GNUPRO™ TOOLKIT

Emotion Engine Simulator User's Guide

October 28, 1999

CYGNUS

This documentation has been prepared by Cygnus Technical Publications; contact the Cygnus Technical Publications staff: doc@cygnus.com

Part #: 300-400-1010029

How to contact Cygnus

Use the following information for contacting Cygnus.

Cygnus Headquarters

Cygnus Solutions
1325 Chesapeake Terrace
Sunnyvale, CA 94089 USA
Telephone (toll free): +1 800 CYGNUS-1
Telephone (main line): +1 408 542 9600
FAX: +1 408 542 9699
(Faxes are answered 8 a.m.–5 p.m., Monday through Friday.)
email: `info@cygnus.com`
Website: `http://www.cygnus.com`

Cygnus Europe

Cygnus Solutions
35-36 Cambridge Place
Cambridge CB2 1NS
United Kingdom
Telephone: +44 1223 728728
FAX: +44 1223 728777
email: `euroinfo@cygnus.com`

Cygnus Canada

Cygnus Solutions
2323 Yonge Street, Suite 502
Toronto, Ontario M4P 2C9
Canada

Cygnus Japan

Nihon Cygnus Solutions
Madre Matsuda Building
4-13 Kioi-cho Chiyoda-ku
Tokyo 102-0094 Japan
Telephone: +81 3 3234 3896
FAX: +81 3 3239 3300
email: `info@cygnus.co.jp`
Website: `http://www.cygnus.co.jp/`

To resolve problems, please contact us via our website:

Website: `http://support.cygnus.com`

Contents

Introduction

Welcome to Cygnus's GNUPro Toolkit Emotion Engine Simulator User's Guide. The GNUPro Toolkit from Cygnus is a complete solution for C and C++ development for the Emotion Engine and co-processors. The tools include the compiler, simulator, interactive debugger, and utilities libraries. Debugger and linker support is included for the SKY simulator.

Tool Naming Conventions

Cross-development tools in the Cygnus GNUPro Toolkit normally have names that reflect the target processor. This allows you to install more than one set of tools in the same binary directory, including both native and cross-development tools.

The complete tool name is a three-part hyphenated string. The first part indicates the processor family (e.g. `ee` for the Emotion Engine). The second part is the generic tool name (e.g. `gcc`). For example, the GCC compiler for the Emotion Engine is:

```
ee-gcc
```

The SKY package includes the following supported tools:

Tool Description	***Tool Name***
GCC compiler	`ee-gcc`
C++ compiler	`ee-c++`
GAS assemblers	`ee-as` `ee-dvp-as`
GNU LD linker	`ee-ld`
Stand-alone simulator	`ee-run`

Binary utilities	`ee-addr2line`
	`ee-ar`
	`ee-c++filt`
	`ee-gasp`
	`ee-nm`
	`ee-objcopy`
	`ee-objdump`
	`ee-ranlib`
	`ee-readelf`
	`ee-size`
	`ee-strings`
	`ee-strip`

GDB debugger	`ee-gdb`
Linker script	`sky.ld`

Note: On Win32 systems, all tool names except `sky.ld` will have the file extension "`.exe`".

Case Sensitivity

The following strings are case-sensitive under UNIX:

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

The following strings are not case-sensitive under UNIX:

- gdb commands
- assembler instructions and register names

Environment Summary

Processor versions

Emotion Engine and custom co-processors

Targets supported

Emotion Engine - SKY simulator

Hosts supported

The following hosts are supported:

Solaris 2.5.1	(SPARC)
IRIX 6.3	(MIPS)
Red Hat Linux 5.0	(Intel x86)
Microsoft Win32 (This includes Windows NT, Windows 95, and Windows 98)	(Intel x86)

Object file format

The SKY tools support the ELF object file format. Refer to Chapter 4, ***System V Application Binary Interface*** (Prentice Hall, 1990).

1

Installation

Use the following procedures for a full installation from CD.

Throughout these procedures, screen samples are shown with a gray background. Code input is shown in plain monofont. Code output is shown in bold monofont. Code variant is shown in italic font. The UNIX command prompt is shown as **`%.`** The Windows command prompt is shown as **`C:\>`**

Throughout these examples, the variable *<yymmdd>* indicates the release date found on the CD.

To install on UNIX, see “Installing on UNIX” on page 6. To install on Win32, see “Installing on Win32” on page 8.

Installing on UNIX

1. ***Mount the CD.***

   The device used will depend on your system configuration (defaults where appropriate are used in the examples). In addition, consult your system administrator if you need assistance.

   In the following installation examples, substitute `/mnt` with the directory where the CD-ROM is mounted, or where the installation files are located.

2. ***Install the tools in a directory that has writable access permissions.***

   Make sure you can write in `/usr/cygnus` using the following input:

   ```
   % su root
   ```

   Enter the root password.

   ```
   % mkdir /usr/cygnus
   ```

   If a `File exists` error appears, ignore it.

   ```
   % chmod 777 /usr/cygnus
   % exit
   ```

3. ***Run the*** `Install` ***script to set up a target machine.***

   Using the following command:

   ```
   % /mnt/ee/Install \
   --file=/mnt/ee/ee.tar.Z
   ```

   Install displays messages about its activity, ending with the following output:

   ```
   Done.
   ```

4. ***Install the source code.***

   Use the following commands to install the source code:

   ```
   % cd /usr/cygnus/ee-<yymmdd>
   % uncompress < /mnt/src.tar.Z | tar xpf -
   ```

5. ***Install the HTML documentation.***

   Use the following commands to install the HTML documentation:

   ```
   % cd /usr/cygnus/ee-<yymmdd>
   % uncompress < /mnt/doc.tar.Z | tar xpf -
   ```

6. ***Build symbolic links to make executable paths easy to negotiate.***

   You may need root access to put the link in `/usr` directory. The *<host>* in the instruction on the fourth line stands for the host machine you're using (for example, `sparc-sun-solaris2.5.1`).

```
% cd /usr/cygnus
% ln -s ee-<yymmdd> ee
% su root
# ln -s /usr/cygnus/ee/H-<host> /usr/ee
# exit
```

7. ***Enable the problem reporting utility.***

   Using your Cygnus customer ID (see cover letter), enable the electronic mail tool for reporting problems. With the following command, enable the send-pr problem reporting database tool:

```
% /usr/ee/bin/install-sid <customer-ID>
```

8. ***Remove public write access restrictions from*** `/usr/cygnus`***.***

   See your system administrator for the correct permissions at your site.

   You are done. With `/usr/ee/bin` in your `PATH`, you can use the GNUPro Toolkit.

Installing on Win32

1. ***Insert the CD.***
2. ***Make sure that you have*** `unzip.exe`.

   It should be available on the CD.
3. ***Create a directory.***

   Use the following commands to create a directory called `C:\>CYGNUS` and `cd` into that directory:

   ```
   C:\> mkdir C:\CYGNUS
   C:\> cd C:\CYGNUS
   C:\CYGNUS>
   ```
4. ***Extract the zipfile.***

   Note: all subsequent instructions assume that you are installing from the CD and that the CD-ROM drive is `D:\`.

   Use the following command:

   ```
   C:\CYGNUS> d:\unzip d:\ee.zip
   ```

   For the Win32 toolchain the libraries are installed in different locations. Therefore, the Win32 hosted toolchain requires the following environmental settings to function properly. Assume the release is installed in `C:\CYGNUS`.

   ```
   SET PROOT=C:\cygnus\ee-<yymmdd>
   SET PATH=%PROOT%\H-i586-cygwin32\BIN;%PATH%
   SET INFOPATH=%PROOT%\info
   REM Set TMPDIR to point to a ramdisk if you have one
   SET TMPDIR=%PROOT%
   ```
5. ***Add the bin directory that is created by the extraction, to your path.***

   On Windows NT, click on **Start** -> **Settings** -> **Control Panel**. The Control Panel window opens. Select the **System** icon. In the System Properties window, select the **Environment** tab. Enter the appropriate settings (as shown below) in the Variable and Value dialog boxes.

   On Windows 95 or Windows 98, open the file `autoexec.bat` in the root directory of the boot drive. Add the following variable settings:

   ```
   SET PROOT=C:\cygnus\ee-<yymmdd>
   SET PATH=%PROOT%\H-i586-cygwin32\BIN;%PATH%
   ```
6. ***Set the*** `INFOPATH` ***environment variable.***

   Using the following command:

   ```
   SET INFOPATH=%PROOT%\info
   ```

7. ***Set the*** `TMPDIR` ***environment variable.***

   Use the following command:

   ```
   SET TMPDIR=%PROOT%
   ```

   You are done. Everything is set up and you can use the GNUPro Toolkit.

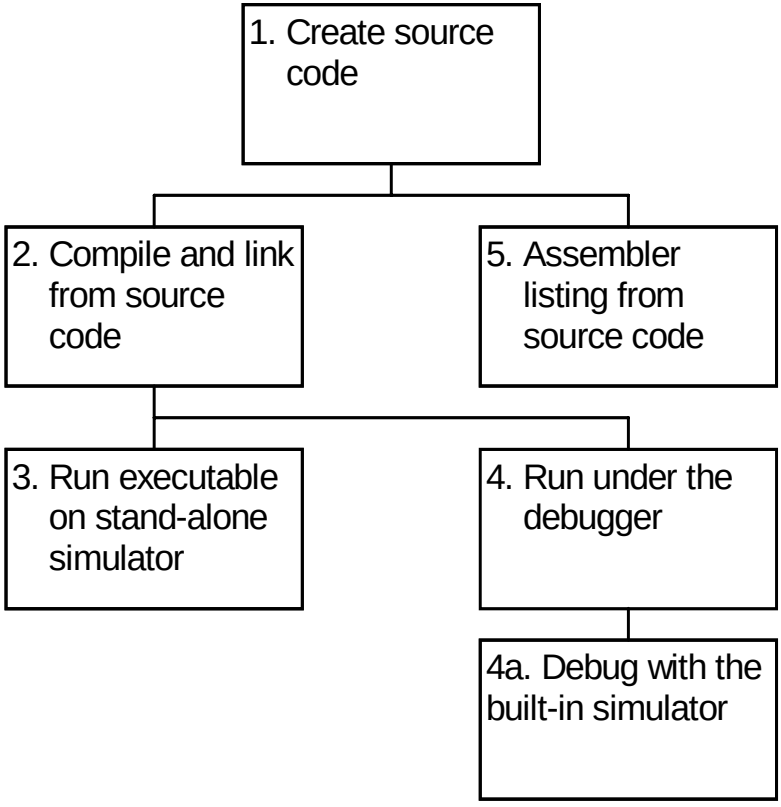
2

Procedures

This section demonstrates the main utilities. Follow the sequence to verify the correct installation and operation of the utilities. For more information, refer to the standard GNUPro documentation.

It is important to note that the GNUPro Toolkit is case-sensitive on UNIX operating systems. Therefore, if you are using UNIX you must enter all commands and options exactly as indicated in this document.

The following chart demonstrates one all-inclusive sequence; however, the steps following compilation are up to the user and may be considered optional.



Create Source Code

There are four sample programs provided to help you verify installation. These samples are located in Appendix A of this User's Guide. See "Sample Code" on page 85.

The sample source files are:

- `sky_main.c`
- `sky_test.dvpasm`
- `sky_test.vuasm`
- `sky_refresh.s`

The instructions in this section provide the commands needed to compile, assemble, link, and run the sample programs.

Compile, Assemble and Link Sample Code

Throughout these examples, screen samples are shown with a gray background. Code input is shown in plain monofont. Code output is shown in bold monofont. All samples are displayed in UNIX format. The UNIX command prompt is shown as **%**. On Win32 systems, the samples are exactly the same, but the command prompt is shown as **C:\>**.

To compile, assemble and link this example for the simulator, enter the following command:

```
% ee-gcc -g -c sky_main.c -o sky_main.o
% dvp-elf-as sky_test.dvpasm -o sky_test.o
% dvp-elf-as sky_refresh.s -o sky_refresh.o
% ee-gcc -g -Tsky.ld sky_main.o \
sky_test.o sky_refresh.o -o sky_main.exe
```

You must specify a linker script (for example, `sky.ld`). The `-T` option specifies the linker script. The syntax requires that there be no space between the `-T` and the linker script name.

The option `-g` generates debugging information and the option `-o` 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 perform no optimizations on the code. Refer to "GNU CC Command Options" in *Using GNU CC* in ***GNUPro Compiler Tools*** for a complete list of available options.

Run Executable on the Stand-Alone Simulator

To run this program on the stand-alone simulator, enter:

```
% ee-run sky_main.exe
```

On UNIX, the simulator executes the program, and displays a multi-color cube onscreen.

On Win32 systems, the text information will be sent to the screen.

Run Under Debugger

The GNU debugger, GDB can debug programs by using the built-in simulator (this does not require access to any hardware).

Debug with the built-in simulator

Run this executable under GDB using the built-in simulator, as follows:

```
% ee-gdb -nw sky_main.exe
GNU gdb 4.17-sky-980617
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License,
and you are welcome to change it and/or distribute copies of
it under certain conditions.
Type "show copying" to see the conditions. This version of GDB is
supported for customers of Cygnus Solutions. Type "show warranty"
for details.
This GDB was configured as "--host=sparc-sun-solaris2.5-gnulibc1
--target=ee"...
(gdb) target sim
Connected to the simulator.
(gdb) load
Loading section .text, size 0x21dc vma 0x10000
Loading section .rodata, size 0x100 vma 0x121e0
Loading section .vutext, size 0x20c30 vma 0x122e0
Loading section .data, size 0x728 vma 0x32f10
Loading section .dmadata, size 0x10 vma 0x33640
Loading section .sdata, size 0x38 vma 0x33650
Start address 0x10004
Transfer rate: 1160160 bits in <1 sec.
(gdb) break main
Breakpoint 1 at 0x10344: file sky_main.c, line 106.
(gdb) run
Starting program: sky_main.exe
CPU context is now master
```

```
Breakpoint 1, main () at sky_main.c:106
106  enable_cop2();
(gdb) step 3
main () at sky_main.c:109
109  start_DMA_ch1_source_chain(&My_dma_start);
(gdb) list
104  int main()
105  {
106  enable_cop2();
107
108  /* Send the data through the simulator.         */
109  start_DMA_ch1_source_chain(&My_dma_start);
110  wait_until_idle();
111
112  /* Send a screen refresh though the simulator.  */
113   start_DMA_ch1_source_chain(&screen_refresh);
(gdb) break 110
Breakpoint 2 at 0x1035c: file sky_main.c, line 110.
(gdb) continue
Continuing.

Breakpoint 2, main () at sky_main.c:110
110  wait_until_idle();
(gdb) set cpu vif1
(gdb) info registers
vif1_pc: 0x3364c    vif1_pcx: 0x7
     vif1_stat      vif1_fbrst        vif1_err       vif1_mark
      01000000        00000000        00000000        00000000
    vif1_cycle       vif1_mode        vif1_num       vif1_mask
      00000404        00000000        00000000        00000000
     vif1_code      vif1_itops       vif1_base       vif1_ofst
      20000000        00000000        00000000        00000000
     vif1_tops       vif1_itop        vif1_top        vif1_dbf
      00000000        00000000        00000000        00000000
       vif1_r0         vif1_r1         vif1_r2         vif1_r3
```

```
      00000000        00000000        00000000        00000000
       vif1_c0         vif1_c1         vif1_c2         vif1_c3
      00000000        00000000        00000000        00000000
(gdb) next
CPU context is now master

Breakpoint 2, main () at sky_main.c:110
110  wait_until_idle();
(gdb) backtrace
#0  main () at sky_main.c:110
(gdb) continue
Continuing.

Program exited normally.
```

Assembler Listing from Sample Code

If you want to see how the compiler implements certain language constructs, such as function calls, or if you want to learn more about assembly language, you can produce an assembly listing of a source module.

Use the following command to produce an assembler listing:

```
% ee-gcc -c -g -O2 -Wa,-alh sky_main.c
```

The compiler option `-c` compiles without linking. The compiler option `-g` gives the assembler the necessary debugging information. The `-O2` option produces more fully optimized code. The option `-Wa` tells the compiler to pass along the comma-separated list of options that follow it to the assembler. The assembler option `-alh` requests a listing with high-level language and assembler language interspersed.

Here is a partial excerpt of the output:

```
103:sky_main.c      ****
104:sky_main.c      **** int main()
105:sky_main.c      **** {
341                 .stabn 68,0,105,$LM35
342                 .stabs "sky_main.c",132,0,0,$Ltext0
343                 .ent   main
344                  main:
345                 .frame  $fp,32,$31
346                 .mask  0xc0000000,-16
347                 .fmask0x00000000,0
348 020c E0FFBD27    subu  $sp,$sp,32
349 0210 1000BF7F    sq $31,16($sp)
350 0214 0000BE7F    sq $fp,0($sp)
351 0218 2DF0A003    move  $fp,$sp
352 021c 0000000C    jal __main
352      00000000
353                  $LM36:
106:sky_main.c      ****   enable_cop2();
354                 .stabn 68,0,106,$LM36
355 0224 3500000C    jal enable_cop2
355      00000000
```

3

Reference

This section contains reference information about the various components of the SKY toolchain.

- To compile C or C++ source code, refer to “Compiler” on page 22.
- To learn how the Emotion Engine should interface with the operating system, refer to “Emotion Engine ABI Summary” on page 24.
- To assemble Emotion Engine source code, refer to “Emotion Engine Assembler” on page 31.
- To assemble DMA tags, GIF tags, VIF instructions, and VU instructions, refer to “DVP Assembler” on page 41.
- To link with GNU `ld`, refer to “Linker” on page 54.
- To debug with GDB, refer to “Debugger” on page 57.
- To use the SKY simulator, refer to “Simulator” on page 71.

Compiler

This section describes Emotion Engine-specific features of the GNUPro Compiler.

