TSIM2 Simulator User's Manual

ERC32/LEON2/LEON3/LEON4

TSIM2-UM
Version 2.0.39
June 2015

Kungsgatan 12
411 19 Gothenburg
Sweden

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

TSIM2 Simulator User's Manual

Table of Contents

1. Introduction

1.1. General

TSIM is a generic SPARC[1] architecture simulator capable of emulating ERC32- and LEON-based computer systems.

TSIM provides several unique features:

- Emulation of ERC32 and LEON2/3/4 processors
- Superior performance: up to 60 MIPS on high-end PC (Intel i7-2600K @3.4GHz)
- Accelerated processor standby mode, allowing faster-than-realtime simulation speeds
- Standalone operation or remote connection to GNU debugger (gdb)
- Also provided as library to be included in larger simulator frameworks
- 64-bit time for practically unlimited simulation periods
- Instruction trace buffer
- EDAC emulation (ERC32)
- MMU emulation (LEON2/3/4)
- SRAM emulation and functional emulation of SDRAM (with SRAM timing) (LEON2/3/4)
- Local scratch-pad RAM (LEON3/4)
- Loadable modules to include user-defined I/O devices
- Non-intrusive execution time profiling
- Code coverage monitoring
- Instruction trace buffer
- Stack backtrace with symbolic information
- Check-pointing capability to save and restore complete simulator state
- Unlimited number of breakpoints and watchpoints
- Pre-defined functional simulation modules for GR712, UT699, UT700 and AT697

1.2. Supported platforms and system requirements

TSIM supports the following platforms: Solaris 2.8, Linux, Linux-x64, Windows XP/7, and Windows XP/7 with Cygwin Unix emulation.

1.3. Obtaining TSIM

The primary site for TSIM is 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 support@gaisler.com.

[1]SPARC is a registered trademark of SPARC International

2. Installation

2.1. General

TSIM is distributed as a tar-file (e.g. tsim-erc32-2.0.39.tar.gz) with the following contents:

Table 2.1. TSIM content

doc	TSIM documentation
samples	Sample programs
iomod	Example I/O modules
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-erc32-2.0.39.tar.gz | tar xf -
```

2.2. License installation

TSIM is licensed using a HASP USB hardware key. Before use, a device driver for the key must be installed. The latest drivers can be found at http://sentinelcustomer.safenet-inc.com/sentineldownloads.

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. Starting 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*]

tsim-leon4 [*options*] [*input_files*]

The following command line options are supported by TSIM:

`-ahbm` *ahb_module*
: Use *ahb_module* as loadable AHB module rather than the default ahb.so (LEON only). If multiple -ahbm switches are specified up to 16 AHB modules can be loaded. The enviromental variable TSIM_MODULE_PATH can be set to a ':' separated (';' in WIN32) list of search paths.

`-ahbstatus`
: Adds AHB status register support.

`-asi1noallocate`
: Makes ASI 1 reads not allocate cache lines (LEON3/4 only).

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

`-banks` *ram_banks*
: Sets how many RAM banks the SRAM is divided on. Supported values are 1, 2 or 4. Default is 1. (LEON only).

`-bopt`
: Enables idle-loop optimisation (see Section 4.1.5).

`-bp`
: Enables emulation of LEON3/4 branch prediction

`-c` *file*
: Reads commands from *file* and executes them at startup.

`-cfg` *file*
: Reads extra configuration options from *file*.

`-cfgreg_and` *and_mask*, `-cfgreg_or` *or_mask*
: LEON2 only: Patch the Leon Configuration Register (0x80000024). The new value will be: *(reg & and_mask)* | *or_mask*.

`-covtrans`
: Enable MMU translations for the coverage system. Needed when MMU is enabled and not mapping 1-to-1.

`-cpm` *`cp_module`*
Use *`cp_module`* as loadable co-processor module file name (LEON). The enviromental variable TSIM_MODULE_PATH can be set to a ‘:’ separated (‘;’ in WIN32) list of search paths.

`-cas`
When running a VXWORKS SMP image the SPARCV9 “casa” instruction is used. The option `-cas` enables this instruction (LEON3/4 only).

`-dcsize` *`size`*
Defines the set-size (KiB) of the LEON data cache. Allowed values are powers of two in the range 1 - 64 for LEON2 and 1-256 for LEON3/4. Default is 4 KiB.

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

`-dlram` *`addr`* *`size`*
Allocates *`size`* KiB of local dcache scratchpad memory at address *`addr`*. (LEON3/4)

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

`-drepl` *`repl`*
Sets the replacement algorithm for the LEON data cache. Allowed values are `rnd` (default for LEON2) for random replacement, `lru` (default for LEON3/4) for the least-recently-used replacement algorithm and `lrr` for the least-recently-replaced replacement algorithm.

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

`-exc2b`
Issue 0x2b memory exception on memory write store error (LEON2 only)

`-ext` *`nr`*
Enable extended irq ctrl with extended irq number *`nr`* (LEON3/4 only)

`-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). The enviromental variable TSIM_MODULE_PATH can be set to a ‘:’ separated (‘;’ in WIN32) list of search paths.

`-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.

`-gdb`
Listen for GDB connection directly at start-up.

`-gdbuartfwd`
Forward output from first UART to GDB.

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

`-gr712rc`
Set parameters to emulate the GR712RC device. Must be used when using the GR712 AHB module.

`-grfpu`
Emulate the GRFPU floating point unit, rather then Meiko or GRFPU-lite (LEON only).

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

`-icsize` *size*
Defines the set-size (KiB) of the LEON instruction cache. Allowed values are powers of two in the range 1 - 64 for LEON2 and 1-256 for LEON3/4. Default is 4 KiB.

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

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

`-ilram` *addr size*
Allocates *size* bytes of local icache scratchpad memory at address *addr*. (LEON3/4)

`-ilsize` *size*
Sets the line size of the LEON instruction cache (in bytes). Allowed values are 16 or 32. Default is 16 for LEON2/3 and 32 for LEON4.

`-iom` *io_module*
Use *io_module* as loadable I/O module rather than the default io.so. The enviromental variable TSIM_MODULE_PATH can be set to a ‘:’ separated (‘;’ in WIN32) list of search paths.

`-irepl` *repl*
Sets the replacement algorithm for the LEON instruction cache. Allowed values are `rnd` (default for LEON2) for random replacement, `lru` (default for LEON3/4) for the least-recently-used replacement algorithm and `lrr` for the least-recently-replaced replacement algorithm.

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

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

`-l2wsize` *size*
Enable emulation of L2 cache (LEON4 only) with *size* KiB. The *size* must be binary aligned (e.g. 16, 32, 64 ...).

`-logfile` *filename*
Logs the console output to *filename*. If *filename* is preceded by ‘+’ output is append.

`-mfailok`
Do not fail on startup even if explicitely requested io/ahb modules fails to load.

`-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 (LEON only).

`-nb`
Do not break on error exceptions 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).

`-noeditline`
Disable use of editline for history and tab completion.

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

`-nothreads`
Disable threads support.

`-notimers`
Disable the LEON timer unit.

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

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

`-nrtimers` *`val`*
Adds support for more than 2 timers. Value *`val`* can be in the range of 2 - 8 (LEON3/4 only). Default: 2. See also the `-sametimerirq` and `-timerirqbase` *`number`* switches.

`-numbp` *`num`*
Sets the upper limit on number of possible breakpoints.

`-numwp` *`num`*
Sets the upper limit on number of possible watchpoints.

`-nwin` *`win`*
Defines the number of register windows in the processor. The default is 8. Only applicable to LEON3/4.

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

`-pr`
Enable profiling.

`-ram` *`ram_size`*
Sets the amount of simulated RAM (KiB). Default is 4096.

`-rest` *`file_name`*
Restore saved state from file_name.tss. See Section 3.8.

`-rom` *`rom_size`*
Sets the amount of simulated ROM (KiB). 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.

`-rtems` *`ver`*
Override autodetected RTEMS version for thread support. *`ver`* should be 46, 48, 48-edisoft or 410.

`-sametimerirq`
Force the irq number to be the same for all timers. Default: separate increasing irqs for each timer. (LEON3/4 only). See also the `-nrtimers` *`val`* and `-timerirqbase` *`number`* switches.

`-sdram` *`sdram_size`*
Sets the amount of simulated SDRAM (MiB). Default is 0. (LEON only)

`-sdbanks <1|2>`
Sets the SDRAM banks. This parameter is used to calculate the used SDRAM in conjunction with the mcfg2.sdramsize field. The actually used SDRAM at runtime is sdbanks*mcfg2.sdramsize. Default:1 (LEON only)

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

`-timer32`
Use 32 bit timers instead of 24 bit. (LEON2 only)

`-timerirqbase` *number*
Set the irq number of the first timer to interrupt number *number* (LEON3/4 only). Default: 8. See also the `-nrtimers` *val* and `-sametimerirq` switches.

`-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.

`-uartX` *device*
Here X, can be 1 or 2. 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. On windows use //./com1, //./com2 etc. to access the serial ports. The serial port settings can be adjusted by doubleclicking the "Ports (COM and LPT)" entry in controlpanel->system->hardware->devicemanager. Use the "Port Setting" tab in the dialogue that pops up.

`-ut699`
Set parameters to emulate the UT699 device. Must be used when using the UT699 AHB module.

`-ut699e`
Set parameters to emulate the UT699E device. Must be used when using the UT699E AHB module.

`-ut700`
Set parameters to emulate the UT700 device. Must be used when using the UT700 AHB module.

`-wdfreq` *freq*
Specify the frequency of the watchdog clock. (ERC32 only)

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.

Command line options can also be specified in the file .tsimcfg in the home directory. This file will be read at startup and the contents will be appended to the command line.

3.3. Standalone mode commands

If the file .tsimrc exists in the home directory, it will be used as a batch file and the commands in it will be executed at startup.

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** [*address*] [*count/time*] command for how to specify count or time.

coverage <**enable** | **disable** | **save** [*file_name*] | **clear** | **print** *address [len]*>
Code coverage control. Coverage can be enabled, disabled, cleared, saved to a file or printed to the console.

dump *file address length*
Dumps memory content to file *file*, in whole aligned words. The *address* argument can be a symbol.

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. See the **inst** [*length*] command for displaying only a part of the instruction 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.

inst [*length*]

Display the latest *length* (default 30) instructions in the instruction trace buffer. See the **hist** [*length*] command for how to enable the instruction trace buffer.

leon

Display LEON peripherals registers.

load *files*

Load *files* into simulator memory.

l2cache

Display contents of L2 cache. (LEON4 only)

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 [*addr*] [*count/time*]

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

save *file*

Save simulator state to *file*.

step
Execute and disassemble one instruction. See also **trace** [*num*] .

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

trace [*num*]
Executes and disassembles unbounded or up to *num* instruction(s), until finished, hitting a breakpoint/watchpoint or some other reason to stop.

version
Prints the TSIM version and build date.

walk *address* [**iswrite**|**isid**|**issu**]*
If the MMU is enabled printout a table walk for the given address. The flags iswrite, isid and issu are specifying the context: iswrite for a write access (default read), isid for a icache access (default dcache), issu for a supervisor access (default user).

watch *address*
Adds a watchpoint at *address*.

wmem, wmemh, wmemb *address value*
Write a word, half-word or byte directly to simulated memory.

xwmem *asi address value*
Write a word to simulated memory using ASI=*asi*. Applicable to LEON3/4.

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> break 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 bootproms 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 a good 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 clear	reset coverage data

The coverage data for each 32-bit word of memory consists of a 5-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. Bit3 and bit4 indicates the presence of a branch instruction; if bit3 is set then the branch was taken while bit4 is set if the branch was not taken.

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: Only the internally emulated memory (PROM, SRAM and SDRAM) are subject for code coverage. Any memory emulated in the user's I/O module must be handled by a user-defined coverage function.

The address ranges that are monitored are based on TSIM's startup parameters. For instance, the range corresponding to the SDRAM for LEON will begin at address 0x40000000 if TSIM was started with -nosram or -ram 0, or will begin at 0x60000000 otherwise. Reconfiguration of the memory controller during execution will not be taken into account for monitored address ranges. Coverage information on memory reads will be recorded even for cache hits.

NOTE on MMU and coverage: The monitored ranges are based on physical addresses. The TSIM coverage system does no address translations by default, for performance reasons. To get proper physical address coverage when the MMU is is enabled and not mapping 1-to-1, use the `-covtrans` option. There is no support for getting virtual address coverage.

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 Section 6.2). Enabling and disabling the coverage functionality from the function interface should be done using `tsim_cmd()`.

Example scripts for annotating C code using saved coverage information from TSIM can be found in the coverage sub-directory.

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

NOTE: AT697, UT699, UT700 and GR712 simulation modules do not support check-pointing.

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

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.

3.12. Thread support

TSIM has thread support for the RTEMS operating system. Additional OS support will be added to future versions. The GDB interface of TSIM is also thread aware and the related GDB commands are described later.

3.12.1. TSIM thread commands

thread info - lists all known threads. The currently running thread is marked with an asterisk. (The wide example output below has been split into two parts.)

