The ERC32 GNU Cross-Compiler System

Version 1.2
June 1997

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. Discussions are provided for the following topics :

- contents and directory structure of ERC32CC S
- compiling and linking ERC32 application s
- usage of SIS and MKPRO M
- debugging ERC32 application with GDB/SI S

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:

- GCC C/C++ compile r
- GNU binary utilities (as, ld, ar, ranlib & size)
- RTEMS real-time kerne l
- Standalone C-librar y
- SIS ERC32 simulato r
- GDB debugge r
- MKPROM boot-prom builde r

1.2 News in version 1.2

This version of ERC32CCS contains the following changes with respect to 1.0 :

- DDD version 2.1, including new documentatio n
- Modified assembler to compensate the FPU rev.B bug s
- Re-compiled floating-point library (libm.a) to incorporate the FPU fixe s
- GDB version 4.16. 1
- SIS version 2.7. 1

1.3 Support

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

2 Installation and directory structur e

2.1 Obtaining ERC32CCS

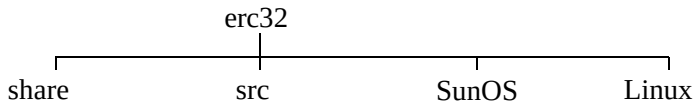
ERC32CCS is only distributed via anonymous ftp. The primary home of ERC32CCS is the ESTEC ftp-server at ftp://ftp.estec.esa.nl/pub/ws/wsd/erc32/erc32ccs. The binary release for the three following platforms is available: Sun (Solaris & SunOs) and PC/Linux. The binary releases can be found in ftp://ftp.estec.esa.nl/pub/ws/wsd/erc32/erc32ccs/binaries. The source of each package is available in ftp://ftp.estec.esa.nl/pub/ws/wsd/erc32/erc32ccs/source. The installation instructions in this document only cover the binary distribution, users wishing to perform a source installation have to rely on the documentation provided in each individual package .

2.2 Installation

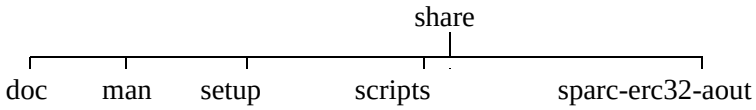
The ERC32CCS directory tree is compiled to reside in /usr/local/erc32. After obtaining the compressed tarfile with the binary distribution, un-compress and un-tar it in a suitable location - if this is not /usr/local/erc32 then a link have to be created to point to the location of the ERC32CCS directory.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

```
gunzip -c erc32ccs-1.2.tar.gz | tar xf -
```

This will create the erc32 directory in the current directory. The ERC32 CCS has the following structure :



Typically, only one of the host-specific directories (Sunos or Linux) is present. Note that the SunOS library contain binaries that run on both Sunos-4.1.x and Solaris 2.x The share directory contains host-independent files and doc u-mentation:



Documentation for the various packages can be found in doc, while unix-style man-pages are in man. setup contains setup files for the supported host platforms while sparc-erc32-* contains RTEMS and C libraries. Note that the d i-rectory structure for ERC32CCS-1.1 has changed and is not compatible with ERC32CCS-1.0.

2.3 Environment

The ERC32CCS uses a number of environment variables to find the proper tools and libraries. The file erc32/share/setup/setup.csh contains an init script for csh. Either put this script in your .cshrc file or source it into your current shell:

```
source /usr/local/erc32/share/setup/setup.csh
```

Users of other shells will have to modify the setup file according to their needs.

2.4 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
sparc-erc32-aout-ar	library archiver
sparc-erc32-aout-as	modified cross-assembler (FPU rev.B fixes)
sparc-erc32-aout-c++	C++ cross-compiler
sparc-erc32-aout-c++filt	utility to demangle C++ symbols
sparc-erc32-aout-g++	same as sparc-erc32-aout-c++
sparc-erc32-aout-gasp	assembler pre-processor
sparc-erc32-aout-gcc	C/C++ cross-compiler
sparc-erc32-aout-gdb	debugger with ERC32 simulator and remote target interface
sparc-erc32-aout-ld	linker
sparc-erc32-aout-nm	utility to print symbol table
sparc-erc32-aout-objcopy	utility to convert between binary formats
sparc-erc32-aout-objdump	utility to dump various parts of executables
sparc-erc32-aout-ranlib	library sorter
sparc-erc32-aout-sis	ERC32 simulator
sparc-erc32-aout-size	utility to display segment sizes
sparc-erc32-aout-strings	utility to dump strings from executables
sparc-erc32-aout-strip	utility to remove symbol table
unprotoize	GNU unprotoize utility

Documentation on how to invoke the tools can be found in erc32/share/doc and erc32/share/man.