Emotion Engine-specific command-line options

For a list of available generic compiler options, refer to "GNU CC Command Options" in *Using GNU CC* in ***GNUPro Compiler Tools***. The defaults for the following Emotion Engine-specific command-line options have changed:

`-mhard-float`

This option is on by default. It causes the compiler to generate output containing floating point instructions.

To turn this option off, specify `-msoft-float`. This causes the compiler to generate output containing library calls for floating point operations.

`-msingle-float`

This option is on by default. It tells GCC to assume that the floating point co-processor only supports single precision operations.

To turn this option off, specify `-mdouble-float`. This permits GCC to use double precision operations.

`-EL`

This option is on by default. Compiles code for the processor in little endian mode.

`-G` *num*

Puts global and static items less than or equal to *num* bytes into the `small data` or `bss` section, instead of the `normal data` or `bss` section. This allows the assembler to emit one word memory reference instructions based on the global pointer (`gp` or `$28`), instead of the normal two words used. By default, *num* is set at 8 bytes. The `-G` *num* switch is also passed to the assembler and linker. All modules should be compiled with the same `-G` *num* value.

`-fno-edge-lcm`

Turns off all the new optimization work added for the flow graph edge based global common sub-expression elimination (gcse) load motion and store motion.

`-fno-edge-lm`

Turns off just the new load motion work, but still performs the edge based gcse optimization.

`-fno-edge-sm`

Turns off just the new store motion work, but still performs the edge based gcse optimization.

Preprocessor symbols

The compiler supports the following preprocessor symbols:

`__mips__`
`__ee__`
`__MIPSEL__`

Each of these is always defined.

`__mips_single_float`
`__mips_soft_float`

One of these will be defined depending on the floating point compilation mode. The default is `__mips_single_float.`

Emotion Engine-specific attributes

There are no Emotion Engine-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 more information.

Emotion Engine ABI Summary

Data types and alignment

This section describes the MIPS EABI, which the Emotion Engine tools adhere to, by default, with one exception. Pointers are 4 bytes instead of 8 bytes.

Data type sizes and alignments

The following table shows the size and alignment for all data types:

Table 1: Emotion Engine Data Type Sizes and Alignments

Type	*Size*	*Alignment*
`char`	1 byte	1 byte
`short`	2 bytes	2 bytes
`int`	4 bytes	4 bytes
`unsigned`	4 bytes	4 bytes
`long`	8 bytes	8 bytes
`long long`	8 bytes	8 bytes
`float`	4 bytes	4 bytes
`double`	8 bytes	8 bytes
`pointer`	4 bytes	4 bytes
`TItype`	16 bytes	16 bytes
`UTItype`	16 bytes	16 bytes

- Alignment within aggregates (structs and unions) is as above, with padding added if needed
- Aggregates have alignment equal to that of their most aligned member
- Aggregates have sizes which are a multiple of their alignment

You can specify 128-bit data types as follows:

```
typedef          int TItype  __attribute__ ((mode (TI)));
typedef unsigned int UTItype __attribute__ ((mode (TI)));
```

This defines two types, `TItype` and `UTItype`, which are 128-bit signed and unsigned types, respectively.

The only operations allowed on 128-bit types are `load, store, copy` and `constant` initialization. No arithmetic, logical, conversion, comparison or other operations are allowed. Any attempt to use an operation that is not allowed will cause the compiler to emit an error message.

Subroutine calls

The following describes the calling conventions for subroutine calls. The first table outlines the registers used for passing parameters. The second table outlines other register usage.

Table 2: Registers used for passing parameters

Parameter Registers:	
general-purpose	`r4-r11`
floating point	`f12-f19`

Table 3: Register usage

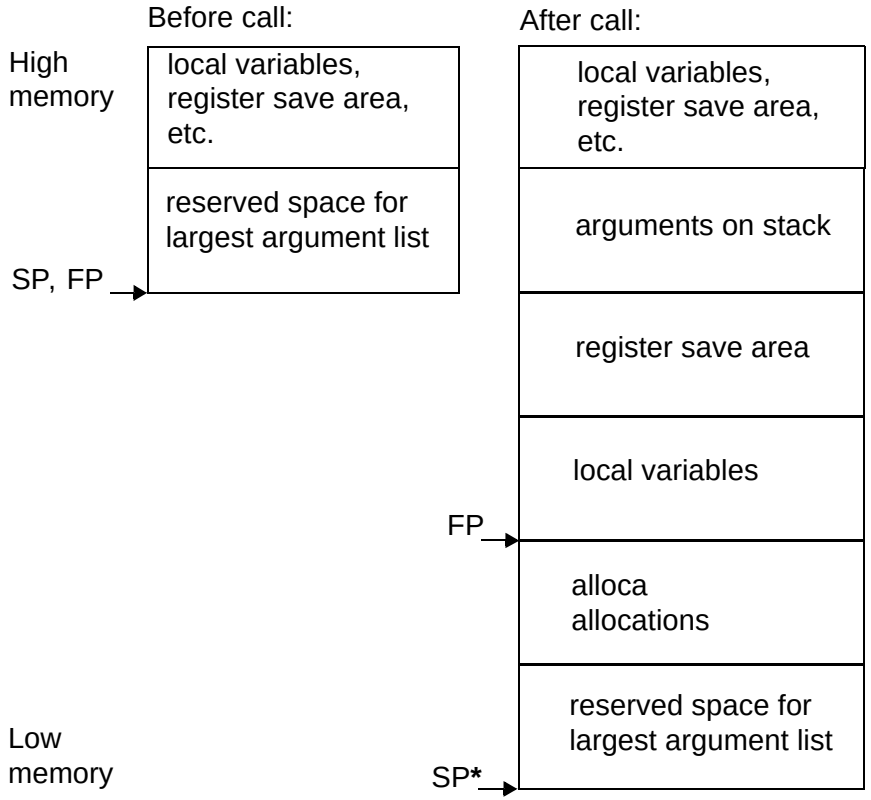
Register Usage:	
fixed 0 value	`r0`
volatile	`r1-r15, r24, r25`
non-volatile	`r16-r23, r30`
kernel reserved	`r26, r27`
gp (SDA base)	`r28`
stack pointer	`r29`
frame pointer	`r30` (if needed)
return address	`r31`

- General-purpose and floating point parameter registers are allocated independently.
- Structures that are less than or equal to 32 bits are passed as values.
- Structures that are greater than 32 bits are passed as pointers.

The stack frame

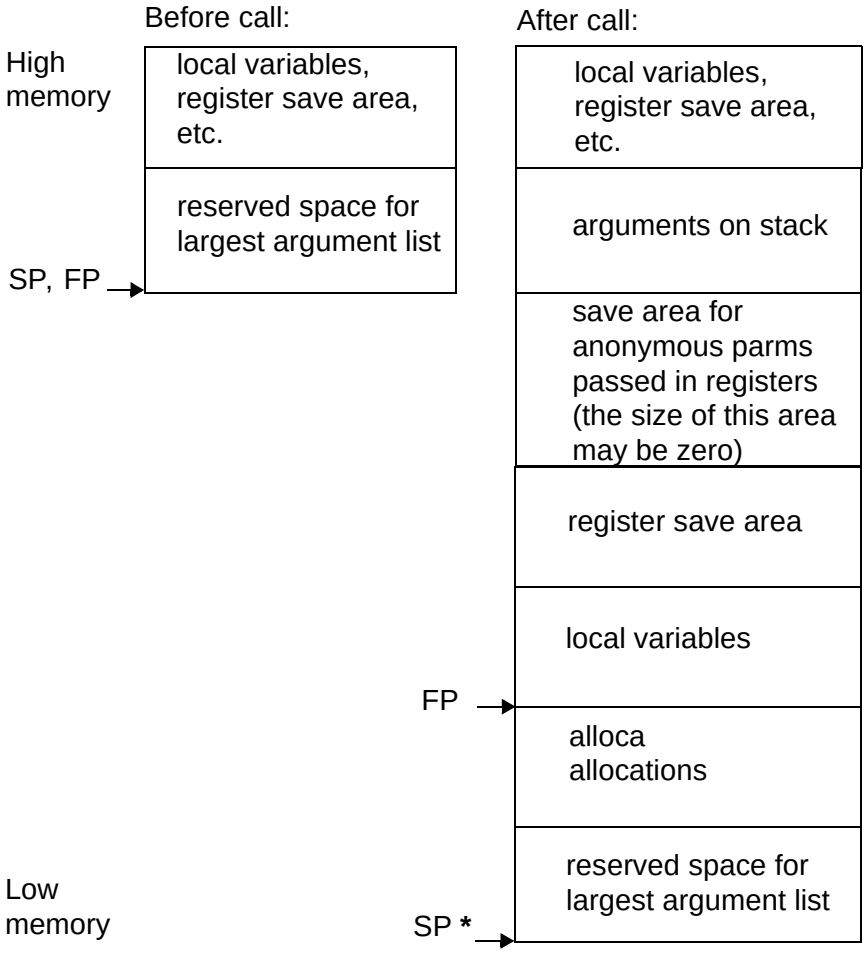
- The stack grows downwards from high addresses to low addresses.
- A leaf function need not allocate a stack frame if it does not need one.
- A frame pointer (FP) need not be allocated.
- The stack pointer (SP) shall always be aligned to 16-byte boundaries.

Stack frames for functions that take a fixed number of arguments look like this:



***** The frame pointer points to both the end of "local variables" and to the "reserved space for the largest argument list". In a function that calls “alloca”, FP may actually point into the “alloca allocations” area. In a function that does not call “alloca”, FP and SP will have the same value.

Stack frames for functions that take a variable number of arguments look like this:



***** The frame pointer points to both the end of "local variables" and to the "reserved space for the largest argument list". In a function that calls “alloca”, FP may actually point into the “alloca allocations” area. In a function that does not call “alloca”, FP and SP will have the same value.

Parameter assignment to registers

Consider the parameters in a function call as ordered from left (first parameter) to right. In this algorithm, FR contains the number of the next available floating-point register (or register pair for modes in which floating-point registers hold only 32 bits). `GR` contains the number of the next available general-purpose register. STARG is the address of the next available stack parameter word.

INITIALIZE:

Set GR=r4, FR=f12, and STARG to point to parameter word 1.

SCAN:

If there are no more parameters, terminate. Otherwise, select one of the following depending on the type of the next parameter:

DOUBLE OR FLOAT:

If FR > f19, go to STACK. Otherwise, load the parameter value into floating-point register FR and advance FR to the next floating-point register (or register pair in 32-bit mode). Then go to SCAN.

SIMPLE ARG:

A SIMPLE ARG is one of the following:

- One of the simple integer types that will fit into a general-purpose register
- A pointer to an object of any type
- A struct or union small enough to fit in a register
- A larger struct or union, which shall be treated as a pointer to the object or to a copy of the object (see below for when copies are made)

If GR > r11, go to STACK. Otherwise, load the parameter value into general-purpose register GR and advance GR to the next general-purpose register. Values shorter than the register size are sign-extended or zero-extended depending on whether they are signed or unsigned. Then go to SCAN.

LONG LONG in 32-bit mode:

If GR > r10, go to STACK. Otherwise, if GR is odd, advance GR to the next register. Load the 64-bit `long long` value into register pair GR and GR+1. Advance GR to GR+2 and go to SCAN.

STACK:

Parameters that are not otherwise handled above are passed in the parameter words of the caller's stack frame. SIMPLE ARGs, as defined above, are considered to have size and alignment equal to the size of a general-purpose register, with simple argument types shorter than this sign- or zero-extended to this width. Float arguments are considered to have size and alignment equal to the size of a floating-point register. In 64-bit mode, floats are stored in the low-order 32 bits of the 64-bit space allocated to them. double and long long are considered to have 64-bit size and alignment. Round STARG up to a multiple of the alignment requirement of the parameter and copy the argument byte-for-byte into STARG, STARG+1, ... STARG+size-1. Set STARG to STARG+size and go to SCAN.

Specified registers for local variables

Stores into local register variables may deleted when they appear to be dead according to dataflow analysis. References to local register variables may be deleted or moved or simplified.

Structure passing

As noted above, code that passes structures and unions by value is implemented specially. (In this section, "struct" will refer to structs and unions inclusively.) Structs small enough to fit in a register are passed by value in a single register or in a stack frame slot the size of a register. Larger structs are handled by passing the address of the structure. In this case, a copy of the structure will be made if necessary in order to preserve the pass-by-value semantics.

Copies of large structs are made under the following rules:

Table 4: Structure passing rules

	ANSI mode	***K&R Mode***
Normal param	Callee copies if needed	Caller copies
Varargs (...) param	Caller copies	Caller copies

In the case of normal (non-varargs) large-struct parameters in ANSI mode, the callee is responsible for producing the same effect as if a copy of the structure were passed, preserving the pass-by-value semantics. This may be accomplished by having the callee make a copy, but in some cases the callee may be able to determine that a copy is not necessary in order to produce the same results. In such cases, the callee may choose to avoid making a copy of the parameter.

Varargs handling

No special changes are needed for handling varargs parameters other than the caller knowing that a copy is needed on struct parameters larger than a register (see above).

The varargs macros set up a two-part register save area, one part for the general-purpose registers and one part for floating-point registers, and maintain separate pointers for these two areas and for the stack parameter area. The register save area lies between the caller and callee stack frame areas.

In the case of software floating-point, only the general-purpose registers need to be saved. Because the save area lies between the two stack frames, the saved register parameters are contiguous with parameters passed on the stack. This allows the varargs macros to be much simpler. Only one pointer is needed, which advances from the register save area into the caller's stack frame.

Function return values

Data types and register usage for return values.

Table 5: Function return values

Type	Register
`int`	`r2`
`short`	`r2`
`long`	`r2`
`long long`	`r2-r3` (32-bit mode)
`float`	`f0`
`double`	`f0-f1` (32-bit mode)
`struct/union`	see below

Structures and unions, which will fit into two general-purpose registers, are returned in `r2,` or in `r2` and `r3` if necessary. They are aligned within the register according to the endianness of the processor; e.g. on a big-endian processor the first byte of the struct is returned in the most significant byte of `r2,` while on a little-endian processor the first byte is returned in the least significant byte of `r2.` The caller handles larger structures and unions, by passing, as a "hidden" first argument, a pointer to space allocated to receive the return value.

Emotion Engine Assembler

MIPS-specific command-line options

For a list of available generic assembler options, refer to "Command-Line Options" in *Using AS* in ***GNUPro Utilities***.

`-EL`

Any MIPS configuration of the assembler can select big-endian or little-endian output at run time.

Use `-EL` for little-endian. The default is little-endian.

Syntax

For information about the MIPS instruction set, see ***MIPS RISC Architecture***, (Kane and Heinrich, Prentice-Hall). For an overview of MIPS assembly conventions, see "Appendix D: Assembly Language Programming" in the same volume.

Register names

There are thirty-two 64-bit general (integer) registers, named `'$0..$31'`. There are thirty-two 64-bit floating point registers, named `'$f0..$f31'`.

The symbols `$0` through `$31` refer to the general-purpose registers.

The following symbols can be used as aliases for individual registers:

Table 6: Emotion Engine Register names

Register	***Symbol***
`$at`	`$1`
`$kt0`	`$26`
`$kt1`	`$27`
`$gp`	`$28`
`$sp`	`$29`
`$fp`	`$30`

Assembler directives

This is a complete list of Emotion Engine assembler directives

Table 7: Emotion Engine Assembler directives

.abicalls	.dcb.b	.fail	.irepc	.psize
.abort	.dcb.d	.file	.irp	.purgem
.aent	.dcb.l	.fill	.irpc	.quad
.align	.dcb.s	.float	.lcomm	.rdata
.appfile	.dcb.w	.fmask	.lflags	.rep
.appline	.dcb.x	.format	.linkonce	.rept
.ascii	.debug	.frame	.list	.rva
.asciiz	.double	.global	.livereg	.sbttl
.asciz	.ds	.globl	.llen	.sdata
.balign	.ds.b	.gpword	.loc	.set
.balignl	.ds.d	.half	.long	.short
.balignw	.ds.l	.hword	.lsym	.single
.bgnb	.ds.p	.if	.macro	.skip
.bss	.ds.s	.ifc	.mask	.space
.byte	.ds.w	.ifdef	.mexit	.spc
.comm	.ds.x	.ifeq	.mri	.stabd
.common	.dword	.ifeqs	.name	.stabn
.common.s	.eject	.ifge	.noformat	.stabs
.cpadd	.else	.ifgt	.nolist	.string
.cpload	.elsec	.ifle	.nopage	.struct
.cprestore	.end	.iflt	.octa	.text
.data	.endb	.ifnc	.offset	.title
.dc	.endc	.ifndef	.option	.ttl
.dc.b	.endif	.ifne	.org	.verstamp
.dc.d	.ent	.ifnes	.p2align	.word
.dc.l	.equ	.ifnotdef	.p2alignl	.word
.dc.s	.equiv	.include	.p2alignw	.xdef
.dc.w	.err	.insn	.page	.xref
.dc.x	.exitm	.int	.plen	.xstabs
.dcb	.extern	.irep	.print	.zero

MIPS synthetic instructions for the Emotion Engine

The Emotion Engine GAS assembler supports the typical MIPS synthetic instructions (macros). What follows is a list of synthetic instructions supported by the assembler, as well as an example expansion of each instruction.

`R1, R2,` etc. represent integer register operands.

`I1, I2,` etc. represent integer immediate operands.

`F1, F2,` etc. represent floating-point register operands.

Note: `I?` represents an immediate integer value that is calculated from the alignment offset of one of the other immediate integer values, and is determined at assemble time.

