GRSIM

A generic SPARC simulator capable of emulating LEON based systems

2017 User's Manual

The most important thing we build is trust

COBHAM

GRSIM Simulator User's Manual

COBHAM

Table of Contents

1. Introduction 4
 1.1. Supported platforms and system requirements 4
 1.2. Obtaining GRSIM 4
 1.3. Installation 4
 1.4. License installation 4
 1.5. Problem reports 4
2. Operation 5
 2.1. Operational modes 5
 2.2. Command line options 5
  2.2.1. Switches specific to certain modules 5
  2.2.2. (VENDOR_GAISLER) 5
  2.2.3. (VENDOR_ESA) 9
 2.3. Interactive Commands 10
 2.4. Simulator configuration 11
  2.4.1. Constructing a simulator configuration 11
 2.5. Running applications 13
 2.6. Inserting breakpoints and watchpoints 14
 2.7. Displaying registers 14
 2.8. Code coverage 14
 2.9. Symbolic debug information and profiling 15
 2.10. Displaying memory contents 16
 2.11. Disassembly of memory 16
 2.12. Loadable command module 17
 2.13. GDB interface 17
  2.13.1. Attaching to gdb 17
  2.13.2. Debugging of applications 18
  2.13.3. Detaching 18
  2.13.4. Specific GDB optimization 18
  2.13.5. Limitations of gdb interface 18
3. Simulator modules 19
 3.1. The basic structure 19
 3.2. Read and Write operations 20
  3.2.1. The parameters explained 20
  3.2.2. Ordinary accesses 20
  3.2.3. Diagnostic accesses 20
 3.3. Predefined modules 20
  3.3.1. (ESA_LEON2) LEON2 CPU 20
  3.3.2. (ESA_MCTRL) LEON2 memory controller 20
  3.3.3. (GAISLER_APBUART) GRLIB APBUART 21
  3.3.4. (GAISLER_ETHMAC) GRETH 10/100 Ethernet MAC module 21
  3.3.5. (GAISLER_GPTIMER) GRLIB APB General Purpose Timer 23
  3.3.6. (GAISLER_GRGPIO) General Purpose I/O Port 23
  3.3.7. (GAISLER_IRQMP) Multiprocessor Interrupt controller with AMP support 25
  3.3.8. (GAISLER_L2C) LEON2 Compatibility 26
  3.3.9. (GAISLER_L2IRQ) LEON2 Interrupt controller 26
  3.3.10. (GAISLER_L2TIME) LEON2 Timer 27
  3.3.11. (GAISLER_LEON3) LEON3 CPU 27
  3.3.12. (GAISLER_LEON4) LEON4 CPU 27
  3.3.13. (GAISLER_PCIFBRG) GRLIB GRPCI master/target interface 28
  3.3.14. (GAISLER_SDCTRL) GRLIB SDRAM Controller 30
  3.3.15. (GAISLER_SPW) GRSPW SpaceWire controller 30
  3.3.16. (GAISLER_SPW2) GRSPW2 SpaceWire controller 33
  3.3.17. (GAISLER_SRCTRL) GRLIB SRAM/PROM Controller 33
4. Library 34
 4.1. Function interface 34

4.2. Multi-threading 35
4.2.1. Limitations to multi-threading support 36
4.3. UART handling 36
4.4. Linking an application with the GRSIM library 36
4.5. GRSIM library without a simconf module 36
5. Support 37
A. HASP 38
A.1. Installing HASP Device Driver 38
A.1.1. On a Linux platform 38
B. Sample simulator configuration 40
C. Simple Example Amba Device 43
C.1. MemCtrl.h 43
C.2. MemCtrl.c 43
D. grcommon.h 46

1. Introduction

GRSIM is a generic SPARC[1] simulator capable of emulating LEON based systems.

GRSIM includes the following functions:

- Read/write access to all LEON registers and memory
- Built-in disassembler and trace buffer management
- Downloading and execution of LEON applications
- Breakpoint and watchpoint management
- Remote connection to GNU debugger (gdb)

1.1. Supported platforms and system requirements

GRSIM is currently provided for linux-2.2/glibc-2.3.

1.2. Obtaining GRSIM

The primary site for GRSIM is the Cobham Gaisler website [http://www.gaisler.com] where the latest version of GRSIM can be ordered.

1.3. Installation

GRSIM can be installed anywhere on the host computer - for convenience the installation directory should be added to the search path. The commercial versions use a HASP4 license key.

1.4. License installation

GRSIM 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 Aladdin website [http://www.aladdin.com] or the Cobham Gaisler website [http://www.gaisler.com]. See Appendix A, for installation instructions of device drivers.

1.5. Problem reports

Please send problem reports or comments to `<support@gaisler.com>`.

[1]SPARC is a registred trademark of SPARC International

2. Operation

This chapter describes how to use GRSIM.

2.1. Operational modes

GRSIM can operate in two modes: stand-alone and attached to gdb. In stand-alone mode, LEON applications can be loaded and debugged using a command line interface. A number of commands are available to examine data, insert breakpoints and advance execution, etc. When attached to gdb, GRSIM acts as a remote gdb target, and applications are loaded and debugged through gdb (or a gdb front-end such as DDD or Emacs GUD-mode).

2.2. Command line options

GRSIM is started as follows on a command line: `grsim [options]`

The following command line options are supported by GRSIM:

`-c` *`file`*
: Reads commands from `file` instead of stdin.

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

`-port` *`gdbport`*
: Set the port number for gdb communications. Default is 2222.

`-v`
: Turn on verbose mode. (debug 1)

`-vv`
: Turn on verbose mode. (debug 2)

`-simconf` *`file`*
: Use *file* as simulator configuration module, instead of the default *simconf.so*

`-simconfcfg` *`opts`*
: Pass *opts* as parameters to the simulator configuration module. E.g "-simconfcfg single_argument" or "-simconfcfg 'three arguments passed'".

`-nosimconf`
: Tell the simulator not to load a configuration module. This switch is only available in the library version of GRSIM.

`-sci` *`size`*
: Enable *size*kb instruction scratch pad memory.

`-scd` *`size`*
: Enable *size*kb data scratch pad memory.

`-ucmd` *`file`*
: Load a user command module. (See `'examples/UserCmd'` in the distribution for an example)

`-f` *`input_files`*
: Executable files to be loaded into memory. The input file is loaded into the target memory according to the entry point for each segment. Recognized formats are elf32 and S-record.

2.2.1. Switches specific to certain modules

Here follows switches that are specific to certain simulator models. If more than one instance of a model is used, it might be a good idea to pass the options via the configuration module, See Appendix C, to allow different parameters

2.2.2. (VENDOR_GAISLER)

Switches for modules in package VENDOR_GAISLER

2.2.2.1. (GAISLER_APBUART) GRLIB APB UART

`-uart` *`device`*
: By default, the UART is connected to stdin/stdout. In Linux, this switch can be used to connect the uart to another device. E.g. "-uart /dev/ptypc" will connect the UART to the pseudo device ptypc. (To communicate with ptypc, the user should connect to /dev/ttypc).

`-fast_uart`
: Run the uart at infinite speed, rather than the default (slow) baud rate.

`-uartlocalecho`
: This will echo all input characters locally when the terminal is connected to stdin/stdout.

`-uartfifo`
: Selects the size of the Receiver and Transmitter FIFOs. Valid values are 1, 2, 4, 8, 16, 32. Default is 1. Cannot be used together with “-fast_uart”.

`-sbits`
: Selects the number of bits in the scaler register. Valid values are 12-32.

2.2.2.2. (GAISLER_ETHMAC) GRLIB GRETH 10/100 Ethernet MAC

`-idx` *`i`*
: Set unique index for specific GRETH core. Needs to be set in simulation configuration file.

`-phyrstadr` *`i`*
: Set reset value of PHY address in MDIO Control/Status register mdio register. This also sets the bus that will repond to MDIO operations. Default 0.

2.2.2.3. (GAISLER_GPTIMER) GRLIB APB General Purpose Timer

`-pirq` *`num`*
: Irq number to use for timer interrupt. By default the amba configuration's irq is used.

`-sepirq`
: Use separate irq's for the *n* timers.

`-sbits`
: Number of bits for the scalar. Default is 16.

`-nbits`
: Number of bits for the counter. Default is 32.

`-ntimers`
: Number of timers (1-7). Default is 2.

2.2.2.4. (GAISLER_GRGPIO) General Purpose I/O Port

`-idx` *`nr`*
: The index among gpio cores

`-v`
: Turn on verbose output

`-nbits`
: Sets the GRGPIO *nbits* generic. Default is 16.

`-imask` *`mask`*
: Sets the GRGPIO *imask* generic. Default is 0.

`-irqgen`
: Sets the GRGPIO *irqgen* generic. Note that this generic is available from version 1 of the core and decides whether interrupt map registers are implemented or not. Default is 0 which is also how version 0 behaves.

`-driver` *`so/dll file symbol`*
: Loads an I/O driver from the given so/dll file, where the given symbol must be a pointer to a struct grgpio_iodriver.

`-output-zeroes-data`
: Sets quirk that a line that drives output will always read 0 for that line in the data register.

`-restricted-output` *`mask`*
: Sets quirk that only the lines for which the corresponding bit in the given mask is set can drive output. The other lines behaves as if direction is always set to in (even when the corresponding bit in the in the direction register is set to 1).

2.2.2.5. (GAISLER_IRQMP) Multiprocessor Interrupt controller options

`-broadcast`
: Enables the broadcast functionality.

`-amp` *cnt*
: Implement the AMP functionality of the IRQMP controller for *cnt* instances.

`-ext` *nr*
: Implement the extended irq controller functionality with extended interrupt *nr*.

`-cpubus` *busid*
: If the interrupt controller is located on a separate bus, specify the bus *busid* where the CPUs are located.

2.2.2.6. (GAISLER_L2C) LEON2 Compatibility, UART

`-uart` *device*
: By default, the UART is connected to stdin/stdout. In Linux, this switch can be used to connect the uart to another device. E.g. "-uart /dev/ptypc" will connect the UART to the pseudo device ptypc. (To communicate with ptypc, the user should connect to /dev/ttypc).

`-fast_uart`
: Run the uart at infinite speed, rather than the default (slow) baud rate.

2.2.2.7. (GAISLER_LEON3|4) LEON3|4 CPU

`-smpid` *num*
: The cpu's id that will appear in the cpu index field of %asr17.

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

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

`-dlock`
: Enable data cache line locking. Default is disabled.

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

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

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

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

`-isets` *sets*
: Defines the number of sets in the instruction cache. Allowed values are 1 - 4.

`-ilock`
: Enable instruction cache line locking.

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

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

`-ca` *cache area generic*
: Leon4 only: The value of the cached region vhdl-generic. Default: 0x10ff.

2.2.2.8. (GAISLER_SDCTRL) GRLIB SDRAM Controller

`-sdram` *size*
: Sets the amount of simulated SDRAM (Mbytes). Default is 16 Mbytes.

`-sdbanks` *num*
: Number of sdram banks. Default is 1.

2.2.2.9. (GAISLER_SPW) GRLIB GRSPW SpaceWire controller

`-idx` *i*
: Set unique index for specific GRSPW core. Needs to be set in simulation configuration file.

`-grspw1X_connect [ip address]`
: Same as the *grspw1*X*_connect* command. (See Section 3.3.15)

`-grspw1X_server [port]`
: Same as the *grspw1*X*_server* command. (See Section 3.3.15)

`-grspw1_normap`
: Disable hardware RMAP handler.

`-grspw1_rmap`
: Enable hardware RMAP handler.

`-grspw1_rmapcrc`
: Only enable hardware RMAP CRC calculations. (No hardware RMAP handler).

`-grspw1X_dbg` *flag*
: Turn on debug information. For a list of flags use the *grspw1X_dbg help* command, see Section 3.3.15.

2.2.2.10. (GAISLER_SPW2) GRLIB GRSPW2 SpaceWire controller

`-idx` *i*
: Set unique index for specific GRSPW2 core. Needs to be set in simulation configuration file.

`-grspwX_connect [ip address]`
: Same as the *grspw*X*_connect* command. (See Section 3.3.16)

`-grspwX_server [port]`
: Same as the *grspw*X*_server* command. (See Section 3.3.16)

`-grspwX_dbg` *flag*
: Turn on debug information. For a list of flags use the *grspwX_dbg help* command, see Section 3.3.16.

2.2.2.11. (GAISLER_SRCTRL) GRLIB SRAM/PROM Controller

`-sram` *size*
: Sets the amount of simulated RAM (kbyte). Default is 2048. 0 disables the sram.

`-prom` *size*
: Sets the amount of simulated PROM (kbyte). Default is 4096.

`-promstart` *address*
: Start address of prom. Default is bar 1 of the amba configuration. (Note that the amba bars have to cover the address)

`-sramstart` *address*
: Start address of sram. Default is bar 2 of the amba configuration or bar 2 if prom size is 0. (Note that the amba bars have to cover the address)

`-sramws` *ws*
: Sets the number of SRAM waitstates to *ws*. Default is 1.

`-promws` *size*
: Sets the number of PROM waitstates to *ws*. Default is 1.

2.2.2.12. (GAISLER_PCIFBRG) GRLIB GRPCI PCI master/target interface

`-pci_abits` *abits*
: Defines size of BAR0 PCI address space: 2^*abits*. Upper half accesses PAGE0 register. Lower half is translated to AHB accesses.

`-pci_dmaabits` *dmaabits*
: Defines size of BAR1 PCI address space: 2^*dmaabits*.

`-pci_nothost`
: If pci_nothost is specified the device is modeled as if not in a PCI host slot.

`-pci_blen` *bits*
: Number of bits in burst length register of PCIDMA core.

`-grpciip <so|dll>`
: Defines user defined PCI input provider. (See Section 3.3.13.2)

`-grpciiparg=<str>`
: Define argument to be given to the input providers grpci_inp_setup() callback on startup. -grpciiparg can be given multiple times.

2.2.3. (VENDOR_ESA)

Switches for modules in package VENDOR_ESA

2.2.3.1. (ESA_LEON2) Leon2 CPU

`-sci` *size*
: Enable instruction scratchpad RAM of *size* kb. Valid sizes are 1-64kb(in binary steps).

