG header. EJTAG_TPC_M controls the number of TPC pins that are used to serially output the PC Trace information when a change in execution occurs (i.e. jump, branch, exception). At the chip level, connect these pins according to the table below.

EJTAG_TPC_M	Pins to connect
1	TDO/TPC[1]
2	TPC[2], TDO/TPC[1]
4	TPC[4:2],TDO/TPC[1]
8	TPC[8:2],TDO/TPC[1]

The setting of EJTAG_DCLK_N specifies the number of sets of the 3-bit PCST that are required. At the chip level, connect these signals by following the table below.

EJTAG_DCLK_N	Pins to connect
1	PCST[2:0]
2	PCST[5:0]
3	PCST[8:0]
4	PCST[11:0]

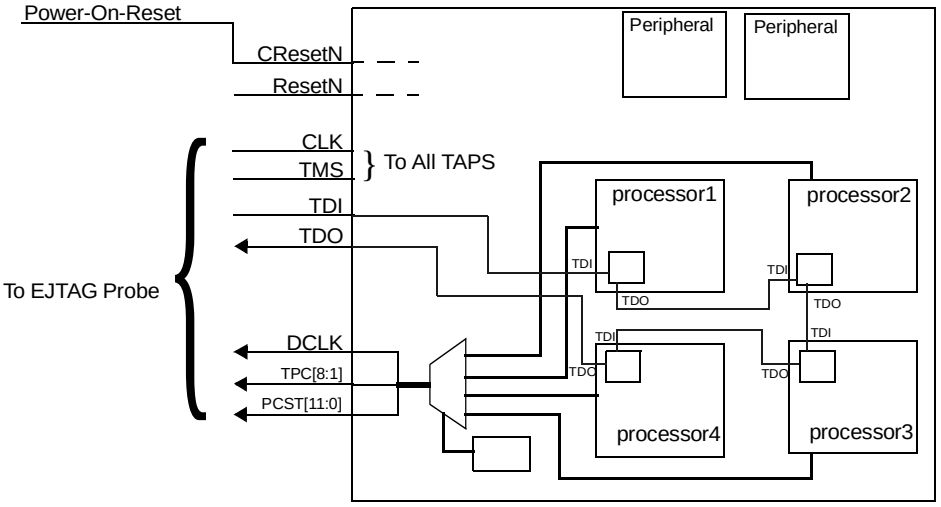
An EJTAG probe model is provided to check these connections. See Section 8.2.8, EJTAG Customer Probe Model

8.2.2 Multi-processor Debugging

In order to facilitate multi-processor debugging, a feature not supported by the EJTAG 2.0.0 specification, Lexra worked with Embedded Performance Inc. (EPI) to come up with an extended and improved EJTAG specification. Currently EPI has the only probes that are tested and verified to support these features. Check with your probe vendor for compliancy.

To minimize complexity and I/O resources, multi-processor debugging can be achieved with only a single EJTAG tap interface. This is made possible by daisy chaining the EJTAG TAP controllers for each Lexra processor. This solution enables independent simultaneous control of each processor with the same pinout as the single processor configuration.

In order to allow seamless operation with external probes, Lexra's EJTAG TAP controller chain must have no register stages between it and the external probe. Any multiplexing of the external pins must be set such that there is a direct connection to and from the Lexra TAP controller chain while using EJTAG.



As shown in the figure above, the TAP controllers are daisy chained together such that the TDO from the first processor is connected to the TDI of processor 2 and the TDO of processor 2 is connected to the TDI of processor 3 etc. The TDO of the last processor on the chain is connected to the TDO pin of the EJTAG probe.

The TAP control signals (CLK, TMS) should fan out from their respective top level pins to each TAP controller. That is, they should be broadcasted to each TAP on the chain.

The signal TRST is removed so that the chip level pin count stays the same when TDO/TPC[1] is de-muxed. TPC[1] is added to the TPC[8:2] bus resulting in TPC[8:1]. With the removal of TRST, the EJTAG probe must be configured to generate a TAP reset by asserting TMS for 5 clock periods.

The demuxing of TDO/TPC[1] is done via the `lconfig` option, JTAG_TRST_IS_TPC. The de-muxing is necessary to avoid confusing the TAP chain when one of the processors is in PC Trace mode and outputting data on it's

TPC pins. The EJTAG probe will need to be configured to accept the TPC[1] signal via the TRST wire on it's EJTAG cable (EPI configuration). This may require a jumper change on the probe cabling.

As shown in the previous figure, the PC Trace signals from each processor are muxed into a single set of PC Trace pins that connect to the EJTAG probe header. Various ways to control the mux can be used, but it is common to map the mux select to a register accessible by the system bus. Using the debugger software, the EJTAG probe can DMA to the system bus to change the mux selection from one processor to another.

Due to the single set of PC Trace pins going to the EJTAG probe, only one processor can be running PC Trace at any given time. Before enabling PC Trace in the debugger, be sure to switch the PC Trace mux to the correct processor. While PC Trace is being captured by the EJTAG probe, the TAP chain can be still be used by the other debuggers communicating to the other processors on the TAP chain.

8.2.3 Clocking

There are two clock domains in the EJTAG circuitry, JTAG_CLOCK and SYSCLK. The maximum frequency of JTAG_CLOCK is 40MHz. (Check with your probe vendor). The two clocks can be asynchronous. The EJTAG design in the processor contains the synchronizing circuitry needed to prevent metastability and other problems involving signals crossing clock domains. The EJTAG DCLK output is synchronized to the SYSCLK; therefore, it does not represent an additional clock domain.

The multiple clock domains have implications for both gate-level simulations and scan testing. Therefore, it helps to know where the clock domain crossings occur. For gate-level simulations, you usually need to modify the SDF (standard delay format for annotating simulation models with timing) entries of the cell instances below to remove the setup/hold time checks. This prevents unknowns from propagating through the simulation.

The following registers are the synchronizing flops at the boundaries of each clock domain.

From JTAG_CLOCK to SYSCLK domain:

lx1/ejtag/ejtag_control/ecr_probeen/REG_D1_R
lx1/ejtag/ejtag_control/ecr_pcasid/REG_D1_R
lx1/ejtag/ejtag_control/jtagbreak/set_toggle2edge/Toggle_D0_R
lx1/ejtag/ejtag_control/pctrace/clr_toggle2edge/Toggle_D0_R
lx1/ejtag/ejtag_control/dma/set_toggle2edge/Toggle_D0_R
lx1/ejtag/ejtag_control/tif/set_toggle2edge/Toggle_D0_R
lx1/ejtag/ejtag_control/tof/clr_toggle2edge/Toggle_D0_R
lx1/ejtag/ejtag_control/pracc/clr_toggle2edge/Toggle_D0_R

From SYSCLK to JTAG_CLOCK domain:

lx1/ejtag/ejtag_control/jtagbreak/clr_toggle2edge/Toggle_D0_R
lx1/ejtag/ejtag_control/pctrace/set_toggle2edge/Toggle_D0_R
lx1/ejtag/ejtag_control/dma/clr_toggle2edge/Toggle_D0_R
lx1/ejtag/ejtag_control/tif/clr_toggle2edge/Toggle_D0_R
lx1/ejtag/ejtag_control/tof/set_toggle2edge/Toggle_D0_R
lx1/ejtag/ejtag_control/pracc/set_toggle2edge/Toggle_D0_R

8.2.4 Using the Lexra EJTAG TAP Controller

The Lexra processor includes an IEEE 1149.1 compliant TAP controller. If you enable EJTAG, you must use Lexra's TAP controller. If you don't implement EJTAG, you have the option of using Lexra's TAP controller.

To enable the TAP controller, set the `lconfig` parameter JTAG = EXPORT. The TAP controller is instantiated as tap in the lx1 module. The five JTAG pins (TDI, TDO, TCLK, TRST, and TMS) appear as ports to the lx2 module. If you enable EJTAG, you can use the TAP controller to shift data in and out of the appropriate EJTAG control and data registers.

The TAP controller implements the SAMPLE, BYPASS and IDCODE instructions. You may want to use the TAP controller to implement other functions like boundary scan control, internal scan control and EXTEST. To make this possible, you can output key signals from the TAP controller state machine to the lx2 module boundary by setting the `lconfig` parameter JTAG = EXPORT_EXTENDED. When you choose this option, `lconfig` brings the pins defined in the table below to the `lx2` boundary.

Decode the TAP instructions on the JTAG_IR bus as follows. When you select JTAG=EJTAG_EXTENDED it is up to you to implement the italicized instructions. New instructions can be defined using one of the four user defined opcodes reserved for this purpose.

EXTEST	**5'h00**
IDCODE	5'h01
SAMPLE	5'h02
EJTAG_IMPLEMENTATION	5'h03
INTEST	5'h04
HIZ	5'h05
CLAMP	5'h06
BYPASS1	5'h07
EJTAG_ADDRESS	5'h08
EJTAG_DATA	5'h09
EJTAG_CONTROL	5'h0a
EJTAG_ALL	5'h0b
EJTAG_PCTRACE	5'h10
user defined instruction 0	5'h18
user defined instruction 0	5'h19
user defined instruction 0	5'h1a
user defined instruction 0	5'h1b
BYPASS	5'h1f

8.2.5 Reset Issues

Using EJTAG, you have several ways of resetting the processor, the JTAG TAP controller and the EJTAG control logic:

- cold reset using CResetN
- warm reset using ResetN
- TAP reset using TRST
- software reset of core
- software reset of TAP

8.2.5.1 Cold Reset

The CResetN pin asserts cold reset or power-on reset. When you assert this pin low, the hardware resets the processor, TAP controller and the EJTAG control logic including the breakpoint match logic. The processor begins to fetch instructions from logical address `0xBFC0_0000` after you deassert CResetN.

Since CResetN resets the breakpoint control logic, you can't use CResetN alone to debug boot code using EJTAG. For example, it will not be possible to set breakpoints or single-step through the boot code. You must use the warm reset for that purpose.

8.2.5.2 Warm Reset

The warm reset, ResetN, resets the processor core along with the LMI, LBC, coprocessors and custom engines. The only EJTAG registers not affected are the probe_enable and the processor_reset bits. Connect ResetN to the EJTAG probe.

For example, by using the warm reset function EJTAG can gain immediate contol of the processor after reset. This is done by the EJTAG probe asserting the probe_enable bit and then issuing a warm reset. The EJTAG control logic indicates to the processor that the EJTAG probe is present and thus changing the reset vector to `0xFF20_0000`, instead of the normal `0xBFC0_0000`. The processor will then download real boot code through the EJTAG probe.

8.2.5.3 Software Reset

There is a software reset of the core that behaves like the warm reset. Enable it by setting bit 16 of the EJTAG control register to 1 and then set it to 0 to deassert reset.

You can also reset the TAP controller and the EJTAG control logic without resetting either the processor or the breakpoint logic. The three ways to do it are:

- assert TRST signal (unless configured for multiprocessor debug)
- set JTAG_TMS signal to 1 for a minimum of five JTAG_CLOCK cycles
- set bit 7 of the COP0 debug register to 1

Depending on you how you configure the connection of the EJTAG probe to the lexra TAP controller and whether the implementation supports multiple processors on a single TAP chain, various settings in the debugger and probe will need to be specified. These settings define the correct method for the probe to use to reset the processor and accompanying EJTAG and TAP controller.

8.2.6 Gate Count per Breakpoint

The number of gates required per breakpoint varies depending on the type of breakpoint you select. Instruction breakpoints only analyze an address match and are the least costly in terms of area. Data bus breakpoints are the most costly as they have both address and data matching.

Instruction breakpoints are the most useful. If you set a breakpoint at an instruction address, the breakpoint exception occurs before the instruction completes.

You can set data breakpoints on matching either address or data. The processor checks for breakpoint matches during either cached or uncached data fetches. Data breakpoints are useful for analyzing load/store operations. The processor takes the breakpoint exception after the subsequent instruction completes.

Instruction breakpoints require approximately 1K gates per breakpoint. Data bus breakpoints on the other hand require approximately 2K gates per breakpoint.

8.2.7 Memory Addressing

We reserve portions of the MIPS address space for the EJTAG probe and debug registers. The EJTAG probe addresses, including the debug exception handler, occupy the address space from `0xFF20_0000` through `0xFF2F_FFFF`. The debug control and breakpoint registers occupy the address space from `0xFF30_0000` through `0xFF3F_FFFF`. When the processor is in debug mode, these memory locations are mapped to the EJTAG probe space and the EJTAG debug control and breakpoint registers, respectively. Access to these registers is valid only when the processor is in debug mode.

When the processor is not in debug mode, these memory locations default to uncached system memory in kseg2. Therefore, **do not** use these memory locations for system devices.

8.2.8 EJTAG Customer Probe Model

To help customers validate the EJTAG connections between the processor(s) and the chip level pins, Lexra provides an EJTAG Probe model & testbench. This model and testbench used with the chip level netlist verify the connectivity of all PC Trace and TAP signals including daisy-chained TAP controllers. See the `$LX_HOME/testbed/README.probecust` file that comes with the probe model for detailed information on how to use it.

8.3 Implementation Issues

8.3.1 Special Requirements

The PC Trace signals, PCST and TPC, can only change within a +/- 2ns window from the rising edge of the DCLK signal. Check with the EJTAG probe manufacturer for any changes to this specification.

Because of this tight timing requirement, you need to pay special attention to the timing relationship between DCLK, PCST and TPC (including TDO). Depending on the configuration, synthesis constraints and target ASIC library, you may need to modify the synthesized netlist in order to meet this timing requirement.

This timing requirement is only necessary if DCLK is running at its maximum frequency of 100MHz. The PC Trace probe samples PCST and TPC/TDO on the falling edge of DCLK, so this timing specifications gives the probe 3 ns of setup and 3ns of hold time on these signals relative to DCLK. Therefore, this timing requirement can be relaxed if DCLK is running slower than 100MHz.

8.3.2 Unimplemented Features from EJTAG Specification

Lexra's EJTAG implementation supports the required features of Revision 2.0.0 of the EJTAG specification. It does not implement the following optional features:

- complex break
- processor break
- data break enhancements
- EJTAG memory map in normal mode

- memory overlay
- DMA operation ABORT request
- indication of debug mode by hardware signal
- support for address space identifier (ASID) in break or PC trace

8.3.3 Implemented Optional Features from EJTAG Specification

The processor implements the following optional feature from Revision 2.0.0 of the EJTAG specification:

- debug exception vector in normal memory
- profiling by DMA read of PC/ASID

Chapter
9

Testability

The Lexra cores are designed for testability and complies with DFT (design for testability) rules. In addition, the Lexra core includes various options to synthesize test structures in the design automatically.

You can configure three test structures:

- internal scan
- a memory scan collar
- RAM test access interface

The internal scan is based on a full scan, muxed flip-flop scan architecture. If selected, scan insertion runs automatically during Lexra synthesis. The scan architecture includes special features to allow easy integration into your ASIC.

An optional memory scan collar provides additional observability and controllability of the Lexra core logic in the shadow of the instruction and data cache RAMs. It allows a combinational ATPG engine to achieve high fault coverage with black boxed memories (not provided by Lexra).

The RAM test access interface, also called the RAM BIST interface, allows external logic such as a BIST engine to directly access the pins on the RAMs, thereby simplifying the testing of cache tag and store RAMs.

You configure testability just as you do all other options, using the `lconfig` form.

9.1 Internal Scan

9.1.1 Scan Methodology Overview

The Lexra core includes scripts for scan insertion and ATPG. Scan is inserted using Design-Compiler XP™ from Synopsys during normal logic synthesis. During scan insertion, all the scannable flip-flops are replaced with their scan equivalent assuming a muxed flip-flop architecture. LSSD architectures are not supported by the Lexra provided scripts.

There are 3 possible approaches to scan and ATPG with the Lexra core:

- Scan insertion and ATPG performed at the Lexra core level. Using this approach, scan insertion is done during the logic synthesis of Lexra core. The `lconfig` options for scan insertion are described in this chapter, while logic synthesis is described in detail in Chapter 11 Synthesizing the Lexra CPU. Lexra's provided ATPG scripts can be used to perform ATPG at the Lexra core level. This approach is recommended for most customers, especially when multiple Lexra cores are instantiated in the design.

- Scan insertion done at the Lexra core level, with ATPG done at the whole chip level. Using this approach, scan insertion is done during the logic synthesis of the Lexra processor. The `lconfig` options for scan insertion are described in this chapter, while logic synthesis is described in detail in Chapter 11 Synthesizing the Lexra CPU. This approach can simplify the overall design flow by reducing the number of steps in the ATPG process. The customer must still ensure that they maintain complete controllability on the clock and reset pins on the Lexra core. The full list of signals is shown in Section 9.1.4, Internal Scan Interface. **Please note that it is not possible to use Lexra's ATPG scripts when performing ATPG at the chip level.** Also, note that if many Lexra cores are instantiated, performing ATPG at the chip level may exceed the capacity of the ATPG EDA tools

- Whole chip scan insertion and ATPG using customer provided scan insertion methodology and scripts. Using this approach, the Lexra core is synthesized with scan insertion disabled. Scan insertion and ATPG are performed at a higher level of the customer's design database. This approach can simplify the overall design flow by

reducing the number of steps in the scan insertion and ATPG processes. The customer must still ensure that they maintain complete controllability on the clock and reset pins on the Lexra core. The full list of signals is shown in Section 9.1.4, Internal Scan Interface. **Please note that it is not possible to use Lexra's scan insertion or ATPG scripts when performing scan insertion at the chip level.** Also, note that if many Lexra cores are instantiated, performing scan insertion or ATPG at the chip level may exceed the capacity of the scan test and ATPG EDA tools.

