VMIPS Programmer’s Manual

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.

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/setup 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
```

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 TLB entries.

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.

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 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`'. There is probably a way to configure the cross compiler so that it won't try to do this; it will be documented here once it is discovered. A simple workaround is to call the entry function `entry` instead of `main`.

4.6.2 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 [-D] [-o option_string] ... rom_file
vmips --help
vmips --version
vmips --print-config
```

This is what the different command line options mean:

'-D' Turns on debugging of option parsing. This is not generally useful to the end user unless you are confused about VMIPS's interpretation of your command line or startup file(s). VMIPS has to be compiled with -DOPTIONS_DEBUG for this option to be available.

'--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; usually this is in '`/usr/local/share/vmipsrc`', but it may have been moved by the site administrator. (This is configurable in the source file '`options.h`', and by specifying the –prefix and –sharedir 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. 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 line exceeds BUFSIZ (usually 1,024) characters. Whitespace separates options from one another. 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 will be dumped on halt. The default value is FALSE.

'`haltdumpcp0`' (type: Boolean)

Controls whether the system control coprocessor (CP0) registers will be dumped on halt. 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. 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 TRUE. 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 will be dumped after every instruction. The default value is FALSE. The output is in the following format:

```
Reg Dump: PC=bfc00080 Last Instr=241f001f HI=00000000 LO=00000000
DELAY_STATE = NORMAL ; DELAY_PC=00000000 ; NEXT_EPC = bfc0007c
R00=00000000 R01=00000001 R02=00000002 R03=00000003 R04=00000004
...
R30=0000001e R31=0000001f
```

(Some values have been omitted for brevity.) Here, PC is the program counter, Last Instr is the last instruction executed, HI and LO are the multiplication/division result registers, DELAY_STATE and DELAY_PC are used in delay slot processing, NEXT_EPC is what the Exception PC would be if an exception were to occur, and R00 ... R31 are the CPU general purpose registers. All values are in hexadecimal.

'dumpcp0' (type: Boolean)

Controls whether the system control coprocessor (CP0) registers will be dumped after every instruction. The default value is FALSE. The output is in the following format:

```
CP0 Dump Registers:
R00=00000100 R01=00001f00 R02=06a5ee00 R03=00000000
R04=7fffca10 R05=00000000 R06=00000000 R07=00000000
R08=7fb7e0aa R09=00000000 R10=6f6dd980 R11=00000000
R12=00485e18 R13=30002110 R14=4c04a8af R15=0000703b
```

Each of the R00 .. R15 are coprocessor zero registers. Their values are displayed in hexadecimal.

'haltibe' (type: Boolean)

If haltibe is set to TRUE, the virtual machine will halt when an instruction fetch causes a bus error (exception code 6, Instruction bus error). This is useful if you are expecting execution to jump into unmapped areas of memory, and you want it to stop instead of calling the exception handler. The default value is TRUE.

'haltjrra' (type: Boolean)

If haltjrra is set to TRUE, the virtual machine will halt when the instruction "jr $31" (also written "jr $ra") is encountered. Since this is the instruction for a procedure call to return, this is useful if you have a simple procedure to run and you want execution to terminate when it finishes. The default value is FALSE.

'`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. 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:

```
733737 instructions executed in 5.81484 seconds
126183.545 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".

'`configfile`' (type: string)

This is the name of the user configuration file. It will be ~username-expanded and checked for configuration options before the virtual machine boots. The default value is "~/.vmipsrc".

'`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. You might want to round this off to the nearest page; you can determine the pagesize using utils/getpagesize.cc. The default value is 0x100000.

'`memdump`' (type: Boolean)

If memdump is set, then the virtual machine will dump its RAM into a file named "memdump.bin" at the end of processing. The default value is FALSE.

'`reportirq`' (type: Boolean)

If reportirq is set, then any change in the interrupt inputs from a device will be reported on stderr. The default value is FALSE.

'`usetty`' (type: Boolean)

If usetty is set, then the SPIM-compatible console device will be configured. If it is not set, then no console device will be available to the virtual machine. The default value is TRUE.

'`ttydev`' (type: string)

This pathname will be used as the device from which reads from the console device will take their data, and to which writes to the console device will send their data. If the OS supports ttyname(3), that call will be used to guess the default pathname. The default value is "/dev/tty".

'`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.

'`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 200.

'`clockspeed`' (type: number)

If the realtime option is not set, this option gives the speed of simulated time in Hz. It has no effect if realtime is set. The default value is 25000000.

'`clockintr`' (type: number)

This option gives the frequency of clock interrupts, in nanoseconds of simulated time. The default value is 1000.

7 Invoking vmipstool

`vmipstool` is intended to be a friendly front-end to the process of compiling, linking, and assembling code for VMIPS.

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 PROG (or FILE.o)
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.)

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 `periodic()`. 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 through an interface to GDB, the GNU debugger. GDB talks to VMIPS using its built-in remote serial protocol. See the "Remote Serial" section of the GDB manual for details of the protocol.

If you want to take advantage of 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
```

