Final Evaluation of MIPS M/500

Daniel V. Klein
Robert Firth

November 1987

Final Evaluation of MIPS M/500

Daniel V. Klein
Robert Firth

Software for Reduced Instruction Set Computers (RISC) Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the SEI Joint Program Office HQ ESC/AXS

5 Eglin Street

Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF, SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Copyright 1987 by Carnegie Mellon University.

This document is available through Research Access, Inc. / 800 Vinial Street / Pittsburgh, PA 15212. Phone: 1-800-685-6510. FAX: (412) 321-2994. RAI also maintains a World Wide Web home page at http://www.rai.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service / U.S. Department of Commerce / Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides acess to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential con tractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218. Phone: 1-800-225-3842 or 703-767-8222.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Final Evaluation of Mips M/500

Abstract: In response to a request from the DoD, an analysis of a Reduced Instruction Set Computer (RISC) processor, the Mips M/500, was performed. All aspects of processor capabilities and support software were evaluated, tested, and compared to familiar Complex Instruction Set Computer (CISC) architectures. In all cases, the RISC computer and its support software performed better than a comparable CISC computer. This report provides the general and specific results of these analyses, along with the recommendation that the DoD and other government agencies seriously consider this or other RISC architectures as a highly viable and attractive alternative to the more familiar but less efficient CISC architectures.

1. Introduction

This report describes our evaluation of the Mips M/500 RISC processor[1] as part of our ongoing research into RISC class architectures. Our intention was to review the general class of RISC architectures using the Mips M/500 as an example of this type of machine, rather than to specifically evaluate the Mips M/500. Although it is difficult to generalize about the behavior of all RISC processors from the performance of a single example, we have tried to point out the strengths and weaknesses of the Mips M/500 in relation to other architectures, and we have tried to demonstrate how the shortcomings and positive aspects of the Mips M/500 can be extrapolated to other RISC class machines.

This report covers our findings, offering insights into the strong and weak points of the Mips M/500, often by comparing it to the Vax (for both hardware evaluation and compiler evaluation purposes). In analyzing the shortcomings of the Mips M/500, we try to offer possible solutions and present comparisons to the general RISC class of architectures.

I entered into the RISC assessment project with a strong bias toward CISC architectures. I looked at this project as an interesting exercise in which I would have my suspicions about RISC processors confirmed, and one in which quite consistently I would find vindication for the CISC side in the great *"RISC versus CISC"* debate.

I have, however, come to the opposite conclusion. My research on this project has convinced me (quite consistently, I might add) that, if there is a "right" side of the debate to be on, it is the RISC side. In all features – execution speed, compiler efficiency, language consistency, and code size – the concept of a *reduced* instruction set computer has proven to be the correct architectural choice. The term *reduced* has in no way implied *restricted*, nor has it caused the horrific increases in code size that CISC proponents tout to support their cause. In fact, comparing the Mips M/500 instruction set usage versus the Vax instruction set usage, we found that the instructions used by the Mips M/500 compilers closely paralleled those used by the Vax. The main deviation was in the area of

---

[1]The Mips M/500 is produced by Mips, Incorporated, Sunnyvale, CA. It is one implementation of the R2000 processor architecture. All references in this report to the Mips M/500 architecture refer to the R2000, while all performance statistics refer to the Mips M/500. Mips, Incorporated also manufactures faster versions of the R2000 – the Mips M/800 and the Mips M/1000. These processors were not evaluated for this report.

addressing modes, but, by and large, the VAX compilers poorly used the complex modes provided by the VAX hardware.

Daniel V. Klein
Principal Investigator

2. Evaluation Methodology

Our studies concentrated on three areas:

1. instruction set conformance

2. benchmark performance

3. compiler and assembler effectiveness

The results obtained in these three areas of research are elaborated in their respective chapters. Here we describe the methodology used to evaluate the MIPS M/500.

2.1. Compliance with DoD CORE MIPS ISA

MIPS Incorporated had previously enrolled in the DoD CORE ISA standard.[2] In brief, this standard allows a hardware manufacturer to specify its own RISC class architecture as long as the architecture either conforms directly to the standard, or can provide assembly language translators from the manufacturer ISA to the CORE ISA and the manufacturer's ISA. Technically speaking, even the VAX satisfies these requirements, but the extra features provided by the VAX are considered detrimental by the standard.

To determine whether the MIPS M/500 satisfies the requirements of the CORE ISA, we evaluated the architecture with these evaluation criteria:

1. Justification for extra instructions – Are the instructions that MIPS added to the CORE in their implementation reasonable? That is, can they be generated by a compiler, are they needed to perform special operating system functions, or are they reasonable for use in specialized high level applications?

2. Justification for removed instructions – Did MIPS exercise reasonable judgement in eliding instructions from the CORE in their implementation? That is, can the functions of the missing instructions be carried out with other instructions or combinations of instructions? Can an automatic translator perform this translation?

3. Number and classes of registers – Does the number of registers meet or exceed the requirements of the CORE? Are the registers general, or are there special case registers which must be used in special ways? How do special case registers affect the overall design?

4. Is it RISC? This is a very difficult question to answer since we have not yet established what "RISC" means. We did, however, attempt to classify the MIPS M/500.

---

[2]*CORE set of Assembly Language Instructions for MIPS Based MicroProcessors*, Version 3.2, January 1987; originally written by Thomas Gross, Carnegie Mellon University; maintained by Robert Firth, Software Engineering Institute.

2.2. Benchmark Performance

We ran many standard and non-standard benchmarks on the MIPS M/500 (and on the VAX for comparison purposes). Some of the results are presented in chapter 4, although not all of our tests are reported. We have not withheld any useful information; however, some of the benchmarks were inconclusive or inapplicable. The benchmark suite consisted of:

1. BYTE Benchmarks – the benchmark suite from BYTE magazine, August 1983 and August 1984.

2. Whetstones – the quintessential floating point benchmark (although we show how this is an inadequate benchmark to use).

3. Dhrystones – an integer benchmark similar in functionality to the Whetstone benchmark.

4. EUUG Workstation Performance – a set of simple programs released by the European UNIX Users Group to test a computer's performance under varying loads.

5. LinPak – Jack Dongarra's matrix manipulation benchmarks, written at Argonne National Laboratories.

6. Spice – a circuit simulator often used to measure processor efficiency. This benchmark heavily loads the floating point hardware.

7. LLNL Loops – a set of FORTRAN kernels, released through Lawrence Livermore National Laboratories, designed to exercise the floating point system.

8. Buchholz – an artificial benchmark designed at IBM to measure system load handling capabilities.

9. FORTRAN FP – a simplistic benchmark for measuring the time to execute various FORTRAN floating point operations. This was deemed too simplistic to report on.

10. FFT – a simple FFT algorithm, analyzed to compare compiler efficiency.

11. 20 Queens – an extension of the 8 queens placement problem, analyzed to compare compiler efficiency.

12. UNIX *lex* – a lexer generator for which a number of degenerate lexical specifications exist. These specifications heavily load the lexer generator, and provide a reasonable "natural" benchmark.

13. Ackermann's Function – a test devised to evaluate the efficiency of a compiler's recursive code analysis and generation.

2.3. Compiler Performance

The MIPS compilers were carefully examined for a number of characteristics. Many of these characteristics are specific to the MIPS M/500, but some of our findings can easily be generalized to other compilers. The areas of compiler performance that we examined are:

1. Compiler speed – that is, speed of compilation during the parsing, code generation, and optimization phases, as well as time spent on assembly and assembly level code reorganization. This phase of analysis looked at how long a user would have to wait for a compilation to run, regardless of the optimality of the generated code.

2. Speed of generated code – that is, how fast the compiled code would run. This test

was varied over different levels of compiler optimization and was tested with many of the benchmarks described above in section 2.2.

3. Optimizer efficiency – that is, how good is the code that is generated by the compiler and assembler. To evaluate this, we looked at four aspects of optimization:

   a. Optimization techniques used – which methods are used, and which are not used. The optimization techniques we looked for were include: α-motion, ρ-motion, ω-motion, routine hoisting, loop-invariant code motion, common sub-expression elimination, arithmetic expression reorganization, branch optimization, multi-way branch evaluation, etc.

   b. Register usage – how well the registers are allocated. We examined register tracking, register re-use, and type of register use (i.e.,, addressing mode inter-actions with register use), as well as register usage in parameter passing.

   c. Instruction utilization – how well the instruction set of the native machine is used, including evaluations on the efficiency of code idioms used, and on the optimality of the generated code. We also examined the code that was generated for algorithms that could be written in the three languages available on the MIPS M/500: FORTRAN, C, Pascal.

   d. Instruction coverage – how completely the instruction set of the native machine is used, including percentages of used versus unused instruction.

   These topics are all discussed in chapter 6.

4. Assembly reorganization and pipelining – that is, how well the assembler reorganizer kept the pipeline filled, how efficiently `nop` instructions were eliminated, and what the reorganizer was able to accomplish in final stage peephole optimization. We also looked at code idioms that would enhance reorganization, and at those we found that hindered reorganization. This topic is discussed in chapters 3 and 7, and again in appendix A.

2.4. Applicability

We also examined the applicability of the MIPS M/500 (and of RISC architectures in general) in common environments. The problem areas that we considered were:

1. Usable in a workstation environment – a single user (or small number of users) development station, either for general software, or software targeted for an embedded application.

2. Usable in embedded applications – placing the MIPS M/500 processor chip on board a platform for real-time analysis and control.

3. Usable in included applications – using the MIPS M/500 in a network (either as a stand alone chip or as a workstation) with other, potentially different, processors.

3. Analysis of MIPS Assembler Reorganizer

The MIPS assembler reorganizer is the system program that takes MIPS assembly language instructions and translates them into the MIPS M/500 native machine code. As one of its side functions, it also reorganizes the machine code to eliminate the `nop` instructions that must follow instructions such as branches and jumps may be eliminated. This reorganization takes advantage of the pipeline nature of the MIPS M/500 hardware. That is, once an instruction is loaded in the pipeline, it will be executed. This unconditional execution occurs in spite of any jumps or branches that may be taken.

3.1. Assembly Reorganization

As a simple example of assembly reorganization, consider the instruction sequence shown in figure 3-1. In this simple example, the numbers in registers $5 and $6 are subtracted and the result is placed into register $4.[3] If the result of the subtraction is non-zero, branch to the label `foo` otherwise, increment register $4 by 1 and continue.

```
sub     $4,$5,$6
bne     $4,$0,foo
add     $4,1
```

Figure 3-1: Sample Assembler Input

The first and last instructions take one clock cycle each. However, the branch instruction takes two clock cycles – one to determine whether the condition is true or not, and another to load the program counter with the address of the new instruction if the condition is true.[4] Thus, given the assembler input in figure 3-1, the assembler would generate the machine language code seen in figure 3-2.

```
sub     a0,a1,a2
bne     a0,zero,foo
nop
addi    a0,1
```

Figure 3-2: Sample Machine Language Output

The assembler reorganizer has added a `nop` instruction following the conditional branch. Since the cycle following the branch instruction can be filled with an instruction, the assembler reorganizer must take care to ensure that the `addi` instruction[5] is executed *only if* the branch is not taken. Since the destination of the subtract instruction is the source operand of the comparison, the reorganizer cannot perform any assembly reorganization. However, consider the sample assembler source in figure 3-3.

In this case, we have changed the source of the conditional branch to register $5, which is not a

---

[3]These register names will be changed to the logical names `a1`, `a2`, and `a0`, respectively, by the disassembler. However, the location of the registers is the same, regardless of their names. The list of register number to register name mappings is found in table A-1 in appendix A.

[4]The second cycle is expended whether or not the branch is taken. Additionally, it is worth mentioning that the new program counter is loaded at the *end* of the second cycle, so that the whole cycle may be filled with an instruction execution.

[5]Note the change of instruction name between the MIPS assembler input and the MIPS M/500 machine language output.

```
sub      $4,$5,$6
bne      $5,$0,foo
add      $4,1
```

Figure 3-3: Sample Assembler Input

direct result of the subtract instruction. What the assembler reorganizer will produce, given this input, is shown in figure 3-4.

```
bne      a1,zero,foo
sub      a0,a1,a2
addi     a0,1
```

Figure 3-4: Sample Reorganizer Output

Notice that the order of the MIPS M/500 machine instructions is no longer the same as that of the MIPS assembly language input. In fact, it would appear that the subtraction occurs only *after* the branch is rejected. This is not the case, however. Recall that the branch instruction always takes two cycles to execute, and that the instruction following the branch is *always* executed regardless of whether or not the branch is taken. Therefore, even though the instruction stream does not look correct, it is correct. The subtract instruction is always executed, whether or not the branch is taken. Thus, register $4 (that is, a0) will always have the correct result in it, and the addi instruction is only executed if the branch is not taken.

3.2. Translation of MIPS Assembly Instructions

As mentioned earlier, the assembler reorganizer will change the name of an assembler instruction to match the MIPS M/500 native machine language. It is not the case, however, that every MIPS assembly instruction has a corresponding MIPS M/500 native machine language instruction. Sometimes (as is the case for conditional branches), the inverse condition is tested with reversed arguments, at no extra instruction count expense. Often, however, multiple MIPS M/500 native instructions are substituted for a single MIPS assembler instruction. Consider the example shown in figure 3-5. In this case, we have substituted the the mulo (multiply with overflow) instruction for sub instruction.

```
mulo     $4,$5,$6
bge      $4,$0,foo
add      $4,1
```

Figure 3-5: Sample Assembler Input

What happens in this case (as shown in figure 3-6) is that the assembler reorganizer translates the single mulo instruction into a sequence of 8 MIPS M/500 native machine language instructions. The additional instructions are required to effect the overflow checking that the documentation for the mulo instruction advertises (as being part of a single instruction). The net effect of this legerdemain is that what appears to be a single instruction (taking a single machine cycle) is instead a sequence of 8 instructions taking 24 cycles to execute.

Compilers (such as those for strongly typed languages like Ada™) are not required to use the mulo instruction and may implement their own overflow checking software (see section 3.2.1). In general, however, the instruction counts obtained from the assembler output of compilers is not to be trusted

```
mult    a1,a2
mflo    a0
sra     a0,a0,31
mfhi    at
beq     a0,at,0x1c
mflo    a0
break   6
nop
bne     a0,zero,foo
nop
add     a0,1
```

Figure 3-6: Machine Language Output

as a measure of execution cycles (see section 4.1 on Ackermann's function for a discussion of this subject). Instead, the actual executable image must must be examined to determine *exactly* what instructions will be executed.[6] A comprehensive table of all instruction translations (and accompanying commentary) is in appendix A. The reader is strongly encouraged to read this appendix to correctly understand the translation from the MIPS high level instruction set to the MIPS M/500 native instruction set.

3.2.1. Interesting Effects of Multiplication

The MIPS instruction set provides a number of different multiply and divide instructions. Although most instructions on the MIPS M/500 take only a single cycle to execute, the multiply and divide instructions take far longer. Thus, it is in the best interest of the execution speed for the assembler/reorganizer to change multiply instructions into sequences of shifts and adds or subtracts. The only time this is valid is when the value of one of the multiplicands is known (i.e., it is a constant value). The assembler/reorganizer will substitute the appropriate sequence of simpler instructions *only* when the execution time of a multiply exceeds that of a sequence of shifts and adds. When neither of the multiplicands is a constant value, the assembler/reorganizer uses the appropriate MIPS M/500 multiplication instruction.[7]

In the worst case, the number of instructions that will be generated for a multiply are *n-1* adds and *n* shifts, where *n* is the number of 1 bits that are present in the constant multiplier. Thus, to multiply by the constant value 42, the instruction

```
mul    $15,$14,42
```

is converted to the sequence shown in figure 3-7. The number 42 (or 2#101010) has three 1 bits, and so the number of instructions is 3 shifts and 2 adds.

Where there are runs of 1's with no intervening 0's, the number of instructions is reduced. Multiplying by 79, for example, produces only 2 shifts and two adds (one of the adds is a subtraction), even though 79 (or 2#1001111) contains 5 bits that are 1. The MIPS M/500 code is shown in figure 3-8.

_____

[6]Altering a single instruction may subtly change the actions of the assembler reorganizer, and critical sections of code must be examined with great care following any modification.

[7]Shifts and adds can always be used for multiplication. The problem is that there is a large chance that the time it takes to execute these instructions is greater than the multiply instruction. When both the multiplier and the multiplicand are constant values, the compiler precalculates the value instead of generating runtime code to perform the function.

```
0x0:    000e7880        sll     t7,t6,2
0x4:    01ee7821        addu    t7,t7,t6
0x8:    000f7880        sll     t7,t7,2
0xc:    01ee7821        addu    t7,t7,t6
0x10:   000f7840        sll     t7,t7,1
```

Figure 3-7: MIPS M/500 Code for Multiplication by 42

```
0x0:    000e7880        sll     t7,t6,2
0x4:    01ee7821        addu    t7,t7,t6
0x8:    000f7900        sll     t7,t7,4
0xc:    01ee7823        subu    t7,t7,t6
```

Figure 3-8: MIPS M/500 Code for Multiplication by 79

Figure 3-9 shows a worst case expansion – a multiplication by 2730 (or 2#101010101010), which contains 6 discontiguous 1 bits. In this example, $n = 6$, and a single multiply is expanded to 6 shifts and 5 adds.

```
0x0:    000e7880        sll     t7,t6,2
0x4:    01ee7821        addu    t7,t7,t6
0x8:    000f7880        sll     t7,t7,2
0xc:    01ee7821        addu    t7,t7,t6
0x10:   000f7880        sll     t7,t7,2
0x14:   01ee7821        addu    t7,t7,t6
0x18:   000f7880        sll     t7,t7,2
0x1c:   01ee7821        addu    t7,t7,t6
0x20:   000f7880        sll     t7,t7,2
0x24:   01ee7821        addu    t7,t7,t6
0x28:   000f7840        sll     t7,t7,1
```

Figure 3-9: MIPS M/500 Code for Multiplication by 2730

For many users of the MIPS M/500, however, this scheme presents an interesting set of problems.

1. Obviously, when minimizing code size is a paramount consideration, multiplications can cause image code size to grow. Since the speed/space tradeoff of the assembler/reorganizer is weighted on speed, multiply instructions are allowed to grow to 14 times their original size.

2. Algorithms written in different languages may run at vastly different speeds. Languages may implement constant values in different ways; thus, multiplications may be implemented in different ways. Multiplying by the constant value 2 takes substantially less time than multiplication by a variable containing the value 2.

3. A good compiler may actually generate code that runs slower than a bad compiler. A compiler that compresses arithmetic expressions to eliminate spurious multiplies may create code that expands into a larger sequence of shifts and adds than a compiler that does not do compression (see section 7.7).

4. Altering a constant value (e.g., a C `#define` constant) may change the size and speed of a program, even though the variable does not affect the number of iterations in loops.

Users of the MIPS assembler/reorganizer must therefore be very careful when generating space-critical or speed-critical code. Figure 3-10 shows the time required to perform a multiply by a constant value and by a variable using the `mul` instruction (the timing will be different for the `mulo` instruction).

Figure 3-10: Relative Multiplication Speeds

Notice the widely varying times required to perform the multiplications. The two curves represent constant values ranging from 0 to 100 for the solid curve, and from 2700 to 2800 for the stippled curve. The solid line at the top of the graph is the time required to perform a multiply using the actual `mul` instruction. Note that the time required to execute the shifts and adds never exceeds this time.

MIPS therefore has taken pains to correctly weight the `mul` instruction expansion by the assembler/reorganizer. This will *usually* result in faster program execution (except in cases similar to that shown in section 7.7), although predicting actual execution time can be difficult. When this is of critical importance, actual instruction counting must be done.

3.2.2. Retargeting of Branch Instructions

One interesting effect of the assembler/reorganizer is that it will occasionally re-target a branch instruction. This re-targeting will occur when at least the following three conditions are true:

1. The delay slot that must follow the branch cannot be filled with an instruction from immediately before the branch.

2. The original target of the branch is not relatively relocatable – that is, the target must be within the same module. Jump instructions that refer to addresses outside of the local scope are ineligible.

3. The targeted instruction must not cause an exception.

When these conditions are met, the assembler/reorganizer will fill the delay slot following the branch instruction with the instruction that was originally targeted by the branch, and will move the target of the branch to the next instruction following the original branch target. Consider the example source code shown in figure 3-11.

```
        .ent    foo 2
foo:
        bge     $2, 0, $41
        negu    $2, $2
$41:
        subu    $24, $18, $17
        beq     $24, $2, $43
        negu    $3, $3
$43:
        jal     foo
        .end    foo
```

Figure 3-11: Example of Branch Target Relocation – Assembler Source

In this example, the negation of register **$2** is only performed if **$2** is less than 0. Otherwise, a branch is executed to label **$41**, which subtracts registers **$17** and **$18**. This is then followed by another conditional branch and another negation. One might expect that this code fragment would yield two **nop** instructions, one following each of the branch instructions.[8] However, as can be seen in figure 3-12, this is not the case.

```
        foo:
0x0:    04410003        bgez    v0,0x10
0x4:    0251c023        subu    t8,s2,s1
0x8:    00021023        subu    v0,zero,v0
0xc:    0251c023        subu    t8,s2,s1
0x10:   13020002        beq     t8,v0,0x1c
0x14:   00000000        nop
0x18:   00031823        subu    v1,zero,v1
0x1c:   0c000000        jal     0
0x20:   00000000        nop
```

Figure 3-12: Example of Branch Target Relocation – MIPS M/500 Output

The anticipated second **nop** instruction is present at address 0x18, but the first delay slot has been filled with the original target of the branch instruction, and the branch target has been moved from 0xc to 0x10. The reader is encouraged to trace the control flow of this fragment (remembering the rules of reorganization around branch instructions) to verify that the output of the assembler/reorganizer is correct. Notice that location 0x4 contains the same instruction as location 0xc (the original target). However, only one of these instructions is ever executed.

Notice that this reorganization technique does not reduce the size of the program at all (nor does it increase it). It does, however, speed up program execution by substituting nop instructions with other "real" instructions; this is especially effective when the delay slot following a branch back to the top of a loop can be filled with the first instruction of the loop (ρ-motion). The assembler/reorganizer could also decrease program size with this technique. All of the "come from" points of an instruction are known to the assembler.[9] If an instruction has no "come from" points (that is, no instruction will "fall through" to the instruction, and all branches have been retargeted), then that instruction may be

---

[8]The first would be seen because the negate follows the branch in the original instruction stream. The second would be present because the branch is contingent on the result of the subtraction, so the subtract can not be moved after the branch.

[9]A "come from" point is either a branch instruction that executes a "go to" an instruction, or a prior instruction that "falls through" to that instruction.

removed. In this case, the instruction at address 0xc will never be executed, and thus it could be elided by the assembler/reorganizer.

3.3. Local Conclusions

The MIPS user-level instruction set and the MIPS M/500 native instruction set are inherently similar, though radically different in some cases. Evaluating a compiler on the basis of the MIPS assembly code that it produces would therefore be a mistake. It is necessary to examine the reorganized native machine code produced by the assembler reorganizer. This has the disadvantage of presenting to the reader a somewhat confusing picture, because some instructions (such as branches and jumps) do not take effect immediately upon being scanned.

Also, although most instructions take a single cycle to execute, some instructions (notably the `mul` and `div` instructions, and co-processor instructions) take more than a single cycle. Evaluating the predicted worst case runtime of a section of code can therefore be tricky, even without considering the effects of the instruction cache (as discussed in section 5.3). Measurement is the only reliable guide, and even measurements need careful interpretation.

It also appears that, wherever the assembler writers thought it appropriate, special case code has been introduced to handle operands of zero. Since the assembler reorganizer is taking the liberty of effectively rewriting the assembly program into a functionally equivalent, though structurally different form, it is perfectly acceptable to interpret the constant value 0 and the zero register as identical. Unfortunately, it is all too often the case that the two are not treated equivalently. This, combined with the absence of many other special case tests (such as checking for an addend or dividend of 0), suggests a non-uniform approach to the assembler reorganizer. It seems that the assembler writers have considered each special test in line, rather than developing a rigorous solution to all of the special conditions.[10] The code that is generated by the assembler reorganizer is correct, although it is sometimes suboptimal.[11]

_____

[10]It could be argued that a "good" compiler would never generate code that uses many of these special cases (i.e., generating code that has a divisor or dividend of zero). It is usually on assumptions like these that catastrophes, and theses on catastrophe theory, are based. We discovered a number of examples of this type of failure in the course of our investigations.

[11]See, for example, the differences in code expansion for the `seq` instruction on page 174. For this instruction, a different set of instructions are generated for a source of the zero register and for the constant value 0, even though the two are identical values.

4. Analysis of Benchmarks

In general, benchmarks set out to do two things:

1. Produce some gross determination on the suitability of using given compiler generated code for a given processor by providing some measure of it's efficiency.

2. Determine the relative performance of various processors.

Regrettably, most published benchmarks fail to achieve these goals, and instead only report on a given processor's ability to run a specific benchmark. The people who publish benchmark statistics for a given machine are generally concentrating on the second factor only. By claiming that their machine can execute "273 deka-Floppystones," they divulge almost no useful information. Yet the notion of benchmarks as measures of performance is that we felt compelled to present some statistics, in spite of our feelings about their inapplicability.

The "art" of benchmarking is still in the stone age – the Whetstone and Dhrystone benchmarks were written with a specified mix of instructions in mind (as well as a specific compiler technology), and they test only that instruction mix. The Dhrystone benchmark even requires that certain optimizations *not* be used when compiling the benchmark to most effectively test the features for which it was designed.

A benchmark really tests two things:

1. A compiler's effectiveness in generating machine code from source language.

2. The hardware's speed in executing that code.

These two parts are inseparable halves of the whole – one may not eliminate either part, but must examine both the generated machine code and the speed at which it is executed. In restricting the level of optimization that may be used, the Dhrystone benchmark considers only one half of the compiler/machine couplet. If a given compiler has features which enable it to process source language in an efficient way, those features should be tested in the benchmark since they will also be used in real life. On the MIPS M/500, these features include:

- cross-module optimization

- interprocedure register allocation

- routine in lining (hoisting).[12]

We believe that these are valuable compiler functions and therefore have gathered all of our benchmark statistics with these features enabled.

---

[12]Routine inlining is the process of removing a routine call and substituting it with the body of the routine. This action is also called *routine hoisting* and increases the speed of a program by removing the overhead of parameter passing and routine calling. When a routine is called from only one place in a program, routine inlining almost always results in a performance improvement. However, as the number of call sites for a routine increases, the performance improvement begins to be offset by an increased program image size. The decision to inline a routine is usually based on the number of call sites, the size of the routine body versus the size of the call and return sequence, and on various specifics of register allocation.

We present in this chapter the results and analyzes of four benchmarks.

1. Ackermann's Function [Wichmann 76] – this deceptively simple function is used to examine the behavior of the compiler on a well-known fundamental problem. Ackermann's Function is a highly recursive function that serves no "useful" purpose in that it does not calculate anything of importance. However, the way in which a compiler generates code for this function can be fairly easily reduced to a pair of meaningful numbers. We evaluate these numbers and comment on their significance.

2. Whetstones [Curnow 76] – one of the numbers that hardware manufacturers like to publicize to show off their computer's efficiency. In our opinion, all that this benchmark measures is how efficiently a compiler/computer pair can execute the Whetstone benchmark (and not how fast they can execute a real floating-point program). However, since it is customary to measure this aspect of a computer's performance, we provide (again, with a careful analysis) the results of the MIPS M/500's performance in this benchmark.

3. Dhrystones [Weicker 84] – another of the numbers that is produced to tout a computer's performance. The Dhrystone measure concentrates on integer operations of a mix calculated to simulate average integer programs. Unfortunately, it presents an artificial picture of routine loading and parameter passing.

4. 20 Queens – a small integer-based program that calculates a mutually non-threatening placement of twenty queens on a 20 x 20 chessboard. This benchmark was chosen because it, too, was small enough to analyze in detail. The relative run times at the various levels of optimization are presented to give a feel for optimizer efficiency on this small scale.

We examined numerous other benchmarks. Some of the standard ones that we rejected are:

- The CMU MCF benchmark suite [Barbacci 78] – these benchmarks are designed to test the efficiency of numerous military processors by having humans write the most efficient assembly code they could to perform a number of functions, including:

    • character string search

    • integer array manipulation

    • linked list insertion

    • character to floating-point conversion.

    • record packing and unpacking

    These benchmarks were never executed in the original tests, but they measured the applicability of different instruction sets to these tasks. The results of the evaluation consisted of measuring the memory and register usage based on a high-level simulation of the machines on real hardware and not execution speed. These benchmarks are much too small to consider alone.

- Quicksort – While this is a reasonable function to test for, the Quicksort algorithm is so small that it does not really test the efficiency of the compiler. Also, it is somewhat data dependent, so that even a machine-independent set of data does not really test the algorithm.

- FFT – Rejected for the same reason as Quicksort.

- BYTE benchmarks – Rejected for the same reason as Quicksort.

- EUUG benchmarks – These benchmarks showed that the MIPS M/500 is useful as a workstation, but the statistics that they are of little significance.

In all cases, it should be remembered that benchmarks are useless unless a detailed analysis of the *reasons* for their performance is conducted. Simply presenting a set of unrelated numbers tells nothing about a machine. A benchmark's behavior on a given machine is also highly correlated with the efficiency of the compiler that is generating code for it, and ignoring the compiler's effect ignores the truth.

Readers are also cautioned to first read section 5.3 before attempting to generate or execute benchmarks on their own. It is insufficient to run a benchmark once or twice to determine its execution speed. The graphs shown in figures 5-1 and 5-2 are the distillation of data acquired from running 768 different programs a total of 4608 times. The graphs shown in figures 5-3 and 5-4 are pictures of the data collected from 520 individual programs executed over 6000 times. The primary reason for this huge collection of data was to eliminate any special factors which could influence the run time of the test programs. Many factors influence the execution speed of a benchmark; simply asking all other users to log off is insufficient. As with the results of a benchmark, the ancillary influencing factors must also be analyzed before any meaningful results can be extracted from the mass of data.

