cadence®

Xtensa®
Instruction Set Simulator (ISS)

User's Guide

Cadence Design Systems, Inc.
2655 Seely Ave.
San Jose, CA 95134
www.cadence.com

Issue Date:10/2018

RI-2018.0

Cadence Design Systems, Inc.
2655 Seely Ave.
San Jose, CA 95134
www.cadence.com

Contents

List of Tables

Preface

This guide is written for customers who are experienced in the programming and debugging of software.

The applicable keywords for this guide are: simulation, multiprocessor, debugging, ISS.

Notation

- *italic_name* indicates a program or file name, document title, or term being defined.
- $ represents your shell prompt, in user-session examples.
- **`literal_input`** indicates literal command-line input.
- *`variable`* indicates a user parameter.
- `literal_keyword` (in text paragraphs) indicates a literal command keyword.
- `literal_output` indicates literal program output.
- *`... output ...`* indicates unspecified program output.
- *`[optional-variable]`* indicates an optional parameter.
- **`[`***`variable`***`]`** indicates a parameter within literal square-braces.
- **`{`***`variable`***`}`** indicates a parameter within literal curly-braces.
- **`(`***`variable`***`)`** indicates a parameter within literal parentheses.
- | means *OR*.
- *`(var1|var2)`* indicates a required choice between one of multiple parameters.
- *`[var1|var2]`* indicates an optional choice between one of multiple parameters.
- *`var1 [, varn]*`* indicates a list of 1 or more parameters (0 or more repetitions).

Terms

- *0x* at the beginning of a value indicates a hexadecimal value.
- *b* means bit.
- *B* means byte.
- *Mb* means megabit.
- *MB* means megabyte.
- *PC* means program counter.
- *word* means 4 bytes.

Changes from the Previous Version

1. Introduction

The Xtensa Instruction Set Simulator (ISS) is a standard component of the Xtensa software development tools package. You can use the simulator to test and debug your software, and analyze and tune the performance of your applications before committing to the fabrication of a large SoC. This introduction provides an overview of the Xtensa ISS features that can help you during the software development process.

1.1 Stand-Alone Mode

In the most basic usage, you can invoke the ISS as a stand-alone program to simulate the execution of an application targeted for the Xtensa processor. You can do this either from the command line, using the `xt-run` command, or from the Xtensa Xplorer, using the Run dialog.

When requested, the ISS provides the performance summary output, which includes cycle counts for various events that occurred during the simulated program execution. The simulator can also generate execution traces, as well as the target program profiling data, which can be analyzed with the `xt-gprof` profiler or the Xtensa Xplorer. Comparisons of profiles from different Xtensa processor configurations can help you select an optimal set of configuration options.

You can debug an Xtensa target program using either the GNU command-line debugger (`xt-gdb`) or the Xtensa Xplorer debugger. By default, the debugger automatically runs the simulator to execute the target program.

The ISS also includes a command-loop mode for an interactive, low-level access to the simulator. This mode is similar to the command-line debugger interface, with two main differences: the command loop does not provide symbolic information about the target program, but it can give more low-level details about the simulated processor state.

The basic ISS operation is extended through client packages (or simply clients) that supply added functionality on demand. The ISS clients are used for various tasks, such as program execution tracing, application profiling and memory checking. The behavior of each client package can be controlled by command-line options, or by the client command mechanism, which allows for changes during the simulated program execution.

1.2 System-Level Modeling with XTSC

Increasingly complex system-on-chip (SoC) designs often contain multiple processors and various other devices. For system-level modeling, Cadence provides an application programming interface to the ISS — Xtensa SystemC package (XTSC). You can use XTSC to write your own customized, multi-threaded simulators to model complex SoCs. This allows you to develop, debug, profile, and verify combined hardware and software systems early in the design process.

XTSC provides the SystemC model of the Xtensa processor and a well-defined set of TLM (transaction-level modeling) interfaces for communication with SystemC models of other SoC components. For details about XTSC, see the *Xtensa SystemC (XTSC) User's Guide*.

Note: XTSC is an optional feature of the Xtensa Software Development Toolkit, and is available only to users who have purchased the system simulation option.

1.3 Cycle-Accurate Versus Fast Functional Simulation

The Xtensa ISS has two major simulation modes. In the default cycle-accurate simulation mode, rather than executing one instruction at a time the ISS directly simulates multiple instructions in the pipeline, cycle by cycle. The simulator models most micro-architectural features of the Xtensa processor and maintains a very high degree of cycle accuracy. In this mode, the ISS can be used as a reference model for a detailed hardware simulation, thus serving as a verification tool.

The second major simulation mode is the fast functional simulation, or *TurboXim*. In the TurboXim mode, the ISS does not model micro-architectural details of the Xtensa processor, but it does perform architecturally correct simulation of all Xtensa instructions and exceptions. For long-running programs, the fast functional simulation can be 40 to 80 times faster than the cycle-accurate simulation, which makes it ideally suited for high-level functional verification of your application software.

In the TurboXim mode, cycles lost due to pipeline stalls and replays, memory delays, and branch penalties are not taken into account, and each instruction is assumed to take a single cycle to complete. However, you can combine the fast functional mode with the cycle-accurate mode in the same simulator invocation; this allows you to generate very accurate profiles for long-running programs at speeds much faster than the cycle-accurate simulation can provide.

Note: TurboXim is an optional feature of the Xtensa Software Development Toolkit and is available only to users who have purchased the accelerated simulation option.

2. Basic Usage

This chapter describes the simulator invocation and basic functionality. It also provides some details about important simulator features: semi-hosting, fast functional simulation, and simulator client services used in common software development tasks.

2.1 Running the Simulator

You can run the simulator in the Xtensa Xplorer by using the `Run`, `Debug`, or `Profile` dialog. For more information, refer to the Xtensa Xplorer documentation.

Outside of the Xtensa Xplorer, you can run the simulator in one of the three execution modes: command-line, debugger-backend, or command-loop. This section provides an overview of the first two modes, while the command-loop mode is described in detail in Chapter 4.

2.1.1 Command-Line Mode

You can invoke the simulator from your Linux or Windows shell using the `xt-run` command as follows:

```
xt-run [simulator-options] target-program [target-arguments]
```

The simulator command-line options are described in Chapter 3. Target program is an ELF-format executable generated by the Xtensa Tools compiler, assembler and linker. The name of the target program is optionally followed by its own command-line arguments.

For example, assuming that the directory where you have installed the simulator is in your path, and assuming that your target program is:

```
main()
{
  printf("Hello world\n");
}
```

and assuming that you have used the Xtensa Tools to compile and link this program into an executable named `hello`, you would run the simulator on the target program with a command such as the following:

```
$ xt-run hello
Hello world
$
```

If you include `--summary` (page 50) or `--mem_model` (page 31) on your command line, the simulator displays a summary of its run (see Chapter 6 for the full description of the summary output). For example:

```
$ xt-run --summary hello
Hello world

Xtensa Core: "DC_B_212GP"  ISS Version: 10.0.0

Time for Simulation = 0.01 seconds

Current PC = 0x60005b4b

Events                            Number  Number
                                          per 100
                                          instrs

Committed instructions             3870 ( 100.00 )
Taken branches                      513 (  13.26 )
Exceptions                           29 (   0.75 )
   WindowOverflow                    15 (   0.39 )
   WindowUnderflow                   14 (   0.36 )
Loads                               428 (  11.06 )
Stores                              272 (   7.03 )

Cycles: total = 8030
                                          Summed |           Summed
                                  CPI      CPI   |% Cycle  % Cycle
Committed instructions    3870 ( 1.0000   1.0000 |  48.19    48.19 )
Taken branches            1107 ( 0.2860   1.2860 |  13.79    61.98 )
Pipeline interlocks        113 ( 0.0292   1.3152 |   1.41    63.39 )
Exceptions                 145 ( 0.0375   1.3527 |   1.81    65.19 )
Sync replays              2180 ( 0.5633   1.9160 |  27.15    92.34 )
Special instructions       606 ( 0.1566   2.0726 |   7.55    99.89 )
Loop overhead                4 ( 0.0010   2.0736 |   0.05    99.94 )
Reset                        5 ( 0.0013   2.0749 |   0.06   100.00 )
```

To run the ISS in the fast functional simulation mode, use the `--turbo` command-line option **(page 59)**. Because the simulator running in the TurboXim mode assumes that each instruction takes a single cycle, the performance summary output will be less verbose:

```
$ xt-run --turbo --summary hello
Hello world

Xtensa Core: "DC_B_212GP"  TurboXim Version: 9.0.2

Time for Simulation = 0.01 seconds (+ 0.55 seconds for Initialization)

Current PC = 0x60005b4b

Events                            Number  Number
                                          per 100
                                          instrs

Committed instructions             3870 ( 100.00 )
Taken branches                      513 (  13.26 )
Exceptions                           29 (   0.75 )
   WindowOverflow                    15 (   0.39 )
   WindowUnderflow                   14 (   0.36 )
```

2.1.2 Debugger-Backend Mode

The `xt-gdb` command-line debugger automatically launches the simulator to run your Xtensa program (unless you explicitly request another target). For example, a very simple debugging session may look similar to the following:

```
$ xt-gdb hello
GNU gdb 7.1 Xtensa Tools 9.0.2
... copyright and configuration notice ...
(xt-gdb) run
Starting program: hello
Starting the ISS simulator.
Remote debugging using localhost:16572
Hello world

Program exited normally.
(xt-gdb) quit
$
```

To specify the simulator command-line options in the debugger, use the `target iss` command before entering the `run` command. For example, a debugging session in which you run the ISS in the fast functional simulation mode would look similar to the following:

```
$ xt-gdb hello
GNU gdb 7.1 Xtensa Tools 9.0.2
... copyright and configuration notice ...
(xt-gdb) target iss --turbo
```

```
Connected to the simulator.
(xt-gdb) break main
(xt-gdb) run
Starting program: hello
Starting the ISS simulator.
Remote debugging using localhost:52932

Breakpoint 1, main() at hello.c:7
(xt-gdb) iss mode

        Simulation mode: turbo (fast functional)

(xt-gdb) continue
Hello world

Program exited normally.
(xt-gdb) quit
$
```

You can invoke command-loop commands (Chapter 4) from the debugger-backend mode by issuing the `iss` *cmd* command in `xt-gdb` (*cmd* is any command-loop command). In this example, the command-loop `mode` command is used to print the current simulation mode.

For the full description of `xt-gdb` commands, see the *GNU Debugger User's Guide*.

2.2 Semi-Hosting SIMCALL Interface

The Xtensa Instruction Set Architecture (ISA) includes the `SIMCALL` instruction, which can be used only in the ISS. Execution of a `SIMCALL` instruction is a request to the simulator to perform system services for the target application using the host operating system. Through this mechanism the ISS provides a semi-hosting system call library, which allows you to execute reasonably complex programs without a target operating system. For the complete list of services provided by the ISS semi-hosting library, see the System Calls chapter in the *Red Hat newlib C Library Reference Manual*.

The `SIMCALL` instruction interface is also used to dynamically modify the ISS behavior from the simulated target program. Two important examples of this are switching between the cycle-accurate and fast functional simulation during a single ISS invocation, and modifying the operation of simulator client packages by passing commands to them from the target application.

2.3 Fast Functional Simulation (TurboXim)

As mentioned in Section 1.3, the default mode in the Xtensa ISS is the cycle-accurate simulation. However, at various stages of the application program development, this level of accuracy may not be necessary, and the simulation speed may be more important. To help you expedite the software development process and facilitate rapid functional verification of your programs, the simulator provides a fast-functional simulation mode or TurboXim (see the `--turbo` command-line option on page 59). For long-running programs (100 million instructions or more), the TurboXim mode can be 40 to 80 times faster than the cycle-accurate simulation.

In the fast functional simulation mode, the ISS faithfully models the architectural effects of Xtensa instructions and exceptions, but it does not take into account pipeline stalls, cache misses, memory latencies and branch penalties — each instruction is modeled as taking just one cycle to complete.

2.3.1 Simulation Mode Switching

The TurboXim mode and the cycle-accurate simulation mode are fully interoperable, and the ISS provides several mechanisms for switching between the two modes.

By using the `--sample` command-line option (page 47), you can instruct the simulator to automatically and periodically switch between modes. This is most useful for performance profiling of long-running applications, since it allows you to generate very accurate profiling data at very high simulation speeds.

In an interactive simulation session, you can change the simulation mode from the command loop with the `mode` command (page 96), or from the `xt-gdb` debugger with the `iss mode` command.

Finally, you can request the simulation mode switch programmatically from your Xtensa target application. The `SIMCALL`-based interface for communication between the target program and the simulator includes the function

```
int xt_iss_switch_mode(int mode);
```

whose prototype is provided in the `<xtensa/sim.h>` header file. The `mode` value `XT_ISS_CYCLE_ACCURATE` switches to the cycle-accurate modeling, and the `mode` value `XT_ISS_FUNCTIONAL` switches to the fast functional simulation. The non-zero return value is an indication that the simulation mode could not be changed. For example, this can happen if the `--sample` command-line option is specified — automatic and program-directed simulation mode switching are incompatible.

2.3.2 TurboXim Caveats

Because the ISS counts one cycle per instruction in the fast functional mode, the reported cycle count will be smaller than in the cycle-accurate mode. The target program can access this cycle information through the use of the `times()` system call or by reading `CCOUNT` special registers. If the application behavior depends on the cycle count values (for example, as is the case with timer interrupts), then its execution may take different paths in the two simulation modes.

In the TurboXim mode, the simulator does not model instruction or data caches. These can be architecturally visible when cache-locking is used. If cache locking is used, the data locked in the cache must be the same as the memory location it represents.

To correctly simulate a program when switching to the fast functional mode when the write-back data cache is dirty, the simulator will write back any dirty cache lines before it executes the next instruction in the TurboXim mode. Before switching back to the cycle-accurate mode, the caches are invalidated. Thus, the first instructions executed in the cycle-accurate mode after a switch will have higher-than-normal cache misses. When measuring the performance of an application through sampling, the number of instructions executed in the cycle-accurate mode should be large enough to mitigate these effects.