```
tsim> thread info

  Name | Type     | Id         | Prio | Time (h:m:s)  | Entry point              ...
-------------------------------------------------------------------------------- ...
  Int. | internal | 0x09010001 |  255 |   5:30.682722 | bsp_idle_thread          ...
-------------------------------------------------------------------------------- ...
  UI1  | classic  | 0x0a010001 |  100 |      0.041217 | system_init              ...
-------------------------------------------------------------------------------- ...
  ntwk | classic  | 0x0a010002 |  100 |      0.251199 | soconnsleep              ...
-------------------------------------------------------------------------------- ...
  ETH0 | classic  | 0x0a010003 |  100 |      0.000161 | soconnsleep              ...
-------------------------------------------------------------------------------- ...
* TA1  | classic  | 0x0a010004 |    1 |      0.034739 | prep_timer               ...
-------------------------------------------------------------------------------- ...
  TA2  | classic  | 0x0a010005 |    1 |      0.025740 | prep_timer               ...
-------------------------------------------------------------------------------- ...
  TA3  | classic  | 0x0a010006 |    1 |      0.021357 | prep_timer               ...
-------------------------------------------------------------------------------- ...
  TTCP | classic  | 0x0a010007 |  100 |      0.002914 | rtems_ttcp_main          ...
-------------------------------------------------------------------------------- ...

... | PC                                             | State
... -------------------------------------------------------
... | 0x40044bec _Thread_Dispatch + 0xd8             | READY
... -------------------------------------------------------
... | 0x40044bec _Thread_Dispatch + 0xd8             | SUSP
... -------------------------------------------------------
... | 0x40044bec _Thread_Dispatch + 0xd8             | READY
... -------------------------------------------------------
... | 0x40044bec _Thread_Dispatch + 0xd8             | Wevnt
... -------------------------------------------------------
... | 0x40006a28 printf + 0x4                        | READY
... -------------------------------------------------------
... | 0x40044bec _Thread_Dispatch + 0xd8             | DELAY
... -------------------------------------------------------
... | 0x40044bec _Thread_Dispatch + 0xd8             | DELAY
... -------------------------------------------------------
... | 0x40044bec _Thread_Dispatch + 0xd8             | Wevnt
... -------------------------------------------------------
```

thread bt *id* prints a backtrace of a thread.

```
tsim> thread bt 0x0a010007

    %%pc
#0   0x40044bec   _Thread_Dispatch + 0xd8
#1   0x400418f8   rtems_event_receive + 0x74
#2   0x40031eb4   rtems_bsdnet_event_receive + 0x18
```

```
#3   0x40032050   soconnsleep + 0x50
#4   0x40033d48   accept + 0x60
#5   0x4000366c   rtems_ttcp_main + 0xda0
```

A backtrace of the current thread (equivalent to normal bt command):

```
tsim> thread bt
    %pc          %sp
#0   0x40006a28   0x4008d7d0   printf + 0x0
#1   0x40001c04   0x4008d838   Test_task + 0x178
#2   0x4005c88c   0x4008d8d0   _Thread_Handler + 0xfc
#3   0x4005c78c   0x4008d930   _Thread_Evaluate_mode + 0x58
```

3.12.2. GDB thread commands

TSIM needs the symbolic information of the image that is being debugged to be able to check for thread information. Therefore the symbols needs to be read from the image using the **sym** command before issuing the **gdb** command. When a program running in GDB stops TSIM reports which thread it is in. The command **info threads** can be used in GDB to list all known threads.

```
Program received signal SIGINT, Interrupt.
[Switching to Thread 167837703]

0x40001b5c in console_outbyte_polled (port=0, ch=113 ’q’) at ../../../../../../../../../rtems-
4.6.5/c/src/lib/libbsp/sparc/leon3/console/debugputs.c:38
38            while ( (LEON3_Console_Uart[LEON3_Cpu_Index+port]->status &amp; LEON_REG_UART_STATUS_THE)
== 0 );

(gdb) info threads

  8 Thread 167837702 (FTPD Wevnt) 0x4002f760 in _Thread_Dispatch () at ../../../../../../rtems-
4.6.5/cpukit/score/src/threaddispatch.c:109
  7 Thread 167837701 (FTPa Wevnt) 0x4002f760 in _Thread_Dispatch () at ../../../../../../rtems-
4.6.5/cpukit/score/src/threaddispatch.c:109
  6 Thread 167837700 (DCtx Wevnt) 0x4002f760 in _Thread_Dispatch () at ../../../../../../rtems-
4.6.5/cpukit/score/src/threaddispatch.c:109
  5 Thread 167837699 (DCrx Wevnt) 0x4002f760 in _Thread_Dispatch () at ../../../../../../rtems-
4.6.5/cpukit/score/src/threaddispatch.c:109
  4 Thread 167837698 (ntwk ready) 0x4002f760 in _Thread_Dispatch () at ../../../../../../rtems-
4.6.5/cpukit/score/src/threaddispatch.c:109
  3 Thread 167837697 (UI1 ready) 0x4002f760 in _Thread_Dispatch () at ../../../../../../rtems-
4.6.5/cpukit/score/src/threaddispatch.c:109
  2 Thread 151060481 (Int. ready) 0x4002f760 in _Thread_Dispatch () at ../../../../../../rtems-
4.6.5/cpukit/score/src/threaddispatch.c:109
* 1 Thread 167837703 (HTPD ready ) 0x40001b5c in console_outbyte_polled (port=0, ch=113 ’q’)
   at ../../../../../../../../../rtems-4.6.5/c/src/lib/libbsp/sparc/leon3/console/debugputs.c:38
```

Using the **thread** command a specified thread can be selected:

```
(gdb) thread 8

[Switching to thread 8 (Thread 167837702)]#0 0x4002f760 in _Thread_Dispatch () at ../../../../
../../rtems-4.6.5/cpukit/score/src/threaddispatch.c:109
109         _Context_Switch( &amp;executing->Registers, &amp;heir->Registers );
```

Then a backtrace of the selected thread can be printed using the **bt** command:

```
(gdb) bt

#0   0x4002f760 in _Thread_Dispatch () at ../../../../../../rtems-4.6.5/cpukit/score/src/thread-
dispatch.c:109
#1   0x40013ee0 in rtems_event_receive (event_in=33554432, option_set=0, ticks=0,
event_out=0x43fecc14)
      at ../../../../leon3/lib/include/rtems/score/thread.inl:205
#2   0x4002782c in rtems_bsdnet_event_receive (event_in=33554432, option_set=2, ticks=0,
event_out=0x43fecc14)
      at ../../../../../../rtems-4.6.5/cpukit/libnetworking/rtems/rtems_glue.c:641
#3   0x40027548 in soconnsleep (so=0x43f0cd70) at ../../../../../../rtems-4.6.5/cpukit/libnetwork-
ing/rtems/rtems_glue.c:465
#4   0x40029118 in accept (s=3, name=0x43feccf0, namelen=0x43feccec) at ../../../../../../rtems-
4.6.5/cpukit/libnetworking/rtems/rtems_syscall.c:215
#5   0x40004028 in daemon () at ../../../../../../rtems-4.6.5/c/src/libnetworking/rtems_servers/
ftpd.c:1925
#6   0x40053388 in _Thread_Handler () at ../../../../../../rtems-4.6.5/cpukit/score/src/threadhan-
dler.c:123
```

```
#7   0x40053270 in __res_mkquery (op=0, dname=0x0, class=0, type=0, data=0x0, datalen=0,
newrr_in=0x0, buf=0x0, buflen=0)
      at ../../../../../../../rtems-4.6.5/cpukit/libnetworking/libc/res_mkquery.c:199
```

It is possible to use the **frame** command to select a stack frame of interest and examine the registers using the **info registers** command. Note that the **info registers** command only can see the following registers for an inactive task: g0-g7, l0-l7, i0-i7, o0-o7, pc and psr. The other registers will be displayed as 0:

```
(gdb) frame 5

#5  0x40004028 in daemon () at ../../../../../../rtems-4.6.5/c/src/libnetworking/rtems_servers/
ftpd.c:1925
1925       ss = accept(s, (struct sockaddr *)&addr, &addrLen);

