UNISIM
TMS320C3X Manual

Gilles Mouchard
Daniel Gracia Pérez
Reda Nouacer

CEA List

1 User guide

1.1 Simulator features

The TMS320C3X is a 32-bit floating-point DSP from Texas Instrument. The UNISIM TMS320C3X 2.0 simulator features:

- Written for SystemC TLM 2.0
- Simulation of the TMS320C3X instruction set
- A simulation speed average around 11 MIPS and up to 14 MIPS on a 2.4 Ghz Core2 Duo machine under Linux
- Support for instruction cache
- Support for TI COFF v0, v1, and v2 (either with big-endian or little-endian headers)
- Built-in debugger (Inline Debugger)
- Support for GDB serial remote protocol (GDB server)
- Support for TI C I/O

1.2 Status of implementation

The UNISIM TMS320C3X has been developed using the following documentation:

- TMS320C3x Users Guide (SPRU031F, 2558539-9761 revision L, March 2004)
- TMS320C3x/C4x Assembly Language Tools Users Guide (SPRU035D, June 1998)
- TMS320C3x/C4x Optimizing C Compiler Users Guide (SPRU034H, June 1998)

The simulator current implementation completely decodes the TMS320C3X instruction set. All registers are present but no on-chip devices are implemented. The simulator has complete support for:

- integer instructions (2-ops, 3-ops, parallel ops, load/store)
- floating point instructions (2-ops, 3-ops, parallel ops, load/store)
- control instructions (branches, delayed branches, RPTS, RPTB), but `iack` and `swi` instructions

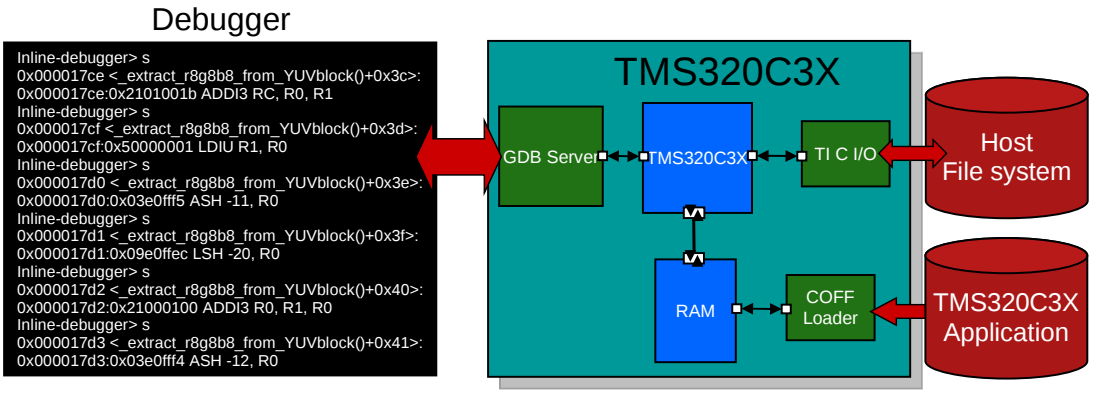

Figure 1: TMS320C3X simplified schematic.

- interlocked instructions, but `sigi` instruction
- power instructions
- interrupt handling

The current status of the simulator allows to run any integer or floating-point benchmark. However, during the validation process of the UNISIM TMS320C3X simulator, four hardware bugs have been found on our development board, and one software bug in Code Composer. The UNISIM TMS320C3X simulator can emulate these bugs (see Section 1.8) if they are enabled:

- `LDF || LDF` bug: From our experiments on the development board, uncomprehensibly `src1` is not correctly transformed to a valid `0.0` when the `src1` exponent is `0x80`. Simulator parameter `cpu.enable-parallel-load-bug` enables this bug.
- `STF || STF` and `STI || STI` bugs: From our experiments on the development board, the first store is never performed. Simulator parameter `cpu.enable-parallel-store-bug` enables these bugs.
- `RND` bug: TMS320C3x Users Guide says that the `rnd` instruction does not affect the `Z` flag however the real hardware systematically sets `Z` to 0. Simulator parameter `cpu.enable-rnd-bug` enables this bug.
- `lseek` bug: From our experiments on the development board, function `lseek` from `RTS30.LIB` has a 32-bit return value truncated to 16 bits. Simulator parameter `ti-c-io.enable-lseek-bug` enables this bug.
- floating point instructions bug: All the float instructions can use non-extended registers (all the registers different than `R0-R7`). However their behavior when using non-extended registers is not documented, and from our experiments on the development board their behavior is unexpected. By default, the simulator does not allow the use of non-extended registers for float instructions (obviously with the exception of the `FIX` and `FLOAT` instructions when the use of non-extended registers is documented). Simulator parameter `cpu.enable-float-ops-with-non-ext-regs` allows the use of non-extended registers for float instructions. Note that the behavior of the instructions when using non-extended registers has been deduced from our experiments with the evaluation board, but that they can not be validated due to the lack of documentation and unexpected behavior.

1.3 Compiling the simulator

Up-to-date instructions for compiling the simulator are available in the `INSTALL` file.

1.4 Invoking the simulator

The general command line format for invoking the simulator is the following:

```
unisim-tms320c3x-2.0 [<options>] <binary to simulate>
```

The binary to simulate must be a TI's COFF v0, v1 or v2 file. See 1.5 to generate such files.

The command line options of the simulator are:

- `--set <param=value> or -s <param=value>`: set value of parameter 'param' to 'value'
- `--config <XML file> or -c <XML file>`: configures the simulator with the given XML configuration file
- `--get-config <XML file> or -g <XML file>`: get the simulator configuration XML file (you can use it to create your own configuration. This option can be combined with -c to get a new configuration file with existing variables from another file
- `--list or -l`: lists all available parameters, their type, and their current value
- `--warn or -w`: enable printing of kernel warnings
- `--doc <Latex file> or -d <Latex file>`: enable printing a latex documentation
- `--version or -v`: displays the program version information
- `--share-path <path> or -p <path>`: the path that should be used for the share directory (absolute path)
- `--help or -h`: displays this help

1.5 The Texas Instrument cross-compiler for TMS320C3X

To compile programs for the TMS320C3X simulator, you can use the free evaluation cross-compiler for TMS320C3X running on a Windows host (SPRC147, TMS320C3x DSK Software) available at `http://focus.ti.com/docs/toolsw/folders/print/tmdsdsk33.html`. This cross-compiler also runs under other x86 operating systems such as Linux or MacOSX using Wine, a Windows emulator (`http://www.winehq.org/`).

Note: Be aware that any call to the C standard library requires linking the program with `RTS30.LIB`. Moreover, any call to I/O functions (open, close, read, write, printf, ...) requires TI C I/O support enabled in the TMS320C3X simulator.

The cross-compiler tool chain (`CL30.EXE, LNK30.EXE, ASM30.EXE, MK30.EXE, ar30.EXE, ...`) should be in your `PATH`. The shell variable `C_DIR` points to the location where the cross-compiler should search for the standard C headers and libraries. Suppose the tool chain is installed in `C:\TI`. Windows users should add the following in their `AUTOEXEC.BAT`:

```
set PATH=C:\TI\TIC3X4X\BIN;%PATH%
set C_DIR=C:\TI\TIC3X4X\INCLUDE;C:\TI\TIC3X4X\LIB
```

Wine and GNU bash users should add the following in their `.bashrc`:

```
export PATH=${HOME}/.wine/drive_c/TI/TIC3X4X/BIN:${PATH}
export C_DIR=C:\\TI\\TIC3X4X\\INCLUDE\;C:\\TI\\TIC3X4X\\LIB
```

1.6 The GNU binutils

The GNU binutils are a set of open source tools to manipulate binaries. They provide an assembler, a linker, and an object dump utility among others. The last version, at the time of writing this document, is available at: `ftp://ftp.gnu.org/gnu/binutils/binutils-2.19.1.tar.gz` The GNU binutils support TI COFF v0, v1 and v2 binary files for both TMS320C3X and TMS320C4X targets.

To compile the binutils and install them into `/opt/c4x-coff`:

```
$ ./configure --target=c4x-unknown-coff --prefix=/opt/c4x-coff
$ make
$ make install
```

A key feature of the GNU binutils is the ability of `objdump` to dump/disassemble a TI COFF binary for the TMS320C3X. For instance, the following command will dump file `test.out` into file `dump.txt`:

```
$ /opt/c4x-coff/bin/c4x-unknown-coff-objdump -D test.out > dump.txt
```

1.7 The GNU GDB debugger

Version 4.16 of GDB was patched to support C3x/C4x (see `http://www.elec.canterbury.ac.nz/c4x/doc/c4x-tools.html` and `ftp://ftp.rtems.com/pub/c4x-tools`). We've slightly patched again this port to make it work on a modern Linux distribution. It runs on 32-bit x86 Linux hosts. It is available at: `http://unisim-vp.org/site/downloads/other/crosstool/c4x-coff-gdb-4.16.tar.gz`.

To build this special version of GDB, do the following commands:

```
$ tar x c4x-coff-gdb-4.16.tar.gz
$ cd c4x-coff-gdb-4.16
$ ./build.sh all
```

That special GDB can connects to the UNISIM TMS320C3X simulator:

```
$ unisim-tms320c3x-2.0 -s enable-gdb-server=true -s gdb-server.tcp-port=1234
```

```
$ ./c4-coff-gdb/bin/c4-coff-gdb
(gdb) set machine 30
(gdb) target remote localhost:1234
```

1.8 Simulator configuration

The simulator stores its configuration (a set of parameters) in a XML configuration file.

The simulator can provide the user with a default XML configuration file with option `-g`:

```
$ unisim-tms320c3x-2.0 -g default_sim_config.xml
```

The simulator can load a XML configuration file with option `-c`:

```
$ unisim-tms320c3x-2.0 -c sim_config.xml
```

Note: Although it's not strictly necessary, parameter ***`inline-debugger.memory-atom-size`*** *should be set to value 4 as the TMS320C3X memory is not byte-addressable. If this parameter is not set to 4, presentation of the memory content and disassembly may seem unconventional in the inline debugger.*
The available parameters are summarized in table below:

Global

Name: `enable-gdb-server` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable GDB server instantiation.

Name: `enable-inline-debugger` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable inline debugger instantiation.

Name: `enable-press-enter-at-exit` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable pressing key enter at exit.

Name: `kernel_logger.file` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Keep logger output in a file.

Name: `kernel_logger.filename` **Type:** `parameter`
Default: `logger_output.txt` **Data type:** `string`

Description:
Filename to keep logger output (the option file must be activated).

Name: `kernel_logger.std_err` **Type:** `parameter`
Default: `true` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Show logger output through the standard error output.

Name: `kernel_logger.std_err_color` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Colorize logger output through the standard error output (only works if std_err is active).

Name: `kernel_logger.std_out` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Show logger output through the standard output.

Name: `kernel_logger.std_out_color` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Colorize logger output through the standard output (only works if std_out is active).

Name: `kernel_logger.xml_file` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Keep logger output in a file xml formatted.

Name: `kernel_logger.xml_file_gzipped` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
If the xml_file option is active, the output file will be compressed (a .gz extension will be automatically added to the xml_filename option.

Name: `kernel_logger.xml_filename` **Type:** `parameter`
Default: `logger_output.xml` **Data type:** `string`

Description:
Filename to keep logger xml output (the option xml_file must be activated).

cpu

Name: `cpu.max-inst` **Type:** `parameter`
Default: `0xffffffffffffffff` **Data type:** `unsigned 64-bit integer`

Name: `cpu.trap-on-instruction-counter` **Type:** `parameter`
Default: `0xffffffffffffffff` **Data type:** `unsigned 64-bit integer`

Name: `cpu.mimic-dev-board` **Type:** `parameter`
Default: `true` **Data type:** `boolean`
Valid: `true`, `false`

Name: `cpu.enable-parallel-load-bug` **Type:** `parameter`
Default: `true` **Data type:** `boolean`
Valid: `true`, `false`

Description:
When using parallel loads (LDF src2, dst2 —— LDF src1, dst1) the src1 load doesn't transform incorrect zero values to valid zero representation, instead they copy the contents of the memory to the register. Set to this parameter to false to transform incorrect zero values..

Name: `cpu.enable-rnd-bug` **Type:** `parameter`
Default: `true` **Data type:** `boolean`
Valid: `true`, `false`

Description:
If enabled the 'rnd' instruction sets the Z flag to 0 systematically, as it is done in the evaluation board. Otherwise, Z is unchanged as it is written in the documentation..

Name: `cpu.enable-parallel-store-↪bug` **Type:** `parameter`
Default: `true` **Data type:** `boolean`
Valid: `true`, `false`

Description:
If enabled, when using parallel stores (STF src2, dst2 —— STF src1, dst1) the first store is treated as a NOP..

Name: `cpu.enable-float-ops-with-↪non-ext-regs` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
If enabled non extended registers can be used on all the float instructions, however the behavior is not documented and can differ between chips revision. If disabled, it stops simulation when using non extended registers on float instructions..

Name: `cpu.verbose-all` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Name: `cpu.verbose-setup` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

loader

Name: `loader.verbose` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable verbosity.

Name: `loader.verbose-parser` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable verbosity of parser.

Name: `loader.filename` **Type:** `parameter`
Default: `c31boot.out` **Data type:** `string`

Description:
List of files to load. Syntax: [[filename=]<filename1>[:[format=]<format1>]][,[filename=]<filename2>[:[form
(e.g. boot.bin:raw,app.elf).

loader.memory-mapper

Name: `loader.memory-mapper.verbose` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable verbosity.

Name: `loader.memory-mapper.verbose-↪parser` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable verbosity of parser.

Name: `loader.memory-mapper.mapping` **Type:** `parameter`
Default: `memory=memory:0x0-0xffffffff` **Data type:** `string`

Description:
Memory mapping. Syntax: [[(memory=]<memory1>[:[range=]<low1-high1>]][,[(memory=]<memory2>[:[range=]<low2-high2>]]... (e.g. ram:0x0-0x00ffff,rom:0xff0000-0xffffff).

memory

Name: `memory.org` **Type:** `parameter`
Default: `0x0000000000000000` **Data type:** `unsigned 64-bit integer`

Description:
memory origin/base address.

Name: `memory.bytesize` **Type:** `parameter`
Default: `0` **Data type:** `unsigned 64-bit integer`

Description:
memory size in bytes.

ti-c-io

Name: `ti-c-io.enable` **Type:** `parameter`
Default: `true` **Data type:** `boolean`
Valid: `true`, `false`

Description:
enable/disable TI C I/O support.

Name: `ti-c-io.warning-as-error` **Type:** `parameter`
Default: `false` **Data type:** `boolean`

Valid: `true`, `false`

Description:
Whether Warnings are considered as error or not.

Name: `ti-c-io.pc-register-name` **Type:** `parameter`
Default: `PC` **Data type:** `string`

Description:
Name of the CPU program counter register.

Name: `ti-c-io.c-io-buffer-symbol-name` **Type:** `parameter`
Default: `__CIOBUF_` **Data type:** `string`

Description:
C I/O buffer symbol name.

Name: `ti-c-io.c-io-breakpoint-symbol-name` **Type:** `parameter`
Default: `C$$IO$$` **Data type:** `string`

Description:
C I/O breakpoint symbol name.

Name: `ti-c-io.c-exit-breakpoint-symbol-name` **Type:** `parameter`
Default: `C$$EXIT` **Data type:** `string`

Description:
C EXIT breakpoint symbol name.

Name: `ti-c-io.verbose-all` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
globally enable/disable verbosity.

Name: `ti-c-io.verbose-io` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
enable/disable verbosity while I/Os.

Name: `ti-c-io.verbose-setup` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:	
enable/disable verbosity while setup.	
Name: `ti-c-io.enable-lseek-bug`	**Type:** `parameter`
Default: `false`	**Data type:** `boolean`
Valid: `true`, `false`	
Description:	
enable/disable lseek bug (as code composer).	

1.9 Debugging the target program

The command line option `-s enable-inline-debugger=true` enables the inline debugger. The inline debugger has support for controlling the program execution, inspecting the program and its data, and putting breakpoints and watchpoints. The user can interact with the debugger using the following commands:

- Execution commands:
  - `<c | cont | continue> [<symbol | *address>]`:
    Continue to execute instructions until program reaches a breakpoint, a watchpoint, a 'symbol' or an 'address'.
  - `<s | si | step | stepi>`:
    Execute one instruction.
  - `<n | ni | next | nexti>`:
    Continue to execute instructions until the processor reaches next contiguous instruction, a breakpoint or a watchpoint.
  - `<r | run>`:
    Restart the simulation from the beginning (not yet supported).
- Inspection commands:
  - `<dis | disasm | disassemble> [<symbol | *address>]`:
    Continue to disassemble starting from 'symbol', 'address', or after the previous disassembly.
  - `<d | dump> [<symbol | *address>]`:
    Dump memory starting from 'symbol', 'address', or after the previous dump.
  - `<register name>`:
    Display the register value.
  - `<m | monitor> [<variable name>]`:
    Display the given simulator variable (displays all variable names if none is given).
  - `<p | prof | profile>`
    `<p | prof | profile> program`
    `<p | prof | profile> data`
    `<p | prof | profile> data read`
    `<p | prof | profile> data write`:
    Display the program/data profile.
- Breakpoints/Watchpoints commands:
  - `<b | break> [<symbol | *address>]`:
    Set a breakpoint at 'symbol' or 'address'. If 'symbol' or 'address' are not specified, display the breakpoint list.

- `<w | watch> [<symbol | *address[:<size>]>] [<read | write>]`:
  Set a watchpoint at 'symbol' or 'address'. When using 'continue' and 'next' commands, the debugger will spy CPU loads and stores. The debugger will return to the command line prompt once a load or a store accesses to the given 'symbol' or 'address'.
- `<del | delete> <symbol | *address>`:
  Delete the breakpoint at 'symbol' or 'address'.
- `<delw | delwatch> <symbol | *address> [<read | write>] [<size>]`:
  Delete the watchpoint at 'symbol' or 'address'.

- Miscellaneous commands:
  - `<h | ? | help>`:
    Display the integrated help.
  - `<quit | q>`:
    Quit the built-in debugger.

2 Developer guide

The TMS320C3X simulator is the combination of several software components:

- A service infrastructure in `unisim/kernel/service` (see Section 2.2).
- A built-in logger in `unisim/kernel/logger` (see Section 2.4.5).
- Several small utility classes in `unisim/util` (see Section 2.5).
- A TMS320C3X instruction set simulator in `unisim/component/cxx/processor/tms320` (see Section 2.1.1).
- A memory in `unisim/component/cxx/memory/ram` (see Section 2.1.2).
- Service interface definitions in `unisim/service/interfaces` (see Section 2.3).
- A multi-format loader (COFF, ELF, S-Rec, Raw) service and especially a COFF loader in `unisim/service/loader/coff_loader` (see Section 2.4.1).
- A TI C I/O service in `unisim/service/os/ti_c_io` (see Section 2.4.2).
- An inline debugger service in `unisim/service/debug/inline_debugger` (see Section 2.4.3).
- A GDB server service in `unisim/service/debug/gdb_server` (see Section 2.4.4).

2.1 Simulation Components

2.1.1 TMS320C3X instruction set simulator

The instruction set simulator source code is located in directory:
`unisim/component/cxx/processor/tms320`.
The UNISIM TMS320C3X instruction set simulator uses an instruction set simulator generator, GenISSLib. GenISSLib uses an instruction set description (`.isa` files) located in sub-directory `isa` of the instruction set simulator source code directory. Most computations (e.g., integer computation) are directly performed in these description files. See the GenISSLib manual for additional informations about the GenISSLib instruction set description language. The simulator is implemented in class `unisim::component::cxx::processor::tms320::CPU`, and its main methods are:

- `StepInstruction`: Executes one instruction.
- `PrWrite`: Write a word into memory using service import `memory_import` (see Section 2.2 for details about services). This method is virtual so that it can be reimplemented into a derived class.
- `PrRead`: Read a word from memory using service import `memory_import` (see Section 2.2 for details about services). This method is virtual so that it can be reimplemented into a derived class.
- `SetIRQLevel`: Set level (0/1, true/false) of an IRQ. IRQ numbering is same as register `IF` bit numbering.
- `ComputeIndirEA`: Compute the effective address for indirect addressing modes.
- `ComputeDirEA`: Compute the effective address for direct addressing modes.

This class is a client and a service (see Section 2.2 for details) that can be connected to a debugger, a loader, and a memory.

Each register (R0, R1, R2, R3, R4, R5, R6, R7, ar0, ar1, ar2, ar3, ar4, ar5, ar6, ar7, DP, IR0, IR1, BK, SP, ST, IE, IF, IOF, RS, RE, and RC) is implemented by an instance of class `unisim::component::cxx::processor::tms320::Register`. This class has methods to get/set value of a register and to perform floating point computations.
The table below summarizes the API of the CPU:

Module CPU

Class Name:
`unisim::component::cxx::processor::tms320::CPU`

Header:
`unisim/component/cxx/processor/tms320/cpu.hh`

Description:
This C++ class implements the TMS320C3X instruction set simulator.

Template Parameters

Name: `CONFIG` **Type:** `class`
Default value: `none`

Description:
This is a configuration class that is a collection of definitions to parameterize the simulation model.

Name: `DEBUG` **Type:** `bool`
Default value: `false`

Description:
Enable/disable debug.

Run-Time Parameters

Name: `max-inst` **Type:** `uint64_t`
Default value: $2^{64} - 1$

Description:
Maximum number of instructions to simulate. Once this threshold is reached, the CPU calls virtual method `Stop` to stop simulation.

Name: `trap-on-instruction-counter` **Type:** `uint64_t`
Default value: $2^{64} - 1$

Description:
Number of instructions to simulate before trapping, i.e., calling `ReportTrap` through service import `trap_import`. This is useful to inform the debugger that the CPU has simulated a certain amount of instructions, so that user can take control of the simulation at this point.

Name: `verbose-setup` **Type:** `bool`
Default value: `false`

Description:
Enable/disable verbosity of the CPU while setup.

Name: `verbose-all` **Type:** `bool`
Default value: `false`

Description:
Globally enable/disable verbosity of CPU.

Name: `enable-parallel-load-bug` **Type:** `bool`
Default value: `true`

Description:
When using parallel loads (`LDF src2, dst2 || LDF src1, dst1`) the `src1` load doesn't transform incorrect zero values to valid zero representation, instead they copy the contents of the memory to the register. Set this parameter to false to transform incorrect zero values.

Name: `enable-rnd-bug` **Type:** `bool`
Default value: `true`

Description:
If enabled the `rnd` instruction sets the Z flag to 0 systematically, as it is done in the evaluation board. Otherwise, Z is unchanged as described in the TMS320C3X documentation.

Name: `enable-parallel-store-bug` **Type:** `bool`
Default value: `true`

Description:
If enabled, when using parallel stores (`STF src2, dst2 || STF src1, dst1` or `STI src2, dst2 || STI src1, dst1`) the first store is treated as a NOP.

Name: `enable-float-ops-with-non-ext-regs` **Type:** `bool`
Default value: `false`

Description:
If enabled, float instructions can operate over non-extended registers. If disabled, the use of non-extended registers on float instructions will stop the program execution.

Service Exports

Name: `disassembly_export` **Interface:** `unisim::service::interfaces::Disassembly`

Description:
The CPU provides clients (e.g. debuggers) with a disassembly capability through this service export.

Name: `registers_export` **Interface:** `unisim::services::interfaces::Registers`

Description:
The CPU provides clients (e.g. debuggers) with an access to its registers through this service export.

Name: memory_export	**Interface:** unisim::service::interfaces ↪::Memory

Description:
The CPU provides clients (e.g debuggers) with an access to memory space through this service export. Accesses to memory space are non-intrusive, i.e. they do not affect timing or data placement (e.g. in caches or TLBs).

Name: memory_injection_export	**Interface:** unisim::service::interfaces ↪::MemoryInjection

Description:
The CPU provides clients (e.g debuggers) with an access to memory space through this service export. Accesses to memory space are intrusive, i.e., they affect timing and data placement (e.g., in caches or TLBs).

Name: memory_access_reporting_control	**Interface:** unisim::service::interfaces ↪::MemoryAccessReportingControl

Description:
The CPU allows a client to enable/disable memory access reporting through this service export.

Service Imports

Name: debug_control_import **Mandatory connected:** no	**Interface:** unisim::service::interfaces ↪::DebugControl

Description:
This service import allows to interactively control the CPU. Method FetchDebugCommand of the service import interface returns the control command for CPU: either execute one instruction or stop simulation.