4.1. Ackermann's Function

Ackermann's Function is a reasonable measure of the efficiency of a compiler's treatment of routine calls and the associated integer arithmetic. It is a useful benchmark in that it can be used to simply quantify (without running the program) the performance of a compiler.

4.1.1. Method of Analysis

The first number that can be derived from the output of a compiler is the size (in bytes) of the generated code. This number gives a reasonable handle on the overall efficiency of a compiler, particularly compared to other compilers for machines with similar instruction set complexity.

The second number is a fair indicator of the speed of the generated code. This number is the average of the number of instructions needed to execute either the first or third leg of the conditional expression comprising Ackermann's function (see figure 4-1 for a statement of the function). The average of the first and third legs of the conditional is used because these comprise the predominant run-time load of the function.[13]

Ideally, the lower the numbers for both measures, the better the compiler. This generalization, however, can be misleading. For example, the VAX calls instruction is very expensive, yet clever usage of it can reduce the second measure considerably, at very little improvement in run-time performance. We will attempt to objectively evaluate the performance of the MIPS C and Pascal

---

[13]The first and third legs are executed nearly the same number of times, which are disproportionately frequent compared to the second leg. For acker(3,8), the first leg is executed 1,391,982 times and the third leg is executed 1,391,981 times, while the second leg is executed only 2,036 times. Since the second leg accounts for only 0.073% of the total function load, it may be ignored.

```
acker(n,m)
{
    if (n == 0)
        return m+1;
    else if (m == 0)
        return acker(n-1,1);
    else
        return acker(n-1,acker(n,m-1));
}
```

Figure 4-1: C Source Code for Ackermann's Function

compilers and contrast them to comparable compilers on comparable architectures.[14]

```
function acker(n,m : integer) : integer;
begin
    if n = 0 then
        acker := m+1
    else if m = 0 then
        acker := acker(n-1,1)
    else
        acker := acker(n-1,acker(n,m-1));
end;
```

Figure 4-2: Pascal Source Code for Ackermann's Function

4.1.2. Analysis of C and Pascal

The C and Pascal source code for Ackermann's Function is shown in figures 4-1 and 4-2, respectively. The assembly language output of the C compiler[15] (seen in figure 4-3) shows a total byte count of 92 (23 instructions of 4 bytes each), with a mean instruction count of 14.[16] To show how this latter number is arrived at, we have added the tag **"[1]"** for instructions that are executed when the first leg of the conditional is executed, and the tag **"[3]"** for those that are executed when the third leg is executed.

The numbers for C compare quite favorably with the other architectures and compilers evaluated by Wichmann. Given that the architecture of the MIPS M/500 is RISC in nature, these values are *very* good (in fact, they are quite respectable for CISC architectures, too). However, these accolades must be held in abeyance for a little while. As discussed in section 3.2, the instructions that are emitted by the code generator are not necessarily the instructions that are executed by the MIPS M/500. Since the MIPS M/500 native instruction set is not identical to the MIPS assembly language, we must use the disassembler to look at the actual machine language image before we can come up with accurate values for the Ackermann Function analysis.

Figure 4-4 shows the actual MIPS M/500 native machine code that is executed for Ackermann's Function when compiled with the C compiler at optimization level 2.

---

[14]Brian Wichmann has accumulated many measurements of the code generated for Ackermann's function in [Wichmann 82]. References to other compilers are from that report.

[15]This example was compiled with the **-o** switch, which selects optimization level 2. There is no extra benefit for optimization levels 3 or 4 for a simple function like this.

[16]There are 11 instructions in the first leg, 17 in the third, with an average of (11+17)/2=14.

```
 #   1   acker(n,m)
 #   2   {
acker:   subu    $sp, 24         [1][3]
         sw      $31, 20($sp)    [1][3]
         sw      $16, 16($sp)    [1][3]
         move    $16, $4         [1][3]
         move    $3, $5          [1][3]
 #   3       if (n == 0)
         bne     $16, 0, $32     [1][3]
 #   4           return m+1;
         addu    $2, $3, 1       [1]
         b       $34             [1]
 #   5       else if (m == 0)
$32:     bne     $3, 0, $33          [3]
 #   6           return acker(n-1,1);
         addu    $4, $16, -1
         li      $5, 1
         jal     acker
         b       $34
 #   7       else
 #   8           return acker(n-1,acker(n,m-1));
$33:     move    $4, $16             [3]
         addu    $5, $3, -1          [3]
         jal     acker               [3]
         addu    $4, $16, -1         [3]
         move    $5, $2              [3]
         jal     acker               [3]
$34:     lw      $16, 16($sp)    [1][3]
         lw      $31, 20($sp)    [1][3]
         addu    $sp, 24         [1][3]
         j       $31             [1][3]
```

Figure 4-3: Assembly Output from the C Compiler

```
         acker:
0x0:     27bdffe8        addiu   sp,sp,-24       [1] [3]
0x4:     afbf0014        sw      ra,20(sp)       [1] [3]
0x8:     afb00010        sw      s0,16(sp)       [1] [3]
0xc:     00808021        move    s0,a0           [1] [3]
0x10:    16000003        bne     s0,zero,0x20    [1] [3]
0x14:    00a01821        move    v1,a1           [1] [3]
0x18:    1000000e        b       0x54            [1]
0x1c:    24620001        addiu   v0,v1,1         [1]
0x20:    14600007        bne     v1,zero,0x40        [3]
0x24:    02002021        move    a0,s0               [3]
0x28:    2604ffff        addiu   a0,s0,-1
0x2c:    0c000000        jal     acker
0x30:    24050001        li      a1,1
0x34:    10000008        b       0x58
0x38:    8fbf0014        lw      ra,20(sp)
0x3c:    02002021        move    a0,s0
0x40:    0c000000        jal     acker               [3]
0x44:    2465ffff        addiu   a1,v1,-1            [3]
0x48:    2604ffff        addiu   a0,s0,-1            [3]
0x4c:    0c000000        jal     acker               [3]
0x50:    00402821        move    a1,v0               [3]
0x54:    8fbf0014        lw      ra,20(sp)       [1] [3]
0x58:    8fb00010        lw      s0,16(sp)       [1] [3]
0x5c:    03e00008        jr      ra              [1] [3]
0x60:    27bd0018        addiu   sp,sp,24        [1] [3]
```

Figure 4-4: MIPS M/500 Native Machine Code for Ackermann's Function

---

In this case, we come up with a total byte count of 96 (24 instructions of 4 bytes each), with a mean instruction count of 14.5.[17] To show how this latter number is arrived at, we have again added the tag **"[1]"** for instructions that are executed when the first leg of the conditional is executed, and the tag **"[3]"** for those that are executed when the third leg is executed.[18] The counts have increased somewhat, although not markedly. Still, because the instructions that are actually executed by the MIPS M/500 are different from those emitted by the code generator, one must be careful when evaluating the expected run-time of any program. In this case, the time increase is a little more than 3.5%, but there are cases in which a single MIPS assembly language instruction will be expanded to 12 or more times its original size when converted to MIPS M/500 native instructions. (See the table of instruction conversions starting on page 144 for more details on this feature of the assembler reorganizer.)

	Optimization Level				
	–O0	–O1	–O2	–O3	–O4
Byte Count	192	176	100	100	100
Instruction Average	27	18.5	14.5	14.5	14.5

Table 4-1: C Compiler Efficiency Measures Using Ackermann's Function

The values shown in tables 4-1 and 4-2 show the code size and average number of instructions executed for the C and Pascal versions of Ackermann's Function at varying levels of optimization.

	Optimization Level				
	–O0	–O1	–O2	–O3	–O4
Byte Count	200	152	108	108	108
Instruction Average	29	21	16	16	16

Table 4-2: Pascal Compiler Efficiency Measures Using Ackermann's Function

---

[17]There are 12 instructions in the first leg, 17 in the third, with an average of (12+17)/2=14.5

[18]See section 3.1 for an explanation as to why the instructions after the branches are still considered in the instruction counts.

The marked improvement for both compilers for optimization level 2 over optimization 0 demonstrates conclusively the positive effects of an optimizer for even simple programs like this. In fact, even optimization level 1 causes a noticeable shrinkage in code size and execution count.[19] Optimization levels 3 and 4 (which are fairly sophisticated) do not have any effect on programs that are this simple.[20]

```
          acker:
0x0:    27bdffe0        addiu   sp,sp,-32
0x4:    afbf001c        sw      ra,28(sp)
0x8:    afb00014        sw      s0,20(sp)
0xc:    afb10018        sw      s1,24(sp)
0x10:   00808021        move    s0,a0
0x14:   00a03021        move    a2,a1
0x18:   16000003        bne     s0,zero,0x28
0x1c:   00408821        move    s1,v0
0x20:   10000012        b       0x6c
0x24:   24c30001        addiu   v1,a2,1
0x28:   14c00008        bne     a2,zero,0x4c
0x2c:   02002021        move    a0,s0
0x30:   2604ffff        addiu   a0,s0,-1
0x34:   24050001        li      a1,1
0x38:   0c000000        jal     acker
0x3c:   02201021        move    v0,s1
0x40:   1000000a        b       0x6c
0x44:   00401821        move    v1,v0
0x48:   02002021        move    a0,s0
0x4c:   24c5ffff        addiu   a1,a2,-1
0x50:   0c000000        jal     acker
0x54:   02201021        move    v0,s1
0x58:   2604ffff        addiu   a0,s0,-1
0x5c:   00402821        move    a1,v0
0x60:   0c000000        jal     acker
0x64:   02201021        move    v0,s1
0x68:   00401821        move    v1,v0
0x6c:   00601021        move    v0,v1
0x70:   8fb00014        lw      s0,20(sp)
0x74:   8fbf001c        lw      ra,28(sp)
0x78:   8fb10018        lw      s1,24(sp)
0x7c:   03e00008        jr      ra
0x80:   27bd0020        addiu   sp,sp,32
```

Figure 4-5: MIPS M/500 Machine Language Output from Figure 4-2

---

[19]Optimization level 1 is performed by default, unless explicitly switched off.

[20]One optimization that would have an effect on this program is tail recursion elimination. The MIPS optimizer does not do this particular type of optimization. When this and other hand optimizations are performed on this module (specifically, improving the calling convention used by this routine and reducing the entry/exit protocol), the byte count can be reduced to 72, and the average instruction count to 9 (see section 4.1.4).

The values for C are noticeably better than those for Pascal,[21] even though we would predict that at level 4 optimization, there should be no difference between the two. The C code generator is apparently able to take advantage of the fact that the values returned do not need to be assigned to intermediate storage locations, while the Pascal compiler, being constrained to always store the return value of a function, does not. The Pascal compiler could have achieved equally good results by not making the mistake of using two registers (specifically $v0$ and $v1$) to hold return values and intermediate results, and then needing to perform a register shuffle. This shortcoming can be corrected by a better register tracking and assignment algorithm in the Pascal compiler.

4.1.3. Analysis of BCPL

The same analysis was performed for the BCPL version of Ackermann's Function. The BCPL source code is shown in figure 4-6.

```
LET acker(m,n) =

  m=0 -> n+1,
  n=0 -> acker(m-1,1),
          acker(m-1,acker(m,n-1))
```

Figure 4-6: BCPL Source for Ackermann's Function

The output of the BCPL compiler is shown in figure 4-7, with the same tagging notation used in the earlier examples. The front end of the compiler has performed code hoisting of the function return sequence so that each of the three arms ends with a direct return (instead of branching to the return sequence as is done with C and Pascal).

A total of 27 instructions is needed to implement the function. The average number of instructions per call is (6+17)/2 or 11.5. The compiler has only one (fairly low) level of optimization. Even so, these numbers are fairly good. However, figure 4-7 shows the unreorganized, high-level MIPS code. The reorganizer changes it to what is shown in figure 4-8.

The actual MIPS M/500 native code uses 32 instructions, or 128 bytes of code to implement the routine. The mean number of instructions per call is (8+20)/2 or 14. Note that the code could be improved by α-motion of the code at 0x28 and 0x48, and by eliminating the nop at 0x24.

---

[21]When the C compiler is coerced into recognizing common subexpressions by changing the code to read:

```
acker(n,m)
{
    if (n == 0)
        return m+1;
    else
        return acker(n-1, m==0 ? 1 : acker(n,m-1));
}
```

the total code size measure improves even more (reducing it to 88 bytes of code). While this last optimization does nothing to improve the execution speed of the routines (in fact, it hinders it somewhat, raising the average number of instructions executed to 15), it does reduce the overall size of the routine. This optimization, carried out over larger programs, will have the effect of reducing overall program size, and, hence, the amount of paging the system needs to perform. Since run-time may be increased, however, it is up to the program implementors to decide where the tradeoff is to be made. Ideally, both the C and Pascal compilers should recognize this inherent commonality, and should produce somewhat smaller code with no increase in execution speed. The results shown here, however, are still very favorable.

```
        #define u0  $2
        #define u1  $3
        #define rz  $0
        #define ru $16            # holds 1
        #define rp $22            # Ocode stack pointer
        #define rl $31


        LA1:
         sw     rl,0(rp)          [1] [3]
         sd     u0,8(rp)          [1] [3]
         bne    u0,rz,LA3         [1] [3]
         add    u0,u1,ru          [1]
         lw     rl,0(rp)          [1]
         j      rl                [1]
        LA3:
         bne    u1,rz,LA5             [3]
         sub    u0,u0,ru
         move   u1,ru
         add    rp,16
         bal    LA1
         lw     rl,(0-16)(rp)
         sub    rp,16
         j      rl
        LA5:
         sub    u0,u0,ru              [3]
         sw     u0,24(rp)             [3]
         sub    u1,u1,ru              [3]
         lw     u0,8(rp)              [3]
         add    rp,28                 [3]
         bal    LA1                   [3]
         move   u1,u0                 [3]
         lw     u0,(24-28)(rp)        [3]
         sub    rp,12                 [3]
         bal    LA1                   [3]
         lw     rl,(0-16)(rp)         [3]
         sub    rp,16                 [3]
         j      rl                    [3]
```

Figure 4-7: Assembly Language Output from BCPL Compiler

4.1.4. Results of Hand Coding in MIPS Assembly Language

The hand coded version, which is the best we can currently come up with, is seen in figure 4-9. This is reorganized into what we see in figure 4-10.

The hand optimized code results in 18 instructions (or 72 bytes), and the mean number of instructions per call is (4+14)/2 or 9. This is a substantial improvement over the MIPS C compiler and the BCPL compiler, and reflects the power that a compiler could achieve with the proper optimizations. The special optimizations that were done are:

- Tail recursion elimination

- Procedure call protocol elimination

- Writing the MIPS assembly language to eliminate all possible nop instructions. It would have been easier to write directly in the MIPS M/500 native instruction set, but this option was not available to us.

```
0x0:     aedf0000          sw     ra,0(s6)        [1] [3]
0x4:     aec20008          sw     v0,8(s6)        [1] [3]
0x8:     14400005          bne    v0,zero,0x20    [1] [3]
0xc:     aec3000c          sw     v1,12(s6)       [1] [3]
0x10:    8edf0000          lw     ra,0(s6)        [1]
0x14:    00701020          add    v0,v1,s0        [1]
0x18:    03e00008          jr     ra              [1]
0x1c:    00000000          nop                    [1]
0x20:    14600009          bne    v1,zero,0x48        [3]
0x24:    00000000          nop                        [3]
0x28:    00501022          sub    v0,v0,s0
0x2c:    02001821          move   v1,s0
0x30:    0411fff3          bgezal zero,0x0
0x34:    22d60010          addi   s6,s6,16
0x38:    8edffff0          lw     ra,-16(s6)
0x3c:    22d6fff0          addi   s6,s6,-16
0x40:    03e00008          jr     ra
0x44:    00000000          nop
0x48:    00501022          sub    v0,v0,s0            [3]
0x4c:    aec20018          sw     v0,24(s6)           [3]
0x50:    00701822          sub    v1,v1,s0            [3]
0x54:    8ec20008          lw     v0,8(s6)            [3]
0x58:    0411ffe9          bgezal zero,0x0            [3]
0x5c:    22d6001c          addi   s6,s6,28            [3]
0x60:    00401821          move   v1,v0               [3]
0x64:    8ec2fffc          lw     v0,-4(s6)           [3]
0x68:    0411ffe5          bgezal zero,0x0            [3]
0x6c:    22d6fff4          addi   s6,s6,-12           [3]
0x70:    8edffff0          lw     ra,-16(s6)          [3]
0x74:    22d6fff0          addi   s6,s6,-16           [3]
0x78:    03e00008          jr     ra                  [3]
0x7c:    00000000          nop                        [3]
```

Figure 4-8: Machine Language Output from Figure 4-7

```
LA1:
  sw     $31,0($22)       # no need to store parameters yet
  bne    $2,$0,LA3
  add    $2,$3,$16
  j      $31              # Return
LA3:
  sub    $2,$2,$16        # moved up to fill branch slot
  bne    $3,$0,LA4
  move   $3,$16
  b      LA1              # Call (tail recursion elimination)
LA4:
  sw     $2,8($22)        # save across inner call
  sub    $3,$3,$16
  add    $2,$2,$16
  add    $22,12           # true call follows - must move stack
  bal    LA1
  move   $3,$2
  lw     $2,(8-12)($22)
  lw     $31,(0-12)($22)
  sub    $22,12           # should fill branch slot
  b      LA1              # tail recursion elimination
```

Figure 4-9: Hand Optimized Version of Ackermann's Function

```
0x0:    14400003        bne     v0,zero,0x10    [1] [3]
0x4:    aedf0000        sw      ra,0(s6)        [1] [3]
0x8:    03e00008        jr      ra              [1]
0xc:    00701020        add     v0,v1,s0        [1]
0x10:   14600003        bne     v1,zero,0x20        [3]
0x14:   00501022        sub     v0,v0,s0            [3]
0x18:   1000fff9        b       0x0
0x1c:   02001821        move    v1,s0
0x20:   00701822        sub     v1,v1,s0            [3]
0x24:   aec20008        sw      v0,8(s6)            [3]
0x28:   00501020        add     v0,v0,s0            [3]
0x2c:   0411fff4        bgezal  zero,0x0            [3]
0x30:   22d6000c        addi    s6,s6,12            [3]
0x34:   00401821        move    v1,v0               [3]
0x38:   8ec2fffc        lw      v0,-4(s6)           [3]
0x3c:   8edffff4        lw      ra,-12(s6)          [3]
0x40:   1000ffef        b       0x0                 [3]
0x44:   22d6fff4        addi    s6,s6,-12           [3]
```

Figure 4-10: Machine Language Output for Figure 4-9

4.1.5. Comparison

Table 4-3 gives the results for each language analyzed. It shows the total size of the function in bytes, the average number of instructions per call, and the time to execute **acker(3,8)**. In all cases, the figures are for the reorganized code (i.e., the native MIPS M/500 code), not the high level assembly language.

Language	Function Size (bytes)	Instructions per Call	Execution Time of acker(3,8) (seconds)
C	100	14.5	5.6
Pascal	108	16	9.0
BCPL	128	14	6.1
Assembler	72	9	3.8

Table 4-3: Summary of Statistics For Ackermann's Function

4.1.6. Local Conclusions

All things considered, the measures obtained by analyzing the compilers' treatment of Ackermann's Function are quite favorable, especially for a RISC-style architecture. Clearly, the MIPS compilers and hardware are on to something. However, the hand optimizations shown in section 4.1.4 indicates that there is still a large degree of improvement that can be obtained. RISC architectures, especially when pipelined, can be tricky machines to generate code by. MIPS has done a very reasonable first pass at the development of a set of good compilers, (especially for a system that has been developed *ex nihilo*) and has demonstrated that a RISC architecture is a good choice. While the measure of Ackermann's Function is only one indication of the efficiency of a compiler and hardware combination, the results shown here are *very* promising. Nonetheless, MIPS needs to apply additional effort in their compiler team. %!PS-Adobe-1.0 md begin

F T -37 -34 873 669 87 72 72 1320 psu (Daniel V. Klein; document: Whetstone Chart)jn 0 mf od op 0

0 xl 1 1 pen 183 70 gm (nc 27 70 235 395 6 rc)kp 48 gr 183 394 lin 132 70 gm 132 394 lin 81 70 gm 81 394 lin 30 70 gm 30 394 lin 234 70 gm (nc 14 16 271 496 6 rc)kp 0 gr 29 70 lin 234 67 gm 234 72 lin 183 67 gm 183 72 lin 132 67 gm 132 72 lin 81 67 gm 81 72 lin 30 67 gm 30 72 lin 234 70 gm 234 394 lin 237 70 gm 232 70 lin 237 151 gm 232 151 lin 237 232 gm 232 232 lin 237 313 gm 232 313 lin 237 394 gm 232 394 lin 189 70 gm (nc 28 70 237 395 6 rc)kp 181 151 lin 180 151 gm 178 232 lin 175 313 lin 174 313 gm 144 394 lin 229 70 gm 217 151 lin 216 151 gm 200 232 lin 197 313 lin 196 313 gm 184 394 lin 207 70 gm 187 151 lin 186 151 gm 168 232 lin 170 313 lin 164 394 lin 102 70 gm 63 151 lin 62 151 gm 31 232 lin 35 313 lin 29 394 lin 205 70 gm 194 151 lin 193 151 gm 175 232 lin 171 313 lin 170 313 gm 143 394 lin 172 70 gm 160 151 lin 159 151 gm 140 232 lin 139 232 gm 137 313 lin 113 394 lin 202 70 gm 202 151 lin 203 151 gm 191 232 lin 190 232 gm 188 313 lin 187 313 gm 174 394 lin 189 70 gm (nc 24 66 238 399 6 rc)kp 64 gr 185 66 193 74 1 ov 0 gr 186 67 192 73 1 ov 180 151 gm 64 gr 176 147 184 155 1 ov 0 gr 177 148 183 154 1 ov 178 232 gm 64 gr 174 228 182 236 1 ov 0 gr 175 229 181 235 1 ov 174 313 gm 64 gr 170 309 178 317 1 ov 0 gr 171 310 177 316 1 ov 144 395 gm 64 gr 140 391 148 399 1 ov 0 gr 141 392 147 398 1 ov 229 70 gm 64 gr 225 66 233 74 1 ov 0 gr 226.5 67.5 231.5 72.5 0 ov 217 151 gm 64 gr 213 147 221 155 1 ov 0 gr 214.5 148.5 219.5 153.5 0 ov 200 232 gm 64 gr 196 228 204 236 1 ov 0 gr 197.5 229.5 202.5 234.5 0 ov 197 313 gm 64 gr 193 309 201 317 1 ov 0 gr 194.5 310.5 199.5 315.5 0 ov 184 395 gm 64 gr 180 391 188 399 1 ov 0 gr 181.5 392.5 186.5 397.5 0 ov 207 70 gm 64 gr 203 66 211 74 1 rc 0 gr 204 67 210 73 1 rc 187 151 gm 64 gr 183 147 191 155 1 rc 0 gr 184 148 190 154 1 rc 168 232 gm 64 gr 164 228 172 236 1 rc 0 gr 165 229 171 235 1 rc 170 313 gm 64 gr 166 309 174 317 1 rc 0 gr 167 310 173 316 1 rc 164 395 gm 64 gr 160 391 168 399 1 rc 0 gr 161 392 167 398 1 rc 102 70 gm 64 gr 98 66 106 74 1 rc 0 gr 99.5 67.5 104.5 72.5 0 rc 62 151 gm 64 gr 58 147 66 155 1 rc 0 gr 59.5 148.5 64.5 153.5 0 rc 31 232 gm 64 gr 27 228 35 236 1 rc 0 gr 28.5 229.5 33.5 234.5 0 rc 35 313 gm 64 gr 31 309 39 317 1 rc 0 gr 32.5 310.5 37.5 315.5 0 rc 29 395 gm 64 gr 25 391 33 399 1 rc 0 gr 26.5 392.5 31.5 397.5 0 rc 202 70 gm 64 gr pr 202 70 pl 210 74 pl 210 66 pl 202 70 pl 1 ep 203 70 gm 0 gr pr 203 70 pl 209 73 pl 209 67 pl 203 70 pl 1 ep 190 151 gm 64 gr pr 190 151 pl 198 155 pl 198 147 pl 190 151 pl 1 ep 191 151 gm 0 gr pr 191 151 pl 197 154 pl 197 148 pl 191 151 pl 1 ep 172 232 gm 64 gr pr 172 232 pl 180 236 pl 180 228 pl 172 232 pl 1 ep 173 232 gm 0 gr pr 173 232 pl 179 235 pl 179 229 pl 173 232 pl 1 ep 167 313 gm 64 gr pr 167 313 pl 175 317 pl 175 309 pl 167 313 pl 1 ep 168 313 gm 0 gr pr 168 313 pl 174 316 pl 174 310 pl 168 313 pl 1 ep 139 395 gm 64 gr pr 139 395 pl 147 399 pl 147 391 pl 139 395 pl 1 ep 140 395 gm 0 gr pr 140 395 pl 146 398 pl 146 392 pl 140 395 pl 1 ep 169 70 gm 64 gr pr 169 70 pl 177 74 pl 177 66 pl 169 70 pl 1 ep 170 70 gm 0 gr pr 170 70 pl 176 73 pl 176 67 pl 170 70 pl 1 ep 172 70 gm 64 gr pr 172 70 pl 175 71 pl 175 69 pl 172 70 pl 1 ep 156 151 gm pr 156 151 pl 164 155 pl 164 147 pl 156 151 pl 1 ep 157 151 gm 0 gr pr 157 151 pl 163 154 pl 163 148 pl 157 151 pl 1 ep 159 151 gm 64 gr pr 159 151 pl 162 152 pl 162 150 pl 159 151 pl 1 ep 136 232 gm pr 136 232 pl 144 236 pl 144 228 pl 136 232 pl 1 ep 137 232 gm 0 gr pr 137 232 pl 143 235 pl 143 229 pl 137 232 pl 1 ep 139 232 gm 64 gr pr 139 232 pl 142 233 pl 142 231 pl 139 232 pl 1 ep 133 313 gm pr 133 313 pl 141 317 pl 141 309 pl 133 313 pl 1 ep 134 313 gm 0 gr pr 134 313 pl 140 316 pl 140 310 pl 134 313 pl 1 ep 136 313 gm 64 gr pr 136 313 pl 139 314 pl 139 312 pl 136 313 pl 1 ep 109 395 gm pr 109 395 pl 117 399 pl 117 391 pl 109 395 pl 1 ep 110 395 gm 0 gr pr 110 395 pl 116 398 pl 116 392 pl 110 395 pl 1 ep 112 395 gm 64 gr pr 112 395 pl 115 396 pl 115 394 pl 112 395 pl 1 ep 202 70 gm 198 66 206 74 1 rc 199 67 gm 0 gr 204 72 lin 204 67 gm 199 72 lin 203 151 gm 64 gr 199 147 207 155 1 rc 200 148 gm 0 gr 205 153 lin 205 148 gm 200 153 lin 190 232 gm 64 gr 186 228 194 236 1 rc 187 229 gm 0 gr 192 234 lin 192 229 gm 187

234 lin 188 313 gm 64 gr 184 309 192 317 1 rc 185 310 gm 0 gr 190 315 lin 190 310 gm 185 315 lin 175 395 gm 64 gr 171 391 179 399 1 rc 172 392 gm 0 gr 177 397 lin 177 392 gm 172 397 lin 175 395 gm (nc 227 22 241 63 6 rc)kp 64 gr 228 22 241 62 4 rc 238 23 gm 0 gr 0 0[1 dtb 0 dfs 1.04 9 mul dfz bu fc 2 F /|_____Helvetica fnt bn pw dgf (1200.00)39 dam dkb (nc 176 22 190 63 6 rc)kp 64 gr 177 22 190 62 4 rc 187 23 gm 0 gr 0 0[1 dtb dgf (1500.00)39 dam dkb (nc 125 22 139 63 6 rc)kp 64 gr 126 22 139 62 4 rc 136 23 gm 0 gr 0 0[1 dtb dgf (1800.00)39 dam dkb (nc 74 22 88 63 6 rc)kp 64 gr 75 22 88 62 4 rc 85 23 gm 0 gr 0 0[1 dtb dgf (2100.00)39 dam dkb (nc 23 22 37 63 6 rc)kp 64 gr 24 22 37 62 4 rc 34 23 gm 0 gr 0 0[1 dtb dgf (2400.00)39 dam dkb (nc 241 61 255 80 6 rc)kp 64 gr 243 61 256 79 4 rc 253 62 gm 0 gr 0 0[1 dtb dgf (-O0)17 dam dkb (nc 241 142 255 161 6 rc)kp 64 gr 243 142 256 160 4 rc 253 143 gm 0 gr 0 0[1 dtb dgf (-O1)17 dam dkb (nc 241 223 255 242 6 rc)kp 64 gr 243 223 256 241 4 rc 253 224 gm 0 gr 0 0[1 dtb dgf (-O2)17 dam dkb (nc 241 304 255 323 6 rc)kp 64 gr 243 304 256 322 4 rc 253 305 gm 0 gr 0 0[1 dtb dgf (-O3)17 dam dkb (nc 241 385 255 404 6 rc)kp 64 gr 243 385 256 403 4 rc 253 386 gm 0 gr 0 0[1 dtb dgf (-O4)17 dam dkb (nc 0 0 701 622 6 rc)kp 76 418 198 495 4 rc 64 gr 70 412 192 489 4 rc 0 gr 70.5 412.5 191.5 488.5 0 rc (nc 71 413 191 488 6 rc)kp 64 gr 77 418 87 457 4 rc 80 419 gm (nc 77 418 87 457 6 rc)kp 0 gr 80 425 lin 80 422 gm 64 gr 76 418 84 426 1 ov 0 gr 77 419 83 425 1 ov 83 429 gm 0 0[1 dtb 6 dfz bu fc 2 F /|_____Helvetica fnt bn pw dgf (C Double)26 dam dkb (nc 71 413 191 488 6 rc)kp 64 gr 93 418 103 455 4 rc 96 419 gm (nc 93 418 103 455 6 rc)kp 0 gr 96 425 lin 96 422 gm 64 gr 92 418 100 426 1 ov 0 gr 93.5 419.5 98.5 424.5 0 ov 99 429 gm 0 0[1 dtb dgf (C Single)24 dam dkb (nc 71 413 191 488 6 rc)kp 64 gr 109 418 119 480 4 rc 112 419 gm (nc 109 418 119 480 6 rc)kp 0 gr 112 425 lin 112 422 gm 64 gr 108 418 116 426 1 rc 0 gr 109 419 115 425 1 rc 115 429 gm 0 0[1 dtb dgf (FORTRAN Double)49 dam dkb (nc 71 413 191 488 6 rc)kp 64 gr 126 418 136 478 4 rc 129 419 gm (nc 126 418 136 478 6 rc)kp 0 gr 129 425 lin 129 422 gm 64 gr 125 418 133 426 1 rc 0 gr 126.5 419.5 131.5 424.5 0 rc 132 429 gm 0 0[1 dtb dgf (FORTRAN Single)47 dam dkb (nc 71 413 191 488 6 rc)kp 64 gr 142 418 152 471 4 rc 145 419 gm (nc 142 418 152 471 6 rc)kp 0 gr 145 425 lin 141 422 gm 64 gr pr 141 422 pl 149 426 pl 149 418 pl 141 422 pl 1 ep 142 422 gm 0 gr pr 142 422 pl 148 425 pl 148 419 pl 142 422 pl 1 ep 148 429 gm 0 0[1 dtb dgf (Pascal Double)40 dam dkb (nc 71 413 191 488 6 rc)kp 64 gr 158 418 168 469 4 rc 161 419 gm (nc 158 418 168 469 6 rc)kp 0 gr 161 425 lin 157 422 gm 64 gr pr 157 422 pl 165 426 pl 165 418 pl 157 422 pl 1 ep 158 422 gm 0 gr pr 158 422 pl 164 425 pl 164 419 pl 158 422 pl 1 ep 160 422 gm 64 gr pr 160 422 pl 163 423 pl 163 421 pl 160 422 pl 1 ep 164 429 gm 0 gr 0 0[1 dtb dgf (Pascal Single)38 dam dkb (nc 71 413 191 488 6 rc)kp 64 gr 174 418 184 475 4 rc 177 419 gm (nc 174 418 184 475 6 rc)kp 0 gr 177 425 lin 177 422 gm 64 gr 173 418 181 426 1 rc 174 419 gm 0 gr 179 424 lin 179 419 gm 174 424 lin 180 429 gm 0 0[1 dtb dgf (C Forced Single)44 dam dkb (nc 256 190 270 275 6 rc)kp 64 gr 257 190 270 274 4 rc 267 191 gm 0 gr 0 0[1 dtb 1.04 9 mul dfz bu fc 2 F /|_____Helvetica fnt bn pw dgf (Optimization Level)83 dam dkb F T cp cd end

4.2. Dhrystone Benchmark

The Dhrystone benchmark is another artificial benchmark, constructed to measure the integer performance of a machine. Since it is an artificial benchmark, the results of this benchmark are of questionable value in analyzing the true performance of the MIPS M/500. As artificial benchmarks go, however, the Dhrystone benchmark appears to be a fairly reasonable test. One argument against it is that it has a fairly high percentage of routine calls, which unfairly biases the results against those

machines with an expensive procedure call interface. However, since practically all machine comparisons include the results of the Dhrystone benchmark, we felt it would be appropriate to include it in our analysis. The actual source code of the benchmark is not reproduced here.

