VMIPS Programmer's Manual

This is the VMIPS Programmer's Manual, Sixth Edition, for version 1.5.

1 Overview

VMIPS is a simulator for a machine compatible with the MIPS R3000 RISC architecture. VMIPS consists entirely of software. No special hardware is required to run programs on VMIPS—that is, VMIPS is a virtual machine.

Since VMIPS is based on an already-existing architecture, it is relatively easy to find tools to build programs that will run on VMIPS. Since VMIPS is based on a RISC architecture, its primitive machine-language commands are all fairly simple to understand and implement.

VMIPS is easily extended by programmers to include more virtual devices, such as frame buffers, disk drives, etc. VMIPS is written in C++ and uses a fairly simple class structure. Furthermore, VMIPS is intended to be a "concrete" virtual machine which its users can modify at will—"concrete" meaning that it maintains a tight correspondence between its structures and structures which actually appear in modern physical computer hardware. For example, a programmer who wished to modify the CPU simulation could easily extract the `CPU` class from the VMIPS source code, and replace it with one which was more to his/her liking.

VMIPS is also designed with debugging and testing in mind, offering an interface to the GNU debugger GDB by which programs can be debugged while they run on the simulator. As such, it is intended to be a practical simulator target for compilers and assembly language/hardware-software interface courses.

VMIPS is free software. This means that you are free to share VMIPS with everyone, and we encourage you to do so, but we do not give you the freedom to restrict others from sharing it with everyone. For a comprehensive explanation please read the GNU General Public License.

2 Getting Started

Step 0. If VMIPS is installed on your system, you can start building programs with it right away. Otherwise, you (or your system administrator) will have to compile VMIPS first; see the appendix on Installation.

Step 1. First, compile your program. You should have a MIPS cross-compiler available. VMIPS supports the GNU C Compiler; most installations of VMIPS will also have an installation of the GNU C Compiler targetting the MIPS architecture. Your easiest interface to the C compiler will probably be through the '`vmipstool`' program; to run the MIPS compiler that VMIPS was installed with, use the '`vmipstool --compile`' command.

Step 2. Link your program with any support code necessary. VMIPS comes with some canned support code, in the share/vmips directory, or you can write your own support code. VMIPS comes with a linker script for simple standalone programs, which you can run with '`vmipstool --link`', or you can write your own linker script.

Step 3. Build a ROM image. This is necessary because the current version of VMIPS does not read in executables. Most real machines don't; they have an embedded program on a piece of flash ROM that reads in the first executable and runs it. This makes development a little more realistic, but not quite so convenient; this may change in the future, but for now it's necessary. To build a ROM image, use the script that comes with VMIPS, by running '`vmipstool --make-rom`'.

Step 4. Start the simulator using '`vmips ROMFILE`', where '`ROMFILE`' is the name of your ROM image. Your program should run to completion, and if you are using the canned setup code that comes with VMIPS, the simulator should halt when it hits the first `break` instruction, which should happen right after your `entry` function returns.

3 An Example

Let's assume you have VMIPS already compiled, and that you have some setup code in '`setup.s`', and a standalone program (i.e., not one meant to run under an operating system) in '`hello.c`'.

First assemble the setup code.

```
vmipstool --assemble -o setup.o setup.s
```

Note: if you get an 'installation error' from vmipstool, you may need to edit your '`/etc/vmipsrc`' the first time you use vmipstool. See the 'Post-Installation Setup' section of the 'Installation' appendix for more information.

Compile your program:

```
vmipstool --compile -c hello.c
```

Then, link your program and the setup code together to produce an executable:

```
vmipstool --link -o hello setup.o hello.o
```

Build a ROM image from the executable:

```
vmipstool --make-rom hello hello.rom
```

Run the program.

```
vmips hello.rom
```

The program will terminate, by default, when your setup code generates a breakpoint exception (using the `break` instruction, for example). This termination condition can be changed by adding one of the '`halt`' options to the file '`.vmipsrc`' in your home directory.

4 Building Programs

4.1 Source Languages

Programs for VMIPS are generally built out of C or assembly-language source code. It is theoretically possible to use C++ or other languages, but the infrastructure required has not yet been investigated or documented.

4.2 ROM Programs

The easiest way to get VMIPS to run a program is to install that program as the VMIPS ROM. Building a C program as a ROM requires that you link it with some setup code.

4.3 Default Setup Code

This section describes the default VMIPS setup code. It also describes the minimal set of things you need to do before you can run C code from the ROM, since that is the intended purpose of the default VMIPS setup code.

Start by clearing out registers and initializing TLB entries. The default VMIPS setup code sets up an identity virtual-to-physical address mapping in the TLB with valid entries for the first few pages of physical RAM.

Set yourself up a stack pointer ($sp). Usually this can just be some number of megabytes above the end of your code's data segment. You can get the address of the end of your code's data segment from your linker script.

Set up your globals pointer ($gp), if your code uses global data. You can get the right address from your linker script.

If you have writable data in ROM, your C code probably doesn't realize that it's in ROM, and it will want to write to it. You should copy the writable data to RAM. There is code to do this in the canned setup code provided with VMIPS.

Note: The canned setup code is hard-wired for 1 MByte of memory. It operates with a very simple memory map: writable data and bss (uninitialized data) above `DATA_START`, and the stack grows down from `DATA_START`. The linker script and the canned setup code share some hard-wired constants related to this memory map; you should be careful to coordinate your changes if you wish to change the memory map.

Finally, your setup code should finish by calling the entry point of your C code. Usually this will have a name like `entry`; using the name `main` is not recommended, because many versions of GCC assume that they can call standard C runtime setup functions (such as are normally found in '`crt0.o`') from the beginning of `main`. You may or may not want this.

When the C code returns, you will probably want to halt the machine; the default way to do this is by executing a break instruction. Read the following section for details.

4.4 Exceptions

4.4.1 Handling exceptions

Your startup code should have some kind of exception support. If you don't, exceptions are likely to make your program loop forever, because the jump to the exception vector will result in the execution of garbage or in a unmapped access, either of which are likely to cause exceptions.

An absolutely minimal exception handler is a break instruction at address 0xbfc00180, which will halt the machine on any exception, providing you have the '`haltbreak`' option set. This is also a handy way to halt the machine after your program ends, if you are writing kernel code; just follow the jump to your kernel code by a `break` instruction.

4.4.2 Exception vectors

If the Boot-time Exception Vectors are in use, exceptions use the base address 0xbfc00100 (which is in unmapped, uncached kernel space), otherwise they use the base address 0x80000000 (which is in unmapped, cached kernel space). You can control this by setting or clearing the Boot-time Exception Vector bit (bit 22, or 0x00400000) in the Status register (register 12 of coprocessor zero). If the bit is set, the Boot-time Exception Vectors will be used.

User-space TLB Miss exceptions have a special vector, which is obtained by adding 0 to the base address. All other exceptions use the general vector, which is obtained by adding 0x080 to the base address. This obviously places a bit of a restriction on the layout of the beginning of your ROM code: the setup code must either fit in the first 0x100 bytes, or it must be structured so that it jumps past the exception vectors.

4.4.3 Exception codes and their meanings

Whenever control is transferred to your exception handler, the ExcCode field of the Cause register, that is, bits 6 - 2 (0x007c) of register 13 of coprocessor 0, are filled in with one of the following exception codes. Each exception code has a canonical short name, included in parentheses next to the exception code number, and is followed by a short description of the circumstances where it occurs.

`0 (Int)` Hardware or software interrupt. Some device or process is trying to get the processor's attention.

`1 (Mod)` TLB modification exception. The memory address translation mapped to a TLB entry, but that entry's "dirty" bit was set.

`2 (TLBL)` TLB exception caused by a data load (i.e., a load word or similar instruction) or instruction fetch. The memory address translation did not match any valid TLB entry.

`3 (TLBS)` TLB exception caused by a data store (i.e., a store word or similar instruction). The memory address translation did not match any valid TLB entry.

`4 (AdEL)` Address error exception caused by a data load or instruction fetch. The PC was not word-aligned, or the address the load instruction wanted to load from was not aligned to the width of the load instruction. (For example, load halfword instructions must be 2-byte aligned.)

`5 (AdES)` Address error exception caused by a data store. The address the store instruction wanted to store to was not aligned to the width of the store instruction. (For example, store halfword instructions must be 2-byte aligned.)

`6 (IBE)` Bus error caused by an instruction fetch. The PC does not correspond to any real area of memory.

`7 (DBE)` Bus error caused by a data load or store. The target address of the load or store instruction does not correspond to any real area of memory.

`8 (Sys)` SYSCALL exception. Some code was trying to call the operating system, using a SYSCALL instruction. This exception is the processor's way of transferring control to the operating system.

`9 (Bp)` Breakpoint exception. Some process executed a BREAK instruction. This is the processor's way of allowing the operating system to stop the process and do whatever is appropriate (alert the user using the debugger, for example).

`10 (RI)` Reserved instruction exception. Some code executed something which wasn't a valid MIPS-1 instruction.

`11 (CpU)` Coprocessor Unusable. Some code executed an instruction which tried to reference a coprocessor that isn't configured in VMIPS.

`12 (Ov)` Arithmetic Overflow. Some code executed an instruction whose arithmetic answer was too big to fit in a register using two's-complement arithmetic. The processor issues this exception so that the operating system can stop or otherwise signal the process.

`13 (Tr)` Trap. This exception is only issued on the R4000 or R6000 processor and compatibles.

`14 (NCD)` LDCz or SDCz (coprocessor load/store) using an address which wasn't in the cache. This exception is only issued on the R6000 processor and compatibles.

`14 (VCEI)` Virtual Coherency Exception (instruction). This exception is only issued on the R4000 processor and compatibles.

`15 (MV)` Machine check exception. This exception is only issued on the R6000 processor and compatibles.

`15 (FPE)` Floating-point exception. This exception is only issued on the R4000 processor and compatibles.

`16-22` Reserved, not used.

`23 (WATCH)`
Reference to WatchHi/WatchLo address detected. This exception is only issued on the R4000 processor and compatibles.

`24-30` Reserved, not used.

`31 (VCED)` Virtual Coherency Exception (data). This exception is only issued on the R4000 processor and compatibles.

4.4.4 Exception prioritizing

It is possible for more than one exception to occur during the emulation of the same instruction. The MIPS architecture has a system for determining which of a set of conflicting exceptions is reported to the exception handler.

When two or more exceptions occur on the same execution of the same instruction, only one is reported, according to the priority list, below. The ordering is by exception code (EXCCODE) and mode of memory access (MODE), where applicable. Each ordered pair (EXCCODE, MODE) below has the priority listed in brackets. * denotes a position where any value matches.

This prioritization is implemented in the `exception_priority()` member function of class CPU.

`[1]` Address error - instruction fetch (AdEL, INSTFETCH)

`[2]` TLB refill - instruction fetch TLB invalid - instruction fetch (TLBL, INSTFETCH) (TLBS, INSTFETCH)

`[3]` Bus error - instruction fetch (IBE, *)

`[4]` Integer overflow, Trap, System call, Breakpoint, Reserved Instruction, or Coprocessor Unusable (Ov, *) (Tr, *) (Sys, *) (Bp, *) (RI, *) (CpU, *)

`[5]` Address error - data load or data store (AdEL, DATALOAD) (AdES, *)

`[6]` TLB refill - data load or data store TLB invalid - data load or data store (TLBL, DATALOAD) (TLBS, DATALOAD) (TLBL, DATASTORE) (TLBS, DATASTORE)

`[7]` TLB modified - data store (Mod, *)

`[8]` Bus error - data load or data store (DBE, *)

`[9]` Interrupt (Int, *)

4.5 Linking

You want the text section of your program to start with the setup code, so link in the setup code first — that is, put the name of the object file containing the setup code first on the linker command line.

You want the setup code to start at 0xbfc00000, which is the MIPS reset exception vector. In practical terms, when VMIPS starts up, it will reset. When VMIPS resets, it jumps to 0xbfc00000, which is the beginning of your setup code.

4.6 Common Errors in Compilation

If the linker complains about not being able to find the symbol `_gp_disp`, you should turn on the GCC option '`-mno-abicalls`'. `_gp_disp` is used by the SGI N32 ABI for MIPS ELF. One reliable reference source claims, "`_gp_disp` is a reserved symbol defined by the linker to be the distance between the lui instruction and the context pointer." The GNU linkers currently in use do not appear to support this function.

If you get lots of `R_MIPS_GPREL16` relocation failures from the linker, there are two workarounds: either combine all the files together first with '`ld -x -r -o bigfile.o <`*all your files*`>`' and then use '`vmipstool --link`' on '`bigfile.o`', or compile with '`-G 0`' in your `CFLAGS`.

4.6.1 Dealing with kernel code in GCC

If you have a `main()` function in your code, GCC expects it to return an int. If you don't like this, use '`-ffreestanding`' or '`-Wno-main`'. You have to have GCC 2.95.2 or later for this to work, though; it won't work in EGCS 1.1.1.

If you have a `main()` function in your code, GCC will try to call `__main` or some other kind of setup function even if you use '`-ffreestanding`'. If you don't want this call made, a simple workaround is to call the entry function `entry` instead of `main`. You can also try defining `void __main(void)` to be an empty function.

4.6.2 MIPS position-independent code

In Linux/MIPS, PIC (position-independent code) is the default. An important implication is that GCC configured for `mips-linux` or `mipsel-linux` will, by default, use the `$gp` register to access global variables. This is fine in most user programs, but you may find that it causes you trouble if you are writing a ROM for use in VMIPS because the `$gp` register may not contain a valid pointer. The easiest thing to do is turn off this behavior by giving gcc the '`-fno-pic`' option, which also turns on the gas '`-non_shared`' option. If you compile with vmipstool, it will turn on these flags for you by default.