Name: memory_access_reporting_import **Mandatory connected:** no	**Interface:** unisim::service::interfaces ↪::MemoryAccessReporting

Description:
The CPU reports memory accesses (e.g. to a debugger) using this service import.

Name: trap_reporting **Mandatory connected:** no	**Interface:** unisim::service::interfaces ↪::TrapReporting

Description:
The CPU informs a remote service (e.g. a debugger) that an event has occurred using this service import.

Name: `symbol_table_lookup_import`	**Interface:** `unisim::service::interfaces ↪::SymbolTableLookup`
Mandatory connected: no	

Description:
The CPU can obtain a translation from an address to a symbol name using this service import.

Name: `memory_import`	**Interface:** `unisim::service::interfaces ↪::Memory`
Mandatory connected: no	

Description:
The CPU accesses to an external memory using this service import.

Name: `ti_c_io_import`	**Interface:** `unisim::service::interfaces ↪::TI_C_IO`
Mandatory connected: yes	

Description:
The CPU allows a remote service (e.g. TI C I/O service) to capture `SWI` instructions. Such service should translate target program I/Os to host I/Os.

2.1.2 Memory

The source of class `unisim::component::cxx::memory::ram::Memory` is in directory: `unisim/component/cxx/memory/ram`.
Methods `ReadMemory` and `WriteMemory` respectively implement read and write memory accesses.
This simulation component provides the interface `unisim::service::interfaces::Memory` to other simulation components (e.g. CPU) or services (e.g. the COFF loader).
The table below summarizes the API of the memory:

Module Memory

Class Name:
`unisim::component::cxx::memory::ram::Memory`

Header:
`unisim/component/cxx/memory/ram/memory.hh`

Description:
This C++ class models a RAM.

Template Parameters

Name: `PHYSICAL_ADDR`
Type: `class`
Default value: `none`

Description:
This is the C++ type of a memory address (typically uint32_t or uint64_t).

Name: `PAGE_SIZE`
Type: `uint32_t`
Default value: `1 MB`

Description:
This is the size of a memory page in the implementation. This parameter is absolutely not related to an architectural parameter but only a hint to speed-up simulation (memory usage vs. speed).

Run-Time Parameters

Name: `org`
Type: `PHYSICAL_ADDR`
Default value: `0`

Description:
Starting address of the memory (typically 0).

Name: `bytesize`
Type: `PHYSICAL_ADDR`
Default value: `0`

Description:
Size in bytes of the memory.

Service Exports

Name: `memory_export`
Interface:
`unisim::service::interfaces::Memory`

Description:
The memory provides clients (e.g debuggers, loaders or CPUs) with an access to memory space through this service export. Accesses to memory space are non-intrusive, i.e. they do not affect timing or data placement.

2.2 Service infrastructure

Designing a new emulator, and particularly for a research purposes, means implementing an instruction set emulator but also involves several software components not directly related to pure instruction set execution. The most obvious needed software components are memories, debuggers, loaders, but components such as chipsets and peripherals are still mandatory to enable running unmodified real world applications. Abstracting the underlying host hardware is also something useful to emulators. Making all these components running together requires programming interfaces as much standard as possible.

Usually the programmer faces to the problems of sharing source codes among several emulators, reusing existing source codes, and building a fully functional emulator from all these heterogeneous pieces of source codes. Most of the time, the software components are strongly dependent of each other: components are statically linked together through explicit function calls and adhoc interfaces. Replacing these adhoc interfaces with C++ pure interfaces (C++ classes with only unimplemented virtual methods, see your C++ manual for more details) and linking the components through pointers is a step toward avoiding such strong dependencies between the components. But still finding a standard manner to initialize those pointers is necessary. This can be done either by directly writing in those pointers or calling special functions to do the job.

Another problem with heterogeneous software components is the manner to instantiate and parameterize them in a standard way, so that it is easier for the component's user to use a new component. Usually, parameterizing a component means passing arguments to an initialization function or a class constructor. It implies that the programmers agree on using only one of these two solutions or both. Still the programmers must know the setup order of these components: it is an error prone process because determining a correct order from the components documentation will likely fail the first times.

In this section, we present the standard way to share, reuse, link, parameterize and setup the software components within the TMS320C3X simulator. C++ object oriented programming and pure C++ interfaces enable sharing and reuse. In few words, some special pointers (classes `ServiceImport` and `ServiceExport`) linking the software components (classes `Service` and `Client`) together with some base software component classes have been introduced, thus enabling easier component composition and connection. The parameterization have been standardized (class `Parameter`) and the framework (class `ServiceManager`) uses additional dependency informations to provide the user with an automatic setup order.

2.2.1 Class hierarchy

Each software component of the UNISIM TMS320C3X simulator is an object (a client and/or a service). The term `Client` refers to an object that calls methods of a `Service` through a `ServiceImport`. The term `Service` refers to an object that exposes its interface to client through a `ServiceExport`. `ServiceImport` acts as gate for a client to call remote methods of a service. `ServiceExport` is a mean for a service to export its interface, so that a client `ServiceImport` can be bound to it.

Figure 2 presents the object/class hierarchy of the service infrastructure. This class hierarchy allows the `ServiceManager` to see clients and services as a service graph. The base class of the class hierarchy is `class Object`. It provides composition (it's a container) and naming of objects. Template class `Service<SERVICE_IF>` represents a service implementing interface `SERVICE_IF` while template class `Client<SERVICE_IF>` represents a client using a service implementing interface `SERVICE_IF`. On Figure 2, classes `MyService` and `MyClient` are respectively examples of a service and a client with interface `SERVICE_IF`. Example class `MyService` has a

member of type `ServiceExport<SERVICE_IF>` to export `SERVICE_IF` to the outside world. Example class `MyClient` has a member of type `ServiceImport<SERVICE_IF>` to import interface `SERVICE_IF` from a remote service.

Classes `ServiceImport<SERVICE_IF>` and `ServiceExport<SERVICE_IF>` provide a C++ operator `>>` to allow binding a service import to a service export, so that client is bound to a service. In the example of Figure 2, class `MyClient` would use service `MyService` as soon as `ServiceImport` of class `MyClient` is bound to `ServiceExport` of class `MyService`. A concrete use of service binding import and service is provided in the next section.

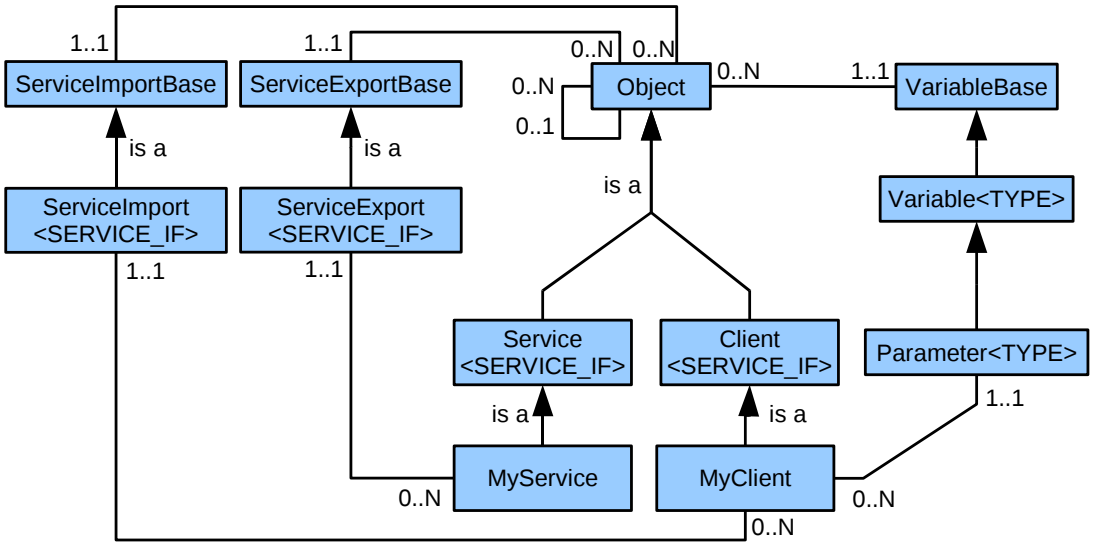

Figure 2: Service/Client/Run-time Parameters Object hierarchy.

2.2.2 Building a service graph

Using services implies building a service graph. For instance, consider that the client is a loader, and the service is a memory. The programmer creates objects `loader` and `memory`, see Figure 3.

```
1 Loader loader("loader");
2 Memory memory("memory");
```

Figure 3: Client/Service instantiation.

Object `loader` is a client because it needs a service (reading/writing in memory) from object `memory` to load the program. `loader` has a member `import` named `memory_import` whereas `memory` object has a member `export` named `memory_export`. The programmer connects the loader to the memory using `loader.memory_import` and `memory.memory_export`, see Figure 4.

```
1 loader.memory_import >> memory.memory_export;
```

Figure 4: Import/Export connection.

Once the programmer has created a service graph, he must perform a call to `ServiceManager::Setup()`. `ServiceManager::Setup()` returns `true` if setup of each service and client in the graph has been successful, otherwise it returns `false`.

2.2.3 Designing a service

A service is a C++ object inheriting from template class `Service<SERVICE_INTERFACE>` ❶, see Figure 5. `SERVICE_INTERFACE` is a C++ abstract class defining the virtual methods implemented by the service. To export its interface, a service must have a member of type `ServiceExport<SERVICE_INTERFACE>` ❷. For normalization purposes, the service constructor should only take two parameters ❸: the service name and the pointer to the parent (a container service). The pointer to the parent is `null` if the service is a top level service (no parent). The base `Object` constructor ❹ and the base `Service` constructor ❺ must be called with the name and the pointer to the parent. `ServiceExport` member constructor must be called with the export name and a pointer to the owner, i.e. the service itself ❻.

```
 1 class MyService : public Service<MyInterface> ❶
 2 {
 3 public:
 4   ServiceExport<MyInterface> my_interface_export; ❷
 5
 6   MyService(const char *name, Object *parent = 0) : ❸
 7     Object(name, parent), ❹
 8     Service<MyInterface>(name, parent), ❺
 9     my_interface_export("my-interface-export", this) ❻
10   {
11   }
12 };
```

Figure 5: Simple service.

2.2.4 Designing a client

A client is a C++ object inheriting from template class `Client<SERVICE_INTERFACE>` ❶, see Figure 6. `SERVICE_INTERFACE` is a C++ abstract class defining the virtual methods implemented by the service the client can call. To import an interface, a client must have a member of type `ServiceImport<SERVICE_INTERFACE>` ❷. For normalization purposes, the client constructor should only take two parameters ❸: the client name and the pointer to the parent (a container client). The pointer to the parent is `null` if the client is a top level client (no parent). The base `Object` constructor ❹ and the base `Client` constructor ❺ must be called with the name and the pointer to the parent. `ServiceImport` member constructor must be called with the import name and a pointer to the owner, i.e. the client itself ❻.

```
 1 class MyClient : public Client<ServiceInterface> ❶
 2 {
 3 public:
 4   ServiceImport<ServiceInterface> my_interface_import; ❷
 5
 6   MyClient(const char *name, Object *parent = 0) : ❸
 7     Object(name, parent), ❹
 8     Client<ServiceInterface>(name, parent), ❺
 9     my_interface_import("my-interface-import", this) ❻
10   {
11   }
12 };
```

Figure 6: Simple client.

2.2.5 Run-time parameters

Run-time parameterization can be added to a service or a client. "Run-time parameterization" means that the service and/or client can be reconfigured at run-time. It is opposed to "Static parameterization" or "template parameterization" which allows configuring a service and/or client at compilation-time. To expose a member variable as a run-time parameter, a client/service must have a member variable of type `Parameter<TYPE>`, where `TYPE` is the C++ type of the exposed member variable, see Figure 7. Multiple `Parameter` variables with different `TYPE`s can be defined within a client/service. Consider that a service would expose a member variable `x` ❶. An instance of class `Parameter` is defined as a member of the service ❷. The parameter is bound to the exposed variable ❸ in the service/client constructor.

```
 1 class MyService : public Service<MyInterface>
 2 {
 3 public:
 4   Parameter<unsigned int> param_x; ❷
 5
 6   MyService(const char *name, Object *parent = 0) :
 7     Object(name, parent),
 8     Service<MyInterface>(name, parent),
 9     param_x("x", this, x) ❸
10   {
11   }
12 private:
13   unsigned int x; ❶
14 };
```

Figure 7: Exposing a service/client member variable as a run-time parameter.

2.2.6 Setup Order

As explained in section 2.2.2, method `Setup` of class `ServiceManager` calls all `Setup` methods in the simulator. A problem may occur if setup order is important. For instance, consider two services: service `A` and `B`. `A::Setup()` uses service `B`. A correct setup order consist to first setup service `B` and then service `A`. To solve such setup dependency, programmer should call method `ServiceExportBase::SetupDependsOn` (e.g. in the class constructor) so that the service manager can ensure correct setup order. If the service manager finds a cyclic dependency, `ServiceManager::Setup()` fails: it generally means that clients and services have been badly designed.

2.3 Service Interfaces

All service interfaces are declared in namespace `unisim::service::interfaces` and located in directory `unisim/service/interfaces`.

2.3.1 Memory Interfaces

These interfaces allow reading/writing from/to memory space. The memory interfaces comes in two flavors:

- Non-intrusive memory access (`unisim::service::interfaces::Memory`): It should not affect timing and data placement (e.g. in caches and TLBs).
- Intrusive memory access (`unisim::service::interfaces::MemoryInjection`): It can affect timing and data placement.

The two C++ interfaces are:

```
 1 template <class ADDRESS>
 2 class Memory
 3 {
 4 public:
 5   Memory(){}
 6   virtual  Memory(){}
 7
 8   virtual void Reset() = 0;
 9   virtual bool ReadMemory(ADDRESS addr, void *buffer, uint32_t size) = 0;
10   virtual bool WriteMemory(ADDRESS addr, const void *buffer, uint32_t size) = 0;
11 };
```

```
 1 template <class ADDRESS>
 2 class MemoryInjection
 3 {
 4 public:
 5   MemoryInjection(){}
 6   virtual  MemoryInjection(){}
 7
 8   virtual bool InjectReadMemory(ADDRESS addr, void *buffer, uint32_t size) = 0;
 9   virtual bool InjectWriteMemory(ADDRESS addr, const void *buffer, uint32_t size) = 0;
10 };
```

The arguments to methods `ReadMemory`, `InjectReadMemory`, `WriteMemory`, `InjectWriteMemory` are:

- `addr`: the starting address of the data transfer between the memory and the buffer
- `buffer`: a pointer to the buffer of bytes
- `size`: the length in bytes to transfer between the memory and the buffer

2.3.2 Debugging Interfaces

These interfaces are intended for the connection of the simulation components (e.g. CPU, memory, devices, ...) with a debugger (e.g. inline-debugger, GDB server, ...).

Instruction disassembly. CPU components provide a disassembly capability of the instruction set using the `unisim::service::interfaces::Disassembly` interface for the debuger.

```
1 template <class ADDRESS>
2 class Disassembly
3 {
4 public:
5   virtual std::string Disasm(ADDRESS addr, ADDRESS& next_addr) = 0;
6 };
```

Method `Disasm` arguments are:

- `addr`: the byte address of the instruction to disassemble
- `next_addr`: the byte address of the next instruction

and returns a string with the disassembly of the instruction.

Register access. A CPU or a device provides an access to its registers using the `unisim::service::interf` interface for the debugger.

```
1 class Registers
2 {
3 public:
4   virtual unisim::util::debug::Register *GetRegister(const char *name) = 0;
5 };
```

Method `GetRegister` arguments are:

- `name`: the name of the register to retrieve the register interface

and returns a pointer to an `unisim::util::debug::Register` interface.

```
1 class Register
2 {
3 public:
4   virtual  Register() {}
5   virtual const char *GetName() const = 0;
6   virtual void GetValue(void *buffer) const = 0;
7   virtual void SetValue(const void *buffer) = 0;
8   virtual int GetSize() const = 0;
9 };
```

Method `GetName` returns the register name. Method `GetValue` fills in a buffer with the register value. Method `SetValue` sets the register value from a buffer. Method `GetSize` returns the register size in bytes.

Step by step execution. A simulation component (e.g. a CPU) leaves control to a debugger with the `unisim::service::interfaces::DebugControl` interface.

```
1 template <class ADDRESS>
2 class DebugControl
3 {
4 public:
5   typedef enum { DBG_STEP, DBG_SYNC, DBG_KILL, DBG_RESET } DebugCommand;
6
7   virtual DebugCommand FetchDebugCommand(ADDRESS cia) = 0;
8 };
```

Method `FetchDebugCommand` takes the current program counter as argument and returns a command for the simulation component: either finish the simulation or execute one instruction.

Monitoring memory accesses. An instrumented simulation component provides a memory access trace using the `unisim::service::interfaces::MemoryAccessReporting` interface. Such memory trace is useful for a debugger to monitor memory access.

```
1 template <class ADDRESS>
2 class MemoryAccessReporting
3 {
4 public:
5   typedef enum { MAT_NONE = 0, MAT_READ = 1, MAT_WRITE = 2 } MemoryAccessType;
6   typedef enum { MT_DATA = 0, MT_INSN = 1 } MemoryType;
7
8   virtual void ReportMemoryAccess(MemoryAccessType mat, MemoryType mt, ADDRESS addr, uint32_t size)
↪    = 0;
9   virtual void ReportFinishedInstruction(ADDRESS next_addr) = 0;
10 };
```

Method `ReportMemoryAccess` takes as arguments the memory access type (either read or write), the memory type (either data or instruction memory), the address of the access, and the size of the memory access. Method `ReportFinishedInstruction` takes as argument the address of next instruction to be executed.

Trap reporting. An instrumented simulation component informs a debugger about an important event using the `unisim::service::interfaces::TrapReporting` interface. Such event is useful for a debugger to pause simulation when such event occurs.

```
1 class TrapReporting
2 {
3 public:
4   virtual void ReportTrap() = 0;
5 };
```

Method `ReportTrap` takes no arguments.

Symbol. A service (e.g. a loader) provides lookup to the symbol table using the `unisim::service::interf` interface. This interface is useful for translating addresses to symbol names, and vice-versa.

```
1 template <class T>
2 class Symbol {
3 public:
4   enum Type {
5     SYM_NOTYPE = 0,
6     SYM_OBJECT = 1,
7     SYM_FUNC = 2,
8     SYM_SECTION = 3,
9     SYM_FILE = 4,
10    SYM_COMMON = 5,
11    SYM_TLS = 6,
12    SYM_NUM = 7,
13    SYM_LOOS = 8,
14    SYM_HIOS = 9,
15    SYM_LOPROC = 10,
16    SYM_HIPROC = 11
17  };
18
19  Symbol(const char *name, T addr, T size, typename unisim::util::debug::Symbol<T>::Type type, T
↪    memory_atom_size);
20  const char *GetName() const;
21  T GetAddress() const;
22  T GetSize() const;
23  typename unisim::util::debug::Symbol<T>::Type GetType() const;
24  string GetFriendlyName(T addr) const;
25 };
26
27 template <class T>
28 class SymbolTableLookup {
29 public:
30  virtual const typename unisim::util::debug::Symbol<T> *FindSymbol(
31    const char *name,
32    T addr,
33    typename unisim::util::debug::Symbol<T>::Type type) const = 0;
34
35  virtual const typename unisim::util::debug::Symbol<T> *FindSymbolByAddr(T addr) const = 0;
36  virtual const typename unisim::util::debug::Symbol<T> *FindSymbolByName(const char *name) const
↪    = 0;
37  virtual const typename unisim::util::debug::Symbol<T> *FindSymbolByName(
38    const char *name,
39    typename unisim::util::debug::Symbol<T>::Type type) const = 0;
40  virtual const typename unisim::util::debug::Symbol<T> *FindSymbolByAddr(
41    T addr,
42    typename unisim::util::debug::Symbol<T>::Type type) const = 0;
43 };
```

Efficient instrumentation. To limit the impact on simulation performance of memory access instrumentation in the simulation components, such instrumentation can be enabled or disabled at run-time using interface `unisim::service::interfaces::MemoryAccessReportingControl`.

```
1 class MemoryAccessReportingControl
2 {
3 public:
4   virtual void RequiresMemoryAccessReporting(bool report) = 0;
5   virtual void RequiresFinishedInstructionReporting(bool report) = 0;
6 };
```

2.3.3 Loader Interface

This interface provides basic informations about the loaded program.

```
1 template <class T>
2 class Loader
3 {
4 public:
5   virtual void Reset() = 0;
6   virtual T GetEntryPoint() const = 0;
7   virtual T GetTopAddr() const = 0;
8   virtual T GetStackBase() const = 0;
9 };
```

2.3.4 Time Interface

This interface provides the current simulation time of the component using it.

```
1 class Time
2 {
3 public:
4   virtual double GetTime() = 0; // in seconds
5 };
```

2.3.5 TI C I/O Interface

An instrumented TMS320C3X instruction set simulator provides a trace of `SWI` instructions using the `unisim::service::interfaces::ti_c_io` interface. This interface is useful for the TI C I/O service to capture target program I/Os and translate them to host I/Os.

```
1 class TI_C_IO
2 {
3 public:
4   typedef enum
5   {
6     ERROR = -1,
7     OK    = 0,
8     EXIT  = 1,
9   } Status;
10
11  virtual Status HandleEmulatorInterrupt() = 0;
12 };
```

2.4 Services

2.4.1 COFF loader service

This service provides UNISIM TMS320C3X simulator with a support for TI COFF v0, v1, and v2 binary files either with little-endian or big-endian headers (see TMS320C3x/C4x Assembly Language Tools Users Guide, Appendix A). The COFF loader service loads the programs into memory while setup (simulator initialization). The loader can interpret `.cinit` section if option `-cr` of TI C cross-compiler has been used while building the target program (see *TMS320C3x/C4x Optimizing C Compiler Users Guide*, section 4.8.1: *Autoinitialization of variables and constants*). To configure the COFF loader service see Section 1.8. The source code of COFF loader service is located in directory `unisim/service/loader/coff_loader`. The table below summarizes the COFF Loader service API:

Service COFF Loader	

Class Name:
`unisim::service::loader::coff_loader`
`↪::CoffLoader`
	Header:
	`unisim/service/loader/coff_loader`
	`↪/coff_loader.hh`

Description: The COFF loader service allows to load a COFF binary program into a memory and fill a symbol table. The loader also provides information about the loaded le such as the code and data locations (base address and size). The COFF loader loads the program during setup.	
Template Parameters	
Name: `MEMORY_ADDR` **Default value:** `none`	**Type:** `class`
Description: This is the C++ type of a memory address (e.g. `uint32_t` or `uint64_t`).	
Run-Time Parameters	
Name: `filename` **Default value:** empty string	**Type:** `string`
Description: The COFF file name to load into the connected memory.	
Name: `dump-headers` **Default value:** false	**Type:** `boolean`
Description: If true this parameter makes the COFF loader print the file headers on the screen (file header, section headers, symbol table ...) while loading the program.	
Service Exports	

Name: `logger_export`	**Interface:**
	`unisim::service::interfaces::`
	`↪Loader<MEMORY_ADDR>`

Description: The COFF loader provides information about the code and data location through this export.	

Name: symbol_table_lookup_export	**Interface:** unisim::service::interfaces:: ↪SymbolTableLookup<MEMORY_ADDR>
Description: The COFF loader provides symbol lookup through this export.	

Service Imports	
Name: memory_import **Mandatory connected:** no	**Interface:** unisim::service::interfaces:: ↪Memory<uint32_t>
Description: The COFF loader accesses to the memory through this import.	

2.4.2 TI C I/O service

This service provides low level I/O (open, read, write, close, ...) support on the host machine for target programs. The TI Run-time support libraries (`RTS*.lib`) implement a software stack for standard C I/Os (see *TMS320C3x/C4x Optimizing C Compiler Users Guide* (SPRU034H, June 1998), Appendix B). A development board debugger captures target program I/Os at `C$$IO$$`. The Run-time support library puts the I/Os in a communication buffer (`__CIOBUF_`) that the development board debugger translates to host I/Os. The debugger also captures target program termination at `C$$EXIT`. The UNISIM TI C I/O service captures and translates target program I/Os and termination in the same manner as a development board built-in debugger. To configure the TI C I/O service see Section 1.8. The source code of the COFF loader service is located in directory `unisim/service/os/ti_c_io`. The table below summarizes the TI C I/O service API:

Service TI C I/O	
Class Name:	**Header:**
`unisim::service::os::ti_c_io` ↪`::TI_C_IO`	`unisim/service/os/ti_c_io` ↪`/ti_c_io.hh`
Description: The TI C I/O service provides low level I/O (open, read, write, close, ...) support on the host machine for target programs.	
Template Parameters	
Name: `MEMORY_ADDR`	**Type:** `class`
Default value: `none`	
Description: This is the C++ type of a memory address (e.g. uint32_t or uint64_t).	
Run-Time Parameters	
Name: `ti_c_io.enable`	**Type:** `bool`
Default value: `false`	
Description: Enable/disable TI C I/O support.	
Name: `ti-c-io.warning-as-error`	**Type:** `bool`
Default value: `false`	
Description: Whether Warnings are considered as error or not.	
Name: `ti-c-io.pc-register-name`	**Type:** `string`
Default value: `"PC"`	
Description: Name of the CPU program counter register.	
Name: `ti-c-io.c-io-buffer-symbol-name`	**Type:** `string`
Default value: `"__CIOBUF_"`	
Description: C I/O buffer symbol name.	