When performing scan insertion at the Lexra core level, the user can specify the number of scan chains and the crossing of clock domains using `lconfig`.

When using the Lexra provided ATPG scripts, the user can specify the degree of controllability and observability on the I/O signals of the Lexra core. There are two options available to the user:

- A worst-case assumption in which only the scan in, scan enable, clock, reset, and test signals are controllable and only the scan out signals are observable. Under this scenario, logic between the input signals and their destination flip-flop and logic between the source flip-flops and output signals are not testable.

- A best-case assumption in which all of the Lexra core I/O's are controllable and observable. While this scenario provides the best fault coverage, it does require the use of a scan isolation collar around the Lexra core. The design and implementation of this scan collar is the responsibility of the customer.

Refer to Section 9.5, Testability Statistics for details on testability statistics under best and worst case conditions.

9.1.2 Internal Scan Options

You can configure the Lexra core with or without internal scan.

The Lexra supported scan architecture is a full scan, muxed flip-flop methodology.

You can specify the number of scan chains to be 1,2,4 or 8. The synthesis tool creates the appropriate number of scan ports in the RTL code (where these ports are not connected) and hooks up the scan chains in the synthesis flow.

You can also specify whether all flip-flops within the same scan chain must be in the same clock domain. If you allow mixed clocks within a scan chain, Design-Compiler(tm) inserts lockup latches in the scan chain between clock domains. Separate clock domains typically have separate clock distribution trees. While the clock skew within the same clock domain can be tightly controlled with careful design methodology, it is much more difficult to control clock skew between clock domains. Therefore, lockup latches, which typically operate on the negative edge of the same clock as the leading flip-flop, are needed to avoid potential clock skew problems between separate clock domains. The lockup latch has the effect of delaying the transition of the scan output of the leading flip-flop by one-half a clock cycle, thereby providing sufficient hold time to the scan input of the trailing flip-flop. Multiple clocks per chain give Test-Compiler more flexibility to create balanced scan chains, resulting in shorter maximum scan chain length and therefore reduced test time.

9.1.3 `Lconfig` Options

The following `lconfig` options configure the Lexra core to have scan chains.

To control insertion of testability ports, and scan insertion during synthesis, select:

SCAN_INSERT = YES | NO

- If you select YES as the SCAN_INSERT option, scan will be inserted in the core as well as in the optional MAC, LBC, and EJTAG (if selected) and their scan enable and chains will be brought out to the `lx2` level. Turning on scan insertion will also result in the following pins being exported outside of `lx2`:

SEN	Scan enable
SIN	Scan input (one per scan chain)
SOUT	Scan output (one per scan chain)
TMODE	Test mode signal

- If you select NO as the SCAN_INSERT option, scan insertion will be disabled during logic synthesis. No scan signals will appear at the boundary of `lx2`.

To control how many scan chains are inserted in the Lexra core (`lx2` level), set:

SCAN_NUM_CHAINS = 1 | 2 | 3 | 4 | 8

- 1 -- One scan chain will be present in the core.
- 2 -- Two scan chains will be present in the core.
- 3 -- Three scan chains will be present in the core.
- 4 -- Four scan chains will be present in the core.
- 8 -- Eight scan chains will be present in the core.

To specify whether you will allow multiple clocks per scan chain, set:

SCAN_MIX_CLOCKS = YES | NO

- Choose YES if you would like to allow the scan chains to cross clock boundaries.
- Choose NO if you do not want scan chains to not cross clock boundaries.

Note that the Lexra core has up to 3 clock domains depending upon configuration (SYSCLK, BUSCLK, and JTAGCLK). If you set SCAN_MIX_CLOCKS to NO, you must ensure that you specify enough scan chains with the SCAN_NUM_CHAINS options.

9.1.4 Internal Scan Interface

We designed the Lexra processor so that you can get high fault coverage by controlling and observing a small subset of the I/O pins of the `LX2` boundary.

The worst case fault coverage number as shown in Section 9.5.2, Example assumes that the test hardware has access only to the following pins:

input	SYSCLKF	Free-running system clock (only present if SLEEP is selected)
input	BUSCLKF	Free-running bus clock (only present if SLEEP is selected and LBC_SYNC_MODE is ASYNCHRONOUS)
input	SYSCLK	Gated system clock.
input	BUSCLK	Gated bus clock (only present if LBC_SYNC_MODE is ASYNCHRONOUS)
input	JTAG_CLOCK	Clock for JTAG and EJTAG
input	JTAG_TRST_N	Reset signal for JTAG and EJTAG
input	ResetN	Standard reset signal
input	CResetN	Power-up reset signal
input	JTAG_RESET	Buffered version of EJTAG reset domain
input	TAP_RESET_N	Buffered version of TAP reset domain
input	RESET_D1_R_N	Buffered reset for SYSCLK clock domain
input	RESET_D1_BR_N	Buffered reset for BUSCLK clock domain
input	SEN	Scan enable
input	TMODE	Test mode
input	SIN[N-1:0]*	Scan In bus
output	SOUT[N-1:0]*	Scan Out bus

*N is the value SCAN_NUM_CHAINS selects.

The TMODE, or Test Mode, signal is used to configure the Lexra core for testability. For example, in configurations that use an asynchronous reset, the reset signals must be gated off by TMODE so that the asynchronous reset can be explicitly controlled during ATPG. Otherwise, spurious resets could occur during scan chain shifting or scan capture. The Lexra core uses TMODE to gate off asynchronous resets. TMODE is also used to enable the memory scan collar.

Refer to section Section 9.7.3, Reset Distribution for further information regarding reset signals and their impact on test.

9.1.5 Scan Enable Distribution

The scan enable signal, SEN, is a global signal. SEN requires proper distribution and buffering. While it is rare for the scan chain to run at the full system clock speed, it is necessary to ensure that the scan enable signal is properly buffered or the design rule constraints of the ASIC or COT library will be violated. The following `lconfig` options control the buffering and distribution of scan enable:

SEN_DIST = GLOBAL | LOCAL_BUFFERED | NONE

- Choosing GLOBAL causes a global scan enable signal to be distributed throughout the core. No buffers will be placed on scan enable, except optionally at the `lx2` module level (refer to SEN_BUFFERS option description below). Choose this option if you prefer to use your own clock distribution scheme for scan enable.

- Choosing LOCAL_BUFFERED causes the scan enable signal to be buffered at each module boundary during scan synthesis. Choose this option if you prefer to have synthesis buffer the scan enable signal for you.

- Choose NONE only if you set SCAN_INSERT to be NO.

SEN_BUFFERS = LX2 | EXTERNAL | NONE

- Choosing `LX2` causes a special buffer for scan enable signal to be placed in the `lx2` module boundary. The module will be called `lx2_senbuf`. It will be necessary to replace this module with your own clock buffer during final synthesis of the `lx2` module. Choose this option only if you choose GLOBAL for SEN_DIST and you wish to buffer scan enable inside `lx2` rather than at the full chip level.

- Choosing EXTERNAL results in no buffers being placed inside the `lx2` module boundary for scan enable. Choose this option if you select LOCAL_BUFFERED for SCAN_DIST. Alternatively, you can choose this option if you set SCAN_DIST to GLOBAL and you wish to buffer scan enable at the full chip level rather than at the `lx2` level.

- Choose NONE only if you set SCAN_INSERT to be NO.

9.2 Memory Scan Collar

9.2.1 Scan Collar Overview

Most ATPG tools model RAM and ROM memories as black box elements. As a result, any combinatorial logic on the RAM inputs and outputs will be untestable, resulting in loss of fault coverage. Typically, the coverage loss can be as much as 3-5%, depending upon the cache and local memory configuration.

Therefore, the Lexra core includes an optional memory scan collar, which provides the ability to observe the output of the logic cone connected to the RAM inputs and to control the input of the logic cone connected to the RAM outputs. The structure of the memory scan collar includes an observability flip-flop that captures the signals between the logic internal to the `lx1` module boundary and the RAM inputs. This flip-flop, which is scannable, is also used to control the input of a 2:1 mux which resides between the RAM outputs and the internal logic cone inside `lx1`.

You can only implement the memory scan collar when you configure the Lexra core with internal scan. The memory scan collar does induce a small area penalty and also adds a 2:1 mux in the critical path of the RAM outputs. The RTL includes the scan collar in the `lx0c`, between the `lx0` and `lx1` levels of the Lexra hierarchy.

The memory scan collar does not provide test access to the RAM pins themselves; such access is provided by the separate RAM BIST interface described later in this chapter. Also, the memory scan collar does not provide controllability or observability of the inputs and outputs of the `lx2` module boundary. Such controllability must be provided by an external scan collar, which is not provided by Lexra.

9.2.2 `lconfig` Option

To control whether the memory scan collar is included, set the following option in `lconfig`:

SCAN_SCL = YES | NO

- YES is only valid if you configure the Lexra core for internal scan. When you specify a scan collar, it provides coverage on

all memories in your configuration. Note that the scan collar is provided on all of the memory interfaces, including store and tag RAMs. It is not possible to individually select which RAMs are isolated using the scan collar.

- By choosing NO, no memory scan collar will be inserted.

9.2.3 Scan Collar Interface

Synthesis automatically merges the scan collar with existing scan chains for other blocks. Therefore, the scan collar synthesis and ATPG is completely transparent to the customer. The scan collar itself is not faulted during ATPG.

9.3 RAM Testing

9.3.1 RAM Test

To assist in the testing of the on-chip instruction and data memories specific to the Lexra core, Lexra provides an optional RAM test port called the RAM BIST interface.

The RAM BIST interface logic resides with the memory scan collar inside the `lx0c` module. The RAM test logic is implemented using a 2:1 MUX on all of the RAM cell inputs. One input of the mux is the normal signal between the Lexra core and the RAM input. The other input of the mux is a signal external to `lx2` which is driven by the customer's test logic. The output of the mux goes directly to the RAM input. The mux select is a signal external to `lx2` which is also driven by the customer's test logic. The RAM outputs are also brought out to the `lx2` module boundary so that the external test logic can observe them.

The RAMs can be tested in a variety of ways once the BIST interface is enabled; although, the most common method is to use an on-chip BIST engine. Lexra does not provide any special vectors to test the RAM, nor does Lexra provide the BIST logic itself. User's should consult directly with their memory vendor or foundry regarding specific RAM test requirements and the proper implementation of any BIST logic.

Selecting the RAM BIST interface does result in a 2:1 mux to be placed in the critical path on the RAM inputs. It is safest to test the RAMs while you hold the rest of the core in reset through the assertion of ResetN or CResetN.

9.3.2 `Lconfig` Option

To specify a module providing access for memory testing in the Lexra Processor, set:

RAM_BIST_MUX = YES | NO

You can set testing for RAMs whether or not you have implemented scan chains and scan collars.

9.3.3 RAM Test Interface

The pins of the RAM test interface are described in detail in the product datasheet. A typical configuration would include the following pins.

input	RBC_SEL[7:0]	selects memory to test (see below)
input	RBC_WE	write enable to selected memory
input	RBC_RE	read enable to selected memory
input	RBC_CS	chip select to selected memory
input	RBC_ADDR[15:0]	address bits to selected memory
input	RBC_DATAWR[63:0]	write data bus to selected memory
output	RBC_DATARD[63:0]	read data bus from selected memory

The RBC_SEL signal is used to control which memory is being tested. Only one memory is selected for test at any given time, as there is only one set of interface pins on the `lx2` boundary:

RBC_SEL[7:0]	**Memory selected**
10000000	IMEM
01000000	DMEM
00100000	Data cache data store
00010000	Data cache tag store
00001000	Instruction cache tag store, set 1
00000100	Instruction cache data store, set 1
00000010	Instruction cache tag store, set 0
00000001	Instruction cache data store, set 0
00000000	none, RAM BIST inactive

Please note that the above encoding for the RBC_SEL pins are subject to change. For the latest encoding, please refer to the product datasheet. The above encodings are primarily intended for illustrative purposes.

Both the RBC_ADDR and RBC_SEL vector are a fixed width no matter what the user configuration. If a customer configuration does not implement a specific memory (for instance, no DMEM), the behavior of the RAM test hardware is undefined when RBC_SEL[6] is active. It should be always assigned logic 0. Similarly, the number of RBC_ADDR bits in use in each memory depends on the memory configuration. The customer is responsible for tying the high-order bits inactive.

Some memories do not use the full complement of data bits and you need to mask them.

The BIST, ATE or DMA engine needs to write and read the tag bits through RBC_DATA_WR(N:1) and RBC_DATA_RD(N:1), where N is the width specified in the RAM requirement summary.

For example, for the ICACHE TAG bits for set 1 (indicated by tag store and `LRU flag` in the table), the width is 25 bits. The BIST, ATE, or DMA engine needs to write them through RBC_DATA_WR(25:1) and read them through RBC_DATA_RD(25:1).

For more details on memory requirements, see Chapter 4 Local Memory.

9.4 ATPG Vectors

9.4.1 ATPG Overview

Lexra provides ATPG scripts to generate internal scan vectors using Tetramax™ from Synopsys.

To achieve the highest possible fault coverage using the Lexra core, it is recommended that scan insertion and ATPG is done using the provided scripts. Using the provided scripts also provides the user with a modular method of performing DFT. Such a methodology can alleviate problems with EDA tool capacity that can occur if scan insertion and ATPG are done flat at the top level. Also, this approach also simplifies fault isolation in production test.

ATPG can be run at either the `lx1` or `lx2` module boundaries. Most users will prefer to run at the `lx2` module boundary, as that is the natural division between the Lexra core and the customer logic. In some cases, however, the RAM architecture assumed by `lx2` may not match that provided by the RAM vendor, in which case additional customer logic is required inside the RAM wrappers. For example, the only RAM available may be asynchronous (Lexra cores require synchronous RAMs). Lexra's synthesis scripts do not synthesize logic inside the RAM wrappers; therefore, it will not incorporate any such logic into the scan chain. In such cases, it is desirable to perform ATPG at the `lx1` module boundary instead of `lx2`.

Regardless of the chosen module boundary, there are 2 conditions under which ATPG may be run, best case and worst case. Under best case conditions, controllability and observability of all the module I/O's are assumed. Under worst case conditions, minimal controllability and observability is assumed. In either case, controllability and observability is assumed for those signals listed in the table in Section 9.1.4, Internal Scan Interface. If any of the signals listed in Section 9.1.4, Internal Scan Interface are uncontrollable or unobservable, it will not be possible to use Lexra's ATPG scripts, nor will it be possible to generate vectors that result in high fault coverage.

To summarize the four conditions under which ATPG may be run:

- **lx2_best:** Assumes complete controllability of all of the I/O's at the lx2 module boundary. For such an assumption to be true, an external scan collar around all of the module I/O's must be implemented by the customer.

- **lx2_worst:** Assumes controllability and observability only for the module I/O's listed in the table in Section 9.1.4, Internal Scan Interface.

- **lx1_best:** Assumes complete controllability of all of the I/O's at the lx1 module boundary. For such an assumption to be true, an external scan collar around all of the module I/O's must be implemented by the customer. Additionally, use of Lexra's memory scan collar is required. Refer to Section 9.2, Memory Scan Collar for information on the memory scan collar.

- **lx1_worst:** Assumes controllability and observability only for the module I/O's listed in the table in Section 9.1.4, Internal Scan Interface.

Lexra's ATPG scripts do not generate test vectors for the RAMs inside the lx2 module boundary.

9.4.2 ATPG Generation Process

To generate scan test vectors using ATPG, please perform the following steps:

1. Edit your `lconfig` form to configure the Lexra core for scan. You must set SCAN_INSERT option to be YES. Additionally, it is recommended that you set SCAN_SCL to YES.

2. Run regression simulation.

3. Synthesize the core to the `lx2` level (or `lx1` if ATPG is to be done at the `lx1` level). Lexra recommends that ATPG be performed at the `lx2` module boundary.

4. Check the `syn/lx2/chk_logs.log` file for errors or warnings. If none are present, and synthesis results are satisfactory, you can proceed to ATPG generation.

