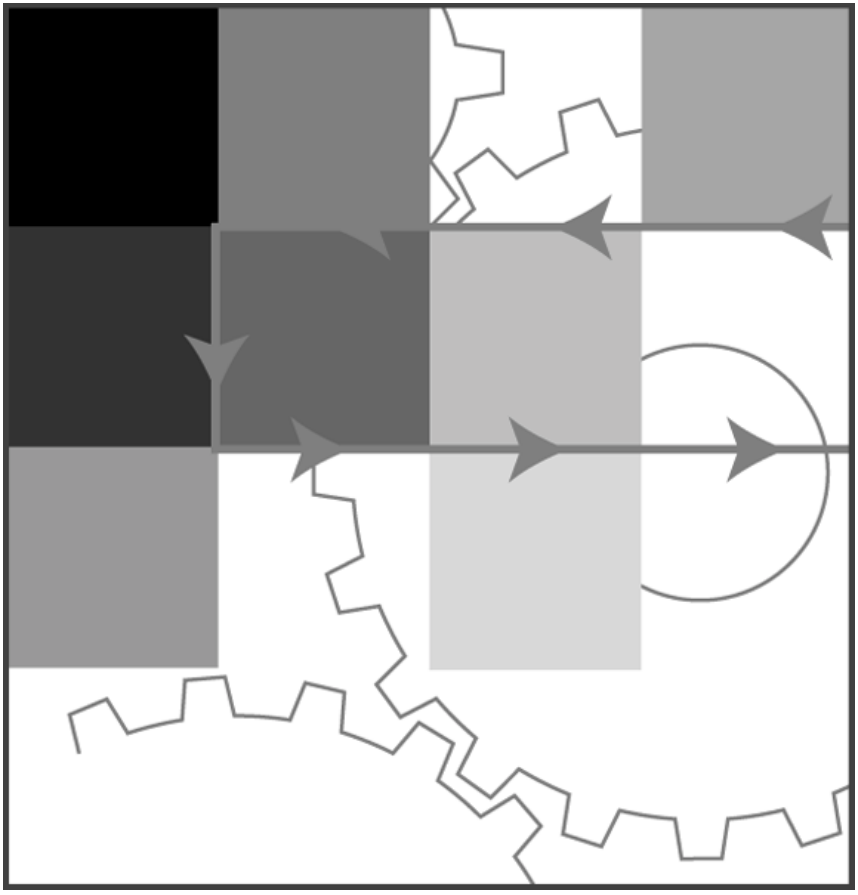
redhat.

GNUPro® Toolkit User's Guide for Fujitsu™ FR-V Architecture



How to Contact Red Hat

Red Hat Corporate Headquarters

1801 Varsity Drive
Raleigh, NC 27606 USA
Telephone (toll free): +1 888 REDHAT 1
Telephone (main line): +1 919 754 3700
Telephone (FAX line): +1 919 754 3701
Website: `http://www.redhat.com/`

Contents

Introduction

GNUPro® Toolkit from Red Hat® is a complete development system for the Fujitsu® FR-V architecture. For installation and the most current release notes, find the `README` at the top level directory of your files. For what's new with this release, see "What's New for Fujitsu FR-V Architecture" on page 4.

Tools for this architecture have support for the operating systems in Table 1.

Table 1: Supported host operating systems

Operating systems	*Central processing unit (CPU)*
Microsoft Windows 2000/XP	x86
Red Hat RHEL3, AS2.1	x86
Sun Solaris® 2.6, 2.7, 2.8, 2.9	SPARC®

Customers using older versions of RedHat Linux RHL 7.2, 7.3, 8.0 and 9 are supported to the extent that they can use the toolchain on their operating systems. RHEL3 is, however, the recommened OS for rebuilding the toolchain.

This documentation describes the features of GNUPro Toolkit specific to FR-V architecture, including information on the compiler, interactive debugger, binary utilities, libraries, and other tools. This documentation provides an introduction to the features of the tools, as well as a tutorial and reference for the FR-V architecture; see `http://www.redhat.com/docs/manuals/gnupro/` for more documentation.

There is support for the tools in Table 2; these cross-development tools have names

that reflect the target processor and the object file format that is output by the tools (ELF). This makes it possible to install more than one set of tools in the same binary directory, including both native and cross-development tools. A tool's complete tool name is a three-part hyphenated string, with the first part indicating the processor family and the mode of operation (`frv`), the second part indicating the object file format output by the tool (`elf`), and the third part indicating the generic tool name (`gcc`).

Table 2: GNUPro supported tools

Tool description	***Tool name***
GNU assembler	`frv-elf-as`

GNU binary utilities	`frv-elf-ar`
	`frv-elf-nm`
	`frv-elf-objcopy`
	`frv-elf-objdump`
	`frv-elf-ranlib`
	`frv-elf-readelf`
	`frv-elf-size`
	`frv-elf-strings`
	`frv-elf-strip`

GNU compiler collection	`frv-elf-gcc`
GNU debugger	`frv-elf-gdb`
GNU linker	`frv-elf-ld`
GNU simulator	`frv-elf-run`

IMPORTANT! Binaries for the Windows hosted toolchain use an `.exe` suffix; however, the `.exe` suffix does not need to be specified when running the executable.

For the tools to function properly on your hardware, you must have the following environment variables set.

- For the Microsoft Windows operating system, use the following examples as input for setting environment variables for the tools.

  Replace*`installdir`* with your installation directory; *`yymmdd`* indicates the name for your release (the processor name and a date, such as `frv-031205`).

  Replace `H-`*`host`* with `H-i686-pc-cygwin` as a triplet name.

  ```
  SET PROOT=C:\installdir\frv-yymmdd
  SET PATH=%PROOT%\H-host\BIN;%PATH%
  SET INFOPATH=%PROOT%\info
  REM Set TMPDIR to point to a ramdisk if you have one
  SET TMPDIR=%PROOT%
  ```

- For the Sun Solaris and Red Hat Linux operating systems, use the following examples as input for setting environment variables for the tools.

  Replace *`installdir`* with your installation directory; *`yymmdd`* indicates the name for your release (the processor name and a date, such as `frv-031205`).

Replace `H-host` (where *host* signifies the toolchain's triplet name) with `H-i686-pc-linux-gnulibc2.2` for Red Hat Linux 7.*x*, RHEL2.1 and `H-sparc-sun-solaris2.6` for Sun Solaris 2.6 host systems.

- For Bourne-compatible shells (`/bin/sh`, bash, or Korn shell), use the following example's input:

```
PROOT=installdir/frv-yymmdd
PATH=$PROOT/H-host/bin:$PATH
INFOPATH=$PROOT/info
export PATH SID_EXEC_PREFIX INFOPATH
```

- For C shells, use the following example's input:

```
set PROOT=installdir/frv-yymmdd
set path=($PROOT/H-host/bin $path)
setenv INFOPATH $PROOT/info
```

Case sensitivity for Windows is dependent on system configuration. By default, file names under Windows are not case sensitive. File names are case sensitive under UNIX. File names are case sensitive when passed to the GNU C compiler (GCC), regardless of the operating system. The following strings are case sensitive:

- command line options
- assembler labels
- linker script commands
- section names
- file names within makefiles

The following strings are not case sensitive:

- debugger commands
- assembler instructions and register names

This documentation uses some general conventions (see Table 3):

Table 3: Documentation conventions

Documentation usage	***Significance***
Bold Font	Represents menus, window names, and tool buttons.
Bold Italic Font	Denotes book titles, both hardcopy and electronic.
`Plain Typewriter Font`	Denotes code fragments, command lines, file contents, and command names; also indicates directory, file, and project names where they appear in text.
`Italic Typewriter Font`	Represents a variable to substitute.
`Bold Typewriter Font`	Indicates command lines, options, and text output generated by the program.

What's New for Fujitsu FR-V Architecture

GNUPro Toolkit has the following improvements for the FR-V architecture.

- For working with the compiler tools:
  - Added support for new FR405 and FR450 builtin functions.
  - Added options to select FR405, FR450 and FR550 code generation (`-mcpu=fr405, -mcpu=fr450` and `-mcpu=fr550` respectively).
  - Added support for scheduling and packing FR450 and FR550 code.
- For working with the debugger:
  - Implemented debugging for remote targets with RedBoot for FR451 board.
- For working with the simulator:
  - Supporting the new instructions
  - Implemented new cache size defaults for the FR450 and FR550 architectures
  - Changed the FR400 cache size defaults to match the FR405.
  - Implemented new machine models for the FR450 and FR550 architectures.
    - Provided resource constraints
    - Implemented packing restrictions
    - Implemented memory map functionality
    - Implemented exception model functionality
    - Implemented profiling model functionality (cycle counting)
- For the ABI and other general enhancements:
  - Improved ABI conformance
  - Improved error checking for builtin-in media functions

1

Tutorial

The following documentation provides tutorials for using the tools.

To get other more general information not specific to the Fujitsu architectures, see `http://www.redhat.com/docs/manuals/gnupro/` for more GNUPro Toolkit documentation.

To rebuild the tools with Microsoft Windows NT systems, see “Rebuild GNUPro for Cygwin/ Windows NT/2000/XP Systems” on page 24, and to get more information on Cygwin, see `http://www.redhat.com/docs/manuals/gnupro/`.

Create Source Code

Using a text editor, create the sample source code in Example 1; save it as `hello.c`. Use this program to verify correct installation.

Example 1: `hello.c` sample source code

```
#include <stdio.h>

int a, c;

void foo(int b)
{
  c = a + b;
  printf("%d + %d = %d\n", a, b, c);
}

int main()
{
  int b;

  a = 3;
  b = 4;
  printf("Hello, world!\n");
  foo(b);
  return 0;
}
```

Compile from Source Code

Using a bash shell, compile the example code to run on the simulator.

On Windows, type:

```
frv-elf-gcc -g hello.c -o hello.exe
```

On Linux and Solaris, type:

```
frv-elf-gcc -g hello.c -o hello.x
```

The `-g` option generates debugging information and the `-o` option specifies the name of the executable to be produced. Other useful options include `-O` for standard optimization, and `-O2` for extensive optimization. When no optimization option is specified, GCC will not optimize. See “GNU CC Command Options” in *Using GCC* in ***GNUPro Compiler Tools*** for a complete list of available options.

See “Compiler Features” on page 28 and “Assembler Features” on page 45 for special functionality when developing with the Fujitsu FR-V target.

Run the Executable on the Simulator

Using a bash shell, run your executable on the stand-alone simulator.

On Windows, type:

```
frv-elf-run hello.exe
```

On Linux and Solaris, type:

```
frv-elf-run hello.x
```

The program returns:

```
Hello world!
3 + 4 = 7
```

The simulator executes the program and returns when the program exits. See “Simulator Features” on page 55 for special functionality for working with the simulator.

Run the Debugger through an Executable

GDB can be used to debug executables using the GNUPro simulator; for information on using RedBoot when debugging a target, see “Get RedBoot for Debugging” on page 8, “RedBoot Features” on page 54, and see also `http://sources.redhat.com/redboot/`

Using a bash shell, from the `~/bin` directory, start GDB; on Windows, type:

```
frv-elf-gdb hello.exe
```

Using a bash shell, from the `~/bin` directory, start GDB; on Linux and Solaris, type:

```
frv-elf-gdb hello
```

IMPORTANT! The frv-elf-gdb command invokes the command line version of GDB. A graphical interface to GDB, called Insight, is also provided. It is invoked via the frv-elf-insight command. For more information on the debugger’s graphical user interface, see “Debug with Insight” on page 13.

After the initial copyright and configuration information, GDB returns its own prompt, **`(gdb)`**.

For details on the debugging process, see “Debug with the Simulator” on page 10 and begin at Step 3.

To stop debugging with the command line approach, type `quit` at the **`(gdb)`** prompt.

Get RedBoot for Debugging

To use RedBoot for targets, set up the the Fujitsu FR-V board as described in VDK Setting Guide available from Fujitsu.

Refer to the appropriate section for your board:

- Section 1. MB93401A CPU board and Main board
- Section 2. MB93403 CPU board and Main board
- Section 3. MB93403 CPU board, Main board and MB93493 Digital AV board
- Section 4. MB93555 CPU board and Main board
- Section 5. MB93555 CPU board, Main board and MB93493 Digital AV board
- Section 6. MB93405 CPU board (Stand alone mode)
- Section 7. MB93405 CPU board and MB93493 Digital AV board (Stand alone mode)
- Section 8. MB93405 CPU board and Main board
- Section 9. MB93405 CPU board, Main board and MB93493 Digital AV board

See `http://sources.redhat.com/redboot/` for downloading RedBoot; see Example 2 for a sample bash shell session of downloading the RedBoot image into flash for the FR-V target board (with the `frv.ROM` image).

Note: the following are provided as examples only, the actual sessions will vary a little for each supported hardware platform.

Example 2: Download the `frv.ROM` image for RedBoot for the FR-V target

```
************************************************************
** VDK LOADER for FR400 (BOOT ROM:IC8)                    **
**                                                        **
**                                            Version 1.02  **
** ALL RIGHTS RESERVED, COPYRIGHT(C) FUJITSU LIMITED 2000  **
************************************************************

Would you like to check SDRAM and SRAM ? (Y/N) : N

>r 3e00000
Flash ROM : IC7. OK ? (Y/N) Y
Blank check
Blank error !! Erase ? (Y/N) Y
Erase...
Work memory clear
Hex Data Offset Address=0x03E00000
Recieve....
```

1. At this point, send the `frv.ROM` file using ASCII protocol. Using the `dl_slow` script, set up Minicom like Example 3 shows.

