The ERC32 GNU Cross-Compiler System

Version 1.3
January 1998

Jiri Gaisler
European Space Research and Technology Centre (ESA/ESTEC)

European Space Agency

jgais@ws.estec.esa.nl

The ERC32 GNU cross-compiler system

1 Introduction

1.1 General

This document describes the ERC32 GNU cross-compiler system version 1.3. Discussions are provided for the following topics:

- contents and directory structure of ERC32CCS
- compiling and linking ERC32 applications
- usage of SIS and MKPROM
- debugging ERC32 application with GDB/SIS

The ERC32 GNU cross-compiler system is a multi-platform development system based on the GNU family of freely available tools with additional 'point' tools developed by Cygnus, OAR and ESTEC. The ERC32CCS consists of the following packages:

- EGCS/GCC C/C++ compiler
- GNAT Ada95 compiler
- GNU binary utilities
- RTEMS C/C++ real-time kernel
- Newlib standalone C-library
- SIS ERC32 simulator
- GDB debugger with ERC32 remote debugging monitor (rdbmon)
- DDD graphical front-end for GDB
- MKPROM boot-prom builder

1.2 News in version 1.3

This version of ERC32CCS contains the following changes with respect to 1.2:

- DDD version 2.2, including new documentation
- Modified assembler to compensate newly discovered FPU rev.B bugs
- Egcs-1.0 C/C++ compiler
- RTEMS 4.0 beta release with simplified compile procedure
- Experimental Ada95 compiler based on gnat-3.10 and rtems-4.0
- Re-compiled floating-point library (libm.a) to incorporate the FPU fixes
- GDB version 4.16.1 with gnat support
- SIS version 2.7.6

2 Installation and directory structure

2.1 Obtaining ERC32CCS

ERC32CCS is only distributed via anonymous ftp. The primary home of ERC32CCS is the ESTEC ftp-server at ftp://ftp.estec.esa.nl/pub/ws/wsd/erc32/erc32ccs. Two platforms are supported: SUN Sparcstaion (SunOS & Solaris) and PC/Linux. Sources for rdbmon and mkprom are provided with ERC32CCS, the remaining sources can be found at the usual GNU sites.

2.2 Installation

The ERC32CCS directory tree is compiled to reside in `/usr/local/erc32`. After obtaining the compressed tar-file with the binary distribution, un-compress and un-tar it in a suitable location - if this is not `/usr/local/erc32` then a link have to be created to point to the location of the ERC32CCS directory. The distribution can be installed with the following command:

```
gunzip -c erc32ccs-1.3.tar.gz | tar xf -
```

This will create the `erc32` directory in the current directory. The `erc32` contains the following sub-directories:

bin	executables
doc	documentation
include	host includes
lib	host libraries
man	man pages
rtems	rtems libraries
rtemsnp	rtems libraries (no posix)
sparc-rtems	target libraries (ERC32)
src	various sources

Documentation for the various packages can be found in `doc`, while unix-style man-pages are in `man`. Note that the directory structure for ERC32CCS-1.3 has changed and is not compatible with ERC32CCS-1.2.

2.3 Environment

ERC32CCS does not longer require any environment variables to be set; just add /usr/local/erc32/bin to your search path.

2.4 Support

For technical support regarding GNU tools, contact Cygnus at http://www.cygnus.com/. For RTEMS support contact OAR at http://www.oarcorp.com/. For SIS and MkProm support, contact J.Gaisler (ESTEC) at jgais@ws.estec.esa.nl.