In many MIPS operating systems, register `$t9` is reserved for the ABI as a globals pointer when compiling position-independent code, and the assembly writer (programmer or compiler) must issue `.cpsave` and `.cprestore` assembler directives upon function entry and exit to put the right globals pointer values into this register. Similarly to the '`-fno-pic`' option above, it is easy to turn this off while writing ROM code by using the gcc '`-mno-abicalls`' option. If you compile with vmipstool, it will turn this flag on for you by default.

4.6.3 Building ROMs

If it takes a long time to build a ROM or the ROM file fills the disk, make sure all the sections your linker is producing are accounted for in the linker script. Do an '`objdump -x`' on the executable which you are using to build the ROM image, and make sure that the difference between any two of the LMAs (load memory addresses) of the sections in the file is not a lot bigger than the total size of the executable. This metric is strictly a rule of thumb, but it easily identifies when a section has not been put into the linker script: if a load memory address for some section is expecting to be in RAM (0xa0000000, for example),

and the load memory address for all the other sections is in ROM (around 0xbfc00000), then you will lose because writing out a memory image to be used as a ROM file would take roughly 0xbfc00000 - 0xa0000000 = 532676608 bytes (about 500 megs). The solution is to make sure that all LMAs in the executable are sane with respect to the '`loadaddr`' variable in your '`.vmipsrc`', usually by adding any new sections you find to either the .text, .data, or .bss section of the linker script.

5 Invoking vmips

VMIPS is started by running the "vmips" program from the command line. The format of the VMIPS command line is any one of the following:

```
vmips [-n] [-F FILE] [-o option_string] ... rom_file
vmips --help
vmips --version
vmips --print-config
```

This is what the different command line options mean:

'`-F FILE`' Read options from FILE instead of the '`.vmipsrc`' in your home directory.

'`-n`' Do not read the system-wide configuration file, usually called '`/usr/local/etc/vmipsrc`'.

'`--help`' Prints a short summary of VMIPS command line options, and exits successfully.

'`--version`'
Prints a short summary of VMIPS version and copyright information, and exits successfully.

'`--print-config`'
Prints a short summary of VMIPS compile-time configuration information, and exits successfully.

'`-o something`'
Set the option "something" as if "something" were in your .vmipsrc file. See the "VMIPS options" section of the "Customizing" chapter for more information on what kind of things can go in your .vmipsrc file. You can use as many -o options on the command line as your shell will let you.

'`rom_file`'
Use the named file as the ROM file VMIPS should boot. This option is mandatory.

6 Customizing

6.1 VMIPS options

The VMIPS simulator gets runtime options from four different sources, in this order: first, it checks its compile-time defaults, which are set by the site administrator in the source file '`optiontbl.h`'. Then, the system-wide configuration file is read, unless you specify the '`-n`' option; usually this file is called '`/usr/local/etc/vmipsrc`', but it may have been moved by the site administrator or by the maintainers of your distribution. (This is configurable in the source file '`options.h`', and by specifying the –prefix and –sysconfdir options to the GNU `configure` script when building VMIPS.) Next, it checks the user's own configuration file, usually the file '`.vmipsrc`' in your home directory, or whatever file you specify using the '`-F`' option. Last, it reads the command line, and gets any options listed there.

6.2 Format of the configuration file

The configuration file may contain as many options per line as you want, provided no single line exceeds 1,024 characters in length. Whitespace separates options from one another. Single quotes and backslash are valid in the configuration file. Their meanings are similar to those found in the Bourne shell: any text within paired single quotes is uninterpreted, as is any character immediately following a backslash. A comment is any text starting from a hash mark to the end of the line, inclusive.

A string or number option named NAME can appear as NAME=VALUE, where VALUE is the string or number in question. If the number begins with 0x, it will be interpreted as a 32-bit hexadecimal number, and if it begins with 0, it will be interpreted as octal. Otherwise, it will be interpreted as a decimal number. Numbers are always unsigned. A Boolean option named NAME can appear as either NAME (to set it to TRUE) or noNAME (to set it to FALSE).

6.3 Summary of configuration options

The following is a list of the configuration options present in this version of VMIPS.

'`haltdumpcpu`' (type: Boolean)

Controls whether the CPU registers and stack will be dumped on halt. For the output format, please see the description of the '`dumpcpu`' option, below. The default value is FALSE.

'`haltdumpcp0`' (type: Boolean)

Controls whether the system control coprocessor (CP0) registers and the contents of the translation lookaside buffer (TLB) will be dumped on halt. For the output format, please see the description of the '`dumpcp0`' option, below. The default value is FALSE.

'`excpriomsg`' (type: Boolean)

Controls whether exception prioritizing messages will be printed. These messages attempt to explain which of a number of exceptions caused by the same instruction will be reported. The default value is FALSE.

‘`excmsg`’ (type: Boolean)

Controls whether every exception will cause a message to be printed. The message gives the exception code, a short explanation of the exception code, its priority, the delay slot state of the virtual CPU, and states what type of memory access the exception was caused by, if applicable. Interrupt exceptions are only printed if ‘`reportirq`’ is also set; when they occur, they also have Cause and Status register information printed. TLB misses will have fault address and user/kernel mode information printed. The default value is FALSE.

‘`bootmsg`’ (type: Boolean)

Controls whether boot-time and halt-time messages will be printed. These include ROM image size, self test messages, reset and halt announcements, and possibly other messages. The default value is TRUE.

‘`instdump`’ (type: Boolean)

Controls whether every instruction executed will be disassembled and printed. The default value is FALSE. The output is in the following format:

```
PC=0xbfc00000 [1fc00000]    24000000 li $zero,0
```

The first column contains the PC (program counter), followed by the physical translation of that address in brackets. The third column contains the machine instruction word at that address, followed by the assembly language corresponding to that word. All of the constants except for the assembly language are in hexadecimal.

‘`dumpcpu`’ (type: Boolean)

Controls whether the CPU registers and stack will be dumped after every instruction. The default value is FALSE. The output is in the following format:

```
Reg Dump: [ PC=bfc00180  LastInstr=0000000d  HI=00000000  LO=00000000
            DelayState=NORMAL  DelayPC=bfc00308  NextEPC=bfc00308
 R00=00000000  R01=00000000  R02=00000000  R03=a00c000e  R04=0000000a
 ...
 R30=00000000  R31=bfc00308  ]
Stack: 00000000 00000000 00000000 00000000 a2000008 a2000008 ...
```

(Some values have been omitted for brevity.) Here, PC is the program counter, LastInstr is the last instruction executed, HI and LO are the multiplication/division result registers, DelayState and DelayPC are used in delay slot processing, NextEPC is what the Exception PC would be if an exception were to occur, and R00 ... R31 are the CPU general purpose registers. Stack represents the top few words on the stack. All values are in hexadecimal.

‘`dumpcp0`’ (type: Boolean)

Controls whether the system control coprocessor (CP0) registers and the contents of the translation lookaside buffer (TLB) will be dumped after every instruction. The default value is FALSE. The output is in the following format:

```
CP0 Dump Registers: [        R00=00000000  R01=00003200
 R02=00000000  R03=00000000  R04=001fca10  R05=00000000
 R06=00000000  R07=00000000  R08=7fb7e0aa  R09=00000000
 R10=00000000  R11=00000000  R12=00485e60  R13=f0002124
 R14=bfc00308  R15=0000703b ]
Dump TLB: [
Entry 00: (00000fc000000000) V=00000 A=3f P=00000 ndvg
```

```
Entry 01: (00000fc000000000) V=00000 A=3f P=00000 ndvg
Entry 02: (00000fc000000000) V=00000 A=3f P=00000 ndvg
Entry 03: (00000fc000000000) V=00000 A=3f P=00000 ndvg
Entry 04: (00000fc000000000) V=00000 A=3f P=00000 ndvg
Entry 05: (00000fc000000000) V=00000 A=3f P=00000 ndvg
...
Entry 63: (00000fc000000000) V=00000 A=3f P=00000 ndvg
]
```

Each of the R00 .. R15 are coprocessor zero registers, in hexadecimal. The Entry 00 .. 63 lines are TLB entries. The 64-bit number in parentheses is the hexadecimal raw value of the entry. V is the virtual page number. A is the ASID. P is the physical page number. NDVG are the Non-cacheable, Dirty, Valid, and Global bits, uppercase if on, lowercase if off.

'`haltibe`' (type: Boolean)

If '`haltibe`' is set to TRUE, the virtual machine will halt after an instruction fetch causes a bus error (exception code 6, Instruction bus error). This is useful if you are expecting execution to jump to nonexistent addresses in memory, and you want it to stop instead of calling the exception handler. It is important to note that the machine halts after the exception is processed. The default value is TRUE.

'`haltbreak`' (type: Boolean)

If '`haltbreak`' is set to TRUE, the virtual machine will halt when a breakpoint exception is encountered (exception code 9). This is equivalent to halting when a `break` instruction is encountered. It is important to note that the machine halts after the breakpoint exception is processed. The default value is TRUE.

'`haltdevice`' (type: Boolean)

If '`haltdevice`' is set to TRUE, the halt device is mapped into physical memory, otherwise it is not. The default value is TRUE.

'`instcounts`' (type: Boolean)

Set '`instcounts`' to TRUE if you want to see instruction counts, a rough estimate of total runtime, and execution speed in instructions per second when the virtual machine halts. The default value is FALSE. The output is printed at the end of the run, and is in the following format:

```
7337 instructions in 0.0581 seconds (126282.271 instructions per second)
```

'`romfile`' (type: string)

This is the name of the file which will be initially loaded into memory (at the address given in '`loadaddr`', typically 0xbfc00000) and executed when the virtual machine is reset. The default value is `"romfile.rom"`.

'`loadaddr`' (type: number)

This is the virtual address where the ROM will be loaded. Note that the MIPS reset exception vector is always 0xbfc00000 so unless you're doing something incredibly clever you should plan to have some executable code at that address. Since the caches and TLB are in an indeterminate state at the time of reset, the load address must be in uncacheable memory which is not mapped through the TLB (kernel segment `"kseg1"`). This effectively

constrains the valid range of load addresses to between 0xa0000000 and 0xc0000000. The default value is 0xbfc00000.

‘`memsize`’ (type: number)

This variable controls the size of the virtual CPU’s "physical" memory in bytes. The default value is 0x100000.

‘`memdump`’ (type: Boolean)

If ‘`memdump`’ is set, then the virtual machine will dump its RAM into a file, whose name is given by the ‘`memdumpfile`’ option, at the end of the simulation run. The default value is FALSE.

‘`memdumpfile`’ (type: string)

This is the name of the file to which a RAM dump will be written at the end of the simulation run. The default value is "memdump.bin".

‘`reportirq`’ (type: Boolean)

If ‘`reportirq`’ is set, then any change in the interrupt inputs from a device will be reported on stderr. Also, any Interrupt exception will be reported, if ‘`excmsg`’ is also set. The default value is FALSE.

‘`spimconsole`’ (type: Boolean)

When set, configure the SPIM-compatible console device. This is incompatible with ‘`decserial`’. The default value is TRUE.

‘`ttydev`’ (type: string)

This pathname will be used as the device from which reads from the SPIM-compatible console device’s Keyboard 1 will take their data, and to which writes to Display 1 will send their data. If the OS supports ttyname(3), that call will be used to guess the default pathname. If the pathname is the single word ‘`off`’, then the device will be disconnected. If the pathname is the single word ‘`stdout`’, then the device will be connected to standard output, and input will be disabled. The default value is "/dev/tty".

‘`ttydev2`’ (type: string)

See ‘`ttydev`’ option; this one is just like it, but pertains to Keyboard 2 and Display 2. The default value is "off".

‘`debug`’ (type: Boolean)

If debug is set, then the gdb remote serial protocol backend will be enabled in the virtual machine. This will cause the machine to wait for gdb to attach and ‘`continue`’ before booting the ROM file. If debug is not set, then the machine will boot the ROM file without pausing. The default value is FALSE.

‘`debugport`’ (type: number)

If debugport is set to something nonzero, then the gdb remote serial protocol backend will use the specified TCP port. The default value is 0.

‘`realtime`’ (type: Boolean)

If ‘`realtime`’ is set, then the clock device will cause simulated time to run at some fraction of real time, determined by the ‘`timeratio`’ option. If realtime is not set, then simulated time will run at the speed given by the ‘`clockspeed`’ option. The default value is FALSE.

‘`timeratio`’ (type: number)

If the ‘`realtime`’ option is set, this option gives the number of times slower than real time at which simulated time will run. It has no effect if ‘`realtime`’ is not set. The default value is 1.

‘`clockspeed`’ (type: number)

If the ‘`realtime`’ option is not set, you should set this option to the average speed in MIPS instructions per second at which your system runs VMIPS. You can get suitable values from turning on the ‘`instcounts`’ option and running some of your favorite programs. If you increase the value of ‘`clockspeed`’, time will appear to pass more slowly for the simulated machine; if you decrease it, time will pass more quickly. (To be precise, one instruction is assumed to take 1.0e9/‘`clockspeed`’ nanoseconds.) This option has no effect if ‘`realtime`’ is set. The default value is 250000.

‘`clockintr`’ (type: number)

This option gives the frequency of clock interrupts, in nanoseconds of simulated time, for the clock device. It does not affect the DECstation-compatible realtime clock. The default value is 200000000.

‘`clockdeviceirq`’ (type: number)

This option gives the interrupt line to which the clock device is connected. Values must be a number 2-7 corresponding to an interrupt line reserved for use by hardware. The default value is 7.

