GNUPRO™ TOOLKIT

Emotion Engine Tools User's Guide

August 30, 2000

Red Hat Inc.

Part #: 300-400-1010029

How to contact Red Hat Inc.

Use the following information for contacting Red Hat Inc.

Red Hat Inc.
2600 Meridan Parkway
Durham, NC 27713 USA
Telephone (toll free): +1 888-REDHAT-1
Telephone (main line): +1 919 547-0012

To resolve problems, please contact us via our website:

`http://support.redhat.com`

Contents

Introduction 1

Tool Naming Conventions 2

Case Sensitivity 3

Environment Summary 3

Processor versions 3

Object file format 3

Procedures 5

Create Source Code 6

Compile, Assemble and Link Sample Code 6

Assembler Listing from Sample Code 7

Reference 9

Compiler 10

Emotion Engine-specific command-line options 10

Preprocessor symbols 11

Emotion Engine-specific attributes 11

Rules for Inlining Functions 11

SIMD Built-in Functions 14

Emotion Engine ABI Summary 19

Data types and alignment 19

Data type sizes and alignments 19

Subroutine calls 20

The Stack Frame 22

Parameter assignment to registers 24

Specified registers for local variables 25

Structure passing 25
Varargs handling 25
Function return values 26
Emotion Engine Assembler **27**
MIPS-specific command-line options 27
Syntax 27
Register names 27
Assembler directives 28
MIPS synthetic instructions for the Emotion Engine 29
DVP Assembler **38**
Emotion Engine-specific command-line options 38
Syntax 39
Literals 40
Directives 40
VU opcodes 42
VIF opcodes 42
DMA tag instructions 47
GIF tag instructions 49
Linker **52**
SKY-specific command-line options 52
Linker script 52
Debugger **55**
Examining source files 55
Stopping and continuing 56
Examining registers 58
Examining the Emotion Engine SIMD Registers 59
Examining data 60

Sample Code **61**
Sample 1: sky_main.c **61**
Sample 2: sky_test.dvpasm **64**
Sample 3: sky_test.vuasm **67**
Sample 4: sky_refresh.s **72**

Bibliography **73**

Index **75**

Introduction

Welcome to Red Hat’s GNUPro Toolkit Emotion Engine Tools User’s Guide. The GNUPro Toolkit from Red Hat is a complete solution for C and C++ development for the Emotion Engine and co-processors. The tools include the compiler,interactive debugger, and utilities libraries.

Tool Naming Conventions

Cross-development tools in the Red Hat 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`

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`

Case Sensitivity

The following strings are case-sensitive:

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

The following strings are not case-sensitive:

- gdb commands
- assembler instructions and register names

Environment Summary

Processor versions

Emotion Engine and custom co-processors

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

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, so you must enter all commands and options exactly as indicated in this document.

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 61.

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 and link the program.

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 `%`.

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

```
% ee-gcc -g -c sky_main.c -o sky_main.o
% ee-dvp-as sky_test.dvpasm -o sky_test.o
% ee-dvp-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.

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
```

2

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 10.
- To learn how the Emotion Engine should interface with the operating system, refer to “Emotion Engine ABI Summary” on page 19.
- To assemble Emotion Engine source code, refer to “Emotion Engine Assembler” on page 27.
- To assemble DMA tags, GIF tags, VIF instructions, and VU instructions, refer to “DVP Assembler” on page 38.
- To link with GNU `ld`, refer to “Linker” on page 52.
- To debug with GDB, refer to “Debugger” on page 55.

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.

`-mhandle-ee-div-pipeline-bug`

This option is on by default. It ensures that the compiler will not place a `div.s`, `sqrt.s` or `rsqrt.s` instruction in the following places:

- in the delay-slot of a branch instruction
- within two instructions after the delay-slot of branch instruction, or
- within two instructions after a branch target address.

`-ftrapv`

Raise exceptions on signed overflow for addition, subtraction and multiplication operations.

`-fsingle-precision-constant`

This option tells the compiler to treat floating point constants as single precision constants instead of implicitly converting them to double precision constants.

`--target-help`

When gcc is invoked with this option, it prints a description of target-specific command line options for each tool to the standard output.

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.

Rules for Inlining Functions

Functions are categorized into three basic types:

- leaf functions
- non-recursive functions
- recursive functions.

A leaf function does not call other functions. Generally, you invoke leaf functions as normal calls.

A non-recursive function contains one or more calls to other functions. At the end of the execution of these called functions, control returns to the calling function. A call to another non-recursive function made anywhere other than at the end of the function is known as a sibling call. A call made at the end of the function is known as a tail call.

Recursive functions call themselves. Some recursive functions are tail-recursive, which means that they have certain properties that allow recursive calls to be transformed into a loop.

Leaf functions, non-recursive functions, and tail-recursive functions can be inlined.

Declaring a Function Inline

Declaring a function inline means integrating that function's code into the code for its caller. This makes execution faster by eliminating the function-call overhead. In addition, if any of the actual argument values are constant, their known values may permit simplifications at compile time so that not all of the inline function's code needs to be included.

The effect of inlining on code size is less predictable; object code may be larger or smaller with function inlining, depending on the particular case. Inlining of functions is an optimization and it really works only in optimizing compilation. To permit functions to be inlined, you must use the `-O` command line option.

To declare a function inline, use the inline keyword in its declaration, such as in the following example:

```
inline int
inc (int *a)
{
   (*a)++;
}
```

For ANSI C programs, use the `__inline__` keyword instead of the `inline` keyword.

You can also make all simple enough functions inline with the `-finline-functions` option. If compiled with `-finline-functions`, the compiler makes decisions based on the size of the function and the registers used to inline a function, even if the function is not declared explicitly.

