GAISLER RESEARCH

TSIM2 Simulator User's Manual

TSIM ERC32/LEON2/LEON3

Version 2.0.7
January 2007

Första Långgatan 19
413 27 Göteborg
Sweden

tel +46 31 7758650
fax +46 31 421407
www.gaisler.com

Table of contents

1 Introduction

1.1 General

TSIM is a generic SPARC* architecture simulator capable of emulating ERC32- and LEON-based computer systems.

TSIM provides several unique features:

- Accurate and cycle-true emulation of ERC32 and LEON2/3 processors
- Superior performance: +30 MIPS on high-end PC (AMD64@2.4 GHz)
- Accelerated simulation during processor standby mode
- Standalone operation or remote connection to GNU debugger (gdb)
- 64-bit time for unlimited simulation periods
- Instruction trace buffer
- EDAC emulation (ERC32)
- MMU emulation (LEON2/3)
- SDRAM emulation (LEON2/3)
- Local scratch-pad RAM (LEON3)
- Loadable modules to include user-defined I/O devices
- Non-intrusive execution time profiling
- Code coverage monitoring
- Dual-processor synchronisation
- Stack backtrace with symbolic information
- Check-pointing capability to save and restore complete simulator state
- Also provided as library to be included in larger simulator frameworks

1.2 Supported platforms and system requirements

TSIM supports the following platforms: solaris-2.8, linux, linux-x64, Windows 2K/XP, and Windows 2K/XP with Cygwin unix emulation.

1.3 Obtaining TSIM

The primary site for TSIM is http://www.gaisler.com/, where the latest version of TSIM can be ordered and evaluation versions downloaded.

1.4 Problem reports

Please send problem reports or comments to tsim@gaisler.com.

**. SPARC is a registered trademark of SPARC International*

2 Installation

2.1 General

TSIM is distributed as a tar-file (e.g. tsim-erc32-2.0.3.tar.gz) with the following contents:

```
doc                             TSIM documentation
samples                         Sample programs
iomod                           Example I/O module
tsim/cygwin                     TSIM binary for cygwin
tsim/linux                      TSIM binary for linux
tsim/linux-x64                  TSIM binary for linux-x64
tsim/solaris                    TSIM binary for solaris
tsim/win32                      TSIM binary for native windows
tlib/cygwin                     TSIM library for cygwin
tlib/linux                      TSIM library for linux
tlib/linux-x64                  TSIM library for linux-x64
tlib/solaris                    TSIM library for solaris
tlib/win32                      TSIM library for native windows
```

The tar-file can be installed at any location with the following command:

```
gunzip -c tsim-erc32-2.0.3.tar.gz | tar xf -
```

2.2 License installation

TSIM is licensed using a HASP hardware key. Two versions of the key are available, HASP4/HL for node-locked licenses (blue keys), and HASP4/HL Net for floating licenses (red keys). Before use, a device driver for the key must be installed. The latest drivers can be found at www.ealaddin.com or www.gaisler.com/tsim.html. If a floating-license key is used, the HASP4/HL network license server also has to be installed and started. The necessary server installation documentation can be obtained from the distribution CD or from www.ealaddin.com. See *appendix A* for installation of device drivers under Windows, Linux and Mac platforms.

3 Operation

3.1 Overview

TSIM can operate in two modes: standalone and attached to gdb. In standalone mode, ERC32 or LEON applications can be loaded and simulated using a command line interface. A number of commands are available to examine data, insert breakpoints and advance simulation. When attached to gdb, TSIM acts as a remote gdb target, and applications are loaded and debugged through gdb (or a gdb front-end such as ddd).

3.2 Staring TSIM

TSIM is started as follows on a command line:

tsim-erc32 [*options*] [*input_files*]

tsim-leon [*options*] [*input_files*]

tsim-leon3 [*options*] [*input_files*]

The following command line options are supported by TSIM:

-**ahbm** *ahb_module_path*

Use *ahb_module*_path as loadable AHB module rather than the default ahb.so (LEON only).

-**at697e** Configure caches according to the Atmel AT697E device (LEON2 only).

-**banks** *ram_banks*

Sets how many ram banks (1 - 4) the ram is divided on. Default is 1. (LEON only).

-**bopt** Enables idle-loop optimisation (see text).

-**c** *file* Reads commands from *file* instead of stdin. If the file .tsimrc exists in the home directory, it will be automatically executed before the commands in *file*.

-**cfg** *file* Reads configuration options from *file* instead of reading options from the command line. If the file .tsimcfg exists in the home directory, it will be automatically read if *file* is not specified.

-**cpm** *cp_module*

Use *cp_module* as loadable co-processor module file name (LEON).

-**dcsize** *size* Defines the set-size (kbytes) of the LEON dcache. Allowed values are 1 - 64 in binary steps. Default is 4 kbytes.

-**dlock** Enable data cache line locking. Default is disabled. (LEON only).

-**dlsize** *size* Sets the line size of the LEON data cache (in bytes). Allowed values are 8, 16 or 32. Default is 16.

-**dlram** *addr size*

Allocates *size* Kbytes of local dcache scratchpad memory at address *addr*. (LEON3)

-**dsets** *sets* Defines the number of sets in the LEON data cache. Allowed values are 1 - 4.

-**drepl** *repl* Sets the replacement algorithm for the LEON data cache. Allowed values are rnd (default) for random replacement, lru for the least-recently-used replacement algorithm and lrr for least-recently-replaced replacement algorithm.

-**freq** *system_clock*

Sets the simulated system clock (MHz). Will affect UART timing and performance statistics. Default is 14 for ERC32 and 50 for LEON.

-**fast_uart** Run UARTs at infinite speed, rather than with correct (slow) baud rate.

-**fpm** *fp_module*

Use *fp_module* as loadable FPU module rather than the default fp.so (LEON only).

-**gr702rc** Set cache parameters to emulate the GR702RC device.

-**hwbp** Use TSIM hardware breakpoints for gdb breakpoints.

-**icsize** *size* Defines the set-size (kbytes) of the LEON icache. Allowed values are 1 - 64 in binary steps.

-**isets** *sets* Defines the number of sets in the LEON instruction cache. Allowed values are 1(default) - 4.

-**ift** Generate illegal instruction trap on IFLUSH. Emulates the IFT input on the ERC32 processor.

-**ilock** Enable instruction cache line locking. Default is disabled.

-**iom** *io_module*

Use *io_module* as loadable I/O module rather than the default io.so.

-**ilsize** *size* Sets the line size of the LEON instruction cache (in bytes). Allowed values are 8, 16 or 32.

-**irepl** *repl* Sets the replacement algorithm for the LEON instruction cache. Allowed values are rnd (default) for random replacement, lru for the least-recently-used replacement algorithm and lrr for least-recently-replaced replacement algorithm.

-**ilram** *addr size*

Allocates *size* bytes of local icache scratchpad memory at address *addr*. (LEON3)

-**iwde** Set the IWDE input to 1. Default is 0. (TSC695E only)

-**logfile** *filename*

Logs the console output to *filename*. If *filename* is preceded by '+' output is append.

-**mflat** This switch should be used when the application software has been compiled with the gcc -mflat option, and debugging with gdb is done.

-mmu Adds MMU support (LEON2/3 only).

-**nb** Do not break on error exeptions when debugging through GDB

-**nfp** Disables the FPU to emulate system without FP hardware. Any FP instruction will generate an FP disabled trap.

-**nomac** Disable LEON MAC instruction. (LEON only).

-**nov8** Disable SPARC V8 MUL/DIV instructions (LEON only).

-**nosram** Disable sram on startup. SDRAM will appear at 0x40000000 (LEON only).

-**notimers** Disable the LEON timer unit.

-**nouart** Disable emulation of UARTs. All access to UART registers will be routed to the I/O module.

-**nwin** *win* Defines the number of register windows in the processor. The default is 8. Only applicabale to TSIM/LEON3.

-port *portnum*

Use *portnum* for gdb communication (port 1234 is default)

-pr Enable profiling.

-**ram** *ram_size*

Sets the amount of simulated RAM (kbyte). Default is 4096.

-**rom** *rom_size*

Sets the amount of simulated ROM (kbyte). Default is 2048.

-**rom8, -rom16**

By default, the prom area at reset time is considered to be 32-bit. Specifying -rom8 or -rom16 will initialise the memory width field in the memory configuration register to 8- or 16-bits. The only visible difference is in the instruction timing.

-**sdram** *sdram_size*

Sets the amount of simulated SDRAM (kbyte). Default is 0. (LEON only)

-**sym** *file* Read symbols from *file*. Useful for self-extracting applications

-tsc691 Emulate the TSC691 device, rather than TSC695

-tsc695e Obsolete. TSIM/ERC32 now always emulates the TSC695 device rather that the early ERC32 chip-set.

-**uart[1,2]** *device*

By default, UART1 is connected to stdin/stdout and UART2 is disconnected. This switch can be used to connect the uarts to other devices. E.g., '-uart1 /dev/ptypc' will attach UART1 to ptypc. On linux '-uart1 /dev/ptmx' can be used in which case the pseudo terminal slave's name to use will be printed. If you use minicom to connect to the uart then use minicom's *-p <pseudo terminal>* option.

input_files Executable files to be loaded into memory. The input file is loaded into the emulated memory according to the entry point for each segment. Recognized formats are elf32, aout and srecords.