‘`clockdevice`’ (type: Boolean)

If this option is set, then the clock device is enabled. This will allow MIPS programs to take advantage of a high precision clock. The default value is TRUE.

‘`dbemsg`’ (type: Boolean)

If this option is set, then the physical addresses of accesses that cause data bus errors (DBE exceptions) will be printed. The default value is FALSE.

‘`decrtc`’ (type: Boolean)

If this option is set, then the DEC RTC device will be configured. The default value is FALSE.

‘`deccsr`’ (type: Boolean)

If this option is set, then the DEC CSR (Control/Status Register) will be configured. The default value is FALSE.

‘`decstat`’ (type: Boolean)

If this option is set, then the DEC CHKSYN and ERRADR registers will be configured. The default value is FALSE.

‘`decserial`’ (type: Boolean)

If this option is set, then the DEC DZ11 serial device will be configured. This is incompatible with ‘`spimconsole`’. The default value is FALSE.

‘`tracing`’ (type: Boolean)

If this option is set, VMIPS will keep a trace of the last few instructions executed in memory, and write it out when the machine halts. This incurs a substantial performance

penalty. Use the '`tracesize`' option to set the size of the trace you want. The default value is FALSE.

'`tracesize`' (type: number)

Set this option to the maximum number of instructions to keep in the dynamic instruction trace. This has no effect if '`tracing`' is not set. The default value is 100000.

'`bigendian`' (type: Boolean)

If this option is set, then the emulated MIPS CPU will be in Big-Endian mode. Otherwise, it will be in Little-Endian mode. You must set it to correspond to the type of binaries that your assembler and compiler are configured to produce, which is not necessarily the same as the endianness of the CPU on which you are running VMIPS. (The default may not be meaningful for your setup!) The default value is FALSE.

'`tracestartpc`' (type: number)

If the tracing option is set, then this is the PC value which will trigger the start of tracing. Otherwise it has no effect. The default value is 0.

'`traceendpc`' (type: number)

If the tracing option is set, then this is the PC value which will trigger the end of tracing. Otherwise it has no effect. The default value is 0.

'`mipstoolprefix`' (type: string)

vmipstool uses this option to locate your MIPS-targetted cross compilation tools, if you have them installed. If your MIPS GCC is installed as /opt/mips/bin/mips-elf-gcc, then you should set this option to "/opt/mips/bin/mips-elf-". vmipstool looks for the "gcc", "ld", "objcopy" and "objdump" programs starting with this prefix. This option should be set in your installed system-wide VMIPS configuration file (vmipsrc) by the "configure" script; the compiled-in default is designed to cause an error. The default value is "/nonexistent/mips/bin/mipsel-ecoff-".

'`execname`' (type: string)

Name of executable to be loaded by automatic kernel loader. This is an experimental, unfinished feature. The option value must be the name of a MIPS ECOFF executable file, or 'none' to disable the option. The executable's headers must specify load addresses in KSEG0 or KSEG1 (0x80000000 through 0xbfffffff). The default value is "none".

'`fpu`' (type: Boolean)

True to enable hooks in the CPU to communicate with a floating-point unit as coprocessor 1. The floating-point unit is not implemented; only the hooks in the CPU are. This is an experimental, unfinished feature. The default value is FALSE.

'`testdev`' (type: Boolean)

True to enable a memory-mapped device that is used to test the memory-mapped device interface. The VMIPS test suite turns this device on as necessary; you should not normally need to enable it. The default value is FALSE.

7 Invoking vmipstool

`vmipstool` is intended to be a friendly front-end to the process of compiling, linking, and assembling code for VMIPS using the GNU Compiler Collection (GCC) and GNU Binutils.

Note that you do not need to use `vmipstool`, or even GCC, to compile programs for VMIPS; you can use any MIPS compiler and assembler you have handy.

The format of the vmipstool command line is as follows:

```
vmipstool [ --verbose ] [ --dry-run ] --compile [ FLAGS ]
    FILE.c -o FILE.o
vmipstool [ --verbose ] [ --dry-run ] --preprocess [ FLAGS ] FILE
vmipstool [ --verbose ] [ --dry-run ] --assemble [ FLAGS ]
    FILE.s -o FILE.o
vmipstool [ --verbose ] [ --dry-run ] [ --ld-script=T ] --link
    [ FLAGS ] FILE1.o ... FILEn.o -o PROG
vmipstool [ --verbose ] [ --dry-run ] --make-rom PROG PROG.rom
vmipstool [ --verbose ] [ --dry-run ] --disassemble-rom PROG.rom
vmipstool [ --verbose ] [ --dry-run ] --disassemble-word PC INSTR
vmipstool [ --verbose ] [ --dry-run ] --disassemble PROG (or FILE.o)
vmipstool [ --verbose ] [ --dry-run ] --swap-words INPUT OUTPUT
vmipstool --help
vmipstool --version
```

This is what the different command line options mean:

‘`--help`’ Display this help message and exit.

‘`--version`’
Display the version of vmipstool and exit.

‘`--verbose`’
Echo commands as they are run.

‘`--dry-run`’
Don’t actually run anything; use with –verbose.

‘`--ld-script=T`’
Use T as the linker script (instead of default script); use with –link.

‘`--compile`’
Compile C code. The remainder of the command line must consist of arguments to the GNU C compiler.

‘`--preprocess`’
Preprocess C source code or assembly code. The remainder of the command line must consist of arguments to the GNU C preprocessor.

‘`--assemble`’
Translate assembly code to object files. The remainder of the command line must consist of arguments to the GNU assembler.

‘`--link`’ Link objects together to create an executable. The remainder of the command line must consist of arguments to the GNU linker.

‘`--make-rom`’
: Write a program into a ROM file. The next 2 arguments are the executable and the ROM file, respectively.

‘`--disassemble`’
: Disassemble a relocatable object file (*.o file) or an executable.

‘`--disassemble-rom`’
: Disassemble arbitrary data, possibly including ROM files. (More information is available with –disassemble, but it only works on programs which have not been written into ROMs.)

‘`--disassemble-word`’
: Disassemble an instruction whose binary encoding is passed as a command-line argument. Because instructions can have PC-relative immediate arguments, the PC must also be specified.

‘`--swap-words`’
: Copy the input file to the output file, reversing the byte-order of each 32-bit word in the process. This can be used to translate chunks of data from big-endian to little-endian, or vice-versa.

Note that `vmipstool` consults your ‘`/etc/vmipsrc`’ and ‘`~/.vmipsrc`’ to determine where the MIPS cross compiler, assembler, and `objdump` and `objcopy` tools are. If you install new cross-tools, you should edit these configuration files to reflect the new location of the MIPS tools, and to reflect their default endianness, by changing the settings of ‘`mipstoolprefix`’ and ‘`bigendian`’.

8 Programming

In this section we attempt to give some hints about writing code for VMIPS. They are primarily intended for assembly language programmers, but should be helpful to anyone interested in the MIPS architecture. This section will not replace a good MIPS reference; check the "References" section for more information about these. However, any help is appreciated for making this section more complete.

8.1 Delay slot handling

MIPS branch instructions' effects are delayed by one instruction; the instruction following the branch instruction is always executed, regardless of whether the branch is taken. This is a consequence of the pipeline which is not important to virtual machine architecture, except that it has to be emulated correctly.

VMIPS emulates delay slot handling by means of a tiny state machine, whose state is called the delay state. The virtual CPU can be in a delay state of `DELAYING`, `DELAYSLOT`, or `NORMAL` at the beginning of the call to `step()`. The VMIPS delay slot state machine's state is displayed when you use the '`dumpcpu`' option. See the "Summary of configuration options" section of the "Customizing" chapter for more information about this option.

A delay state of `NORMAL` corresponds to execution in the non-branch case.

A delay state of `DELAYING` means that the instruction being executed caused a branch to be taken, and the next instruction to execute is in the delay slot.

A delay state of `DELAYSLOT` means that the instruction just executed was in the delay slot, and the next instruction to execute is the branch target. If there is an exception, the exception PC will be the PC of the branch instruction, not of this one.

9 Debugging

VMIPS supports debugging programs running on the virtual machine by providing an interface to GDB, the GNU debugger. GDB talks to VMIPS using its built-in remote serial protocol, over a local TCP connection. See the "Remote Serial" section of the GDB manual for details of the protocol.

You must use a MIPS-targetted GDB to debug programs running on VMIPS; that is, you must use a copy of GDB that understands MIPS assembly language and registers. Usually, a copy of GDB configured this way will have a name starting with '`mips`', e.g., '`mipsel-ecoff-gdb`'. See the "Installation" section of the manual for more information on configuring and building a MIPS-targetted GDB.

If you want to use the VMIPS GDB interface, set the '`debug`' flag on the command line. VMIPS will wait for you to attach GDB and type '`continue`' at the GDB prompt before booting the ROM file.

To attach GDB to VMIPS, look for the line in the VMIPS startup message that reads:

```
Use this command to attach debugger: target remote 127.0.0.1:3371
```

Make a note of the `target remote` command VMIPS printed out. The host and port numbers ('`127.0.0.1:3371`') are likely to be different on your machine than are shown here. If you want to force a particular port to be used, specify the '`-o debugport=PORT`' option when starting VMIPS.

When VMIPS pauses and says '`Waiting for connection from debugger`', open up GDB in another window or on another terminal on the program you are debugging. Do not try to open GDB on the ROM file, because GDB doesn't understand ROM files. Instead, give GDB the name of the executable you used as input when creating the ROM file. For best results, it should have been compiled with '`-g`', so that it will contain extra debug information, and should not have had `strip` run on it.

Once GDB is open, type in to GDB the `target remote` command that VMIPS printed out. GDB will connect to VMIPS, which will be stopped at the first instruction of your setup code. Then you can set breakpoints, single step, or just let the program continue. VMIPS will return control to GDB on exceptions.

If you change your mind while VMIPS is waiting for a debug connection, type Control-C to VMIPS to cancel.

Here is what the whole setup process looks like in VMIPS:

```
% ./vmips -o debug boot.rom
Auto-size ROM image: 4096 words.
Running self tests.
Little-Endian host processor detected.
Self tests passed.
Use this command to attach debugger: target remote 127.0.0.1:33891
Mapping ROM image (boot.rom): 4096 words at 0xbfc00000 [1fc00000]
Attached SerialHost(fd 5) at 0x808cab8 to SPIMConsole [host=0x808cac8]
Attached SPIMConsole [host=0x808cac8] to phys addr 0x2000000
Connecting IRQ2-IRQ6 to console.
Mapped (host=0x401a4008) 1024k RAM at base phys addr 0
```

```
*************RESET*************

Waiting for connection from debugger.
```

Here is what the whole setup process looks like in GDB:

```
% mips-dec-ultrix4.5-gdb boot.exe
GNU gdb 4.17
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=i586-pc-linux-gnu --target=mips-dec-ultrix4.5"...
(gdb) target remote 127.0.0.1:33891
Remote debugging using 127.0.0.1:33891
__start () at setup.S:24
24              move $1, $0
Current language:  auto; currently asm
```

9.1 GDB, VMIPS and Signals

Since VMIPS does not know what operating system you are running on it, and GDB does not believe in hardware exceptions (only operating system signals), VMIPS has its own mapping of hardware exceptions to signals.

The mapping is as follows: Each signal is followed by a list of the hardware exceptions that map to it.

SIGINT

Interrupt

SIGSEGV

TLB modification exception
TLB exception (load or instruction fetch)
TLB exception (store)
Address error exception (load or instruction fetch)
Address error exception (store)

SIGBUS

Instruction bus error
Data (load or store) bus error

SIGTRAP

SYSCALL exception
Breakpoint exception (BREAK instruction)
Processor reset (only at VMIPS startup)

SIGILL

Reserved instruction exception

SIGFPE

Coprocessor Unusable
Arithmetic Overflow

SIGHUP

(Anything else.)

9.1.1 Startup behavior

Upon connecting to the VMIPS socket, gdb asks for the number of the signal that stopped VMIPS. Of course, there was no exception, since no instructions have executed, but we have to give a reason anyway. The signal that is always returned is the signal corresponding to the breakpoint exception – hence the listing for processor reset in the signal table above, even though reset is not really an ordinary exception.

9.2 ROM Breakpoints

VMIPS supports the setting of breakpoints in ROM. This would not be extraordinary except that MIPS breakpoints are usually implemented by GDB's remote serial protocol by overwriting instructions with MIPS `break` instructions. VMIPS keeps a single bit for each word of ROM, in order to tell whether that instruction is really a breakpoint. GDB keeps track of setting and unsetting the breakpoints.

9.3 Limitations

9.3.1 Detaching Limitations

When the debugger disconnects or detaches from VMIPS, the system will halt and VMIPS will exit.

9.3.2 Protocol Limitations

The GDB remote serial protocol supports lots of packets, but VMIPS does not support all of them. The following subset of the GDB remote serial protocol is implemented.

packet 'g': Read registers
packet 'G': Write registers
packet 'm': Read memory
packet 'M': Write memory
packet 'c': Continue
packet 'D': Detach
packet 's': Single step
packet 'k': Kill target
packet 'H': Set thread

packet 'T': Poll thread

packet '?': What was the last signal?

packet 'Z': Set breakpoint

packet 'z': Remove breakpoint

packet 'q': Query

For some of these packets, not all the arguments are supported. For example, in general, VMIPS acts as if it has exactly one thread.

9.4 Using Insight as a GUI for VMIPS

You can use the Insight graphical front end for GDB as a graphical front end for VMIPS.