4.2.1. Method of Analysis

The analysis of the data generated by the Dhrystone benchmark is usually interpreted as a straightforward measure of the hardware's efficiency in performing integer calculations. However, in truth there is a much more subtle interaction with the source language and the compiler's optimizing capabilities than most sources would admit. One feature of the MIPS compilers that serves it in good stead with this benchmark (and, of course, with real-life applications programs), is its ability to do routine hoisting (see page 15). This is especially true for this benchmark, which has a high percentage of procedure calls relative to actual computation.

To portray the MIPS M/500 as accurately as possible, the Dhrystone benchmark was executed in C at all levels of optimization.

4.2.2. Results

Table 4-4 shows the results obtained for the three languages at the highest level of optimization (along with the values for the VAX with the Berkeley 4.3 compiler for comparison purposes). The larger the value for the Dhrystone benchmark, the greater the machine/compiler performance.

MIPS C (Register)	14184
MIPS C (Non-Register)	14167
VAX C (Register)	*1394*
VAX C (Non-Register)	*1380*

Table 4-4: Dhrystone Numbers for MIPS and VAX

Again, the benchmark results demonstrate that the MIPS M/500 is a fast machine, clocking in at over 10 times faster than the MicroVAX. However, a lot of the MIPS speed (or actually, the VAX's lack of speed) is attributable to the high percentage of routine calls used in this benchmark. The Berkeley compiler uses the `calls` linkage exclusively, even though it is a very expensive subroutine linkage.

The most interesting aspect of this benchmark is the performance as the level of optimization is increased. As shown in figure 4-11, as the level of optimization is increased from level 0 (i.e., no optimization) to level 4 (the highest level). The optimizer nearly doubles the performance of the non-register Dhrystone benchmark.[22]

It is also interesting to note that the C compiler is faithfully observing the benchmark's request to put certain variables in registers. This is reflected in the fact that the register version of the benchmark runs faster than the non-register version with low level optimization. However, when the optimization level rises to level 2 (the first serious set of optimizations that are performed), the compiler ignores

---

[22]It is also worthy of note that even with all optimizations turned off, the MIPS M/500 is still able to execute the Dhrystone benchmark over 5 times faster than the *optimized* register version on the MicroVAX.

Figure 4-11: Dhrystone Benchmark Performance

the benchmark's requests, and decides for itself which variables belong in registers. The net effect is to immediately bring the non-register version of the benchmark up to par with the register version. This effectively proves that an automatic register allocator can be just as good (or better) a judge of which variables belong in registers.

The code size of the MIPS M/500 is only 25% larger than on the MicroVAX when at optimization level 2. This sort of code size expansion due to the more reduced instruction set complexity of the MIPS M/500 is predicted. However, when optimization level 4 is used (i.e., routine hoisting), the size of the MIPS M/500 executable image is 3% ***smaller*** than that of the VAX! This is a clear example of the desirability of a powerful compiler, and further exemplifies the applicability of a RISC architecture in any area.

4.2.3. Local Conclusions

Although the Whetstone benchmark leaves some room for improvement, the Dhrystone benchmark results show unequivocally that the concept of a RISC architecture is a viable one. The use of routine hoisting is a very valuable optimization, and the increase in performance obtained by simplifying the routine interface is substantial.

4.3. The Eight Queens Problem

In order to introduce some non-artificial benchmark statistics into the test suite for the MIPS machine, the classic problem of the Eight Queens was generated. In its pure form, the Eight Queens problem is to find a placement for eight queens on an 8 x 8 chessboard such that no piece threatens any other in a static placement. The problem may be generalized for the placement of n queens on an n x n chessboard. Although there are no solutions for $n = 2$ or $n = 3$, there exist solutions for $n = 4$

through at least $n = 26$. To bring execution time to a reasonable level (the complexity of the algorithm is O(n^3)), we chose $n = 20$ as our board size for running this benchmark.

```c
#ifdef ABS
int abs(i) int i;
{
    return (i<0?-i:i);
}
#endif

main()
{
    register int r, i, j, low, high;
    int row[20];

    for (i = 0; i < 20; i++)
        row[i] = -1;
    r = 0;
    while (r < 20) {                    /* Main loop */
        if (++row[r] == 20)             /* Nothing can go on this row */
            if (r == 0)
                break;                  /* Failure - no solution */
            else {
                row[r--] = -1;          /* Reset current row (for later) */
                continue;               /* Back up and try again */
            }
        for (i = r-1; i >= 0; i--) {
            if (row[i] == row[r])
                break;                  /* Test vertical */
#ifdef ABS
            if (abs(row[r]-row[i]) == r-i)
                break;                  /* Test both diagonals */
#else
            if (row[r]-row[i] == r-i)
                break;                  /* Test left diagonal */
            if (row[i]-row[r] == r-i)
                break;                  /* Test right diagonal */
#endif
        }
        if (i < 0)                      /* Loop completed, no collisions */
            r++;
    }
}
```

Figure 4-12: Source Code to 20 Queens Problem

The problem was solved using two similar algorithms, seen in the same body of code in figure 4-12. The conditional compilation bounded by the compile time constant ABS selects which method of examining the squares diagonal to the location are to be tested. If the constant expression ABS is FALSE, the diagonals are examined directly, first the left side, and then the right. If the constant expression ABS is TRUE, the diagonals are examined simultaneously through the use of the abs() routine. While the latter method shortcuts some evaluation, we would expect this version of the program to run slightly slower because of the additional overhead of a routine call. The actual values of the run-times in this test are unimportant, since we are not interested in using this test as a benchmark against other processors. What is important is the relative speeds of the two algorithms at the various optimization levels.

Figure 4-13: Runtime of 20 Queens Placement at Differing Optimization Levels

As shown in figure 4-13, the direct examination of the diagonals generally runs faster then the routine call examination. The slight anomaly at optimization level 1 can be attributed to the fact that level 1 optimizations are quite simple, and do very little by way of flow analysis. In any case, since optimization level 1 is billed as *"all the optimizations that can be done quickly"*, the optimizer cannot be faulted for inadequately optimizing one version of the program. In fact, the only difference be-tween the level 0 and level 1 optimizations for *this* example is the removal of extraneous assembler labels. This removal allows the assembler reorganizer to better manipulate the assembly output,[23] and causes the major influence on program run-time to evidence itself.[24] When more substantial optimizations are performed at level 2 (specifically, the elimination of redundant code), the direct examination once again performs better than the routine call examination.

For this simple test, there is no difference in execution speed between optimization levels 2, 3, and 4 for the direct examination of the diagonals (the extra optimization simply has no effect on a program that is this simply and tightly coded). However, a noticeable difference occurs at optimization level 4 for the absolute value routine-call examination of the diagonals. At this level, the optimizer hoists the absolute value routine into the main body of code, which results in a faster run-time performance.

_____

[23]The assembler reorganizer does not (or cannot) consider two adjacent labels as being identical. Consequently, it is unable to move instructions around a pair of labels, and the resulting executable image is larger.

[24]In this case, it is the four array accesses for direct examination of the diagonals *versus* two array accesses for the routine call examination of the diagonals. Without any common subexpression elimination, this results in four multiplies and four adds for direct examination *versus* two multiplies, two adds, and a routine call for the routine call examination. The latter code is faster in this case, since multiply instructions are very expensive (see section 3.2.1).

The hoisted code that examines the diagonals with an absolute value routine still runs slower than the direct examination of the diagonals because the optimizer (and the assembly reorganizer) still have some troubles with register tracking.

```
#   49        if (row[r]-row[i] == r-i)   /* Test left diagonal */
        subu    $3, $18, $17
        subu    $24, $4, $2
        beq     $24, $3, $41
#   50        break;
#   51        if (row[i]-row[r] == r-i)   /* Test right diagonal */
        subu    $25, $2, $4
        beq     $25, $3, $41
#   52        break;
```

Figure 4-14: Direct Examination of Diagonals

As shown in figure 4-14, the direct examination of the diagonals results in 20 bytes of code being generated. However, as figure 4-15 shows, the hoisted code uses 32 bytes of code.[25] Since the remainder of the code generated for these two cases is identical, the extra bytes of code are directly responsible for the performance difference.

```
#   46        if (abs(row[r]-row[i]) == r-i)
        subu    $2, $4, $3
        bge     $2, 0, $41
        negu    $3, $2
        b       $42
$41:
        move    $3, $2
$42:
        move    $2, $3
        subu    $24, $18, $17
        beq     $24, $3, $43
#   47        break;                      /* Test both diagonals */
```

Figure 4-15: Hoisted Routine Examination of Diagonals

The optimizer is having some difficulty in tracking the usage of registers 2 and 3, especially since the `move` instruction immediately following the label $42 serves no purpose (since the value in register 2 is not used anywhere else in the routine). A more efficient extraction of this routine shown in figure

---

[25]These code size values are somewhat misleading, since they measure the size of the assembly code, and not the size of the actual executable image. The assembler reorganizer must occasionally insert `nop` instructions into the code stream. The actual size of the code in figure 4-14 is 28 bytes, while the size of the code in figure 4-15 is 36 bytes.

```
subu    v1,s2,s1                    subu    v0,a0,v1
subu    t8,a0,v0                    bgez    v0,0x4002a8
beq     t8,v1,0x4002b0              move    v1,v0
nop                                 b       0x4002a8
subu    t9,v0,a0                    subu    v1,zero,v0
beq     t9,v1,0x4002b0              move    v1,v0
nop                                 subu    t8,s2,s1
                                    beq     t8,v1,0x4002c0
                                    move    v0,v1
```

Machine code for direct **examination (28 bytes)**	**Machine code for hoisted** **routine (36 bytes)**

4-16. In the latter case, the size of the assembly language routine is again 20 bytes of code.[26]

```
  #  46       if (abs(row[r]-row[i]) == r-i)
        subu    $2, $4, $3
        bge     $2, 0, $41
        negu    $2, $2
  $41:
        subu    $24, $18, $17
        beq     $24, $2, $43
  #  47         break;                      /* Test both diagonals */
```

Figure 4-16: Hand Optimized Hoisted Routine Examination of Diagonals

4.3.1. Influence of Assembler Reorganizer

Although routine hoisting is a valuable optimization, the combination of the code generator and the assembly reorganizer has, in this case as in others, deleteriously affected the quality of the executable image.

```
0x400290:        00831023         subu    v0,a0,v1
0x400294:        04410003         bgez    v0,0x4002a4
0x400298:        0251c023         subu    t8,s2,s1
0x40029c:        00021023         subu    v0,zero,v0
0x4002a0:        0251c023         subu    t8,s2,s1
0x4002a4:        13020004         beq     t8,v0,0x4002b8
0x4002a8:        00000000         nop
```

Figure 4-17: Machine Language Output for Hand Optimized Code in Figure 4-16

Figure 4-17 shows the actual machine instructions generated for the code in figure 4-16. Notice that the instructions at location 0x400298 and 0x4002a0 are identical. As outlined in section 3.2.2, the assembly reorganizer has filled in the nop instruction that follows the bgez with the instruction that was originally targeted by the branch (the subu instruction at 0x4002a0 calculating r-i), and has moved the target of the branch to the next instruction that follows the original target (to 0x4002a4). While this behavior is entirely correct, the reorganizer has missed the fact that the moved instruction may be *removed* from its original location, reducing the size of and increasing the speed of the final executable image. It could be argued that the the subu instruction cannot be deleted because it immediately follows a label. However, because the assembler knows of all the jumps and branches that target the label, it can easily determine that the instruction is removable. In this case, only a single branch targets that label. (See chapter 7 for further discussion on this and other reorganizer drawbacks).

When the assembly language output (shown in figure 4-16) is modified again so that the calculation of r-i is moved before the branch (effectively forcing a different reorganization strategy), the resulting executable image is generated more intelligently with a size of only 24 bytes (not all of which are always executed) and a concomitant speed improvement (seen figures 4-18 and 4-19).

_____

[26]The actual executable image size is now really 28 bytes. This means that by eliminating the needless register shuffle, the hoisted code is now slightly faster than the original inline evaluation of the diagonals, which is what we would expect. This is due to the fact that roughly half of the time row[r]-row[i] is positive, so not all 28 bytes of code are executed at each pass.

```
  #  46      if (abs(row[r]-row[i]) == r-i)
        subu      $2, $4, $3
        subu      $24, $18, $17
        bge       $2, 0, $41
        negu      $2, $2
$41:
        beq       $24, $2, $43
  #  47          break;                      /* Test both diagonals */
```

Figure 4-18: Further Modification of Hoisted Code

```
0x400290:       00831023        subu    v0,a0,v1
0x400294:       04410002        bgez    v0,0x4002a0
0x400298:       0251c023        subu    t8,s2,s1
0x40029c:       00021023        subu    v0,zero,v0
0x4002a0:       13020004        beq     t8,v0,0x4002b4
0x4002a4:       00000000        nop
```

Figure 4-19: Machine Language Output of Further Optimization in Figure 4-18

4.3.2. Local Conclusions

The routine hoisting optimization performed by the MIPS compiler back end can be tremendously effective in reducing the overall execution time of programs. However, as shown in the simple example above, some work still needs to be done on the register tracking algorithms in the code generator, and in the expression tracking algorithms in the assembly reorganizer. The ability to consider two adjacent labels as being identical targets for jumps and branches would also be a desirable feature.

5. Hardware Effects on Program Performance

In this chapter, we describe our measurements of the hardware's interaction with simple software constructs. Initially, we set out to ask what would be the time required to perform a routine call. However, as our work progressed, we discovered that there was not a straightforward answer to the question. Rather, it was dependent on the instruction cache (which on our version of the hardware is 16 K bytes) and how the host operating system (UNIX) places programs in virtual memory. In the following sections, we describe the compiler's interaction with these features and interpret our results.

5.1. Routine Call Overhead

One tidbit of information about a machine and its compilers is how fast it can execute a routine call. On the VAX, the answer depends on the type of routine linkage that is used (i.e., `jsb` or `callx`), the number of local registers used by the routine, how many parameters it is passed, and the instructions that are used to pass them (i.e., `pushr`, `pushax`, `pushx`, or `movx`). If one uses the `callx` routine linkage, the answer is "very expensive", no matter what the other factors. This is due to the fact that while the `callx` linkage is very easy to use from an assembly language standpoint (and also from a compiler standpoint), it is a complex instruction that incurs a great deal of overhead, whether or not any of the special features are used.

The MIPS machine has three instructions for subroutine calls: `bgezal`, `jal`, and `jalr`. The last two are the most commonly used (in fact, as discussed in section 6.1.1.2 the MIPS compiler suite does not generate the `bgezal` instruction[27]). These two instructions are relatively simple. They store the return address in register 31 and jump to the specified address (`jal` is a jump to an address, while `jalr` is a jump indirectly through a register). We were interested in discovering how long a simple routine call would take, given a specified number of parameters. In our test cases, the target routine was a dummy routine that did nothing (although the compiler still generated code to save the actual parameters on the stack).

Our test cases were broken up into a number of classes. First, we subdivided the test programs into the number of parameters that we would pass into a routine. This number was varied from 0 through 15 parameters. Next, we tested for the type of parameter. Since our examples were constructed using C, we used all of the types available to the language (which corresponded nicely to the types available in the MIPS M/500 hardware): `char`, `short`, `int`, `long`, `float`, and `double`. We also used pointers to each of these types of variables. Finally, to round out the problems, we examined the compiler's behavior with each of the four possible variable allocation classes: local, global, local-own (i.e., local scope declared `static`), and global-own (i.e., global scope declared `static`). We generated 768 different programs (using an automatic program generation test bed) and executed each one a number of times.

Each program consisted of a loop, executed 2048 times, surrounding 512 calls to a routine. We

---

[27]The proposed MIPS LISP implementation will use it.

used a large high number of routine calls so that they would substantially outweigh the overhead of the loop. When multiple parameters were passed to a routine, the actual parameters were rotated through the set of formal parameters to eliminate the possibility of any special optimizations that the compiler might have for detecting common subexpressions.[28]

Initially, our study discovered the following items:

1. The first four parameters to a routine are passed in registers 4 through 7 (or register `a0` through `a3`; see table A-1). The remaining parameters are passed on the stack (the reorganizer has an interesting part to play in this convention; see figures 5-1 and 5-2). Passing 4 parameters in registers is wise. Most routines are called with 3 parameters or fewer.[29]

2. All integer data types (i.e., `char`, `short`, `int`, and `long`) took the same amount of time to pass as parameters to the test routine. This is because the `lb`, `lh`, and `lw` instructions all execute in a single cycle (plus a single delay slot).

3. All address data types (i.e., a pointer to any of `char`, `short`, `int`, `long`, `float`, or `double`) took the same amount of time to pass as parameters to the test routine. This is because all addresses are loaded using the `la` instruction.

4. Passing floating-point parameters took longer than passing integer parameters. This is due to the interactions and synchronization between the Mips M/500 CPU and the floating-point co-processor. Although no `nop` instructions are in evidence in the object code, there are implicit delays whenever data is passed from one processor to another.

5. Double precision floating-point parameters took less time to pass than single precision floating-point parameters. This was an artifact of the C language calling convention, which requires that single precision numbers be converted to double precision in routine calls. This effect is also discussed in section WHETSTONE.CC.

6. Passing local variables took less time than passing global or statically allocated variables. Local variables are usually stored in registers, and passing them as parameters requires a register move (i.e., 1 CPU cycle). Global and statically allocated variables are stored in main memory and must be loaded into a register (i.e., 1 CPU cycle plus a delay slot). Multiple loads can be overlapped, but the last load required one extra cycle to fill the delay slot.[30]

7. In our test cases, passing an address as a parameter took less time than passing a value. This result is misleading, though, since the optimizer was able to recognize the addresses we were passing as common sub-expressions and translate that knowledge into a reduced complexity program. In actual practice, passing an address takes the

---

[28]When the number of parameters was 15, the optimizer used over 14 M bytes of memory while trying to optimize the code. This resulted in literally millions of page faults for each separate compilation, and a flurry of complaints directed towards Mips Inc. When main memory was increased from 4 Mb to 8 Mb, the number of page faults (and the compilation time) decreased markedly. However, for a compiler to use 14 Mb of data space to optimize 35 Kb of code is uncalled for. This translates to a data expansion of 400 : 1, or over 26 Kb of optimizer memory for each line of source code. We admit that this example is an unusual one, and that typical optimizer memory usage is not this high. However, this is one example that we hold in disfavor when evaluating the Mips compiler suite.

[29]Of the nearly 1000 individual routines declared in the three integer applications in section 6.3.1, only half a dozen (less than 1%) of them had more than 4 formal parameters. The vast majority of them had 2 or less parameters. This finding closely correlates with the results in [Cook 82], [DePrycker 82], [Tannenbaum 78], and [Zeigler 83], who report 0.9, 2.1, 1.5/2.0, and 1.3 average parameters per routine, respectively.

[30]The delay slot could be filled with the `jal` instruction, but then the delay slot for that instruction could not be filled. See section 3 for more information on delay slots.

---

same amount of time as (for statically allocated variables) or longer (for register variables) than passing a value parameter. Compare the expansions of the `la` instruction on page 144 with that of the `lw` instruction on page 146 and recall that, to construct the address of a register variable, the value of that variable must first be *stored* in a memory location on the stack, requiring an extra `sw` instruction.

We discuss a number of other interesting phenomena in the following sections.

5.2. Reorganizer Effects on Parameter Passing

When passing local parameters to a routine, the MIPS compilers generate `move` or `li` instructions to move the first four parameters into the argument registers, and store instructions to push the remaining parameters on the stack. Thus, one would expect there to be two breakpoints in the time *versus* number of parameters graph – between 0 and 1 parameter, and between 4 and 5 parameters. However, as can be shown in figure 5-1, it takes the same amount of time to call a routine with one integer parameter as it does to call it with none.[31]

Figure 5-1: Routine Overhead for Local Integer Parameters

The predicted breakpoint in the curve occurs between 4 and 5 parameters. Yet 1 parameter takes no longer to pass than zero. The reason for this lies in the assembler reorganizer. A simple routine with no parameters is called with a `jal` instruction, which, according to the MIPS M/500 hardware constraints, has a single delay slot following it. Thus, a simple call takes two CPU cycles to execute. However, a local value class parameter is passed by executing a `move` into argument register `a0`, and this instruction can be moved into the delay slot of the `jal`.

_____

[31]The high/low bars on the graph indicate the maximum variance between different runs of the test programs, while the line indicates the average. We are concentrating on the average time now, and will discuss the variance in section 5.3. The actual times are irrelevant, since our test programs are contrived examples and do not represent real-life examples.

Thus, a routine call with a single parameter takes no longer than a call with none. Both require 2 CPU cycles to execute, but, in the former case, the second cycle is spent executing a `nop`, while in the latter, it is spent executing a `move`.

When global (or statically allocated) variables are passed as parameters, another discrepancy between predicted and actual results occurs. As shown in figure 5-2, the second breakpoint in the curve occurs between 5 and 6 parameters not between 4 and 5 parameters as predicted.

Figure 5-2: Routine Overhead for Global Integer Parameters

The reason for this effect is similar to that seen in figure 5-1, except that in this case, the effect is delayed. The first four global value parameters are loaded into registers with the `lb`, `lh`, or `lw` instructions. Each load instruction takes one cycle plus one delay slot. However, each delay slot except the last is filled with the next load instruction, and the delay slot for the last load instruction is filled by the `jal` instruction (the delay slot of the `jal` instruction is then of necessity a `nop` instruction). Thus, when there are from 0 through 4 global value parameters passed into a routine, each extra parameter requires one extra instruction cycle to pass it. Note that the `jal` delay slot cannot be by a parameter load (which has its own delay slot), since the first instruction of the called routine might access that parameter.

Unlike the first four parameters, which are simply loaded into registers, the fifth and following parameters must also be pushed onto the stack with an `sw` instruction. Thus each extra parameter past 4 requires two instructions to pass it, and the slope of the curve for these instructions should be twice what it is for the first 4 parameters. However, when we examine the object code for 5 parameters, we notice that the delay slot for the `jal` is filled with the `sw` for the fifth parameter. Since this delay slot was previously a `nop` instruction, passing a fifth parameter has effectively taken only a single instruction more than passing four parameters (even though two extra instructions are actually executed). When the sixth parameter is passed, two instructions are required, and since the

delay slot of the `jal` is already filled, twice the work needs be done to load this and subsequent parameters.

Thus, the assembly reorganization needed to satisfy the MIPS M/500 pipeline actually benefits sub-routine parameter passing by delaying the effects of adding extra parameters to subroutines. In general, these benefits will manifest themselves regardless of the type of parameter that is passed, since the benefits are derived for both local and global value parameters, and at the low and high end of the number of variables.

5.3. Effects of Instruction Caching

As we said in the previous section, figures 5-1 and 5-2 show not only the average run-time for various procedure call overheads, but also the variance across different runs of the same program. The fact that the run-times varied at all was discovered accidentally. We ran each program 6 times and generated a graph from the results because there were some wild aberrations in the graph. When we re-ran the tests, we got a very different graph with different abnormalities.

At first we thought that the discrepancies were due to glitches in the CPU time accounting of the MIPS UNIX system. We rejected this idea when we observed the following:

1. Successive runs of the same program gave almost perfectly consistent run times, but if other programs were run in between tests, the CPU time varied.

2. The actual amounts of CPU time required to run a given test case did not fluctuate in a continuous spectrum, but fell into a limited set of quanta.

3. The run-times of the very large and very small tests cases did not vary much, but run-times of the the medium sized test cases varied considerably.

Figure 5-3 shows the distribution of the run-times needed by the various programs. Each curve on the graph represents a different number of parameters being fed to the test routine. The `x` axis is the CPU time required to execute the program, and the `Y` axis is the frequency of occurrence of that time value.

Notice that the individual programs, when run at different times, exhibit different run-times, and that these run-times fall into well defined quantile points. There are three reasons for this:

1. The MIPS M/500 has a 16 K byte hardware instruction cache. Programs smaller than one page (i.e., 4 K bytes) will reside either wholly within the cache or wholly outside of it. Programs larger than one page but less than 4 pages will reside wholly or partially in the cache, or they may not be in the cache at all. Programs larger than 16 K bytes will have some variable percentage of their code in the instruction cache. The fraction of a program that is in the cache will, to a large degree, determine how fast that program runs.

2. Whether a page resides in the instruction cache depends on a number of factors, one of which is the physical address of the start of the page. On any single run of a program, the UNIX operating system will place successive pages of a program image into the first available pages from the memory pool. These pages are not necessarily contiguous, so there may be collisions between two or more pages in a program in the

Figure 5-3: Distribution of Execution Times for Similar Programs

instruction cache hash table.[32] The fewer collisions there are within a given program, the more effective the cache is at speeding up the execution time of that program.

3. The file access mechanism on UNIX relies on an *i-node* which points at the pages of a program when it is on disk, and also serves to reference the program's pages when it is in main memory. Once a program has been executed, it remains in memory (even if it is not presently being executed) until its pages need to be reclaimed. If a program is run a number of times in succession, the pages that comprise the program will remain at the *same* virtual and physical address across each run. If, however, other programs are run in between executions, then the pages for the program may be reclaimed, and on subsequent execution may have to be reloaded from disk at potentially different addresses in memory. Additionally, if a copy of the program is made (effectively creating a new *i-node*), then the program (now referenced by a different, non-resident *i-node*) must be loaded into memory, probably at different page addresses. Each time the addresses of the pages of a program change, the program may run at a different speed, due to the reasons cited in the first two items.

The net effect of all of this is that, in general, no single number can be quoted as the "run-time" of a program. We can in general speak of best case, worst case, or average run-times. However, for *any* processor that has an instruction cache, the hit rate on the cache is determined by a number of factors – all of which are out of the control of the user in the case of the MIPS M/500 running UNIX. *Ergo*, the actual run-time of a program is non-deterministic and unpredictable, although the *range* of values in which the run-time will fall *is* predictable. Figure 5-4 shows the ranges of execution times for a set of similar programs. In this case, the program that was used was one of our routine-call test cases, except that here we simply varied the size of the loop being executed, rather than varying the number of parameters.

---

[32]A later release of the MIPS system software has fixed this problem.

Figure 5-4: Variance of Execution Times of Similar Programs

In figure 5-4, the solid line represents the average execution time as the program size increases. The high-low bars indicate the minimum and maximum execution times, and the dotted lines represent the extrapolation of the extremes. The distance between the extremes is probably highly correlated to the size of the instruction cache, but we cannot verify this predication because we could not change the hardware cache size. We do know, however, that without an instruction cache, the average execution time would probably be at the same level as the extrapolated maximum, with very little variation between runs.