5. Go to the `user/tech` directory and create a file called `lib_tmax`. This file must contain a pointer to the standard cell library to be used by Tetramax. An example of this file will contain the command:

   read netlist /<path_to_library>/<lib_name>.v -noabort -library

6. In the `atpg` directory, you will see 4 directories:

   lx2_best
   lx2_worst
   lx1_best
   lx1_worst

   Choose the directory that matches the conditions under which you plan to run ATPG. For example:

   cd lx2_best

7. The ATPG process is Makefile driven. To create the ATPG vectors, simply type **make** at the command line.

The ATPG process itself is a two-step process. First, TestCompiler™ is run to check the scan design rules and generate the control files for TetraMax™. The following files are produced:

lx2_atpg_scan.order: The scan order file, derived from the Verilog netlist located at `syn/lx2/lx2.hv`.

lx2_atpg.tpf: The test protocol file used by TetraMax.

lx2_atpg.spf: The signal protocol file used by TetraMax.

lx2_atpg.autoxp: The detailed listing of the scan chains.

lx2_scan.scr.log: The log file from TestCompiler.

The output of TestCompiler will show some scan design rule violations. These can be grouped into two categories:

- Non-scanned elements. The JTAG TAP controller is not scanned; therefore, there will be violations being reported. These can be ignored.
- Falling edge flip-flops. There are 2 falling edge flip-flops in the design. One drives the TDO output of the JTAG TAP controller. The second generates the DCLK output when PC_TRACE option in EJTAG is selected. Both can be ignored.

As a rule, you can ignore the reported design rule violations unless your fault coverage after ATPG is unexpectedly low.

Once TestCompiler completes, TetraMax is run to generate the scan vectors. In most cases, the two steps will complete in one **make** step, so it will be invisible to the user. In some cases, the script may abort after running TestCompiler. If this should occur, simply run **make** again to run TetraMax.

The following files are created by TetraMax:

lx2_atpg_pat.v: The scan test vectors, along with a testbed that loads the scan chain with the test patterns in a parallel format.

lx2_atpg_pat_s.v: A serial scan testbed, that simulates the serial shifting in and out of the scan chain. Only 16 scan vectors are simulated serially.

faults.AU, faults.UD: List of untestable and undectectable faults as reported by Tetramax.

lx2_atpg_tmax.log: Log file produced by Tetramax.

Currently, the ATPG script only produces Verilog based test vectors. If different vector formats are desired, you should rerun ATPG:

8. Type **make clean** at the Unix command prompt to delete the old files.

9. Edit the file `lx2_atpg_tmax.scr`, and look for the line:

   exit -force

   Prior to this line, add the following command, which will create TSSI vectors:

   write patterns lx2.wgl -format wgl -internal -replace

   This will create the file **lx2.wgl**, which will create patterns in a TSSI format. Other formats include:

   ftdl: Fujitsu TDL
   stil: IEEE P1450.1
   stil99: IEEE P1450.0
   tdl91: TI TDL91
   testgen: Sunrise
   tstl2: Toshiba TSTL2

10. Rerun the Makefile script:

    make

9.5 Testability Statistics

9.5.1 Overview

The testability results after ATPG will vary based on a number of factors, including:

- Product type (LX4189, LX4280, LX5180, LX5280, LX8000).
- Product options, such as MAC, EJTAG, LBC, or TLB.
- Configuration options, including cache sizes, use of coprocessors or custom engines, reset methodology (synchronous vs. asynchronous).
- Testability options, such as the optional memory scan collar.
- If Lexra does the porting and layout of the design, Lexra may add special enhancements to improve area or performance such as latch-based register files or tri-state muxes.

Due to the wide variety of options available, it is not possible to guarantee a specific level of fault coverage for the design. Most configurations should see at least 90% fault coverage, with the majority exceeding 95% fault coverage.

9.5.2 Example

The following example illustrates the fault coverage results for a particular configuration of the LX4189. Again, results will vary with configuration. The illustrated configuration options from the `lconfig` form are shown below:

```
PRODUCT = "LX4189";
PRODUCT_TYPE = "RTL";
REGFILE_TECH = "FLOP";
TECHNOLOGY = "CUSTOM";
TESTBED_ENV = "CHIP";
RESET_TYPE = "ASYNCHRONOUS";
RESET_DIST = "GLOBAL";
SEN_DIST = "GLOBAL";
SEN_BUFFERS = "EXTERNAL";
SLEEP = "YES";
RESET_BUFFERS = "EXTERNAL";
```

```
CLOCK_BUFFERS = "EXTERNAL";
RAM_CLOCK_BUFFERS = "NO";
COP1 = "NONE";
COP2 = "NONE";
COP3 = "NONE";
CE0 = "CE_HL";
CE1 = "NONE";
M16_SUPPORT = "YES";
MEM_LINE_ORDER = "SEQUENTIAL";
MEM_FIRST_WORD = "DESIRED";
MEM_GRANULARITY = "BYTE";
SYSTEM_INTERFACE = "LBUS";
LBC_WBUF = 4;
LBC_RBUF = "2";
LBC_RDBYPASS = "YES";
LBC_SYNC_MODE = "SYNCHRONOUS";
LINE_SIZE = "4";
ICACHE = "16K_1";
DCACHE = "16K_1";
IMEM = "NONE";
IROM = "NONE";
IMEM_IS_ROM = "NO";
DMEM = "NONE";
LMI_DATA_GRANULARITY = "BYTE";
LMI_RANGE_SOURCE = "HARDWIRED";
LMI_RAM_ARB = "NO";
JTAG = "EXPORT_EXTENDED";
EJTAG = "YES";
EJTAG_INST_BREAK = 2;
EJTAG_DATA_BREAK = 1;
JTAG_TRST_IS_TPC = "YES";
PC_TRACE = "EXPORT";
EJTAG_DCLK_N = "2";
EJTAG_TPC_M = "1";
EJTAG_XV_BITS = "4";
EJTAG_PC_ISABIT = "YES";
SCAN_INSERT = "YES";
SCAN_MIX_CLOCKS = "YES";
SCAN_NUM_CHAINS = "8";
RAM_BIST_MUX = "YES";
```

The results of this configuration are shown in the table below, both with and without the optional memory scan collar around the cache RAMs.

	lx2_worst	lx2_best
No memory scan collar (SCAN_SCL="NO")	95.1%	95.7%
Memory scan collar present (SCAN_SCL="YES")	96.7%	97.4%

In this configuration, the memory scan collar improves fault coverage by close to 2%. It is also apparent that the difference between best and worst case ATPG conditions is only about 0.5% for this configuration.

9.5.3 Interpreting ATPG Results

TetraMax produces a log file, `lx2_atpg_tmax.log`**.** The conclusion of this log file includes a table that shows the testability results. An example log file is shown below (from lx2_worst, SCAN_SCL=YES configuration from the above table):

```
  Collapsed Stuck Fault Summary Report
-----------------------------------------------
fault class                     code   #faults
-----------------------------------------------
Detected                         DT     137044
  detected_by_simulation         DS    (100860)
  detected_by_implication        DI     (36184)
Possibly detected                PT        590
  atpg_untestable-pos_detected   AP       (590)
Undetectable                     UD       2895
  undetectable-unused            UU          (2)
  undetectable-tied              UT      (1389)
  undetectable-blocked           UB       (778)
  undetectable-redundant         UR       (726)
ATPG untestable                  AU       1470
  atpg_untestable-not_detected   AN      (1470)
Not detected                     ND          3
  not-observed                   NO          (3)
-----------------------------------------------
total faults                              142002
test coverage                              98.73%
fault coverage                             96.72%
ATPG effectiveness                        100.00%
-----------------------------------------------
```

Some definition of the terms is in order:

- **Detected** faults are faults that were detected by the ATPG vectors. A detected fault is one that causes a hard 1 vs. 0 failure during pattern checking.
- **Possibly** detected faults are faults that cause an X value to propagate to the output.
- **Not Detected** faults are faults that the ATPG tool failed to cover when generating vectors. This number is normally very low.
- **Undetectable** are faults that do not propagate any hard 1 vs. 0 failures during pattern checking. These faults are typically either not controllable (e.g., an input pin tied to VDD) are not observable (e.g., an unused output of cell).
- **Untestable** faults are faults for which the ATPG tool cannot generate a known good reference model. These faults are commonly caused by non-scannable flip-flops or black-box models such as memories.
- **Fault coverage** is the ratio of detected and possibly detected faults to the total number of faults in the design. This is the most conservative definition of fault coverage.
- **Test coverage** is similar to fault coverage, except that the undetectable faults are removed from consideration. This definition is less conservative than fault coverage.
- **ATPG effectiveness** removes all untestable and undetectable faults from consideration. It is used to indicate how complete the ATPG patterns test the faults that could be detected.

Customers should refer to the TetraMax documentation for detailed descriptions of these fault classes.

9.6 TAP Controller

We have designed the Lexra TAP controller which is required for implementation of the EJTAG option and optional in other configurations, to act as the only TAP controller in a chip. Lexra provides an extended interface that allows the Lexra TAP to control your boundary scan register and implement four of your proprietary instructions. With this interface, it is possible to use one or more of the proprietary instructions to control the internal scan chains inside the Lexra core.

See Chapter 8 EJTAG for the specification of the interface, and relevant `lconfig` options.

9.7 Additional Considerations for Reset and Clock Distribution.

Like the distribution of scan enable, there are several options for the distribution and buffering of clock and reset. While strictly not a testability feature, the distribution of these signals can affect the overall test strategy.

9.7.1 Clock Distribution

The following pins on the `lx2` module boundary are used for clocking the Lexra core:

- **SYSCLK** is the main processor clock. If SLEEP mode is enabled, this clock is gated by the SLEEP mode signal.
- **SYSCLKF** is the free-running version of SYSCLK. This pin exists only if SLEEP mode is enabled. The SYSCLKF domain clocks the wake-up circuitry inside the sleep mode logic.
- **BUSCLK** is the clock for the LBC, and is present only if LBC_SYNC_MODE is ASYNCHRONOUS. This clock is gated by the SLEEP mode signal if SLEEP mode is enabled.
- **BUSCLKF** is the free-running version of BUSCLK. This pin exists only if SLEEP mode is enabled. The BUSCLKF domain clocks the wake-up circuitry inside the sleep mode logic.
- **JTAG_CLOCK** is the clock for the JTAG TAP controller and some of the EJTAG logic. It is used only if JTAG is set to EXPORT or EXPORT_EXTENDED. This clock domain is NOT gated by SLEEP.
- **SL_SLEEPSYS_R** is used to indicate that the core has entered SLEEP mode. This signal is synchronized to SYSCLK, and exists only if SLEEP is set to YES.

- **SL_SLEEPBUS_BR** is used to indicate that the core has entered SLEEP mode. This signal is synchronized to BUSCLK and exists only if LBC_SYNC_MODE is ASYNCHRONOUS and SLEEP is set to YES.

Additionally, the following `lconfig` options control clock buffering

CLOCK_BUFFERS = EXTERNAL | LX2

- Setting this parameter to EXTERNAL causes the clock inputs to propagate directly from the `lx2` module boundary to the individual clocked elements in the design. No buffering is done inside the Lexra core. This is the recommended setting. If this option is set and SLEEP mode is set to YES, please refer to Section 9.7.2, SLEEP and Clock Distribution for details on proper clock distribution.

- Setting this parameter to `LX2` will cause special clock buffers to be instantiated inside the Lexra core. These clock buffer modules are used as non-synthesizable place holders, which must be replaced by the technology specific clock buffers available from your ASIC or COT vendor.

Most users prefer to control clock distribution at the top-level of the design, so setting this parameter to EXTERNAL will work for most technologies. If you set this parameter to `LX2` and SLEEP is not enabled, the following clock buffer cells will be instantiated:

- lx2_sysclkbuf: Buffer for SYSCLK domain.

- lx2_busclkbuf: Clock buffer for BUSCLK domain. Only if LBC_SYNC_MODE is ASYNCHRONOUS.

- lx2_jtagclkbuf: Clock buffer for JTAG TAP controller.

If, however, SLEEP is set to YES, the following clock buffers are instantiated instead:

- lx2_slsysclkbuf: Buffer for SYSCLK domain. Also gates SYSCLK with SLEEP signal.

- lx2_slsysclkfbuf: Buffer for SYSCLKF domain.

- lx2_slbusclkbuf: Clock buffer for BUSCLK domain. Also gates BUSCLK with SLEEP signal. Used only if LBC_SYNC_MODE is ASYNCHRONOUS.

- lx2_slbusclkfbuf: Clock buffer for BUSCLKF domain. Used only if LBC_SYNC_MODE is ASYNCHRONOUS.

- lx2_jtagclkbuf: Clock buffer for JTAG TAP controller.

In addition to the main clock buffers, there is an `lconfig` option for the clock signals on the cache RAMs:

RAM_CLOCK_BUFFERS = YES | NO

- Setting this parameter to NO causes the SYSCLK signal to route directly to the clock pins on the cache and local memory RAMs. This is the recommended setting for most technologies.

- Setting this parameter to YES will cause special clock buffers to be placed inside the lmi modules. These clock buffer modules are used as non-synthesizable place holders, which must be replaced by the technology specific clock buffers available from your ASIC or COT vendor.

Depending upon your configuration, if you set RAM_CLOCK_BUFFERS to be YES, the following clock buffers will appear in the `lmi_icache` module:

- ic_ramclkbuf: Buffer for instruction cache RAM clock. Used only if ICACHE is enabled in `lconfig`.

- iw_ramclkbuf: Buffer for instruction memory RAM clock. Used only if IMEM is enabled in `lconfig`.

- ir_ramclkbuf: Buffer for instruction ROM clock. Used only if IROM is enabled in `lconfig`.

Depending upon your configuration, if you set RAM_CLOCK_BUFFERS to be YES, the following clock buffers will appear in the `lmi_dcache` module:

- dc_ramclkbuf: Buffer for data cache RAM clock. Used only if DCACHE is enabled in `lconfig`.

- dw_ramclkbuf: Buffer for data memory RAM clock. Used only if DMEM is enabled in `lconfig`.

9.7.2 SLEEP and Clock Distribution

If SLEEP mode is enabled, clock distribution requires more attention. The SLEEP logic requires that the SYSCLK and BUSCLK domains be disabled when the core enters SLEEP mode. However, the free-running clocks, SYSCLKF and BUSCLKF, must still continue to toggle so that they can clock the wake-up logic.

If the `lconfig` option CLOCK_BUFFERS is set to LX2, the gating logic for SYCLK and BUSCLK domains occurs inside the 2 clock buffer placeholders in `lx2`, `lx2_slsysclkbuf` and `lx2_slbusclkbuf`. The design of these clock buffers and the gating logic is the responsibility of the user.

If the `lconfig` option CLOCK_BUFFERS is set to EXTERNAL, the SLEEP mode signals, SL_SLEEPSYS_R and SL_SLEEPBUS_BR will be exported out of `lx2`. The user must then use these 2 signals to gate their incoming clocks. The gated clock signals are then connected to SYSCLK and the BUSCLK inputs.

Under no circumstances should the free running clocks, SYSCLKF and BUSCLKF, be gated with the SLEEP mode signal, or a deadlock condition will result. These free running clocks are used to clock the wakeup logic inside the Lexra core.

9.7.3 Reset Distribution

Reset of the Lexra core is complex. There are several domains, pins, and `lconfig` options controlling reset and its distribution and buffering.

There are as many as four reset domains in the Lexra core: logic clocked by SYSCLK, logic clocked by BUSCLK, the EJTAG logic clocked by the TAP clock, and the TAP controller itself.

The TAP and EJTAG domains are similar. When the TAP controller is reset, it immediately enters the TEST_LOGIC_RESET state. The EJTAG logic is reset one cycle after the TAP controller has entered its TEST_LOGIC_RESET state.

The following signal pins on `lx2` boundary control the reset of the core:

- **CResetN:** Cold Reset, also called PowerOn reset. This is the primary reset of the core. When asserted, all 4 reset domains are reset. When the core comes out of reset, it will fetch instructions from the reset vector, 0xbfc0_0000.

- **ResetN:** Warm Reset. When asserted, only the SYSCLK and BUSCLK domains are reset. This reset pin allows an EJTAG probe to reset the core without resetting all of the EJTAG logic, which is useful in software debug. Also, if an EJTAG probe is present, the core will fetch instructions from the EJTAG probe when it comes out of reset. There is also a software warm reset, which is accessible only through the EJTAG probe.

- **JTAG_TRST_N:** The JTAG TAP reset. The TAP controller can also be reset through the assertion of JTAG_TMS high for 5 clocks, so the use of this pin is optional. Finally, there is a software reset of the TAP controller, which is accessible only through the EJTAG probe.

There are three `lconfig` options that control reset:

RESET_TYPE = ASYNCHRONOUS | SYNCHRONOUS

- This parameter controls the type of reset flip-flop that will be synthesized. Some ASIC/COT libraries use asynchronous resettable flip-flops, others use synchronous resettable flops, and others use both. This setting is normally determined by customer design methodology and library vendor. With either setting, the reset will be disabled during scan shifting. When set to ASYNCHRONOUS, the reset will be disabled by TMODE, which is typically asserted in both scan shift and capture.