As with GDB, you must use a MIPS-targetted Insight to debug programs running on VMIPS; that is, you must use a copy of Insight that understands MIPS assembly language and registers. Usually, a copy of Insight configured this way will have a name starting with '`mips`', e.g., '`mipsel-ecoff-gdb`'. (Confusingly, Insight binaries are also named '`gdb`'.)

Now let's walk through an example scenario where we want to use Insight to debug a program running in ROM on VMIPS.

1. Start VMIPS using the '`-o debug`' command line flag, to activate the debugging interface, and specify the name of the ROM file containing the ROM you want to debug.

2. Start Insight.

3. Choose Open... from the File menu. Select the executable file corresponding to the ROM file you just loaded in to VMIPS.

4. VMIPS will have printed out a message like:

```
Use this command to attach debugger: target remote 127.0.0.1:3082
```

To tell Insight what to do, choose Target Settings... from the File menu. In the Connection panel, set the Target to Remote/TCP, and set the Hostname to 127.0.0.1, and set the Port to 3082. Then hit OK.

5. Now, choose Connect to target from the Run menu. This will probably bring up a dialog box affirming that the connection was successful. Now you can look at registers, step through code, and whatnot, till your heart's content.

10 Interaction

The interactive inspector (also known as the interactor) provides a way to inspect a running vmips system without attaching a debugger.

You can enter the interactive inspector at any time while the simulation is running. To enter the interactive inspector, hit Control-underscore (`^_`) which enters an ASCII 0x1F (US) byte at the terminal. This will cause the simulation to pause and the `"vmips=>"` interactor prompt to be printed.

The interactor will loop prompting you for commands and processing them until you enter a command that exits the interactor.

10.1 Command table

'`go`' Continue execution and exit the interactive inspector. You can reenter the interactive inspector again later by hitting Control-underscore again.

'`step`' Attempt to step one instruction. After a single-step attempt, the PC will be printed.

'`halt`' Halt machine, exit the interactive inspector, and quit VMIPS.

'`quit`' Same as halt.

'`regs`' Print the CPU registers.

'`cp0`' Print the CP0 registers and TLB.

'`mem A`' Print the first few words of memory starting from address A. The address provided must be a 32-bit, word-aligned address.

'`dis A`' Disassemble the first few words of memory at address A. The address provided must be a 32-bit, word-aligned address. Due to a limitation in the current implementation, disassembly will invariably be printed to standard error.

'`stack`' Print the first few words of memory starting from the stack pointer.

'`help`' Print a list of available commands.

11 Devices

VMIPS comes with a few standard devices.

11.1 SPIM-compatible console device

The SPIM-compatible Console Device models a serial controller with two 200-baud full-duplex communication lines and a 1 Hz clock providing timer interrupts. This console device is currently the standard console device used in VMIPS.

11.1.1 Memory-mapped registers

The SPIM-compatible console device communicates with the CPU by means of a series of 9 32-bit-wide control and data registers, for a total of 36 memory-mapped bytes. The control registers are used for enabling and disabling specific devices' interrupt request mechanisms, and for determining which device(s) is/are ready for data when polling or during interrupt processing.

The following table details the offset of each register within the console device's mapped memory:

offset 0x00	Keyboard 1 Control
offset 0x04	Keyboard 1 Data
offset 0x08	Display 1 Control
offset 0x0c	Display 1 Data
offset 0x10	Keyboard 2 Control
offset 0x14	Keyboard 2 Data
offset 0x18	Display 2 Control
offset 0x1c	Display 2 Data
offset 0x20	Clock Control

Within each control register, Bit 2 of each word is the Device Interrupt Enable bit, and bit 1 is the Device Ready bit. Only the Device Interrupt Enable bits of the control registers are writable; other bits must be written as zero. Only Device Interrupt Enable and Device Ready are readable; other bits read as zero. Initially the Interrupt Enable bits on all SPIM console control words are unset.

Within each data register, writes are allowed only to the least-significant 8 bits; the other 24 bits read as zero and must be written as zero.

11.1.2 Interrupts

With a SPIM-compatible Console Device configured, the following interrupt lines are enabled.

Interrupt line 2 (Cause bit 0x0400) is wired to the Clock
Interrupt line 3 (Cause bit 0x0800) is wired to the #1 Keyboard
Interrupt line 4 (Cause bit 0x1000) is wired to the #1 Display
Interrupt line 5 (Cause bit 0x2000) is wired to the #2 Keyboard
Interrupt line 6 (Cause bit 0x4000) is wired to the #2 Display

When any one of the console devices is both ready and has its Device Interrupt Enable bit set, it requests an interrupt. (You must have the interrupt mask and interrupt enable bits of the CP0 Status register set for this request to succeed.) It follows that if the device becomes ready and then the user sets the Device Interrupt Enable bit, the device will immediately attempt to request an interrupt. You can determine which device requested the interrupt by examining the Interrupt Pending field of the CP0 Cause register in your interrupt handler code.

11.1.3 Display

The display data register is write-only. On a write to the data register, the display becomes unready and writes a char to the connected serial interface; it becomes ready again in 40 ms.

11.1.4 Clock

The Clock has no data register and becomes ready at most every second. A read from the Clock Control register makes the clock unready. Writes to the clock control register are as above.

11.1.5 Keyboard

The keyboard is initially unready; whenever the connected serial interface has a byte waiting on input, and the keyboard is unready, the keyboard reads the byte into its buffer, and becomes ready. If the keyboard is ready for more than 40 ms., it will check the connected serial interface again. If there is another byte available, it will read it and save it in the buffer, writing over the one which was originally in the buffer. No provision is made for detection of these buffer overruns. Updates to the keyboard buffer happen at most once per instruction fetched.

The keyboard data register is read-only. On a read from the data register, if the keyboard is ready it becomes unready and returns the byte in its holding buffer. If the keyboard data register is read while the keyboard is unready, the data in the buffer is the same as when the keyboard was last ready.

11.1.6 Compatibility

The SPIM-compatible console device is based on the SPIMSAL 4.4.2 version, which generally provides a superset of the functionality of the console device provided in SPIM 5.x and 6.x.

In SPIM 5.x/6.x, the keyboard controller appears at virtual address 0xffff0000. Keyboard 2, Display 2 and the Clock device are not available. (This is the same layout used in Patterson and Hennessy's Computer Organization and Design textbook.) Therefore, in order to get compatible behavior from the VMIPS SPIM-compatible console device, your startup code should configure the TLB to map virtual page number 0xffff0 to the physical addresses where the SPIM-compatible console device is configured.

In SPIM, when you read or write to a memory-mapped I/O register, only the virtual address and the data value stored are considered, not the width of the access. This means

that on a big-endian machine, you can (for example) write the display at the most-significant byte of the display data word (using a store byte instruction), or at the least-significant byte of the word (using a store word instruction). In VMIPS, you must always write the least-significant byte.

In SPIMSAL, it is believed to be the case that reads always read from keyboard 1, never from keyboard 2; whereas the user may write to either display, but data written to either display are invariably written to the simulator's standard output. Compatibility with these bugs is not supported.

11.1.7 Disconnected operation

The SPIM console device can be configured to turn off either the first or the second display/keyboard pair. Use the special keyword '`off`' in place of a device name, e.g., '`-o ttydev=off`', to turn off a console line. When a console line is turned off, it is described as 'disconnected', and behaves as follows:

Interrupts can be turned on and off as usual.

A disconnected keyboard always has Device Ready clear and always returns ASCII NULs (zero bytes) when its data word is read.

A disconnected display discards bytes written to it, and always has Device Ready set, but if its interrupts are turned on, it will not generate an interrupt.

11.2 Standard clock device

This section documents the standard clock device for VMIPS. It is intended to support user programs' access to real and simulated time. The clock device supports a hardware clock interrupt to notify MIPS programs of the passage of a prespecified number of nanoseconds, determined by the user's setting of the '`clockintr`' option. This clock provides a much higher resolution than the SPIM-compatible console device's 1Hz clock. The clock is enabled or disabled with the '`clockdevice`' option.

11.2.1 Memory-mapped registers

The standard clock device has 5 registers, configured to be mapped into memory at physical address 0x01010000. The following table defines the layout of the memory-mapped clock device registers:

`offset 0x00`
: real time, seconds

`offset 0x04`
: real time, microseconds

`offset 0x08`
: simulated time, seconds

`offset 0x0c`
: simulated time, microseconds

`offset 0x10`
: control word

Writing any of the clock's real time words is undefined. Writing a clock's simulated time word sets that component of the simulated time if the number written is a non-negative signed integer, otherwise there is no effect.

The control word has 32 bits. Bit 2 of the control word is the interrupt enable bit (`CTL_IE` is defined as 0x00000002) and bit 1 is the device ready bit (`CTL_RDY` is defined as 0x00000001). All other bits in the control word are currently reserved and read as zero. Writing any of the other bits of the control word is undefined. The interrupt enable bit in the clock device control word is initially unset.

11.2.2 Interrupts

The standard clock device is connected to the hardware interrupt line specified by the '`clockdeviceirq`' option, which must be a number corresponding to an interrupt line reserved for use by hardware (2 through 7). See the "Summary of configuration options" section of the "Customizing" chapter for more information. The clock requests an interrupt whenever the clock is in the ready state and the interrupt enable bit on the control word is set.

The '`clockintr`' option gives the frequency of clock interrupts in nanoseconds of simulated time. See the "Summary of configuration options" section of the "Customizing" chapter for more information.

11.2.3 Real vs. simulated time

Real time is obtained from the host's `gettimeofday(2)` system call, so it should be close to the host's view of the current time. No sophisticated algorithms are used to calibrate the real time clock, so it will drift a little.

The speed of simulated time is determined by the '`realtime`', '`timeratio`', and '`clockintr`' options. See the "Summary of configuration options" section of the "Customizing" chapter for more information. Increasing the speed of simulated time will most likely make the simulation run more slowly because it will increase the average number of system calls per instruction.

11.3 Halt device

This section documents the halt device. It is provided so that simulated operating systems can stop the simulator in a controlled manner, without having to rely on specific instructions or exceptional conditions. The halt device is enabled or disabled with the '`haltdevice`' option.

11.3.1 Memory-mapped registers

The halt device has 1 register, configured to be mapped into memory at physical address 0x01010024. The following table defines the layout of the memory-mapped halt device register:

`offset 0x00`
: control word

Writing a non-zero value to the halt device control word halts the simulation. Writing zero has no effect. The control word is always read as zero.

11.4 DECstation 5000/200-compatible devices

VMIPS contains partial support for the built-in memory-mapped devices on the motherboard of the Digital Equipment DECstation 5000/200. This support should be considered "beta" quality in this release. All the DECstation-compatible devices are disabled by default; see the options starting with '`dec`' in the Customizing chapter for information about how to turn them on.

The following devices are supported:

DECstation 5000/200's Dallas Semiconductor DS1287-based real-time clock (RTC) chip (use '`-o decrtc`' to enable; mapped into memory at physical address 0x1fe80000)

DECstation 5000/200's Control/Status register (use '`-o deccsr`' to enable; mapped into memory at physical address 0x1ff00000)

DECstation 5000/200's Check Syndrome (CHKSYN) and Error Address (ERRADR) registers (use '`-o decstat`' to enable; mapped into memory at physical addresses 0x1fd00000 and 0x1fd80000 respectively)

DECstation 5000/200's DZ11-based serial chip (use '`-o decserial`' to enable; mapped into memory at physical address 0x1fe00000; note: interrupt-driven I/O does not work well with this device – use polling instead)

In the future, we plan to finish these device emulations, test them more thoroughly, and document them more completely in this manual.

The devices are documented thoroughly in the document: "DECstation 5000/200 KN02 System Module Functional Specification", published August 1990 by Digital Equipment Corporation. You may be able to find it by doing a web search.

12 Monitor

The main use of the ROM monitor is to load executables into RAM and run them. Currently, only COFF executables are supported.

12.1 Monitor Commands

The ROM monitor is structured around a simple command processor that prompts the user to enter a command, reads a command from the first serial line, and executes it. This loop continues until you boot a kernel (which may or may not ever return control to the monitor) or halt the system.

The following are commands supported by the ROM monitor. The first word is the command name; other words are arguments that you specify. Arguments in brackets are optional.

`rx` Receive file via XModem protocol over serial device into RAM.

`quit` Halt the boot monitor.

`help` Print a brief list of commands.

`peek addr nwords`
Read and print 32-bit value at address addr.

`poke addr value`
Write the given 32-bit value to address addr.

`info` Print information about a COFF program which was previously loaded.

`boot [args]`
Run a program which has previously been loaded by calling its entrypoint.

`rom addr nwords`
Prepare to boot a file from ROM address addr, having length nwords (rather than RAM).

`printenv` Print the boot environment.

`setenv varname value`
Set the boot environment variable named varname to value.

`unsetenv varname`
Delete the boot environment variable named varname.

`call addr [args]`
Call procedure at addr.

12.2 Loading Your Program

The main facility provided for transferring a file to the boot monitor is the XModem serial file transfer protocol. On the host side, you can use a program such as `sx` to send files via XModem.

COFF executable files are loaded to `BASE_ADDR` (0x80000000 by default), and then moved to their proper location at boot time.

Instead of transferring a file to the boot monitor, you can put the file into a separate area of ROM and load it using the `rom` command.

Yet another alternative is to use the `catrom` script, which will build you a ROM image containing the boot monitor and a COFF executable of your choice, and patch the boot monitor to load it automatically. A boot monitor thus patched will not require user interaction and is very often the quickest way to test out a new program. A dummy `argv[0]` will always be passed to the booted program in this mode.