NOTE: In C and Objective C, unlike C++, the inline keyword does not affect the linkage of the function.

GNU CC automatically inlines member functions defined within the class body of C++ programs, even if they are not explicitly declared inline. (You can override this with the `-fno-default-inline` option.)

Using Inline Functions

When a function is both inline and static, if all calls to the function are integrated into the caller and the function's address is never used, then the function's own assembler code is never referenced. In this case, GNU CC does not actually output assembler code for the function, unless you specify the `-fkeep-inline-functions` option. Some calls cannot be integrated for various reasons. In particular, calls that precede the function's definition cannot be integrated, and neither can recursive calls within the definition. If there is a nonintegrated call, then the function is compiled to assembler code as usual. The function must also be compiled as usual if the program refers to its address, because that cannot be inlined.

When an inline function is not static, the compiler must assume that there may be calls from other source files. Since a global symbol can be defined only once in any program, the function must not be defined in the other source files, so the calls therein cannot be integrated. Therefore, a non-static inline function is always compiled on its own in the usual fashion.

If you specify both inline and extern in the function definition, then the definition is used only for inlining. In no case is the function compiled on its own, not even if you refer to its address explicitly. Such an address becomes an external reference, as if you had only declared the function, and had not defined it.

This combination of inline and extern has almost the effect of a macro. The way to use it is to put a function definition in a header file with these keywords, and put another copy of the definition (lacking inline and extern) in a library file. The definition in the header file will cause most calls to the function to be inlined. If any uses of the function remain, they will refer to the single copy in the library.

You cannot inline functions that:

- are too big
- have variable arguments
- have nested functions
- have a nested function that jumps to a non-local label
- have label addresses used in initializers
- have a transparant union type parameter
- have a nonlocal goto
- have complex parameters
- contain computed jumps
- contain alloca and/or setjmp
- return variable size parameters.

Even if the function is not explicitly inlined, the compiler inlines functions under the following circumstances only if invoked with `-O3` command line option:

- if the called function has just one call.
- if the size of function body is smaller than the size of the function call.

SIMD Built-in Functions

A set of built-in functions has been implemented in order to provide access to the SIMD instruction set of the Emotion Engine. There is one built-in function for each SIMD instruction. We have provided two headers to help you work with the built-in functions. The files are called `mmintrin.h` and `mmutils.h.` Both files are located in the `<install>/lib/gcc-lib/ee/<release>/include` directory, where `<install>` is the diretory in which you installed your files, and `<release>` is the name of the release that you are using.

The `mmintrin.h` header file contains the definitions of built-in functions and the data types they operate on. These are the built-in functions that we have included:

Table 1: SIMD Built-in Functions

Function Name	**Macro**
`__v8hi  __builtin_mips5900_pabsh (__v8hi A)`	`#define _pabsh(A)`
`__v4si  __builtin_mips5900_pabsw (__v4si A)`	`#define _pabsw(A)`
`__v16qi __builtin_mips5900_paddb (__v16qi A, __v16qi B)`	`#define _paddb(A,B)`
`__v8hi  __builtin_mips5900_paddh (__v8hi A, __v8hi B)`	`#define _paddh(A,B)`
`__v16qi __builtin_mips5900_paddsb (__v16qi A, __v16qi B)`	`#define _paddsb(A,B)`
`__v8hi  __builtin_mips5900_paddsh (__v8hi A, __v8hi B)`	`#define _paddsh(A,B)`
`__v4si  __builtin_mips5900_paddsw (__v4si A, __v4si B)`	`#define _paddsw(A,B)`
`__v16qi __builtin_mips5900_paddub (__v16qi A, __v16qi B)`	`#define _paddub(A,B)`
`__v8hi  __builtin_mips5900_padduh (__v8hi A, __v8hi B)`	`#define _padduh(A,B)`
`__v4si  __builtin_mips5900_padduw (__v4si A, __v4si B)`	`#define _padduw(A,B)`
`__v4si  __builtin_mips5900_paddw (__v4si A, __v4si B)`	`#define _paddw(A,B)`
`__v8hi  __builtin_mips5900_padsbh (__v8hi A, __v8hi B)`	`#define _padsbh(A,B)`
`__v2di  __builtin_mips5900_pand (__v2di A, __v2di B)`	`#define _pand(A,B)`
`__v16qi __builtin_mips5900_pceqb (__v16qi A, __v16qi B)`	`#define _pceqb(A,B)`
`__v8hi  __builtin_mips5900_pceqh (__v8hi A, __v8hi B)`	`#define _pceqh(A,B)`
`__v4si  __builtin_mips5900_pceqw (__v4si A, __v4si B)`	`#define _pceqw(A,B)`
`__v16qi __builtin_mips5900_pcgtb (__v16qi A, __v16qi B)`	`#define _pcgtb(A,B)`
`__v8hi  __builtin_mips5900_pcgth (__v8hi A, __v8hi B)`	`#define _pcgth(A,B)`
`__v4si  __builtin_mips5900_pcgtw (__v4si A, __v4si B)`	`#define _pcgtw(A,B)`
`__v8hi  __builtin_mips5900_pcpyh (__v8hi A)`	`#define _pcpyh(A)`

