intel

i860™ 64-Bit Microprocessor Simulator and Debugger Reference Manual

Version 3

January 1990

240437-003

Intel Corporation makes no warranty for the use of its products and assumes no responsibility for any errors which may appear in this document nor does it make a commitment to update the information contained herein.

Intel retains the right to make changes to these specifications at any time, without notice.

Contact your local sales office to obtain the latest specifications before placing your order.

The following are trademarks of Intel Corporation and may only be used to identify Intel products:

376, 386, 387, 486, 4-SITE, Above, ACE51, ACE96, ACE186, ACE196, ACE960, BITBUS, COMMputer, CREDIT, Data Pipeline, DVI, ETOX, FaxBACK, Genius, i, $\hat{\imath}$, i486, i750, i860, ICE, iCEL, ICEVIEW, iCS, iDBP, iDIS, I^2ICE, iLBX, iMDDX, iMMX, Inboard, Insite, Intel, int$_e$l, Intel386, int$_e$lBOS, Intel Certified, Intelevision, int$_e$ligent Identifier, int$_e$ligent Programming, Intellec, Intellink, iOSP, iPAT, iPDS, iPSC, iRMK, iRMX, iSBC, iSBX, iSDM, iSXM, Library Manager, MAPNET, MCS, Megachassis, MICROMAINFRAME, MULTIBUS, MULTICHANNEL, MULTIMODULE, MultiSERVER, ONCE, OpenNET, OTP, Pro750, PROMPT, Promware, QUEST, QueX, Quick-Erase, Quick-Pulse Programming, Ripplemode, RMX/80, RUPI, Seamless, SLD, SugarCube, ToolTALK, UPI, Visual Edge, VLSiCEL, and ZapCode, and the combination of ICE, iCS, iRMX, iSBC, iSBX, iSXM, MCS, or UPI and a numerical suffix.

MDS is an ordering code only and is not used as a product name or trademark. MDS® is a registered trademark of Mohawk Data Sciences Corporation.

MULTIBUS is a patented Intel bus.

CHMOS and HMOS are patented processes of Intel Corp.

Intel Corporation and Intel's FASTPATH are not affiliated with Kinetics, a division of Excelan, Inc. or its FASTPATH trademark or products.

OS/2 and AIX are trademarks and Personal System/2 and IBM are registered trademarks of International Business Machines Corporation.

UNIX is a registered trademark of AT&T.

Additional copies of this manual or other Intel literature may be obtained from:

Intel Corporation
Literature Sales
P.O. Box 7641
Mt. Prospect, IL 60056-7641

CG-010290

Preface

The Intel i860™ Microprocessor delivers supercomputing performance in a single VLSI component. The 64-bit design of the i860 Microprocessor balances integer, floating point, and graphics performance for applications such as engineering workstations, scientific computing, 3-D graphics workstations, and multiuser systems. Its parallel architecture achieves high throughput with RISC design techniques, pipelined processing units, wide data paths, large on-chip caches, million-transistor design, and fast one-micron CHMOS IV silicon technology.

The i860 Debugger is an interactive, symbolic debugging tool for applications for the i860 Microprocessor. The Debugger can be used with or without an i860 Microprocessor in the system. The i860 Simulator takes the place of the i860 Microprocessor for debugging when the hardware is unavailable.

This manual is a complete source of information about the use of both the Debugger and the Simulator. It is useful both to software managers and software engineers. Software managers will learn how these tools can accelerate development of applications for the i860 Microprocessor. Software engineers will find detailed descriptions of the use of these tools.

System Requirements

The software documented in this manual executes in the following environments:

- UNIX System V/386, release 3.2
- OS/2 version 1.1 or later
- AIX version 1.1 or later

Prerequisites

Because the Debugger deals with machine-level instructions, you should already be familiar with the architecture and instruction set of the i860 Microprocessor as presented in the *i860 64-Bit Microprocessor Programmer's Reference Manual.*

How to Use This Manual

- Chapter 1, "Introduction" presents an overview of both the Debugger and the Simulator and identifies the relationship between the two.
- Chapter 2, "Using the Debugger" defines all the Debugger's user interfaces.
- Chapter 3, "Using the Simulator" explains all the user interfaces with the Simulator.
- Chapter 4, "Example Debugging Session" illustrates the use of both the Debugger and Simulator for debugging an actual program.
- Appendix A, "Debugger Quick Command Reference" provides a convenient list of commonly used commands.

Related Documentation

The following books contain additional material concerning the i860 Microprocessor:

- *i860 64-Bit Microprocessor* (Data Sheet), order number 240296
- *i860 64-Bit Microprocessor Programmer's Reference Manual,* order number 240329
- *i860 64-Bit Microprocessor Assembler and Linker Reference Manual,* order number 240436

Notation and Conventions

gbold	When a name, symbol, or other sequence of characters is used in exactly the same way that it is intended to be entered into the Simulator, Debugger, or other software product, it is printed in a contrasting type style, gothic bold; for example, **dirbase**.
gitalic	Gothic italic type indicates a metasymbol that is to be replaced with an item that fulfills the rules for that symbol; for example, ***objfil*** is to be replaced by an object-file identifier that fulfills the rules for file names.
[ ]	Brackets indicate optional arguments or parameters.
[ ]	Brackets, when printed in the contrasting type style, are required, and must be entered as shown.
{ }	One and only one of the enclosed entries must be selected unless the field is also surrounded by brackets, in which case it is optional.
{ }...	At least one of the enclosed entries must be selected unless the field is also surrounded by brackets, in which case it is optional. The items may be used in any order, unless otherwise noted.

...	Ellipses indicate that the preceding argument or parameter may be repeated. When an ellipsis follows a right bracket or brace, the entire unit enclosed by the brackets or braces may be repeated.
<^CHAR>	Angled brackets surround characters that do not appear on the screen when typed. A circumflex (**^**) represents the control key (usually labeled **Ctrl**). To type <**^D**>, for example, hold down the control key while you type the **D** key.
.,/$	Punctuation other than brackets and braces must be entered as shown.

Table of Contents

Tables

Examples

Chapter 1
Introduction to the Simulator and Debugger

1.1 The Debugger

The i860 Microprocessor Debugger is the common user-interface module for debugging either with a real i860 Microprocessor or with the Simulator module. The Debugger accepts debugging requests and executes them by manipulating the program being debugged. With the i860 Microprocessor Debugger, programmers can:

- Refer to memory and registers via symbolic names.
- Examine and set user and simulator registers (when used with the Simulator module).
- Examine and set user memory sections with data in various formats.
- Display pipeline stages.
- Set code and data breakpoints in user programs.
- Disassemble instructions symbolically.
- Display call frames from the stack.
- Single step programs, optionally stepping over procedure calls.
- Debug high-level language programs at the source-code level.

Debugger requests are similar to those of the UNIX debugger **adb**. Programmers familiar with that environment can start debugging immediately.

1.2 The Simulator

The Simulator plays the role of the i860 Microprocessor in systems that do not contain an i860 Microprocessor. It is an instruction simulator that enables software developers to run complete applications for the i860 Microprocessor and examine their performance and correctness.

The Simulator uses the Debugger as its human interface, giving programmers the ability to examine and modify the simulated processor registers and memory, to set code and data breakpoints, to single-step, etc.