2.4 Clients Overview

The ISS client packages are designed to facilitate common software development tasks, such as various aspects of debugging, and application performance analysis and tuning. You can request the simulator to load a client library by using the `--client` (page 14) or `--client_commands` (page 15) command-line option. These options may also include client arguments that allow you to configure client behavior at the time you invoke the simulator.

Some clients can also accept run-time commands for dynamic control during the simulation. In an interactive simulation session, you can send a command to a client package from the command loop by using the `client` command (page 77), or from the `xt-gdb` debugger or the Xtensa Xplorer GDB Console by using the `iss client` command. In addition, you can issue client commands programmatically through the `SIMCALL`-based function

```
int xt_iss_client_command(const char *client, const char *command);
```

In the following summary of client functionality, clients are grouped based on the tasks they are designed to address:

- program tracing: `pchistory`, `trace`
- application profiling: `isa_profile`, `profile`, `stackuse`, `summary`
- memory error checking: `ferret`
- binary image loading: `loadbin`.

For details on all the ISS clients, including their command-line arguments and dynamic commands, refer to Chapter 5.

2.5 Virtual Breakpoints – Unhandled Exceptions

When a simulated program encounters an unhandled exception, the simulator may report a virtual breakpoint. The word *virtual* does not mean that this is a breakpoint on a virtual address; it is better understood as a `SIMCALL`-based breakpoint. If the program is running under `xt-gdb`, a virtual breakpoint returns control to the debugger. If not, the ISS terminates the simulation.

It takes at least a few tens of instructions before an unhandled exception in the simulated program leads to a virtual breakpoint, because the simulated processor will execute code from the exception vector. To find the underlying problem, find the last program instruction before the exception by using the `--pchistory` command-line option (page 40), which allows you to easily see the last *n* instructions executed. Alternatively, you can use the `--trace` command-line option (page 58) to trace program execution, but choosing this option may lead to very large trace files. To use these features within the Xtensa Xplorer, you can select the “Enable PC history” or the “Trace Execution” checkbox from the Advanced tab of the Run/Debug dialog.

3. Command-Line Options

The following sections describe each of the simulator's command-line options. The options affect not only simulations invoked in the command-line mode, but also simulations invoked in the command-loop mode or debugger-backend mode. The simulator also interprets any text specified in the environment variable `ISS_EXTRA_ARGS` as additional command-line arguments. Such options will prefix any options given in the traditional manner. All command-line options are POSIX-compliant, and thus require a double dash instead of a single dash.

Note: All command-line options are case-sensitive and must be lower case.

--alt_reset_vec - Set alternate reset vector address

Syntax

```
--alt_reset_vec=n
```

Description

Specify the alternate external reset vector input bus value *n* when the external reset vector sub-option of the relocatable vectors option is configured. This option sets the reset vector address to be used when the alternate static vector base is selected (**--vector=1**). See the Exception Vectors section in the *Xtensa LX Microprocessor Data Book* for details about relocatable vectors.

Note: This option does not apply to Xtensa NX processors.

Example

```
--alt_reset_vec=0x60001000
```

Default

Alternate static vector base address assigned at the processor configuration time.

Prerequisites

```
--vector=1
```

Related

```
--vector
```

--blockrepeat - Set memory block-read repeat delay

Syntax

```
--blockrepeat=n
```

Description

This is a deprecated version of `--read_repeat` (page 45).

Example

```
--blockrepeat=2
```

Default

```
--blockrepeat=0
```

Prerequisites

```
--mem_model
```

Related

```
--read_repeat
```

--client - Load multiple clients

Syntax

--client=*file*

Description

Load each client specified in the given *file*. The file can contain blank lines, which are ignored, comment lines beginning with the '#' character, and invocation entries. An entry consists of the client name followed by any client options on a single line. See Chapter 5 for more details on client names and options.

Example

--client=profile_and_trace.txt

where `profile_and_trace.txt` contains:

```
profile --instructions gmon.out
trace --level=6 trace_6.t
```

Default

No clients are explicitly loaded.

Prerequisites

None

Related

--client_commands

--client_commands - Load a single client

Syntax

```
--client_commands=client-name
--client_cmds=client-name
--client_commands="client-name[ client-option]*"
--client_cmds="client-name[ client-option]*"
```

Description

Load the specified client. The string after the equals sign consists of the client name followed by a space-separated list of client options. If any client options are specified, then the string must be quoted with double quotes. If more than one client is to be loaded, each must be specified with a separate `--client_commands` option. See Chapter 5 for more details on client names and options.

Example

```
--client_commands=ferret
--client_cmds="profile --dcmiss dcmiss.out"
```

Default

No clients are explicitly loaded.

Prerequisites

None

Related

```
--client
```

--cmdinit - Execute command-loop commands on startup

Syntax

```
--cmdinit=file
```

Description

Execute command-loop commands from the given *file* immediately after entering command-loop mode. This can be useful for establishing custom aliases. The option has no effect unless the `--cmdloop` (page 17) option is also given.

Example

```
--cmdinit=my_aliases
```

Default

No command-loop commands are executed upon entering command-loop mode.

Prerequisites

```
--cmdloop
```

Related

None

--cmdloop - Start in the command-loop mode

Syntax

```
--cmdloop
```

Description

Enable a command loop at the end of simulator initialization. In the command loop, the simulator prompts you for command-loop commands. The command loop is described in detail in Chapter 4.

Example

```
--cmdloop
```

Default

No command loop.

Prerequisites

None

Related

```
--cmdinit
```

--cycle_limit - Set the simulation cycle limit

Syntax

```
--cycle_limit=n
```

Description

Set a limit on the number of cycles the simulator is to run. When this limit is reached, the simulation terminates.

Example

```
--cycle_limit=1000000
```

Default

No limit.

Prerequisites

None

Related

None

--dcline - Set data cache line size

Syntax

```
--dcline=n
```

Description

Set the line size of the data cache to *n* bytes, where *n* is a power of 2 between 16 and 256, but no greater than 16 times the byte width of the external bus interface. The option has no effect if the data cache is not configured.

Note: For an Xtensa NX processor configured with the vector cache option, `--dcline` applies to the scalar (L1S) data cache.

Example

```
--dcline=32
```

Default

`--dcline=`*n* where *n* is the configured line size of the data cache.

Prerequisites

```
--mem_model
```

Related

```
--dcsize
--dcways
```

--dcsize - Set data cache size

Syntax

--dcsize=*n*

Description

Set the size of the data cache to *n* bytes. *n* divided by the data cache associativity (set by `--dcways`) must be a power of 2. The option has no effect if the data cache is not configured.

Note: For an Xtensa NX processor configured with the vector cache option, `--dcsize` applies to the scalar (L1S) data cache.

Example

--dcsize=16k

Default

--dcsize=*n* where *n* is the configured size of the data cache.

Prerequisites

--mem_model

Related

--dcline
--dcways
--vecdcsize

--dcways - Set data cache associativity

Syntax

```
--dcways=n
```

Description

Set the associativity of the data cache to *n* ways. Supported values are 1,2,3,4. The data cache size divided by *n* must be a power of 2. The option has no effect if the data cache is not configured.

Note: For an Xtensa NX processor configured with the vector cache option, `--dcways` applies to the scalar (L1S) data cache.

Example

```
--dcways=2
```

Default

`--dcways=`*n* where *n* is the configured associativity of the data cache.

Prerequisites

```
--mem_model
```

Related

```
--dcline
--dcsize
```

--exit_location - Set exit address

Syntax

```
--exit_location=address
```

Description

Set the virtual *address* used to exit the target program. Back-to-back writes to this location will terminate the target program execution. This option allows you to simulate hardware diagnostic tests built with an LSP (Linker Support Package) other than `sim`, `sim-local`, or `sim-rom`. See the “Running C Programs on the Reference Test Bench” section in the *Xtensa Hardware User’s Guide* for more details.

Example

```
--exit_location=0x60000000
```

Default

No exit location.

Prerequisites

None

Related

None

--exit_with_target_code - Return target exit code

Syntax

```
--exit_with_target_code
```

Description

Return the exit status of the target program. Without this option, the exit code of the simulator will always indicate success, unless there was an error during the simulation. Note that the target exit status is available only after the target program has terminated.

Example

```
--exit_with_target_code
```

Default

The exit code indicates the status of the simulation, not of the target program.

Prerequisites

None

Related

None

--help - Display command-line options

Syntax

```
--help
```

Description

Display a list of the command-line options with summary descriptions and immediately exits.

Example

```
--help
```

Default

No help.

Prerequisites

None

Related

None

--icline - Set instruction cache line size

Syntax

```
--icline=n
```

Description

Set the line size of the instruction cache to *n* bytes, where *n* is a power of 2 between 16 and 256, but no greater than 16 times the byte width of the external bus interface. The option has no effect if the instruction cache is not configured.

Example

```
--icline=32
```

Default

`--icline=`*n* where *n* is the configured line size of the instruction cache.

Prerequisites

```
--mem_model
```

Related

```
--icsize
--icways
```

--icsize - Set instruction cache size

Syntax

--icsize=*n*

Description

Set the size of the instruction cache to *n* bytes. *n* divided by the instruction cache associativity (set by `--icways`) must be a power of 2. The option has no effect if the instruction cache is not configured.

Example

--icsize=16384

Default

--icsize=*n* where *n* is the configured size of the instruction cache.

Prerequisites

--mem_model

Related

--icline
--icways

--icways - Set instruction cache associativity

Syntax

--icways=*n*

Description

Set the associativity of the instruction cache to *n* ways. Supported values are 1,2,3,4. The instruction cache size divided by *n* must be a power of 2. The option has no effect if the instruction cache is not configured.

Example

--icways=2

Default

--icways=*n* where *n* is the configured associativity of the instruction cache.

Prerequisites

--mem_model

Related

--icline
--icsize

--lbsize - Set loop buffer size

Syntax

--lbsize=*n*

Description

Set the size of the L0 loop buffer to *n* bytes. *n* must be a power of 2 between 16 and 256, but no less than 4 times the instruction fetch byte width.

Note: This option does not apply to Xtensa NX processors.

Example

--lbsize=64

Default

--lbsize=*n* where *n* is the configured size of the L0 loop buffer.

Prerequisites

--mem_model

--load - Load Xtensa ELF binary file

Syntax

```
--load=file
```

Description

Specify an Xtensa ELF binary *file* to load before loading the target program. The option can be repeated if multiple loads are needed. It allows you to load interrupt handlers, reset code, and other such programs before loading the target program.

Example

```
--load=user.exe --load=kernel.exe
```

This loads two exception handlers, one for user mode and one for kernel mode.

Default

Only the target program is loaded.

Prerequisites

None

Related

```
--loadbin
--noload
```

--loadbin - Load a binary image

Syntax

```
--loadbin=file@address
```

Description

Load a raw binary image from the specified *file* into the target program virtual memory at the given *address*. A raw image is a contiguous sequence of data that does not contain any auxiliary information such as size, symbols, or checksums, which are usually embedded in popular file formats.

This option can be specified multiple times. Its functionality is implemented by the `loadbin` client package (Section 5.3).

Example

```
--loadbin=my_reset_vector.raw@0x80001000
```

Loads the entire contents of the file `my_reset_vector.raw` into memory starting at virtual address `0x80001000`.

Default

No image is loaded.

Prerequisites

If the MMU option is configured, a mapping for the given address must exist at processor reset.

Related

```
--load
--noload
```

--mem_model - Simulate memory subsystem

Syntax

```
--mem_model
```

Description

Enable the simulation of internal memories (instruction and data caches, RAMs, and ROMs) and a parameterized simple delay model of system memories. The simulator will model cache hits and misses, local memory bank and sub-bank conflicts, and external bus delays. At the end of simulation, it will display a performance summary.

Note: This option has no effect when the ISS is in the fast functional simulation mode.

Example

```
--mem_model
```

Default

Without this option, the simulator does not model caches, local memory banks and sub-banks, or external bus delays. It assumes instant accesses for all memory operations.

Prerequisites

None

Related

```
--dcline
--dcsize
--dcways
--icline
--icsize
--icways
--read_delay
--read_repeat
--vecdcsize
--wbsize
--write_delay
--write_repeat
```

--meminit - Set uninitialized memory value

Syntax

```
--meminit=n
```

Description

Specify a 32-bit number *n* to be used as the default value for uninitialized memory locations. The default is `0xe8e8e8e8`, and recognizing this pattern can help you identify uninitialized memory bugs in your target program. In previous releases, all memory locations were zero-initialized. To get the same behavior, use `--meminit=0`.

Example

```
--meminit=0xa5a5a5a5
```

Default

```
--meminit=0xe8e8e8e8
```

Prerequisites

None

Related

None

--memlimit - Set memory limit

Syntax

```
--memlimit=n
```

Description

Specify an upper bound of *n* megabytes on host memory allocated by the simulator during the target program execution. The simulator's memory requirement is approximately 3 times the target application's footprint. There should be more than *n* megabytes of swap space available on the machine hosting the simulation.

Example

```
--memlimit=512
```

Default

```
--memlimit=256
```

Prerequisites

None

Related

None

--mlatency - Set memory read delay

Syntax

```
--mlatency=n
```

Description

This is a deprecated version of `--read_delay` (page 44).

Example

```
--mlatency=2
```

Default

```
--mlatency=0
```

Prerequisites

```
--mem_model
```

Related

```
--read_delay
```

--no_debug_flush - Do not flush pipeline on break

Syntax

```
--no_debug_flush
```

Description

When the ISS running in the debugger-backend mode encounters a breakpoint or a watchpoint, it flushes the pipeline in a way similar to the behavior of real Xtensa hardware. In addition, in order for the user-visible state to be updated appropriately, the core is cycled until instructions in the commit stage and beyond are completed before the breakpoint is taken. Instructions in stages prior to the commit stage are killed. The store and write buffers are processed and emptied.

Upon resumption, fetch processing begins anew, and it takes several cycles before the pipeline fills to the commit stage. Thus, a breakpoint will necessarily cause the simulation to proceed somewhat differently from the case where no breakpoints are encountered. In particular, the number of cycles executed will be greater in the former case.

You can use the `--no_debug_flush` option to change this behavior. This option precludes the usual pipeline flush, and on resumption, the fetch engine continues precisely where it left off. However while the simulator is stopped, register and state values might not be consistent since they are in the midst of being processed through the pipeline. Similarly, updates to memory may not yet be visible because the store and write buffers are not flushed.