   Example 3: Minicom file protocol

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
x  Name  Program                          Name U/D Full IO  Multi x
x                                                 Scr  -Red.     x
x A zmodem  /usr/bin/sz -vv -b              Y   U   N    Y   Y    x
x B ymodem  /usr/bin/sb -vv                 Y   U   N    Y   Y    x
x C xmodem  /usr/bin/sx -vv -k              Y   U   N    Y   N    x
x D zmodem  /usr/bin/rz -vv -b -E           N   D   N    Y   Y    x
x E ymodem   /usr/bin/rb -vv                N   D   N    Y   Y    x
x F xmodem  /usr/bin/rx -vv                 Y   D   N    Y   N    x
x G kermit   /usr/bin/kermit -i -l %l -s    Y   U   Y    N   N    x
x H kermit  /usr/bin/kermit -i -l %l -r     N   D   Y    N   N    x
x I ascii  /usr/bin/ascii-xfr -dsv          Y   U   N    Y   N    x
x J slow  /home/yourdir/bin/dl_slow         Y   U   N    Y   N    x
x K -                                                             x
x L -                                                             x
x M Zmodem download string activates... D                         x
x N Use filename selection window...... No                        x
x O Prompt for download directory...... No                        x
x                                                                 x
x Change which setting? (SPACE to delete)                         x
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

2. You will know if the file is transferring correctly by the LED indicator lights blinking, showing the IP address being loaded. When the download completes (using the **Enter** key to download), you will see output like Example 4.

   Example 4: Output after downloading the `frv.ROM` image

```
Write Start...
Write OK!
Verify Start...
Verify OK!
Complete !!

>
```

3. Change SW1-1 on the motherboard to be down (x). Press reset (the bottom blue button on the board). See Example 5 for the output you will see when RedBoot is active.

   Example 5: Output after set up of the `frv.ROM` image with RedBoot active

   ```
   RedBoot(tm) bootstrap and debug environment [ROMRAM]
   Version - 2001
   Platform: MB93091-CB10 evaluation board (Fujitsu FR400)
   Copyright (C) 2000, 2001, Red Hat, Inc.

   RAM: 0x00000000-0x04000000, 0x00007000-0x03fed000 available
   FLASH: 0xff000000 - 0xff200000, 32 blocks of 0x00010000 bytes.
   RedBoot>
   ```

4. Initialize the flash.

   ```
   fis init
   ```

You are ready to begin debugging. See “Debug with the Simulator” on page 10 for the tutorial for debugging; begin at Step 3.

Debug with the Simulator

The following is a sample debugging session with GDB, using the simulator. See “Simulator Features” on page 55 for special functionality for working with the simulator.

1. Using a bash shell, from where you located the binary, start the debugger.
   - On Windows, type:

     ```
     frv-elf-gdb hello.exe
     ```

   - On Linux and Solaris, type:

     ```
     frv-elf-gdb hello.x
     ```

2. To specify the target on which to debug (in this tutorial’s case, the simulator), type:

   ```
   target sim
   ```

   The program returns:

   ```
   Connected to the simulator.
   ```

3. Then, to begin debugging your program (in this tutorial’s case, either `hello.exe` for Windows or `hello.x` for Linux and Solaris operating systems), type:

   ```
   load
   ```

   The debugger returns:

```
Loading section .text, size 0x8cc0 vma 0xc8000000
Loading section .rodata, size 0x218 vma 0xc8008cc0
Loading section .data, size 0x768 vma 0xc8008ed8
Start address 0xc8000000
Transfer rate: 307712 bits in <1 sec.
```

4. To set a breakpoint, type:

   ```
   break main
   ```

   The program returns:

   ```
   Breakpoint 1 at 0xc8000168: file hello.c, line 15.
   ```

5. To run the program, type:

   ```
   run
   ```

   - On Windows, the program returns:

     ```
     Starting program: C:\hello.exe
     Breakpoint 1, main () at hello.c:15
     15        a = 3;
     ```

   - On Linux and Solaris, the program returns:

     ```
     Starting program: hello.x
     Breakpoint 1, main () at hello.c:15
     15        a = 3;
     ```

6. To print the value of variable, a, type:

   ```
   print a
   ```

   The program returns:

   ```
   $1 = 0
   ```

7. To execute the next command, type:

   ```
   step
   ```

   The program returns:

   ```
   16        b = 4;
   ```

8. To display the value of a again, type:

   ```
   print a
   ```

   The program returns:

   ```
   $2 = 3
   ```

9. To display the program being debugged, type:

   ```
   list
   ```

   The program returns:

```
11      int main()
12      {
13        int b;
14
15        a = 3;
16        b = 4;
17        printf("Hello, world!\n");
18        foo(b);
19        return 0;
20      }
```

10. To list a specific function code, use the `list` command with the name of the function to be displayed. For example, type:

```
list foo
```

The program returns:

```
1       #include <stdio.h>
2
3       int a, c;
4
5       void foo(int b)
6       {
7         c = a + b;
8         printf("%d + %d = %d\n", a, b, c);
9       }
10
```

11. To set a breakpoint at line seven, type:

```
break 7
```

You can set a breakpoint at any line by typing `break` *`linenumber`*, where *`linenumber`* is the specific linenumber to break. The program returns:

```
Breakpoint 2 at 0xc6: file hello.c, line 7.
```

12. To resume normal execution of the program until the next breakpoint, type:

```
continue
```

The program returns:

```
Continuing.
Hello, world!
Breakpoint 2, foo (b=4) at hello.c:7
7         c = a + b;
```

13. To step to the next instruction and execute it, type:

```
step
```

The program returns:

```
8         printf("%d + %d = %d\n", a, b, c);
```

14. To print the value of variable, `c`, type:

```
print c
```

The program returns:

```
$3 = 7
```

15. To see how you got to where you are, type:

```
backtrace
```

The program returns:

```
#0  foo (b=4) at hello.c:9
#1  0xc800018c in main () at hello.c:18
```

16. To exit the program and quit the debugger, type:

```
quit
```

Debug with Insight

The following documentation serves as a general reference for debugging with GNUPro Toolkit's graphical user interface, Insight; for more information, see Insight's **Help** menu for discussion of general functionality and use of menus, buttons or other features; see also "Insight, GDB's Alternative Interface" and the "Examples of Debugging with Insight" documentation in ***GNUPro Debugging Tools*** (see `http://www.redhat.com/docs/manuals/gnupro/`).

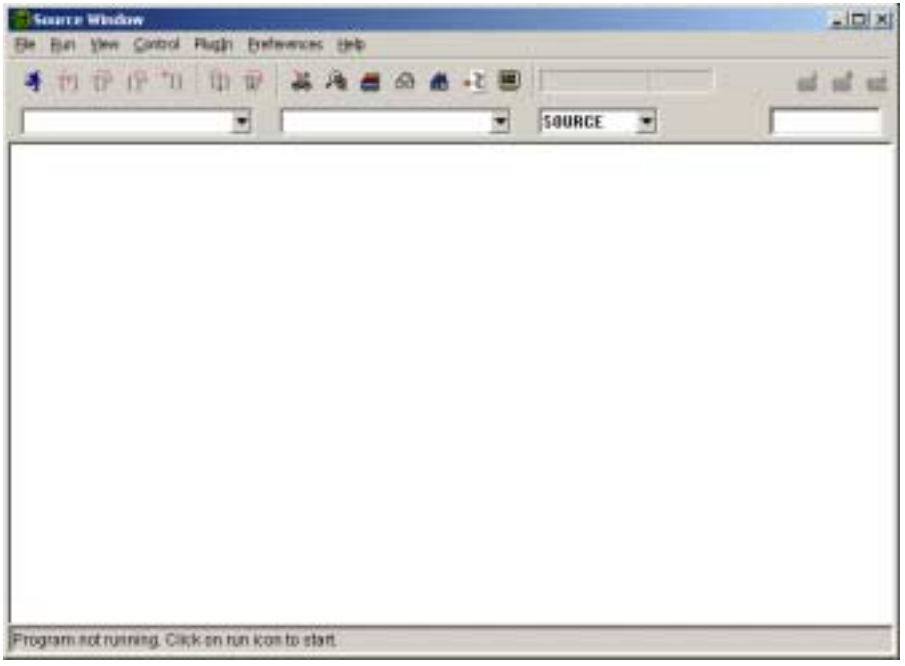
IMPORTANT! Insight is invoked via the frv-elf-insight command.

1. From a shell window, enter the following input:

```
frv-elf-insight
```

Insight launches, displaying the **Source Window** (Figure 1).

Figure 1: Source Window, the main window interface for Insight



The menu selections in the **Source Window** are **File**, **Run**, **View**, **Control**, **Plugin**, **Preferences**, and **Help**. To work with the other windows for debugging purposes specific to your project, use the **View** menu or the buttons in the toolbar.

2. To open a specific file as a project for debugging, select **File** → **Open** in the **Source Window**. The file's contents will then pass to the GDB interpreter.
3. To start debugging, click the **Run** button (Figure 2) from the **Source Window**.

Figure 2: Run button

When the debugger runs, the button turns into the **Stop** button (Figure 3).

Figure 3: Stop button

The **Stop** button interrupts the debugging process for a project, provided that the underlying hardware and protocols support such interruptions. Generally, machines that are connected to boards cannot interrupt programs on those boards. In such cases, a dialog box appears as a prompt asking if you want to abandon the session and if the debugger should detach from the target.

For an embedded project, click **Run**; then click the **Continue** button (Figure 4).

Figure 4: Continue button

WARNING! When debugging a target, do not click on the **Run** button during an active debugging process, since using the **Run** button will effectively restart the session with all work unrecoverable.

For more information on Insight, see its **Help** menu. For examples of debugging session procedures for using Insight, see the following documentation (the content assumes familiarity with debugging procedures).

- “Selecting and Examining a Source File” on page 15
- “Setting Breakpoints and Viewing Local Variables” on page 18
- “Setting Breakpoints on Multiple Threads” on page 22

To specify how source code appears and to change debugging settings, from the **Preferences** menu, select **Source**.

IMPORTANT! When debugging remote targets with RedBoot, the processor name and identification codes display when connecting to the target.

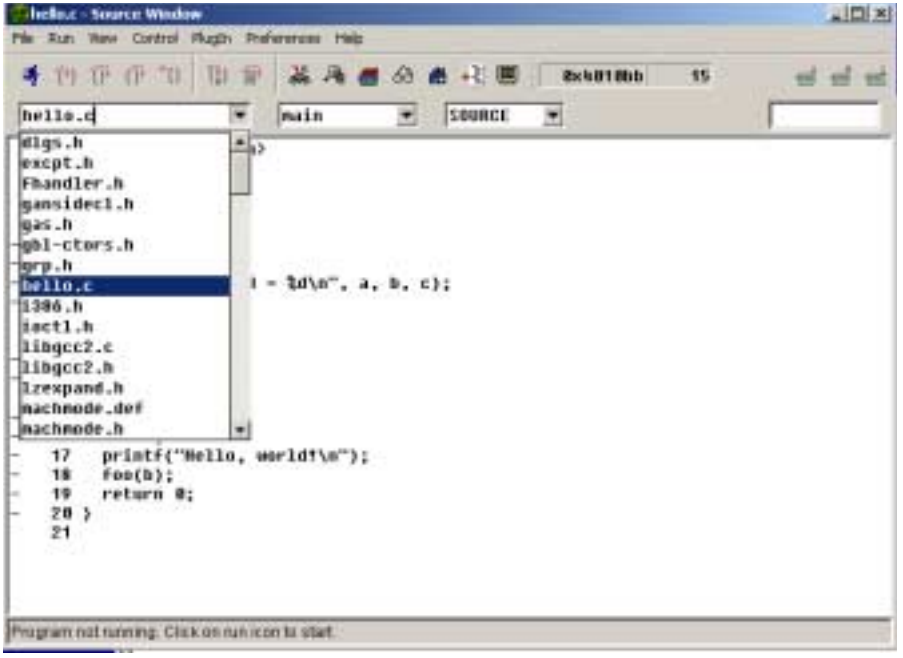
To obtain the same information, from the **Source Window**, select **Plugin** → *`target`* → **CPU Information**, information which is based on interpretation of the processor response to the CPUID instruction (*`target`* changes for every target architecture for Insight; environment variables that you set help Insight automatically to determine this functionality) . To add identification codes to the debugger’s table of Intel processors, see the ***GDB Internals*** documentation, distributed with the source code.

Selecting and Examining a Source File

To select a source file, or to specify what to display when examining a source file when debugging, use the following processes.

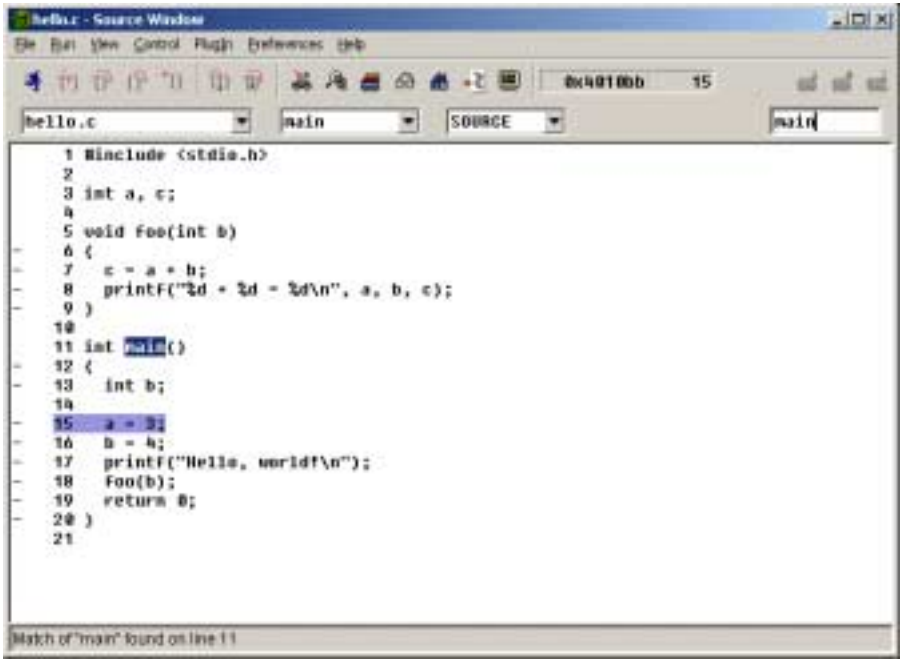
1. Select a source file from the file drop-down list with the **Source Window** (`hello.c` in Figure 5).