The Simulator supports all instructions, interrupt mechanism, memory management, caches, floating-point arithmetic and exceptions. In addition, it provides a means of measuring the cache-hit ratio and the performance of whole programs or arbitrary portions of programs.

The Simulator supplies systems services to programs being simulated. Applications programs running on the Simulator can call systems services such as file manipulation.

Note that the simulator requires that a numerics coprocessor be present in the system.

Chapter 2
Using the Debugger

The interface with the Debugger is largely the same, regardless of whether the Debugger is combined with the Simulator or with an actual i860 Microprocessor.

2.1 Command-Line Syntax

When the debugger is part of the Simulator, it is activated by invoking the Simulator with a command of the form:

sim860 [**-d**] [**-w**] [**-R**] [**-r**[*reg_type*]] [**-I***dir*] [**-S** *dir*[~*dir*]...] [**-F**[*cmd_file*]] *objfil* [*parms*]

Options must be separated from one another by one or more spaces.

The *objfil* is an executable program file or a data file[1].

The **-d** option tells the Simulator to start an interactive debugging session, reading commands from the standard input. If **-d** is not specified, the Simulator executes the program as a batch session, i.e., without further user interaction.

The **-w** option opens *objfile* for reading and writing, so that this file can be modified by the Debugger.

The **-R** option instructs the Debugger to display all processor and Simulator registers when a batch session is terminated. This aids in testing when comparing the results of the current run with previous runs of the same program.

The **-r** option displays the values of selected registers when a batch session is terminated. It may be followed (without intervening space) by a one-character *reg_type*, specifying the type of registers to display. Only one **-r** option can be specified. The register types are:

No type	Integer registers and control registers.
f	Floating point registers as single-precision.
d	Pairs of floating-point registers as double-precision.
i	Pairs of floating-point registers as 64-bit integers.
p	Pipelines.
s	Simulator registers.

The **-I** option specifies the directory that contains files to be read with the commands **$<** or **$<<**. The default is the current directory. (No space is permitted between **-I** and *dir*.)

1. One might ask, "Why would I want to debug a data file?" A data file cannot be "debugged," as such; however, the Debugger's commands can be used to search, modify, and create a formatted listing for a data file.

The **-S** option provides a list of directories to be used during source debugging when searching for source files. The current directory is automatically added at the beginning of the list.

The **-F** argument tells the Debugger to execute the requests found in *cmd_file* immediately when starting the debug session. The *cmd_file* usually contains commands to set Simulator parameters, to set standard breakpoints, and to initialize processor registers. The Debugger searches for this file in the same directory specified by the **-I** option. The **-F** option is in effect only when the **-d** argument is specified. When *cmd_file* has been exhausted and no **$q** command has been encountered, the debugger displays its prompt and begins the interactive session. If no *cmd_file* is specified with the **-F** option, a file name of *objfil*.**cmd** is assumed. (No space is permitted between **-F** and *cmd_file*.)

The optional parms are command-line parameters to be passed to the program in *objfil* when executed in batch mode. If the **-d** option is specified, then these parameters are ignored. Any command-line parameters for the object program must follow *objfil*; options for the Debugger must precede *objfil*.

2.2 Requests

The general format of a request is:

[*address_exp*] [, *count_exp*] *command* [*modifier*] [;]

The *address_exp* is usually (but not always) an integer valued expression that specifies a memory address. Some commands require other expression types. These cases are clearly identified. Some commands neither use nor permit an *address_exp*.

The Debugger maintains a *current address* called *dot*, which is a pointer into *objfil* or into the image of the executing program in main memory. If a request needs an address, and no *address_exp* is specified, *dot* is used. Many requests modify *dot* implicitly.

The *count_exp* is an integer-valued expression that specifies the number of times the command is repeated. Some commands neither use nor permit a *count_exp*.

The *modifier* depends on the type of command and is documented with each command.

The semicolon (;) separates one command from another when multiple commands are entered in one line.

Two consecutive slashes (//) introduce a comment. They can appear anywhere in a line; the comment extends from the slashes to the end of the line.

2.2.1 Expressions

Numeric values are represented by expressions. Both 32-bit integer and 32-bit real values are supported. Complex expressions are formed from primitive expressions using operators and parentheses.

2.2.1.1 Primitive Expressions

A numeric value may be identified by any of the following primitive expressions:

. The value of *dot*, an integer.

" The last address used, an integer.

int_con An integer constant of the form [*prefix*]*digit_string*. The *prefix* indicates the radix of the number...

0o or **0O** Octal. The *digit_string* is formed of the digits **01234567**.

0t or **0T** Decimal. The *digit_string* is formed of the digits **0123456789**.

0x or **0X** Hexadecimal. The *digit_string* is formed of the digits **0123456789abcdefABCDEF**.

If no *prefix* is present, the *default radix* is used. Initially the default radix is hexadecimal, but it may be changed by a Debugger request. Note that a hexadecimal number may not begin with an alphabetic character; a leading zero can be added to the *digit_string* if the default radix is hexadecimal. For example, **F0A3** is not valid, but **0F0A3** is valid if the default radix is hexadecimal. The number **0xF0A3** is valid, regardless of the default radix.

integer.fraction[**e**{[+][–]}*exponent*]
A 32-bit real number. The *integer*, *fraction*, and *exponent* are all decimal integer constants without a prefix.

'cccc' A string of up to four ASCII characters. To include ' in the string, enter \'. Other special characters can be entered as a two-character sequence beginning with \ as in the C language **printf** format. The value of this expression is the integer formed by the concatenation of the ASCII codes for the characters.

<name The contents of a register or variable. Section 2.2.2 presents a complete list of valid names. The value may be or type integer or real, depending on the register.

symbol The value of a symbol from the symbol table of *objfil*. A symbol is a string of uppercase alphabetic characters, lowercase alphabetic characters, underscore characters, and digits. A symbol cannot begin with a digit. The value of a symbol is always an integer. Programming tools such as C compilers sometimes add an underscore (_) at the beginning of global symbols. This underscore must be explicitly specified when referring to such symbols.

The wildcard character * may be entered as the last character of an abbreviated symbol string, in which case, the debugger finds a symbol that starts with the specified string. This eliminates the need to write the full name of the symbol. If more than one symbol begins with the string entered, the Debugger arbitrarily chooses one.

2.2.1.2 Operators

Given that exp is an expression, then the following are also expressions:

uop exp — The *uop* is one of the unary operators of Table 2-1.

exp1 bop exp2 — The *bop* is one of the binary operators of Table 2-1. If one expression is real and the other integer, the result is real.

(*exp*) — Paired parentheses may be used freely to clarify or override operator precedence. The value is the value of the *exp*.

In the absence of overriding parentheses, binary operators are evaluated according to the following precedence groups. Group one is the group with highest precedence (the first to be evaluated).