Table 1: SIMD Built-in Functions

Function Name	**Macro**
`__v2di  __builtin_mips5900_pcpyld (__v2di A, __v2di B)`	`#define _pcpyld(A,B)`
`__v2di  __builtin_mips5900_pcpyud (__v2di A, __v2di B)`	`#define _pcpyud(A,B)`
`void    __builtin_mips5900_pdivbw (__v4si A, __v8hi B)`	`#define _pdivbw(A,B)`
`void    __builtin_mips5900_pdivuw (__v2di A, __v2di B)`	`#define _pdivuw(A,B)`
`void    __builtin_mips5900_pdivw (__v2di A, __v2di B)`	`#define _pdivw(A,B)`
`__v4si  __builtin_mips5900_phmadh (__v8hi A, __v8hi B)`	`#define _phmadh(A,B)`
`__v4si  __builtin_mips5900_phmsbh (__v8hi A, __v8hi B)`	`#define _phmsbh(A,B)`
`__v4si  __builtin_mips5900_pmaddh (__v8hi A, __v8hi B)`	`#define _pmaddh(A,B)`
`__v2di  __builtin_mips5900_pmadduw (__v4si A, __v4si B)`	`#define _pmadduw(A,B)`
`__v2di  __builtin_mips5900_pmaddw (__v4si A, __v4si B)`	`#define _pmaddw(A,B)`
`__v4si  __builtin_mips5900_pmsubh (__v8hi A, __v8hi B)`	`#define _pmsubh(A,B)`
`__v2di  __builtin_mips5900_pmsubw (__v4si A, __v4si B)`	`#define _pmsubw(A,B)`
`__v4si  __builtin_mips5900_pmulth (__v8hi A, __v8hi B)`	`#define _pmulth(A,B)`
`__v2di  __builtin_mips5900_pmultuw (__v4si A, __v4si B)`	`#define _pmultuw(A,B)`
`__v2di  __builtin_mips5900_pmultw (__v4si A, __v4si B)`	`#define _pmultw(A,B)`
`__v8hi  __builtin_mips5900_pexch (__v8hi A)`	`#define _pexch(A)`
`__v4si  __builtin_mips5900_pexcw (__v4si A)`	`#define _pexcw(A)`
`__v8hi  __builtin_mips5900_pexeh (__v8hi A)`	`#define _pexeh(A)`
`__v4si  __builtin_mips5900_pexew (__v4si A)`	`#define _pexew(A)`
`__v4si  __builtin_mips5900_pext5 (__v8hi A)`	`#define _pext5(A)`
`__v16qi __builtin_mips5900_pextlb (__v16qi A, __v16qi B)`	`#define _pextlb(A,B)`
`__v8hi  __builtin_mips5900_pextlh (__v8hi A, __v8hi B)`	`#define _pextlh(A,B)`
`__v4si  __builtin_mips5900_pextlw (__v4si A, __v4si B)`	`#define _pextlw(A,B)`
`__v16qi __builtin_mips5900_pextub (__v16qi A, __v16qi B)`	`#define _pextub(A,B)`
`__v8hi  __builtin_mips5900_pextuh (__v8hi A, __v8hi B)`	`#define _pextuh(A,B)`
`__v4si  __builtin_mips5900_pextuw (__v4si A, __v4si B)`	`#define _pextuw(A,B)`
`__v8hi  __builtin_mips5900_pinth (__v8hi A, __v8hi B)`	`#define _pinth(A,B)`
`__v8hi  __builtin_mips5900_pinteh (__v8hi A, __v8hi B)`	`#define _pinteh(A,B)`
`__v2si  __builtin_mips5900_plzcw (__v2si A)`	`#define _plzcw(A)`
`__v8hi  __builtin_mips5900_pmaxh (__v8hi A, __v8hi B)`	`#define _pmaxh(A,B)`
`__v4si  __builtin_mips5900_pmaxw (__v4si A, __v4si B)`	`#define _pmaxw(A,B)`
`__v8hi  __builtin_mips5900_pminh (__v8hi A, __v8hi B)`	`#define _pminh(A,B)`
`__v4si  __builtin_mips5900_pminw (__v4si A, __v4si B)`	`#define _pminw(A,B)`
`__v2di  __builtin_mips5900_pnor (__v2di A, __v2di B)`	`#define _pnor(A,B)`
`__v2di  __builtin_mips5900_por (__v2di A, __v2di B)`	`#define _por(A,B)`