(The host and port numbers (127.0.0.1:3371) may be different on your machine.) 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; rather, give GDB the name of the program you used to create the ROM file. Then type the "target remote..." command that VMIPS printed out, and 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.

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.
Waiting for packet 0
```

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 GDB remote serial protocol implementation

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 's': Single step

packet 'k': Kill target

packet 'H': Set thread

packet '?': What was the last signal?

9.3 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.

10 Devices

VMIPS comes with a few standard devices.

10.1 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.

10.1.1 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.

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.

10.1.2 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 of the console devices becomes ready while its Device Interrupt Enable bit is set, it also sets the appropriate bits in the Cause register of CP0 and signals an Interrupt exception.

10.1.3 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.

10.1.4 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.

10.1.5 Keyboard

The keyboard is initially unready; whenever the connected serial interface has a char waiting on input, and the keyboard is unready, the keyboard reads the char 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 char 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.

10.2 Standard clock device

This section documents the standard clock device for VMIPS. It is intended to support user programs' access to real time, for the benefit of realistic user perceptions, and simulated time, for repeatability of experiments. It also supports a hardware clock interrupt, and provides access to TOY (i.e., Time of Year) for those special occasions when you want the clock to tell you what time it is, not just how fast things are going.

The reason this clock is to be preferred over the SPIM-compatible console device's clock is because it provides much better resolution (1 MHz vs. 1 Hz).

10.2.1 Memory-mapped registers

The standard clock device has 5 registers, configured by default to be mapped into memory at address 0xa1010000. 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 time words is undefined. Support for setting the time from a MIPS program may be added in the future.

The control word has 32 bits. The least-significant bit, when set (`CDC_INTERRUPTS_ENABLED` is defined as 0x00000001), enables the clock interrupt. All other bits of the control word are currently reserved and read as zero. Writing any of the other bits of the control word is undefined.

10.2.2 Interrupts

The standard clock device is connected to hardware interrupt line 7 in the default configuration.

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.) When a clock interrupt is triggered, it will be deasserted during the following instruction.

10.2.3 Real vs. simulated time

Real time is obtained using the host's `gettimeofday(2)` system call, so it should run pretty close to whatever the host thinks the current time is. No sophisticated algorithms are used to calibrate the real time clock, though.

The speed of simulated time is determined by the 'realtime', 'clockspeed', and 'timeratio' options. (See the "Summary of configuration options" section of the "Customizing" chapter for more information.) Increasing the speed of simulated time is unlikely to make the simulator appreciably faster.

11 Extending

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

11.1 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 is all 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 classes `Range` and `ProxyRange`. This is meant to be logically and physically separate from the TLB, which is implemented as part of the system control coprocessor.

The system control coprocessor (MIPS coprocessor zero) and the TLB are implemented in '`cpzero.cc`' and '`cpzero.h`', as class `CPZero`.

There is a cache implementation in '`cache.cc`' and '`cacheline.cc`' (classes `Cache` and `CacheLine`), but these modules are not finished yet.

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`'); the only really important feature of this class right now is the `exception` instance method and its implementation in class `CPU`.

The disassembler, which uses GNU libopcodes (part of GNU Binutils), is in '`stub-dis.cc`'.

The SPIM-compatible console device, which is comprised of two serial line TTYs and a clock, is in '`spimconsole.cc`' and '`spimconsole.h`'.

The standard clock device is in '`clockdev.cc`' and '`clockdev.h`'.

'`zschip.cc`' contains an unfinished emulation of the Zilog Z85C30 serial chip, commonly found in Macintoshes and Suns.

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`'.

Many 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. Any part of the VMIPS system which needs to be called back periodically to update its state can inherit from class `Periodic` (described in '`periodic.h`') and implement the `periodic()` member function. The `CPU` class, for example, fetches, decodes, and executes one instruction each time its `periodic()` function is called. Devices, such as the SPIM-compatible console device, use their `periodic()` function check for and complete pending I/O and trigger interrupts as necessary.

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 periodic() member function(s). The vmips class is not a very smart or a very flexible configuration mechanism; it will eventually be replaced with a configuration language of some sort.

The ROM bootstrap loader code (also known as the ROM monitor) is in the directory 'sample_code/xmboot'. The current ROM monitor uses the XMODEM protocol.

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

A bunch of interesting test cases are in the directory 'test_code'. Most of the interesting sample code is being moved into the directory 'sample_code', but some interesting sample code continues to be stored in the 'test_code' directory.

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

11.2 Endianness issues

There are no known bugs involving endianness issues.

However, 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). The configuration procedure determines the endianness of the VMIPS target and of the host processor, and will define the C preprocessor symbol BYTESWAPPED if the two are different. You can then call the swap_word() or swap_halfword() static methods of class Mapper to do the translation between host and target, when necessary.

As you might expect, this is most important when dealing with reading from memory or from memory-mapped devices.

Most endianness problems can be dealt with using the BYTESWAPPED symbol, except those problems originating in third-party libraries which you might hook up to VMIPS. If you are calling external code that has to know whether to expect big-endian or little-endian instructions or data, or whether the host processor is big-endian or little-endian, you can use the C preprocessor symbols TARGET_LITTLE_ENDIAN and TARGET_BIG_ENDIAN for the target, and testing for the presence or absence of the definition of WORDS_BIGENDIAN for the host.

11.3 Memory-Mapped Devices

Memory-mapped devices must inherit from class DeviceMap. They must have a constructor and a destructor, and override the following abstract methods:

```
uint32 fetch_word(uint32 offset, int mode, DeviceExc *client);
uint16 fetch_halfword(uint32 offset, DeviceExc *client);
uint8 fetch_byte(uint32 offset, DeviceExc *client);
uint32 store_word(uint32 offset, uint32 data, DeviceExc *client);
uint16 store_halfword(uint32 offset, uint16 data, DeviceExc *client);
uint8 store_byte(uint32 offset, uint8 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.

11.4 Exception behavior

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

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

Type must be one of the standard MIPS exception codes, which are defined in '`regnames.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.

11.4.1 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.

11.5 Mapping memory-mapped devices

You can map memory-mapped devices at one location, or more than one location, if you want. 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);
```

Therefore, if you want to have multiple base-addresses for a device, you can. You can add as many calls to the Mapper instance method add_device_mapping(device, addr) as you want. *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.

11.6 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.

11.6.1 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.

11.6.2 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.

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.

11.6.3 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.

11.6.4 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.

11.7 Weird things

11.7.1 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.

Appendix A Installation

The VMIPS build process assumes that you have a full set of GNU MIPS cross compilation tools installed, because you'll need them to do anything useful with VMIPS. For a concise summary of how to build the necessary MIPS cross tools, read "Building MIPS Cross Tools", below.

VMIPS uses the GNU Autoconf/Automake system for configuration management. This provides the familiar `configure` shell script interface for setting configuration variables. 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 Building from CVS

If you retrieved your sources from the CVS repository, you will need Automake version 1.4 or later, Autoconf version 2.13 or later, and libtool 1.2f 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.2 Options that configure supports

You will need to tell `configure` the configuration prefix you used to install the MIPS cross tools, by specifying it as the value to the '`--with-mips`' argument. For example, if your MIPS cross compiler is '`/opt/mips/bin/mips-dec-ultrix4.3-gcc`' and your MIPS-targeted libopcodes libtool library (which should have been installed by the binutils Makefile) is '`/opt/mips/lib/libopcodes.la`', then you should specify '`--with-mips=/opt/mips`' on the `configure` command line.

Other options that "configure" supports are as follows:

'`--with-mips=MDIR`'
: Specify installation prefix of MIPS cross tools (default MDIR = /opt/mips).

'`--with-mips-lib=DIR`'
: Specify path to MIPS cross tools' libraries (default MDIR/lib).

'`--with-mips-bin=DIR`'
: Specify path to MIPS cross tools' executables (default MDIR/bin).

'`--with-mips-include=DIR`'
: Specify path to MIPS cross tools' includes (default MDIR/include).

'`--with-mips-bfdtarget=TARG`'
: Specify MIPS cross tools BFD target name (defaults to the first target listed in the output of objdump -i). Normally you can let configure guess this, unless you built your MIPS tools for a target (mips-ecoff or mips-elf, for example) which supports both big-endian and little-endian data.

'`--with-mips-endianness=VAL`'
: Specify endianness of the VMIPS simulated machine, which must match the MIPS cross tools target's endianness. VAL may be specified as `big` or `little`.

It is best to let configure guess this (using 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.

'`--disable-debug`'
: Strip debugging symbols and turn on all the compiler optimizations. The default is not to do this (i.e., leave in the debugging symbols, and turn off all the compiler optimizations.)

'`--disable-tty`'
: Do not include (default=include) support for the emulated serial interface.

A.3 Building MIPS Cross Tools

First decide on an installation prefix. The following examples will use the prefix "/opt/mips", as above.

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

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

We recommend getting version 2.10.1. The file you will need would be named '`binutils-2.10.1.tar.gz`'.

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