3 Using ERC32CCS

3.1 ERC32CCS tools

The following tools are included in ERC32CCS:

ddd	graphic X11 front-end to GDB
fcheck	utility to check for FPU rev.B bugs
mkprom	boot-prom builder
protoize	GNU protoize utility
sis	ERC32 simulator
sis64	ERC32 simulator (with 64-bit time)
sparc-rtems-ar	library archiver
sparc-rtems-as	modified cross-assembler (FPU rev.B fixes)
sparc-rtems-c++	C++ cross-compiler
sparc-rtems-c++filt	utility to demangle C++ symbols
sparc-rtems-g++	same as sparc-rtems-c++
sparc-rtems-gasp	assembler pre-processor
sparc-rtems-gcc	C/C++ cross-compiler
sparc-rtems-gdb	debugger with ERC32 simulator and remote target interface
sparc-rtems-gdb64	debugger with ERC32 simulator (64-bit time)
sparc-rtems-gnatcmd	Utility to print all GNAT command switches
sparc-rtems-gnatmake	Ada make utility
sparc-rtems-gnatbind	Ada binder
sparc-rtems-gnatf	Ada syntax checker and cross-reference generator
sparc-rtems-gnatprep	Ada pre-processor
sparc-rtems-gnatbl	Ada bind and link
sparc-rtems-gnatkr	Ada file name kruncher
sparc-rtems-gnatpsta	Utility to print the *Standard* package
sparc-rtems-gnatchop	Ada source code splitter
sparc-rtems-gnatlink	Ada linker
sparc-rtems-gnatpsys	Utility to display the *System* package
sparc-rtems-gnatchp	Ada source code splitter
sparc-rtems-gnatls	Ada library lister
sparc-rtems-ld	GNU linker
sparc-rtems-nm	utility to print symbol table
sparc-rtems-objcopy	utility to convert between binary formats
sparc-rtems-objdump	utility to dump various parts of executables
sparc-rtems-ranlib	library sorter
sparc-rtems-size	utility to display segment sizes
sparc-rtems-strings	utility to dump strings from executables
sparc-rtems-strip	utility to remove symbol table
unprotoize	GNU unprotoize utility

3.2 Documentation

An extensive set of documentation for all tools can be found in `doc` and `man`. The following documents are provided:

as.bdf	Using **as** - the GNU assembler
bfd.pdf	Libbfd - the binary file description
binutils.pdf	The GNU binary utilities
cpp.pdf	The C Preprocessor
ddd.pdf	DDD - The Data Display Debugger
gcc.pdf	Using and porting GCC
gdb.pdf	Debugging with GDB
gnat_rm.pdf	GNAT reference manual
gnat_ug.pdf	GNAT User's guide
ld.pdf	Using ld - the GNU linker
rtems_dev.pdf	RTEMS Development environment guide
rtems_relnotes.pdf	RTEMS Release notes
rtems_sparc.pdf	RTEMS SPARC Applications supplement
rtems_user.pdf	RTEMS C User's manual (this is the one you want!)
sis.pdf	SIS - SPARC instruction set simulator manual
sparcv7.pdf	SPARC V7 Instruction set manual

Data sheets for the ERC32 chip-set are also provided:

mecspec.pdf	MEC rev.A Device specification
tsc961e.pdf	TSC961 Integer Unit User's manual
tsc962e.pdf	TSC962 Floating-point Unit user's manual
sysover.pdf	ERC32 System overview

The documents are all provided in PDF format, with searchable indexes. The GNU documents have embedded hyper-links and searchable document text. A free PDF viewer ('acrobat') can be down-loaded from Adobe (http://www.adobe.com/).

3.3 Development flow

The compilation and debugging of an ERC32-based applications is done in the following steps:

1. Compile and link program with gcc
2. Debug program in standalone simulator (SIS) or with gdb
3. Debug program on remote target with gdb
4. Create boot-prom for a standalone application

The ERC32CCS-1.3 supports three types of applications; ordinary sequential C-programs, multi-tasking real-time C--programs based on the RTEMS kernel and Ada-95 programs. Compiling and linking is done in much the same manner as with the host-based gcc and GNAT.