Table 1: SIMD Built-in Functions

Function Name	**Macro**
`__v8hi  __builtin_mips5900_ppac5 (__v4si A)`	`#define _ppac5(A)`
`__v16qi __builtin_mips5900_ppacb (__v16qi A, __v16qi B)`	`#define _ppacb(A,B)`
`__v8hi  __builtin_mips5900_ppach (__v8hi A, __v8hi B)`	`#define _ppach(A,B)`
`__v4si  __builtin_mips5900_ppacw (__v4si A, __v4si B)`	`#define _ppacw(A,B)`
`__v8hi  __builtin_mips5900_prevh (__v8hi A)`	`#define _prevh(A)`
`__v4si  __builtin_mips5900_prot3w (__v4si A)`	`#define _prot3w(A)`
`__v8hi  __builtin_mips5900_psllh (__v8hi A, int B)`	`#define _psllh(A,B)`
`__v2di  __builtin_mips5900_psllvw (__v2di A, __v2di B)`	`#define _psllvw(A,B)`
`__v4si  __builtin_mips5900_psllw (__v4si A, int B)`	`#define _psllw(A,B)`
`__v8hi  __builtin_mips5900_psrah (__v8hi A, int B)`	`#define _psrah(A,B)`
`__v2di  __builtin_mips5900_psravw (__v2di A, __v2di B)`	`#define _psravw(A,B)`
`__v4si  __builtin_mips5900_psraw (__v4si A, int B)`	`#define _psraw(A,B)`
`__v8hi  __builtin_mips5900_psrlh (__v8hi A, int B)`	`#define _psrlh(A,B)`
`__v2di  __builtin_mips5900_psrlvw (__v2di A, __v2di B)`	`#define _psrlvw(A,B)`
`__v4si  __builtin_mips5900_psrlw (__v4si A, int B)`	`#define _psrlw(A,B)`
`__v16qi __builtin_mips5900_psubb (__v16qi A, __v16qi B)`	`#define _psubb(A,B)`
`__v8hi  __builtin_mips5900_psubh (__v8hi A, __v8hi B)`	`#define _psubh(A,B)`
`__v16qi __builtin_mips5900_psubsb (__v16qi A, __v16qi B)`	`#define _psubsb(A,B)`
`__v8hi  __builtin_mips5900_psubsh (__v8hi A, __v8hi B)`	`#define _psubsh(A,B)`
`__v4si  __builtin_mips5900_psubsw (__v4si A, __v4si B)`	`#define _psubsw(A,B)`
`__v16qi __builtin_mips5900_psubub (__v16qi A, __v16qi B)`	`#define _psubub(A,B)`
`__v8hi  __builtin_mips5900_psubuh (__v8hi A, __v8hi B)`	`#define _psubuh(A,B)`
`__v4si  __builtin_mips5900_psubuw (__v4si A, __v4si B)`	`#define _psubuw(A,B)`
`__v4si  __builtin_mips5900_psubw (__v4si A, __v4si B)`	`#define _psubw(A,B)`
`__v2di  __builtin_mips5900_pxor (__v2di A, __v2di B)`	`#define _pxor(A,B)`
`void    __builtin_mips5900_mtsab (int A, int B)`	`#define _mtsab(A,B)`
`void    __builtin_mips5900_mtsah (int A, int B)`	`#define _mtsah(A,B)`
`void    __builtin_mips5900_mtsa (__m64 A)`	`#define _mtsa(A)`
`__m64   __builtin_mips5900_mfsa (void)`	`#define _mfsa()`
`__m128  __builtin_mips5900_qfsrv (__m128 A, __m128 B)`	`#define _qfsrv(A,B)`
`__v2di  __builtin_mips5900_pmfhi (void)`	`#define _pmfhi()`
`__v2di  __builtin_mips5900_pmflo (void)`	`#define _pmflo()`
`__v4si  __builtin_mips5900_pmfhl_lw (void)`	`#define _pmfhl_lw()`
`__v4si  __builtin_mips5900_pmfhl_uw (void)`	`#define _pmfhl_uw()`
`__v2di  __builtin_mips5900_pmfhl_slw (void)`	`#define _pmfhl_slw()`

Table 1: SIMD Built-in Functions

Function Name	**Macro**
`__v8hi  __builtin_mips5900_pmfhl_lh (void)`	`#define _pmfhl_lh()`
`__v8hi  __builtin_mips5900_pmfhl_sh (void)`	`#define _pmfhl_sh()`
`void    __builtin_mips5900_pmthi (__v2di A)`	`#define _pmthi(A)`
`void    __builtin_mips5900_pmtlo (__v2di A)`	`#define _pmtlo(A)`
`void    __builtin_mips5900_pmthl_lw (__v4si A)`	`#define _pmthl_lw(A)`

SIMD Data Types

There are also new data types that represent the vectors used by the built-in functions. The `mmutils.h` header file contains functions and macros designed to help you manipulate the SIMD data types themselves. This includes functions and macros for setting and obtaining the values of vectors and vector elements, and for printing the values of vectors.

Table 2: SIMD data types

Data Type	***Value***
`__m128`	a 128 bit integer
`__m64`	a 64 bit integer
`__v2di`	two 64-bit ("double") integers
`__v2si`	two 32-bit ("single") integers
`__v4si`	four 32-bit ("single") integers
`__v8hi`	eight 32-bit ("half") integers
`__v16qi`	sixteen 32-bit ("quarter") integers

Example of a Program Using SIMD Data Types

The following is an example of a program that uses SIMD Data Types:

```
#include "mmutils.h"
#include <stdio.h>
#include <stdlib.h>

static void
pass (void)
{
  puts ("pass");
}

static void
fail (void)
{
  puts ("fail");
  exit(1);
}

static void
compare (__v4si result, __v4si expected)
{
  int i;
  for (i = 0; i < 4; ++i)
    {
      if (_MM_INT ((__m128)result, i) != _MM_INT ((__m128)expected, i))
      fail ();
    }
}


main () {
  __v8hi vec;
  __v4si expected, result;

  vec = (__v8hi)_m_set_v8hi (0xdead, 0x0000, 0xbeef, 0x8000,
                             0xdead, 0x7c1f, 0xbeef, 0x83e0);
  expected = (__v4si)_m_set_v4si (0x00000000, 0x80000000,
                                  0x00f800f8, 0x8000f800);

  result = _pext5 (vec);
#if DEBUG
  _m_print_v8hi (vec);
  _m_print_v4si (result);
  _m_print_v4si (expected);
#endif
  compare (result, expected);

  pass ();
  return 0;
}
```

Emotion Engine ABI Summary

Data types and alignment

This section describes the MIPS EABI, which the Emotion Engine tools generally adhere to, with some exceptions:

- MIPS pointers are normally 8 bytes, but Emotion Engine pointers are 4 bytes.
- Unlike most MIPS processors, the Emotion Engine passes parameters that have the type double via a general-purpose register, rather than via a floating-point register or register pair.
- Some MIPS processors have both 32-bit mode and 64-bit mode; however, the Emotion Engine only has 64-bit mode.

Data type sizes and alignments

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

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. The third table describes floating-point register usage.