Table 8: MIPS Synthetic Instructions

Instruction	*Expansion*

`abs R1 R2`	`bgez R2,end_abs`
	`move R1,R2`
	`neg R1,R2`
	`end_abs:`

`add R1 R2 I1`	`addi R1,R2,I1`
`addu R1 R2 I1`	`addiu R1,R2,I1`
`and R1 R2 I1`	`andi R1,R2,I1`
`beq R1 I1 I2`	`li $at,I1` `beq R1,$at,+I2`
`beql R1 I1 I2`	`li $at,I1` `beql R1,$at,+I2`
`bge R1 R2 I1`	`slt $at,R1,R2` `beqz $at,+I1`
`bge R1 I1 I2`	`slti $at,R1,I1` `beqz $at,+I2`
`bgel R1 R2 I1`	`slt $at,R1,R2` `beqzl $at,+I1`
`bgel R1 I1 I2`	`slti $at,R1,I1` `beqzl $at,+I2`
`bgeu R1 R2 I1`	`sltu $at,R1,R2` `beqz $at,+I1`
`bgeu R1 I1 I2`	`sltiu $at,R1,I1` `beqz $at,+I2`
`bgeul R1 R2 I1`	`sltu $at,R1,R2` `beqzl $at,+I2`

Table 8: MIPS Synthetic Instructions

Instruction	*Expansion*
`bgeul R1 I1 I2`	`sltiu $at,R1,I1` `beqzl $at,+I2`
`bgt R1 R2 I1`	`slt $at,R2,R1` `bnez $at,+I1`
`bgt R1 I1 I2`	`slti $at,R1,I1+1` `beqz $at,+I2`
`bgtl R1 R2 I1`	`slt $at,R2,R1` `bnezl $at,+I1`
`bgtl R1 I1 I2`	`slti $at,R1,I1+1` `beqzl $at,+I2`
`bgtu R1 R2 I1`	`sltu $at,R2,R1` `bnez $at,+I2`
`bgtu R1 I1 I2`	`sltiu $at,R1,I2-1` `beqz $at,+I2`
`bgtul R1 R2 I1`	`sltu $at,R2,R1` `bnezl $at,+I1`
`bgtul R1 I1 I2`	`sltiu $at,R1,I1+1` `beqzl $at,+I2`
`ble R1 R2 I1`	`slt $at,R2,R1` `beqz $at,+I1`
`ble R1 I1 I2`	`slti $at,R1,I1+1` `bnez $at,+I2`
`blel R1 R2 I1`	`slt $at,R2,R1` `beqzl $at +I2`
`blel R1 I1 I2`	`slti $at,R1,I1+1` `bnezl $at,+I2`
`bleu R1 R2 I1`	`sltu $at,R2,R1` `beqz $at,+I1`
`bleu R1 I1 I2`	`sltiu $at,R1,I1+1` `bnez $at,+I2`
`bleul R1 R2 I1`	`sltu $at,R2,R1` `beqzl $at,+I1`
`bleul R1 I1 I2`	`sltiu $at,R1,I1+1` `bnezl $at,+I2`
`blt R1 R2 I1`	`slt $at,R1,R2` `bnez $at,+I1`
`blt R1 I1 I2`	`slti $at,R1,I1` `bnez $at,+I2`

Table 8: MIPS Synthetic Instructions

Instruction	*Expansion*
`bltl R1 R2 I1`	`slt $at,R1,R2` `bnezl $at,+I1`
`bltl R1 I1 I2`	`slti $at,R1,I1` `bnezl $at,+I2`
`bltu R1 R2 I1`	`sltu $at,R1,R2` `bnez $at,+I1`
`bltu R1 I1 I2`	`sltiu $at,R1,I1` `bnez $at,+I2`
`bltul R1 R2 I1`	`sltu $at,R1,R2` `bnezl $at,+I1`
`bltul R1 I1 I2`	`sltiu $at,R1,I1` `bnezl $at,+I2`
`bne R1 I1 I2`	`li $at,I1` `bne R1,$at,+I2`
`bnel R1 I1 I2`	`li $at,I1` `bnel R1,$at,+I2`

`dabs R1 R2`	`bgez R2,end_dabs`
	`move R1,R2`
	`dneg R1,R2`
	`end_dabs:`

`dadd R1 R2 I1`	`daddi R1,R2,I1`
`daddu R1 R2 I1`	`daddiu R1,R2,I1`

`ddiv R1 R2 R3`	`bnez R3,L1_ddiv`
	`ddiv $zero,R2,R3`
	`break 0x7`
	`L1_ddiv:`
	`daddiu $at,$zero,-1`
	`bne R3,$at,L2_ddiv`
	`daddiu $at,$zero,1`
	`dsll32 $at,$at,0x1f`
	`bne R2,$at,L2_ddiv`
	`nop`
	`break 0x6`
	`L2_ddiv:`
	`mflo R1`

`ddiv R1 R2 I1`	`li $at,I1`
	`ddiv $zero,R2,$at`
	`mflo R1`

`ddivu R1 R2 I1`	`li $at,I1`
	`ddivu $zero,R2,$at`
	`mflo R1`

Table 8: MIPS Synthetic Instructions

Instruction	*Expansion*

`ddivu R1 R2 R3`	`bnez R3,L1_ddivu`
	`ddivu $zero,R2,R3`
	`break 0x7`
	`L1_ddivu`
	`mflo R1`

`div R1 R2 R3`	`bnez R3,L1_div`
	`div $zero,R2,R3`
	`break 0x7`
	`L1_div:`
	`li $at,-1`
	`bne R3,$at,L2_div`
	`lui $at,0x8000`
	`bne R2,$at,L2_div`
	`nop`
	`break 0x6`
	`L2_div:`
	`mflo R1`

`div R1 R2 I1`	`li $at,I1`
	`div $zero,R2,$at`
	`mflo R1`

`divu R1 R2 R3`	`bnez R3,L1_divu`
	`divu $zero,R2,R3`
	`break 0x7`
	`L1_divu:`
	`mflo R1`

`divu R1 R2 I1`	`li $at,I1`
	`divu $zero,R2,$at`
	`mflo R1`

`dla R1 I1(R2)`	`li R1,I1` `addu R1,R1,R2`
`dli R1 I1`	`li R1,I1`

`drem R1 R2 R3`	`bnez R3,L1_drem`
	`ddiv $zero,R2,R3`
	`break 0x7`
	`L1_drem`
	`daddiu $at,$zero,-1`
	`bne R3,$at,300`
	`daddiu $at,$zero,1`
	`dsll32 $at,$at,0x1f`
	`bne R2,$at,L2_drem`
	`nop`
	`break 0x6`
	`L2_drem:`
	`mfhi R1`

Table 8: MIPS Synthetic Instructions

Instruction	*Expansion*

`drem R1 R2 I1`	`li $at,I1`
	`ddiv $zero,R2,$at`
	`mfhi R1`

`dremu R1 R2 R3`	`bnez R3,L1_dremu`
	`ddivu $zero,R2,R3`
	`break 0x7`
	`L1_dremu:`
	`mfhi R1`

`dremu R1 R2 I1`	`li $at,I1`
	`ddivu $zero,R2,$at`
	`mfhi R1`

`dsub R1 R2 I1`	`daddi R1,R2,-I1`
`dsubu R1 R2 I1`	`daddiu R1,R2,-I1`
`flush R1 I1(R2)`	`lwr R1,I1(R2)`
`jal R1 R2`	`jalr R1,R2`
`jal R1`	`jalr R1`
`la R1 I1(R2)`	`li R1,I1` `addu R1,R1,R2`
`l.d F1 I1(R1)`	`ldc1 F1,I1(R1)`
`ldc3 R1 I1(R2)`	`ld R1,I1(R2)`
`li.d R1 I1`	`li R1,0x????` `dsll32 R1,R1,0xf`

`li.d F1 I1`	`li $at,0x????`
	`$at, $at,0xf`
	`dmtc1 $at,F1`

`li.s R1 I1`	`lui R1,0x????`
`li.s F1 I1`	`lui $at,0x????` `mtc1 $at,F1`
`lwc0 R1 I1(R2)`	`ll R1,I1(R2)`
`l.s F1 I1(R1)`	`lwc1 F1,I1(R1)`
`lcache R1 I1(R2)`	`lwl R1,I1(R2)`
`lwr R1 I1(R2)`	`lwr R1,I1(R2)`
`nor R1 R2 I1`	`ori R1,R2,I1` `nor R1,R1,$zero`
`or R1 R2 I1`	`ori R1,R2,I1`

Table 8: MIPS Synthetic Instructions

Instruction	*Expansion*

`rem R1 R2 R3`	`bnez R3,L1_rem`
	`div $zero,R2,R3`
	`break 0x7`
	`L1_rem:`
	`li $at,-1`
	`bne R3,$at,4c0`
	`lui $at,0x8000`
	`bne R2,$at,L1_rem`
	`nop`
	`break 0x6`
	`L2_rem:`
	`mfhi R1`

`rem R1 R2 I1`	`li $at,I1`
	`div $zero,R2,$at`
	`mfhi R1`

`remu R1 R2 R3`	`bnez R3,L1_remu`
	`divu $zero,R2,R3`
	`break 0x7`
	`L1_remu:`
	`mfhi R1`

`remu R1 R2 I1`	`li $at,I1`
	`divu $zero,R2,$at`
	`mfhi R1`

`rol R1 R2 R3`	`negu $at,R3`
	`srlv $at,R2,$at`
	`sllv R1,R2,R3`
	`or R1,R1,$at`

`rol R1 R2 I1`	`sll $at,R2,I1`
	`srl R1,R2,32-I1`
	`or R1,R1,$at`

`ror R1 R2 R3`	`negu $at,R3`
	`sllv $at,R2,$at`
	`srlv R1,R2,R3`
	`or R1,R1,$at`

`ror R1 R2 I1`	`srl $at,R2,I1`
	`sll R1,R2,32-I1`
	`or R1,R1,$at`

`sdc3 R1 I1(R2)`	`sd R1,I1(R2)`
`s.d F1 I1(R1)`	`sdc1 F1,I1(R1)`
`seq R1 R2 R3`	`xor R1,R2,R3` `sltiu R1,R1,1`

Table 8: MIPS Synthetic Instructions

Instruction		*Expansion*	
seq	R1 R2 I1	xori sltiu	R1,R2,I1 R1,R1,1
sge	R1 R2 R3	slt xori	R1,R2,R3 R1,R1,0x1
sge	R1 R2 I1	slti xori	R1,R2,I1 R1,R1,0x1
sgeu	R1 R2 R3	sltu xori	R1,R2,R3 R1,R1,0x1
sgeu	R1 R2 I1	sltiu xori	R1,R2,I1 R1,R1,0x1
sgt	R1 R2 R3	slt	R1,R3,R2
sgt	R1 R2 I1	li slt	$at,I1 R1,$at,R2
sgtu	R1 R2 R3	sltu	R1,R3,R2
sgtu	R1 R2 I1	li sltu	$at,I1 R1,$at,R2
sle	R1 R2 R3	slt xori	R1,R3,R2 R1,R1,0x1
sle	R1 R2 I1	li slt xori	$at,I1 R1,$at,R2 R1,R1,0x1
sleu	R1 R2 R3	sltu xori	R1,R3,R2 R1,R1,0x1
sleu	R1 R2 I1	li sltu xori	$at,I1 R1,$at,R2 R1,R1,0x1
slt	R1 R2 I1	slti	R1,R2,I1
sltu	R1 R2 I1	sltiu	R1,R2,I1
sne	R1 R2 R3	xor sltu	R1,R2,R3 R1,$zero,R1
sne	R1 R2 I1	xori sltu	R1,R2,I1 R1,$zero,R1
sub	R1 R2 I1	addi	R1,R2,-I1
subu	R1 R2 I1	addiu	R1,R2,-I1
swc0	R1 I1(R2)	sc	R1,I1(R2)
s.s	F1 I1(R1)	swc1	F1,I1(R1)

Table 8: MIPS Synthetic Instructions

Instruction	*Expansion*
`scache R1 I1(R2)`	`swl R1,I1(R2)`
`invalidate R1 I1(R2)`	`swr R1,I1(R2)`
`teq R1 I1`	`teqi R1,I1`
`tge R1 I1`	`tgei R1,I1`
`tgeu R1 I1`	`tgeiu R1,I1`
`tlt R1 I1`	`tlti R1,I1`
`tltu R1 I1`	`tltiu R1,I1`
`tne R1 I1`	`tnei R1,I1`
`trunc.w.d F1 F2 R1`	`trunc.w.d F1,F2`
`trunc.w.s F1 F2 R1`	`trunc.w.s F1,F2`
`uld R1 I1(R2)`	`ldl R1,I?(R2)` `ldr R1,I1(R2)`

`ulh R1 I1(R2)`	`lb R1,I?(R2)`
	`lbu $at,I1(R2)`
	`sll R1,R1,0x8`
	`or R1,R1,$at`

`ulhu R1 I1(R2)`	`lbu R1,I?(R2)`
	`lbu $at,I1(R2)`
	`sll R1,R1,0x8`
	`or R1,R1,$at`

`ulw R1 I1(R2)`	`lwl R1,I?(R2)` `lwr R1,I1(R2)`
`usd R1 I1(R2)`	`sdl R1,I?(R2)` `sdr R1,I1(R2)`

`ush R1 I1(R2)`	`sb R1,I1(R2)`
	`srl $at,R1,0x8`
	`sb $at,I?(R2)`

`usw R1 I1(R2)`	`swl R1,I?(R2)` `swr R1,I1(R2)`
`xor R1 R2 I1`	`xori R1,R2,I1`

DVP Assembler

The DVP Assembler allows a combination of DMA tags, GIF tags, VIF instructions, and VU instructions to be assembled from a single input stream to produce an ELF object file.

Emotion Engine-specific command-line options

For a list of available generic assembler options, refer to "Command-Line Options" in *Using AS* in ***GNUPro Utilities***.

`-no-dma`

Do not include DMA instructions in the output.

`-no-dma-vif`

Do not include DMA or VIF instructions in the output.

Syntax

For a list of available generic description of assembler syntax, refer to "Syntax" in *Using AS* in ***GNUPro Utilities***.

There are two types of comments: inline comments and line comments. In both cases, the comment is equivalent to one space. Anything from `'/*'` to `'*/'` is an inline comment. Inline comments can span multiple lines but cannot be nested. A line comment is everything from `';'` to the next newline.

A statement ends at a newline character (`\n`). There is no line separator character; there can be no more than one statement on a line.

A statement begins with an optional label, optionally followed by a key symbol and appropriate operands. A label is a symbol that is immediately followed by a colon (:). The key symbol (opcode) determines the syntax of the rest of the statement. The mode of the assembler determines which set of opcodes is acceptable.

Literals

For a generic description of numeric and character constants (literals), refer to "Syntax" in *Using AS* in ***GNUPro Utilities***.

The DVP and MIPS assemblers support a new format for quadword literals. The format of this literal is `'0x'` followed by four 1-word hexadecimal values separated by the underscore (`'_'`) character.

For example, the literal:

```
0x333_0_12345678_1
```

can be used, and is equivalent to:

```
0x00000333000000000123456780000001
```

Directives

The DVP assembler accepts a subset of the directives described in *Using AS* in ***GNUPro Utilities***. In addition, the DVP assembler also accepts a number of new directives.

This is a complete list of DVP assembler directives:

Table 9: DVP Assembler directives

.ascii	.EndGif	.if	.quad
.asciz	.endm	.ifdef	.rept
.balign	.EndMpg	.ifndef	.sbttl
.byte	.EndUnpack	.include	.section
.data	.equ	.int	.set
.DmaData	.equiv	.irp	.short
.DmaPackVif	.err	.irpc	.skip
.eject	.exitm	.list	.space
.else	.extern	.macro	.string
.EndDirect	.fill	.nolist	.text
.EndDmaData	.float	.org	.title
.endfunc	.func	.p2align	.vu
.endif	.global	.psize	.word

For information on most supported directives, refer to *Using AS* in ***GNUPro Utilities.*** Some existing assembler directives have been modified for the DVP Assembler. In addition, the DVP Assembler also supports several new directives.

The following list describes the new and modified directives that the DVP assembler supports:

`.DmaData`

Builds a labeled block of DMA data for use with `DMAref`.

`.DmaPackVif`

Specifies whether a DMA tag will be packed with two subsequent VIF instructions. The argument to `.DmaPackVif` is either `'1'` or `'0'`.

`.EndDirect`

Terminates the list of data following `direct` or `directhl` VIF opcodes.

`.EndDmaData`

Terminates the list of data following `.DmaData`.

`.EndGif`

Terminates the list of data following `GIFpacked`, `GIFreglist` and `GIFimage`.

`.endfunc`

Denotes the end of the function started with `.func`.

`.endm`

Terminates a macro definition.

`.EndMpg`

Terminates the list of VU instructions following `mpg`.

`.EndUnpack`

Terminates the list of data following `unpack`.

`.func`

Emits debugging information for user-written assembler functions. Must assemble with `--gstabs` for this directive to have any effect.

Syntax: `.func function_name [,symbol_name]`

Note that `symbol_name` is optional. It is used when the function name, as viewed from the debugger, is different than what is written in the assembler code (for example, some systems have a leading underscore (`'_'`) for C functions).

`.quad`

Assembles a 16-byte integer.

`.vu`

Sets the assembler to compile VU instructions.

Must be present before any VU instructions are assembled.

`.word`

Assembles a 4-byte integer.

VU opcodes

For detailed information on the VU machine instruction set, see ***VU Specifications Version 2.10***.

To set the `I, E, M, D` and `T` bits in an instruction, specify the appropriate letters as a bracketed suffix to the upper opcode. Case is ignored.

For example:

```
NOP[DT]     IADDIU  VI01, VI00, LOOP
```

VIF opcodes

For detailed information on the VIF instruction set, see ***SCEI CPU2 Specifications Version 2.10***. (Note that the "**PKE**" has been renamed to "**VIF**".)

Some tags support the following optional bits, enclosed in brackets "[]" as a suffix:

- `i:` `interrupt`
- `m:` `mask` (`unpack` instruction only)
- `r:` `TOPS` relative address (`unpack` instruction only)
- `u:` unsigned `UNPACK` (`unpack` instruction only)

The following instructions are supported:

`base[i]` *`<expr>`*

Sets the VIF1 Base register.