Figure 5: Source file selection



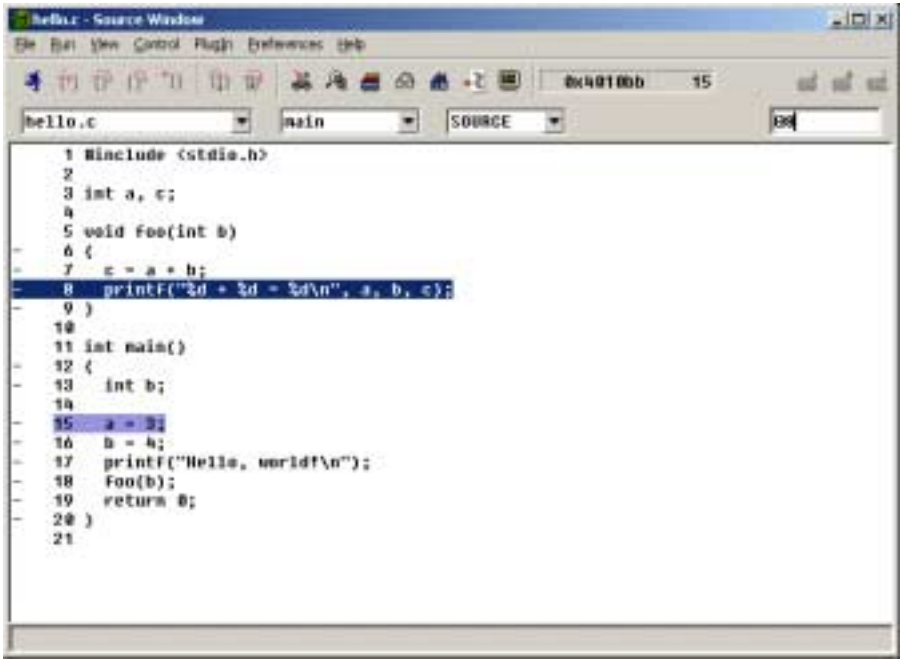
2. Select a function from the function drop-down list to the right of the file drop-down list, or type its name in the text field above the list to locate the function (in Figure 6, see the executable line 11, where the `main` function displays).

Figure 6: Search for functions



3. Use the **Enter** key to repeat a previous search. Use the **Shift** and **Enter** keys simultaneously to search backwards.
4. Type `@` with a number in the search text box in the top right of the **Source Window**. Press **Enter**. Figure 7 shows a jump to line 8 in the `hello.c` source file.

Figure 7: Searching for a specific line in source code



Setting Breakpoints and Viewing Local Variables

A breakpoint can be set at any executable line in a source file.

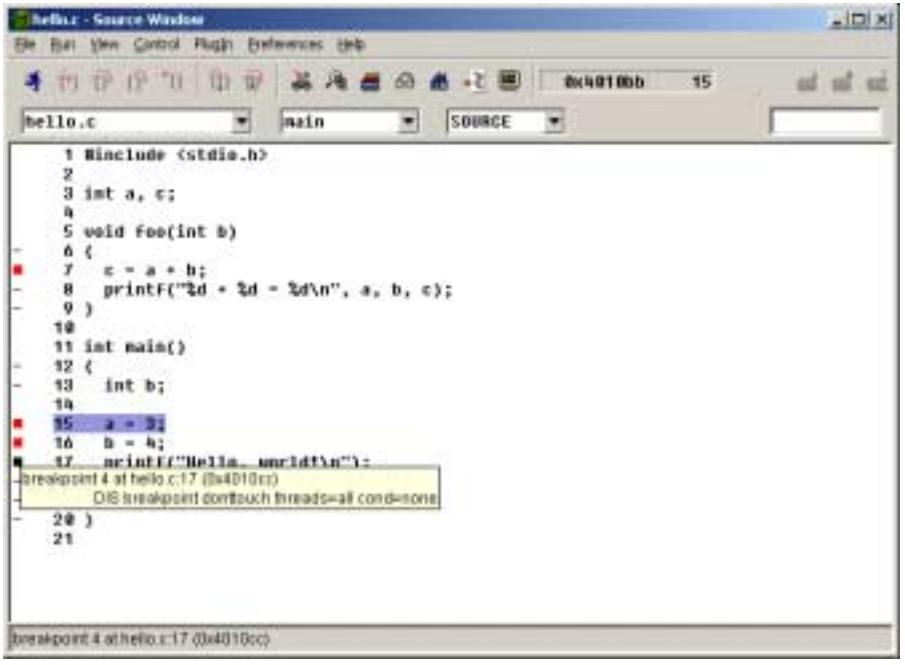
Executable lines are marked by a minus sign in the left margin of the **Source Window**. When the cursor is over a minus sign for an executable line, the cursor changes to a circle. When the cursor is in this state, a breakpoint can be set. The **Breakpoints** window is for managing the breakpoints: disabling them, enabling them, or erasing them; an enabled breakpoint is one for which the debugging session will stop, a disabled breakpoint is one which the debugging session ignores.

The following exercise steps you through setting four breakpoints in a function, as well as running the program and viewing changed values in local variables.

1. To set a breakpoint, have an active the `hello.c` source file open in the **Source Window**, and, with the cursor over a minus sign on a line, click the left mouse button. When you click on the minus sign, a red square appears for the line, signifying a set breakpoint (see the highlighted line 15 in Figure 8 for a set breakpoint).

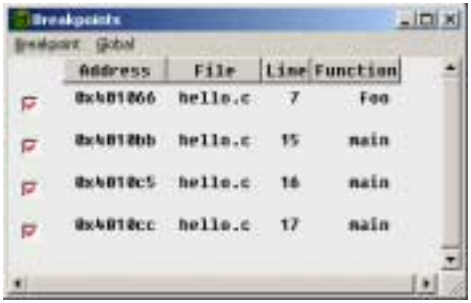
   Clicking the line again will remove the breakpoint.

Figure 8: Results of setting breakpoint for line 17



2. Open the **Breakpoints** window (Figure 9) using the **Breakpoints** button from the **Source Window**. See a line with a check box in the window appears showing that you set a breakpoint for a corresponding line in the **Source Window** frame. With the cursor over a breakpoint, a *breakpoint information balloon* displays in the **Source Window** (the information details the breakpoint, its address, its associated source file and line, its state, whether enabled, temporary, or erased, and the association to all threads for which the breakpoint will cause a stop; see also “Setting Breakpoints on Multiple Threads” on page 22 for details about threads).

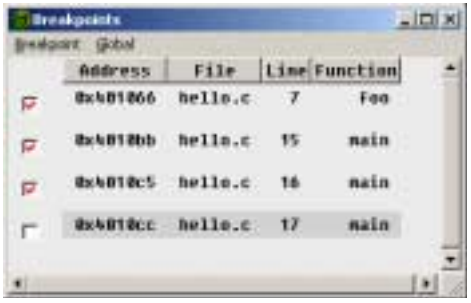
Figure 9: Breakpoints window



3. The debugger ignores disabled breakpoints, lines indicated having a black square

over them in the **Source Window** frame (see line 17 in Figure 8). Click on a breakpoint to disable the breakpoint. Figure 10 shows the results in the **Breakpoints** window of disabling a breakpoint. Re-enable a breakpoint at a line by clicking on the check box in the **Breakpoints** window. Once a breakpoint is enabled for a line, it will again have a red square in the **Source Window** frame.

Figure 10: Results of disabling a breakpoint at line 17



4. Repeat the process to set breakpoints at specific lines.
5. Click **Run** in the **Source Window** to start the executable. The debugger runs until it finds a breakpoint. When the target stops at a breakpoint, the debugger highlights a line (see highlighted line 17 in Figure 13, where the debugging stopped). For more information about breakpoints, see the standard documentation for Insight: “Insight, GDB’s Alternative Interface” and the “Examples of Debugging with Insight” documentation in ***GNUPro Debugging Tools***; see `http://www.redhat.com/docs/manuals/gnupro/`).
6. Open the **Local Variables** window by clicking its button in the tool bar for the **Source Window**; the **Local Variables** window displays the values of the variables (see Figure 11 for the `b` variable in `hello.c`).

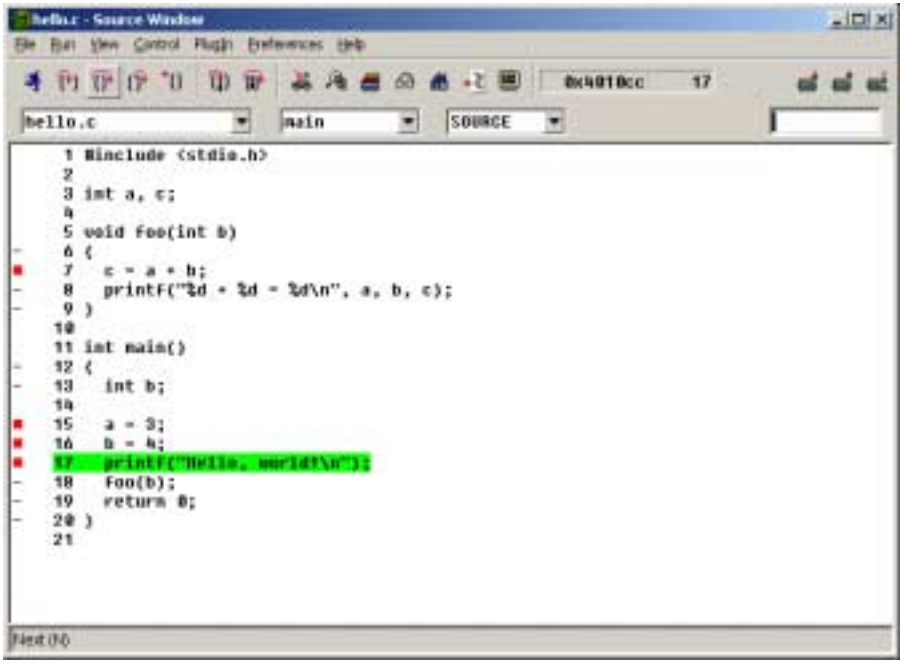
   Figure 11: Local Variables window

7. Click the **Continue** button in the **Source Window** tool bar to move to the next breakpoint. The variables that changed value turn color in the **Local Variables** window (see results in Figure 12 for the `b` variable in `hello.c`).

Figure 12: Local Variables window after setting breakpoints

8. Click the **Continue** button two more times to step through the next two breakpoints (until execution stops at line 17) and see the values of the local variables change (compare results from `hello.c` in Figure 8 and results in Figure 13).

Figure 13: Executable after changing local variable's values



Setting Breakpoints on Multiple Threads

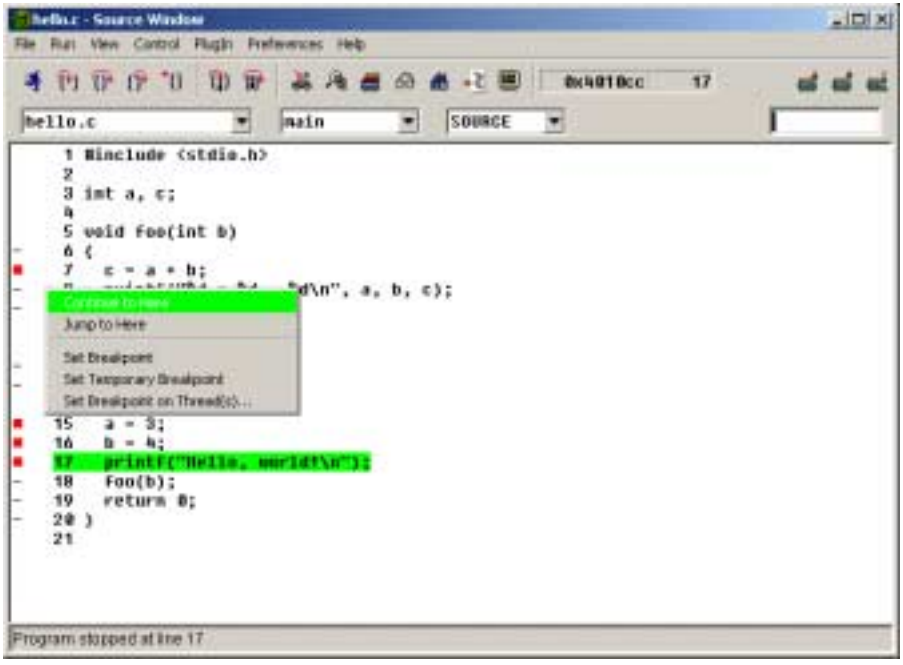
Select threads and set breakpoints on one or more threads when debugging a multi-threaded application with Insight.

WARNING! Working with multiple threads does not function similarly on all embedded targets. When debugging C++ code, for instance, breakpoints and exceptions may not work on multiple threads.

A process can have multiple threads running concurrently, each performing a different task, such as waiting for events or something time-consuming that a program does not need to complete before resuming. The thread debugging facility allows you to observe all threads while your program runs. However, whenever the debugging process is active, one thread in particular is always the focus of debugging. This thread is called the *current thread.* The precise semantics of threads and the use of threads differs depending on operating systems. In general, the threads of a single program are like multiple processes, except that they share one address space (that is, they can all examine and modify the same variables). Additionally, each thread has its own registers and execution stack and, perhaps, private memory.