(gdb) info reg

g0            0x0      0
g1            0x0      0
g2            0xffffffff       -1
g3            0x0      0
g4            0x0      0
g5            0x0      0
g6            0x0      0
g7            0x0      0
o0            0x3      3
o1            0x43feccf0       1140772080
o2            0x43feccec       1140772076
o3            0x0      0
o4            0xf34000e4       -213909276
o5            0x4007cc00       1074252800
sp            0x43fecc88       0x43fecc88
o7            0x40004020       1073758240
l0            0x4007ce88       1074253448
l1            0x4007ce88       1074253448
l2            0x400048fc       1073760508
l3            0x43feccf0       1140772080
l4            0x3      3
l5            0x1      1
l6            0x0      0
l7            0x0      0
i0            0x0      0
i1            0x40003f94       1073758100
i2            0x0      0
i3            0x43ffafc8       1140830152
i4            0x0      0
i5            0x4007cd40       1074253120
fp            0x43fecd08       0x43fecd08
i7            0x40053380       1074082688
y             0x0      0
psr           0xf34000e0       -213909280
wim           0x0      0
tbr           0x0      0
pc            0x40004028       0x40004028 <daemon+148>
npc           0x4000402c       0x4000402c <daemon+152>
fsr           0x0      0
csr           0x0      0
```

It is not supported to set thread specific breakpoints. All breakpoints are global and stops the execution of all threads. It is not possible to change the value of registers other than those of the current thread.

4. Emulation characteristics

4.1. Common behaviour

4.1.1. Timing

The simulator time is maintained and incremented according the IU and FPU instruction timing. The parallel execution between the IU and FPU is modelled, as well as stalls due to operand dependencies. Instruction timing has been modelled after the real devices. Integer instructions have a higher accuracy than floating-point instructions due to the somewhat unpredictable operand-dependent timing of the ERC32 and LEON MEIKO FPU. Typical usage patterns have higher accuracy than atypical ones, e.g. having vs. not having caches enabled on LEON systems. Tracing using the **inst** or **hist** 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 and will not always be identical to the actual ERC32/LEON hardware. The simulator implements (to some extent) data-dependant execution timing as in the real MEKIO FPU (ERC32/LEON2). For LEON3/4, the `-grfpu` switch will enable emulation of the GRFPU instruction timing.

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.

4.1.6. 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 Section 6.2). 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.

4.1.7. Chip-specific errata

Incorrect behavior described in errata documents for specific devices are not emulated by TSIM in general.

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 table outlines the implemented MEC registers:

Table 4.1. Implemented MEC registers

Register	Address	Status
MEC control register	0x01f80000	implemented
Software reset register	0x01f80004	implemented
Power-down register	0x01f80008	implemented

Register	Address	Status
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 KiB and 32 MiB, the ROM size between 128 KiB and 4 MiB. 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 KiB of ROM and 256 KiB 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 performance when there are inserted errors in the memory. To obtain maximum simulation performance, any diagnostic software should remove all inserted errors after 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 extended 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 behavior of the LEON2 VHDL model. The (optional) MMU can be emulated using the `-mmu` switch.

4.3.2. Cache memories

TSIM/LEON2 can emulate any permissible cache configuration using the `-icsize`, `-ilsize`, `-dcsize` and `-dlsize` options. Allowed sizes are 1 - 64 KiB with 16 - 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. 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 functionally modelled SDRAM (with SRAM timing) 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.

4.3.8. DSU and hardware breakpoints

The LEON debug support unit (DSU) and the hardware watchpoints (%asr24 - %asr31) are not emulated.

4.4. LEON3 specific emulation

4.4.1. General

The LEON3 version of TSIM emulates the behavior 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 32-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 configured with 2 - 32 register windows using the `-nwin` switch. The MMU can be emulated using the `-mmu` switch. Local scratch pad RAM can be added with the `-ilram` and `-dlram` switches.

4.4.3. Cache memories

The evaluation version of TSIM/LEON3 implements 2*4 KiB 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 - 256 KiB with 16 - 32 bytes/line. The characteristics of the LEON multi-way 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 functionally modelled SDRAM (with SRAM timing) 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 LEON3-FT is not implemented.

Options regarding memory characteristics are not available in the evaluation version of TSIM/LEON3.

4.4.8. CASA instruction

The SPARCV9 “casa” command is implemented if the `-cas` switch is given. The “casa” instruction is used in VXWORKS SMP multiprocessing to synchronize using a lock free protocol.

4.4.9. 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.

4.4.10. DSU and hardware breakpoints

The LEON debug support unit (DSU) and the hardware watchpoints (%asr24 - %asr31) are not emulated.

4.4.11. AHB status registers

When using `-ahbstatus` or a chip option for a chip that has AHB status registers, AHB status registers are enabled. As TSIM/LEON3 does not emulate FT, the CE bit will never be set. Furthermore, the HMASTER field is set to 0 when the CPU caused the error and 1 when any other master caused the error.

4.5. LEON4 specific emulation

4.5.1. General

The LEON4 version of TSIM emulates the behavior of the LEON4MP template VHDL model distributed in the GRLIB-1.0.x IP library. The system includes the following modules: LEON4 processor, APB bridge, IRQMP interrupt controller, LEON2 memory controller, L2 cache, GPTIMER timer unit with two 32-bit timers, two APBUART uarts. The AHB/APB plug&play information is provided at address 0xFFFFF000 - 0xFFFFFFFF (AHB) and 0x800FF000 - 0x800FFFFF (APB).

4.5.2. Processor

The instruction timing of the emulated LEON4 processor is modelled after LEON4 VHDL model in GRLIB IP library. The processor can be configured with 2 - 32 register windows using the `-nwin` switch. The MMU can be emulated using the `-mmu` switch. Local scratch pad RAM can be added with the `-ilram` and `-dlram` switches.

4.5.3. L1 Cache memories

TSIM/LEON4 can emulate any permissible cache configuration using the `-icsize`, `-ilsize`, `-dcsize` and `-dlsize` options. Allowed sizes are 1 - 256 KiB with 16 - 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.

4.5.4. L2 Cache memory

The LEON4 L2 cache is emulated, and placed between the memory controller and AHB bus. Both the PROM and SRAM/SDRAM areas are cached in the L2. The size of the L2 cache is default 64 KiB, but can be configured to any (binary aligned) size using the `-l2wsize` switch at start-up. Setting the size to 0 will disable the L2 cache. The L2 cache is implemented with one way and 32 bytes/line. The contents of the L2 cache can be displayed with the **l2cache** command.

4.5.5. Power-down mode

The LEON4 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.

4.5.6. LEON4 peripherals registers

The LEON4 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’).

4.5.7. 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.

4.5.8. Memory emulation

The LEON2 memory controller is emulated in the LEON4 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 functionally modelled SDRAM (with SRAM timing) 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 LEON4-FT is not implemented.

4.5.9. CASA instruction

The SPARCV9 “casa” command is implemented if the `-cas` switch is given. The “casa” instruction is used in VXWORKS SMP multiprocessing to synchronize using a lock free protocol.

4.5.10. SPARC V8 MUL/DIV/MAC instructions

TSIM/LEON4 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.

4.5.11. GRFPU emulation

By default, TSIM-LEON4 emulates the GRFPU-Lite FPU. If the simulator is started with `-grfpu`, the fully pipelined GRFPU is emulated. Due to the complexity of the GRFPU, the instruction timing is approximated and might show some discrepancies compared to the real hardware.

4.5.12. DSU and hardware breakpoints

The LEON debug support unit (DSU) and the hardware watchpoints (%asr24 - %asr31) are not emulated.

4.5.13. AHB status registers

When using `-ahbstatus`, AHB status registers are enabled. As TSIM/LEON4 does not emulate FT, the CE bit will never be set. Furthermore, the HMASTER field is set to 0 when the CPU caused the error and 1 when any other master caused the error.

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 straightforward: 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 startup, 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. The enviromental variable TSIM_MODULE_PATH can be set to a ':' separated (';' in WIN32) list of search paths.

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 */
   char *args;                       /* concaterated argv */
   void (*stop_event_arg)(void (*cfunc)(),int arg,int op);

   /* Restorable events */
   unsigned short (*reg_revent)(void (*cfunc) (unsigned long arg));
   unsigned short (*reg_revent_prearg)(void (*cfunc) (unsigned long arg),
                                      unsigned long arg);
   int (*revent)(unsigned short index, unsigned long arg, uint64 offset);
   int (*revent_prearg)(unsigned short index, uint64 offset);
   void (*stop_revent)(unsigned short index);
};
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.

NOTE: For save and restore support, restorable events should be used instead.

`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.

`char *args;`

Arguments supplied when starting tsim. The arguments are concatenated as a single string.

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

Similar to `stop_event()` but differentiates between 2 events with same `cfunc` but with different *`arg`* given when inserted into the event queue via `event()`. Used when simulating multiple instances of an entity. Parameter *`op`* should be 1 to enable the *`arg`* check.

`unsigned short (*reg_revent)(void (*cfunc) (unsigned long arg));`

Registers a restorable event that will use `cfunc` as callback. The returned index should be used when calling `revent()`. The event argument is supplied when calling `revent()`. The call to `reg_revent()` should be done once at I/O or AHB module initialization.

`unsigned short (*reg_revent_prearg)(void (*cfunc) (unsigned long arg), unsigned long arg);`

Registers a restorable event that will use `cfunc` as callback and *`arg`* as argument. This can be used to register an argument that is a pointer to a data structure. The returned index should be used when calling `revent_prearg()`. The call to `reg_revent_prearg()` should be done once at I/O or AHB module initialization.

`int (*revent)(unsigned short index, unsigned long arg, uint64 offset);`

This inserts an event registered by `reg_revent()` into the event queue with the registered `cfunc` for the given *`index`*. Multiple events with the same *`index`* can be in the event queue at the same time. The *`arg`* and *`offset`* arguments are the same as for the `event()` function.

NOTE: See the description of `event()` for limitations on number of events and from which contexts events can be added.

`int (*revent_prearg)(unsigned short index, uint64 offset);`

This inserts an event registered by `reg_revent_prearg()` into the event queue with the registered `cfunc` and *`arg`* for the given *`index`*. Multiple events with the same *`index`* can be in the event queue at the same time. The *`offset`* argument is the same as for the `event()` function.

NOTE: See the description of `event()` for limitations on number of events and from which contexts events can be added.

`void (*stop_revent)(unsigned short index);`

This removes all events from the event queue that has been entered by `revent()` or `revent_prearg()` using the given *`index`*.

NOTE: The `stop_revent()` function may *not* be called from a signal handler installed by the I/O module.

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);
   int (*dma_write_sub)(uint32 addr, uint32 *data, int sz);
};
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.

`int (*dma_write_sub)(uint32 addr, uint32 *data, int sz);`

Performs DMA transactions to/from the emulated processor memory on the AMBA AHB bus. Available for LEON only. On bus error, 1 is returned, otherwise 0. Write size is indicated by *`sz`* as follows: 0 = byte, 1 = half-word, 2 = word, 3 = double-word.

5.1.3. Structure to be provided by I/O device

```
struct io_subsystem {
   void (*io_init)(struct sim_interface sif, struct io_interface iif); /* 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)(struct sim_interface sif, struct io_interface iif);`
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 %lt; 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 */
};
```

The same method is also used in the AHB and FPU/CP modules.

5.2. LEON AHB emulation interface

In addition to the above described I/O modules, TSIM also allows emulation of the LEON2/3/4 processor core with a completely user-defined memory and I/O architecture. This is in other words not applicable to ERC32. 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. The enviromental variable TSIM_MODULE_PATH can be set to a ‘:’ separated (‘;’ in WIN32) list of search paths.

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)();
   void (*set_irq_level)(int level, int set);
   void (*set_irq)(uint32 irq, uint32 level); /* generate external interrupt */
};
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 can be used to invalidate data cache lines (regardless of whether data cache snooping is enabled or not). The tags to the given address will be checked, and if a match is detected the corresponding cache lines will be flushed (i.e. the tag will be cleared). If an MMU is present and is enabled the argument should be a virtual address. See also the snoop function in `struct ahb_interface`.

`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 LEON3/4 accesses the cache controller through a separate ASI load/store instruction.

`void (*power_down)();`
The LEON processor enters power down-mode when called.

`void (*set_irq_level)(int level, int set);`
Callback `set_irq_level` can be used to emulate level triggered interrupts. Parameter `set` should be 1 to activate the interrupt level specified in parameter `level` or 0 to reset it. The interrupt level will remain active after it is set until it is reset again. Multiple calls can be made with different `level` parameters in which case the highest level is used.

`void (*set_irq)(uint32 irq, uint32 level);`
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.

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; /* No longer used */
};

struct pp_amba {
   int is_apb;
   unsigned int vendor, device, version, irq;
   struct {
     unsigned int addr, p, c, mask, type;
   } bars[4];
};

struct ahb_subsystem {
   void (*init)(struct proc_interface procif);/* 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 */
   int (*plugandplay)(struct pp_amba **); /* LEON3/4: get plug & play information */
   void (*intpend)(unsigned int pend); /* LEON3/4 only: interrupt pending change */
   int   meminit; /* tell tsim weather to initialize mem */
   struct sim_interface *simif; /* initialized by tsim */
   unsigned char *(*get_mem_ptr_ws)(); /* initialized if meminit was set */
   void (*snoop) (unsigned int addr);    /* initialized with cache snoop routine */
   struct io_interface *io; /* initialized by tsim */
   void (*dprint)(char *p); /* initialized by tsim, prints out a debug string */
   struct proc_interface *proc; /* initialized by tsim, access to proc_interface */
   int (*cacheable)(uint32 addr, uint32 size); /* Cacheability of area */
   int (*lprintf)(const char *format, ...); /* initialized by tsim */
   int (*vlprintf)(const char *format, va_list ap); /* initialized by tsim */
};

extern struct ahb_subsystem *ahbsystem;      /* imported AHB emulation functions */
```

The elements of the structure have the following meanings:

`void (*init)(struct proc_interface procif);`
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.

`void int (*read)(struct ahb_access *ahbacc);`
Processor AHB read. The processor always reads one or more 32-bit words from the AHB bus. The following fields of *ahbacc* is used. The ahbacc.addr field contains the read address of the first word to read. The ahbacc.data field points to a buffer that the module can fill in. The module can also change the pointer to point to a different buffer. The ahbacc.ws field should be set by the module to the number of cycles for the complete access. The ahbacc.rnum field contains the number of words to be read. The function should return 0 for a successful access, 1 for failed access and -1 for an area not handled by the module. The ahbacc.wsize field is not used during read cycles. The ahbacc.cache field is no longer in use (use `struct ahb_subsystem.cacheable` instead).

`int (*write)(struct ahb_access *ahbacc);`
Processor AHB write. The processor can write 1, 2, 4 or 8 bytes per access. The following fields of *ahbacc* is used. The ahbacc.addr field contains the address of the write. The ahbacc.data field points to the data to write; either one word for byte, half word or word writes or two words for double-word writes. The ahbacc.wsize field defines write size as follows: 0 = byte, 1 = half-word, 2 = word, 3 = double-word. The function should return 0 for a successful access, 1 for failed access and -1 for an area not handled by the module. The ahbacc.rnum field is not used during write cycles. The ahbacc.cache field is no longer in use (use `struct ahb_subsystem.cacheable` instead).