Name: ti-c-io.c-io-breakpoint-symbol-name **Type:** string
Default value: "C$$IO$$"

Description:
C I/O breakpoint symbol name. The TI C I/O service installs a SWI instruction at this point to capture target program I/O.

Name: ti-c-io.c-exit-breakpoint-symbol-name **Type:** string
Default value: "C$$EXIT"

Description:
C EXIT breakpoint symbol name. The TI C I/O service installs a SWI instruction at this point to capture target program exit.

Name: ti-c-io.verbose-all **Type:** bool
Default value: false

Description:
Globally enable/disable verbosity of TI C I/O service.

Name: ti-c-io.verbose-io **Type:** bool
Default value: false

Description:
Enable/disable verbosity of TI C I/O service while performing I/Os.

Name: ti-c-io.verbose-setup **Type:** bool
Default value: false

Description:
Enable/disable verbosity of TI C I/O service while setup.

Service Exports

Name: ti_c_io_export **Interface:** unisim::interfaces::TI_C_IO<MEMORY_ADDR>

Description:
The TI C I/O provides target to host I/O translation through this service export.

Service Imports

Name: memory_import **Interface:** unisim::service::interfaces::Memory<MEMORY_ADDR>

Mandatory connected: no

Description:
The TI C I/O service accesses to the memory while setup through this import. While in setup it installs two SWI instructions to capture both target I/O and program exit.

Name: memory_injection_import **Interface:** unisim::service::interfaces::MemoryInjection<MEMORY_ADDR>

Mandatory connected: no

Description:
The TI C I/O service accesses to the memory while simulation through this import. It accesses to the I/O buffer in the target program memory and then interprete the content of this buffer to translate target program I/Os to host I/Os.

Name: `registers_import`
Mandatory connected: yes
Interface:
`unisim::service::interfaces ↪::Registers`

Description:
This service import should be connected to a CPU module. The TI C I/O service calls method `GetRegister` through this service import to get an interface to the CPU registers. The TI C I/O service uses methods `GetName`, `GetValue`, `GetSize` and `SetValue` of that interface to access to CPU registers. This import is mainly used to get the current PC, so that the TI C I/O service can distinguish target program I/Os from target program exit.

Name: `symbol_table_lookup_import`
Mandatory connected: yes
Interface:
`unisim::service::interfaces ↪::SymbolTableLookup<MEMORY_ADDR>`

Description:
The TI C I/O service uses this service import to get the address of the breakpoints and I/O buffer from their symbol name.

2.4.3 Inline debugger

The inline debugger service is a built-in debugger with a text-based user interface, see 1.9. It provides instruction level debugging of the target program. Table below summarizes the inline debugger service API:

Service Inline Debugger

Class Name:
`unisim::service::debug`
`↪::inline_debugger::InlineDebugger`

Header:
`unisim/service/debug`
`↪/inline_debugger/inline_debugger.hh`

Description:
The inline debugger service provides a simple text-based interface to interactively debug a target application running on a CPU module for the user. The debug is at the instruction level. The inline debugger may be connected to a CPU module.

Template Parameters

Name: `ADDRESS`
Type: `class`
Default value: none

Description:
This is the C++ type of a memory address (e.g. uint32_t or uint64_t).

Run-time parameters

Name: `inline-debugger.memory-atom-size`
Type: `unsigned integer`
Default value: `1`

Description:
Size of the smallest addressable element in memory.

Service Exports

Name: `debug_control_export`
Interface:
`unisim::service::interfaces`
`↪::DebugControl<ADDRESS>`

Description:
This service export should be connected to a CPU module. The CPU module calls method `FetchDebugCommand` through its service import to leave control to the debugger and fetch a new debug command.

Name: `memory_access_reporting_export`
Interface:
`unisim::service::interfaces`
`↪MemoryAccessReporting<ADDRESS>`

Description:
This service export should be connected to a CPU module. The CPU module calls methods `ReportMemoryAccess` and `ReportFinishedInstruction` through its service import. This allows the debugger to spy memory accesses and thus handle breakpoints and watchpoints.

Name: `trap_reporting_export`
Interface:
`unisim::service::interfaces`
`↪::TrapReporting`

Description:
This service export should be connected to a CPU module. A CPU module calls method `ReportTrap` through its service import. This allows the debugger to break execution on the simulated CPU once a trap condition is detected by the CPU module.

Service Imports

Name: `disasm_import`
Mandatory connected: yes
Interface:
`unisim::service::interfaces ↪::Disassembly<ADDRESS>`

Description:
This service import should be connected to a CPU module. The CPU module should implement method `Disassemble` which provides disassembling of the instructions for the debugger.

Name: `memory_import`
Mandatory connected: yes
Interface:
`unisim::service::interfaces ↪::Memory<ADDRESS>`

Description:
This service import should be connected to a CPU or a memory module. The debugger uses this service import to access to memory using methods `ReadMemory` and `WriteMemory`.

Name: `memory_access_reporting ↪_control_import`
Mandatory connected: no
Interface:
`unisim::service::interfaces ↪::MemoryAccessReportingControl`

Description:
This service import should be connected to a CPU module. The debugger calls methods `RequiresMemoryAccessReporting` and `RequiresFinishedInstructionReporting` through this service import to enable/disable memory access reporting from the CPU module.

Name: `registers_import`
Mandatory connected: yes
Interface:
`unisim::service::interfaces ↪::Registers`

Description:
This service import should be connected to a CPU module. The debugger calls method `GetRegister` through this service import to get an interface to the CPU registers. The debugger uses methods `GetName`, `GetValue`, `GetSize` and `SetValue` of that interface to access to CPU registers.

Name: `symbol_table_lookup_import`
Mandatory connected: no
Interface:
`unisim::service::interfaces ↪::SymbolTableLookup`

Description:
This service import should be connected to a symbol table. The debugger calls method `FindSymbol`, `FindSymbolByAddr`, `FindSymbolByName` through this service import to translate addresses to symbols and vice-versa.

2.4.4 GDB server

The GDB server service emulates the GDB remote serial protocol over TCP/IP (see *Debugging with GDB*, Appendix D. *GDB Remote serial protocol*), so that a GDB client can connect to the simulator and debug the target program as if it were run on the real hardware. This service uses an architecture XML description file defined by the `architecture-description-filename` run-time parameter (see table below). A sample configuration file for a dummy `XYZ` big-endian architecture, with four 32-bit general purpose registers named `r0`, `r1`, `r2`, `r3` and a program counter named `pc` would be the following:

```
<architecture name="XYZ" endian="big">
    <program_counter name="pc"/>
    <register name="r0" size="4"/>
    <register name="r1" size="4"/>
    <register name="r2" size="4"/>
    <register name="r3" size="4"/>
</architecture>
```

Table below summarizes the GDB server service API:

Service GDB Server	
Class Name:	**Header:**
`unisim::service::debug` ↪`::gdb_server::GDBServer`	`unisim/service/debug` ↪`/gdb_server/gdb_server.hh`
Description: The GDB server service allows debugging a software running on a simulated hardware by connecting (over TCP/IP) a GDB client to it (and thus to the simulator). The GDB client can be either the standard text based client (i.e. command `gdb`), a graphical front-end to GDB (e.g. `ddd`), or even Eclipse CDT. The GDB server service directly speaks to the GDB serial remote protocol (over TCP/IP), so that a GDB client can connect (over TCP/IP) to the simulator using GDB command `target remote`. The GDB server service may be connected to a CPU module.	
Template Parameters	
Name: `ADDRESS`	**Type:** `class`
Default value: none	
Description: This is the C++ type of a memory address (e.g. uint32_t or uint64_t).	
Run-Time Parameters	
Name: `tcp-port`	**Type:** `int`
Default value: `12345`	
Description: The TCP port used by GDB server service to communicate with the GDB client.	
Name: `architecture-description` ↪`-filename`	**Type:** `string`
Default value: empty string	

Description:
The path to the architecture description file that the GDB server service must use. The description file provides retargetability to the GDB server service. The following files brings support of the ARM, PowerPC, TMS320C3X and HCS12X processors to the GDB server service:

- `unisim/service/debug/gdb_server/gdb_armv4l.xml`
- `unisim/service/debug/gdb_server/gdb_armv5b.xml`
- `unisim/service/debug/gdb_server/gdb_powerpc.xml`
- `unisim/service/debug/gdb_server/gdb_tms320c3x.xml`
- `unisim/service/debug/gdb_server/gdb_hcs12x.xml`

Service Exports

Name: `debug_control_export`
Interface: `unisim::service::interfaces ↪::DebugControl<ADDRESS>`

Description:
This service export should be connected to a CPU module. The CPU module calls method `FetchDebugCommand` through its service import to leave control to the debugger and fetch a new debug command.

Name: `memory_access_reporting_export`
Interface: `unisim::service::interfaces ↪MemoryAccessReporting<ADDRESS>`

Description:
This service export should be connected to a CPU module. The CPU module calls methods `ReportMemoryAccess` and `ReportFinishedInstruction` through its service import. This allows the debugger to spy memory accesses and thus handle breakpoints and watchpoints.

Name: `trap_reporting_export`
Interface: `unisim::service::interfaces ↪::TrapReporting`

Description:
This service export should be connected to a CPU module. A CPU module calls method `ReportTrap` through its service import. This allows the debugger to break execution on the simulated CPU when a trap condition is detected by the CPU module.

Service Imports

Name: `memory_import`
Mandatory connected: yes
Interface: `unisim::service::interfaces ↪::Memory<ADDRESS>`

Description:
This service import should be connected to a CPU or a memory module. The debugger uses this service import to access to memory using methods `ReadMemory` and `WriteMemory`.

Name: `memory_access_reporting` ↪`_control_import`	**Interface:** `unisim::service::interfaces` ↪`::MemoryAccessReportingControl`
Mandatory connected: no	

Description:
This service import should be connected to a CPU module. The debugger calls methods `RequiresMemoryAccessReporting` and `RequiresFinishedInstructionReporting` through this service import to enable/disable memory access reporting from the CPU module.

Name: `registers_import`	**Interface:** `unisim::service::interfaces` ↪`::Registers`
Mandatory connected: yes	

Description:
This service import should be connected to a CPU module. The debugger calls method `GetRegister` through this service import to get an interface to the CPU registers. The debugger uses methods `GetName`, `GetValue`, `GetSize` and `SetValue` of that interface to access to CPU registers.

2.4.5 Built-in Logger

UNISIM provides you a centralized log system to debug modules and simulators. It should be used to show all debug messages, instead of using the traditional C++ stream output mechanism (`cerr` and `cout`). However, as you will see below the UNISIM log system works much like the C++ stream output mechanism.

It provides the following advantages:

- Categorization: messages can be categorized on information, warning and error messages
- Atomic messages: messages will not be mixed (something which happens when programming concurrent/parallel systems like UNISIM/SystemC)
- Multiple outputs: your messages can be written simultaneously to different outputs, for example:
  - console (error output or standard output)
  - raw file
  - XML formatted file
  - ...
- Simple configuration: the log system configuration is integrated to the UNISIM parameter mechanism provided by UNISIM service, see 1.8.

To use the UNISIM logger you need to include unisim/kernel/logger/logger.hh and declare that you are using the unisim::kernel::logger namespace:

```
1 #include "unisim/kernel/logger/logger.hh"
2
3 using namespace unisim::kernel::logger;
```

The logger can only be used by UNISIM objects, that is, classes that inherit from `unisim::kernel::service::Object`. So if you want to use the UNISIM log system your class must inherit from a UNISIM object.

```
1 class MyObject : public Object {
2     ...
3 };
```

You will need to create a member variable of the `unisim::kernel::logger::Logger` type. And at the construction of your object use its default constructor `Logger(const unisim::kernel::service::Object &obj)`. For example:

```
 1 #include "unisim/kernel/service/service.hh"
 2 #include "unisim/kernel/logger/logger.hh"
 3
 4 using unisim::kernel::service::Object;
 5 using namespace unisim::kernel::logger;
 6
 7 class MyObject : public Object {
 8 private:
 9     Logger logger;
10
11 public:
12     MyObject(const char *name, const Object *parent = 0) :
13             Object(name, parent),
14             logger(*this) {
15         ...
16     }
17 };
```

Once you have initialized your member logger variable you can start using it in your class methods. Basically it works like an standard C++ output stream, with the `<<` operator. However, it requires that you indicate when a message starts and ends, and its category (information, warning or error) with the following keywords:

- `DebugInfo` and `EndDebugInfo` to start and end an information message
- `DebugWarning` and `EndDebugWarning` to start and end a warning message
- `DebugError` and `EndDebugError` to start and end an error message

You can use the keyword `EndDebug` instead of `EndDebugInfo`, `EndDebugWarning` or `EndDebugWarning` to indicate that a message ends. The log system will automatically decide which kind of message you are ending. Between the start and the end of a message you can use the logger as a normal C++ output stream. Some of examples of its use:

```
 1 /* displaying an information message */
 2 logger << DebugInfo << "This is an information message" << EndDebugInfo;
 3
 4 /* displaying an information message using   */
 5 /* the EndDebug keyword to close the message */
 6 logger << DebugInfo << "This is an information message" << EndDebug;
 7
 8 /* displaying a warning message written in multiple steps */
 9 logger << DebugWarning << "This is the start of a warning message" << endl;
10 logger << "This is the end of the warning message." << EndDebug;
11
12 /* displaying an error message using variables */
13 unsigned int error = 25;
14 logger << DebugError << "This is an error message using variable ërror¨ with value "
15        << error << EndDebug;
```

2.5 Utility classes

The utility classes source code is in `unisim/util`.

2.5.1 Arithmetic and Logical helper functions

These functions located in `unisim/util/arithmetic` implement fast integer arithmetic computations (assembly on i386 machines):

- Full Adders (8, 16, 32, and 64 bits)
- Full Substractors (8, 16, 32, and 64 bits)
- Full Adders with signed saturation (8, 16, 32, and 64 bits)
- Full Substractors with signed saturation (8, 16, 32, and 64 bits)
- Specific Adders (e.g. reversed carry propagation adder)
- Rotates (left, right, through an additional virtual bit, with bit in, with bit out)
- Logical Shifts (left, right, through an additional virtual bit, with bit in, with bit out)
- Arithmetic Shifts (left, right, an additional virtual bit, with, with bit in, with bit out)
- Bit Scanning (from left to right, and from right to left)
- Base 2 Logarithm
- 2's complement sign Extension

2.5.2 Debugging support

Directory `unisim/util/debug` provides several C++ classes that support:

- Symbol management (symbol table)
- Profile to keep software activity during a run (e.g. in inline debugger service)
- Breakpoint/watchpoint registry
- Register debugging support
- Network stub for implementing a fake device, remotely control the simulator, and cosimulate with another external simulation environment

2.5.3 Endianness support

Directory `unisim/util/endian` provides support for fast endian conversion (assembly on i386 machines).

2.5.4 Hash Table

Directory `unisim/util/hash_table` provides support for fast table lookup (e.g. for memory).

2.5.5 XML

Directory `unisim/util/xml` provides support for bare XML file (e.g. for GDB server service).

3 Validation guide

3.1 Setup

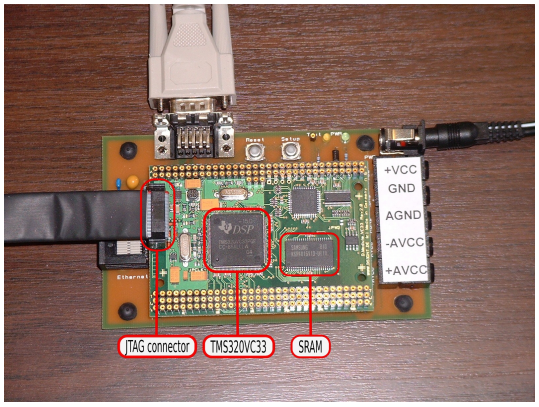

Figure 8: TMS320VC33 Board.

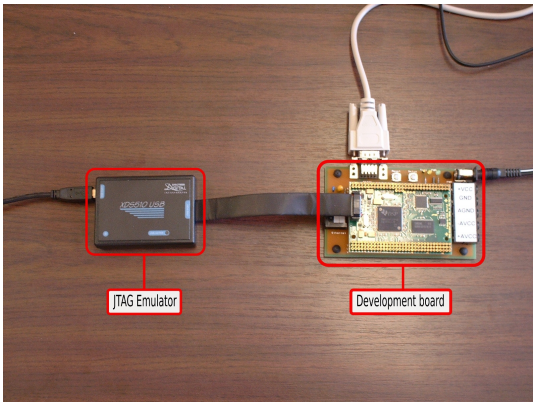

Figure 9: TMS320VC33 board with JTAG.

The simulator validation involved using the TI C cross-compiler for Windows (see Section 1.5) with the following versions:

- TMS320C3x/4x ANSI C Compiler Version 5.11
- TMS320C3x/4x ANSI C Optimizer Version 5.13
- TMS320C3x/4x ANSI C Code Generator Version 5.13
- TMS320C3x/4x COFF Assembler Version 5.12
- TMS320C3x/4x COFF Linker Version 5.11

The cross-compiler has generated COFF v2 files for TMS320C3X/C4X with little-endian headers matching the endianness of our building host machine (Windows XP x86).
The host machine configurations used to test both compilation and run of the simulator are:

- Redhat Linux RHEL4 x86/gcc 3.4.6 (32-bit little-endian machine)
- Mandriva Linux 2009.1 x86/gcc 4.3.2 (32-bit little-endian machine)
- Mandriva Linux 2010.0 x86/gcc 4.4.1 (32-bit little-endian machine)
- Mageia Linux 3/gcc 4.7.2 (32-bit little-endian machine)
- Ubuntu Linux 7.04 powerpc/gcc 4.1.2 (32-bit big-endian machine)
- Ubuntu Linux 9.04 AMD64/x86_64/gcc 4.3.3 (64-bit little-endian machine)
- Ubuntu Linux 10.04 AMD64/x86_64/gcc 4.4.3 (64-bit little-endian machine)
- Mac OS X Leopard v10.5 x86/gcc 4.3.3 and gcc 4.4.2 (32-bit little-endian machine)
- Windows XP x86/gcc mingw32 4.4.0 (32 bit little-endian machine)

Note that the UNISIM TMS320C3X has also been run under the control of `valgrind` (`http://valgrind.org`), a tool tracking memory related bugs such as memory leaks, unitialized memory reads, and control statements that depends on unitialized variables.

The following developement board (see Figures 8 and 9) has been used to compare the simulator results against a real TMS320C3X DSP:

- A D.SignT DK.VC33-150-S2 development board
- A D.SignT D.Module.VC33-150-S2 module including a TI TMS320VC33PGE (150 MFLOPS)
- A 256K 32 bits SRAM with 1 Wait state
- A 512K 8 bits Flash Memory
- A Spectrum Digital XDS510USB JTAG Emulator
- Code Composer IDE 4.10.36 SP2 C3X'C4X for Windows

The UNISIM TMS320C3X has been validated using integer benchmarks, floating point benchmarks, and unit tests of individual instructions. These tests and benchmarks are available for download at `http://unisim-vp.org/site/download.html`. The next sections provide details about the validation process.

3.2 Benchmarks

This section presents the validation process of UNISIM TMS320C3X simulator using some application benchmarks. For that purpose, several integer benchmarks have been ported from the MiBench benchmark suite to the TMS320C3X compiler tool chain. The floating-point benchmarks have been extracted from the TMS320C3x DSK Software. The following document has been used for selecting these floating-point benchmarks:

- *TMS320C3x General Purpose Applications User's Guide* (SPRU194, January 1998)

These benchmarks have been run into the UNISIM TMS320C3X simulator using the application profiling capability of the inline debugger:

```
$ unisim-tms320c3x-2.0 -c sim_config.xml -s enable-inline-debugger=true
Loading xml parameters from: sim_config.xml
Parameters set using file "sim_config.xml"
....

....
loader: Loading symbol table
loader: Loading string table
ti-c-io: TI C I/O support is enabled
ti-c-io: Using __CIOBUF_ at 0xeac00 as I/O buffer
ti-c-io: Installing emulator breakpoint (SWI) for I/O at 0x2003c34 (symbol C$$IO$$)
ti-c-io: Installing emulator breakpoint (SWI) for EXIT at 0x20001cc (symbol C$$EXIT)
Starting simulation at system privilege level
0x00800048 <_c_int00>:
0x00800048:0x08700080 LDP @0x800000
inline-debugger> profile program on
inline-debugger> break C$$EXIT
inline-debugger> continue
...
0x00800073 <C$$EXIT>:
0x00800073:0x66000000 SWI
inline-debugger> profile program
0x00800000 <_enable_insn_cache>:1 times:0x08750800 LDI 2048, ST
0x00800001 <.firm+0x1>:1 times:0x78800000 RETSU <@0x8012a9>
....
0x00801279 <_fclose()+0x39>:3 times:0x0e240000 POP R4
```

```
0x0080127a <_fclose()+0x3a>:3 times:0x18740002 SUBI 2, SP
0x0080127b <_fclose()+0x3b>:3 times:0x68000001 BU R1 <_exit()+0x12>
inline-debugger> quit
```

We extracted the instruction coverage from these applications profiles. The table below shows the instruction coverage for each benchmark. Benchmarks `Fibo`, `Quick sort`, `CRC32`, `Rijndael`, `Sha`, and `ADPCM` are integer benchmarks written in C. Benchmarks `LP`, `BP`, `IIR`, and `FFT` are floating-point benchmarks written in C and assembly. Each general addressing mode (see Table 9) of each TMS320C3X instruction (see Tables 1, 2, 3, 4, 5, 6 and 7), actually has a row in the table. A tick into a cell at the intersection of a row and a column indicates that the instruction of that row is covered by the benchmark of that column.

Although the integer benchmarks (written in C) have been selected to address the digital signal processing application domain, they have only validated the general operations of the simulator: program loading, program debugging, basic integer computation, control flow instructions, basic addressing modes ... The reason of that limited validation scope is that the C compiler does not generate many different instructions and addressing modes among the integer benchmarks. For instance, unusual addressing modes as the indirect addressing with circular modify and indirect addressing with bit reversed modify were not generated at all by the C compiler. Thus solely relying on these integer benchmarks for testing integer computation would have resulted in quite poor instruction coverage. For instance, Instructions `addc`, `negb`, `rol`, `rolc`, `ror`, `rorc`, `subrb`, `addc3`, `subb3`, and most of parallel instructions but `ldi || ldi`, `ldi || sti`, and `sti || sti` were not covered at all. Although the case of floating point benchmarks (written in C and assembly) is similar with an incomplete instruction coverage, the use of assembly has improved coverage of addressing modes and parallel instructions. For instance, benchmarks `IIR` and `FFT` cover indirect addressing with circular modify and indirect addressing with bit reversed modify. Nevertheless floating-point benchmarks insufficiently cover parallel instructions. Particularly, Instructions `absf || stf`, `fix || sti`, `float || sti`, `negf || stf` were not covered at all.