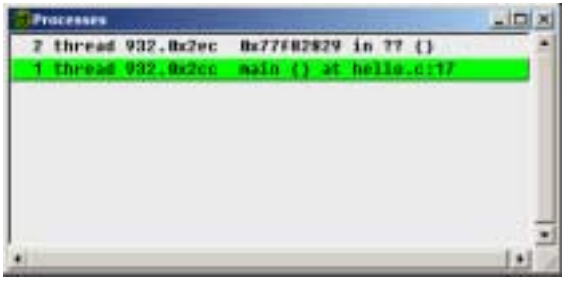
1. In the **Source Window**, right click on an executable line without a breakpoint to open the breakpoint pop-up menu (Figure 14).

   Figure 14: Breakpoint pop-up menu in the **Source Window**



2. Select **Set Breakpoint on Thread(s)** to display a window allowing you to choose the threads with which you set breakpoints. The **Processes** window (see Figure 15), available from the **Source Window**'s **View** → **Threads List** menu, displays all the available threads in the system and allows you to switch the current thread. See *Debugging with GDB* in ***GNUPro Debugging Tools*** (see `http://www.redhat.com/docs/manuals/gnupro/`) for more general information about threads.

   Figure 15: Processes window with threads



Get Assembler Listing from Source Code

To produce assembler listing information, using a bash shell, type:

```
frv-elf-gcc -g -O2 -Wa,-al -c hello.c
```

The `-g` compiler debugging option gives the assembler the necessary debugging information. The `-O2` option produces better looking code output. The `-Wa` option tells the compiler to pass the text immediately following the comma as a command line to the assembler. The `-al` assembler option requests an assembler listing. The `-c` option tells GCC to compile or assemble the source files, but not to link. Example 6 shows a partial excerpt of the output for producing an assembly listing.

Note:

> The following is provided as an example only. The actual assembler listing will be different.

Example 6: Output for assembler listing for `hello.c`

```
63                              .globl main
64                              .type   main,@function
65                      main:
66                      .LFB2:
67                      .LM11:
68                      .LBB2:
69                      .LBE2:
70 0050 82401FF0                addi sp,#-16,sp
71                      .LCFI4:
72 0054 05481000                sti.p fp, @(sp,0)
73                      .LCFI5:
74 0058 84881000                mov sp, fp
75                      .LCFI6:
76 005c 880D01C5                movsg lr, gr5
77 0060 0B482008                sti.p gr5, @(fp,8)
78                      .LCFI7:
79 0064 803C0000                call __main
80                      .LM12:
81 0068 08FC0003                setlos.p #3, gr4
82                      .LM13:
83 006c 90F80000                sethi #hi(.LC1), gr8
84                      .LM14:
85 0070 09490000                sti.p gr4, @(gr16,#gprel12(a))
86                      .LM15:
87 0074 90F40000                setlo #lo(.LC1), gr8
88 0078 803C0000                call puts
```

For more information on using the assembler tool for the Fujitsu Fujitsu FR-V targets, see “Compiler Features” on page 28 and “Assembler Features” on page 45.

Rebuild GNUPro for Cygwin/ Windows NT/2000/XP Systems

The following instructions are for rebuilding GNUPro Toolkit for Windows XP operating system in order to use the Cygwin tools, which allow you to work as if on UNIX systems. These examples show the C: drive as default for working; substitute the appropriate corresponding drive letter for the drive you use. Rebuilding requires at least 1 GB free on the drive you select.

WARNING! Do not use other Cygwin installations from another release, including any web release. Those contents may not be appropriate for configuring with your

specific release.

1. Cygwin now has a graphical installer which is used to install. Make sure there is a `Typical´ install of GNUPro 03r1 Cygwin, or, in case of a custom install, that at least the following components are installed: `Compilers´, `Utilities for rebuilding from source´ and `Contrib´
2. For the contents for rebuilding your *releasename*, see Table 4 (*releasename* signifies the name given your release, which includes the tool name, `frv`, and a release date, *yymmdd*; for example, in a previous release, you used `frv-031205`).

Table 4: Microsoft Windows rebuilding tools

File name	*Usage*
`tools-src.zip`	Compressed file of patch sources

3. Unpack your sources that you received (`tools-src.zip`) into the `C:\cygwin` directory.

   ```
   unzip tools-src.zip
   ```

   Unpacking `tools-src.zip` creates a `C:\cygwin\src` directory.
4. Make build and installation directories at the same level of the directory structure as your `C:\cygwin\src` directory.

   ```
   mkdir builddir installdir
   ```

5. Navigate to the *builddir* directory.

   ```
   cd builddir
   ```

6. Configure the tools using the following commands as input. This should be typed as a single line.

   ```
   ‘pwd‘/../src/configure --host=i686-pc-cygwin --target=target \
   --prefix=‘pwd‘/../builddir \
   --exec-prefix=‘pwd‘/../builddir/H-i686-pc-cygwin \
   >& ../configure.log
   ```

   Using the following command in your *builddir* directory, watch what a `configure.log` file produces.

   ```
   tail -f configure.log
   ```

   Start as many `bash` sessions as you require. At a minimum, you should have at least two `bash` windows open, one in which to execute the `configure`, `build`, and `install` process, and another in which to watch the progress using the `tail -f` command.
7. Make the tools with the following input’s syntax.

   ```
   make all >& ../build.log
   ```

8. Install the tools with the following input’s syntax.

   ```
   make install >& ../install.log
   ```

This step allows you to save disk space by eventually deleting your build directory without losing your logs of the build process. Do not delete your build directory until after the build process is complete and after you are confident that the tools work. At this point, you should have three log files in the `C:\cygwin` directory (`configure.log`, `build.log`, and `install.log`). You can watch the `build.log` or the `install.log` with the `tail -f` command as you did in Step 6 with `configure.log`.

Rebuilding is now complete.

If you move binaries to another machine where Cygwin is not installed, you will need to copy (using the `cp` command) the `cygwin1.dll` file from the *`installdir`* to the new directory. Ask your system administrator if you need assistance with this task.

2

Reference

The following documentation describes the Application Binary Interface (ABI) and Fujitsu FR-V architecture specific features of the GNUPro tools.

To get other more general information not specific to the Fujitsu architectures, see `http://www.redhat.com/docs/manuals/gnupro/` for more GNUPro Toolkit documentation.

Compiler Features

The following documentation describes FR-V specific features of the GNUPro Compiler Collection (GCC). For generic compiler options, see "GNU CC Command Options" in *Using GCC* in ***GNUPro Compiler Tools***.

`-mcpu=`*`CPU`*
Generates code for *`CPU`*. This option selects the hardware features normally associated with *`CPU`* such as the number of registers and the availability of floating-point and media instructions. It also controls the scheduling and packing of instructions (when enabled). `-mcpu=fr500` serves as the default if no `-mcpu=` option is given. Supported values for *`CPU`* are:

- `-mcpu=fr550`
  Compile for the FR550. This option implies `-mgpr-64`, `-mfpr-64`, `-macc-8`, `-mhard-float`, `-mmedia`, `-mdword`, `-mno-double`, and `-mno-muladd`.
- `-mcpu=fr500`
  Compile for the FR500. This option implies `-mgpr-64`, `-mfpr-64`, `-macc-8`, `-mhard-float`, `-mmedia`, `-mdword`, `-mno-double`, and `-mno-muladd`.
- `-mcpu=fr450`
  Compile for the FR450. This option implies `-mgpr-32`, `-mfpr-32`, `-macc-8`, `-msoft-float`, `-mmedia`, `-mdword`, `-mno-double`, and `-mno-muladd`.
- `-mcpu=fr405`
  Compile for the FR405. The only difference between this option and `-mcpu=fr400` is that `-mcpu=fr405` allows the use of FR405-specific built-in functions. See page 38 for more information about built-in functions.
- `-mcpu=fr400`
  Compile for the FR400. This option implies `-mgpr-32`, `-mfpr-32`, `-macc-4`, `-msoft-float`, `-mmedia`, `-mdword`, `-mno-double`, and `-mno-muladd`.

`-mno-pack`
Disable VLIW packing. This option also implies `-msoft-float` and `-mno-media`.

`-mlibrary-pic`
Enable the generation of position-independent code EABI code. See page 38 for details.

`-mfdpic`
Select the FDPIC (uClinux) ABI, that uses function descriptors to represent pointers to functions. Without any PIC/PIE-related options, it implies `-fPIE`. With `-fpic` or `-fpie`, it assumes GOT entries and small data are within a 12-bit range from the GOT base address; with `-fPIC` or `-fPIE`, GOT offsets are computed with 32 bits.

`-minline-plt`
: Enable inlining of PLT entries in function calls to functions that are not known to bind locally. It has no effect without `-mfdpic`. It's enabled by default if optimizing for speed and compiling for shared libraries (i.e., `-fPIC` or `-fpic`), or when an optimization option such as `-O3` or above is present in the command line.

`-mpgrel-ro`
: Enable the use of `GPREL` relocations in the FDPIC ABI for data that is known to be in read-only sections. It's enabled by default, except for option `-fpic` or `-fpie`, even though it may help make the global offset table smaller, it trades 1 instruction for 4. With `-fpic` or `-fpie`, it trades 3 instructions for 4, one of which may be shared by multiple symbols, and it avoids the need for a GOT entry for the referenced symbol, so it is more likely to be a win. If it is not, `-mno-gprel-ro` can be used to disable it.

`-mlinked-fp`
: Follow the EABI requirement of always creating a frame pointer whenever a stack frame is allocated. It is enabled by default, and can be disabled with `-mno-linked-fp`.

`-mlong-calls`
: Use indirect addressing to call functions outside the current compilation unit. This allows the functions to be placed anywhere within the 32-bit address space

`-fpscr`
: Enable resource-constrained software pipelining.

`-fpic`
: Compiles position independent code, using a 4096 byte global offset table.

`-fPIC`
: Compile position independent code. Unlike `-fpic`, there is no size limit for the global offset table, though it takes more instructions to refer to static and global variables.

`-fpie/-fPIE`
: Same as `-fpic/-fPIC`, respectively, but generated position independent code can be only linked into executables. On frv-elf, `-fPIE` is implied by `-mfdpic`.

`-pie`
: Produce a position independent executable on targets which support it. Code must have been compiled with the FDPIC ABI, and the executable will need a dynamic loader to run. For frv-uclinux, or when linking with frv-elf `-mfdpic`, the only differences are to force the creation of a dynamic executable, and to turn any `.rofixup` entries that would have been generated in a position dependent executable into dynamic relocations, that makes the executable relocatable by the dynamic loader, instead of by itself.

`-shared`
: Create a dynamic library. Object code must have been compiled to conform to the FDPIC ABI, and with `-fPIC` or `-fpic`.

`-static`
: Create a static executable. Dynamic libraries that might be used to satisfy link dependencies will be disregarded, and static libraries will be required instead.

`-G`*n*
: Puts statics/globals less than *n* bytes into the small data area.

The following options are only needed if you want to override the hardware features selected by `-mcpu=` compiler option.

`-mgpr-64`
`-mgpr-32`
: Select the number of general-purpose registers.

`-mfpr-64`
`-mfpr-32`
: Select the number of floating-point registers. These options only have an effect when either floating-point or media instructions are enabled.

`-macc-8`
`-macc-4`
: Select the number of accumulators and accumulator guards. These options only have an effect when media instructions are enabled.

`-mno-media`
: Disable media instructions.

`-mhard-float`
`-msoft-float`
: Specify whether the compiler should generate single-precision floating-point instructions.

`-mdouble`
`-mno-double`
: `-mdouble` enables and `-mno-double` disables double-precision floating-point instructions. These options only have an effect when single-precision instructions are enabled.

`-mdword`
`-mno-dword`
: Specify whether the target supports double-register loads and stores (`ldd`, `stdd`, `lddf`, and `stdf`).

`-mmuladd`
`-mno-muladd`
: `-mmuladd` enables and `-mno-muladd` disables the floating-point multiply-add and multiply-subtract instructions.

`-mcond-move`
`-mno-cond-move`
: `-mcond-move` enables and `-mno-cond-move` disables using conditional execution to move alternate values to a register. `-mcond-move` is on by default.

`-mscc`
`-mno-scc`
: `-mscc` enables and `-mno-scc` disables using conditional execution to set a register to 0/1 based on the results of a comparison. `-mscc` is on by default.

`-mcond-exec`
`-mno-cond-exec`
: `-mcond-exec` enables and `-mno-cond-exec` disables converting small IF-THEN and IF-THEN-ELSE statements to use conditional execution if optimizing. `-mcond-exec` is on by default.

The compiler supports the following preprocessor symbols:

`__frv__`
: Is always defined.

`__FRV_GPR__`
: Is the number of general purpose registers.

`__FRV_FPR__`
: Is the number of floating-point registers. It is 0 if both floating-point and media instructions are disabled.

`__FRV_DWORD__`
: Is defined if the target supports double-word load and store instructions.

`__FRV_ACC__`
: Is the number of media accumulators. It is 0 if media instructions are disabled.

`__FRV_HARD_FLOAT__`
: Is defined if the target supports hardware floating-point instructions.

`__FRV_VLIW__`
: Is defined if VLIW packing is enabled. When defined, it is the number of instructions packed together; 2 for `-mcpu=fr400`, `-mcpu=fr405` and `-mcpu=450`; 4 for `-mcpu=fr500`; and 8 for `-mcpu=fr550`.

`__FRV_FDPIC__`
: Is defined when the FDPIC ABI is in effect.

`__CPU_FR550__`
: Is defined by `-mcpu=fr550`.

`__CPU_FR500__`
: Is defined by `-mcpu=fr500`.

`___CPU_FR450__`
: Is defined by `-mcpu=fr450`.

`__CPU_FR405__`

Is defined by `-mcpu=fr405`.