`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.

`int (*plugandplay)(struct pp_amba **p);`

Leon3/4 only: `The plugandplay()` function is called at startup. `optioplugandplay()` should return in p a static pointer to an array with elements of type `struct pp_amba` and return the number of entries in the array. The callback `plugandplay()` is used to add entries in the AHB and APB configuration space. Each `struct pp_amba` entry specifies an entry: If is_apb is set to 1 the entry will appear in the APB configuration space and only member bars[0] will be used. If is_apb is 0 then the entry will appear in the AHB slave configuration space and bars[0-3] will be used. If is_apb is 2 then the entry will appear in the AHB master configuration space and bars[0-3] will be used. The members of the struct resemble the fields in a configuration space entries. The entry is mapped to the first free slot.

`void (*intpend)(unsigned int pend);`

Leon3/4 only: The `intpend()` function is called when the set of pending interrupts changes. The *`pend`* argument is a bitmask with the bits of pending interrupts set to 1.

`int meminit;`

If all loaded AHB modules sets meminit to 1, TSIM will initialize and emulate initialize and emulate SRAM/SDRAM/PROM memory. Thus, the AHB module should initialize `meminit` with 1 if TSIM (or another AHB module) should handle memory simulation. Calls to read and write should return -1 (undecoded area) for the memory regions in which case TSIM (or possibly some other AHB module) will handle them. If `meminit` is set to 0 the AHB module itself should emulate the memory address regions.

`struct sim_interface *simif;`

Entry `simif` is initialized by tsim with the global `struct sim_interface` structure.

`unsigned char *(*get_mem_ptr_ws) (unsigned int addr, int size, int *wws, int *rws)`

If `meminit` was set to 1 tsim will initialize `get_mem_ptr_ws` with a callback that can be used to query a pointer to the host memory emulating the LEON's memory, along with waitstate information. Note that the host memory pointer returned is in the hosts byte order (normally little endian on a PC). The *`size`* parameter should be the length of the access (1 for byte, 2 for short, 4 for word and 8 for double word access). The *`wws`* and *`rws`* parameters will return the calculated write and read waitstates for a possible access. See also `snoop` below for responsibilities when DMA writes are done via pointers from this function.

`void (*snoop) (unsigned int addr)`

The callback `snoop` is initialized by tsim. If data cache snooping is enabled (and functioning, i.e. not ut699) it flushes (i.e. invalidates) data cache lines corresponding to physical address *`addr`* (on LEON3/4 even when MMU is enabled). If the AHB module is doing DMA writes directly to memory pointers, it is the responsibility of the AHB module to call this for all changed words for snooping to work correctly.

`struct io_interface *io;`

Initialized with the I/O interface structure pointer.

`void (*dprint)(char *);`

Initialized by tsim with a callback pointer to the debug output function. Output ends up in log, when logging is enabled and gets forwarded to gdb when running TSIM via gdb. See `lprintf` and `vlprintf` for the formatted couterparts.

`struct proc_interface *proc;`

Initialized with the procif structure pointer.

`int (*cacheable)(uint32 addr, uint32 size)`

The `cacheable` callback is initialized by the module to NULL or a function returning cacheability for a memory area. The function should return 1 if the range [addr,addr+size) is cacheable, 0 if it is uncacheable or -1 if the memory area it is not handled by the module. If all modules return -1 and/or lack the `cacheable` callback, the area will be considered cacheable for memory areas [0x00000000,0x20000000) and [0x40000000-0x80000000) and non-cacheable for all other areas. NOTE: For any (correspondingly aligned) area as large as the largest data cache or instruction cache line size in the system, the `cacheable` callback may not return different results for different words in the area.

`int (*lprintf)(const char *format, ...)`
Initialized by TSIM with a function for formatted loggable debug output. The function interface works like for printf.

`int (*vlprintf)(const char *format, va_list ap)`
Initialized by TSIM with a function for formatted loggable debug output. The function interface works like for vprintf.

5.2.3. Big versus little endianess

SPARC 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)();                                   /* calledon 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 */
   int set_fsr(uint32 fsr);                              /* initialized by tsim */
};
extern struct cp_interface *cp;   /* imported co-processor emulation functions */
```

5.3.2. Structure elements

`void (*cp_init)(struct sim_interface sif, struct io_interface iif);`
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_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.

`int (*set_fsr)(char * cmd); /* initialized by tsim */`

This callback is initialized by tsim and can be called to set the FPU status register.

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. The enviromental variable TSIM_MODULE_PATH can be set to a ':' separated (';' in WIN32) list of search paths. 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. See 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 co-processor.

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

The professional version of TSIM is also available as a library, allowing the simulator to be integrated in a larger simulation frame-work. 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 just before each executed instruction, *when the history is enabled*. At this point the instruction to be executed can be seen as the last entry in the history. 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 Section 4.1.6. Calling `tsim_ext_ins()` with a NULL pointer will remove the handler.

`int tsim_lastbp (int *addr)`

When simulation stopped due to breakpoint or watchpoint hit (SIGTRAP), this function will return the address of the break/watchpoint in **addr*. The function return value indicates the break cause; 0 = breakpoint, 1 = watchpoint.

6.3. AHB modules

AHB modules can be loaded by adding the "`-ahbm <name>`" switch to the `tsim_init()` string when starting. See Section 5.2 for further information.

6.4. 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).

6.5. 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.

6.6. 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, for all other platform a static library is provided (.a). Support for dynamic libraries on Linux or Solaris is not available.

6.7. 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.

7. Cobham UT699/UT699e AHB module

7.1. Overview

This chapter describes the UT699 loadable AHB module for the TSIM2 simulator. The AHB module provides simulation models for the Ethernet, SpaceWire, PCI, GPIO and CAN cores in the UT699 processor. For more information about this chip see the Cobham UT699 user manual.

The interfaces are modelled at packet/transaction/message level and provides an easy way to connect the simulated UT699 to a larger simulation framework.

The following files are delivered with the UT699 TSIM module:

Table 7.1. Files delivered with the UT699 TSIM module

File	Description
ut699/linux/ut699.so	UT699 AHB module for Linux
ut699/linux/ut699e.so	UT699e AHB module for Linux
ut699/win32/ut699.dll	UT699 AHB module for Windows
ut699/win32/ut699e.dll	UT699e AHB module for Windows
out699/examples/input	The input directory contains two examples of PCI user modules
ut699/examples/input/README.txt	Description of the user module examples
ut699/examples/input/pci.c	PCI user module example that makes UT699 PCI initiator accesses
ut699/examples/input/pci_target.c	PCI user module example that makes UT699 PCI target accesses
ut699/examples/input/gpio.c	GPIO user module example
ut699/examples/input/ut699inputprovider.h	Interface between the UT699 module and the user defined PCI module
ut699/examples/input/pci_input.h	UT699 PCI input provider definitions
ut699/examples/input/input.h	Generic input provider definitions
ut699/examples/input/tsim.h	TSIM interface definitions
ut699/examples/input/end.h	Defines the endian of the local machine
ut699/examples/test	The test directory contains tests that can be executed in TSIM
ut699/examples/test/README.txt	Description of the tests
ut699/examples/test/Makefile	Makefile for building the tests
ut699/examples/test/cansend.c	CAN transmission test
ut699/examples/test/canrec.c	CAN reception test
ut699/examples/test/pci.c	PCI interface test
ut699/examples/test/pcitest.h	Header file for PCI test

7.2. Loading the module

The module is loaded using the TSIM2 option `-ahbm`. All core specific options described in the following sections need to be surrounded by the options `-designinput` and `-designinputend`, e.g:

On Linux:

```
tsim-leon3 -ut699 -ahbm ./ut699/linux/ut699.so
          -designinput ./ut699/examples/input/pci.so -designinputend
```

On Windows:

```
tsim-leon3 -ut699 -ahbm ut699/win32/ut699.dll
        -designinput ./ut699/examples/input/pci.dll -designinputend
```

The option `-ut699` needs to be given to TSIM to enable the UT699 processor configuration. Note that when `-ut699` is given, snooping will be set as non-functional.

7.3. UT699e

To enable the UT699e version of the UT699 replace *ut699.[so|dll]* with *ut699e.[so|dll]* and option `-ut699` with `-ut699e`. This:

- Enables snooping opposed to the non-functional snooping of the `-ut699`
- Sets UT699e build-id
- Changes MMU status/ctrl registers layout
- Contains GRSPW2 cores instead of GRSPW cores (the TSIM command, flag and packet interface is the same however)

7.4. Debugging

To enable printout of debug information the `-ut699_dbgon` *flag* switch can be used. Alternatively one can issue the **ut699_dbgon** *flag* command on the TSIM2 command line. The debug flags that are available are described for each core in the following sections and can be listed by **ut699_dbgon help**.

7.5. 10/100 Mbps Ethernet Media Access Controller interface

The Ethernet core simulation model is designed to functionally model the 10/100 Ethernet MAC available in the UT699. For core details and register specification please see the UT699 manual.

The following features are supported:

- Direct Memory Access
- Interrupts

7.5.1. Start up options

Ethernet core start up options

`-grethconnect host[:port]`
Connect Ethernet core to a packet server at the specified host and port. Default port is 2224.

7.5.2. Commands

Ethernet core TSIM commands

greth_connect host[:port]
Connect Ethernet core to a packet server at the specified host and port. Default port is 2224.

greth_status
Print Ethernet register status

7.5.3. Debug flags

The following debug flags are available for the Ethernet interface. Use the them in conjunction with the **ut699_dbgon** command to enable different levels of debug information.

Table 7.2. Ethernet debug flags

Flag	Trace
GAISLER_GRETH_ACC	GRETH accesses

Flag	Trace
GAISLER_GRETH_L1	GRETH accesses verbose
GAISLER_GRETH_TX	GRETH transmissions
GAISLER_GRETH_RX	GRETH reception
GAISLER_GRETH_RXPACKET	GRETH received packets
GAISLER_GRETH_RXCTRL	GRETH RX packet server protocol
GAISLER_GRETH_RXBDCTRL	GRETH RX buffer descriptors DMA
GAISLER_GRETH_RXBDCTRL	GRETH TX packet server protocol
GAISLER_GRETH_TXPACKET	GRETH transmitted packets
GAISLER_GRETH_IRQ	GRETH interrupts

7.5.4. Ethernet packet server

The simulation model relies on a packet server to receive and transmit the Ethernet packets. The packet server should open a TCP socket which the module can connect to. The Ethernet core is connected to a packet server using the `-grethconnect` start-up parameter or using the **greth_connect** command.

An example implementation of a packet server, named `greth_config`, is included in TSIM distribution. It uses the TUN/TAP interface in Linux, or the WinPcap library on Windows, to connect the GRETH core to a physical Ethernet LAN. This makes it easy to connect the simulated GRETH core to real hardware. It can provide a throughput in the order of magnitude of 500 to 1000 KiB/sec. See its distributed README for usage instructions.

7.5.5. Ethernet packet server protocol

Ethernet data packets have the following format. Note that each packet is prepended with a one word length field indicating the length of the packet to come (including its header).

31 0

0x0	LENGTH
31:0	LENGTH, specifies length of packet including the header

Header

31 16 15 8 7 5 4 0

0x4	RES	IPID=1	TYPE=0	RES
31:16	RES, reserved for future use			
15:8	IPID, IP core ID, must equal 1 for Ethernet			
7:5	TYPE, packet type, 0 for data packets			
4:0	RES, reserved for future use			

Payload

0x8 -	Ethernet frame

Figure 7.1. Ethernet data packet

7.6. SpaceWire interface with RMAP support

The UT699 AHB module contains 4 GRSPW cores which models the GRSPW cores available in the UT699. For core details and register specification please see the UT699 manual.

The following features are supported:

- Transmission and reception of SpaceWire packets
- Interrupts
- RMAP
- Modifying the link state

7.6.1. Start up options

SpaceWire core start up options

`-grspwXconnect host:port`
Connect GRPSW core X to packet server at specified server and port.

`-grspwXserver port`
Open a packet server for core X on specified port.

`-grspw_normap`
Disable the RMAP handler. RMAP packets will be stored to the DMA channel.

`-grspw_rmap`
Enable the RMAP handler. All RMAP packages will be simulated in hardware. Includes support for RMAP CRC. (Default)

`-grspw_rmapcrc`
Enable support for RMAP CRC. Performs RMAP CRC checks and calculations in hardware.

`-grspw_rxfreq freq`
Set the RX frequency which is used to calculate receive performance.

`-grspw_txfreq freq`
Set the TX frequency which is used to calculate transmission performance.

X in the above options has the range 1-4.

7.6.2. Commands

SpaceWire core TSIM commands

grspwX_connect host:port
Connect GRSPW core X to packet server at specified server and TCP port.

grspwX_server port
Open a packet server for core X on specified TCP port.

grspw_status
Print status for all GRSPW cores.

X in the above commands has the range 1-4.

7.6.3. Debug flags

The following debug flags are available for the SpaceWire interfaces. Use the them in conjunction with the **ut699_dbgon** command to enable different levels of debug information.

Table 7.3. SpaceWire debug flags

Flag	Trace
GAISLER_GRSPW_ACC	GRSPW accesses

Flag	Trace
GAISLER_GRSPW_RXPACKET	GRSPW received packets
GAISLER_GRSPW_RXCTRL	GRSPW rx protocol
GAISLER_GRSPW_TXPACKET	GRSPW transmitted packets
GAISLER_GRSPW_TXCTRL	GRSPW tx protocol

7.6.4. SpaceWire packet server

Each SpaceWire core can be configured independently as a packet server or client using either `-grspwXserver` or `-grspwXconnect`. TCP sockets are used for establishing the connections. When acting as a server the core can only accept a single connection.

For more flexibility, such as custom routing, an external packet server can be implemented using the protocol specified in the following sections. Each core should then be connected to that server.

7.6.5. SpaceWire packet server protocol

The protocol used to communicate with the packet server is described below. Three different types of packets are defined according to the table below.

Table 7.4. Packet types

Type	Value
Data	0
Time code	1
Modify link state	2

Note that all packets are prepended by a one word length field which specified the length of the coming packet including the header.

Data packet format:

	31				0
0x0	LENGTH				
31:0	LENGTH, specifies length of packet including the header				

Header

	31 16	15 8	7 5	4 1	0
0x4	RES	IPID=0	TYPE=0	RES	EEP
31:16	RES, reserved for future use				
15:8	IPID, IP core ID, must equal 0 for SpaceWire				
7:5	TYPE, packet type, 0 for data packets				
4:1	RES, reserved for future use, must be set to 0				
0	EEP, Error End of Packet. Set when the packet is truncated and terminated by an EEP.				

Payload

0x8 -	SpaceWire packet

Figure 7.2. SpaceWire data packet

Time code packet format:

31 … 0	
0x0	LENGTH
31:0	LENGTH, specifies length of packet including the header

Header

	31 … 16	15 … 8	7 … 5	4 … 0
0x4	RES	IPID=0	TYPE=1	RES
31:16	RES, reserved for future use, must be set to 0			
15:8	IPID, IP core ID, must equal 0 for SpaceWire			
7:5	TYPE, packet type, 1 for time code packets			
4:0	RES, reserved for future use, must be set to 0			

Payload

	31 … 8	7 … 6	5 … 0
0x8	RES	CT	CN
31:8	RES, reserved for future use, must be set to 0		
7:6	CT, time control flags		
5:0	CN, value of time counter		

Figure 7.3. SpaceWire time code packet

Link state packet format:

31 … 0	
0x0	LENGTH
31:0	LENGTH, specifies length of packet including the header

Header

	31 … 16	15 … 8	7 … 5	4 … 3	2 … 0
0x4	RES	IPID=0	TYPE=2	RES	LS
31:16	RES, reserved for future use, must be set to 0				
15:8	IPID, IP core ID, must equal 0 for SpaceWire				
7:5	TYPE, packet type, 2 for link state packets				
4:3	RES, reserved for future use, must be set to 0				
2:0	LS, Link State:	0	Error reset		
		1	Error wait		
		2	Ready		
		3	Started		
		4	Connecting		
		5	Run		

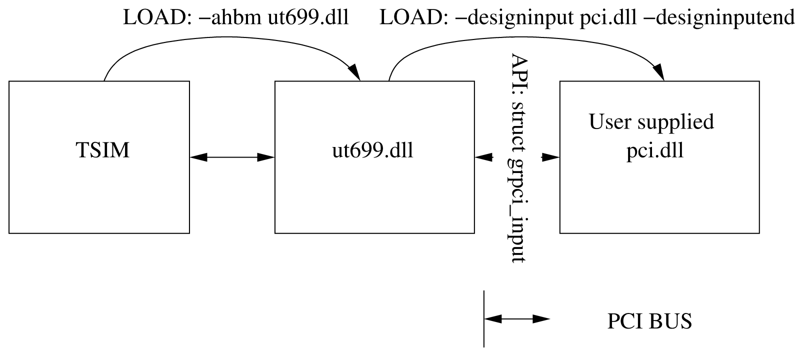
Figure 7.4. SpaceWire link state packet

7.7. PCI initiator/target and GPIO interface

The UT699 AHB module models the GPIO and PCI core available in the UT699 ASIC. For core details and register specification please see the UT699 manual.

The GPIO/PCI emulation is implemented by a two stage model:

1. The TSIM AHB module ut699.dll implements the GPIO and PCI core itself
2. A user supplied dynamic library models the devices on the PCI bus and the GPIO pins.



To load a user supplied dynamic library use the following command line switch:

```
-designinput <pciexample> <switches> -designinputend
```

This will load a user supplied dynamic library “pciexample”. In addition the switches between `-designinput` and `-designinputend` are local switches only propagated to the user dynamic library “pciexample”.

7.7.1. Commands

PCI Commands

pci_status
Print status for the PCI core

7.7.2. Debug flags

The following debug flags are available for the PCI interface. Use them in conjunction with the ut699_dbgon command to enable different levels of debug information.

Table 7.5. PCI interface debug flags

Flag	Trace
GAISLER_GRPCI_ACC	AHB accesses to/from PCI core
GAISLER_GRPCI_REGACC	GRPCI APB register accesses
GAISLER_GRPCI_DMA_REGACC	PCIDMA APB register accesses
GAISLER_GRPCI_DMA	GRPCI DMA accesses on the AHB bus
GAISLER_GRPCI_TARGET_ACC	GRPCI target accesses
GAISLER_GRPCI_INIT	Print summary on startup

7.7.3. User supplied dynamic library

The user supplied dynamic library should expose a public symbol `ut699inputsystem` of type `struct ut699_subsystem *`. The `struct ut699_subsystem` is defined as:

```
struct ut699_subsystem {
     void (*ut699_inp_setup) (int id, struct ut699_inp_layout *l,
                              char **argv, int argc);
     void (*ut699_inp_restart) (int id, struct ut699_inp_layout *l);
     struct sim_interface *simif;
};
```

At initialization the callback `ut699_inp_setup` will be called once, supplied with a pointer to a structure of type `struct ut699_inp_layout`.

```
struct ut699_inp_layout {
    struct grpci_input grpci;
    struct gpio_input gpio;
};
```

The callback ut699_inp_restart will be called every time the simulator restarts and the PCI user module can access the global TSIM `struct sim_interface` structure through the `simif` member. See Chapter 5 for more details.

The user supplied dynamic library should claim the `ut699_inp_layout.grpci` member of the structure by using the `INPUT_CLAIM(l->grpci)` macro (see the example below). A `struct grpci_input` consists of callbacks that model the PCI bus (see Section 7.7.4).

A typical user supplied dynamic library would look like this:

```
#include "tsim.h"
#include "inputprovider.h"
int pci_acc(struct grpci_input *ctrl, int cmd, unsigned int addr, unsigned int wsize,
            unsigned int *data, unsigned int *abort, unsigned int *ws) {

    ... BUS access implementation ...

}

static void ut699_inp_setup (int id, struct ut699_inp_layout *l, char **argv, int argc)
{
   printf("Entered PCI setup\n");

   if (INPUT_ISCLAIMED(l->grpci)) {
         printf("module user for PCI already allocated \n");
         return;
   }

   for(i = 0; i &lt; argc; i++) {
      ... do argument processing ...
   }

   l->grpci.acc = pci_acc;

   ... do module setup ...

   printf("ut699_inp_setup: Claiming %s\n", l->grpci._b.name);
   INPUT_CLAIM(l->grpci);
   return;
}

static struct ut699_subsystem ut699_pci = {
   ut699_inp_setup,0,0
};

struct ut699_subsystem *ut699inputsystem = &amp;ut699_pci;
```

A typical Makefile that would create a user supplied dynamic library pci.(dll|so) from pci.c would look like this:

```
M_DLL_FIX = $(if $(strip $(shell uname | grep MINGW32)),dll,so)
M_LIB   = $(if $(strip $(shell uname | grep MINGW32)),-lws2_32 -luser32 -lkernel32 -
   lwinmm,)
all:pci.$(M_DLL_FIX)

pci.$(M_DLL_FIX) : pci.o
$(CC) -shared -g pci.o -o pci.$(M_DLL_FIX) $(M_LIB)

pci.o:   pci.c \
inputprovider.h
```

```
$(CC) -fPIC -c -g -O0 pci.c -o pci.o
clean:
-rm -f *.o *.so
```

7.7.4. PCI bus model API

The structure `struct grpci_input` models the PCI bus. It is defined as:

```
/* ut699 pci input provider */

struct grpci_input {
   struct input_inp _b;

   int (*acc)(struct grpci_input *ctrl, int cmd, unsigned int addr,
              unsigned int *data, unsigned int *abort, unsigned int *ws);

   int (*target_acc)(struct grpci_input *ctrl, int cmd, unsigned int addr,
                     unsigned int *data, unsigned int *mexc);
};
```

The acc callback should be set by the PCI user module at startup. It is called by the UT699 module whenever it reads/writes as a PCI bus master.

Table 7.6. acc callback parameters

Parameter	Description
cmd	Command to execute, see Section 7.7.1 details
addr	PCI address
data	Data buffer, fill for read commands, read for write commands
wsize	0: 8-bit access 1: 16-bit access, 2: 32-bit access, 3: 64-bit access. 64 bit is only used to model STD instructions to the GRPCI AHB slave
ws	Number of PCI clocks it shall to complete the transaction
abort	Set to 1 to generate target abort, 0 otherwise

The return value of acc determines if the transaction terminates successfully (1) or with master abort (0).

The callback target_acc is installed by the UT699 AHB module. The PCI user dynamic library can call this function to initiate an access to the UT699 PCI target.

Table 7.7. target_acc parameters

Parameter	Description
cmd	Command to execute, see Section 7.7.1 for details. I/O cycles are not supported by the UT699 target.
addr	PCI address
data	Data buffer, returned data for read commands, supply data for write commands
wsize	0: 8-bit access 1: 16-bit access, 2: 32-bit access
mexc	0 if access is successful, 1 in case of target abort

If the address matched MEMBAR0, MEMBAR1 or CONFIG target_acc will return 1 otherwise 0.

7.7.5. GPIO model API

The structure `struct gpio_input` models the GPIO pins. It is defined as:

```
/* GPIO input provider */
struct gpio_input {
    struct input_inp _b;
    int (*gpioout)(struct gpio_input *ctrl, unsigned int out);
```