3.4 RTEMS applications

The primary application type for ERC32CCS is now RTEMS programs. As of ERC32CCS-1.3, compiling and linking of RTEMS applications is greatly simplified. The usage of a RTEMS specific makefile is no longer required, the gcc compiler driver supplies all necessary paths and libraries. To compile and link a RTEMS application, use 'sparc-rtems-gcc':

```
sparc-rtems-gcc -g -O3 rtems-hello.c -o rtems-hello.exe
```

The various compilation switches are explained in the gcc manual (gcc.pdf) and the man-pages. RTEMS is provided in two versions; with and without POSIX threads interface. If applications are written with the POSIX interface, add the -posix switch during compilation and linking:

```
sparc-rtems-gcc -posix -g -O3 posix-app.c -o posix-app.exe
```

The default load address is start of RAM, i.e. 0x2000000. Any load address can be specified through the -Ttext option (see gcc manual).

Extensive documentation is provided on `RTEMS in doc/rtems_user.pdf.`

3.5 Compiling sequential C-programs

Ordinary sequential C programs are still supported by providing the -nortems switch to the compiler driver:

```
sparc-rtems-gcc -nortems -g -O2 hello.c -o hello.exe
```

3.6 Compiling Ada95 programs

Compiling and linking an Ada95 program is easiest done through gnatmake:

```
sparc-rtems-gnatmake -g -O3 -gnatp dais.adb
```

Individual units can be compiled through gcc:

```
sparc-rtems-gcc -c -g -O3 -gnatp dais.adb
```

If linking is done through gnatbl, the -posix and -qgnat flags has to be added:

```
sparc-rtems-gnatbl -posix -qgnat -g -O3 -gnatp dais.adb
```

For details on how to use gnat, see the GNAT User's Manual (`gnat_us.pdf`) and GNAT Reference Manual (`gnat_rm.pdf`). NOTE: the gnat compiler is only provided for testing purposes, and is not by any means validated or guaranteed. A commercial, validated version is available from Ada Core Technology (http://www.gnat.com/).

The gnat compiler is configured for a maximum of 20 tasks and 30 mutexes. If you need a different configuration, go to the erc32/src/libio directory, edit gnatinit.c and do a 'make install'.

3.7 Making boot-proms

Both sequential C-programs and RTEMS applications are linked to run from beginning of ram at address 0x2000000. To make a boot-prom that will run on a standalone target, use the `mkprom` utility. This will create a compressed boot image that will load the application to the beginning of ram, initiate various MEC register, and finally start the application. `mkprom` will set all target dependent parameters, such as memory sizes, number of memory banks, waitstates, baudrate, and system clock. The applications do not set these parameters themselves, and thus do not need to relinked for different board architectures. The the example below creates a boot-prom for a system with 1 Mbyte RAM, one waitstate during write, 3 waitstates for rom access, and 12 MHz system clock. For more details see the mkprom manual

```
mkprom -ramsz 0x100000 -ramws 1 -romws 3 hello.exe -freq 12 hello.srec
```

To let the real-time clock generate the correct time base, the loader passes board frequency to the program via the memory location at %tbr[0x7e0]. When running programs on top of sparcmon without using `mkprom`, this is not done and the default frequency of 14 MHz is assumed. This can be changed by setting the first word in trap 0x7e to the value of the clock frequency (in MHz) after the program has been loaded but before it is started. This needs to be done only to get the time correct, the run-time system will not modify the UART baudrate if it already has been set. The example below sets the clock frequency to 10 MHz for a program that is linked to run from address 0x2020000:

```
monitor> ex -l 20207e0
0x020207e0: 0x91d02000 ? a
```

3.8 Simple examples

Following example compiles the famous ‘hello world’ program and creates a boot-prom in SRECORD format:

```
tellus > sparc-rtems-gcc -nortems -g -O2 hello.c -o hello
tellus > mkprom hello -o hello.exe
tellus> sparc-rtems-objcopy -v -O srec hello.exe hello.srec
copy from prom.out(a.out-sunos-big) to hello.srec(srec)
tellus>
```

An Ada application compiled through gnatmake:

```
tellus > sparc-rtems-gnatmake -g -O3 -gnatp dais.adb
sparc-rtems-gcc -c -g -O3 -gnatp dais.adb
sparc-rtems-gnatbind -aO./ -I- -r -x dais.ali
sparc-rtems-gnatlink -g -posix -qgnat dais.ali
tellus >sparc-rtems-size dais
text    data    bss     dec     hex     filename
204720  6392    32268   243380  3b6b4   dais
```

Several example C, C++ and Ada program can be found in `src/examples`. The RTEMS validation tests can be found in `src/examples/RTEMS`.