12.3 Booting Your Loaded Program

For the boot and call commands, the code being called is assumed to have the same prototype as main. Providing arguments is optional. Any arguments provided are passed to the procedure in argc/argv form. It is a good idea always to pass at least one argument (`argv[0]`) to the booted program.

The boot-time exception vectors (BEV bit in CP0 Status register) will typically be disabled upon booting a program and reenabled if that program returns control to the monitor.

12.4 Other Monitor Facilities

The boot monitor contains a set of basic library routines and a jump table that mimics to some extent the one found in the DECstation ROM.

Exceptions caught in the boot monitor will generally cause the monitor to print a message and halt.

12.5 Hacking the Monitor

Whether to use the SPIM-compatible console device or the DECstation-compatible serial device is determined at the time you build the boot monitor. Either the `SPIM_CONSOLE` or the `DZ_CONSOLE` macro at the top of '`serial.c`' must be set to 1.

The monitor directory contains a '`README`' file with more information about the structure of the code.

If you are trying to get standalone programs working before you have a working C library or proper serial drivers, you may get some mileage out of the standalone library ('`lib.c`' and '`lib.h`') routines used by the boot monitor.

The boot monitor directory also contains some tests, and an example ('`loadtest.c`') program that you may be able to compile and boot via the monitor. Of course, the monitor code is under the same license as VMIPS.

13 Extending

This chapter is intended to be a hacker's guide to adding or modifying VMIPS functionality.

13.1 Road map to the VMIPS source code

This section is intended to help interested persons find various things in the VMIPS source code, and get a general idea of how the various software modules are structured.

The processing of command-line options and of options in your '`.vmipsrc`' is directed by routines in '`options.cc`' and in class `Options`. The default options and the option documentation are found in '`optiontbl.h`'.

The memory mapping unit has a high-level interface to the rest of the code, which is defined in '`mapper.cc`' and '`mapper.h`', and in class `Mapper`. The memory mapping unit uses a bunch of low-level data structures, which are defined in '`range.cc`' and '`range.h`', in class `Range`. This is meant to be logically and physically separate from the TLB, which is implemented as part of the system control coprocessor. The actual chunks of host virtual memory which are used for the virtual machine's physical memory are encapsulated in class `MemoryModule`, which is implemented in '`memorymodule.h`'.

The memory mapping unit is responsible for handling memory accesses via the cache, or without using the cache, as appropriate. The cache is defined in class `Cache`, also in '`mapper.cc`' and '`mapper.h`'. Two instances of this class exist as the `icache` and `dcache` members of class `Mapper`.

The system control coprocessor (MIPS coprocessor zero) and the TLB are implemented in '`cpzero.cc`' and '`cpzero.h`', as class `CPZero`. The structure of TLB entries is defined in '`tlbentry.h`', and constants related to the register set of MIPS coprocessor zero are defined in '`cpzeroreg.h`'.

The CPU (class `CPU`) and the default exception handling behavior are implemented in '`cpu.cc`' and '`cpu.h`'. Exception handling behavior is an interface described by class DeviceExc (in '`deviceexc.h`'); this class provides for the `exception` instance method and its implementations in class `CPU` and class `Debug`. Constants for the different kinds of exceptions which are implemented by MIPS processors are defined in '`excnames.h`'.

The disassembler uses code from GNU libopcodes (part of GNU Binutils); it is located in the '`libopcodes_mips`' directory. The high-level C++ interface to the disassembler is in '`stub-dis.cc`'.

The GNU debugger interface is separated into a high-level part (which deals with the various debugger requests) in '`debug.cc`' and '`debug.h`', and a low-level part (which assembles and disassembles the GDB remote serial protocol packets), in '`remotegdb.cc`' and '`remotegdb.h`'.

A few parts of the VMIPS system have a central procedure which needs to be run periodically in a loop in order to update the part of the simulation that they are responsible for. These parts typically have instance methods named `step()`. The `CPU` class, for example, fetches, decodes, and executes one instruction each time its `step()` function is called.

The `vmips` class, implemented in '`vmips.cc`', is used to tie all the components of the system together. This class, and specifically its `run()` member function, is responsible for setting up and configuring all system components and calling the `step()` member function(s). The vmips class is not a very smart or flexible configuration mechanism; it eventually ought to be replaced with a configuration language of some sort.

The simulator's idea of time is managed by classes in '`clock.cc`' and '`clock.h`'. VMIPS programs gain access to the simulated clock by using the memory-mapped clock device, which is implemented in files '`clockdev.cc`' and '`clockdev.h`', and whose register map is available in '`clockreg.h`'. The clock manages tasks, which are basically function objects that can be cancelled or fire at a later time. Tasks are defined in '`task.h`'.

VMIPS provides standard error-reporting functions, which your code can use. They are defined in '`error.cc`' and '`error.h`'.

Some of VMIPS's simulated devices share common semantics for control register bits, constants for which are defined in '`devreg.h`'.

VMIPS provides a halt device, which can halt the machine even when the options such as '`haltbreak`' are turned off. It is implemented in '`haltdev.cc`' and '`haltdev.h`', and its register map is defined in '`haltreg.h`'.

The SPIM-compatible console device (implemented in '`spimconsole.cc`' and '`spimconsole.h`', with a register map in '`spimconsreg.h`') is based on a generic terminal controller, which is implemented in '`terminalcontroller.cc`' and '`terminalcontroller.h`'.

The various DECstation-compatible devices are implemented in the files whose names start with '`dec`'.

The ROM monitor code is in the directory '`sample_code/xmboot`'. See the Monitor chapter for more details.

The manual, and any random bits of hacking information which have not yet been incorporated into the manual, are in the directory '`doc`'.

The VMIPS automated regression test suite is in the directory '`test_code`'. Some interesting sample code, including the canned ROM setup code used to build ROM files out of C programs for the test suite, is in the directory '`sample_code`'.

Various scripts used by the maintainers to help maintain the code are in the directory '`utils`'.

VMIPS provides a simple front-end to GNU MIPS cross-compilation tools, called Vmipstool. Its implementation is in the file '`vmipstool.cc`'; it shares a little bit of options- and error-handling code with VMIPS.

Interfaces to the host system's C++ standard library are included in '`sysinclude.h`'. '`wipe.h`' is a template utility function used for deleting all the objects contained in standard C++ containers.

Please read the rest of this chapter for information about the rest of the files in the VMIPS source directory.

13.2 Endianness issues

When you are making extensions to VMIPS, it is important not to assume that your host processor is little-endian (or to assume that it is big-endian). One of the first things that VMIPS does when it starts is to determine the endianness of the host processor. The endianness of the VMIPS target processor is determined by the '`bigendian`' command-line option. Your extension can query the `machine->host_bigendian` flag to determine whether the host processor is big-endian or not, and it can query the '`bigendian`' command-line option, via the public interface of class `Options`, to determine whether the VMIPS target processor is big-endian or not.

The physical memory system (class `Mapper`) defines some convenience methods which you can use. You can call the `swap_word()` or `swap_halfword()` methods of class `Mapper`, or the wrapper functions `host_to_mips_word()` and `mips_to_host_word()`, to do endianness translation between the host machine and target machine, when necessary.

When you define memory-mapped devices, you should return data to the Mapper in host endianness. It is recommended that memory-mapped devices also store their data in host endianness, unless there is a good reason.

13.3 Memory-Mapped Devices

Memory-mapped devices must inherit from class `DeviceMap`, which is defined in the files '`devicemap.cc`' and '`devicemap.h`' in the VMIPS source directory.

Memory-mapped devices must have a constructor and a destructor. The constructor must call the DeviceMap constructor with a single parameter, called *extent*. It should be equal to the number of bytes which are mapped into the processor's memory; this figure must be a multiple of 4. The device must also override the following abstract methods:

```
uint32 fetch_word(uint32 offset, int mode, DeviceExc *client);
uint32 store_word(uint32 offset, uint32 data, DeviceExc *client);
```

The meanings of the parameters are as follows:

offset Byte offset from the beginning of the device's memory-mapped region that is being read or written. The width of the read (fetch) or write (store) is either a word (4 bytes), halfword (2 bytes), or a single byte, depending on the call. Since this value is a byte offset, if you want to figure out which word of your device is being accessed, you should divide it by 4.

mode This tells you whether the memory access is a data load (`DATALOAD`), data store (`DATASTORE`), or instruction fetch (`INSTFETCH`). These constants are defined in '`accesstypes.h`'. For narrow (< 1 word) fetches, the mode is always `DATALOAD`. For stores, the mode is always `DATASTORE`. The only case in which this is ambiguous is for the `fetch_word` case, where mode may be either `DATALOAD` or `INSTFETCH`. Most devices do not need to bother with the mode, except when there is an illegal access. See the section on exception behavior, below.

client Every memory access is requested by a client, which is responsible for handling any exceptions which may arise. Any component of the VMIPS system which may access memory must either inherit from class `DeviceExc` (i.e., "a device

which may handle exceptions"), or have a pointer to a device which does. See the section on exception behavior, below.

data When the client is storing a value, you will receive the value as the *data* parameter.

13.4 Exception behavior

Whenever there is an exception, the device must make the call

```
client->exception(type, mode);
```

whose precise prototype is defined in '`deviceexc.h`'.

Type must be one of the standard MIPS exception codes, which are defined in '`cpzeroreg.h`', and elsewhere in this manual. *Mode* is the mode of the memory access; see the table entry for *mode* above.

Please note that you should not call the `exception` method in order to generate a hardware interrupt (i.e., the Interrupt exception). Interrupts are managed by class `IntCtrl`, and your device should call the `assertInt` function to generate them. See the "Interrupt-generating devices" section for more details on what you should do. If you are curious about the inner workings of the interrupt controller, you can read its source in '`intctrl.cc`' and '`intctrl.h`'.

13.4.1 Coprocessors

If your device is part of a MIPS coprocessor, you should pass a third argument to the `client->exception()` call, which is the number of the coprocessor; it may meaningfully be 0, 1, 2, or 3. Ordinarily, that is to say in situations not involving coprocessors, this parameter defaults to -1 and does not need to be specified explicitly.

Coprocessor 0 is the MIPS system control coprocessor, responsible for TLB and paging management. It is implemented as class `CPZero` in '`cpzero.cc`' and '`cpzero.h`'. It has 16 registers, each of which has some read-only bits and some read/write bits. Extension code should not attempt to misrepresent itself as being coprocessor zero without a good reason.

One of the jobs of the `CPZero` class is to ensure that attempts to write to these registers are only allowed to write to the bits which are writable, so if you are interested in implementing read-only and read/write registers in your virtual hardware, look through '`cpzero.cc`' for *read_masks* and *write_masks*.

Coprocessor 1 is the floating point coprocessor, but it is not implemented. It may, however, be implemented in the future. Volunteers to begin such a task would be more than welcome.

The default behavior of MIPS coprocessors 1, 2, and 3 in the VMIPS system is to assume that they are not connected to the system and that accesses to them should therefore trigger the `CpU` (Coprocessor Unusable) exception.

13.5 Mapping memory-mapped devices

You can map each memory-mapped device at a single physical address in the machine's memory. The instantiation process is as follows: Assume that `TestDev` is a memory-mapped device class which derives from class `DeviceMap`, that *testdev* is an instance of class `TestDev`, and that *physmem* is a `Mapper` (memory manager) object.

```
/* Test device at base phys addr 0x01000000 */
testdev = new TestDev();
physmem->add_device_mapping(testdev, 0x01000000);
```

A device should have a single base address; you should have a single call to the Mapper instance method `add_device_mapping(device, addr)` to set it. *device* is an instance of a class deriving from class `DeviceMap`. *addr* is the physical address where you want the device to appear in memory.

This code is generally executed as part of the `vmips->run()` method in '`vmips.cc`'. Look there for more information and some examples of what to do.

13.6 Interrupt-generating devices

VMIPS provides support for virtual devices that generate hardware interrupts to communicate with the processor. These virtual devices should inherit from class `DeviceInt` (defined in '`deviceint.h`'). This section outlines some information about how to write such virtual devices.

13.6.1 Connecting devices to the interrupt controller

There are 8 interrupt lines in the R3000/R3000A, 6 of which (7..2) are hardware interrupts (readable by software), and the other 2 of which (1..0) are software interrupts (readable/writable by software).

The class `IntCtrl` instance method `connectLine(irq, device)` is used in '`vmips.cc`' to notify the interrupt controller and the device that the interrupt line specified by *irq* is connected to *device*. *irq* must be one of the hardware interrupt constants defined in '`deviceint.h`' and *device* must be an object of a class deriving from `DeviceInt`.

13.6.2 Generating and cancelling interrupt requests

The class `DeviceInt` instance method `assertInt(irq)` is used to request an interrupt from the processor. Your device should only request interrupts that have previously been connected to it using the interrupt controller (see above). Your device may share an interrupt request line with another device. In practical terms, asserting an interrupt request line will cause a trap to the general exception vector before the next instruction. If your device asserts an interrupt, it stays asserted until it is explicitly de-asserted.

The instance method `deassertInt(irq)` will turn off the interrupt request for your device; this should be done when the condition that caused the device to request an interrupt has become satisfied. Note that this does not necessarily imply that the interrupt request for the processor will be turned off, as there may be another device trying to use that interrupt request line.

For both calls, the IRQ parameter must be one of the hardware interrupt constants defined in '`deviceint.h`'. It is not a good idea to use the general `exception()` method to cause interrupt exceptions, because this could cause excess interrupts to be generated.

The place where you should make these calls and do these checks is when your device's code is called through the `periodic()` callback. Your device will get `periodic()` calls fairly often.