Table 2-1. Operators

Operator Type	Operator	Functions	Operand Type[1]
@exp[2]	Unary	The contents of the location exp in *objfil*.	I
-exp	Unary	Arithmetic negation	I or R
~exp	Unary	Bitwise complement	I
+exp	Unary	(Has no effect.)	I or R
exp1+exp2	Binary	Addition.	I or R
exp1-exp2	Binary	Subtraction	I or R
*exp1*exp2*	Binary	Multiplication.	I or R
exp1%exp2	Binary	Division.	I or R
exp1&exp2	Binary	Bitwise AND	I
exp1|exp2	Binary	Bitwise OR.	I

1 I = integer; R = real.

2 On some terminals using the @ may cause deletion of entered text and jumping to a new line. If so, precede the @ with \

1. ***, %**
2. **+, -**
3. **<, >, =**
4. **&, |**

Operators with the same precedence are evaluated left-to-right. Unary operators have precedence over binary operators.

2.2.1.3 Line-Number Address Expressions

Line-number address expressions permit reference to the memory address that corresponds to a specific line of a source program file. The general format of a line-number address is:

#line_number[*~source_file*]

The Debugger maintains a record of the current source file, so that, if *source_file* is not given, the current source file is assumed. The Debugger searches for *source_file* in the directories specified by the **-S** command-line option.

Note that source-level debugging can be used only if the C or FORTRAN program is complied with the **-g** option to insert debugging information into the object modules. When linking, the **-s** option must **not** be used, as it would strip the debugging information from the object module.

2.2.2 Predefined Names

A name in an *address_exp* or in the *modifier* of the > operator may be a processor register, a Simulator environment register, or a Debugger variable. Some of the registers are divided into fields, in which case the field can be identified as *name.field*.

2.2.2.1 Processor Registers

All of the processor registers are accessible through the Debugger. They can be accessed by the *<name=* and various forms of the **$r** commands or altered with the **>** command.

r31..r0	The integer registers.
f31..f0	The floating-point registers, single precision.
d30..d0	Pairs of floating-point registers, double precision. (Only even numbered register can be specified.)
i30..i0	Pairs of floating-point registers treated as 64-bit integers. (Only even numbered register can be specified.)
t_, ki, kr	The special registers of the dual-operation instructions.
merge_lo, merge_hi	The MERGE register.
fir, psr, epsr, dirbase, db, fsr	The control registers.

The control registers have the fields defined in the *i860 Microprocessor Programmer's Reference Manual*. These are repeated here for convenience:

psr	**br, bw, cc, lcc, im, pim, u, pu, it, in, iat, dat, ft, ds, dim, knf, sc, ps, pm**
epsr	**il, wp, int, dcs, pbm, be, of, pt, sn**
dirbase	**ate, dps, bl, iti, cs8, rb, rc, dtb**
fsr	**fz, ti, rm, u, fte, si, se, mu, mo, mi, ma, au, ao, ai, aa, rr, ae, lrp, irp, mrp, arp**

To access the **fz** field of the **fsr**, for example, enter **fsr.fz**.

2.2.2.2 Simulator Environment Registers

When the Debugger is used with the Simulator, any of the Simulator Environment Registers defined in Chapter 3 can be displayed by the *<name=* and **$rs** commands or altered with the **>** command. Refer to Chapter 3 for a list of these registers along with descriptions of their functions. (When the Debugger is used with an actual i860 Microprocessor, these registers are undefined.)

2.2.2.3 Debugger Variables

The Debugger provides a number of integer variables. These have the predefined, one-character names **a** through **z** and **0**, **1**, **2**, and **9**. They can be accessed by the *<name=* and **$v** commands or altered by the **>** command.

The following variables have predefined functions:

0 The last value displayed.
9 The value of *count_exp* in the last **$<** or **$<<** request.
b The base address of the data section.
d The size of the data section.
e The entry point.
m The "magic number" of the i860 Microprocessor.
s The size of the stack section.
t The size of the text section.

All other variables are available for storing values via the Debugger commands. Their initial values are all zero.

2.2.3 Commands

Within the following commands, **?** refers to the original contents of *objfil* on disk while **/** refers to the executing memory image of *objfil.* The same memory addresses are used in either case. To access *objfil* directly, the debugger translates memory addresses into file-relative disk addresses. (There may be a space after **{?/}**.)

Several of these commands display values according to a format specifier. If the format is not supplied, the Debugger uses the last supplied format. The *format* is a character string defined as {[*repeat_count*]*format_char*}..., where *repeat_count* is an integer constant and *format_char* is one of the characters defined by Table 2-2.

Table 2-2. Formatting Characters

Format Char	Bytes Accessed	Contents and Format of Display	Effect on *dot*
a	-	Symbol that corresponds to *dot*	Does not change *dot*
b	1	Unsigned octal integer	Increment by 1
B	4	**dirbase** format	Increment by 4
c	1	ASCII character	Increment by 1
C	1	ASCII character, but nongraphic characters displayed symbolically	Increment by 1
d	2	Signed decimal integer	Increment by 2
D	4	Signed decimal integer	Increment by 4
f	4	Single-precision floating-point	Increment by 4
F	8	Double-precision floating-point	Increment by 8
i	4	Disassembled machine instruction	Increment by 4
I	-	Source line number that corresponds to *address_exp*	Does not change *dot*
n	-	New-line character sequence	Does not change *dot*
o	2	Unsigned octal integer	Increment by 2
O	4	Unsigned octal integer	Increment by 4
p	-	Symbol that corresponds to *address_exp*	Does not change *dot*
P	4	**psr** format	Increment by 4
q	2	Signed octal integer	Increment by 2
Q	4	Signed octal integer	Increment by 4
r	-	Space character	Does not change *dot*
R	4	**fsr** format	Increment by 4
s	variable	ASCII string terminated by NUL	Increment by length
S	variable	ASCII string terminated by NUL, nongraphic characters displayed symbolically	Increment by length
t	-	Tab to next tab stop*	Does not change *dot*
u	2	Unsigned decimal integer	Increment by 2
U	4	Unsigned decimal integer	Increment by 4
x	2	Hexadecimal integer	Increment by 2
X	4	Hexadecimal integer	Increment by 4
Y	4	Date as seconds since 00:00 GMT 1970	Increment by 4
"strg"		Character string *strg*	Does not change *dot*
^	-	(none)	Decrement by the current format length
+	-	(none)	Increment by one
-	-	(none)	Decrement by one

* The spacing between tab stops is defined by the *repeat_count* that precedes the **t**. For example, **8t** means move to the next tab stop where tab stops are set every eight positions.