Note: This option has no effect when the ISS is in the fast functional simulation mode.

Example

```
--no_debug_flush
```

Default

The pipeline is flushed by default when a breakpoint is encountered.

Prerequisites

None

Related

None

--no_zero_bss - Disable early initialization of .bss segment

Syntax

```
--no_zero_bss
```

Description

This option instructs the simulator not to zero initialize the `.bss` segment when loading the target program (Xtensa ELF binary). It is primarily intended for the case where the MMU option is configured and the `.bss` segment is mapped dynamically. If you use this option, your software must initialize the `.bss` segment.

Note that on real hardware, the `.bss` segment must always be initialized by software.

Example

```
--no_zero_bss
```

Default

The simulator initializes the `.bss` segment while loading the target program.

Prerequisites

None

Related

None

--noload - Disable loading of the target program

Syntax

```
--noload
```

Description

This option prevents the simulator from loading the target program. It is useful when you want to initialize all the system memories by using `--load` (page 29) and `--loadbin` (page 30) options.

Example

```
--noload
```

Default

The target program is loaded by default.

Prerequisites

None

Related

```
--load
--loadbin
```

--noreset - Skip reset vector

Syntax

```
--noreset
```

Description

This option causes the program execution to begin at the target application's start address specified in the ELF file header (see the `-e`/`--entry` linker flag or the `ENTRY` linker script command in the *GNU Linker User's Guide* for more information). The standard hardware reset as specified by the Xtensa Instruction Set Architecture (ISA) is still performed. This option should be used with care as critical initialization code could be skipped.

Example

```
--noreset
```

Default

Begin execution at the reset vector.

Prerequisites

None

Related

None

--nosummary - Disable simulation summary

Syntax

```
--nosummary
```

Description

Disable summary reporting at the completion of each run.

Example

```
--nosummary
```

Default

The summary is printed when the simulator is invoked with the `--mem_model` option.

Prerequisites

None

Related

```
--mem_model
--summary
```

--pchistory - Print the last *n* execution addresses

Syntax

```
--pchistory=n
```

Description

Print the PCs for the last *n* instructions executed. See the `pchistory` client package description in Section 5.4 for details.

Example

```
--pchistory=100
```

Default

PC history is not displayed.

Prerequisites

None

Related

```
--client
--client_commands
--trace
```

--prefetch - Set prefetch control register

Syntax

```
--prefetch=n
```

Description

Set the value of the processor's special register PREFCTL to *n*. Bits [3:0] of *n* specify the DataCtl field, bits [7:4] specify the InstCtl field, bit [12] specifies the DL1 field, and bits [19:16] specify the BlockCtl field of PREFCTL (see the Prefetch Unit Option chapter in the *Xtensa Microprocessor Data Book* for details). Valid values for DataCtl and InstCtl are: 0, 1, 3, 4, 5, and 8; valid values for DL1 are 0 and 1; valid values for BlockCtl are 0 through 9.

For Xtensa NX processors, only the setting of lower 8 bits (fields DataCtl and InstCtl) is taken into account. Upper bits are always treated as being hard-wired to DL1 = 1 and BlockCtl = 0.

Value specified by --prefetch is used to control hardware prefetching during the entire simulation, overriding any target program updates of the PREFCTL special register.

Example

```
--prefetch=0x61044
```

Default

Value of PREFCTL written by the target program is used.

Prerequisites

```
--mem_model
```

Related

None

--proc_id - Specify processor ID

Syntax

```
--proc_id=n
```

Description

Set the value of the processor's special register PRID to the 16-bit quantity *n*. In hardware, the value of the PRID register is sampled from external pins at reset. In either case, the value is held constant after reset.

Example

```
--proc_id=0xabcd
```

Default

Indeterminate

Prerequisites

None

Related

None

--profile - Enable profiling

Syntax

```
--profile=file
```

Description

Dump profiling data to the given *file* in a format suitable for `xt-gprof` or the Xtensa Xplorer profiler. This includes the call-graph information and either cycle or instruction counts. In the cycle-accurate or the sampling mode, execution cycles are profiled; in the fast functional mode (with `--turbo`, but without `--sample`) only instructions are counted.

See the `profile` client package description in Section 5.5 for details, as well as the *GNU Profiler User's Guide*.

Example

```
--profile=gmon.out
```

Default

No profiling.

Prerequisites

None

Related

```
--client
--client_commands
--sample
```

--read_delay - Set memory read delay

Syntax

```
--read_delay=n
```

Description

Specify the number of wait state cycles required for a system memory to respond to a single read request or send the first response transfer for a block-read or burst-read request. This option is applicable only when `--mem_model` (page 31) is specified.

The default value is zero wait states, which implies that the memory responds in one cycle. This does not mean that an uncached access or a cache miss will complete in one cycle, since the simulator also models bus delays and instruction replays due to external bus accesses.

Note: The ISS system memory model assumes that there is sufficient request buffering to allow overlapping of delays for multiple requests. For example, with a read delay of 10 wait cycles, a read request issued in cycle 0 will get a response in cycle 11. If a second read request is issued in cycle 2, its delay will be partially overlapped with the wait cycles of the first request, and the second response will arrive in cycle 13.

Example

```
--read_delay=2
```

Default

```
--read_delay=0
```

Prerequisites

```
--mem_model
```

Related

```
--read_repeat
--write_delay
--write_repeat
```

--read_repeat - Set memory block-read repeat delay

Syntax

```
--read_repeat=n
```

Description

Specify the number of wait state cycles required for a system memory to return each additional bus-width transfer during a block-read or burst-read transaction. This option is applicable only when `--mem_model` (page 31) is specified.

The default value is zero wait states, which implies that each transfer completes one cycle after the previous one.

Example

```
--read_repeat=2
```

Default

```
--read_repeat=0
```

Prerequisites

```
--mem_model
```

Related

```
--read_delay
--write_delay
--write_repeat
```

--reference - Simulate TIE references

Syntax

```
--reference
```

Description

If a TIE instruction is described both in a reference (or operation) section and in a semantic section, the semantic description is used in the simulation. With the `--reference` option, the simulator will use the reference (or operation) description instead. This can help you verify the intended behavior of TIE instructions, and it usually improves the simulation speed.

For cycle-accurate simulation of user TIE instruction references, the TIE compiler must be invoked with `-reflib`. If not, a warning will be generated and TIE semantics will be simulated. For fast functional simulation of TIE instruction references, if the TIE compiler was not invoked with `-reflib`, there can be a significant simulation startup overhead.

For details about reference, operation and semantic sections for TIE instructions, see the *Tensilica Instruction Extension (TIE) Language Reference Manual*.

Example

```
--reference
```

Default

Simulate TIE semantics.

Prerequisites

None

Related

None

--sample - Switch simulation mode periodically

Syntax

```
--sample
```

Description

Automatically and periodically switch between the cycle-accurate and fast functional simulation modes. This sampling is most useful for generating fast and accurate profiles of long-running programs.

In the sampling mode, by default, the simulator will alternate between executing 1,000,000 instructions in the cycle-accurate mode, and 100 times that many in the fast functional mode. The first number can be changed with the `--sample_insns` option (page 48), and the second number can be changed with `--sample_ratio` option (page 49).

Without the `--turbo` option (page 59), the ISS will start in the cycle-accurate mode; with the `--turbo` option, it will start in the fast functional mode.

The `--sample` option cannot be used in the command-loop or the debugger-backend mode.

While in the automatic sampling mode, the ISS ignores target-program requests to switch the simulation mode — calls to `xt_iss_switch_mode()` will fail.

Example

```
--sample
```

Default

No automatic simulation-mode switching.

Prerequisites

None

Related

```
--profile
--sample_insns
--sample_ratio
--turbo
```

--sample_insns - Set cycle-accurate instruction sample

Syntax

```
--sample_insns=n
```

Description

Specify the number of instructions that will be executed in the cycle-accurate mode when the simulator is automatically switching modes (`--sample`, page 47). The ISS will simulate *n* instructions in the cycle-accurate mode, and, by default, 100**n* instructions in the fast functional mode. You can change this ratio with the `--sample_ratio` option (page 49).

Example

```
--sample_insns=500000
```

Default

```
--sample_insns=1000000
```

Prerequisites

```
--sample
```

Related

```
--sample
--sample_ratio
```

--sample_ratio - Set functional to accurate sample ratio

Syntax

```
--sample_ratio=n
```

Description

Specify the ratio of instructions that will be executed in the fast functional mode to those in the cycle-accurate mode when the simulator is automatically switching modes (`--sample`, page 47). By default, the ISS will simulate 1,000,000 instructions in the cycle-accurate mode, and 1,000,000*_n_ instructions in the fast functional mode. You can change the size of the cycle-accurate sample with the `--sample_insns` option (page 48).

A smaller ratio will increase the number of accurate samples taken during a program execution, but it will reduce the simulation speed. A larger number will result in the faster simulation, but it will reduce the number of instructions executed in the cycle-accurate mode.

Example

```
--sample_ratio=150
```

Default

```
--sample_ratio=100
```

Prerequisites

```
--sample
```

Related

```
--sample
--sample_insns
```

--summary - Enable simulation summary

Syntax

```
--summary
```

Description

Enable summary reporting at the completion of each run. If the simulation included instruction executed in both cycle-accurate and fast functional simulation modes, two summaries will be printed, one for the instructions executed in each mode.

For the full description of the performance summary format, see Chapter 6.

Example

```
--summary
```

Default

The summary is not printed unless the simulator was invoked with the `--mem_model` option.

Prerequisites

None

Related

```
--client
--client_commands
--nosummary
```

--target_input - Specify target program input file

Syntax

```
--target_input=file
```

Description

Specify the *file* from which the simulator is to read the target program input.

Example

```
--target_input=file1
```

Default

Input is read from the host `stdin`.

Prerequisites

None

Related

```
--target_output
```

--target_output - Specify target program output file

Syntax

--target_output=*file*

Description

Specify the *file* to which the simulator is to write the target program output. Target `stdout` and `stderr` will be combined.

Example

--target_output=file2

Default

Target program `stdout` and `stderr` are written to the host `stdout` and `stderr`, respectively.

Prerequisites

None

Related

--target_input

--tieports - Specify TIE port value file

Syntax

```
--tieports=spec-file
```

Description

Specify the *spec-file* that defines how TIE ports, queues, and lookups are simulated.

TIE ports, queues, and lookups allow you to create direct interfaces between Xtensa processors and other devices in your system. To accurately model their behavior, you should implement these interfaces as a part of an XTMP or XTSC simulation. However, for the simplified testing of the code with TIE ports you can use this option to specify how the sequences of TIE port values are read from or written to a file.

Note: You should not use `--tieports` for TIE lookups with the `lookup_memory` property. The ISS automatically models such lookups as specialized memories.

The *spec-file* contains a sequence of sections, one for each TIE port. The format of each section is either

```
tie-port-name value-file
```

where *value-file* contains one *value* per line, or

```
tie-port-name
value
value
value
...
```

In other words, values for each input TIE port can come from a separate *value-file*, or be read directly from the *spec-file* (which is more convenient when there are only a few values). For output TIE ports, only the first format is applicable, and it specifies that the TIE port values are written into the *value-file*.

Each *value* is a space-separated sequence of (hexadecimal or decimal) numbers, each of which represents a 32-bit chunk of a TIE port value, ordered from high to low bit. If the TIE port width is not a multiple of 32, the extra bits from the high-order word are masked out.

If a sequence of values is exhausted, that is, if an input TIE port is accessed more times than the number of values specified for that port, the simulator will cycle back to the beginning of the sequence. For example, if only one value is provided, that value will be used for all accesses to the corresponding TIE port.

tie-port-name is the same as the one used in connecting TIE port callbacks in XTMP. For import wires and exported states, the names are the same as those specified in TIE `import_wire` and `state export` constructs, respectively. For an input queue `INQ`, the names of the TIE ports are `INQ_PopReq`, `INQ_Empty`, and `INQ`. For an output queue `OUTQ`, the names of TIE ports are `OUTQ_PushReq`, and `OUTQ_Full`. For a TIE lookup `TABLE`, the names of TIE ports are `TABLE_Out_Req`, `TABLE_In`, and `TABLE_Rdy`.

Note: For an Xtensa NX processor, the names of an input queue `INQ` TIE ports are `INQ_Pop`, `INQ_Entries` (indicates the number of valid data entries) and `INQ`. For an output queue `OUTQ`, the names of TIE ports are `OUTQ_Push`, and `OUTQ_Entries` (indicates how many free entries are available) and `OUTQ`.

Note: Xtensa processors read TIE import wires and input queues speculatively, but even a speculative read of a *tie-port-name* consumes a value from the input sequence specified with `--tieports`. As a consequence, using more than one value for an import wire or anything other than `0` for `INQ_Empty`, may make your program timing-dependent. The program behavior may vary when you run it with or without memory modeling, or when you step through it inside the debugger.

Writes to an output queue `OUTQ` are recorded even when `OUTQ_Full` is `1`. This means that the output file for `OUTQ_PushReq` may contain repeated values whenever the input sequence of `OUTQ_Full` values includes at least one `1`.

A *spec-file* section can also have the following format that allows modeling of a ROM table accessed via a TIE lookup:

```
tie-lookup-name rom-table-file
```

where *rom-table-file* contains one <*address,value*> pair per line:

```
address value
address value
address value
...
```

tie-lookup-name is the actual name of a TIE lookup construct, and not the name of one of its ports. Each *address* is a single hexadecimal or decimal number, at most 32-bit wide.

For a TIE lookup `TABLE`, if `TABLE_Out_Req` matches an *address* specified in the *rom-table-file*, `TABLE_In` returns the corresponding *value*. If no matching *address* is found, the simulator will generate an error. To avoid this, you can specify * as an *address*, and the corresponding *value* will be used as the default mapping.

If you use the ROM table format for a TIE lookup, you should not specify value sequences for its ports, as they will be ignored. If a TIE lookup TABLE has the Rdy signal, TABLE_Rdy always returns 1.

A *spec-file*, *value-file* or *rom-table-file* line that begins with // or # is treated as a comment.