3.9 FPU rev.B bugs

The FPU rev.B have a bug that will make certain lddf/stdf sequences fail. The compiler only rarely emits these sequences. The occurrence of such sequence can be check with the provided fcheck program. A modified assembler is also provided which will automatically insert NOPs in the failing sequences to correct this problem. The modified assembler also emits NOPs between ldf/fpop sequences with dependencies to circumvent a second FPU bug which is only occur if waitstates are used.

4 Execution and debugging

The applications built by ERC32CCS can be executed in four different ways; on the standalone simulator, on gdb with integrated simulator, on a remote target connected to gdb and on a standalone target board from prom.

4.1 Standalone simulator

The standalone simulator can run both application produced by the compiler and srecord images produced by Mk-Prom. The following example shows how the ‘hello world’ program is run:

```
tellus > sis hello

 SIS - SPARC instruction simulator 2.7.6,  copyright Jiri Gaisler 1995-1998
 Bug-reports to jgais@ws.estec.esa.nl

loading hello:
section .text at 0x02000000 (26032 bytes)
section .data at 0x020065b0 (1304 bytes)
section .bss at 0x02006ac8 (40 bytes)(not loaded)
serial port A on stdin/stdout
sis> go
Hello world
IU in error mode (257)
     2567  02000800  91d02000  ta  0
sis>
```

Note that the program was started from address 0x2000000, the default start address. Programs always halt the IU after they have terminated, that is why the IU goes into error mode. The boot-prom image can also be simulated:

```
tellus > sparc-rtems-sis hello.srec

SIS - SPARC instruction simulator 2.7.6,  copyright Jiri Gaisler 1995-1998
 Bug-reports to jgais@ws.estec.esa.nl

loading hello.srec:
section .sec1 at 0x00000000 (16784 bytes)
serial port A on stdin/stdout
sis> run
ERC32 boot loader v1.0

  initialising RAM
  decompressing .text
  decompressing .data

  starting hello

Hello world!
IU in error mode (257)
sis>
```

4.2 GDB with simulator

To do symbolic debugging of both C and Ada applications, use gdb. After gdb is started, the simulator has to attached and the program loaded. It is important that the applications have been compiled with the -g switch. Below is a sample session:

```
tellus > sparc-rtems-gdb hello
(gdb)tar sim

 SIS - SPARC instruction simulator 2.7.6
 Bug-reports to Jiri Gaisler ESA/ESTEC (jgais@ws.estec.esa.nl)
serial port A on stdin/stdout
Connected to the simulator.
(gdb)
(gdb) load
(gdb) break main
Breakpoint 1 at 0x20014e4: file hello.c, line 4.
(gdb) run
Starting program: /home/jgais/erc32/src/examples/hello

Breakpoint 1, main () at hello.c:4
4               printf(“Hello world!\n”);
(gdb) cont
Continuing.
Hello world!

Program exited normally.
(gdb)
```

4.3 GDB with remote target

To attach gdb to a remote targets similar to attaching to the simulator. The baud rate for the serial port has to be specified and the remote target monitor has to run on the target. Also, a tip window should be connected to UART A to see the application output. Below is a sample session with a remote target:

```
tellus> xterm -e tip -38400 /dev/ttya &
[234]
tellus > sparc-rtems-gdb hello
(gdb) set remotebaud 38400
(gdb) tar erc32 /dev/ttyb
Remote debugging using /dev/ttyb
0x2000000 in trap_table ()
(gdb) lo
Loading section .text, size 0x65e8 vma 0x2000000
Loading section .data, size 0x4d0 vma 0x20065e8
(gdb) bre main
Breakpoint 1 at 0x20014e4: file hello.c, line 3.
(gdb) run
Starting program: /home/jgais/erc32/src/examples/hello
main () at hello.c:4
3           printf(“Hello world!\n”);
```

```
(gdb) cont
Continuing.

Program exited with code 03.
(gdb)
```

Note that the program has to be loaded each time before it is started with ‘run’. This is to initialise the data segment to the proper start values. It is possible to switch between several targets (real or simulated) in the same GDB session. Use the GDB command **detach** to disconnect from the present target before attaching a new one.