`__CPU_FR400__`

Is defined by `-mcpu=fr400`

There are no FR-V architecture specific attributes; see “Declaring Attributes of Functions” and “Specifying Attributes of Variables” in “Extensions to the C Language Family” in *Using GNU CC* in ***GNUPro Compiler Tools*** for information.

EABI Summary of Features

The Fujitsu FR-V toolchain supports the Fujitsu FR-V EABI (*E*mbedded *A*pplication *B*inary *I*nterface), which programs use as a standard for interfacing with operating systems, including specifications such as executable format, calling conventions, chip-specific requirements, and other prerequisites.

Table 5 shows the size and alignment for all data types.

Table 5: Data type sizes and alignments

Type	***Size (bytes)***	***Alignment (bytes)***
`char`	1 byte	1 byte
`short`	2 bytes	2 bytes
`int`	4 bytes	4 bytes
`unsigned`	4 bytes	4 bytes
`long`	4 bytes	4 bytes
`long long`	8 bytes	8 bytes
`float`	4 bytes	4 bytes
`double`	8 bytes	8 bytes
`long double`	8 bytes	8 bytes
pointer	4 bytes	4 bytes

The structure/union data size is a multiple of the maximum boundary alignment size of the members. Boundary alignment for the area itself is accomplished by member maximum boundary alignment.

The individual members are subject to boundary alignment in accordance with the member type.

Table 6 shows the function calling sequence.

Table 6: Function calling sequence

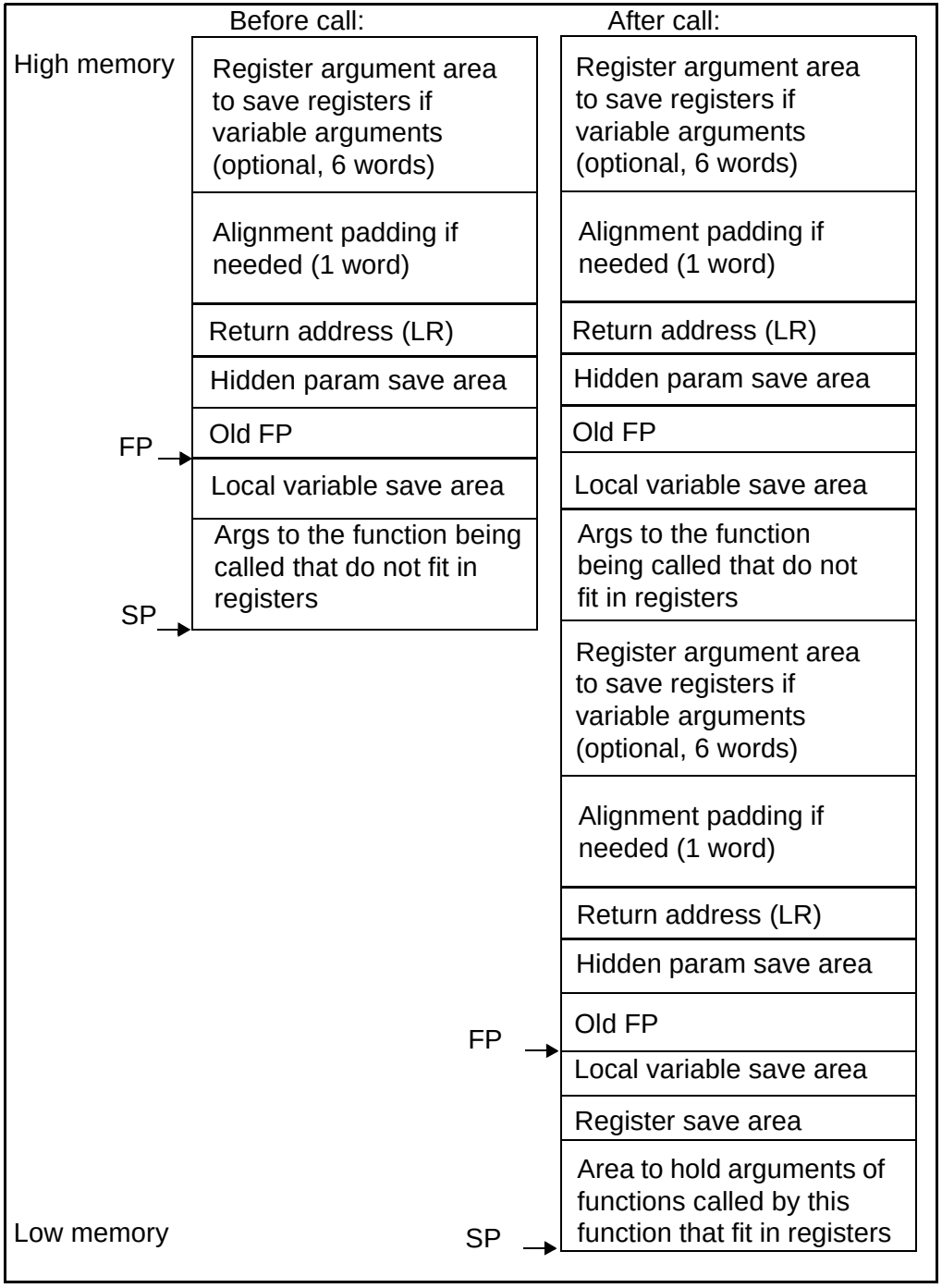
Register type	*Register name*	*Caller/Callee save*
Zero register	GR0	-
Stack pointer (SP)	GR1	-
Frame pointer (FP)	GR2	-
Hidden parameter	GR3	caller
-	GR4-GR7	caller
Argument register	GR8-GR13	caller
-	GR14-GR15	caller
-	GR16-GR31	callee
-	GR32-GR47	caller
-	GR48-GR63	callee
-	FR0-FR15	caller
-	FR16-FR31	callee
-	FR32-FR47	caller
-	FR48-FR63	callee

The following documentation describes FR-V stack frame.

- The stack grows downwards from high addresses to low addresses.
- A leaf function is not required to allocate a stack frame if one is not needed.
- The EABI requires a frame pointer always be allocated if any stack is allocated. In other words, a leaf function that uses no stack does not allocate a frame pointer, but a leaf function that uses stack or a non-leaf function requires a frame pointer.
- If `-mdword` is used then the stack will be aligned to 8-byte boundaries. If `-mno-dword` is used the stack will be aligned to 4-byte boundaries. The default is `-mdword`.

Figure 16 shows the stack frame usage for functions that take a fixed number or variable number of arguments.

Figure 16: Stack frame for functions taking a fixed or variable number of arguments



GR8 through GR13 registers carry arguments to functions, with lower-numbered registers being allocated to earlier arguments. When all six registers have been filled, any remaining arguments are placed in the stack argument area, allocating from lower to higher addresses.

To pass and return values which are not structures or unions:

- Each argument four bytes in size or smaller is allocated one complete register.
- Eight-byte arguments are allocated two consecutive complete registers. The lower-numbered register holds the most significant word, and the higher-numbered register holds the least significant word. If registers GR8 through GR12 have already been allocated, a subsequent eight-byte argument is split between registers and arguments; its most significant half is passed in GR13, and its least significant half is passed as the first word of the stack argument area.
- Values four bytes or smaller are returned in GR8. For eight-byte values, the most significant half is returned in GR8, and the least significant half is returned in GR9.

For structures and unions, the rules are as follows:

- To pass a structure or union of any size by value, the caller copies the value to a buffer in its own local variable area. The caller then passes the address of this buffer to the callee like a normal pointer argument, either in registers or on the stack. The size of the buffer must be a multiple of four bytes.
- To return a structure or union of any size by value, the caller allocates a buffer of the appropriate size in its own local variable area, and passes the address of this buffer to the callee in GR3 (the “hidden parameter”). The size of this buffer must also be a multiple of four bytes.

If the callee takes a variable number of arguments, it stores all its argument registers in an argument register save area. This area is six words long, just large enough to hold all the argument registers, and allocated just below any arguments received on the stack. Thus, once the registers have been saved, all the function’s arguments appear in a contiguous block of memory, starting with the argument register save area. To walk the argument list, the callee needs only advance a pointer from lower to higher addresses.

Table 7 shows how relocation names and numbers are used.

Table 7: Relocation names and numbers

Number	*Name*	*Usage*
0	`R_FRV_NONE`	None
1	`R_FRV_32`	32 bit relocation
2	`R_FRV_LABEL16`	Used with `bicc` instructions
3	`R_FRV_LABEL24`	Used with call instruction
4	`R_FRV_LO16`	Used with `setlo`, `setlos`
5	`R_FRV_HI16`	Used with `sethi`
6	`R_FRV_GPREL12`	Used with immediate instructions for global pointer-relative references
7	`R_FRV_GPRELU12`	for unsigned operands, used with immediate instructions for global pointer-relative references
8	`R_FRV_GPREL32`	Not used.
9	`R_FRV_GPRELHI`	Used with `sethi` for global pointer-relative access
10	`R_FRV_GPRELLO`	Used with `setlos`, `setlo` for global pointer-relative access
200	`R_FRV_GNU_VTINHERIT`	Generated for `.vtinherit` assembler directive
201	`R_FRV_GNU_VTENTRY`	Generated for `.vtentry` assembler directive

See Table 8 for the flags and values used in the `e_flags` field of the ELF header.

Table 8: Flags and values used in the `e_flags` field of the ELF header

Flag	*Value*	*Usage*
EF_FRV_GPR32	0x00000001	Compiled with -mgpr-32
EF_FRV_GPR64	0x00000002	Compiled with -mgpr-64
EF_FRV_FPR32	0x00000004	Compiled with -mfpr-32
EF_FRV_FPR64	0x00000008	Compiled with -mfpr-64
EF_FRV_FPR_NONE	0x0000000c	Compiled with -msoft-float
EF_FRV_DWORD_YES	0x00000010	Compiled with -mdword
EF_FRV_DWORD_NO	0x00000020	Compiled with -mno-dword
EF_FRV_DOUBLE	0x00000040	Compiled with -mdouble
EF_FRV_MEDIA	0x00000080	Compiled with -mmedia
EF_FRV_PIC	0x00000100	Compiled with -fpic
EF_FRV_NON_PIC_RELOCS	0x00000200	Used non-pic relocs
EF_FRV_BIGPIC	0x00000800	Compiled with -fPIC
EF_FRV_LIBPIC	0x00001000	Compiled with -mlibrary-pic
EF_FRV_G0	0x00002000	All modules compiled with -G 0
EF_FRV_NOPACK	0x00004000	Compiled with -mno-pack
EF_FRV_FDPIC	0x00008000	Compiled with -mfdpic
EF_FRV_MULADD	0x00000400	Compiled with -mmuladds
EF_FRV_CPU_GENERIC	0x00000000	Generic FRV
EF_FRV_CPU_FR500	0x01000000	Compiled for the FR500
EF_FRV_CPU_FR400	0x05000000	Compiled for the FR400
EF_FRV_CPU_FR550	0x06000000	Compiled for the FR550
EF_FRV_CPU_FR405	0x07000000	Compiled for the FR405
EF_FRV_CPU_FR450	0x08000000	Compiled for the FR450

Grouping small global data items together results in more efficient code generation for the FR-V. Consider the following code sequence.

```
sethi %hi(_smallvar), gr22
setlo %lo(_smallvar), gr22
ld @(gr22, gr0), gr23
```

Instead, the following code will generate to load a value from the small data area:

```
ldi @(gr16, _smallvar), gr23
```

To facilitate this optimization, `gr16` is reserved for use as the global data pointer (`gp`). By default, all global data items which are less than 8 bytes will be placed in two special sections named `.sbss` and `.sdata`. It is possible to address up to 4K of globals using this scheme. The `-G` compiler switch is provided to change the default size of items which are placed in these sections. The “section” attribute may also be used to control placement of globals.

The middle address of the small data area will be defined by an entry in the linker script. The `gp` register is initialized by the startup code.

The `PIC` register (`gr17`) is reserved for use with position independent code. The compiler gathers static pointers in a special section named `.rofixup`. Within this section the `.picptr` is used to mark those pointers which are considered to be valid for use with position independent code. For programs compiled with `-fpic`, code will be generated upon procedure entry to set up the `PIC` register (`gr17`). All addresses will then be loaded relative to the `PIC` register. The code which sets up the `PIC` register looks like the code in Example 7.

Example 7: Procedure prologue code which sets up the `PIC` register

```
            call .LCF0
.LCF0:
            movsg lr, gr17
            sethi %gprelhi(.LCF0), gr5
            setlo %gprello(.LCF0), gr5
           sub gr17,gr5,gr17
```

The `%gprelhi` - `%gprello` syntax triggers a `_gp` relative relocation for the `sethi` and `setlo` instructions.

To access a data item once the `PIC` register has been set up, the following code sequence is then used:

```
ldi @(gr17,_y), gr8
```

FDPIC ABI Summary

FDPIC enables the creation of executables and dynamic libraries that enables a multi-process system to share the text segments of multiple processes running the same program or using the same dynamic library, even on a machine without a memory management unit. This is accomplished by using gr15 as the PIC register, that points to a global offset table (GOT). Every address computation uses the PIC register, either by adding an offset to it when an address is part of a data segment, or by loading an address from the GOT otherwise. Function calls sequences must set gr15 to the same value it had at the function entry, and not expect it to remain unchanged after the call. Calls to functions in other translation units may go through a procedure linkage table stub. Pointers to functions do not point to the function entry point, but rather to a function descriptor, that holds not only the address of the function entry point, but also a pointer to the GOT address that must be in the PIC register in order to call the function within the context of the process. Additional details are available in a separate document, that specifies the FDPIC ABI.

Built-in Functions

GCC provides many FR-V-specific built-in functions for accessing features such as saturated arithetic, cache prefetching and media operations. The term "built-in" refers to the fact that the functions are integrated into GCC itself; there is no need to include a special header file.