Example

--tieports=tp.txt

```
$ cat tp.txt
INQ_Empty
0
1
INQ in.txt
OUTQ_PushReq out.txt
OUTQ_Full
0
TABLE rom.txt

$ cat rom.txt
12 144
15 225
* 99
```

In this example, the input queue INQ will be empty every other time it is accessed (periodic sequence of values 0 and 1), the INQ data (when not empty) will be read from file in.txt, the OUTQ data will be written to file out.txt, and the output queue OUTQ will never be full (always 0).

Values for TIE lookup TABLE are specified in file rom.txt: when TABLE_Out_Req is 12, TABLE_In will be 144, and when TABLE_Out_Req is 15, TABLE_In will be 225. For all other values of TABLE_Out_Req, TABLE_In will be 99.

For an Xtensa NX processor:

--tieports=tp.txt

```
$ cat tp.txt
INQ_Entries
1
INQ in.txt
OUTQ_Push out.txt
OUTQ_Entries
1
```

In this example, the input queue `INQ` will always have one available entry, and the `INQ` data will be read from file `in.txt`. The `OUTQ` data will be written to file `out.txt`, and the output queue `OUTQ` will never be full (always has one free entry).

Default

TIE ports are not simulated in the command-line mode. Accessing an output TIE port generates a warning; accessing an input TIE port generates an error.

Prerequisites

None

Related

None

--tieprint - Enable TIEprint statements

Syntax

```
--tieprint[=file]
```

Description

Write the output from TIEprint statements to the specified *file*, or to the standard output if the *file* argument is ommitted. You can also combine the TIEprint output with the program execution trace by invoking the trace client package with the `--tieprint` argument (refer to page 122). For more information about TIEprint statements, see the *Tensilica Instruction Extension (TIE) Language Reference Manual*.

Example

```
--tieprint
```

```
--tieprint=tie.log
```

Default

No output from TIEprint statements is generated.

Prerequisites

None

Related

```
--trace
```

`--trace` - Enable execution tracing

Syntax

```
--trace=level
```

Description

Generate a human/machine readable execution trace. See the `trace` client package description in Section 5.8 for details.

Example

```
--trace=6
```

Default

No tracing.

Prerequisites

None

Related

```
--client
--client_commands
```

--turbo - Enable fast functional simulation

Syntax

```
--turbo
```

Description

Simulate the target program execution using the fast functional mode (TurboXim). The simulation will be much faster, but only instruction-accurate (instead of cycle-accurate).

Even without the `--turbo` option, the ISS can switch to the fast functional mode either automatically with `--sample` (page 47), or after an explicit request from the command loop, debugger, or a target program. See Section 2.3.1 for more details.

Example

```
--turbo
```

Default

Simulate in the cycle-accurate mode.

Prerequisites

None

Related

```
--sample
--sample_insns
--sample_ratio
```

--turbocache - Enable TurboXim context caching

Syntax

--turbocache=*file*

Description

Enable context caching when running in the fast functional mode. This option is useful when the same target program is executed multiple times. If the specified *file* exists and contains a valid simulation context, TurboXim performance may improve. When the simulation terminates, the *file* will be updated with the new simulation context.

Example

--turbocache=MyTurboCtxt

Default

TurboXim simulation context is not cached.

Prerequisites

--turbo

Related

--turbo

--vecdcsize - Set vector data cache size

Syntax

```
--vecdcsize=n
```

Description

Set the size of the vector data cache (L1V) to *n* bytes.

Note: This option applies only to Xtensa NX processors configured with the vector cache option.

Example

```
--vecdcsize=128k
```

Default

`--vecdcsize=`*n* where *n* is the configured size of the vector data cache.

Prerequisites

```
--mem_model
```

Related

```
--dcline
--dcsize
--dcways
```

--vector - Select static vector base

Syntax

```
--vector=n
```

Description

Specify the static vector select pin value *n*, which must be 0 for the default configured static vector base address, or 1 for the alternate. When the relocatable vectors are configured, this option selects one of the two possible locations for the reset vector and optionally the memory error vector. See the Exception Vectors section in the *Xtensa Microprocessor Data Book* for details about relocatable vectors.

Example

```
--vector=1
```

Default

```
--vector=0
```

Prerequisites

None

Related

```
--alt_reset_vec
```

--version - Display simulator version

Syntax

```
--version
```

Description

Print simulator version information and immediately exit.

Example

```
--version
```

Default

Do not print the version.

Prerequisites

None

Related

None

--wbsize - Set write buffer size

Syntax

--wbsize=*n*

Description

Set the size of the write buffer to *n* entries. *n* must be a power of 2 between 1 and 32. This option is applicable only when `--mem_model` (page 31) is specified. It has no effect if the PIF is not configured.

Note: This option does not apply to Xtensa NX processors.

Example

--wbsize=16

Default

--wbsize=*n* where *n* is the configured size of the write buffer.

Prerequisites

--mem_model

Related

None

--write_delay - Set memory write delay

Syntax

```
--write_delay=n
```

Description

Specify the number of wait state cycles required for the ISS system memory model to respond to a single write request. For a block-write or burst-write request this number represents the part of the total response delay associated with the first bus-width data transfer. This option is applicable only when `--mem_model` (page 31) is specified.

The default value is zero wait states, which means that the system memory model will respond to a single write request one cycle later.

The ISS system memory model enforces in-order request processing. Even if external write responses are not configured, the write delay value affects when the memory will be able to service a subsequent read, block-read, or burst-read request. When external write responses are configured, the write delay value also determines when a write ID is freed and made available for a subsequent write, block-write, or burst-write request.

Note: The ISS system memory model assumes that there is sufficient request buffering to allow overlapping of delays for multiple requests. For example, with read and write delays both set to 10 wait cycles, after a write request issued in cycle 0 the earliest time when the memory could respond to a subsequent read request is cycle 11. If that read request is issued in cycle 5, its delay will be partially overlapped with the wait cycles of the write request, and the read response will arrive in cycle 16.

Example

```
--write_delay=2
```

Default

```
--write_delay=0
```

Prerequisites

```
--mem_model
```

Related

```
--read_delay
--read_repeat
--write_repeat
```

--write_repeat - Set memory block-write repeat delay

Syntax

```
--write_repeat=n
```

Description

Specify the number of wait state cycles that represents the part of the total response delay associated with each additional bus-width data transfer during a block-write or burst-write transaction. This option is applicable only when `--mem_model` (page 31) is specified.

The default value is zero wait states, which means that each transfer adds one additional cycle to the total response delay.

Given `--write_delay=`*`wdelay`* `--write_repeat=`*`wrepeat`*, and a block-write transaction consisting of *`nblocks`* bus-width transfers, the total number of cycles between the start of the request and the system memory response will be

```
(wdelay + 1) + (nblocks - 1) * (wrepeat + 1)
```

Example

```
--write_repeat=2
```

Default

```
--write_repeat=0
```

Prerequisites

```
--mem_model
```

Related

```
--read_delay
--read_repeat
--write_delay
```

--xtensa-params - Specify Xtensa parameter file

Syntax

```
--xtensa-params=file
```

Description

Specify the location of the TIE Development Kit (TDK) that was produced by running the TIE Compiler (`tc`). If *`file`* identifies a directory rather than a file, the parameters are read from a file named `default_params` (if it exists in that directory). The parameter file may also be specified by setting the `XTENSA_PARAMS` environment variable. The parameters specified with this option take precedence over parameters specified with the environment variable. See the *Xtensa Software Development Toolkit User's Guide* for more information.

Example

```
--xtensa-params=./params
```

Default

No TDK is used.

Prerequisites

None

Related

None

4. Command Loop

The command loop is an internal ISS debugging facility that allows you to run the simulator interactively. In comparison with the external debugger-backend mode (`xt-gdb`), the command-loop gives you access to more detailed low-level information about the simulated processor state, but it does not provide symbolic information about the target program.

To run the ISS interactively in the command loop, simply include the `--cmdloop` option (page 17) when you invoke the simulator. The simulator will display a prompt, at which you can enter a sequence of commands to control the simulation.

4.1 Interactive Session Examples

To start the simulation in the command-loop mode, enter the `continue` command at the prompt. The default command-loop prompt shows the instruction (`I:`) and cycle (`C:`) counts. To obtain a summary of the run, use the `summary` command. The `exit` command gets you out of the command loop and exits the simulator. For example:

```
$ xt-run --cmdloop hello

     Welcome to the ISS command loop.
     The 'alias' command prints a list of default aliases.
     The 'help' command provides online help.

I:0 C:0 => continue
Hello world
I:3870 C:8030 => summary

Xtensa Core: "DC_B_212GP"  ISS Version: Xtensa 9.0.2

Current PC = 0x60005b4b

Events                            Number  Number
                                          per 100
                                          instrs

Committed instructions             3870 ( 100.00 )
Taken branches                      513 (  13.26 )
Exceptions                           29 (   0.75 )
   WindowOverflow                    15 (   0.39 )
   WindowUnderflow                   14 (   0.36 )
Loads                               428 (  11.06 )
Stores                              272 (   7.03 )
```

```
Cycles: total = 8030
                                        Summed |          Summed
                                  CPI    CPI   |% Cycle  % Cycle
Committed instructions    3870 ( 1.0000  1.0000 |  48.19    48.19 )
Taken branches            1107 ( 0.2860  1.2860 |  13.79    61.98 )
Pipeline interlocks        113 ( 0.0292  1.3152 |   1.41    63.39 )
Exceptions                 145 ( 0.0375  1.3527 |   1.81    65.19 )
Sync replays              2180 ( 0.5633  1.9160 |  27.15    92.34 )
Special instructions       606 ( 0.1566  2.0726 |   7.55    99.89 )
Loop overhead                4 ( 0.0010  2.0736 |   0.05    99.94 )
Reset                        5 ( 0.0013  2.0749 |   0.06   100.00 )

I:46859 C:68127 => exit
```

Following is a more realistic example of a command-loop session that:

1. Lists the default aliases
2. Sets a breakpoint using the default alias for the break command
3. Runs from the beginning of the program until it reaches the breakpoint
4. Requests a summary of the run
5. Dumps the current AR register window to the screen
6. Runs again from the breakpoint until it loops to the same breakpoint
7. Dumps the current AR register window again to the screen
8. Clears the breakpoint
9. Runs from the breakpoint to the end
10. Requests a summary of the run
11. Exits the simulator

```
$ xt-run --cmdloop factorial 10

     Welcome to the ISS command loop.
     The 'alias' command prints a list of default aliases.
     The 'help' command provides online help.

I:0 C:0 => alias

     a     alias
     b     break
     c     cycle
     cont  continue
     ds    dstep
     e     exit
     h     help
     hi    history
```

```
     i     info
     ip    info pipe
     ir    info reg
     is    info state
     p     prompt
     q     exit
     quit  exit
     run   continue
     s     step
     so    source
     su    summary
     u     unalias
     w     watch
     wm    write mem
     wr    write reg
     ws    write state

I:0 C:0 => b 0x600005d7

     Setting breakpoint at address 0x600005d7

I:0 C:0 => continue
I:1980 C:5294 Breakpoint at 0x600005d7
=> summary

Xtensa Core: "DC_B_212GP"  ISS Version: Xtensa 9.0.2

Current PC = 0x600005d7

Events                            Number  Number
                                          per 100
                                          instrs

Committed instructions             1980 ( 100.00 )
Taken branches                      222 (  11.21 )
Loads                                40 (   2.02 )
Stores                                7 (   0.35 )

Cycles: total = 5294
                                          Summed |          Summed
                                    CPI     CPI  |% Cycle  % Cycle
Committed instructions    1980 ( 1.0000  1.0000 |  37.40    37.40 )
Taken branches             510 ( 0.2576  1.2576 |   9.63    47.03 )
Pipeline interlocks         31 ( 0.0157  1.2732 |   0.59    47.62 )
Sync replays              2179 ( 1.1005  2.3737 |  41.16    88.78 )
Special instructions       585 ( 0.2955  2.6692 |  11.05    99.83 )
Loop overhead                2 ( 0.0010  2.6702 |   0.04    99.87 )
Reset                        5 ( 0.0025  2.6727 |   0.09    99.96 )
Breakpoints                  2 ( 0.0010  2.6737 |   0.04   100.00 )
```

```
I:1980 C:5294 => info arw

        AR window from WindowBase 3:
        AR[ 0/12] = 0xa00006d4
        AR[ 1/13] = 0x6fffff00
        AR[ 2/14] = 0x0000000a
        AR[ 3/15] = 0x00000008
        AR[ 4/16] = 0x00000000
        AR[ 5/17] = 0x6fffff59
        AR[ 6/18] = 0x00000000
        AR[ 7/19] = 0x00000000
        AR[ 8/20] = 0x00000000
        AR[ 9/21] = 0x00000000
        AR[10/22] = 0x00000000
        AR[11/23] = 0x00000000
        AR[12/24] = 0x00000000
        AR[13/25] = 0x00000000
        AR[14/26] = 0x00000000
        AR[15/27] = 0x00000000

I:1980 C:5294 => continue
I:1984 C:5310 Breakpoint at 0x600005d7
=> info arw

        AR window from WindowBase 5:
        AR[ 0/20] = 0xa00005e3
        AR[ 1/21] = 0x6ffffee0
        AR[ 2/22] = 0x00000009
        AR[ 3/23] = 0x00000000
        AR[ 4/24] = 0x00000000
        AR[ 5/25] = 0x00000000
        AR[ 6/26] = 0x00000000
        AR[ 7/27] = 0x00000000
        AR[ 8/28] = 0x00000000
        AR[ 9/29] = 0x00000000
        AR[10/30] = 0x00000000
        AR[11/31] = 0x00000000
        AR[12/ 0] = 0x00000000
        AR[13/ 1] = 0x6fffff40
        AR[14/ 2] = 0x00000002
        AR[15/ 3] = 0x6fffff40

I:1984 C:5310 => break -c 0x600005d7

        Clearing breakpoint at address 0x600005d7

I:1984 C:5310 => continue
PASS: x = 3628800
```

```
I:3187 C:7030 => summary

Xtensa Core: "DC_B_212GP"  ISS Version: 9.0.2

Current PC = 0x6000081a

Events                            Number  Number
                                          per 100
                                          instrs

Committed instructions             3187 ( 100.00 )
Taken branches                      385 (  12.08 )
Exceptions                           19 (   0.60 )
   WindowOverflow                    10 (   0.31 )
   WindowUnderflow                    9 (   0.28 )
Loads                               180 (   5.65 )
Stores                               91 (   2.86 )

Cycles: total = 7030
                                          Summed |           Summed
                                    CPI     CPI  |% Cycle  % Cycle
Committed instructions    3187 ( 1.0000  1.0000 |  45.33    45.33 )
Taken branches             851 ( 0.2670  1.2670 |  12.11    57.44 )
Pipeline interlocks         74 ( 0.0232  1.2902 |   1.05    58.49 )
Exceptions                  95 ( 0.0298  1.3201 |   1.35    59.84 )
Sync replays              2179 ( 0.6837  2.0038 |  31.00    90.84 )
Special instructions       613 ( 0.1923  2.1961 |   8.72    99.56 )
Loop overhead               10 ( 0.0031  2.1992 |   0.14    99.70 )
Reset                        5 ( 0.0016  2.2008 |   0.07    99.77 )
Breakpoints                 16 ( 0.0050  2.2058 |   0.23   100.00 )

I:3187 C:7030 => q
```

4.2 Command Input

Command-loop commands accept numerical constants in the following formats:

- Signed and unsigned decimal
- Unsigned hexadecimal
- Unsigned octal
- Unsigned binary

For better readability, you can use any number of underscore characters inside the hexadecimal, octal, and binary constants.