`-scd` *size*
: Enable data scratchpad RAM of *size* kb. Valid sizes are 1-64kb(in binary steps).

`-sci_base` *address*
: Set the base address of the instruction scratchpad RAM to *address*. (Defaults to 0x8e000000)

`-scd_base` *address*
: Set the base address of the data scratchpad RAM to *address*. (Defaults to 0x8f000000)

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

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

`-dlock`
: Enable data cache line locking. Default is disabled.

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

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

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

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

`-isets` *sets*
: Defines the number of sets in the instruction cache. Allowed values are 1 - 4.

`-ilock`
: Enable instruction cache line locking.

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

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

2.2.3.2. (ESA_MCTRL) Leon2 Memory Controller

`-onlyrom`
: Only allocate ROM area memory.

`-sram` *size*
: Sets the amount of simulated RAM (kbyte). Default is 4096.

`-rom` *size*
: Sets the amount of simulated ROM (kbyte). Default is 2048.

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

2.3. Interactive Commands

GRSIM dynamically loads libreadline.so if available on your host system, and uses readline() to enter or edit monitor commands. If libreadline.so is not found, fgets() is used instead (no history, poor editing capabilities and no tab-completion). Below is a description of those commands available, when used in stand-alone mode.

`batch`
: execute a batch file of grsim commands

`break`
: print or add breakpoint

`cont`
: continue execution

`cpu [<enable | disable | active> <num | all>]`
: Without parameters the 'cpu' command shows cpu status. The 'cpu active' command sets active cpu, i.e. the cpu to control from command line. The 'cpu enable/disable' command enables/disables one or all cpus.

`dcache`
: show data cache

`debug`
: change or show debug level

`delete`
: delete breakpoint(s)

`disassemble`
: disassemble memory

`echo`
: echo string in monitor window

`ep <addr>`
: set the entry point for the active cpu

`exit`
: see 'quit'

`float`
: display FPU registers

`gdb`
: connect to gdb debugger

`go [addr]`
: start execution without initialization

`hbreak`
: print breakpoints or add hardware breakpoint (if available)

`help`
: show available commands or usage for specific command

`xhelp`
: show debugging and status commands for specific cores

`icache`
: show instruction cache

`info <sys|libs|drivers|cpu|bus>`
: show information about the system.

`load`
: load a file

`mem`
: see 'x'(examine memory)

`vmem`
: like 'x'(examine memory) but with virtual address

`register`
: show/set integer registers

`reset`
: reset active GRSIM

`run`
: reset and start execution at last load address

`set <cpu <num>| bus <num>>`
: set which cpu/bus to control from command line

`shell`
: execute a shell command

`stack <addr>`
: set the stack pointer for the active cpu

`step`
: single step one or [n] times

`symbols`
: show symbols or load symbols from file

`profile <interval>`
: show and enables profiling with <interval> accuracy

`perf`
: Display execution statistics

`statistic`
: Display access statistics for cpu and bus

`quit`
: exit grsim

`version`
: show version

`watch`
: print or add watchpoints

`wmem`
: write word to memory

`x`
: examine memory

Typing a **`Ctrl-C`** will interrupt a running program. Short forms of the commands are allowed, e.g c, co, or con, are all interpreted as cont. Tab completion is available for commands, text-symbols and filenames.

2.4. Simulator configuration

Before starting the simulator, a configuration module has to be created. A configuration module defines which devices are present in the simulated system and where they appear in the address space. Appendix B, for a complete sample simulator configuration.

2.4.1. Constructing a simulator configuration

To construct a simulator configuration (referred to as *simconf* from now on), a `SimConf_T` structure has to be exported.

```
typedef struct simconf
{
   /* data provided by grsim */
   void         *sim;

   /* operations provided by grsim */
   int (*CreateBus)(void *sim);
   int (*AddLib    )(struct vendor_lib *lib);
   int (*AddDriver)(AmbaUnit_T *driver);
   int (*AddDevice)(void *simulator,
                    int busid,
                    unsigned int vendor,
                    unsigned int devid,
                    int irq, int numbars,
                    ...);
   int (*AddParams)(int idx, char **params, char *str);

   /* data */
   char **params;

   /* operations */
   int (*init)(char *str);
   int (*exit)(void);
   int (*cmd) (char  *cmd, void* arg);
```

```
    int (*ctrl)(int  ctl, void* arg);


    int (*SetMBApbMaster)(void *simulator, int busid, int apbid);
    int (*SetBusPnp)(void *simulator, int busid, unsigned int addr );
} SimConf_T;
```

where,

`sim`
: is a pointer to the current simulator instance.

`AddLib`
: is used to add a library to the built-in ones in grsim. This allows reuse of device driver ID.

`AddDriver`
: adds a driver to one of the existing libraries.

`CreateBus`
: creates a bus in the system. The return value from this function is the busid of the newly created bus. A bus must, of course, be created before any device is added to the system.

`AddDevice`
: adds a device to a bus in the simulated system. The first 6 parameters to AddDevice() are fixed. *numbars* designate the number of entries to follow. Each entry is headed by a tag followed by variable arguments depending on the tag. The possible tags are:

  `APB <start> <size> <iscache>`
  : Add a APB slave memory bar to the last added APB master.

  `MASTER`
  : Add a AHB master memory bar to the bus.

  `SLAVE <start> <size> <iscache>`
  : Add a AHB slave memory bar to the bus.

  `AHBIO <start> <size> <iscache>`
  : Add a AHB slave io bar to the bus.

  In addition there are tags that are used when configuring a multibus system:

  `SWITCHBUS <busid>`
  : Switch membar allocation target to bus with id <busid>. This is used when configuring a device with multiple interfaces to different buses.

  `SETVERSION`
  : Set the version field of all following entries. The default version is 0.

  `SETCUSTOM <idx> <val>`
  : Set the custom0-2 membar entry.

`AddParams`
: adds configuration parameters to the device with id `idx`. The value of idx for a specific device is the return value from the call to `AddDevice`, thus `AddDevice` must be called prior to `AddParams`. The params argument is the argument that the user supplied in simconf.params . The AddParams call will simply copy the argument of its call there.

`params`
: is a pointer to a pointer to the first string in an array of strings. (i.e. `char *foo[]`). These parameters are passed only to the specified device and allows for different configurations of multiple instances of the same device. E.g. the second of two UART instances might be passed *"-uart /dev/ptypc"* to avoid both uarts from using stdin/stdout.

`init`
: are called from the simulator at startup. Arguments can be passed to this function by the *-simconfcfg* command line option.

`exit`
: are called from the simulator when the simulator exits. It is not mandatory to provide this from the simconf module.

`cmd`
: are not used, but provided for future extensions of the simconf module.

`ctrl`
: are not used, but provided for future extensions of the simconf module.

`SetMBApbMaster`
: for a bus with multiple APB bridges specify the APB bridge that the next APB-device added with AddDevice should associate with. The parameter apbid is the return value of the call to AddDevice when allocating the APB bridge. The default APB bridge is the first one added.

`SetBusPnp`
: Currently not used.

For a more complete example, see the simulator configuration module that is supplied with the GRSIM release. Also, see Appendix B, for detailed code listing.

2.5. Running applications

To run a program, first use the load command to download the application and then run to start it.

```
john@venus% ./grsim.exe

 GRSIM LEON MP Simulator v1.1.39 professional version

 Copyright (C) 2004-2009 Aeroflex Gaisler - all rights reserved.
 For latest updates, go to http://www.gaisler.com/
 Comments or bug-reports to support@@gaisler.com

 Creating a simulator instance.
 Loading Config Module,"simconf.so"
 Loaded simconf.so

added library "Test Vendor"
added driver "Simple EXAMPLE Mem Controller"

 Initialising...
icache: 1 * 4 kbytes, 16 bytes/line (4 kbytes total)
dcache: 1 * 4 kbytes, 16 bytes/line (4 kbytes total)
allocated 4096 K RAM memory, in 1 bank(s)
allocated 2048 K ROM memory
Initialising the Interrupt controller...
Initializing UART
serial port on stdin/stdout


 Component                             Vendor
 Leon2 SPARC V8 Processor              ESA
 Leon 2 Memory Controller              ESA
 Leon2 compat module                   Gaisler Research
 Leon2 compat module                   Gaisler Research
 Leon2 compat module                   Gaisler Research
 AHB/APB Bridge                        Gaisler Research
 Leon2 Timer and Watchdog module       Gaisler Research
 Leon2 IRQ Controller                  Gaisler Research
 APB UART                              Gaisler Research

grsim> load samples/stanford.prom
total size: 28048 bytes (in <1 sec)
read 37 symbols
entry point: 0x00000000
grsim> run


  MkProm LEON boot loader v1.2
  Copyright Gaisler Research - all right reserved

  system clock   : 50.0 MHz
  baud rate      : 19171 baud
  prom           : 512 K, (2/2) ws (r/w)
  sram           : 2048 K, 1 bank(s), 0/0 ws (r/w)
  edac           : disabled

  decompressing .text
  decompressing .data

  starting stanford

Starting
    Perm  Towers  Queens    Intmm      Mm  Puzzle   Quick  Bubble    Tree     FFT
      50      50      33       67      33     184      33      33     317      50

Nonfloating point composite is        137

Floating point composite is        168
```

```
Program exited normally.
```

2.6. Inserting breakpoints and watchpoints

GRSIM supports execution breakpoints and write data watchpoints. In stand-alone mode, hard- ware 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. Data write watch- points are inserted using the 'watch' command. A watchpoint can only cover one word address, block watchpoints are not available.

2.7. Displaying registers

The current register window can be displayed using the *register* command:

```
grsim> register

          INS        DATA       OUTS       GLOBALS
   0:   00000027   40013000   00000027   00000000
   1:   40650E97   40001024   4000A470   00000001
   2:   9999999A   40001028   401FDF00   0000000A
   3:   0000029D   40013344   00000000   401FD687
   4:   00000770   00000000   40650E97   4000A800
   5:   401FDF30   00000000   9999999A   00000770
   6:   401FDEB8   00000000   401FDE50   00000001
   7:   40001208   00000000   40004E08   00000000

 psr: 004010C5   wim: 00000080   tbr: 40000800   y: 01800000

 pc:  40000800  91d02000   ta  0x0
 npc: 40000804  01000000   nop
```

Other register windows can be displayed using reg n, when n denotes the window number. Use the float command to show the FPU registers (if present).

2.8. Code coverage

The normal GRSIM doesn't include code coverage. There is a special GRSIM binary marked with a **-coverage** suffix. This binary implements the coverage commands similar to those present in TSIM. The GRSIM version with suffix **-coverage** runs slower due to the overhead imposed. Coverage is implemented for LEON3 and LEON4 processor cores and is done on a per-cpu basis. 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 [cpunum] enable`
: enable coverage

`coverage [cpunum] disable`
: disable coverage

`coverage [cpunum] save [filename]`
: write coverage data to file (file name optional)

`coverage [cpunum] print address [len]`
: print coverage data to console, starting at address

`coverage [cpunum] 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:

```
grsim> coverage 0 print start
40000000 : 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
40000080 : 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
```

```
40000100 : 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
40000180 : 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

When the code coverage is saved to file, only blocks with at least one coverage field set are written to the file. Block that have all the coverage fields set to zero are not saved in order to decrease the file size. Note that all memory (prom and ram) are subject for code coverage. Coverage also supports virtual addresses.

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:

```
grsim> dis 0
 00000000  0x88100000 1  clr  %g4
 00000004  0x0900003a 1  sethi  %hi(0xe800), %g4
 00000008  0x81c12098 1  jmp  %g4 + 0x98
 0000000c  0x01000000 1  nop
 00000010  0xa1480000 0  mov  %psr, %l0
 00000014  0xa7500000 0  mov  %wim, %l3
 00000018  0x10803a1a 0  ba  0x0000e880
 0000001c  0xac102001 0  mov  1, %l6
```

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

2.9. Symbolic debug information and profiling

GRSIM will automatically extract (.text) symbol information from elf-files. It is also possible to read symbols from an alternative (elf) file, which is very useful when debugging self extracting applications, such as those created by mkprom.

The symbols can be used wherever an address is expected.:

```
grsim> load samples/stanford.prom
total size: 28048 bytes (in <1 sec)
read 37 symbols
entry point: 0x00000000
grsim> break main
breakpoint 1 at 0x000067f8:  main
grsim> symbols samples/stanford
read 195 symbols
entry point: 0x40000000
grsim> break main
breakpoint 2 at 0x40001ac8:  main
grsim>
```

The symbols command can also be used to display all currently loaded symbols:

```
grsim> symbols samples/hello
read 71 symbols
grsim> symbols
0x40000000 L _trap_table
0x40000000 L start
0x4000102c L _window_overflow
0x40001084 L _window_underflow
0x400010dc L _fpdis
0x400011a4 T _flush_windows
0x400011a4 T _start
0x40001218 L fstat
0x40001220 L isatty
0x40001228 L getpid
0x40001230 L kill
0x40001238 L _exit
0x40001244 L lseek
...
```

The `profile` command can be used to show usage of program functions. Enabling profile is done by `profile <interval>` where <interval> is the interval time in clock cycles for which a probe should reoccur. On each profiling interval event the callstack is used to update a execution count for each symbol present. Recursive functions may therefore cause expected though correct results.

```
grsim> profile 100
Profiling enabled: interval 100
grsim> run
Starting
    Perm  Towers  Queens   Intmm      Mm
...
grsim> profile
function            samples     ratio(%),
 tower                691022      27.93
 Permute              269377      10.88
 _hardreset_real      256988      10.38
 _start               256986      10.38
 main                 256265      10.35
 Try                  185961       7.51
 Intmm                 73309       2.96
 Innerproduct          66984       2.70
 Towers                53463       2.16
 Perm                  48513       1.96
 Mm                    47115       1.90
 Move                  45831       1.85
 rInnerproduct         38796       1.56
 Queens                33579       1.35
 Doit                  33551       1.35
...
```

2.10. Displaying memory contents

Any memory location can be displayed using the x command. If a third argument is provided, that is interpreted as the number of bytes to display. Text symbols can be used instead of a numeric address. If the mmu is switched on the vmem command can be used to translate the address first. The bus on which to issue the access has to be set using the *set bus <idx>* command.