```
        int (*gpioin) (struct gpio_input *ctrl, unsigned int in);
    };
```

The `gpioout` callback should be set by the user module at startup. The `gpioin` callback is set by the U699 AHB module. The `gpioout` callback is called by the UT699 module whenever a GPIO output pin changes. The `gpioin` callback is called by the user module when the input pins should change. Typically the user module would register an event handler at a certain time offset and call `gpioin` from within the event handler.

Table 7.8. gpioout callback parameters

Parameter	Description
out	The values of the output pins

Table 7.9. gpioin callback parameters

Parameter	Description
in	The input pin values

The return value of `gpioin/gpioout` is ignored.

7.8. CAN interface

The UT699 AHB module contains 2 CAN_OC cores which models the CAN_OC cores available in the UT699. For core details and register specification please see the UT699 manual.

7.8.1. Start up options

CAN core start up options

`-can_ocX_connect host:port`
: Connect CAN_OC core X to packet server to specified server and TCP port.

`-can_ocX_server port`
: Open a packet server for CAN_OC core X on specified TCP port.

`-can_ocX_ack [0|1]`
: Specifies whether the CAN_OC core will wait for a acknowledgment packet on transmission. This option must be put after `-can_ocX_connect.`

X in the above options is in the range 1-2.

7.8.2. Commands

CAN core TSIM commands

can_ocX_connect host:port
: Connect CAN_OC core X to packet server to specified server and TCP port.

can_ocX_server port
: Open a packet server for CAN_OC core X on specified TCP port.

can_ocX_ack <0|1>
: Specifies whether the CAN_OC core will wait for a acknowledgment packet on transmission. This command should only be issued after a connection has been established.

can_ocX_status
: Prints out status information for the CAN_OC core.

X in the above commands is in the range 1-2.

7.8.3. Debug flags

The following debug flags are available for the CAN interfaces. Use them in conjunction with the **ut699_dbgon** command to enable different levels of debug information.

Table 7.10. CAN debug flags

Flag	Trace
GAISLER_CAN_OC_ACC	CAN_OC register accesses
GAISLER_CAN_OC_RXPACKET	CAN_OC received messages
GAISLER_CAN_OC_TXPACKET	CAN_OC transmitted messages
GAISLER_CAN_OC_ACK	CAN_OC acknowledgements
GAISLER_CAN_OC_IRQ	CAN_OC interrupts

7.8.4. Packet server

Each CAN_OC core can be configured independently as a packet server or client using either `-can_ocX_server` or `-can_ocX_connect`. When acting as a server the core can only accept a single connection.

7.8.5. CAN packet server protocol

The protocol used to communicate with the packet server is described below. Four different types of packets are defined according to the table below.

Table 7.11. CAN packet types

Type	Value
Message	0x00
Error counter	0xFD
Acknowledge	0xFE
Acknowledge config	0xFF

7.8.5.1. CAN message packet format

Used to send and receive CAN messages.

31 0

0x0	LENGTH
31:0	LENGTH, specifies the length of the rest of the packet

CAN message

Byte #	Description	Bits (MSB-LSB)							
		7	6	5	4	3	2	1	0
4	Protocol ID = 0	Prot ID 7-0							
5	Control	FF	RTR	-	-	DLC (max 8 bytes)			
6-9	ID (32 bit word in network byte order)	ID 10-0 (bits 31 - 11 ignored for standard frame format) ID 28-0 (bits 31-29 ignored for extended frame format)							
10-17	Data byte 1 - DLC	Data byte *n* 7-0							

Figure 7.5. CAN message packet format

7.8.5.2. Error counter packet format

Used to write the RX and TX error counter of the modelled CAN interface.

31	0
0x0	LENGTH
31:0	LENGTH, specifies the length of the rest of the packet

Error counter packet

Byte #	Field	Description
4	Packet type	Type of packet, 0xFD for error counter packets
5	Register	0 - RX error counter, 1 - TX error counter
6	Value	Value to write to error counter

Figure 7.6. Error counter packet format

7.8.5.3. Acknowledge packet format

If the acknowledge function has been enabled through the start up option or command the CAN interface will wait for an acknowledge packet each time it transmits a message. To enable the CAN receiver to send acknowledge packets (either NAK or ACK) an acknowledge configuration packet must be sent. This is done automatically by the CAN interface when **can_ocX_ack** is issued.

31	0
0x0	LENGTH
31:0	LENGTH, specifies the length of the rest of the packet

Acknowledge packet

Byte #	Field	Description
4	Packet type	Type of packet, 0xFE for acknowledge packets
5	Ack payload	0 - No acknowledge, 1 - Acknowledge

Figure 7.7. Acknowledge packet format

7.8.5.4. Acknowledge packet format

This packet is used for enabling/disabling the transmission of acknowledge packets and their payload (ACK or NAK) by the CAN receiver. The CAN transmitter will always wait for an acknowledge if started with `-can_ocX_ack` or if the **can_ocX_ack** command has been issued.

31	0
0x0	LENGTH
31:0	LENGTH, specifies the length of the rest of the packet

Acknowledge configuration packet

Byte #	Field	Description	
4	Packet type	Type of packet, 0xFF for acknowledge configuration packets	
5	Ack configuration	bit 0	Unused
		bit 1	Ack packet enable, 1 - enabled, 0 - disabled
		bit 2	Set ack packet payload, 1 - ACK, 0 - NAK

Figure 7.8. Acknowledge configuration packet format

8. Cobham UT700 AHB module

8.1. Overview

The UT700 AHB module is very similar to the UT699 AHB module described in the previous chapter. The differences between the UT700 and the UT699 models is the added SPI model that is only present in the UT700 AHB module and that it has GRSPW2 cores instead of GRSPW cores and that the debug flag toggling command is **ut700_dbgon**,

For information on the CAN, Spacewire, PCI and GPIO interfaces of the UT700 module, see the UT699 documentation in Chapter 7. The TSIM command, flag and packet interface is the same for both GRSPW and GRSPW2.

The following files are delivered with the UT700 TSIM module:

Table 8.1. Files delivered with the UT700 TSIM module

File	Description
ut700/linux/ut700.so	UT700 AHB module for Linux
ut700/win32/ut700.dll	UT700 AHB module for Windows
ut700/examples/input	The input directory contains two examples of PCI user modules
ut700/examples/input/README.txt	Description of the user module examples
ut700/examples/input/Makefile	Makefile for building the user modules
ut700/examples/input/pci.c	PCI user module example that makes UT700 PCI initiator accesses
ut700/examples/input/pci_target.c	PCI user module example that makes UT700 PCI target accesses
ut700/examples/input/ut700inputprovider.h	Interface between the UT700 module and the user defined PCI module
ut700/examples/input/pci_input.h	UT700 PCI input provider definitions
ut700/examples/input/input.h	Generic input provider definitions
ut700/examples/input/tsim.h	TSIM interface definitions
ut700/examples/input/end.h	Defines the endian of the local machine
ut700/examples/test	The test directory contains tests that can be executed in TSIM
ut700/examples/test/README.txt	Description of the tests
ut700/examples/test/Makefile	Makefile for building the tests
ut700/examples/test/cansend.c	CAN transmission test
ut700/examples/test/canrec.c	CAN reception test
ut700/examples/test/pci.c	PCI interface test
ut700/examples/test/pcitest.h	Header file for PCI test

8.2. Loading the module

The module is loaded using the TSIM2 option `-ahbm`. All core specific options described in the following sections need to be surrounded by the options `-designinput` and `-designinputend`, e.g:

On Linux:

```
tsim-leon3 -ut700 -ahbm ./ut700/linux/ut700.so
          -designinput ./ut700/examples/input/pci.so -designinputend
```

On Windows:

```
tsim-leon3 -ut700 -ahbm ut700/win32/ut700.dll
           -designinput ./ut700/examples/input/pci.dll -designinputend
```

The option `-ut700` needs to be given to TSIM to enable the UT700 processor configuration.

8.3. SPI bus model API

The UT700 user supplied so/dll differs from that of the UT699 in the addition of the SPI bus model API. The structure `struct spi_input` models the SPI bus. It is defined as:

```
/* Spi input provider */

struct spi_input {
    struct input_inp _b;
    int (*spishift)(struct spi_input *ctrl, uint32 select, uint32 bitcnt,
                      uint32 out, uint32 *in);
};
```

The spishift callback should be set by the SPI user module at startup. It is called by the UT700 module whenever it shifts a word through the SPI bus.

Table 8.2. spishift callback parameters

Parameter	Description
select	Slave select bits
bitcnt	Number of bits set in the MODE register, if bitcnt is -1 then the operation is not a shift and the call is to indicate a *select* change, i.e. if the core is disabled.
out	Shift out (tx) data
in	Shift in (rx) data

9. Cobham Gaisler GR712 AHB module

9.1. Overview

GR712 AHB module is a loadable AHB module that implements the GR712 peripherals including: GPIO, GRTIMER with latch, SPI, CAN, GRETH, SPACEWIRE, AHBRAM and extra UARTS.

The following files are delivered with the GR712 TSIM module:

Table 9.1. Files delivered with the GR712 TSIM module

File	Description
gr712/linux/gr712.so	GR712 AHB module for Linux
gr712/win32/gr712.dll	GR712 AHB module for Windows
gr712/examples/input	The input directory contains two examples of user modules
gr712/examples/input/README.txt	Description of the user module examples
gr712/examples/input/Makefile	Makefile for building the user modules
gr712/examples/input/spi.c	SPI user module example emulating a Intel SPI flash
gr712/examples/input/gpio.c	GPIO user module emulating GPIO bit toggle
gr712/examples/input/gr712inputprovider.h	Interface between the GR712 module and the user module

9.2. Loading the module

The module is loaded using the TSIM2 option `-ahbm`. All core specific options described in the following sections need to be surrounded by the options `-designinput` and `-designinputend`, e.g:

On Linux:

```
tsim-leon -gr712rc -ahbm ./gr712/linux/gr712.so
          -designinput ./gr712/examples/input/spi.so -designinputend
```

On Windows:

```
tsim-leon -gr712rc -ahbm ./gr712/win32/gr712.dll
          -designinput ./gr712/examples/input/spi.dll -designinputend
```

The option `-gr712rc` needs to be given to TSIM to enable the GR712 processor configuration. The above line loads the GR712 AHB module ./gr712.so which in turn loads the SPI user module ./spi.so. The SPI user module ./spi.so communicates with ./gr712.so using the user module interface described in gr712inputprovider.h,, while ./gr712.so communicates with TSIM via the AHB interface.

9.3. Debugging

To enable printout of debug information the `-gr712_dbgon` *`flag`* switch can be used. Alternatively one can issue the **gr712_dbgon** *`flag`* command on the TSIM2 command line. The debug flags that are available are described for each core in the following sections and can be listed by **gr712_dbgon help**.

9.4. CAN interface

The GR712 AHB module contains 2 CAN_OC cores which models the CAN_OC cores available in the GR712. For core details and register specification please see the GR712 manual.

9.4.1. Start up options

CAN core start up options

`-can_ocX_connect host:port`
: Connect CAN_OC core X to packet server to specified server and TCP port.

`-can_ocX_server port`
: Open a packet server for CAN_OC core X on specified TCP port.

`-can_ocX_ack [0|1]`
: Specifies whether the CAN_OC core will wait for a acknowledgment packet on transmission. This option must be put after `-can_ocX_connect.`

X in the above options is in the range 0-1.

9.4.2. Commands

CAN core TSIM commands

can_ocX_connect host:port
: Connect CAN_OC core X to packet server to specified server and TCP port.

can_ocX_server port
: Open a packet server for CAN_OC core X on specified TCP port.

can_ocX_ack <0|1>
: Specifies whether the CAN_OC core will wait for a acknowledgment packet on transmission. This command should only be issued after a connection has been established.

can_ocX_status
: Prints out status information for the CAN_OC core.

X in the above commands is in the range 0-1.

9.4.3. Debug flags

The following debug flags are available for the CAN interfaces. Use them in conjunction with the **gr712_dbgon** command to enable different levels of debug information. To toggle debug output for individual cores, use the **can_ocX_dbg** command, where X is in the range 0-1.

Table 9.2. CAN debug flags

Flag	Trace
GAISLER_CAN_OC_ACC	CAN_OC register accesses
GAISLER_CAN_OC_RXPACKET	CAN_OC received messages
GAISLER_CAN_OC_TXPACKET	CAN_OC transmitted messages
GAISLER_CAN_OC_ACK	CAN_OC acknowledgements
GAISLER_CAN_OC_IRQ	CAN_OC interrupts

9.4.4. Packet server

Each CAN_OC core can be configured independently as a packet server or client using either `-can_ocX_server` or `-can_ocX_connect`. When acting as a server the core can only accept a single connection.

9.4.5. CAN packet server protocol

The protocol used to communicate with the packet server is described below. Four different types of packets are defined according to the table below.

Table 9.3. CAN packet types

Type	Value
Message	0x00

Type	Value
Error counter	0xFD
Acknowledge	0xFE
Acknowledge config	0xFF

9.4.5.1. CAN message packet format

Used to send and receive CAN messages.

	31	0
0x0	LENGTH	
31:0	LENGTH, specifies the length of the rest of the packet	

CAN message

Byte #	Description	Bits (MSB-LSB)							
		7	6	5	4	3	2	1	0
4	Protocol ID = 0	Prot ID 7-0							
5	Control	FF	RTR	-	-	DLC (max 8 bytes)			
6-9	ID (32 bit word in network byte order)	ID 10-0 (bits 31 - 11 ignored for standard frame format) ID 28-0 (bits 31-29 ignored for extended frame format)							
10-17	Data byte 1 - DLC	Data byte *n* 7-0							

Figure 9.1. CAN message packet format

9.4.5.2. Error counter packet format

Used to write the RX and TX error counter of the modelled CAN interface.

	31	0
0x0	LENGTH	
31:0	LENGTH, specifies the length of the rest of the packet	

Error counter packet

Byte #	Field	Description
4	Packet type	Type of packet, 0xFD for error counter packets
5	Register	0 - RX error counter, 1 - TX error counter
6	Value	Value to write to error counter

Figure 9.2. Error counter packet format

9.4.5.3. Acknowledge packet format

If the acknowledge function has been enabled through the start up option or command the CAN interface will wait for an acknowledge packet each time it transmits a message. To enable the CAN receiver to send acknowledge packets (either NAK or ACK) an acknowledge configuration packet must be sent. This is done automatically by the CAN interface when **can_ocX_ack** is issued.

	31	0
0x0	LENGTH	
31:0	LENGTH, specifies the length of the rest of the packet	

Acknowledge packet

Byte #	Field	Description
4	Packet type	Type of packet, 0xFE for acknowledge packets
5	Ack payload	0 - No acknowledge, 1 - Acknowledge

Figure 9.3. Acknowledge packet format

9.4.5.4. Acknowledge packet format

This packet is used for enabling/disabling the transmission of acknowledge packets and their payload (ACK or NAK) by the CAN receiver. The CAN transmitter will always wait for an acknowledge if started with `-can_ocX_ack` or if the **can_ocX_ack** command has been issued.

	31	0
0x0	LENGTH	
31:0	LENGTH, specifies the length of the rest of the packet	

Acknowledge configuration packet

Byte #	Field	Description	
4	Packet type	Type of packet, 0xFF for acknowledge configuration packets	
5	Ack configuration	bit 0	Unused
		bit 1	Ack packet enable, 1 - enabled, 0 - disabled
		bit 2	Set ack packet payload, 1 - ACK, 0 - NAK

Figure 9.4. Acknowledge configuration packet format

9.5. 10/100 Mbps Ethernet Media Access Controller interface

The Ethernet core simulation model is designed to functionally model the 10/100 Ethernet MAC available in the GR712. For core details and register specification please see the GR712 manual.

The following features are supported:

- Direct Memory Access
- Interrupts

9.5.1. Start up options

Ethernet core start up options

`-grethconnect host[:port]`
Connect Ethernet core to a packet server at the specified host and port. Default port is 2224.

9.5.2. Commands

Ethernet core TSIM commands

greth_connect host[:port]
Connect Ethernet core to a packet server at the specified host and port. Default port is 2224.

greth_status
Print Ethernet register status

9.5.3. Debug flags

The following debug flags are available for the Ethernet interface. Use the them in conjunction with the **gr712_dbgon** command to enable different levels of debug information.

Table 9.4. Ethernet debug flags

Flag	**Trace**
GAISLER_GRETH_ACC	GRETH accesses
GAISLER_GRETH_L1	GRETH accesses verbose
GAISLER_GRETH_TX	GRETH transmissions
GAISLER_GRETH_RX	GRETH reception
GAISLER_GRETH_RXPACKET	GRETH received packets
GAISLER_GRETH_RXCTRL	GRETH RX packet server protocol
GAISLER_GRETH_RXBDCTRL	GRETH RX buffer descriptors DMA
GAISLER_GRETH_RXBDCTRL	GRETH TX packet server protocol
GAISLER_GRETH_TXPACKET	GRETH transmitted packets
GAISLER_GRETH_IRQ	GRETH interrupts

9.5.4. Ethernet packet server

The simulation model relies on a packet server to receive and transmit the Ethernet packets. The packet server should open a TCP socket which the module can connect to. The Ethernet core is connected to a packet server using the `-grethconnect` start-up parameter or using the **greth_connect** command.

An example implementation of a packet server, named `greth_config`, is included in TSIM distribution. It uses the TUN/TAP interface in Linux, or the WinPcap library on Windows, to connect the GRETH core to a physical Ethernet LAN. This makes it easy to connect the simulated GRETH core to real hardware. It can provide a throughput in the order of magnitude of 500 to 1000 KiB/sec. See its distributed README for usage instructions.

9.5.5. Ethernet packet server protocol

Ethernet data packets have the following format. Note that each packet is prepended with a one word length field indicating the length of the packet to come (including its header).

	31	0
0x0	LENGTH	
31:0	LENGTH, specifies length of packet including the header	

Header

	31 16	15 8	7 5	4 0
0x4	RES	IPID=1	TYPE=0	RES
31:16	RES, reserved for future use			
15:8	IPID, IP core ID, must equal 1 for Ethernet			
7:5	TYPE, packet type, 0 for data packets			
4:0	RES, reserved for future use			

Payload

0x8 -	Ethernet frame

Figure 9.5. Ethernet data packet

9.6. SpaceWire interface with RMAP support

The GR712 AHB module contains 6 GRSPW2 cores which models the GRSPW2 cores available in the GR712. For core details and register specification please see the GR712 manual.

The following features are supported:

- Transmission and reception of SpaceWire packets
- Interrupts
- Time codes
- RMAP
- Modifying the link state

9.6.1. Start up options

SpaceWire core start up options

`-grspwXconnect host:port`
: Connect GRPSW core X to packet server at specified server and port.

`-grspwXserver port`
: Open a packet server for core X on specified port.

`-grspw_normap`
: Disable the RMAP handler. RMAP packets will be stored to the DMA channel.

`-grspw_rmap`
: Enable the RMAP handler. All RMAP packages will be simulated in hardware. Includes support for RMAP CRC. (Default)

`-grspw_rmapcrc`
: Enable support for RMAP CRC. Performs RMAP CRC checks and calculations in hardware.

`-grspw_rxfreq freq`
: Set the RX frequency which is used to calculate receive performance.

`-grspw_txfreq freq`
: Set the TX frequency which is used to calculate transmission performance.

X in the above options has the range 0-5.

9.6.2. Commands

SpaceWire core TSIM commands

grspwX_connect host:port
: Connect GRSPW2 core X to packet server at specified server and TCP port.

grspwX_server port
: Open a packet server for core X on specified TCP port.

grspw_status
: Print status for all GRSPW2 cores.

X in the above commands has the range 0-5.

9.6.3. Debug flags

The following debug flags are available for the SpaceWire interfaces. Use the them in conjunction with the **gr712_dbgon** command to enable different levels of debug information. To toggle debug output for individual cores, use the **grspwX_dbg** command, where X is in the range 0-5.

Table 9.5. SpaceWire debug flags

Flag	Trace
GAISLER_GRSPW_ACC	GRSPW accesses
GAISLER_GRSPW_RXPACKET	GRSPW received packets
GAISLER_GRSPW_RXCTRL	GRSPW rx protocol
GAISLER_GRSPW_TXPACKET	GRSPW transmitted packets
GAISLER_GRSPW_TXCTRL	GRSPW tx protocol
GAISLER_GRSPW_RMAP	GRSPW RMAP accesses
GAISLER_GRSPW_RMAPPACKET	GRSPW RMAP packet dumps
GAISLER_GRSPW_RMAPPACKDE	GRSPW RMAP packet decoding
GAISLER_GRSPW_DMAERR	GRSPW DMA errors

9.6.4. SpaceWire packet server

Each SpaceWire core can be configured independently as a packet server or client using either `-grspwXserver` or `-grspwXconnect`. TCP sockets are used for establishing the connections. When acting as a server the core can only accept a single connection.

For more flexibility, such as custom routing, an external packet server can be implemented using the protocol specified in the following sections. Each core should then be connected to that server.

9.6.5. SpaceWire packet server protocol

The protocol used to communicate with the packet server is described below. Three different types of packets are defined according to the table below.

Table 9.6. Packet types

Type	Value
Data	0
Time code	1

Type	Value
Modify link state	2

Note that all packets are prepended by a one word length field which specified the length of the coming packet including the header.

Data packet format:

	31 0
0x0	LENGTH
31:0	LENGTH, specifies length of packet including the header

Header

	31 16	15 8	7 5	4 1	0
0x4	RES	IPID=0	TYPE=0	RES	EEP
31:16	RES, reserved for future use				
15:8	IPID, IP core ID, must equal 0 for SpaceWire				
7:5	TYPE, packet type, 0 for data packets				
4:1	RES, reserved for future use, must be set to 0				
0	EEP, Error End of Packet. Set when the packet is truncated and terminated by an EEP.				

Payload

0x8 -	SpaceWire packet

Figure 9.6. SpaceWire data packet

Time code packet format:

	31 0
0x0	LENGTH
31:0	LENGTH, specifies length of packet including the header

Header

	31 16	15 8	7 5	4 0
0x4	RES	IPID=0	TYPE=1	RES
31:16	RES, reserved for future use, must be set to 0			
15:8	IPID, IP core ID, must equal 0 for SpaceWire			
7:5	TYPE, packet type, 1 for time code packets			
4:0	RES, reserved for future use, must be set to 0			

Payload

	31 8	7 6	5 0
0x8	RES	CT	CN
31:8	RES, reserved for future use, must be set to 0		
7:6	CT, time control flags		
5:0	CN, value of time counter		

Figure 9.7. SpaceWire time code packet

Link state packet format:

	31	0
0x0	LENGTH	
31:0	LENGTH, specifies length of packet including the header	

Header

	31 16	15 8	7 5	4 3	2 0
0x4	RES	IPID=0	TYPE=2	RES	LS
31:16	RES, reserved for future use, must be set to 0				
15:8	IPID, IP core ID, must equal 0 for SpaceWire				
7:5	TYPE, packet type, 2 for link state packets				
4:3	RES, reserved for future use, must be set to 0				
2:0	LS, Link State:		0	Error reset	
			1	Error wait	
			2	Ready	
			3	Started	
			4	Connecting	
			5	Run	

Figure 9.8. SpaceWire link state packet

9.7. SPI and GPIO user modules

The user supplied dynamic library should expose a public symbol `gr712inputsystem` of type `struct gr712_subsystem *`. The `struct gr712_subsystem` is defined in gr712inputprovider.h as:

```
struct gr712_subsystem {
    void (*gr712_inp_setup) (int id,
                    struct gr712_inp_layout * l,
                    char **argv, int argc);
    void (*gr712_inp_restart) (int id,
                     struct gr712_inp_layout * l);
    struct sim_interface *simif;
};
```

The callback gr712_inp_restart will be called every time the simulator restarts. At initialization the callback gr712_inp_setup will be called once, supplied with a pointer to structure `struct gr712_inp_layout` defined in gr712inputprovider.h (see Section 9.7.1 and Section 9.7.2 for details):

```
struct gr712_inp_layout {
    struct gpio_input gpio[2];
    struct spi_input spi;
};
```

The user module can access the global TSIM `struct sim_interface` structure through the `simif` member. See Chapter 5 for more details.

The user supplied dynamic library should claim the gr712_inp_layout.gpio or gr712_inp_layout. spi members by using the INPUT_CLAIM macro, i.e. INPUT_CLAIM(l->gpio) (see the example below).

A typical user supplied dynamic library would look like this:

```
/* simple gpio user module that toggles all input bits */
#include <stdio.h>
#include <string.h>
#include "tsim.h"
```