RESET_DIST = GLOBAL | LOCAL_BUFFERED | LOCAL_SAMPLED

- When set to GLOBAL, a global reset signal is routed to the reset pins of all the flip-flops in all reset domains. No logic or buffering is placed on the reset signal. When set to GLOBAL, the incoming reset signal will NOT be synchronized to any clock domain. It is recommended that the customer synchronize the de-assertion of reset to SYSCLK if this parameter is set to GLOBAL and RESET_TYPE is set to ASYNCHRONOUS. Also, the use of an asynchronous LBC (LBC_SYNC_MODE set to ASYNCHRONOUS) is not recommended when this parameter is set to GLOBAL.

- When set to LOCAL_BUFFERED, the incoming reset signal will be re-synchronized to the appropriate clock domain using a 2-stage synchronizer, and then distributed throughout the rest of the core. During synthesis, buffers will be placed on the reset signals as they enter each module boundary.

- When set to LOCAL_SAMPLED, the incoming reset signal will be re-synchronized to the appropriate clock domain using a 2-stage synchronizer, and then distributed throughout the rest of the core. The reset signal will then be re-clocked at each module boundary. This setting does simplify the timing analysis of reset, and is recommended for high-performance designs. The only disadvantage to this setting is an area penalty, and an increase in latency during the reset sequence itself.

The reset domains inside the Lexra core are composed of logical functions of the 3 main reset pins, plus various software resets controlled by EJTAG. The `reset_dist` module generates the actual reset signals based on the states of the reset pins and software reset values. **It is these generated resets, not the 3 reset pins listed above, that propagate throughout the registers of the design.** Therefore, these generated resets must be buffered. The buffering of these reset signals is controlled through the use of the RESET_BUFFERS option in `lconfig`:

RESET_BUFFERS = EXTERNAL | LX2

When CLOCK_BUFFERS is set to `lx2`, special clock buffer placeholder cells are instantiated inside the `lx2` module boundary. These placeholders must be replaced by either a high drive clock buffer or buffer tree prior to synthesis of `lx2`. The reset buffers inside `lx2` are as follows:

- lx2_rstd1rbuf: This buffer distributes reset to the flip-flops clocked by SYSCLK. Logical combination of cold and warm reset (CResetN and ResetN). Also triggered by EJTAG software reset.

- lx2_rstd1brbuf: This buffer distributes the reset to the flip-flops clocked by BUSCLK. Logical combination of cold and warm reset (CResetN and ResetN). Also triggered by EJTAG software reset.

- lx2_rstjtagbuf: This buffer distributes the reset to selected registers inside EJTAG. Triggered only when TAP controller enters TEST_LOGIC_RESET state.

- lx2_rsttapbuf: This buffer distributes the reset domain to the TAP controller. Triggered by the 3 conditions for TAP reset: TRST_N assertion, TMS=1 for 5 clocks, or TAP software reset.

Additionally, if the RESET_DIST option is NOT set to GLOBAL, the following buffers will also appear:

- lx2_rstpwrbuf: A buffered version of the CResetN, not synchronized to any clock. This domain is used to reset asynchronous handshaking logic inside the EJTAG module and to reset the TAP.

- lx2_rstpwrd1buf: A buffered version of CResetN synchronized to BUSCLK. Not used at this time.

If RESET_BUFFERS is set to EXTERNAL instead, the reset signals are assumed to be buffered externally. Since the reset signals are generated deep inside the Lexra core, it becomes necessary to output the generated reset signals from `lx2`, and to create input pins for the buffered signals. Therefore, the following signals will appear at the `lx2` module boundary:

- **RESET_D1_R_N_O:** The generated version of the primary core reset signal. A logical combination of CResetN, ResetN, and the EJTAG software reset, synchronized to SYSCLK. An output of `lx2`. (Note: signal is not synchronized to SYSCLK if RESET_DIST=GLOBAL).

- **RESET_D1_R_N:** The buffered version of RESET_D1_R_N_O. Resets core flip-flops clocked by SYSCLK. An input to lx2.

- **RESET_D1_BR_N_O:** The generated version of the primary core reset signal. A logical combination of CResetN, ResetN, and the EJTAG software reset, synchronized to BUSCLK. An output of `lx2`. (Note: signal is not synchronized to BUSCLK if RESET_DIST=GLOBAL).

- **RESET_D1_BR_N:** The buffered version of RESET_D1_BR_N_O. Resets core flip-flops clocked by BUSCLK. An input to `lx2`.

- **RESET_PWRON_C1_N_O:** The generated version of the PowerOn reset signal, used to reset asynchronous logic inside EJTAG and the TAP controller. A logical equivalent of CResetN, synchronized to SYSCLK. An output of `lx2`. (Note: signal is not present if RESET_DIST=GLOBAL).

- **RESET_PWRON_C1_N:** The buffered version of RESET_PWRON_C1_N_O. Not preset if RESET_DIST=GLOBAL. An input to `lx2`.

- **RESET_PWRON_D1_LR_N_O:** The generated version of the PowerOn reset signal, used to reset asynchronous logic inside EJTAG and the TAP controller. A logical equivalent of CResetN, synchronized to BUSCLK. An output of `lx2`. (Note: signal is not present if RESET_DIST=GLOBAL).

- **RESET_PWRON_D1_LR_N:** The buffered version of RESET_PWRON_D1_LR_N_O. Not present if RESET_DIST=GLOBAL. An input to `lx2`.

- **JTAG_RESET_O:** Generated version of EJTAG reset signal. Logically asserted one cycle after the TAP controller enters TEST_LOGIC_RESET state. Used to reset registers in EJTAG clocked by JTAG_CLOCK. An output of `lx2`.

- **JTAG_RESET:** Buffered version of JTAG_RESET. An input to `lx2`.

- **TAP_RESET_N_O:** Generated version of TAP reset. Asserted whenever JTAG_TRST_N or CResetN is asserted, when TMS is asserted high for 5 clock cycles, or when the TAP software reset is enabled by the EJTAG probe. An output `lx2`.

- **TAP_RESET_N:** Buffered version of TAP_RESET_N_O. An input to `lx2`.

It is the responsibility of the user to ensure proper buffering exists between the generated and buffered versions of the reset signals listed above. Also, Lexra's ATPG scripts assume that the input signals listed above are completely controllable. Failure to provide controllability of these buffered reset inputs will seriously degrade testability.

Chapter
10

Using the Rundvt Regression Environment

Lexra RTL releases include `rundvt` and the regression test suite that Lexra uses to validate the processor configuration. This test suite is used to verify the customized RTL against a specific user configuration of the Lexra processor. `Rundvt` and the Lexra testbed are not designed to be the primary means of validating the entire ASIC application, nor does it easily integrate peripherals and bus agents, develop software drivers, or do other ASIC development work. Modifications are permitted; however, Lexra does not support the extended `rundvt` environment for the entire ASIC development.

`Rundvt` is a software utility automating the simulation of self checking assembly and C tests within a Verilog simulation environment. It automates the simulation of the regression suite using test lists. Each test list is a simple Perl script. This script calls `rundvt` subroutines that associate configuration information with a specific test name. Section 10.4, Working with Test Lists gives more details.

`Rundvt` allows commands to be given to the simulator. Many command line options can override `rundvt`'s default behavior. See Section 10.3.1, Standard Command-Line Options for the most common and useful options, and Section 10.3.2, Advanced Options for more advanced features. Thus, simulating the configured core in the provided testbed with `rundvt` requires minimal changes.

`Rundvt` is ordinarily used to verify the `lconfig` options within the Lexra testbed after a configuration has been chosen. Additional tests may be written to check specific features in the Lexra core. ASCII debug tracing and gate level simulation are supported. These are described in Section 10.6, Generating ASCII Traces in the Simulation Output and Section 12.6, Gate Level Simulation. Overall, `rundvt` provides more than sufficient resources to run regression tests on the Lexra core.

10.1 Rundvt Simulators

`Rundvt` currently offers support for the following Verilog simulators:

- VCS
- Verilog-XL
- NC-Verilog

Lexra may add support for other popular Verilog simulators in upcoming releases.

10.2 Setup

Refer to Section , Chapter 1Section , Lexra Development Environment for `rundvt` and Perl installation setups.

10.3 Using the Command-line Options

The `rundvt` script is a generic front end for various Verilog simulators. `Rundvt` automates regression testing by providing a complete interface to the testbed Verilog module. The `rundvt` command line options include many of the testbed's features.

`Rundvt` passes command line options it doesn't recognize to the Verilog simulator in the same order that it sees them. This allows Verilog command line options, top level Verilog modules, or gate level netlists to be passed to the simulator.

The `Getopt::Long` Perl package and `rundvt`'s `&process_verilog_args` subroutine parse the `rundvt` commands. For more information on the operation of the PERL package, see the documentation in the LSDK as specified below.

$LSDKDIR/perl/html/lib/Getopt/Long.html

The `rundvt` script contains a summary of the options it supports. Lexra often enhances the script between product revisions, so there may be some differences. It is recommend that new features are checked by invoking `rundvt` with the `-help` option.

rundvt -help

`Rundvt` is insensitive to the order of options, tests, or Verilog command line arguments with the exception of `-help`. This option must be the first argument after `rundvt`.

rundvt [-help] [options] <test/test list> [Verilog commands]...

10.3.1 Standard Command-Line Options

This section describes the most common command line options. They are useful when using `rundvt` as a means of performing basic validation of the current RTL configuration, `rundvt's` most common use.

`Rundvt` supports the following often-used command line options.

OPTION	DESCRIPTION
-help	Displays usage
<test/test list>	Runs the specified test or test list
<Verilog commands>	Commands, options, or arguments external to *rundvt*

rundvt <test/testlist>

One or more tests, or test lists (files automating the simulation of multiple tests) may be specified. Test source code should be placed in the $LX_HOME/`tests` directory. Assembly test files should have a "`.s`" extension and C programs should have a "`.c`" extension. Test lists should be placed in the $LX_HOME/`regression` directory and be given a "`.pl`" extension. See Section 10.3.3, Passing Tests to Rundvt Through the Command Line for more information on how `rundvt` knows which command line arguments are tests. Examples of different test arguments:

rundvt regression.pl
rundvt -from_regression Logical.s Add.s
rundvt -from_regression hello.c

rundvt <Verilog commands> <test/testlist>

`Rundvt` passes any unknown commands to the Verilog simulator. By default, `rundvt` uses Synopsys VCS for Verilog simulation.

Examples of Verilog commands are relative paths, plus-args, and defines to Verilog files. In the example below, `RTL_MON` is a Verilog macro definition and `+trace_all` is a plus-arg.

rundvt regression.pl +define+RTL_MON +trace_all

`Rundvt` passes several compiler options such as

-l vcompile.log -Mupdate -V +cli +incdir+..

to the VCS compiler by default. The only way to change these arguments is to edit `rundvt`. For more information on the cli and Verilog acc routines, see the VCS man pages. **`-Mupdate`** causes VCS to create a `csrc` subdirectory containing Makefile and C objects related to the Verilog compilation for use in incremental compile.

rundvt -sim <vcs, ncv, vxl > <test/testlist>

VCS, NC-Verilog, or Verilog-XL simulators can be specified with `rundvt` from the command line. `Rundvt`'s default verilog simulator is VCS. When invoking VCS with **rundvt -sim <vcs>** is not required.

OPTION	DESCRIPTION
vcs	Synopsys VCS
ncv	NC-Verilog
vxl	Verilog-XL

Note: Only certain Verilog simulators permit plus-args.

The default Verilog simulator arguments assumed by `rundvt` might not be correct for your environment. If the simulator is not listed above and the default Verilog simulator arguments are not correct for the simulator, then `rundvt` must be edited to support it.

See regression/*.inpfiles for more information about the files used in Verilog compilation (specifically: $LX_HOME/regression/*.inpfiles).

10.3.2 Advanced Options

Lexra developers are the principal users of advanced command line options as described in this section. These options are useful in tracing or debugging a suspected problem in the RTL.

Lexra reserves the right to add or remove options. For the latest documentation on available command line options, please use the on-line help **rundvt -help**.

rundvt -nomake <test/testlist>

Stops `rundvt` from compiling test programs. If this option is present, `rundvt` will not call the `$LX_HOME/tests/Makefile` to compile the test programs into a `<test>.bin` (ASCII binary file) format. `Rundvt` will expect a `<test>.bin` file to be available in the `$LX_HOME/tests/obj` directory when this option is used.

rundvt -norun <test/testlist>

No tests are performed. This option lists the tests that `rundvt` would run. It shows the individual test file name in a testlist file.

rundvt -tools <string> <test/testlist>

The test or testlist is compiled and simulated in Lexra's testbed for a specific software tool chain other than the specified default. The specified default for software compiler and execution target is defined in `$LX_HOME/tests/tools.mk` file which is generated by `lconfig`. Software tool chains supported are shown below.

Software Tool-chain Options

STRING	DESCRIPTION
lsdk	Lexra Software Developer's Kit
lsdk-mips16	Lexra Software Developer's Kit MIPS16
ghs	Green Hills
ghs-mips16	Green Hills MIPS16

rundvt -sim <asym, pass2> <test/testlist>

In addition to vcs, ncv, and vxl, the `-sim` option also compiles and simulates tests in Lexra's testbed for the Instruction Set Simulators.

OPTION	DESCRIPTION
asym	Instruction Set Simulation Debugger
pass2	Cycle Accurate Instruction Set Simulator

rundvt -simopts <string> <test/testlist>

The `-simopts` is used in conjunction with **-**`sim` for non-Verilog simulators. The `-simopts <string>` option passes non-Verilog simulator options to the simulator. The example below instructs the pass2 simulator to display the log to STDOUT instead of a file.

rundvt -sim=pass2 -simopts='-v 2'

Specific options for the non-verilog simulators can be displayed using:

rundvt -help -sim=<non-verilog simulator>
perl simif.pl -help
perl simif.pl -simif"=<non-verilog simulator> -help"

rundvt -top <file> <test/testlist>

Overrides the default top level Verilog modules and passes one Verilog file from the command line to the Verilog simulator. `Rundvt` uses one pair of Verilog top level modules, `$LX_HOME/testbed/testbed.v` and `$LX_HOME/chip/topchip.v`. Set the `lconfig` option TESTBED_ENV to "CHIP" in `$LX_HOME/user/lx----.form` file. For customized top level simulation, specify the top verilog module name using the `-top` option as in the example below.

rundvt -top your_top_file.v...

rundvt -notop <file> <test/testlist>

Stops `rundvt` from adding the default top level Verilog modules to the Verilog simulator's command line and allows for a different top level module.

rundvt -notop top1.v top2.v topN.v Logical.s

rundvt -sim_only <test/testlist>

When using VCS, the default Verilog simulator, this option stops `rundvt` from calling VCS Verilog compiler. This feature assumes that the VCS **simv** executable already exists in the `regression` directory and runs the simulations using this executable.

rundvt -gen_only <test/testlist>

When using VCS, the default Verilog simulator, this option stops `rundvt` from calling **`simv`**, the Verilog simulator executable. It will only compile. This feature calls the VCS simulator without the **`-R`** option that causes **simv** to execute immediately following a successful compilation.

rundvt -quiet <test/testlist>

Stops `rundvt` from printing verbose messages to standard output.

rundvt -notty <test/testlist>

Stops `rundvt` from printing of nearly all messages to standard output.

rundvt -continue <test/testlist>

`Rundvt` calls the verilog simulator many times with different command line options, depending on the nature of the tests in test files or test lists. If `rundvt` encounters an error within a verilog simulation with specified verilog command line options, it will finish that simulation with error messages. Subsequent simulation calls within the rundvt test list will not occur. `Rundvt` exits regression testing with an error message similar to the one shown below. Using `-continue` option allows `rundvt` to continue and finish regression testing of all tests.

```
V C S  S i m u l a t i o n  R e p o r t
Time: 87026000 ps
processor Time:   1.110 seconds; Data structure size: 18.4Mb
Fri Apr 13 11:49:24 2001
INFO: rundvt Results of simulation
INFO: rundvt   PASS: 1; FAIL: 0
INFO: rundvt Execution Complete
ERROR: rundvt    Expected 2 total passes but observed 1
ERROR: rundvt   Total PASS 1
ERROR: rundvt   Total FAIL 0
```

rundvt -batch <test/testlist>

When using VCS 6.0, the `-batch` command configures the Verilog compilation flags for faster simulation by turning off CLI.

rundvt -gates ../syn/xxx/xxx.hv <test/testlist>

Allows Verilog to compile and run gate level simulation. All hierarchical references which are made by the Verilog testbed are disabled. This is done automatically with `-gates` option which invokes `-nopeeking`. **`Rundvt -gates`** adds a file called `$LX_HOME/user/custom/tech/gate.f` which includes the ASIC cell libraries. See Section 12.6, Gate Level Simulation.

rundvt -nopeeking <test/testlist>

This option is also invoked as part of `-gates`. Hierarchical references are disabled by removing `+define+LOCAL_MON` in the verilog testbed. See Section 10.6.1, Tracing Through Hierarchical References.

rundvt -seed <#> <test/testlist>

Sets the random seed number with either the number from the command line or the default time or'ed with process ID. `Rundvt` passes this number to the Verilog testbed, where it becomes the initial seed for a 16-bit random number generator. The Verilog testbed uses this random number generator during internal Lexra testing to randomize the behavior of an ICACHE pre-loader and to define random values for exported `lconfig` options.

rundvt -nops <#> <test>

Causes the test object to be loaded into the memory model with 0 to 3 words of offset from the `.text` segment. Default: 0.