Notes:

1. Only bits 15:0 of the expression are encoded into the instruction
2. Only bits 9:0 are used by the register.

`direct[i]` *`<filename>`*
`direct[i]` *`<expr>`*
`direct[i] *`

Send data directly to GIF via path 2. The operand is either the name of a binary file containing only GIF tagged data, or the number of 128-bit quadwords following, or an asterisk for the assembler to compute the number of quadwords following. The last two forms of `direct` should end the data with the directive `.EndDirect`. Only valid for execution on VIF1.

Examples:

```
DIRECT *
GIFpacked REGS={A_D}, NLOOP=13, EOP
.int  0x000a0000, 0x00000000, 0x0000004c, 0x00000000
.int  0x027f0000, 0x01df0000, 0x00000040, 0x00000000
.int  0x00000001, 0x00000000, 0x0000001a, 0x00000000
.int  0x01000096, 0x00000000, 0x0000004e, 0x00000000
.int  0x00000001, 0x00000000, 0x00000046, 0x00000000
```

```
  .int  0x00000000, 0x00000000, 0x00000047, 0x00000000
  .int  0x00000000, 0x00000000, 0x00000018, 0x00000000
  .int  0x00000006, 0x00000000, 0x00000000, 0x00000000
  .int  0x00000000, 0x00000000, 0x00000001, 0x00000000
  .int  0x00000000, 0x00000000, 0x00000004, 0x00000000
  .int  0x1e002800, 0x00000000, 0x00000004, 0x00000000
  .int  0x00070000, 0x00000000, 0x00000047, 0x00000000
  .int  0x00006c00, 0x00007100, 0x00000018, 0x00000000
  .EndGif
  .EndDirect
```

`directhl[i] <filename>`
`directhl[i] <expr>`
`directhl[i] *`

The syntax of `directhl` is identical to that of `direct`.

`flush[i]`

Waits until the VU1 program and Paths 1 and 2 to GIF are idle. Only valid for execution on VIF1.

`flusha[i]`

Waits until the VU1 program and all Paths to GIF are idle. Only valid for execution on VIF1.

`flushe[i]`

Waits until the VU program is idle.

`itop[i] <expr>`

Sets the VIF ITOPS register.

Notes:

1. Only bits 15:0 of the expression are encoded into the instruction.
2. Only bits 9:0 are used by the register.

`mark[i] <abs-expr>`

Sets the VIF Mark register. Only bits 15:0 of the expression are encoded into the instruction.

`mpg[i] <expr>, <filename>`
`mpg[i] <expr>, <expr>`
`mpg[i] <expr>, *`

As `flushe`, then load a program into micro memory. The first operand is always the address at which to load the program. If specified as "*", the current value of `$.MpgLoc` is used. The second operand is either the name of a binary file containing only VU instructions, or the number of following 64-bit doublewords, or an asterisk for the assembler to compute the number of following doublewords.

The assembler will automatically insert VIF code (`mpg`, `direct` or `directhl`) if the written data is longer than the specified length. In addition to assembling the `mpg` instruction, `mpg` causes the following to be executed:

```
$.MpgLoc = value of 1st operand ;update the mpg location counter
$. = $.MpgLoc ;set the secondary location counter
```

The symbol `$.MpgLoc` is reserved for the `mpg` location counter. Users should not set it. The last two forms of `mpg` should end the VU instructions with the directive `.EndMpg`. It will switch the assembler back to non-VU mode and cause the following to be executed:

```
$.MpgLoc = $. ;update the mpg location counter
```

Examples:

```
        mpg 0x20,"test00.vubin"
        mpg *,*
 main:  NOP IADDIU VI01,VI00,0 ;Euler angle,transfer vector
        NOP IADDIU VI02,VI00,22 ; omatrix(SCREEN/LOCAL)
        NOP NOP
        NOP BAL VI15,$RotMatrix
        NOP NOP
        NOP END ;wait VIF
        NOP NOP
        NOP B $main ;repeat
        NOP NOP
        .EndMpg
```

`mscal[i]` *<expr>*

As `flushe,` then sets the VU1 start address to *<expr>* and executes via `callms.`

`mscalf[i]` *<expr>*

As `flush,` then sets the VU1 start address to *<expr>* and executes via `callms.`

`mscnt[i]`

As `flushe,` then continues to execute a micro program from the next address of the last program counter on VU.

`mskpath3[i]` *<ability>*

Enables or disables path 3 transfers.

Notes:

1. Bit 15 of the instruction is set via *<ability>*

   DISABLE = 1 AND ENABLE = 0.

2. Bits 14:0 are always set to 0.
3. Only valid for execution on VIF1.

`offset[i] <expr>`

Sets the VIF1 Offset register.

Notes:

1. Only bits 15:0 of the expression are encoded into the instruction.
2. Only bits 9:0 are used by the register.

`stcol[i] <expr0>, <expr1>, <expr2>, <expr3>`

Set the VIF `COL` registers `C0, C1, C2` and `C3`. All 32-bits are used from each expression.

`stcycl[i] <abs-expr>, <abs-expr>`

Sets the VIF `WL` and `CL` values in the cycle register from the 1st and 2nd operands, respectively. Only bits 7:0 are used from each expression.

`stmask[i] <expr>`

Sets VIF `MASK` register to *`<expr>.`* All 32 bits are used.

`stmod[i] <mode>`

Sets the VIF `MODE` register, where *`<mode>`* is one of `DIRECT, ADD,` or `ADDROW.`

Notes:

1. Bits 15:2 of the instruction are always set to 0.
2. Bits 1:0 are set via *`<mode>`*: `DIRECT = 0x00, ADD = 0x01,` and `ADDROW = 0x02`

`strow[i] <expr0>, <expr1>, <expr2>, <expr3>`

Sets the VIF `ROW` registers `R0`, `R1`, `R2` and `R3`. All 32 bits are used from each expression.

`unpack[imru] <wl>, <cl>, <type>, <addr>, <filename>`
`unpack[imru] <wl>, <cl>, <type>, <addr>, *`
`unpack[imru] <wl>, <cl>, <type>, <addr>, <num>`

Unpack data to an address in VU memory according to type.

The above forms emit two machine instructions. The first, `stcycl,` sets the VIF `WL` and `CL` values in the cycle register from the 1st and 2nd operands, respectively. The second, `unpack,` uses the remaining operands. These instructions are emitted together because assembling the unpack instruction requires knowledge of the `WL` and `CL` values.

The following forms are deprecated, and assume the WL and CL values from the most recently assembled stcycl instruction:

```
unpack[imru] <type>, <addr>, <filename>
unpack[imru] <type>, <addr>, *
unpack[imru] <type>, <addr>, <num>
```

Unpack data to an address in VU memory according to type. The *<type>* operand is the type of data.

```
S_32 scalar, 32-bit
S_16 scalar, 16-bit
S_8 scalar, 8-bit
V2_32 vector of 2 * 32-bit
V2_16 vector of 2 * 16-bit
V2_8 vector of 2 * 8-bit
V3_32 vector of 3 * 32-bit
V3_16 vector of 3 * 16-bit
V3_8 vector of 3 * 8-bit
V4_32 vector of 4 * 32-bit
V4_16 vector of 4 * 16-bit
V4_8 vector of 4 * 8-bit
```

V4_5 vector of 4 * 5-bit (actually 1*1-bit and 3*5-bit)

The *<addr>* operand is always the address at which to load the data. The final operand is either the name of a binary file containing only data, or an asterisk for the assembler to compute the number of data writes to be performed.

A PackV4_5 macro will be provided to encode the V4_5 packed data for this instruction. An example of its use is:

Examples:

```
Unpack 1, 1, V4_5, loadAddr, *
PackV4_5 1,29,7,12
PackV4_5 0,18,31,4
.EndUnpack

unpack 1, 1, V4_32, loadAddr, "euler.vubin"
unpack 1, 1, V4_32, 0X20, *
.float   0,   -0.7, 0,  0.7 ; Euler angle (sin)
.float   0,   -0.7, 0, -0.7 ; Euler angle (cos)
.float 1.0, 1024.0, 0,    0 ; Transfer vector
.EndUnpack
```

```
            unpack 1, 1, V3_8, 0X40, *
            .byte 0x00, 0x10, 0x20
            .byte 0x30, 0x40, 0x50
            .EndUnpack
```

`vifnop[i]`
: VIF no-op instruction.

DMA tag instructions

"Source" DMA (from memory) is done by one of eight DMA tags:

`DMAcnt`
: Inline data, inline next-tag

`DMAnext`
: Inline data, ptr to next-tag

`DMAref`
: `ptr` to data, inline next-tag

`DMArefe`
: `ptr` to data, inline next-tag and stop

`DMArefs`
: `ptr` to data, inline next-tag (stall control)

`DMAcall`
: Inline data, ptr to next-tag (push `addr` of inline tag)

`DMAret`
: Inline data, pop next-tag `addr`

`DMAend`
: Inline data and stop

“Destination" DMA (to memory) is not supported.

Each tag supports the following optional bits, enclosed in brackets "[]" as a suffix:

- `0` D_PCR.PCE=0
- `1` D_PCR.PCE=1
- `I` Interrupt request
- `S` SPR address (N/A for DMA channels 0, 1, 2)

Most tags have two operands: a quadword count and address. For `DMAcnt`, `DMAret` and `DMAend` the second operand is not permitted.

The DMA tags may be coded in one of two forms:

`*, DmaData_label`

The asterisk indicates that the quadword count should be taken from the attributes of the label on a `DmaData` macro.

`QWcount, address`

The quadword count is specified along with the address or label of the data block.

Since DMA tags must be 16-byte aligned but are only 8 bytes long, the hardware supports packing two VIF instructions into the unused upper words. By default, the assembler will automatically pack the next two instructions into a DMA tag if they are VIF instructions. This behavior can be disabled by `.DmaPackVif 0` or re-enabled with `.DmaPackVif 1.`

If the assembler is not filling the unused upper words with subsequent VIF instructions, they will be set to VIF `NOP` instructions.

Examples of how to use DMA tags:

```
DMAref[I1] *, data1
DMAend  4
.word   0, 0, 0, 0
.EndDmaData

.DmaData data1
MPG     *, *
NOP     IADDIU  VI01, VI00, 0x00000100
NOP     LQI.xyzw VF04, (VI01++)
NOP[d]  LQI.xyzw VF05, (VI01++)
NOP     LQI.xyzw VF06, (VI01++)
NOP     LQI.xyzw VF07, (VI01++)
NOP[e]  NOP
NOP     NOP
.endmpg
.EndDmaData
```

GIF tag instructions

There are three forms of the GIF tag: `PACKED`, `REGLIST` and `IMAGE`.

```
GIFpacked  PRIM=0Xxx, REGS={ rr, rr, ..}, NLOOP=c, EOP
```

`PRIM`

Optional.

The least-significant 11 bits will be transferred to the `PRIM` register. The other bits are ignored.

`REGS`

Required.

One to sixteen instances of the following register names may be specified in any order, combination or repetitions.

The sixteen possible register names and encodings are:

```
0x0 PRIM
0x1 RGBAQ
0x2 ST
0x3 UV
0x4 XYZF2
0x5 XYZ2
0x6 TEX0_1
0x7 TEX0_2
0x8 CLAMP_1
0x9 CLAMP_2
0xA XYZF
0xB RESERVED
0xC XYZF3
0xD XYZ3
0xE A_D
0xF NOP  (skips a quadword of data)
```

NLOOP

Optional. If unspecified, its value is computed from the location of `.EndGif.`

The least-significant 15 bits are used as an iteration count for the list of registers. The length of the following data must be `NREG*NLOOP` quadwords.

EOP

Optional.

If specified, there is no subsequent primitive in this packet

(End-Of-Packet).

Note:

1. If both `PRIM` and `REGS` are given, the `PRIM` value is transferred first.
2. The assembler will produce a warning if the length of the following data is not `NREG*NLOOP` 128-bit quadwords.
3. The tag and any data must be followed by the pseudo-op `.EndGif.`

```
GIFreglist  REGS={ rr, rr, ..}, NLOOP=c, EOP
```

REGS

Required.

One to sixteen instances of the following register names may be specified in any order, combination or repetitions.

The sixteen possible register names and encodings are:

```
0x0 PRIM
0x1 RGBAQ
0x2 ST
0x3 UV
0x4 XYZF2
0x5 XYZ2
0x6 TEX0_1
0x7 TEX0_2
0x8 CLAMP_1
0x9 CLAMP_2
0xA XYZF
0xB RESERVED
0xC XYZF3
0xD XYZ3
0xE A_D
```

`0xF NOP` (skips 64 bits of data)

`NLOOP`

Optional. If unspecified, its value is computed from the location of `.EndGif`.

The least-significant 15 bits are used as an iteration count for the list of registers.

`EOP`

Optional.

If specified, there is no subsequent primitive in this packet

(End-Of-Packet).

Notes:

1. The length of subsequent data must be `NREG*NLOOP` 64-bit words.
2. The data is contained in quadwords. If the number of 64-bit words is odd, the most-significant 64-bits of the last quadword are ignored.
3. The data must be followed by the pseudo-op `.EndGif`.

`GIFimage NLOOP=c, EOP`

`NLOOP`

Optional. If unspecified, its value is computed from the location of `.EndGif`.

The least-significant 15 bits are used as an count of the number of data values to be transferred to `HWREG (0x54)`.

`EOP`

Optional.

If specified, there is no subsequent primitive in this packet

(End-Of-Packet).

Notes:

1. The length of subsequent data must be `NLOOP` 128-bit quadwords.
2. The data must be followed by the pseudo-op `.EndGif`.

Linker

SKY-specific command-line options

For a list of available generic linker options, refer to "Command Language" in *Using LD* in ***GNUPro Utilities***. There are no SKY-specific command-line linker options.

Linker script

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.

Note: Sections `.lit4` and `.lit8` are used for storing integer and double literals. From their position in the linker script, they appear to be referenced via `_gp`. The sections `.rdata` and `.rodata` are used to store read-only data.

The following linker script is the contents of `sky.ld`

```
/* This is just the basic idt.ld linker script file, except that it has
   a different OUTPUT_ARCH to force the linker into EE mode which
   uses 32bit addresses.  */

ENTRY(_start)
OUTPUT_ARCH("mips:5900")
OUTPUT_FORMAT("elf32-littlemips")
GROUP(-lc -lidt -lgcc)
SEARCH_DIR(.)
__DYNAMIC  =  0;

/*
* Allocate the stack to be at the top of memory, since the stack grows
* down.  (The first access will pre-decrement to the top of memory.)
*/
PROVIDE (__stack = 0);

/*
* The following two init_hook symbols are referenced by the supplied
* crt0 startup code.  If they are set to a non-zero value by
* linking in a routine by that name, they will be called during startup.
* If some other startup code is used, the following definitions are
* not needed.
```

```
*/
PROVIDE (hardware_init_hook = 0);
PROVIDE (software_init_hook = 0);

SECTIONS
{
  . = 0x00010000;
  .text : {
     _ftext = . ;
    *(.init)
     eprol  =  .;
    *(.text)
    *(.mips16.fn.*)
    *(.mips16.call.*)
    *(.rel.sdata)
    *(.fini)
     etext  =  .;
     _etext  =  .;
  }
  . = .;
  .rdata : {
    *(.rdata)
  }
   _fdata = ALIGN(16);
  .data : {
    *(.data)
    CONSTRUCTORS
  }

/*
* The processor has a number of instructions that can access data more
* efficiently with a restricted offset range, in this case +-32K.
* Therefore, small data items are collected together in memory and
* accessed relative to _gp.  Since the maximum size of the data is 64K,
* we ensure that all of that data can be accessed by setting _gp to the
* middle of the area (beginning + 32K).
*/
  . = ALIGN(8);
  _gp = . + 0x8000;
  __global = _gp;
```

```
  .lit8 : {
    *(.lit8)
  }
  .lit4 : {
    *(.lit4)
  }
  .sdata : {
    *(.sdata)
  }
  . = ALIGN(4);
   edata  =  .;
   _edata  =  .;
   _fbss = .;
  .sbss : {
    *(.sbss)
    *(.scommon)
  }
  .bss : {
    _bss_start = . ;
    *(.bss)
    *(COMMON)
  }
   end = .;
   _end = .;

  /* DVP overlay support */
  .DVP.ovlytab 0 : { *(.DVP.ovlytab) }
  .DVP.ovlystrtab 0 : { *(.DVP.ovlystrtab) }

  /* interrupt vectors, for BEV=0 */
  .eit_v 0x80000180 : { *(.eit_v) }
}
```

Debugger

The debug environment is based on GDB, a powerful debugger that has already been ported to many platforms. However, there are several features of the SKY simulator that impose special challenges with respect to the GDB interface. In particular, GDB was not designed to handle multiple heterogeneous processors, or the multiple memory spaces introduced by the local VU instruction and data memory.

This section describes the extensions that have been added to GDB in order to support the SKY simulator. Any commands or functionality not explicitly mentioned are assumed to behave as in standard GDB.

You can communicate with the simulator through a single GDB session, which supports debugging of the Emotion Engine core processor and the VU and VIF units. Although GDB always assumes that the complete system is present, you can use it to debug subsystems A and B. For more information about the subsystems, see “Subsystems A, B and C” on page 76. GDB does not support source-level debugging of the GIF.

Running the simulator under GDB

GDB's built-in software simulation of the Emotion Engine processor allows you to debug programs compiled for the Emotion Engine without access to actual hardware. Activate this mode in GDB by entering `target sim`. Any run-time arguments that can be specified on the `ee-run` command line can also be specified here. Then, load code into the simulator by entering the `load` command. Debug it in the normal fashion.

If the simulator needs to be restarted from the beginning, use the following command sequence:

```
(gdb) sim reset
(gdb) load
(gdb) run
```

This ensures that all devices are reinitialized appropriately.

SKY-specific commands

Processor contexts serve as the primary method of differentiating between multiple processors and address spaces.