3 Compilation

3.1 Development flow

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

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

The ERC32CCS-1.1 supports three types of applications; ordinary sequential C-programs, multi-tasking real-time programs based on the RTEMS kernel and Ada-95 programs. Compiling and linking of sequential C-programs and Ada-95 is done pretty much the same as with the host-based gcc and GNAT. RTEMS applications are built through the use of an RTEMS-specific makefile .

3.2 Compiling sequential C-programs

The various compilation switches are explained in the gcc manual (gcc.ps) and the man-pages. The only real difference is during linking, where a linker file is used to build an executable for ERC32. Compiling and linking is easiest done through the compiler driver. The following example compiles and link an application in 'hello.c '

```
sparc-erc32-aout-gcc -N -Tram.M hello.c -o hello.exe
```

The -N switch is needed to concatenate the data and text segments. The link script in ram.M will include all necessary startup files and libraries.

3.3 RTEMS applications

RTEMS applications are built through a RTEMS-specific makefile. The ERC32CCS contains a pre-compiled RTEMS kernel in erc32/src/rtems. There are several sample applications provided, use those as a template for new RTEMS applications. The sample applications are found in rtems/c/src/tests. Refer to the RTEMS manuals (er32/share/doc/rtems) for more details.

3.4 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 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 .

3.5 Examples

Following example compiles the famous 'hello world' program and creates a boot-prom :

```
tellus > sparc-erc32-aout-gcc -N -Tram.M -g -O2 hello.c -o hello
```

```
tellus > mkprom hello

loading hello:
section .text at 0x02000000 (26088 bytes)
Uncoded stream length: 26088 bytes
Coded stream length: 13534 bytes
Compression Ratio: 1.928

section .data at 0x020065e8 (1232 bytes)
Uncoded stream length: 1232 bytes
Coded stream length: 431 bytes
Compression Ratio: 2.858

section .bss at 0x02006ab8 (40 bytes)(not loaded)
tellus> sparc-erc32-aout-objcopy -v -O srec prom.out hello.srec
copy from prom.out(a.out-sunos-big) to hello.srec(srec)
tellus>
```

In the following example, we build a sample RTEMS application:

```
tellus > cd ../rtems/c/src/tests/samples/ticker/
tellus > make
test -d o-erc32 || /bin/mkdir -p o-erc32
/usr/local/erc32/SunOS/bin/sparc-erc32-aout-gcc -O4 -mno-v8 -Wall -pipe -ansi -fasm -g -nostdinc -
DRTEMS_NEWLIB -DMALLOC_PROVIDED -DNO_TABLE_MOVE
.
.          (text deleted)
.
/usr/local/erc32/SunOS/bin/sparc-erc32-aout-ld -u _sbrk -N -T /usr/local/SunOS/bin/sparc-erc32-aout-nm -g -n o-
erc32/ticker.exe > o-erc32/ticker.num
/usr/local/erc32/SunOS/bin/sparc-erc32-aout-size o-erc32/ticker.exe
text    data    bss     dec     hex     filename
79072   2840    19840   101752  18d78   o-erc32/ticker.exe
tellus > mkprom -o ticker.srec o-erc32/ticker.exe

loading o-erc32/ticker.exe:
section .text at 0x02000000 (79072 bytes)
Uncoded stream length: 79072 bytes
Coded stream length: 40248 bytes
Compression Ratio: 1.965

section .data at 0x020134e0 (2840 bytes)
Uncoded stream length: 2840 bytes
Coded stream length: 1049 bytes
Compression Ratio: 2.707

section .bss at 0x02013ff8 (19840 bytes)(not loaded)
```