13.6.3 Software interrupts vs. hardware interrupts

Two of the interrupt lines (IRQ 0 and 1) are reserved for software use. Only the interrupts which are not reserved for software use (IRQ 2 through 7) may be triggered by VMIPS devices.

13.6.4 Turning interrupts off and on

There is a global Interrupt Enable bit for the whole system; this is the IEc (Interrupt Enable (current)) bit, bit 0 (mask 0x001) of the Status register (coprocessor zero register 12). If this bit is turned off, no interrupt will be triggered. Be sure to turn on your Interrupt Enable and Interrupt Mask (below) bits when you are testing your new interrupt-generating device.

Additionally, bits 15 - 8 (mask 0x0ff00) of the Status register are individual Interrupt Mask bits. Each bit represents a global interrupt enable/disable bit for the entire system per interrupt-request line. For example, if you turn off bit 10 of this register (mask 0x0400), the IRQ2 line will be disabled for the whole system.

Finally, it is not uncommon for individual devices to have their own interrupt enable/disable bits that you can set or clear. See the documentation for each individual device for more information.

13.7 Error reporting

When your code needs to emit warning or error messages, we recommend you use the following functions from '`error.cc`':

```
void error(const char *msg, ...) throw();
void fatal_error(const char *msg, ...) throw();
void warning(const char *msg, ...) throw();
```

`fatal_error` will result in a call to abort() after printing the error message. All of these functions will print a newline after MSG.

13.8 Weird things

13.8.1 Branch on Coprocessor Zero True/False

These instructions are not supposed to cause reserved instruction exceptions, even though the behavior of BC0F and BC0T instructions on MIPS-1 machines is not specified in most canonical references.

On some DEC MIPS machines, the coprocessor 0 condition bit (which BC0F and BC0T test) is wired to the external write-buffer-empty bit; that is, when all stores have completed, the write buffer becomes empty, and the bit goes to true. This makes it possible for a hacker to write the line '`1: bc0f 1b`' and thereby loop until the write buffer is empty. However, this is not true of all DECstations, or of the Sony NEWS 3400.

The coprocessor zero condition bit has an entirely different use on the R4400 and compatible processors; it is used to tell when you got a cache hit with a CACHE operation. The R10000 also implements this condition, but the bit is not wired to the coprocessor zero condition.

Since VMIPS does not support CACHE operations, and does not have a write buffer, VMIPS emulates the case where the CpCond bit for CP0 is always TRUE, i.e., applications that look for the writebuffer will find that it is always empty.

14 Test Suite

VMIPS has a test suite with a small number of regression tests. It uses the DejaGNU test framework, which is written in the Expect language. Expect was written by Don Libes, and is a dialect of Tcl, the Tool Command Language by Ousterhout et al.

14.1 How to run the test suite

In order to run the test suite or add tests to it, you will need to have Expect and DejaGNU installed. Any version of DejaGNU later than 1.3 should work fine. We have mostly used Expect version 5.32 or later; older versions may have bugs that may lead to unexpected test failures.

The next step is to configure and build VMIPS from source. This is important, because currently, you can only run tests on a freshly built copy of VMIPS; no provision exists for testing a previously installed copy of VMIPS. Simply run '`./configure`' and '`make`' from the top level source directory to build VMIPS. See the "Installation" section of the manual for more details.

To run the test suite, run '`make check`' from the top level source directory. This will invoke the DejaGNU `runtest` command. It will take a minute or so, and then you will get a count of tests that passed and failed. If you want to see a more comprehensive listing from `runtest`, pass it the appropriate options through the Makefile, by typing, for example: '`make RUNTESTFLAGS=--verbose check`'.

You can also run the test suite by changing into the '`test_code`' directory in the source tree and running '`runtest --tool vmips`'.

As a general rule, no tests should fail in a released version of VMIPS; however, CVS builds may have test failures from time to time.

14.2 Test suite frameworks

VMIPS has two comprehensive testing frameworks: the '`regcheck`' framework, whose tests live in the '`vmips.regcheck`' directory, and the '`outcheck`' framework, whose tests live in the '`vmips.outcheck`' directory. The former looks at the final values of registers after a test case is run, and the latter looks at the output that VMIPS prints out when a test case is run.

Each of these main testing frameworks has its own '`.exp`' file that runs it. You can easily run the subset of the test suite controlled by a given '`.exp`' file, by passing its name on the `runtest` command line. For example, if you want to run all the '`regcheck`' tests, you would type: '`runtest --tool vmips regcheck.exp`'.

There are a few test cases that do not use either of these frameworks, because they have special requirements of some kind or are otherwise unique in some inconvenient way. These test cases have their own Expect drivers ('`.exp`' files) and live in the '`vmips.misc-tests`' directory.

In addition, each test case is defined by a '`.par`' file that contains the parameters of the test. If you only want to run a single test from among the '`regcheck`' tests, specify its '`.par`' file after an equal sign. For example, you might type: '`runtest --tool vmips regcheck.exp=mumble.par`' to run only the '`mumble.par`' '`regcheck`' test.

14.3 How to add a test to the test suite

To write a new test case, first decide whether it is easier to have your test case print out something or to look at the '`haltdumpcpu`' option's output to verify it. This will tell you whether it should be an '`outcheck`' test (if it prints out something) or a '`regcheck`' case (if you look at the register values). Then write up a '`.par`' file for the test case — the best way to learn how to do this, for the time being, is to examine the examples in the '`test_code`' directory. Then move the '`.par`' file and the test code (assembly or C) to the appropriate subdirectory for the framework you chose, '`vmips.regcheck`' or '`vmips.outcheck`'. Now you can try to make sure that it passes (or fails, as the case may be), by running '`runtest --tool vmips`' on it.

Also, note that you shouldn't add another test to '`vmips.misc-tests`' unless you can't find a way to fit it into any of the existing testing frameworks. Adding a test to '`vmips.misc-tests`' is tricky – you may be able to make progress by looking at the other '`.exp`' files in that directory.

14.4 Common problems

You can get screenfuls of '`endian_option was not set`' errors if you run `runtest` without first making a DejaGNU '`site.exp`' configuration file first. To make a '`site.exp`' file, just run '`make site.exp`' in the '`test_code`' directory.

14.5 Additional test cases needed

Test cases should probably be added for the following categories of VMIPS behaviors. (This list is from 2002; it is probably a good start, but it may be out of date in the sense that it is unlikely to be exhaustive.)

Vmipstool.

The debugger interface.

Exceptions: exception prioritizing, non-boot-time exception vectors, all the TLB exceptions, Unimplemented Coprocessor exceptions, Reserved Instruction exceptions, and exceptions due to PC address translation.

Check that the TLB does the right thing when you have a dirty entry vs. a non-dirty one.

dumpcpu, dumpcp0, haltdumpcp0, bootmsg options.

tilde_expand() where it's not your own home directory.

Address translations in KSEG0 or KSEG2 or KUSEG or User mode.

Appendix A Installation

VMIPS uses the GNU Autoconf/Automake system for configuration management. This provides the familiar `configure` shell script interface for setting configuration variables before compiling VMIPS. This means that the traditional '`./configure; make; make install`' sequence should work. For more information about the special options that VMIPS `configure` accepts, read on, or give the '`--help`' option to `configure` for an abridged version.

A.1 Prerequisites

The VMIPS build process assumes that you have a C++ compiler installed on the host machine. Any reasonably recent, ISO C++-compliant compiler with working template function handling should work. Some ancient compilers, such as the system compiler on Red Hat Linux 6.x systems, will not work. `configure` contains checks for a few known compiler problems which will prevent VMIPS from working, and will print an error message if it detects such a problem.

If you want to build any of the sample code which is included with the VMIPS source distribution, you must have a full set of GNU MIPS cross compilation tools installed when you configure VMIPS. You will need to tell `configure` the configuration prefix you used to install the MIPS tools, by specifying it as the value to the '`--with-mips`' argument. For example, if your MIPS cross compiler is '`/opt/mipsTools/bin/mips-dec-ultrix4.3-gcc`', then you should specify '`--with-mips=/opt/mipsTools`' on the `configure` command line. Additionally, you will also need to tell `configure` the target you used to configure the MIPS cross tools, by specifying it as the value to the '`--target`' argument (see below). For a concise summary of how to build the necessary MIPS cross tools, read "Building MIPS Cross Tools", below.

If you want to run the test suite, you must additionally have Expect and DejaGNU installed (any version published since 2000 should be fine). Once VMIPS is compiled, you can type '`make check`' to run the test suite.

A.2 Building from CVS

If you retrieved your sources from the CVS repository, you will need Automake version 1.11.1 or later, Autoconf version 2.68 or later, and libtool 2.2.10 or later. You will need perl 5 to build the documentation. Your distribution will be missing many important files, including `configure`. To generate these, run `utils/bootstrap`. To automatically run configure once it has been generated, you can run '`utils/bootstrap -c CONFIGURE-ARGS`'.

A.3 Options that configure supports

If you want to build VMIPS with particular compiler optimizations or with debug symbols, see the example in '`INSTALL`' describing how you can set *CFLAGS*. You will want to do the same for *CXXFLAGS*. By default, if your system compiler is GNU gcc, VMIPS will be built using '`-g -O2`'.

Some of the interesting options that '`configure`' supports are as follows:

‘`--target=T`’
: Specify the target used to configure your MIPS cross tools. ‘`T`’ must match the value of the ‘`--target`’ option provided to GNU Binutils `configure`.

‘`--with-mips=MDIR`’
: Specify installation prefix of MIPS cross tools (default MDIR = /opt/mips). If you do not have MIPS cross tools, the VMIPS test suite will not be available and sample code will not be built. If you want `configure` to ignore your MIPS cross tools, you can indicate this by specifying ‘`--without-mips`’.

  This flag is used to set the default value of the vmipsrc ‘`mipstoolprefix`’ option. You can always edit the ‘`mipstoolprefix`’ option setting in ‘`/etc/vmipsrc`’ after installation if you want to change its value.

‘`--with-mips-bin=DIR`’
: Specify path to MIPS cross tools’ executables (default MDIR/bin). This option may be useful as an override if –with-mips isn’t working for you, but beware — it is rarely tested by the developers.

‘`--with-mips-endianness=VAL`’
: Specify the default endianness of the VMIPS simulated machine, which must match the MIPS cross tools target’s endianness. VAL may be specified as `big` or `little`. If you have installed MIPS cross tools, it is best to let configure guess this (which it will do by running ‘`mips-objdump -i`’), unless you have reason to believe it is guessing wrong, because if you get it wrong, vmipstool may compile ROMs that do not run correctly under vmips. If you are configuring without MIPS cross tools installed, this will default to little-endian.

  This flag is used to set the default value of the vmipsrc ‘`bigendian`’ option. You can always edit the ‘`bigendian`’ option setting in ‘`/etc/vmipsrc`’ after installation if you want to change this value.

‘`--enable-profiling`’
: Include (default=do not include) profiling instrumentation in the VMIPS binary. This is only interesting if you are maintaining VMIPS and trying to figure out why it is running more slowly than it should be. When we last tried it, this option didn’t seem to work very well on Mac OS X.

A.4 Post-Installation Setup

If you are an end-user with a binary package for VMIPS and a MIPS cross compiler, you will probably want to make Vmipstool use the cross compiler when you run commands such as `vmipstool --compile`.

You should edit your ‘`/etc/vmipsrc`’ or ‘`~/.vmipsrc`’ file and change the ‘`bigendian`’ and ‘`mipstoolprefix`’ options to correspond to the installed MIPS cross tools. (See the ‘Customizing’ chapter for more information on the syntax of these options.) Then, test it by trying to compile a C file by running `vmipstool --compile -c foo.c`. You should get an object file (‘`foo.o`’) of the right endianness and object format; you can check this using the `file` command on most Unix systems.

A.5 Packaging VMIPS

If you are a system integrator or distributor who is building a package for VMIPS intended for distribution, you may be able to start by looking at the RPM vmips.spec file or the Debian dpkg packaging files included in the source distribution.

Your VMIPS package need not require a set of MIPS cross tools either at the build or install stage. Starting with VMIPS 1.2, it is perfectly possible to build VMIPS without a cross-compiler, cross-assembler, or cross-linker. Vmipstool will not be very useful without cross-tools, but it will build; however, an end-user can install cross-tools and edit the system-wide '`vmipsrc`' file to make the '`mipstoolprefix`' option value contain their location.

The VMIPS binary package should probably include the following files:

the two executables `vmips` and `vmipstool`

the two man pages '`vmips.1`' and '`vmipstool.1`'

the VMIPS Programmer's Manual in some format (for example, the Info file '`vmips.info`')

the default GNU linker script used by vmipstool, '`ld.script`'

the system-wide configuration file '`/etc/vmipsrc`'

the assembler convenience header file '`asm_regnames.h`'

Help keep VMIPS free! As VMIPS is released under the GNU General Public License, please make an effort to distribute sources (or at least, post a link to the sources) if you distribute binaries or binary packages. Thanks!

A.6 Building MIPS Cross Tools

First decide on an installation prefix. The following examples will use the prefix '`/opt/mips`', as above, which is the default place that the VMIPS `configure` script looks for them; you can however use any prefix you wish.

Download a copy of Binutils, from any GNU mirror, or from the URL:

```
ftp://sources.redhat.com/pub/binutils/releases
```

The most recently-tested version is 2.20.1.