Following are some examples of valid numerical constants:

`45`	Signed decimal constant
`0x56, 0x2000_0004`	Hexadecimal constants preceded by `0x`
`0o77`	Octal constant preceded by `0o`
`0b10001, 0b111_0011`	Binary constants preceded by `0b`

Binary and hexadecimal constants support a zero filler feature that frees you from having to input long strings of zeros. If a decimal point is given as a part of a hexadecimal or binary constant, it expands to the number of zeros required to bring the total length of the constant to 32 bits. For example:

`0x42.1` is equivalent to `0x42000001`

`0b1.1` is equivalent to `0b1000_0000_0000_0000_0000_0000_0000_0001`

For quick computations, such as address displacements or cache line indices, the command-loop has a built-in calculator that supports `-`, `+`, `*`, `/` and `%` operations, as well as parenthesized expressions. Computation results are displayed in all the supported formats.

Multiple command-loop commands may be issued on a single line and must be delimited by semicolons. Some command-loop commands (for example, `alias`) must be the last command in the line, because the last command argument is defined as every character until the end of the line.

For example,

```
cycle; info arw
```

simulates one cycle and then prints the contents of the current address register window.

A single `Enter` issued from an empty command-line prompt executes the last command or commands given.

4.3 List of Commands

The rest of this chapter provides the complete list of command-loop commands with detailed descriptions of their usage.

alias - Define or print aliases

Syntax

alias *[-f=file] [alias-name [alias-string]]*

Description

The alias command defines *alias-name* as a substitution for *alias-string*, where *alias-string* includes every character between *alias-name* and end of the line. Therefore, alias must be the last command on the line.

When *alias-name* is met as a recognizable token, it is substituted to *alias-string*. The alias command, without parameters, prints all existing aliases. The alias command with *alias-name* only prints *alias-string* for the given *alias-name* (if it exists). The –f flag specifies the *file* into which all aliases are dumped. The source command (page 100) may be used to redefine these aliases in a future session.

An existing alias cannot be redefined but must be deleted using the unalias command (page 103); after that, a new alias may be created.

Notes

It is possible to define aliases that refer to or include other aliases. No loop checking is performed.

Examples

alias c continue	Define c as an alias for continue
alias c	Print alias-string for alias c
alias si step; info arw	Define si as an alias for “step; info arw”
alias	Print all existing aliases
alias –f=alias.dump	Dump all existing aliases into file alias.dump

break - Set, clear or print breakpoints

Syntax

```
break [-c] [address] [name]
```

Description

The `break` command (without the `-c` flag) specifies a PC at which the simulator should stop. The execution halts before the instruction at the given *`address`* is executed and the pipeline is flushed (unless you invoked the simulator with the `--no_debug_flush` command-line option). Multiple breakpoints can be set before running a simulation. Without any arguments, the `break` command prints the list of currently set breakpoints.

If you need to refer to a program location symbolically, use either the debugger or some other method (such as the disassembler) to find out the numerical values of symbols.

Optional *`name`* argument associates a descriptive label with the breakpoint and can be printed when the breakpoint is reached (see the `prompt` command on page 98). This feature simplifies debugging with multiple breakpoints, given the lack of symbolic information.

With the `-c` flag, the `break` command clears the breakpoint at the given `address` or with the given *`name`*. If neither *`address`* nor *`name`* is provided, `break -c` will clear all the breakpoints.

Examples

`break 0x2000_0040`	Set a breakpoint at address `0x2000_0040`
`break 0x2000_0040 loop`	Set a breakpoint at address `0x2000_0040` and assign name `loop` to it
`break`	Print all current breakpoints
`break loop`	Print address of breakpoint with name `loop`
`break –c 0x2000_0040`	Clear a breakpoint at address `0x2000_0040`
`break –c loop`	Clear a breakpoint named `loop`

client - Pass commands to clients

Syntax

```
client client-name client-command
```

Description

The `client` command passes a *`client-command`* string to the client *`client-name`*. See Chapter 5 for the full description of all client packages and the list of commands that each client accepts.

Notes

The `client` command must be the last command on the line.

Examples

`client profile disable`	Disable profiling
`client trace level 6`	Set the tracing level to 6
`client all dump`	Send the `dump` command to all the clients that accept it

commit - Run until an instruction commits

Syntax

```
commit PC
```

Description

The `commit` command continues the simulation until the instruction at the given *PC* reaches the commit stage in the pipeline, at which point the simulator stops.

There are two differences between `break` and `commit`. First, `break` stops the simulation before the given instruction is executed, while `commit` stops the simulation when the given instruction has already reached the commit stage. Second, at a breakpoint the simulator, by default, flushes the pipeline by letting all the instructions that precede the given PC complete, whereas `commit` does not modify the pipeline state when it stops the simulation.

Note: At the point when an instruction commits, it is possible for some of it effects not to have yet occurred.

Examples

`commit 0x2000_0040`	Run until the instruction at address `0x2000_0040` commits

continue - Continue simulation

Syntax

```
continue [number-of-cycles]
```

Description

The `continue` command starts (or continues) the simulation until the target program terminates with an `exit` system call, or until a breakpoint or a watchpoint is reached. If the *`number-of-cycles`* is specified, the simulation will continue for at most that many cycles.

Examples

`continue`	Run the simulator until the program is completed or a breakpoint occurs
`continue 100`	Same as above, but run at most 100 cycles

cycle - Advance simulation by cycle

Syntax

```
cycle [n]
```

Description

The `cycle` command continues the simulation for *n* cycles. If *n* is omitted, the default is 1 cycle.

Examples

`cycle`	Run the simulator for 1 cycle
`cycle 100`	Run the simulator for 100 cycles

dstep - Advance by instruction and flush the pipeline

Syntax

```
dstep [n]
```

Description

The `dstep` command continues the simulation until *n* more instructions are committed (if *n* is omitted, the default is 1). When the simulation stops, the pipeline is flushed, making the architectural state consistent.

Examples

`dstep 100`	Run the simulator for 100 committed instructions, and flush the pipeline
`dstep`	Run the simulator until the next instruction commits, and flush the pipeline

exit - Exit the simulator

Syntax

```
exit
```

Description

The `exit` command quits the simulator.

Examples

`exit`

help - Print help information

Syntax

help *[command]*

Description

The help command prints the detailed information about the given *command* during a command-loop session. If help is issued without command, or if *command* does not exist, it prints the list of all available commands with a brief description of each.

Notes

Aliases are not processed for *command*. For example, if c is an alias for command-loop command continue, help c will produce a parse error. help continue is the correct form of the command.

Examples

help step Print the detailed information about the step command

history - Print previously executed commands

Syntax

history *[-f=file] [n]*

Description

The `history` command prints the last *n* commands that the command-loop executed during the current session. If no parameter is given, all commands are printed. If the `-f` option is given, the commands are saved in the specified *`file`*. They can be executed again later using the `source` command.

Notes

`!`*`N`* executes the command number *`N`* from the history.

Examples

`history 5`	Print the last five commands
`!20`	Execute the command 20 from the history
`history -f=h.file`	Save all the executed commands in file `h.file`

info - Print runtime processor information

Syntax

info *resource*

resource is one of the following:

arw	Current AR register window
config	Processor configuration
dcache	Data cache
dtlb	Data TLB
icache	Instruction cache
itlb	Instruction TLB
lookup	TIE lookup memory
mem	Memory locations
pipe	Pipeline status
reg	Register files
regstage	Per-stage register files
state	Special registers and TIE states
tie	TIE wires
windows	All AR register windows

Description

The `info` command prints the runtime information about the specified *resource*. For some resources, the `info` command may need additional arguments, and those are described in the sections that follow.

Examples

info arw	Print the contents of AR registers in the current window
info config	Print the processor configuration parameters

info dtlb	Print the data TLB information
info itlb	Print the instruction TLB information
info windows	Print all the register windows and the contents of AR registers in them

info dcache - Print data cache information

Syntax

info dcache *cache-item[,cache-item]**

cache-item : (address | addr-start[word-count] | addr-low:addr-high)

Description

The `info dcache` command prints the data cache information for the given address or address range. The printed information includes the index, tag, an indication of whether the address hit or missed in the cache, the data in memory, and the data in cache.

Notes

The `info dcache` command is valid only if the data cache is configured. Addresses are aligned on 32-bit boundaries.

Examples

info dcache 0x2.8	Print contents of the data cache for address `0x2000_0008`
info dcache 0x2.8:0x8.18	Print contents of the data cache for addresses ranging from `0x2000_0008 to 0x8000_0018`

info icache - Print instruction cache information

Syntax

info icache *cache-item[,cache-item]**

cache-item : (address | addr-start[word-count] | addr-low:addr-high)

Description

The `info icache` command prints the instruction cache information for the given address or address range. The printed information includes the index, tag, an indication of whether the address hit or missed in the cache, the data in memory, and the data in cache.

Notes

The `info icache` command is valid only if the instruction cache is configured. Addresses are aligned on 32-bit boundaries.

Examples

info icache 0x2.8	Print contents of the instruction cache for address `0x2000_0008`
info icache 0x2.8:0x8.18	Print contents of the instruction cache for addresses ranging from `0x2000_0008 to 0x8000_0018`

info lookup - Print TIE lookup memory contents

Syntax

info lookup *lookup-memory-name*

info lookup *lookup-memory-name*[*address*]

Description

The `info lookup` command prints the contents of a specialized memory with the TIE `lookup_memory` property. The first form prints the entire contents; the second form prints the value at the specified address.

Examples

info lookup RAM	Print the contents of the TIE lookup memory named `RAM`.
info lookup RAM[10]	Print the value at the address `10` of the TIE lookup memory named `RAM`.

info mem - Print information about memory locations

Syntax

info mem *[–p] mem-item[,mem-item]**

mem-item : (address | addr-start[word-count] | addr-low:addr-high)

Description

The `info mem` command prints a combined picture of memory location, relevant instruction and data cache information (if caches are configured), and translation information for the given address or address range.

The value in the cache is printed only if it differs from the value in memory; otherwise only a hit or miss indication is printed. Addresses on cache boundaries are marked by `C_`. A warning is printed if the data TLB translation is different from the instruction TLB translation.

With the `-p` flag, the specified addresses are considered physical (the default is virtual).

Notes

Addresses are aligned on 32-bit boundaries.

Examples

info mem 0x2.8, 0x2.20	Prints contents of address `0x2000_0008` and `0x2000_0020`
info mem –p 0x2.8[6]	Prints 6 words starting from physical address `0x2000_0008`
info mem 0x2.8:0x2.20	Prints contents of memory between addresses `0x2000_0008` and `0x2000_0020`

info pipe - Print pipeline status

Syntax

```
info pipe [-c] [-r]
```

Description

The `info pipe` command prints the instruction or pipeline stall cause in each pipeline stage. With the `-c` flag, the output is color-coded (on terminals that support colors): pipeline stalls are printed in red and instruction opcodes in blue. The `-r` flag implies that the order of pipeline stages is from first to last (by default, the last pipeline stage is printed first).

Examples

`info pipe -r`	Print the pipeline information, ordering stages from first to last

info reg - Print register file information

Syntax

```
info reg [register-file[register-index]]
```

Description

Without any arguments, the `info reg` command prints the list of all register files. If only *`register-file`* is specified, committed values for all the registers in the given *`register-file`* are displayed. If *`register-index`* is also provided, the committed value of the single specified register is displayed.

For the windowed AR registers, *`register-index`* represents the physical index in the register file, and not the index in the current register window.

Examples

`info reg`	Print the list of all register files
`info reg AR`	Print the values of all AR registers
`info reg AR[10]`	Print the value of the AR register with index 10

info regstage - Print per-stage register file information

Syntax

```
info regstage register-file[register-index]
```

Description

If only *register-file* is specified, the `info regstage` command prints per-stage values for all the registers in the given *register-file*. If *register-index* is also provided, values of the single specified register for each pipeline stage are displayed.

Examples

`info regstage AR`	Print the pipeline stage values of all AR registers
`info regstage AR[10]`	Print the pipeline stage value of the AR register with index 10

info state - Print processor state information

Syntax

```
info state [state-name]
```

Description

The `info state` command prints the information for the named processor state (which may map to a user or a special register). With no arguments, it prints the values of all processor states.

Examples

`info state`	Print the size and current values of all processor states
`info state WindowBase`	Print the current value of the `WindowBase` special register

info tie - Print TIE wire information Not for Diamond cores

Syntax

```
info tie [slot-index] [block-name[:function-name]* wire-name]
```

Description

The `info tie` command prints the information for the named wire from a TIE block. The block specified by *`block-name`* represents a TIE operation (reference or semantic) or a shared function; it may be followed by an optional list of function names separated by colons. If *`wire-name`* is omitted, all wires from the specified block are printed. If *`block-name`* is also omitted, all wires from all TIE instructions are printed. If the optional *`slot-index`* is specified, only the instructions in the given slot are considered.