?[*format*]	Display the contents of the disk file *objfil* starting at memory address *address_exp* according to *format*.
/[*format*]	Display the contents of memory starting at address *address_exp* according to *format*. This command is valid only after execution has been started (by the **:r**, **:R**, **:s**, **:S**, **:i**, or **:I** commands).
=[*format*]	Display the value of *address_exp* according to format.
>*name*	Assign the value of *address_exp* to *name*. The *name* can refer to any of the processor registers, Debugger variables, or Simulator environment registers. Refer to section 2.2.2. The *address_exp* must have a real value when *name* is a floating-point register.
{**?**/} **l** *value* [*mask*]	Starting at *address_exp* and masking each 16 bits with *mask*, search for the integer *value*. When a match occurs set *dot* to the matched location; otherwise, do not change *dot*. Use a mask of all one bits if *mask* is not supplied.
{**?**/} **L** *value* [*mask*]	Starting at *address_exp* and masking each 32 bits with *mask*, search for the integer *value*. When a match occurs set *dot* to the matched location; otherwise, do not change *dot*. Use a *mask* of all one bits if mask is not supplied.
{**?**/} **w** *value*	Assign the integer *value* to the 16 bits at *address_exp*.
{**?**/} **W** *value*	Assign the integer *value* to the 32 bits at *address_exp*.
?m[*] [*b* [*e* [*f*]]]	Change address mapping parameters. Refer to section 2.3. If * is present, change the data mapping; otherwise, change the text mapping.
:*command modifier*	One of the series of program execution commands. The : commands are defined in section 2.2.3.2.
$*command modifier*	One of the series of debugger control commands. The **$** commands are defined in section 2.2.3.3.
<newline>	Depends on the previous command:
	:s, **:S**, **:i**, **:I** — Repeat with a *count_exp* of one.
	$r*n*, **$rf***n*, **$rd***n*, or **$ri***n* — Display the next higher numbered register of the same type.
	?[*format*] or /[*format*] — Repeat with the value of *dot* as *address_exp*.
! *command_line*	Execute system command *command_line*. Under AIX and UNIX this invokes **/bin/sh**; under OS/2 it invokes **cmd.exe**.
<^C>	Interrupt the executing program and return control to the terminal.

2.2.3.1 Examples of Formats

The formatting requests shown in Example 2-1 were entered while debugging a program that has a 16-entry vector of single-precision floating-point numbers beginning at the symbolic address **sparray**. (Refer to Chapter 4 for even more formatting examples.)

```
*sparray/f // What is the value of first vector entry?
sparray:   1
*sparray/"First element: "f     // Display with caption
sparray:   First element: 1
*sparray/f^X                    // Display once, then back up and redisplay in
hex, too
sparray:   1                    0x3f800000
*sparray+4/// What is the value of the second entry?
sparray+0x4: *sparray/4f        // Display first four elements of vector
sparray:   1                    2               3    4
*sparray,4/4f                   // Display entire vector as 4x4 array
sparray:   1                    2               3    4
           5                    6               7    8
           9                    10              11   12
           13                   14              15   16
*="Every other entry" // Display text only
Every other entry
*=16t"COL 1"16t"COL 2"16t"COL 3"16t"COL 4"
// Text with tabs COL 1         COL 2           COL 3COL 4
*sparray,2/f4+f4+f4+f4+ // Display every other entry
sparray:   1                    3               5    7
           9                    11              13   15
*<r17="Dimension in r17: "UX // Display a register in two formats with caption
Dimension in r17: 16            0x10
*'m'=c16tx // What is the ASCII code for the letter 'm'?
0x6d       Dimension in r17:
0x6d: unable to access data
*sparray=X // What is the address of the array?
           0x1180
*<r16=p    // What is the address in r16 for?
           sparray:
*$q        // Terminate session
Goodbye!
```

Example 2-1. Formatting Examples

2.2.3.2 Program Execution Commands

These commands, which begin with a colon (:), control the execution of a program; i.e., starting and stopping execution, and setting and clearing breakpoints. The following restrictions apply to the placement of breakpoints:

- For instructions that execute in dual-instruction mode, breakpoints can be set only on the core instructions.
- A breakpoint cannot be set on the instruction that follows a delayed branch instruction.

- When delayed branch instructions are used in dual-instruction mode, a breakpoint cannot be set on any of the three instructions that follow the delayed branch.

After a single step or after stopping at a breakpoint, the Debugger normally displays a single instruction, the one that will be executed next. However, when the next instruction is a delayed control transfer (e.g., **call** or **bri**), both that instruction and the following instruction are displayed. A subsequent single-step command executes the control transfer and either executes or skips the second instruction, depending on whether the branch is taken.

When executing in dual-instruction mode, code breakpoints and single stepping (both of which are implemented with the core **trap** instruction) can be effected only on the core instruction of a dual-instruction pair. Therefore, after a single step or after stopping at a breakpoint, the Simulator displays two instructions: the floating-point instruction (which has already been executed) and the corresponding core instruction (which will be executed next).

When in dual-instruction mode and the core instruction is a delayed control transfer, four instructions (two dual-instruction pairs) are displayed. A subsequent single-step command executes the core control transfer and either executes or skips the second instruction pair, depending on whether the branch is taken.

With the following general-purpose commands, *address_exp* can be either or a line-number expression or an integer expression that refers to an address.

:b [*cmd*] — Set a code breakpoint at *address_exp*. If *count_exp* is given, the instruction at *address_exp* is executed *count_exp*–1 times before trapping. The *cmd* is a Debugger command to execute after the trap.

:brd [*cmd*] — Set a data breakpoint for read access at *address_exp*. If *count_exp* is given, the instruction at *address_exp* is executed *count_exp*–1 times before trapping. The *cmd* is a Debugger command to execute after the trap.

:bwr [*cmd*] — Set a data breakpoint for write access at *address_exp*. If *count_exp* is given, the instruction at *address_exp* is executed *count_exp*–1 times before trapping. The *cmd* is a Debugger command to execute after the trap.

:ba [*cmd*] — Set a data breakpoint for either read or write access at *address_exp*. If *count_exp* is given, the instruction at *address_exp* is executed *count_exp*–1 times before trapping. The *cmd* is a Debugger command to execute after the trap.

:c [*sig_num*] — Continue program execution at *address_exp*. Used to continue after a breakpoint. If a *count_exp* is given, skip the next *count_exp*–1 breakpoints. If a *sig_num* of 0t16 is entered, the Simulator emulates an external interrupt, setting the IN bit of **psr**.

:d — If *address_exp* is explicitly entered, delete the breakpoint at *address_exp*. If no *address_exp* is entered, delete all breakpoints.

:r [*parms*] — Run *objfil*. Command-line parameters may be specified after **:r**. The *address_exp*, if given, use it as the entry point, otherwise begin the program at its standard entry point. If a *count_exp* is given, skip the next *count_exp*–1

breakpoints. Reinitializes all program variables but leaves breakpoints, processor registers, and Debugger variables intact. Those Simulator environment registers that are used for performance measurement are cleared.

:s {[*sig_num*] [*parms*]} Single-step *count_exp* times starting at *address_exp*. If the **:r** command is not previously entered, the Debugger implicitly begins execution of *objfil*, in which case *sig_num* is not permitted and the value of *parms* is passed to the program as in the **:r** command. The backslash \ can be used instead of **:s**. If a *sig_num* of 0t16 is entered, the Simulator emulates an external interrupt, setting the IN bit of **psr**.

:S {[*sig_num*] [*parms*]} Similar to **:s**, but steps over procedure calls. Two consecutive backslashes \\ can be used instead of **:S**.

The remaining program execution commands are used solely for source debugging. The *address_exp* must be a line-number address expression.

:a Display the instructions that derived from the line specified by *address_exp*.

:i {[*sig_num*] [*parms*]} Single-step *count_exp* source lines starting at *address_exp*. If the **:r** command is not previously entered, the Debugger implicitly begins execution of *objfil*, in which case *sig_num* is not permitted and the value of *parms* is passed to the program as in the **:r** command. If a *sig_num* of 0t16 is entered, the Simulator emulates an external interrupt, setting the IN bit of **psr**.