3.3 Standalone mode commands

TSIM dynamically loads libreadline.so if available on your host system, and uses `readline()` to enter and edit simulator commands. If libreadline.so is not found, `fgets()` is used instead (no history and poor editing capabilities).

Below is a description of commands that are recognized by the simulator when used in standalone mode:

batch *file* Execute a batch file of TSIM commands.

+bp, break *address*

Adds an breakpoint at *address*.

bp, break Prints all breakpoints and watchpoints

-**bp, del**[*num*]

Deletes breakpoint/watchpoint *num*. If *num* is omitted, all breakpoints and watchpoints are deleted.

bt Print backtrace

cont [*count*/time]

Continue execution at present position. See the **go** command for how to specify count or time.

coverage [*enable* | *disable* | *gcc* | *save* [*file_name*] | *print address* <*len*>]

Code coverage control. Coverage can be enabled, disabled, saved to a file or printed to the console.

dis [*addr*] [*count*]

Disassemble [*count*] instructions at address [*addr*]. Default values for count is 16 and *addr* is the program counter address.

echo *string* Print <string> to the simulator window.

edac [*clear* | *cerr* | *merr* <*address*>]

Insert EDAC errors, or clear EDAC checksums (ERC32 only)

event Print events in the event queue. Only user-inserted events are printed.

flush [*all* | *icache* | *dcache* | *addr*]

Flush the LEON caches. Specifying *all* will flush both the icache and dcache. Specifying *icache* or *dcache* will flush the respective cache. Specifying *addr* will flush the corresponding line in both caches.

float Prints the FPU registers

gdb Listen for gdb connection.

go [*address*] [*count*/*time*]

The **go** command will set pc to *address* and npc to *address* + 4, and resume execution. No other initialisation will be done. If address is not given, the default load address will be assumed. If a *count* is specified, execution will stop after the specified number of instructions. If a time is given, execution will continue until *time* is reached (relative to the current time). The time can be given in micro-seconds, milliseconds, seconds, minutes, hours or days by adding 'us', 'ms', 's', 'min', 'h' or 'd' to the time expression. Example: go 0x40000000 400 ms. **Note**: for the **go** command, if the *count/time* parameter is given, *address* must be specified.

help Print a small help menu for the TSIM commands.

hist [*length*] Enable the instruction trace buffer. The *length* last executed instructions will be placed in the trace buffer. A **hist** command without *length* will display the trace buffer. Specifying a zero trace length will disable the trace buffer.

icache, dcache

Dumps the contents of tag and data cache memories (LEON only).

inc *time* Increment simulator time without executing instructions. Time is given in the same format as for the **go** command. Event queue is evaluated during the advancement of time.

load *files* Load *files* into simulator memory.

leon Display LEON peripherals registers.

mp <*1*|*2*> Synchronize two TSIM instances. See manual for details.

mec Display ERC32 MEC registers.

mem [*addr*] [*count*]

Display memory at *addr* for *count* bytes. Same default values as for **dis**. Unimplemented registers are displayed as zero.

vmem [*vaddr*] [*count*]

Same as **mem** but does a MMU translation on *vaddr* first (LEON only).

mmu Display the MMU registers (LEON only).

quit Exits the simulator.

perf [*reset*] The **perf** command will display various execution statistics. A 'perf reset' command will reset the statistics. This can be used if statistics shall be calculated only over a part of the program. The **run** and **reset** command also resets the statistic information.

prof [0|1] [*stime*]

Enable ('prof 1') or disable ('prof 0') profiling. Without parameters, profiling information is printed. Default sampling period is 1000 clock cycles, but can be changed by specifying *stime*.

reg [*reg_name value*]

Prints and sets the IU registers in the current register window. **reg** without parameters prints the IU registers. **reg** *reg_name value* sets the corresponding register to *value*. Valid register names are psr, tbr, wim, y, g1-g7, o0-o7 and l0-l7. To view the other register windows, use **reg** *w*n, where n is 0 - 7.

reset Performs a power-on reset. This command is equal to **run** 0.

restore <*file*>

Restore simulator state from *file*.

run [*count/time*]

Resets the simulator and starts execution from address 0. The event queue is emptied but any set breakpoints remain. See the **go** command on how to specify the time or count.

save <*file*> Save simulator state *file*.

step Equal to **trace** 1.

sym [*file*] Load symbol table from *file*. If *file* is omitted, prints current (.text) symbols.

version Prints the TSIM version and build date.

watch *address*

Adds a watchpoint at *address*.

wmem, wmemh, wmemb<*address*> <*value*>

Write a word, half-word or byte directly to simulated memory.

Typing a 'Ctrl-C' will interrupt a running simulator. Short forms of the commands are allowed, e.g **c**, **co**, or **con**, are all interpreted as **cont**.

3.4 Symbolic debug information

TSIM will automatically extract (.text) symbol information from elf-files. The symbols can be used where an address is expected:

```
tsim> bre main
breakpoint 3 at 0x020012f0: main
tsim> dis strcmp 5
02002c04 84120009 or      %o0, %o1, %g2
```

```
02002c08 8088a003 andcc   %g2, 0x3, %g0
02002c0c 3280001a bne,a   0x02002c74
02002c10 c64a0000 ldsb    [%o0], %g3
02002c14 c6020000 ld      [%o0], %g3
```

The **sym** command can be used to display all symbols, or to read in symbols from an alternate (elf) file:

```
tsim> sym /opt/rtems/src/examples/samples/dhry
read 234 symbols
tsim> sym
0x02000000 L _text_start
0x02000000 L _trap_table
0x02000000 L text_start
0x02000000 L start
0x0200102c L _window_overflow
0x02001084 L _window_underflow
0x020010dc L _fpdis
0x02001a4c T Proc_3
```

Reading symbols from alternate files is necessary when debugging self-extracting applications, such as boot-proms created with mkprom or linux/uClinux.

3.5 Breakpoints and watchpoints

TSIM supports execution breakpoints and write data watchpoints. In standalone mode, hardware breakpoints are always used and no instrumentation of memory is made. When using the gdb interface, the gdb 'break' command normally uses software breakpoints by overwriting the breakpoint address with a 'ta 1' instruction. Hardware breakpoints can be inserted by using the gdb 'hbreak' command or by starting tsim with -hwbp, which will force the use of hardware breakpoints also for the gdb 'break' command. Data write watchpoints are inserted using the 'watch' command. A watchpoint can only cover one word address, block watchpoints are not available.

3.6 Profiling

The profiling function calculates the amount of execution time spent in each subroutine of the simulated program. This is made without intervention or instrumentation of the code by periodically sample the execution point and the associated call tree. Cycles in the call graph are properly handled, as well as sections of the code where no stack is available (e.g. trap handlers). The profiling information is printed as a list sorted on highest execution time ration. Profiling is enabled through the **prof 1** command. The sampling period is by default 1000 clocks which typically provides the best compromise between accuracy and performance. Other sampling periods can also be set through the **prof 1** ***n*** command. Profiling can be disabled through the **prof 0** command. Below is an example profiling the dhrystone benchmark:

```
bash$tsim-erc32 /opt/rtems/src/examples/samples/dhry
tsim> pro 1
profiling enabled, sample period 1000
tsim> go
resuming at 0x02000000
Execution starts, 200000 runs through Dhrystone
Stopped at time 23375862 (1.670e+00 s)
tsim> pro

function      samples     ratio(%)
start         36144       100.00
_start        36144       100.00
main          36134        99.97
Proc_1        10476        28.98
Func_2         9885        27.34
strcmp         8161        22.57
Proc_8         2641         7.30
.div           2097         5.80
Proc_6         1412         3.90
Proc_3         1321         3.65
Proc_2         1187         3.28
.umul          1092         3.02
Func_1         777          2.14
Proc_7         772          2.13
Proc_4         731          2.02
Proc_5         453          1.25
Func_3         227          0.62
printf           8          0.02
vfprintf         8          0.02
_vfprintf_r      8          0.02

tsim>
```

3.7 Code coverage

To aid software verification, the professional version of TSIM includes support for code coverage. When enabled, code coverage keeps a record for each 32-bit word in the emulated memory and monitors whether the location has been read, written or executed. The coverage function is controlled by the coverage command:

coverage enable	enable coverage
coverage disable	disable coverage
coverage save [filename]	write coverage data to file (file name optional)
coverage print address [len]	print coverage data to console, starting at address
coverage gcc exec-file [src-file]	print source code with back-annotated coverage information
coverage clear	reset coverage data

The coverage data for each 32-bit word of memory consists of a 3-bit field, with bit0 (lsb) indicating that the word has been executed, bit1 indicating that the word has been written, and bit2 that the word has been read. As an example, a coverage data of 0x6 would indicate that the word has been read and written, while 0x1 would indicate that the word has been executed. When the coverage data is printed to the console or save to a file, it is presented for one block of 32 words (128 bytes) per line:

```
tsim> cov print start
02000000 : 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0
02000080 : 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
02000100 : 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
02000180 : 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

When the code coverage is saved to file, only blocks with at least one coverage field set are written to the file. Block that have all the coverage fields set to zero are not saved in order to decrease the file size. Note that only the internally emulated memory (prom and ram) are subject for code coverage, any memory emulated in the user's I/O module must be handled by a user-defined coverage function.

When coverage is enabled, disassembly will include an extra column after the address, indicating the coverage data. This makes it easier to analyse which instructions has not been executed:

```
tsim> di start

 02000000 1 a0100000   clr      %l0
 02000004 1 29008004   sethi    %hi(0x2001000), %l4
 02000008 1 81c52000   jmp      %l4
 0200000c 1 01000000   nop
 02000010 0 91d02000   ta       0x0
 02000014 0 01000000   nop
 02000018 0 01000000   nop
```

The coverage data is not saved or restored during check-pointing operations. When enabled, the coverage function reduces the simulation performance of about 30%. When disabled, the coverage function does not impact simulation performance. Individual coverage fields can be read and written using the TSIM function interface using the tsim_coverage() call (see "Function interface" on page 34). Enabling and disabling the coverage functionality from the function interface should be done using tsim_cmd().

Coverage information can also be back-annotated to the source code of applications compiled with gcc (sparc-rtems-gcc). The command '**coverage gcc** *exec-file src-file*' will produce a file called *src-file.cov* where each executable line is either marked with the line number (if it has been executed), or '#####' if it has not been executed. The exec-file is the binary file which has been executed by TSIM, while the src-file is (one of) the source files. Note that the binary file must have been compiled by sparc-rtems-gcc with debugging enabled (-g). Also, the LECCS cross-compiler must be installed on the host and in the execution path. TSIM uses the LECSS tools (sparc-rtems-objdump) to exctract the debug information from the binary.

The example below shows and example of the command and annotated source code:

```
tsim> coverage gcc stanford stanford.c
coverage for stanford.c:  95.4%
tsim> q
$ cat stanford.c.cov
.
.

642       kount = 0;
643       if (Fit (0, m))
644           n = Place (0, m);
          else
######        printf ("Error1 in Puzzle\n");
647       if (!Trial (n))
######        printf ("Error2 in Puzzle.\n");
649       else if (kount != 2005)
######        printf ("Error3 in Puzzle.\n");
           };
```

The example above shows that there are three executable lines which has not been executed. Note when the code is compiled with optimisation (-O or O2), some lines which seem to have executable code might be marked as not executable. This is beacause the optimisation process has either removed them or merged them with other lines.

3.8 Check-pointing

The **professional** version of TSIM can save and restore its complete state, allowing to resume simulation from a saved check-point. Saving the state is done with the **save** command:

tsim> save *file_name*

The state is save to *file_name*.tss. To restore the state, use the **restore** command:

tsim> restore *file_name*

The state will be restored from *file_name*.tss. Restore directly at startup can be performed with the '**-rest** *file_name'* command line switch.

Note that TSIM command line options are not stored (such as alternate UART devices, etc.).

3.9 Performance

TSIM is highly optimised, and capable of simulating ERC32 systems faster than realtime. On high-end Athlon processors, TSIM achieves more than 1 MIPS / 100 MHz (cpu frequency of host). Enabling various debugging features such as watchpoints, profiling and code coverage can however reduce the simulation performance with up to 40%.

3.10 Backtrace

The bt command will display the current call backtrace and associated stack pointer;

```
tsim> bt
    %pc          %sp
#0  0x0200190c   0x023ffcc8 Proc_1 + 0xf0
#1  0x02001520   0x023ffd38 main + 0x230
#2  0x02001208   0x023ffe00 _start + 0x60
#3  0x02001014   0x023ffe40 start + 0x1014
```

3.11 Connecting to gdb

TSIM can act as a remote target for gdb, allowing symbolic debugging of target applications. To initiate gdb communication, start the simulator with the **-gdb** switch or use the TSIM **gdb** command:

```
bash-2.04$ ./tsim -gdb

TSIM/LEON - remote SPARC simulator, build 2001.01.10 (demo version)
serial port A on stdin/stdout
allocated 4096 K RAM memory
allocated 2048 K ROM memory
gdb interface: using port 1234
```

Then, start gdb in a different window and connect to TSIM using the extended-remote protocol:

```
bash-2.04$ sparc-rtems-gdb t4.exe

(gdb) tar extended-remote localhost:1234
Remote debugging using localhost:1234
0x0 in ?? ()
(gdb)
```

To load and start the application, use the gdb **load** and **cont** command.

```
(gdb) load
Loading section .text, size 0x14e50 lma 0x40000000
Start address 0x40000000 , load size 87184
Transfer rate: 697472 bits/sec, 278 bytes/write.
(gdb) cont
Continuing
```

To interrupt simulation, Ctrl-C can be typed in both gdb and TSIM windows. The program can be restarted using the gdb **run** command but a **load** has first to be executed to reload the program image into the simulator:

```
(gdb) load
Loading section .text, size 0x14e50 lma 0x40000000
Loading section .data, size 0x640 lma 0x40014e50
Start address 0x40000000 , load size 87184
Transfer rate: 697472 bits/sec, 278 bytes/write.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/jgais/src/gnc/t4.exe
```

If gdb is detached using the **detach** command, the simulator returns to the command prompt, and the program can be debugged using the standard TSIM commands. The simulator can also be re-attached to gdb by issuing the **gdb** command to the simulator (and the **target** command to gdb). While attached, normal TSIM commands can be executed using the gdb **monitor** command. Output from the TSIM commands is then displayed in the gdb console.

TSIM translates SPARC traps into (unix) signals which are properly communicated to gdb. If the application encounters a fatal trap, simulation will be stopped exactly on the failing instruction. The target memory and register values can then be examined in gdb to determine the error cause.

Profiling an application executed from gdb is possible if the symbol table is loaded in TSIM before execution is started. gdb does not download the symbol information to TSIM, so the symbol table should be loaded using the monitor command:

```
(gdb) monitor sym t4.exe
read 158 symbols
```

When an application that has been compiled using the gcc -mflat option is debugged through gdb, TSIM should be started with -mflat in order to generate the correct stack frames to gdb.

4 Emulation characteristics

4.1 Common behaviour

4.1.1 Timing

The TSIM simulator is cycle-true, i.e a simulator time is maintained and incremented according processor instruction timing and memory latency. Tracing using the **trace** command will display the current simulator time in the left column. This time indicates when the instruction is fetched. Cache misses, waitstates or data dependencies will delay the following fetch according to the incurred delay.

4.1.2 UARTs

If the baudrate register is written by the application software, the UARTs will operate with correct timing. If the baudrate is left at the default value, or if the **-fast_uart** switch was used, the UARTs operate at infinite speed. This means that the transmitter holding register always is empty and a transmitter empty interrupt is generated directly after each write to the transmitter data register. The receivers can never overflow or generate errors.

Note that with correct UART timing, it is possible that the last character of a program is not displayed on the console. This can happen if the program forces the processor in error mode, thereby terminating the simulation, before the last character has been shifted out from the transmitter shift register. To avoid this, an application should poll the UART status register and not force the processor in error mode before the transmitter shift registers are empty. The real hardware does not exhibit this problem since the UARTs continue to operate even when the processor is halted.

4.1.3 Floating point unit (FPU)

The simulator maps floating-point operations on the hosts floating point capabilities. This means that accuracy and generation of IEEE exceptions is host dependent. The simulator implements (to some extent) data-dependant execution timing as in the real MEKIO FPU.

4.1.4 Delayed write to special registers

The SPARC architecture defines that a write to the special registers (%psr, %wim, %tbr, %fsr, %y) may have up to 3 delay cycles, meaning that up to 3 of the instructions following a special register write might not 'see' the newly written value due to pipeline effects. While ERC32 and LEON have between 2 and 3 delay cycles, TSIM has 0. This does not affect simulation accuracy or timing as long as the SPARC ABI recommendations are followed that each special register write must always be followed by three NOP. If the three NOP are left out, the software might fail on real hardware while still executing 'correctly' on the simulator.

4.1.5 Idle-loop optimisation

To minimise power consumption, LEON and ERC32 applications will typically place the processor in power-down mode when the idle task is scheduled in the operation system. In power-down mode, TSIM increments the event queue without executing any instructions, thereby significantly improving simulation performance. However, some (poorly written) code might use a busy loop (BA 0) instead of triggering power-down mode. The **-bopt** switch will enable a detection mechanism which will identify such behaviour and optimise the simulation as if the power-down mode was entered.

4.2 ERC32 specific emulation

4.2.1 Processor emulation

TSIM/ERC32 emulates the behaviour of the TSC695 processor from Atmel by default. The parallel execution between the IU and FPU is modelled, as well as stalls due to operand dependencies (IU & FPU). Starting TSIM with the **-tsc691** will enable TSC691 emulation (3-chip ERC32).

4.2.2 MEC emulation

The following list outlines the implemented MEC registers:

Register	Address	Status
MEC control register	0x01f80000	implemented
Software reset register	0x01f80004	implemented
Power-down register	0x01f80008	implemented
Memory configuration register	0x01f80010	partly implemented
IO configuration register	0x01f80014	implemented
Waitstate configuration register	0x01f80018	implemented
Access protection base register 1	0x01f80020	implemented
Access protection end register 1	0x01f80024	implemented
Access protection base register 2	0x01f80028	implemented
Access protection end register 2	0x01f8002c	implemented
Interrupt shape register	0x01f80044	implemented
Interrupt pending register	0x01f80048	implemented
Interrupt mask register	0x01f8004c	implemented
Interrupt clear register	0x01f80050	implemented
Interrupt force register	0x01f80054	implemented
Watchdog acknowledge register	0x01f80060	implemented
Watchdog trap door register	0x01f80064	implemented
RTC counter register	0x01f80080	implemented
RTC counter program register	0x01f80080	implemented
RTC scaler register	0x01f80084	implemented
RTC scaler program register	0x01f80084	implemented
GPT counter register	0x01f80088	implemented
GPT counter program register	0x01f80088	implemented
GPT scaler register	0x01f8008c	implemented
GPT scaler program register	0x01f8008c	implemented
Timer control register	0x01f80098	implemented
System fault status register	0x01f800A0	implemented
First failing address register	0x01f800A4	implemented
GPI configuration register	0x01f800A8	I/O module callback
GPI data register	0x01f800AC	I/O module callback
Error and reset status register	0x01f800B0	implemented
Test control register	0x01f800D0	implemented
UART A RX/TX register	0x01f800E0	implemented
UART B RX/TX register	0x01f800E4	implemented
UART status register	0x01f800E8	implemented

The MEC registers can be displayed with the **mec** command, or using **mem** ('mem 0x1f80000 256'). The registers can also be written using **wmem** (e.g. 'wmem 0x1f80000 0x1234'). When written, care has to be taken not to write an unimplemented register bit with '1', or a MEC parity error will occur.

4.2.3 Interrupt controller

Internal interrupts are generated as defined in the MEC specification. All 15 interrupts can be tested via the interrupt force register. External interrupts can be generated through loadable modules.

4.2.4 Watchdog

The watchdog timer operate as defined in the MEC specification. The frequency of the watchdog clock can be specified using the -**wdfreq** switch. The frequency is specified in MHz.

4.2.5 Power-down mode

The power-down register (0x01f800008) is implemented as in the specification. A Ctrl-C in the simulator window will exit the power-down mode. In power-down mode, the simulator skips time until the next event in the event queue, thereby significantly increasing the simulation speed.

4.2.6 Memory emulation

The amount of simulated memory is configured through the -ram and -rom switches. The ram size can be between 256 kbytes and 32 Mbyte, the rom size between 128 kbyte and 4 Mbyte. Access to unimplemented MEC registers or non-existing memory will result in a memory exception trap.

The memory configuration register is used to decode the simulated memory. The fields RSIZ and PSIZ are used to set RAM and ROM size, the remaining fields are not used. NOTE: after reset, the MEC is set to decode 128 kbytes of ROM and 256 kbytes of RAM. The memory configuration register has to be updated to reflect the available memory. The waitstate configuration register is used to generate waitstates. This register must also be updated with the correct configuration after reset.

4.2.7 EDAC operation

The EDAC operation of ERC32 is implemented on the simulated RAM area (0x2000000 - 0x2FFFFFF). The ERC32 Test Control Register can be used to enable the EDAC test mode and insert EDAC errors to test the operation of the EDAC. The **edac** command can be used to monitor the number of errors in the memory, to insert new errors, or clear all errors. To see the current memory status, use the **edac** command without parameters:

```
tsim> edac
ram error count : 2
 0x20000000 : MERR
 0x20000040 : CERR
```

TSIM keeps track of the number of errors currently present, and reports the total error count, the address of each error, and its type. The errors can either be correctable (CERR) or non-correctable (MERR). To insert an error using the edac command, do '**edac cerr** *addr'* or '**edac merr** *addr*' :

```
tsim> edac cerr 0x2000000
correctable error at 0x02000000
tsim> edac
ram error count : 1
 0x20000000 : CERR
```

To remove all injected errors, do **edac clear**. When accessing a location with an EDAC error, the behaviour of TSIM is identical to the real hardware. A correctable error will trigger interrupt 1, while un-correctable errors will cause a memory exception. The operation of the FSFR and FAR registers are fully implemented.

NOTE: the EDAC operation affect simulator peformance *when there are inserted errors in the memory.* To obtain maximum simulation performance, any diagnostic software should remove all inserted errors aftter having performed an EDAC test.

4.2.8 Extended RAM and I/O areas

TSIM allows emulation of user defined I/O devices through loadable modules. EDAC emulation of exteanded RAM areas is not supported.

4.2.9 SYSAV signal

TSIM emulates changes in the SYSAV output by calling the command() callback in the I/O module with either "sysav 0" or "sysav 1" on each changes of SYSAV.

4.2.10 EXTINTACK signal

TSIM emulates assertion of the EXTINTACK output by calling the command() callback in the I/O module with "extintack" on each assertion. Note that EXTINTACK is only asserted for one external interrupt as programmed in the MEC interrupt shape register.

4.2.11 IWDE signal

The **TSC695E** processor input signal can be controlled by the -iwde switch at start-up. If the switch is given, the IWDE signal will be high, and the internal watchdog enabled. If -iwde is not given, IWDE will be low and the internal watchdog will be disabled. Note that the simulator must started in TSC695E-mode using the -tsc695e switch, for this option to take effect.

4.3 LEON2 specific emulation

4.3.1 Processor

The LEON2 version of TSIM emulates the behaviour of the LEON2 VHDL model. The (optional) MMU can be emulated using the *-mmu* switch.

4.3.2 Cache memories

The evaluation version of LEON implements 2*4kbyte caches, with 16 bytes per line. The commercial TSIM version can emulate any permissible cache configuration using the -icsize, -ilsize, -dcsize and -dlsize options. Allowed sizes are 1 - 64 kbyte with 8 - 32 bytes/line. The characteristics of the leon multi-set caches (as of leon2-1.0.8) can be emulated using the -isets, -dsets, -irepl, -drelp, -ilock and -dlock options. Diagnostic cache reads/writes are implemented. The simulator commands **icache** and **dcache** can be used to display cache contents. Starting TSIM with -at697e will configure that caches according to the Atmel AT697E device.

4.3.3 LEON peripherals registers

The LEON peripherals registers can be displayed with the **leon** command, or using **mem** ('mem 0x80000000 256'). The registers can also be written using **wmem** (e.g. 'wmem 0x80000000 0x1234').

4.3.4 Interrupt controller

External interrupts are not implemented, so the I/O port interrupt register has no function. Internal interrupts are generated as defined in the LEON specification. All 15 interrupts can also be generated from the user-defined I/O module using the set_irq() callback.

4.3.5 Power-down mode

The power-down register 0x80000018) is implemented as in the specification. A Ctrl-C in the simulator window will exit the power-down mode. In power-down mode, the simulator skips time until the next event in the event queue, thereby significantly increasing the simulation speed.

4.3.6 Memory emulation

The memory configuration registers 1/2 are used to decode the simulated memory. The memory configuration registers has to be programmed by software to reflect the available memory, and the number and size of the memory banks. The waitstates fields must also be programmed with the correct configuration after reset. Both SRAM and SDRAM can be emulated.

Using the **-banks** option, it is possible to set over how many ram banks the external SRAM is divided in. Note that software compiled with BCC/RCC, and **not** run through mkprom must **not** use this option. For mkprom encapsulated programs, it is essential that the **same** ram size and bank number setting is used for both mkprom and TSIM.

The memory EDAC of LEON2-FT is not implemented.

4.3.7 SPARC V8 MUL/DIV/MAC instructions

TSIM/LEON optionally supports the SPARC V8 multiply, divide and MAC instruction. To correctly emulate LEON systems which do not implement these instructions, use the **-nomac** to disable the MAC instruction and/or -**nov8** to disable multiply and divide instructions.

4.4 LEON3 specific emulation

4.4.1 General

The LEON3 version of TSIM emulates the behaviour of the LEON3MP template VHDL model distributed in the GRLIB-1.0 IP library. The system includes the following modules: LEON3 processor, APB bridge, IRQMP interrupt controller, LEON2 memory controller, GPTIMER timer unit with two 24-bit timers, two APBUART uarts. The AHB/APB plug&play information is provided at address 0xFFFFF000 - 0xFFFFFFFF (AHB) and 0x800FF000 - 0x800FFFFF (APB).

4.4.2 Processor

The instruction timing of the emulated LEON3 processor is modelled after LEON3 VHDL model in GRLIB IP library. The processor can be be configured with 2 - 32 register windows using the *-nwin* switch. The MMU can be emulated using the *-mmu* switch. Local scratch pad ram is not emulated.

4.4.3 Cache memories

The evaluation version of TSIM/LEON3 implements 2*4kbyte caches, with 16 bytes per line. The commercial TSIM version can emulate any permissible cache configuration using the -icsize, -ilsize, -dcsize and -dlsize options. Allowed sizes are 1 - 64 kbyte with 8 - 32 bytes/line. The characteristics of the leon multi-set caches can be emulated using the -isets, -dsets, -irepl, -drelp, -ilock and -dlock options. Diagnostic cache reads/writes are implemented. The simulator commands **icache** and **dcache** can be used to display cache contents.

4.4.4 Power-down mode

The LEON3 power-down function is implemented as in the specification. A Ctrl-C in the simulator window will exit the power-down mode. In power-down mode, the simulator skips time until the next event in the event queue, thereby significantly increasing the simulation speed.

4.4.5 LEON3 peripherals registers

The LEON3 peripherals registers can be displayed with the **leon** command, or using **mem** ('mem 0x80000000 256'). The registers can also be written using **wmem** (e.g. 'wmem 0x80000000 0x1234').

4.4.6 Interrupt controller

The IRQMP interrupt controller is fully emulated as described in the GRLIB IP Manual. The IRQMP registers are mapped at address 0x80000200. All 15 interrupts can also be generated from the user-defined I/O module using the set_irq() callback.

4.4.7 Memory emulation

The LEON2 memory controller is emulated in the LEON3 version of TSIM. The memory configuration registers 1/2 are used to decode the simulated memory. The memory configuration registers has to be programmed by software to reflect the available memory, and the number and size of the memory banks. The waitstates fields must also be programmed with the correct configuration after reset. Both SRAM and SDRAM can be emulated.

Using the **-banks** option, it is possible to set over how many ram banks the external SRAM is divided in. Note that software compiled with BCC/RCC, and **not** run through mkprom must **not** use this option. For mkprom encapsulated programs, it is essential that the **same** ram size and bank number setting is used for both mkprom and TSIM.

4.4.8 SPARC V8 MUL/DIV/MAC instructions

TSIM/LEON3 optionally supports the SPARC V8 multiply, divide and MAC instruction. To correctly emulate LEON systems which do not implement these instructions, use the **-nomac** to disable the MAC instruction and/or -**nov8** to disable multiply and divide instructions.

4.4.9 Custom instruction emulation

TSIM/LEON allows the emulation of custom (non-SPARC) instructions. A handler for non-standard instruction can be installed using the tsim_ext_ins() callback function (see "Function interface" on page 34). The function handler is called each time an instruction is encountered that would cause an unimplemented instruction trap. The handler is passed the opcode and all processor registers in a pointer, allowing it to decode and emulate a custom instruction, and update the processor state.

The definition for the custom instruction handler is:

```
int myhandler (struct ins_interface *r);
```

The pointer *r is a structure containing the current instruction opcode and processor state:

```
 struct ins_interface {
    uint32          psr;   /* Processor status registers */
    uint32          tbr;   /* Trap base register */
    uint32          wim;   /* Window maks register */
    uint32          g[8]; /* Global registers */
    uint32          r[128];/* Windowed register file */
    uint32          y;     /* Y register */
    uint32          pc;    /* Program counter *
    uint32          npc;   /* Next program counter */
    uint32          inst;/* Current instruction */
    uint32          icnt;/* Clock cycles in curr inst */
    uint32          asr17;
    uint32          asr18;
};
```

SPARC uses an overlapping windowed register file, and accessing registers must be done using the current window pointer (%psr[4:0]). To access registers %r8 - %r31 in the current window, use:

```
cwp = r->psr & 7;
regval = r->r[((cwp << 4) + RS1) % (nwindows * 16)];
```

Note that global registers (%r0 - %r7) should always be accessed by r->g[RS1];

The return value of the custom handler indicates which trap the emulated instruction has generated, or 0 if no trap was caused. If the handler could not decode the instruction, 2 should be returned to cause an unimplemented instruction trap.

The number of clocks consumed by the instruction should be returned in r->icnt; This value is by default 1, which corresponds to a fully pipelined instruction without data interlock. The handler should not increment the %pc or %npc registers, as this is done by TSIM.

5 Loadable modules

5.1 TSIM I/O emulation interface

User-defined I/O devices can be loaded into the simulator through the use of loadable modules. As the real processor, the simulator primarily interacts with the emulated device through read and write requests, while the emulated device can optionally generate interrupts and DMA requests. This is implemented through the module interface described below. The interface is made up of two parts; one that is exported by TSIM and defines TSIM functions and data structures that can be used by the I/O device; and one that is exported by the I/O device and allows TSIM to access the I/O device. Address decoding of the I/O devices is straight-forward: all access that do not map on the internally emulated memory and control registers are forwarded to the I/O module.

TSIM exports two structures: simif and ioif. The simif structure defines functions and data structures belonging to the simulator core, while ioif defines functions provided by system (ERC32/LEON) emulation. At start-up, TSIM searches for 'io.so' in the current directory, but the location of the module can be specified using the **-iom** switch. Note that the module must be compiled to be position-independent, i.e. with the -fPIC switch (gcc). The win32 version of TSIM loads io.dll instead of io.so. See the iomod directory in the TSIM distribution for an example io.c and how to build the .so and .dll modules.

5.1.1 simif structure

The simif structure is defined in sim.h:

```
struct sim_options {
    int phys_ram;
    int phys_sdram;
    int phys_rom;
    double freq;
    double wdfreq;
};
struct sim_interface {
    struct sim_options *options; /* tsim command-line options */
    uint64 *simtime; /* current simulator time */
    void (*event)(void (*cfunc)(), uint32 arg, uint64 offset);
    void (*stop_event)(void (*cfunc)());
    int *irl; /* interrup request level */
    void (*sys_reset)(); /* reset processor */
    void (*sim_stop)(); /* stop simulation */
};
struct sim_interface simif;/* exported simulator functions */
```

The elements in the structure has the following meaning:

```
struct sim_options *options;
```

Contains some tsim startup options. options.freq defines the clock frequency of the emulated processor and can be used to correlate the simulator time to the real time.

```
uint64 *simtime;
```

Contains the current simulator time. Time is counted in clock cycles since start of simulation. To calculate the elapsed real time, divide simtime with options.freq.

```
void (*event)(void (*cfunc)(), int arg, uint64 offset);
```

TSIM maintains an event queue to emulate time-dependant functions. The event() function inserts an event in the event queue. An event consists of a function to be called when the event expires, an argument with which the function is called, and an offset (relative the current time) defining when the event should expire. NOTE: the event() function may NOT be called from a signal handler installed by the I/

O module, but only from event() callbacks or at start of simulation. The event queue can hold a maximum of 2048 events.

```
void (*stop_event)(void (*cfunc)());
```

stop_event() will remove all events from the event queue which has the calling function equal to cfunc(). NOTE: the stop_event() function may NOT be called from a signal handler installed by the I/O module.

```
int *irl;
```

Current IU interrupt level. Should not be used by I/O functions unless they explicitly monitor theses lines.

```
void (*sys_reset)();
```

Performs a system reset. Should only be used if the I/O device is capable of driving the reset input.

```
void (*sim_stop)();
```

Stops current simulation. Can be used for debugging purposes if manual intervention is needed after a certain event.

5.1.2 ioif structure

ioif is defined in sim.h:

```
structio_interface {
    void (*set_irq)(int irq, int level);
    int (*dma_read)(uint32 addr, uint32 *data, int num);
    int (*dma_write)(uint32 addr, uint32 *data, int num);
};
extern struct io_interface ioif; /* exported processor interface */
```

The elements of the structure have the following meaning:

```
void (*set_irq)(int irq, int level);
```

ERC32 use: drive the external MEC interrupt signal. Valid interrupts are 0 - 5 (corresponding to MEC external interrupt 0 - 4, and EWDINT) and valid levels are 0 or 1. Note that the MEC interrupt shape register controls how and when processor interrupts are actually generated. When -nouart has been used, MEC interrupts 4, 5 and 7 can be generated by calling set_irq() with irq 6, 7 and 9 (level is not used in this case).

LEON use: set the interrupt pending bit for interrupt irq. Valid values on irq is 1 - 15. Care should be taken not to set interrupts used by the LEON emulated peripherals. Note that the LEON interrupt control register controls how and when processor interrupts are actually generated. Note that level is not used with LEON.

```
int (*dma_read)(uint32 addr, uint32 *data, int num);
int (*dma_write)(uint32 addr, uint32 *data, int num);
```

Performs DMA transactions to/from the emulated processor memory. Only 32-bit word transfers are allowed, and the address must be word aligned. On bus error, 1 is returned, otherwise 0. For ERC32, the DMA transfer uses the external DMA interface. For LEON, DMA takes place on the AMBA AHB bus.

5.1.3 Structure to be provided by I/O device

io.h defines the structure to be provided by the emulated I/O device:

```
struct io_subsystem {
    void (*io_init)();/* called once on start-up */
    void (*io_exit)();/* called once on exit */
    void (*io_reset)();/* called on processor reset */
    void (*io_restart)();/* called on simulator restart */
    int (*io_read)(unsigned int addr, int *data, int *ws);
    int (*io_write)(unsigned int addr, int *data, int *ws, int size);
    char *(*get_io_ptr)(unsigned int addr, int size);
    void (*command)(char * cmd);/* I/O specific commands */
    void (*sigio)();/* called when SIGIO occurs */
    void (*save)(char *fname);/* save simulation state */
    void (*restore)(char *fname);/* restore simulation state */
};
extern struct io_subsystem *iosystem; /* imported I/O emulation functions */
```

The elements of the structure have the following meanings:

```
void (*io_init)();
```

Called once on simulator startup. Set to NULL if unused.