```
#include "gr712inputprovider.h"
extern struct gr712_subsystem *gr712inputsystem;
static struct gr712_inp_layout *lay = 0;

static void Change(struct gpio_input *ctrl) {
   ...
}

int gpioout(struct gpio_input *ctrl, unsigned int out) {
   ...
}

static void gr712_inp_setup (int id,
                        struct gr712_inp_layout * l,
                        char **argv, int argc) {
   lay = l;
   printf("User-dll: gr712_inp_setup:Claiming %s\n", l->gpio[0]._b.name);
   INPUT_CLAIM(l->gpio[0]);
   l->gpio[0].gpioout = gpioout;
   gr712inputsystem->simif->event(Change,(unsigned long)&l->gpio[0],10000000);
}

static struct gr712_subsystem gr712_gpio = {
   gr712_inp_setup,0,0
};

struct gr712_subsystem *gr712inputsystem = &gr712_gpio;
```

A typical Makefile that would create a user supplied dynamic library gpio.(dll|so) would look like this:

```
   M_DLL_FIX=$(if $(strip $(shell uname|grep MINGW32)),dll,so)
   M_LIB=$(if $(strip $(shell uname|grep MINGW32)),-lws2_32 -luser32 -lkernel32 -lwinmm,)
   all:gpio.$(M_DLL_FIX)

   pci.$(M_DLL_FIX) : gpio.o
       $(CC) -shared -g gpio.o -o gpio.$(M_DLL_FIX) $(M_LIB)

   gpio.o:   gpio.c
       $(CC) -fPIC -c -g -O0 gpio.c -o gpio.o
clean:
       -rm -f *.o *.so
```

The user can then specify the user module to be loaded by the gr712.so AHB module using the `-designinput` and `-designinputend` command line options:

`-designinput ./gr712/examples/input/gpio.so -designinputend`

These switches are interpreted by gr712.so.

9.7.1. SPI bus model API

The structure `struct spi_input` models the SPI bus. It is defined as:

```
/* Spi input provider */

struct spi_input {
   struct input_inp _b;
   int (*spishift)(struct spi_input *ctrl, uint32 select, uint32 bitcnt,
                  uint32 out, uint32 *in);
};
```

The spishift callback should be set by the SPI user module at startup. It is called by the GR712 module whenever it shifts a word through the SPI bus.

Table 9.7. spishift callback parameters

Parameter	Description
select	Slave select bits (in case of GR712 these should be ignored and GPIO used instead)
bitcnt	Number of bits set in the MODE register, if bitcnt is -1 then the operation is not a shift and the call is to indicate a *select* change, i.e. if the core is disabled.
out	Shift out (tx) data

Parameter	Description
in	Shift in (rx) data

The return value of spishift is ignored.

9.7.2. GPIO model API

The structure `struct gpio_input` models the GPIO pins. It is defined as:

```
/* GPIO input provider */
struct gpio_input {
    struct input_inp _b;
    int (*gpioout)(struct gpio_input *ctrl, unsigned int out);
    int (*gpioin) (struct gpio_input *ctrl, unsigned int in);
};
```

The gpioout callback should be set by the user module at startup. The gpioin callback is set by the GR712 AHB module. The gpioout callback is called by the GR712 module whenever a GPIO output pin changes. The gpioin callback is called by the user module when the input pins should change. Typically the user module would register an event handler at a certain time offset and call gpioin from within the event handler.

Table 9.8. gpioout callback parameters

Parameter	Description
out	The values of the output pins

Table 9.9. gpioin callback parameters

Parameter	Description
in	The input pin values

The return value of gpioin/gpioout is ignored.

9.8. UART interfaces

The GR712 module adds five extra UARTS in addition to the one built in UART (the second built in UART is is disabled by the `-gr712rc` option). The extra UARTS are numbered 2 through 6.

9.8.1. Start up options

`-uartX` *`device`*
: Works like the ordinary `-uartX` *`device`* option but for X in the range 2-6, with the extra possibility to set the UART to use stdin and stdout by using `-uartX stdio`.

9.8.2. Commands

uartX_connect *`device`*
: Has the same effect as `-uartX` *`device`* above but can as a command.

uartX_status
: Shows the status of the UART.

uartX_dbg < *`flag`* | **list** | **help** | **clean** >
: Toggle, show, disable or show help for debug options for the given UART.

X in the above commands is in the range 2-6.

10. Atmel AT697 PCI emulation

10.1. Overview

The PCI emulation is implemented as a AT697 AHB module that will process all accesses to memory region 0xa0000000 - 0xf0000000 (AHB slave mode) and the APB registers starting at 0x80000100. The AT697 AHB module implements all registers of the PCI core. It will in turn load the PCI user modules that will implement the devices. The AT697 AHB module is supposed to be the PCI host. Both PCI Initiator and PCI Target mode are supported. The interface to the PCI user modules is implemented on bus level. Two callbacks model the PCI bus.

The following files are delivered with the AT697 TSIM module:

Table 10.1. Files delivered with the AT697 TSIM module

File	Description
at697/linux/at697.so	AT697 AHB module for Linux
at697/win32/at697.dll	AT697 AHB module for Windows
Input	The input directory contains two examples of PCI user modules
at697/examples/input/README.txt	Description of the user module examples
at697/examples/input/Makefile	Makefile for building the user modules
at697/examples/input/pci.c	PCI user module example that makes AT697 PCI initiator accesses
at697/examples/input/pci_target.c	PCI user module example that makes AT697 PCI target accesses
at697/examples/input/at697inputprovider.h	Interface between the AT697 module and the user defined PCI module
at697/examples/input/pci_input.h	AT697 PCI input provider definitions
at697/examples/input/input.h	Generic input provider definitions
at697/examples/input/tsim.h	TSIM interface definitions
at697/examples/input/end.h	Defines the endian of the local machine

10.2. Loading the module

The module is loaded using the TSIM2 option `-ahbm`. All core specific options described in the following sections need to be surrounded by the options `-designinput` and `-designinputend`, e.g:

On Linux:

```
tsim-leon -ahbm ./at697/linux/at697.so
          -designinput ./at697/examples/input/pci.so -designinputend
```

On Windows:

```
tsim-leon -ahbm ./at697/win32/at697.dll
          -designinput ./at697/examples/input/pci.dll -designinputend
```

This loads the AT697 AHB module ./at697.so which in turn loads the PCI user module ./pci.so. The PCI user module ./pci.so communicates with ./at697.so using the PCI user module interface, while ./at697.so communicates with TSIM via the AHB interface.

10.3. AT697 initiator mode

The PCI user module should supply one callback function `acc()`. The AT697 AHB module will call this function to emulate AHB slave mode accesses or DMA accesses that are forwarded via `acc()`. The cmd parameter determines which command to use. Configuration cycles have to be handled by the PCI user module.

10.4. AT697 target mode

The AT697 AHB module supplies one callback `target_acc()` to the PCI user modules to implement target mode accesses from the PCI bus to the AHB bus. The PCI user module should trigger access events itself by inserting itself into the event queue.

10.5. Definitions

```
#define  ESA_PCI_SPACE_IO       0
#define  ESA_PCI_SPACE_MEM      1
#define  ESA_PCI_SPACE_CONFIG   2
#define  ESA_PCI_SPACE_MEMLINE  3

/* atc697 pci input provider */
struct esa_pci_input {
struct input_inp _b;

int (*acc)(struct esa_pci_input *ctrl, int cmd, unsigned int addr,
           unsigned int *data, unsigned int *abort,unsigned int *ws);

int (*target_acc)(struct esa_pci_input *ctrl, int cmd, unsigned int addr,
                  unsigned int *data, unsigned int *mexc);

};
```

10.5.1. PCI command table

```
0000:   "IRQ acknowledge",
0001:   "Special cycle",
0010:   "I/O Read",
0011:   "I/O Write",
0100:   "Reserved",
0101:   "Reserved",
0110:   "Memory Read",
0111:   "Memory Write",
1000:   "Reserved",
1001:   "Reserved",
1010:   "Configuration Read",
1011:   "Configuration Write",
1100:   "Memory Read Mutltiple",
1101:   "Dual Address Cycle",
1110:   "Memory Read Line",
1111:   "Memory Write And Invalidate"
```

10.6. Read/write function installed by PCI module

This function should be set by the PCI user module:

```
int (*acc)(struct esa_pci_input *ctrl, int cmd, unsigned int addr, unsigned int *data,
   unsigned int *abort, unsigned int *ws);
```

If set, the function is called by the AT697 AHB module whenever the PCI interface initiates a transaction. The function is called for AHB-slave mapped accesses as well as AHB-Master/APB DMA.The parameter *`cmd`* specifies the command to execute, see Section 10.5.1. Parameter *`addr`* specifies the address. The user module should return the read data in *`*data`* for a read command or write the *`*data`* on a write command and return the time to completion in *`*ws`* as PCI clocks. A possible target abort should be returned in *`*abort`*. The return value should be: 0: taken, 1: not taken (master abort)

10.7. Read/write function installed by AT697 module

The following function is installed by the AT697 AHB module:

```
int (*target_acc)(struct esa_pci_input *ctrl, int cmd, unsigned int addr, unsigned int
   *data, unsigned int *mexc);
```

The PCI user module can call this function to emulate a PCI target mode access to the AT697 AHB module. Parameter *`cmd`* specifies the command to execute, see Section 10.5.1. The AT697 module is supposed to be the host and accesses to the configuration space is not supported. Parameter `addr` specifies the address. Parameter `*data` should point to a memory location where to return the read data on a read command or point to the write data on a write command. Parameter `*mexc` should point to a memory location where to return a possible error. If the call was hit by MEMBAR0, MEMBAR1 or IOBAR, `target_read()` will return 1 otherwise 0.

10.8. Registers

Table 10.2 contains a list of implemented and not implemented fields of the AT697F PCI Registers. Only register fields that are relevant for the emulated PCI module is implemented.

Table 10.2. PCI register support

Register	Implemented	Not implemented
PCIID1	device id, vendor id	
PCISC	stat 13, stat 12, stat 11, stat 7, stat 6 stat 5, stat 4, com2, com 1, com1	stat15 stat14 stat10_9 stat8 com10 com9 com8 com7 com6 com5 com4 com3
PCIID2	class code, revision id	
PCIBHDLC	[bist, header type, latency timer, cache size] config-space only	
PCIMBAR1	base address, pref, type, msi	
PCIMBAR2	base address, pref, type, msi	
PCIIOBAR3	io base address, ms	
PCISID	subsystem id, svi	
PCICP	pointer	
PCILI	[max_lat min_gnt int_pin int_line] config-space-only	
PCIRT	[retry trdy] config-space-only	
PCICW		ben
PCISA	start address	
PCIIW		ben
PCIDMA	wdcnt, com	b2b
PCIIS	act, xff, xfe, rfe	dmas, ss
PCIIC	mod, commsb	dwr, dww, perr
PCITPA	tpa1, tpa2	
PCITSC		errmem, xff, xfe, rfe, tms
PCIITE	dmaer,imier, tier	cmfer, imper, tbeer, tper, syser
PCIITP	dmaer,imier, tier	cmfer, imper, tbeer, tper, syser
PCIITF	dmaer,imier, tier, cmfer, imper, tbeer, tper, syser	
PCID	dat	
PCIBE	dat	
PCIDMAA	addr	
PCIA		p0, p1, p2, p3

10.9. Debug flags

The switch `-designdbgon` flags can be used to enable debug output. The possible values for flags are as follows:

Table 10.3. Debug flags

ESAPCI_REGACC	Trace accesses to the PCI registers
ESAPCI_ACC	Trace accesses to the PCI AHB-slave address space
ESAPCI_DMA	Trace DMA
ESAPCI_IRQ	Trace PCI IRQ

10.10. Commands

pci

Displays all PCI registers.

11. Support

For support contact the Cobham Gaisler support team at support@gaisler.com.

12. Disclaimer

Aeroflex Gaisler AB, reserves the right to make changes to any products and services described herein at any time without notice. Consult Aeroflex or an authorized sales representative to verify that the information in this document is current before using this product. Aeroflex does not assume any responsibility or liability arising out of the application or use of any product or service described herein, except as expressly agreed to in writing by Aeroflex; nor does the purchase, lease, or use of a product or service from Aeroflex convey a license under any patent rights, copyrights, trademark rights, or any other of the intellectual rights of Aeroflex or of third parties.