:I {[*sig_num*] [*parms*]} Similar to **:i**, but step over procedure calls.

:n Display the next set of source lines using the current display size.

:p Display the previous set of source lines using the current display size.

:w Display *count_exp* lines around the source line specified by *address_exp*, and set the current display size to *count_exp*.

2.2.3.3 Debugger Control Commands

$<[*file*] Read requests from *file*. If file is omitted, revert to reading requests from the terminal. If *count_exp* is zero the **$<** command is ignored; otherwise, the value of *count_exp* is placed in variable **9** before execution of the first command in *file*. If this command is executed from a file of requests, further commands in that file will not be read.

$<<[*file*] Read requests from *file* and return to the current request stream upon termination. If *file* is omitted, return to the prior request stream. Also returns upon exhausting *file*. This command works like a "call" to a "subroutine" of debugging requests. A call can pass one parameter to the subroutine. The parameter is the value of *count_exp*. The subroutine can access the parameter via variable **9**. The prior value

of variable **9** is saved before entering the subroutine and restored after exiting.

$>[*file*] Divert output to *file*. If *file* does not already exist, create it. If *file* is not given, divert output back to the terminal.

$?** Same as **$r.

$b Display breakpoints.

$c Display a call trace by examining the active stack frames. If *address_exp* is given, use it as the address of the current stack frame instead of the contents of **r3**. If *count_exp* is given, display only the first *count_exp* stack frames.

$d Set default radix to decimal.

$e Display names and values of external variables.

$m Display the address mapping parameters. Refer to section 2.3.

$o Set default radix to octal.

$q Quit (exit, terminate) the Debugger. (**<^D>** can be used instead.)

$r Display integer, control, and MERGE registers.

$r*n* Display the integer register specified by *n*.

$rf Display floating-point registers in single precision, along with the T, KI, and KR registers.

$rf*n* Display the floating-point register specified by *n* in single precision.

$rd Display pairs of floating-point registers in double precision, along with the T, KI, and KR registers.

$rd*n* Display the pair of floating-point registers specified by the even number *n* in double precision.

$ri Display pairs of floating-point registers as 64-bit integers, along with the MERGE register.

$ri*n* Display the pair of floating-point registers specified by the even number *n* as a 64-bit integer.

$rp Display pipelines.

$rs Display the Simulator environment registers.

$s Set to *address_exp* the maximum offset between an address and the symbol that is closest to it. Addresses that are farther from any symbol than this offset are displayed as a numeric value rather than a symbolic value (symbol + offset). The default is 255.

$t Display a trace of the last *address_exp* instructions executed by the Simulator. If *address_exp* is not entered, display the last 16 instructions.

$v Display the Debugger variables.

$w Set the page width for output to *address_exp*. The default is 80.

$x Set default radix to hexadecimal.

NOTE

When displaying registers, be aware that some instructions do not store results by the beginning of the next instruction.

2.3 Mapping Addresses to Object File

All addresses used in Debugger commands refer to memory locations, even when the command operates directly on the *objfil* on disk. For this reason, the Debugger implements the concept of *address mapping*.

A *mapping* is a transformation from a memory address to the corresponding location in *objfil*. Let **A** be a memory address, and let **F** be the file address. The mapping is defined in terms of two triplets of parameters: $b1, e1, f1$ and $b2, e2, f2$ as follows:

if $b1 \leq \mathbf{A} \leq e1$ then $\mathbf{F} = \mathbf{A} - b1 + f1$

else if $b2 \leq \mathbf{A} \leq e2$ then $\mathbf{F} = \mathbf{A} - b2 + f2$

else **A** is an invalid address

The first triplet ($b1, e1, f1$) maps addresses in the text section; the second ($b2, e2, f2$) maps addresses in the data section.

For a COFF program file, the Debugger reads the values of the mapping parameters from the file. Their values can be displayed with the **$m** command. Normally there is no need to change them. However, when using the Debugger to process a data file or an object file of another format, it may be useful to change the mapping parameters, using the **?m** command.

Chapter 3
Using the Simulator

Most interaction with the Simulator is mediated by the Debugger module that is bound to the Simulator. A request entered into the Debugger is interpreted by the Debugger, which then fulfills the request by calling Simulator procedures. Simulation consists of:

- Preparing an application for the i860 Microprocessor.
- Invoking the Simulator/Debugger.
- Configuring the Simulator.
- Entering commands into the Debugger.

Invocation of the Simulator/Debugger and entering of Debugger commands are covered in Chapter 2.

3.1 Preparing an Application for the Simulator

The steps for preparing an application for simulation are no different than those for running on a real i860 Microprocessor. The same executable modules can be used for Simulation, for debugging on a real i860 Microprocessor, or for execution on a real i860 Microprocessor without the Debugger.

1. Write the application program modules in a high-level language and/or the assembly language of the i860 Microprocessor.
2. Translate the application program modules using a compiler or assembler for the i860 Microprocessor. The translator may be instructed to generate symbolic debugging information; this is not essential, however.
3. Link the program modules together using the i860 Microprocessor Linker. Systems services can be accessed via the library **libc.a**.
4. Run the application under control of the Debugger. The same object module can be executed either with the Simulator or on an actual i860 Microprocessor.

3.2 The Simulated Machine

The simulated machine is composed of the following components:

- Memory
- Processor Registers
- Pipelines
- Traps

- Simulator Environment Registers
- Data, Instruction, and Address-Translation Caches

3.2.1 Simulator Environment Registers

The Simulator environment registers are used to control the behavior and function of the Simulator. When linked to the Simulator, the Debugger can be used to display and change these registers. The Simulator registers can be assigned new values only before simulation starts, i.e., before an **:r** (or **:s** or **:i**) command. Any value that is assigned to a simulator register during the simulation of a program, takes effect only after the next **:r** command.

3.2.1.1 Simulator Control Register

sim The Simulator control register. Holds configuration options and helps control Simulator operation. The internal fields of the **sim** can be treated as separate one-bit fields, having the following names:

me Memory boundary check enable. When set, the simulator validates all memory references. A reference to memory out of the sections defined by the program causes an error message.

ce Caching enable. When set, the caches of the i860 Microprocessor are simulated.

te Timer enable. When set, the Simulator counts clocks during execution of the program.

log Instruction log enable. When set, the Simulator writes every instruction that it executes into a file with the name *objfil*.**log**. Note that a long simulation can create a *very large* log file.

clog Cache log enable. When **clog** and **ce** are set, the Simulator records all cache hits and misses in *objfil*.**log**.

turbo Fast simulation. When **sim.turbo** is set, many floating-point checks are eliminated, thereby speeding up the simulation.

The following fields of the **sim** register define trap handling. For each trap, if the field is set, such a trap causes execution of the user trap handler (refer to **trap_addr** in section 3.2.1.4). If reset, the corresponding trap causes the debugger to issue an error massage and prompt.

it Instruction trap.
in Interrupt trap.
iat Instruction access trap.
dat Data access trap.
ft Floating-point trap.