In figure 5-4, the average run-time is close to the minimum for small programs and climbs toward the maximum for larger programs. The larger the program, the less likely it will be wholly cache resident. Large programs (i.e., larger than 16 Kb) will never be wholly cache resident, although they can still reap the benefits of an instruction cache by grouping like procedures together (MIPS has a program called *cord* to aid in this process). Further benefits could be derived with a more robust linker.

Floating-point programs exhibit a much smaller range between the minimum and maximum values. We suspect that this is due to the fact that although instructions for the floating-point co-processor may be kept in the instruction cache, they are executed in the co-processor – effectively obviating the cache.[33] We feel (although we have not tested this hypothesis) that programs with a greater percentage of floating-point instructions will demonstrate a reduced variability in execution speeds. Of course, regardless of program content, the larger the program, the lower the variability when the program counter is not kept within a 1 page boundary.

---

[33]MIPS Inc. doubts this hypothesis.

6. Instruction Set Usage by the Compilers

This chapter covers a three part analysis of the use of the machine instruction set by the MIPS compilers. The first part is a static analysis in which we examined the source code for the compilers to determine what instructions could possibly be generated from a source program. The purpose of this test was to get a feel for the utility of the instructions in the instruction set. If an instruction is never used by the compiler, then perhaps it is because it is too difficult for the compiler detect a use for the instruction (or perhaps it is a special instruction that was never expected to be used, such as the *translation lookaside buffer* instructions on the MIPS M/500, or the *context switch* instructions on the VAX).

The second part is a thorough analysis of a specific compiler written for the MIPS. The compiler is for BCPL, a simple, easy-to-implement systems programming language.[34] The purpose of this exercise was to get an instrumented view of the instruction set in relation to a known compiler, and to evaluate patterns of register use. instruction use , instruction mix, addressing mode use, and addressing mode effectiveness.

The third part is an analysis of instruction and register use across a number of large programs. In contrast to the static analysis, this "brute force" overview examines the actual instructions and registers that are used for a set of programs. This analysis does not provide specific insights; we can make some general statements about the compilers' effectiveness and efficiency.

However, in all three sections we compare the MIPS compilers with the VAX Berkeley UNIX compilers. The purpose of the comparison is to provide to:

- Give some feel for the use of a RISC versus a CISC architecture from a compiler standpoint. We hope to quantify our assertion that many instructions in a CISC architecture are not used by the compiler, and thus show that a *reduced* instruction set is reasonable.

- Provide a basis of comparison that most of our readers will be familiar with.

- Highlight the differences between optimizing compilers (on the MIPS) and less sophisticated compilers (on the VAX).

This information is provided to deliver *insights*, not tables of raw figures.

6.1. Static Analysis of Compilers

This section examines the set of instructions that the compiler *can* generate (but not necessarily those instructions that it *will* generate). We collected this information by reading through the source code of the compilers. Through this exercise we hope to shed some light on two aspects of compiler and processor technology:

1. What subset of the instruction set can be effectively used by a compiler (and from this information, what an effective minimum instruction set is).

---

[34]BCPL is one of the ancestors of the C language.

2. What subset of the instruction set cannot be used by a compiler (and from this, which instructions are too specialized or too complex to be effectively fitted to a source code idiom).

To adequately address these questions, we looked at the compilers for both the MIPS and the VAX, the latter being included in our investigation as a CISC architecture, and hence a possible counterexample to our pro-RISC argument. As shall be seen, we show conclusively that a RISC architecture is a much better choice from a compiler standpoint.

6.1.1. MIPS C, FORTRAN, and Pascal Compilers

The MIPS compiler suite currently consists of three different language front end parsers (C, FORTRAN, and Pascal) and a common optimizer and code generator. To analyze the use of the MIPS instruction set by the compilers, we were forced to look at the MIPS compilers from two levels – the high-level instruction set use generated by the compiler, and the low level instruction set executed by the MIPS M/500 hardware. Ultimately, only the low-level instructions get executed, so the most significant tables are in section 6.1.1.2. However, comparing the low-level coverage with the high-level coverage, argues in favor of a *reduced* instruction set. In spite of the large number of conditional instructions provided by the high level assembler (26 set and branch), all of the instructions are easily emulated with less than one third as many real instructions (8 branch and set, plus `xor`).

6.1.1.1. MIPS High-Level Instruction Use

The following table lists the full (high-level) instruction set of the MIPS architecture. The MIPS compilers use many of these instructions. If an instruction is used by the compiler,[35] it is shown in **boldface**. Wherever justifiable, instructions that are not generated by the compiler/optimizer are shown in `plain text`. Instructions that are unjustifiably ignored by the compiler are shown in *(italics)*. Superscripted numbers refer to notes at the end of the table.

abs	**add**	**addu**	**and**	**b**
bal^1	bc0f^2	bc0t^2	**bc1f**	**bc1t**
bc2f^2	bc2t^2	bc3f^2	bc3t^2	**beq**
beqz^4	**bge**	**bgeu**	bgez^4	*(bgezal)*
bgt	**bgtu**	bgtz^4	**ble**	**bleu**
blez^4	**blt**	**bltu**	bltz^4	*(bltzal)*
bne	bnez^4	**break**	c0^2	c1^3
c2^2	c3^2	cfc0^2	cfc1^3	cfc2^2
cfc3^2	ctc0^2	ctc1^3	ctc2^2	ctc3^2
div	**divu**	**j**	**jal**	**la**
lb	**lbu**	**ld**	**lh**	**lhu**
li	lui^5	**lw**	lwc0^2	lwc1^3
lwc2^2	lwc3^2	lwl^6	lwr^6	mfc0^2
mfc1	**mfc1.d**	mfc2^2	mfc3^2	*(mfhi)*
(mflo)	**move**	mtc0^2	**mtc1**	**mtc1.d**
mtc2^2	mtc3^2	*(mthi)*	*(mtlo)*	**mul**
(mulo)	*(mulou)*	*(mult)*	*(multu)*	*(neg)*
negu	**nop**	nor^8	**not**	**or**

---

rem	**remu**	rfe[7]	rol[9]	ror[9]
sb	**sd**	**seq**	**sge**	**sgeu**
sgt	**sgtu**	**sh**	**sle**	**sleu**
sll	**slt**	**sltu**	**sne**	**sra**
srl	*(sub)*	**subu**	**sw**	swc0[2]
swc1[3]	swc2[2]	swc3[2]	swl[6]	swr[6]
syscall[7]	tlbp[10]	tlbr[10]	tlbwi[10]	tlbwr[10]
(ulh)	*(ulhu)*	*(ulw)*	*(ush)*	*(usw)*
xor				

abs.d	**abs.s**	**add.d**	**add.s**	**c.eq.d**
c.eq.s	c.f.d[11]	c.f.s[11]	**c.le.d**	**c.le.s**
c.lt.d	**c.lt.s**	c.nge.d[11]	c.nge.s[11]	c.ngl.d[11]
c.ngl.s[11]	c.ngle.d[11]	c.ngle.s[11]	c.ngt.d[11]	c.ngt.s[11]
c.ole.d[11]	c.ole.s[11]	c.olt.d[11]	c.olt.s[11]	c.seq.d[11]
c.seq.s[11]	c.sf.d[11]	c.sf.s[11]	c.ueq.d[11]	c.ueq.s[11]
c.ule.d[11]	c.ule.s[11]	c.ult.d[11]	c.ult.s[11]	c.un.d[11]
c.un.s[11]	**cvt.d.s**	**cvt.d.w**	**cvt.s.d**	**cvt.s.w**
cvt.w.d	**cvt.w.s**	**div.d**	**div.s**	**l.d**
l.s	**mov.s**	**mov.d**	**mul.d**	**mul.s**
neg.d	**neg.s**	**round.w.d**	**round.w.s**	**s.d**
s.s	**sub.d**	**sub.s**	**trunc.w.d**	**trunc.w.s**

Notes

1. The MIPS compilers suffer from a common problem. The `bal` instruction is not used because the compiler has no facility for determining at compile time the address of the target, and hence no knowledge whether the target will be out of range of a branch. The target will usually be in range when recursion is used, although the MIPS compiler does not take advantage of this knowledge.

2. The MIPS M/500 provides instruction set support for 4 co-processors. However, only co-processor 1 (the floating point co-processor) is presently supported in hardware. Of course, the extra co-processor instructions will not be generated for non-existent hardware.

3. Certain co-processor instructions do not make any sense for the floating point co-processor, since their functions are not supported by a floating point unit.

4. Although instructions are provided in the high level assembly language for conditional branches relative to zero, the compiler simply generates an ordinary conditional branch relative to the zero register. In the end, these instructions are functionally equivalent.

5. The `lui` instruction is available to the high-level assembler, but it is not really needed. It is used primarily to load an immediate value of larger than 16 bits on the real MIPS M/500 hardware (while the high level assembler allows a full 32 bit operand).

6. These special load instructions could conceivably be used in C to load structure components stored in registers, but their primary function is to be used in the unaligned load and store instructions.

7. The `syscall` and `rfe` instructions are used to perform system calls, a function handled by the subroutine libraries.

8. None of the high-level languages on the MIPS has a **nor** function, hence the `nor` instruction is not used.

9. None of the high-level languages on the MIPS has a rotate function, hence the `rol` and `ror` instructions are not used.

10. These instructions reference the translation lookaside buffer and are used primarily in the kernel, and then only in assembly language.

11. These instructions are provided by the floating-point co-processor to supply complete IEEE floating-point compatibility. They are not all necessary for the languages available on the MIPS.

6.1.1.2. MIPS M/500 Low Level Instruction Use

The following table lists the full (native) instruction set of the MIPS M/500 architecture. Since the MIPS compilers do not generate these instructions directly, but rely on the assembler reorganizer, it is only partially true that the compilers use these instructions. If an instruction is used by the compiler,[36] it is shown in **boldface**. Wherever justifiable, instructions that are not generated by the compiler/optimizer are shown in `plain text`. Instructions that are unjustifiably ignored by the compiler are shown in *(italics)*. The superscripted numbers refer to notes at the end of the table.

add	**addi**	**addiu**	**addu**	**and**	**andi**
b	$bc0f^1$	$bc0t^1$	$bc1f^2$	$bc1t^2$	**beq**
bgez	*(bgezal)*	**bgtz**	**blez**	**bltz**	*(bltzal)*
bne	**break**	$c0^1$	$c2^1$	$c3^1$	**cfc1**
ctc1	**div**	**divu**	**j**	**jal**	**jalr**
jr	**lb**	**lbu**	**lh**	**lhu**	**li**
lui	**lw**	$lwc0^1$	**lwc1**	$lwc2^1$	$lwc3^1$
lwl^5	lwr^5	$mfc0^1$	**mfc1**	**mfhi**	**mflo**
move	$mtc0^1$	**mtc1**	$mthi^3$	$mtlo^3$	**mult**
multu	**nop**	**nor**	**or**	**ori**	**sb**
sh	**sll**	**sllv**	**slt**	**slti**	**sltiu**
sltu	**sra**	**srav**	**srl**	**srlv**	**sub**
subu	**sw**	$swc0^1$	**swc1**	$swc2^1$	$swc3^1$
swl^5	swr^5	$syscall^4$	**xor**	**xori**	
abs.d	**abs.s**	**add.d**	**add.s**	**c.eq.d**	**c.eq.s**
$c.f.d^6$	$c.f.s^6$	**c.le.d**	**c.le.s**	**c.lt.d**	**c.lt.s**
$c.nge.d^6$	$c.nge.s^6$	$c.ngl.d^6$	$c.ngl.s^6$	$c.ngle.d^6$	$c.ngle.s^6$
$c.ngt.d^6$	$c.ngt.s^6$	$c.ole.d^6$	$c.ole.s^6$	$c.olt.d^6$	$c.olt.s^6$
$c.seq.d^6$	$c.seq.s^6$	$c.sf.d^6$	$c.sf.s^6$	$c.ueq.d^6$	$c.ueq.s^6$
$c.ule.d^6$	$c.ule.s^6$	$c.ult.d^6$	$c.ult.s^6$	$c.un.d^6$	$c.un.s^6$
cvt.d.s	**cvt.d.w**	**cvt.s.d**	**cvt.s.w**	**cvt.w.d**	**cvt.w.s**
div.d	**div.s**	**mov.d**	**mov.s**	**mul.d**	**mul.s**
neg.d	**neg.s**	**sub.d**	**sub.s**		

Notes

1. The MIPS M/500 provides instruction set support for 4 co-processors. However, only co-processor 1 (the floating-point co-processor) is presently supported in hardware. Of course, the extra co-processor instructions will not be generated for non-existent hardware.

2. Certain co-processor instructions do not make any sense for the floating point co-processor, since their functions are not supported by a floating-point unit.

3. The **hi** (**lo**) registers are documented to *"hold the most (least) significant 32 bits of*

---

[36]In this case, by "compiler" we mean the combination of the language-specific frontend and the common back end and the assembler reorganizer.

multiply, quotient, or divide." Since these are *result* registers, we do not expect that a compiler would have any reason to load them explicitly.

4. The `syscall` instruction is used to perform system calls, a function handled by the subroutine libraries.

5. These special load instructions could conceivably be used in C to load structure components stored in registers, but their primary function is to be used in the unaligned load and store instructions, none of which are generated by the compiler.

6. These instructions are provided by the floating-point co-processor to supply complete IEEE floating-point compatibility. They are not all necessary for the languages available on the MIPS.

6.1.2. Berkeley C and FORTRAN Compilers

By way of comparison, we examined the Berkeley C and FORTRAN compilers and the way they use the VAX assembly instruction suite. The following table lists the full instruction set of the VAX architecture. The Berkeley C and FORTRAN compilers use many of these instructions. If an instruction is used by the compiler,[37] it is shown in **boldface**. Wherever justifiable, instructions that are not generated by the compiler/optimizer are shown in `plain text`. Instructions that are unjustifiably ignored by the compiler are shown in *(italics)*. Superscripted numbers refer to notes at the end of the table.

(acbb)	*(acbd)*	*(acbf)*	acbg^{10}	acbh^{10}
acbl^2	*(acbw)*	adawi^{13}	**addb2**^{4}	*(addb3)*
addd2	**addd3**	**addf2**	**addf3**	addg2^{10}
addg3^{10}	addh2^{10}	addh3^{10}	**addl2**	**addl3**
addp4^{11}	addp6^{11}	**addw2**^{4}	*(addw3)*	*(adwc)*
aobleq^2	**aoblss**^{2}	**ashl**	ashp^{11}	*(ashq)*
bbc^1	*(bbcc)*	bbcci^{13}	*(bbcs)*	**bbs**^{1}
bbsc^1	*(bbss)*	bbssi^{13}	*(bcc)*	*(bcs)*
beql^1	**beqlu**^{1}	**bgeq**^{1}	**bgequ**^{1}	**bgtr**^{1}
bgtru^1	**bicb2**^{4}	*(bicb3)*	**bicl2**	**bicl3**
(bicpsw)	**bicw2**^{4}	*(bicw3)*	**bisb2**^{4}	*(bisb3)*
bisl2	**bisl3**	*(bispsw)*	**bisw2**^{4}	*(bisw3)*
bitb	**bitl**	**bitw**	**blbc**^{1}	**blbs**^{1}
bleq^1	**blequ**^{1}	**blss**^{1}	**blssu**^{1}	**bneq**^{1}
bnequ^1	bpt^9	*(brb)*	**brw**	*(bsbb)*
(bsbw)	bugl^9	bugw^9	*(bvc)*	*(bvs)*
(callg)^{12}	**calls**	*(caseb)*	**casel**	*(casew)*
chme^7	chmk^7	chms^7	chmu^7	**clrb**
clrd	**clrf**	clrg^{10}	clrh^{10}	**clrl**
(clro)	*(clrq)*	**clrw**	**cmpb**	cmpc3^{14}
cmpc5^{14}	**cmpd**	**cmpf**	cmpg^{10}	cmph^{10}
cmpl	cmpp3^{11}	cmpp4^{11}	*(cmpv)*	**cmpw**
(cmpzv)	crc^{14}	**cvtbd**	**cvtbf**	cvtbg^{10}

---

[37]In this case, by "compiler" we mean the combination of the code generator and optimizer, since the Berkeley compiler suite splits these two tasks into two separate programs (which, instead of operating on a common intermediate form, share information in assembler source code format). Some instructions are therefore not generated directly by the compiler, but are inserted by the optimizer to match certain code idioms. The origin of the instruction is unimportant. Rather, it is more important that it is used at all. The C and FORTRAN compilers differ only in the front end – the code generator is shared by both languages.

cvtbh[10]	**cvtbl**	**cvtbw**	**cvtdb**	**cvtdf**
cvtdh[10]	**cvtdl**	**cvtdw**	**cvtfb**	**cvtfd**
cvtfg[10]	cvtfh[10]	**cvtfl**	**cvtfw**	cvtgb[10]
cvtgf[10]	cvtgh[10]	cvtgl[10]	cvtgw[10]	cvthb[10]
cvthd[10]	cvthf[10]	cvthg[10]	cvthl[10]	cvthw[10]
cvtlb	**cvtld**	**cvtlf**	cvtlg[10]	cvtlh[10]
cvtlp[11]	**cvtlw**	cvtpl[11]	cvtps[11]	cvtpt[11]
(cvtrdl)	*(cvtrfl)*	cvtrgl[10]	cvtrhl[10]	cvtsp[11]
cvttp[11]	**cvtwb**	**cvtwd**	**cvtwf**	cvtwg[10]
cvtwh[10]	**cvtwl**	**decb**	**decl**	**decw**
divb2[4]	*(divb3)*	**divd2**	**divd3**	**divf2**
divf3	divg2[10]	divg3[10]	divh2[10]	divh3[10]
divl2	**divl3**	divp[11]	**divw2**[4]	*(divw3)*
editpc[11]	*(ediv)*	*(emodd)*	*(emodf)*	emodg[10]
emodh[10]	*(emul)*	escd	esce	escf
extv	**extzv**	ffc[14]	ffs[14]	halt[8]
incb	**incl**	**incw**	index[14]	insqhi[14]
insqti[14]	insque[14]	**insv**	**jbc**[1,2]	*(jbcc)*
(jbcs)	**jbr**[1]	**jbs**[1,2]	**jbsc**[3]	*(jbss)*
jeql[1]	**jeqlu**[1]	**jgeq**[1]	**jgequ**[1]	**jgtr**[1]
jgtru[1]	**jlbc**[1,2]	**jlbs**[1,2]	**jleq**[1]	**jlequ**[1]
jlss[1]	**jlssu**[1]	**jmp**[1]	**jneq**[1]	**jnequ**[1]
jsb[5]	ldpctx[8]	locc[14]	matchc[14]	*(mcomb)*
mcoml	*(mcomw)*	mfpr[8]	**mnegb**	**mnegd**
mnegf	mnegg[10]	mnegh[10]	**mnegl**	**mnegw**
movab	*(movad)*	*(movaf)*	movag[10]	movah[10]
moval	*(movao)*	**movaq**[2]	**movaw**[2]	**movb**
movc3[6]	movc5[14]	**movd**	**movf**	movg[10]
movh[10]	**movl**	*(movo)*	movp[11]	*(movpsl)*
movq	movtc[14]	movtuc[14]	**movw**	**movzbl**
movzbw	**movzwl**	mtpr[8]	**mulb2**[4]	*(mulb3)*
muld2	**muld3**	**mulf2**	**mulf3**	mulg2[10]
mulg3[10]	mulh2[10]	mulh3[10]	**mull2**	**mull3**
mulp[11]	**mulw2**[4]	*(mulw3)*	nop	polyd[14]
polyf[14]	polyg[10,14]	polyh[10,14]	*(popr)*	prober[8]
probew[8]	**pushab**[2]	*(pushad)*	*(pushaf)*	**pushag**[10]
pushah[10]	**pushal**[2]	*(pushao)*	*(pushaq)*	*(pushaw)*
pushl	*(pushr)*	rei[8]	remqhi[14]	remqti[14]
remque[14]	**ret**	*(rotl)*	*(rsb)*	*(sbwc)*
scanc[14]	skpc[14]	**sobgeq**[2]	**sobgtr**[2]	spanc[14]
subb2[4]	*(subb3)*	**subd2**	**subd3**	**subf2**
subf3	subg2[10]	subg3[10]	subh2[10]	subh3[10]
subl2	**subl3**	subp4[11]	subp6[11]	**subw2**[4]
(subw3)	svpctx[8]	**tstb**	**tstd**	**tstf**
tstg[10]	tsth[10]	**tstl**	**tstw**	xfc[9]
xorb2[4]	*(xorb3)*	**xorl2**	**xorl3**	**xorw2**[4]
(xorw3)				

Notes

1. The `jbxxx` instructions are pseudo-instructions that are converted to either the corresponding branch instruction or an inverse-sense branch/jump instruction pair by the assembler. Correspondingly, the `bxxx` are only generated by the assembler, the `brw` instruction, which is also generated directly by the compiler. The `beqlu` instruction is identical to the `beql` instruction (since an unsigned test for equality is the same as a signed test for equality); the VAX ISA simply provides two mnemonics for the same instruction. The same is true for `bnequ` and `bneq`.

2. These instructions are generated only by the common optimizer pass, not by the compiler. While this is not bad, it indicates a weakness of the common code generator that the optimizer must compensate for.

3. This instruction is used exclusively in the conversion from unsigned longword integers to floating or double variables, not for the intended function of intra-processor semaphore interlocks.

4. The `add`, `sub`, `mul`, `div`, `bis`, `bic`, and `xor` instructions for byte and word operands are produced only in the two-operand form, while the corresponding instructions for long, floating, and double formats are produced in both two- and three-operand form.

5. The `jsb` instruction is not generated in any normal code sequence, but only as an interface mechanism to the run-time profiler. No subroutines are ever called with anything except the `calls` linkage.

6. The `movc3` instruction is used to copy C structures, not to copy character strings. This is because the instruction takes as its first operand the number of bytes to be moved, but the C representation of strings is such that this datum is not readily available.

7. The `chmx` instructions are intended to be used to switch between processor modes, and are of questionable utility to a compiler. The `chmk` instruction is used by the UNIX libraries (written directly in assembly language) to effect kernel calls.

8. These instructions are designed for use in an operating system context and cannot be expected to be generated by a compiler. Additionally, some of them are privileged instructions and can only be executed in kernel mode.

9. These are very special case instructions (for use in debuggers and other applications) that cannot reasonably be generated by a compiler.

10. The `g` and `h` floating-point types are not supported by UNIX languages and are not available on all versions of the VAX. Consequently, it is reasonable to allow a portable compiler to not generate them.

11. The packed decimal instructions are in the VAX ISA primarily for DEC's version of PL/1 (which the Berkeley compilers do not support).

12. The `callg` instruction is designed for FORTRAN static call frames, although Berkeley FORTRAN does not take advantage of it.

13. The `adawi`, `bbssi`, and `bbcci` instructions are designed for multiprocessor applications.

14. The polynomial and crc instructions are designed to make assembly language programming easier. The character manipulation instructions support complex character comparison, matching, and insertion. All of these instructions provide support for high-level functions not present in C or Fortran. It would be unreasonable to expect most compilers to generate these instructions without the corresponding higher level language primitives.

6.1.3. Comparison of Compiler Coverage

The MIPS high-level instruction set contains 192 instructions,[38] of which 94 (or almost 49%) are unused by the compilers. This, however, is a somewhat unfair measure. If we exclude the instructions that are used for non-existent co-processor functions and the extraneous floating-point instructions that are present to satisfy the IEEE standard, then of the remaining 134 instructions, only 36 (or slightly less than 26%) are unused by the compilers. To be still fairer to the MIPS compiler, we count only those instructions that we considered "unjustifiably ignored by the compiler", then only 17 instructions (or approximately 12%) of the instructions are unused.

Because the MIPS assembler/reorganizer is really a macro assembler, we must also look at the coverage of the native instruction set by the compilers, even if this coverage is through one level of indirection. Of the 135 actual instructions (including the floating-point co-processor instructions[39]), 50 instructions (or 37%) are unused by the compiler. However, if we exclude superfluous floating-point and co-processor instructions then of the remaining 92 instructions, only 7 (a mere 7.5%) are unused. Of these, only 2 fall under the category of "unjustifiably ignored" instructions. Clearly, the MIPS M/500 instruction set is sufficiently small to be manageable by the compiler, but sufficiently large to handle the programming tasks it is designed to handle.

By way of comparison, the VAX native instruction set contains a total of 323 instructions,[40] of which 179 (or 55%) are unused by the compiler. This could lead us to believe that either the compiler is terribly inefficient, or, a more likely conclusion, that the instruction set is far too complex. Even when we exclude the "special" instructions for G and H floating-point formats (which are not supported by all VAX processors), then of the remaining 267 instructions, 121 (or 45%) are unused. To be absolutely fair, if we count only those instructions which, in the previous analysis, we felt were "unjustifiably ignored by the compiler", we still find that over 23% of the instruction repertoire of the VAX is used by the compiler, and that another 22% are of a complex or systems programming nature.

By looking at these numbers, it is clear from a compiler standpoint at least that a RISC architecture is better used by a compiler than a CISC architecture. One of the bottlenecks of a CISC computer is instruction decoding. Removing unneeded instructions can speed up a CISC processor (and at the same time, convert it to a RISC processor). The later analysis of instruction set coverage shows that

---

[38]Remember that the number of instructions in the high level assembly language for the MIPS does not reflect the number of instructions found on the MIPS M/500 native instruction set. Many of the high level instructions are simply macros that are expanded by the assembler reorganizer. See chapter 3 for details.

[39]According to the source code for the disassembler program ***dis***, there is the potential for many more instructions available on the MIPS M/500. However, it is unclear how many of these are actually present in the hardware, and how many were planned but never inscribed in silicon. We will use as our instruction count the number of instructions that can be created by the assembler reorganizer, given the set of instructions documented in the *"Assembly Language Programmer's Guide"* and revealed by the translation table in appendix 3.

We note also that a slightly different measurement criterion has been used on the MIPS M/500 than on the VAX. On the MIPS M/500, "add" and "add immediate" are considered two different instructions, while on the VAX, they are considered to be one instruction with two different addressing modes. If we follow the VAX metric, the MIPS M/500 has 14 fewer instructions—a figure which better shows off the RISC nature of the architecture.

[40]This is a count of real instructions and does not include the 21 `jbxx` pseudo-instructions provided by the Berkeley assembler.

---

this is a wise move to make, since a large fraction of a "standard" CISC architecture is never used by the compiler (nor, we suspect, by a human programmer).

6.2. Assessment of BCPL/MIPS

This section contains a brief description and assessment of the BCPL/MIPS compiler created at the SEI. It contains numbers specific to the MIPS RISC-based workstation and some comparisons with the DEC MicroVAX II.

The compiler consists of a front end that translates BCPL into an intermediate form called Ocode, and a back end that translates Ocode into symbolic assembly language, which is then assembled by the target machine assembler program.

The front end, called *bcpl*, is common. The back ends for MIPS and VAX, are called *cgmips* and *cgvax*, respectively. The structure of the two back ends is very similar; *cgvax* performs a few extra peephole optimizations, but otherwise the generated code is of similar quality. This allows us to make a direct comparison between the two machines.

The vehicle for comparison is the *cgmips* code generator itself, which is a BCPL program with about 4400 lines of source, of which about 50% are white space or comment. It is divided into four modules numbered 1 through 4.

The purpose of this assessment is to:

1. Obtain comparative performance measurements in a manner that, as far as possible, reflects the hardware rather than the combination of hardware and compiler.

2. Test the claims that CISC architectures are too complicated and embody expensive but unused features, whereas RISC machines are sufficient for most purposes and more efficient overall.

It is possible to approach such a task in two ways. One can run very large amounts of code through the two compiling systems and accumulate statistics (section 6.3 describes this approach). Or one can use a small amount of code only and try to understand and explain the results. This section follows the latter course.

The host systems on which the analysis was performed were:

- DEC MicroVAX II running Mach (4.3 BSD UNIX). This is considered to be a machine of about 0.9 "mips".

- MIPS M/500 workstation running 4.3 BSD UNIX, with a 16K byte I-cache and an 8K byte D-cache. This is claimed to be a "4 to 5 mips" machine.

6.2.1. Performance Analysis
We collected the following data for both VAX and MIPS:

- code size

- code density

- bcpl execution speed

- cgmips execution speed

- assembler execution speed.

These figures and appropriate totals and ratios are given below, along with explanatory text.

6.2.1.1. Code Size

Module	1	2	3	4	Total
Bytes	4234	4293	4894	2482	15903
Instructions	1025	1192	1304	642	4163
Bytes/Instr					3.80

Table 6-1: Results of *cgmips* Compiled on the VAX

The code size in table 6-1 includes `case` statement jump tables; without them the average bytes per instruction is 3.64. Note that each entry in such a table occupies 2 bytes on the VAX but 4 bytes on the MIPS.

Module	1	2	3	4	Total
Bytes	6420	6112	7480	3756	23768
Instructions	1448	1502	1818	837	5605
Bytes/Instr					4.24

Table 6-2: Results of *cgmips* Compiled on the MIPS

The code size in table 6-2 includes `case` statement jump tables; without them, the average bytes per instruction is 4.00. However, this figure *excludes* any code expansion in the assembler reorganizer. Initial measurements showed this expansion to be considerable (over 40%; see section 3.2); however, much of this expansion was due to assembler decisions that were not entirely appropriate. After making changes in the Assembler source, and minor changes in the order in which the code generator emitted instructions, we were able to reduce the expansion to 25%, and we believe that further work could reduce it to about 9%. This is discussed below.

6.2.1.2. Code Density

by bytes. . . . . 1.50 (1.90 after assembly)

by instructions. . . . . 1.35 (1.70 after assembly)

Table 6-3: Code Expansion MIPS / VAX

Note that the VAX code density is very high because the code generation strategy uses, wherever possible, address modes with short offsets. The density of the output of pcc, for example, is substantially lower.[41] On the average, execution of each VAX instruction required about eight cycles. Execution of each MIPS instruction required a little more than one cycle.