4.4 Using DDD

DDD is a graphical front-end to gdb, and can be used regardless of target. To start DDD with the debugger use:

ddd --debugger sparc-rtems-gdb --attach-window

A small script, `dddx`, is provided to start DDD in this configuration. You might need the full path in front of DDD if you already have a version of ddd installed. To get the source code displayed in the ddd window, click on **locate()**. The required gdb commands to connect to a target can be entered in the command window. See the GDB and DDD manuals for how to set the default settings. If you have problems with getting DDD to run, run it with --check-configuration to probe for necessary libraries etc. DDD has many advanced features, see the manual in erc32/doc (ddd.pdf) or the on-line manual under the ‘Help’ menu.

4.5 Remote target monitor

The directory `src/rdbmon` contains the remote monitor which needs to be running on the target board to allow remote target debugging with gdb. The monitor supports ‘break-in’ into a running program by pressing Ctrl-C in GDB or **interrupt** in DDD. The two timers are stopped during monitor operation to preserve the notion of time for the application. Note that the remote debugger monitor only works with programs compiled with ERC32CCS, and thus NOT with programs compiled with Aonix Ada, VxWorks or similar.

Type make to build the monitor. Depending on desired baudrate type either ‘make m38k4’, ‘make m19k2’ or ‘make m9k6’. Program the resulting *.srec file to you boot-prom. The remote debugger will be attached via UART B, console is on UART A. The maximum baudrate depends on the system clock of the target, 38K4 has been successfully used with a zero-waitstate ERC32 system running at 10 MHz. The monitor installs it self into the top 32K of ram. It therefore needs to know how large the ram is. The default ram size for the monitor is 2 Mbyte, adjust the Makefile if your system has different size.

5 Internals (sequential C-programs)

Below is some information you might need if you wish to modify the way sequential C-programs are built.

5.1 Memory allocation

The resulting executables are in a.out format and has three segments; text, data and bss. The text segment is at address 0x2000000, followed immediately by the data and bss segments. The stack starts at top-of-ram and extends downwards.