Build binutils by running the following commands. We recommend `--disable-nls` because some versions do not build correctly with NLS (linking against '`libopcodes.a`' results in unresolved symbols.)

```
./configure --target=mipsel-ecoff --prefix=/opt/mips \
 --disable-nls --disable-shared
make
make install install-info
```

Download a copy of the GNU Compiler Collection (`gcc`) from any GNU mirror, or from the URL:

```
ftp://gcc.gnu.org/pub/gcc/releases
```

Our examples assume that you want to use the ECOFF binary format, so we recommend you get gcc version 3.0.4. If you would prefer to use the ELF binary format, pretty much any recent version of gcc will work, but note that you will need to pass `--target=mips-elf`

instead of `--target=mipsel-ecoff` when configuring both binutils and gcc. We have most recently tested version 4.2.4 with the ELF format.

You can read the documentation for building the compiler by pointing your World-Wide Web browser at `http://gcc.gnu.org/install`. When you encounter difficulties, you should consider consulting the documentation for building the compiler, because it is more complete than the following summary.

1. Unpack the sources. Let's say you unpack them in '`/usr/build`', creating the directory '`/usr/build/gcc-3.0.4`'.
2. Create the build directory '`/usr/build/gcc-mips-build`'.
3. First, add the directory '`/opt/mips/bin`' (where you just installed Binutils) to your path, so that the compiler configuration process can find your MIPS-targetted assembler and linker.
4. Configure the compiler. Change to the directory '`/usr/build/gcc-mips-build`' and issue the following command. (The back-slash characters represent the usual Unix shell convention of continuing a command on the following line, and are inserted for typesetting purposes.)

   ```
   ../gcc-3.0.4/configure --target=mipsel-ecoff \
    --prefix=/opt/mips --with-gnu-as --with-gnu-ld \
    --disable-threads --disable-nls --disable-shared \
    --enable-languages=c
   ```

5. If the configuration step fails, make sure you have a working native compiler, and/or try a different version of gcc. Otherwise, proceed to compile the compiler:

   ```
   make -k MAKE='make -k TARGET_LIBGCC2_CFLAGS=-Dinhibit_libc' cross
   make -k LANGUAGES=c install
   ```

   The reason '`make -k`' is required is because some parts of the gcc toolkit may fail to build, but the compiler itself may be OK.

   The '`-Dinhibit_libc`' option is required when you are building the compiler in the absence of a MIPS C library, as is often the case with VMIPS users.

   Do not be alarmed by errors in building or installing the compiler; the cross compiler install interface is less than polished.
6. You should be able to use the newly-installed compiler to compile (but not link) a program that does not use any C library functions. If this works, you should be able to use the cross tools you have just built for VMIPS.
7. If you want to use the GNU debugger (GDB) to debug MIPS programs running on VMIPS, you can build that now.
   1. Download a copy of the GNU debugger from any GNU mirror, or from the URL:

      `ftp://ftp.gnu.org/pub/gnu/gdb/`

      We recommend version 6.0 or later. Download the file '`gdb-6.0.tar.gz`'.
   2. Unpack the file and change to the directory '`gdb-6.0`'.
   3. Type the following commands to configure and build GDB:

      ```
      ./configure --prefix=/opt/mips --target=mipsel-ecoff
      make
      make install install-info
      ```

4. You can now use the newly installed '`mipsel-ecoff-gdb`' to debug programs with VMIPS, as described in the "Debugging" section of the manual.

8. If you want to build a MIPS C library, you can also do that now, but it is not strictly required for many useful VMIPS tasks.

   Here is how to build the uClibc C library for use with VMIPS:

   As noted in the uClibc INSTALL file, you will need Linux kernel sources. Just pick a recent version of Linux 2.4; you can download it from `http://www.kernel.org` or one of its mirrors, if you don't have it handy. You will need to configure (but not build) the Linux kernel for MIPS. Here's how:

   Edit '`linux/Makefile`', setting *ARCH* to '`mips`' and setting *CROSS_COMPILE* to a value that corresponds to the path where your MIPS cross compiler is installed, for example: '`/opt/mips-elf/bin/mips-elf-`' (this is just like '`mipstoolprefix`' in your '`vmipsrc`' file).

   Copy '`arch/mips/defconfig`' to '`.config`'.

   Run '`make oldconfig`' and '`make dep`'.

   Next, download uClibc from `http://www.uclibc.org`, and unpack it next to the Linux kernel sources. The last version we tested was 0.9.29. Read the '`INSTALL`' file in that distribution.

   When you configure uClibc with '`make config`', be sure to pick '`mips`' as your Target Architecture, and '`Generic (MIPS I)`' as your Target Processor. Be sure to pick the correct endianness (that is, the one which corresponds to the default endianness of your cross tools.) You should be sure to answer yes to 'Target CPU has a memory management unit (MMU)' and no to 'Enable floating point number support', because current versions of VMIPS do not include floating-point support. If you are intending to use uClibc to build ROMs, you will probably want to turn off position-independent code and shared library support. Turn on only those other features of uClibc as you expect you will need.

   Create a new directory into which uClibc will be installed. This will be your *PREFIX*.

   Run '`make CROSS=/opt/mips-elf/bin/mips-elf-`' to build uClibc. For *CROSS*, you should use the same value as you used for *CROSS_COMPILE* in the Linux Makefile, above.

   Run '`make CROSS=... PREFIX=... RUNTIME_PREFIX=/ DEVEL_PREFIX=/usr/ install`' to install it, specifying the same *CROSS* value and the name of the directory created earlier as *PREFIX*. In the directory you specified for *PREFIX*, you will now have '`usr/include`' and '`usr/lib`' subdirectories. You will now want to rebuild GCC, specifying these directories using the '`--with-headers`' and '`--with-libs`' options to the GCC '`configure`' script, respectively. This will cause the directories to be copied to

   prefix/*target-triplet*/lib (for libraries)

   prefix/*target-triplet*/sys-include (for includes)

Appendix B Reporting Bugs

We are always interested in hearing about VMIPS bugs. Please send mail to `vmips@dgate.org` and tell us about them. Please include at least the following information:

- your operating system
- your host processor type
- your C++ and C compiler type and version
- the version of VMIPS you are using
- how you configured VMIPS
- how to trigger the bug
- what you expected to see
- how what you saw differed from what you expected to see
- how you think it could be fixed (send a patch if you have one)

Appendix C Future Directions

The following is a list of things we would like to add to VMIPS. Please get in touch with us if you think you would be willing to help.

- Cache and DMA support in the memory subsystem.
- A just-in-time translator from MIPS machine code to the host's machine code. X86 and PowerPC would probably be the first hosts to target.
- Add meaningful bindings to an extension language such as Tcl/Tk. In particular, it should be possible to read or write option values, modify the configuration of the machine (by attaching or removing devices, for example), and type commands interactively to VMIPS. There should be some provision for refreshing cached option values if the extension language changes them.
- Modularize the devices, and potentially even the CPU and memory mapper using a shared-library plug-in model.
- Make it so that the debugger can be attached and detached at any time, without the user having had to think of it beforehand and supply the debug option. Delay the debugging interface initialization until a connection is received?
- Graphical user interface with register and memory display and editing, support for loading, running, and stepping programs, and setting breakpoints.
- Extend the current executable loader (in vmips/exeloader.cc) to support:

  loading ELF-format files

  loading files into user space (This will require some sort of built-in TLB-miss handling.)
- Support for non-English languages using NLS.
- A reasonable FPU emulator.
- ROM monitor network booting support.
- Full MIPS32 support.
- Checkpoint and restart of simulations.
- Develop a patch for gas to support software register names. gas supports $sp and $gp but not, say, $t0.
- Consolidate some of the .h files that just contain huge lists of useful constants.

Appendix D Release History

vmips-1.5 was released on 17 November 2014. User-visible changes in version 1.5 (since version 1.4.1):

- New features/improvements:
- VMIPS now includes a basic direct-mapped cache simulation. The cache isolation and cache swap bits in the CP0 Status register are now honored.
- The boot monitor distributed with VMIPS now sets up a dummy `argv[0]` value for the loaded program. Also, it halts by entering an infinite loop rather than attempting to execute a '`break`' instruction when it encounters an unexpected exception.
- The setup assembly routine distributed with VMIPS has been made more TLB-friendly. Identity virtual-to-physical mappings for the first few pages of physical RAM are installed in the TLB at program start time.
- When '`-o excmsg`' is on, TLB miss addresses will be printed to stderr.
- When '`-o ttydev=stdout`' is specified, simulated program output will be sent to VMIPS's standard output, even if it is not a tty.
- The interactor can now disassemble memory. Also, stepping in the interactor now prints the PC after each step.
- VMIPS now supports more of the GDB remote serial protocol, in support of the GDB `info threads` and `detach` commands. The debug protocol TCP port is now configurable via the '`-o debugport`' option. Also, if something halts the program while the debugger interface is active, VMIPS will tell GDB that the program exited.
- Bug fixes:
- A bug was fixed in the debugger interface where disconnecting from the debugger socket could cause vmips to enter an infinite loop.
- The interactor will refuse to dump raw memory words at non-word-aligned addresses.
- The CP0 Cause register IP field is now recomputed whenever the register is read, rather than only when exceptions happen. This makes polling loops with interrupts disabled work correctly.
- A bug was fixed in the DECstation-compatible clock device which was preventing some of its registers from being zeroed properly.
- Some endianness bugs were fixed in the generic memory-mapped device code and in the DECstation-compatible serial device.

vmips-1.4.1 was released on 7 May 2013. User-visible changes in version 1.4.1 (since version 1.4):

- Problems compiling VMIPS with gcc 4.7.x have been fixed.
- An error has been fixed in the documentation regarding the installed location of the canned setup code.
- An error has been fixed in the Makefiles which was preventing the canned setup code from being installed.
- An error has been fixed in the Makefiles which was preventing CFLAGS and LDFLAGS from being set correctly in some cases.

vmips-1.4 was released on 29 January 2012. User-visible changes in version 1.4 (since version 1.3.2):

- A new interaction facility was implemented. This allows you to suspend the simulation, print registers and memory, and then continue by running or single-stepping, without needing to use a full-fledged debugger.
- Some of the memory-mapped I/O devices have had endianness bugs fixed and debugging printouts removed.
- Certain coprocessor zero branch instructions no longer unnecessarily cause reserved instruction exceptions.
- Other minor optimizations.

vmips-1.3.2 was released on 27 March 2007. User-visible changes in version 1.3.2 (since version 1.3.1):

- A bug was fixed in the code which responded to address-translation exceptions by loading the CP0 Context register. It was loading the wrong virtual page number.
- Problems compiling VMIPS with gcc 4.1.x have been fixed.

vmips-1.3.1 was released on 5 January 2005. User-visible changes in version 1.3.1 (since version 1.3):

- VMIPS now compiles and passes most of its regression tests under Microsoft Windows using Cygwin. There is one known failure (misc-tests/echo), but the failure stems from the testsuite, not VMIPS; I suspect that my old version of Expect is at fault.
- The build status report script, '`buildstat.sh`', has been removed from the distribution. (Since it was broken in version 1.3 and no one complained, we're assuming that no one will miss it much!)
- Two bugs in the testsuite that could have led to spurious test failures have been fixed.

vmips-1.3 was released on 8 October 2004. User-visible changes in version 1.3 (since version 1.2.2):

- The VMIPS code base has been cleaned up substantially. The most important visible effect of this is that VMIPS compiles much more quickly now.
- A new, experimental executable-file loader has been integrated. Presently it is only usable for loading ECOFF files into the kernel segments (0x80000000-0xbfffffff). The ROM monitor has a new `call` command you can use to take advantage of this.
- Vmipstool can now disassemble binary instructions from the command line, e.g. '`vmipstool --disassemble-word <pc> <instr>`'.
- The '`excmsg`' option used to report all exceptions, including things like clock interrupts. This made using '`excmsg`' in any nontrivial kernel code almost impossible. Now you must also specify 'reportirq' to get reports of interrupt exceptions.
- The '`haltjrra`' option has been removed.
- The DECstation 5000/200-compatible CSR device now supports delivery of interrupts from devices attached to CSR interrupt lines, such as the DZ11 serial device.
- The DZ11 serial device now supports interrupt-driven I/O properly. It previously only worked correctly if polling was used.

- The documentation has received the usual slight adjustments. In addition, the test-suite documentation has been extensively revised and integrated into the programmer's manual.
- The options-processing code has been revised slightly to print a better error message when an invalid option is specified.

vmips-1.2.2 was released on 23 August 2004. User-visible changes in version 1.2.2 (since version 1.2.1):

- A bug in the Makefiles and configure scripts has been fixed, where some files would not be installed if you didn't configure with cross compilation tools (using '`--with-mips`').
- Vladimir Machulsky found and fixed a bug in the debugger interface, where the register file would be sent to gdb incorrectly. Thanks!
- VMIPS will no longer abort and dump core when bad command-line rguments are passed to it. Instead, it will print an error message and exit with a non-zero exit code.
- A bug in the command-line option processor was fixed, where using a completely empty '`~/.vmipsrc`' or the '`-F /dev/null`' option could cause the options parser to parse garbage, usually resulting in harmless, but annoying error messages.

vmips-1.2.1 was released on 26 July 2004. User-visible changes in version 1.2.1 (since version 1.2):

- Fixed a bug where delay slot execution could be triggered even when emulating a malformed jr (jump-register) instruction.
- Fixed a bug where, upon taking a TLB Miss exception, the ASID of the page that could not be looked up would be overwritten (in the CP0 EntryHi register) with garbage.
- Fixed a bug in VMIPS configurations with multiple interrupt devices, where Interrupt Pending bits of the CP0 Cause from previous interrupts could be left asserted even if the device that asserted them had subsequently deasserted them.
- Fixed a bug where the memory dump file and ROM file would be opened in text mode. They are now opened in binary mode.

vmips-1.2 was released on 26 June 2004. User-visible changes in version 1.2 (since version 1.1.3):

- Many documentation updates have been made. Note that the documentation has been relicensed under the MIT license, instead of the GNU FDL.
- Many facets of VMIPS are now configurable at runtime, instead of at compile time. For instance:

  It is now possible to build and install VMIPS without having previously installed MIPS cross-compiler tools. In particular, VMIPS now incorporates the portions of GNU libopcodes used to implement the MIPS disassembler, so linking against an installed version of libopcodes from GNU binutils is no longer necessary.