Some assembly tests can be arbitrarily relocated without the LSDK tool chain re-linking them. The `-nops` feature allows internal Lexra testing of assembly tests. The Verilog testbed loads the test at four different word offsets, causing cache line crossings between every instruction and the next.

The tool chain must re-link all C programs and assembly programs that use the JAL, JALR, or J instruction, or the LA pseudo instruction, as well as any tests that are not relocatable, to relocate them. Therefore, for these types of tests, only use zero nops to avoid recompilation.

rundvt -exact_nops <#> <test/testlist>

Sets the number of nops in front of the test object. The number of nops can be specified from 0 to 3. The test only runs once, offset in memory by <#> nops.

rundvt -pmon <test>

Boots PMON and runs tests as PMON client applications. The Verilog testbed loads the PMON object into system memory at virtual address 0xBFC0_0000; overwriting the default initialization code (`Trap_BEV1.s` and `Reset_pgm.s`).

rundvt -pmonpath <path> <test>

Allows to load pmon from a different directory location other than the default. Default: `$LSDKDIR/pmon/build/LX4x80/upmon.bin`

rundvt -load <file>=<hex address> <test>

Loads an ASCII binary file into the memory model at the virtual address. The testbed loads this file into the memory model in the Verilog testbed using the `$readmemb` Verilog system call.

`Rundvt` compiles the assembly program using the `$LX_HOME/tests/Makefile`. It puts the resulting ASCII binary file into `$LX_HOME/tests/obj`. In the example below, a file named `myfile.s` exists in the `tests` directory and has been compiled.

rundvt -load myfile.bin=0x00500000 ...

rundvt -set <hex address>=<hex data> <test/testlist>

Sets a word of data at the virtual address specified in the Verilog testbed memory model.

rundvt -script <ASCII binary file> <test/testlist>

Loads a PMON script file into system memory at virtual address 0xA040_0000. This feature can be used two ways:

- If the PMON script has been compiled into an ASCII binary, provide the full path of the compiled ASCII binary script file.

The `$LX_HOME/tests/Makefile` observes that the ASCII binary script file already exists and does not compile the file. The makefile appends 32 bits of zeros to the compiled ASCII binary script file. They serve as a string terminator.

```
img2bin ~/myscript.scr ~/myscript.bin
echo 00000000000000000000000000000000 >> ~/myscript.bin
rundvt -pmon -script ~/myscript.bin ...
```

- Put the PMON script in the $LX_HOME/tests/pmon.usr directory and let `rundvt` convert it.

`Rundvt` converts the PMON script file into an ASCII binary. `Rundvt` calls `$LX_HOME/tests/Makefile` from the `$LX_HOME/tests/obj` directory. Therefore, make the path to the file a path relative to the `$LX_HOME/tests/obj` directory.

The PMON script file must be placed in the `$LX_HOME/tests/pmon.usr` directory, for example `$LX_HOME/tests/pmon.usr/BPscript.scr`. `Rundvt` expects the target to be the compiled PMON script file instead of the source.

rundvt -pmon -script ../pmon.usr/BPscript.bin ...

rundvt -trap_bev1 <file> <test/testlist>

Specifies a file for the testbed to load at 0xBFC0_0100 for the BEV1 trap handler other than the default trap bev1 file. Default: `$LX_HOME/tests/system/Trap_BEV1.s`.

rundvt -trap_bev0 <file> <test/testlist>

Specifies a file for the testbed to load at 0x8000_0000 for the BEV0 trap handler other than the default trap bev0 file. Default: `$LX_HOME/tests/system/Trap_BEV0.s`.

rundvt -reset <file> <test/testlist>

Specifies a file for the testbed to load at 0xBFC0_0000, the boot vector other that the default reset program. Default: `$LX_HOME/tests/system/Reset_pgm.s`.

rundvt -watch <hex physical address> -watch_stop <test> +define+RTL_MON

The `-watch` option causes the `$LX_HOME/testbed/lmon.v` module to watch for a **physical** instruction address on the internal processor bus LBUS or CBUS. When the module sees the address, it either enables ASCII traces if the `rundvt -watch_trace 0xFF` is present or calls `$stop` if the `rundvt` option `-watch_stop` is present, and prints a message.

The `-watch_stop` option is used in conjunction with `-watch <hex address>`. When invoked and when the address in simulation matches <hex address> then the testbed will call the `$stop` Verilog system call. Default is no `$stop`.

These options requires `+define+RTLMON` option.

rundvt -watch 0x40400038 -watch_stop <test> +define+RTL_MON

```
Database tag: ZL  348
Processor id: 0000c401
INST ADDR WATCHPOINT 0x40400038 detected -- enabling trace
$stop at time 19357500 Scope: topsys.lx_base.lx2.LMONW.LMON.inst_monitor File: ../testbed/
lmon.v Line: 1500
cli_0 >
```

rundvt -watch_mask <hex> <test> +define+RTL_MON

The default mask 0xFFFFFFFF is logically anded with the current address. This allow the testbed to watch only for the <hex address>. Clearing any mask bits allows the testbed to watch for a range of addresses. For example, the address range watch for mask 0xFFFF0000 is from 0xFFFF0000 to 0xFFFFFFFF.

rundvt -watch 0x40400038 -watch_mask 0xFFFFFFF0 -watch_stop <test> +define+RTL_MON

```
INST ADDR WATCHPOINT 0x40400030 detected -- enabling trace
$stop at time 14457500 Scope: topsys.lx_base.lx2.LMONW.LMON.inst_monitor File: \../testbed
/lmon.v Line: 1500
cli_0 > .
INST ADDR WATCHPOINT 0x40400034 detected -- enabling trace
$stop at time 14467500 Scope: topsys.lx_base.lx2.LMONW.LMON.inst_monitor File: \../testbed
/lmon.v Line: 1500
cli_1 > .
INST ADDR WATCHPOINT 0x40400030 detected -- enabling trace
$stop at time 19337500 Scope: topsys.lx_base.lx2.LMONW.LMON.inst_monitor File: \../testbed
/lmon.v Line: 1500
```

rundvt -watch_trace <hex> <test> +define+RTL_MON

The **-**`watch_trace <hex>` is used in conjunction with `-watch <hex address>`. When the Verilog simulator reaches a watch point, LMON overwrites the MonPath MONTrace Verilog variable which enables the ASCII trace. 0xFF is equivalent to `+trace_all`. For more information on ASCII traces see Section 10.6, Generating ASCII Traces in the Simulation Output.

rundvt -watch 0x40400038 -watch_trace 0xff <test> +define+RTL_MON

```
Database tag: ZL  348
Processor id: 0000c401
INST ADDR WATCHPOINT 0x40400038 detected -- enabling trace
19362500 M000>>>  DATA 0x404087b4 wr  c 1111 mem  hit 0x00000000
19367500 M000>>>  INST 0x4040003c rd  c 1111 mem  hit 0x0043082a
19372500 M000>>>  DATA 0x404087b8 wr  c 1111 mem  hit 0x00000000
19372500 M000>>>   SYS 0x404087b0 wr  c 1111 mem    - 0x00000000
```

rundvt -Ttext <hex> <test/testlist>

Set the ".text" segment code location. Make the hex address a **logical** address. Default: 0x0040_0000.

rundvt -drambase <hex address> <test> +define+RTL_MON

Overrides the DMEM settings `lconfig` normally sets. `Rundvt` instructs the testbed to use hierarchical assignments to set the DMEM base address which must be a physical address.

rundvt -drambase 0x4040800 <test> +define+RTL_MON

rundvt -irambase <hex address> <test/testlist>

Sets the IMEM base address. The hex address must be a physical address.

rundvt -load_imem <test> +define+RTL_MON

Instruction data is directly loaded into the local instruction memory from main memory at simulation start. The test program does not have to wait for the simulator to load instruction data from main memory to IMEM through the system bus. This option works only with RTL models in Lexra's testbench.

rundvt -from_regression <test>

This option ensures that the test being simulated is from the regression.pl file which contains the list of valid tests.

rundvt -from_regression LdSt_stress.s

rundvt -from <test> <testlist>

Allows a subset of tests in the testlist to be simulated. From a testlist, the simulator will start from the <test> file as specified. The last test can be specified with `-to <test>`. If the last test is not specified, then the simulator will run tests until the last test in the testlist. In the rundvt.log file, list of tests that are skipped is displayed.

rundvt -from LdSt_stress.s regression.pl

rundvt.log
%I-rundvt Skipping LoadStore [219] because it is before -from starting point

rundvt -to <test> <testlist>

Allows a subset of tests in the testlist to be simulated. From a <testlist>, the simulator will start from the beginning of the test list and end when it reaches the end of <test> as specified with `-to`. The first test can be specified with `-from <test>`. In the `rundvt`.log file, list of tests that are skipped is displayed.

rundvt -to LdSt_stress5d.s regression.pl

rundvt.log
%I-rundvt Skipping LdSt_segments [231] because it is after -to ending point

rundvt -from <test> -to <test> <testlist>

Combination of both `-from` and `-to` may be used to run a subset of the test list.

rundvt -from LdSt_stress.s -to LdSt_stress5d.s regression.pl

rundvt -from <#> <testlist>

Allows a subset of tests in the testlist to be simulated. From a <testlist>, the simulator will start from the # as specified. The last test can be specified with `-to <#>`. If the last # is not specified, then the simulator will run tests until the last test in the testlist. Before using this option, the testlist must have been simulated at least once. The # for the corresponding test from the testlist is found in the rundvt.log.

rundvt.log
```
%I-rundvt  run_test   (LdSt_stress.s     [ 221] 0x00400000 NOPS3)
%I-rundvt  run_test   (LdSt_stress2.s    [ 222] 0x00400000 NOPS3)
%I-rundvt  run_test   (LdSt_stress3.s    [ 223] 0x00400000 NOPS3)
%I-rundvt  run_test   (LdSt_stress4.s    [ 224] 0x00400000 NOPS3)
%I-rundvt  run_test   (LdSt_stress5a.s [ 225] 0x00400000 NOPS3)
%I-rundvt  run_test   (LdSt_stress5b.s [ 226] 0x00400000 NOPS3)
%I-rundvt  run_test   (LdSt_stress5c.s [ 227] 0x00400000 NOPS3)
%I-rundvt  run_test   (LdSt_stress5d.s [ 228] 0x00400000 NOPS3)
```

rundvt -to <#> <testlist>

Allows a subset of tests in the testlist to be simulated. From a <testlist>, the simulator will start from the beginning of the test list and end when it reaches the end of # as specified with `-to`. The first test can be specified with `-from <#>`. Before using this option, the testlist must have been simulated at least once. The # for the corresponding test from the testlist is found in the rundvt.log.

rundvt -from 221 -to 228 regression.pl

10.3.3 Passing Tests to Rundvt Through the Command Line

It is sometimes necessary to use tests outside of Lexra's regression suite. Doing so is an advanced option. If necessary, writing and adding new tests to the standard regression suite is allowed.

In examining its own arguments, `rundvt` must be able to differentiate among tests, test lists, `rundvt` options, and similar command-line options. It compares its arguments against the supported options, then compares all other arguments to one of assembly tests, C programs, or lists of tests. Finally, it passes all unrecognized arguments to the target simulator as command line options.

For example,

rundvt regression -seed 8228 +define+RTL_MON +trace_all ...

`Rundvt` first strips away the options it knows about:

-seed 8228

`Rundvt` recognizes the type of file by its extension: "`.s`" (assembly program), "`.c`" (C program), or "`.pl`" (`rundvt` test list). If the file exists, `rundvt` handles it appropriately.

`Rundvt` checks all the options against the files within the regression and the tests directories. If a corresponding file does not exist, `rundvt` passes the option to the Verilog simulator.

For example,

rundvt regression

It tries the three extensions, ".pl" ".s", ".c" and finds `$LX_HOME/regression/regression.pl`.

`Regression.pl` is a test list, namely the comprehensive list of all the tests under the `$LX_HOME/regression` directory.

For example,

rundvt +define+RTL_MON ...

It tries the three extensions, ".pl", ".s", " .c" and fails to find

$LX_HOME/regression/+define+RTL_MON.pl
$LX_HOME/tests/+define+RTL_MON.c
$LX_HOME/tests/+define+RTL_MON.s

Then `rundvt` passes `+define+RTL+MON` directly to the Verilog simulator as a command-line option.

For example,

rundvt +trace_all ...

It tries the three extension and fails to find

$LX_HOME/regression/+trace_all.pl
$LX_HOME/tests/+trace_all.c
$LX_HOME/tests/+trace_all.s

Then `rundvt` passes `+trace_all` directly to the Verilog simulator as a command line option.

If `rundvt` encounters a test list like regression.pl, it executes the file as a PERL program, using the `require` directive.

If `rundvt` identifies an option as a stand-alone test like "`Logical.s`", `rundvt` tries to call `make` to compile the assembly and C programs.

cd ../tests/obj; /opt/lsdk/bin/make -f ../Makefile Logical.bin

`Rundvt` can detect ASCII binary files at the `rundvt` command line. The "`.bin`" file path should be specified so that `rundvt` can load the file into the Verilog simulator.

rundvt /home/your_work/foo.bin
rundvt ~/your_work/foo.bin

Lexra does not recommend using a relative path to an ASCII binary file on `rundvt` command line. `Rundvt` calls the `$LX_HOME/tests/Makefile` from within the `$LX_HOME/tests/obj` directory, referencing the ../ directory where tests are located. For example,

rundvt ../myfiles/foo.bin

The make program reports:

make: Nothing to be done for `../../myfiles/foo.bin'.

If the binary file does not exist relative to the `$LX_HOME/tests/obj` directory, `rundvt` returns the following error:

cd ../tests/obj; /opt/lsdk/bin/make -f ../Makefile foo.bin
make: * No rule to make target `foo.bin'. Stop.**
%E-Testloader Call make compile has failed >512<

The `-nomake` option stops the call to make.

rundvt -nomake foo.bin

10.4 Working with Test Lists

`Rundvt` provides a single interface for running test lists and/or tests at the `rundvt` command line. Before running tests within a test list, `rundvt` will determine if the test matches the configuration restrictions specified in the test list against the current configuration contained within `$LX_HOME/include/lxr_symbols.vh`.

When tests are added to the `rundvt` command line, they will always be

executed regardless of the configuration. This presents the opportunity for regression tests to fail falsely when they do not match the current configuration. Refer to the `$LX_HOME/regression/regression.pl` test list to learn the configuration dependencies of regression tests to prevent false failures. See Section 10.4.2, Running Tests at the Rundvt Command Line for further information.

10.4.1 Test List File Format

A test list file contains PERL code invoking the `&run_test` subroutine that the `rundvt` script defines. A test-list file can only contain legal PERL code. If there are syntax errors in a test list file, `rundvt` fails with syntax errors.

%l-rundvt Found test-list <test list> <testlist.pl>
Semicolon seems to be missing at <test list> line <#>.
syntax error at <test list> line <#>, near "run_test"

You can put any legal PERL code in a test list. The most common command is the `&run_test` subroutine.

run_test (test-name, # nops, lconfig, Verilog args, test-base, ...);

The first four arguments must be used. The test-base is optional as well as any newer arguments in the `run_test` list.

test-name

The name of the test with the source file extension ".s" or ".c". The makefile compiles tests into the `$LX_HOME/tests/obj` directory. It may mangle the binary file names to let one test produce multiple binaries.

nops

The nops argument to `&run_test` sets the number of times to loop through the test with insertion of 0 to 3 nops at the beginning of the code. This causes the test to start at different memory offsets. Another PERL subroutine, `&run_test_raw`, lets the test run only once, with the specified number of nops. This is similar to
- exact_nops <#>.

lconfig

If the test depends on one specific configuration, a simple expression can be composed to prevent the test from running in an unsupported configuration. The operator precedence from highest to lowest is: ==, !=, &, and |. Using these operators require the arguments to be enclosed between double quotes.

run_test("hello.c",
0,"$NOTNVX&(CE0==CE_MAC|CE0==CE_MACD...)", "");

OPTION	DESCRIPTION
"CEx == CE_DVT"	Apply this test if ce is ce_dvt
"CEx == CE_DVT|CE == CE_MAC"	Apply this test if either ce_dvt or ce_mac
"CEx == CE_DVT&CE != CE_MAC"	Apply this test if ce_dvt and not ce_mac

x = 0 or 1

Verilog options

Each test can be set to run in its required specific configurations. The Verilog options field contains a string with any number of Verilog command line options. Each unique string may cause additional Verilog compilations.