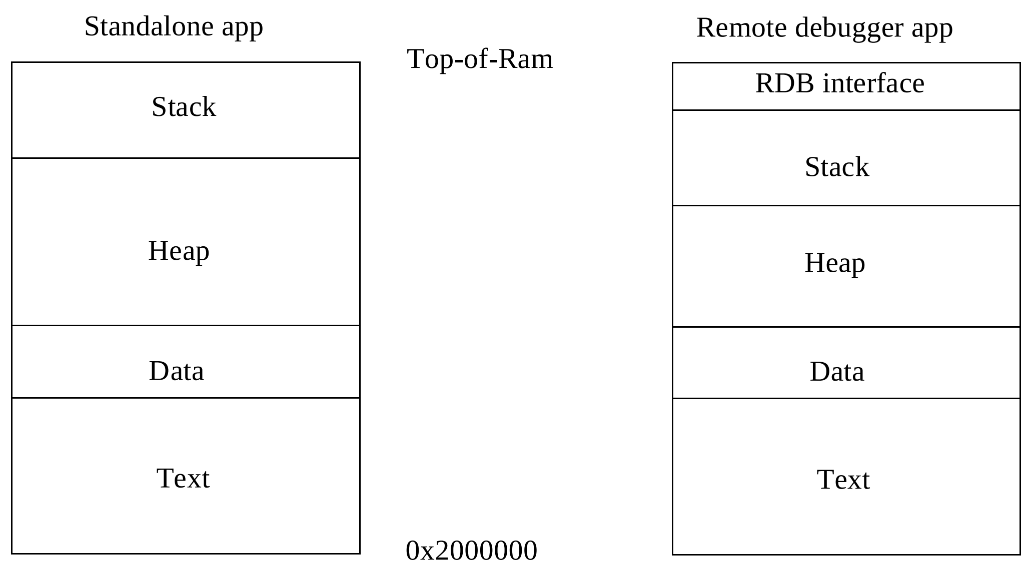

Figure 1: ERC32CCS applications memory map

The link script ram.M in erc32/share/sparc-erc32-aout/lib contains the setting for the available memory. The default setting is 2Mbyte. The applications are not compiled for a specific ram size, the initialisation sequence in the boot-prom (or remote target monitor) will set the top of stack to the highest available memory. The area between the data segment and the stack is used for the heap.

5.2 Libraries

A posix compatible C-library and math library is provided with ERC32CCS. However, no file or other I/O related functions will work, with the exception of I/O to stdin/stdout. Stdin/stdout are mapped on UART A, accessible via the usual stdio functions. UART B can be accessed via file handle 3 (input) or 4 (output). The following function call will write `size` character from `buf` to UART B:

```
write(4,buf,size);
```

At startup of a program, the MEC real-time counter is programmed to increment one per microsecond. The function `clock()` will return the value of the counter. The sources to the board-specific part of the C-library is provided in `erc32/src/libio`. A user can modify the I/O functions according to his needs and install them into the C-library location (`erc32/sparc-rtems/lib`).

6 MkProm version 1.1 manual

NAME

mkprom

SYNOPSYS

mkprom [**-baud** *baudrate*] [**-wdog**] [**-nocomp**] [**-noedac**] [**-nopar**] [**-dump**] [**-freq** *system_clock*] [**-noprot**] [**-o** *filename*] [**-ramsize** *size*] [**-romws** *ws*] [**-romsize** *size*] [**-ramcs** *chip_selects*] [**-ramws** *ws*] *input_files*

DESCRIPTION

The MkProm utility is used to create boot-images for programs compiled with ERC32CCS. It encapsulates the application in a loader suitable to be placed in a boot prom. The application is compressed with a modified LZSS algorithm, typically achieving a compression factor of 2. The loader initiates the system according to the specified parameters. The loader operates in the following steps:

- The register files of IU and FPU (if present) are washed to initialise register parity bits.
- The MEC control, waitstate and memory configuration registers are set according to the specified options.
- The top 32K of the ram is written to initiate the EDAC checksums.
- Part of the loader is moved to the top of ram to speed up operation.
- The remaining ram is written and the application is decompressed and installed.
- The text part of the application is write protected, except the lower 4K where the traptable is assumed to reside.
- Finally, the application is started, setting the stack pointer to the top of ram.

OPTIONS

-baud baudrate

Set UART A and B baudrate to baudrate. Takes into account the system clock. Default value is 19200.

- wdog

Enables the watchdog. By default, the watchdog is disabled.

- nocomp

Don’t compress application. Decreases loading time on the expense of rom size.

-noedac

Disable EDAC. By default, EDAC and parity checking of the ram is enabled.

-nopar

Disable RAM parity checking. Note, never do this if you do have parity on your target board since it could result in driver collision on the DPARIO line.

-freq *system_clock*

Defines the system clock in MHz. This value is used to calculate the divider value for the baud rate generator and the real-time clock. Default is 10.

-noprot

Disable memory write protection. by default, the applications text segment is write-protected against accidental over-write.

- o *outfile*

Put the resulting image in *outfile*, rather then prom.out (default).

-ramsize *size*

Set the ramsize in the memory configuration register to *size*. The default value is 0x200000 (2 Mbyte).

-ramcs *chip_selects*

Set the number of ram banks to *chip_selects*. Default is 1.

-ramws *ws*

Set the number of waitstates during ram writes to *ws*. Default is 0. Ram reads are always zero-wait-state.