You can set or display the processor contexts using the following commands:

`set cpu` *`processor`*

Sets the current CPU context, where *`processor`* is one of:

`master`	Commands are relative to Emotion Engine processor and address space.
`vu0`	Commands are relative to the VU0 processor and its local memory.
`vu1`	Commands are relative to the VU1 processor and its local memory.
`vif0`	Commands are relative to the VIF0 unit.
`vif1`	Commands are relative to the VIF1 unit.
`auto`	Commands refer to the overall system, but will switch to one of the above contexts when certain events are generated.

`show cpu`

Displays the current CPU context.

By specifying the processor context, you provide a reference point for commands that are otherwise ambiguous.

For example, there are five sets of registers between the Emotion Engine, the two VUs, and the two VIF units. By default (i.e. when the context is `auto`), the `info reg` command will show all five sets of registers. If the VU1 context is selected (using `set cpu vu1`), the same command will show only the registers of processor VU1.

To allow easy identification of the current CPU context, the following command has been added:

`set context-in-prompt on|off|1|0|yes|no`

When `on`, shows the current CPU context as part of the prompt. If the CPU context is `auto`, the original prompt is used. When this feature is `off`, GDB will print a single line identifying the new CPU context every time it changes.

`show context-in-prompt`

Shows the current setting of the context-in-prompt feature.

Memory addresses

Memory on the SKY board is natively accessed with three different address granularities: the Emotion Engine uses byte addressability; VU text memory is accessed with double-word granularity; and VU data memory uses quadword addressing. Internally, GDB always assumes byte addressing. However, the casting facilities of GDB can be used to facilitate address specifications while working in the VU or VIF contexts.

Some examples are:

```
x /i (double*) foo + 4
```

Disassemble memory starting 4 double words past the address `foo`.

```
break *($vu1_text + 49)
```

Break at double word address 49 in VU1 text space, where

`set $vu1_text = (double *) 0xb1008000` was used to set the GDB convenience variable `$vu1_text`.

```
print *($vu0_data + 27)
```

Suppose one defines the following structure:

```
typedef struct {
   float x, y, z, w;
} vec;
```

and sets `$vu0_data = (vec *) 0xb1004000`. This command will print the 27th quadword of VU0 data memory as a 4 element floating point vector.

Note that GDB is still using byte addressing internally: the casting is used to scale the constants appropriate to the type of data in each memory section. This means that values will still be reported in bytes: `info break` will show the address as:

```
$vu1_text+49=0xb1008000+49*8=0xb1008188
```

and the disassembly listing will show the offset from `foo` in bytes (i.e. starting at offset 32 in the above example).

Examining source files

You can use the line numbers of source files to locate instructions in memory. Conversely, any memory resident instruction (excluding those allocated on the heap) can be related back to its original source line.

Instructions and data in the local memory of the VUs are dynamically loaded through the DMA and VIF units. However, the mapping between the source lines and location in VU local memory is maintained so as to be transparent to the user.

CPU contexts are used to distinguish between multiple address spaces. This affects the interpretation of the commands that examine source. GDB normally maintains the concept of a current source file and line number. With multiple contexts, this concept is extended so that there is now a current source file and line number for each context.

The detailed behavior of each command and/or argument is as follows:

`list [`*`linespec`*`]`

Prints lines from the source file relative to the current CPU context, where *`linespec`* is one of the following:

`[first][,last]`

Specifies a range of line numbers of the current source file in the current CPU context. If the CPU context is `auto,` the current source file is the last source file referenced in any context.

Either `first` or `last` or both may be omitted. If both are omitted, the last line number in the current CPU context is used.

`+|-offset`

Specifies an offset (either forward or back) from the last line printed in the current CPU context.

`file:`*`linenum`*

Specifies a line number in the source file `file`. This `file` and the *`linenum`* specified in the command becomes the current source location for the current CPU context.

For example, context `vu1` could have `foo.vuasm:23` associated with it; and context `master` could have `main.c:124` as its *`file:linenum`*. If you are in context `master` and specify `list malloc`, the new *`file:linenum`* for the `master` context would become `malloc.c:1028` (assuming that is where `malloc` is located). The *`file:linenum`* for context `vu1` would not change.

The term “current” applies to both the CPU context and the *`file:linenum.`* This is important if you specify a `list` (or break) command whose *`linespec`* is relative to a source location. If the current context is `vu1`, the `list` command will list lines starting from `foo.vuasm:23`; and if the current context is `master,` the lines will be listed from `malloc.c:1028.`

`[file:]`*`function`*

Lists lines starting at *`function.`* The `file` specifier can be used if *`function`* is multiply defined or in a different file from the current source file. This file becomes the current source location for the current CPU context.

`*address`

Specifies the line containing `address` in the program. The address is interpreted relative to the current CPU context, if it is in range (when the CPU context is `vu0` or `vu1`). If the address is out of range for the current context, it is interpreted as an address in main memory. Note that if the CPU context is `auto` or `master`, VU memory is still visible through its aliased main memory address. The source file and the line number for the corresponding address become the current source location for the current CPU context.

Stopping and continuing

Stack frames

Stack frames for assembly source files are not supported. For assembly source, GDB always assumes frame 0 (the innermost frame). The `call` command is not supported for the VUs, since `call` creates and uses a dummy stack for its execution.

Breakpoints

CPU contexts are used to set breakpoints relative to a particular processor's address space. When a breakpoint on any unit is encountered, the CPU context is set to that unit.

The breakpoint commands are:

`break [`*`file`*`:]`*`function`*

Stop on entry to the specified function. The optional `file` specifier is not required unless the function is defined in multiple files. Functions in assembler source are identified using the `.func` and `.endfunc` assembler directives. For more information, refer to “Directives” on page 42.

`break +|-offset`

Set a breakpoint `offset` lines forward or backward from the position at which execution stopped. The position is relative to the current CPU context.

`break [`*`file`*`:]`*`linenum`*

Set a breakpoint at the specified line number of *`file.`* If the *`file`* specifier is omitted, GDB uses the last source file whose text was printed in the current CPU context. If the CPU context is `auto`, the source file is the last one whose text was printed in any context. This method of setting breakpoints is applicable to both C/C++ and assembler source.

`break *address`

Set a breakpoint at *address*. The *address* is relative to the current context, so the local memories of the VUs can be specified. VU and VIF addresses can also be specified from their memory-mapped addresses in the Emotion Engine address space. This is useful when the CPU context is `auto` or `master` and a breakpoint in VU or VIF space is needed.

Note that the memory at `address` must contain the opcode part of a machine instruction. This is a consideration for VIF instructions, which can have operands that are several hundred bytes long. Undefined behavior will result unless the first word of such an instruction is specified.

`break`

If the current context is `master`, the breakpoint is set at the next instruction in the selected stack frame. If the frame is the innermost, which is always true for the VUs, the breakpoint is set at the current location (useful inside loops).

`watch` *expr*

Execution stops when *expr* is written into and its value changes. The current context is used to resolve the address or register specified by *expr.*

`info break [n]`

Show the current breakpoint settings. The address shown is normally relative to the memory space to which the breakpoint applies. However, a VU breakpoint can be set before the instruction is downloaded to the local memory space, or the local instruction could have been over-written by a subsequent VIF download.

`clear`

Delete any breakpoints at the next instruction to be executed in the selected stack frame. Since the context is set to the unit that encountered the breakpoint, this is typically not ambiguous.

`clear [`*file*`:]`*linenum*

Delete any breakpoints at the specified line number of *file.* If the *file* specifier is omitted, GDB uses the last source file whose text was printed in the current context. If the CPU context is `auto,` the source file is the last one whose text was printed in any context.

Breakpoint conditions and commands work as usual, but the expressions should be legal in the current context.

Resuming execution

The simulator always steps the entire machine one instruction at a time. As a result, it is not possible to step a single unit individually. The GDB commands that resume execution are usually specified in terms of source lines. For high-level languages, a single source line can require many machine instructions to implement; for assembly programs, each instruction is normally a separate source line. The exception is if the assembler source line is a macro that expands to multiple instructions.

For the commands that follow, the current CPU context is used to interpret the definition of a single source line.

`step [count]`

Continue the program until it has executed `count` source lines (one line, if `count` is omitted). The scope of a source line is determined by the current CPU context.

`next [count]`

Like step, but any functions encountered are executed without stopping.

`finish`

Continue until just after the function in the selected stack frame returns. For high-level source, the return value (if any) is printed. For assembly source, there is only the inner-most frame, which never returns.

`until`

Continue running until a source line past the current line, in the current stack frame, is reached. The scope of a source line is determined by the current CPU context.

`until location`

Continue running until either the specified location is reached, or the current stack frame returns. Location is any of the forms acceptable to `break,` and so the same context resolution rules apply.

Examining registers

Each of the Emotion Engine and VU registers has a unique global name. The Emotion Engine registers have the standard unprefixed MIPS names (for example, `t2` or `t2h`). The VU and VIF register names are prefixed with the CPU number (for example, `vu1_vi01` or `vif1_stat`). If the CPU context is set to `vu0, vu1, vif0,` or `vif1,` the prefix can be dropped, because the register name is assumed relative to the current context. However, the global name can still be used to refer to a register that is outside the current context.

This register context is incorporated into the following commands:

`info registers`

With no arguments, show integer and special purpose registers relative to the current CPU context. Since the VUs are primarily floating-point processors, the vector registers are also displayed. If the CPU context is `auto`, the registers for all CPUs are displayed.

This command also accepts a list of individual register names. If the context is `auto` or `master`, only global register names may be used. Otherwise, the CPU prefix on the register name is optional.

Because the VU floating point registers are vector registers, GDB permits the use of vector register names to specify that all four elements of the vector should be printed. See the description of the `printvector` command below for more details.

`info all-registers`

Same as `info registers`, except that when no arguments are specified the display includes the floating point registers, as well as the integer and control registers.

<expr>

Register values may be used in expressions by prefixing the register name with a `$` (for example, `print $vu1_vi10`). No unit prefix is necessary to specify registers in the current context. Note that vector registers are not legal expressions; *<expr>* must have a single-valued result.

`printvector reg [...reg]`

This command is similar to `info register reg` in that it prints the value of the register specified as its argument. In particular, it allows `reg` to be a vector register. As a convenience, this command has the alias `pv`.

`set printvector-order [wzyx|xyzw]`

Set the order in which the elements of a vector register are printed. Without an argument, the command resets the order to the default `wzyx` order.

The setting of this variable affects the order of all commands that print vector registers. These commands include `info registers`, `printvector`, and `sim pipe`.

The following session demonstrates how these commands may be used:

```
(gdb) info register vu1_vf00
vu1_vf00wzyx: 1 0 0 0
(gdb) set context-in-prompt on
(gdb-vu1) printvector vf04
vu1_vf04wzyx: 1 0.5 2 2
(gdb-vu1) set printvector-order xyzw
(gdb-vu1) pv 4
vu1_vf04xyzw: 2 2 0.5 1
```

COP2 registers are the VU0 status and control registers as viewed from the Emotion Engine CPU. These registers are displayed as part of the VU0 and VU1 register sets (for example, `vu0_clip` or `vu1_cmsar`) and may be accessed from GDB by their symbolic names.

DMA registers have no symbolic names. They are examined and set using their memory-mapped addresses.

Examining data

The `print` *<expr>* command works as usual, although high-level type information is not recorded for variables that are declared in assembler source files. The cast operator of GDB can be used to overcome this restriction for primitive types, and even for complex types provided they have been declared somewhere in the high-level source part of the program.

Expressions containing addresses or registers are interpreted relative to the current CPU context. If the CPU context is `vu0` or `vu1` and the address is out of the range of the local address space, the address is interpreted as a main memory address. This is true for all the memory examining commands: `print, x` (examine), and `disassem`.

Caution: When using the `display` command, remember that the expression will be interpreted relative to the CPU context that is in effect when the program stops. This context may be different than the one in which the `display` command was issued.

GDB overlay support

Support for tracking and debugging the dynamically downloaded code of the VUs is based on the overlay mechanism already present in GDB. As with the existing overlay support, GDB assumes that VU code sections are downloaded in their entirety, and that a particular section has a unique runtime address. For example, a particular function can be downloaded to either or both VUs, but must run at the same local address in each VU. Unlike the standard overlay support, the SKY extensions do not require custom linker scripts or a user provided overlay manager. Instead, the runtime addresses are obtained from the source code during assembly, and the overlay tracking is managed by the SKY simulator.

GDB provides the following commands to inspect and manage the current overlay state:

`overlay manual|auto`

GDB allows users to manually map and unmap overlays in `manual` mode. For the SKY extensions, GDB starts in `auto` mode to allow the simulator to track the overlays transparently. It is not normally necessary to change this default setting.

`overlay list`

Show the currently active (mapped) overlays and their load and runtime addresses. The overlays listed are relative to the current CPU context, or to all overlays if the context is `auto` or `master`.

`overlay map` *`<section-name>`*

`overlay unmap` *`<section-name>`*

Manually map or unmap the overlay in `section-name` to or from its runtime address. If multiple overlays share the same runtime address range, then mapping a new section implies unmapping the currently mapped section (if one is mapped). Unmapping an overlay informs GDB that `section-name` is no longer resident at its runtime address, and must be accessed from its load-time address.

Debugging with overlays

When GDB's overlay support is active, GDB's concept of a symbol's address is controlled by which overlays are mapped into which memory regions. For instance, if you `print` a variable that is in an overlay which is currently mapped (that is, located in its runtime address region) GDB will fetch the variable's memory from the runtime address. If the variable's overlay is currently not mapped, GDB will fetch it from its load-time address.

Similarly, if you disassemble a function that is in an unmapped overlay, or use a symbol's address to examine memory, GDB will fetch the memory from the symbol's load-time address range instead of the runtime range. If GDB's output contains labels that are relative to an overlay's load-time address instead of the runtime address, the labels will be distinguished like this:

```
(gdb) overlay map .ovly0

(gdb) x /x foo
  0x300000 <foo>:   0x2d7f4ffc

(gdb) overlay unmap .ovly0

(gdb) x /x foo
  0x400000 <*foo*>: 0x2d7f4ffc
```

The asterisks (*) around the label *foo* may be interpreted as meaning that this is where *foo* is presently, but not where it will be when it is in use by the target program.

The `info address` command can tell you what overlay a symbol is in, as well as where it is loaded and mapped. The `info symbol` command can list all of the symbols that are mapped to an address.

```
(gdb) info addr foo
  Symbol "foo" is a function at address 0x300000,
  -- loaded at 0x400000 in overlay section .ovly0.

(gdb) info symbol 0x300000
  foo in mapped overlay section .ovly0
  bar in unmapped overlay section .ovly1
```

Breakpoints

As long as the overlay sections are located in RAM rather than ROM, GDB can set breakpoints in them. The breakpoints work by inserting trap instructions into the load-time address region. When the overlay is mapped into the runtime region, the trap instructions are mapped along with it, and when executed, cause the target program to break out to the debugger. If the overlay regions are located in ROM, you can only set breakpoints in them after they have been mapped into the runtime region in RAM.

Debugging the VUs

The timing and positioning of instructions is determined by the programmer; therefore, debugging support is required. This support is provided through the following command:

`sim pipe [`*`unit`*`]`

Displays the contents of the pipeline for the specified `unit` (`[vu]0` or `[vu]1`). If `unit` is not specified, both pipes are displayed.

The columns of the pipeline display show:

- The instruction address
- The delay, in cycles, before the instruction completes
- The target of the instruction
- The value to be written to the target

The following is an example VU pipe listing:

```
(gdb-vu1)
126            ADDw.x VF04, VF04, VF00w       ESIN P, VF01x
(gdb-vu1) sim pipe vu1
VU1 Pipeline:
Addr  Dly Insn  Tgt       Value
 121  1 MUL  VF05.wzyx <0.000000,-0.000000,0.000000,0.000000>
 122  2 MUL  VF06.wzyx <0.000000,-0.000000,0.000000,0.000000>
 121  1 LQI  VF02.wzyx <0.000000,-0.700000,0.000000,-0.700000>
 122  2 LQI  VF03.wzyx <1.000000,1024.000000,0.000000,0.000000>
(gdb-vu1) step
127              NOP     NOP
(gdb-vu1) sim pipe vu1
VU1 Pipeline:
Addr  Dly Insn  Tgt       Value
 122  1 MUL   VF06.wzyx <0.000000,-0.000000,0.000000,0.000000>
 124  3 ADD   VF04.x    <0.000000,-0.000000,0.000000,1.000000>
 122  1 LQI   VF03.wzyx <1.000000,1024.000000,0.000000,0.000000>
 124 26 ESIN  P         0.644218
```

To further aid debugging, the VU pipelines allow non-intrusive debugging, which means that breakpoints and single-stepping do not affect the calculated results. Note that this is different from the hardware behavior, in which the results can differ after a stopped pipeline is resumed.

Debugging the VIF units

The VIFs are modeled as having infinitely long FIFO streams of instructions. The instructions in these streams are numbered by a pseudo-instruction counter that advances as each VIF instruction is executed. The FIFO contents can be examined using the following command:

`sim list-vif[0|1] [`*`range`*`]`

Prints the instructions in the FIFO of the specified VIF unit. Note that VIF and DMA instructions only are disassembled. Embedded VU code/data is indicated by a single line giving its source address and a byte count.

Instruction ranges are specified as a `start,end` pair or `count` specifier that follows a subset of the *`linespec`* syntax of the standard `list` command (see “Examining source files” on page 59). For example, the command `sim list vif1 15,25` will print the instructions corresponding to pseudo-PC values 15 through 25 of unit VIF1. Open-ended ranges can be specified by omitting either the `start` or `end` specifier. Thus, `',25'` specifies all instructions from 0 to 25, and `'25,'` specifies all instructions from 25 up to the current pseudo-PC. If no range is specified, the default is to print the last 10 instructions. This can be changed by giving a single `count` argument (no commas), which print the last `count` instructions.