Table 3: Parameter Registers

Registers	***Use***
`r4-r11`	general-purpose
`f12-f19`	floating-point

Table 4: Register Usage

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

- 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.

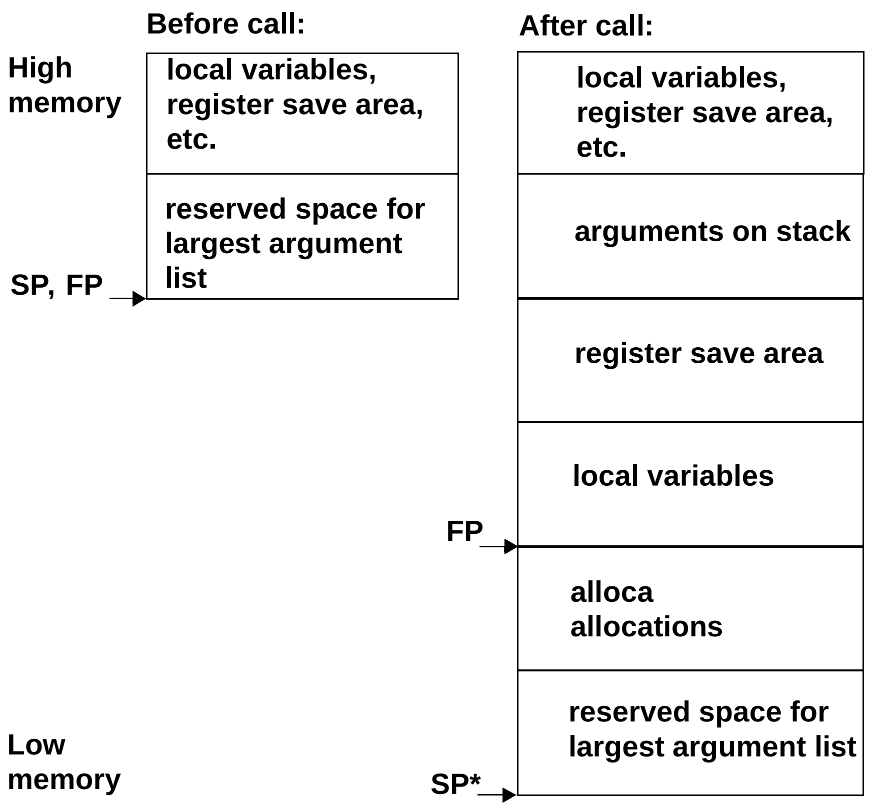
Table 5: Floating-point Register Usage

Register Name:	***Use***
`$f0..$f23`	Temporary floating point values. Their contents are not preserved across function calls. `$f12..$f19` are used to pass floating point arguments. If a function returns a single precision floating point value it is passed in `$f0`.
`$f24..$f31`	Saved floating point values. Their contents are preserved across function calls.
`fcr31`	Control/status register. Contains control and status data for FP operations, including the enabling of FP exceptions. Also indicates floating-point exceptions that have occurred since the register was cleared. This register is read/write.

The Stack Frame

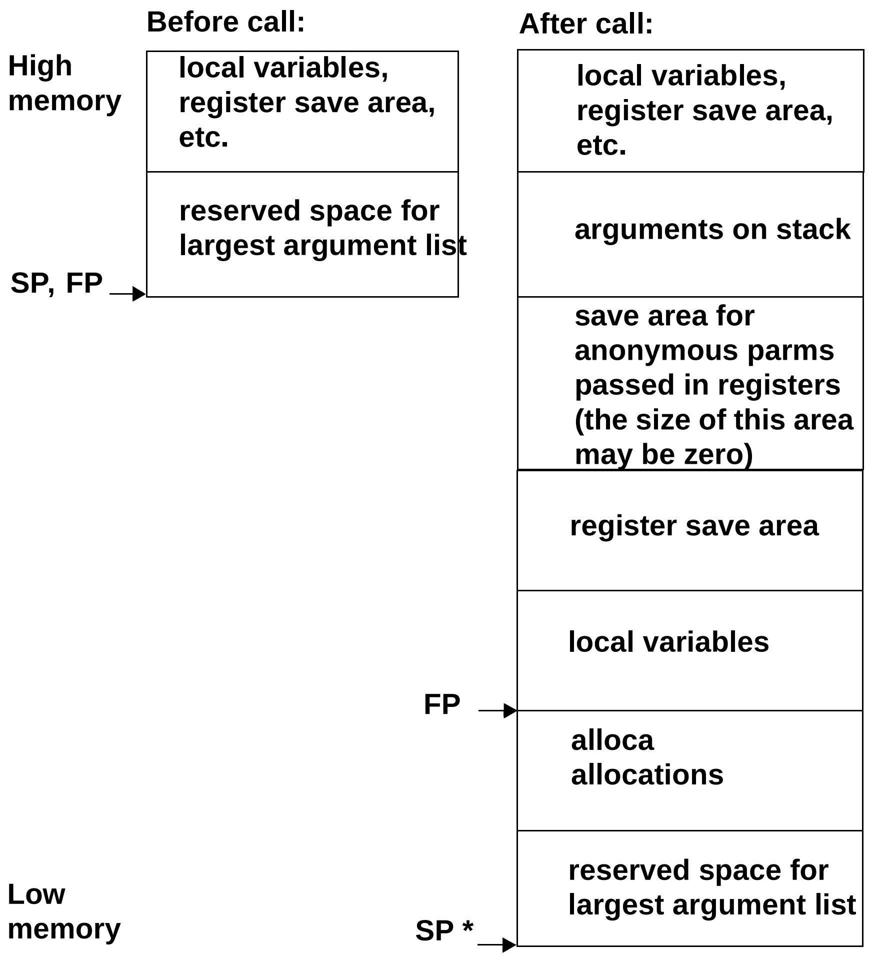
- The stack grows downwards from high addresses to low addresses.
- A leaf function need not allocate a stack frame if it does not need one.
- A frame pointer (FP) need not be allocated.
- The stack pointer (SP) shall always be aligned to 16-byte boundaries.