Argument and return types

The arguments to built-in functions can be divided into three groups: register numbers, compile-time constants and run-time values. In order to make this classification clear at a glance, the arguments and return values are given the following pseudo types.

Table 9: Arguments and return types.

Pseudo Type	*Real C type*	*Constant*	*Description*
`uh`	`unsigned short`	No	an unsigned halfword
`uw1`	`unsigned int`	No	an unsigned word
`sw1`	`int`	No	a signed word
`uw2`	`long long`	No	an unsigned doubleword
`sw2`	`long long`	No	a signed doubleword
`const`	`int`	Yes	an integer constant
`acc`	`int`	Yes	an ACC register number
`iacc`	`int`	Yes	an IACC register number

Note that these pseudo types are not defined by GCC, they are simply a notational convenience used in this manual.

Arguments of type uh, uw1, sw1, uw2 and sw2 are evaluated at run time. They correspond to register operands in the underlying FR-V instruction.

`const` arguments represent immediate operands in the underlying FR-V instruction. They must be compile-time constants.

`acc` arguments are evaluated at compile time and specify the number of an accumulator register. For example, an `acc` argument of 2 will select the ACC2 register.

`iacc` arguments are similar to `acc` arguments but specify the number of an IACC register. See the description of the IACC functions for more details.

Directly-mapped built-in functions

Most of the built-in functions are named after the FR-V instruction which implements them. This section summarizes these functions in tabular form. Each table has three columns:

- Instruction

  The assembly-language syntax for the underlying FR-V instruction. Operands are denoted by lower case letters (`a`, `b`, etc.).

- Function Prototype

  A C-like prototype for the built-in function. See the previous section for the meaning of argument and return types.

- Operand Mapping

  An example of how the function might be used. Variables are denoted `a`, `b`, etc., and correspond to the operands of the same name in column 1.

Please see the FR-V instruction set manuals for a description of what each instruction does.

Table 10: Integer Instructions

Instruction	***Function Prototype***	***Operand Mapping***
`ADDSS a,b,c`	`sw1 __ADDSS (sw1, sw1)`	`c = __ADDSS (a, b)`
`SCAN a,b,c`	`sw1 __SCAN (sw1, sw1)`	`c = __SCAN (a, b)`
`SCUTSS a,b`	`sw1 __SCUTSS (sw1)`	`b = __SCUTSS (a)`
`SLASS a,b,c`	`sw1 __SLASS (sw1, sw1)`	`c = __SLASS (a, b)`
`SMASS a,b`	`void __SMASS (sw1, sw1)`	`__SMASS (a, b)`
`SMSSS a,b`	`void __SMSSS (sw1, sw1)`	`__SMSSS (a, b)`
`SMU a,b`	`void __SMU (sw1, sw1)`	`__SMU (a, b)`
`SMUL a,b,c`	`sw2 __SMUL (sw1, sw1)`	`c = __SMUL (a, b)`
`SUBSS a,b,c`	`sw1 __SUBSS (sw1, sw1)`	`c = __SUBSS (a, b)`
`UMUL a,b,c`	`uw2 __UMUL (uw1, uw1)`	`c = __UMUL (a, b)`

Table 11: Media Instructions

Instruction	*Function Prototype*	*Operand Mapping*
MABSHS a,b	uw1 __MABSHS (sw1)	b = __MABSHS (a)
MADDHUS a,b,c	uw1 __MADDHUS (uw1, uw1)	c = __MADDHUS (a, b)
MADDHSS a,b,c	sw1 __MADDHSS (sw1, sw1)	c = __MADDHSS (a, b)
MADDACCS a,b	void __MADDACCS (acc, acc)	__MADDACCS (b, a)
MAND a,b,c	uw1 __MAND (uw1, uw1)	c = __MAND (a, b)
MASACCS a,b	void __MASACCS (acc, acc)	__MASACCS (b, a)
MAVEH a,b,c	uw1 __MAVEH (uw1, uw1)	c = __MAVEH (a, b)
MBTOH a,b	uw2 __MBTOH (uw1)	b = __MBTOH (a)
MBTOHE a,b	void __MBTOHE (uw1 *, uw1)	__MBTOHE (&b, a)
MCLRACC a,#0	void __MCLRACC (acc)	__MCLRACC (a)
MCLRACC acc0,#1	void __MCLRACCA (void)	__MCLRACCA ()
Mcop1 a,b,c	uw1 __Mcop1 (uw1, uw1)	c = __Mcop1 (a, b)
Mcop2 a,b,c	uw1 __Mcop2 (uw1, uw1)	c = __Mcop2 (a, b)
MCPLHI a,#b,c	uw1 __MCPLHI (uw2, const)	c = __MCPLHI (a, b)
MCPLI a,#b,c	uw1 __MCPLI (uw2, const)	c = __MCPLI (a, b)
MCPXIS a,b,c	void __MCPXIS (acc, sw1, sw1)	__MCPXIS (c, a, b)
MCPXIU a,b,c	void __MCPXIU (acc, uw1, uw1)	__MCPXIU (c, a, b
MCPXRS a,b,c	void __MCPXRS (acc, sw1, sw1)	__MCPXRS (c, a, b)
MCPXRU a,b,c	void __MCPXRU (acc, uw1, uw1)	__MCPXRU (c, a, b)
MCUT a,b,c	uw1 __MCUT (acc, uw1)	c = __MCUT (a, b)
MCUTSS a,b,c	uw1 __MCUTSS (acc, sw1)	c = __MCUTSS (a, b)
MDADDACCS a,b	void __MDADDACCS (acc, acc)	__MDADDACCS (b, a)
MDASACCS a,b	void __MDASACCS (acc, acc)	__MDASACCS (b, a)
MDCUTSSI a,#b,c	uw2 __MDCUTSSI (acc, const)	c = __MDCUTSSI (a, b)
MDPACKH a,b,c	uw2 __MDPACKH (uh, uh, uh, uh)	c = __MDPACKH (a1, a2, b1, b2)
MDROTLI a,#b,c	uw2 __MDROTLI (uw2, const)	c = __MDROTLI (a, b)
MDSUBACCS a,b	void __MDSUBACCS (acc, acc)	__MDSUBACCS (b, a)
MDUNPACKH a,b	void __MDUNPACKH (uw1 *, uw2)	__MDUNPACKH (&b, a)
MEXPDHD a,#b,c	uw2 __MEXPDHD (uw1, const)	c = __MEXPDHD (a, b)
MEXPDHW a,#b,c	uw1 __MEXPDHW (uw1, const)	c = __MEXPDHW (a, b)
MHDSETH a,#b,c	uw1 __MHDSETH (uw1, const)	c = __MHDSETH (a, b)
MHDSETS #a,b	sw1 __MHDSETS (const)	b = __MHDSETS (a)
MHSETHIH #a,b	uw1 __MHSETHIH (uw1, const)	b = __MHSETHIH (b, a)
MHSETHIS #a,b	sw1 __MHSETHIS (sw1, const)	b = __MHSETHIS (b, a)
MHSETLOH #a,b	uw1 __MHSETLOH (uw1, const)	b = __MHSETLOH (b, a)
MHSETLOS #a,b	sw1 __MHSETLOS (sw1, const)	b = __MHSETLOS (b, a)

Table 11: Media Instructions (cont)

Instruction	*Function Prototype*	*Operand Mapping*
MHTOB a,b	uw1 __MHTOB (uw2)	b = __MHTOB (a)
MMACHS a,b,c	void __MMACHS (acc, sw1, sw1)	__MMACHS (c, a, b)
MMACHU a,b,c	void __MMACHU (acc, uw1, uw1)	__MMACHU (c, a, b)
MMRDHS a,b,c	void __MMRDHS (acc, sw1, sw1)	__MMRDHS (c, a, b)
MMRDHU a,b,c	void __MMRDHU (acc, uw1, uw1)	__MMRDHU (c, a, b)
MMULHS a,b,c	void __MMULHS (acc, sw1, sw1)	__MMULHS (c, a, b)
MMULHU a,b,c	void __MMULHU (acc, uw1, uw1)	__MMULHU (c, a, b)
MMULXHS a,b,c	void __MMULXHS (acc, sw1, sw1)	__MMULXHS (c, a, b)
MMULXHU a,b,c	void __MMULXHU (acc, uw1, uw1)	__MMULXHU (c, a, b)
MNOT a,b	uw1 __MNOT (uw1)	b = __MNOT (a)
MOR a,b,c	uw1 __MOR (uw1, uw1)	c = __MOR (a, b)
MPACKH a,b,c	uw1 __MPACKH (uh, uh)	c = __MPACKH (a, b)
MQADDHSS a,b,c	sw2 __MQADDHSS (sw2, sw2)	c = __MQADDHSS (a, b)
MQADDHUS a,b,c	uw2 __MQADDHUS (uw2, uw2)	c = __MQADDHUS (a, b)
MQCPXIS a,b,c	void __MQCPXIS (acc, sw2, sw2)	__MQCPXIS (c, a, b)
MQCPXIU a,b,c	void __MQCPXIU (acc, uw2, uw2)	__MQCPXIU (c, a, b)
MQCPXRS a,b,c	void __MQCPXRS (acc, sw2, sw2)	__MQCPXRS (c, a, b)
MQCPXRU a,b,c	void __MQCPXRU (acc, uw2, uw2)	__MQCPXRU (c, a, b)
MQLCLRHS a,b,c	sw2 __MQLCLRHS (sw2, sw2)	c = __MQLCLRHS (a, b)
MQLMTHS a,b,c	sw2 __MQLMTHS (sw2, sw2)	c = __MQLMTHS (a, b)
MQMACHS a,b,c	void __MQMACHS (acc, sw2, sw2)	__MQMACHS (c, a, b)
MQMACHU a,b,c	void __MQMACHU (acc, uw2, uw2)	__MQMACHU (c, a, b)
MQMACXHS a,b,c	void __MQMACXHS (acc, sw2, sw2)	__MQMACXHS (c, a, b)
MQMULHS a,b,c	void __MQMULHS (acc, sw2, sw2)	__MQMULHS (c, a, b)
MQMULHU a,b,c	void __MQMULHU (acc, uw2, uw2)	__MQMULHU (c, a, b)
MQMULXHS a,b,c	void __MQMULXHS (acc, sw2, sw2)	__MQMULXHS (c, a, b)
MQMULXHU a,b,c	void __MQMULXHU (acc, uw2, uw2)	__MQMULXHU (c, a, b)
MQSATHS a,b,c	sw2 __MQSATHS (sw2, sw2)	c = __MQSATHS (a, b)

Table 11: Media Instruction (cont)

Instruction	*Function Prototype*	*Operand Mapping*
`MQSLLHI a,#b,c`	`uw2 __MQSLLHI (uw2, const)`	`c = __MQSLLHI (a, b)`
`MQSRAHI a,#b,c`	`sw2 __MQSRAHI (sw2, const)`	`c = __MQSRAHI (a, b)`
`MQSUBHSS a,b,c`	`sw2 __MQSUBHSS (sw2, sw2)`	`c = __MQSUBHSS (a, b)`
`MQSUBHUS a,b,c`	`uw2 __MQSUBHUS (uw2, uw2)`	`c = __MQSUBHUS (a, b)`
`MQXMACHS a,b,c`	`void __MQXMACHS (acc, sw2, sw2)`	`__MQXMACHS (c, a, b)`
`MQXMACXHS a,b,c`	`void __MQXMACXHS (acc, sw2, sw2)`	`__MQXMACXHS (c, a, b)`
`MRDACC a,b`	`uw1 __MRDACC (acc)`	`b = __MRDACC (a)`
`MRDACCG a,b`	`uw1 __MRDACCG (acc)`	`b = __MRDACCG (a)`
`MROTLI a,#b,c`	`uw1 __MROTLI (uw1, const)`	`c = __MROTLI (a, b)`
`MROTRI a,#b,c`	`uw1 __MROTRI (uw1, const)`	`c = __MROTRI (a, b)`
`MSATHS a,b,c`	`sw1 __MSATHS (sw1, sw1)`	`c = __MSATHS (a, b)`
`MSATHU a,b,c`	`uw1 __MSATHU (uw1, uw1)`	`c = __MSATHU (a, b)`
`MSLLHI a,#b,c`	`uw1 __MSLLHI (uw1, const)`	`c = __MSLLHI (a, b)`
`MSRAHI a,#b,c`	`sw1 __MSRAHI (sw1, const)`	`c = __MSRAHI (a, b)`
`MSRLHI a,#b,c`	`uw1 __MSRLHI (uw1, const)`	`c = __MSRLHI (a, b)`
`MSUBACCS a,b`	`void __MSUBACCS (acc, acc)`	`__MSUBACCS (b, a)`
`MSUBHSS a,b,c`	`sw1 __MSUBHSS (sw1, sw1)`	`c = __MSUBHSS (a, b)`
`MSUBHUS a,b,c`	`uw1 __MSUBHUS (uw1, uw1)`	`c = __MSUBHUS (a, b)`
`MTRAP void`	`__MTRAP (void)`	`__MTRAP ()`
`MUNPACKH a,b`	`uw2 __MUNPACKH (uw1)`	`b = __MUNPACKH (a)`
`MWCUT a,b,c`	`uw1 __MWCUT (uw2, uw1)`	`c = __MWCUT (a, b)`
`MWTACC a,b`	`void __MWTACC (acc, uw1)`	`__MWTACC (b, a)`
`MWTACCG a,b`	`void __MWTACCG (acc, uw1)`	`__MWTACCG (b, a)`
`MXOR a,b,c`	`uw1 __MXOR (uw1, uw1)`	`c = __MXOR (a, b)`

Other built-in functions

This section describes built-in functions that are not named after a specific FR-V instruction.

`sw2 __IACCreadll (iacc)`

Returns the full 64-bit value of IACC0. The argument is reserved for future expansion and must be 0.