The following is an example VIF FIFO debug session:

```
...
(gdb) list vuctl.dvpasm:23
18      .text
19      DMAref *, data1
20
21      .section ".dmadata", "aw"
22      .DmaData data1
23      MPG *, *
24      NOP     IADDIU VI01, VI00, 912
25      NOP     IADDIU VI02, VI00, 904
(gdb) break 23
Breakpoint 1 at 0x133b0: file vuctl.dvpasm, line 23.
(gdb) continue
Continuing
CPU context is now vif1
```

```
Breakpoint 1, data1 () at vuctl.dvpasm:23
23      MPG *, *
(gdb) sim list vif1
    0: (0x00012280)     dmaref 1,0x000133a0
    2: (0x00012288)     vifnop
    3: (0x0001228c)     vifnop
    4: (0x000133a0)     stcycl 4,4
    5: (0x000133a4)     stmask 0
    7: (0x000133ac)     stmod direct
(gdb) printf "%x, %x\n", $pc, $vif1_pc
133b0, 133b0
(gdb)
...
```

Interactive simulator options

`sim log` *`unit`*`=[on|off]`

Used to enable or disable trace file generation for the specified unit, where unit is one of: `vu0, vu1, vif0, vif1, gif, gif1, gif2, gif3` or `gs`.

The *`unit=`* specifier may be omitted, in which case trace files will either be enabled for all units, or disabled for all units.

`sim log-file` *`unit=file`*

Specifies the file for the output log of the specified unit as file, where *`unit`* is one of: `vu0, vu1, vif0, vif1, gif, gif1, gif2, gif3` or `gs`. For more information, refer to “Emotion Engine-specific command-line options” on page 82.

`sim reset`

Returns the following peripheral devices to their start-up state: `vu0, vu1, vif0, vif1, dmac, gif` and `gs`.

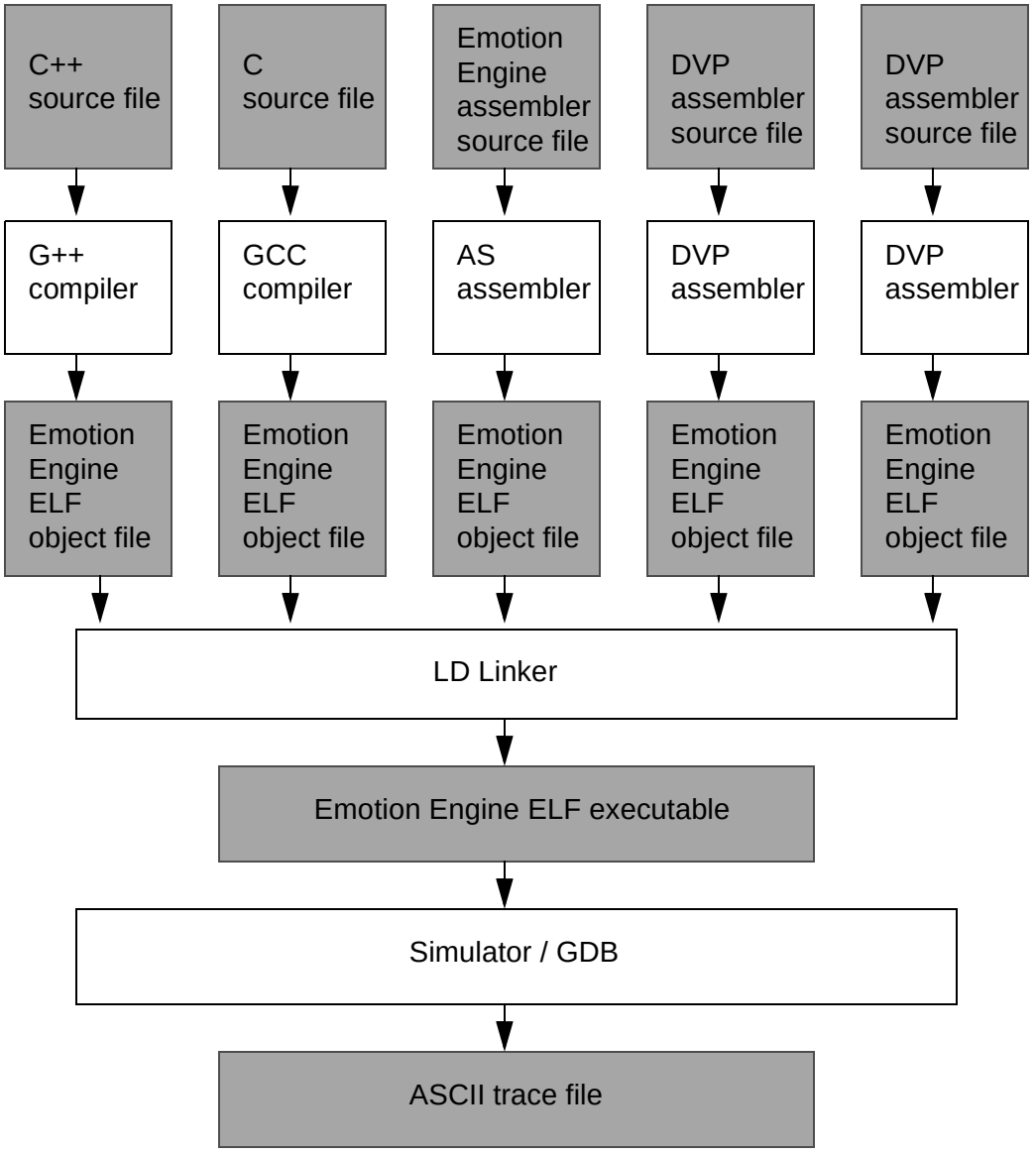
Simulator

Simulator theory of operation

This section provides an overview of how the different tools provided in the SKY simulation environment work together. It specifies the major components of the simulator and describes the behavior of the hardware elements under the simulator.

Building an executable

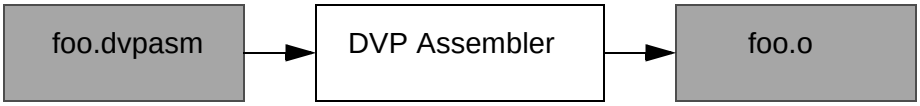
The simulation runs a single Emotion Engine executable. To produce this file, several input files are compiled with one or more of the Emotion Engine assembler, the Emotion Engine GCC and G++ compilers, and the DVP assembler.


C++ source file
C source file
Emotion Engine assembler source file
DVP assembler source file
DVP assembler source file
G++ compiler
GCC compiler
AS assembler
DVP assembler
DVP assembler
Emotion Engine ELF object file
Emotion Engine ELF object file
Emotion Engine ELF object file
Emotion Engine ELF object file
Emotion Engine ELF object file
LD Linker
Emotion Engine ELF executable
Simulator / GDB
ASCII trace file

DVP assembler

A DVP assembler is provided. For a detailed description of the assembler format, refer to “DVP Assembler” on page 41.

The following diagram displays the default output from the assembler:



Input

Assembly language source file (may contain DMA tags, GIF tags, VIF instructions and VU instructions).

Output

ELF object file that is linkable with Emotion Engine ELF objects to create a Emotion Engine ELF executable. By default, the ELF object file resides in the current working directory. You can modify the generated object file name and destination location via command line options.

The assembler has an option to ignore DMA tags and VIF instructions putting out only the VU instructions. A second option causes the assembler to ignore only DMA tags. For more information, refer to “DVP Assembler” on page 41.

You can create a binary-only version of the VU code from the object file by using the `objcopy -output-target=binary` command or the `ld -oformat binary` command.

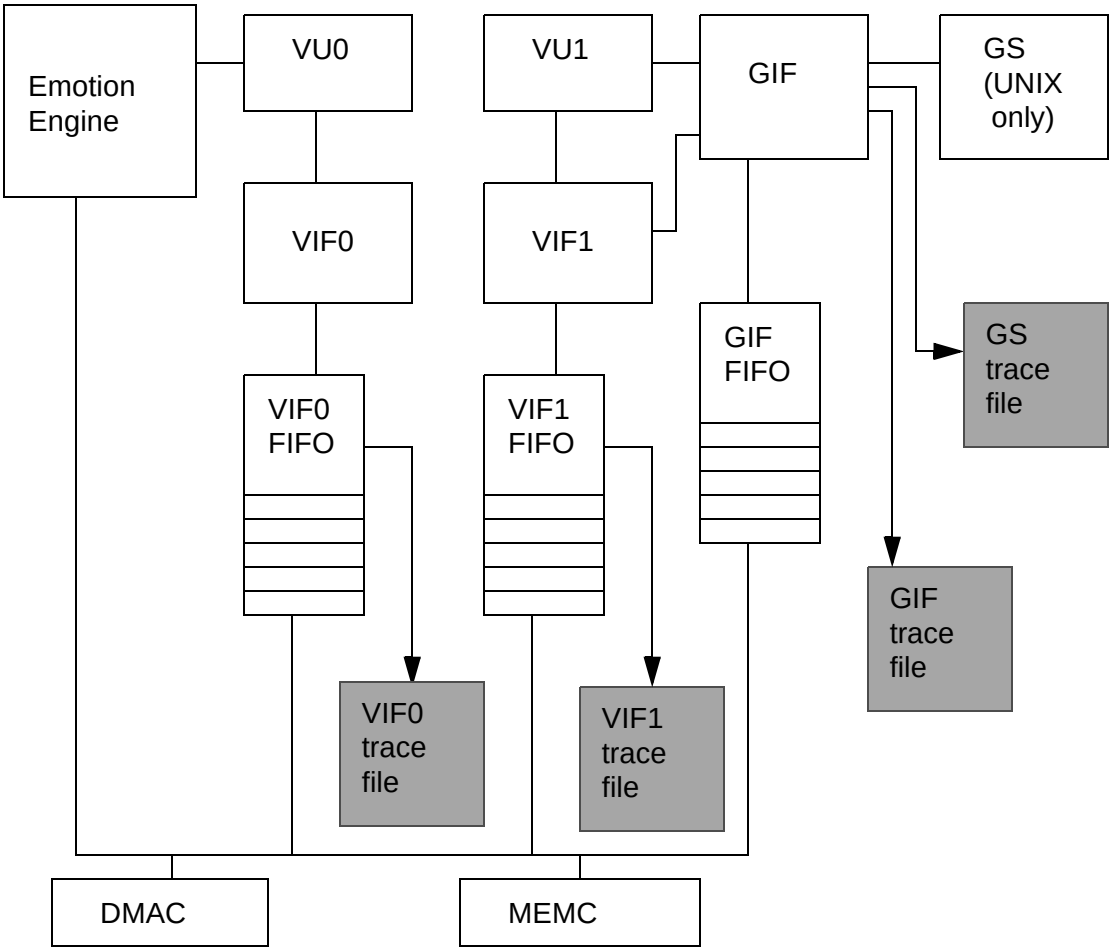
Simulator trace files

The trace files are text files that contain disassembly of the input streams to VIF0, VIF1, GIF and the GS library.

The GIF disassembly contains all GIF tags and quadwords sent to GIF on the requested path(s). The path(s) transferring the data will be identified with a comment beside the GIF tag. The GIF tags and data will appear in the sequence that the data is executed, not received. Disassembly of transfers along `path1` and `path2` are provided for information purposes only. To send `path1` and `path2` information to GIF, use the disassembly file generated by the `--log vif1=on` option. The disassembly of `path3` transfers can be assembled and sent to GIF using the `re-dma.c` sample program with the `-DGIF` option.

The GS trace file contains all input from GIF in hexadecimal form, in the following format:

```
gs-register hi-order-32-bits low-order-32-bits
(for example, 04 00109c69 827784b8)
```



You can specify whether or not to create these trace files by using command line options at simulator invocation (refer to “Emotion Engine-specific command-line options” on page 82). The default is not to trace the binary streams.

If you choose to create trace files, you must specify the streams that you want to trace, using the `--log` option. If you do not specify the streams that you want to trace, you will receive an error. The trace files will be uniquely identified and placed in the current working directory. You can modify the trace file names and destination location via command line options.

The GIF trace file contains data from 3 input paths with information regarding the path origin. You can filter out source paths from the GIF trace file.

A mechanism is provided to turn on (or off) the trace facility once the simulator has been invoked.

Subsystems A, B and C

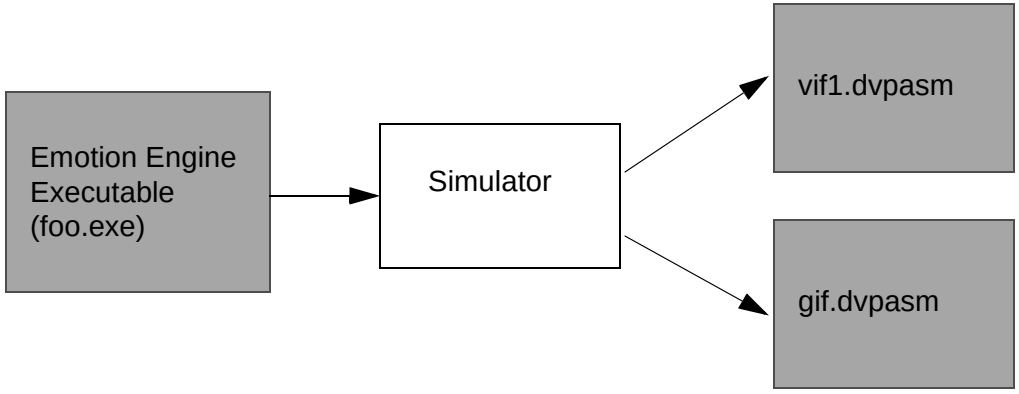
All subsystems are always present. However, each subsystem has file-like interfaces so that it can be run alone with suitable input files. Subsystem A consists of an Emotion Engine and VU0/VIF0. Subsystem B consists of VU1/VIF1 and GIF. On UNIX platforms only, there is also another subsystem, called Subsystem C, which consists of the GS library. Mechanisms are provided to run each subsystem in isolation.

Simulating Subsystem A in isolation.

The simulator itself is built as a single system. However, it allows a mode of operation that behaves as if Subsystem A is running in isolation.

The simulator provides two options to allow this mode of behavior. The first option allows the creation of trace files from VIF1 and the GIF (as described in the previous two sections). The second option causes the simulator to not execute any VIF1 instructions.

These two options together cause the simulator to behave as if Subsystem A was running in isolation.



Input

Emotion Engine ELF executable. For example, the following command:

```
ee-run --log vif1=on --log gif3=on foo.exe
```

executes `foo.exe`, producing a VIF1 and GIF Path 3 disassembly file. By default the resulting files will be called `vif1.dvpasm` and `gif.dvpasm`.

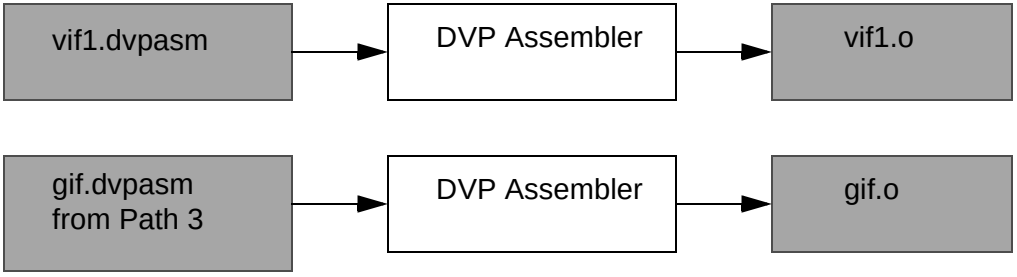
Output

Two DVP assembler files containing the VIF1 and GIF Path 3 instructions. The DVP assembler can then assemble these files into ELF object files for use as input to Subsystem B.

Simulating Subsystem “B” in isolation

The simulator allows a mode of operation that simulates Subsystem B in isolation and is performed by linking a small Emotion Engine skeleton program (provided) with DMA/VU/VIF data created by the DVP assembler. The Emotion Engine program simply executes a DMA transfer to VIF1, and then waits for VU1 to finish.

The input to the DVP assembler can be either a hand-created assembler file or it can be the output from the trace of VIF1.



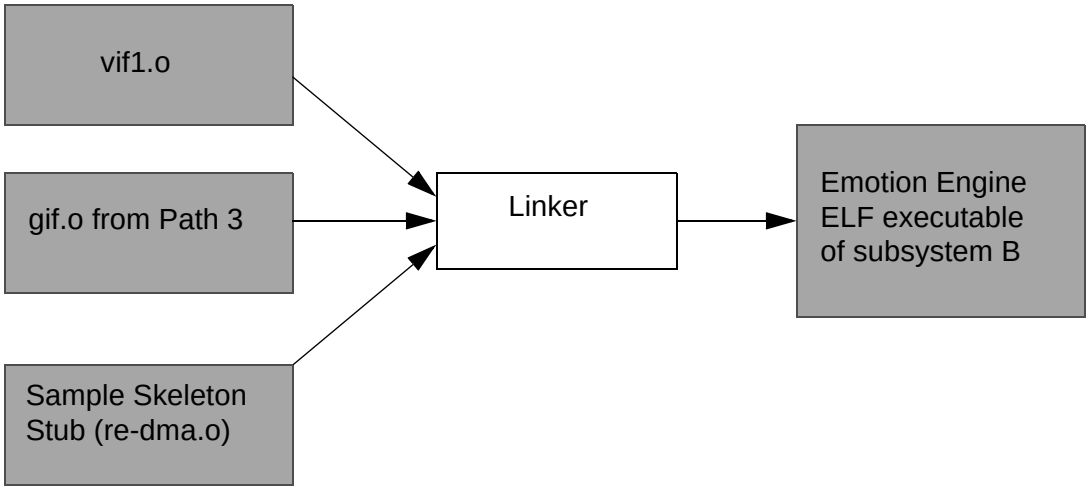
The skeleton program is a C program called `re-dma.c`. This sample program demonstrates how a reassembled log file can be transmitted back to the unit that generated the log. This process allows examination and tweaking of a data stream.

The `re-dma.c` sample code is provided as-is, without warranty of any kind. Compile with `-DVIF1` or `-DGIF`, as appropriate.