```
grsim> x 0x40000000

 40000000  a0100000  29100004  81c52000  01000000    .......). ......
 40000010  91d02000  01000000  01000000  01000000    . .............
 40000020  91d02000  01000000  01000000  01000000    . .............
 40000030  91d02000  01000000  01000000  01000000    . .............

grsim> x 0x40000000 32

 40000000  a0100000  29100004  81c52000  01000000    .......). ......
 40000010  91d02000  01000000  01000000  01000000    . .............

grsim> x main

 40001AC8  9de3bf88  7fffffb1  9007bfe8  d407bfe8    ................
 40001AD8  92102003  912aa001  9002000a  912a2003    . ....*...... *.
 40001AE8  9002000a  40000913  912a2001  81c7e008    .......@. *.....
 40001AF8  91e80008  13000049  92126003  1510006b    ....I....`..k...
```

2.11. Disassembly of memory

Any memory location can be disassembled using the disassemble command. As with the examine command, text symbols can be used as an address. Also, a region can be disassembled.

```
grsim> disassemble 0x40000000 5
 40000000  a0100000   clr  %l0
 40000004  29100004   sethi  %hi(0x40001000), %l4
 40000008  81c52000   jmp  %l4
 4000000c  01000000   nop
 40000010  91d02000   ta  0x0
grsim> disassemble main 3
 40001ac8  1110000e   sethi     %hi(0x40003800), %o0
 40001acc  90122010   or        %o0, 0x10, %o0
 40001ad0  8213c000   or        %o7, %g1
grsim> disassemble 0x40000000 0x4000000c
 40000000  a0100000   clr  %l0
 40000004  29100004   sethi  %hi(0x40001000), %l4
 40000008  81c52000   jmp  %l4
 4000000c  01000000   nop
```

COBHAM

2.12. Loadable command module

It is possible for the user to add commands to grsim by creating a loadable command module. The module should export a pointer to a UserCmd_T called UserCommands, e.g.:

```
UserCmd_T *UserCommands = &CommandExtension;
```

UserCmd_T is defined as:

```
typedef struct
{
   LibIf_T *lib;

/* Functions exported by grmon */
   int  (*MemoryRead  )(LibIf_T *lib, unsigned int addr,
                        unsigned char *data, unsigned int length);
   int  (*MemoryWrite )(LibIf_T *lib, unsigned int addr,
                        unsigned char *data, unsigned int length);
   void (*GetRegisters)(LibIf_T *lib, unsigned int registers[]);
   void (*SetRegisters)(LibIf_T *lib, unsigned int registers[]);
   void (*dprint)(char *string);

/* Functions provided by user */
   int  (*Init)();
   int  (*Exit)();
   int  (*CommandParser)(int argc, char *argv[]);
   char **Commands;
   int    NumCommands;
} UserCmd_T;
```

The first five entries is function pointers that are provided by grsim when loading the module. The other entries has to be implemented by the user. This is how:

- Init and Exit are called when entering and leaving a grsim target.
- CommandParser are called from grsim before any internal parsing is done. This means that you can override internal grsim commands. On success CommandParser should return 0 and on error the return value should be > 200. On error grsim will print out the error number for diagnostics. argv[0] is the command itself and argc is the number of tokens, including the command, that is supplied.
- Commands should be a list of available commands. (used for command completion)
- NumCommands should be the number of entries in Commands. It is crucial that this number matches the number of entries in Commands. If NumCommands is set to 0(zero), no command completion will be done.

A simple example of a command module is supplied with the professional version of GRSIM.

2.13. GDB interface

This section describes how to use gdb with GRSIM.

2.13.1. Attaching to gdb

GRSIM can act as a remote target for gdb, allowing symbolic debugging of target applications. To initiate gdb communications, start the monitor with the -gdb switch or use the GRSIM gdb command:

```
grsim> gdb
gdb interface: using port 2222
```

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

```
(gdb) target extended-remote pluto:2222
Remote debugging using pluto:2222
0x40000800 in start ()
(gdb)
```

While attached, normal GRSIM commands can be executed using the gdb monitor command. Output from the GRSIM commands, such as the register information is then displayed in the gdb console:

```
(gdb) monitor reg

          INS        DATA       OUTS       GLOBALS
   0:  00000004   00000000   00000008   00000000
   1:  400155FC   40001DD4   00000001   40022000
   2:  40015694   40001DD8   00000004   00000073
   3:  00000030   00000010   00000005   00000000
   4:  4001562C   00000800   40015624   4000A8F0
   5:  00000007   00000080   40015694   00000770
   6:  401FD7C8   00000000   401FD760   00000001
   7:  40001E3C   40022000   400155FC   00000000

 psr: 000010E4   wim: 00000004   tbr: 40000050   y: 00000000

 pc:  40001e1c  d4232004   st   %o2, [%o4 + 0x4]
 npc: 40001e20  10800004   ba   0x40001e30
(gdb)
```

2.13.2. Debugging of applications

To load and start an application, use the gdb load and run command.

```
(gdb) lo
Loading section .text, size 0xcb90 lma 0x40000000
Loading section .data, size 0x770 lma 0x4000cb90
Start address 0x40000000, load size 54016
Transfer rate: 61732 bits/sec, 278 bytes/write.
(gdb) bre main
Breakpoint 1 at 0x400039c4: file stanford.c, line 1033.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/john/samples/stanford

Breakpoint 1, main () at stanford.c:1033
1033        fixed = 0.0;
(gdb)
```

To interrupt simulation, `Ctrl-C` can be typed in both GDB and GRSIM windows. The program can be restarted using the GDB run command but a load has first to be executed to reload the program image on the target. Software trap 1 (ta 1) is used by gdb to insert breakpoints and should not be used by the application.

2.13.3. Detaching

If gdb is detached using the detach command, the monitor returns to the command prompt, and the program can be debugged using the standard GRSIM commands. The monitor can also be re-attached to gdb by issuing the gdb command to the monitor (and the target command to gdb).

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

2.13.4. Specific GDB optimization

GRSIM detects gdb access to register window frames in memory which are not yet flushed and only reside in the processor register file. When such a memory location is read, GRSIM will read the correct value from the register file instead of the memory. This allows gdb to form a function traceback without any (intrusive) modification of memory. This feature is disabled during debugging of code where traps are disabled, since no valid stack frame exist at that point.

GRSIM detects the insertion of gdb breakpoints, in form of the `ta 1' instruction. When a breakpoint is inserted, the corresponding instruction cache tag is examined, and if the memory location was cached the tag is cleared to keep memory and cache synchronized.

2.13.5. Limitations of gdb interface

Do not use the gdb command, *"where"*, in parts of an application where traps are disabled (e.g.trap handlers). Since the stack pointer is not valid at this point, gdb might go into an infinite loop trying to unwind false stack frames.

3. Simulator modules

This describes how to write you own simulation model of an AHB slave or APB slave to use in grsim.

3.1. The basic structure

The simulator interface for attaching a device in the simulator is as follows:

```
typedef struct amba_unit
{
   /* identification data */
   short  vendor;
   short  version;
   short  device;
   char   desc[32];

   /* functions */
   int (*read)      (struct ahb_dev_rec *me,
                     struct ahb_dev_rec *master,
                     unsigned int address,
                     unsigned int *data,
                     unsigned int length,
                     unsigned int wsize);
   int (*write)     (struct ahb_dev_rec *me,
                     struct ahb_dev_rec *master,
                     unsigned int address,
                     unsigned int *data,
                     unsigned int length,
                     unsigned int wsize);
   int (*read_done) (AccRes_T *result);
   int (*write_done)(AccRes_T *result);
   int (*ctrl)      (int ctl, struct ahb_dev_rec *me, void *args);
   int (*cmd)       (char *cmd, struct ahb_dev_rec *me);
} AmbaUnit_T;
```

This is the structure that should be exported by each device module.

`vendor`
: The vendor ID of the device.

`version`
: The version of the device.

`device`
: The device ID of the device.

`read`
: The function that is called when a read access is made to the device.

`write`
: The function that is called when a write access is made to the device.

`read_done`
`write_done`
: Not used by AHB slaves or APB slaves.

`ctrl`
: This is the generic control function. It is via this function that grsim communicates with the device. The minimum set of commands that a device has to implement are:

- GRDRV_INIT, is called during simulator initialisation.Read/write access to all LEON registers and memory
- GRDRV_EXIT, is called when the simulator exits. Here is a good place to free memory that has been allocated by the device.Built-in disassembler and trace buffer management
- GRDRV_RESTART, is called during simulator reset.Downloading and execution of LEON applications
- GRDRV_OPTIONS, is actually not required, but it is strongly advisable to at least implement an empty handler for this command to easily beeing able to send configuration parameters to the device.

`cmd`
: This function is called whenever GRSIM doesn't recognize a command as a internal command. E.g. if *run* is issued, cmd will not be called because *run* is an internal grsim command. But, if *my_nice_cmd* is issued, cmd will be called.

3.2. Read and Write operations

There are a couple of implementations constraints when implementing the read and write operations which will be described below.

3.2.1. The parameters explained

`me`
: a pointer to this device's instantiation structure. This can be thought of as *'this'* in C++.

`master`
: a pointer to the callee's device structure.

`address`
: the address to which the access is made. The simulator guaranties that this address always is within the region that the device was attached in the simulator configuration. See Appendix B.

`data`
: a pointer to the data which is read from/written to.

`length`
: the length of the transaction in *wsize* units.

`wsize`
: the wordsize in the transaction:

wsize	word size [bytes]
0	1
1	2
2	4
3	8

3.2.2. Ordinary accesses

Apart from filling in/reading out the requested data, the slave must also finish the transaction by a call to `Grsim_AmbaReadDone`(or `Grsim_AmbaWriteDone` if it was a write).

3.2.3. Diagnostic accesses

The simulator uses the read and write operations for diagnostic accesses, which should not interfere with the simulation. This is done by setting the master to DIAG, see Appendix D. When the slave detects that the master is set to DIAG, it should **not** call `Grsim_AmbaReadDone`(or `Grsim_AmbaWriteDone` if it was a write).

3.3. Predefined modules

This chapter describes the predefined modules available with GRSIM.

3.3.1. (ESA_LEON2) LEON2 CPU

The LEON2 cpu module (vendorid:VENDOR_ESA, deviceid: ESA_LEON2) simulates a LEON2 processor. The configuration module example below shows how to instantiate the module.

```
//leon2:
i = simcfg.AddDevice(simcfg.sim,
                     busid,
                     VENDOR_ESA,
                     ESA_LEON2,
                     0,
                     0);
simcfg.AddParams(i, simcfg.params, "cpu params");
```

3.3.2. (ESA_MCTRL) LEON2 memory controller

The LEON2 memory controller module (vendorid:VENDOR_ESA, deviceid: ESA_MCTRL) simulates rom/sram and sdram. The configuration module example below shows how to instantiate the module. Four bars with range

0x00000000-0x10000000,0x20000000-0x30000000,0x40000000-0x70000000 and 0xc0000000-0xc0000100 are allocated. Inside 0x40000000-0x70000000 there will be 16 mb of sram at 0x4000000-0x4100000 and 16 mb of sdram ar 0x6000000-0x6100000. Inside 0x00000000-0x10000000 there will be 4mb of prom (default) at 0x00000000-0x00400000. The memory controller's registers will appear at 0xc0000000 onward.

```
//mem ctrl:
i = simcfg.AddDevice(simcfg.sim,
                     busid,
                     VENDOR_ESA,
                     ESA_MCTRL,
                     1,
                     4,
                     SLAVE,
                     0x00000000,
                     0x10000000,
                     1,
                     SLAVE,
                     0x20000000,
                     0x10000000,
                     1,
                     SLAVE,
                     0x40000000,
                     0x30000000,
                     1,
                     APB,
                     0xc0000000,
                     0x00000100,
                     0);
simcfg.AddParams(i, simcfg.params, "-sram 16384 -sdram 16");
```

3.3.3. (GAISLER_APBUART) GRLIB APBUART

The GRLIB APB UART (vendorid:VENDOR_GAISLER, deviceid: GAISLER_APBUART) simulates one uart. The configuration module example below shows how to instantiate the module. A uart who's registers are accessible from address 0xc0000100 onward is created.

```
//uart:
i = simcfg.AddDevice(simcfg.sim,
                     busid,
                     VENDOR_GAISLER,
                     GAISLER_APBUART,
                     3,
                     1,
                     APB,
                     0xc0000100,
                     0x00000100,
                     0);
simcfg.AddParams(i, simcfg.params, "-fast_uart");
/*  -uart /dev/ptypc");  */
```

3.3.4. (GAISLER_ETHMAC) GRETH 10/100 Ethernet MAC module

The Ethermac module (vendorid: VENDOR_GAISLER, deviceid: GAISLER_ETHMAC) simulates the GRLIB GRETH 10/100 Ethernet MAC. The simulation model delivers and receives packets through a TCP socket. An example packet server is delivered with GRSIM which uses the tun/tap interface in Linux to attach to a real Ethernet network.

The configuration module example below shows how to instantiate the module.