+define+CLD_BusClkRatio3to1 +trace_all

test-base (optional)

This argument specifies the ".text" segment that determines where the program is loaded in system memory. Leaving it blank indicates the test does not require re-linking to be relocated. If it requires linking, then place RECOMPILE in this argument. C programs must always be recompiled.

10.4.2 Running Tests at the `Rundvt` Command Line

Lexra bundles the `$LX_HOME/regression/regression.pl` test list with the regression suite to automate the simulation. Most regression tests depend on a specific `lconfig` configuration. `Rundvt` runs tests that match the configuration in the `$LX_HOME/include/lxr_symbols.vh` file.

When running tests from the `rundvt` command line there are no such guarantee. The "`PM3_icnt.s`" assembly regression test is designed to test some features of the coprocessor performance counters. This can only be run if the `lconfig` form selected COPTC3 on the coprocessor 3 port. In addition, nops has to be set to zero.

run_test("PM3_icnt.s", 0, "COP3=COPTC3");

If using "rundvt `PM3_icnt.s`" on the command line, it runs the test regardless of the configuration, which might result in false failures or false passes. If tests are to be executed from the command line, use the `-from_regression` option. This will allow `rundvt` to check against `$LX_HOME/include/lxr_symbols` to ensure that the tests are valid for the particular lconfig option and to run the test properly.

10.5 Simulation Flow

The `rundvt` script provides a communication mechanism between PERL and the Verilog simulation environment without using PLI. `Rundvt` encodes commands and puts them in the `$LX_HOME/regression/testloader.hex` file, which the $LX_HOME/`testbed/testbed.v` module interprets during simulation. This provides a robust, one-way communication between PERL and Verilog.

The commands in `testloader.hex` appear in Verilog format in the "`testloader.vh`" file. The `testloader.hex` file is read into a memory array within the `$LX_HOME/testbed/testbed.v` module. If `rundvt` composes commands which exceed the bounds of the memory array, errors will result. The following Verilog code fragment is in `$LX_HOME/testbed/testbed.v`. It defines a memory test list, which the testbed loads with `$LX_HOME/testloader.hex`.

reg [`TEST_LENGTH:1] test_list [`MAXTESTS:1];

10.6 Generating ASCII Traces in the Simulation Output

The Lexra bus monitor module `$LX_HOME/testbed/lmon.v` can be enabled during Verilog simulation to generate ASCII debug traces. The `$LX_HOME/testbed/testbed.v` determines which signals to trace with Verilog plus-args. A subset of traces requires that the Verilog macro `RTL_MON` be defined; therefore, only use them for RTL simulation.

```
`MonPath MONTrace[`MonTraceI] = $test$plusargs("trace_inst");
```

+trace_all

The `+trace_all` option enables the tracing of the internal ICACHE bus, the internal DCACHE bus, and the system bus. These traces are enabled individually with the following plus-args: `+trace_inst`, `+trace_data` and `+trace_system`.

Here is a sample output generated by the `+trace_all` option.

```
13620000 M000>>>> DATA 0xbfa00020 wr c 1111 mem  hit 0xffffffff
13630000 M000>>>> INST 0x00400018 rd c 1111 mem  hit 0x3442aaaa
13657800 M000>>>> SYS 0x1fa00020 wr c 1111 mem    - 0x9fa020e8
13760000 M000>>>> INST 0x0040001c rd c 1111 mem  hit 0x3c03ffff
13770000 M000>>>> INST 0x00400020 rd c 1111 mem miss 0xad00000c
```

The following table defines the labels used in the example.

TYPE	LABEL	DESCRIPTION
Time	decimal number	Time stamp produced by the $time Verilog system task.
ID Tag		Identifies which processor is being traced in a multiprocessor system. A multiprocessor system requires TESTBED_ENV==EXAMPLE.
	M000	Processor M00, thread 0. (note: thread is always 0 in single thread processor).
	M103	Processor M10, thread 3.
Trace Domain		Identifies which field is traced.
	INST	Instruction cache bus.
	DATA	Data cache bus.
	SYS	Lexra system bus.
Address	hexadecimal number	If the trace domain is INST or DATA then this field is a logical address. Otherwise it is a physical address.
Operation	rd	Read.
	wr	Write.
Cache Type	c	Operations to cached address spaces.
	u	Operations to uncached address space.

TYPE	LABEL	DESCRIPTION
Byte Lanes	1111	Bus operations accessing a full word.
	0001,0010,0011, 0100,1000,1100	Bus operations accessing bytes or half words.
Memory		Cached or uncached transaction corresponding to addresses in system memory.
	ram	Transaction mapping to a local D-RAM or I-RAM memory.
	rom	Transaction mapping to a local IRAM memory when using the *lconfig* option IRAM_IS_ROM=YES.
Status	hit	Indicates that the data requested by the instruction is in the cache or that the instruction is performing an uncached transaction.
	miss	The cache-line corresponding to the requested address is not present in instruction or data cache. For write transactions, the data is written to system memory.
	-	Valid only for system bus transactions. For write transactions: "-" always appears For read transactions: "-" means data is invalid and the processor disregards it.
	rdy	The data is valid and the LBC is reading or writing it
Data	hexadecimal number	Data on the bus.

+trace_pipem +define+RTL_MON

This option traces the M-stage of the pipeline and disassembles the instructions in pipe A. In superscaler products, pipe B is also displayed.

Single Issue Processor:

```
13787500 M000>>>PIPEA M-->W: RETIRE: 0x9fa02110: 37bd0008 ori$sp,$sp,x0008
13792500 M000>>>PIPEA M-->W: RETIRE: 0x9fa02114: 8fbd0000   lw$sp,x0000($sp)
13867500 M000>>>PIPEA M-->W: RETIRE: 0x9fa02118: 3c1cbfa0   lui$gp,$0,xbfa0
13872500 M000>>>PIPEA M-->W: RETIRE: 0x9fa0211c: 379c000c  ori$gp,$gp,x000c
13877500 M000>>>PIPEA M-->W: RETIRE: 0x9fa02120: 8f9c0000   lw$gp,x0000($gp)
13917500 M000>>>PIPEA M-->W: RETIRE: 0x9fa02124: 00000000 sll$0,$0,x00
```

In the sample output below, note that pairs of instructions are grouped using underscores ("_"). The output below shows program order, rather than one pipe always first.

Dual Issue Processor:

```
11580000 M000>>>PipeB M-->W: NOTVLD
11580000 M000>>>PipeA M-->W: RETIRE:_0x00400004:_34210020 ori $at,$at,0x0020
11590000 M000>>>PipeA M-->W: DUAL RETIRE:0x00400008:ac3f0000
                sw    $ra,x0000($at)
11590000 M000>>>PipeB M-->W:DUAL RETIRE:0x0040000c:_240a0001
                addiu$t2,$0,0x0001
11600000 M000>>>PipeB M-->W:RETIRE:0x00400010: 254a0001addiu $t2,$t2,0x0001
11600000 M000>>>PipeA M-->W:NOTVLD
11730000 M000>>>PipeB M-->W:NOTVLD
 11730000 M000>>>PipeA M-->W:RETIRE:0x00400014:_254a0001addiu $t2,$t2,0x0001
11740000 M000>>>PipeB M-->W:DUAL RETIRE:0x00400018: 254a0001
                addiu $t2,$t2,0x0001
11740000 M000>>>PipeA_M-->W:DUAL RETIRE:0x0040001c:_10000001
                beq   $0,$0,0x0001
11750000 M000>>>PipeA M-->W:SQUASH:0x00400020: 0002000d break $0,$0,$v0
11750000 M000>>>PipeB_M-->W:NOTVLD
```

The following table defines the labels used in both examples:

TYPE	LABEL	DESCRIPTION
Time	decimal number	The time stamp produced by the $time Verilog system task.
ID Tag		Identifies which processor is being traced in a multiprocessor system. A multiprocessor system requires TESTBED_ENV==EXAMPLE.
	M000	Processor M00, thread 0. (note: thread is always 0 in single thread processor).
	M103	Processor M10, thread 3.
Pipeline		Indicates which pileline is executing what instruction. The "_" character denotes grouping with the previous instruction (i.e. executed in parallel).
	PipeA	Executes load, store, cop0 and ALU opcodes.
	PipeB	Executes MULT, DIV and ALU opcodes.

TYPE	LABEL	DESCRIPTION
Code	NOTVLD	The indicated pipe does not contain a valid instruction. Invalid instructions are generated by the core whenever the code being executed dictates that both pipes cannot execute in parallel because of data dependencies, etc.
	RETIRE	The instruction in the indicated pipe completes while there is an invalid instruction in the other pipe.
	SQUASH	The instruction in the indicated pipe is SQUASHed while there is an invalid instruction in the other pipe.
	DUAL RETIRE	The instructions in both pipes complete successfully.
	HALF RETIRE	The indicated pipe has its instruction completed while the other pipe incurs a "HALF SQUASH." The indicated pipe carries the older instruction.
	HALF SQUASH	The indicated pipe has its instruction invalidated ("squashed") by an exception. The other pipe incurs a "HALF RETIRE." The indicated pipe carries the newer instruction.
	DUAL SQUASH	Both pipes have their current instruction invalidated because an exception is taken by the older pipe.
Address	hexadecimal number	Instruction logical Address (Program Counter)
Data	hexadecimal number	Instruction data (opcode)
Instruction	Instruction	Disassembled instruction and operands

+trace_exception +define+RTL_MON

The processor triggers this trace output when it takes an exception. The current state of the COP0 registers are displayed in hexadecimal. In addition, a count of all exceptions is displayed at the end of simulation.

```
6390000 M000>>>> XCPN 0xbfc00180 EPC:0x0040001c Cause:0x00000024
                                 Status:0x00400000

Count of exceptions
Int     ExcCode=00  BEV0=     0 BEV1=      0
Mod     ExcCode=04  BEV0=     0 BEV1=      1
TLBL    ExcCode=08  BEV0=     0 BEV1=     11
TLBS    ExcCode=0c  BEV0=     0 BEV1=      9
AdEL    ExcCode=10  BEV0=     0 BEV1=      0
```

```
AdES     ExcCode=14  BEV0=    0 BEV1=     0
Sys      ExcCode=20  BEV0=    0 BEV1=     0
Bp       ExcCode=24  BEV0=    0 BEV1=     0
RI       ExcCode=28  BEV0=    0 BEV1=     0
Ov       ExcCode=30  BEV0=    0 BEV1=    27
TLBL     ExcCode=08  UTL0=    0 UTL1=     0
TLBS     ExcCode=0c  UTL0=    0 UTL1=     0
```

+trace_regwrite +define+RTL_MON

Traces changes to register state used in combination with `+trace_pipem`, provides one of most used features of a source code debugger.

```
M000>>> GPR      130 W1A $10  t2 wr 0x55555555
M000>>> GPR      131 W1A $20  s4 wr 0x00001001
M000>>> GPR      132 W1A $11  t3 wr 0xaaaaaaaa
M000>>> GPR      133 W1A $20  s4 wr 0x00001002
M000>>> GPR      134 W1A $12  t4 wr 0x00000000
M000>>> GPR      135 W1A $20  s4 wr 0x00001003
M000>>> GPR      136 W1A $13  t5 wr 0x00000000
M000>>> GPR      137 W1A $20  s4 wr 0x00001004
M000>>> GPR      138 W1A $10  t2 wr 0x55555555
```

The following table defines the labels used in the example.

TYPE	LABEL	DESCRIPTION
ID Tag		Identifies which processor is being traced in a multiprocessor system. A multiprocessor system requires TESTBED_ENV==EXAMPLE.
	M000	Processor M00, thread 0. (note: thread is always 0 in single thread processor).
	M103	Processor M10, thread 3.
	GPR	General Purpose Register
Port Activity	W1A or W2A	Write to port 1 or 2 from Leg A
Register Number	hexadecimal	Identifies the register number being addressed.
Register Name	t2, s4, etc	Identifies the register by its conventional mnemonic name being accessed.
Operation	wr	write
Data	hexadecimal	write data.

+trace_memread

State of the memory model are traced with **+trace_memread** plus-arg.

```
M--->>>  MEM 0x1fa02020 rd          0xad00000c
M--->>>  MEM 0x1fa02024 rd          0xad000010
M--->>>  MEM 0x1fa02028 rd          0xad000014
M--->>>  MEM 0x1fa0202c rd          0x3c08bfa0
```

+trace_memwrite +define+RTL_MON

Changes in the state of the memory model are traced with `+trace_memwrite` plus-arg.

```
M--->>>  MEM 0x1fa00004 wr          0x9fa02000
M--->>>  MEM 0x1fa00000 wr          0x00400000
M--->>>  MEM 0x1fa00460 wr          0x00000010
M--->>>  MEM 0x1fa0000c wr          0x00600000
```

+trace_lbus

Traces the LBUS activities including master and target devices, address, data and commands

```
10640100 Bus0>>> LBUS SELSTALL
10645100 Bus0>>> LBUS LASTDATA    Target = 0010  Data = 0x00000000
10800100 Bus0>>> LBUS ADDR        Master = 0001  Addr = 0x1fa02120  Cmd = 0x30
10805100 Bus0>>> LBUS SELSTALL
10810100 Bus0>>> LBUS DATA        Target = 0010  Data = 0x25080008
10815100 Bus0>>> LBUS DATA        Target = 0010  Data = 0x3c08bfa0
10820100 Bus0>>> LBUS DATA        Target = 0010  Data = 0x35080230
10825100 Bus0>>> LBUS LASTDATA    Target = 0010  Data = 0xad000000
10840100 Bus0>>> LBUS ADDR        Master = 0001  Addr = 0x1fa00030  Cmd = 0x4b
```

Lexra supports many types of traces. For a complete description of all available traces, please see `$LX_HOME/testbed/lmon.v`.

10.6.1 Tracing Through Hierarchical References

To control hierarchical references, only Verilog macros can be used. Lexra has defined a convention where several levels of references are allowed. These options are mainly used with RTL netlists. Because synthesized gate level netlists will not have the same name references as the RTL netlists, RTL_MON and LOCAL_MON will not work in gate level simulation.

+define+LOCAL_MON

This symbol is added to the verilog compiler by default by `rundvt` in order to allow a limited amount of peeking. Without LOCAL_MON, only the system bus may be monitored. The testbed will attempt to peek into lx2 but not deeper. This may be used with gate level netlists below `lx2` level. However, tests may fail when simulating with both behavioral and gate level netlists, especially critical timing at the behavioral/gate boundaries.

+define+RTL_MON

RTL_MON permits any signal in the design to be examined below `lx2` hierarchy. Code can be seen where RTL_MON will enable hierarchical references in `$LX_HOME/testbed/lmonw.v`

```
‘ifdef RTL_MON
        wire [7:0] ProcNum = ‘Lx0Path CFG_PROCNUM;
‘else // not RTL_MON
        wire [7:0] ProcNum = 8’h0;
‘endif
```

The different modes of hierarchical references are summarized in the following table with the various `rundvt` options.

	Allow Hierarchical References	**Add ‘-f gate.f’ to Verilog Compiler**
rundvt default	yes	no
add ‘-gates’	no	yes
files with *hv,*vo,*vm	no	yes
CUSTOM_FILES=YES	yes (default)	yes
SIM_TECH=YES	yes (default)	yes
Add ‘-nopeeking’	no	yes (default)
Add ‘+define+LOCAL_MON’	yes	yes (default)

10.6.2 Sparse Memory Tracing

A 4GB system memory model, sparsememory.v is included in `$LX_HOME/testbed`. The `$LX_HOME/testbed/testbed.v` loads an ASCII binary file into the system memory for testing. The verilog macros below enable informational traces.