-romsize *size*

Set the rom size to *size*. Default is 0x80000 (512 KByte)

-romws *ws*

Set the number of rom waitstates during read to *ws*. Default is 2;

-v

Be verbose; reports compression statistics and compile commands

-dump

The intermediate assembly code with the compressed application and the MEC register values is put in `dump.s` (only for debugging of `mkprom`).

input_files

The input files must be in aout format. If more than one file is specified, all files are loaded by the loader and control is transferred to the first segment of the first file.

7 SIS version 2.7.6 manual

NAME

sis - SPARC instruction set simulator version 2.7.6

SYNOPSYS

sis, sis64 [**-c** *file*] [**-nfp**] [**-ift**] [**-wrp**] [**-rom8**] [**-sparclite**] [**-uart1** *device*] [**-uart2** *device*] [**-freq** *system_clock*] *input_files*

sparc-rtems-gdb *input_files*
sparc-rtems-gdb64

DESCRIPTION

The SIS is a SPARC V7 architecture simulator configured to emulates an ERC32 system with 4 Mbyte ram and 512 Kbyte rom. Two versions of the simulator is provided; sis and sis64. The standard version (**sis**) uses 32-bit time and has a simulation time limit of 2^{32} clock ticks (= 5 minutes at 14 MHz).

COMMANDS

Below is description of commands that are recognized by the simulator. The command-line is parsed using GNU readline. A command history of 64 commands is maintained. Use the up/down arrows to recall previous commands. For more details, see the readline documentation.

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

+**bp** *address* Adds an breakpoint at *address*.

bp Prints all breakpoints

-bp *num* Deletes breakpoint *num*.

cont [*count*]

tcont *[time]* Continue execution at present position, optionally for *count* instructions or for *time* time.

dis [*addr*] [*count*]

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

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

float Prints the FPU registers

go [*address*] [*count*]

tgo [*address*] [*time*]

The **go** command will set pc to *address* and npc to *address* + 4, and resume execution. No other initialisation will be done. If *count* is specified, execution will stop after the specified number of instructions. If address is not given, the default load address will be assumed. The **tgo** command will optionally resume execution until *time* is reached. See **tlim** on how to specify the time.

help Print a small help menu for the SIS commands.

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

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

mem [*addr*] [*count*]

Display memory at *addr* for *count* bytes. Same default values as for **dis**.

quit Exits the simulator.

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

reg [*reg_name value*]

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

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

run [*count*]

trun [*time*] Resets the simulator and starts execution from address 0. If an instruction *count* is given, the simulator will stop after the specified number of instructions. The event queue is emptied but any set breakpoints remain. **trun** command will execute until *time* is reached. See **tlim** on how to specify the time.

step Equal to **trace** 1.

tlim <*time*> Limit the simulated time. Will stop a running simulator after *time*. The time parameter is relative to the current time. The time is given in micro-seconds, but can also be given in milli-seconds, seconds or minutes by adding ‘ms’, ‘s’ or ‘min’ to the time expression. Example: tlim 400 ms.

tra [*count*] Starts the simulator at the present position and prints each instruction it executes. If an *count* is given, the simulator will stop after the specified number of instructions.

Typing a ‘Ctrl-C’ will interrupt a running simulator.

Short forms of the commands are allowed, e.g **c**, **co**, or **con**, are all interpreted as **cont**.

TIMING

The SIS emulates the behaviour of the TSC691E and TSC692E SPARC IU and FPU from Temic/MHS. These are roughly equivalent to the Cypress 7C601 and 7C602. The simulator is cycle true, i.e a simulator time is maintained and incremented according the IU and FPU instruction timing. The parallel execution between the IU and FPU is modelled, as well as stalls due to operand dependencies (IU & FPU). Tracing using the **trace** command will display the current simulator time in the left column. This time indicates when the instruction is fetched. If a dependency is detected, the following fetch will be delayed until the conflict is resolved. Below is a table describing the instruction timing with no resource dependencies:

Instruction	Cycles	Instruction	Cycles
jmpl, rett	2	sqrts	37
load	2	fsqrtd	65
store	3	fsubs	4
load double	3	fsubd	4
store double	4	fdtoi	7
other integer inst	1	fdots	3
fabs	2	fitos	6
fadds	4	fitod	6
faddd	4	fstod	2
fcmps	4		
fcmpd	4		
fdivs	20		
fdivd	35		
fmovs	2		
fmuls	5		
fmuld	9		
fnegs	2		

FPU

The simulator maps floating-point operations on the hosts floating point capabilities. This means that accuracy and generation of IEEE exceptions is host dependent.