```
//ethermac
i = simcfg.AddDevice(simcfg.sim,
                 busid,
                 VENDOR_GAISLER,
                 GAISLER_ETHMAC,
                 12,
                 1,
                 APB,
                 0x80000800,
                 0x00000100,
                 0);
```

3.3.4.1. Commands

Commands for this module:

`grethX_status`
: Print out he status of the buffer descriptors and the registers.

`grethX_connect [ip address]`
: Try to connect to the packet server at address `ip address`, if `ip address` is omitted localhost is used. TCP port 2224 is used. The packet server must have been started in another shell or on another linux box.

`grethX_dbg [<flags>|clean|list|help]`
: Toggle debug output options. Do *grethX_dbg help* for a list of possible options. *grethX_dbg clean* will deactivate all debug output and *grethX_dbg list* will list the current settings.

`grethX_dump [file]`
: Dump packets to Ethereal readable `file`. When a file is not specified the current dumpfile will be closed.

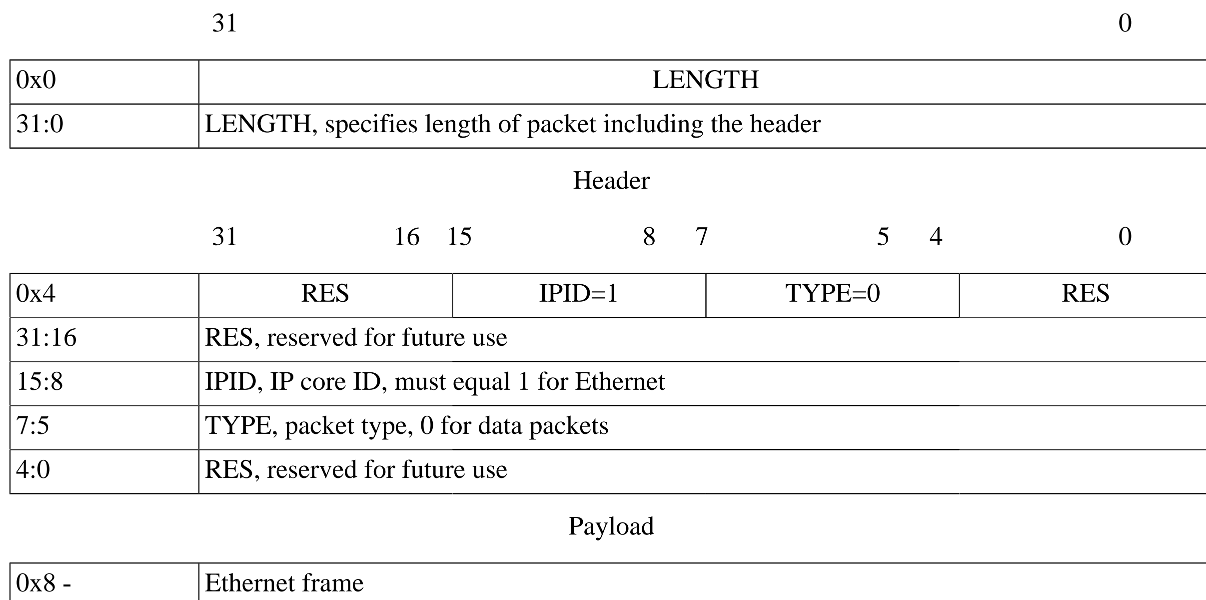
`grethX_ping [ip address]`
: Simulate a ping transmission. Packets will be generated by GRSIM. If `ip address` is not specified the default is 192.168.0.80.

3.3.4.2. Packet server

A example packet server is delivered with GRSIM (`greth_config`). To start the packet server `greth_config` you have to be root. It uses 192.168.0.81 as the tap0 interface's address by default. To specify a different interface ip number use the -ipif <addr> switch. The ipaddress of the operating system running in GRSIM is default assumed to be 192.168.0.80 but can be changed by the -ip <addr> switch. The packet server tries to configure the ip stack by issuing the following commands:

```
echo 1 > /proc/sys/net/ipv4/ip_forward
echo 1 > /proc/sys/net/ipv4/conf/tap0/proxy_arp
route add -host 192.168.0.80 dev tap0
```

Where 192.168.0.80 is the ip address of the interface of the operating system running in the simulator or <num> if -ip <num> was given. `greth_config` tries to issue the commands at startup. In case they can't be executed issue them by hand. When running the server you can enter 1-3 to specify the debugging level. For 3 each received/transmitted packet will be dumped on the screen. The packet format used by the GRETH module is shown below. 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 3.1. Ethernet data packet

3.3.5. (GAISLER_GPTIMER) GRLIB APB General Purpose Timer

The GRLIB APB General purpose Timer (vendorid:VENDOR_GAISLER, deviceid: GAISLER_GPTIMER) simulates a multipurpose interrupt timer. One instance can simulate multiple timers which can eather issue the same interrupt or seperate interrupts. The configuration module example below shows how to instantiate the module. A general purpose Timer with 2 timers (default) using the same interrupt is created. The irq number is that of the amba configuration, 8.

```
//timer:
i = simcfg.AddDevice(simcfg.sim,
                     busid,
                     VENDOR_GAISLER,
                     GAISLER_GPTIMER,
                     8,                //irqnr
                     1,
                     APB,
                     0xc0000300,       //bar1
                     0x00000100,       //bar1
                     0);
simcfg.AddParams(i, simcfg.params, "");
```

3.3.6. (GAISLER_GRGPIO) General Purpose I/O Port

The GRGPIO general purpose I/O port (vendorid:VENDOR_GAISLER, deviceid: GAISLER_GRGPIO) simulates all registers of the GRLIB GRGPIO core except for the bypass register. Note that the available registers depends on the version of the core (capabilities register from version 2) and the generics (interrupt map registers availability depends on -irqgen) that are being used to instantiate it. Section 2.2.2.4 for instantiation options. A user defined I/O driver module is connected to the GRGPIO module to simulate the I/O environment of the core. The configuration module example below shows how to instantiate the module.

```
i = simcfg.AddDevice(simcfg.sim,
                     busid,
                     VENDOR_GAISLER,
                     GAISLER_GRGPIO,
                     0,
                     1,
                     APB,
                     0x80000900,
                     0x00000100,
                     0);
simcfg.AddParams(i, simcfg.params,"-idx 0 -nbits 32 -imask 0xffff");
```

3.3.6.1. Commands

Commands for this module (where the `X` in `gpioX` should be replaced by the index of the core to work with):

`gpioX_driver <so/dll file> <symbol>`
: Loads an I/O driver from the given so/dll file, where the given symbol must be a pointer to a struct grgpio_iodriver.

`gpioX_status`
: Print out the status of registers and I/O provider.

`gpioX_input <value>`
: Set default input value do be used when there is no connected I/O driver.

`gpioX_dbg [<flags>|clean|list|help]`
: Toggle debug output options. Do *grethX_dbg help* for a list of possible options. *grethX_dbg clean* will deactivate all debug output and *grethX_dbg list* will list the current settings.

`gpioX_verbose [0|1]`
: Toggle or set verbose output of module.

3.3.6.2. I/O driver interface

The user should supply a dynamic library that exposes a symbol of type `struct grgpio_iodriver *`. The `struct grgpio_iodriver` is the overall interface between a GRGPIO I/O and GRSIM. It is defined in *grgpio_iodriver.h* as:

```
struct grgpio_iodriver {

        /* See grgpio_core_iodriver.h */
        struct grgpio_core_iodriver core_driver;

        /* ===== Initialized by I/O driver ===== */

        /*
         * Called by GRSIM once at setup.
         * Should return 0 on success and non-zero otherwise.
         */
        int (*setup)(struct grgpio_iodriver* driver);

        /* Called by GRSIM on reset */
        void (*reset)(struct grgpio_iodriver* driver);

        /* Called by GRSIM for tearing down the I/O driver */
        void (*exit)(struct grgpio_iodriver* driver);


        /* ===== Initialized by GRSIM ===== */

        /* I/O driver interface for interfacing with grsim */
        struct iodriverif *iodif;

        /* Index, among grgpio cores, of the core */
        int idx;

        /* Name of grgpio  */
        const char *name;
};
```

The `struct grgpio_core_iodriver` is the interface between a GRGPIO I/O driver and the GRGPIO core logic. It is defined in *grgpio_core_iodriver.h* as:

```
struct grgpio_core_iodriver {
    struct grgpio_core *core; /* Opaque pointer for the grgpio core */

    void *ip_data; /* Pointer for use by the I/O driver */

    /*
     * Gets called by the grgpio core whenever a direction is changed or if an
     * output value (for a pin in the out direction) is changed.
     *
     * Bit x of dir indicates that the grgpio core drives output on line x when
     * 1 and that it does not when it is 0.
     *
     * Function must be initialized by I/O driver.
     */
    void (*output_change)(struct grgpio_core_iodriver *driver,
                          uint32 dir, uint32 output);

    /*
     * Must be called by I/O driver whenever the input to the core changes. May
     * be called by I/O driver even if input data has not changed since last
     * call.
     *
     * Note that is up to the I/O driver to call or arrange for a call of this
     * function to change input value even for lines for which the grgpio core
     * is driving an output value.
     *
     * Function gets initialized by grgpio core.
     */
    void (*set_input)(struct grgpio_core *core, uint32 input);

    /*
     * Can optionally be set by I/O driver. If set, should print I/O driver
     * status.
     */
    void (*print_status)(struct grgpio_core_iodriver *driver);
};
```

The `struct iodriverif` contains functions that an I/O driver can use to interface with GRSIM. It is defined in *iodriver.h* as:

```
struct iodriverif {
    /*
     * Pointer to simulation instance to use as an argument in the functions
     * below
```

```
     */
    void *sim;

    /*
     * Adds an event to the event queue.
     * Returns 0 on success, non-zero on failure.
     */
    int (*add_event)(void *sim, void(*handler)(void *), void *arg, SimTime_T offset);

    /*
     * Stops all event that matches the given handler.
     * Returns the number of stopped events.
     */
    int (*stop_event)(void *sim, void(*handler)(void *));

    /*
     * Stops all event that matches both the given handler and arg.
     * Returns the number of stopped events.
     */
    int (*stop_event_arg)(void *sim, void(*handler)(void *), void *arg);

    /*
     * Returns non-zero in an event exists in the event queue that matches both
     * the given handler and arg.
     */
    int (*event_exists)(void *sim, void(*handler)(void *), void *arg);

    /*
     * Returns the current simulation time.
     */
    SimTime_T (*get_time)(void *sim);

    /*
     * Stops the simulation after current event is finished, but before the next
     * event.
     */
    void (*stop_simulation)(void *sim);

    /*
     * Prints both to stdout and, if logging, to logfile.  Works like printf
     * apart from the sim arugment.
     */
    int (*simprintf)(void *sim, char *format, ...);
};
```

Typically an I/O driver sets up a `struct grgpio_iodriver` initializing the setup, reset and exit functions that are called by GRSIM when the module is loaded, at reset and when the module is unloaded. For the `struct grgpio_iodriver core_driver` field, the `output_change` function pointer must be initialized by the I/O driver with a function that deals with changes of the output from the core, print_status function pointer is typically initialized with a function that prints status information on the I/O driver and the ip_data pointer is typically set to point to some user defined data structure that is used by the I/O driver.

Between the call to the setup function and the exit function of a `struct grgpio_iodriver`, the idx and the name can be read and `core_driver.set_input` and all the functions in `iodif` can be called.

Note that when an GRGPIO I/O driver is loaded, the readout of the data register is not automatically updated due to writes to the output or direction registers. It is up to the GRGPIO I/O driver to arrange for a call of `core_driver.set_input` (by calling it directly or by adding an event that will eventually call it) to have the input, as viewed from the GRGPIO core, change at the appropriate time.

For an example GRGPIO I/O driver see the example in the distributed *grgpio_iodriver_example.c* file.

3.3.7. (GAISLER_IRQMP) Multiprocessor Interrupt controller with AMP support

The IRQMP Interrupt controller (vendorid:VENDOR_GAISLER, deviceid: GAISLER_IRQMP) simulates a multiprocessor processor interrupt controller. The configuration module example below shows how to instantiate the module.

```
//irq:
i = simcfg.AddDevice(simcfg.sim,
                     BUS1,
                     VENDOR_GAISLER,
                     GAISLER_IRQMP,
```

```
                      10,
                      1,
                      APB,
                      0xf0004000,
                      0x00004000,
                      0);
simcfg.AddParams(i, simcfg.params, "-broadcast -amp 4 -ext 10 -cpubus 0 ");
```

3.3.8. (GAISLER_L2C) LEON2 Compatibility

The LEON2 Compatibility module (vendorid:VENDOR_GAISLER, deviceid: GAISLER_L2C) simulates all LEON2 registers from 0x800000000 onward except of the interrupt controller which should be implemented using the LEON2 Interrupt controller module (GAISLER_L2IRQ), the timer module which should be implemented using the LEON2 Timer controller module (GAISLER_L2TIME) and the uart which should be implemented using the apb uart module (GAISLER_APBUART). The configuration module example below shows how to instantiate the module.

```
//leon2 compat
i = simcfg.AddDevice(simcfg.sim,
                      busid,
                      VENDOR_GAISLER,
                      GAISLER_L2C,
                      3,
                      1,
                      APB,
                      0x80000010,
                      0x00000030,
                      0);

//leon2 compat
i = simcfg.AddDevice(simcfg.sim,
                      busid,
                      VENDOR_GAISLER,
                      GAISLER_L2C,
                      3,
                      1,
                      APB,
                      0x80000068,
                      0x00000008,
                      0);

//leon2 compat
i = simcfg.AddDevice(simcfg.sim,
                      busid,
                      VENDOR_GAISLER,
                      GAISLER_L2C,
                      3,
                      1,
                      APB,
                      0x800000a0,
                      0x00000040,
                      0);
simcfg.AddParams(i, simcfg.params, "");
```

3.3.9. (GAISLER_L2IRQ) LEON2 Interrupt controller

The LEON2 Interrupt controller (vendorid:VENDOR_GAISLER, deviceid: GAISLER_L2IRQ) simulates a LEON2 single processor interrupt controller. As for the default LEON2 register layout it should be allocated at address 0x80000090. The configuration module example below shows how to instantiate the module.

```
//irq:
i = simcfg.AddDevice(simcfg.sim,
                      busid,
                      VENDOR_GAISLER,
                      GAISLER_L2IRQ,
                      3,
                      1,
                      APB,
                      0x80000090,
                      0x00000010,
                      0);
simcfg.AddParams(i, simcfg.params, "");
```

3.3.10. (GAISLER_L2TIME) LEON2 Timer

The Leon2 Timer module (vendorid:VENDOR_GAISLER, deviceid: GAISLER_L2TIME) simulates a Leon2 timer. The configuration module example below shows how to instantiate the module.