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


Before call:
After call:
High memory
local variables, register save area, etc.
reserved space for largest argument list
SP, FP
local variables, register save area, etc.
arguments on stack
register save area
local variables
FP
alloca allocations
reserved space for largest argument list
Low memory
SP*

*** 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:


Before call:
After call:
High memory
local variables, register save area, etc.
reserved space for largest argument list
SP, FP
local variables, register save area, etc.
arguments on stack
save area for anonymous parms passed in registers (the size of this area may be zero)
register save area
local variables
FP
alloca allocations
reserved space for largest argument list
Low memory
SP *

*** 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. 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:

FLOAT:

If FR > f19, go to STACK. Otherwise, load the parameter value into floating-point purpose register FR and advance FR to the next floating-point purpose register, then go to SCAN.

DOUBLE:

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.

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. Floats are stored in the low-order 32 bits of the 64-bit space allocated to them. Type long long is 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 6: 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 7: Function return values

Type	**Register**
`int`	`r2`
`short`	`r2`
`long`	`r2`
`long long`	`r2`
`float`	`f0`
`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 8: 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 9: 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 10: 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`
`b      I1`	`beq    $zero,$zero,I1`
`bal    I1`	`bgezal $zero,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 10: 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`

`bltl   R1,R2,I1`	`slt    $at,R1,R2`
	`bnezl  $at,+I1`

Table 10: MIPS Synthetic Instructions

Instruction	***Expansion***

`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`

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

Table 10: MIPS Synthetic Instructions

Instruction	***Expansion***
`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`
`dneg    R1,R2`	`dsub   R1,$zero,R2`
`dnegu   R1,R2`	`dsubu  R1,$zero,R2`
`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`
`drem     R1,R2,I1`	`li     $at,I1`
	`ddiv   $zero,R2,$at`
	`mfhi   R1`

Table 10: MIPS Synthetic Instructions

Instruction	*Expansion*

`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`

`dsll   R1,R2,R3`	`dsllv  R1,R2,R3`
`dsll   R1,R2,I1 # I1 > 31`	`dsll32 R1,R2,I1-32`
`dsra   R1,R2,R3`	`dsrav  R1,R2,R3`
`dsra   R1,R2,I1 # I1 > 31`	`dsra32 R1,R2,I1-32`
`dsrl   R1,R2,R3`	`dsrlv  R1,R2,R3`
`dsrl   R1,R2,I1 # I1 > 31`	`dsrl32 R1,R2,I1-32`
`dsub   R1,R2,I1`	`daddi  R1,R2,-I1`
`dsubu  R1,R2,I1`	`daddiu R1,R2,-I1`
`flush  R1,I1(R2)`	`lwr    R1,I1(R2)`
`j      R1`	`jr     R1`
`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     R1,I1`	`lui    R1,%hi(I1)`
	`ori    R1,%lo(I1)`

`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)`

Table 10: MIPS Synthetic Instructions

Instruction	*Expansion*
`move   R1 R2`	`daddu  R1,R2,0`

`mul    R1,R2,R3`	`multu   R2,R3`
	`mflo    R1`

`mul    R1,R2,I1`	`li      $at,I1`
	`multu   R2,$at`
	`mflo    R1`

`mulo   R1,R2,R3`	`  mult    R2,R3`
	`  mflo    R1`
	`  sra     R1,R1,0x1f`
	`  mfhi    $at`
	`  beq     R1,$at,L1_mulo`
	`  nop`
	`  break   0x6`
	`L1_mulo:`
	`  mflo    R1`

`mulo   R1,R2,I1`	`  li      $at,I1`
	`  mult    R2,$at`
	`  mflo    R1`
	`  sra     R1,R1,0x1f`
	`  mfhi    $at`
	`  beq     R1,$at,L1_mulo`
	`  nop`
	`  break   0x6`
	`L1_mulo`
	`  mflo    R1`

`mulou  R1,R2,R3`	`  multu   R2,R3`
	`  mfhi    $at`
	`  mflo    R1`
	`  beqz    $at, L1_mulou`
	`  nop`
	`  break   0x6`
	`L1_mulou:`

`mulou  R1,R2,I1`	`  li      $at,I1`
	`  multu   R2,$at`
	`  mfhi    $at`
	`  mflo    R1`
	`  beqz    $at, L1_mulou`
	`  nop`
	`  break   0x6`
	`L1_mulou:`

`neg    R1,R2`	`sub    R1,$zero,R2`
`negu   R1,R2`	`subu   R1,$zero,R2`

`nor    R1,R2,I1`	`ori    R1,R2,I1`
	`nor    R1,R1,$zero`

`not    R1,R2`	`nor    R1,R2,$zero`
`or     R1,R2,I1`	`ori    R1,R2,I1`

Table 10: 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
seq	R1,R2,I1	xori R1,R2,I1 sltiu R1,R1,1
sge	R1,R2,R3	slt R1,R2,R3 xori R1,R1,0x1