```
./configure --target=mips-dec-ultrix4.3 --prefix=/opt/mips \
 --disable-nls --enable-shared
make
make install
```

Save a copy of '`include/dis-asm.h`' from the Binutils source distribution. You'll need to install it as '`include/dis-asm.h`' in the VMIPS source distribution, in order to ensure compatibility between the version of Binutils you used and VMIPS.

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

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

We recommend version 2.95.2. Download the file '`gcc-2.95.2.tar.gz`'.

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-2.95.2`'.
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-2.95.2/configure --target=mips-dec-ultrix4.3 \
    --prefix=/opt/mips --with-gnu-as --with-gnu-ld \
    --disable-threads --disable-shared
   ```

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.

   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. Some persons have reported success using the `"newlib"` C library from Cygnus. The GNU C Library (glibc) is fairly difficult to build. Instructions for building a C library will appear here soon.

Appendix B Reporting Bugs

We are always interested in hearing about VMIPS bugs. Please send mail to `vmips@sartre.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 make 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

C.1 Version 1.1 targets

C.1.1 Caches

The caches are not currently finished, and not connected to the rest of the memory subsystem.

Arguments in favor of finishing the caches include the fact that operating systems tend to expect them, and you can learn some things about how many cache misses your code incurs if you simulate them. Arguments in favor of ditching the caches center around the performance penalty that simulating them would cause.

Finishing the implementation of the cache is a two-part task:

1. Implement cache "store partial" mode.
2. Implement the non-"store partial" mode; i.e., where a store causes the cache line to be invalidated unconditionally. (pp. 5-5 & 5-6 in MIPS RISC Architecture).

Understanding how to do this is probably best accomplished by first reading '`cache.cc`' and '`mapper.cc`'.

C.1.1.1 Cache behavior and the Status register

CP0 implements two Diagnostic Status bits in the Status register called IsC and SwC; they can be used by the OS to determine the cache geometry exactly, for example. These can be tested by the appropriate accessor functions in cpzero.cc.

Cache isolation (IsC = 1) means the cache is never refilled and all accesses are considered hits; stores write into the cache and do not write into memory.

Cache swapping (SwC = 1) means the instruction cache and data cache are switched, so that instruction fetches go through the data cache and load/store instructions go through the instruction cache.

C.1.2 Other version 1.1 targets

1. Make the SPIM console device more efficient. Currently the code spends 36% of its time in calls to `select()` and 35% in all the emulation functions, which is unacceptably slow. Probably we will want to use a separate thread to cause interrupts instead of polling. We'll likely have to restructure the interrupt controller for this.

   A thought I had was to use a separate thread for everyone who wants to have `periodic()` called.

   Michael Constant suggesting using `FIOASYNC`, `sigaction()` with `SA_SIGINFO`, and handling `SIGPOLL` by dispatching to the correct interrupt generator. This would be both more portable and easier to understand than threads.
2. Make the build process suck less. Right now it requires drawing a pentagram around your computer on the night of a full moon with torches blazing, not to mention a very

close synchronization between versions of gcc, binutils, gdb, and whatever else you are trying to use. It may be a good idea to build a "combined source tree" including gcc, vmips, gdb, and binutils source together for people to install.