```
void (*io_exit)();
```

Called once on simulator exit. Set to NULL if unused.

```
void (*io_reset)();
```

Called every time the processor is reset (i.e also startup). Set to NULL if unused.

```
void (*io_restart)();
```

Called every time the simulator is restarted (simtime set to zero). Set to NULL if unused.

```
int (*io_read)(unsigned int addr, int *data, int *ws);
```

Processor read call. The processor always reads one full 32-bit word from addr. The data should be returned in *data, the number of waitstates should be returned in *ws. If the access would fail (illegal address etc.), 1 should be returned, on success 0.

```
int (*io_write)(unsigned int addr, int *data, int *ws, int size);
```

Processor write call. The size of the written data is indicated in size: 0 = byte, 1 = half-word, 2 = word, 3 = doubleword. The address is provided in addr, and is always aligned with respect to the size of the written data. The number of waitstates should be returned in *ws. If the access would fail (illegal address etc.), 1 should be returned, on success 0.

```
char * (*get_io_ptr)(unsigned int addr, int size);
```

TSIM can access emulated memory in the I/O device in two ways: either through the io_read/io_write functions or directly through a memory pointer. get_io_ptr() is called with the target address and transfer size (in bytes), and should return a character pointer to the emulated memory array if the address and size is within the range of the emulated memory. If outside the range, -1 should be returned. Set to NULL if not used.

```
int (*command)(char * cmd);
```

The I/O module can optionally receive command-line commands. A command is first sent to the AHB and I/O modules, and if not recognised, the to TSIM. command() is called with the full command string in *cmd. Should return 1 if the command is recognized, otherwise 0. TSIM/ERC32 also calls this callback when the SYSAV bit in the ERSR register changes. The commands "sysav 0" and "sysav 1" are then issued. When TSIM commands are issued through the gdb 'monitor' command, a return value of 0 or 1 will result in an 'OK' response to the gdb command. A return value > 1 will send the value itself as the gdb response. A return value < 1 will truncate the lsb 8 bits and send them back as a gdb error response: 'E*nn*'.

```
void (*sigio)();
```

Not used as of tsim-1.2, kept for compatibility reasons.

```
void (*save)(char *fname);
```

The save() function is called when save command is issued in the simulator. The I/O module should save any required state which is needed to completely restore the state at a later stage. *fname points to the base file name which is used by TSIM. TSIM saves its internal state to fname.tss. It is suggested that the I/O module save its state to fname.ios. Note that any events placed in the event queue by the I/O module will be saved (and restored) by TSIM.

```
void (*restore)(char *fname);
```

The restore() function is called when restore command is issued in the simulator. The I/O module should restore any required state to resume operation from a saved check-point. *fname points to the base file name which is used by TSIM. TSIM restores its internal state from fname.tss.

5.1.4 Cygwin specific io_init()

Due to problems of resolving cross-referenced symbols in the module loading when using Cygwin, the io_init() routine in the I/O module must initialise a local copy of simif and ioif. This is done by providing the following io_init() routine:

```
static void io_init(struct sim_interface sif, struct io_interface iif)
{
#ifdef __CYGWIN32__
/* Do not remove, needed when compiling on Cygwin! */
    simif = sif;
    ioif = iif;
#endif
/* additional init code goes here */
};
```

5.2 LEON AHB emulation interface

In addition to the above described I/O modules, TSIM also allows emulation of the LEON2/3 processor core with a completely user-defined memory and I/O architecture. By loading an AHB module (ahb.so), the internal memory emulation is disabled. The emulated processor core communicates with the AHB module using an interface similar to the AHB master interface in the real LEON VHDL model. The AHB module can then emulate the complete AHB bus and all attached units.

The AHB module interface is made up of two parts; one that is exported by TSIM and defines TSIM functions and data structures that can be used by the AHB module; and one that is exported by the AHB module and allows TSIM to access the emulated AHB devices.

At start-up, TSIM searches for 'ahb.so' in the current directory, but the location of the module can be specified using the **-ahbm** switch. Note that the module must be compiled to be position-independent, i.e. with the -fPIC switch (gcc). The win32 version of TSIM loads ahb.dll instead of ahb.so. See the iomod directory in the TSIM distribution for an example ahb.c and how to build the .so /.dll modules.

5.2.1 procif structure

TSIM exports one structure for AHB emulation: procif. The procif structure defines a few functions giving access to the processor emulation and cache behaviour. The procif structure is defined in tsim.h:

```
struct proc_interface {
    void (*set_irl)(int level); /* generate external interrupt */
    void (*cache_snoop)(uint32 addr);
    void (*cctrl)(uint32 *data, uint32 read);
    void (*power_down)();
};
extern struct proc_interface procif;
```

The elements in the structure have the following meaning:

```
void (*set_irl)(int level);
```

Set the current interrupt level (iui.irl in VHDL model). Allowed values are 0 - 15, with 0 meaning no pending interrupt. Once the interrupt level is set, it will remain until it is changed by a new call to set_irl(). The modules interrupt callback routine should typically reset the interrupt level to avoid new interrupts.

```
void (*cache_snoop)(uint32 addr);
```

The cache_snoop() function emulates the data cache snooping of the processor. The tags to the given address will be checked, and if a match is detected the corresponding cacheline will be flushed (= the tag will be cleared).

```
void (*cctrl)(uint32 *data, uint32 read);
```

Read and write the cache control register (CCR). The CCR is attached to the APB bus in the LEON2 VHDL model, and this function can be called by the AHB module to read and write the register. If read = 1, the CCR value is returned in *data, else the value of *data is written to the CCR. The cctrl() function is only needed for LEON2 emulation, since LEON3 accesses the cache controller through a separate ASI load/store instruction.

```
void (*power_down)();
```

The LEON processor enters power down-mode when called. Only applicable to LEON2.

5.2.2 Structure to be provided by AHB module

tsim.h defines the structure to be provided by the emulated AHB module:

```
struct ahb_access {
    uint32 address;
    uint32 *data;
    uint32 ws;
    uint32 rnum;
    uint32 wsize;
    uint32 cache;
};
struct ahb_subsystem {
    void (*init)();/* called once on start-up */
    void (*exit)();/* called once on exit */
    void (*reset)();/* called on processor reset */
    void (*restart)();/* called on simulator restart */
    int (*read)(struct ahb_access *access);
    int (*write)(struct ahb_access *access);
    char *(*get_io_ptr)(unsigned int addr, int size);
    int (*command)(char * cmd); /* I/O specific commands */
    int   (*sigio)();/* called when SIGIO occurs */
    void (*save)(char * fname); /* save state */
    void (*restore)(char * fname); /* restore state */
    int (*intack)(int level); /* interrupt acknowledge */
};
extern struct ahb_subsystem *ahbsystem;   /* imported AHB emulation functions */
```

The elements of the structure have the following meanings:

```
void (*init)();
```

Called once on simulator startup. Set to NULL if unused.

```
void (*exit)();
```

Called once on simulator exit. Set to NULL if unused.

```
void (*reset)();
```

Called every time the processor is reset (i.e. also startup). Set to NULL if unused.

```
void (*restart)();
```

Called every time the simulator is restarted (simtime set to zero). Set to NULL if unused.

```
int (*read)(struct ahb_access *ahbacc);
```

Processor AHB read. The processor always reads one or more 32-bit words from the AHB bus. The ahb_access structure contains the access parameters: access.addr = read address; access.data = pointer to the first read data; access.ws = should return the number of AHB waitstates used for the complete access; access.rnum = number of words read (1 - 8); access.wsize = not used during read cycles; access.cache = should return 1 if the access is cacheable, else 0. Return values: 0 = access succeeded; 1 = access failed, generate memory exception; -1 = undecoded area, continue to decode address (I/O module or LEON registers).

```
int (*write)(struct ahb_access *ahbacc);
```

Processor AHB write. The processor can write 1, 2, 4 or 8 bytes per access. The access parameters are as for read() with the following changes: access.data = pointer to first write data; access.rnum = not used; access.wsize = defines write size as follows: 0 = byte, 1 = half-word, 2 = word, 3 = double-word. Return values as for read()

```
char * (*get_io_ptr)(unsigned int addr, int size);
```

During file load operations and displaying of memory contents, TSIM will access emulated memory through a memory pointer. get_io_ptr() is called with the target address and transfer size (in bytes), and should return a character pointer to the emulated memory array if the address and size is within the range of the emulated memory. If outside the range, -1 should be returned. Set to NULL if not used.

```
int (*command)(char * cmd);
```

The AHB module can optionally receive command-line commands. A command is first sent to the AHB and I/O modules, and if not recognised, then to TSIM. command() is called with the full command string in *cmd. Should return 1 if the command is recognized, otherwise 0. When TSIM commands are issued through the gdb 'monitor' command, a return value of 0 or 1 will result in an 'OK' response to the gdb command. A return value > 1 will send the value itself as the gdb response. A return value < 1 will truncate the lsb 8 bits and send them back as a gdb error response: 'E*nn*'.