Notes

To use this command, you must compile your TIE file with the `-libdebug` option.

If a TIE instruction has both a reference and a semantic description, the simulator will use the semantic block by default, or the reference block if the `--reference` command-line option is specified.

Examples

`info tie`	Print all wires from all TIE instructions
`info tie 1`	Print all wires from all TIE instructions in slot index 1
`info tie one`	Print all wires from TIE semantic or reference block `one`
`info tie one abc`	Print wire `abc` from TIE semantic or reference block `one`
`info tie one:two`	Print all wires from TIE function `two` called from semantic or reference block `one`
`info tie one:two abc`	Print wire `abc` from TIE function `two` called from semantic or reference block `one`
`info tie 0 one:two`	Print all wires from TIE function `two` called from semantic or reference block `one` in slot index 0

mode - Change or print simulation mode

Syntax

```
mode [cycle|turbo]
```

Description

The `mode` command instructs the simulator to switch to the cycle-accurate or the fast functional simulation mode. Without an argument, it displays the current simulation mode.

Examples

`mode`	Print the current simulation mode
`mode cycle`	Switch to the cycle-accurate simulation
`mode turbo`	Switch to the fast functional simulation

print - Print prompt variable

Syntax

```
print prompt-variable
```

Description

The `print` command displays a single prompt variable. This is useful for obtaining the information that is not included in the command-loop prompt. See the `prompt` command (page 98) for the full list of prompt variables.

Examples

`print %C`	Print the current cycle count
`print %(DIS_W)`	Print the disassembly of the instruction in the W stage

prompt - Change the command-loop prompt

Syntax

prompt *[-c] prompt-string*

prompt *[-c] [predefined-prompt-number]*

Description

The `prompt` command specifies the information that is printed in the command-loop prompt. The prompt-string is a quoted string of ASCII characters. Some characters and parenthesized names preceded by % are prompt variables with special interpretation. The list of current prompt variables is shown on page 99.

Without any arguments, the `prompt` command prints the current prompt string and the numbered list of predefined prompts. A predefined prompt can be selected by specifying the *`predefined-prompt-number`*.

With the `-c` flag, the prompt variables will be color-coded in the same way as with the `info pipe -c` command (page 91).

Prompt variables

`%B or %(bpname)`	Name of the current breakpoint, if it exists
`%C or %(cycle)`	Current cycle count
`%T or %(icount)`	Number of committed instructions
`%L or %(last)`	Cycle in which the last instruction committed
`%D or %(diff)`	Number of cycles since the last instruction committed
`%stage`	Instruction size, PC and disassembly for the given pipeline `stage`, where `stage` is one of `I`, `R`, `E`, `M`, `W`
`%(PC_stage)`	Instruction PC for the given pipeline `stage`, where `stage` is one of `I`, `R`, `E`, `M`, `W`, `COMMIT`
`%(DIS_stage)`	Instruction disassembly for the given pipeline `stage`, where `stage` is one of `I`, `R`, `E`, `M`, `W`,`COMMIT`

Examples

`prompt`	Print the current prompt string and all predefined prompts
`prompt 2`	Install the predefined prompt number 2
`prompt "%(PC_W) %(DIS_W) [%C] => "`	Define the prompt which prints the PC and the disassembly of the instruction in the W stage, the current cycle count in square brackets, followed by an arrow. It will print similar to the following: `0x8010437c movi.n a15,1 [3026] =>`

source - Execute commands from file

Syntax

```
source –f=file
```

Description

The `source` command executes command-loop commands from the given *`file`*. It can be used to install a user's set of aliases and prompts, or bring the simulator to a certain point where it was during a previous invocation. The latter may be achieved by saving all the command-loop commands executed during the session into the file (see the `history` command on page 84).

Notes

A file with command-loop commands may contain other source commands. No loop checking is performed.

Examples

`source -f=init.alias`	Execute command-loop commands from file `init.alias`

step - Advance by instruction

Syntax

```
step [n]
```

Description

The `step` command continues the simulation until *n* more instructions are committed (if *n* is omitted, the default is 1). In contrast to `dstep` (page 81), when the simulation stops, the pipeline is left untouched and the processor state may not be fully consistent.

Examples

`step 100`	Run the simulator for 100 committed instructions
`step`	Run the simulator until the next instruction commits

summary - Print execution statistics

Syntax

```
summary
```

Description

The `summary` command prints the runtime statistics during program execution. See Chapter 6 for a description of summary output.

Examples

```
summary
```

unalias - Remove alias

Syntax

```
unalias alias-name
```

Description

The `unalias` command deletes the *`alias-name`* (if one exists) from the list of aliases. `unalias` without *`alias-name`* does not delete all existing aliases — they must be deleted individually.

Examples

`unalias ir`	Remove alias `ir` (predefined alias for `info reg`)

watch - Set, clear or print watchpoints

Syntax

watch *[–c] watch-item*

watch-item : (address | addr-start[byte-count] | addr-low:addr-high)

Description

The `watch` command sets a watchpoint at the given address or address range. If a single address is specified, then the 32-bit word starting at that address is watched. For an address range, every byte in the range is watched. When a watched item is written to, the simulation will stop.

With the `-c` flag, the `watch` command clears watchpoints for the specified addresses.

Without any arguments, it prints the list of currently watched addresses.

Examples

watch 0x2000_0040	Set the watchpoint for address `0x2000_0040`
watch –c 0x2000_0040	Clear the watchpoint for address `0x2000_0040`
watch	Print all existing watchpoints

write - Write register, state or memory location

Syntax

write reg *register-file[register-index] value*

write state *state-name value*

write mem *[-p] address value*

Description

The `write` command assigns the given `value` to the specified register, state or memory location. For registers and states, the `value` is a sequence of up to four 32-bit numbers. For memory locations, the `value` is a single 32-bit number, and the `address` must be 4-byte aligned. With the `-p` flag, the `address` is considered physical (the default is virtual).

Examples

write reg AR[10] 0xabcd	Assign the value `0xabcd` to the AR register with index 10
write state WindowBase 3	Change the value of the `WindowBase` special register to 3
write mem 0x2.8 0xabcd	Write the value 0xabcd to the memory location at address `0x20000008`

5. Client Packages

A client package is a dynamically loaded program that extends the functionality of the simulator. You can invoke an ISS client with the `--client` or `--client_commands` option. The client invocation syntax is

```
client-name [client-options]
```

With the `--client` command-line option (page 14), you specify a text file, and each line in the file is a single client invocation. With the `--client_commmands` option (page 15), a client invocation is specified directly in the option's string argument. The **Syntax** sections in the remainder of this chapter give the name and invocation options for each ISS client.

Some top-level ISS command-line options for common tracing and profiling tasks, `--pchistory` (page 40), `--profile` (page 43), and `--trace` (page 58), are simply aliases to client invocations. While they do not provide the full flexibility of the client invocation syntax, they are useful shortcuts for most frequently-used client options.

In addition to using command-line options to configure client behavior at the invocation time, you can also send run-time commands to clients for dynamic control during the simulation. You can issue dynamic client commands from the command-loop mode, by using the `client` command (page 77); from the debugger-backend mode, by using the `iss client` command in `xt-gdb`; or programatically, by calling the function:

```
int xt_iss_client_command(const char *client, const char *command);
```

from your Xtensa target program. This function is declared in `<xtensa/sim.h>`; the return value `0` indicates success, and `-1` indicates failure. The `client` parameter is the same client name from the invocation syntax, and the `command` parameter specifies the command that you want to issue. For added convenience, if you pass `"all"` as the `client` parameter, the specified `command` will be sent to all the clients that can accept it. For example, `xt_iss_client_command("all", "enable");` will be accepted by the following clients: `isa_profile`, `profile`, and `summary`. The **Dynamic Commands** sections in the remainder of this chapter list the run-time controls that each client accepts.

Note: Programmatic client control is implemented through the `SIMCALL` instruction interface (Section 2.2), and is therefore available only in the simulator. Trying to use the `xt_iss_client_command()` function in an Xtensa program not targeted for the simulator will result in an unresolved symbol at link time.

5.1 Memory Checking (`ferret`)

Syntax

```
ferret [--warn-unaligned] [--cmdfile=command-file]
```

Description

This client is a basic memory debugger for target programs running under XTOS (a single-threaded runtime described in the *Xtensa System Software Reference Manual*). Although the ISS executes programs compiled for the Xtensa processor, it does not usually monitor memory accesses for validity. It can be very difficult to track down runtime memory access errors in applications. The ferret client can be helpful in identifying such errors. It intercepts each memory access and verifies that it is legitimate. It can detect errors such as:

- Reading or writing beyond the bounds of an allocated array
- Freeing memory more than once
- Reading or writing memory that has been freed
- Reading uninitialized memory
- Writing into system ROM
- Reading or writing outside the bounds of system memory
- Stack overflow

When the ferret client detects an error, it generates a warning message that includes the PC of the instruction that caused the error, the memory address, and the type of error. If you provide the *`command-file`*, you can suppress warnings based on PC, memory address, type, or some combination of the three.

To enable memory debugging, it is generally necessary to link your target program with the ferret library (`-lferret`). This library supplies special versions of `malloc`, `free`, and `sbrk` that communicate information to the ferret client. If you are linking with `xt-xcc`, you should place the flag `-lferret` at the end of the command line. If you are using `xt-ld` to link explicitly then you should place `-lferret` just before the `-lc` flag, so that the ferret memory allocation routines will override those in `-lc` (or `libc.a`).

If you want to build with the ferret library but you do not directly reference `malloc` in your program, you must pass the flag `-u malloc` to the linker using a command such as:

```
xt-xcc main.o -Wl,-u -Wl,malloc -lferret
```

or

```
xt-ld -u malloc -lferret -lc main.o
```

The `-u malloc` flag must precede `-lferret` on the linker command line.

Linking with the ferret library is not necessary to trap all the errors ferret monitors. Detection of writes to system ROM or writes outside the bounds of system memory do not require the ferret library.

Warnings

Ferret detects several types of common memory problems. The tag in parentheses is the token used to identify the warning in the ferret client output. The same tag is used for suppressing the warnings via the `ignore` command, which can be issued dynamically or specified in the *`command-file`*.

Multiple Free (**`mult_free`**)
: A pointer passed to `free` has already been freed before.

```
char *z = malloc(1024);
...
free(z);
...
free(z);  // warning!
```

Bad Free (**`bad_free`**)
: An address was passed to `free` that was not allocated with `malloc`.

```
char *a = malloc(45);
int aa = 56;
free(aa);   // warning!
```

Array Access Out of Bounds (**`array_bnd`**)
: An access occurred a short distance before or after a block allocated with `malloc`.

```
char *str = malloc(64);
*(str-1) = '?';  // warning!
str[64] = '!';   // warning!
```

Access to Freed Memory (**`freed_mem`**)
: An access to memory that has been freed.

```
char *p = malloc(16);
free(p);
p[0] = '?';  // warning!
```

Unaligned Access (**unaligned**)
An access to unaligned memory location.

```
char *str = malloc(64);
*(short*)(str+1) = 1;  // warning!
*(int*)(str+2) = 2;    // warning!
```

No Heap (**no_heap**)
Out of memory.

```
char *p;
while (1) {
  p = malloc(0x1000000); // warning! when memory is exhausted
}
```

Memory Not Initialized (**not_init**)
A read occurred to a location that was not written to first.

```
char *str(int arg) {
  return malloc(arg);
}
...
char *my_str = str(10);
fprintf(stderr, "%s", my_str);  // warning!
```

Orphan Access (**stk_bnd**)
An access occurred to a region outside the stack and heap. A common cause of this type of error is returning a pointer to a local variable.

```
char *foo() {
  char bad[1024]="This doesn't seem right\n";
  return bad;
}
...
char *my_str = foo();
strcat(my_str, "... no it is not\n"); // warning!
```

Note: It is not uncommon for code in interrupt handlers to access memory locations outside the current stack and heap. You can suppress spurious **stk_bnd** warnings by using the `ignore` command (described below).

Access Outside Bounds of System Memory (**mem_bnd**)
An attempt was made to access system memory outside the bounds defined in the configuration file, or an attempt was made to write into system ROM.

Options

By default, the ferret client does not warn about unaligned memory accesses when the simulated Xtensa processor is configured to handle such accesses in hardware. The `--warn-unaligned` option can be used to override that behavior and always force unaligned access warnings.

The optional *`command-file`* contains the list of `ignore` commands (described below) that indicate which warnings should be suppressed. Any text following a # character is ignored up to the end of the line.

Dynamic Commands

`ignore` *`[`***`type=`***`check_type] [`***`pc=`***`value-or-range] [`***`mem=`***`value-or-range] ...`*

The `ignore` command allows you to selectively suppress warnings by type, instruction PC range, memory location range, or some combination of the three. The *`check_type`* is one of the ferret client warning tags. For `pc` and `mem` fields *`value-or-range`* is either a single number, or a range specified by a pair of numbers in brackets:

`[`*`number number`***`]`**

For example:

```
ignore type=unaligned
```

will suppress warnings about unaligned memory accesses.

```
ignore mem=0xc0000004
```

will suppress all warnings for memory location `0xc0000004`.

```
ignore type=not_init pc=[0xa0000000 0xa000ffff]
```

will suppress warnings about uninitialized memory accesses caused by instructions in the range between `0xa0000000` and `0xa000ffff`.

If both the PC range and the memory range are specified, the warnings are suppressed across the union of the two ranges.

5.2 Instruction Statistics (`isa_profile`)

Syntax

```
isa_profile [options] [file]
```

Description

This package provides a human-readable profile of the distribution of instruction formats and opcodes that are executed in a program. The data is recorded per-function in the target program.

Options

`--disable`	Start simulation with profiling turned off. You can turn it on by issuing the `enable` command from the command loop or debugger, or by calling `xt_iss_client_command("isa_profile", "enable")`.
`--program=`*`elf-file`*	Specify the ELF program name used to map instruction addresses to functions.
`--formats`	Generate profiles of the instruction formats executed.
`--no-formats`	Do not generate profiles of the instruction formats executed.
`--slots`	Generate slot-based profiles.
`--no-slots`	Do not generate slot-based profiles.
`--unused-opcodes`	Generate data for the unused opcodes.

If the *`file`* argument is specified, the report is written to the named file; otherwise, the report is printed to the standard output.