---

[41]An average of 5.66 bytes per instruction for our benchmarks.

6.2.1.3. BCPL Execution Speed

Execution speed is given in terms of user process time as measured by UNIX. Since both machines were workstations with a single user, this correlates quite closely with elapsed time.

cgmips compiled from source to Ocode, on VAX					
Module	*1*	*2*	*3*	*4*	*Total*
Time *(sec.)*	15.2	15.8	17.5	7.8	56.3

This is approximately 5000 lines/min.

cgmips compiled from source to Ocode, on MIPS					
Module	*1*	*2*	*3*	*4*	*Total*
Time *(sec.)*	3.0	3.0	3.6	1.4	11.0

This is approximately 25000 lines/min. Overall, this program executes 5.1 times as fast on MIPS.

6.2.1.4. Cgmips Execution Speed

cgmips compiled from Ocode to Assembler, on VAX					
Module	*1*	*2*	*3*	*4*	*Total*
Time *(sec.)*	9.7	10.8	15.8	5.5	41.8

This is approximately 6000 lines/min.

cgmips compiled from Ocode to Assembler, on MIPS					
Module	*1*	*2*	*3*	*4*	*Total*
Time *(sec.)*	2.1	2.6	3.2	1.3	9.2

This is approximately 28000 lines/min. Overall, this program executes 4.5 times faster on MIPS.

6.2.1.5. Combined Execution Speed

cgmips compiled from source to Assembler, on VAX					
Module	*1*	*2*	*3*	*4*	*Total*
Time *(sec.)*	24.9	26.6	32.3	13.3	97.1

This is approximately 2600 lines/min.

cgmips compiled from source to Assembler, on MIPS					
Module	*1*	*2*	*3*	*4*	*Total*
Time *(sec.)*	5.1	5.6	6.8	2.7	20.2

This is approximately 13000 lines/min. Overall, the compiler and code generator execute 4.8 times

faster on the MIPS. For small programs (less than 32k bytes), this is probably an accurate reflection of the intrinsic speed of the machine.

6.2.1.6. Assembler Execution Speed

This comparison is different from the previous tests. We measured the time taken to assemble cgmips on both VAX and MIPS, using in each case the native Assembler program. There are several points to note:

1. Both programs had to have several bugs fixed, which should not have affected their speed.

2. The two programs are assembling different input files, and the VAX input is about 25% smaller. However, the files contain functionally equivalent programs.

3. We are measuring the combined effect of the hardware speed and the software performance.

cgmips assembled from Assembler to Object, on VAX					
Module	*1*	*2*	*3*	*4*	*Total*
Time *(sec.)*	4.6	4.1	4.9	3.0	16.6

This corresponds to a rate of assembly of approximately 14000 lines/min.

cgmips assembled from Assembler to Object, on MIPS					
Module	*1*	*2*	*3*	*4*	*Total*
Time *(sec.)*	14.1	12.1	14.5	8.7	49.4

This corresponds to a rate of assembly of is approximately 6700 lines/min. Overall, the MIPS assembler takes three times as long as the VAX assembler for the same program, or more than twice as long for the same number of instructions. Since it is executing on a machine intrinsically almost five times faster, this represents a difference in software performance of more than an order of magnitude.

Granted, the MIPS Assembler is doing more – for example, it is performing the code reorganization required by the target. Nevertheless, the above data represent an example of how software can degrade objective performance faster than hardware can enhance it.

The full compilation times are:

- *cgmips* compiled from Source to Object, on VAX: 113.7 sec

- *cgmips* compiled from Source to Object, on MIPS: 69.6 sec

On VAX, the assembler pass takes less than 15% of the time; on MIPS, it takes more than 70% of the time.[42]

---

[42]According to Larry Weber at MIPS, the reason that this pass takes so long is due to the assembler front end. With MIPS compilers, this stage is bypassed altogether, with the compiler back ends calling the assembler middle end directly. If BCPL took this approach, these times would be appreciably reduced.

6.2.2. Instruction Reorganization on MIPS

When the generated MIPS code of *cgmips* was first submitted to the reorganizer, the number of instructions increased from 5605 to 8116 (by 44.8%). This was a far greater expansion than we had expected, and reasons were sought.

Our first observation was that the reorganizer was generating the full 32-bit addressing idiom for all static operands, even though the code generator was following the rules for generating only `gp`-relative static data. The error was traced to a bug in the assembler; when this was fixed, the number of `lui` instructions generated was reduced from 1086 to 136.

A further reduction could be made by observing that the assembler, though now generating single instructions for loads and stores of most static data, still accessed local static data using two instructions. This was traced to an interesting feature: the assembler correctly handled `gp`-relative addresses only for operands declared *before* the operation referencing them (see section 9). Making this fix replaced 125 `lui/addi` pairs with 125 simple `li` instructions.

This left the reorganized code count at 7041, an expansion of 1435 instructions (25.6%). These extra instructions exhibit the pattern shown in figure 6-1:

Figure 6-1: Extra Instructions - Pattern of Use

However, even these figures are too high. For reasons explained in section 3.2, the reorganizer must make very pessimistic assumptions about whether it is safe to rearrange load and store instructions. Accordingly, it often generates a `nop` to fill a load delay, when in fact another instruction could be placed there. It also has some problems filling branch delays.

We did not perform a full analysis, but a study of a sample of the 1209 `nop` instructions generated suggests that about 70% could be removed by a reorganizer that had more information about aliasing and block structure, reducing the count to about 360.

Of the 226 other extra instructions, several could be removed by combining reorganization with code generation. For example, the reorganizer always loads a constant operand of a conditional branch into register `at`; the code generator could track this value and/or use more than one temporary register. A sampling suggests that about 40% of these instructions could be removed, leaving about 135.

Taken together, all these changes would reduce the reorganization penalty to about 500 instructions, or a little less than 9%.

6.2.3. Instruction Set Usage – MIPS

The usage pattern for MIPS instructions, address modes, and registers is given in the following tables. These figures are for code compiled from a simple systems implementation language. Superscripted numbers refer to notes at the end of these tables.

`lbu`	6	0.1%	`add`	125	1.8%	`slt`	49	0.7%
`lw`	1799	25.6%	`addi`	536	7.6%	`slti`	33	0.5%
`li`	415	5.9%	`addiu`	9	0.1%	`beq`	184	2.6%
`lui`	11	0.2%	`addu`	7	0.1%	`bne`	201	2.9%
Load	2231	31.7%	`sub`	69	1.0%	`bltz`	31	0.4%
`sb`	9	0.1%	`multu`	5	0.1%	`blez`	2	0.0%
`sw`	902	12.8%	`sll`	84	1.5%[2]	`bgtz`	0	0%
Store	911	12.9%	`div`	7	0.1%	`bgez`	14	0.2%
`move`	262	3.7%[1]	`mflo`	10	0.2%[3]	Cbranch	514	7.3%
Move	3404	48.3%	`mfhi`	2	0.0%	`b`	320	4.5%
			Arithmetic	854	12.1%	`jr`	130	1.8%[5]
			`nor`	8	0.1%	Ubranch	450	6.4%
			`and`	5	0.1%	`bgezal`	1	0.0%
			`andi`	10	0.2%	`jalr`	539	7.7%
			`or`	1	0.0%	Call	540	7.7%[6]
			`ori`	1	0.0%	Control	1504	21.4%
			`xor`	15	0.2%	Noop	1209	17.2%
			`sllv`	3	0.1%[2]	*Total*	*7041*	*100%*
			`srl`	6	0.1%			
			`srlv`	2	0.0%			
			Logical	51	0.7%			
			`slt`	8	0.1%			
			`sltiu`	11	0.2%			
			Boolean	19	0.3%[4]			
			Compute	924	13.1%			

Table 6-4: Instruction Counts – MIPS

Notes

1. A move from one register to another is suspect, since it might be due to inadequate targeting. These instructions were checked by hand, and 38 were found to be removable by arbitrarily better code generation (14% of the moves or 0.5% of the code). The remainder were genuine.

2. Most left shifts are optimizations of multiplication and are counted as arithmetic; only a few are true logical shifts.

3. Every `mul` or `div` must be followed by a `mflo` or `mfhi` to collect the results of the product, quotient, or remainder.

4. This is a false picture. In several places, the source code uses a conditional statement returning TRUE or FALSE instead of a pure Boolean expression. Hand checking shows that there should be about three times as many uses of these instructions (about 1% overall).

5. This is 121 procedure returns, 2 true jumps, and 7 case statements implemented as jump tables. Each procedure has exactly one return jump: in order to improve comparability with *cgvax*, we inhibited code hoisting.

6. This is 540 calls in 121 procedures.

If the `nop` instructions are excluded, the instruction mix is as shown in figure 6-2:

Figure 6-2: BCPL / MIPS M/500 Instruction Mix

This figure shows a fairly typical pattern for a load/store machine. The approximate breakdown of 60% load/store, 15% compute, and 25% control is not unfamiliar. However, it underlines the need for good data caching and wide bandwidth to memory. The high proportion of control instructions is typical of systems code; it shows that a good branch cache or instruction cache is desirable.

The proportion of stores to loads is about 2 : 5, and stores are about 27% of all moves. This is a little higher than one would expect; the reason is that register tracking is eliminating a lot of loads. In detail, we save:

- 570 loads of 0,+1,-1

- 2 loads of other values

- 425 loads from memory

for a total of 997. This is a saving of over 30%.

The very small number of byte loads and stores seems surprising, since the program being analyzed reads and writes text files. However, almost all the code treats strings as atomic objects passed by reference; only in a few primitive routines are the individual characters accessed.

Overall, the only instructions that seem underused are the Boolean `seq` group. As noted, this is partly an artificial result; but at best they would be used only 1% of the time. However, in the true MIPS architecture, they are used to implement some of the branch macro-instructions, so they probably come for free. Moreover, if they were absent, the code sequence that the compiler would have to generate would be quite expensive.

The `nor` instruction was never generated by *cgmips* even though a specific optimization was added to look for a chance to use it. It appears in the reorganized code only as the translation of `not`.

Address Mode Usage					
Conceptual			**Physical**		
Constant	2689	21.2%	Immediate	2119	16.7%
Local	1461	11.5%[1]	Absolute	0	0.0%[4]
Protocol	865	6.8%[2]	Register	7822	61.8%
Static	1105	8.7%	Based	2716	21.5%[6]
Indirect	297	2.3%[3]			
Temporary	6240	49.3%			
Total	10166	100%	Total	10166	100%

Table 6-5: Address Mode Usage – MIPS

In addition, there were 753 branch targets.

Offset and Constant Sizes	
Constant, 0	358
Constant, +1	152
Constant, -1	60[5]
Immediate, 16 bits	2115
Immediate, 32 bits	4
All Numbers	2689
Stack-based, 16 bits	1467
Stack-based, 32 bits	0
pointer-based, no offset	120
pointer-based, 16 bits	177
pointer-based, 32 bits	0

Table 6-6: Offset and Constant Sizes – MIPS

Notes

1. The distribution between value moves and address loads is:

	Value	*Address*	*Total*
Local	1455	6	1461
Static	968	137	1105

2. This refers to operations implementing the procedure entry/exit protocol.

3. This includes all pointer, structure, and array references.

4. The Absolute address mode cannot be generated by this language.

5. Recall that -1 is the BCPL representation of TRUE.

6. Based address mode is of the form `displacement (register)`.

This data is a compelling vindication of the RISC design. The machine has just three data address modes, and they are used in the ratio 17% : 62% : 21%. In fact, register tracking, and the use of three registers to hold constants, inflates the middle figure – for unoptimized code the pattern would be closer to 21% : 53% : 26%. Note that 50% register operands is the minimum possible, since a simple assignment generates two memory references and two register references, while any more complicated expression generates more register references than memory references.

The `li` instruction adds a 16-bit immediate value to the contents of a register and loads the result. This allows it to serve as a *load address* instruction, and it appears in the high level instruction set as `la`. This is clearly a good idea: over 6% of references to based addresses use this idiom.

Inspection of the generated code shows just two places where some further economy could be achieved:

1. The only way to access static tables with a short address mode was to put them in the `.sdata` segment, even though they were conceptually read-only. This could be ameliorated by having a PC-relative address mode, or by allowing the user to set up global base registers.

2. The majority of the conditional branches were of the form "compare register and small constant". A true MIPS instruction that implemented

   ```
   <conditional-branch> <register> <immed> <destination>
   ```

   would be very useful, though we agree it would be hard to fit into 32 bits. With the present machine, this expands into two true instructions (8 bytes); by contrast, the same idiom on the VAX usually takes 5 bytes.

The pattern found for immediate values and offsets confirms that a mode with a 32-bit offset is unnecessary. However, it would be helpful if the MIPS assembler allowed the programmer to use general registers as global base registers, instead of keeping this ability strictly to itself (and not using it to best advantage).

6.2.4. Register Usage – MIPS

The Ocode code generator uses `u0` through `u13` as accumulators and for parameter passing. Up to 14 parameters are passed in registers; any additional parameters are passed on the stack. Results are returned in `u0`. The same registers are used in round-robin fashion for temporaries, starting with `u0` for the first temporary of each basic block. All registers are tracked across linear code and non-looping control structures. All accumulators are assumed destroyed by a procedure call. (This is not the protocol of the other MIPS compilers.)

The registers `rz`, `ru`, and `rm` always hold the values 0, +1, and -1, respectively. Register `rp` is the Ocode stack pointer `rl` is the return link register, and `rw` is a work register.

The accumulator pattern is, as expected, very close to a negative-binomial distribution. It illustrates well the way benefits rapidly diminish with this allocation strategy. Interprocedural register allocation would be a better (but harder) strategy.

There are 2409 references to the Ocode stack pointer, `rp`. Of these, 1461 are accesses to local variables, and 942 are generated by 471 instructions to raise or lower the stack. The stack is moved by the caller before and after every procedure call; canonically that would be 2*540 = 1080 moves, but optimizations remove 609 of them (56.8%), giving a much faster protocol than the conventional one in which the called procedure moves the stack.

The temporary register `rw` is used during a procedure call. This is necessary because BCPL calls procedures indirectly through a transfer vector, so the call sequence is:

```
lw      rw, procoffset(vector)
jalr    rw
```

giving 2*539 = 1078 uses of `rw` for 539 calls of external procedures.

Register Usage				
Accumulators		**Special Registers**		
u0	1778	rz	358	(holds 0)
u1	912	ru	152	(holds +1)
u2	542	rm	60	(holds -1 or TRUE)
u3	366	rp	2409	(stack pointer)
u4	250	rw	1078	(temporary)
u5	157	rl	363	(return link)
u6	120	gp	1096	(MIPS sdata base register)
u7	79			
u8	66			
u9	50			
u10	38			
u11	24			
u12	21			
u13	15			
Total	*4418*	*Total*	*5516*	

Table 6-7: Register Usage – MIPS

6.2.5. Instruction Set Usage – VAX

Here are the same data for the VAX, using the same source program, but compiled by *bcpl* and *cgvax*. It is much harder to understand these tables, since there are many special idioms that perform actions that are not obvious. For example, a constant can be loaded into a register by any of the following:

```
clrl     r0
movl     #1,r0
mcoml    #63,r0
movzbl   #200,r0
cvtbl    #-100,r0
movzwl   #300,r0
cvtwl    #-200,r0
```

and a constant can be added to a register by:

```
incl     r0
decl     r0
addl2    #2,r0
subl2    #2,r0
moval    100(r0),r0
```

Although there are other methods for achieving these results, the reader is assured that each example is indeed the shortest way to accomplish that operation with that specific constant.

The Ocode code generator uses `jsb` exclusively for calls, and builds its own stack on `r12`. There are therefore no occurrences of `callx`, `pushx`, `ret`, or `rsb`.

movb	1	0.0%	**mnegl**	9	0.2%	**tstl**	105	2.5%[4]
clrb	1	0.0%	**incl**	32	0.8%	**cmpl**	266	6.4%
clrl	137	3.3%	**addl**	300	7.2%	**bitl**	0	0.0%
clrq	1	0.0%	**decl**	28	0.7%	**cmpv**	0	0.0%
movl	1194	28.7%	**subl**	212	5.1%	**cmpzv**	0	0.0%
movq	107	2.6%	**moval**	28	0.7%[3]	Compare	371	8.9%
mcoml	58	1.4%[1]	**tstl**	44	1.1%[4]	**beql**	99	2.4%
cvtbl	0	0.0%	**mull**	21	0.5%	**bneq**	182	4.4%
cvtwl	0	0.0%	**divl**	5	0.1%	**blss**	38	0.9%
cvtlb	7	0.2%	**emul**	2	0.0%	**bgtr**	28	0.7%
movzbl	21	0.5%[2]	**ediv**	2	0.0%	**bleq**	32	0.8%
movzwl	5	0.1%	Arithmetic	683	16.4%	**bgeq**	24	0.6%
moval	132	3.1%[3]	**mcoml**	8	0.2%[1]	**blssu**	0	0.0%[5]
Move	1664	40.0%	**bisl**	1	0.0%	**bgtru**	0	0.0%
			bicl	14	0.3%	**blequ**	0	0.0%
			xorl	4	0.1%	**bgequ**	0	0.0%
			rotl	0	0.0%	**casel**	12	0.3%[6]
			ashl	3	0.1%	Cbranch	415	10.0%
			ashq	1	0.0%	**brb**	257	6.2%
			extv	0	0.0%	**brw**	74	1.8%
			extzv	7	0.2%	**jmp**	121	2.9%[7]
			insv	0	0.0%	Ubranch	452	10.9%
			Logical	38	0.9%	**bsbb**	0	0.0%
			Compute	721	17.3%	**bsbw**	1	0.0%
						jsb	539	13.0%
						Call	540	13.0%[8]
						Control	1778	42.7%
						Total	4163	100%

Table 6-8: Instruction Usage – Vax

Notes

1. Most uses of `mcoml` are to load a small negative number, and these are considered moves. A few are genuine bitwise complement operations, and so are considered logical.

2. Most uses of `movzbl`, and all uses of `movzwl`, are to load medium-sized constants.

3. Some uses of `moval` were to add a constant to a register, and these are considered adds. The remainder are genuine moves of addresses.

4. The idiom `tstl (r12)+` is sometimes used to add 4 to the Ocode stack pointer. These are considered additions; the other occurrences of `tstl` are true tests.

5. This language cannot generate unsigned comparisons.

6. That is, one for each case statement implemented as a jump table. The code generator algorithm for choosing between a table and a sequence of tests depends on the number of cases and their sparsity. Since the VAX form of the jump table is half the size of the MIPS form, this algorithm chooses jump tables more often on the VAX.

7. The `jmp` instruction is used only to implement a procedure return. This version of *cgvax* did not support any code hoisting; there are therefore 121 returns in 121 procedures.

8. This is 540 calls in 121 procedures.

This is a different pattern from that found on the MIPS. The main differences of interest are discussed in the next section.

Address Mode Usage					
Conceptual			Physical		
Constant	1380	22.5%[1]	Literal	982	16.4%
Local	1181	19.2%[2]	Indexed	68	1.1%
Protocol	363	6.0%	Register	1946	32.5%
Static	1094	17.8%[3]	Reg deferred	47	0.8%[5]
Indirect	245	4.0%	AutoDecrement	124	2.1%[6]
Indexed	68	1.1%[4]	AutoIncrement	143	2.4%[6]
Temporary	1808	29.4%	AutoInc deferred	121	2.0%[6]
			Displacement	1798	30.0%
			Disp deferred	539	9.0%[7]
			Immediate	93	1.6%
			Absolute	0	
			Relative	126	2.1%
			Rel deferred	0	
Total	6139	100%	Total	5987	100%

Table 6-9: Address Mode Usage – VAX

In addition, there are 720 branch addresses.

Offset and Constant Sizes	
Constant, 0 (`clr`/`tst`)	245
Constant, +1 (`inc`/`dec`)	60[8]
Literal, 6 bits	982
Immediate, 8 bits	84
Immediate, 16 bits	5
Immediate, 32 bits	4
Stack-based, 8 bits	1146
Stack-based, 16 bits	35
Stack-based, 32 bits	0
Pointer-based, no offset	47
Pointer-based, 8 bits	175
Pointer-based, 16 bits	23
Pointer-based, 32 bits	0

Table 6-10: Offset and Constant Sizes – VAX

Notes

1. Of these, 245 zeros are elided into `clr` or `tst` instructions, and 60 occurrences of unity are elided into `inc` or `dec`, leaving 1075 explicit constant operands.

2. Of these, 1172 are to load or store plain values, 6 are to generate the address of local variables, and 3 are indirect accesses through a local pointer.

3. Of these, 422 are plain loads and stores, 126 are to generate addresses, and 536 are indirect accesses through a pointer.

4. An indexed operand also has a base address that is a second logical operand. The base operands are distributed thus:

     • Temporary – 2

     • Static pointer – 35

     • Local pointer – 31.

5. Plus 2 that are base addresses of index mode.

6. These modes are never generated for true operand access. They occur only as part of the procedure entry/exit protocol and as special idioms.

7. Plus 66 that are base addresses of index mode.

8. It is advantageous on the VAX to avoid small negative numbers, e.g.,:

```
addl #-1,r0  → subl #1,r0
movl #-1,x   → mcoml #0,x
```

   Hence, the constant -1 rarely occurs in the generated code.

These statistics are very confusing. However, two things seem clear. First, the 8-bit offset mode, and the 6- and 8-bit literal modes, amply justify themselves. They account for 96% of all offsets and 99% of all literals.

Secondly, the majority of the address modes are hardly ever used. If we exclude the modes generated only by hand-crafted protocol sequences, then just three modes – literal, register, and displacement – account for almost 80% of all operands. The most significant remaining mode – displacement deferred – is generated only by the BCPL calling sequence for external procedures.

6.2.6. Register Usage – VAX

The Ocode code generator uses `r0` through `r7` as accumulators and for parameter passing. Up to 8 parameters are passed in registers; any additional parameters are passed on the stack. Results are returned in `r0`. The same registers are used in round-robin fashion for temporaries, with the exact same conventions as on the MIPS. The register pair `<r7,r8>` is used as a special accumulator for instructions that require two registers, such as `ashq` or `ediv`.

Register `r12` is the Ocode stack pointer, used to address local variables. Register `r10` is the static database pointer, which has much the same purpose as `gp` on the MIPS. The hardware stack pointed to by `sp` is never used, but the return link must be popped off it on entry to every procedure, hence there are 121 references.

Register Usage			
Accumulators		**Special Registers**	
r0	1187	r10	993
r1	369	r12	1941
r2	141	sp	121
r3	56		
r4	29		
r5	15		
r6	0		
r7	8		
r8	3		
Total	*1808*	*Total*	*3055*

Table 6-11: Register Usage – VAX

Once again, the pattern of accumulator usage is typical. VAX code generation seems to require far fewer registers than MIPS code generation, but this is largely a figment of the round-robin strategy which tries to avoid reusing registers when fresh ones are available. Further analysis shows that 6 registers on VAX, or 8 on MIPS, would be enough to allow both good register tracking and efficient expression evaluation. Note, however, that the code generator does not bind local variables to registers.

There are 1941 references to the Ocode stack pointer, `r12`. Of these, 1181 are references to local variables, 242 are part of the procedure entry/exit protocol, and the rest are generated by 475 instructions to move the stack. The strategy on the VAX is the same as on MIPS: the caller moves the stack, and of the canonical 1080 moves required by 540 calls, the code generator can remove 605 (56.2%). This optimization leaves the stack pointer biased between two successive procedure calls; access to local variables then uses a **negative** offset from it. The possibility of this optimization is one reason for preferring offsets from a base register to be signed.

6.2.7. Architectural Comparison

6.2.7.1. Move Versus Load/Store
The VAX instruction breakdown shows far fewer move instructions. This is, of course, because the MIPS is a load/store machine, whereas the VAX may be used as a multi-address machine. Thus, a simple copy

```
a := b
```

is two instructions on MIPS

```
lw      reg,a
sw      reg,b
```

but one instruction on the VAX

```
movl    a,b
```

And a simple addition

```
a := a+b
```

is four instructions on MIPS

```
lw      r1,a
lw      r2,b
add     r2,r2,r1
sw      r2,a
```

but again only one instruction on the VAX

```
addl2   b,a
```

The effect of this is to inflate the number of moves. However, this effect is mitigated by optimization. If the value in A is to be used again, it is often better to compute that value in a register:

```
addl3   a,b,r0
movl    r0,a
```

A sampling of the code shows that about one-half of the "extra" MIPS moves were used to load the right operand of an operation, confirming the popular view that a general-register one-address organization is the best compromise between instruction density and simplicity. A load/store machine generates more instructions but can perform better overall because, since the fetch of an operand is not tightly coupled to its use, the fetch delay can be overlapped with useful work.

6.2.7.2. Three-Address Idiom

Another feature of the VAX is the "three-address" instructions that allow one, for example, to translate

```
a := b + c
```

as

```
addl3   b,c,a
```

These are available for most dyadic operations, and their pattern of use is shown in table 6-12.

Instruction	3-address	Total
addl	38	300
subl	38	212
mull	8	21
divl	5	5
bisl	0	1
bicl	11	14
xorl	2	4
Total	*102* (18.3%)	*557*

Table 6-12: Three Address Mode Usage

The code generator tries very hard to generate the three-address form to save register traffic. However, on the basis of these figures, it is barely worth having: it saved about 6% of the move instructions.

6.2.7.3. Condition Codes and Branches

The pattern of conditional branches is slightly different between MIPS and VAX. This is because *cgvax* looks for idioms such as:

- x ≥ 1 → x > 0 (saves 1 byte)
- x ≥ 64 → x > 63 (saves 4 bytes)

There are fewer branches overall because the VAX code implements more case statements as jump tables, and because the VAX case instruction includes a range check.

However, the VAX code has a far higher proportion of control instructions. One reason is that there are fewer instructions overall, so the same number of control transfers is a larger proportion. But there are also absolutely more such instructions: 1778 versus 1504.

This difference is almost entirely because of the tstl and cmpl instructions. The code generator slaves the condition codes religiously, both through linear code and across control transfers. Nevertheless, of 415 conditional branches, 371 (almost 90%) required a prior test or compare to set the condition codes; of 2169 normal instructions that set the condition codes, only 44 (about 2%) did so to any purpose. It is hard to avoid the conclusion that condition codes are a waste of time, effort, and silicon.

The MIPS machine is not perfect, however. First, because a full set of conditional branches is not available, 82 "set" instructions had to be generated to prepare for 432 branches. There is another, more difficult, problem. The most common kind of test in the program being analyzed is:

```
IF <component-of-structure> = <small-constant> THEN...
```

where the small constant represents a value of a scalar type. If we assume that a pointer to the structure is already in a register, then the VAX code looks like:

```
cmpl    offset(r1), #constant
bneq    else
```

which is 2 instructions and 6 bytes. The MIPS code looks like:

```
lw      u2, offset(u1)
li      at, constant
bneq    u2, at, else
```

which is 3 instructions and 12 bytes. The first instruction is a consequence of the load/store architecture, and can sometimes be optimized out. The second instruction is necessary because the branch operations do not take an immediate operand. Note also that it might be necessary to append a no-op after the branch.

The MIPS instructions have room for a 16-bit relative branch, or a 16-bit immediate operand, but not both. The VAX code is much denser, in part because it uses an 8-bit field for both the relative branch and the immediate operand; for the MIPS to achieve a density it would have to either abandon the fixed 32-bit instruction format or use smaller field sizes in this special case.

It seems that smaller field sizes would improve code density: over 95% of constants would fit, and over 90% of branch destinations (in fact, 12% of the branches on the VAX are not within 8-bit range, but that is with a relative byte address; MIPS uses a relative word address). These branch instructions are perhaps the part of the MIPS order code that suffers most from the simplification of the RISC design.

6.2.7.4. Index Mode

The number of adds and left shifts is much lower on the VAX because of the scaled-index mode. For a simple language, where nearly all arrays have word-sized components, this mode can be used for most array references. For example, to load the value of `ary[i]` into a register:

```
VAX:
        movl    i, rx
        movl    @ary[rx],r0

Mips:
        lw      rx,I
        sll     rx,rx,2
        add     rx,rx,ary
        lw      u0,0(rx)
```

There are in fact 68 occurrences of this mode in a total of 3930 operand references (1.7%).

The existence of a scaled-index mode cannot be justified by these figures. But this is systems code, which has few arrays references. However, compilers for scientific languages implement very

sophisticated loop induction optimizations, which tend to eliminate the need for index scaling. Moreover, the mode is useless for arrays whose components are size other than 1,2,4, or 8 bytes.

6.2.8. Local Conclusions

We can draw the following tentative conclusions:

1. For simple systems programming, a RISC machine is as effective as a CISC machine, and potentially a lot faster.

2. Leaving aside code reorganization, it is certainly no harder to generate code for a RISC machine, and in many respects it is easier. Moreover, a preliminary study of the problem suggests that reasonable code reorganization can be added with little extra effort.

3. In the main areas where RISC machines differ from CISC – simpler instructions, fewer address modes, no side effects – the RISC design is rarely inferior and usually superior.

4. However, the basic system software of the machine must be fast and efficient.

In addition, the specific claims made about the RISC machine under study are corroborated by this work.

6.3. Dynamic Analysis of Compilers

In this section we describe a brute-force analysis of the code generated by the MIPS and VAX compilers. This section contrasts the approach taken in section 6.2, which instrumented a compiler and examined the output. Here, we look at the instruction mix that is output by the compilers in response to two sets of input: a set of integer application programs, and a single large floating point application.

6.3.1. Instruction Use by Integer Applications

The first test we subjected the compilers to was the compilation of a set of integer application programs. The three programs we chose were:

1. *csh* – the UNIX C-Shell. This program is a command interpreter whose function is to scan user commands and run system and user programs. This program consists of nearly 16,000 lines of source code and comments.