MEC EMULATION

The following list outlines the implemented MEC registers:

Register	Address	Status
MEC control register	0x01f80000	implemented
Software reset register	0x01f80004	implemented
Power-down register	0x01f80008	implemented
Memory configuration register	0x01f80010	partly implemented
IO configuration register	0x01f80014	implemented
Waitstate configuration register	0x01f80018	implemented
Access protection base register 1	0x01f80020	implemented
Access protection end register 1	0x01f80024	implemented
Access protection base register 2	0x01f80028	implemented

Access protection end register 2	0x01f8002c	implemented
Interrupt shape register	0x01f80044	implemented
Interrupt pending register	0x01f80048	implemented
Interrupt mask register	0x01f8004c	implemented
Interrupt clear register	0x01f80050	implemented
Interrupt force register	0x01f80054	implemented
Watchdog acknowledge register	0x01f80060	implemented
Watchdog trap door register	0x01f80064	implemented
RTC counter register	0x01f80080	implemented
RTC counter program register	0x01f80080	implemented
RTC scaler register	0x01f80084	implemented
RTC scaler program register	0x01f80084	implemented
GPT counter register	0x01f80088	implemented
GPT counter program register	0x01f80088	implemented
GPT scaler register	0x01f8008c	implemented
GPT scaler program register	0x01f8008c	implemented
Timer control register	0x01f80098	implemented
System fault status register	0x01f800A0	implemented
First failing address register	0x01f800A4	implemented
Error and reset status register	0x01f800B0	implemented
Test control register	0x01f800D0	implemented
UART A RX/TX register	0x01f800E0	implemented
UART B RX/TX register	0x01f800E4	implemented
UART status register	0x01f800E8	implemented

UARTS

The UART transmitters operate at inf

MEMORY EMULATION

The following memory areas are valid for the ERC32 simulator:

0x00000000 - 0x00080000	ROM (512 Kbyte)
0x02000000 - 0x02400000	RAM (4 Mbyte)
0x01f80000 - 0x01f80100	MEC registers

Access to unimplemented MEC registers or non-existing memory will result in a memory exception trap.

The memory configuration register is used to define available memory in the system. The fields RSIZ and PSIZ are used to set RAM and ROM size, the remaining fields are not used. NOTE: after reset, the MEC is set to decode 128 Kbyte of ROM and 256 Kbyte of RAM. The memory configuration register has to be updated to reflect the available memory.

The waitstate configuration register is used to generate waitstates. This register must also be updated with the correct configuration after reset.

GDB-INTEGRATED SIS

To use the GDB-integrated simulator (**gdb** or **gdb64**), use the ‘target sim’ command at the gdb prompt. The only valid options for gdb are **-rom8**, **-nfp**, **-freq**, **-v**, **-sparclite**, **-uben** and **-nogdb**. GDB inserts breakpoints in the form of the ‘ta 1’ instruction. The GDB-integrated simulator will therefore recognize the breakpoint instruction and return control to GDB. If the application uses ‘ta 1’, the breakpoint detection can be disabled with the -nogdb switch. In this case however, GDB breakpoints will not work.

Before control is left to GDB, all register windows are flushed out to the stack. Starting after the invalid window, flushing all windows up to, and including the current window. This allows GDB to do backtraces and look at local variables for frames that are still in the register windows. Note that strictly speaking, this behaviour is **wrong** for several reasons. First, it doesn’t use the window overflow handlers. It therefore assumes standard frame layouts and window handling policies. Second, it changes system state behind the back of the target program. Typically, this will only create problems when debugging trap handlers. The ‘-nogdb’ switch disables the register flushing as well.