Table 10: MIPS Synthetic Instructions

Instruction	*Expansion*

`sge    R1,R2,I1`	`slti   R1,R2,I1`
	`xori   R1,R1,0x1`

`sgeu   R1,R2,R3`	`sltu   R1,R2,R3`
	`xori   R1,R1,0x1`

`sgeu   R1,R2,I1`	`sltiu  R1,R2,I1`
	`xori   R1,R1,0x1`

`sgt    R1,R2,R3`	`slt    R1,R3,R2`

`sgt    R1,R2,I1`	`li     $at,I1`
	`slt    R1,$at,R2`

`sgtu   R1,R2,R3`	`sltu   R1,R3,R2`

`sgtu   R1,R2,I1`	`li     $at,I1`
	`sltu   R1,$at,R2`

`sle    R1,R2,R3`	`slt    R1,R3,R2`
	`xori   R1,R1,0x1`

`sle    R1,R2,I1`	`li     $at,I1`
	`slt    R1,$at,R2`
	`xori   R1,R1,0x1`

`sleu   R1,R2,R3`	`sltu   R1,R3,R2`
	`xori   R1,R1,0x1`

`sleu   R1,R2,I1`	`li     $at,I1`
	`sltu   R1,$at,R2`
	`xori   R1,R1,0x1`

`sll    R1,R2,R3`	`sllv   R1,R2,R3`
`slt    R1,R2,I1`	`slti   R1,R2,I1`
`sltu   R1,R2,I1`	`sltiu  R1,R2,I1`

`sne    R1,R2,R3`	`xor    R1,R2,R3`
	`sltu   R1,$zero,R1`

`sne    R1,R2,I1`	`xori   R1,R2,I1`
	`sltu   R1,$zero,R1`

`sra    R1,R2,R3`	`srav   R1,R2,R3`
`srl    R1,R2,R3`	`srlv   R1,R2,R3`
`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)`
`scache R1,I1(R2)`	`swl    R1,I1(R2)`
`invalidate R1,I1(R2)`	`swr    R1,I1(R2)`
`teq    R1,I1`	`teqi   R1,I1`

Table 10: MIPS Synthetic Instructions

Instruction	*Expansion*
`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`

Use if you do not want to include DMA instructions in the output.

`-no-dma-vif`

Use if you do not want to include DMA or VIF instructions in the output.

`-stalls-pipeline`

Use if you want the DVP Assembler to issue a warning message if the execution of an instruction stalls the pipeline. One of the following reason will be provided:

DIV resource hazards

Occurs when a DIV/SQRT/RSQRT instruction is being executed and another instruction of similar type is executed concurrently. For instance if a DIV instruction and an SQRT instructioned were executed concurrently, the DVP Assembler would issue this warning.

EFU resource hazards

Occurs when a calculation instruction that uses the EFU unit is executed, and the next instruction of this type is executed concurrently.

floating point register data hazards

Occurs when an instruction that uses a floating point register as the destination is executed, and the next instruction—which uses the same floating point register—is executed.

Data hazard checks are perform independently in each field of `x/y/z/w`. VF00 is a constant register and is not subject to Data hazards.

integer register data hazards

Occurs if a `load/store` instruction that uses an integer register is executed and the next instruction that uses the same integer register is executed. VI00 is not subject to integer register data hazards.

For further references, see the ***VU User's Manual, Section 3.4.***

`-no-fetching`

The DVP Assembler will issue a warning if an upper instruction fetches a value that is not the value that the corresponding lower instruction set to I register.

`-no-interdependency`

If the destination register of the instruction executed before an SQ instruction is a floating point register, there should not be any interdependency between the instruction and the SQ instruction. However, if the number of the registers are the same, the hardware thinks that there is interdependency. This option issues a warning message to indicate that the hardware thinks there is an interdependency when there actually is none.

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 11: 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 User's Manual.***

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
```

You can use `LOI` pseudo instruction as lower instruction to load floating point immediate value into the I register.

For example:

```
NOP         LOI 3.141592
```

This pseudo instruction set the `I` bit of a corresponding upper instruction field and put the immediate value in a lower instruction field.

VIF opcodes

For detailed information on the VIF instruction set, see ***EE User's Manual.*** (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).

Notes:

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.

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.

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.

`[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.

`*address`

Specifies the line containing `address` in the program.

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).

Breakpoints

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 40.

`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. This method of setting breakpoints is applicable to both C/C++ and assembler source.

`break *`*`address`*

Set a breakpoint at *address*.

`break`

The breakpoint is set at the next instruction in the selected stack frame. If the frame is the innermost, the breakpoint is set at the current location (useful inside loops).

`watch `*`expr`*

Execution stops when *expr* is written into and its value changes.

`info break [n]`

Show the current breakpoint settings.

`clear`

Delete any breakpoints at the next instruction to be executed in the selected stack frame.

`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.

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

Resuming execution

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.

`step [count]`

Continue the program until it has executed `count` source lines (one line, if `count` is omitted).

`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.

`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 register names are prefixed with the CPU number(for example, `vu1_vi01`).

`info registers`

With no arguments, show integer and special purpose registers. Since the VUs are primarily floating-point processors, the vector registers are also displayed.

This command also accepts a list of individual register names.

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`). 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) printvector vf04
vu1_vf04wzyx: 1 0.5 2 2
(gdb) set printvector-order xyzw
(gdb) 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 the Emotion Engine SIMD Registers

There are some differences between registers that are used in the ordinary way, and registers that are used to hold SIMD values.

When debugging ordinary MIPS assembly code, you might use the following to print the value of register `$a0` as an integer:

```
(gdb) print $a0
$1 = 9
```

However, if you are debugging SIMD code that uses `$a0` to hold a vector value, you can add a ‘`v`’ to end of the register name to see its value as a vector.

For example:

```
(gdb) print $a0v
$2 = {v2di = {-1, 9},
      v2si = {-1, -1},
      v4si = {-1, -1, 9, 0},
      v8hi = {-1, -1, -1, -1, 9, 0, 0, 0},
      v16qi = {-1, -1, -1, -1, -1, -1, -1, -1, 9, 0, 0, 0, 0, 0, 0, 0}}
```

This shows the contents of `$a0` as a vector register in all the SIMD modes supported by the Emotion Engine: a pair of 64-bit integers, four 32-bit integers, and so on.

If you know that `$a0` is being used to hold four 32-bit values, then you can specify the mode explicitly, by selecting a particular element from `$a0v`'s value:

```
(gdb) print $a0v.v4si
$3 = {-1, -1, 9, 0}
```

This provides a less cluttered display. This value is an ordinary array, so you can subscript it to refer to a particular element:

```
(gdb) print $a0v.v4si[2]
$4 = 9
```

You can assign to components of vector registers in the usual way:

```
(gdb) set variable $a0v.v4si[2] = 10
(gdb) print $a0v.v4si
$5 = {-1, -1, 10, 0}
```

If the register holds a vector of unsigned values, you can use the print command with the ‘`/u`’ format:

```
(gdb) print/u $a0v.v4si
$6 = {4294967295, 4294967295, 10, 0}
```

In general, for every general-purpose Emotion Engine register `$X,` GDB now has a new *pseudoregister* called `$Xv,` whose value is a union containing five members, one for each SIMD mode supported by the Emotion Engine:

Table 12: Pseudoregisters

Pseudoregister	***Value***
`v2di`	the register's contents as two 64-bit ("double") integers
`v2si`	the register's contents as two 32-bit ("single") integers
`v4si`	the register's contents as four 32-bit ("single") integers
`v8hi`	the register's contents as eight 32-bit ("half") integers
`v16qi`	the register's contents as sixteen 32-bit ("quarter") integers

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.

A

Sample Code

Use the following sample code to verify that the SKY environment is installed correctly. For detailed instructions, refer to “Procedures” on page 5.

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 | 1;

}

/* 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 | 1;
}

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 2.0, October, 1999, Sony Computer Entertainment Inc.)
EE Core User's Manual
(Version 2.0, October, 1999, Sony Computer Entertainment Inc.)
EE User's Manual
(Version 2.0, October, 1999, Sony Computer Entertainment Inc.)
GS User's Manual
(Version 2.0, October, 1999, Sony Computer Entertainment Inc.)
VU User's Manual
(Version 2.0, October, 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)
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

Symbols

__ee__ 11
__mips__ 11
__mips_single_float 11
__mips_soft_float 11
__MIPSEL__ 11

A

assembler listing 7
assembler mode 39

B

breakpoint commands 56
built-in functions
  SIMD 14

C

case sensitivity 3, 5
comments 39
compiling 6, 10
CPU context 55

D

debugging 55
declaring a function inline 12
directives 40
  .DmaData 41
  .DmaPackVif 41
  .EndDmaData 41
  .endfunc 41
  .EndGif 41
  .Endirect 41
  .endm 41
  .EndMpg 41
  .EndUnpack 41
  .func 41
  .quad 41
  .vu 41
  .word 41
DMA tag instructions 47
  DMAcall 47
  DMAcnt 47
  DMAend 47
  DMAnext 47
  DMAref 47
  DMArefe 47
  DMArefs 47
  DMAret 47
DVP assembler 38

E

-EL option 10, 27
ELF object file 3, 38
Emotion Engine
  assembler 27
  register names 27

examining registers 58

F

floating-point registers 21
-fno-edge-lcm 10
-fno-edge-lm 10
-fno-edge-sm 10
-fsingle-precision-constant 11
-ftrapv 11
function calls 24
function return values 26
functions
  rules for inlining 11

G

-G num 10
GAS 2
GCC 2
GDB 2
GDB commands
  break 56
  clear 57
  finish 57
  info all-registers 58
  info break 56
  info registers 58
  list 55
  next 57
  printvector reg 58
  set printvector-order 58
  step 57
  until 57
  until location 57
  watch 56
GIFimage 51
GIFpacked 49
GIFreglist 50
GIFtag instructions 49
GNU CC 10
--gstabs 41

H

-handle-ee-div-pipeline-bug 11
hazard warnings 38

I

inlining function
  rules 11

K

key symbols 39

L

labels 39
ld 52
linking 6, 52
literals 40

M

macros 29
-mdouble-float 10
-mhard-float 10
MIPS EABI
  exceptions to 19
MIPS Emotion Engine 19, 27
mmintri.h 14
mmutils.h 14
-msingle-float 10
-msoft-float 10

N

naming conventions 2
newline character 39
-no-dma 38
-no-dma-vif 38
-no-fetching 39
-no-interdependency 39

O

opcodes 39
  VIF 42
  VU 42
operands 29

P

parameter registers 20
preprocessor symbols 11

R

registers
  SIMD 59

S

setting breakpoints 56
SIMD
  built-in functions 14
  data types 17
  modes 59
  registers 59
-stalls-pipeline 38
statements 39
stopping the debugger 56
structs 25
structure passing 25
subroutine calls 20
synthetic instructions 29

T

--target-help 11

V

varargs 25
viewing registers 59
VIF
  opcodes 42
VIF opcodes
  base 42
  direct 43
  directhl 43
  flush 43
  flusha 43
  flushe 43
  itop 43
  mark 44
  mpg 44
  mscal 44
  mscalf 45
  mscnt 45
  mskpath3 45
  offset 45
  stcol 45
  stcycl 45
  stmask 45
  stmod 45
  strow 45
  unpack 46
  vifnop 47
VU
  opcodes 42