Dynamic Commands

The isa_profile client accepts the following dynamic commands:

`enable`	Turn profiling on.
`disable`	Turn profiling off.
`report` *`[file]`* **`dump`** *`[file]`*	Dump the current profile data. If *file* is omitted, the report is written to the file specified in the client invocation.
`reset`	Reset the profile data.

5.3 Binary File Loading (`loadbin`)

Syntax

```
loadbin file@address
```

Description

This package provides a facility to load raw binary images directly into the target program's virtual memory. A raw image is a contiguous sequence of raw data, which does not contain any auxiliary information such as size, symbols, or checksums usually embedded in popular file formats.

The entire contents of the specified *`file`* are loaded into memory starting at the given virtual *`address`*, specified as a decimal or hexadecimal integer number.

Options

None

Dynamic Commands

None

5.4 PC Tracing (`pchistory`)

Syntax

```
pchistory n
```

Description

The pchistory client allows you to easily print the PCs for the last *n* executed instructions. The list is printed when the simulator exits or every time the `dump` or `state` dynamic command is issued.

Options

None.

Dynamic Commands

`dump` **`state`**	Print the last *n* PCs.

5.5 Application Profiling (`profile`)

Syntax

```
profile [options] [file]
```

Description

The profiling client associates event counts with executed instructions in a form suitable for processing by `xt-gprof` (see the *GNU Profiler User's Guide*) or Xtensa Xplorer. Multiple events can be profiled simultaneously. For each event type selected, a file will be produced that contains call-graph edge counts and the event counts.

A call-graph edge event is recognized whenever the target program executes a call instruction or encounters an exception. At that time, a counter for the caller-callee or exception source-vector pair is incremented.

You can profile the following events:

- **Instructions** (`--instructions`)
  The number of committed instructions.
- **Cycles** (`--cycles`)
  The number of cycles spent executing each instruction, including penalties for events associated with the instruction: cache misses, taken branch overhead, and so forth. This is the default profiler output.
- **ICache/DCache Misses** (`--icmiss`, `--dcmiss`)
  The number of instruction cache or data cache misses for each executed instruction. These options are meaningless unless the simulation is run with the `--mem_model` option (page 31). For Xtensa NX processors configured with the vector cache option, scalar and vector data cache misses are combined.
- **ICache/DCache Miss Cycles** (`--icmiss-cycles`, `--dcmiss-cycles`)
  The number of instruction cache or data cache miss cycles for each instruction. These options are meaningless unless the simulation is run with the `--mem_model` option (page 31). For Xtensa NX processors configured with the vector cache option, scalar and vector data cache miss cycles are combined.
- **Branch Delays** (`--branch-delay`)
  The number of cycles lost due to taken branch overhead.
- **Pipeline Interlocks** (`--interlock`)
  The number of cycles lost due to functional unit interlocks.
- **Uncached Memory Accesses** (`--uncached-ifetch`, `--uncached-load`)
  The number of cycles for system memory accesses that bypass the instruction or data cache. These options are meaningless unless the simulation is run with the `--mem_model` option (page 31).

If the entire target program is executed in the fast functional simulation mode, only the committed instructions can be profiled. To profile additional events, use the `--sample` command-line option (page 47) to switch the simulation mode periodically. The ISS will collect profiling samples in the cycle-accurate simulation mode, and it will extrapolate gathered data based on the total number of instructions executed in each mode.

Options

`--instructions`	Generate an instruction count profile.
`--cycles`	Generate a cycle count profile (this is the default if no arguments are specified).
`--icmiss`	Generate an instruction cache miss count profile.
`--dcmiss`	Generate a data cache miss count profile.
`--icmiss-cycles`	Generate an instruction cache miss cycle profile.
`--dcmiss-cycles`	Generate a data cache miss cycle profile.
`--branch-delay`	Generate a taken branch overhead profile.
`--interlock`	Generate a pipeline interlock overhead profile.
`--uncached-ifetch`	Generate an uncached instruction fetch cycle profile.
`--uncached-load`	Generate an uncached data load cycle profile.
`--all`	Generate profiles for all applicable events. In the fast functional simulation mode, this implies only `--instructions`. Otherwise, it implies `--instructions --cycles --branch-delay --interlock`. In addition, if the simulation is run with `--mem_model`, it also implies `--icmiss --icmiss-cycles`, if the instruction cache is configured, `--dcmiss --dcmiss-cycles`, if the data cache is configured, and `--uncached-ifetch --uncached-load`, if the PIF is configured.
`--disable`	Start simulation with profiling turned off. You can turn it on by issuing the `enable` command from the command loop or debugger, or by calling `xt_iss_client_command("profile", "enable")`.
`--min-max`	Generate minimum and maximum event counts for each function across all of its calls.

The *`file`* argument specifies the output file base name; if *`file`* is omitted, the profile client uses `gmon.out` as the base name. If more than one profiling option is selected, the base name is suffixed with a type tag. The following list shows the suffixes.

`insn`	Instruction profile
`cyc`	Cycle profile
`icmiss`	Instruction cache miss count profile
`dcmiss`	Data cache miss count profile
`icmiss_cyc`	Instruction cache miss cycle profile
`dcmiss_cyc`	Data cache miss cycle profile
`bdelay`	Branch delay profile
`interlock`	Source interlock profile
`unc_ifetch`	Uncached instruction fetch cycle profile
`unc_load`	Uncached data load cycle profile

If the profile client is invoked from an XTMP or XTSC simulation, the core name forms an additional suffix term.

For example, given the following client line:

```
profile --icmiss --cycles prof
```

Two files are generated: `prof.icmiss` and `prof.cyc`. If the core name was `acorn`, then the files would be named `prof.icmiss.acorn` and `prof.cyc.acorn`.

Starting with the Xtensa Tools Version 9.0.3 (Tensilica release RD-2011.3), the profile client also collects inclusive event counts by annotating all call-site PCs on the call chain for an executed instruction. This enables more accurate profiling of children functions and removes the need for potentially imprecise estimates described in the *GNU Profiler User's Guide*.

Files with inclusive event counts have an additional `incl` suffix; in the above example, the inclusive count files will be named `prof.icmiss.incl` and `prof.cyc.incl`, or `prof.icmiss.acorn.incl` and `prof.cyc.acorn.incl`. These files will automatically be used by the Xtensa Xplorer profiler.

Starting with the Xtensa Tools Version 9.0.4 (Tensilica release RD-2012.4), if the profile client is invoked with the `--min-max` option, it also collects minimum and maximum event counts for each function across all of its calls. In the above example, files with the minimum and maximum cycle counts will be named `prof.cyc.min`, `prof.cyc.min.incl`, `prof.cyc.max`, and `prof.cyc.max.incl`. These files will automatically be used by the Xtensa Xplorer profiler.

Dynamic Commands

disable	Turn profiling off.
enable	Turn profiling on.
reset	Reset all event counts to 0.
dump **save**	Save the current profile data. For every invocation of this command, a new file is generated for each event using index suffixes 0, 1, 2, ...

Programmatic Control Examples

To programmatically disable or enable profiling, reset or save profiling data, you can call the `xt_iss_client_command()` function:

```
xt_iss_client_command("profile", "disable");
xt_iss_client_command("profile", "enable");
xt_iss_client_command("profile", "reset");
xt_iss_client_command("profile", "save");
```

To disable or enable profiling, you can also use the older functions:

```
void xt_iss_profile_disable();
void xt_iss_profile_enable();
```

The prototypes for these functions are provided in the `<xtensa/sim.h>` header file.

If you want to profile only the specific parts of your application, you can start the profiling client with the `--disable` argument and surround the program segments of interest with calls to `xt_iss_profile_enable()` and `xt_iss_profile_disable()`. In the following example:

```
#include <xtensa/sim.h>

main()
{
  ...
  xt_iss_profile_enable();
  computation();
  xt_iss_profile_disable();
  ...
}
```

only the code in function `computation()` would be profiled.

Optimizations performed by the Xtensa C and C++ compiler may move some code across the call to `xt_iss_profile_enable()` or `xt_iss_profile_disable()`. In such cases, it is possible that small portions of the code with only local effects are excluded from the profile.

In addition to the functions described in the `<xtensa/sim.h>` header file, which are available only in the simulator, you can also control profiling by using the following functions prototyped in the `<xtensa/xt_profiling.h>` header file:

```
void xt_profile_disable();
void xt_profile_enable();
int  xt_profile_save_and_reset();
```

The advantage of using these functions is that they are supported both by the simulator and by the hardware-based profiling.

5.6 Stack Analysis (`stackuse`)

Syntax

```
stackuse
```

Description

The stackuse client monitors all changes to the stack pointer and prints out a summary of the stack space used by the program. The stack usage information is printed when the simulator exits or every time the `dump` or `state` dynamic command is issued.

Note: The stackuse client does not support the CALL0 ABI.

Options

None

Dynamic Commands

dump	Dump current stack usage summary.

5.7 Performance Summary (`summary`)

Syntax

```
summary [options] [file]
```

Description

This package collects and reports the same performance data as with the `--summary` command-line option (page 50), with the added functionality for dynamic enabling and disabling of performance data collection. This allows you to generate performance summary data for specific parts of your application.

For the full description of the performance summary format, see Chapter 6.

Options

`--disable`	Start simulation with performance data collection turned off. You can turn it on by issuing the `enable` command from the command loop or debugger, or by calling `xt_iss_client_command("summary", "enable")`.
`--no-report`	Do not print the final summary at the end of simulation. This is useful if you want to see only the summary data explicitly requested via the `report` (`dump`) dynamic command.

If the *`file`* argument is specified, the performance summary is written to the named file; otherwise, the summary is printed to the standard output.

Note: If you invoke the summary client without the `--no-report` option and without the *`file`* argument, and you also use the `--summary` or `--mem_model` command-line option, two copies of the performance summary will be printed to the standard output.

Dynamic Commands

The summary client accepts the following dynamic commands:

`enable`	Enable performance data collection.
`disable`	Disable performance data collection.
`report` *`[tag]`* **`dump`** *`[tag]`*	Dump the current performance summary, optionally labeling it with the *`tag`* string.
`reset`	Reset the performance summary data.

5.8 Execution Tracing (`trace`)

Syntax

```
trace [options] [file]
```

Description

The trace client provides a machine/human readable dump of ISA level state changes, as well as some micro-architectural information such as caches and TLBs. For the full description of the execution trace format, see Appendix A.

Options

`--level=`*n*	Set the trace level to *n*. The default trace level is 2. 0 - no trace 1 - executed instructions, including exceptions and replays 2 - register file reads and writes, state and special register writes 3 - load/store and instruction fetch unit activity 4 - PIF activity 5 - cache and TLB activity 6 - timing information
`--start=`*n*	Start the trace at the instruction number *n*. This is the absolute number of instructions that have completed since simulator invocation and not the value of the ICOUNT register. By default, tracing starts at the beginning of simulation.
`--stop=`	Stop the trace at the instruction number *n*. This is the absolute number of instructions that have completed since simulator invocation and not the value of the ICOUNT register. By default, tracing stops at the end of simulation.
`--tieprint`	Include the TIEprint output in the trace. This allows you to see the effects of TIEprint statements interleaved with instructions that contain them.
`--wtime`	With the trace level 6, the default time stamp for each instruction is the cycle number in which that instruction completes the execution of the last pipeline stage. This option changes the time stamp to be the cycle number in which the instruction enters the commit (W) stage.

If the *`file`* argument is specified, the trace is written to the named file; otherwise, the trace is printed to the standard output.

Dynamic Commands

`level` *n*	Set the trace level to *n*.
`start` *n*	Start the trace at the instruction number *n*.
`stop` *n*	Stop the trace at the instruction number *n*.

Programmatic Control Examples

You can programmatically change the trace level by calling the `xt_iss_client_command()` function, as in:

```
xt_iss_client_command("trace", "level 6");
```

Alternatively, you can use the more specific backward-compatible function:

```
void xt_iss_trace_level(unsigned level);
```

The prototypes for these functions are provided in the `<xtensa/sim.h>` header file.

If you want to trace only specific parts of your application, you can run the trace client at level 0 and change the level only around the program segments of interest. In the following example:

```
#include <xtensa/sim.h>

main()
{
  ...
  xt_iss_trace_level(6);
  computation();
  xt_iss_trace_level(0);
  ...
}
```

only the code in function `computation()` would be traced.

You can achieve a similar effect during an interactive debugging session. For example:

```
(xt-gdb) target iss --client_cmds="trace --level=0 trace.log"
(xt-gdb) break computation
(xt-gdb) run
(xt-gdb) iss client trace level 6
(xt-gdb) finish
(xt-gdb) iss client trace level 0
```

Note that you must load the trace client prior to changing the trace level programmatically. If the trace client is not loaded with one of the `--client`, `--client_commands`, or `--trace` command-line options, calls to `xt_iss_trace_level()` will have no effect.