Instruction	Fibonacci	Quick sort	CRC32	Rijndael	Sha	ADPCM coder	ADPCM decoder	DCT/Quantization	LP	BP	IIR	FFT
lde												
`lde reg, reg`												√
`lde dir, reg`												
`lde indir, reg`												
`lde imm, reg`												√
ldf												
`ldf reg, reg`											√	√
`ldf dir, reg`												
`ldf indir, reg`									√	√	√	√
`ldf imm, reg`									√	√		√
ldf*cond*												
`ldf`*`cond`* `reg, reg`									√	√	√	√
`ldf`*`cond`* `dir, reg`									√	√	√	√
`ldf`*`cond`* `indir, reg`										√	√	√
`ldf`*`cond`* `imm, reg`									√	√	√	√

Instruction	Fibonacci	Quick sort	CRC32	Rijndael	Sha	ADPCM coder	ADPCM decoder	DCT/Quantization	LP	BP	IIR	FFT
ldi												
`ldi reg, reg`	√	√	√	√	√	√	√	√	√		√	√
`ldi dir, reg`	√	√	√	√	√	√	√	√	√	√	√	√
`ldi indir, reg`		√	√	√	√	√	√		√	√	√	√
`ldi imm, reg`	√	√	√	√	√	√	√	√	√		√	√
ldi*cond*												
`ldi`*cond* `reg, reg`	√	√	√	√	√	√	√	√	√	√	√	√
`ldi`*cond* `dir, reg`	√	√	√	√	√	√	√	√	√	√	√	√
`ldi`*cond* `indir, reg`	√	√	√	√	√	√	√	√	√	√	√	√
`ldi`*cond* `imm, reg`	√	√	√	√	√	√	√	√	√	√	√	√
ldm												
`ldm reg, reg`												
`ldm dir, reg`												
`ldm indir, reg`												
`ldm imm, reg`												√
ldp												
`ldp src`	√	√	√	√	√	√	√	√	√	√	√	√
pop												
`pop reg`	√	√	√	√	√	√	√	√	√	√	√	√
popf												
`popf reg`				√	√	√	√	√	√	√	√	√
push												
`push reg`	√	√	√	√	√	√	√	√	√	√	√	√
pushf												
`pushf reg`	√	√	√	√	√	√	√	√	√	√	√	√
stf												
`stf reg, dir`											√	
`stf reg, indir`									√	√	√	√
sti												
`sti reg, dir`	√	√	√	√	√	√	√	√	√	√	√	√
`sti reg, indir`	√	√	√	√	√	√	√	√	√	√	√	√
ldfi												
`ldfi dir, reg`												
`ldfi indir, reg`												
ldii												
`ldii dir, reg`												
`ldii indir, reg`												
sigi												
`sigi`												
stfi												
`stfi reg, dir`												

Instruction	Fibonacci	Quick sort	CRC32	Rijndael	Sha	ADPCM coder	ADPCM decoder	DCT/Quantization	LP	BP	IIR	FFT
`stfi reg, indir`												
stii												
`stii reg, dir`												
`stii reg, indir`												
b*cond*												
`b`*`cond`* `reg`	√	√	√	√	√	√	√	√				√
`b`*`cond`* `disp`	√	√	√	√	√	√	√	√	√	√	√	√
b*cond*d												
`b`*`cond`*`d reg`	√	√	√	√	√	√	√	√				√
`b`*`cond`*`d disp`	√	√	√	√	√	√	√	√	√	√	√	√
br												
`br src`												
brd												
`brd src`												
call												
`call src`	√	√	√	√	√	√	√	√		√	√	√
call*cond*												
`call`*`cond`* `reg`	√	√	√	√	√	√	√	√	√	√	√	√
`call`*`cond`* `disp`									√			√
db*cond*												
`db`*`cond`* `ar`$_n$`, reg`												
`db`*`cond`* `ar`$_n$`, disp`				√				√				√
db*cond*d												
`db`*`cond`*`d ar`$_n$`, reg`												
`db`*`cond`*`d ar`$_n$`, disp`								√	√	√		√
iack												
`iack dir`												
`iack indir`												
idle												
`idle`												
nop												
`nop reg`	√	√	√	√	√	√	√	√	√	√	√	√
`nop indir`												
reti*cond*												
`reti`*`cond`*												
rets*cond*												
`rets`*`cond`*	√	√	√	√	√	√	√	√	√	√	√	√
rptb												
`rptb src`	√	√	√	√	√	√	√	√	√	√	√	√
rpts												
`rpts reg`		√	√	√	√	√	√		√	√	√	√

Instruction	Fibonacci	Quick sort	CRC32	Rijndael	Sha	ADPCM coder	ADPCM decoder	DCT/Quantization	LP	BP	IIR	FFT
`rpts dir`												
`rpts indir`												
`rpts imm`												
swi												
`swi`												
trap*cond*												
`trap`*`cond`* `n`												
absf												
`absf reg, reg`									√			√
`absf dir, reg`												
`absf indir, reg`												
`absf imm, reg`												
absi												
`absi reg, reg`		√						√	√	√	√	√
`absi dir, reg`												
`absi indir, reg`												
`absi imm, reg`												
addc												
`addc reg, reg`												
`addc dir, reg`												
`addc indir, reg`												
`addc imm, reg`												
addf												
`addf reg, reg`									√	√		√
`addf dir, reg`									√			√
`addf indir, reg`												√
`addf imm, reg`												
addi												
`addi reg, reg`	√	√	√	√	√	√	√	√	√	√	√	√
`addi dir, reg`	√	√	√	√	√	√	√	√	√	√	√	√
`addi indir, reg`	√	√	√	√	√	√	√	√	√	√	√	√
`addi imm, reg`	√	√	√	√	√	√	√	√	√	√	√	√
and												
`and reg, reg`	√	√	√	√	√	√	√	√	√	√	√	√
`and dir, reg`				√							√	
`and indir, reg`	√	√	√	√	√	√	√	√	√	√		√
`and imm, reg`	√	√	√	√	√	√	√	√	√	√	√	√
andn												
`andn reg, reg`									√	√	√	√
`andn dir, reg`												
`andn indir, reg`												

Instruction	Fibonacci	Quick sort	CRC32	Rijndael	Sha	ADPCM coder	ADPCM decoder	DCT/Quantization	LP	BP	IIR	FFT
`andn imm, reg`	√	√	√	√	√	√	√	√	√	√	√	√
ash												
`ash reg, reg`												
`ash dir, reg`												
`ash indir, reg`												
`ash imm, reg`	√	√	√	√	√	√	√	√	√	√	√	√
cmpf												
`cmpf reg, reg`									√	√	√	√
`cmpf dir, reg`												
`cmpf indir, reg`												
`cmpf imm, reg`									√	√	√	√
cmpi												
`cmpi reg, reg`	√	√	√	√	√	√	√	√	√	√	√	√
`cmpi dir, reg`	√	√	√	√	√	√	√	√	√	√	√	√
`cmpi indir, reg`	√	√	√	√	√	√	√	√	√	√	√	√
`cmpi imm, reg`	√	√	√	√	√	√	√	√	√	√	√	√
fix												
`fix reg, reg`									√	√	√	√
`fix dir, reg`												
`fix indir, reg`												
`fix imm, reg`												
float												
`float reg, reg`	√	√	√	√	√	√	√	√	√	√	√	√
`float dir, reg`												√
`float indir, reg`												
`float imm, reg`												
lsh												
`lsh reg, reg`	√	√	√	√	√	√	√	√	√	√	√	
`lsh dir, reg`				√				√				
`lsh indir, reg`												
`lsh imm, reg`	√	√	√	√	√	√	√	√	√	√	√	√
mpyf												
`mpyf reg, reg`									√	√	√	√
`mpyf dir, reg`									√			√
`mpyf indir, reg`											√	
`mpyf imm, reg`												√
mpyi												
`mpyi reg, reg`	√	√	√	√	√	√	√	√	√	√		√
`mpyi dir, reg`												
`mpyi indir, reg`												
`mpyi imm, reg`												

Instruction	Fibonacci	Quick sort	CRC32	Rijndael	Sha	ADPCM coder	ADPCM decoder	DCT/Quantization	LP	BP	IIR	FFT
negb												
`negb reg, reg`												
`negb dir, reg`												
`negb indir, reg`												
`negb imm, reg`												
negf												
`negf reg, reg`									√	√	√	√
`negf dir, reg`												
`negf indir, reg`												
`negf imm, reg`												
negi												
`negi reg, reg`	√	√	√	√	√	√	√	√	√	√	√	√
`negi dir, reg`												
`negi indir, reg`												
`negi imm, reg`												
norm												
`norm reg, reg`									√	√		√
`norm dir, reg`												
`norm indir, reg`												
`norm imm, reg`												
not												
`not reg, reg`			√						√			√
`not dir, reg`											√	
`not indir, reg`												
`not imm, reg`												
or												
`or reg, reg`												
`or dir, reg`												
`or indir, reg`	√	√	√	√	√	√	√	√	√	√	√	√
`or imm, reg`		√	√	√	√	√	√	√	√	√	√	√
rnd												
`rnd reg, reg`									√	√	√	√
`rnd dir, reg`												
`rnd indir, reg`												
`rnd imm, reg`												
rol												
`rol reg`												
rolc												
`rolc reg`												
ror												
`ror reg`												

Instruction	Fibonacci	Quick sort	CRC32	Rijndael	Sha	ADPCM coder	ADPCM decoder	DCT/Quantization	LP	BP	IIR	FFT
rorc												
`rorc reg`												
subb												
`subb reg, reg`												
`subb dir, reg`												
`subb indir, reg`												
`subb imm, reg`												
subc												
`subc reg, reg`	√	√	√	√	√	√	√	√	√	√	√	√
`subc dir, reg`												
`subc indir, reg`												
`subc imm, reg`												
subf												
`subf reg, reg`												√
`subf dir, reg`												
`subf indir, reg`												√
`subf imm, reg`												
subi												
`subi reg, reg`	√	√	√	√	√	√	√	√	√	√	√	√
`subi dir, reg`												
`subi indir, reg`												
`subi imm, reg`	√	√	√	√	√	√	√	√	√	√	√	√
subrb												
`subrb reg, reg`												
`subrb dir, reg`												
`subrb indir, reg`												
`subrb imm, reg`												
subrf												
`subrf reg, reg`									√	√	√	√
`subrf dir, reg`												
`subrf indir, reg`												
`subrf imm, reg`									√			√
subri												
`subri reg, reg`	√	√	√	√	√	√	√	√	√	√	√	√
`subri dir, reg`									√			√
`subri indir, reg`	√	√	√	√	√	√	√	√	√	√		√
`subri imm, reg`	√	√	√	√	√	√	√	√	√	√	√	√
tstb												
`tstb reg, reg`												
`tstb dir, reg`												
`tstb indir, reg`	√	√	√	√	√	√	√	√	√	√	√	√

Instruction	Fibonacci	Quick sort	CRC32	Rijndael	Sha	ADPCM coder	ADPCM decoder	DCT/Quantization	LP	BP	IIR	FFT
`tstb imm, reg`	√	√	√	√	√	√	√	√	√	√	√	√
xor												
`xor reg, reg`									√	√		
`xor dir, reg`												
`xor indir, reg`				√	√							
`xor imm, reg`												
addc3												
`addc3 reg, reg, reg`												
`addc3 indir, reg, reg`												
`addc3 reg, indir, reg`												
`addc3 indir, indir, reg`												
addf3												
`addf3 reg, reg, reg`									√	√	√	√
`addf3 indir, reg, reg`												
`addf3 reg, indir, reg`												
`addf3 indir, indir, reg`												√
addi3												
`addi3 reg, reg, reg`	√	√	√	√	√	√	√	√	√	√	√	√
`addi3 indir, reg, reg`												
`addi3 reg, indir, reg`	√	√	√	√	√	√	√	√	√	√	√	√
`addi3 indir, indir, reg`	√	√	√	√	√	√	√	√	√	√	√	√
and3												
`and3 reg, reg, reg`				√	√				√	√		
`and3 indir, reg, reg`												
`and3 reg, indir, reg`	√	√	√	√	√	√	√	√	√	√	√	√
`and3 indir, indir, reg`												
andn3												
`andn3 reg, reg, reg`					√							
`andn3 indir, reg, reg`												
`andn3 reg, indir, reg`												
`andn3 indir, indir, reg`												
ash3												
`ash3 reg, reg, reg`	√	√	√	√	√	√	√	√	√	√	√	√
`ash3 indir, reg, reg`												
`ash3 reg, indir, reg`		√	√	√	√	√	√	√	√	√	√	√
`ash3 indir, indir, reg`												
cmpf3												
`cmpf3 reg, reg`									√	√	√	√
`cmpf3 indir, reg`												
`cmpf3 reg, indir`												
`cmpf3 indir, indir`												

Instruction	Fibonacci	Quick sort	CRC32	Rijndael	Sha	ADPCM coder	ADPCM decoder	DCT/Quantization	LP	BP	IIR	FFT
cmpi3												
`cmpi3 reg, reg`	√	√	√	√	√	√	√	√	√	√	√	√
`cmpi3 indir, reg`	√	√	√		√	√	√	√	√	√	√	√
`cmpi3 reg, indir`												
`cmpi3 indir, indir`												
lsh3												
`lsh3 reg, reg, reg`				√	√							
`lsh3 indir, reg, reg`												
`lsh3 reg, indir, reg`	√	√	√	√	√	√	√	√	√	√	√	√
`lsh3 indir, indir, reg`												
mpyf3												
`mpyf3 reg, reg, reg`									√	√	√	√
`mpyf3 indir, reg, reg`											√	√
`mpyf3 reg, indir, reg`												
`mpyf3 indir, indir, reg`											√	√
mpyi3												
`mpyi3 reg, reg, reg`		√	√	√	√	√	√	√	√	√		√
`mpyi3 indir, reg, reg`												
`mpyi3 reg, indir, reg`												
`mpyi3 indir, indir, reg`												
or3												
`or3 reg, reg, reg`	√	√	√	√	√	√	√	√	√	√	√	√
`or3 indir, reg, reg`												
`or3 reg, indir, reg`	√	√	√	√	√	√	√	√	√	√	√	√
`or3 indir, indir, reg`												
subb3												
`subb3 reg, reg, reg`												
`subb3 indir, reg, reg`												
`subb3 reg, indir, reg`												
`subb3 indir, indir, reg`												
subf3												
`subf3 reg, reg, reg`									√			√
`subf3 indir, reg, reg`												√
`subf3 reg, indir, reg`												
`subf3 indir, indir, reg`												√
subi3												
`subi3 reg, reg, reg`	√	√	√	√	√	√	√	√	√	√	√	√
`subi3 indir, reg, reg`	√	√	√	√	√	√	√	√	√	√	√	√
`subi3 reg, indir, reg`	√	√	√	√	√	√	√	√	√	√		√
`subi3 indir, indir, reg`		√	√	√	√	√	√	√	√	√		√
tstb3												

Instruction	Fibonacci	Quick sort	CRC32	Rijndael	Sha	ADPCM coder	ADPCM decoder	DCT/Quantization	LP	BP	IIR	FFT
`tstb3 reg, reg`												
`tstb3 indir, reg`			√		√							
`tstb3 reg, indir`												
`tstb3 indir, indir`												
xor3												
`xor3 reg, reg, reg`		√			√			√	√	√	√	√
`xor3 indir, reg, reg`												
`xor3 reg, indir, reg`			√	√								
`xor3 indir, indir, reg`				√								
absf || stf												
`absf indir, reg || stf reg, indir`												
`absf reg, reg || stf reg, indir`												
absi || sti												
`absi indir, reg || sti reg, indir`												
`absi reg, reg || sti reg, indir`												
addf3 || stf												
`addf3 reg, indir, reg || stf reg, indir`												
`addf3 reg, reg, reg || stf reg, indir`												√
addi3 || sti												
`addi3 reg, indir, reg || sti reg, indir`												
`addi3 reg, reg, reg || sti reg, indir`												
and3 || sti												
`and3 reg, indir, reg || sti reg, indir`												
`and3 reg, reg, reg || sti reg, indir`												
ash3 || sti												
`ash3 count, indir, reg || sti reg, indir`												
`ash3 count, reg, reg || sti reg, indir`												
fix || sti												
`fix indir, reg || sti reg, indir`												
`fix reg, reg || sti reg, indir`												
float || sti												
`float indir, reg || sti reg, indir`												
`float reg, reg || sti reg, indir`												
ldf || ldf												
`ldf indir, reg || ldf indir, reg`												√
`ldf reg, reg || ldf indir, reg`												
ldf || stf												
`ldf indir, reg || stf reg, indir`												√
`ldf reg, reg || stf reg, indir`												
ldi || ldi												
`ldi indir, reg || ldi indir, reg`		√	√	√	√	√	√	√	√	√		√

Instruction	Fibonacci	Quick sort	CRC32	Rijndael	Sha	ADPCM coder	ADPCM decoder	DCT/Quantization	LP	BP	IIR	FFT
`ldi reg, reg || ldi indir, reg`												
ldi || sti												
`ldi indir, reg || sti reg, indir`	√	√	√	√	√	√	√	√	√	√	√	√
`ldi reg, reg || sti reg, indir`												
lsh3 || sti												
`lsh3 count, indir, reg || sti reg, indir`												
`lsh3 count, reg, reg || sti reg, indir`												
mpyf3 || addf3												
`mpyf3 indir, indir, reg || addf3 reg, reg, reg`									√	√	√	√
`mpyf3 indir, reg, reg || addf3 reg, reg, reg`												
`mpyf3 reg, indir, reg || addf3 reg, reg, reg`												
`mpyf3 reg, reg, reg || addf3 reg, reg, reg`												
`mpyf3 indir, reg, reg || addf3 indir, reg, reg`												√
`mpyf3 reg, reg, reg || addf3 indir, reg, reg`												
`mpyf3 reg, reg, reg || addf3 indir, indir, reg`												√
`mpyf3 reg, reg, reg || addf3 reg, indir, reg`												
`mpyf3 indir, reg, reg || addf3 reg, indir, reg`												
mpyf3 || stf												
`mpyf3 indir, reg, reg || stf reg, indir`												√
`mpyf3 reg, reg, reg || stf reg, indir`												√
mpyf3 || subf3												
`mpyf3 indir, indir, reg || subf3 reg, reg, reg`												
`mpyf3 indir, reg, reg || subf3 reg, reg, reg`												
`mpyf3 reg, indir, reg || subf3 reg, reg, reg`												
`mpyf3 reg, reg, reg || subf3 reg, reg, reg`												
`mpyf3 indir, reg, reg || subf3 indir, reg, reg`												
`mpyf3 reg, reg, reg || subf3 indir, reg, reg`												
`mpyf3 reg, reg, reg || subf3 indir, indir, reg`												√
`mpyf3 reg, reg, reg || subf3 reg, indir, reg`												
`mpyf3 indir, reg, reg || subf3 reg, indir, reg`												
mpyi3 || addi3												
`mpyi3 indir, indir, reg || addi3 reg, reg, reg`												
`mpyi3 indir, reg, reg || addi3 reg, reg, reg`												
`mpyi3 reg, indir, reg || addi3 reg, reg, reg`												
`mpyi3 reg, reg, reg || addi3 reg, reg, reg`												
`mpyi3 indir, reg, reg || addi3 indir, reg, reg`												
`mpyi3 reg, reg, reg || addi3 indir, reg, reg`												
`mpyi3 reg, reg, reg || addi3 indir, indir, reg`												
`mpyi3 reg, reg, reg || addi3 reg, indir, reg`												
`mpyi3 indir, reg, reg || addi3 reg, indir, reg`												
mpyi3 || sti												

Instruction	Fibonacci	Quick sort	CRC32	Rijndael	Sha	ADPCM coder	ADPCM decoder	DCT/Quantization	LP	BP	IIR	FFT
`mpyi3 indir, reg, reg || sti reg, indir`												
`mpyi3 reg, reg, reg || sti reg, indir`												
mpyi3 || subi3												
`mpyi3 indir, indir, reg || subi3 reg, reg, reg`												
`mpyi3 indir, reg, reg || subi3 reg, reg, reg`												
`mpyi3 reg, indir, reg || subi3 reg, reg, reg`												
`mpyi3 reg, reg, reg || subi3 reg, reg, reg`												
`mpyi3 indir, reg, reg || subi3 indir, reg, reg`												
`mpyi3 reg, reg, reg || subi3 indir, reg, reg`												
`mpyi3 reg, reg, reg || subi3 indir, indir, reg`												
`mpyi3 reg, reg, reg || subi3 reg, indir, reg`												
`mpyi3 indir, reg, reg || subi3 reg, indir, reg`												
negf || stf												
`negf indir, reg || stf reg, indir`												
`negf reg, reg || stf reg, indir`												
negi || sti												
`negi indir, reg || sti reg, indir`												
`negi reg, reg || sti reg, indir`												
not || sti												
`not indir, reg || sti reg, indir`												
`not reg, reg || sti reg, indir`												
or3 || sti												
`or3 reg, indir, reg || sti reg, indir`												
`or3 reg, reg, reg || sti reg, indir`												
stf || stf												
`stf reg, indir || stf reg, indir`												√
`stf reg, reg || stf reg, indir`												
sti || sti												
`sti reg, indir || sti reg, indir`								√				
`sti reg, reg || sti reg, indir`												
subf3 || stf												
`subf3 reg, indir, reg || stf reg, indir`												
`subf3 reg, reg, reg || stf reg, indir`												√
subi3 || sti												
`subi3 reg, indir, reg || sti reg, indir`												
`subi3 reg, reg, reg || sti reg, indir`												
xor3 || sti												
`xor3 indir, reg, reg || sti reg, indir`												
`xor3 reg, reg, reg || sti reg, indir`												
idle2												
`idle2`												

Instruction	Fibonacci	Quick sort	CRC32	Rijndael	Sha	ADPCM coder	ADPCM decoder	DCT/Quantization	LP	BP	IIR	FFT
lopower												
`lowpower`												
maxspeed												
`maxspeed`												

3.2.1 Fibonacci

This benchmark recursively (and quite inefficiently) computes the Fibonacci numbers:

$$\begin{cases} F_1 = 1 \\ F_2 = 1 \\ F_n = F_{n-2} + F_{n-1} \quad \text{where } n > 2 \end{cases}$$

It has validated general simulator operations such as program loading, stack management and function calls. This benchmark requires the TI C I/O service enabled to run in the TMS320C3X simulator. A precompiled binary (`fibo.out`) is provided together with a GNU Make compatible `Makefile`. A simulation configuration (`sim_config.xml`) for this simulator is also provided, so that the simulator can run the benchmark using the following command:

```
$ unisim-tms320c3x-2.0 -c sim_config.xml
```

The expected output on the screen of the benchmarks is:

```
Fibo(1)=1 (0x1)
Fibo(2)=1 (0x1)
Fibo(3)=2 (0x2)
Fibo(4)=3 (0x3)
Fibo(5)=5 (0x5)
Fibo(6)=8 (0x8)
Fibo(7)=13 (0xd)
Fibo(8)=21 (0x15)
Fibo(9)=34 (0x22)
Fibo(10)=55 (0x37)
Fibo(11)=89 (0x59)
Fibo(12)=144 (0x90)
Fibo(13)=233 (0xe9)
Fibo(14)=377 (0x179)
Fibo(15)=610 (0x262)
Fibo(16)=987 (0x3db)
Fibo(17)=1597 (0x63d)
Fibo(18)=2584 (0xa18)
Fibo(19)=4181 (0x1055)
Fibo(20)=6765 (0x1a6d)
Fibo(21)=10946 (0x2ac2)
Fibo(22)=17711 (0x452f)
Fibo(23)=28657 (0x6ff1)
Fibo(24)=46368 (0xb520)
```

```
Fibo(25)=75025 (0x12511)
Fibo(26)=121393 (0x1da31)
Fibo(27)=196418 (0x2ff42)
Fibo(28)=317811 (0x4d973)
Fibo(29)=514229 (0x7d8b5)
Fibo(30)=832040 (0xcb228)
Fibo(31)=1346269 (0x148add)
Fibo(32)=2178309 (0x213d05)
Fibo(33)=3524578 (0x35c7e2)
Fibo(34)=5702887 (0x5704e7)
```

3.2.2 Quick sort

This benchmark sorts 65536 integer numbers using the quick sort recursive algorithm. It has validated general simulator operations such as program loading, stack management, function calls, comparisons, arrays, and file I/O. The input data set is in file `random.txt` that contains random generated integer numbers. The output data set after the benchmark run is in file `sort.sim.txt`.

This benchmark requires the TI C I/O service enabled to run in the TMS320C3X simulator. A precompiled binary (`quicksort.out`) is provided together with a GNU Make compatible `Makefile`. A simulation configuration (`sim_config.xml`) for this simulator is also provided, so that the simulator can run the benchmark using the following command:

```
$ unisim-tms320c3x-2.0 -c sim_config.xml
```

The expected output data set is in file `sort.ref.txt`.