```
//timer:
i = simcfg.AddDevice(simcfg.sim,
                     busid,
                     VENDOR_GAISLER,
                     GAISLER_L2TIME,
                     3,
                     1,
                     APB,
                     0x80000040,
                     0x00000028,
                     0);

simcfg.AddParams(i, simcfg.params, "");
```

3.3.11. (GAISLER_LEON3) LEON3 CPU

The LEON3 cpu module (vendorid:VENDOR_GAISLER, deviceid: GAISLER_LEON3) simulates a LEON3 processor. The configuration module example below shows how to instantiate the module. A multiprocessor system with 2 cpus is created.

```
//leon3:
i = simcfg.AddDevice(simcfg.sim,
                     busid,
                     VENDOR_GAISLER,
                     GAISLER_LEON3,
                     0,
                     0);
simcfg.AddParams(i, simcfg.params, "-smpid 0");
i = simcfg.AddDevice(simcfg.sim,
                     busid,
                     VENDOR_GAISLER,
                     GAISLER_LEON3,
                     0,
                     0);
simcfg.AddParams(i, simcfg.params, "-smpid 1");
```

3.3.11.1. Commands

Commands for this module:

`cctrl show`
: Shows decoded cache control register.

3.3.12. (GAISLER_LEON4) LEON4 CPU

The LEON4 cpu module (vendorid:VENDOR_GAISLER, deviceid: GAISLER_LEON4) simulates a LEON4 processor. The configuration module example below shows how to instantiate the module. A multiprocessor system with 2 cpus is created.

```
//leon4:
i = simcfg.AddDevice(simcfg.sim,
                     busid,
                     VENDOR_GAISLER,
                     GAISLER_LEON4,
                     0,
                     0);
simcfg.AddParams(i, simcfg.params, "-smpid 0");
i = simcfg.AddDevice(simcfg.sim,
                     busid,
                     VENDOR_GAISLER,
                     GAISLER_LEON4,
                     0,
                     0);
simcfg.AddParams(i, simcfg.params, "-smpid 1");
```

3.3.12.1. Commands

Commands for this module:

`cctrl show`
: Shows decoded cache control register.

`cctrl<cpunum> <val>`
: Write <val> into cache control register of cpu <cpunum>, possibly flushing caches.

3.3.13. (GAISLER_PCIFBRG) GRLIB GRPCI master/target interface

The GRPCI module (vendorid: VENDOR_GAISLER, deviceid: GAISLER_PCIFBRG) simulates the GRLIB GR-PCI PCI controller. The user models all devices on the PCI bus through a user supplied dynamic library.

The configuration module example below shows how to instantiate the module.

```
i = simcfg.AddDevice(simcfg.sim,
                     busid,
                     VENDOR_GAISLER,
                     GAISLER_PCIFBRG,
                     9,
                     3,
                     SLAVE,
                     0xa0000000,
                     0x20000000,
                     0,
                     AHBIO,
                     0xfff20000,
                     0x00020000,
                     0,
                     APB,
                     0x80000a00,
                     0x00000100,
                     0);
```

The GAISLER_PCIDMA module is an optional DMA engine addon to the GAISLER_PCIFBRG module. It can only be used in combination with GAISLER_PCIFBRG. The example below shows how to intantiate it.

```
i = simcfg.AddDevice(simcfg.sim,
                     busid,
                     VENDOR_GAISLER,
                     GAISLER_PCIDMA,
                     0,
                     1,
                     APB,
                     0x80000b00,
                     0x00000100,
                     0);
```

3.3.13.1. Commands

Commands for this module:

`grpci_help`
: Shows information about GRPCI module commands.

`grpci_status`
: Show the status of the GRPCI core.

`grpci_dbg [<flags>|clean|list|help]`
: Toggle debug output options. Do grpci_dbg help for a list of possible options. grpci_dbg clean will deactivate all debug output and grpci_dbg list will list the current settings.

`grpci_ip <input provider> <args>`
: Load a GRPCI input provider. The <args> parameter is passed to grpci_inp_setup() of the input provider.

3.3.13.2. GRPCI User supplied dynamic library

The user supplied dynamic library should expose a public symbol grpciinputsystem of type struct grpci_subsystem *.

The struct grpci_subsystem is defined as:

```
struct grpci_subsystem {
    void (*grpci_inp_setup) (int id, struct grpci_input *l, char **argv, int argc);
    void (*grpci_inp_restart) (int id, struct grpci_input * l);

    void (*event)(struct grpci_input * l, void (*cfunc)(), uint32 arg, uint64 offset);
    void (*stop_event_arg)(struct grpci_input * l, void (*cfunc)(),int arg);
};
```

At initialization the callback grpci_inp_setup will be called once, supplied with a pointer to structure struct grpci_input. The grpci_inp_restart callback should be set to 0.

The user supplied dynamic library should claim the grpci_input structure by using the INPUT_CLAIM() macro (see the example below). The struct grpci_input consists of callbacks that model the PCI bus (see the section PCI bus model API).

A typical user supplied dynamic library would look like this:

```
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 grpci_inp_setup (int id, struct grpci_input *l, char **argv, int argc) {

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

    l->acc = pci_acc;

    ... do module setup ...

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

static struct grpci_subsystem grpci_pci = {
    grpci_inp_setup,0,0
};

struct grpci_subsystem *grpciinputsystem = &grpci_pci;
```

3.3.13.3. PCI bus model API

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

```
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 GRPCI module whenever it reads/writes as a PCI bus master.

`cmd`
: Command to execute, see the PCI command table for details. I/O cycles not support by the GRPCI target.

`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 GRPCI module. The PCI user dynamic library can call this function to initiate an access to the GRPCI target.

`cmd`
: Command to execute, see the PCI command table for details. I/O cycles not support by the GRPCI 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 succesful, 1 in case of target abort.

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

3.3.14. (GAISLER_SDCTRL) GRLIB SDRAM Controller

The GRLIB SDRAM Controller (vendorid:VENDOR_GAISLER, deviceid: GAISLER_SDCTRL) simulates a sdram region for a given address range. The configuration module example below shows how to instantiate the module. One bar with range 0x40000000-0x60000000 is defined in which 16 mb of sdram are allocated (0-0x1000000). The AHBIO entry allocates the bar entry for the AMBA ahb configuration records for this instance.

```
//sdram mem ctrl:
i = simcfg.AddDevice(simcfg.sim,
                     busid,
                     VENDOR_GAISLER,
                     GAISLER_SDCTRL,
                     1,
                     2,
                     SLAVE,
                     0x40000000,    //bar1
                     0x20000000,    //bar1
                     1,
                     AHBIO,
                     0xfff00400,    //ahb io entry
                     0x00000200,    //ahb io entry
                     0
                     );
simcfg.AddParams(i, simcfg.params, "-sdram 16 ");
```

3.3.15. (GAISLER_SPW) GRSPW SpaceWire controller

The SpaceWire module (vendorid: VENDOR_GAISLER, deviceid: GAISLER_SPW) simulates the GRLIB GR-SPW SpaceWire controller. The simulation model delivers and receives packets through a TCP socket and can either act as a server or client which means you can easily connect two GRSIM simulators through SpaceWire.

The configuration module example below shows how to instantiate the module.

```
i = simcfg.AddDevice(simcfg.sim,
                 busid,
                 VENDOR_GAISLER,
                 GAISLER_SPW,
                 14,
                 1,
                 APB,
                 0x80000a00,
                 0x00000100,
                 0);
simcfg.AddParams(i, simcfg.params, "-idx 0 ");
```

3.3.15.1. Commands

Commands for this module:

`grspw1X_status`

Print out he status of the buffer descriptors and the registers

`grspw1X_connect [ip address[:port]]`

Try to connect to the packet server at address `ip address` port `port`. If `ip address` is omitted localhost is used. If `port` is omitted, TCP port 2225 is used.

`grspw1X_server [port]`

Start server at address `port`, if `port` is omitted 2225 is used.

`grspw1X_dbg [<flags>|clean|list|help]`

Toggle debug output options. Do grspw1X_dbg help for a list of possible options. grspw1X_dbg clean will deactivate all debug output and grspw1X_dbg list will list the current settings.

3.3.15.2. Packet server

Each SpaceWire core can be configured independently as a packet server or client using either grspwX_server or grspwX_connect. 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 below.

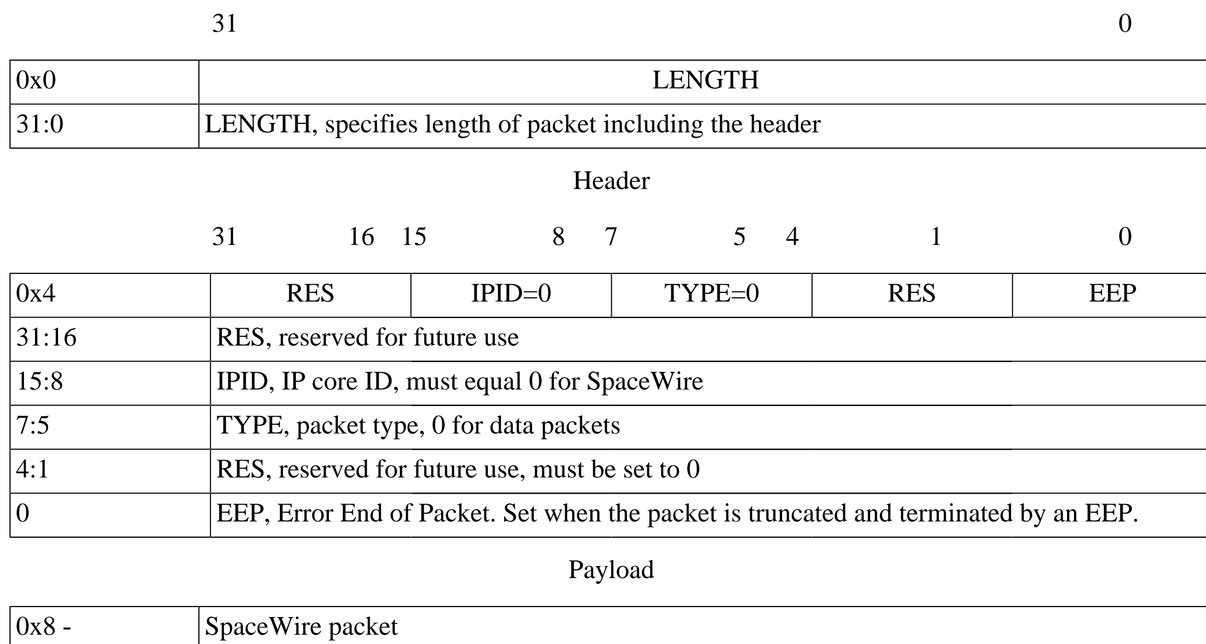
Three different types of packets are defined according to the table below.

Table 3.1. 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

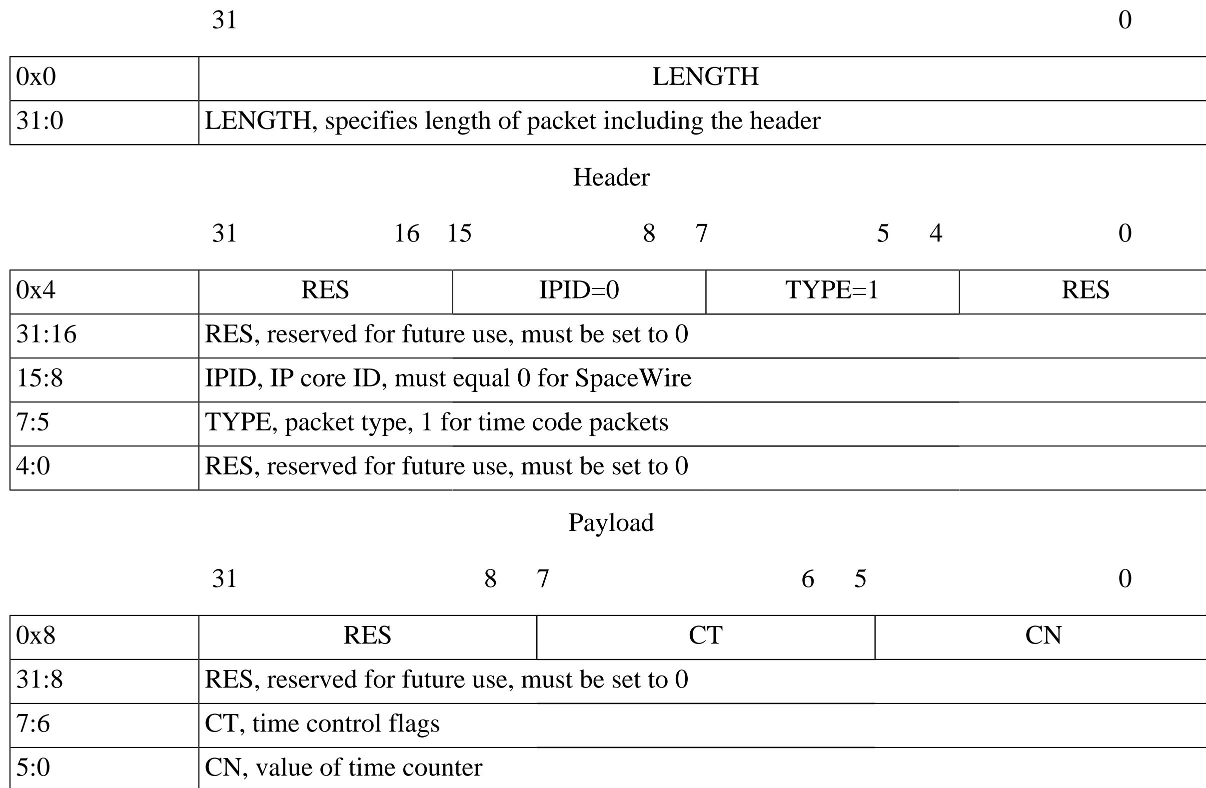
31:16	RES, reserved for future use
15:8	IPID, IP core ID, must equal 0 for SpaceWire
7:5	TYPE, packet type, 0 for data packets
4:1	RES, reserved for future use, must be set to 0
0	EEP, Error End of Packet. Set when the packet is truncated and terminated by an EEP.