```
void (*save)(char *fname);
```

The save() function is called when save command is issued in the simulator. The AHB module should save any required state which is needed to completely restore the state at a later stage. *fname points to the base file name which is used by TSIM. TSIM save its internal state to fname.tss. It is suggested that the AHB module save its state to fname.ahs. Note that any events placed in the event queue by the AHB module will be saved (and restored) by TSIM.

```
void (*restore)(char * fname);
```

The restore() function is called when restore command is issued in the simulator. The AHB module should restore any required state to resume operation from a saved check-point. *fname points to the base file name which is used by TSIM. TSIM restores its internal state from fname.tss.

```
int (*intack)(int level);
```

intack() is called when the processor takes an interrupt trap (tt = 0x11 - 0x1f). The level of the taken interrupt is passed in level. This callback can be used to implement interrupt controllers. intack() should return 1 if the interrupt acknowledgement was handled by the AHB module, otherwise 0. If 0 is returned, the default LEON interrupt controller will receive the intack instead.

5.2.3 Big versus little endianess

SPARC is conforms to the big endian byte ordering. This means that the most significant byte of a (half) word has lowest address. To execute efficiently on little-endian hosts (such as intel x86 PCs), emulated memory is organised on word basis with the bytes within a word arranged according the endianess of the host. Read cycles can then be performed without any conversion since SPARC always reads a full 32-bit word. During byte and half word writes, care must be taken to insert the written data properly into the emulated memory. On a byte-write to address 0, the written byte should be inserted at address 3, since this is the most significant byte according to little endian. Similarly, on a half-word write to bytes 0/1, bytes 2/3 should be written. For a complete example, see the prom emulation function in io.c.

5.3 TSIM/LEON co-processor emulation

5.3.1 FPU/CP interface

The professional version of TSIM/LEON can emulate a user-defined floating-point unit (FPU) and co-processor (CP). The FPU and CP are included into the simulator using loadable modules. To access the module, use the structure 'cp_interface' defined in io.h. The structure contains a number of functions and variables that must be provided by the emulated FPU/CP:

```
/* structure of function to be provided by an external co-processor */
struct cp_interface {
    void (*cp_init)();               /* called once on start-up */
    void (*cp_exit)();               /* called once on exit */
    void (*cp_reset)();              /* called on processor reset */
    void (*cp_restart)();            /* called on simulator restart */
    uint32 (*cp_reg)(int reg, uint32 data, int read);
    int (*cp_load)(int reg, uint32 data, int *hold);
    int (*cp_store)(int reg, uint32 *data, int *hold);
    int (*cp_exec)(uint32 pc, uint32 inst, int *hold);
    int (*cp_cc)(int *cc, int *hold); /* get condition codes */
    int *cp_status;                  /* unit status */
    void (*cp_print)();              /* print registers */
    int (*command)(char * cmd);      /* CP specific commands */
};
extern struct cp_interface *cp;   /* imported co-processor emulation functions */
```

5.3.2 Structure elements

```
void (*cp_init)();
```

Called once on simulator startup. Set to NULL if not used.

```
void (*cp_exit)();
```

Called once on simulator exit. Set to NULL if not used.

```
void (*cp_reset)();
```

Called every time the processor is reset. Set to NULL if not used.

```
void (*cp_restart)();
```

Called every time the simulator is restarted. Set to NULL if not used.

```
uint32 (*cp_reg)(int reg, uint32 data, int read);
```

Used by the simulator to perform diagnostics read and write to the FPU/CP registers. Calling cp_reg() should not have any side-effects on the FPU/CP status. 'reg' indicates which register to access: 0-31 indicates %f0-%f31/%c0- %31, 34 indicates %fsr/%csr. 'read' indicates read or write access: read==0 indicates write access, read!=0 indicates read access. Written data is passed in 'data', the return value contains the read value on read accesses.

```
int (*cp_load)(int reg, uint32 data, int *hold);
```

Used to perform FPU/CP load instructions. 'reg' indicates which register to access: 0-31 indicates %f0-%f31/%c0- %31, 34 indicates %fsr/%csr. Loaded data is passed in 'data'. If data dependency is emulated, the number of stall cycles should be return in *hold. The return value should be zero if no trap occurred or the trap number if a trap did occur (0x8 for the FPU, 0x28 for CP). A trap can occur if the FPU/CP is in exception_pending mode when the load is executed.

```
int (*cp_store)(int reg, uint32 *data, int *hold);
```

Used to perform FPU/CP store instructions. 'reg' indicates which register to access: 0-31 indicates %f0-%f31/%c0- %31, 32 indicates %fq/%cq, 34 indicates %fsr/%csr. Stored should be assigned to *data. During a STDFQ, the %pc should be assigned to data[0] while the instruction opcode to data[1]. If data dependency is emulated, the number of stall cycles should be return in *hold. The return value should be zero if no trap occurred or the trap number if a trap did occur (0x8 for the FPU, 0x28 for CP). A trap can occur if the FPU/CP is in exception_pending mode when the store is executed.

```
int (*cp_exec)(uint32 pc, uint32 inst, int *hold);
```

Execute FPU/CP instruction. The %pc is passed in 'pc' and the instruction opcode in 'inst'. If data dependency is emulated, the number of stall cycles should be return in *hold. The return value should be zero if no trap occurred or the trap number if a trap did occur (0x8 for the FPU, 0x28 for CP). A trap can occur if the FPU/CP is in exception_pending mode when a new FPU/CP instruction is executed.

```
int (*cp_cc)(int *cc, int *hold); /* get condition codes */
```

Read condition codes. Used by FBCC/CBCC instructions. The condition codes (0 - 3) should be returned in *cc. If data dependency is emulated, the number of stall cycles should be return in *hold. The return value should be zero if no trap occurred or the trap number if a trap did occur (0x8 for the FPU, 0x28 for CP). A trap can occur if the FPU/CP is in exception_pending mode when a FBCC/ CBCC instruction is executed.

```
int *cp_status;/* unit status */
```

Should contain the FPU/CP execution status: 0 = execute_mode, 1 = exception_pending, 2 = exception_mode.

```
void (*cp_print)();/* print registers */
```

Should print the FPU/CP registers to stdio.

```
int (*command)(char * cmd); /* CP specific commands */
```

User defined FPU/CP control commands. NOT YET IMPLEMENTED.

5.3.3 Attaching the FPU and CP

At startup the simulator tries to load two dynamic link libraries containing an external FPU or CP. The simulator looks for the file fp.so and cp.so in the current directory and in the search path defined by ldconfig. The location of the modules can also be defined using **-cpm** and **-fpm** switches. Each library is searched for a pointer 'cp' that points to a cp_interface structure describing the co-processor. Below is an example from fp.c:

```
struct cp_interface test_fpu = {
    cp_init, /* cp_init */
    NULL, /* cp_exit */
    cp_init, /* cp_reset */
    cp_init, /* cp_restart */
    cp_reg, /* cp_reg */
    cp_load, /* cp_load */
    cp_store, /* cp_store */
    fpmeiko, /* cp_exec */
    cp_cc, /* cp_cc */
    &fpregs.fpstate,/* cp_status */
    cp_print, /* cp_print */
    NULL   /* cp_command */
};
struct cp_interface *cp = &test_fpu; /* Attach pointer!! */
```

5.3.4 Big versus little endianess

SPARC is conforms to the big-endian byte ordering. This means that the most significant byte of a (half) word has lowest address. To execute efficiently on little-endian hosts (such as intel x86 PCs), emulated register-file is organised on word basis with the bytes within a word arranged according the endianess of the host. Double words are also in host order, and the read/write register functions must therefore invert the lsb of the register address when performing word access on little-endian hosts. Se the file fp.c for examples (cp_load(), cp_store()).

5.3.5 Additional TSIM commands

float	Shows the registers of the FPU
cp	Shows the registers of the CP

5.3.6 Example FPU

The file fp.c contains a complete SPARC FPU using the co-processor interface. It can be used as a template for implementation of other co-processors. Note that data-dependency checking for correct timing is not implemented in this version (it is however implemented in the built-in version of TSIM).

6 TSIM library (TLIB)

6.1 Introduction

TSIM is also available as a library, allowing the simulator to be integrated in a larger simulation framework. The various TSIM commands and options are accessible through a simple function interface. I/O functions can be added, and use a similar interface to the loadable I/O modules described earlier.

6.2 Function interface

The following functions are provided to access TSIM features:

```
int tsim_init (char *option);/* initialise tsim with optional params. */
```

Initialize TSIM - must be called before any other TSIM function (except tsim_set_diag()) are used. The options string can contain any valid TSIM startup option (as used for the standalone simulator), with the exception that **no** filenames for files to be loaded into memory may be given. tsim_init() may only be called once, use the TSIM **reset** command to reset the simulator without exiting. tsim_init() will return 1 on success or 0 on failure.