`sw1 __IACCreadl (iacc)`

`__IACCreadl (0)` returns the value of IACC0H.

`__IACCreadl (1)` returns the value of IACC0L

`void __IACCsetll (iacc, sw2)`

Sets the full 64-bit value of IACC0 to the second argument. The first argument is reserved for future expansion and must be 0.

`void __IACCsetl (iacc, sw1)`

`__IACCsetl (0, X)` sets IACC0H to `X`.
`__IACCsetl (1, X)` sets IACC0L to `X`.

`void __data_prefetch0 (const void *)`

`__data_prefetch0 (X)` preloads one data cache line from address `X`. It is implemented as `dcpl X, gr0, #0`.

`void __data_prefetch (const void *)`

This function is like `__data_prefetch0` but uses the `nldub` instruction. The instruction will be issued in slot I1.

Example

Save the following code as `example.c`:

```
void f (unsigned int *z, unsigned int *x, unsigned int *y)
{
__MMULHU (2, x[0], y[0]);
__MMACHU (2, x[1], y[1]);
z[0] = __MRDACC (2);
z[1] = __MRDACC (3);
}
```

and compile it with:

```
frv-elf-gcc -O2 -mcpu=fr550 -S example.c
```

The implementation of `f` in `example.s` will be something like:

```
ldf.p @(gr10,gr0), fr1
ldf @(gr9,gr0), fr3
ldfi.p @(gr10,4), fr0
ldfi @(gr9,4), fr2
mmulhu fr3, fr1, acc2
mmachu fr2, fr0, acc2
mrdacc acc2, fr1
mrdacc acc3, fr0
```

```
stf.p fr1, @(gr8,gr0)
stfi.p fr0, @(gr8,4)
ret
```

(Note that the exact output may vary between releases.)

Assembler Features

The following documentation describes FR-V specific features of the GNUPro assembler. For generic assembler options, see “Command Line Options” in *Using* `as` in ***GNUPro Auxilairy Development Tools***. For more information, see “Get Assembler Listing from Source Code” on page 23. The instruction set is defined in the Fujitsu manual, ***FRV Architecture Specification***, Volume 1.

`-mpic`
: Assembles position independent code (compiler passes `-mpic` to the assembler if passed `-fpic`).

`-mPIC`
: Assembles large position independent code (compiler passes `-mPIC` to the assembler if passed `-fPIC`).

The following options are the same as the GCC equivalents, see page 28 for details

```
-mcpu
-mno-pack
-mlibrary-pic
-mfdpic
-G
-mgpr-64
-mgpr-32
-mfpr-64
-mfpr-32
-mno-media
-mhard-float
-msoft-float
-mdouble
-mno-double
-mdword
-mno-dword
-mmuladd
-mno-muladd
```

Opcodes are not case sensitive.

The assembler uses the register names in Table 12.

Table 12: Registers and naming conventions

Register usage	*Registers*
General purpose registers	`gr0` through `gr63`
Floating point registers	`fr0` through `fr63`
Coprocessor registers	`cpr0` through `cpr63`
Condition code registers	`icc0`, `icc1`, `icc2`, `icc3`, `fcc0`, `fcc1`, `fcc2`, `fcc3`, `cc0`, `cc1`, `cc2`, `cc3`, `cc4`, `cc5`, `cc6`, and `cc7`
Special purpose registers	There are 1024 special purpose registers.

Assembler special characters and directives

The FR-V-specific special characters are:

`#`

which starts a comment that extends to the end of the line, but only if it is the first non-whitespace character on the line.

`;`

which starts a comment that extends to the end of the line. It can be used anywhere on the line, even if non-whitespace characters have preceeded it.

`!`

which seperates two instructions on the same line. In effect the ! character is treated as if it were a new-line character.

The FR-V-specific assembler directives are:

`.eflags`

which allows the user to set, and optionally clear, the flag bits which are stored in the `e_flags` field of the ELF header. The format of the directive is:

`.eflags <set_bit_mask> [, <clear_bit_mask>]`

Any bits present in `<set_bit_mask>` will be set in the `e_flags` field. If the `<clear_bit_mask>` is also specified then any bits in it will be removed from the `e_flags field`.

`.picptr`

which produces the same output as .4byte, but which is considered safe for use in

position-independent code. It also supports the syntax:

```
.picptr funcdesc(f)
```

which generates an `R_FRV_FUNCDESC` relocation against "`f`".

Linker Features

The following documentation describes FR-V specific features of the GNUPro linker.

There are no FR-V specific command line linker options. For generic linker options, see “Linker Scripts” in *Using* `ld` in ***GNUPro Developer Tools***.

The GNU linker uses a linker script to determine how to process each section in an object file, and how to lay out the executable. The linker script is a declarative program consisting of a number of directives. For instance, the `ENTRY()` directive specifies the symbol in the executable that will be the executable’s entry point.

When building executables to run under the simulator, the GNU linker uses its built in linker script, which is a generic ELF linker script. Using a bash shell and having navigated to the `~/bin` directory, display the script with the following command:

```
frv-elf-ld --verbose
```

See Example 8 for the linker script specific to the FR-V. The linker script in the example cannot be used to create FDPIC binaries. GCC implicitly passes the `-melf32frvfd` option to the linker when linking FDPIC binaries, that causes it to use a different set of linker scripts by default. .

Example 8: FR-V linker script

```
/* Default linker script, for normal executables */
OUTPUT_FORMAT("elf32-frv", "elf32-frv","elf32-frv")
OUTPUT_ARCH(frv)
ENTRY(_start)
SEARCH_DIR("/usr/local/frv-elf/lib");
/* Do we need any of these for elf?
__DYNAMIC = 0; */SECTIONS
{/* Read-only sections, merged into text segment: */
PROVIDE (__executable_start = 0x10000); . = 0x10000;
.interp : { *(.interp) }
.hash : { *(.hash) }
.dynsym : { *(.dynsym) }
.dynstr : { *(.dynstr) }
.gnu.version : { *(.gnu.version) }
.gnu.version_d : { *(.gnu.version_d) }
.gnu.version_r : { *(.gnu.version_r) }
.rel.init : { *(.rel.init) }
.rela.init : { *(.rela.init) }
.rel.text : { *(.rel.text .rel.text.* .rel.gnu.linkonce.t.*) }
.rela.text : { *(.rela.text .rela.text.* .rela.gnu.linkonce.t.*) }
.rel.fini : { *(.rel.fini) }
.rela.fini : { *(.rela.fini) }
.rel.rodata : { *(.rel.rodata .rel.rodata.* .rel.gnu.linkonce.r.*) }
.rela.rodata : { *(.rela.rodata .rela.rodata.* .rela.gnu.linkonce.r.*)
.rel.data : { *(.rel.data .rel.data.* .rel.gnu.linkonce.d.*) }
.rela.data : { *(.rela.data .rela.data.* .rela.gnu.linkonce.d.*) }
.rel.tdata : { *(.rel.tdata .rel.tdata.* .rel.gnu.linkonce.td.*) }
.rela.tdata : { *(.rela.tdata .rela.tdata.* .rela.gnu.linkonce.td.*) }
.rel.tbss : { *(.rel.tbss .rel.tbss.* .rel.gnu.linkonce.tb.*) }
.rela.tbss : { *(.rela.tbss .rela.tbss.* .rela.gnu.linkonce.tb.*) }
.rel.ctors : { *(.rel.ctors) }
.rela.ctors : { *(.rela.ctors) }
.rel.dtors : { *(.rel.dtors) }
.rela.dtors : { *(.rela.dtors) }
.rel.got : { *(.rel.got) }
.rela.got : { *(.rela.got) }
.rel.sdata : { *(.rel.sdata .rel.sdata.* .rel.gnu.linkonce.s.*) }
.rela.sdata : { *(.rela.sdata .rela.sdata.* .rela.gnu.linkonce.s.*) }
.rel.sbss : { *(.rel.sbss .rel.sbss.* .rel.gnu.linkonce.sb.*) }
.rela.sbss : { *(.rela.sbss .rela.sbss.* .rela.gnu.linkonce.sb.*) }
.rel.sdata2 : { *(.rel.sdata2 .rel.sdata2.* .rel.gnu.linkonce.s2.*) }
.rela.sdata2 : { *(.rela.sdata2 .rela.sdata2.* rela.gnu.linkonce.s2.*)
.sdata2 : { *(.sdata2 .sdata2.* .gnu.linkonce.s2.*) }
}
```

Example 8: FR-V linker script (cont'd)

```
.rel.sbss2 : { *(.rel.sbss2 .rel.sbss2.* .rel.gnu.linkonce.sb2.*) }
.rela.sbss2 : { *(.rela.sbss2 .rela.sbss2.* .rela.gnu.linkonce.sb2.*)
}
.rel.bss : { *(.rel.bss .rel.bss.* .rel.gnu.linkonce.b.*) }
.rela.bss : { *(.rela.bss .rela.bss.* .rela.gnu.linkonce.b.*) }
.rel.plt : { *(.rel.plt) }
.rela.plt : { *(.rela.plt) }
.init :
{
KEEP (*(.init))
} =0x80000000
.plt : { *(.plt) }
.text :
{
*(.text .stub .text.* .gnu.linkonce.t.*)
/* .gnu.warning sections are handled specially by elf32.em. */
*(.gnu.warning)
} =0x80000000
.fini :
{
KEEP (*(.fini))
} =0x80000000
PROVIDE (__etext = .);
PROVIDE (_etext = .);
PROVIDE (etext = .);
.rodata : { *(.rodata .rodata.* .gnu.linkonce.r.*) }
.rodata1 : { *(.rodata1) }
.sbss2 : { *(.sbss2 .sbss2.* .gnu.linkonce.sb2.*) }
.rofixup : { *(.rofixup) }
.eh_frame_hdr : { *(.eh_frame_hdr) }
/* Adjust the address for the data segment. We want to adjust up to
the same address within the page on the next page up. */
. = ALIGN(256) + (. & (256 - 1));
/* Ensure the __preinit_array_start label is properly aligned. We
could instead move the label definition inside the section, but
the linker would then create the section even if it turns out to
be empty, which isn't pretty. */
. = ALIGN(32 / 8);
PROVIDE (__preinit_array_start = .);
.preinit_array : { *(.preinit_array) }
PROVIDE (__preinit_array_end = .);
PROVIDE (__init_array_start = .);
```

Example 8: FR-V linker script (cont'd)

```
.init_array : { *(.init_array) }
PROVIDE (__init_array_end = .);
PROVIDE (__fini_array_start = .);
.fini_array : { *(.fini_array) }
PROVIDE (__fini_array_end = .);
.data :
{
__data_start = . ;
*(.data .data.* .gnu.linkonce.d.*)
SORT(CONSTRUCTORS)
}
.data1 : { *(.data1) }
.tdata : { *(.tdata .tdata.* .gnu.linkonce.td.*) }
.tbss : { *(.tbss .tbss.* .gnu.linkonce.tb.*) *(.tcommon) }
.eh_frame : { KEEP (*(.eh_frame)) }
.gcc_except_table : { *(.gcc_except_table) }
.dynamic : { *(.dynamic) }
.ctors :
{
/* gcc uses crtbegin.o to find the start of the constructors, so we
make sure it is
first. Because this is a wildcard, it doesn't matter if the user does
not actually link against crtbegin.o; the linker won't look for a file
to match a wildcard. The wildcard also means that it doesn't matter
which directory crtbegin.o is in. */
KEEP (*crtbegin*.o(.ctors))
/* We don't want to include the .ctor section from
from the crtend.o file until after the sorted ctors.
The .ctor section from the crtend file contains the
end of ctors marker and it must be last */
KEEP (*(EXCLUDE_FILE (*crtend*.o *frvend.o) .ctors))
KEEP (*(SORT(.ctors.*)))
KEEP (*(.ctors))
}
.dtors :
{
KEEP (*crtbegin*.o(.dtors))
KEEP (*(EXCLUDE_FILE (*crtend*.o *frvend.o) .dtors))
KEEP (*(SORT(.dtors.*)))
KEEP (*(.dtors))
}
.jcr : { KEEP (*(.jcr)) }
. = ALIGN(8); _gp = . + 2048;
PROVIDE (gp = _gp);
.got : { *(.got.plt) *(.got) }
```

Example 8: FR-V linker script (cont'd)

```
/* We want the small data sections together, so single-instruction
offsets can access them all, and initialized data all before
uninitialized, so
we can shorten the on-disk segment size. */
.sdata :
{
*(.sdata .sdata.* .gnu.linkonce.s.*)
}
_edata = .;
PROVIDE (edata = .);
__bss_start = .;
.sbss :
{
PROVIDE (__sbss_start = .);
PROVIDE (___sbss_start = .);
*(.dynsbss)
*(.sbss .sbss.* .gnu.linkonce.sb.*)
*(.scommon)
PROVIDE (__sbss_end = .);
PROVIDE (___sbss_end = .);
}
.bss :
{
*(.dynbss)
*(.bss .bss.* .gnu.linkonce.b.*)
*(COMMON)
/* Align here to ensure that the .bss section occupies space up to
_end. Align after .bss to ensure correct alignment even if the
.bss section disappears because there are no input sections. */
. = ALIGN(32 / 8);
}
. = ALIGN(32 / 8);
_end = .;
__end = .;
PROVIDE (end = .);
```