+define+SPARSEMEMORY_TRACE

The sparse memory model uses a single level hash to provide fast access to the memory pages. The dump function will be invoked when the `SPARSEMEMORY_TRACE` is defined. It will display the state of the hash. A linked list of collision pool records backs up the hash.

```
Allocating page    key=00001045 tag=40405400 page=00001f00
Allocating page    key=00001046 tag=40405800 page=00002000
Allocating page    key=00001047 tag=40405c00 page=00002100
Allocating page    key=00001048 tag=40406000 page=00002200
Allocating page    key=00001049 tag=40406400 page=00002300
Allocating page    key=0000104a tag=40406800 page=00002400
Allocating page    key=0000104b tag=40406c00 page=00002500
Allocating page    key=0000104d tag=40407400 page=00002700

HASH[00001044] tag=40405000 page=00001e00 count=   354
HASH[00001045] tag=40405400 page=00001f00 count=   255
HASH[00001046] tag=40405800 page=00002000 count=   255
HASH[00001047] tag=40405c00 page=00002100 count=   255
HASH[00001048] tag=40406000 page=00002200 count=   388
HASH[00001049] tag=40406400 page=00002300 count=   354
HASH[0000104a] tag=40406800 page=00002400 count=   407
HASH[0000104b] tag=40406c00 page=00002500 count=   255
```

+define+SPARSEMEMORY_TRACE_HIT

Trace read and write hits to hash.

```
Hit page    key=0000047e tag=405ffc00 page=00003400
Hit page    key=00000015 tag=1c000000 page=00003600
Hit page    key=00000015 tag=1c000000 page=00003600
Hit page    key=0000047e tag=405ffc00 page=00003400
Hit page    key=0000047e tag=405ffc00 page=00003400
Hit page    key=00000015 tag=1c000000 page=00003600
Hit page    key=00000015 tag=1c000000 page=00003600
Hit page    key=0000047e tag=405ffc00 page=00003400
```

+define+SPARSEMEMORY_TRACE_VERBOSE

Trace every read and write accesses to the system memory model.

```
get_memory(1fa00120) memory[00000748]=00000000
get_memory(1fa000a4) memory[00000729]=00000000
get_memory(1fa00124) memory[00000749]=00000000
get_memory(1fa000a8) memory[0000072a]=00000000
get_memory(1fa00128) memory[0000074a]=00000000
get_memory(1fa000ac) memory[0000072b]=00000000
get_memory(1fa0012c) memory[0000074b]=00000000
get_memory(1fa00230) memory[0000078c]=00000000
```

Chapter
11

Synthesizing the Lexra CPU

11.1 Overview

The Lexra database comes with a complete synthesis environment. To setup this environment for your design only a few files need to be modified. Example modifications are: library name and location, cpu clock speed, RAM timing information. We provide default values for several parameters and you can change them to gain more control over the provided synthesis script. Section 11.2, Setting up the Synthesis Environment explains the required and optional setups.

Section 11.3, Running Synthesis explains how to synthesize the database and Section 11.4, Synthesis Output Files explains the output files generated by the synthesis session.

You can configure the RTL code to synthesize to gates at the push of a button. This provides high-quality synthesis results. There are back-end optimizations that are not practical to automate. Consequently, Lexra's synthesis procedure is not a step-by-step set of instructions to get from RTL to silicon. Section 11.5, Considerations discusses these topics.

The final Section 11.6, Structure of the Synthesis Environment goes into more detail about how the Lexra Makefile driven synthesis environment works.

11.2 Setting up the Synthesis Environment

This discusses the three files you need to modify to customize your environment:

- .synopsys_dc.setup
- dont_use.scr
- techvars.scr

11.2.1 .synopsys_dc.setup

You need to customize the file `$LX_HOME/user/tech/.synopsys_dc.setup which is created after lconfig is run`.

This file specifies the Synopsys library file(s) to use and the search path to the file(s). Be sure to add RAM libraries to your link_library list.

The `tech` directory as shown above is a link created by `lconfig`. It points to the directory specified by the following variable in the `lconfig` form.

TECHNOLOGY = "CUSTOM"

Usually TECHNOLOGY will be set to CUSTOM and thus the `tech` directory will point to the `custom` directory. However, if synthesis to multiple technologies is required using the same Lexra database, modifying the TECHNOLOGY variable to CUSTOM_<my_name> and re-running `lconfig` will relink the `tech` directory to the new directory CUSTOM_<my_name>. This helps in managing the database when synthesizing to multiple technologies.

You can also use this file to set default values on `dc_shell` variables for all blocks in the RTL database. For example, you can specify `bus_naming_style` or `define_name_rules` in this file. All `dc_shell` scripts in the environment use this file.

note: The synthesis methodology writes out a netlist for each <block> at each level of the hierarchy and reads that netlist when synthesizing the next level of the hierarchy. This requires that the bus naming style be consistent between the RTL and the synthesized netlist. Therefore, we recommend that you do not change `dc_shell` variables such as `bus_naming_style` or `define_name_rules`. If you change them, you must maintain the consistency described above or the hierarchical make process will not be successful.

Parameters to Set

Variable	Required/optional	Example
link_library	required	link_library = { "*" , tech.db}
target_library	required	target_library = { tech.db }
search_path	required	search_path = search_path + "path_to_library_file"
symbol_library	optional	symbol_library = tech.sdb

11.2.2 dont_use.scr

You may customize the file `$LX_HOME/user/tech/dont_use.scr`. This file allows you specify cells in the synthesis library that the synthesis tool may not use.

Do not remove these two `set_dont_use` statements in the `dont_use.scr` file:

```
set_dont_use { standard.sldb/DW01_add/rpl, standard.sldb/DW01_addsub/rpl, standard.sldb/
DW01_sub/rpl }

set_dont_use { standard.sldb/DW01_inc/rpl, standard.sldb/DW01_incdec/rpl, standard.sldb/
DW01_dec/rpl }
```

These commands prohibit Design Compiler from using DesignWare ripple-carry implementations of adders, subtractors, addsubs, incrementers, decrementers and incdecs. Many versions of Design Compiler have trouble swapping out such cells when a faster implementation is needed to meet timing constraints. Therefore, it is often difficult to get good timing results without prohibiting these DesignWare parts.

When SCAN_INSERT=NO, the synthesis tool may take advantage of scan flops to implement logic that requires a mux followed by a non-scan flop. If your goal is to insert scan post-synthesis, this will cause a problem. One way to avoid the problem in this case is to `set_don't_use` on scan flops. This can be done using the following example as a guide:

```
if (SCAN ==0) {
  set_dont_use { TECHLIB + "/SDFFHQX1" }
  set_dont_use { TECHLIB + "/SDFFHQX2" }
  set_dont_use { TECHLIB + "/SDFFHQX4" }
}
```

11.2.3 techvars.scr

You must customize the file `$LX_HOME/user/tech/techvars.scr`.

The file `techvars.scr` offers you many variables for controlling the synthesis environment. You need to set some variables, but frequently the default settings are the right ones.

Below are tables of available variables. The first table displays the variables you must customize to get the synthesis scripts to run without errors. The second table displays the variables whose default values the synthesis script can run without errors.

Variables Needing Customizing

Variable	Function
TECHLIB	Synopsys technology library name (not a file name)
BCOPCOND	best case operating condition for checking hold time violations (NOT USED at this time)
WCOPCOND	worst case operating condition in `TECHLIB` for optimizing setup violations
CLKPERIOD, BUSCLKPERIOD, JTAGCLKPERIOD[1]	Sysclk, busclk and jtagclk clock period
DCELL	basic driving cell, like size 1 or size 2 inverter for calculating standard drive of input signal.
DCELLPIN	output pin on `DCELL`
MAXAREASCALE	variable for scaling the Synopsys max_area limit from the original reported area
MAXFANOUTV[2]	limits fanout on ports of each module (is a floating point number which is an integral multiple of 1.0)
RAM_DCACHE_DATA_INDEX_SETUP[3]	setup time from positive edge of clock (sysclk) for address bus on DCACHE
RAM_DCACHE_DATA_WE_SETUP[3]	setup time from positive edge of clock (sysclk) for write enable on DCACHE
RAM_DCACHE_DATA_WR_SETUP[3]	setup time from positive edge of clock (sysclk) for write data bus on DCACHE
RAM_DCACHE_DATA_RD_DELAY[3]	read access delay from positive edge of clock (sysclk) to valid output on memory for DCACHE

Variable	Function
RAM_DCACHE_DATA_RE_SETUP[3]	setup time from positive edge of clock (sysclk) for read enable
RAM_DCACHE_DATA_CS_SETUP[3]	setup time from positive edge of clock (sysclk) for chip select
ROM_IROM_INST_INDEX_SETUP	setup time from positive edge of clock (sysclk) for address bus on IROM
ROM_IROM_INST_RE_SETUP	setup time from positive edge of clock (sysclk) for read enable
ROM_IROM_INST_CS_SETUP	setup time from positive edge of clock (sysclk) for chip select
ROM_IROM_INST_RD_DELAY	read access delay from positive edge of clock (sysclk) to valid output on IROM

1 CLKPERIOD is the system clock period, BUSCLKPERIOD is the bus clock period, and JTAGCLKPERIOD is the JTAG clock period. The Lexra LBC has an optional asynchronous interface that operates correctly with any bus clock frequency relative to system clock frequency. To make the synthesis tool work effectively, set these variables so that their ratio is a small integer, for example BUSCLKPERIOD/CLKPERIOD = 1, 2, 4, 8.

2 Many synthesis libraries model fanout restrictions poorly. For example, they may model fanout for input pins exactly as they do capacitance or for output pins exactly as they do drive strength. The synthesis tool, using only this information, provides poor ability to control fanout. The Lexra synthesis environment therefore, puts a default max_fanout attribute on all cells' output pins. Specifically, executing the `lib_fanout.scr` script puts a default max_fanout attribute on all library cells' output pins. The default is MAXFANOUTV. This means that no cell can drive a node with a fanout_load exceeding MAXFANOUTV, without causing a max_fanout design rule violation. Thus, the MAXFANOUTV variable provides an efficient way for you to control the fanout in the design.

3 These variables describe timing characteristics of the memories for the synthesis tool. This table shows them for the Dcache data RAM only. The `techvars.scr` file provides the same parameters for the Instruction cache store, Instruction cache tag, Data Memory (DMEM, no tags) and Instruction memory (IMEM, no tags). Each type of RAM has six parameters. Their naming style is consistent with those the Dcache has in the table.

There are six possible types of RAMs and one type of ROM in the Lexra CPU core. Choosing a 2-way set associative instruction cache will cause multiple instances of instruction cache data and tag RAMs.

DCACHE_DATA

DCACHE_TAG

DRAM_DATA

ICACHE_INST

ICACHE_TAG

IRAM_INST

IROM_INST

Each RAM has six timing parameters.

INDEX_SETUP
WE_SETUP
RE_SETUP
CS_SETUP
WR_SETUP
RD_DELAY

The IROM has four timing parameters.

INDEX_SETUP
RE_SETUP
CS_SETUP
RD_DELAY

The `techvars.scr` variable names are the concatenation of the `string: "RAM_" (or "ROM_"), the memory name, and the parameter name`:

RAM_DCACHE_DATA_INDEX_SETUP
RAM_DCACHE_DATA_WE_SETUP
RAM_DCACHE_DATA_RE_SETUP
RAM_DCACHE_DATA_CS_SETUP
RAM_DCACHE_DATA_WR_SETUP
RAM_DCACHE_DATA_RD_DELAY

Variables Not Needing Customizing

Variable	Function
TECHLIB2	synopsys technology library name for a second library (optional)
TECHNAME	local library name useful for customizing scripts, not in use
RPTCLKPERIOD, RPTBUSCLKPERIOD, RPTJTAGCLKPERIOD[1]	report clock periods. report timing uses these for system clock, bus clock and jtag clock
RPTMAX	argument to Synopsys command `report_timing -max_path`
RPTWORST	argument to Synopsys command `report_timing -worst`

Variable	Function
IODELAYSCALE[2]	scaling factor for `input_delay` and `output_delay` values. Do not modify. Should be `CLKPERIOD/10`.
LXDEBUG	set to 1 for verbose logging in synthesis log file
CLKMUNCERT, BUSCLKMUNCERT, JTAGCLKMUNCERT	minus uncertainty applied to system clock, bus clock and jtag clock for modeling clock tree effects before layout (computing clock edge times). Use minus uncertainty when checking setup (maximum path) delays. It effectively shifts the clock edge to the left, making the clock edge earlier. Larger uncertainty implies tighter setup constraints.
CLKPUNCERT, BUSCLKPUNCERT, JTAGCLKPUNCERT	plus uncertainty applied to system clock, bus clock and jtag clock for modeling of clock tree effects before layout (computing clock edge times). Use plus uncertainty when checking hold (minimum path) delays. It shifts the clock edge to the right, making it later. Larger uncertainty implies tighter hold constraints.
RPTCLKMUNCERT, RPTBUSCLKMUNCERT, RPTJTAGCLKMUNCERT	same as `CLKMUNCERT`, `BUSCLKMUNCERT`, and `JTAGCLKMUNCERT` but for the report clocks
RPTCLKPUNCERT, RPTBUSCLKPUNCERT, RPTJTAGCLKPUNCERT	same as `CLKPUNCERT`, `BUSCLKPUNCERT`, and `JTAGCLKPUNCERT` but for the report clocks.
CLKHPERIOD, BUSCLKHPERIOD, JTAGCLKHPERIOD, RPTCLKHPERIOD, RPTBUSCLKHPERIOD, RPTJTAGCLKHPERIOD	half period for sysclk, busclk, jtagclk and corresponding report clocks. Default is 50% duty cycle.
CRITRANGE	critical range for compile command. Specifies a margin of delay for path groups in optimization. A critical range of 0.0 means that only the most critical paths (the ones with the worst violation) are optimized. For a nonzero critical range, near-critical paths within that amount of the worst path are also optimized if possible. Default: critical range is 10% of the clock period. Set to zero or small portion of clock period time since it has a large impact on runtimes.

1 The synthesis tool frequently produces better timing results when the timing is overconstrained. Accurate static timing analysis requires the actual clock period and clock uncertainty. Currently, the positive uncertainty is increased to overconstrain the synthesis tool. For each clock, there is a clock period (CLKPERIOD) and a reporting clock period (RPTCLKPERIOD). By default, all reporting clock periods are equal to the actual clock period.

2 Constraints are based on a 10ns clock. The file constraints.template contains this constraint data on each port. These constraints are scaled linearly for different clock frequencies in the constraint files (`io_*.con`). The scale factor by default is `IODELAYSCALE = CLKPERIOD/10`.

11.2.4 Using Pre-defined Technologies

If you use one of the technologies Lexra has already defined, setting up the synthesis environment requires only the following steps.

1. Set the `lconfig` option TECHNOLOGY and run `lconfig` on the edited form.

2. Customize the `.synopsys_dc.setup`, `dont_use.scr`, and `techvars.scr` files. They are in the `$LX_HOME/user/tech` directory.

11.2.5 Synthesis Wire Load Models

Wire load models are controlled by the synthesis variables SYNWLM, DPWLM and DWWLM found in `techvars.scr` and `<module>.scr.template` files. Setting these variables to "`auto`" causes the synthesis tool to automatically select the Synopsys library specific wire load model for the module being synthesized based on the wire_load_selection function in the library.

Setting the three variables to "`modulewlm`" will allow the use of a unique wire load model for each module in the design.

note: A "module" signifies one of the major modules in the Lexra RTL hierarchy. These modules, many times, will contain submodules.

When using "`modulewlm`", the name of the wireload model must be named `<module>_wire.lib`. This option allows you to use layout tool generated wireload models to achieve better synthesis results. You will need to create a `<module>_wire.lib` file for each synthesized module. That is, there is a directory `$LX_HOME/syn/<module>` for each major module in the Lexra RTL hierarchy. If using "`modulewlm`", each directory must contain a file `<module>_wire.lib`. You can confirm which wire load models were used in synthesis by looking at the `timing_real.rpt` file.

Setting SYNWLM, DPWLM and DWWLM to "`<model name>`" allows the use of a single wireload model for all modules. This will look for the file `<model_name>_wire.lib`.

note: Do not change SYNWLM in the following four files: `lx0.scr.template`, `lx0c.scr.template`, `lx1.scr.template`, `lx2.scr.template`. SYNWLM must be set here to "zero" (a Lexra included wireload model) because the routing parasitics normally handled by the wire load model are included in the constraints file io_rpt.con for these modules.

11.3 Running Synthesis

To synthesize the entire Lexra CPU:

- set up the synthesis environment by updating the following three files as described in the previous section:

  .synopsys_dc.setup
  dont_use.scr
  techvars.scr

- Verify your setup by synthesizing a small block:

  cd $LX_HOME/syn/<block_name>/ ; make

  for example:

  cd $LX_HOME/syn/reset_dist/ ; make

- Synthesize Lexra Processor at either the `LX1` or `LX2` level of the hierarchy:

  cd $LX_HOME/syn/lx2; make

11.4 Synthesis Output Files

Each Synopsys script writes out the following files:

<block>.hv	hierarchical Verilog netlist
<block>.db	synthesized database in Synopsys database format
timing_real.rpt	timing report (using real constraints) using *RPTCLKPERIOD* and *RPTBUSCLKPERIOD*
constraint_real.rpt	constraint violation report (using real/report constraints)
design.rpt	*check_design* output
port.rpt	verbose information on port constraints
pin_timing.rpt	timing report on I/O ports only
<block>.autoxp	scan path report (only when SCAN_INSERT=YES)
<block>.tpf	test protocol file (only when SCAN_INSERT=YES)
test.rpt	output of check_test (only when SCAN_INSERT=YES)

11.5 Considerations

11.5.1 Synthesizing Clock Trees

During initial synthesis treat clocks as ideal and don't try to force synthesis to create the clock trees for you. Having synthesis create the clock buffer tree produces poor results since the tool has no layout information to work with. Instead, use back-end tools to create clock trees based on layout and timing constraints.