3.2.3 CRC32 (check sum)

This benchmark is based on CRC32 benchmark from MiBench Version 1.0 (`http://www.eecs.umich.edu/mibench`). It performs a 32-bit Cyclic Redundancy Check (CRC) on a file. CRC checks are often used to detect errors in data transmission. This benchmark has been selected because of its sensitivity to simulator failures. The benchmark reads file `small.pcm` and prints the check sum on the screen

This benchmark requires the TI C I/O service enabled to run in the TMS320C3X simulator. A precompiled binary is provided together with a GNU Make compatible `Makefile`. A simulation configuration (`sim_config.xml`) for this simulator is also provided, so that the simulator can run the benchmark using the following command:

```
$ unisim-tms320c3x-2.0 -c sim_config.xml
```

The expected output on the screen of the benchmarks is in file `ref.txt`:

```
32 BIT ANSI X3.66 CRC checksum:
Opening input file "small.pcm"
.....................................................................................
Total number of bytes read: 1368864
CRC32: 6da5b639
```

3.2.4 Rijndael (encryption/decryption)

This benchmark is based on Rijndael benchmark from MiBench Version 1.0 (`http://www.eecs.umich.edu/mibench`). Rijndael was selected as the National Institute of Standards and Technologies Advanced Encryption Standard (AES). It is a block cipher with the option of 128-,

192-, and 256-bit keys and blocks. This benchmark has been selected because of its sensitivity to simulator failures.

In this benchmark, encryption is followed by decryption so that input data set and output data set should be identical. The benchmark uses this hexadecimal encryption key:

```
1234567890abcdeffedcba09876543211234567890abcdeffedcba0987654321
```

The benchmark reads file `input_small.asc`, and encrypt it into file `output_small.sim.enc`. It decrypts `output_small.sim.enc` into file `output_small.sim.dec`.

This benchmark requires the TI C I/O service enabled to run in the TMS320C3X simulator. A precompiled binary (`rijndael.out`) is provided together with a GNU Make compatible `Makefile`. A simulation configuration (`sim_config.xml`) for this simulator is also provided, so that the simulator can run the benchmark using the following command:

```
$ unisim-tms320c3x-2.0 -c sim_config.xml
```

It is expected that files `input_small.asc` and `output_small.sim.dec` be identical after the benchmark run.

3.2.5 Sha (encryption/decryption)

This benchmark is based on SHA benchmark from MiBench Version 1.0 (`http://www.eecs.umich.edu/mibench`). SHA is the secure hash algorithm that produces a 160-bit message digest from a given input. It is often used in the secure exchange of cryptographic keys and for generating digital signatures. It is also used in the well-known MD4 and MD5 hashing functions. This benchmark has been selected because of its sensitivity to simulator failures.

The benchmark reads its input data set from file `input_small.asc` and prints the SHA digest on the screen.

A precompiled binary (`sha.out`) is provided together with a GNU Make compatible `Makefile`. A simulation configuration (`sim_config.xml`) for this simulator is also provided, so that the simulator can run the benchmark using the following command:

```
$ unisim-tms320c3x-2.0 -c sim_config.xml
```

The expected output on the screen of the benchmark is in file `ref.txt`:

```
NIST Secure Hash Algorithm:
Opening input file "input_small.asc"
Computing SHA digest
SHA digest:
320c22e9 7b1ed440 77d2e55a bbe2481a 2b24a55b
```

3.2.6 ADPCM (sound encoding/decoding)

This benchmark is based on ADPCM benchmark from MiBench Version 1.0 (`http://www.eecs.umich.edu/mibench`). It performs ADPCM encoding/decoding. Adaptive Differential Pulse Code Modulation (ADPCM) is a variation of the well-known standard Pulse Code Modulation (PCM). A common implementation takes 16-bit linear PCM samples and converts them to 4-bit samples, yielding a compression rate of 4:1. The input data are speech samples. This benchmark has been selected because it is a typical application in digital signal processing. The ADPCM coder benchmark reads file `small.pcm` and writes the compressed data in file `output_small.sim.adpcm`. The ADPCM decoder benchmark reads file `small.adpcm` and writes the uncompressed data in file `output_small.sim.pcm`.

This benchmark requires the TI C I/O service enabled to run in the TMS320C3X simulator. Precompiled binaries (`coder.out` and `decoder.out`) are provided together with a GNU Make

compatible `Makefile`. Simulation configurations (`coder_sim_config.xml` and `decoder_sim_config.xml`)) for this simulator are also provided, so that the simulator can run the benchmarks using the following command:

```
$ unisim-tms320c3x-2.0 -c coder_sim_config.xml
$ unisim-tms320c3x-2.0 -c decoder_sim_config.xml
```

The expected output data set of the ADPCM coder benchmark is in file `output_small.ref.adpcm`. The expected output data set of the ADPCM decoder benchmark is in file `output_small.ref.pcm`.

3.2.7 DCT/Quantization (image processing)

This benchmark is based on XVID video codec (`http://www.xvid.org`). The benchmarks has the following steps that are the base of the JPEG lossy image compression standard:

1. Load a Windows 24-bit RGB Bitmap from a `.bmp` file;
2. Convert from RGB to YUV 4:4:4 for each 8x8 pixel blocks;
3. Compute a DCT on each 8x8 pixel blocks;
4. Quantize each 8x8 pixels blocks;
5. Dequantize each 8x8 pixels blocks;
6. Compute an iDCT on each 8x8 pixel blocks;
7. Convert from YUV 4:4:4 to RGB each 8x8 pixel blocks;
8. Save the resulting Windows 24-bit RGB bitmap into a `.bmp` file.

This benchmark has been selectedd because it is a typical application in imaging and digital signal processing. The benchmark reads the input image from file `image.bmp` and the quantization matrix from file `quant_mat.txt`. It save the resulting image in file `output_image.sim.bmp`.

This benchmark requires the TI C I/O service enabled to run in the TMS320C3X simulator. A precompiled binary (`dct_quant.out`) is provided together with a GNU Make compatible `Makefile`. A simulation configuration (`sim_config.xml`) for this simulator is also provided, so that simulator can run the benchmark using the following command:

```
$ unisim-tms320c3x-2.0 -c sim_config.xml
```

The expected output image is in file `output_image.ref.bmp`.

3.2.8 LP (Lowpass Finite Filter)

This benchmark performs the computation of a LowPass Finite (LP) Filter over a digital signal. The LP Filter is programmed in assembler using the *z-transform*, which is widely utilized for the analysis of discrete-time signals, simular to the Laplace transform for continuous-time signals. The implementation is based on the algorithm description provided in "*Digital Signal Processing: Laboratory Experiments Using C and the TMS320C31 DSK*" book by Rulph Chassaing (1999, John Wiley & Sons, Inc.).

The benchmark is mainly written in assembler, and it has been modified to accept an input signal within the "`input_signal.txt`" file, to automatically compute the coefficients depending on the input signal length and to generate an output on the "`output.txt`" file.

This benchmark has been selected to globally check the sequential behavior of floating point instructions and the parallel floating point instructions.

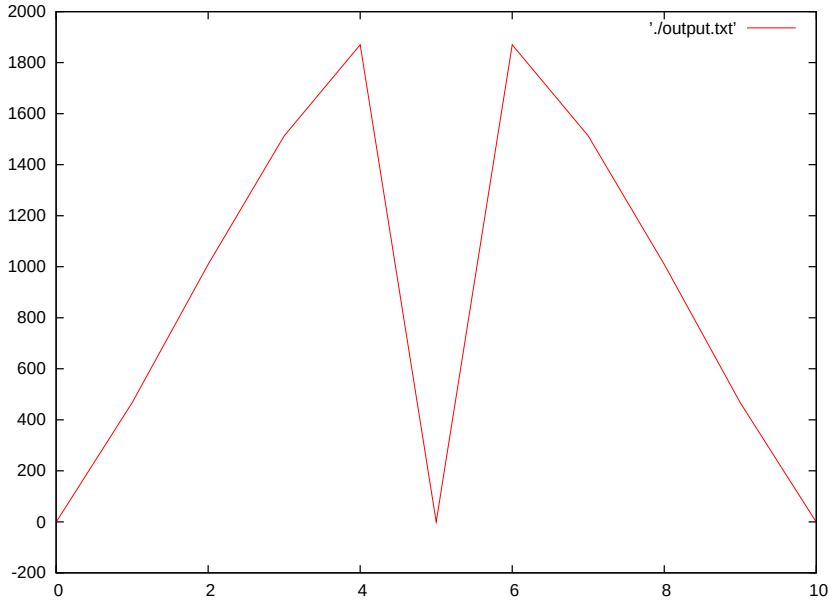

Figure 10: LP (Lowpass Finite Filter) output plot.

This benchmark requires the TI C I/O service enabled to run in the TMS320C3X simulator. A precompiled binary (`bp45.out`) is provided together with a GNU Make compatible Makefile. A simulation configuration file (`sim_config.xml`) for this benchmark is also provided, so that the simulator can run the benchmark with the following command:

```
$ unisim-tms320c3x-2.0 -c sim_config.xml
```

The expected output data set is in the `output.ref.txt` file. You can use plotting tools like *gnuplot* to plot the generated output.

Figure 10 shows the plot of `output.txt` using the following command under *gnuplot*:

```
gnuplot > plot './output.txt' with lines
```

3.2.9 BP (Bandpass Finite Filter)

This benchmark performs the computation of a BandPass Finite (LP) Filter over a digital signal. The LP Filter is programmed in assembler using the *z-transform*, which is widely utilized for the analysis of discrete-time signals, simular to the Laplace transform for continuous-time signals. The implementation is based on the algorithm description provided in "*Digital Signal Processing: Laboratory Experiments Using C and the TMS320C31 DSK*" book by Rulph Chassaing (1999, John Wiley & Sons, Inc.). The benchmark is mainly written in assembler, and it has been modified to accept an input signal (only 45 coefficients are considered) within the "`input_signal.txt`" file, and to generate an output on the "`output.txt`" file.

As for the LowPass filter benchmark, this benchmark has been selected to globally check the sequential behavior of floating point instructions and some parallel floating point instructions.

This benchmark requires the TI C I/O service enabled to run in the TMS320C3X simulator. A precompiled binary (`bp45.out`) is provided together with a GNU Make compatible Makefile. A simulation configuration file (`sim_config.xml`) for this benchmark is also provided, so that the simulator can run the benchmark with the following command:

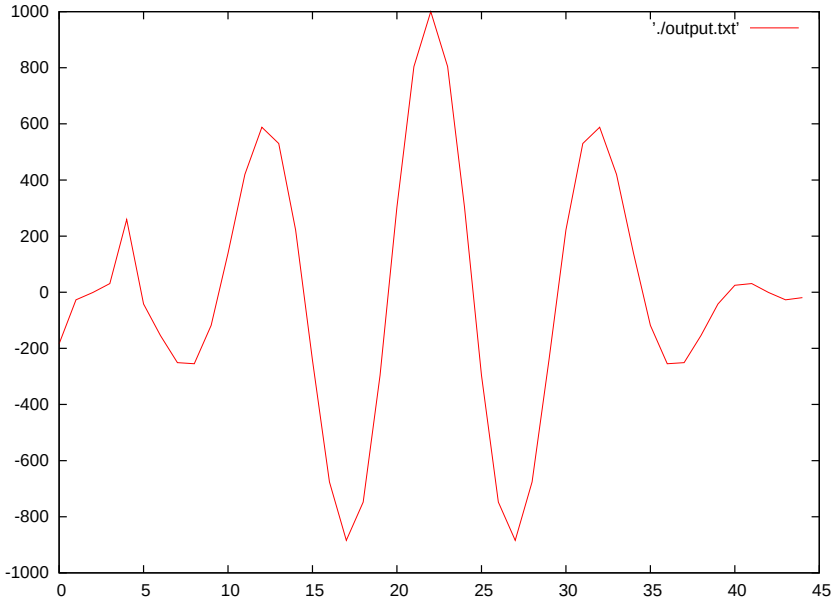

Figure 11: BP (Bandpass Finite Filter) output plot.

```
$ unisim-tms320c3x-2.0 -c sim_config.xml
```

The expected output data set is in the `output.ref.txt` file. You can use plotting tools like *gnuplot* to plot the generated output.

Figure 11 shows the plot of `output.txt` using the following command under *gnuplot*:

```
gnuplot > plot './output.txt' with lines
```

3.2.10 IIR (Biquad Infinite Filter)

This benchmarks performs the computation of an Infinite Impulse Response (IIR) Filter. The previous filter benchmarks (see Sections 3.2.8 and 3.2.9) do not have analog counterpart. This filter benchmark makes use of the vast knowledge already acquired with analog filters. The design procedure involves the conversion of an analog filter to an equivalent discrete filter using the bilinear transformation (BLT) technique. As such, the BLT procedure converts a transfer function of an analog filter in the s-domain into an equivalent discrete-time transfer function in the z-domain.

This benchmark is based on two implementations of the biquad algorithm provided by the TI DSK 3 for the TMS320C3x. The first implementation is done in pure C and uses floating point computation, and the second one is a fast version of the biquad algorithm programmed in assembler. The benchmark provides at the end two different outputs, `b_output.txt` for the C implementation and `fb_output.txt` for the implementation in assembler. Both outputs should be the same.

The IIR benchmark has been selected for the following reasons:

1. It serves to check the correct sequential behavior of programs with an important use of floating point computations.

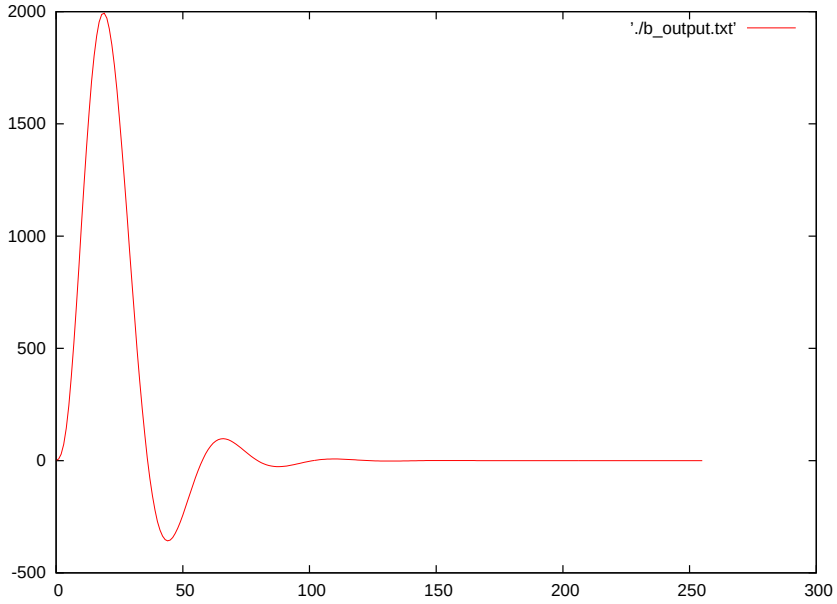

Figure 12: Plot of the IIR program using *gnuplot*.

2. It tests both floating point computations as generated by the TI C compiler and assembler code, which uses specialized instructions as parallel float computations.

3. It tests complex addressing modes and specially the fast biquad implementation uses bit reverse addressing mode.

This benchmark requires the TI C I/O service enabled to run in the TMS320C3X simulator. A precompiled binary (`biquad4.out`) is provided together with a GNU Make compatible Makefile. A simulation configuration file (`sim_config.xml`) for this benchmark is also provided, so that the simulator can run the benchmark with the following command:

```
$ unisim-tms320c3x-2.0 -c sim_config.xml
```

The expected output data set is in the `b_output.ref.txt` and `fb_output.ref.txt` files. You can use plotting tools like *gnuplot* to plot the generated outputs.

Figure 12 shows the plot of `b_output.txt` using the following command under *gnuplot*:

```
gnuplot > plot './b_output.txt' with lines
```

3.2.11 FFT (Fast Fourier Transform)

This benchmark simply computes a 512-point FFT (Fast Fourier Transform) using a Complex Radix 2 given a signal input. It is based on the FFT codes provided by the TI DSK 3 for the TMS320C3x, modified to accept an input signal described as frequency and amplitude in two different input files: *freq_input.txt* (for the frequency) and *ampl_input.txt* (for the amplitude). The benchmarks performs ten FFT iterations over the input signal and generates an output file for each of the iterations (*output*.txt*, where "*" is the iteration number).

The FFT benchmark has been selected for the following reasons:

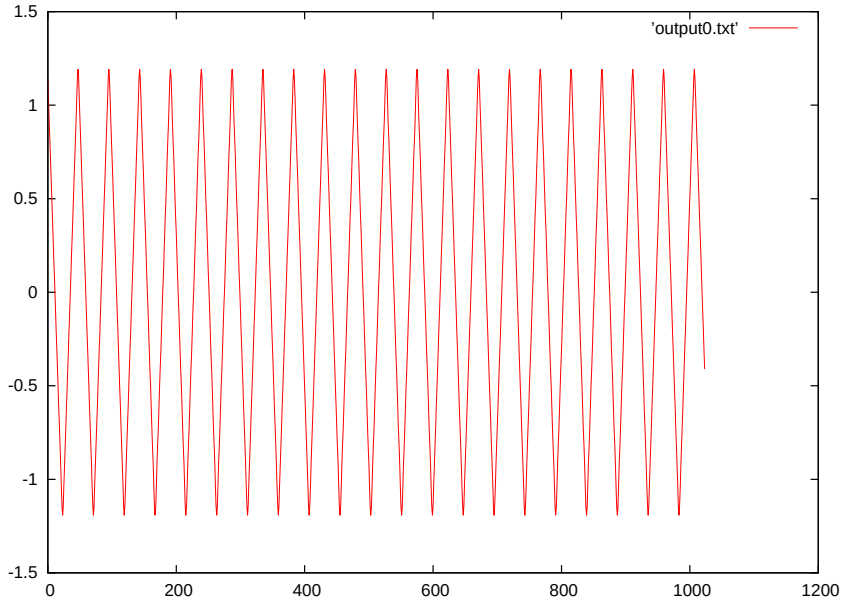

Figure 13: Plot of the FFT512 program first iteration using *gnuplot*.

1. As for the other floating point benchmarks it serves to check the correct sequential behavior of programs with an important use of floating point computations.
2. Most of the program is written in assembler, using parallel float instructions that would otherwise have not been tested by the C compiler.
3. The used FFT assembler implementation uses the bit reverse addressing mode, which is particularly well suited for FFT computations.

This benchmark requires the TI C I/O service enabled to run in the TMS320C3X simulator. A precompiled binary (`fft.out`) is provided together with a GNU Make compatible Makefile. A simulation configuration file (`sim_config.xml`) for this benchmark is also provided, so that the simulator can run the benchmark with the following command:

```
$ unisim-tms320c3x-2.0 -c sim_config.xml
```

The expected output data set is in the following files: `output0.ref.txt`, `output1.ref.txt`, `output2.ref.txt`, `output3.ref.txt`, `output4.ref.txt`, `output5.ref.txt`, `output6.ref.txt`, `output7.ref.txt`, `output8.ref.txt`, and `output9.ref.txt`. You can use plotting tools like *gnuplot* to plot the generated outputs.

Figure 13 shows the plot of `output0.txt` using the following command under *gnuplot*:

```
gnuplot > plot './output0.txt' with lines
```

3.3 Instruction level unit tests

As explained in Section 3.2, although they have validated general operations of the UNISIM TMS320C3X simulator, both the integer and floating-point benchmarks have insufficiently covered the TMS320C3X instructions. Extensive testings at the instruction level are essential to

gain greater confidence in the UNISIM TMS320C3X simulator representativity. This section presents the validation process of UNISIM TMS320C3X simulator at the instruction level. A unit testing environment, in the form of a Makefile for GNU Make, has been developed to allow testing individual instructions for the UNISIM TMS320C3X simulator. Testing an instruction involves writing some "glue" code (C and Assembly) around the instruction under test to provide it with the input operands read from the host filesystem, and to save instruction output operands into a file on the host filesystem, so that results of instruction under test can be observed and compared. A unit test generator, that is part of the testing environment, automatically generates that "glue" code, making writing and maintaining the instruction level unit tests easier.

Section 3.3.1 presents the validation process and the test plan. Section 3.3.2 contains the testing status at instruction level of UNISIM TMS320C3X simulator. Section 3.3.3 presents the unit tests generator flow. Section 3.3.4 presents the testing environment and Section 3.3.5 shows how to use it as a regression test for the UNISIM TMS320C3X simulator.

3.3.1 Validation process

For the purpose of validating the UNISIM TMS320C3X simulator, a factorial plan has been established. The factorial plan parameters are:

- The general instruction under test, e.g. `ldf`, see Tables 1, 2, 3, 4, 5, 6 and 7
- The condition code, e.g. `eq` in `ldfeq`, see Table 8
- The general addressing mode (e.g. `indir` in `ldfeq indir, reg`), see Table 9:
  - For an immediate addressing mode, the immediate value
  - For an indirect addressing mode, one of 26 available indirect addressing modes, see Table 10
- The input value set of the instruction, e.g. the value of `indir` memory operand in `ldfeq indir, reg`

Instructions	Description
lde	Load Floating-Point Exponent
ldf	Load Floating-Point Value
ldf*cond*	Load Floating-point Value Conditionally
ldi	Load Integer
ldi*cond*	Load Integer Conditionally
ldm	Load Floating-Point Mantissa
ldp	Load Data-Page Pointer
pop	Pop Integer
popf	Pop Floating-Point Value
push	Push Integer
pushf	Push Floating-Point Value
stf	Store Floating-Point Value
sti	Store Integer

Table 1: TMS320C3X Load/Store Instructions.

Instructions	Description
ldfi	Load Floating-Point Value, Interlocked
ldii	Load Integer, Interlocked
sigi	Signal, Interlocked
stfi	Store Floating-Point Value, Interlocked
stii	Store Integer, Interlocked

Table 2: TMS320C3X Interlocked Instructions.

Instructions	Description
b*cond*	Branch Conditionally (Standard)
b*cond*d	Branch Conditionally (Delayed)
br	Branch Unconditionally (Standard)
brd	Branch Unconditionally (Delayed)
call	Call Subroutine
call*cond*	Call Subroutine Conditionally
db*cond*	Decrement and Branch Conditionally (Standard)
db*cond*d	Decrement and Branch Conditionally (Delayed)
iack	Interrupt Acknowledge
idle	Idle Until Interrupt
nop	No Operation
reti*cond*	Return From Interrupt Conditionally
rets*cond*	Return From Subroutine Conditionally
rptb	Repeat Block
rpts	Repeat Single Instruction
swi	Software Interrupt
trap*cond*	Trap Conditionally

Table 3: TMS320C3X Control Instructions.

Instructions	Description
absf	Absolute Value of Floating-Point
absi	Absolute Value of Integer
addc	Add Integer With Carry
addf	Add Floating-Point Values
addi	Add Integer
and	Bitwise-Logical AND
andn	Bitwise-Logical AND With Complement
ash	Arithmetic Shift
cmpf	Compare Floating-Point Value
cmpi	Compare Integer
fix	Floating-Point-to-Integer Conversion
float	Integer-to-Floating-Point Conversion
lsh	Logical Shift
mpyf	Multiply Floating-Point Value
mpyi	Multiply Integer
negb	Negative Integer With Borrow
negf	Negative Floating-Point Value
negi	Negate Integer
norm	Normalize
not	Bitwise-Logical Complement
or	Bitwise-Logical OR
rnd	Round Floating-Point Value
rol	Rotate Left
rolc	Rotate Left Through Carry
ror	Rotate Right
rorc	Rotate Right Through Carry
subb	Substract Integer With Borrow
subc	Substract Integer Conditionally
subf	Substract Floating-Point Value
subi	Substract Integer
subrb	Substract Reverse Integer With Borrow
subrf	Substract Reverse Floating-Point Value
subri	Substract Reverse Integer
tstb	Test Bit Fields
xor	Bitwise-Exclusive OR

Table 4: TMS320C3X 2-operand Instructions.

Instructions	Description
addc3	Add Integer With Carry, 3-Operand
addf3	Add Floating-Point, 3-Operand
addi3	Add Integer, 3-Operand
and3	Bitwise-Logical AND, 3-Operand
andn3	Bitwise-Logical AND With Complement, 3-Operand
ash3	Arithmetic Shift, 3-Operand
cmpf3	Compare Floating-Point Value, 3-Operand
cmpi3	Compare Integer, 3-Operand
lsh3	Logical Shift, 3-Operand
mpyf3	Multiply Floating-Point Value, 3-Operand
mpyi3	Multiply Integer, 3-Operand
or3	Bitwise-Logical OR, 3-Operand
subb3	Substract Integer With Borrow, 3-Operand
subf3	Substract Floating-Point Value, 3-Operand
subi3	Substract Integer, 3-Operand
tstb3	Test Bit Fields, 3-Operand
xor3	Bitwise-Exclusive OR, 3-Operand

Table 5: TMS320C3X 3-operand Instructions.