2. *vi* – a UNIX visual editor. This program is a terminal-independent screen editor. It provides all of the standard editor functions in a screen optimal fashion, updating as each change is made. This program contains over 20,000 lines of source code and comments.

3. *uboat* – a proprietary authoring language. This program provides a terminal independent foundation for writing computer aided courseware, menu systems, and demonstration drivers. It contains of almost 10,000 lines of source code and comments.

These three programs were chosen as reasonable representatives of integer-based application programs. By their very nature, none of them are highly compute intensive, although they do perform a great deal of data manipulation. We present the statistics for the three programs together,

rather than inundating the reader with individual analyses. In truth, the compiler generated roughly the same instruction mix for each program, so we present the average mix for each compiler.

6.3.1.1. Analysis of MIPS C Compiler

Table 6-13 shows the instruction mix generated for the three integer applications. We list the actual MIPS M/500 instructions that were generated instead of the high-level instruction set. The reason for this is that the low-level instructions are the ones that are actually executed, thus, their frequency of occurrence is much more significant than the high level macro instructions.[43]

_____

[43]The instruction counts shown in table 6-13 correspond to the output of the compiler at optimization level 4 (this includes cross module register allocation and optimization, and requires that all modules be compiled together). We also did *not* count the instructions in the run-time libraries or the C initialization or finalization routines.

`lb`	13	0.02%	`add.d`	3	0.00%	`beq`	4096	5.34%
`lbu`	1778	2.32%	`addiu`	7032	9.16%	`bgez`	219	0.29%
`lh`	2130	2.77%	`addu`	1098	1.43%	`bgtz`	92	0.12%
`lhu`	159	0.21%	`div`	85	0.11%	`blez`	181	0.24%
`li`	3871	5.04%	`divu`	3	0.00%	`bltz`	154	0.20%
`lui`	1259	1.64%	`mul.d`	5	0.01%	`bne`	3325	4.33%
`lw`	9650	12.57%	`multu`	42	0.05%	CBranch	8067	10.50%
`lwc1`	6	0.01%	`sll`	1367	1.78%	`b`	2992	3.90%
Load	18866	24.57%	`sllv`	10	0.01%	`break`	113	0.15%
`sb`	930	1.21%	`sra`	762	0.99%	`jr`	1041	1.36%
`sh`	1213	1.58%	`srav`	4	0.01%	UBranch	4146	5.40%
`sw`	5656	7.37%	`srl`	5	0.01%	`jal`	6090	7.93%
`swc1`	2	0.00%	`subu`	549	0.72%	`jalr`	37	0.05%
Store	7801	10.16%	Arithmetic	10965	14.34%	Call	6127	7.98%
`cvt.d.s`	14	0.02%	`and`	85	0.11%	Control	18340	23.89%
`cvt.s.d`	5	0.01%	`andi`	1003	1.31%	nop	11074	14.43%
`cvt.w.d`	3	0.00%	`nor`	3	0.00%	*Total*	*76762*	*100%*
`mfc1`	3	0.00%	`or`	24	0.03%			
`mfhi`	40	0.05%	`ori`	206	0.27%			
`mflo`	103	0.13%	`xor`	32	0.04%			
`move`	6635	8.64%	`xori`	87	0.11%			
`mtc1`	11	0.01%	Logical	1440	1.87%			
Shuffle	6814	8.87%	`cfc1`	6	0.01%			
Move	33481	43.61%	`ctc1`	6	0.01%			
			`slt`	383	0.50%			
			`slti`	286	0.37%			
			`sltiu`	282	0.37%			
			`sltu`	499	0.65%			
			Boolean	1462	1.90%			
			Compute	13889	18.12%			

Table 6-13: Integer Application Instruction Usage – MIPS

The first observation we make is that there is a distressingly large number of `nop` instructions in the final executable code – over 14% of the total instruction count are `nops`. Figure 6-3 displays the instruction mix in graphical form.

Although the MIPS compiler is generating fairly good code, a more sophisticated code generator could create programs that run another 10% faster (based on the work described in section 6.2.2, page 55).[44]

---

[44]It should be noted that while the instruction mix shown in figure 6-13 represents the instruction frequencies that are present in the executable image, and not necessarily the frequency of instructions that are executed, we have found that the concurrence between these two figures is usually very high.

Figure 6-3: Instruction Distribution – Integer Applications

When the **nop** instructions are excluded, the resulting instruction mix follows the pattern shown in figure 6-4.

Figure 6-4: Instruction Distribution – Integer Applications (Minus **nop**s)

This instruction breakdown correlates roughly with the mix shown in figure 6-2 on page 57. In these examples, however, the ratio of compute instructions to move instructions is somewhat higher than the standard 15 : 25 mix of a load/store architecture. This is attributable to three factors:

1. The applications use more dynamic (i.e., local or register) variables than static variables; thus, fewer load/store operations are necessary.

2. The applications themselves are performing more computational actions than the standard program.

3. Perhaps more likely the MIPS compilers are sufficiently well tuned to efficiently reduce the total number of load/store operations that need to be performed, and instead turn the major effort more towards actual computation. We feel that this is a more likely explanation, since the level 4 optimizer has an interprocedural optimizer and register allocation mechanism (a feature that is lacking in the BCPL compiler discussed in section 6.2).

The address mode usage by these integer applications is shown in table 6-14. These figures correspond very closely to those in table 6-5 on page 58. This is not surprising, since all the application programs are similar in their general nature.

Address Mode Usage		
Immediate	27333	19.2%
Absolute	6089	4.2%
Register	87085	61.2%
Displacement	21537	15.1%
Floating-point	102	<0.1%
Total	*142146*	*100%*

Table 6-14: Address Mode Usage – Integer Applications on MIPS

Examining the register usage pattern shown in table 6-15 shows a number of interesting things:

- The large number of direct references to the stack pointer **sp** is caused by every routine moving the stack at entry and exit with an **addiu** instruction. There are two references to **sp** per instruction, and if the number of references is divided by 4, the result is 3660 / 4 = 915, the number of procedures defined in the three applications.

- The even larger number of indirect references to **sp** are caused by saving and restoring local registers per procedure.

- Because the compiler partitions registers into classes (rather than considering them identical), some observations about register usage are muddied. However, we may make the following general statements:

  - Temporary registers are allocated on a round-robin basis, and so show a fairly uniform distribution of use. Registers **t6**, **t7**, and **t8** are usually allocated first, and thus their reference count is a fraction higher than the other temporaries.

  - The saved registers **s0** through **s9** are used for local variables and must be saved across procedure calls. They are allocated in order and show a steadily decreasing frequency of reference from **s0** through **s9**, indicating that there are more procedures with a small number of local variables than there are with a large number of them.

  - The number of references to the argument registers follows a pattern that supports our statements in footnote [29] on page 36, to the effect that most procedures are called with four or less parameters. The number of references to **a3** (the fourth parameter register) account for less than 3% of the total references to the argument registers.

- The assembler temporary register **at** is used in 8% of all direct register references, indicating a fairly high percentage of interaction with the assembler reorganizer. It is likely that a large fraction of these references (and their instructions) could be eliminated were the compilers to deal directly with the low level instruction set, instead of the high-level macro instruction set.

- The kernel registers **k0** and **k1** are never referenced (not surprisingly). They are used exclusively by the MIPS UNIX kernel.

Integer			Floating Point	
Register	**Value**	**Offset**	**Register**	**Total**
zero	7431	0	f0	2
at	7164	514	f1	0
v0	8732	419	f2	0
v1	4015	241	f3	0
a0	7668	290	f4	10
a1	3870	110	f5	0
a2	1634	57	f6	10
a3	372	16	f7	2
t0	1681	99	f8	8
t1	1526	136	f9	0
t2	1355	79	f10	8
t3	1310	62	f11	0
t4	1283	65	f12	0
t5	1168	73	f13	0
t6	2659	142	f14	0
t7	2377	116	f15	0
s0	6329	977	f16	14
s1	4196	759	f17	2
s2	3150	265	f18	8
s3	2193	149	f19	2
s4	1417	83	f20	20
s5	1166	58	f21	4
s6	875	19	f22	0
s7	708	13	f23	0
t8	2070	114	f24	0
t9	1891	112	f25	0
k0	0	0	f26	0
k1	0	0	f27	0
gp	1745	7705	f28	0
sp	3660	8853	f29	0
fp/s8	577	11	f30	0
ra	2823	0	f31	12
hi	40	0		
lo	0	0		
Total	*87075*	*21537*	*Total*	*102*

Table 6-15: Register Usage – Integer Applications on MIPS

6.3.1.2. Comparison with VAX UNIX C Compiler

When the same integer application programs were fed through the Berkeley VAX C compiler, the instruction mix that was observed is shown in table 6-16.

clrb	342	0.8%	**addd2**	4	0.0%	**bitb**	25	0.1%
clrl	830	1.8%	**addl2**	638	1.4%	**bitl**	22	0.0%
clrw	293	0.6%	**addl3**	486	1.1%	**bitw**	71	0.2%
cvtbl	566	1.2%	**decb**	1	0.0%	**cmpb**	420	0.9%
cvtbw	9	0.0%	**decl**	366	0.8%	**cmpl**	2428	5.3%
cvtdf	2	0.0%	**decw**	29	0.1%	**cmpw**	252	0.6%
cvtdl	2	0.0%	**divd2**	2	0.0%	**tstb**	641	1.4%
cvtfd	7	0.0%	**divd3**	1	0.0%	**tstl**	1756	3.9%
cvtlb	361	0.8%	**divl2**	73	0.2%	**tstw**	691	1.5%
cvtld	6	0.0%	**divl3**	104	0.2%	Compare	6306	13.8%
cvtlw	521	1.1%	**incb**	33	0.1%			
cvtwb	5	0.0%	**incl**	640	1.4%	**acbl**	7	0.0%
cvtwl	1296	2.8%	**incw**	81	0.2%	**aobleq**	5	0.0%
mcoml	13	0.0%	**muld2**	5	0.0%	**aoblss**	14	0.0%
mnegb	8	0.0%	**muld3**	2	0.0%	**casel**	30	0.1%
mnegl	93	0.2%	**mull2**	154	0.3%	**jbc**	181	0.4%
mnegw	32	0.1%	**mull3**	158	0.3%	**jbcc**	9	0.0%
movab	118	0.3%	**subd3**	2	0.0%	**jbs**	83	0.2%
moval	649	1.4%	**subl2**	459	1.0%	**jbss**	42	0.1%
movaq	2	0.0%	**subl3**	530	1.2%	**jeql**	2862	6.3%
movb	219	0.5%	Arithmetic	3768	8.2%	**jgeq**	344	0.8%
movc3	15	0.0%				**jgequ**	3	0.0%
movd	5	0.0%	**ashl**	226	0.5%	**jgtr**	349	0.8%
movl	4689	10.3%	**bicb2**	4	0.0%	**jgtru**	5	0.0%
movq	2	0.0%	**bicl2**	49	0.1%	**jlbc**	39	0.1%
movw	137	0.3%	**bicl3**	61	0.1%	**jlbs**	13	0.0%
movzbl	107	0.2%	**bicw2**	24	0.1%	**jleq**	383	0.8%
movzbw	3	0.0%	**bisb2**	2	0.0%	**jlequ**	1	0.0%
movzwl	26	0.1%	**bisl2**	33	0.1%	**jlss**	422	0.9%
pushab	6	0.0%	**bisl3**	16	0.0%	**jneq**	2684	5.9%
pushal	2238	4.9%	**bisw2**	40	0.1%	**sobgeq**	12	0.0%
pushl	4986	10.9%	**extzv**	25	0.1%	**sobgtr**	7	0.0%
Move	17588	38.6%	**xorb2**	2	0.0%	Cbranch	7495	16.4%
			xorl2	6	0.0%			
			xorl3	3	0.0%	**jbr**	2559	5.6%
			Logical	491	1.0%	**ret**	1412	3.1%
			Compute	4259	9.3%	UBranch	3971	8.7%
						calls	5930	13.0%
						Control	23702	52.0%
						Total	*45549*	*100%*

Table 6-16: Integer Application Instruction Usage – VAX

The slightly lower percentage of move class instructions on the VAX is predictable, since the VAX is not a load/store architecture. However, the 38.6% figure is still higher than expected. What is most surprising is the markedly decreased number of compute instructions – a figure we expected to see increase when the move instructions decreased. The two fractions can be brought more at a par with each other when it is remembered that many of the `addiu` instructions on the MIPS M/500 are used to calculate addresses, not actual numeric results.

The number of call instructions is roughly the same on the VAX and the MIPS M/500, although due to the decreased number of instructions required on the CISC VAX, they comprise a larger percentage of the total. The larger fraction of conditional branches on the VAX is compensated somewhat on the MIPS M/500 by breaking conditionals into two parts, half of which are considered under booleans.

What is most interesting, however, is the under-use of the VAX instruction set. Many instructions are used only 0.1% or 0.2% of the time, indicating that a large amount of hardware effort is being spent for a very small software gain. When one considers the frequency with which the three operand address mode is used (shown in figure 6-5), we see that many features of the CISC instruction set are simply not used effectively at all.

Figure 6-5: Operand Type – Integer Applications on VAX

To further demonstrate this point, examine table 6-17, which shows the frequency of use of the various modes available on the VAX. When the VAX was first produced, the indexed addressing modes were claimed to be highly beneficial in array accessing. However, the indexed addressing modes are used little more than 0.6% of the time. Other addressing modes are similarly underused.

Address mode coverage			
Address Mode	Example	Count	Percentage
Immediate	$270	3701	5.4%
Literal	$24 (n < 64)	9518	14.0%
Absolute	$*label	0	0.0%
Absolute Indexed	$*label[r4]	0	0.0%
Relative	label	26367	38.9%
Relative Indexed	label[r4]	392	0.5%
Relative Deferred	*label	202	0.2%
Relative Deferred Indexed	*label[r4]	0	0.0%
Register	r3	16619	24.5%
Deferred	(r3)	1365	2.0%
Deferred Indexed	(r3)[r4]	78	0.1%
Autoincrement	(r3)+	394	0.5%
Autoincrement Indexed	(r3)+[r4]	0	0.0%
Deferred Autoincrement	*(r3)+	0	0.0%
Deferred Autoincrement Indexed	*(r3)+[r4]	0	0.0%
AutoDecrement	-(r3)	776	1.1%
AutoDecrement Indexed	-(r3)[r4]	0	0.0%
Displacement	24(r3)	7894	11.6%
Displacement Indexed	24(r3)[r4]	25	0.0%
Displacement Deferred	*24(r3)	419	0.6%
Displacement Deferred Indexed	*24(r3)[r4]	11	0.0%
Total		*67761*	*100%*

Table 6-17: Address Mode Usage – Integer Applications on VAX

In fact, when the use of the address modes is displayed graphically (as in figure 6-6), we see that 94.6% of the address modes used on the VAX are filled by immediate (literal being a subset of immediate), relative, register, and displacement modes – exactly the modes provided by the MIPS M/500 instruction set. Yet on the VAX each instruction must go through the effort of decoding which addressing mode is used, even though (for the most part) only 5 of the possible 16 VAX address modes are ever really used.

Table 6-18 shows another interesting artifact of the Berkeley C compiler (that serves to show off the MIPS compiler as a better example of compiler writing).

In the Berkeley compiler, local registers which are explicitly declared to be of type register are

Figure 6-6: Addressing Mode Usage – Integer Applications on VAX

Register Usage by Class			
	Value	Pointer	Index	Total
r0	8564	1014	314	**9892**
r1	1360	167	50	**1577**
r2	173	12	1	**186**
r3	5	0	0	**5**
r4	0	0	0	**0**
r5	0	0	0	**0**
r6	62	4	3	**69**
r7	95	21	6	**122**
r8	419	122	5	**546**
r9	1083	258	18	**1359**
r10	1939	687	39	**2655**
r11	2559	1337	70	**3966**
r12	8	2304	0	**2312**
r13	0	4192	0	**4192**
r14	352	844	0	**1196**
r15	0	0	0	**0**
Total	*16619*	*10962*	*506*	*28087*

Table 6-18: Register Usage – Integer Applications on VAX

allocated starting at register **r11**, working downwards. If a variable is not declared `register`, it is allocated on the stack. This explains the decreasing frequency of register references from **r11** to **r6**.

As an additional artifact, `r0` (and `r1`) are the function return registers, while registers `r4` and `r5` are rarely allocated, due to their interaction with the `movc` instructions.[45]

Registers `r12` and `r13` are the frame and argument pointers and are referenced almost exclusively as a pointer to the stack. The stack pointer `r14` is referenced both indirectly and directly.

6.3.2. Instruction Use by Floating-Point Applications

In this next test case, we gave the compiler a large floating-point application. We used the *SPICE* program, a large circuit-simulation program written in FORTRAN, consisting of over 18,000 lines of dense, ugly code and comments. We regret that only a single program was used in this test, however the instruction count generated by this program nearly equaled that of the combined integer applications, so we feel our choice was not a bad one. We realize that it is difficult to compare FORTRAN and C compilers, since the semantics of the source languages differ so greatly. However, since the code generator and optimizer in both the VAX and the MIPS programming environment are common to both languages, we feel that there is sufficient similarity between the two compilers to warrant a broad comparison.

6.3.2.1. Analysis of MIPS FORTRAN Compiler

Table 6-19 shows the instruction mix generated by the MIPS compiler for the *SPICE* program. As in section 6.3.1.1, we have listed only the low-level MIPS M/500 instructions that were generated by the level 4 optimizer. We have not counted the FORTRAN run-time library routines, or the FORTRAN initialization or finalization code.

---

[45]The DEC compilers do not suffer from these aberrations of register allocation behavior.

lb	1	0.0%	add.d	943	1.2%	bc1f	510	0.7%
lbu	12	0.0%	add.s	138	0.2%	bc1t	335	0.4%
lh	1	0.0%	addiu	5934	7.8%	beq	1147	1.5%
li	1928	2.5%	addu	4812	6.3%	bgez	32	0.0%
lui	6270	8.2%	div	33	0.0%	bgtz	27	0.0%
lw	7142	9.4%	div.d	726	1.0%	blez	45	0.1%
lwc1	10582	13.9%	mul.d	1977	2.6%	bltz	60	0.1%
Load	25936	34.1%	mul.s	218	0.3%	bne	811	1.1%
sb	7	0.0%	multu	30	0.0%	CBranch	2967	3.9%
sh	10	0.0%	neg.d	288	0.4%	b	1555	2.0%
sw	3573	4.7%	neg.s	12	0.0%	break	40	0.1%
swc1	6273	8.3%	sll	2617	3.4%	jr	179	0.2%
Store	9863	12.9%	sra	16	0.0%	UBranch	1774	2.3%
cvt.d.s	192	0.3%	sub.d	912	1.2%	jal	3120	4.1%
cvt.d.w	69	0.1%	sub.s	126	0.2%	Call	3120	4.1%
cvt.s.d	184	0.2%	subu	110	0.1%	Control	5053	6.6%
cvt.w.d	37	0.0%	Arithmetic	18892	24.8%	nop	5718	7.5%
mfc1	37	0.0%	ori	42	0.1%	*Total*	*76024*	*100%*
mfhi	4	0.0%	xori	39	0.1%			
mflo	58	0.1%	Logical	81	0.2%			
mov.d	634	0.8%	c.eq.d	269	0.4%			
mov.s	46	0.1%	c.le.d	416	0.5%			
move	2675	3.5%	c.lt.d	163	0.2%			
mtc1	2108	2.8%	cfc1	74	0.1%			
Move	6044	8.0%	ctc1	74	0.1%			
Shuffle	41361	54.3%	slt	381	0.5%			
			slti	148	0.2%			
			sltiu	64	0.1%			
			sltu	40	0.1%			
			Boolean	1629	2.1%			
			Compute	20602	27.0%			

Table 6-19: Floating-Point Application Instruction Usage – MIPS

The table of values for the floating-point performance of the compiler differs from the integer performance (shown in figures 6-3 and 6-4) in a number of ways. First, there are fewer **nop** instructions and a higher percentage of move instructions. This is shown graphically in figure 6-7.

Figure 6-7: Instruction Distribution – Floating-Point Application

The decreased number of **nop** instructions is somewhat surprising, given the increased number of load class instructions. The substantially decreased control operations, however, may offset this statistic.[46]

The decreased number of **nop** instructions does not imply that floating-point applications generate better code than integer applications, nor that FORTRAN generates better code than C. It is simply the nature of this particular program, which has a control structure that did not require the insertion of many **nop** instructions. On the other hand, though, the reader should be aware of "hidden" delays in the floating-point computations. While most MIPS M/500 instructions are executed in a single clock cycle, the floating-point instructions are not, and they require synchronization between the MIPS M/500 and the floating-point co-processor. In truth, then, the number of null operations that the MIPS M/500 is executing during floating-point operations is much higher than these tables of statistics would suggest.

Removing the **nop** instructions from consideration, we see the instruction mix shown in figure 6-8.

Figure 6-8: Instruction Distribution – Floating-Point Applications (Minus **nop**s)

This chart shows a much higher percentage of move class instructions than seen in figure 6-4, only a

---

[46]The load and jump/branch instructions have a delay slot following them that must be filled. If the assembler reorganizer is unable to move instructions around the load or jump/branch, it fills the delay slot with a **nop** instruction.

small fraction of which (8.5% of the total) are actual register-to-register movement. The dominating factor is load instructions. We suspect that this is a language and application dependency – the program makes heavy use of FORTRAN COMMON, a factor which effectively defeats interprocedural register allocation by making register slaving of the values of COMMON variables very difficult. Thus, the compiler is forced to load variables before each use. This is not a fault of the compiler, or of RISC architectures, but is a result of the antiquated nature of the FORTRAN language. The heavy use of global variables, a practice highly discouraged by most modern software engineering dogmas, extracts its price in program performance. This would also be the case if Pascal or another modular language used global variables with the frequency of FORTRAN. The MIPS FORTRAN compiler could be strengthened somewhat by placing the *addresses* of COMMON variables, or the address of the start of COMMON blocks into globally allocated registers. This would eliminate some of the `lui` and `addiu` instructions, which are currently used for accessing COMMON and passing parameters by reference.

Figure 6-8 chart also shows a much higher percentage of compute instructions, with a decreased percentage of control operations. We feel that this is another language and application artifact – FORTRAN is basically a "straight-line" language, with few deviations from the top-to-bottom execution model. The *SPICE* circuit simulator similarly has few decisions to make – most of the calculations, though elaborate, are rather straightforward.

The list of the frequency of address mode usage is shown in table 6-20. The pattern of usage is very similar to that shown for integer applications in table 6-14. The differences are that (obviously) a larger fraction of floating-point registers are used in the *SPICE* benchmark, and there is a slight increase in the use of displacement mode. This latter effect is probably caused by two factors – the large number of variables stored in `common`, and the fact that FORTRAN passes parameters to routines by reference instead of by value. Other than this, the addressing mode patterns are fairly consistent.

Address Mode Usage		
Immediate	21662	13.9%
Absolute	3078	1.9%
Register	64787	41.7%
Displacement	27601	17.7%
Floating-point	39188	25.2%
Total	*155316*	*100%*

Table 6-20: Address Mode Usage – Floating-Point Application on MIPS

The register usage patterns are shown in table 6-21. The frequency of use of many of the registers differs greatly from that of for the integer applications shown in table 6-15. This is caused by a number of factors:

- The assembler/reorganizer temporary register **at** is used more frequently in offset mode. This is due largely to the fact that COMMON variables are addressed relative to the base of their respective COMMON regions, and global address references translate to an offset from **at** by the assembler reorganizer.

Integer			Floating Point	
Register	Value	Offset	Register	Total
zero	3306	0	f0	2860
at	9825	5655	f1	636
v0	3179	1093	f2	1772
v1	1549	1077	f3	411
a0	3607	382	f4	4285
a1	2431	339	f5	1495
a2	1885	201	f6	4175
a3	1010	152	f7	1473
t0	1147	119	f8	4042
t1	1426	151	f9	1373
t2	1410	183	f10	4293
t3	1409	174	f11	1493
t4	1667	144	f12	1405
t5	1565	152	f13	260
t6	3334	422	f14	961
t7	3254	440	f15	171
s0	2671	940	f16	882
s1	2438	820	f17	215
s2	1709	455	f18	1059
s3	1439	287	f19	304
s4	1093	207	f20	1140
s5	905	433	f21	383
s6	930	114	f22	863
s7	978	21	f23	256
t8	3251	387	f24	751
t9	3126	436	f25	233
k0	0	0	f26	556
k1	0	0	f27	156
gp	1026	708	f28	494
sp	2018	12005	f29	140
fp	701	102	f30	406
ra	494	2	f31	245
hi	4	0		
lo	0	0		
Total	*64787*	*27601*	*Total*	*39188*

Table 6-21: Register Usage – Floating-Point Application on MIPS

- Temporary registers are allocated on a round-robin basis, and so show a fairly uniform distribution of use. Registers `t6`, `t7`, and `t8` are usually allocated first, and thus their reference count is a fraction higher than the other temporaries.

- Floating-point registers are used with much greater frequency (the *SPICE* circuit simulator *is* a floating-point program. The registers show an interesting pattern of use, though:

  - The odd numbered registers are used much less frequently than are the even numbered ones. This is because double precision floating-point numbers are stored in two registers (and referenced by the low order register of the pair). The vast majority of floating-point variables in *SPICE* are double precision variables.

  - The register allocation algorithm for floating-point variables does not appear to be the same round-robin scheme that is used for temporary registers. Instead, floating-point registers show a roughly exponentially decreasing frequency of use from register `f4` to `f30`.

- Subroutine parameters are passed in registers `a0` through `a4`, but the pattern seen in table 6-15 does not show up here. This is because double-precision floating-point variables are passed in two argument registers (instead of one for integer variables), and so the usage curve decays more slowly.

- The kernel registers `k0` and `k1` are never referenced (not surprisingly). They are used exclusively by the MIPS UNIX kernel.

- The saved registers `s0` through `s9` are used for local variables and must be saved across procedure calls. They are allocated in order, and show a steadily decreasing frequency of reference from `s0` through `s9`, indicating that there are more procedures with a small number of local variables than there are with large numbers of them.

6.3.3. Comparison with VAX UNIX FORTRAN Compiler

The *SPICE* benchmark was also given to the VAX FORTRAN compiler for comparison purposes. The data on instruction usage is shown in table 6-22.

`clrd`	248	0.7%	`addd2`	379	1.0%	`cmpd`	606	1.6%
`clrf`	2	0.0%	`addd3`	602	1.6%	`cmpl`	652	1.7%
`clrl`	281	0.7%	`addf3`	142	0.4%	`tstd`	236	0.6%
`cvtbl`	5	0.0%	`addl2`	381	1.0%	`tstl`	937	2.5%
`cvtdf`	190	0.5%	`addl3`	2826	7.4%	Compare	2431	6.3%
`cvtdl`	37	0.1%	`divd2`	170	0.4%	`acbl`	130	0.3%
`cvtfd`	362	1.0%	`divd3`	505	1.3%	`aobleq`	217	0.6%
`cvtld`	75	0.2%	`divl2`	12	0.0%	`casel`	31	0.1%
`cvtlw`	12	0.0%	`divl3`	33	0.1%	`jeql`	632	1.7%
`cvtwl`	16	0.0%	`incl`	4	0.0%	`jgeq`	192	0.5%
`mnegd`	298	0.8%	`muld2`	520	1.4%	`jgtr`	333	0.9%
`mnegf`	12	0.0%	`muld3`	1540	4.0%	`jleq`	251	0.7%
`mnegl`	14	0.0%	`mulf3`	222	0.6%	`jlss`	163	0.4%
`movab`	82	0.2%	`mull2`	11	0.0%	`jneq`	974	2.6%
`moval`	117	0.3%	`mull3`	52	0.1%	Cbranch	2923	7.6%
`movb`	1	0.0%	`subd2`	174	0.5%	`jbr`	1191	3.1%
`movd`	2774	7.3%	`subd3`	732	1.9%	`ret`	130	0.3%
`movf`	166	0.4%	`subf3`	126	0.3%	Ubranch	1321	3.4%
`movl`	9172	24.1%	`subl2`	231	0.6%	`calls`	2904	7.6%
`movw`	2	0.0%	`subl3`	263	0.7%	Control	9579	25.1%
`movzbl`	2	0.0%	Arithmetic	8925	23.4%	**Total**	**38095**	**100%**
`pushab`	1742	4.6%	`ashl`	313	0.8%			
`pushal`	2460	6.5%	Logical	313	0.8%			
`pushaq`	308	0.8%	Compute	9238	24.2%			
`pushaw`	2	0.0%						
`pushl`	898	2.4%						
Move	19278	50.6%						

Table 6-22: Floating-Point Application Instruction Usage – VAX

As with the MIPS instruction mix in table 6-19, we see a decrease in the number of control type instructions, and an increase in the number of arithmetic instructions. Again, we see the unusually high number of move instructions, even though the VAX is not a load/store architecture.

What is most interesting is the number of compare instructions in the VAX instruction mix. There are no compare instructions on the MIPS; instead, the instructions used to perform conditional branches contain the operands to be compared. On the VAX, two instructions need to be executed to perform most conditional branches: a compare and a branch. Rarely, if ever, are the condition codes used. Thus, even though the VAX has a more complex instruction set, the MIPS M/500 has the mechanism

for performing a conditional branch in a single instruction.[47]

Also as before, many instructions are underused. Instructions such as **mnegl** which moves the negative of a number into a register (saving 3 bytes of instruction), are used less than one-tenth of one percent of the time. It would be better to load a negative number directly, or to load a positive one and then negate it, than to waste the processor floorspace to implement the function in a single instruction that is rarely used.

The address mode usage on the VAX by the *SPICE* simulator is shown in table 6-23. With the exception of the 10% use of the Relative indexed mode, the distribution of address modes is similar to that shown in table 6-17.