Payload

0x8 -	SpaceWire packet

Figure 3.2. 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			

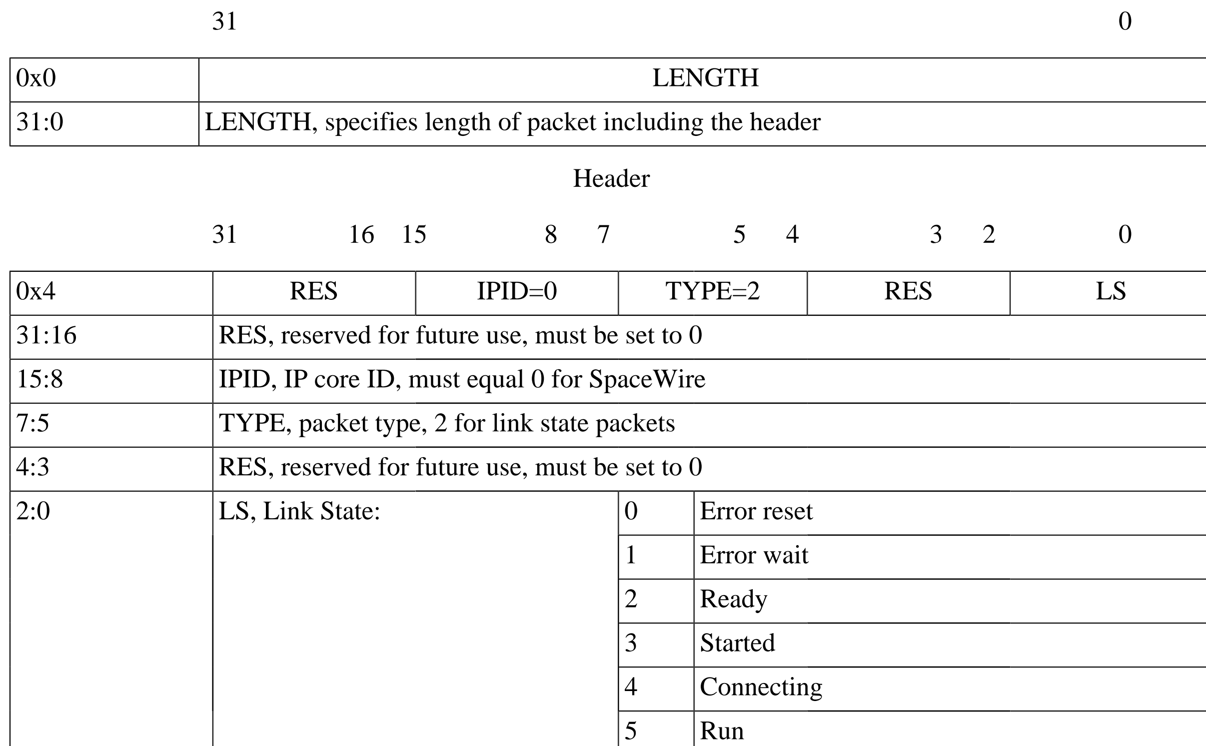
Payload

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 3.3. 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 3.4. SpaceWire link state packet

3.3.16. (GAISLER_SPW2) GRSPW2 SpaceWire controller

The SpaceWire module (vendorid: VENDOR_GAISLER, deviceid: GAISLER_SPW2) simulates the GRLIB GR-SPW2 SpaceWire controller. The simulation model delivers and receives packets through a TCP socket and can either act as a server or client which means you can easily connect two GRSIM simulators through SpaceWire. The packet format is described in @ref{GRSPW Packet server}.

The configuration module example below shows how to instantiate the module.

```
i = simcfg.AddDevice(simcfg.sim,
                 busid,
                 VENDOR_GAISLER,
                 GAISLER_SPW2,
                 14,
                 1,
                 APB,
                 0x80000a00,
                 0x00000100,
                 0);
simcfg.AddParams(i, simcfg.params, "-idx 0 ");
```

3.3.16.1. Commands

Commands for this module (replace *X* with the cores index, specified with *-idx*):

`grspwX_status`
: Print out he status of the buffer descriptors and the registers.

`grspwX_connect [ip address[:port]]`
: Try to connect to the packet server at address `ip address` port `port`. If `ip address` is omitted localhost is used. If `port` is omitted, TCP port 2225 is used.

`grspwX_server [port]`
: Start server at address `port`, if `port` is omitted 2225 is used.

`grspwX_dbg [<flags>|clean|list|help]`
: Toggle debug output options. Do *grspwX_dbg help* for a list of possible options. *grspwX_dbg clean* will deactivate all debug output and *grspwX_dbg list* will list the current settings.

3.3.17. (GAISLER_SRCTRL) GRLIB SRAM/PROM Controller

The GRLIB SRAM/PROM Controller (vendorid:VENDOR_GAISLER, deviceid: GAISLER_SRCTRL) simulates a prom or sram for a given address range. Ether one region prom/sram region or 2 regions, one prom and one sram, can be specified. The configuration module example below shows how to instantiate the module. One bar with range 0-0x10000000 is defined in which 2 mb of ram are allocated (0-0x200000). The below example doesn't allocate sram (-sram 0).

```
//sram/prom mem ctrl:
i = simcfg.AddDevice(simcfg.sim,
                     busid,
                     VENDOR_GAISLER,
                     GAISLER_SRCTRL,
                     1,
                     1,
                     SLAVE,
                     0x00000000,        //bar 1
                     0x10000000,        //bar 1
                     1
                     );
//no sram, 2mb prom
simcfg.AddParams(i, simcfg.params, "-sram 0 -prom 2048 ");
```

4. Library

GRSIM are also available as a library, allowing the simulator to be integrated in a larger simulation frame-work. The various GRSIM commands and options are accessible through a simple function interface. The library version of GRSIM is fully reentrant and thread safe.

4.1. Function interface

The following functions are provided to access GRSIM features:

```
void *Grsim_LibCreate(void);
```

Create an instance of the library. This must be done first. A pointer to the new library is returned.

```
void Grsim_LibExit(LibIf_T *lib);
```

Exit the library in a clean way.

```
void Grsim_LibInit(LibIf_T *lib, char *str);
```

Initialize the Library. (i.e. allocate space for data structures, and initialize them)) This call is NOT thread safe and should be done in a controlled way.

```
int Grsim_Cmd(LibIf_T *lib, char *cmd);
```

Execute a GRSIM command. Any valid GRSIM 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

```
int Grsim_IncTime(LibIf_T *lib, uint64 leap);
```

Increment simulator time with 'leap' ticks.

```
void Grsim_GetRegs(LibIf_T *lib, unsigned int regs[]);
void Grsim_SetRegs(LibIf_T *lib, unsigned int regs[]);
```

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

```
void Grsim_Read(LibIf_T *lib, uint32 addr, uint32 *data);
void Grsim_Write(LibIf_T *lib, uint32 addr, uint32 *data);
```

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

```
void Grsim_GdbIf(LibIf_T *lib,
                 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.

```
int32 Grsim_AddEvent(void *sim,
                     void(*EventHandler)(),
                     uint32 arg, SimTime_T offset);
```

Add an event to the event queue. 'EventHandler' will be called 'offset' simulator ticks from now. 'arg' is passed as an argument to 'EventHandler'.

```
int Grsim_StopEvent(void *sim, void(*EventHandler)())
```

Removes all events with the given 'EventHandler' from the event queue. Returns the number of events removed.

```
int Grsim_StopEventArg(void *sim, void(*EventHandler)(), void *arg)
```

Removes all events with the given 'EventHandler' and 'arg' from the event queue. Returns the number of events removed.

```
int Grsim_EventExists(void *sim, void(*EventHandler)(), void *arg)
```

Returns 1 if an event with the given 'EventHandler' and 'arg' exists in the event queue, or 0 otherwise.

```
int32 Grsim_CreateBus(void *sim);
```

Create A bus in the simulator 'sim', returns the bus index.

```
SimTime_T Grsim_GetTime(void *sim);
```

The current time in the simulator 'sim'.

```
void Grsim_IRQ(struct ahb_dev_rec *dev, int level);
```

Set the interrupt pending bit for interrupt *level*. Valid values are 0-15.

```
void Grsim_Stop(LibIf_T *lib)
```

Stop the simulation **now**. The event that is executing will be finished, but the next event in the event queue will not be executed.

```
int Grsim_SetConfigParams(LibIf_T *lib, char *conf[])
```

Add configuration parameters. This function is to be used if the library is started with the *-nosimconf* switch, i.e. grsim is started without a configurations module.

```
int32 Grsim_GetStatistics(void *sim, Grsim_GetStatisticsT *stat)
```

Get statistics for bus and cpu. Grsim_GetStatisticsT is defined in simif.h.

4.2. Multi-threading

The library version of GRSIM is designed to be thread safe and reentrant, thus providing the opportunity to run several instances of the library in a multi-threaded application.

4.2.1. Limitations to multi-threading support

Since the HASP libraries used in GRSIM for licensing are **not** thread safe, the user must make sure that multiple calls to Grsim_LibInit are made in a non-concurrent way. Otherwise the application might fail to get a license.

4.3. UART handling

By default, the library is using the same UART handling as the stand-alone simulator. This means that the UARTs can be connected to the console, or any Unix device (pseudo-ttys, pipes, fifos).

4.4. Linking an application with the GRSIM library

Six sample application are provided.

`app1.c`
: exemplifies the basic usage of the library version of GRSIM.

`app2.c`
: shows how to use the library in a multi-threaded environment.

`app3.c`
: exemplifies the use of the GDB interface.

`app4.c`
: a multithreaded example where one thread is stopped from the main thread using the Grsim_Stop command.

`app5.c`
: shows how to use the GRSIM library without a configuration module.

`app6.c`
: demonstrates the usage of the internal gdb server.

They are built by typing *make* in the example directory.

4.5. GRSIM library without a simconf module

When using the library version of GRSIM, it is possible to not use a simulator configuration module but configuring the simulator by calling the function interface directly. However, there are some restrictions in how this may be done, which is described below.

1. Call Grsim_LibCreate;
2. Call Grsim_PreConfigure;
3. Create all buses and attach your modules using Grsim_CreateBus, Grsim_AddLib, Grsim_AddDriver, Grsim_AddDevice and Grsim_AddParams. Provide your own array of strings (*char[]) as the second argument to Grsim_AddParams.
4. Call int Grsim_SetConfigParams(LibIf_T *lib, char *conf[]) Here 'conf' is supposed to be the array of strings you used as the second parameter to Grsim_AddParams
5. Call Grsim_LibInit with the option ``*-nosimconf*''

Please see the provided *app5.c* for an example.

5. Support

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

Appendix A. HASP

Table of Contents

A.1. Installing HASP Device Driver 38
A.1.1. On a Linux platform 38

This appendix describes how to install the HASP drivers.

A.1. Installing HASP Device Driver

A.1.1. On a Linux platform

The HASP software for Linux includes the following:

- Kernel mode drivers for various kernel versions
- Utilities to query the driver version and to display parallel ports
- HASP library

It is contained in the redhat-1.05-1.i386.tar.gz, suse-1.5-1.i386.tar.gz or the haspdriver.tar.gz archive in the Linux directory on the GRMON CD. The latest drivers are also available from the Aladdin website, http://www.aladdin.com.

For detailed information on the components refer to the readme files in the archive.

Note: All described action should be executed as root.

A.1.1.1. Enabling Access to USB Keys

In order for the daemon to access USB keys, the so-called usbdevfs must be mounted on /proc/bus/usb. On newer distributions it is mounted automatically (e.g SuSe 7.0). To mount usbdevfs manually use the following command:

```
mount -t usbdevfs none /proc/bus/usb
```

Enabling Access to Parallel Keys To enable access to parallel port keys, the kernel driver aksparlnx must be installed before starting aksusbd.

A.1.1.2. Loading the Daemon

Load the daemon by starting it:

```
<path>/aksusbd
```

The daemon will fork and put itself into the background.

The status message is generated in the system log informing you if the installation has been successful or not. It reports its version, the version of the API used for USB and the version of the API inside the kernel driver (for parallel port keys). If the kernel driver happens to be unavailable when aksusbd is launched, parallel port keys cannot be accessed, but USB keys are still accessible. The system log reflects this status.

If */proc/bus/usb* is not mounted when launching aksusbd, USB keys cannot be accessed.

Preferably the daemon should be started at system boot up time with some script located in */etc/rc.d/init.d* or */etc/init.d* (depending on Linux distribution).

Command Line Switches for aksusbd (Linux)

`-v`
: Print version number as decimal, format xx.xx.

`-l <value>`

Select type of diagnostic messages. Possible values are: 0 - only errors, 1 - normal (default), 2 - verbose, 3 - ultra verbose. The messages are logged in syslog with priority kern.info (and kern.debug). Refer to */etc/syslog.conf* to see where the messages will be put, usually it is the file */var/log/* messages.

`-u <value>`

Specifies the permission bits for the socket special file. Default is 666 (access for everyone).

`-h`

Print command line help

Appendix B. Sample simulator configuration

Here is a sample simulator configuration defining a standard Leon 2 setup.