Instructions	Description
absf || stf	Parallel absf and stf
absi || sti	Parallel absi and sti
addf3 || stf	Parallel addf3 and stf
addi3 || sti	Parallel addi3 and sti
and3 || sti	Parallel and3 and sti
ash3 || sti	Parallel ash3 and sti
fix || sti	Parallel fix and sti
float || sti	Parallel float and stf
ldf || ldf	Parallel ldf and ldf
ldf || stf	Parallel ldf and stf
ldi || ldi	Parallel ldi and ldi
ldi || sti	Parallel ldi and sti
lsh3 || sti	Parallel lsh3 and sti
mpyf3 || addf3	Parallel mpyf3 and addf3
mpyf3 || stf	Parallel mpyf3 and stf
mpyf3 || subf3	Parallel mpyf3 and subf3
mpyi3 || addi3	Parallel mpyi3 and addi3
mpyi3 || sti	Parallel mpyi3 and sti
mpyi3 || subi3	Parallel mpyi3 and subi3
negf || stf	Parallel negf and stf
negi || sti	Parallel negi and sti
not || sti	Parallel not and sti
or3 || sti	Parallel or3 and sti
stf || stf	Parallel Store Floating-Point Value
sti || sti	Parallel sti and sti
subf3 || stf	Parallel subf3 and stf
subi3 || sti	Parallel subi3 and sti
xor3 || sti	Parallel xor3 and sti

Table 6: TMS320C3X Parallel Instructions.

Instructions	Description
idle2	Low-Power Idle
lopower	Divide Clock by 16
maxspeed	Restore Clock to Regular Speed

Table 7: TMS320C3X Parallel Instructions.

Condition codes (20)	Description
`u`	unconditional
`lo`	lower than
`ls`	lower than or same as
`hi`	higher than
`hs`	higher than or same as
`eq`	equal
`ne`	not equal
`lt`	less than
`le`	less than or equal
`gt`	greater than
`ge`	greater than or equal
`nv`	no overflow
`v`	overflow
`nuf`	no floating-point underflow
`uf`	floating-point underflow
`nlv`	no overflow
`lv`	overflow
`nluf`	no latched floating-point underflow
`luf`	latched floating-point underflow
`zuf`	zero or floating-point underflow

Table 8: Condition codes.

General Addressing Modes (4)	Description
reg	register addressing mode
dir	direct addressing mode
imm	immediate addressing mode
indir	indirect addressing mode (see table)

Table 9: General addressing modes.

Indirect Addressing Modes (26)	**Tests**	**Description**
*++ar$_n$(disp)	*++ar$_n$(1)	indirect addressing with predisplacement add
*-ar$_n$(disp)	*-ar$_n$(1)	indirect addressing with predisplacement subtract
*++ar$_n$(disp)	*++ar$_n$(1)	indirect addressing with predisplacement add and modify
*--ar$_n$(disp)	*--ar$_n$(1)	indirect addressing with predisplacement substract and modify
*ar$_n$++(disp)	*ar$_n$++(1)	indirect addressing with postdisplacement add and modify
*ar$_n$--(disp)	*ar$_n$--(1)	indirect addressing with postdisplacement substract and modify
*ar$_n$++(disp)%	*ar$_n$++(1)% bk $\in$ {4,5}	indirect addressing with postdisplacement add and circular modify
*ar$_n$(disp)%	*ar$_n$(disp)% bk $\in$ {4,5}	indirect addressing with postdisplacement substract and circular modify
*++ar$_n$(ir0)	*++ar$_n$(ir0) ir0 $\in$ [0,15]	indirect addressing with preindex (ir0) add
*-ar$_n$(ir0)	*-ar$_n$(ir0) ir0 $\in$ [0,15]	indirect addressing with preindex (ir0) substract
*++ar$_n$(ir0)	*++ar$_n$(ir0) ir0 $\in$ [0,15]	indirect addressing with preindex (ir0) add and modify
*--ar$_n$(ir0)	*--ar$_n$(ir0) ir0 $\in$ [0,15]	indirect addressing with preindex (ir0) substract and modify
*ar$_n$++(ir0)	*ar$_n$++(ir0) ir0 $\in$ [0,15]	indirect addressing with postindex (ir0) add and modify
*ar$_n$--(ir0)	*ar$_n$--(ir0) ir0 $\in$ [0,15]	indirect addressing with postindex (ir0) substract and modify
*ar$_n$++(ir0)%	*ar$_n$++(ir0)% ir0 $\in$ [0,15] bk $\in$ {4,5}	indirect addressing with postindex (ir0) add and circular modify
*ar$_n$--(ir0)%	*ar$_n$--(ir0)% ir0 $\in$ [0,15] bk $\in$ {4,5}	indirect addressing with postindex (ir0) substract and circular modify
*++ar$_n$(ir1)	*++ar$_n$(ir1) ir1 $\in$ [0,15]	indirect addressing with preindex (ir1) add
*-ar$_n$(ir1)	*-ar$_n$(ir1) ir1 $\in$ [0,15]	indirect addressing with preindex (ir1) substract
*++ar$_n$(ir1)	*++ar$_n$(ir1) ir1 $\in$ [0,15]	indirect addressing with preindex (ir1) add and modify
*--ar$_n$(ir1)	*--ar$_n$(ir1) ir1 $\in$ [0,15]	indirect addressing with preindex (ir1) substract and modify
*ar$_n$++(ir1)	*ar$_n$++(ir1) ir1 $\in$ [0,15]	indirect addressing with postindex (ir1) add and modify
*ar$_n$--(ir1)	*ar$_n$--(ir1) ir1 $\in$ [0,15]	indirect addressing with postindex (ir1) substract and modify
*ar$_n$++(ir1)%	*ar$_n$++(ir1)% ir1 $\in$ [0,15] bk $\in$ {4,5}	indirect addressing with postindex (ir1) add and circular modify
*ar$_n$--(ir1)%	*ar$_n$--(ir1)% ir1 $\in$ [0,15] bk $\in$ {4,5}	indirect addressing with postindex (ir1) substract and circular modify
*ar$_n$	*ar$_n$	indirect addressing
*ar$_n$++(ir0)b	*ar$_n$++(ir0)b ir1 $\in$ [0,15]	indirect addressing with postindex (ir0) add and bit-reversed modify

Table 10: Indirect addressing modes.

This plan results in lot of instruction alternatives being tested (several condition codes and addressing modes). A full exploration of the factorial plan would have resulted in an unreasonable number of unit tests. To limit the number of unit tests and to still achieve a good testing status, the following choices have been done:

- The amount of immediate addressing have been limited because each unit test of immediate addressing results in one program. The following integer values have been tested: `0`, `-1`, `+1`, `-32768`, or `+32767`. These integer values have a special role in most integer computations (neutral element, bound of integer immediate value ...). The following floating-point values have been tested: `0.0`, `1.0`, `-1.0`, `1.5`, `-1.5`, $2.5594 \cdot 10^2$, $7.8125 \cdot 10^{-3}$, $-7.8163 \cdot 10^{-3}$, $-2.56 \cdot 10^2$. These floating-point value have a special role in most floating-point computations (neutral element, smallest/largest positive/negative immediate floating-point values ...).
- Condition codes have been varied exhaustively for conditional load/store and control instructions, see Table 8.
- Each general addressing mode has been tested for load/store, control, 2-operand, 3-operand, and parallel instructions (note: some instructions allow only few of them), see Table 9.
- All of the 26 indirect addressing modes have been tested for load/store instructions and 2-operand instructions (28 tests per instructions), see Table 10.
- Only one (`*ar`$_n$) of the 26 indirect addressing modes have been tested for 3-operand instructions and parallel instructions because testing all combinations of the 26 indirect addressing modes would have resulted in an unreasonable number of unit tests. The rational behind this choice is that the instructions implementations in the UNISIM TMS320C3X simulator share the same source code for the indirect addressing modes.
- 2-operand instructions with register addressing have been tested 10000 times with random inputs.
- 3-operand instructions, load/store instructions, and parallel instructions with register, direct and indirect addressing modes have been tested 100 times with random inputs.
- Additional tests have been written to check `ar`$_n$ update ordering when instruction has several operands with indirect addressing mode, or when `ar`$_n$ is both updated by an indirect addressing mode and the instruction itself.
- Random input integer values have an uniform distribution. Table 12 shows the distribution for the floating point numbers. Some remarkable values (neutral, smallest/largest positive/negative floating-point values ...) have non-null probability of occurrence.

These choices still have resulted in 3757 unit test programs for a total of 694282 unit tests.

3.3.2 Testing status

The table below summarizes the testing status of all instructions. The total number of unit tests is shown and the detail for the computation of that number is explained between parenthesis. 100_{rand} means 100 tests with random inputs. 5_{imm} means 5 tests with immediate addressing. 20_{cond} means 20 tests for each condition codes. 28_{indir} means 28 tests for each 26 indirect addressing mode. 1_{indir} means only `ar`$_n$ indirect addressing mode tested. 28_{isr} means 28 interrupt service routines tested. 1_{ar} means one test for `ar`$_n$ update ordering.

Instruction	Tested?	Description
lde		
`lde reg, reg`	Yes	100 unit tests (100_{rand})
`lde dir, reg`	Yes	100 unit tests (100_{rand})
`lde indir, reg`	Yes	2800 unit tests ($28_{indir} \times 100_{rand}$)
`lde imm, reg`	Yes	5 unit tests (5_{imm})
ldf		
`ldf reg, reg`	Yes	100 unit tests (100_{rand})
`ldf dir, reg`	Yes	100 unit tests (100_{rand})
`ldf indir, reg`	Yes	2800 unit tests ($28_{indir} \times 100_{rand}$)
`ldf imm, reg`	Yes	5 unit tests (5_{imm})
ldf*cond*		
`ldf`*cond* `reg, reg`	Yes	2000 unit tests ($20_{cond} \times 100_{rand}$)
`ldf`*cond* `dir, reg`	Yes	2000 unit tests ($20_{cond} \times 100_{rand}$)
`ldf`*cond* `indir, reg`	Yes	56K unit tests ($28_{indir} \times 20_{cond} \times 100_{rand}$)
`ldf`*cond* `imm, reg`	Yes	100 unit tests (100_{rand})
ldi		
`ldi reg, reg`	Yes	100 unit tests (100_{rand})
`ldi dir, reg`	Yes	100 unit tests (100_{rand})
`ldi indir, reg`	Yes	2900 unit tests ($(28_{indir} + 1_{ar}) \times 100_{rand}$)
`ldi imm, reg`	Yes	5 unit tests (5_{imm})
ldi*cond*		
`ldi`*cond* `reg, reg`	Yes	2000 unit tests ($20_{cond} \times 100_{rand}$)
`ldi`*cond* `dir, reg`	Yes	2000 unit tests ($20_{cond} \times 100_{rand}$)
`ldi`*cond* `indir, reg`	Yes	56K unit tests ($(28_{indir} + 1_{ar}) \times 20_{cond} \times 100_{rand}$)
`ldi`*cond* `imm, reg`	Yes	100 unit tests (100_{rand})
ldm		
`ldm reg, reg`	Yes	100 unit tests (100_{rand})
`ldm dir, reg`	Yes	100 unit tests (100_{rand})
`ldm indir, reg`	Yes	2800 unit tests ($28_{indir} \times 100_{rand}$)
`ldm imm, reg`	Yes	5 unit tests (5_{imm})
ldp		
`ldp src`	Yes	Any benchmark and unit test
pop		
`pop reg`	Yes	Any benchmark and unit test
popf		
`popf reg`	Yes	Any floating-point benchmark and unit test
push		
`push reg`	Yes	Any benchmark and unit test
pushf		
`pushf reg`	Yes	Any floating-point benchmark and unit test
stf		
`stf reg, dir`	Yes	100 unit tests (100_{rand})
`stf reg, indir`	Yes	2800 unit tests ($28_{indir} \times 100_{rand}$)
sti		

Instruction	Tested?	Description
`sti reg, dir`	Yes	100 unit tests (100_{rand})
`sti reg, indir`	Yes	2900 unit tests ($(28_{indir} + 1_{ar}) \times 100_{rand}$)
ldfi		
`ldfi dir, reg`	No	interlocked instruction behavior depends on environment
`ldfi indir, reg`	No	Instruction behavior depends on environment
ldii		
`ldii dir, reg`	No	Instruction behavior depends on environment
`ldii indir, reg`	No	Instruction behavior depends on environment
sigi		
`sigi`	No	Unimplemented. Instruction behavior depends on environment
stfi		
`stfi reg, dir`	No	Instruction behavior depends on environment
`stfi reg, indir`	No	Instruction behavior depends on environment
stii		
`stii reg, dir`	No	Instruction behavior depends on environment
`stii reg, indir`	No	Instruction behavior depends on environment
b*cond*		
`b`*cond* `reg`	Yes	2000 unit tests ($20_{cond} \times 100_{rand}$)
`b`*cond* `disp`	Yes	2000 unit tests ($20_{cond} \times 100_{rand}$)
b*cond*d		
`b`*cond*`d reg`	Yes	2000 unit tests ($20_{cond} \times 100_{rand}$)
`b`*cond*`d disp`	Yes	2000 unit tests ($20_{cond} \times 100_{rand}$)
br		
`br src`	Yes	100 unit tests (100_{rand})
brd		
`brd src`	Yes	100 unit tests (100_{rand})
call		
`call src`	Yes	100 unit tests (100_{rand})
call*cond*		
`call`*cond* `reg`	Yes	2000 unit tests ($20_{cond} \times 100_{rand}$)
`call`*cond* `disp`	Yes	2000 unit tests ($20_{cond} \times 100_{rand}$)
db*cond*		
`db`*cond* `ar`$_n$`, reg`	Yes	2000 unit tests ($20_{cond} \times 100_{rand}$)
`db`*cond* `ar`$_n$`, disp`	Yes	2000 unit tests ($20_{cond} \times 100_{rand}$)
db*cond*d		
`db`*cond*`d ar`$_n$`, reg`	Yes	2000 unit tests ($20_{cond} \times 100_{rand}$)
`db`*cond*`d ar`$_n$`, disp`	Yes	2000 unit tests ($20_{cond} \times 100_{rand}$)
iack		

Instruction	Tested?	Description
`iack dir`	No	Unimplemented. Instruction depends on circuitry
`iack indir`	No	Unimplemented. Instruction depends on circuitry
idle		
`idle`	No	Instruction depends on external environment
nop		
`nop reg`	Yes	State does not change
`nop indir`	Yes	State does not change
reti*cond*		
`reti`*cond*	Yes	2000 unit tests ($20_{cond} \times 100_{rand}$)
rets*cond*		
`rets`*cond*	Yes	2000 unit tests ($20_{cond} \times 100_{rand}$)
rptb		
`rptb src`	Yes	100 unit tests (100_{rand})
rpts		
`rpts reg`	Yes	100 unit tests (100_{rand})
`rpts dir`	Yes	100 unit tests (100_{rand})
`rpts indir`	Yes	100 unit tests ($1_{indir} \times 100_{rand}$)
`rpts imm`	Yes	1 unit tests (1_{imm})
swi		
`swi`	No	Instruction depends on external environment
trap*cond*		
`trap`*cond* `n`	Yes	4800 unit tests ($(20_{cond} \times 1_{isr} \times 100_{rand}) + (1_{cond} \times 28_{isr} \times 100_{rand})$)
absf		
`absf reg, reg`	Yes	10K unit tests ($10K_{rand}$)
`absf dir, reg`	Yes	100 unit tests (100_{rand})
`absf indir, reg`	Yes	2800 unit tests ($28_{indir} \times 100_{rand}$)
`absf imm, reg`	Yes	5 unit tests (5_{imm})
absi		
`absi reg, reg`	Yes	10K unit tests ($10K_{rand}$)
`absi dir, reg`	Yes	100 unit tests (100_{rand})
`absi indir, reg`	Yes	2900 unit tests ($(28_{indir} + 1_{ar}) \times 100_{rand}$)
`absi imm, reg`	Yes	5 unit tests (5_{imm})
addc		
`addc reg, reg`	Yes	10K unit tests ($10K_{rand}$)
`addc dir, reg`	Yes	100 unit tests (100_{rand})
`addc indir, reg`	Yes	2900 unit tests ($(28_{indir} + 1_{ar}) \times 100_{rand}$)
`addc imm, reg`	Yes	5 unit tests (5_{imm})
addf		
`addf reg, reg`	Yes	10K unit tests ($10K_{rand}$)
`addf dir, reg`	Yes	100 unit tests (100_{rand})
`addf indir, reg`	Yes	2800 unit tests ($28_{indir} \times 100_{rand}$)
`addf imm, reg`	Yes	5 unit tests (5_{imm})

Instruction	Tested?	Description
addi		
`addi reg, reg`	Yes	10K unit tests ($10K_{rand}$)
`addi dir, reg`	Yes	100 unit tests (100_{rand})
`addi indir, reg`	Yes	2900 unit tests ($(28_{indir} + 1_{ar}) \times 100_{rand}$)
`addi imm, reg`	Yes	5 unit tests (5_{imm})
and		
`and reg, reg`	Yes	10K unit tests ($10K_{rand}$)
`and dir, reg`	Yes	100 unit tests (100_{rand})
`and indir, reg`	Yes	2900 unit tests ($(28_{indir} + 1_{ar}) \times 100_{rand}$)
`and imm, reg`	Yes	5 unit tests (5_{imm})
andn		
`andn reg, reg`	Yes	10K unit tests ($10K_{rand}$)
`andn dir, reg`	Yes	100 unit tests (100_{rand})
`andn indir, reg`	Yes	2900 unit tests ($(28_{indir} + 1_{ar}) \times 100_{rand}$)
`andn imm, reg`	Yes	5 unit tests (5_{imm})
ash		
`ash reg, reg`	Yes	10K unit tests ($10K_{rand}$)
`ash dir, reg`	Yes	100 unit tests (100_{rand})
`ash indir, reg`	Yes	2900 unit tests ($(28_{indir} + 1_{ar}) \times 100_{rand}$)
`ash imm, reg`	Yes	5 unit tests (5_{imm})
cmpf		
`cmpf reg, reg`	Yes	10K unit tests ($10K_{rand}$)
`cmpf dir, reg`	Yes	100 unit tests (100_{rand})
`cmpf indir, reg`	Yes	2800 unit tests ($28_{indir} \times 100_{rand}$)
`cmpf imm, reg`	Yes	5 unit tests (5_{imm})
cmpi		
`cmpi reg, reg`	Yes	10K unit tests ($10K_{rand}$)
`cmpi dir, reg`	Yes	100 unit tests (100_{rand})
`cmpi indir, reg`	Yes	2900 unit tests ($(28_{indir} + 1_{ar}) \times 100_{rand}$)
`cmpi imm, reg`	Yes	5 unit tests (5_{imm})
fix		
`fix reg, reg`	Yes	10K unit tests ($10K_{rand}$)
`fix dir, reg`	Yes	100 unit tests (100_{rand})
`fix indir, reg`	Yes	2900 unit tests ($(28_{indir} + 1_{ar}) \times 100_{rand}$)
`fix imm, reg`	Yes	5 unit tests (5_{imm})
float		
`float reg, reg`	Yes	10K unit tests ($10K_{rand}$)
`float dir, reg`	Yes	100 unit tests (100_{rand})
`float indir, reg`	Yes	2900 unit tests ($(28_{indir} + 1_{ar}) \times 100_{rand}$)
`float imm, reg`	Yes	5 unit tests (5_{imm})
lsh		
`lsh reg, reg`	Yes	10K unit tests ($10K_{rand}$)
`lsh dir, reg`	Yes	100 unit tests (100_{rand})
`lsh indir, reg`	Yes	2900 unit tests ($(28_{indir} + 1_{ar}) \times 100_{rand}$)
`lsh imm, reg`	Yes	5 unit tests (5_{imm})
mpyf		
`mpyf reg, reg`	Yes	10K unit tests ($10K_{rand}$)

Instruction	Tested?	Description
`mpyf dir, reg`	Yes	100 unit tests (100_{rand})
`mpyf indir, reg`	Yes	2800 unit tests ($28_{indir} \times 100_{rand}$)
`mpyf imm, reg`	Yes	5 unit tests (5_{imm})
mpyi		
`mpyi reg, reg`	Yes	10K unit tests ($10K_{rand}$)
`mpyi dir, reg`	Yes	100 unit tests (100_{rand})
`mpyi indir, reg`	Yes	2900 unit tests ($(28_{indir} + 1_{ar}) \times 100_{rand}$)
`mpyi imm, reg`	Yes	5 unit tests (5_{imm})
negb		
`negb reg, reg`	Yes	10K unit tests ($10K_{rand}$)
`negb dir, reg`	Yes	100 unit tests (100_{rand})
`negb indir, reg`	Yes	2900 unit tests ($(28_{indir} + 1_{ar}) \times 100_{rand}$)
`negb imm, reg`	Yes	5 unit tests (5_{imm})
negf		
`negf reg, reg`	Yes	10K unit tests ($10K_{rand}$)
`negf dir, reg`	Yes	100 unit tests (100_{rand})
`negf indir, reg`	Yes	2800 unit tests ($28_{indir} \times 100_{rand}$)
`negf imm, reg`	Yes	5 unit tests (5_{imm})
negi		
`negi reg, reg`	Yes	10K unit tests ($10K_{rand}$)
`negi dir, reg`	Yes	100 unit tests (100_{rand})
`negi indir, reg`	Yes	2900 unit tests ($(28_{indir} + 1_{ar}) \times 100_{rand}$)
`negi imm, reg`	Yes	5 unit tests (5_{imm})
norm		
`norm reg, reg`	Yes	10K unit tests ($10K_{rand}$)
`norm dir, reg`	Yes	100 unit tests (100_{rand})
`norm indir, reg`	Yes	2800 unit tests ($28_{indir} \times 100_{rand}$)
`norm imm, reg`	Yes	5 unit tests (5_{imm})
not		
`not reg, reg`	Yes	10K unit tests ($10K_{rand}$)
`not dir, reg`	Yes	100 unit tests (100_{rand})
`not indir, reg`	Yes	2900 unit tests ($(28_{indir} + 1_{ar}) \times 100_{rand}$)
`not imm, reg`	Yes	5 unit tests (5_{imm})
or		
`or reg, reg`	Yes	10K unit tests ($10K_{rand}$)
`or dir, reg`	Yes	100 unit tests (100_{rand})
`or indir, reg`	Yes	2900 unit tests ($(28_{indir} + 1_{ar}) \times 100_{rand}$)
`or imm, reg`	Yes	5 unit tests (5_{imm})
rnd		
`rnd reg, reg`	Yes	10K unit tests ($10K_{rand}$)
`rnd dir, reg`	Yes	100 unit tests (100_{rand})
`rnd indir, reg`	Yes	2800 unit tests ($28_{indir} \times 100_{rand}$)
`rnd imm, reg`	Yes	5 unit tests (5_{imm})
rol		
`rol reg`	Yes	10K unit tests ($10K_{rand}$)
rolc		
`rolc reg`	Yes	10K unit tests ($10K_{rand}$)

Instruction	Tested?	Description
ror		
`ror reg`	Yes	10K unit tests ($10K_{rand}$)
rorc		
`rorc reg`	Yes	10K unit tests ($10K_{rand}$)
subb		
`subb reg, reg`	Yes	10K unit tests ($10K_{rand}$)
`subb dir, reg`	Yes	100 unit tests (100_{rand})
`subb indir, reg`	Yes	2900 unit tests ($(28_{indir} + 1_{ar}) \times 100_{rand}$)
`subb imm, reg`	Yes	5 unit tests (5_{imm})
subc		
`subc reg, reg`	Yes	10K unit tests ($10K_{rand}$)
`subc dir, reg`	Yes	100 unit tests (100_{rand})
`subc indir, reg`	Yes	2900 unit tests ($(28_{indir} + 1_{ar}) \times 100_{rand}$)
`subc imm, reg`	Yes	5 unit tests (5_{imm})
subf		
`subf reg, reg`	Yes	10K unit tests ($10K_{rand}$)
`subf dir, reg`	Yes	100 unit tests (100_{rand})
`subf indir, reg`	Yes	2800 unit tests ($28_{indir} \times 100_{rand}$)
`subf imm, reg`	Yes	5 unit tests (5_{imm})
subi		
`subi reg, reg`	Yes	10K unit tests ($10K_{rand}$)
`subi dir, reg`	Yes	100 unit tests (100_{rand})
`subi indir, reg`	Yes	2900 unit tests ($(28_{indir} + 1_{ar}) \times 100_{rand}$)
`subi imm, reg`	Yes	5 unit tests (5_{imm})
subrb		
`subrb reg, reg`	Yes	10K unit tests ($10K_{rand}$)
`subrb dir, reg`	Yes	100 unit tests (100_{rand})
`subrb indir, reg`	Yes	2900 unit tests ($(28_{indir} + 1_{ar}) \times 100_{rand}$)
`subrb imm, reg`	Yes	5 unit tests (5_{imm})
subrf		
`subrf reg, reg`	Yes	10K unit tests ($10K_{rand}$)
`subrf dir, reg`	Yes	100 unit tests (100_{rand})
`subrf indir, reg`	Yes	2800 unit tests ($28_{indir} \times 100_{rand}$)
`subrf imm, reg`	Yes	5 unit tests (5_{imm})
subri		
`subri reg, reg`	Yes	10K unit tests ($10K_{rand}$)
`subri dir, reg`	Yes	100 unit tests (100_{rand})
`subri indir, reg`	Yes	2900 unit tests ($(28_{indir} + 1_{ar}) \times 100_{rand}$)
`subri imm, reg`	Yes	5 unit tests (5_{imm})
tstb		
`tstb reg, reg`	Yes	10K unit tests ($10K_{rand}$)
`tstb dir, reg`	Yes	100 unit tests (100_{rand})
`tstb indir, reg`	Yes	2900 unit tests ($(28_{indir} + 1_{ar}) \times 100_{rand}$)
`tstb imm, reg`	Yes	5 unit tests (5_{imm})
xor		
`xor reg, reg`	Yes	10K unit tests ($10K_{rand}$)
`xor dir, reg`	Yes	100 unit tests (100_{rand})