3.2.1.2 Timing Control Registers

When **sim.te** is set, the Simulator counts processor clocks during execution of a program. These counts provide an approximation of program performance on an actual i860 Microprocessor. The following Simulator registers assist in obtaining timing data:

total_time — Number of processor clocks used by all instructions of the simulated program.

range_time — Number of processor clocks used by the instructions located between the addresses in **t_start** and **t_end**.

t_start, t_end — Address range that specifies which instructions are to be included in the clock count maintained in **range_time**. Note that if control passes to a subroutine outside the specified range, the time used by that subroutine is **not** included in **range_time**.

mem_spec — The effect of wait states in the memory system can be taken into account by using this register, which contains the three fields **ads2na**, **cas_ws**, **ras_ws**.

t1, t2 — Each of these registers can store one address. Every time the Simulator executes the instruction at one of these addresses, it displays the clock count before and after the instruction. The value is displayed in decimal.

Note that, if breakpoints are set or if single-stepping is used, the clock count is incorrect, due to the extra trap instructions and due to invalidation of the instruction cache when trap instructions are inserted and removed. (However, setting a breakpoint at the end of the program before starting simulation has only an insignificant effect on clock counts. Another method of terminating a program, calling **exit**, has only a slightly less insignificant effect on clock counts.)

Timing is also dependent on the memory subsystem. The Simulator assumes a pipelined bus. The **mem_spec** register allows specification of other memory subsystem parameters. The default values are adequate for measuring performance under the assumption that the memory does not impose delays on the processor. This register can be changed to add memory delays in accord with an anticipated memory implementation. The **mem_spec** register comprises three parameters:

ads2na
This parameter specifies, for a pipelined bus, by how many clocks the NA# signal follows the ADS# signal. The value of **ads2na** cannot be less that 1, and typically is equal to one.

cas_ws and **ras_ws**
DRAM is logically arranged in rows and columns. When the processor issues a request for memory access, both row logic and column logic must be strobed. Usually it takes longer for the row logic to respond. These parameters specify the delay in clocks from the moment a request is issued to the moment the requested data is ready on the bus. There are two cases:

1. In the simpler case, the time from request to data (from ADS# to READY#) is **max(ras_ws, cas_ws)**, because both circuits work in parallel.

2. If a new address is issued and it happens to be in the same row as the last address issued, the i860 Microprocessor issues a NENE# (next near) signal. In this case, only the column logic is activated, because the NENE# notifies the memory controller that the row logic is already in the correct state. The access will require only **cas_ws** clocks.

For the most accurate timings, cache simulation should also be enabled by setting **sim.ce** whenever **sim.te** is set.

3.2.1.3 Cache Control Registers

These registers allow changing the various cache sizes and counting cache hit ratios.

i_cache_size	Size of the simulated instruction cache in bytes.
d_cache_size	Size of the simulated data cache in bytes.
tlb_size	Size of the simulated address-translation cache (TLB) in bytes.
i_chrc	Instruction cache hit ratio counter.
d_chrc	Data cache hit ratio counter.
tlb_chrc	Address translation cache (TLB) hit ratio counter.

3.2.1.4 Trap Control Register

This register controls trap handling.

trap_addr Address of trap handler. A trap handler can be assigned by entering its address in **trap_addr**. This causes the user-written trap handler to be activated by the Simulator when a trap occurs that has been assigned to the user (refer to the **sim** register in section 3.2.1.1).

3.2.1.5 Other Simulator Registers

pc Holds the program counter (current execution address).

num_instr Accumulates the number of machine instructions that have been executed. This normally differs from the value of **total_time**.

3.3 Configuring the Simulator

The configurable options of the Simulator can be combined to suit the changing needs of the development processes. Sometimes it is desirable to sacrifice simulation speed in order to have more Simulator functions active. This happens mainly when enabling the timer, cache, or memory boundary check. In addition, address translation, when enabled, slows down the Simulator. (Paging is usually turned off but may be turned on when the running application sets the ATE bit of **dirbase**). Setting data breakpoints, though not changing the nature of the simulated machine, also reduces execution speed.

3.3.1 Typical Simulation Cases

Following are some typical simulation cases:

1. Verify correctness and debug an application program. Fast simulation, no performance analysis.
2. Verify an application program and analyze its performance, mainly to check speed of numeric algorithms. Though caching is disabled, RAM wait states can be left zero, allowing a fairly accurate time analysis.
3. Verify system-level program. Traps, cache, and paging are enabled. No performance analysis.

Table 3-1. Simulation Cases

Parameter	Case 1	Case 2	Case 3	Case 4	Default
mem_spec.ras_ws	n.a.	D	n.a.	U (2)	0
mem_spec.cas_ws	n.a.	D	n.a.	U (2)	0
mem_spec.ads2na	n.a.	D	n.a.	U (2)	1
sim.te	D	Y	D	Y	N
t_start	n.a.	U	n.a.	U	0
t_end	n.a.	U	n.a.	U	FFFFFFFF
sim.me	D	D	U	U	Y
d_cache_size	n.a.	U	U	U	8 Kbytes
i_cache_size	n.a.	U	U	U	4 Kbytes
tlb_size	n.a.	U	U	U	256 bytes
sim.ce	N	N	Y	Y	N
sim.it	D	D	U (A)	U (A)	S
sim.ft	D	D	U	U	S
sim.iat	D	D	U (A)	U (A)	S
sim.dat	D	D	U (A)	U (A)	S
sim.in	D	D	U (A)	U (A)	S

D The default value should be used.
n.a. Not applicable. Default value is fine (though not used by the Simulator).
Y Enabled.
N Disabled.
A Trap is assigned to application.
S Trap is assigned to the Simulator.
U The value is up to the user. The default value is usually sufficient in this case; however, a typical value accompanies the U abbreviation for use when the default value will not do.

4. Verify system-level program and an application program and analyze performance. May be used to check performance of an application.

For each of the cases above, Table 3-1 specifies the values of all the configurable parameters.

3.3.2 Default Values of Configurable Parameters

Table 3-2 shows the default values of the configurable parameters.

Table 3-2. Configuration Defaults

Parameter	Default	Comments
mem_spec.ras_ws mem_spec.cas_ws	0 0	Allows fair performance analysis of applications that do not handle cache. Applicable only if timer enabled.
sim.te	Disabled	
t_start	0	Applicable only if timer enabled.
t_end	0xFFFFFFFF	Applicable only if timer enabled.
sim.me	Enabled	
sim.ce	Disabled	
d_cache_size	8K	Applicable only if cache enabled.
i_cache_size	4K	Applicable only if cache enabled.
tlb_size	64 entries	Applicable only if cache enabled.
sim.it sim.ft sim.iat sim.dat sim.in	Handled by Simulator	Typical user does not want to handle interrupts.

3.4 Debugging Differences with Simulator

In general, debugging with the Simulator and debugging with a real i860 Microprocessor are identical processes. There are, however, a few differences:

1. The environmental registers of the Simulator are not available when debugging with a real i860 Microprocessor. Attempts to access these registers produce error messages.
2. The i860 Microprocessor (having only one data breakpoint register) directly supports only one data breakpoint. The Simulator provides twenty data breakpoints.
3. The **$t** command (display instruction trace) and the logging feature are not available when debugging with a real i860 Microprocessor.

3.5 Run-Time Support

The Simulator supports a set of run-time interfaces that are independent of specific hardware configurations. The interfaces consist of:

- Stack, invocation parameters, and environment.
- Calls to operating-system services.