6. Performance Summary

During each simulation run, the ISS accumulates various statistics that vary from simple ISA-level details (for example, taken branches) to micro-architectural unit usage. With the `--mem_model` (page 31) or the `--summary` (page 50) option, the simulator displays the summary output of collected performance data. The output generated with the `--mem_model` option is a superset of the output produced when only the `--summary` option is given; if the memory modeling is not enabled, the summary output will not contain any information related to caches, local memories and PIF activity. Here is a sample summary output of a simulator run with the memory modeling enabled:

```
Xtensa Core:  "DC_B_212GP"   ISS Version: 9.0.2

Time for Simulation = 0.41 seconds

Current PC = 0x60005d3f

Cache Configuration:
  ICache: 8192 bytes (8KB), 2-way set associative, 32-byte line
  DCache: 8192 bytes (8KB), 2-way set associative, 32-byte line, writeback

Events                           Number  Number
                                         per 100
                                         instrs

Committed instructions          338428 ( 100.00 )
Instruction fetches             241747 (  71.42 )
   Uncached                       1782 (   0.53 )
   ICache fetches               239935 (  70.90 )
      ICache misses                152 (   0.04 ) 0.06% of ICache fetches
Taken branches                    1783 (   0.53 )
Exceptions                          29 (   0.01 )
   WindowOverflow                   15 (   0.00 )
   WindowUnderflow                  14 (   0.00 )
Loads                           114470 (  33.82 )
   Uncached                         10 (   0.00 )
   DCache loads                 114460 (  33.82 )
      DCache load misses            35 (   0.01 ) 0.03% of DCache loads
Stores                           99365 (  29.36 )
   DCache stores                 99365 (  29.36 )
      DCache store misses           47 (   0.01 ) 0.05% of DCache stores
DCache castouts                     53 (   0.02 )
PIF transfers (4 bytes each)      4055 (   1.20 )
   IFetch reads                   2965 (   0.88 )
   Data reads                      666 (   0.20 )
   Data writes                     424 (   0.13 )
```

```
Cycles: total = 359690
                                        Summed |           Summed
                                  CPI    CPI   |% Cycle  % Cycle
Committed instructions  338428 ( 1.0000  1.0000 |  94.09    94.09 )
Taken branches            3606 ( 0.0107  1.0107 |   1.00    95.09 )
Pipeline interlocks        139 ( 0.0004  1.0111 |   0.04    95.13 )
ICache misses             1854 ( 0.0055  1.0165 |   0.52    95.65 )
DCache misses             1175 ( 0.0035  1.0200 |   0.33    95.97 )
Exceptions                 148 ( 0.0004  1.0205 |   0.04    96.01 )
Uncached ifetches         9252 ( 0.0273  1.0478 |   2.57    98.59 )
Uncached loads              61 ( 0.0002  1.0480 |   0.02    98.60 )
Sync replays              2004 ( 0.0059  1.0539 |   0.56    99.16 )
Special instructions      1014 ( 0.0030  1.0569 |   0.28    99.44 )
Loop overhead             2004 ( 0.0059  1.0628 |   0.56   100.00 )
Reset                        5 ( 0.0000  1.0628 |   0.00   100.00 )
```

The first lines up to the `Events` line make up the header. Header information is composed of the following items: the identification of the Xtensa processor configuration and the simulator version; the time to run the simulation; the PC of the last instruction executed; and the information about the cache configuration (if applicable).

Following the header is the information about events that occurred during the simulation. The output is divided into three columns: a description of the event; the event count; and the ratio of the event count to the total number of committed instructions. Additionally, the summary of the cache-related events contains a fourth column that expresses the event count as a percentage of the unit's total activity. For example, cacheable loads that missed in the data cache accounted for 0.03% of the data cache read activity, or 35 / 114460.

For Xtensa processors configured with the Prefetch Unit Option, the total number of PIF transfers includes those initiated by the prefetch unit; instruction cache and data cache prefetch transfers are reported separately. The summary also gives the numbers of instruction cache and data cache misses that were serviced by the prefetch unit (prefetch hits), both as raw counts and as percentages of total cache line allocations.

Note that the number of instruction fetches may be larger than the number of committed instructions. The simulator models precisely the behavior of the Xtensa processor fetch engine, and the number of instruction fetches includes fetch requests for the instructions that are killed (due to exceptions or taken branches) or replayed.

The final section contains the execution cycle count and contributions of various events to the total number of cycles. The columns—from left to right—are: event description; number of cycles attributable to the event; the event's contribution to the CPI (cycle contribution / instructions committed); the running sum of the CPI; the event's contribution to the total cycle count; and the running sum of total cycle percentage. Descriptions of various events are given in Table 6–1.

Table 6–1. Performance Summary Events

Category	Description
`Committed instructions`	The total number of committed instructions.
`Taken branches`	The number of cycles lost due to taken branches, both conditional and unconditional. For each taken branch, there is a penalty of two cycles on a 5-stage pipeline or three cycles on a 7-stage pipeline, plus an additional cycle if the branch target is not aligned. For details, see the "Branch Penalties" section in the *Xtensa LX Microprocessor Data Book*.
`Mispredicted branches`	The number of cycles lost due to incorrectly predicted branches, both conditional and unconditional. For each incorrectly predicted branch, there is typically a penalty of 6 or 7 cycles. This category applies only to Xtensa NX processors. For details, see the "Branch Cycles" section in the *Xtensa Microprocessor Data Book*.
`Predicted branch stalls`	The number of cycles lost due to correctly predicted taken branches, both conditional and unconditional. This category applies only to Xtensa NX processors. For details, see the "Branch Cycles" section in the *Xtensa Microprocessor Data Book*.
`Pipeline interlocks`	The number of def-use stalls, i.e., cycles spent waiting to resolve instruction operand dependencies. An example of this is a pipeline bubble that must be inserted between a load instruction and an immediately following instruction that uses the data from the load. For details, see the "Pipeline Data and Resource Dependency Bubbles" section in the *Xtensa Microprocessor Data Book*.
`Icache misses`	The number of cycles spent processing instruction cache misses.
`Dcache misses`	The number of cycles spent processing data cache misses. For Xtensa NX processors, this category is split into `Scalar DCache misses` and `Vector DCache misses`.
`Dcache bank conflicts`	The number of cycles that the processor pipeline was frozen when two load operations try to access the same data cache bank.
`Load bank conflicts`	The number of cycles that the processor pipeline was stalled when two load operations try to access the same data cache or data RAM bank. This category applies only to Xtensa NX processors.
`Memory RAW stalls`	The number of cycles that the processor pipeline was stalled due to dependencies between stores and subsequent loads. This category applies only to Xtensa NX processors. For details, see the "Pipeline RAW Stall Conditions" section in the *Xtensa Microprocessor Data Book*.
`Exceptions`	The number of overhead cycles due to exceptions, not counting the cycles for executing instructions in exception handlers.
`Uncached ifetches`	The number of cycles spent on fetching instructions from uncached memories.
`Uncached loads`	The number of cycles spent on loading data from uncached memories.

Table 6–1. Performance Summary Events (continued)

Category	Description
Sync replays	The number of overhead cycles incurred when an instruction causes the pipeline replay of the next instruction. Examples include the ISYNC instruction, stores to instruction RAM, and instruction-cache manipulation instructions. For details, see the "Pipeline Replay Penalty and Conditions" section in the *Xtensa Microprocessor Data Book*.
Special instructions	The number of overhead cycles due to various special instructions. Examples include the SIMCALL instruction, barrier-like instructions (MEMW, EXCW, EXTW, S32RI, S32C1I), data-cache manipulation instructions, and loads from instruction RAM or ROM.
Loop overhead	The number of loop bubbles, i.e., overhead cycles incurred during the processing of LOOP, LOOPGTZ, and LOOPNEZ instructions. For details, see the "Loop Bubbles" section in the *Xtensa Microprocessor Data Book*.
Prefetch fill hazards	The number of cycles spent on the arbitration between data cache fills from the prefetch unit and load/store instructions in the pipeline.
Store buffer stalls	The number of cycles that the processor pipeline was frozen due to the store-buffer-full condition.
GatherD global stalls	The number of cycles that the processor pipeline was frozen due to a gatherD operation when the SuperGather option is configured.
Scatter global stalls	The number of cycles that the processor pipeline was frozen due to a scatter operation when the SuperGather option is configured.
TIE global stalls	The number of cycles that the processor pipeline was frozen during blocking accesses to TIE queues or lookups.
Non-TIE global stalls	The number of cycles that the processor pipeline was frozen due to conditions other than accessing TIE queues and lookups. Examples include assertion of the external RunStall signal, pending store when write buffer and store buffer are full, and a load from a busy local memory. For details, see the “GlobalStall Causes” section in the *Xtensa Microprocessor Data Book*.
Reset	The number of cycles at the startup before the first instruction commits.
Breakpoints	The number of cycles spent on flushing and restarting the pipeline when processing breakpoints in the debugger-backend or command-loop mode.
Simulator overhead	The number of cycles spent on the ISS internal activity, such as the simulation mode switching.

A. Execution Trace Description

The execution trace generated by the `trace` client is composed of an arbitrary number of steps, each representing a single instruction. The trace for each instruction may contain various events associated with that instruction's execution.

At level 1, only the executed instructions are traced. For example,

```
[ 0x601059c7 0x601059c7 4 ] (2403) 0xf0 0x1d "retw.n"
```

represents a valid completed instruction.

The first field,

```
[ 0x60003923 0x60003923 6 ]
```

gives the virtual and physical address of the instruction, and the memory attribute value. For details on memory attributes, see "The Memory Access Process" section in the *Xtensa Instruction Set Architecture (ISA) Reference Manual*.

The second field,

```
(2403)
```

is the instruction count number; this is not the value of the `ICOUNT` special register, but the total number of completed valid instructions.

```
0xf0 0x1d
```

is the stream of bytes that form the instruction. These values depend on the processor endianness. The bytes shown are for a little-endian configuration; for a big-endian configuration they would show as `0xd1 0x0f`.

```
"retw.n":
```

is the assembler mnemonic of the executed instruction.

In addition to completed instructions, at level 1 the trace also shows the instructions that were replayed, as in:

```
[ 0x60105941 0x60105941 4 ] (2883) 0x0e 0x28 "l32i.n a2,a14,0"
     $replay
```

and the instructions that caused exceptions:

```
[ 0x6010c1cb 0x6010c1cb 4 ] (3429) 0x02 0x9d "mov.n a9,a2"
     $exception("WindowOverflow")
```

The instruction count is not incremented for replays or exceptions.

Starting with the Xtensa Tools Version 9.0.2, replayed instructions are shown only for the conditions that cause the processor to replay the *current* instruction. The conditions that cause the processor to replay the *next* instruction are indicated by the `$replay_next` tag on the previous instruction:

```
[ 0x5000007e 0x5000007e 2 ] (527) 0x00 0x20 0x00 "isync"
     $replay_next
```

For details on these conditions, see the "Pipeline Replay Penalty and Conditions" section in the *Xtensa Microprocessor Data Book*.

At level 2, the additional information includes register file reads and writes, as well as writes to special registers and TIE states. For example, in the lines

```
AR[37] -> 0x601ffbf0
AR[46] <- 0x601ffc08
```

the right arrow shows that value `0x601ffbf0` was read from register `AR[37]`, and the left arrow shows that value `0x601ffc08` was written to register `AR[46]`. Register numbers are indices into the physical register file, and not into the current register window. In another example,

```
WindowBase <- 0xb
```

indicates that the value `0xb` was written into the `WindowBase` special register.

At level 3, the load/store and instruction fetch unit activity is shown. For example:

```
$translation( 0x601ffed8 0x601ffed8 4 "load" )

$translation( 0x6010c828 0x6010c828 4 "store" )

$translation( 0x60003920 0x60003920 2 "fetch" )
```

Each line shows the virtual and physical address, the memory attribute value, and the type of memory access.

Note: The speculative nature of the instruction fetch unit and its interaction with pipeline redirection events (branches, exceptions, replays) make it hard to accurately capture the complete instruction fetch activity in a trace organized by completed instructions. For example, multiple fetch events per instruction may be shown in the trace, a fetch event for an instruction may be shown with an instruction that precedes it, and a fetch event for a killed instruction may not be shown at all (although it would still be counted in the performance summary).

At level 4, the execution trace also includes PIF activity. For example,

```
pif [ 0x60003920 ] -> 0x1d055c65 data-read
```

shows a 32-bit PIF data read from the physical address `0x60003920` that returns value `0x1d055c65`. A read is indicated with a right arrow, and a write is indicated by a left arrow; for example, the following line shows a 64-bit PIF write to address `0x80020ae8`:

```
pif [ 0x80020ae8 ] <- 0x32203129_20283320 write
```

The types of PIF transactions shown are: `inst-read`, `data-read`, `icache-fill`, `dcache-fill`, `castout`, and `write`.

At level 5, the trace shows cache and TLB events. Most cache actions have the following format:

```
$action( "cache", way, index )
```

`action` is one of: `hit`, `miss`, `fill`, `repl`, or `evict`. *`cache`* is one of `dcache` or `icache`. *`way`* is not shown for `miss` events or for any other events if the cache is direct-mapped. `evict` events are only applicable to a write-back data cache. For example,

```
$repl( "dcache", 1, 204)
```

shows that the data-cache line with index `204` in the way `1` has been replaced.

TLB actions have the same format; *`action`* is either `hit` or `miss`, and *`cache`* is either `dtlb` or `itlb`.

Cache tag updates are shown in the following format:

```
cache [way][index] <- tag
```

`cache` is one of `dcache` or `icache`, and as with other cache events, *`way`* is not shown if the cache is direct-mapped.

Finally, at level 6, a time summary is provided. The first number (`4`) indicates how many cycles have elapsed between the completion of the previous and the current instruction. The second number (`1248`) is the cycle count when the current instruction completed.

```
$time( "iss", "clocks", 4, "cycle", 1248 )
```

An instruction is considered completed when it finishes the processing of the last pipeline stage. The last stage may be beyond the commit (W) stage, for example, when you have a TIE instruction that defines a state or a register in a stage later than W. If you invoke the `trace` client with the `--wtime` option, the reported cycles will correspond to each instruction's commit (W) stage.

B. Cycle Accuracy and Known Limitations

The ISS does not model the following optional features of Xtensa processors:

- Memory Parity and ECC for caches and local RAMs (insertion and detection of memory errors is not supported).
- Xtensa Debug Module, including Traceport, TRAX, and Performance Monitor options.

In the cycle-accurate simulation mode, the ISS models the Xtensa processor pipeline with a very high degree of accuracy. However, there may be small differences between the actual hardware and its ISS model in some cases, mostly related to the contention between multiple functional units and/or external devices for accessing processor's caches and local memories. Listed below are the most common sources of inaccuracy:

- The arbitration between multiple load/store units and, optionally, the prefetch unit for accessing the processor's data cache.
- The arbitration between inbound slave port requests and internal processor requests to local memories (which applies only to XTSC system simulations).
- The ISS only maintains the hit and miss information for the L0 loop buffer, and it does not suppress actual instruction fetch requests from a local memory.

The measured cycle accuracy on a number of applications and diagnostic programs is within 1% for Xtensa LX processors with no PrefetchToL1 option configured; for processors with Prefetch2L1 configured, the accuracy is within 2%.

For Xtensa NX processors the measured cycle accuracy is within 5% for applications that access instructions and data from local memories. For external memory accesses, the ISS memory model is unlikely to match any real memory implementation. The recommended approach for cycle-accurate modeling of external memories is to implement the interconnect and memory model using XTSC.

Index