Instruction	Tested?	Description
`xor indir, reg`	Yes	2900 unit tests $((28_{indir} + 1_{ar}) \times 100_{rand})$
`xor imm, reg`	Yes	5 unit tests (5_{imm})
addc3		
`addc3 reg, reg, reg`	Yes	100 unit tests (100_{rand})
`addc3 indir, reg, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`addc3 reg, indir, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`addc3 indir, indir, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
addf3		
`addf3 reg, reg, reg`	Yes	100 unit tests (100_{rand})
`addf3 indir, reg, reg`	Yes	100 unit tests $(1_{indir} \times 100_{rand})$
`addf3 reg, indir, reg`	Yes	100 unit tests $(1_{indir} \times 100_{rand})$
`addf3 indir, indir, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
addi3		
`addi3 reg, reg, reg`	Yes	100 unit tests (100_{rand})
`addi3 indir, reg, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`addi3 reg, indir, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`addi3 indir, indir, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
and3		
`and3 reg, reg, reg`	Yes	100 unit tests (100_{rand})
`and3 indir, reg, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`and3 reg, indir, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`and3 indir, indir, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
andn3		
`andn3 reg, reg, reg`	Yes	100 unit tests (100_{rand})
`andn3 indir, reg, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`andn3 reg, indir, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`andn3 indir, indir, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
ash3		
`ash3 reg, reg, reg`	Yes	100 unit tests (100_{rand})
`ash3 indir, reg, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`ash3 reg, indir, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`ash3 indir, indir, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
cmpf3		
`cmpf3 reg, reg`	Yes	100 unit tests (100_{rand})
`cmpf3 indir, reg`	Yes	100 unit tests $(1_{indir} \times 100_{rand})$
`cmpf3 reg, indir`	Yes	100 unit tests $(1_{indir} \times 100_{rand})$
`cmpf3 indir, indir`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
cmpi3		
`cmpi3 reg, reg`	Yes	100 unit tests (100_{rand})
`cmpi3 indir, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$

Instruction	Tested?	Description
`cmpi3 reg, indir`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`cmpi3 indir, indir`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
lsh3		
`lsh3 reg, reg, reg`	Yes	100 unit tests (100_{rand})
`lsh3 indir, reg, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`lsh3 reg, indir, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`lsh3 indir, indir, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
mpyf3		
`mpyf3 reg, reg, reg`	Yes	100 unit tests (100_{rand})
`mpyf3 indir, reg, reg`	Yes	100 unit tests $(1_{indir} \times 100_{rand})$
`mpyf3 reg, indir, reg`	Yes	100 unit tests $(1_{indir} \times 100_{rand})$
`mpyf3 indir, indir, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
mpyi3		
`mpyi3 reg, reg, reg`	Yes	100 unit tests (100_{rand})
`mpyi3 indir, reg, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`mpyi3 reg, indir, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`mpyi3 indir, indir, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
or3		
`or3 reg, reg, reg`	Yes	100 unit tests (100_{rand})
`or3 indir, reg, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`or3 reg, indir, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`or3 indir, indir, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
subb3		
`subb3 reg, reg, reg`	Yes	100 unit tests (100_{rand})
`subb3 indir, reg, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`subb3 reg, indir, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`subb3 indir, indir, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
subf3		
`subf3 reg, reg, reg`	Yes	100 unit tests (100_{rand})
`subf3 indir, reg, reg`	Yes	100 unit tests $(1_{indir} \times 100_{rand})$
`subf3 reg, indir, reg`	Yes	100 unit tests $(1_{indir} \times 100_{rand})$
`subf3 indir, indir, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
subi3		
`subi3 reg, reg, reg`	Yes	100 unit tests (100_{rand})
`subi3 indir, reg, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`subi3 reg, indir, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`subi3 indir, indir, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
tstb3		
`tstb3 reg, reg, reg`	Yes	100 unit tests (100_{rand})

Instruction	Tested?	Description
`tstb3 indir, reg, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`tstb3 reg, indir, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`tstb3 indir, indir, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
xor3		
`xor3 reg, reg, reg`	Yes	100 unit tests (100_{rand})
`xor3 indir, reg, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`xor3 reg, indir, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`xor3 indir, indir, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
absf || stf		
`absf indir, reg || stf reg, indir`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`absf reg, reg || stf reg, indir`	Yes	100 unit tests $(1_{indir} \times 100_{rand})$
absi || sti		
`absi indir, reg || sti reg, indir`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`absi reg, reg || sti reg, indir`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
addf3 || stf		
`addf3 reg, indir, reg || stf reg, indir`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`addf3 reg, reg, reg || stf reg, indir`	Yes	100 unit tests $(1_{indir} \times 100_{rand})$
addi3 || sti		
`addi3 reg, indir, reg || sti reg, indir`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`addi3 reg, reg, reg || sti reg, indir`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
and3 || sti		
`and3 reg, indir, reg || sti reg, indir`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`and3 reg, reg, reg || sti reg, indir`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
ash3 || sti		
`ash3 count, indir, reg || sti reg, indir`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`ash3 count, reg, reg || sti reg, indir`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
fix || sti		
`fix indir, reg || sti reg, indir`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`fix reg, reg || sti reg, indir`	Yes	100 unit tests $(1_{indir} \times 100_{rand})$
float || sti		
`float indir, reg || sti reg, indir`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`float reg, reg || sti reg, indir`	Yes	100 unit tests $(1_{indir} \times 100_{rand})$
ldf || ldf		
`ldf indir, reg || ldf indir, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`ldf reg, reg || ldf indir, reg`	Yes	100 unit tests $(1_{indir} \times 100_{rand})$
ldf || stf		

Instruction	Tested?	Description
`ldf indir, reg || stf reg, indir`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`ldf reg, reg || stf reg, indir`	Yes	100 unit tests $(1_{indir} \times 100_{rand})$
ldi || ldi		
`ldi indir, reg || ldi indir, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`ldi reg, reg || ldi indir, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
ldi || sti		
`ldi indir, reg || sti reg, indir`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`ldi reg, reg || sti reg, indir`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
lsh3 || sti		
`lsh3 count, indir, reg || sti reg, indir`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`lsh3 count, reg, reg || sti reg, indir`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
mpyf3 || addf3		
`mpyf3 indir, indir, reg || addf3 reg, reg, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`mpyf3 indir, reg, reg || addf3 reg, reg, reg`	Yes	100 unit tests $(1_{indir} \times 100_{rand})$
`mpyf3 reg, indir, reg || addf3 reg, reg, reg`	Yes	100 unit tests $(1_{indir} \times 100_{rand})$
`mpyf3 reg, reg, reg || addf3 reg, reg, reg`	Yes	100 unit tests (100_{rand})
`mpyf3 indir, reg, reg || addf3 indir, reg, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`mpyf3 reg, reg, reg || addf3 indir, reg, reg`	Yes	100 unit tests $(1_{indir} \times 100_{rand})$
`mpyf3 reg, reg, reg || addf3 indir, indir, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`mpyf3 reg, reg, reg || addf3 reg, indir, reg`	Yes	100 unit tests $(1_{indir} \times 100_{rand})$
`mpyf3 indir, reg, reg || addf3 reg, indir, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
mpyf3 || stf		
`mpyf3 indir, reg, reg || stf reg, indir`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`mpyf3 reg, reg, reg || stf reg, indir`	Yes	100 unit tests $(1_{indir} \times 100_{rand})$
mpyf3 || subf3		
`mpyf3 indir, indir, reg || subf3 reg, reg, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`mpyf3 indir, reg, reg || subf3 reg, reg, reg`	Yes	100 unit tests $(1_{indir} \times 100_{rand})$
`mpyf3 reg, indir, reg || subf3 reg, reg, reg`	Yes	100 unit tests $(1_{indir} \times 100_{rand})$
`mpyf3 reg, reg, reg || subf3 reg, reg, reg`	Yes	100 unit tests (100_{rand})
`mpyf3 indir, reg, reg || subf3 indir, reg, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`mpyf3 reg, reg, reg || subf3 indir, reg, reg`	Yes	100 unit tests $(1_{indir} \times 100_{rand})$
`mpyf3 reg, reg, reg || subf3 indir, indir, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`mpyf3 reg, reg, reg || subf3 reg, indir, reg`	Yes	100 unit tests $(1_{indir} \times 100_{rand})$
`mpyf3 indir, reg, reg || subf3 reg, indir, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$

Instruction	Tested?	Description
mpyi3 || addi3		
`mpyi3 indir, indir, reg || addi3 reg, reg, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`mpyi3 indir, reg, reg || addi3 reg, reg, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`mpyi3 reg, indir, reg || addi3 reg, reg, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`mpyi3 reg, reg, reg || addi3 reg, reg, reg`	Yes	100 unit tests (100_{rand})
`mpyi3 indir, reg, reg || addi3 indir, reg, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`mpyi3 reg, reg, reg || addi3 indir, reg, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`mpyi3 reg, reg, reg || addi3 indir, indir, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`mpyi3 reg, reg, reg || addi3 reg, indir, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`mpyi3 indir, reg, reg || addi3 reg, indir, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
mpyi3 || sti		
`mpyi3 indir, reg, reg || sti reg, indir`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`mpyi3 reg, reg, reg || sti reg, indir`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
mpyi3 || subi3		
`mpyi3 indir, indir, reg || subi3 reg, reg, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`mpyi3 indir, reg, reg || subi3 reg, reg, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`mpyi3 reg, indir, reg || subi3 reg, reg, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`mpyi3 reg, reg, reg || subi3 reg, reg, reg`	Yes	100 unit tests (100_{rand})
`mpyi3 indir, reg, reg || subi3 indir, reg, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`mpyi3 reg, reg, reg || subi3 indir, reg, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`mpyi3 reg, reg, reg || subi3 indir, indir, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`mpyi3 reg, reg, reg || subi3 reg, indir, reg`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
`mpyi3 indir, reg, reg || subi3 reg, indir, reg`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
negf || stf		
`negf indir, reg || stf reg, indir`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`negf reg, reg || stf reg, indir`	Yes	100 unit tests $(1_{indir} \times 100_{rand})$
negi || sti		
`negi indir, reg || sti reg, indir`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`negi reg, reg || sti reg, indir`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
not || sti		
`not indir, reg || sti reg, indir`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`not reg, reg || sti reg, indir`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
or3 || sti		
`or3 reg, indir, reg || sti reg, indir`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$

Instruction	Tested?	Description
`or3 reg, reg, reg || sti reg, indir`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
stf || stf		
`stf reg, indir || stf reg, indir`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`stf reg, reg || stf reg, indir`	Yes	100 unit tests $(1_{indir} \times 100_{rand})$
sti || sti		
`sti reg, indir || sti reg, indir`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`sti reg, reg || sti reg, indir`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
subf3 || stf		
`subf3 reg, indir, reg || stf reg, indir`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`subf3 reg, reg, reg || stf reg, indir`	Yes	100 unit tests $(1_{indir} \times 100_{rand})$
subi3 || sti		
`subi3 reg, indir, reg || sti reg, indir`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`subi3 reg, reg, reg || sti reg, indir`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
xor3 || sti		
`xor3 indir, reg, reg || sti reg, indir`	Yes	200 unit tests $((1_{indir} \times 1_{indir} + 1_{ar}) \times 100_{rand})$
`xor3 reg, reg, reg || sti reg, indir`	Yes	200 unit tests $((1_{indir} + 1_{ar}) \times 100_{rand})$
idle2		
`idle2`	No	Instruction depends on external environment
lopower		
`lowpower`	No	No effect on machine state.
maxspeed		
`maxspeed`	No	No effect on machine state.

3.3.3 Unit tests generator

To ease the writing of each unit test of the above test plan, a unit test generator has been developped, see Figure 14.

The generator needs an assembly pattern and some substitution strings to generate the unit test source code, that is:

- an assembly function `unit_test` (in file `test.asm`) with the C calling convention and stack parameter passing convention,
- some random inputs for function `unit_test` (`random.txt`),
- and a testbench written in C (`main.c`) that in a loop, reads inputs, calls function `unit_test`, and write outputs.

An assembly pattern is an assembly source code with special tags. These tags start with character `%`. Most of them represent input/outputs that are substituted by real processor registers during assembly source code generation. Tag `%subst` is substituted by a substitution string passed as a command line argument to the generator. Tag `%clobber` says to the generator that assembly pattern explicitely clobber register following that tag and that the generator should not allocate that register while substituing inputs and outputs. Table 11 lists the available tags.

Figure 16 shows an example of assembly pattern and Figure 17 shows the core of generated assembly. During the generation process, each assembly pattern tag is replaced by real processor registers or substitution strings:

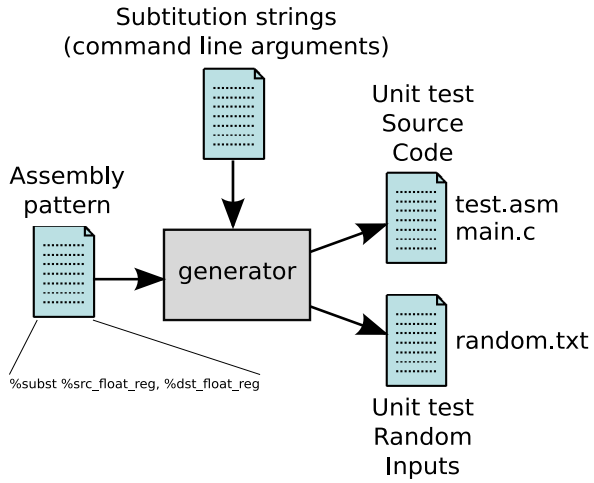

Figure 14: UNISIM TMS320C3X unit test generator.

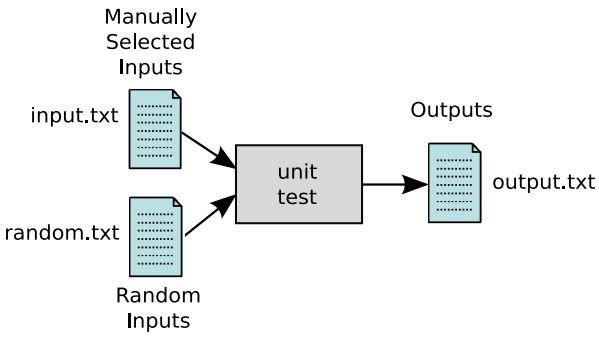

Figure 15: A generated unit test.

```
 1   ldiu %subst, bk  ; ❶ load block size (should be at most 8)
 2   and %0 - 1, %src_int_reg ; ❷ crop random value between 0 and bk - 1
 3   ldiu %2, %clobber ir0; ❸ load ir0 with this random value
 4   ldiu %src_float_buf[16], %dst_aux_reg ; ❹ load a pointer to a buffer of 16 words
 5   addi %subst, %6 ; ❺
 6   andn %7, %6  ; ❻ align circular buffer start on block size
 7   ldiu %st_in, %clobber st ; ❼
 8   %subst *%6++(ir0)%%, %src_dst_float_reg; <-- instruction under test ❽
 9   ldiu st, %st_out ; ❾
10
```

Figure 16: Example of assembly pattern.

```
; ---- INPUTS ----
;   r0: a 32-bit integer register
;   ar2: an auxiliary register pointing to an array of 16 32-bit floating point values
;   r1: a 32-bit integer register (value for st)
;   r2: a 40-bit floating point register
; ---- OUTPUTS ----
;   ar4: an auxiliary register
;   r2: a 40-bit floating point register
;   r3: a 32-bit integer register (value of st)
ldiu 5, bk                ; ❶  load block size (should be at most 8)
and 5 - 1, r0             ; ❷  crop random value between 0 and bk - 1
ldiu r0, ir0              ; ❸  load ir0 with this random value
ldiu ar2, ar4             ; ❹  load a pointer to a buffer of 16 words
addi 7, ar4               ; ❺
andn 7, ar4               ; ❻  align circular buffer start on block size
ldiu r1, st               ; ❼
addf *ar4++(ir0)%, r2     ; ❽  ← instruction under test
ldiu st, r3               ; ❾
```

Figure 17: Generated assembly from assembly pattern of Figure 16 and substitution strings `"5"`, `"7"`, and `"addf"`.

❶ `%subst` is substituted by integer constant `5` passed as command line argument of the generator;

❷ `%0` is substituted as in ❶, while `%src_int_reg` is substituted with register `r0`;

❸ `%clobber ir0` is substituted with register `ir0` and register `ir0` is marked as clobbered;

❹ `%src_float_buf[16]` is substituted with register `ar2` that points to an array of 16 32-bit floating-point values allocated on the stack; `%dst_aux_reg` is substituted with register `ar4`;

❺ `%subst` is substituted with integer constant `7` passed as command line argument to the generator; `%6` is substituted as `%dst_aux_reg` in ❹;

❻ `%7` is substituted as `%subst` in ❺ and `%6` is substituted as `%dst_aux_reg` in ❹;

❼ `%st_in` is substituted with register `r1` and register `r1` is marked as containing state of register st to enable pretty printing of its content, while `%clobber st` is substituted by register `st` and register `st` is marked as clobbered;

❽ `%6` is substituted as `%dst_aux_reg` in ❹; `%%` is substituted with `%`; `%src_dst_float_reg` is substituted with register `r2`;

❾ `%st_out` is substituted with register `r3` and register `r3` is marked as containing state of register `st` to enable pretty printing of its content.

The random inputs are obtained with a KISS (Keep It Simple Stupid) random number generator (see `http://www.math.niu.edu/~rusin/known-math/99/RNG`) embedded in the unit tests generator. The generator creates a uniform distribution for integer numbers. Table 12 shows the distribution for the floating point numbers.

The unit test source code can be compiled for the development board, and run on both the development board and the UNISIM TMS320C3X simulator. As a unit test uses the TI C I/O functions, it can reads inputs and write outputs from/to files of the host file system, see Figure 15. Such capability has considerably reduced the complexity of testing the assembly pattern under test on both the development board and the UNISIM TMS320C3X simulator.

A unit test reads manually selected inputs from file `input.txt` and some random generated inputs from file `random.txt`. It writes outputs into file `output.txt`.

Tag	Type	Storage type
%src_int_reg	source	32-bit integer register
%src_float_reg	source	40-bit floating-point register
%src_aux_reg	source	auxiliary register
%st_in	source	32-bit integer register (value for st)
%dst_int_reg	destination	32-bit integer register
%dst_float_reg	destination	40-bit floating-point register
%dst_aux_reg	destination	auxiliary register
%st_out	destination	32-bit integer register (value of st)
%src_dst_int_reg	source & destination	32-bit integer register
%src_dst_float_reg	source & destination	40-bit floating-point register
%src_dst_aux_reg	source & destination	auxiliary register
%tmp_int_reg	temporary	32-bit integer register
%tmp_float_reg	temporary	40-bit floating-point register
%tmp_aux_reg	temporary	auxiliary register
%src_int_buf[dim]	source	auxiliary register pointing to an array of *dim* 32-bit integer values
%dst_int_buf[dim]	destination	auxiliary register pointing to an array of *dim* 32-bit integer values
%src_float_buf[dim]	source	auxiliary register pointing to an array of *dim* 32-bit floating-point values
%dst_float_buf[dim]	destination	auxiliary register pointing to an array of *dim* 32-bit floating-point values
%subst	substitution	N/A
%clobber reg	clobber	N/A
%0, %1, %2, ...	reference	N/A

Table 11: UNISIM TMS320C3X unit test generator assembly patterns tags.

Category	Probability
-inf	1/37
smallest negative number	1/37
zero	1/37
real zero	1/37
smallest positive number	1/37
near to integer	2/37
+inf	1/37
small	2/37
large	2/37
mantissa near previously generated, fully random exponent	5/37
exponent near previously generated, fully random mantissa	5/37
float near previously generated	5/37
fully random	10/37

Table 12: UNISIM TMS320C3X unit test generator floating point distribution.

3.3.4 The testing environment

A testing environment has been set up using the unit test generator and assembly patterns. A GNU Makefile is provided to run the unit tests on both the development board (our reference) and the UNISIM TMS320C3X simulator. The test plan is located in the companion GNU bash script `factorial.sh`, and more precisely in function '`factorial`'. The goal of this function is to generate an auxiliary `Makefile` (`Makefile.aux`) that contains building rules of planned unit tests. A companion C++ program, `generator` is driven by this auxiliary Makefile to generate the actual unit tests source code. The generated source code is then compiled for the simulated target using the Texas Instruction cross-compilation tool chain. The resulting cross-compiled binaries are executed on both a real TMS320C3X DSP using 'Code Composer', and the UNISIM TMS320C3X simulator. The real/reference execution results and the simulation results are compared, and a failure diagnostic (PASSED or FAILED) is established for each generated unit test program.

As explained in Section 3.3.3, the unit test program output is in file `output.txt`. To clearly distinguish simulation results from real execution results, file `output.txt` (the unit test output) is renamed `output.ref` when run on the development board or `output.sim` when run on the UNISIM TMS320C3X simulator.
The list of supported `Makefile` targets is the following:

- `generator(.exe)`: compile the unit tests generator (`generator(.exe)`)
- `compile`: compile the unit tests for the TMS320C3X development board (objects/binaries are `test.obj`, `main.obj` and `test.out`)
- `rnd`: generate the random input files (`random.txt`)
- `execute`: execute the unit tests on the TMS320C3X development board (`EXECUTE` must be set) (execution result is in `output.ref`)
- `simulate`: run the simulator (SIMULATE must be set) (simulation results are in `output.sim`)
- `check`: compare simulator vs. reference (depends on diff) (check result is in `output.check`)
- `diff`: generate difference between simulation vs. reference (depends on `execute` and `simulate`) (diff result is in `output.diff`)
- `regression-test`: launch a regression test of the UNISIM TMS320C3X simulator
- `doc`: generate unit tests documentation (`unit_tests.pdf`)
- `dist`: distribute the testing environment together with reference outputs (`output.ref`)
- `clean`: clean everything (but execution results, use `cleanref` for that)
- `cleangen`: clean generator executable (`generator(.exe)`)
- `cleansrc`: clean TMS320C3X generated source files (`test.asm` and `main.c`)
- `cleanrnd`: clean generated random input files (`random.txt`)
- `cleanbin`: clean TMS320C3X executable files (`test.out`)
- `cleanobj`: clean TMS320C3X object files (`test.obj` and `main.obj`)
- `cleangel`: clean GEL scripts (`run.gel`)
- `cleansim`: clean simulation results (`output.sim`)

- `cleanref`: clean execution results (`output.ref`)
- `cleandiff`: clean diff files (`output.diff`)
- `cleancheck`: clean check files (`output.check`)
- `cleandoc`: clean latex files (`test.tex`)

The following `Makefile` variables are available for tuning the `Makefile`:

- Mandatory:
  - `SIMULATE`: path to the UNISIM TMS320C3X simulator binary
  - `EXECUTE`: path to Code Composer executable (i.e. `cc_app.exe`)
  - `DIST_DIR`: path to destination directory for a distribution
- Optional:
  - `COMPILER_PREFIX`: prefix to add before the compiler tools executables names (default: empty)

The provided `Makefile` uses a GNU bash script `factorial.sh`, that contains a factorial plan, to generate most of the `Makefile` rules in file `Makefile.aux`. To obtain the reference outputs from the developement board, do the following at the command prompt:

```
$ make execute EXECUTE=cc_app.exe
```

Figure 18 shows the testing environment running on Windows and launching unit tests on the development board.