```
#include <stdio.h>

#include "simconf.h"
#include "grcommon.h"


#define VENDOR_TEST 17

extern AmbaUnit_T MyMemCtrl;

static char      *drvparams[22];
static SimConf_T  simcfg;

static const struct amba_unit *testdrivers[22] = {NULL};
static const struct vendor_lib test_lib =
{
   .name         = "Test Vendor",
   .vendor       = VENDOR_TEST,
   .version      = 1,
   .drivers      = NULL,
   .simdrivers   = (AmbaUnit_T **) testdrivers,
};


static int ConfInit(void)
{
   int i, j;
   int busid;
   int result = 0;

   /* Add a new library */
   simcfg.AddLib((void *)&test_lib);
   /* Add a driver to the newly added library */
   simcfg.AddDriver(&MyMemCtrl);

   /* Create a bus */
   busid = simcfg.CreateBus(simcfg.sim);

   /* Add some devices on the bus */

   /* leon2 CPU: */
   i = simcfg.AddDevice(simcfg.sim,
                        busid,
                        VENDOR_ESA,
                        ESA_LEON2,
                        0,
                        0);
   simcfg.AddParams(i, simcfg.params, "cpu params");

   /* mem ctrl: */
   i = simcfg.AddDevice(simcfg.sim,
                        busid,
                        VENDOR_ESA,
                        ESA_MCTRL,
                        1,
                        4,
                        SLAVE,
                        0x00000000,
                        0x10000000,
                        1,
                        SLAVE,
                        0x20000000,
                        0x10000000,
                        1,
                        SLAVE,
                        0x40000000,
                        0x30000000,
                        1,
                        APB,
                        0x80000000,
                        0x00000010,
```

```
                         0);
    simcfg.AddParams(i, simcfg.params, "SRAM Ctrl params e.g -ram 2048");

    /* leon2 compat */
    i = simcfg.AddDevice(simcfg.sim,
                         busid,
                         VENDOR_GAISLER,
                         GAISLER_L2C, /* internal device: leon2compat */
                         3,
                         1,
                         APB,
                         0x80000010,
                         0x00000030,
                         0);

    /* leon2 compat */
    i = simcfg.AddDevice(simcfg.sim,
                         busid,
                         VENDOR_GAISLER,
                         GAISLER_L2C, /* internal device: leon2compat */
                         3,
                         1,
                         APB,
                         0x80000068,
                         0x00000008,
                         0);

    /* leon2 compat */
    i = simcfg.AddDevice(simcfg.sim,
                         busid,
                         VENDOR_GAISLER,
                         GAISLER_L2C, /* internal device: leon2compat  */
                         3,
                         1,
                         APB,
                         0x800000a0,
                         0x00000040,
                         0);
    simcfg.AddParams(i, simcfg.params, "");



    /* apb master */
    i = simcfg.AddDevice(simcfg.sim,
                         busid,
                         VENDOR_GAISLER,
                         GAISLER_APBMST,
                         3,
                         1,
                         SLAVE,
                         0,
                         0,
                         0);
    simcfg.AddParams(i, simcfg.params, "");

    /* leon2 timer */
    i = simcfg.AddDevice(simcfg.sim,
                         busid,
                         VENDOR_GAISLER,
                         GAISLER_L2TIME,
                         3,
                         1,
                         APB,
                         0x80000040,
                         0x00000028,
                         0);

    simcfg.AddParams(i, simcfg.params, "");



    /* Interrrupt Controller */
    i = simcfg.AddDevice(simcfg.sim,
                         busid,
                         VENDOR_GAISLER,
                         GAISLER_L2IRQ,
                         3,
                         1,
                         APB,
                         0x80000090,
                         0x00000010,
                         0);
    simcfg.AddParams(i, simcfg.params, "Int Ctrl...");
```

```
    /* UART */
    i = simcfg.AddDevice(simcfg.sim,
                         busid,
                         VENDOR_GAISLER,
                         GAISLER_APBUART,
                         3,
                         1,
                         APB,
                         0x80000070,
                         0x00000020,
                         0);
    simcfg.AddParams(i, simcfg.params, "-fast_uart");

    return result;
}

static int ConfExit(void)
{
    int result = 0;
    return result;
}

static int ConfCmd(char  *cmd, void* arg)
{
    int result = 0;
    return result;
}

static int ConfCtrl(int  ctl, void* arg)
{
    int result = 0;
    return result;
}


static SimConf_T simcfg =
{
    .sim     = NULL,

    .AddLib     = NULL,
    .AddDriver = NULL,
    .CreateBus = NULL,
    .AddDevice = NULL,
    .AddParams = NULL,

    .params = drvparams,

    .init    = ConfInit,
    .exit    = ConfExit,
    .cmd     = ConfCmd,
    .ctrl    = ConfCtrl,
};


SimConf_T *SimConf = &simcfg;
```

Appendix C. Simple Example Amba Device

Table of Contents

C.1. MemCtrl.h 43
C.2. MemCtrl.c 43

Here are presented a simple example amba device. This device, a very simple memory controller, is also supplied as source code with the grsim distribution.

C.1. MemCtrl.h

```
#ifndef _MEMCTRL_H
#define _MEMCTRL_H

#include "grcommon.h"

#define RAMSIZE 0x4000000

typedef struct
{
   unsigned char *ramb;
   unsigned int   size;
   AccRes_T       accres;
} MemCtrl_T;

#define DATA(dev) ( (MemCtrl_T *)((dev)->data) )

#endif /* _MEMCTRL_H */
```

C.2. MemCtrl.c

```
#include <stdlib.h>
#include <stdio.h>

#include "types.h"
#include "simif.h"
#include "MemCtrl.h"


static int32 memctrl_init(struct ahb_dev_rec *me)
{
   printf("Initialising the memory controller...\n");
   DATA(me)       = calloc(1, sizeof(MemCtrl_T));
   DATA(me)->ramb = calloc(1, RAMSIZE);
   DATA(me)->size = RAMSIZE;
   return 0;
}

static int32 memctrl_exit(struct ahb_dev_rec *me)
{
   printf("Exiting the memory controller...\n");

   /* Free the malloc:ed  memory */
   free( ((MemCtrl_T *)me->data)->ramb );
   free( me->data );

   return 0;
}


int32 memctrl_read(struct ahb_dev_rec *me, struct ahb_dev_rec *master,
                   uint32 address, uint32 *data, uint32 length, uint32 wsize)
{
   int        i;
   uint32     *mem  = (uint32 *)((MemCtrl_T *)(me)->data)->ramb;
   uint32      addr = address & (DATA(me)->size-1);
```

COBHAM

```
   for (i=0; i<length; i++)
   {
      data[i] = *((uint32*)&mem[(addr>>2) + i]);
   }

   if (NULL != master)
   {
      (struct abh_dev_rec *) DATA(me)->accres.dev  = master;
      DATA(me)->accres.mexc = 0;

      Grsim_AmbaReadDone(me , &DATA(me)->accres, (SimTime_T)length);
   }
   return 0;
}

int32 memctrl_write(struct ahb_dev_rec *me, struct ahb_dev_rec *master,
                    uint32 address, uint32 *data, uint32 length, uint32 wsize)
{
   int        i;
   char      *mem  = ((MemCtrl_T *)(me)->data)->ramb;
   uint32     addr = address & (DATA(me)->size-1);
   uint32     addr2;

   for (i=0; i<length; i++)
   {
      switch(wsize)
      {
         case 0:
            addr2 = addr + i;
#ifdef HOST_LITTLE_ENDIAN
            addr2 ^= 0x3;
#endif
            mem[addr2] = ((unsigned char*) data)[i];
            break;
         case 1:
            addr2 = addr + (i*2);
#ifdef HOST_LITTLE_ENDIAN
            addr2 ^= 0x2;
#endif
            *((unsigned short *) &mem[addr2]) = (unsigned short) data[i];
            break;
         case 2:
            *((unsigned int *) &mem[addr + (i*4)]) = (unsigned int) data[i];
            break;
         case 3:
            *((unsigned int *) &mem[addr   + (i*8)]) = (unsigned int) data[2*i];
            *((unsigned int *) &mem[addr+4 + (i*8)]) = (unsigned int) data[2*i+1];
      }
   }

   if (NULL != master)
   {
      (struct abh_dev_rec *) DATA(me)->accres.dev  = master;
      DATA(me)->accres.mexc = 0;

      Grsim_AmbaWriteDone(me , &DATA(me)->accres, (SimTime_T)length);
   }

   return 0;
}


static int32 memctrl_ctrl(int ctl, struct ahb_dev_rec *me, void *arg)
{
   switch (ctl)
   {
      case GRDRV_INIT:
         memctrl_init(me);
         break;
      case GRDRV_OPTIONS:
         printf("MemCtrl: %s\n", (char*)arg);
         break;
      case GRDRV_EXIT:
         memctrl_exit(me);
         break;
      default:
         break;
   }
   return 0;
}


AmbaUnit_T MyMemCtrl =
```

```
{
   .vendor     = 17,
   .version    = 1,
   .device     = 13,
   .desc       = "Simple EXAMPLE Mem Controller",

   .read       = memctrl_read,
   .write      = memctrl_write,
   .read_done  = NULL,
   .write_done = NULL,
   .cmd        = NULL,
   .ctrl       = memctrl_ctrl,
};
```

Appendix D. grcommon.h

```
#ifndef _GRCOMMON_H
#define _GRCOMMON_H

# ifdef DEBUG /* enums make debugging easier */

enum
{
   VENDOR_GAISLER    = 0x01,
   VENDOR_PENDER     = 0x02,
   VENDOR_ESA        = 0x04,
   VENDOR_OPENCORES  = 0x08,
   VENDOR_GLEICHMANN = 0x10
};

/* Gaisler cores */
enum
{
   GAISLER_LEON2DSU = 0x002,
   GAISLER_LEON3    = 0x003,
   GAISLER_LEON3DSU = 0x004,
   GAISLER_ETHAHB   = 0x005,
   GAISLER_APBMST   = 0x006,
   GAISLER_AHBUART  = 0x007,
   GAISLER_SRCTRL   = 0x008,
   GAISLER_SDCTRL   = 0x009,
   GAISLER_SSRCTRL  = 0x00A,
   GAISLER_APBUART  = 0x00C,
   GAISLER_IRQMP    = 0x00D,
   GAISLER_AHBRAM   = 0x00E,
   GAISLER_GPTIMER  = 0x011,
   GAISLER_PCITRG   = 0x012,
   GAISLER_PCISBRG  = 0x013,
   GAISLER_PCIFBRG  = 0x014,
   GAISLER_PCITRACE = 0x015,
   GAISLER_PCIDMA   = 0x016,
   GAISLER_AHBTRACE = 0x017,
   GAISLER_ETHDSU   = 0x018,
   GAISLER_CANAHB   = 0x019,
   GAISLER_GRGPIO   = 0x01A,
   GAISLER_AHBJTAG  = 0x01C,
   GAISLER_ETHMAC   = 0x01D,
   GAISLER_SPW      = 0x01F,
   GAISLER_SPACEWIRE= 0x01f,
   GAISLER_AHB2AHB  = 0x020,
   GAISLER_DDRMP    = 0x023,
   GAISLER_NUHOSP3  = 0x02B,
   GAISLER_DDR2SP   = 0x02E,

   GAISLER_GRTM      = 0x030,
   GAISLER_GRTC      = 0x031,
   GAISLER_GRPW      = 0x032,
   GAISLER_GRCTM     = 0x033,

   GAISLER_GRHCAN    = 0x034,
   GAISLER_GRFIFO    = 0x035,
   GAISLER_GRADCDAC  = 0x036,
   GAISLER_GRPULSE   = 0x037,
   GAISLER_GRTIMER   = 0x038,
   GAISLER_AHB2PP    = 0x039,
   GAISLER_GRVERSION = 0x03A,

   GAISLER_LEON4     = 0x048,
   GAISLER_L2CACHE   = 0x04B,

   GAISLER_FTAHBRAM = 0x050,
   GAISLER_FTSRCTRL = 0x051,
   GAISLER_AHBSTAT  = 0x052,
   GAISLER_LEON3FT  = 0x053,
   GAISLER_FTMCTRL  = 0x05F,

   GAISLER_KBD      = 0x060,
   GAISLER_VGA      = 0x061,
   GAISLER_LOGAN    = 0x062,
   GAISLER_B1553BC  = 0x070,
   GAISLER_B1553RT  = 0x071,
   GAISLER_B1553BRM = 0x072,
```

```
   GAISLER_LEON2    = 0xffb,
   GAISLER_L2IRQ    = 0xffc, /* internal device: leon2 irq*/
   GAISLER_L2TIME   = 0xffd, /* internal device: leon2 timer */
   GAISLER_L2C      = 0xffe, /* internal device: leon2compat */
   GAISLER_PLUGPLAY = 0xfff  /* internal device: plug & play configarea */

};

/* ESA cores */
enum
{
  ESA_LEON2     = 0x002,
  ESA_LEON2APB  = 0x003,
  ESA_L2IRQ     = 0x005,
  ESA_L2TIMER   = 0x006,
  ESA_L2UART    = 0x007,
  ESA_L2CFG     = 0x008,
  ESA_L2IO      = 0x009,
  ESA_MCTRL     = 0x00F,
  ESA_PCIARB    = 0x010,
  ESA_HURRICANE = 0x011,
  ESA_SPW_RMAP  = 0x012,
  ESA_AHBUART   = 0x013,
  ESA_SPWA      = 0x014,
  ESA_BOSCHCAN  = 0x015,
  ESA_L2IRQ2    = 0x016,
  ESA_L2STAT    = 0x017,
  ESA_L2WPROT   = 0x018
};

enum
{
  GLEICHMANN_CUSTOM = 0x001,
  GLEICHMANN_GEOLCD01 = 0x002,
  GLEICHMANN_DAC = 0x003
};

enum
{
  GRDRV_IRQ_SETLEVEL=0,
  GRDRV_IRQ_SETRUNNING=1,
  GRDRV_IRQ_GETRUNNING=2
};

enum
{
   GRDRV_NOP                      = 0,
   GRDRV_APBINIT                  = 1,
   GRDRV_OPTIONS                  = 2,
   GRDRV_INIT                     = 3,
   GRDRV_EXIT                     = 4,
   GRDRV_REPORT                   = 5,
   GRDRV_RESTART                  = 6,
   GRDRV_APBCONFREAD              = 7,
   GRDRV_IRQCTRL_ATTACH           = 8,
   GRDRV_IRQROUTE_ATTACH          = 9,
   GRDRV_GDB_MODE                 = 10,
   GRDRV_CPU_SETPC                = 11,
   GRDRV_CPU_GETPC                = 12,
   GRDRV_CPU_ATTACH               = 13,
   GRDRV_CPU_DISP_REGS            = 14,
   GRDRV_CPU_ADD_WP               = 15,
   GRDRV_CPU_DEL_WP               = 16,
   GRDRV_CPU_SHOW_WP              = 17,
   GRDRV_CPU_ADD_BP               = 18,
   GRDRV_CPU_DEL_BP               = 19,
   GRDRV_CPU_DEL_BP_GDB           = 20,
   GRDRV_CPU_SHOW_BP              = 21,
   GRDRV_CPU_STEP                 = 22,
   GRDRV_CPU_CONT                 = 23,
   GRDRV_CPU_GET_REGS             = 24,
   GRDRV_CPU_SET_REGS             = 25,
   GRDRV_CPU_GET_REGS_GDB         = 26,
   GRDRV_CPU_SET_REGS_GDB         = 27,
   GRDRV_CPU_FLUSH                = 28,
   GRDRV_CPU_FLUSH_WIN            = 29,
   GRDRV_CPU_GET_STATUS           = 30,
   GRDRV_CPU_GDB_SYNC_REGS        = 31,
   GRDRV_CPU_CACHE_SNOOP          = 32,
   GRDRV_AMBA_READ_DONE           = 33,
   GRDRV_AMBA_WRITE_DONE          = 34,
```