```
int tsim_cmd (char *cmd);/* execute tsim command */
```

Execute TSIM command. Any valid TSIM command-line command may be given. The following return values are defined:

SIGINT	Simulation stopped due to interrupt
SIGHUP	Simulation stopped normally
SIGTRAP	Simulation stopped due to breakpoint hit
SIGSEGV	Simulation stopped due to processor in error mode
SIGTERM	Simulation stopped due to program termination

```
void tsim_exit (int val);
```

Should be called to cleanup TSIM internal state before main program exits.

```
void tsim_get_regs (unsigned int *regs);
```

Get SPARC registers. regs is a pointer to an array of integers, see tsim.h for how the various registers are indexed.

```
void tsim_set_regs (unsigned int *regs);
```

Set SPARC registers. *regs is a pointer to an array of integers, see tsim.h for how the various registers are indexed.

```
void tsim_disas(unsigned int addr, int num);
```

Disassemble memory. addr indicates which address to disassemble, num indicates how many instructions.

```
void tsim_set_diag (void (*cfunc)(char *));
```

Set console output function. By default, TSIM writes all diagnostics and console messages on stdout. tsim_set_diag() can be used to direct all output to a user defined routine. The user function is called with a single string parameter containing the message to be written.

```
void tsim_set_callback (void (*cfunc)(void));
```

Set the debug callback function. Calling tsim_set_callback() with a function pointer will cause TSIM to call the callback function after each executed instruction, **when the history is enabled**. History can be enabled with the tsim_cmd() function.

```
void tsim_gdb (unsigned char (*inchar)(), void (*outchar)(unsigned char c));
```

Controls the simulator using the gdb 'extended-remote' protocol. The inchar parameter is a pointer to a function that when called, returns next character from the gdb link. The outchar parameter is a pointer to a function that sends one character to the gdb link.

```
void tsim_read(unsigned int addr, unsigned int *data);
```

Performs a read from *addr*, returning the value in **data*. Only for diagnostic use.

```
void tsim_write(unsigned int addr, unsigned int data);
```

Performs a write to *addr*, with value *data*. Only for diagnostic use.

```
void tsim_stop_event(void (*cfunc)(), int arg, int op);
```

tsim_stop_event() can remove certain event depending on the setting of arg and op. If op = 0, all instance of the callback function cfunc will be removed. If *op* = 1, events with the argument = *arg* will be removed. If *op* = 2, only the first (earliest) of the events with the argument = *arg* will be removed. NOTE: the stop_event() function may NOT be called from a signal handler installed by the I/O module.

```
void tsim_inc_time(uint64);
```

tsim_inc_time() will increment the simulator time without executing any instructions. The event queue is evaluated during the advancement of time and the event callbacks are properly called. Can not be called from event handlers.

```
int tsim_trap(int (*trap)(int tt), void (*rett)());
```

tsim_trap() is used to install callback functions that are called every time the processor takes a trap or returns from a trap (RETT instruction). The trap() function is called with one argument (*tt*) that contains the SPARC trap number. If tsim_trap() returns with 0, execution will continue. A non-zero return value will stop simulation with the program counter pointing to the instruction that will cause the trap. The rett() function is called when the program counter points to the RETT instruction but before the instruction is executed. The callbacks are removed by calling tsim_trap() with a NULL arguments.

```
int tsim_cov_get(int start, int end, char *ptr);
```

tsim_cov_get() will return the coverage data for the address range >= *start* and < *end.* The coverage data will be written to a char array pointed to by *ptr, starting at ptr[0]. One character per 32-bit word in the address range will be written. The user must assure that the char array is large enough to hold the coverage data.

```
int tsim_cov_set(int start, int end, char val);
```

tsim_cov_set() will fill the coverage data in the address range limited by start and end (see above for definition) with the value of *val*.

```
int tsim_ext_ins (int (*func) (struct ins_interface *r));
```

tsim_ext_ins() installs a handler for custom instructions. *func* is a pointer to an instruction emulation function as described in "Custom instruction emulation" on page 23. Calling tsim_ext_ins() with a NULL pointer will remove the handler.

6.3 I/O interface

The TSIM library uses the same I/O interface as the standalone simulator. Instead of loading a shared library containing the I/O module, the I/O module is linked with the main program. The I/O functions (and the main program) has the same access to the exported simulator interface (simif and ioif) as described in the loadable module interface. The TSIM library imports the I/O structure pointer, iosystem, which must be defined in the main program.

An example I/O module is provided in `tlib/<platform>/io.c`, which shows how to add a prom.

A second example I/O module is provided in simple_io.c This module provides a simpler interface to attach I/O functions. The following interface is provided:

```
void tsim_set_ioread (void (*cfunc)(int address, int *data, int *ws));
```

This function is used to pass a pointer to a user function which is to be called by TSIM when an I/O **read** access is made. The user function is called with the address of the access, a pointer to where the read data should be returned, and a pointer to a waitstate variable that should be set to the number of waitstates that the access took.

```
void tsim_set_iowrite (void (*cfunc)(int address, int *data, int *ws, int size));
```

This function is used to pass a pointer to a user function which is to be called by TSIM when an I/O **write** access is made. The user function is called with the address of the access, a pointer to the data to be written, a pointer to a waitstate variable that should be set to the number of waitstates that the access took, and the size of the access (0=byte, 1=half-word, 2=word, 3=double-word).

6.4 UART handling

By default, the library is using the same UART handling as the standalone simulator. This means that the UARTs can be connected to the console, or any unix device (pseudo-ttys, pipes, fifos). If the UARTs are to be handled by the user's I/O emulation routines, tsim_init() should be called with '-nouart', which will disable all internal UART emulation. Any access to the UART register by an application will then be routed to the I/O module read/write functions.

6.5 Linking a TLIB application

Three sample application are provided, one that uses the simplified I/O interface (app1.c), and two that uses the standard loadable module interface (app2 and app3). They are built by doing a 'make all' in the tlib directory. The win32 version of TSIM provides the library as a DLL.

6.6 Limitations

On Windows/cygwin hosts TSIM is not capable of reading UART A/B from the console, only writing is possible. If reading of UART A/B is necessary, the simulator should be started with -nouart, and emulation of the UARTs should be handled by the I/O module.

APPENDIX A: Installing HASP Device Driver

A.1 Installing HASP Device Driver

Two versions of the HASP USB hardware key are available, HASP4 M1 for node-locked licenses (blue key), and HASP4 Net for floating licenses (red key). Before use, a device driver for the key must be installed. The latest drivers can be found at www.ealaddin.com or www.gaisler.com. If a floating-license key is used, the HASP4 network license server also has to be installed and started. The necessary server installation documentation can be obtained from the distribution CD or from www.ealaddin.com.

A.1.1 On a Windows NT/2000/XP host

The HASP device driver is installed automatically when using the HASPUserSetup.exe located in hasp/windows/driver directory the TSIM CD. It automatically recognize the operating system in use and install the correct driver files at the required location.

Note: To install the HASP device driver under Windows NT/2000/XP, you need administrator privileges.

A.1.2 On a Linux host

The linux HASP driver consists of aksusbd daemon. It is contained in the hasp/linux/driver on the TSIM CD. The driver comes in form of RPM packages for Redhat and Suse linux distributions. The packages should be installed as follows:

Suse systems:

```
rpm -i aksusbd-suse-1.8.1-2.i386.rpm
```

Redhat systems:

```
rpm -i aksusbd-redhat-1.8.1-2.i386.rpm
```

The driver daemon can then be started by re-booting the most, or executing:

```
/etc/rc.d/init.d/aksusbd start
```

Note: All described action should be executed as root.

On other linux distributions, the driver daemon will have to be started manually. This can be done using the HDD_Linux_dinst.tar.gz, which also contains instruction on how to install and start the daemon.

A.2 Installing HASP4Net License Manager

The following steps are necessary to install HASP4 Net in a network:

• Install the appropriate HASP daemon and connect the HASP4 Net key.
• Install and start the HASP License Manager on the same machine.

• Customize the HASP License Manager and the HASP4 Net client, if necessary.

A.2.1 On a Windows NT/2000/XP host

The HASP License Manager for Windows NT/2000/XP is *nhsrvice32.exe*. Use the setup file *lmsetup.exe* to install it. It is recommended that you install the HASP License Manager as an NT service, so there is no need to log in to the station to provide the functionality.

1. Install the HASP device driver and connect the HASP4 Net key to a station.

2. Install the License Manager by running lmsetup.exe from the TSIM CD and following the instructions of the installation wizard. As installation type, select Service.

To activate the HASP License Manager application, start it from the Start menu or Windows Explorer. The HASP License Manager application is always active when any protocol is loaded and a HASP4 Net key is con ected. To deactivate it, select Exit from the main menu.

A.2.2 On Linux host

Before installing the LM you must install the HASP driver and *aksusbd* daemon as described above. The license manager can then be installed through rpm on Redhat and Suse systems, or started manually on other systems. The license manager is located in hasp/linux/server on the TSIM CD.

If you're using SuSE 8.x or 9.x:

```
rpm -i hasplm-suse-8.30-1.i386.rpm
```

RedHat 8.x or 9.x:

```
rpm -i hasplm-redhat-8.30-1.i386.rpm
```

It you're running a different Linux distribution, you must start the HASP LM manually:

```
./hasplm
```