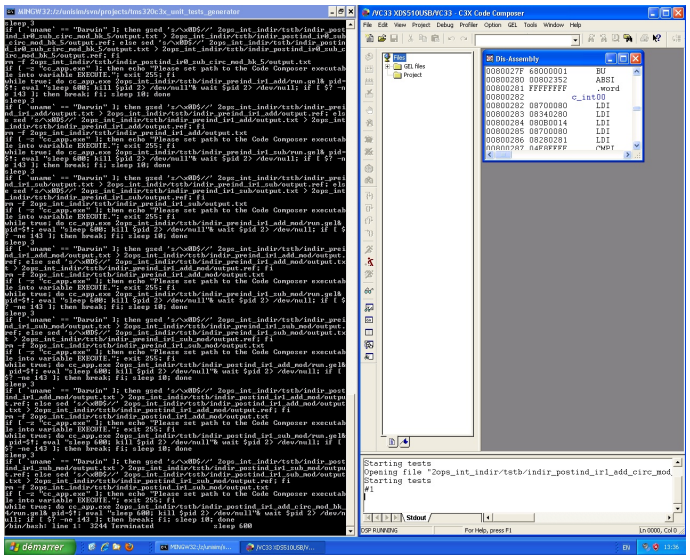

Figure 18: GNU Make (on the left) launching unit tests on the development board using code composer (on the right)

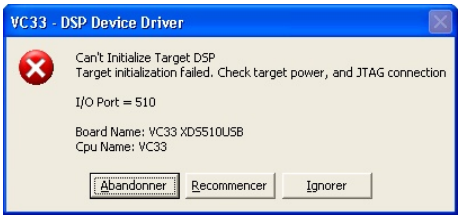

Figure 19: VC33 device driver communication problem.

Note: You may experience frequent failures of the JTAG Emulator (red LED that indicates activity over USB get stuck), making Code Composer complain in a dialog box that it can't initialize target DSP, see Figure 19. Disconnect and reconnect the USB cable on the JTAG emulator, and then click button "Retry" to resume execution.

To obtain the simulation outputs from the UNISIM TMS320C3X simulator, do the following at the command prompt:

```
$ make simulate SIMULATE=path-to-unisim-tms320c3x-2.0
```

3.3.5 Regression tests

The testing environment also acts as a regression test for UNISIM TMS320C3X simulator as the expected results (`output.ref`) are already provided in the testing environment.

To cross-compile the unit tests programs, do the following at the command prompt on the cross-compilation host:

```
$ make generator
$ make compile
$ make c31boot.out
```

Then to check that all unit tests successfully run on the UNISIM TMS320C3X simulator, do the following at the command prompt on the simulation host:

```
$ make cleangen
$ make generator
$ make regression-test SIMULATE=path-to-unisim-tms320c3x-2.0
```

The result for each unit test program is either `PASSED` or `FAILED`.

Note: At most one instance of the Texas Instrument Cross-compiler can be run at a time (at least on a Windows host and Wine). Using `flag -j` of GNU Make when compiling the unit tests programs results in unexpected behaviors.

Note: Unit test program `parallel_float/stf_stf/reg_buf` will likely fail because of a bug in instruction `STF || STF` in the TMS320VC33 of our development board.

Appendices

Appendix A: Simulator technical reference (generated)

This documentation has been automatically generated from the simulator `UNISIM tms320c3x` version 2.0 on Oct 18 2013.

A.1 Introduction

UNISIM tms320c3x, a TMS320C3X DSP simulator with support of TI COFF binaries, and TI C I/O (RTS run-time).

Section A.2 gives licensing informations about the simulator. Section A.3 shows the set of modules and services that compose the simulator. Section A.4 shows how to invoke the simulator at the command line prompt. Section A.5 gives the simulator parameters. Section A.6 gives the simulator statistic counters. Section A.7 gives the simulator statistic formulas.

A.2 Licensing

UNISIM tms320c3x 2.0

Copyright (C) 2009-2013, Commissariat a l'Energie Atomique (CEA)

License: BSD (see file COPYING)

Authors: Gilles Mouchard <gilles.mouchard@cea.fr>, Daniel Gracia Pérez <daniel.gracia-perez@cea.fr>

A.3 Simulated configuration

The UNISIM tms320c3x simulator is composed of the following modules and services:

- **cpu**: this module implements a TMS320C3X DSP core
- **debugger**
- **dint-stub**: An initiator stub
- **gdb-server**: this service implements the GDB server remote serial protocol over TCP/IP. Standards GDB clients (e.g. gdb, eclipse, ddd) can connect to the simulator to debug the target application that runs within the simulator.
- **host-time**: this service is an abstraction layer for the host machine time
- **inline-debugger**: this service implements a built-in debugger in the terminal console
- **int0-stub**: An initiator stub
- **int1-stub**: An initiator stub
- **int2-stub**: An initiator stub
- **int3-stub**: An initiator stub
- **loader**: A multi-format loader that supports ELF32, ELF64, S19, COFF and Raw binary files
- **loader.file0**
- **loader.memory-mapper**: A memory mapper
- **loader.tee-backtrace**: This service/client implements a tee ('T'). It unifies the backtrace capability of several services that individually provides their own backtrace capability
- **loader.tee-blob**: This service/client implements a tee ('T'). It unifies the statement lookup capability of several services that individually provides their own statement lookup capability

- **loader.tee-loader**: This service/client implements a tee ('T'). It unifies the loader capability of several services that individually provides their own loader capability
- **loader.tee-stmt-lookup**: This service/client implements a tee ('T'). It unifies the statement lookup capability of several services that individually provides their own statement lookup capability
- **loader.tee-symbol-table-lookup**: This service/client implements a tee ('T'). It unifies the symbol table lookup capability of several services that individually provides their own symbol table lookup capability
- **memory**: this module implements a memory
- **profiler**
- **rint0-stub**: An initiator stub
- **rint1-stub**: An initiator stub
- **tee-memory-access-reporting**
- **tee-memory-access-reporting.tee-memory-access-reporting.control_selector[0]**
- **tee-memory-access-reporting.tee-memory-access-reporting.control_selector[10]**
- **tee-memory-access-reporting.tee-memory-access-reporting.control_selector[11]**
- **tee-memory-access-reporting.tee-memory-access-reporting.control_selector[12]**
- **tee-memory-access-reporting.tee-memory-access-reporting.control_selector[13]**
- **tee-memory-access-reporting.tee-memory-access-reporting.control_selector[14]**
- **tee-memory-access-reporting.tee-memory-access-reporting.control_selector[15]**
- **tee-memory-access-reporting.tee-memory-access-reporting.control_selector[1]**
- **tee-memory-access-reporting.tee-memory-access-reporting.control_selector[2]**
- **tee-memory-access-reporting.tee-memory-access-reporting.control_selector[3]**
- **tee-memory-access-reporting.tee-memory-access-reporting.control_selector[4]**
- **tee-memory-access-reporting.tee-memory-access-reporting.control_selector[5]**
- **tee-memory-access-reporting.tee-memory-access-reporting.control_selector[6]**
- **tee-memory-access-reporting.tee-memory-access-reporting.control_selector[7]**
- **tee-memory-access-reporting.tee-memory-access-reporting.control_selector[8]**
- **tee-memory-access-reporting.tee-memory-access-reporting.control_selector[9]**
- **ti-c-io**
- **time**: this service is an abstraction layer for the SystemC kernel time
- **tint0-stub**: An initiator stub
- **tint1-stub**: An initiator stub
- **xint0-stub**: An initiator stub
- **xint1-stub**: An initiator stub

A.4 Using the UNISIM tms320c3x simulator

The UNISIM tms320c3x simulator has the following command line options:

Usage: `unisim-tms320c3x-2.0 [<options>] [...]`
Options:

- `--set <param=value> or -s <param=value>`: set value of parameter 'param' to 'value'
- `--config <XML file> or -c <XML file>`: configures the simulator with the given XML configuration file
- `--get-config <XML file> or -g <XML file>`: get the simulator configuration XML file (you can use it to create your own configuration. This option can be combined with -c to get a new configuration file with existing variables from another file
- `--list or -l`: lists all available parameters, their type, and their current value
- `--warn or -w`: enable printing of kernel warnings
- `--doc <Latex file> or -d <Latex file>`: enable printing a latex documentation
- `--version or -v`: displays the program version information
- `--share-path <path> or -p <path>`: the path that should be used for the share directory (absolute path)
- `--help or -h`: displays this help

A.5 Configuration

Simulator configuration (see below) can be modified using command line Options `--set <param=value>` or `--config <config file>`.

Global	
Name: `enable-gdb-server` **Default:** `true` **Valid:** `true`, `false` **Description:** Enable/Disable GDB server instantiation.	**Type:** `parameter` **Data type:** `boolean`
Name: `enable-inline-debugger` **Default:** `true` **Valid:** `true`, `false` **Description:** Enable/Disable inline debugger instantiation.	**Type:** `parameter` **Data type:** `boolean`
Name: `enable-press-enter-at-exit` **Default:** `false` **Valid:** `true`, `false` **Description:** Enable/Disable pressing key enter at exit.	**Type:** `parameter` **Data type:** `boolean`

Name: kernel_logger.file **Type:** parameter
Default: false **Data type:** boolean
Valid: true, false

Description:
Keep logger output in a file.

Name: kernel_logger.filename **Type:** parameter
Default: logger_output.txt **Data type:** string

Description:
Filename to keep logger output _(the option file must be activated).

Name: kernel_logger.std_err **Type:** parameter
Default: true **Data type:** boolean
Valid: true, false

Description:
Show logger output through the standard error output.

Name: kernel_logger.std_err_color **Type:** parameter
Default: false **Data type:** boolean
Valid: true, false

Description:
Colorize logger output through the standard error output _(only works if std_err is active).

Name: kernel_logger.std_out **Type:** parameter
Default: false **Data type:** boolean
Valid: true, false

Description:
Show logger output through the standard output.

Name: kernel_logger.std_out_color **Type:** parameter
Default: false **Data type:** boolean
Valid: true, false

Description:
Colorize logger output through the standard output _(only works if std_out is active).

Name: kernel_logger.xml_file **Type:** parameter
Default: false **Data type:** boolean
Valid: true, false

Description:
Keep logger output in a file xml formatted.

Name: kernel_logger.xml_file_gzipped **Type:** parameter
Default: false **Data type:** boolean
Valid: true, false

Description:
If the xml_file option is active, the output file will be compressed (a .gz extension will be automatically added to the xml_filename option.

Name: kernel_logger.xml_filename **Type:** parameter
Default: logger_output.xml **Data type:** string

Description:
Filename to keep logger xml output _(the option xml_file must be activated).

cpu

Name: cpu.max-inst **Type:** parameter
Default: 0xffffffffffffffff **Data type:** unsigned 64-bit integer

Name: cpu.trap-on-instruction-counter **Type:** parameter
Default: 0xffffffffffffffff **Data type:** unsigned 64-bit integer

Name: cpu.mimic-dev-board **Type:** parameter
Default: true **Data type:** boolean
Valid: true, false

Name: cpu.trap-on-trap-instruction **Type:** parameter
Default: 0x7400003f **Data type:** unsigned 32-bit integer

Description:
if not zero, encoding of trap instruction that should trap into debugger.

Name: cpu.enable-parallel-load-bug **Type:** parameter
Default: true **Data type:** boolean
Valid: true, false

Description:
When using parallel loads (LDF src2, dst2 —— LDF src1, dst1) the src1 load doesn't transform incorrect zero values to valid zero representation, instead they copy the contents of the memory to the register. Set to this parameter to false to transform incorrect zero values..

Name: cpu.enable-rnd-bug **Type:** parameter
Default: true **Data type:** boolean
Valid: true, false

Description:
If enabled the 'rnd' instruction sets the Z flag to 0 systematically, as it is done in the evaluation board. Otherwise, Z is unchanged as it is written in the documentation..

Name: cpu.enable-parallel-store-↪bug **Type:** parameter
Default: true **Data type:** boolean
Valid: true, false

Description:
If enabled, when using parallel stores (STF src2, dst2 —— STF src1, dst1) the first store is treated as a NOP..

Name: `cpu.enable-float-ops-with-↪non-ext-regs` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
If enabled non extended registers can be used on all the float instructions, however the behavior is not documented and can differ between chips revision. If disabled, it stops simulation when using non extended registers on float instructions..

Name: `cpu.verbose-all` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Name: `cpu.verbose-setup` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Name: `cpu.cpu-cycle-time` **Type:** `parameter`
Default: `13333 ps` **Data type:** `sc_time`

Description:
cpu cycle time.

Name: `cpu.nice-time` **Type:** `parameter`
Default: `1 us` **Data type:** `sc_time`

Description:
maximum time between synchonizations.

Name: `cpu.ipc` **Type:** `parameter`
Default: `1` **Data type:** `double precision floating-point`

Description:
targeted average instructions per second.

Name: `cpu.enable-dmi` **Type:** `parameter`
Default: `true` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable TLM 2.0 DMI (Direct Memory Access) to speed-up simulation.

Name: `cpu.debug-dmi` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable debugging of DMI (Direct Memory Access).

debugger

Name: `debugger.verbose` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable verbosity.

Name: `debugger.dwarf-to-html-output-directory` **Type:** `parameter`
Default: **Data type:** `string`

Description:
DWARF v2/v3 to HTML output directory.

Name: `debugger.dwarf-register-number-mapping-filename` **Type:** `parameter`
Default: **Data type:** `string`

Description:
DWARF register number mapping filename.

Name: `debugger.parse-dwarf` **Type:** `parameter`
Default: `true` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable parsing of DWARF debugging informations.

Name: `debugger.debug-dwarf` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable debugging of DWARF.

dint-stub

Name: `dint-stub.enable` **Type:** `parameter`
Default: `true` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable a lazy implementation of TLM 2.0 method interface.

Name: `dint-stub.verbose` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable verbosity.

gdb-server

Name: gdb-server.memory-atom-size **Type:** parameter
Default: 0x00000004 **Data type:** unsigned 32-bit integer

Description:
size of the smallest addressable element in memory.

Name: gdb-server.tcp-port **Type:** parameter
Default: 12345 **Data type:** signed 32-bit integer

Description:
TCP/IP port to listen waiting for a GDB client connection.

Name: gdb-server.architecture-description-filename **Type:** parameter
Default: gdb_tms320c3x.xml **Data type:** string

Description:
filename of a XML description of the connected processor.

Name: gdb-server.verbose **Type:** parameter
Default: false **Data type:** boolean
Valid: true, false

Description:
Enable/Disable verbosity.

inline-debugger

Name: inline-debugger.memory-atom-size **Type:** parameter
Default: 0x00000004 **Data type:** unsigned 32-bit integer

Description:
size of the smallest addressable element in memory.

Name: inline-debugger.search-path **Type:** parameter
Default: **Data type:** string

Description:
Search path for source (separated by ';').

Name: inline-debugger.init-macro **Type:** parameter
Default: **Data type:** string

Description:
path to initial macro to run when debugger starts.

Name: inline-debugger.output **Type:** parameter
Default: **Data type:** string

Description:
path to output file where to redirect the debugger outputs.

int0-stub

Name: `int0-stub.enable` **Type:** `parameter`
Default: `true` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable a lazy implementation of TLM 2.0 method interface.

Name: `int0-stub.verbose` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable verbosity.

int1-stub

Name: `int1-stub.enable` **Type:** `parameter`
Default: `true` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable a lazy implementation of TLM 2.0 method interface.

Name: `int1-stub.verbose` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable verbosity.

int2-stub

Name: `int2-stub.enable` **Type:** `parameter`
Default: `true` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable a lazy implementation of TLM 2.0 method interface.

Name: `int2-stub.verbose` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable verbosity.

int3-stub

Name: `int3-stub.enable` **Type:** `parameter`
Default: `true` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable a lazy implementation of TLM 2.0 method interface.

Name: `int3-stub.verbose` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable verbosity.

loader

Name: `loader.verbose` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable verbosity.

Name: `loader.verbose-parser` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable verbosity of parser.

Name: `loader.filename` **Type:** `parameter`
Default: `c31boot.out` **Data type:** `string`

Description:
List of files to load. Syntax: [[filename=]<filename1>[:[format=]<format1>]][,[filename=]<filename2>[:[form
(e.g. boot.bin:raw,app.elf).

loader.file0

Name: `loader.file0.filename` **Type:** `parameter`
Default: `c31boot.out` **Data type:** `string`

Description:
the COFF filename to load.

Name: `loader.file0.dump-headers` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable dump of COFF file headers while loading.

Name: `loader.file0.verbose` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable verbosity.

loader.memory-mapper

Name: `loader.memory-mapper.verbose` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable verbosity.

Name: `loader.memory-mapper.verbose-parser` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable verbosity of parser.

Name: `loader.memory-mapper.mapping` **Type:** `parameter`
Default: `memory=memory:0x0-0xffffffff` **Data type:** `string`

Description:
Memory mapping. Syntax: [[(memory=]<memory1>[:[range=]<low1-high1>]][,[(memory=]<memory2>[:[range=]<low2-high2>]]... (e.g. ram:0x0-0x00ffff,rom:0xff0000-0xffffff).

memory

Name: `memory.org` **Type:** `parameter`
Default: `0x00000000` **Data type:** `unsigned 32-bit integer`

Description:
memory origin/base address.

Name: `memory.bytesize` **Type:** `parameter`
Default: `4294967295` **Data type:** `unsigned 32-bit integer`

Description:
memory size in bytes.

Name: `memory.initial-byte-value` **Type:** `parameter`
Default: `0x00` **Data type:** `unsigned 8-bit integer`

Name: `memory.cycle-time` **Type:** `parameter`
Default: `13333 ps` **Data type:** `sc_time`

Description:
memory cycle time.

Name: `memory.read-latency` **Type:** `parameter`
Default: `13333 ps` **Data type:** `sc_time`

Description:
memory read latency.

Name: `memory.write-latency` **Type:** `parameter`
Default: `0 s` **Data type:** `sc_time`

Description:
memory write latency.

Name: `memory.verbose` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
enable/disable verbosity.

profiler

Name: `profiler.min-data-read-prof-↪addr` **Type:** `parameter`
Default: `0x00000000` **Data type:** `unsigned 32-bit integer`

Description:
Minimum address for data read profiling.

Name: `profiler.max-data-read-prof-↪addr` **Type:** `parameter`
Default: `0xffffffff` **Data type:** `unsigned 32-bit integer`

Description:
Maximum address for data read profiling.

Name: `profiler.min-data-write-prof-↪addr` **Type:** `parameter`
Default: `0x00000000` **Data type:** `unsigned 32-bit integer`

Description:
Minimum address for data write profiling.

Name: `profiler.max-data-write-prof-↪addr` **Type:** `parameter`
Default: `0xffffffff` **Data type:** `unsigned 32-bit integer`

Description:
Maximum address for data write profiling.

Name: `profiler.min-insn-fetch-prof-↪addr` **Type:** `parameter`
Default: `0x00000000` **Data type:** `unsigned 32-bit integer`

Description:
Minimum address for instruction fetch profiling.

Name: `profiler.max-insn-fetch-prof-↪addr` **Type:** `parameter`
Default: `0xffffffff` **Data type:** `unsigned 32-bit integer`

Description:
Maximum address for instruction fetch profiling.

Name: `profiler.min-insn-exec-prof-↪addr` **Type:** `parameter`
Default: `0x00000000` **Data type:** `unsigned 32-bit integer`

Description:
Minimum address for instruction execution profiling.

Name: `profiler.max-insn-exec-prof-↪addr` **Type:** `parameter`
Default: `0xffffffff` **Data type:** `unsigned 32-bit integer`

Description:
Maximum address for instruction execution profiling.

Name: `profiler.enable-data-read-↪prof` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable data read profiling.

Name: `profiler.enable-data-write-↪prof` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable data write profiling.

Name: `profiler.enable-insn-fetch-↪prof` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable instruction fetch profiling.

Name: `profiler.enable-insn-exec-↪prof` **Type:** `parameter`

Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable instruction execution profiling.

Name: `profiler.verbose` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable verbosity.

rint0-stub

Name: `rint0-stub.enable` **Type:** `parameter`
Default: `true` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable a lazy implementation of TLM 2.0 method interface.

Name: `rint0-stub.verbose` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable verbosity.

rint1-stub

Name: `rint1-stub.enable` **Type:** `parameter`
Default: `true` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable a lazy implementation of TLM 2.0 method interface.

Name: `rint1-stub.verbose` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable verbosity.

ti-c-io

Name: `ti-c-io.enable` **Type:** `parameter`
Default: `true` **Data type:** `boolean`
Valid: `true`, `false`

Description:
enable/disable TI C I/O support.

Name: `ti-c-io.warning-as-error` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Whether Warnings are considered as error or not.

Name: `ti-c-io.pc-register-name` **Type:** `parameter`
Default: `PC` **Data type:** `string`

Description:
Name of the CPU program counter register.

Name: `ti-c-io.c-io-buffer-symbol-name` **Type:** `parameter`
Default: `__CIOBUF_` **Data type:** `string`

Description:
C I/O buffer symbol name.

Name: `ti-c-io.c-io-breakpoint-symbol-name` **Type:** `parameter`
Default: `C$$IO$$` **Data type:** `string`

Description:
C I/O breakpoint symbol name.

Name: `ti-c-io.c-exit-breakpoint-symbol-name` **Type:** `parameter`
Default: `C$$EXIT` **Data type:** `string`

Description:
C EXIT breakpoint symbol name.

Name: `ti-c-io.verbose-all` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
globally enable/disable verbosity.

Name: `ti-c-io.verbose-io` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
enable/disable verbosity while I/Os.

Name: `ti-c-io.verbose-setup` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
enable/disable verbosity while setup.

Name: `ti-c-io.enable-lseek-bug` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
enable/disable lseek bug (as code composer).

tint0-stub

Name: `tint0-stub.enable` **Type:** `parameter`
Default: `true` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable a lazy implementation of TLM 2.0 method interface.

Name: `tint0-stub.verbose` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable verbosity.

tint1-stub

Name: `tint1-stub.enable` **Type:** `parameter`
Default: `true` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable a lazy implementation of TLM 2.0 method interface.

Name: `tint1-stub.verbose` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable verbosity.

xint0-stub

Name: `xint0-stub.enable` **Type:** `parameter`
Default: `true` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable a lazy implementation of TLM 2.0 method interface.

Name: `xint0-stub.verbose` **Type:** `parameter`
Default: `false` **Data type:** `boolean`
Valid: `true`, `false`

Description:
Enable/Disable verbosity.

xint1-stub

Name: `xint1-stub.enable` **Type:** `parameter`

Default: `true`	**Data type:** `boolean`
Valid: `true`, `false`	
Description: Enable/Disable a lazy implementation of TLM 2.0 method interface.	
Name: `xint1-stub.verbose`	**Type:** `parameter`
Default: `false`	**Data type:** `boolean`
Valid: `true`, `false`	
Description: Enable/Disable verbosity.	

A.6 Statistics

Simulation statistic counters are listed below:

cpu	
Name: `cpu.instruction-counter`	**Type:** `statistic` **Data type:** `unsigned 64-bit integer`
Name: `cpu.insn-cache-accesses`	**Type:** `statistic` **Data type:** `unsigned 64-bit integer`
Description: Instruction cache accesses.	
Name: `cpu.insn-cache-misses`	**Type:** `statistic` **Data type:** `unsigned 64-bit integer`
Description: Instruction cache misses.	
memory	
Name: `memory.memory-usage`	**Type:** `statistic` **Data type:** `unsigned 32-bit integer`
Description: target memory usage in bytes (page granularity of 1048576 bytes).	
Name: `memory.read-counter`	**Type:** `statistic` **Data type:** `unsigned 64-bit integer`
Description: read access counter (not accurate when using SystemC TLM 2.0 DMI).	
Name: `memory.write-counter`	**Type:** `statistic` **Data type:** `unsigned 64-bit integer`

Description:
write access counter (not accurate when using SystemC TLM 2.0 DMI).

A.7 Formulas

Simulation statistic formulas are listed below:

cpu	
Name: `cpu.insn-cache-miss-rate`	**Type:** `formula`
Formula: `cpu.insn-cache-misses /` `↪cpu.insn-cache-accesses`	**Data type:** `double precision floating-point`
Description: instruction cache miss rate.	