A typical compile command might look as follows:

```
ee-gcc -DGIF re-dma.c gif.o -o re-dma16.run -Tsky.ld
```

The following diagram displays the linker linking these files to create a Emotion Engine executable:



Input

ELF object files and the provided Emotion Engine skeleton stub (in ELF object form).

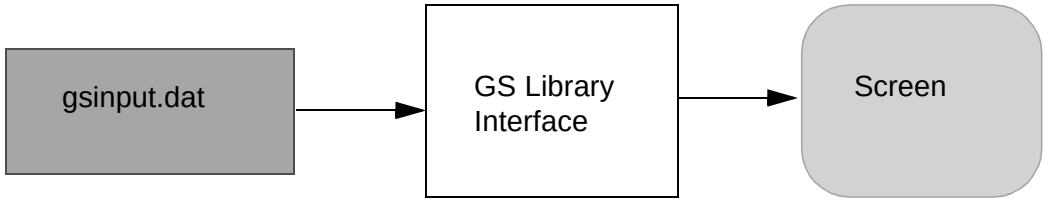
Output

An Emotion Engine ELF executable file. This file can be used to simulate the isolation of a subsystem (in the example, subsystem B) on a subsequent run of the simulator. The ELF executable file resides in the current working directory. You can modify the generated executable file name and destination location via command line options.

Simulating Subsystem C in isolation

Note that Subsystem C does not apply to Win32 platforms.

The simulator allows a mode of operation that simulates Subsystem C in isolation, and is performed by sending the GS trace input to the GS library interface.



Input

Text file containing GS registers and data from a simulator run. For example, the following:

```
ee-run --log gs=on foo.exe
```

executes `foo.exe`, producing a GS input file. By default the resulting file will be named `gsinput.dat`.

Output

Screen display image.

Hardware component level simulation

This section describes how the simulated environment differs from the actual hardware specifications.

Main Bus

- Timing, contention and arbitration of the main bus are not simulated.

DMAC

- Timing of DMA transfers is not accurately simulated. A complete chain of DMA transfers takes place between two core CPU instructions, in the clock cycle after the CORE initiates a data transfer.
- The simulator only supports DMA channels 0 - 2.
- The simulator only supports DMA transfers *from* main memory; it does not support DMA transfers *to* main memory.
- DMA stalls are not simulated. To allow this, all co-processor FIFOs are modeled as infinite length.

- DMA transfers are not sliced.
- Because of the restrictions above, some functions of the D_CTRL register are not accurately simulated:
  - `RELE`: cycle stealing does not apply to the model of DMA used.
  - `STS`, `STD`: DMA transfers do not stall.

Core Processor

- Instruction timings are not cycle accurate.
- Cache and scratch pad RAM (SPR) are not simulated.
- The COP0 Random Register changes value on a cache miss, not every cycle.
- The COP0 Count, Compare, PRId, Debug, Performance Counter, TagLo, and TagHi registers are not simulated.
- COP2 macro-instructions execute in a non-pipelined fashion. Any in-progress VU0 instructions are first driven to completion. Then the macro-instruction is fed to the VU0 pipeline, and also driven to completion. This means that sequential COP2 macro-instructions act as if the VU0 pipeline was very fast and perfectly interlocked.
- `CTC2`, `QMTC2`, `CFC2`, and `QMFC2` instructions without the interlock flag execute in a more realistic manner, because they do not drive in-progress VU0 instructions to completion.
- All COP2 instructions produce "illegal instruction" exceptions unless the CU2 (enable COP2, #30) bit is set in the Emotion Engine status register.
- Illegal COP2 instruction bit patterns are not specifically detected by the simulated Emotion Engine. Synthetic VU instructions are instead propagated to VU0, and are often detected as illegal there.

FPU

- The simulator has a runtime option (`--float-type`) that allows it to use either IEEE arithmetic or hardware-accurate arithmetic.

VIF0 and VIF1

- All VIF operations are modeled to complete in one cycle, after stall conditions are cleared.
- `VIF_NUM` is maintained; however, because operations complete in one cycle, only the extreme values may be observed by the simulated Emotion Engine.
- `VIF_STAT.PPS` accurately indicates idle/stalled state. Active state is not observed by the Emotion Engine for the same reason as above.
- `VIF_STAT.FQC` counter value is 0 or 1 for empty or non-empty state.

- The Emotion Engine can write units smaller than quadwords to the VIF FIFO addresses. However, these units are not inserted into the VIF FIFO until an entire quadword is written. For example, sixteen consecutive bytes may be written by Emotion Engine code to create one complete VIF quadword.

VU0

- The simulator has a runtime option (`--float-type`) that allows it to use either IEEE arithmetic or hardware-accurate arithmetic.

VU1

- The simulator has a runtime option (`--float-type`) that allows it to use either IEEE arithmetic or hardware-accurate arithmetic.
- The `XGKICK` instruction runs to completion in a single VU1 instruction cycle.

GIF

- To prevent stalls, the GIF FIFO is of infinite length.
- Timing of GIF transfers is not accurately simulated. Transfers of primitives complete in one cycle at the end of a primitive or IMT (image mode transfer) burst.
- Reads of GIF register addresses are simulated for the following registers only: `GIF_STAT`, `GIF_P3CNT` and `GIF_P3TAG`. Reads of non-simulated register addresses will return a zero value.

GS library

- On UNIX, local-host DMA transfers from the GS library to the main memory are not simulated.
- On UNIX, reads of GS register addresses are not simulated and will return a zero value.
- Writes to GS register addresses are simulated for the following registers only: `GS_PMODE`, `GS_DISPFB1`, `GS_DISPLAY1`, `GS_DISPFB2` and `GS_DISPLAY2`.

  Note: On Win32 systems, the call will only appear in the text file that is produced.

General

- Simulated devices are not reset upon the GDB `run` command. Simulator initialization occurs at the GDB `target sim` command only.
- The following physical address ranges are used for internal purposes in the simulator. Emotion Engine user code should not use these addresses:
  - `0x19800000 - 0x1980FFFF` [VU address tracking]
  - `0x11007000 - 0x110073FF` [VU1 registers]
  - `0x10000C00 - 0x10000FFF` [VU0 registers]

- 0x10006000 - 0x1000602F [GIF]
- 0x19810000 - 0x1981FFFF [VIF address tracking]

Registers

The SKY simulator supports both the thirty-two 128-bit general-purpose registers and the thirty-two 32-bit floating-point registers. In addition, the special registers: SA, FSR, FIR, HI and LO are also supported.

By default, the simulator allocates 16MB of unified memory at the address 0xa0000000. The 16MB block is replicated at 0x80000000 and at 0x0.

Emotion Engine-specific command-line options

The number of options available is extensive. The following is a summary of the most relevant options supported by the simulator.

--help

Outputs a full list of simulator options. The following example shows only the first few lines of output.

```
% ee-run --help
Usage: ee-run [options] program [program args]
Options:
 --list-VIF0 [range]  Show VIF0 FIFO   (interactive mode only)
 --list-VIF1 [range]  Show VIF1 FIFO   (interactive mode only)
 --pipe [[vu]0|1]     Show VU pipeline (interactive mode only)
 --pipe-order wzyx|xyzw  Vector order in VU pipeline display
 -l all|gif|gif1|gif2|gif3|gs|vif0|vif1=on|off,
 --log all|gif|gif1|gif2|gif3|gs|vif0|vif1=on|off
                          Unit disassembly/input logging
 --log-file gif|gs|vif0|vif1=FILENAME
                          Specify unit and file name for
                          disassembly/input logging
.
.
.
```

`--list-VIF0 [range]`
`--list-VIF1 [range]`
`--pipe [[vu]0|1]`
`--pipe-order wzyx|xyzw`

These options are only used in interactive mode. For more information, see "Debugging the VUs" on page 68 and "Debugging the VIF units" on page 69.

`-l [all|gif|gif1|gif2|gif3|gs|vif0|vif1=on|off]`
`--log [all|gif|gif1|gif2|gif3|gs|vif0|vif1=on|off]`

Generates a readable log of instructions, or disassembly of the input, to a device.

`all:` generate all component level logs

`gif:` generate disassembly log of all GIF paths

`gif1:` generate disassembly log of GIF PATH1 input only

`gif2:` generate disassembly log of GIF PATH2 input only

`gif3:` generate disassembly log of GIF PATH3 input only

`gs:` generate GS input data log

`vif0:` generate disassembly log of VIF0 input

`vif1:` generate disassembly log of VIF1 input

By default no log file is generated.

You can specify the log file names using the `--log-file` option.

`--log-file [gif|gs|vif0|vif1=`*`filename`*`]`

Directs component-level logged output to a specified file.

All GIF data goes to one file, even if more than one path is specified. The default file names are as follows:

`gif:` **`gif.dvpasm`**

`gs:` **`gsinput.dat`**

`vif0:` **`vif0.dvpasm`**

`vif1:` **`vif1.dvpasm`**

`--float-type fast|accurate`

Specifies whether Emotion Engine and VU floating point should be done in the host hardware or in a target-accurate software library.

`fast`: host (default)

`accurate`: target

The `accurate` mode is much slower than the `fast` mode.

`--reset`

Returns the following peripheral devices to their start-up state: `vu0, vu1, vif0, vif1, dmac, gif` and `gs`. Used in interactive mode only.

`--enable-gs [on|off]`

Alters the destination of the GIF output between the GS library and `stdout`.

`on:` GIF output goes to the GS library

`off:` GIF output goes to `stdout`.

Note: On Win32, the `--enable-gs` option is a no-op (`--enable-gs off` only).

`--gs-refresh1 <regAddress0=64bitValue:regAddress1=64bitValue>`
`--gs-refresh2 <regAddress0=64bitValue:regAddress1=64bitValue>`

User-defined GS display refresh buffers, which allow you to define the registers and values.

For example, if you wish to have the GS buffers assigned as follows:

```
GS display1 register set ( DISPFB1=0x00000000_00004000,
DISPLAY1=0x00100180_00040080 )
GS display2 register set ( DISPFB1=0x00000000_00014080,
DISPLAY1=0x00100180_00080100 )
```

the runtime option would be:

```
% ee-run --gs-refresh1
0x12000070=0x0000000000004000:0x12000080=0x0010018000040080
--gs-refresh2
0x12000070=0x0000000000014080:0x12000080=0x0010018000080100
```

Both `--gs-refresh1` and `--gs-refresh2` must be defined for the option to take effect.

`--screen-refresh [on|off]`

Inserts a call to the GS refresh buffer (`0x7f`) at the end of every primitive. The default value is `off`.

Note: The GS unit of the simulator displays the images onto the window when you write data to the address `0x7f` of GS (this is not compatible with the hardware).

If your program changes the draw buffer and the display buffer by every frame, and you use the simulator with `--screen-refresh` option, you should specify both buffers using the `--gs-refresh1,2` options. If you do not do this, the simulator may not display the image correctly.

`--load-next <exec>`

Specifies the name of the next program to be loaded. When the MIPS processor executes a `break 0xfff1` instruction the next program is loaded. The program entry point is returned in the register `a0 ($4)`.

A

Sample Code

Use the following sample code to verify that the SKY simulation environment is installed correctly. For detailed instructions, refer to “Procedures” on page 11

Sample 1: sky_main.c

```
/* sky_main.c                                */

extern int printf(const char *, ...);

extern char My_dma_start[];
extern char gpu_refresh;

/* ------------- VU defines ---------------*/
#define VPU_STAT_VBS1_MASK 0x00000100
/* ----------end of VU defines ------------*/


/* ------------- VIF defines --------------*/
#define VIF1_STAT   (volatile int *) 0xb0003C00
#define VIF1_STAT_FQC_MASK 0x1F000000
#define VIF1_STAT_PPS_MASK 0x00000003
/* ----------end of VIF defines -----------*/

/* -------------- DMA defines -------------*/
#define DMA_D0_CHCR  (volatile int*)0xb0008000
#define DMA_D0_MADR  (volatile int*)0xb0008010
#define DMA_D0_QWC   (volatile int*)0xb0008020
#define DMA_D0_TADR  (volatile int*)0xb0008030
#define DMA_D0_ASR0  (volatile int*)0xb0008040
#define DMA_D0_ASR1  (volatile int*)0xb0008050
```

```
#define DMA_D1_CHCR  (volatile int*)0xb0009000
#define DMA_D1_MADR  (volatile int*)0xb0009010
#define DMA_D1_QWC   (volatile int*)0xb0009020
#define DMA_D1_TADR  (volatile int*)0xb0009030
#define DMA_D1_ASR0  (volatile int*)0xb0009040
#define DMA_D1_ASR1  (volatile int*)0xb0009050

#define DMA_D2_CHCR  (volatile int*)0xb000a000
#define DMA_D2_MADR  (volatile int*)0xb000a010
#define DMA_D2_QWC   (volatile int*)0xb000a020
#define DMA_D2_TADR  (volatile int*)0xb000a030
#define DMA_D2_ASR0  (volatile int*)0xb000a040
#define DMA_D2_ASR1  (volatile int*)0xb000a050

#define DMA_D_CTRL   (volatile int*)0xb000e000
#define DMA_D_STAT   (volatile int*)0xb000e010

/* Dn_CHCR definition values               */
#define MODE_NORM       0
#define MODE_CHAIN      (1 << 2)
#define MODE_INTR       (2 << 2)
#define DMA_START       (1 << 8)
#define DMA_Dn_CHCR__TTE  0x00000040
#define DMA_Dn_CHCR__DIR  0x00000001
/* ----------end of DMA defines -----------*/

void DMA_enable (void)
{
  *DMA_D_CTRL = 0x01;   /* DMA enable.     */
}

/* If DMA mode is source chain.            */
void start_DMA_ch1_source_chain (void* data)
{
  *DMA_D_CTRL  = 0x01;  /* DMA enable.     */
  *DMA_D1_QWC  = 0x00;
  *DMA_D1_TADR = (int)data;
  *DMA_D1_CHCR = MODE_CHAIN | DMA_START | DMA_Dn_CHCR__TTE |
DMA_Dn_CHCR__DIR;

}

/* If DMA mode is normal.                  */
void start_DMA_ch1_normal (void* data, int qwc)
{
  *DMA_D_CTRL  = 0x01;  /* DMA enable.     */
  *DMA_D1_QWC  = qwc;   /* 8 is sample.    */
  *DMA_D1_MADR = (int)data;
  *DMA_D1_CHCR = MODE_NORM | DMA_START | DMA_Dn_CHCR__TTE |
DMA_Dn_CHCR__DIR;
}
```

```
void enable_cop2()
{
  asm ("mfc0 $3,$12; dli $2,0x40000000; or $3,$2,$2; mtc0 $3,$12");
}

int check_VPU_STAT()
{
  asm ("cfc2 $2, $29");
}

void wait_until_idle()
{
  int vif1_stat, vpu_stat;

  do
  {
    vif1_stat = *VIF1_STAT;
    vpu_stat = check_VPU_STAT();
  } while (!( (vif1_stat & VIF1_STAT_PPS_MASK) == 0
           && (vif1_stat & VIF1_STAT_FQC_MASK) == 0
           && (vpu_stat  & VPU_STAT_VBS1_MASK) == 0));
}

void wait_a_while ()
{
  int i;
  for (i=0; i<200000; i++) {}
}

int main()
{
  enable_cop2();
  start_DMA_ch1_source_chain(&My_dma_start);
  wait_until_idle();
  start_DMA_ch1_source_chain(&gpu_refresh);
  wait_a_while();

  return 0;
}
```