Example 8: FR-V linker script (cont'd)

```
/* Stabs debugging sections. */
.stab 0 : { *(.stab) }
.stabstr 0 : { *(.stabstr) }
.stab.excl 0 : { *(.stab.excl) }
.stab.exclstr 0 : { *(.stab.exclstr) }
.stab.index 0 : { *(.stab.index) }
.stab.indexstr 0 : { *(.stab.indexstr) }
.comment 0 : { *(.comment) }
/* DWARF debug sections.
Symbols in the DWARF debugging sections are relative to the beginning
of the section so we begin them at 0. */
/* DWARF 1 */
.debug 0 : { *(.debug) }
.line 0 : { *(.line) }
/* GNU DWARF 1 extensions */
.debug_srcinfo 0 : { *(.debug_srcinfo) }
.debug_sfnames 0 : { *(.debug_sfnames) }
/* DWARF 1.1 and DWARF 2 */
.debug_aranges 0 : { *(.debug_aranges) }
.debug_pubnames 0 : { *(.debug_pubnames) }
/* DWARF 2 */
.debug_info 0 : { *(.debug_info .gnu.linkonce.wi.*) }
.debug_abbrev 0 : { *(.debug_abbrev) }
.debug_line 0 : { *(.debug_line) }
.debug_frame 0 : { *(.debug_frame) }
.debug_str 0 : { *(.debug_str) }
.debug_loc 0 : { *(.debug_loc) }
.debug_macinfo 0 : { *(.debug_macinfo) }
/* SGI/MIPS DWARF 2 extensions */
.debug_weaknames 0 : { *(.debug_weaknames) }
.debug_funcnames 0 : { *(.debug_funcnames) }
.debug_typenames 0 : { *(.debug_typenames) }
.debug_varnames 0 : { *(.debug_varnames) }
.stack 0x200000 :
{
_stack = .;
*(.stack)
}
/DISCARD/ : { *(.note.GNU-stack) }
}
```

Debugger Features

The following documentation describes FR-V specific features of the GNUPro debugger. For information on debugging, see “Run the Debugger through an Executable” on page 7, “Debug with the Simulator” on page 10, and *Debugging with GDB* in ***GNUPro Debugging Tools***. Use RedBoot for remote debugging; see “RedBoot Features” on page 54 and see the ***RedBoot User’s Guide*** (`http://sources.redhat.com/redboot/`). Having connected the serial ports (host operating system and target board), use the following GDB commands from a bash shell:

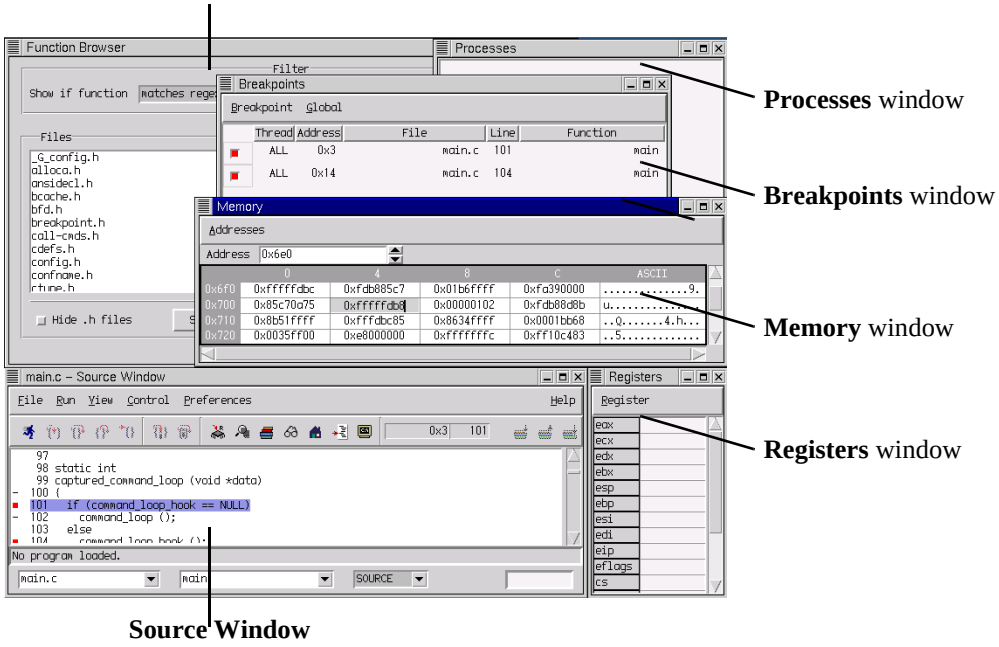
```
set remotebaud 38400
target remote com1
```

To debug, using a bash shell, use the `frv-elf-gdb` *`myapp`*`.exe` command (substituting your executable file’s name for *`myapp`*). Copyright text displays are followed by the **`(gdb)`** prompt, waiting for you to enter commands like `run` or `help`. If your program crashes and you want to determine why it crashed, type `run` and let the debugging process run. After it crashes, use the `where` command to determine where it crashed, or the `info locals` command to see the values of all the local variables. There is also the `print` command that lets you examine individual variables. If your program is doing something unexpected, use the `break` command to stop the debugging process when the process gets to a specific function or line number and use other commands to look at the state of your program at that point, to modify variables, or to step through your program’s statements one at a time.

Insight Features

For debugging, GNUPro Toolkit also includes Insight, a graphical user interface. Insight is invoked by `frv-elf-insight` command. Insight works on a range of host systems and target microprocessors, allowing development with complete access to a program’s state, for source and assembly level, with the ability to manage breakpoints, variables, registers, memory, threads, and other functionality. Providing an interface into the debugging process, Insight gives you a wide range of system information. See Figure 17 for an example of the main windows that Insight uses for analyzing and debugging programs.

Figure 17: A composite view of working with Insight



For developing with Insight, see *Insight, the GNUPro Debugger GUI Interface* in ***GNUPro Debugging Tools***.

RedBoot Features

To debug a program running the GNU debugger, GDB, you can use remote debugging by booting with RedBoot (see “Get RedBoot for Debugging” on page 8). RedBoot helps with manipulating a target system’s environment, for both product development (debug support) and for end product deployment (flash and network booting). Using serial (terminal) or Ethernet (telnet) connectivity, RedBoot has integrated GDB stubs (sub-routines) for connection to a host-based debugger (Ethernet connectivity is limited to local network) with attribute configuration (for control of aspects such as system time and date, default flash image from which to boot, a default failsafe image, static IP address, etc.). Extensible, specifically adapted to a target’s environment with network bootstrap support including setup and download, with capability of using BOOTP, DHCP and TFTP (not available for all systems or targets), RedBoot can include X/Y Modem support for image download. For more information on RedBoot, see “Run the Debugger through an Executable” on page 7 and see `http://sources.redhat.com/redboot/`

Simulator Features

The following content discusses special simulator functionality for use with the Fujitsu FR-V architecture. See also “Debug with the Simulator” on page 10. Use the `--help` option to the simulator with the following syntax.

```
frv-elf-run [options] program [program args]
```

The simulator supports general registers, `gr0` through `gr63`, the floating-point registers, `fr0` through `fr63`, the co-processor registers, `cpr0` through `cpr63`, and any special purpose registers. The simulator allocates a contiguous chunk of memory starting at address zero (`0`). The default memory size is 8 MB.

The following options are for the simulator.

`--architecture` *`machine-type`*
: Allows for specifying `simple`, `fr400`, `fr450`, `fr500`, `fr550`, `frv` for *`machine-type`*. Default is `fr500`.

`--architecture-info`
`--info-architecture`
: Lists supported architectures.

`--alignment strict|nonstrict|forced`
: Sets memory access alignment. `nonstrict` is the only accepted alignment.

`-D`
`--debug`
: Prints debugging messages.

`--debug-insn`
: Prints instruction debugging messages.

`-debug-file` *`filename`*
: Specifies the debugging output file.

`--environment user|virtual|operating`
: Sets the running environment.

`-H`
`--help`
: Displays a complete list of options recognized by the simulator.

`-c[[`*`size`*`]]`
`--scache-size [[=`*`size`*`]]`
: Specifies the size of the simulator execution cache.

`--data-cache[=`*`ways`*`[,`*`sets`*`[,`*`linesize`*`]]]`
`--insn-cache[=`*`ways`*`[,`*`sets`*`[,`*`linesize`*`]]]`
: `--data-cache` enables the data cache. `--insn-cache` enables the instruction cache. Defaults differ, depending on the setting of the `--architecture` option. These options enable simulations of the data cache and instruction cache respectively. The caches are disabled by default since the `HSR0.ICE` and `HSR0.DCE`

bits are 0 initially. The program being simulated can also enable the caches by setting these bits to 1.

ways is an integer specifying how many cache lines are associated with each SET; the default is 4 for the FR500 and FR550 architectures and 2 for the FR400 architecture (specifying 0 results in the default).

sets is an integer specifying how many sets are in the cache; the default is 64 for the FR500 architecture and 128 for the FR400, FR450 and FR550 architectures (specifying 0 results in the default).

linesize is an integer specifying the size of each cache line; the default is 64 bytes for the FR500 and FR550 architectures and 32 bytes for the FR400 and FR450 architectures (specifying 0 results in the default).

`-p`
`--profile`
: These options perform profiling. `-p` option displays information about the execution of the simulated program. In addition, for the FR-V architecture, the `-p` option, when used with the `-t` option (simulation trace) will display information about data hazards, resource hazards and instruction fetch hazards. This information is interspersed with the instruction trace and provides information on the number of cycles which the program must wait for resolution of these hazards.

`--profile-cache[=on|off]`
: Profiles caches. Displays access statistics for both caches at the end of the simulation. Also enabled by the `-p` flag.

`--profile-scache`
: Performs simulator execution cache profiling.

`--profile-core`
: Performs CORE profiling.

`--profile-file` *`filename`*
: Specifies the profile output file.

`--profile-insn`
: Performs instruction profiling.

`--profile-memory`
: Performs memory profiling.

`--profile-model`
: Performs model profiling.

`--profile-parallel[=on|off]`
: Profiles parallelism. Displays statistics on parallel execution at the end of the simulation. This option is also enabled by the `-p` flag.

`--profile-range` *`start,end`*
: Specifies the range of addresses for instruction and model profiling.

`--timer` *`cycles`*`,`*`interrupt`*
Sets the Interrupt Timer.

`--memory-alias` *`address`*`,`*`size`*`[,`*`address`*`]`
Adds a memory shadow.

`--memory-clear`
Clears all memory regions.

`--memory-delete` *`address`*`|all`
`--delete-memory` *`address`*
Deletes memory at *`address`* (or with `all`, all addresses).

`--memory-info`
`--info-memory`
Lists configurable memory regions.

`--memory-latency` *`cycles`*
This option sets the latency of memory, by setting the number of cycles required to access main memory during the simulation. The default is 24 cycles.

`--memory-region` *`address`*`,`*`size`*`[,`*`modulo`*`]`
Adds a memory region.

`--memory-size` *`size`*
Adds memory at address zero.

`--memory-latency` *`cycles`*
Sets for configuring memory latency. The default latency is assumed to be 24 cycles (read and write). Address translation is not implemented. Latency for loads and stores use the standards in Table 13.

Table 13: Configuring latency

Usage	*Cache Hit*	*Cache Miss*
GR load/store	2	`--memory-latency`
FR load/store	3	`--memory-latency + 1`
Instruction fetch	2	`--memory-latency`

`--model` *`model`*
Specifies a model to simulate. `frvbf` is the only model accepted for *`MODEL`*.

`--target` *`BFDname`*
Specifies the object-code format for the object files. `frv-unknown-elf` is the only accepted target for *`BFDname`*.

`--timer` *`cycles`*`,`*`interrupt`*
Sets the interrupt timer. The timer expires periodically after a fixed number of execution cycles. When the timer expires an external interrupt is generated. The arguments are used to configure the timer properties:

- *`cycles`* specifies the number of cycles between interrupts.
- *`interrupt`* specifies the number of the interrupt generated and must be an integer between 1 and 15.

`-t[=on|off]`
`--trace [=on|off]`
Traces useful things.

`--trace-insn [=on|off]`
Performs instruction tracing.

`--trace-extract [=on|off]`
Traces instruction extraction.

`--trace-linenum [=on|off]`
Performs line number tracing (implies `--trace-insn`).

`--trace-semantics [=on|off]`
Performs ALU, FPU, MEMORY, and BRANCH tracing.

`--trace-core [=on|off]`
Traces core operations.

`--trace-events [=on|off]`
Traces events.

`--trace-range =`*`start,end`*
Specifies range of addresses for instruction tracing.

`--trace-debug [=on|off]`
Adds information useful for debugging the simulator to the tracing output.

`--trace-file =`*`filename`*
Specifies tracing output file.

`-v`
`--verbose`
Specifies verbose output.

The following interrupts are available when using the simulator.

- `RESET`
  A software reset may be initiated by setting `RSTR.SR` or `RSTR.HR` to 1. `RSTR` is located at address `0xfeff0500`. Hardware reset is currently not supported.
- `BREAK`
  The `BREAK` instruction is supported. No other `BREAK` interrupts are supported.
- `PROGRAM`
  Program interrupts work as documented in the ***FRV Architecture***, Volume 1.
- `SOFTWARE`
  Software interrupts are supported.
- `EXTERNAL`
  External interrupts work in order to implement the timer interrupt.

Cygwin Features

The Cygwin tools that GNUPro Toolkit provides allow you to work on Windows systems as if emulating a UNIX system. For more information, see the current `http://www.redhat.com/docs/manuals/gnupro/` documentation. Cygwin, a full-featured Win32 porting layer for UNIX programs, is compatible with Win32 hosts (currently, these are Microsoft Windows NT/2000/XP systems). With Cygwin, you can make all directories have similar behavior, with all the UNIX default tools in their familiar place. Scripting languages include `bash`, `tsh`, and `tcsh`. Tools such as Perl, Tcl/Tk, `sed`, `awk`, `vim`, Emacs, xemacs, `telnetd` and `ftpd`. In order to emulate a UNIX kernel to access all processes that can run with it, use the Cygwin DLL (dynamically linked library). The Cygwin DLL will create shared memory areas so that other processes using separate instances of the DLL can access the kernel.

For more details, see `http://sources.redhat.com/cygwin/` for documentation.

Index