3.5.1 Stack, Invocation Parameters, and Environment

Before the Simulator begins executing a program, a stack is allocated and the following registers are set:

r2 Points to the top of the stack.

r16 Contains the number of command-line parameters supplied when the program was started (by the **:r**, **:s**, **:S**, **:i** or **:I** command). Corresponds to the C parameter **argc**.

r17 Points to an array of pointers to the command-line arguments. Corresponds to the C parameter **argv**.

r18 Points to an area containing the environment in which the Simulator was invoked. Corresponds to the C parameter **envp**.

3.5.2 Operating-System Services

Programs being debugged may contain calls to operating-system services just as if they were executing under direct control of an operating system. Calling conventions are defined in the "Programming Model" chapter of the *i860 Microprocessor Programmer's Reference Manual.*

Chapter 4
Example Debugging Session

This chapter is an example of a debugging session. It demonstrates some of the most frequently used features of the Simulator and explains how to create a program for debugging.

Example 4-1 shows the source code of the program to be debugged. Assume that this program is in a file called **example.s**. The program accepts two integer numbers and divides them, giving the quotient and remainder.

The program must first be assembled and linked. This is done by:

mas860 example.s

ld860 example.o libc.a -o example -e ep

The **-e** option sets **ep** as the entry point to the program. The file **libc.a** is the C interface library.

```
                                        // SIGNED INTEGER DIVIDE .data
one_plus_eps:: .double 1.000000005
BNpBC::        .long          0x80000000
               .long          0x43300000
double_2::     .double 2.0
               .text
                                        // r16 - numerator (dividend);
                                        // r17 - denominator (divisor)
                                        // r16 <- quotient;
                                        // f2..f11 temporaries.
idiv::                                  // Procedure to be tested
                                        // Convert Denominator and Numerator
fld.d          BNpBC,          f6       // Sum of biases
xorh           0x8000, r17,    r18      // Add BC to denominator
ixfr           r18,            f4       // Construct low half
fmov.ss f7,    f5                       // Set exponent in high half
xorh           0x8000, r16,    r16      // Add BC to numerator
fsub.dd f4,    f6,             f4       // Extract biases
ixfr           r16,            f2       // Construct low half
fmov.ss f7, f3                                 // Set exponent in high half
fsub.dd f2,    f6,             f2       // Extract biases
                                        // Do Floating-Point Divide
fld.d          double_2,       f10      // Load floating-point two
frcp.dd f4,    f6                       // First guess has 2**-8 error
fmul.dd f4,    f6,             f8       // Guess * divisor
fsub.dd f10,   f8,             f8       // 2 - guess * divisor
fmul.dd f6,    f8,             f6       // Second guess has 2**-15 error
fmul.dd f4,    f6,             f8       // Avoid using f6 as src1
fsub.dd f10,   f8,             f8       // 2 - guess * divisor
fmul.dd f6,    f8,             f6       // Third guess has 2**-29 error
fmul.dd f4,    f6,             f8       // Avoid using f6 as src1
fsub.dd f10,   f8,             f8       // 2 - guess * divisor
```

```
fmul.dd f6,    f2,           f6        // Guess * dividend
fmul.dd f8,    f6,           f8        // Result = third guess * dividend
                                       // Convert Quotient to Integer
fld.d          one_plus_eps, f10       //
fmul.dd f8,    f10,          f8        // Force quotient bigger than integer
ixfr           r17,          f10       // Get denominator to compute
                                       // remainder
ftrunc.dd      f8,           f8        // Convert to integer
bri            r1                      // Exit procedure after next
                                       // instruction
fxfr           f8,           r16       // Transfer quotient to integer
                                       // register
fp_excep::                             // User trap handler
ld.c           psr,          r31       //
xor            0x1000, r31,  r31       // Reset FT bit
st.c           r31,          psr       //
call           exit                    // Terminate simulated execution
mov            -1,           r16       // Exit code
ep::call       idiv                    // Entry point for debugging purposes
nop                                    //
xp::call       exit                    // Terminate simulated execution
nop                                    // Exit code is result of division
```