Sample 2: sky_test.dvpasm

```
.org 0x20000

.macro iwzyx  i1, i2, i3, i4
.int \i4, \i3, \i2, \i1
.endm

.global My_dma_start
.text
My_dma_start:
.DmaPackVif 0


DMAref *, data0

.section ".dmadata", "aw"
.DmaData data0
STCYCL 4, 4
STMASK 0x00000000
STMOD direct
.EndDmaData

.text
DMAcnt *
MPG *, *
.include "sky_test.vuasm"
.endmpg
.EndDmaData

DMAcnt *
DIRECT *
GIFpacked REGS={A_D}, NLOOP=13, EOP
iwzyx 0x00000000, 0x0000004c, 0x00000000, 0x000a0000
iwzyx 0x00000000, 0x00000040, 0x01df0000, 0x027f0000
iwzyx 0x00000000, 0x0000001a, 0x00000000, 0x00000001
iwzyx 0x00000000, 0x0000004e, 0x00000000, 0x01000096
iwzyx 0x00000000, 0x00000046, 0x00000000, 0x00000001
iwzyx 0x00000000, 0x00000047, 0x00000000, 0x00000000
iwzyx 0x00000000, 0x00000018, 0x00000000, 0x00000000
iwzyx 0x00000000, 0x00000000, 0x00000000, 0x00000006
iwzyx 0x00000000, 0x00000001, 0x00000000, 0x00000000
iwzyx 0x00000000, 0x00000004, 0x00000000, 0x00000000
iwzyx 0x00000000, 0x00000004, 0x00000000, 0x1e002800
iwzyx 0x00000000, 0x00000047, 0x00000000, 0x00070000
iwzyx 0x00000000, 0x00000018, 0x00007100, 0x00006c00
.endgif
.EndDirect
.EndDmaData
```

```
DMAcnt *
unpack V4_32, 0, *
iwzyx 0x00000000, 0xbf333333, 0x00000000, 0x3f333333
iwzyx 0x00000000, 0xbf333333, 0x00000000, 0xbf333333
iwzyx 0x3f800000, 0x44800000, 0x00000000, 0x00000000
.EndUnpack
.EndDmaData


DMAcnt *
unpack V4_32, 22, *
iwzyx 0x00000000, 0x00000000, 0x00000000, 0x3f800000
iwzyx 0x00000000, 0x00000000, 0x3f800000, 0x00000000
iwzyx 0x00000000, 0x3f800000, 0x00000000, 0x00000000
iwzyx 0x3f800000, 0x44000000, 0x00000000, 0x00000000
.EndUnpack
unpack V4_32, 26, *
iwzyx 0x00000000, 0x00000000, 0x00000000, 0x44000000
iwzyx 0x00000000, 0x00000000, 0x44000000, 0x00000000
iwzyx 0x3f800000, 0x46746000, 0x45000000, 0x45000000
iwzyx 0x00000000, 0x4e746119, 0x00000000, 0x00000000
.EndUnpack
unpack V4_32, 30, *
iwzyx 0x00000000, 0x00000000, 0x00000000, 0x00000000
iwzyx 0x00000000, 0x00000000, 0x00000000, 0x00000000
iwzyx 0x00000000, 0x00000000, 0x00000000, 0x00000000
iwzyx 0x00000000, 0x00000000, 0x00000000, 0x00000000
.EndUnpack
MSCAL 0
BASE 40
OFFSET 30
.EndDmaData

DMAcnt *
unpack[r] V4_32, 0, *
iwzyx 0x00000000, 0x00000041, 0x20064000, 0x00008008
.EndUnpack
unpack[r] V4_32, 1, *
iwzyx 0x00000000, 0x42800000, 0x437f0000, 0x437f0000
iwzyx 0x00000000, 0x437f0000, 0x42800000, 0x437f0000
iwzyx 0x00000000, 0x437f0000, 0x437f0000, 0x42800000
iwzyx 0x00000000, 0x42800000, 0x42800000, 0x437f0000
iwzyx 0x00000000, 0x42800000, 0x437f0000, 0x42800000
iwzyx 0x00000000, 0x437f0000, 0x42800000, 0x42800000
iwzyx 0x00000000, 0x437f0000, 0x437f0000, 0x437f0000
iwzyx 0x00000000, 0x42800000, 0x42800000, 0x42800000
.EndUnpack
unpack[r] V4_32, 9, *
iwzyx 0x3f800000, 0xc2c80000, 0x42c80000, 0x42c80000
iwzyx 0x3f800000, 0xc2c80000, 0x42c80000, 0xc2c80000
iwzyx 0x3f800000, 0xc2c80000, 0xc2c80000, 0x42c80000
iwzyx 0x3f800000, 0xc2c80000, 0xc2c80000, 0xc2c80000
iwzyx 0x3f800000, 0x42c80000, 0xc2c80000, 0x42c80000
```

```
iwzyx 0x3f800000, 0x42c80000, 0xc2c80000, 0xc2c80000
iwzyx 0x3f800000, 0x42c80000, 0x42c80000, 0x42c80000
iwzyx 0x3f800000, 0x42c80000, 0x42c80000, 0xc2c80000
.EndUnpack
unpack[r] V4_32, 17, *
iwzyx 0x3f800000, 0xbf13d07d, 0x3f13d07d, 0x3f13d07d
iwzyx 0x3f800000, 0xbf13d07d, 0x3f13d07d, 0xbf13d07d
iwzyx 0x3f800000, 0xbf13d07d, 0xbf13d07d, 0x3f13d07d
iwzyx 0x3f800000, 0xbf13d07d, 0xbf13d07d, 0xbf13d07d
iwzyx 0x3f800000, 0x3f13d07d, 0xbf13d07d, 0x3f13d07d
iwzyx 0x3f800000, 0x3f13d07d, 0xbf13d07d, 0xbf13d07d
iwzyx 0x3f800000, 0x3f13d07d, 0x3f13d07d, 0x3f13d07d
iwzyx 0x3f800000, 0x3f13d07d, 0x3f13d07d, 0xbf13d07d
.EndUnpack
MSCNT
unpack[r] V4_32, 0, *
iwzyx 0x00000000, 0x00000041, 0x20064000, 0x00008008
.EndUnpack
unpack[r] V4_32, 1, *
iwzyx 0x00000000, 0x42800000, 0x437f0000, 0x437f0000
iwzyx 0x00000000, 0x437f0000, 0x42800000, 0x437f0000
iwzyx 0x00000000, 0x437f0000, 0x437f0000, 0x42800000
iwzyx 0x00000000, 0x42800000, 0x42800000, 0x437f0000
iwzyx 0x00000000, 0x42800000, 0x437f0000, 0x42800000
iwzyx 0x00000000, 0x437f0000, 0x42800000, 0x42800000
iwzyx 0x00000000, 0x437f0000, 0x437f0000, 0x437f0000
iwzyx 0x00000000, 0x42800000, 0x42800000, 0x42800000
.EndUnpack
unpack[r] V4_32, 9, *
iwzyx 0x3f800000, 0xc2c80000, 0xc2c80000, 0x42c80000
iwzyx 0x3f800000, 0x42c80000, 0xc2c80000, 0x42c80000
iwzyx 0x3f800000, 0xc2c80000, 0x42c80000, 0x42c80000
iwzyx 0x3f800000, 0x42c80000, 0x42c80000, 0x42c80000
iwzyx 0x3f800000, 0xc2c80000, 0x42c80000, 0xc2c80000
iwzyx 0x3f800000, 0x42c80000, 0x42c80000, 0xc2c80000
iwzyx 0x3f800000, 0xc2c80000, 0xc2c80000, 0xc2c80000
iwzyx 0x3f800000, 0x42c80000, 0xc2c80000, 0xc2c80000
.EndUnpack
unpack[r] V4_32, 17, *
iwzyx 0x3f800000, 0xbf13d07d, 0x3f13d07d, 0x3f13d07d
iwzyx 0x3f800000, 0xbf13d07d, 0x3f13d07d, 0xbf13d07d
iwzyx 0x3f800000, 0xbf13d07d, 0xbf13d07d, 0x3f13d07d
iwzyx 0x3f800000, 0xbf13d07d, 0xbf13d07d, 0xbf13d07d
iwzyx 0x3f800000, 0x3f13d07d, 0xbf13d07d, 0x3f13d07d
iwzyx 0x3f800000, 0x3f13d07d, 0xbf13d07d, 0xbf13d07d
iwzyx 0x3f800000, 0x3f13d07d, 0x3f13d07d, 0x3f13d07d
iwzyx 0x3f800000, 0x3f13d07d, 0x3f13d07d, 0xbf13d07d
.EndUnpack
MSCNT
.EndDmaData

DMAend
```

Sample 3: sky_test.vuasm

```
vu_main: SUB.xyzw VF16, VF00, VF00      IADDIU VI01, VI00, 0
         NOP                            IADDIU VI02, VI00, 22
         NOP                            NOP
         NOP                            BAL VI15, RotMatrix
         NOP                            NOP
         NOP                            IADDIU VI01, VI00, 26
         NOP                            IADDIU VI02, VI00, 22
         NOP                            IADDIU VI03, VI00, 30
         NOP                            NOP
         NOP                            BAL VI15, MulMatrix
         NOP                            NOP
         NOP                            IADDIU VI04, VI00, 30
         NOP                            NOP
         NOP                            NOP
         NOP                            NOP
         NOP                            LQI.xyzw VF04, (VI04++)
         NOP                            LQI.xyzw VF05, (VI04++)
         NOP                            LQI.xyzw VF06, (VI04++)
         NOP                            LQI.xyzw VF07, (VI04++)
         NOP                            IADDIU VI01, VI00, 0x7fff
         NOP                            IADDIU VI03, VI00, 1
         NOP                            IADDIU VI09, VI00, 0
         NOP                            NOP
         NOP[e]                         NOP
         NOP                            NOP
LOOPE:   NOP                            IBNE VI09, VI00, CONT
         NOP                            XTOP VI05
         NOP                            IADDIU VI12, VI00, 95
CONT:    NOP                            NOP
         NOP                            NOP
         NOP                            ILW.x VI11, 0(VI05)
         NOP                            IADDIU VI08, VI05, 1
         NOP                            NOP
         NOP                            NOP
         NOP                            IAND VI11, VI11, VI01
         NOP                            NOP
         NOP                            NOP
         NOP                            NOP
         NOP                            IADD VI06, VI08, VI11
         NOP                            IADD VI02, VI11, VI00
         NOP                            NOP
         NOP                            NOP
         NOP                            LQI.xyzw VF30, (VI06++)
         NOP                            IADDIU VI07, VI12, 4
         NOP                            IADDIU VI13, VI12, 5
LOOP0:   NOP                            LQI.xyzw VF21, (VI08++)
         NOP                            IADDI VI11, VI11, -1
         NOP                            NOP
         NOP                            NOP
```

```
            FTOI0.xyzw VF22, VF21          NOP
            NOP                            NOP
            NOP                            NOP
            NOP                            NOP
            NOP                            SQ.xyzw VF22, 0(VI13)
            NOP                            IBNE VI11, VI00, LOOP0
            NOP                            IADDIU VI13, VI13, 2
            NOP                            IADD VI11, VI02, VI00
            NOP                            NOP
LOOP1:      MULw.xyzw VF29, VF31, VF00w    DIV Q, VF00w, VF31w
            MULAx.xyzw ACC, VF04, VF30x    SQ.xyzw VF27, 0(VI12)
            MADDAy.xyzw ACC, VF05, VF30y   IADDI VI11, VI11, -1
            MADDAz.xyzw ACC, VF06, VF30z   IADDI VI12, VI12, 2
            MADDw.xyzw VF31, VF07, VF30w   NOP
            FTOI4.xyzw VF27, VF28          LQI.xyzw VF30, (VI06++)
            NOP                            IBNE VI11, VI00, LOOP1
            MULq.xyzw VF28, VF29, Q        NOP
            NOP                            DIV Q, VF00w, VF31w
            NOP                            SQ.xyzw VF27, 0(VI12)
            NOP                            IADDI VI12, VI12, 2
            NOP                            LQ.xyzw VF20, 0(VI05)
            NOP                            NOP
            FTOI4.xyzw VF27, VF28          NOP
            NOP                            NOP
            MULq.xyzw VF28, VF31, Q        SQ.xyzw VF20, 0(VI07)
            NOP                            NOP
            NOP                            SQ.xyzw VF27, 0(VI12)
            NOP                            IADDI VI12, VI12, 2
            FTOI4.xyzw VF27, VF28          NOP
            NOP                            NOP
            NOP                            NOP
            NOP                            NOP
            NOP                            SQ.xyzw VF27, 0(VI12)
            NOP                            IADDI VI12, VI12, 2
            NOP                            NOP
            NOP                            NOP
            NOP                            XGKICK VI07
            NOP                            ISUB VI09, VI03, VI09
            NOP                            NOP
            NOP[e]                         NOP
            NOP                            NOP
            NOP                            B LOOPE
            NOP                            NOP
MulMatrix: NOP                             LQI.xyzw VF08, (VI02++)
            NOP                            LQI.xyzw VF04, (VI01++)
            NOP                            LQI.xyzw VF05, (VI01++)
            NOP                            LQI.xyzw VF06, (VI01++)
            NOP                            LQI.xyzw VF07, (VI01++)
            MULAx.xyzw ACC, VF04, VF08x    LQI.xyzw VF09, (VI02++)
            MADDAy.xyzw ACC, VF05, VF08y   NOP
            MADDAz.xyzw ACC, VF06, VF08z   NOP
            MADDw.xyzw VF12, VF07, VF08w   NOP
            MULAx.xyzw ACC, VF04, VF09x    LQI.xyzw VF10, (VI02++)
```

```
            MADDAy.xyzw ACC, VF05, VF09y   NOP
            MADDAz.xyzw ACC, VF06, VF09z   NOP
            MADDw.xyzw VF13, VF07, VF09w   SQI.xyzw VF12, (VI03++)
            MULAx.xyzw ACC, VF04, VF10x    LQI.xyzw VF11, (VI02++)
            MADDAy.xyzw ACC, VF05, VF10y   NOP
            MADDAz.xyzw ACC, VF06, VF10z   NOP
            MADDw.xyzw VF14, VF07, VF10w   SQI.xyzw VF13, (VI03++)
            MULAx.xyzw ACC, VF04, VF11x    NOP
            MADDAy.xyzw ACC, VF05, VF11y   NOP
            MADDAz.xyzw ACC, VF06, VF11z   NOP
            MADDw.xyzw VF15, VF07, VF11w   SQI.xyzw VF14, (VI03++)
            NOP                            NOP
            NOP                            NOP
            NOP                            NOP
            NOP                            SQI.xyzw VF15, (VI03++)
            NOP                            NOP
            NOP                            JR VI15
            NOP                            NOP
RotMatrix: MULx.xyzw VF04, VF00, VF00x  LQI.xyzw VF01, (VI01++)
            MULx.xyzw VF05, VF00, VF00x    LQI.xyzw VF02, (VI01++)
            MULx.xyzw VF06, VF00, VF00x    LQI.xyzw VF03, (VI01++)
            NOP                            LOI 1.5707963
            ADDw.x VF04, VF04, VF00w       ESIN P, VF01x
            NOP                            NOP
            NOP                            NOP
            ADDi.xyzw VF02, VF02, I        NOP
            NOP                            WAITP
            NOP                            MFP.z VF05z, P
            NOP                            MFP.y VF06y, P
            NOP                            ESIN P, VF02x
            NOP                            NOP
            NOP                            NOP
            SUB.xyzw VF06, VF16, VF06      NOP
            NOP                            WAITP
            NOP                            MFP.y VF05y, P
            NOP                            MFP.z VF06z, P
            MULx.xyzw VF07, VF00, VF00x    ESIN P, VF01y
            MULx.xyzw VF08, VF00, VF00x    NOP
            MULx.xyzw VF09, VF00, VF00x    NOP
            NOP                            NOP
            NOP                            NOP
            ADDw.y VF08, VF08, VF00w       NOP
            NOP                            WAITP
            NOP                            MFP.z VF07z, P
            NOP                            MFP.x VF09x, P
            NOP                            ESIN P, VF02y
            NOP                            NOP
            SUB.xyzw VF07, VF16, VF07      NOP
            NOP                            WAITP
            NOP                            MFP.x VF07x, P
            NOP                            MFP.z VF09z, P
            MULx.xyzw VF10, VF00, VF00x    ESIN P, VF01z
            MULx.xyzw VF11, VF00, VF00x    NOP
```

```
MULx.xyzw VF12, VF00, VF00x    NOP
MULAx.xyz ACC, VF04, VF07x     NOP
MADDAy.xyz ACC, VF05, VF07y    NOP
MADDz.xyz VF07, VF06, VF07z    NOP
ADDw.z VF12, VF12, VF00w       NOP
MULAx.xyz ACC, VF04, VF08x     NOP
MADDAy.xyz ACC, VF05, VF08y    NOP
MADDz.xyz VF08, VF06, VF08z    NOP
MULAx.xyz ACC, VF04, VF09x     NOP
MADDAy.xyz ACC, VF05, VF09y    NOP
MADDz.xyz VF09, VF06, VF09z    NOP
NOP                            WAITP
NOP                            MFP.y VF10y, P
NOP                            MFP.x VF11x, P
NOP                            ESIN P, VF02z
NOP                            NOP
NOP                            NOP
SUB.xyzw VF11, VF16, VF11      NOP
NOP                            WAITP
NOP                            MFP.x VF10x, P
NOP                            MFP.y VF11y, P
NOP                            NOP
NOP                            NOP
MULAx.xyz ACC, VF07, VF10x     NOP
MADDAy.xyz ACC, VF08, VF10y    NOP
MADDz.xyz VF10, VF09, VF10z    NOP
MULAx.xyz ACC, VF07, VF11x     NOP
MADDAy.xyz ACC, VF08, VF11y    NOP
MADDz.xyz VF11, VF09, VF11z    NOP
MULAx.xyz ACC, VF07, VF12x     NOP
MADDAy.xyz ACC, VF08, VF12y    NOP
MADDz.xyz VF12, VF09, VF12z    NOP
NOP                            NOP
NOP                            SQI.xyz VF10, (VI02++)
NOP                            SQI.xyz VF11, (VI02++)
NOP                            SQI.xyz VF12, (VI02++)
NOP                            SQI.xyz VF03, (VI02++)
NOP                            NOP
NOP                            JR VI15
NOP                            NOP
```

Sample 4: sky_refresh.s

```
.macro iwzyx i1, i2, i3, i4
.int \i4, \i3, \i2, \i1
.endm

.global gpu_refresh
DMAcnt *
direct *
GIFpacked REGS={A_D}, NLOOP=1, EOP
iwzyx 0x00000000, 0x0000007f, 0x00000000, 0x00000000
.endgif
.EndDirect
.EndDmaData
DMAend
```

B

Bibliography

EE Core Instruction Set Manual
(Version 1.00, April 1, 1999, Sony Computer Entertainment Inc.)
EE Core User's Manual
(Version 1.00, April 1, 1999, Sony Computer Entertainment Inc.)
Emotion Engine User's Manual
(Version 1.10, April 1, 1999, Sony Computer Entertainment Inc.)
Graphics Synthesizer User's Manual
(Version 1.11, April 1, 1999, Sony Computer Entertainment Inc.)
VU User's Manual
(Version 1.10, April 1, 1999, Sony Computer Entertainment Inc.)
MIPS RISC Architecture
(Kane and Heinrich, Prentice-Hall)
SCEI CPU2 Specifications Version 2.10.
(Version 2.10, August 13, 1997)
System V Application Binary Interface
(Prentice Hall, 1990)
VU Specifications
(Version 2.10, January 31, 1997 [Translated March 3, 1997])

Getting Started with GNUPro Toolkit
(Sunnyvale: Cygnus Solutions, 1998)
GNUPro Compiler Tools
(Sunnyvale: Cygnus Solutions, 1998)
GNUPro Debugging Tools
(Sunnyvale: Cygnus Solutions, 1998)
GNUPro Libraries
(Sunnyvale: Cygnus Solutions, 1998)
GNUPro Utilities
(Sunnyvale: Cygnus Solutions, 1998)
GNUPro Advanced Topics
(Sunnyvale: Cygnus Solutions, 1998)
GNUPro Tools for Embedded Systems
(Sunnyvale: Cygnus Solutions, 1998)

Index

F

G

H

I

K

L

M

N

O

P

R

S

T

V