Address Mode	Example	Count	Percentage
Immediate	`$270`	870	1.1%
Literal	`$24 (n < 64)`	5873	8.0%
Absolute	`$*label`	0	0.0%
Absolute Indexed	`$*label[r4]`	0	0.0%
Relative	`label`	16323	22.3%
Relative Indexed	`label[r4]`	7339	10.0%
Relative Deferred	`*label`	0	0.0%
Relative Deferred Indexed	`*label[r4]`	0	0.0%
Register	`r3`	18394	25.2%
Deferred	`(r3)`	233	0.3%
Deferred Indexed	`(r3)[r4]`	14	0.0%
Autoincrement	`(r3)+`	0	0.0%
Autoincrement Indexed	`(r3)+[r4]`	0	0.0%
Deferred Autoincrement	`*(r3)+`	0	0.0%
Deferred Autoincrement Indexed	`*(r3)+[r4]`	0	0.0%
AutoDecrement	`-(r3)`	431	0.5%
AutoDecrement Indexed	`-(r3)[r4]`	0	0.0%
Displacement	`24(r3)`	22217	30.4%
Displacement Indexed	`24(r3)[r4]`	284	0.3%
Displacement Deferred	`*24(r3)`	944	1.2%
Displacement Deferred Indexed	`*24(r3)[r4]`	18	0.0%
Total		*72930*	*100%*

Table 6-23: Address Mode Usage – Floating-Point Application on VAX

---

[47]On those occasions when the MIPS assembler reorganizer must expand a conditional to two or three instructions, the **sltx** and **xor** instructions are used. The total of these instructions does not come close to the amount of compare instructions used on the VAX. Apparently, then, the MIPS M/500 does conditional branches more efficiently than the VAX.

The extra high use of the Relative Indexed mode is either because of FORTRAN's parameter passing mechanism or its access to `common` arrays. Other than this, we make the same observation that we made for the integer applications: of the 16 addressing modes available on the VAX, only 5 are ever really used (basically the same addressing modes that are available on the MIPS M/500). The CPU could thus be substantially simplified without any major loss in efficiency of compiled code.

Looking at table 6-24, we see the same symptoms as we found in table 6-18, except that in this case, register allocation is even worse. Of the registers **r6** to **r11**, only **r11** is ever really used.

Register Usage by Class			
	Value	Pointer	Index	Total
r0	11351	127	4383	15861
r1	1228	91	555	1874
r2	2087	2	487	2576
r3	1	0	1	2
r4	87	0	0	87
r5	0	0	0	0
r6	227	4	16	247
r7	280	8	18	306
r8	391	11	45	447
r9	839	14	65	918
r10	1580	6	261	1847
r11	193	17150	1824	19167
r12	0	1277	0	1277
r13	0	5020	0	5020
r14	130	431	0	561
r15	0	0	0	0
Total	*18394*	*24143*	*7655*	*50192*

Table 6-24: Register Usage – Floating-Point Application on VAX

This underuse is predominantly a failing in the Berkeley FORTRAN compiler, and not inherent to the VAX. If the Berkeley compiler had an adequate register allocation algorithm, we would see a much better pattern of register use. As it is, however, some registers are over used, and some are badly underused.

6.3.4. Local Conclusions

It is interesting to note that the MIPS compilers generated 73 out of the 85 possible instructions[48] for the MIPS M/500 on these four programs. The use of 85% of the possible instructions attests to the validity of this test as a fair coverage of the instruction spectrum of a machine. When we look at the Berkeley VAX compilers, we find that of the 146 possible instructions, 111 (75%) were generated for our test programs.

If we examine instead the use of the entire instruction set by the compilers, we find that the MIPS compilers use 54% of the total MIPS M/500 instruction repertoire (73 out of 135 instructions), while the Berkeley compilers could only use 34% of the VAX instruction set (111 out of 323 instructions). The fact that, in both comparisons, a lower percentage of instructions was used by the VAX attests to the overcomplicated nature of the VAX CISC architecture.

If we compare the number of instructions that were generated, we find that the MIPS program (with 152786 instructions) used only 1.82 times more instructions than the VAX (with 83644 instructions). When the byte count is compared (a much more valid measure), the MIPS uses 611144 bytes versus the VAX's use of 474224 bytes, the code size increase is actually only 1.29 : 1. This is because MIPS instructions are always 4 bytes long, while VAX instructions vary in length depending on the addressing modes used. Since the MIPS M/500 is far more than 1.29 times faster than the VAX, we may assume that the penalty of more instructions being required to perform a task which is incurred by moving to a RISC architecture, is more than offset by the increased performance a RISC architecture provides.

From the three analyses that we have performed (static compiler analysis, an instrumented example, and dynamic compiler performance), the choice of a RISC architecture has won out over a CISC architecture. Each of the analyses, considered independently or collectively, shows that it is easier for a compiler to generate code for a RISC architecture, and that that code executes more efficiently. One might be tempted to look at the results from the VAX and conclude that the VAX compilers need to be made more robust. A better conclusion, however, is that the instructions and addressing modes that are not used by the VAX compilers are simply not needed.

---

[48]The possible instructions are those that the compiler can generate, not those that the MIPS M/500 can execute (see section 6.1.1.2)

---

7. General Drawbacks of Assembler-only Code Reorganization

The MIPS compiler suite uses an assembler reorganizer (described in chapter 3) to translate from a high-level assembly language to the MIPS M/500 native machine code. The assembler reorganizer also serves the function of making sure that the restrictions of the instruction pipeline are observed. These restrictions include a one-cycle delay following:

- a branch or jump instruction

- a load from memory before the value is available

- a double precision move operation

- a co-processor control operation

and a two-cycle delay following:

- a move from the `lo` or `hi` register.

It is possible, without knowing the semantics of a program, to use value tracking to determine when an instruction will modify the source of a subsequent instruction. The MIPS assembler reorganizer uses this information to move instructions forward in the execution order to fill in the delay slots required by the pipeline (see section 3.1 for details). This eliminates a large number of delay slots that would otherwise have to be filled with `nop` instructions. However, it is our contention that a reorganizer belongs in the compiler, not in the assembler.

Clearly, the assembler must verify that the pipeline constraints are satisfied. However, the MIPS assembler also translates the high-level instructions into the MIPS M/500 native machine-level instructions, sometimes expanding simple instructions into a sequence of instructions. While this makes it easier to write code in assembly language by hand, it has deleterious effects on compilers. We therefore assert that the proper place for a reorganizer is in the compiler, and *not* in a post-processing assembler.

To support our claim, we cite the following seven issues (which will be explained in greater detail in later sections):

1. The code generator knows a lot more about aliasing[49] than the assembler. Although it is difficult to detect aliasing in a compiler, it is even more difficult to detect it in the language-context free environment that is presented to the assembler. Since a reorganizer must consider aliasing effects, it is better to put a reorganizer in the compiler.

2. The compiler understands about the alignment of variables. It can know when it is not necessary to reload the top 16 bits of an address[50] by ensuring that the top 16 bits are the same for two variable components (i.e., a FORTRAN `complex` type). The assembler could make similar deductions from carefully placed `.align` directives, but

_____

[49]Aliasing is the condition under which two address expressions reference the same memory location.

[50]The MIPS M/500 can only store the low 16 bits of an address in an instruction. If a 32 bit address must be generated, it must be done in two instructions.

seems not to do so – and anyway, the compilers do not generate `.align` directives other than for word alignment.

3. Since the assembler reorganizer may need to perform some intermediate calculations in the MIPS M/500 native instruction set to implement the high-level instructions that are given to it, the assembler must reserve a temporary register for this purpose (i.e., `at`). This leaves one less register for the compiler to use, and often results in the needless recalculation or reloading of temporary values that a compiler could store in one of its registers.

4. The assembler assumes only a single base register (i.e., `gp`). However, it is often much more efficient to allow the compiler to allocate multiple base registers – for example, one for a given routine, or one for read-only data, etc.

5. With a knowledge of the reorganization requirements of the hardware, a compiler can make intelligent decisions about delaying arithmetic calculations. The assembler reorganizer must be very pessimistic about moving arithmetic instructions forward or backward, for fear of affecting numeric results. With the expression semantics available to it, the compiler is much more able to move instructions to avoid `nop` delays.

6. The assembler cannot easily reverse the effects of code hoisting (either α-motion or ω-motion). In this case, compiler optimization effectively reduces the strength of the final assembly code.

7. Since the MIPS assembler is, in effect, a macro-assembler, the final peephole optimization performed by the compilers is defeated by the macro expansion performed by the assembler. The assembler reorganizer must then supplement the compilers' optimization with a peephole optimization of its own, but this is less efficient than doing all optimization in the compiler.

We will now present a number of simple examples that demonstrate the problems cited above. These examples are all somewhat contrived, and are designed to illustrate the problem in as small a space as possible. Thus, the code fragments themselves may look somewhat unreasonable. The reader is assured, however, that real-life examples that trigger these same symptoms exist in profusion.

7.1. Alignment Problems in the Reorganizer

On the MIPS M/500, all addresses are stored as 32-bit quantities. However, an instruction that references a global variable must first load the upper 16 bits of the address of the variable with an `lui` instruction, followed by an instruction that references the low 16-bits of the address. Very often, the assembler cannot know the alignment of two variables relative to each other. Consequently, it must load the upper 16-bits of the address of each global variable *each time* it references one of them (since it is unable to determine whether the variables are in the same 16-bit address space – a fact which may change between assembly and link time, especially if the variables are declared in different modules).

When the components of a variable that is larger than a single word (i.e., a FORTRAN **complex** variable, or a C structure), the compiler can align the variable on a known boundary in such a way that it is guaranteed that the upper 16 bits of the address of all of the components of the variable are the same. Then the upper 16 bits need only be loaded once for a sequence of accesses.

```
            .data
            .align 3         # align on 2**3 byte boundary
cmplx:      .word 0,0
            .text
align:
            lb $2,cmplx
            lb $3,cmplx+1
            lh $4,cmplx+2
            lw $5,cmplx+4
```

Figure 7-1: Alignment Problem - Assembler Source Code

Examine figure 7-1. Note that the variable `cmplx` is a double-word quantity aligned on a double-word boundary. The top 16 bits should be the same for *all* of the above operand addresses (`cmplx`, `cmplx+1`, etc.). Even if the assembler/reorganizer cannot recognize the alignment of the two words comprising the double word, at least the first three instructions can share one load of `$at`.

```
            align:
0x0:        3c010000         lui     at,0x0
0x4:        80220028         lb      v0,40(at)
0x8:        3c010000         lui     at,0x0
0xc:        80230029         lb      v1,41(at)
0x10:       3c010000         lui     at,0x0
0x14:       8424002a         lh      a0,42(at)
0x18:       3c010000         lui     at,0x0
0x1c:       8c25002c         lw      a1,44(at)
0x20:       00000000         nop
```

Figure 7-2: Alignment Problem - MIPS M/500 Code

As shown in figure 7-2, the register `$at` is loaded afresh for every operand, quite needlessly.[51] The excuse that the individual loads might reference words that are in different 64Kb segments is fallacious, since the `.align` directive ensures that this is not the case. A compiler would be aware of the alignment of every object, while the assembler reorganizer is not. For unaligned objects, a compiler could load the true start address into a base register. For aligned objects (such as that shown in figure 7-1), it could load the top 16 bits into a base register **once**, and not reload it each time. In this example, the code could be reduced to a little more than half of its original size.

7.2. Problems with Aliasing

The MIPS M/500 assembler reorganizer knows nothing about the sources or targets of load and store operations. Thus, when it is dealing with registers that are pointing to data (i.e., based address mode), it must assume that the registers are aliased – that is, it must assume that since two registers *may* contain the same value, they *may* point at the same data item. Therefore, the assembler reorganizer *must* avoid reorganizing around load/stores that involve based address mode.[52]

---

[51]The `lui` will not necessarily load the value 0. Instead, the linker will fill in this value at link time with the correct base address.

[52]In fact, for addresses that are declared external, the assembler/reorganizer must not reorganize around load/stores that involve relocatable or absolute addresses. This is because it has no guarantee that the two addresses will not be the same at link time (i.e., two labels referring to the same data location).

```
#           *ptr1 = *ptr2;
        lw      $12, 0($8)
        sw      $12, 0($9)
#           *ptr3 = *ptr4;
        lw      $13, 0($10)
        sw      $13, 0($11)
```

Figure 7-3: Aliasing Problem - Assembly Source

When faced with the problem of generating code for a copy from one set of pointers to another, a compiler might generate the code shown in figure 7-3. The code is straightforward and concise – the variables are loaded using based address mode and stored the same way. However, consider the MIPS M/500 code that the assembler reorganizer generates.

```
0x0:    8dcf0000        lw      t4,0(t0)
0x4:    00000000        nop
0x8:    af0f0000        sw      t4,0(t1)
0xc:    8f280000        lw      t5,0(t2)
0x10:   00000000        nop
0x14:   ad280000        sw      t5,0(t3)
```

Figure 7-4: Aliasing Problem - MIPS M/500 Code

As shown in figure 7-4, the MIPS M/500 code that is generated contains a `nop` instruction between each load and store. This satisfies the pipeline delay that is required before the values become valid in the registers. If the compiler is given the true instruction set of the machine to operate with, instead of a high-level assembly language, these `nop` instructions can be avoided.

The assembler reorganizer is not filling in these `nop` slots because it cannot tell whether the target of the store operation is the same as (i.e., if it is aliased to) the source of the second load. A compiler could determine whether aliasing was a concern, and if it determined that it was not, it could rewrite the code as in figure 7-5.

```
0x0:    8dcf0000        lw      t4,0(t0)
0x4:    8f280000        lw      t5,0(t2)
0x8:    af0f0000        sw      t4,0(t1)
0xc:    ad280000        sw      t5,0(t3)
```

Figure 7-5: Aliasing Problem Corrected

Notice that the delay slots have been filled by reorganizing the code. Since the first store does not affect the second load, that load may be moved in front of the second store. Since the assembler/reorganizer is unaware of the presence or absence of any aliasing, it is unable to perform this function – a strong argument in favor of putting the reorganizer function in the compiler, which can do much stronger analysis of aliasing. For example, in a strongly typed language, two pointers with different base types cannot be aliases. A compiler would know this, since it knows the types; the assembler cannot know this, since it has no type information.

We would like to note that the reorganizer is pretty good about moving code that is unaffected by aliasing. For example, had a set or arithmetic operations followed the load/stores, using different registers as sources and destinations, the assembler reorganizer would have moved them upward to fill in the delay slots. There are numerous cases, however, where this sort of action will be precluded.

7.3. Delaying Calculations to Avoid No-Ops

Because the MIPS M/500 assembler reorganizer cannot know the code generators intent when scanning a piece of assembly code, it be must very pessimistic about reorganizing code. Even when it knows all of the "come-from" locations, it will not reorganize around a label. We assert that a compiler, armed with the semantics of the source language (and thus mindful of the programmer's intentions) can, with much greater confidence, rearrange the assembly language that it produces.

```
int     g1,g2,g3;
int     h1,h2,h3;

delay()
{

    g1 = g2 + g3;
    h1 = h2 + h3;
}
```

Figure 7-6: Example of Assembly Rearrangement - C Source

Figure 7-6 shows a simple example of a routine that adds two pairs of global variables and places the results in a third pair. The assembly language that is generated (figure 7-7) is perfectly reasonable – the values `g2` and `g3` are loaded into memory, added together, and stored in `g1`. Then the values `h2` and `h3` are loaded into memory, added together, and stored in `h1`.

```
delay:
 #    7       g1 = g2 + g3;
         lw       $14, g2
         lw       $15, g3
         addu     $24, $14, $15
         sw       $24, g1
 #    8       h1 = h2  sf+ h3;
         lw       $25, h2
         lw       $8, h3
         adduu    $9, $25, $8
         sw       $9, h1
```

Figure 7-7: Example of Assembly Rearrangement - Assembly Output

For a machine that is not pipelined, this is perfectly reasonable behavior on behalf of the compiler. However, recalling the pipeline restrictions, the assembler must provide a one-cycle delay following each load (the `lw` instructions) before the value in the register becomes valid. Thus, the delay slot of the first load is filled with the second load, but the delay slot for the second load must be observed before the `addu` instructions can be executed.

As shown in figure 7-8, the assembler reorganizer is unable to move any instructions downward to fill either of these delay slots. Examining either the assembly code or the MIPS M/500 code, however, shows that the code can be reorganized in a better way. Instructions can be moved forward and *backward* to fill in the delay slots. Figure 7-9 shows this hand-optimized reorganization.

Notice that the load of `h2` has been moved backward to fill the delay slot required after the load of `g3`. The store to `g1` has been moved forward to fill in the delay slot after the load of `h3`. The net result is that, while the code in figure 7-9 performs exactly the same function as the code in figure

```
        delay:
0x0:    8f8e0000        lw      t6,0(gp)
0x4:    8f8f0000        lw      t7,0(gp)
0x8:    00000000        nop
0xc:    01cfc021        addu    t8,t6,t7
0x10:   af980000        sw      t8,0(gp)
0x14:   8f990000        lw      t9,0(gp)
0x18:   8f880000        lw      t0,0(gp)
0x1c:   00000000        nop
0x20:   03284821        addu    t1,t9,t0
0x24:   af890000        sw      t1,0(gp)
```

Figure 7-8: Example of Assembly Rearrangement - MIPS M/500 Code

```
        delay:
0x0:    8f8e0000        lw      t6,0(gp)
0x4:    8f8f0000        lw      t7,0(gp)
0x8:    8f990000        lw      t9,0(gp)
0xc:    01cfc021        addu    t8,t6,t7
0x10:   8f880000        lw      t0,0(gp)
0x14:   af980000        sw      t8,0(gp)
0x18:   03284821        addu    t1,t9,t0
0x1c:   af890000        sw      t1,0(gp)
```

Figure 7-9: Example of Assembly Rearrangement - Optimized MIPS M/500 Code

7-8, it is 20% smaller. We do not claim that a 20% increase in speed can be obtained with this optimization technique. However, as was explained in section 6.3.1.1, over 14% of the code generated by the MIPS C compiler were `nop` instructions – a figure which could be substantially reduced with this and other optimizations.[53]

7.4. Macro Expansion Defeating Peephole Optimization

It was often observed in the Berkeley compilers that the so-called optimization phase was not a true optimizer, but rather a neatener. This is basically all that a peephole optimizer is able to do – neaten the generated code somewhat. The MIPS M/500 assembler reorganizer suffers from this same problem. After the compiler has done a good job of optimizing for the MIPS virtual machine,[54] the assembler reorganizer expands each of these instructions into the corresponding MIPS M/500 in-structions, effectively messing up the optimization. The peephole optimizer in the assembler reor-ganizer can then only "neaten up" after it has rumpled the previously elegant code.

Consider that a compiler has the conditional expression

    ((a <= b) and (c > 5)) or ((a > b) and (c == 0))

for which to generate code. It would certainly be reasonable for the compiler to calculate a <= b and negate the result (and thus have the result of both a <= b and a > b). One reasonable way of doing this on the MIPS would be as shown in figure 7-10.

---

[53]A random sampling of `nop` instructions (shown in section 6.2.2, page 55) found that over 70% of the `nops` in a given system application could be eliminated. There is probably room, therefore, for approximately another 10% increase in speed in program execution by performing better `nop` elimination.

[54]The assembly language that is available to the user and to the compilers is not the actual machine language used by the MIPS M/500. The assembler reorganizer translates high-level instructions into low-level MIPS M/500 machine instructions.

```
sle       $8,$4,$5
not       $9,$8
```

Figure 7-10: Assembler Reorganizer Defeating Optimization - Assembler Source

When this code is presented to the assembler reorganizer, the code that it generates is changed somewhat, as in figure 7-11. Instead of the `sle` that the compiler requested, the assembler/reorganizer has changed it into an `slt` (with reversed operands), followed by an `xori`. This is then followed by the compiler-requested `not`.

```
0x0:      00a4402a        slt     t0,a1,a0
0x4:      39080001        xori    t0,t0,0x1
0x8:      01004827        nor     t1,t0,zero
```

Figure 7-11: Assembler Reorganizer Defeating Optimization - MIPS M/500 Code

Since the `slt` instruction just sets the lower bit of `t0`, the exclusive xor with the constant 1 is a complementation (i.e., a `not`), which is immediately complemented again by the `nor` instruction. What results is that the compiler has generated what it believes to be good code, but the final effect of the assembler reorganizer is to generate poor code, because the macro expansion *follows* the low-level optimization.

7.5. Drawbacks of Reserving a Temporary Register for the Assembler

Because the assembler reorganizer must rewrite the code that is given to it by the compiler, it often must add instructions into the assembly stream to overcome the shortcomings of the MIPS M/500 native instruction set. Very often it needs to use a temporary register to hold some intermediate values. This register is `at`, and is reserved by the assembler reorganizer for its own use.

We assert that this use of a register unavailable to the compiler is a mistake for at least two reasons:

1. The compiler is denied the use of this register, and so has fewer registers to allocate. Although this is a minor point with 25 other registers to use,[55] it does reduce the efficiency of the machine somewhat.

2. There are times when it is more efficient to store two temporary values, but the assembler is constrained to building work-arounds.

We feel the latter reason is the more important, and we demonstrate our reasons in the following example. Consider the source code shown in figure 7-12. All that the code is doing is incrementing two (global) variables by 1.

```
x := x + 1;
y := y + 1;
```

Figure 7-12: Temporary Register Problem - High Level Source

A compiler that is somewhat aware of the reorganization requirements of the target machine might

---

[55]Although the MIPS M/500 has 32 registers, 7 are reserved. These are the zero register, `at`, `k0`, `k1`, `gp`, `sp`, and `ra`.

generate code of the form shown in figure 7-13. The code is interleaving the loads and adds to avoid the delay following a load from memory required by the pipeline.

```
            .data
            .align 2
x:          .word 0
y:          .word 0

            .text
tempreg:
            lw        $2,x
            lw        $3,y
            add       $2,$2,1
            add       $3,$3,1
            sw        $2,x
            sw        $3,y
```

Figure 7-13: Temporary Register Problem - Assembly Code

As shown in figure 7-14, the assembler reorganizer takes the interleaved code and messes it up somewhat. Because x and y are not directly addressable, the assembler reorganizer must build the addresses of each 16 bits at a time. Because of the interleaving generated by the compiler, it cannot use one register for both x and y. But because it has only one register available to it (i.e at), it must load and reload that one register.

```
            tempreg:
0x0:        3c010000          lui       at,0x0
0x4:        8c220028          lw        v0,40(at)
0x8:        3c010000          lui       at,0x0
0xc:        8c23002c          lw        v1,44(at)
0x10:       20420001          addi      v0,v0,1
0x14:       20630001          addi      v1,v1,1
0x18:       3c010000          lui       at,0x0
0x1c:       ac220028          sw        v0,40(at)
0x20:       3c010000          lui       at,0x0
0x24:       ac23002c          sw        v1,44(at)
```

Figure 7-14: Temporary Register Problem - MIPS M/500 Code

A code generator could use two temporary registers, one each for the top 16 bits of the address of x and y, and so save the second two lui instructions.[56] Once again, macro expansion is inhibiting or defeating other optimizations.

7.6. Shortcomings of Using a Single Global Pointer

In the current implementation, the MIPS compilers load global registers using relocatable or indexed-relocatable address modes. On the MIPS M/500, this is translated by the assembler reorganizer to a sequence of instructions that always performs a lui instruction. This is required, since the linker may have relocated the target address so that the upper 16 bits are significant.

---

[56]Note that the assembler reorganizer could certainly save one lui by reversing the order of the stores. This is safe, since it is evident that x and y do not overlap (i.e., there is no aliasing problem to be reckoned with here, so the ordering can be altered).

To circumvent this problem somewhat, the assembler reorganizer provides two data segments in addition to the UNIX standard of `.data` and `.bss` segments.[57] These are the `.sdata` and `.sbss` segments, which are equivalent to the `.data` and `.bss` segments, respectively, except that they are addressed via the global pointer `gp`.

The `gp` register is loaded by the program prelude, and the initial value is specified by the linker. The problem with this scheme is that it limits the compilers somewhat. It would be better to allow the compilers to make intelligent decisions on register allocation based on variable usage rather than restricting them by the requirements of the assembler. Two specific examples of how compiler performance could be increased are:

- In FORTRAN, the compiler could allocate a global pointer to point at the beginning of a common block. Currently, the compiler must always use relocatable address expressions, which require two MIPS M/500 instructions to fetch an address. Using based address mode (with the compiler allocated register) requires only one native instruction.

- In C, array accesses are performed using the indexed relocatable address mode, which requires three MIPS M/500 instructions. Array accesses could be simplified into two native instructions by allocating a base register at compile time for those arrays which are accessed heavily in a routine.

The problem with these optimizations is that currently, they are "difficult." The compiler views as its target architecture the MIPS pseudo-machine, when in fact it should be generating code for the MIPS M/500 native machine. On the pseudo-machine, based address mode is no more complicated than relocatable mode (whereas on the real machine, they are quite different). For this and other optimizations to be feasible, the assembler reorganizer should be eliminated (or at least simplified), and the compilers should target the native MIPS M/500, not the MIPS pseudo-machine.

7.7. Arithmetic Optimizations on Native Hardware

To save execution time, the assembler reorganizer will substitute a multiply with a sequence of shifts and adds whenever possible (see section 3.2.1). This "optimization," however, has a strictly peephole effect in that it can sometimes cause a program to run slower overall.

```
extern w,x,y,z;

mult()
{
        x = 465*y + 1890*z;
}
```

Figure 7-15: Optimistic Approach to Multiplication - C Source

Consider the source code fragment shown in figure 7-15. In this simplistic example, a variable is loaded with the sum of two products. Since the compiler only knows about the instruction set of the MIPS pseudo-machine, it generates the instruction sequence shown in figure 7-16.

---

[57]The `.bss` segment is for uninitialized data, which, under UNIX, defaults to being initialized to zero. The `.data` segment is for all explicitly initialized data.

```
#    5            x = 465*y + 1890*z;
        lw      $14, y
        mul     $15, $14, 465
        lw      $24, z
        mul     $25, $24, 1890
        addu    $8, $15, $25
        sw      $8, x
```

Figure 7-16: Optimistic Approach to Multiplication - Assembler Source

This is an entirely reasonable thing for the compiler to do, since it has been told that a multiply is a single instruction. As shown in section 3.2.1, however, a single multiply can be expanded to a large sequence of shifts and adds. In this case, both multiplications are by constant values, so this is exactly what happens. As shown in figure 7-17, the first multiply is translated into 6 instructions, and the second into 7 instructions.

```
0x0:    8f8e0000        lw      t6,0(gp)
0x4:    8f980000        lw      t8,0(gp)
0x8:    000e78c0        sll     t7,t6,3
0xc:    01ee7823        subu    t7,t7,t6
0x10:   000f7880        sll     t7,t7,2
0x14:   01ee7821        addu    t7,t7,t6
0x18:   000f7900        sll     t7,t7,4
0x1c:   01ee7821        addu    t7,t7,t6
0x20:   0018c900        sll     t9,t8,4
0x24:   0338c823        subu    t9,t9,t8
0x28:   0019c880        sll     t9,t9,2
0x2c:   0338c823        subu    t9,t9,t8
0x30:   0019c900        sll     t9,t9,4
0x34:   0338c821        addu    t9,t9,t8
0x38:   0019c840        sll     t9,t9,1
0x3c:   01f94021        addu    t0,t7,t9
0x40:   03e00008        jr      ra
0x44:   af880000        sw      t0,0(gp)
```

Figure 7-17: Optimistic Approach to Multiplication - MIPS M/500 Code

Since the assembler reorganizer is trying to discourage the use of the actual MIPS M/500 multiply instruction, it has taken efficient code and translated it into code that is far less efficient than it could be. The algorithm to convert a multiply into shifts and adds is quite simple, and could be placed in the compiler instead of the assembler reorganizer. The extra information needed to make this a worthwhile investment (i.e., the semantics of the arithmetic operations and their interactions with other variables) also resides in the compiler.

```
#    5            x = 31*(15*y + 63*z);
        lw      $14, y
        mul     $15, $14, 15
        lw      $24, z
        mul     $25, $24, 63
        addu    $8, $15, $25
        mul     $9, $8, 31
        sw      $9, x
```

Figure 7-18: A Better Approach to Multiplication - Assembler Source

Since the compiler knows the semantics of the expression, it can calculate the least common denominators of the multiplicands and rewrite the expression into what at first appears to be a less

optimal form, as shown in figure 7-18. This form includes not two, but *three*, multiplications, which seems to be a worse implementation. However, when this code is fed to the assembler reorganizer (which will convert the multiplications by constant values to shifts and adds), we get what is shown in figure 7-19.

```
0x0:     8f8e0000          lw      t6,0(gp)
0x4:     8f980000          lw      t8,0(gp)
0x8:     000e7900          sll     t7,t6,4
0xc:     01ee7823          subu    t7,t7,t6
0x10:    0018c980          sll     t9,t8,6
0x14:    0338c823          subu    t9,t9,t8
0x18:    01f94021          addu    t0,t7,t9
0x1c:    00084900          sll     t1,t0,4
0x20:    01284823          subu    t1,t1,t0
0x24:    03e00008          jr      ra
0x28:    af890000          sw      t1,0(gp)
```

Figure 7-19: A Better Approach to Multiplication - MIPS M/500 Code

In this case, the multiplication by 15 is translated into 2 instructions, the multiplication by 63 into 2 instructions, and the multiplication by 31 into 2 instructions. The net result is that, by writing a more "pessimistic" assembly source, we can reduce the actual instruction count of the arithmetic from 14 instructions to 7 – a reduction of 50%.

While the results may not always be this spectacular, if the compiler were armed with knowledge of the real MIPS M/500 assembly language, instead of relying on the assembler reorganizer to translate from the MIPS pseudo instruction set, the compiler could generate more efficient code. In general, reducing arithmetic expressions to their simplest factored form can allow the compiler to generate tighter code. For most architectures, this is a pessimization, not an optimization. Due to the expense of the multiply instruction, however, reducing the number of true multiplies by increasing the number of shifts and adds pays off.