3.6 Real-time clock initialisation

To let the real-time clock generate the correct time base, the actual board frequency is passed to the program via the memory location at %tbr[0x7e0]. When running programs on top of sparcmon, this is not done and the default fr e-quency 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
```

3.7 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 s e-quences. 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 M k-Prom. The following example shows how the 'hello world' program is run :

```
tellus > sparc-erc32-aout-sis hello

loading hello:
section .text at 0x02000000 (26088 bytes)
section .data at 0x020065e8 (1232 bytes)
section .bss at 0x02006ab8 (40 bytes)(not loaded)
serial port A on stdin/stdout
sis> go 0x2000000
Hello world
IU in error mode (257)
    2708  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-erc32-aout-sis hello.srec

loading hello.srec:

section .sec1 at 0x00004020 (8 bytes)
section .sec2 at 0x00000000 (16416 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)
  2788502  02000800  91d02000  ta   0
sis>
```

4.2 GDB with simulator

To do symbolic debugging of target applications, use gdb. After gdb is started, the simulator has to attached and the program loaded. Below is a sample session :

```
tellus > sparc-erc32-aout-gdb hello
GDB is free software and you are welcome to distribute copies of it
under certain conditions; type “show copying” to see the conditions.
There is absolutely no warranty for GDB; type “show warranty” for details.
GDB 4.15.1 (sparc-sun-sunos4.1.3_U1 --target sparc-erc32-aout),
Copyright 1995 Free Software Foundation, Inc..
(gdb) tar sim
 SIS - SPARC instruction simulator 2.5
 Bug-reports to Jiri Gaisler ESA/ESTEC (jgais@wd.estec.esa.nl)
serial port A on stdin/stdout
Connected to the simulator.
(gdb) lo
loading /users/wdi/jgais/erc32/src/test/hello :
section .text at 0x02000000 (26088 bytes)
section .data at 0x020065e8 (1232 bytes)
section .bss at 0x02006ab8 (40 bytes)(not loaded)
(gdb) break main
Breakpoint 1 at 0x20014bc: file hello.c, line 3.
(gdb) run
Starting program: /users/wdi/jgais/erc32/src/test/hello