3. Generally make things faster. Some other strategies for making things faster:
4. The way we do options must die. Replace it with

   a bunch of global (or instance-variable-local) variables, to replace class Options, and

   a real configuration language (to replace '`.vmipsrc`'). Probably Tcl or Guile Scheme will be used. Tcl is attractive because it is well-integrated with the Cygnus combined source trees. Guile Scheme is attractive because it is the official GNU language for extension.
5. The console device stuff, above.
6. Galen's dynamic recompilation stuff, or a suitable replacement using GNU Lightning.
7. We need to have a benchmark suite in place in order to quantify all this.
8. Some of the data structures could use some improvement:

   memory mapping range list

   maybe tlb
9. Write a testing framework. DejaGNU would be the place to begin.

C.2 Version 1.5 targets

1. When configuring a serial device, the user should be able to select a machine-dependent serial front-end (e.g., unix-xterm, posix-serial, mac-serial, mac-ttywidget, etc.) and a machine-independent serial back-end (zilog-z85c30, spim-console, etc.) These would be good to implement using shared libs. Also, modularizing the CPU, memory mapper and other pieces (freezing APIs, building as shlibs) would be a Very Good Thing Indeed. (In general, we need to throw out vmips.cc and replace it with a smart configuration mechanism, probably based on Guile Scheme or Tcl.)
2. Ideally, debugger should be able to be attached at any time, without the user having had to think of it. Delay the debugging interface initialization until a connection is received?
3. NLS (i.e., i18n). This will require some un-American activities.

C.3 Version 2.0 targets

1. One thing VMIPS could really use is an emulation of the MIPS R3010 FPU. A good way to get started on building an FPU would be to use SoftFloat: `http://www.cs.berkeley.edu/~jhauser/arithmetic/softfloat.html`.

C.4 Other things to do

Future directions which are not associated with any particular release target:

1. For the ROM monitor, support TFTP (UDP) over SLIP with some fixed kernel name and a configurable address (but probably not ARP).

2. For the `ZSChip` module, finish the emulation, and validate it against some stable free-software serial drivers.
3. For the debugging interface, support more and more different GDB remote serial protocol packets. It would have been nice, for example, to use the remote Z-packet interface for breakpoints.
4. Full MIPS32 support.
5. Checkpoint and restart of simulations.
6. Develop a patch for gas to support software register names. gas supports $sp and $gp but not, say, $t0.
7. Consolidate some of the .h files that just contain huge lists of useful constants.

Appendix D References

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

`http://www.sgi.com/processors/r10k/manual.html`

as of May 24, 2001.

> 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 May 24, 2001.

> 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 May 24, 2001.

> 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.

> 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.

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

> 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.

MIPS ABI Group Incorporated. MIPS Processor ABI Conformance Guide, Version 1.2.2, 1996. Available at

`http://www.eagercon.com/resources/MIPSabi12/toc.html`

as of June 3, 2001.

> Describes, among other things, a position independent coding model (PIC) for MIPS.

Delorie, DJ. DJGPP COFF Spec. October, 1996. Available from `http://www.delorie.com/djgpp/doc/coff`

as of June 3, 2001.

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:80/spec/refspecs/elf.pdf`

as of June 3, 2001.

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, Inc. System V Application Binary Interface: MIPS RISC Processor Supplement. 3rd ed., 1996. Available from

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

as of June 4, 2001.

The part of the System V application binary interface guide that pertains specifically to MIPS RISC processors.

Also worth checking out is

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

which points to many MIPS Technologies, Inc. publications.

Appendix E 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 GNU Free Documentation License; see below for details.

E.1 GNU Free Documentation License

Version 1.1, March 2000

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other written document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies of the Document numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

   B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

   C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

   D. Preserve all the copyright notices of the Document.

   E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

   F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

   G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.
I. Preserve the section entitled “History”, and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled “History” in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the “History” section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. In any section entitled “Acknowledgments” or “Dedications”, preserve the section’s title, and preserve in the section all the substance and tone of each of the contributor acknowledgments and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section entitled “Endorsements”. Such a section may not be included in the Modified Version.
N. Do not retitle any existing section as “Endorsements” or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgments", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

E.1.1 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.1
or any later version published by the Free Software Foundation;
with the Invariant Sections being list their titles, with the
Front-Cover Texts being list, and with the Back-Cover Texts being list.
A copy of the license is included in the section entitled ‘‘GNU
Free Documentation License’’.
```

If you have no Invariant Sections, write “with no Invariant Sections” instead of saying which ones are invariant. If you have no Front-Cover Texts, write “no Front-Cover Texts” instead of “Front-Cover Texts being *list*”; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

Table of Contents