```
    GRDRV_CPU_READ_SCRATCH         = 35,
    GRDRV_CPU_WRITE_SCRATCH        = 36,
    GRDRV_CPU_DISAS                = 37,
    GRDRV_MCTRL_GET_USEABLE_STACK = 38,
    GRDRV_CPU_GET_SMPID            = 39,
    GRDRV_CPU_PRINT_STATUS         = 40,
    GRDRV_CPU_TRANSADDR            = 41,
    GRDRV_CPU_GETNPC               = 42,
    GRDRV_BUS_GET_FREQ             = 43,
    GRDRV_RESTART_CPU              = 44,
    GRDRV_CPU_GET_SREGS            = 45,
    GRDRV_CPU_REPORT_CACHE         = 46,
    GRDRV_PREINIT                  = 47,
    GRDRV_SYSREG                   = 48,
    GRDRV_PREINIT2                 = 49,
    GRDRV_PRELOAD                  = 50
};

# else

/* Vendor codes */
#define VENDOR_GAISLER   0x01
#define VENDOR_PENDER    0x02
#define VENDOR_ESA       0x04
#define VENDOR_OPENCORES 0x08
#define VENDOR_GLEICHMANN 0x10


/* Gaisler cores */
#define GAISLER_LEON2DSU 0x002
#define GAISLER_LEON3    0x003
#define GAISLER_LEON3DSU 0x004
#define GAISLER_ETHAHB   0x005
#define GAISLER_APBMST   0x006
#define GAISLER_AHBUART  0x007
#define GAISLER_SRCTRL   0x008
#define GAISLER_SDCTRL   0x009
#define GAISLER_SSRCTRL  0x00A
#define GAISLER_APBUART  0x00C
#define GAISLER_IRQMP    0x00D
#define GAISLER_AHBRAM   0x00E
#define GAISLER_GPTIMER  0x011
#define GAISLER_PCITRG   0x012
#define GAISLER_PCISBRG  0x013
#define GAISLER_PCIFBRG  0x014
#define GAISLER_PCITRACE 0x015
#define GAISLER_PCIDMA   0x016
#define GAISLER_AHBTRACE 0x017
#define GAISLER_ETHDSU   0x018
#define GAISLER_CANAHB   0x019
#define GAISLER_GRGPIO   0x01A
#define GAISLER_AHBJTAG  0x01C
#define GAISLER_ETHMAC   0x01D
#define GAISLER_SPW      0x01F
#define GAISLER_SPACEWIRE 0x01f
#define GAISLER_AHB2AHB  0x020
#define GAISLER_DDRMP    0x023
#define GAISLER_NUHOSP3  0x02b
#define GAISLER_DDR2SP   0x02E
#define GAISLER_GRTM     0x030
#define GAISLER_GRTC     0x031
#define GAISLER_GRPW     0x032
#define GAISLER_GRCTM    0x033
#define GAISLER_GRHCAN   0x034
#define GAISLER_GRFIFO   0x035
#define GAISLER_GRADCDAC 0x036
#define GAISLER_GRPULSE  0x037
#define GAISLER_GRTIMER  0x038
#define GAISLER_AHB2PP   0x039
#define GAISLER_GRVERSION 0x03A

#define GAISLER_LEON4     0x048
#define GAISLER_L2CACHE   0x04B

#define GAISLER_FTAHBRAM 0x050
#define GAISLER_FTSRCTRL 0x051
#define GAISLER_AHBSTAT  0x052
#define GAISLER_LEON3FT  0x053
#define GAISLER_FTMCTRL  0x05F

#define GAISLER_KBD      0x060
#define GAISLER_VGA      0x061
#define GAISLER_LOGAN    0x062
```

```
#define GAISLER_B1553BC  0x070
#define GAISLER_B1553RT  0x071
#define GAISLER_B1553BRM 0x072

#define GAISLER_LEON2    0xffb
#define GAISLER_L2IRQ    0xffc /* internal device: leon2 interrupt controller */
#define GAISLER_L2TIME   0xffd /* internal device: leon2 timer */
#define GAISLER_L2C      0xffe /* internal device: leon2compat */
#define GAISLER_PLUGPLAY 0xfff /* internal device: plug & play configarea */

/* ESA cores */
#define  ESA_LEON2       0x002
#define  ESA_LEON2APB    0x003
#define  ESA_L2IRQ       0x005
#define  ESA_L2TIMER     0x006
#define  ESA_L2UART      0x007
#define  ESA_L2CFG       0x008
#define  ESA_L2IO        0x009
#define  ESA_MCTRL       0x00F
#define  ESA_PCIARB      0x010
#define  ESA_HURRICANE   0x011
#define  ESA_SPW_RMAP    0x012
#define  ESA_AHBUART     0x013
#define  ESA_SPWA        0x014
#define  ESA_BOSCHCAN    0x015
#define  ESA_L2IRQ2      0x016
#define  ESA_L2STAT      0x017
#define  ESA_L2WPROT     0x018

/* GLEICHMANN cores */
#define GLEICHMANN_CUSTOM 0x001
#define GLEICHMANN_GEOLCD01 0x002
#define GLEICHMANN_DAC 0x003

/* irq<->cpu cmds */
#define GRDRV_IRQ_SETLEVEL   0
#define GRDRV_IRQ_SETRUNNING 1
#define GRDRV_IRQ_GETRUNNING 2

/* driver commands */
#define GRDRV_NOP                    0
#define GRDRV_APBINIT                1
#define GRDRV_OPTIONS                2
#define GRDRV_INIT                   3
#define GRDRV_EXIT                   4
#define GRDRV_REPORT                 5
#define GRDRV_RESTART                6
#define GRDRV_APBCONFREAD            7
#define GRDRV_IRQCTRL_ATTACH         8
#define GRDRV_IRQROUTE_ATTACH        9
#define GRDRV_GDB_MODE               10
#define GRDRV_CPU_SETPC              11
#define GRDRV_CPU_GETPC              12
#define GRDRV_CPU_ATTACH             13
#define GRDRV_CPU_DISP_REGS          14
#define GRDRV_CPU_ADD_WP             15
#define GRDRV_CPU_DEL_WP             16
#define GRDRV_CPU_SHOW_WP            17
#define GRDRV_CPU_ADD_BP             18
#define GRDRV_CPU_DEL_BP             19
#define GRDRV_CPU_DEL_BP_GDB         20
#define GRDRV_CPU_SHOW_BP            21
#define GRDRV_CPU_STEP               22
#define GRDRV_CPU_CONT               23
#define GRDRV_CPU_GET_REGS           24
#define GRDRV_CPU_SET_REGS           25
#define GRDRV_CPU_GET_REGS_GDB       26
#define GRDRV_CPU_SET_REGS_GDB       27
#define GRDRV_CPU_FLUSH              28
#define GRDRV_CPU_FLUSH_WIN          29
#define GRDRV_CPU_GET_STATUS         30
#define GRDRV_CPU_GDB_SYNC_REGS      31
#define GRDRV_CPU_CACHE_SNOOP        32
#define GRDRV_AMBA_READ_DONE         33
#define GRDRV_AMBA_WRITE_DONE        34
#define GRDRV_CPU_READ_SCRATCH       35
#define GRDRV_CPU_WRITE_SCRATCH      36
#define GRDRV_CPU_DISAS              37
#define GRDRV_MCTRL_GET_USEABLE_STACK 38
#define GRDRV_CPU_GET_SMPID          39
#define GRDRV_CPU_PRINT_STATUS       40
#define GRDRV_CPU_TRANSADDR          41
```

```
#define GRDRV_CPU_GETNPC              42
#define GRDRV_BUS_GET_FREQ            43
#define GRDRV_RESTART_CPU             44
#define GRDRV_CPU_GET_SREGS           45
#define GRDRV_CPU_REPORT_CACHE        46
#define GRDRV_PREINIT                 47
#define GRDRV_SYSREG                  48
#define GRDRV_PREINIT2                49
#define GRDRV_PRELOAD                 50

# endif /* DEBUG */


# ifndef __ASSEMBLER__

/* AMBA configuration ID */

#define VENID(x)  ( ((x) >> 24) & 0x0ff )
#define DEVID(x)  ( ((x) >> 12) & 0x0fff)
#define CFGVER(x) ( ((x) >> 10) & 0x03  )
#define VERID(x)  ( ((x) >>  5) & 0x01f )
#define IRQNUM(x) ( ((x) >>  0) & 0x1f  )


/* Used as master for diagnostic accesses on the AMBA bus */
#define DIAG NULL

struct ahb_cfg_rec
{
   int devid;
   int custom_cfg[3];
   int bar[4];
};

struct ahb_cfg_area
{
   struct ahb_cfg_rec master[64];
   struct ahb_cfg_rec slave[64];
};

struct apb_cfg_rec
{
   int devid;
   int bar;
};

struct apb_cfg_area
{
   struct apb_cfg_rec slave[64];
};


struct ahb_mem_rec
{
   unsigned int start;
   unsigned int end;
   unsigned int type;
};

struct ahb_dev_rec;

struct grdriver
{
   short vendor;
   short version;
   short device;
   char  desc[32];
   int (*ctrl)(int,    struct ahb_dev_rec *);
   int (*cmd) (char *, struct ahb_dev_rec *);
};

typedef struct acc_res
{
   struct ahb_dev_rec *dev;
   int                 mexc;
} AccRes_T;

typedef int (*IrqFn_T)(struct ahb_dev_rec *dev, int cmd, int arg);
typedef struct
{
   struct ahb_dev_rec  *dev;
   IrqFn_T              fn;
} IrqRouteEntry_T;
```

```
typedef struct amba_unit
{
   /* identification data */
   short  vendor;
   short  version;
   short  device;
   char   desc[32];

   /* functions */
   int (*read)      (struct ahb_dev_rec *me, struct ahb_dev_rec *master, unsigned int address,
                     unsigned int *data, unsigned int length, unsigned int wsize);
   int (*write)     (struct ahb_dev_rec *me, struct ahb_dev_rec *master, unsigned int address,
                     unsigned int *data, unsigned int length, unsigned int wsize);
   int (*read_done) (AccRes_T *result);
   int (*write_done)(AccRes_T *result);
   int (*ctrl)      (int ctl, struct ahb_dev_rec *me, void *args);
   int (*cmd)       (char *cmd, struct ahb_dev_rec *me);
} AmbaUnit_T;


struct ahb_dev_rec
{
   struct ahb_mem_rec  mem[4];
   struct ahb_mem_rec  apb;
   struct grdriver    *drv;
   struct amba_unit   *simdrv;
   void               *data;
   unsigned int        devid;
   unsigned char       irq;
   unsigned char       vendor;
   unsigned short      device;
   unsigned short      version;
   unsigned char       type;
   unsigned char       index;

   // access with timing accumulated
   unsigned char       acc_access;
   unsigned int        acc_ws;

   unsigned int        bus;
   unsigned int        custom_cfg[3];

   /* Entries only used in simulator */
   void               *sim;
   unsigned int        busid;
};

struct bus_dev_rec
{
   struct ahb_dev_rec dev[128];
   int                ndevs;
   int                ioarea[4];
   int                cfgarea;
   int                 nbus;
};

struct vendor_lib
{
   char name[64];
   int                vendor;
   int                version;
   struct grdriver  **drivers;
   struct amba_unit **simdrivers;
};

typedef struct addr_data
{
   unsigned int  addr;
   unsigned int *data;
} AddrData_T;

enum regnames
{
   G0, G1, G2, G3, G4, G5, G6, G7,
   O0, O1, O2, O3, O4, O5, SP, O7,
   L0, L1, L2, L3, L4, L5, L6, L7,
   I0, I1, I2, I3, I4, I5, FP, I7,

   F0, F1, F2, F3, F4, F5, F6, F7,
   F8, F9, F10, F11, F12, F13, F14, F15,
   F16, F17, F18, F19, F20, F21, F22, F23,
```

```
   F24, F25, F26, F27, F28, F29, F30, F31,
   Y, PSR, WIM, TBR, PC, NPC, FPSR, CPSR
};

extern void  GetRegisters  (void *lib, unsigned int regs[]);

struct mbus_rec
{
  int          found;
  unsigned int cfgarea;
  float        ffact;
};

# endif /* __ASSEMBLER__ */
#endif /* _GRCOMMON_H  */
```

COBHAM

Cobham Gaisler AB
Kungsgatan 12
411 19 Gothenburg
Sweden
www.cobham.com/gaisler
sales@gaisler.com
T: +46 31 7758650
F: +46 31 421407

Cobham Gaisler AB, reserves the right to make changes to any products and services described herein at any time without notice. Consult Cobham or an authorized sales representative to verify that the information in this document is current before using this product. Cobham 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 Cobham; nor does the purchase, lease, or use of a product or service from Cobham convey a license under any patent rights, copyrights, trademark rights, or any other of the intellectual rights of Cobham or of third parties. All information is provided as is. There is no warranty that it is correct or suitable for any purpose, neither implicit nor explicit.

Copyright © 2015 Cobham Gaisler AB