Example 4-1. Simulation Example

Following is a log of a debugging session. The **$** is the operating system's prompt; the * is the Debugger's prompt, following which the user enters a debugging request. Other lines are responses to the preceding request.

```
$ sim860 -d example
*$d                                    // Change default radix to decimal
*200>r16                               // Set dividend
*10>r17                                // Set divisor
*$r                                    // Display integer registers in decimal
r0 = 0      r1 = 0              r2 = 0          r3 = 0
r4 = 0      r5 = 0              r6 = 0          r7 = 0
r8 = 0      r9 = 0              r10 = 0         r11 = 0
r12 = 0     r13 = 0             r14 = 0         r15 = 0
r16 = 200   r17 = 10            r18 = 0         r19 = 0
r20 = 0     r21 = 0             r22 = 0         r23 = 0
r24 = 0     r25 = 0             r26 = 0         r27 = 0
r28 = 0     r29 = 0             r30 = 0         r31 = 0
fir = 0xf040013c                psr = 0x0       epsr = 0x40101       db = 0x0
fsr = 0x20 merge = 0x0
dirbase: dtb=0 rc=0 rb=0 dps=0
psr: pm=0 ps=0 sc=0
epsr: pt=1 sn=1 dcs=1
fsr: ae=0 rr=0 fte rm=0
*$x                                    // Change default radix to hexadecimal
*$r                                    // Display integer registers in hex
r0 = 0x0    r1 = 0x0            r2 = 0x0        r3 = 0x0
r4 = 0x0    r5 = 0x0            r6 = 0x0        r7 = 0x0
r8 = 0x0    r9 = 0x0            r10 = 0x0       r11 = 0x0
```

```
r12 = 0x0   r13 = 0x0          r14 = 0x0       r15 = 0x0
r16 = 0xc8  r17 = 0xa          r18 = 0x0       r19 = 0x0
r20 = 0x0   r21 = 0x0          r22 = 0x0       r23 = 0x0
r24 = 0x0   r25 = 0x0          r26 = 0x0       r27 = 0x0
r28 = 0x0   r29 = 0x0          r30 = 0x0       r31 = 0x0
fir = 0xf040013c               psr = 0x0       epsr = 0x40101       db = 0x0
fsr = 0x20 merge = 0x0
dirbase: dtb=0 rc=0 rb=0 dps=0
psr: pm=0x0 ps=0x0 sc=0x0
epsr: pt=0x1 sn=0x1 dcs=0x1
fsr: ae=0 rr=0x0 fte rm=0
*ep:b                                          // Set a breakpoint at the entry
point
*double_2:ba                                   // Set a data access breakpoint
*one*:brd                                      // Set a data read breakpoint at
                                               // one_plus_eps
*                                              // using wildcard abbreviation
*                                              // Note that Simulator supports
                                               // multiple data breakpoints
*                                              // Display all breakpoints
*$b
code breakpoints
count       breakpoint          command
0x1         ep                                 // Set a breakpoint at the entry
point
data breakpoints
count       breakpoint          type command
0x1         double_2            a     // Set a data access breakpoint
0x1         one_plus_eps        rd    // Set a data read breakpoint at
                                      // one_plus_eps
*                                     // Disassemble first several
                                      // instructions
// Must use ? because program has not been run yet*idiv,10?ia
idiv:       orh                 0,r0,r31
idiv+0x4:   fld.d               0x11a8(r31),f6
idiv+0x8:   xorh                0x8000,r17,r17
idiv+0xc:   ixfr                r17,f4
idiv+0x10:  fmov.ss             f7,f5
idiv+0x14:  xorh                0x8000,r16,r16
idiv+0x18:  fsub.dd             f4,f6,f4
idiv+0x1c:  ixfr                r16,f2
idiv+0x20:  fmov.ss             f7,f3
idiv+0x24:  fsub.dd             f2,f6,f2
idiv+0x28:  orh                 0,r0,r31
idiv+0x2c:  fld.d               0x11b0(r31),f10
idiv+0x30:  frcp.dd             f4,f6
idiv+0x34:  fmul.dd             f4,f6,f8
idiv+0x38:  fsub.dd             f10,f8,f8
idiv+0x3c:  fmul.dd             f6,f8,f6
idiv+0x40:
*:r                                   // Run the process.  Will stop at
                                      // breakpoint at ep.
<< Code breakpoint at 0xf040013c >>
```

```
ep:         call                idiv
            nop
*:s                                         // Single step
idiv:       orh                 0,r0,r31
*\                                          // Another form of single step
idiv+0x4:   fld.d               0x11a8(r31),f6
*:c                                         // Continue.  Will stop at
breakpoint
                                            // on double_2.
<< Data read breakpoint at 0xf04000dc, read from 0x11b0 >>
idiv+0x2c: fld.d                0x11b0(r31),f10
*0x11b0=p                                   // Make sure the stop address is
                                            // double_2
            double_2:
*0t12$t                                     // How did we get here?
idiv:       orh                 0,r0,r31
idiv+0x4:   fld.d               0x11a8(r31),f6
idiv+0x8:   xorh                0x8000,r17,r17
idiv+0xc:   ixfr                r17,f4
idiv+0x10:  fmov.ss             f7,f5
idiv+0x14:  xorh                0x8000,r16,r16
idiv+0x18:  fsub.dd             f4,f6,f4
idiv+0x1c:  ixfr                r16,f2
idiv+0x20:  fmov.ss             f7,f3
idiv+0x24:  fsub.dd             f2,f6,f2
idiv+0x28:  orh                 0,r0,r31
idiv+0x2c:  fld.d               0x11b0(r31),f10
*                                           // Disassemble memory image
starting
                                            // from here
*<pc,9/X^ia                                 // in both hex and instruction
format.
idiv+0x2c:  0x27ea11b0          fld.d       0x11b0(r31),f10
idiv+0x30:  0x488601a2          frcp.dd     f4,f6
idiv+0x34:  0x48c821a0          fmul.dd     f4,f6,f8
idiv+0x38:  0x490851b1          fsub.dd     f10,f8,f8
idiv+0x3c:  0x490631a0          fmul.dd     f6,f8,f6
idiv+0x40:  0x48c821a0          fmul.dd     f4,f6,f8
idiv+0x44:  0x490851b1          fsub.dd     f10,f8,f8
idiv+0x48:  0x490631a0          fmul.dd     f6,f8,f6
idiv+0x4c:  0x48c821a0          fmul.dd     f4,f6,f8
idiv+0x50:
*\                                          // Single step
idiv+0x30:  frcp.dd             f4,f6
*                                   // <newline> continues single stepping
idiv+0x34:  fmul.dd             f4,f6,f8
*                                   // <newline> continues single stepping
idiv+0x38:  fsub.dd             f10,f8,f8
*$rd                                // Display floating-point registers in
                                    // double precision
d0 = 0      d2 = 200
d4 = 10     d6 = 0.099853515625
d8 = 0      d10 = 2
```

```
d12 = 0     d14 = 0
d16 = 0     d18 = 0
d20 = 0     d22 = 0
d24 = 0     d26 = 0
d28 = 0     d30 = 0
t_ = 0
ki = 0
kr = 0
*:c                                          // Continue.  Will stop at
breakpoint
                                             // on one_plus_eps.
<< Data read breakpoint at 0xf0400110, read from 0x11a0 >>
idiv+0x60: fld.d                0x11a0(r31),f10
*one_plus_eps/F                              // Display value as double-
precision floating-point.
*:c                                          // Continue to end of program.
Ends by
                                             // calling exit.
<< Process terminated, returned value: 20 >>
*$q
Goodbye!
$
```

Appendix A
Debugger Quick Command Reference

$d	Decimal format for displays (default is hexadecimal)
$r	display integer Registers (plus control registers)
$rf	display Floating-point registers, KI,KR,T (single precision)
$rp	display Pipeline contents
$rs	display Simulator variables (**total_time, sim.te,** ...)
$rd	display floating registers, KI, KR, T (Double precision)
$b	display Breakpoints
$e	display "Externals" (i.e., program variables)
\	single step (**:s** is equivalent)
:r	Run (initiate) simulation, or Restart if already running
:c	Continue simulation after breakpoint
{*addr*}**:b** [*cmd*]	Breakpoint for instruction fetch [do *cmd* upon break]
{*addr*}**:ba** [*cmd*]	Breakpoint for data Access [do *cmd* upon break]
:d	Delete all breakpoints
{*addr*}**:d**	Delete breakpoint at *addr*
$q	Quit (exit) simulator
$t	display Trace of last 16 instructions executed
$x	heXadecimal format for displays
$<< [*file*]	Read requests from *file*
$> [*file*]	Divert output of simulator to *file*
2>r2	Put value 2 into integer register **r2**
0x9a>r31	Put hexadecimal value 9a into integer register **r31**
1.989e+3>f1	Put real 1,989 into single-precision register **f1**
1234.56>d2	Put real 1234.56 into double-precision register **f3:f2**
<r1>r2	Copy value of register **r1** into r**2**
<total_time=D	Display value of simulator variable **total_time**
1>sim.ce	Enable caches
1>sim.te	Enable clock-count (timer)
0>sim.me	Disable memory limit checks
{*addr*}**>t_start**	Enable **range_time** count whenever PC > *addr*
{*addr*}**>t_end**	Disable **range_time** count whenever PC > *addr*

[*addr*]/X	Display memory at *addr* as a four-byte hexadecimal integer
[*addr*]/f	Display memory at *addr* as floating point
[*addr*],10/X	Starting at *addr*, display 10 four-byte hexadecimal integers
[*addr*],10/i	Starting at *addr*, disassemble 10 instructions
<pc,10/ia	Starting at current program location, disassemble 10 instructions
[*label*]=X	Display value of *label* as four-byte hexadecimal
{*addr*}/W {*val*}	Write *val* to four bytes of memory at *addr*
1000$s	Set limit to 1000 bytes for backwards search for symbolic addresses

Notes:

1. Square brackets [ ] indicate an optional parameter.
2. Curly brackets { } indicate a required parameter.
3. **Do not** type the [ ] or { } shown in the above commands.
4. Instruction breakpoints occur **before** the execution of the instruction.
5. The value of **total_time** is inflated when many breakpoints are enabled or when single-stepping.
6. Use **:C**, not **:r**, to continue after breakpoints. **:r** restarts everything.