  It is now possible to switch the VMIPS CPU from big-endian to little-endian mode or vice-versa using a command-line option.

  It is now possible to change the name of the configuration file that VMIPS reads on startup using a command-line option. The '`-o configfile`' option, which never worked, has been removed.

- vmipstool now shares command-line processing code with vmips. It also has a new –swap-words option that allows you to byteswap all the 4-byte words in a file.
- You can now specify numeric constants in command line options which are multiples of 1024, 1024^2, or 1024^3 using the K, M, or G suffixes.
- Several new emulated devices based on the DECstation 5000/200 have been added. In particular, it is no longer possible to disable the serial device at compile time (or with '`-o nousetty`'), but you can choose whether to use the DECstation serial chip or the SPIM-compatible serial console (or neither) using new command-line options. In addition, the ROM program distributed with VMIPS supports a "boot environment" similar to that provided by the DECstation PROM. The new DECstation-compatible devices have not been comprehensively validated; they should be considered "beta" quality at this point.
- There is a new tracing framework you can use to generate runtime execution traces of programs that run in VMIPS. See the documentation of the "tracing" command-line option and the other options that begin with "trace" for more details.
- The VMIPS interface to GDB now supports the "remote Z-packet" interface for setting breakpoints.
- VMIPS now prints more informative diagnostic messages about the causes of bus errors and interrupts.
- You can now use ^C (your terminal's Interrupt key) to stop VMIPS and break into the debugger, if the '`-o debug`' command line option was set, or ^\ (your terminal's Quit key) to halt VMIPS in an orderly manner at any time.
- The '`--disable-debug`' option to `configure` now strips the compiled VMIPS binaries, although it has been documented to do this for some time now.

vmips-1.1.3 was released on 24 October 2003. User-visible changes in version 1.1.3 (since version 1.1.2):

- A bug in comparing large immediate constants near UINT_MAX using the sltiu instruction has been fixed.
- A bug where the meaning of the 'dirty' bit in TLB entries was accidentally reversed has been fixed.
- A bug in specifying which bits are treated as writable in the CP0 EntryLo register by the tlbr instruction has been fixed. Thanks to Mingyu Chen for pointing out these last two bugs.

vmips-1.1.2 was released on 20 August 2003. User-visible changes in version 1.1.2 (since version 1.1.1):

- An instruction that turns on interrupts in the system control coprocessor (coprocessor 0) will no longer immediately take an exception if an interrupt is pending; now it will happen at least one instruction later.
- When building VMIPS in debug mode, the coprocessor 0 Cause register may no longer contain random, non-clearable interrupt-pending bits.
- The coprocessor 0 Cause register's interrupt-pending and coprocessor-error bits are now correctly updated when an exception is taken.

- The VMIPS physical-memory manager will now allow you to place two devices directly adjacent to one another in memory; it used to complain that they overlapped, when they actually did not.
- A typo was fixed in the testsuite documentation.

vmips-1.1.1 was released on 16 June 2003. User-visible changes in version 1.1.1 (since version 1.1):

- A bug involving shifts by 0 or 32 bits (e.g., using instructions like srlv) producing incorrect results has been fixed.
- The absence of an FPU attached to coprocessor slot 1 can now be detected by user programs.
- The system control coprocessor (coprocessor 0) now returns the correct ID number for the R3000A processor in the PRId register.
- Emulation of store-byte and store-halfword operations for emulated devices that only support store-word should now work correctly on byte-swapped configurations.
- A bug in vmipstool which caused it to fail to find alternate linker scripts (supplied with the –ld-script option) has been fixed.
- A gcc warning caused by a (harmless) comment in asm_regnames.h has been quashed.
- A bug where little-endian vmips configurations would dump core when the disassembler was used (e.g., when the instdump option was turned on) has been fixed.
- 'configure' no longer ignores its –with-mips-include option.

vmips-1.1 was released on 14 March 2003. User-visible changes in version 1.1 (since version 1.0.4):

- If you are using g++ to compile VMIPS, you must use g++ version 2.95 or later. Also, you must specify the –target option to 'configure', with an argument that matches the –target option to binutils 'configure'. See 'INSTALL'.
- VMIPS now supports GNU standard –help and –version options.
- The manual now includes improved documentation for using gdb and Insight to control VMIPS, a section on the differences between SPIM's console and the SPIM-compatible console device, a section on proper error reporting from VMIPS extensions, and many other minor revisions.
- Using gdb to control VMIPS is now much more robust in the face of errors and exceptions.
- VMIPS 1.1 makes it possible to use the 2nd display/keyboard channel of the SPIM-compatible console device, using the 'ttydev2' option.
- VMIPS now includes a halt device, which can be used to halt the simulator without using a special instruction.
- VMIPS 1.1 is much faster than VMIPS 1.0.4. This is due in large part to the efforts of Paul Twohey, who reimplemented the clock and SPIM-compatible console devices, and cleaned up the code a bit.
- The VMIPS distribution now includes a regression test suite, based on DejaGNU (replaces the old test_code directory). In order to support this, the formats of lots of random messages that VMIPS prints out have changed.

- We now support 64-bit hosts (e.g., Alpha).

vmips-1.0.4 was released on 28 April 2002. User-visible changes in version 1.0.4 (since version 1.0.3):

- Fixed bug where the first instruction of an interrupt handler would be executed twice in a row. Thanks to Paul Twohey.
- Workaround some problems in the way casts are used in VMIPS by specifying -fno-strict-aliasing to gcc in configure; this should help avoid miscompilations for now, until the code can be rewritten for cast-safety.
- Fix bug where vmips would dereference a NULL pointer when trying to use a SPIMConsole device with a non-attached serial host. Thanks to Paul Twohey.
- Reorder some tests in the SPIMConsole code for a slight speedup when interrupts are turned off.
- Fix bug where virtual page numbers were being incorrectly interpreted by the system control coprocessor, resulting in excess TLB misses being generated. Thanks to Sanjeev Datla.

vmips-1.0.3 was released on 12 January 2002. User-visible changes in version 1.0.3 (since version 1.0.2):

- Fixed bug in debugger interface; TLB translations caused by the debugger trying to access user space could fail if the virtual page number was not 0.
- Fixed bugs in the following instructions: lb, lh, lwr, lwl, swr, swl, nor.
- Clarified some especially confusing comments in the source code, in one of the test cases, and in the testsuite README.
- The sample setup code now halts on all exceptions, including user TLB miss exceptions.
- Fixed some warnings compiling xmboot using recent versions of gcc.

vmips-1.0.2 was released on 17 December 2001. User-visible changes in version 1.0.2 (since version 1.0.1):

- Add warning message to man page directing the interested reader to the Info manual.
- New default handling of '-o ttydev' option; manual updated.
- /etc/vmipsrc contains only comments by default. These two fixes work around a bug in Darwin select(2)'s tty handling.
- Only print host processor endianness if '-o bootmsg' is set. Fixes bug where '-o bootmsg' should (but doesn't) turn this message off.
- Remove test_code/testdev.S from the branch. This is a test for a feature (TestDev) which wasn't distributed with 1.0.x.
- Comment out extra tokens after #endif in test_code/{sort.c,tester.c}. Fixes bug where gcc 3.0 gives warnings compiling these files.

vmips-1.0.1 was released on 27 November 2001. User-visible changes in version 1.0.1 (since version 1.0):

- Fixed format-string bug in vmipstool.
- Fixed "extra tokens after #endif" warning in sample_code/asm_regnames.h.

vmips-1.0 was released on 28 October 2001. User-visible changes in version 1.0 (since version 0.9):

- A standard clock device has been added.
- The "vmipstool" front end to compilation tools was added, and most of the random scripts lying around have been consolidated into it.
- The memory-mapped test device known as "TestDev" has been removed.
- Debian and Red Hat packaging files have been added.
- The installation procedure has been greatly improved.
- The VMIPS manual has been completed.
- Many, many bugs fixed as a result of a concerted attempt at testing everything. Doubtless, some bugs still remain.
- Some portability fixes.
- Dumping of the stack (when registers are dumped) now works.
- Many strange messages the debugger backend used to print out have been quelled. Also, killing the debugger connection should no longer cause a crash.
- vmips now supports –help, –version and –print-config command-line options.
- The SPIM-compatible console device now works better when the host machine is byte-swapped with respect to the MIPS target.

vmips-0.9 was released on 9 May 2001. User-visible changes in version 0.9 (since version 20001014):

- vmips.debug is no longer built.
- Read-only memory is now really read-only. This allowed us to add debugger support for breakpoints in ROM.
- Debugger support for exception handling – CPU exceptions are now delivered to your GDB, if it is attached, giving you a chance to poke around before the target is made aware of the condition.
- Many tasty configury updates, making it easier to build.
- Miscellaneous bugs fixed in many places.
- Branch on coprocessor 0 instructions now have defined behavior.
- Many places which erroneously had text after #endif in the source have been corrected to use comments instead, allowing you to compile vmips with newer C++ compilers.
- Some old test code that used to run at the beginning of every vmips job has been removed, so you will no longer see the familiar messages it printed out.

User-visible changes since version 20000517:

- Emulating a little-endian MIPS on a big-endian host passes preliminary tests. Build process queries MIPS tools for their endianness automatically. Builds tested on Solaris SPARC, i386 Linux, FreeBSD and HP-UX. Distribution building (gmake distcheck) almost works.

User-visible changes since version 19991114:

- Polling input and output via the SPIM-compatible console device now work.

User-visible changes since version 19990829:

- Build processes seeks out MIPS tools; Installation documentation now briefly documents how to build cross tools.

User-visible changes since version 19990823:

- Spim console documentation completed; first attempt at spim console code.

User-visible changes since version 19990801:

- Documentation Makefile.in has been built.
- Some documentation has been updated.
- VMIPS is being adapted to use GNU Autoconf and Automake.

Appendix E References

Silicon Graphics, Inc. *The R10000 Microprocessor User's Manual - Version 2.0.* Available from

`http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi/hdwr/bks/SGI_Developer/R10K_UM`

as of June 1, 2004.

This is a good reference about a typical 64-bit MIPS processor, and also has some useful application notes. However, the processor it describes is currently much more advanced than the VMIPS simulation.

Silicon Graphics, Inc. *SGI TechPubs Library: The ABI(5) manual page.* Available from

`http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi?coll=0650&db=man&fname=/usr/share/catman/p_man/cat5/abi.z`

as of June 1, 2004.

This is a short manual page about the three prevalent MIPS ABIs (application binary interfaces), termed O32, N32, and N64.

Silicon Graphics, Inc. *SGI TechPubs Library: The MIPS_EXT(5) manual page.* Available from

`http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi?cmd=getdoc&coll=0650&db=man&fname=5%20mips_ext`

as of June 1, 2004.

This short manual page is a good summary of the differences between the various MIPS ISA levels (MIPS-II, MIPS-III, MIPS-IV).

Kane, Gerry, and Joe Heinrich. *MIPS RISC Architecture.* Upper Saddle River, New Jersey: Prentice-Hall, 1992. ISBN 0135904722.

This is a good all-around reference for the 32-bit MIPS processors which VMIPS is modelled upon, and it includes a complete list of all the 32-bit MIPS-II instructions as well as a description of the MIPS TLB, virtual memory, exception behavior, and caches. It is not a particularly good reference for the 64-bit versions of the MIPS architecture, though.

Sweetman, Dominic. *See MIPS Run.* San Francisco: Morgan Kaufmann Publishers, 1999. ISBN 1558604103.

This is a general reference in the style of Kane and Heinrich, but updated for the MIPS-III, MIPS-IV, and MIPS-V ISAs, and written in a much more experienced and less minimalist style, with attempts to include useful pieces of MIPS lore.

Delorie, DJ. DJGPP COFF Spec. October, 1996. Available from

`http://www.delorie.com/djgpp/doc/coff`

as of June 1, 2004.

A good online reference for the COFF file format, a form of which was heavily used on DEC MIPS implementations.

Tool Interface Standard Committee. Executable and Linking Format Specification. Version 1.2, May 1995. Available from

`http://www.linuxbase.org/spec/refspecs/elf/elf.pdf`

as of June 1, 2004.

An online reference for the ELF file format, now the preferred object file format for Unix systems. This document is highly Intel architecture-specific, but it provides a lot of useful background material.

The Santa Cruz Operation. System V Application Binary Interface: MIPS RISC Processor Supplement, 3rd Edition, February 1996. Available at

`ftp://www.linux-mips.org/pub/linux/mips/doc/ABI/mipsabi.pdf`

as of June 1, 2004.

The part of the System V application binary interface guide that pertains specifically to MIPS RISC processors. It describes, among other things, a position independent coding model (PIC) for MIPS.

Also worth checking out is

`http://www.mips.com/publications/index.html`

which points to many MIPS Technologies, Inc. publications.

Appendix F Copying

VMIPS and its source code are governed by the GNU General Public License, which you should have received a copy of along with VMIPS. It is in the source code distribution in the file '`COPYING`'.

VMIPS's documentation is governed by the MIT license. A copy of that license follows:

Index

Table of Contents