Lexra's synthesis scripts model the clock distribution tree as if it were driven by an ideal driver (infinite drive strength). The synthesis environment does provide parameters for modeling clock uncertainty in terms of plus and minus uncertainty of the arrival of the clock edge. Use these parameters (see Section 11.2.3, techvars.scr) with the Synopsys command `set_clock_skew` in the `syn/opt.scr` file as follows:

```
set_clock_skew -ideal \
        -delay 0 \
        -minus_uncertainty CLKMUNCERT \
        -plus_uncertainty CLKPUNCERT \
        find(clock, Clock)
```

We recommend that you not modify the default values, which are percentages of the actual clock period.

11.5.2 Back-end and IPO Considerations

Your first synthesis ideally results in a netlist without any timing or design rule violations. The next design phase is to floorplan and/or place and route the netlist. After this step, your design is likely to have timing and/or design rule violations because the layout tool provides you with actual wire parasitic resistance and capacitance.

You have to resynthesize the netlist incrementally, using back-annotated data from the back-end flow unless the violations are acceptable. This kind of iteration, performing incremental re-synthesis based on back-annotated data, is often called IPO (in-place optimization) in the synthesis level or ECO (engineering change order) in the layout level.

It is very complicated to create a generic, automated and still efficient IPO or ECO synthesis flow and it is beyond the scope of the synthesis environment provided by Lexra. Some consideration in the initial synthesis may help later, when you do ECO or IPO types of optimizations.

Be conservative but not too limited when specifying constraints.

Consider disabling (`set_dont_use`) strong drive cells in the synthesis so you have them available for IPO optimization.

If the vendor's technology library has poor wire-load models, consider generating custom wire-load models from a floorplan and place and route. Then re-synthesize from scratch using the newly generated wire-load models.

Model clock tree uncertainties as accurately as possible in initial synthesis but use `set_clock_skew -propagated_delay` with back-annotated SDF data when doing IPO or ECO types of optimization.

11.5.3 Reordering Scan Chains

Lexra's synthesis environment inserts and connects scan chains if you set the `SCAN_INSERT` option. You usually reorder the scan chains in the back-end based on layout considerations.

Scan chain reordering as well as the IPO/ECO loop outlined in the previous section is hard to make generic, automated, and still efficient. It is beyond the scope of Lexra's synthesis environment. You must recreate ATPG vectors if you re-optimize the scan chains.

11.5.4 Library Recommendations

The Lexra CPU is a generic, technology independent RTL design that does not rely on any specific library components.

For optimal synthesis results, we recommend that the library contains

- cells with many different drive strengths per logic function
- scan flip-flops
- a rich selection of wire-load tables
- several different operating conditions or different library files for different operation conditions (i.e typical, slow, fast)

11.6 Structure of the Synthesis Environment

When you run `lconfig`, it reads your form and sets up your environment for both simulation and synthesis. The synthesis files reside in the `$LX_HOME/syn/<block>` directories after `lconfig` is run. The lconfig script figures out if you have technology specific files for synthesis (for example RAM wrappers, sleep clock buffers, etc) based on finding `*.v` files in the `$LX_HOME/*/<tech>` directories. If these files exist, `lconfig` creates appropriate links to them from the appropriate `syn` directory.

When it locates your technology specific file, `lconfig` displays the message

using technology specific file <pathname> for synthesis

For example, if TECHNOLOGY is set to CUSTOM and you are using a 1K DCACHE, you create the RAM wrapper `$LX_HOME/chip/custom/sram_dc_data_256x32.v`. The `lconfig` script then creates a link in the `$LX_HOME/syn/lx2` directory:

sram_dc_data_256x32.v -> ../../chip/custom/sram_dc_data_256x32.v

When you synthesize the `lx2` module, the synthesis script reads the linked file. For certain modules, RAM wrappers for example, `lconfig` issues an error message if it is unable to locate the technology-specific file to use for synthesis.

Prior to running `lconfig`, there are no `syn/<block>` directories. The lconfig script creates the `syn/<block>` directories and copies the `<block>.files & <block>.scr.template` files from `syn/syntrol` to them.

The lconfig script reads the `<block>.files` file and your form and creates the following files in each `syn/<block>` directory that needs to be synthesized for your configuration.

Makefile	
`.synopsys_dc.setup`	link to technology specific file
`chk_logs`	link to script to detect problems in Synopsys log file
`include`	link to include directory
`<blocks>_subs.scr`	list of Verilog submodules read by synthesis script

The `lconfig` script then creates a `makefile` in the unused `syn/<block>`

directories so if you stumble into one of these by accident, the `makefile` tells you that this configuration doesn't use that `<block>`.

Constraint information for each block is located in the file `constraints.template`. The lconfig script and the synthesis `makefile` use the information in `constraints.template` to create the module level constraint files requrired for synthesis (`syn/<module>/io_synth.con`).

In addition to `set_input_delay` and `set_output_delay` constraints and `set_false_path` timing exceptions, the module level constraint files include `set_load` commands to add a load attribute to a specified value on all output ports and internal nets that connect to output ports of synthesized sub-blocks. There are five load values (*OUT_LOAD_W1*, ... *OUT_LOAD_W5*) specified in the `/$LX_HOME/user/tech/techvars.scr` file. For example:

```
OUT_LOAD_W1   = 0.01  /* load assigned by mk_syn_con, smallest connection */
OUT_LOAD_W2   = 0.02  /* load assigned by mk_syn_con,typical nearby connection*/
OUT_LOAD_W3   = 0.05  /* load assigned by mk_syn_con,in-between-ish connection*/
OUT_LOAD_W4   = 0.10  /* load assigned by mk_syn_con, typical far connection */
OUT_LOAD_W5   = 0.20  /* load assigned by mk_syn_con, biggest connection */
```

The `build_constraints` script called by the synthesis makefile checks the blocks in the hierarchy to ensure that the constraints are consistent between levels of hierarchy.

Each `Makefile` does the following:

- change directory to `../syn/syntrol` and calls the local `makefile` with an argument to provide the desired functionality.

This functionality includes commands that:

- run `build_constraints` to create a `<block>.data` file containing the I/O timing.
- run `mk_syn_con` on the `<block>.data` file to get `io_{synth,rpt}.con` synthesis constraint files
- runs `vpp` on the `<block>.v` file to generate a local `<block>.v` file without any ifdef logic (since the synthesis tool may not be able to parse this)

- for subblocks not synthesized separately, runs `vpp` to generate a local `<subblock>.v` file

- if the `<block>.scr.template` file exists, runs `vpp` to create the configuration-specific `<block>.scr` synthesis script. The `<block>.scr` file sets some variables and includes other synthesis command scripts.

- runs `dc_shell` on the `<block>.scr` file and write the results to the screen and to the `<block>.scr.log` file.

- runs the script `chk_logs` after the synthesis job is complete and writes potential problems to the screen and to `chk_logs.log`.

To get started with synthesis after your `lconfig` form is complete, run `lconfig` and try synthesizing a small block:

cd $LX_HOME/syn/reset_dist
make

If you don't have any problems, you can try synthesizing a high-level block:

cd $LX_HOME/syn/lx2
make

You can run **make -n** to see what commands `make` wants to run without executing them, or you can run **make -nd** to see explanations as well.

Most of the targets in the makefiles depend on `syn/syntrol/<block>.vh` that are written by `lconfig`. Therefore when you modify a form and re-run `lconfig`, the makefile will know that it only has to update blocks where the configuration has changed.

Chapter
12

Simulation Guidelines

There are no special simulation requirements for Lexra's processors. The RTL code uses standard coding techniques and single edge clocked flip flops. Engineers have simulated synthesized gate level netlists with numerous third party libraries, for example Artisan TSMC 0.25µm, 0.18µm and 0.13µm, Aspec IBM 0.25µm, Faraday UMC 0.25µm, Nurlogic IBM 0.25µm, and STMicroelectronics 0.25µm.

Nevertheless, this chapter offers several useful notes on simulation issues users may encounter.

12.1 Verilog

The syntax of Verilog options may differ between each Verilog simulator. Verilog-XL, NC-Verilog and VCS all accept Verilog simulation options using the same syntax.

12.1.1 Verilog Macro Definition on Simulator Command Line

If the user wanted to define a Verilog macro for use by all Verilog files then the following syntax would be used to define a macro call EXAMPLE.

Command line syntax: **+define+EXAMPLE**

Verilog usage:

```
`ifdef EXAMPLE
// Conditionally enabled code at compile time
`endif
```

12.1.2 Verilog System Function $test$plusargs

The $test$plusargs function looks for the existence of a user defined plusarg at runtime.

Command line syntax: **+example**

Verilog usage:

```
if ($test$plusargs("example"))
begin
// Conditionally enabled code at runtime
end
```

12.1.3 Verilog Simulator Specific Options

For more information and use of these options refer to the simulator users guide. A common example is the -RI option in VCS:

rundvt -RI -from_regression Logical.s

The compiler option automatically starts the Virsim wave form view and graphical simulation control following a successful Verilog compilation.

12.2 RAM Models

The RAM behavioral models shipped with Lexra's processor include event filters for eliminating the race conditions. These conditions can result when simulating the processor gate level netlists with RAM behavioral models. Be aware of the potential for race conditions when simulating with RAM behavioral models not supplied by Lexra. See the information provided in the Lexra behavioral RAMs found in the `$LX_HOME/chip` directory.

There is a wide range of quality across library vendors' structural RAM models. Some don't support back annotation with delays from SDF. Others can't turn off post layout timing. Lexra has experienced numerous difficulties getting the structural models integrated into the simulation environment. Therefore, it is recommended to test the vendors' RAM models as early as possible. The ram wrapper for the vendor rams should be pin compatible with the tsyncram_example model that `lconfig` generates. The model is found in `$LX_HOME/chip/custom` directory after running `lconfig`.

When connecting vendor specific ram models to the Lexra processor, be careful of uninitialized contents, X's since they can cause problems in simulation. In the Lexra simulation environment, these problems can be diagnosed pretty quickly because it has checkers to warn the user that something in the RAM interface is X. See Section 4.4, Using Library Vendors' RAM Models for more details.

12.3 Reset

When simulating the core outside of the Lexra regression environment, ensure that the reset signal duration is at least ten times longer than the system clock or bus clock period. This allows the processor to exit reset properly. While the processor and LBC are held in reset, the LBUS may be used by other peripheral such as I^2C loading bootstrap code into sdram.

The normal boot sequence occurs after reset is de-asserted. If there are caches, the cache controllers initialize the tags for ICACHE and DCACHE. This takes one cycle per cache line. The cache with the largest number of lines determines the number of cycles since ICACHE and DCACHE tags are initialized in parallel. The tag initialization is completed before the first instruction is fetched. Additional cycles are required for the instruction request to propagate to LBUS. Therefore, it will take the largest of number of DCACHE lines or ICACHE lines plus some additional cycles before the first instruction fetch activity is seen on LBUS.

The first instruction fetch for normal operation begins at the reset vector 0xBFC0_0000. This location usually contains the starting address of the user's reset routine or bootup code. The boot code can copy another program or RTOS to memory. After it finishes copying to memory, the program can jump to the new program in memory.

12.4 Testbed Models

All of the testbed modules Lexra supplies sample their inputs on the rising edge of the clock and update their outputs on the falling edge of the clock. This avoids potential race conditions between the processor and the testbed, even when the processor is simulated at the gate level. It is recommended that any new testbed modules the user creates should follow this approach. This will also make gate simulations with timing probably fail if tested at speed.

12.5 Libraries

It is recommended that the user follows the library vendors' simulation guidelines. Add any simulator command-line arguments recommended by the library vendor to the technology specific gate file. Running `rundvt -gates` adds '`-f /user/custom/gate.f`' to the simulator command line.

12.6 Gate Level Simulation

It is strongly recommend that simulation with post layout timing and gate level netlist are performed. They should pass the regression tests. Although, the entire regression suite does not have to be run. A subset of the regression testlist may be specified by using `-from` and `-to rundvt` options. See 10.3.2, Advanced Options.

Do not run gate level simulation at high frequencies (e.g. too close to the STA verified speed) since the gate level simulation models and timings are not exactly the same as the STA models. Otherwise, critical timing on a behavioral/gate boundary will cause simulation to fail.

There are several steps to take when using `rundvt` to run regression testing on the synthesized gate level design. In the file `$LX_HOME/user/tech/gate.f`, ensure the appropriate verilog files are present including technology specific rams where applicable. For example in `gate.f`,

```
-v library/tsmc/artisan/C018/aci/sc/verilog/tsmc18_lxr_udp_dff.v
-v $LX_HOME/chip/custom/RA1SH_512x20.v
+ notimingchecks
+ nospecify
```

By default, in the way the netlists are setup, the synthesized netlists entered on the command line will override the ".v" files since the ".v" files are treated as library files. To ensure that no RTL models are used in the gate level simulation, comment out the `lx0`, `lx0c`, `lx1` and `lx2` inpfiles in `$LX_HOME/regression/vcs.inpfiles`. For example in `vcs.inpfiles`,

```
-f board.inpfiles
-f chip.inpfiles
//-f lx0.inpfiles
//-f lx0c.inpfiles
//-f lx1.inpfiles
//-f lx2.inpfiles
-f tm_lbus.inpfiles
```

```
-f testbed.inpfiles
-f system.inpfiles
```

To run the regression testing on gate level netlist, use `-gates` option which will include `$LX_HOME/user/custom/gate.f` for ASIC cell libraries and `-nopeeking` option to disable hierarchical references in the testbed.

rundvt -gates -nopeeking <your_path/synthesized_netlist> <test>

12.6.1 Back Annotation

Verilog simulators support SDF for timing back annotation. To run the gate level simulation with real annotated timings in Lexra's testbed environment, there are several additional steps that must be taken. Verilog simulators do not support DSPF.

In the file `$LX_HOME/user/tech/gate.f`, comment out the last two lines so the simulator can back-annotate the timing and do timing checks (e.g. setup & hold).

```
-v library/tsmc/artisan/C018/aci/sc/verilog/tsmc18_lxr_udp_dff.v
-v $LX_HOME/chip/custom/RA1SH_512x20.v
//+notimingchecks
//+nospecify
```

In the file `$LX_HOME/system/control.v`, the following needs to be fixed.

1) SDF file name
2) top-level module hierarchical reference (the level at which SDF is annotated)
3) log file name

Make sure the SDF file is in the `$LX_HOME/regression` directory.

For VCS:

% **cd $LX_HOME/regression**

% **rundvt -gates [options] system/control.v +compsdf +maxdelays +sdfverbose +neg_tchk**

VCS will have a top level module control in addition to the normal topsys and testbed. `+compsdf` tells VCS to use compiled SDF. If `+compsdf` is not used then VCS will need an additional file sdf.tab. The file sdf.tab is not recommended because it is difficult to create and causes VCS to run slower. `+maxdelay`s tells VCS to use the MAX delays in the SDF file. Other options are `+typdelays` or `+mindelays`. `+neg_tchk` depends on the specific SDF or library.

For Verilog-XL:

The `+compsdf` argument is not needed because Verilog-XL is interpreted. Verilog-XL executable must be compiled with the SDF annotator included. It does not come this way on the installation CD from Cadence. However, most sites have built it this way.

Check the log file to make sure the annotation takes place. The biggest source of errors is the hierarchical reference in `$LX_HOME/system/control.v`. Refer to Verilog-XL or VirSim for the correct reference. For example, to back annotate an lx1 SDF file, the control file is modified as shown below.

```
*************************** control.v ******************************
module control;
initial
  begin
      $display ("Annotating topsys.lx_base.lx2.lx1 using %s",
                                        ("+mindelays":"+typdelays":"+maxdelays"));
      $sdf_annotate("lx1.sdf",topsys.lx_base.lx2.lx1, , "lx1_sdf.log",
                                        "TOOL_CONTROL", , );
  end
endmodule
*************************** control.v ******************************
```

To simulate "at speed", modify $LX_HOME/include/lxr_symbols.vh.

For example,

`define SYSCLK_PERIOD 8 (8ns clock period = 125MHz)
(the default is SYSCLK_PERIOD 50 => 50ns period = 20MHz).

12.7 Asynchronous-mode LBC

When using the LBC in asynchronous mode, simulating at the gate-level with post layout timing annotation and synchronizing flip-flops in the LBC generates setup time violations. When a timing violation occurs, most flip flop models produce unknown values, which corrupt the simulation. This is unavoidable because of the asynchronous relationship between the system clock and the bus clock. This problem can be neutralized by copying the flip flop model to a new cell, disabling the unknown value generated on a timing violation, and replacing the synchronizing instances with the new cell.

12.8 Runtime Limitations

When running regression tests, run only one simulation at a time per regression directory.

Do not rerun lconfig during a simulation. Otherwise, the simulation may fail since `Rundvt` may recompile the models to run certain tests. For simultaneous simulations, create a separate directory for each simulation run.

`Lconfig` may report synthesis setup errors if the technology files have not been completely setup, but RTL simulation may still work.