Breakpoint 1, main () at hello.c:3
3           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. However, 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-erc32-aout-gdb hello
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type “show copying” to see the conditions.
There is absolutely no warranty for GDB; type “show warranty” for details.
GDB 4.15.1 (sparc-sun-sunos4.1.3_U1 --target sparc-erc32-aout),
Copyright 1995 Free Software Foundation, Inc..
(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 0x200000 0
Loading section .data, size 0x4d0 vma 0x20065e 8
(gdb) bre main
Breakpoint 1 at 0x20014bc: file hello.c, line 3 .
(gdb) run
Starting program: /users/wdi/jgais/erc32/src/test/hello
main () at hello.c:3
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-erc32-aout-gdb --attach-windo w
```

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 .

4.5 Remote target monitor

The directory erc32/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.

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, co n-sole 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 and link file (rdmon.M) 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.

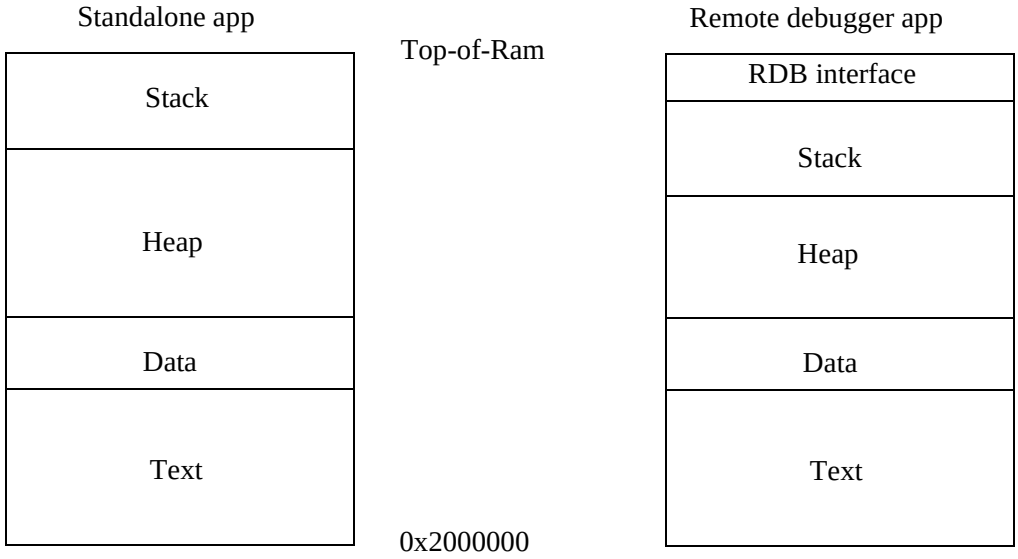


Figure 1: ERC32CCS applications memory map

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/share/sparc-erc32-aout/lib).

6 MkProm manual

NAME

mkprom

SYNOPSYS

mkprom [**-baud** *baudrate*] [**-wdog**] [**-nocomp**] [**-noedac**] [**-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 washed to initiate the EDAC checksums.
- Part of the loader is moved to the top of ram to speed up operation.
- The remaining ram is washed 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.

-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-wai t-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;

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 manual

NAME

sis (version 2.6)

SYNOPSYS

sparc-erc32-aout-sis [**-c** *file*] [**-nfp**] [**-ift**] [**-wrp**] [**-rom8**] [**-sparclite**] [**-uart1** *device*] [**-uart2** *device*] [**-freq** *system_clock*] *input_files*

DESCRIPTION

The SIS is a SPARC V7 architecture simulator. It consist of two parts, the simulator core and a user defined memory module. The simulator core executes the instructions while the memory module emulates memory and peripherals. The integrated memory module emulates an ERC32 system with IU, FPU, MEC, 4 Mbyte ram and 512 Kbyte rom.

OPTIONS

-c *file*

Reads commands from *file* instead of stdin.

-nfp

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

-wrp

Sets the prom to be writable (ROMWRT input on the MEC) .

-rom8

By default, the prom area is considered to be 32-bit. Specifying -rom8 will simulate a 8-bit prom. The only visible difference is in the instruction timing .

-sparclite

Enables two sparclite instructions, SMUL and DIVSCC .

-uart[1,2] *device*

By default, UART A is connected to stdin/stdout while UART B is disconnected. This switch can be used to connect the uarts to other devices. E.g., '-uart1 /dev/ptypc' will attach UART A to pse u-do-device ptypc. Use 'tip /dev/ttypc' to connect to i t

-freq *system_clock*

Set the emulated system clock (MHz). Only used for the performance report. Default is 14 .

input_files

Each input file is loaded into the emulated memory according to the entry point for each segment. Recognized formats are aout, srecords and tektronix hex .

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

dis [*addr*] [*count*]

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

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

float Prints the FPU registers

go [*address* [*count*]]

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. With out address, execution will start at the program load point.

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. **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.

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

run [*count*] 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.

step Equal to **trace** 1.

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 infinite speed, which means that the transmitter holding register always is empty and a transmitter empty interrupt is generated directly after each write to the transmitter data register. The receivers can never overflow or generate errors .

INTERRUPT CONTROLLER

External interrupts are not implemented, so the interrupt shape register has no function. The only internal interrupts that are generated are the real-time clock, the general purpose timer and UARTs. All 15 interrupts

can be tested via the interrupt force register .

WATCHDOG

The watchdog clock is always the system clock regardless of WDCS bit in MEC configuration register .

POWER-DOWN MODE

The power-down register (0x01f800008) is implemented as in the specification. However, if the simulator event queue is empty, power-down mode is not entered since no interrupt would be generated to exit from the mode. A Ctrl-C in the simulator window will exit the power-down mode. The simulator runs at least 100 times faster in power-down mode .

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, use the ‘target sim’ command at the gdb prompt. The only valid options for gdb are **-nfp**, **-freq**, **-v**, **-sparclite** 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 .