8. Validation of MIPS Pascal Compiler

This chapter describes the results of the Pascal validation suite[58] as applied to the MIPS M/500. The validation suite tests the Pascal compiler against the BS 6192:1982 *"Specification for Computer Programming Language Pascal"*[59] and reports any discrepancies. In this chapter, we list those discrepancies, along with our evaluation of the ramifications. The discrepancies are listed under four categories: portability, conformance, incorrectly generated code, and extensions. In all cases, the section number listed to in the discrepancy reports reference the section number in BS 6192:1982. Please note that this chapter refers only to those failures which the validation set was able to discover; it does not report on those tests which passed correctly. It should also be noted that the MIPS Pascal compiler is a level 0 implementation of Pascal, which is to say that it does *not* support conformant arrays. According to the Standard, this is an acceptable reduction in compiler strength, although we feel that conformant array support is still desirable.

According to MIPS Inc., their Pascal compiler is an implementation of ANSI standard Pascal (ASNI/IEEE 770X3.97-1983). As such, there will be slight differences between it and the BS 6192:1982 Pascal. The differences, however, are far fewer in number than we found as discrepancies in the following sections.

8.1. Portability

This section lists those features under which the MIPS Pascal compiler deviates from the standard in a way that may affect program portability. Generally, these deviations are expressed as extensions to the language.

Section	Symptom and Comments
`6.1.2-2`	The unrestricted words `otherwise`, `return`, `separate`, `subtype`, `double`, and `cobol` are reserved words in MIPS Pascal. - In the case of `double`, `return` and `otherwise`, MIPS Pascal is providing what we feel to be needed extra functionality to the language. The other additional reserved words serve other functions. In any event, this is a legal language extension, provided it is documented.
`6.1.6-5`	The MIPS Pascal compiler allows labels to exceed the range of 1..9999. - The Pascal standard states that labels must be restricted to the range of 1..9999. By allowing labels to exceed that limitation, programs developed on the MIPS Pascal compiler may have portability problems. In practice, however, it is unlikely that a programmer would use a sufficient number of labels to make a simple translation unfeasible.

---

[58]The Pascal validation suite was obtained from Software Consulting Services, 3162 Bath Pike, Nazareth, PA 18046. All test programs from the validation suite are copyrighted by A. H. J. Sale and the British Standards Institution, 1982.

[59]Also known as ISO 7185, and available from the British Standards Institute, 2 Park Street, London W1A 2BS, England.

Section	Symptom and Comments
`6.1.6-6`	The MIPS Pascal compiler allows labels to contain alphabetic characters.
	- -
	The Pascal standard states that labels must be numeric. By allowing a label to contain alphabetic characters, the MIPS Pascal compiler presents a possible portability problem.

Section	Symptom and Comments

`6.1.7-5`	The MIPS Pascal compiler allows string variables to be stored in ordinary (unpacked) arrays. - The Pascal standard specifically states that character strings are of type `packed array[1..n] of char`. By allowing unpacked arrays to hold strings, the MIPS Pascal compiler presents a possible portability problem. In general, however, this is a rather simple addition to the language, and can be worked around easily enough.
`6.1.7-11`	The MIPS Pascal compiler allows for the null string. - The Pascal standard states that a character string is a sequence of characters surrounded by apostrophes – hence there can be no null string. Although this introduces a portability problem, we do not feel that it presents any real issue.
`6.1.8-5`	The MIPS Pascal compiler allows the expression `    i := 10div j;` to pass without error. - The expression (notice the missing space character) is clearly unambiguous, even though it is in violation of the standard. We do not expect this deviation to be of any consequence.
`6.2.1-8` `6.2.1-9` `6.2.1-10`	The MIPS Pascal compiler allows for declarations outside of the standard-specified order, and for multiple declarations of any given type. - The Standard requires that declarations be in the order: `    1. label` `    2. type` `    3. const` `    4. var` `    5. procedure/function` Since many Pascal compilers allow these deviations, we feel that this is of little consequence.
`6.2.2-8`	This program fragment compiles successfully, even though it is in violation of the Standard: ``` const red = 1; violet = 2; procedure ouch; const m = red; n = violet; type a = array[m..n] of integer; var v : a; color : (blue,red,indigo,violet); begin v[1]:=1; color:=red end; ``` - The Pascal Standard requires that the defining-point of an identifier shall precede all applied occurrences of that identifier, with the exception of pointer-type declarations. The scope of an identifier is its whole region, which, in most cases, is a block. The rules prohibit a reference to

Section	Symptom and Comments
(continued)	an outer identifier of the same spelling preceding the defining-point. The test includes two exactly similar violations of the rules in the use of the identifiers **red** and **violet** in the declarations of **m** and **n**. The MIPS Pascal compiler is treating the declarations in a top-down manner, instead of considering them in a block-oriented manner. This particular error is *very* hard for a 1-pass front end to get right.
| **6.2.2-12** | The MIPS Pascal compiler allows an applied occurrence of a type to be in the same scope as a field designator of the same name:<br><br>```
type
    rec = record
               ptr : ^fred;
               fred : integer
          end;
    fred = rec;
```<br>- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -<br>This deviation from the standard presents a significant portability problem. Should a programmer take advantage of this "feature", it could be rather difficult to undo its use when attempting to port a program written on the MIPS M/500. |
| **6.3-2**<br>**6.3-4**<br>**6.3-5**<br>**6.7.2.2-5** | The MIPS Pascal compiler allows characters and booleans to be signed, for example:<br><br>```
const
    dot = '.';
    plusdot = + dot;
```<br>or:<br><br>```
const
    truth = true;
    plustruth = + truth;
```<br>- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -<br>While it is not anticipated that a programmer would use this feature, the failure of the MIPS Pascal compiler to catch this error suggests that hidden program flaws may pass through the compiler undetected. |
| **6.3-6** | This program deviates because constants must not appear in their own definition:<br><br>```
const
    ten = 10;

procedure p;
const
    ten = ten;
begin
end;
```<br>- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -<br>The Standard explicitly forbids a constant to appear in its own definition. In this program, the definition `ten = ten` is in the scope of the second use of `ten` and, accordingly, is in error. While it is not anticipated that a programmer would use this feature, the failure of the MIPS Pascal compiler to catch this error suggests that hidden program flaws may pass through the compiler undetected. |
| **6.3-7** | The MIPS Pascal compiler allows the value `nil` to be used in the constant definition part:<br><br>```
const
    nothing = nil;
```<br>- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -<br>This deviation allows the programmer to define a synonym for `nil`. For portability purposes, this presents only a small problem, since a global textual substitution will solve the compilation problems. |

Section	Symptom and Comments
| **6.3–9** | By allowing this example to compile, the MIPS Pascal compiler deviates from the Standard since expressions cannot appear in a constant-definition:<br><br>```
const
    linelength=80;
    lineoflo=linelength+1;
```<br><br>- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -<br><br>The const-part contains definitions of identifiers in terms of simple constants. Standard Pascal does not permit expressions to be used, even if their values are compile-time determinable. The authors have opposing viewpoints on this restriction. Should it present a portability problem, however, it is easily worked around. |
| **6.4.1–3** | The MIPS Pascal compiler allows the use of a type in the same scope as its definition:<br><br>```
type
    x  = integer;

    procedure p;
    type
       x  = record
                  y : x
            end;
    begin
    end;
```<br><br>- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -<br><br>In this case, the definition of the component $y$ in the record $x$ is of type integer, although the scope of the type $x$ is the same as the declaration. Because the MIPS Pascal compiler allows this to compile, it suggests that the compiler does not place a type in the symbol table until it is fully defined. While it is not anticipated that a programmer would use this feature, the failure of the MIPS Pascal compiler to catch this error suggests that hidden program flaws may pass through the compiler undetected. It will also present a nasty portability problem if this feature is used. |
| **6.4.3.2–5** | Strings must have a subrange of integers as an index-type. The following fragment compiles without error.<br><br>```
type
    color = (red,blue,yellow,green);
    cl1 = blue..green;
var
    s: packed array[cl1] of char;
begin
    s:='ABC';
end.
```<br><br>- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -<br><br>It is incorrect to have a subrange of an enumerated-type as the index-type, even if the ord of the lower bound is one. As with other examples of this type, we feel it unlikely that a Pascal programmer will use this feature of the MIPS Pascal compiler. However, in this case we feel that by allowing this code to pass through without error, the MIPS Pascal compiler is allowing other, perhaps undetected, errors to pass through by equating some instances of sets with integers. |

Section	Symptom and Comments
6.4.3.3-18	This test deviates, since all values of a *tag-type* of a record must appear as case-constants. ``` type color=(pink,red,green,blue,yellow); colored=record case c:color of pink:(p:array [1..2] of color); red:(r:array [1..3] of color); blue,yellow:(b:array [1..5] of color); end; ``` - This deviation is another of little consequence. The requirement that all of the values of the tag-type appear as case-constants is primarily for completeness. The actual value of the tag (in this case **c**) is not used to access the variant part, so assigning **green** to **c** will not cause a range violation error on the variant part.
6.4.3.5-13	This test deviates, since the *component-type* of a *file-type* should not include a *file-type*. ``` var f1 : file of text; ``` - The Pascal predeclared entity `text` is a *file-type*. By allowing this program fragment to compile, the MIPS Pascal compiler is introducing a possible portability problem. It appears easy enough to change in the source program, however, to merit little concern.
6.4.3.5-14	This test deviates for the same reason as 6.4.3.5-13. ``` type rec = record f1 : text; f2 : file of char; end; var f3 : file of rec; ``` - In this example, the compiler is essentially allowing a `file of file of char` to be a legal type. This will generally not compile on other Pascal compilers.
6.4.5-12	The following fragment compiles without error: ``` if 'CAT' < 'HOUND' then ``` - The Pascal Standard permits compatibility only between string-types having the same number of components, while the MIPS Pascal compiler allows compatibility between different string types. This is a nice extension to the language, although finding and correcting all such instances in a program to be ported could prove to be a difficult venture.

Section	Symptom and Comments
`6.4.5-16`	This test violates the type rules for relational-operators using sets as operands.

```
type
    BType  = set of boolean;
    PType  = packed set of false..true;
var
    flag:boolean; B:BType; P:PType;
begin
    B:=[true,false];
    P:=[true];
    flag:=(B >= P); { B,P, incompatible }
```

- -

A relational-operator between values of a set type can either have compatible operands or be of the same canonical set-of-T type. In this instance, the T is not the same (one packed, the other unpacked). The MIPS Pascal compiler makes no distinction between packed and unpacked datatypes, so this is of little consequence on the MIPS machine. However, serious difficulties could arise in porting.

Section	Symptom and Comments
`6.4.6-4`	This test deviates, since assignment of reals to integers is not permitted.

```
var
    r : real;
    i : integer;

begin
    r:=6.0;
    i:=r;
end.
```

- -

The Pascal Standard allows assignment of integers to reals, but not reals to integers. To perform this latter assignment, the program writer must use the explicit built-in functions `trunc` or `round` (the MIPS Pascal compiler is performing an implicit `trunc` operation). While this feature is of little consequence on the MIPS, chasing down all instances of this feature in a program to be ported could prove harrowing.

Section	Symptom and Comments
`6.4.6-6`	The MIPS Pascal compiler allows this program to compile and execute without error:

```
type
    rekord = record
                   f : text;
                   a : integer
                end;
var
    record1 : rekord;
    record2 : rekord;
begin
    record1.a:=1;
    rewrite(record1.f);
    rewrite(record2.f);
    record2:=record1;
    writeln(' DEVIATES...6.4.6-6')
end.
```

- -

Structured-types containing a file component should not be assigned to each other. The Pascal Standard states that the two types `T1` and `T2` (in determining assignment compatibility) must not be a structured-type with a file component. This feature of the MIPS Pascal compiler seems to be a little more threatening regarding the portability issue.

Section	Symptom and Comments
6.5.4-4	This program deviates because a function-identifier cannot be used as a pointer-variable. ```pascal type ptr = ^integer; var p : ptr; function f : ptr; var p : ptr; begin new(p); f := p; f^ := 10 end; begin p := f; writeln(p^); end. ``` - The MIPS Pascal compiler takes a short-cut and treats a function-identifier as a local variable when it appears on the left-hand side of an assignment. This is illegal according to the Standard, and presents a noticeable portability problem.
6.6.1-3	This program shows that a procedure call is incorrectly bound to the wrong defining occurrence. ```pascal procedure p; begin writeln(' OUTER PROCEDURE') end; procedure q; procedure qq; begin p end; procedure p; begin writeln(' INNER PROCEDURE') end; begin qq end; begin q; end. ``` - Since the applied occurrence is before the defining occurrence (in `qq`), the program deviates. The MIPS Pascal compiler should issue a compile time error indicating that the procedure `p` is not declared at the time of its use. Instead, it uses the outer procedure, even though the scope of the inner procedure overrides it. If, within the procedure `q`, the procedure `p` is declared to be of type `forward`, the *inner* procedure is called, alluding to the linear creation of the symbol reference table within the compiler.

Section	Symptom and Comments
| 6.6.1–4 | This program shows another example of a procedure binding to the wrong occurrence:<br><br>```pascal
var
    i:integer;
procedure p;
    begin
    i := ord( 'A' )
    end;
function ord(c:char): integer;
    begin
    ord := - maxint
    end;
begin
p;
end.
```<br><br>- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -<br><br>This test uses a standard function rather than nested procedures.  We feel that it is unlikely that a programmer will redefine a built-in function in this manner.  However, the MIPS Pascal compiler should nonetheless issue an error message for this program. |
| 6.6.3.2–1 | The assignment compatibility rules prohibit a type with a file component being used as a value parameter.<br><br>```pascal
type
    f = record
        x: integer;
        y: text
        end;
var
    v: f;
procedure p( q: f);
    begin
    rewrite( q.y )
    end;
begin
    v.x := 1;
    p(v);
end.
```<br><br>- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -<br><br>Since a `file` is conceptually an area on a secondary storage medium, it cannot have a "value".  By allowing a `file` to be passed as a value parameter, the MIPS Pascal compiler introduces a severe portability problem. |
| 6.6.3.3–4 | This test deviates, since an actual variable parameter shall not denote a field which is the selector of a variant-part.<br><br>```pascal
type
    shape  = (triangle,rectangle);
    figure = record
            area   :real;
            case s :shape of
                triangle : (base,height :real);
                rectangle: (side1,side2 :real)
            end;
var
    ptr    : ^figure;
``` |

Section	Symptom and Comments
(continued)	```pascal procedure findarea(var s : shape); begin case s of triangle : ptr^.area := (ptr^.base*ptr^.height)/2; rectangle: ptr^.area := ptr^.side1*ptr^.side2 end end; begin new(ptr); ptr^.s := rectangle; ptr^.side1 := 3; ptr^.side2 := 4; findarea(ptr^.s); {illegal} if ptr^.area = 12 then writeln(' VAR PARAMETER PASSING') else writeln(' VAR PARAMETER DEVIANCE') end. ``` - This deviation opens the door to some ***major*** problems. What the MIPS Pascal compiler is allowing the user to do is the following: A variant record is used with one part of the variant in one place in the program. While this variant part is in use, the variant is passed by reference to another routine, which then has the liberty to change the selector field – without "advising" the caller of the routine. Although the MIPS Pascal compiler is getting the value of **area** right (i.e., it is 12), it is providing a major loophole in the Pascal type checking rules (effectively permitting FORTRAN equivalencing or the unconstrained **C** `union` operator in a language which forbids this type of construct).
6.6.3.3-5	This program deviates from the standard, since an actual variable parameter may not denote a component of a packed variable. ```pascal type card = packed array[1..80] of char; var image : card; function headercard(var col1 :char) : boolean; begin if col1 = 'H' then headercard := true else headercard := false end; begin image[1] := ' '; if headercard(image[1]) then writeln(' VAR PARAMETER PASSING(1)') else writeln(' VAR PARAMETER PASSING(2)') end. ``` - The MIPS Pascal compiler considers packed and unpacked arrays and records to be equivalent, thus, for the MIPS, this deviation from the standard is of little consequence. However, for portability's sake, this feature should be changed.

Section	Symptom and Comments
`6.6.3.6-10`	The MIPS Pascal compiler does not adhere to standard parameter list congruity rules: ```pascal var aa,bb : integer; procedure p(procedure formal(var a,b : integer)); begin formal(aa,bb) end; procedure actual(var a : integer; var b : integer); begin writeln(' DEVIATES') end; begin p(actual) end. ``` - This example merely points out a simple extension to Pascal (and thus, a small portability problem), since the declaration parts of **formal** and **actual** are essentially identical.
`6.7.1-10`	Although the compiler should generate errors for each of the three string assignments, it generates errors only for the last two: ```pascal var string1 : packed array[1..4] of char; string2 : packed array[1..6] of char; begin string1:='AB'; string2:=string1; string1:='ABCDEFG'; end. ``` - The Pascal Standard states that string types are compatible only if they have the same number of components. The MIPS Pascal compiler is allowing assignment of one string type to another, padding out with spaces if they are not of the same length, when the source of the string assignment is a string constant. While this is felt to be a reasonable action, it may pose portability problems.
`6.7.2.5-6`	The MIPS Pascal compiler allows assignments and comparisons on records and arrays: ```pascal var c,d : record f1 : integer; f2 : real end; begin c.f1 := 0; c.f2 := 3.1; if (c <> d) then c := d; end. ``` - This is a rather nice extension to the Pascal Standard, which, unfortunately will cause some big headaches in porting. The comparisons are implemented on a component-by-component basis, as are the assignments (i.e., they are done correctly). However, although this shortcut is nice to have, it will prove annoying to anyone porting a program originally written under the MIPS Pascal compiler.

Section	Symptom and Comments
6.8.1-1 6.8.1-2	The MIPS Pascal compiler allows `gotos` between alternative arms of a conditional statement and case statements: ``` i:=5; if (i<10) then goto 1 else 1:write(' DEVIATES...6.8.1-1,'); if (i>10) then 2:writeln(' GOTO ALTERNATE BRANCH OF IF') else goto 2 ``` - A conditional (or case) statement is considered a compound statement by the standard. A `goto` may only reference a simple statement and may not reference a part of a compound. One of the reasons for this restriction is to prohibit code that skips over loop initialization code (see 6.8.1-4 below) or block initialization code (see 6.8.1-7 in section 8.3). In general, the MIPS Pascal compiler is implementing the semantics of C in allowing this feature. Programs that utilize this feature will be unportable or may produce unpredictable results on other compilers.
6.8.1-4 6.8.1-5	The MIPS Pascal compiler allows a `goto` in the middle of a `for` loop: ``` j := 0; for i := 1 to 0 do begin 100: writeln('OOPS') end; i := 0; if j = 0 then goto 100 ``` - This feature is just asking for trouble in that it allows the initialization code of a loop to be skipped. A "clever" programmer could use this feature to advantage but would be violating the Pascal standard. It is interesting to note that, if the `goto` is coded as in the example, the string "OOPS" is printed. If, however, the `goto` is coded as a non-local goto, no message is printed. We feel that this particular feature is a dangerous one to include in a production language – especially when it is disallowed by the Pascal Standard. In general, the MIPS Pascal compiler is implementing the semantics of C in allowing this feature. Programs that use this feature will be unportable or may produce unpredictable results on other compilers.
6.8.3.5-7	Subrange lists are allowed in case elements: ``` case foo of 1..4: writeln('low'); 5: writeln('high') end; ``` - According to the Standard, only lists of case elements (i.e., `1,2,3,4`) are allowed in case elements, and not subranges (i.e., `1..4`). This is a simple extension to the language and should not present too much of a portability problem. Difficulties will arise when a range that includes elements of a set is used, since it is not as obvious a list as integers.

Section	*Symptom and Comments*
`6.8.3.9-6` `6.8.3.9-7` `6.8.3.9-8` `6.8.3.9-9` `6.8.3.9-10` `6.8.3.9-15` `6.8.3.9-16` `6.8.3.9-21` `6.8.3.9-22` `6.8.3.9-24`	The MIPS Pascal compiler allows a loop control variable to be passed as a `var` parameter: ``` var i:integer; procedure verynasty (var n:integer); begin end; begin for i:=1 to 10 do begin verynasty(i) end; writeln('OOPS') end. ``` - In this example, the procedure **verynasty** can change the value of the loop control variable. This threat is prohibited by the Standard, and by allowing it, the MIPS Pascal compiler introduces a nasty portability (and debugging) feature. Other threats that the compiler allows to pass through undetected are: • Using a non-local variable as a loop control variable. • Using a global variable as a loop control variable. • Using a local variable for loop control, but permitting its use in another local procedure. • Modifying the loop control variable with a `read` statement. • Using an actual-value parameter as a loop control variable. • Using the value of the loop control variable after loop execution has completed. • Allowing the value of the loop control variable to extend past the legal subrange of the variable. As in other cases, this implementation follows the unconstrained semantics seen in `C`, and should be changed.
`6.9.1-11` `6.9.3.1-6` `6.9.3.6-3`	The MIPS Pascal compiler allows values of type other than integer, real, and character to be read and written from/to a `text` file: ``` var one:boolean; f1 :text; begin rewrite(f1); one := true; writeln(f1,one); reset(f1); read(f1,one); end. ``` - Although this is a clear deviation from the standard, other than creating a portability problem, we feel that this extension is a valid one. Since the MIPS Pascal compiler considers packed and unpacked arrays of characters to be equivalent, it also allows reads/writes of packed arrays. This, too, is valid extension.

Section	*Symptom and Comments*
6.10-1 **6.10-7**	In the program specification, declaring `output` is not required. Also, a file may be a program parameter but not be declared.
	- -
	In the former case, the MIPS Pascal compiler is adhering to the standard but deviating from Jensen and Wirth. In the latter, the type of the variable may be inferred. In both cases, we feel this is of small consequence.

8.2. Conformance

This section lists those features under which the MIPS Pascal compiler deviates from the standard in a way that may affect program compilation. Generally, these deviations are expressed as failures of the language to meet certain minimum requirements.

Section	*Symptom and Comments*
6.1.5-2	A program with a very large floating-point number (i.e., an integer part with 3 digits followed by a 35 digit fraction) causes the compiler to issue a fatal error in *ugen*. `    const` `        reel = 123.45678901234567890123456789;` - The compiler should allow an arbitrary length floating-point number to be expressed in Pascal. Whether this value can be accurately represented in an internal form is irrelevant – the compiler must accept the number as input.
| **6.2.3.5-1**<br>**6.4.3.3-11** | The MIPS Pascal compiler does not detect the use of an uninitialized variable:<br><br>```
procedure q;
var
    i,j : integer;
begin
    i:=2;
    j:=3
end;

procedure r;
var
    i,j : integer;
begin
    j := i-4;
    writeln(' THE VALUE OF I IS ', i)
end;

begin
    q;
    r
end.
```<br><br>- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -<br><br>The value printed out for **i** is 0, which happens to be the value that was in the register allocated for **i** when the program was compiled. The same kind of unpredictable behavior occurs when an uninitialized portion of a variant record is used. The compiler should report the use of a variable before it is initialized (as is done with *lint* for the **C** compiler). Instead, no indication is given. We feel that this is a shortcoming of the compiler. |

Section	Symptom and Comments
`6.4.2.4-5`	Using strings in a subrange declaration crashes the compiler with the error *"Fatal error"* and no line number indication. ``` firstindex = 'AB' .. 'CD'; ``` - While this program fragment is illegal, the ungraceful error handling of the compiler is un-acceptable. At least a specific error message should be printed. However, the compiler simply dumps core and terminates execution.
`6.4.3.3-10`	The MIPS Pascal compiler does not generate an error when accessing a field of an inactive variant: ``` type two = (a,b); var variant : record case tagfield:two of a: (m:integer); b: (n:integer) end; i : integer; begin variant.tagfield:=a; variant.m:=1; i:=variant.n; {illegal} end. ``` - This deviation is another of little consequence. The requirement that all of the values of the tag-type match the access type is primarily for completeness. The actual value of the tag (in this case **a**) is not used to access the variant part, so accessing **n** while the variant part is set to **c** should not cause any problems (even though, technically, an error message should be printed).

`6.4.4-4`	This program, which tests that the *domain type* of a *pointer type* may be a *file type*, generates a segmentation fault:
	
	``
	type
	    fileptr = ^text;
	var
	    ptr1,ptr2,ptr4 : fileptr;
	
	procedure copyandadd(var fromfile,tofile:text; ch:char);
	    begin
	    while not eoln(fromfile) do
	        begin
	        write(tofile,fromfile^);
	        get(fromfile)
	        end;
	    write(tofile,ch);
	    reset(fromfile); reset(tofile)
	    end;
	
	procedure swapptr(var first,second:fileptr);
	var
	    helpptr : fileptr;
	    begin
	    helpptr := first; first := second; second := helpptr
	    end;
	``

Section	Symptom and Comments

(continued)

```
        procedure checkcontents(thefile:fileptr; expectedvalue:integer);
        var
            actualvalue : integer;
        begin
            readln(thefile^,actualvalue);
        end;

        begin
            new(ptr1);  new(ptr2);  new(ptr4);
            rewrite(ptr1^);  rewrite(ptr2^);  rewrite(ptr4^);
            write(ptr1^,'1');
            reset(ptr1^);
            copyandadd(ptr1^,ptr2^,'4');
            swapptr(ptr2,ptr4);
            checkcontents(ptr4,1);
        end.
```

- -

This example fails due to some internal consistency error in the run-time library. Whatever the cause, the Pascal run-time should never dump core, but should issue some reasonable run-time error message.

6.4.5-15
6.4.6-9
6.4.6-10
6.4.6-12
6.7.2.4-4

The MIPS Pascal compiler does not always detect out-of-range errors correctly, even when the **-C** switch is used:

```
        type
            subrange = 0..5;
        var
            i : subrange;

        procedure test(a : subrange);
        begin
            writeln(' THE VALUE OF A IS ', a);
        end;

        begin
            i:=5;
            test(i*2);    { error }
        end.
```

- -

In this specific example, the compiler is able to track the value of **i** into the procedure **test** when the optimizer is enabled and when range checking is enabled. If, however, the optimizer is not used, or if range checking is not explicitly enabled, no error message is issued. While the latter is an acceptable constraint, we do not feel that the presence of the optimizer should influence range checking. In this example, and many others, range checking was only performed at compile time, not at run-time. In addition to parameter passing, range checking also fails with:

- simple variable assignments

- array indexing

- incompatible (non-overlapping) set assignments

- sets passed as parameters

This is very bad behavior for a Pascal compiler to exhibit, especially since Pascal is supposed to be a strongly typed, range checking language. Note again that these errors occurred even when range checking was enabled during compilation.

Section	Symptom and Comments

6.5.5-2
 6.5.5-3
	The run-time error in this program is not detected:
	
	 ``
	var
	    fyle : text;
	procedure naughty(var f : char);
	    begin
	        if f='G' then
	            put(fyle)
	    end;
	begin
	    rewrite(fyle);
	    fyle^:='G';
	    naughty(fyle^);
	end.
	``
	 -
	 This program causes an error by changing the current file position of a file, while the buffer-variable is an actual variable parameter to a procedure. The error should be detected by the run-time.

6.6.3.1-9	The following program fragment does not compile:
	
	 ``
	type
	    t = 0..10;
	function f(t: integer): t;
	``
	 The error that is given is that `t` (the second instance) is *"Identifier is not of appropriate class"*.
	 -
	 The problem is that the compiler is not keeping type declarations and variable declarations in different name spaces. The declaration of a local variable `t` correctly overrides all other enclosing declarations. However, the declaration also obscures the declaration of the *type* `t`, which is incorrect.

6.6.3.2-3	The MIPS Pascal compiler passes all arrays by reference, regardless of the presence of a **var** qualifier. - This is bad news for portability. It is acceptable for a Pascal compiler to pass a non-**var** array by reference, provided it is treated as a read-only array in the called routine. However, the MIPS Pascal compiler does not even do this check, and simply passes the address of the array into the routine, allowing full access to the array body. Truly, it is very inefficient to copy the entire contents of an actual array parameter into a formal array parameter, but if that is the action desired by the programmer (and demanded by the Standard), then the compiler must perform this action.

6.6.3.5-2	The MIPS Pascal compiler does not check for function return-type congruity:
	
	 ``
	type
	    natural=0..maxint;
	var
	    k:integer;
	
	function actual(i:natural):natural;
	begin
	    actual:=i
	end;
	
	procedure p(function formal(i:natural):integer);
	begin
	    k:=formal(10)
	end;
	``

Section	Symptom and Comments
| *(continued)* | ```
    begin
        p(actual);
    end.
``` |
| | - |
| | The return types of the function **formal** do not match those of the function **actual**. This is a severe portability problem because the compiler does not check for an incompatibility that other compilers will surely complain about. In addition, it violates the strongly typed nature of Pascal. |
| **6.6.3.6-2** **6.6.3.6-4** | The MIPS Pascal compiler does not check for parameter list congruity, whether the parameters are of type **var** or not:<br>```
    program failure(output);
    type
        natural = 0..maxint;

    procedure actual(i:integer; n:natural);
    begin
        i:=n
    end;

    procedure p(procedure formal(a:integer;b:integer));
    var
        k,l:integer;
    begin
        k:=1; l:=2;
        formal(k,l)
    end;

    begin
        p(actual);
    end.
``` |
	- -
	The parameter types of the procedure **formal** do not match those of the procedure **actual**. This is a severe portability problem. In addition, it violates the strongly typed nature of Pascal.
6.6.5.2-19	Calling the built-in function **get** with no parameters causes a fatal error in /usr/lib/upas.
	- -
	This shortcoming in the MIPS Pascal compiler is indicative of the rather sparse error recovery system built into the compiler. Section 8.5 discusses this shortcoming in more detail.
| **6.6.2-7** | The MIPS Pascal compiler fails to detect when a function assignment is not executed:<br>```
    function area(a : real) : real;
    var
        x : real;
    begin
        if a > 0 then x:=3.1415926*a*a
                else area:=0
    end;

    begin
        writeln(area(2.0));
    end.
``` |
| | - |
| | The Pascal Standard states that the result of a function will be the last value assigned to its identifier. If no assignment occurs, then the result is undefined. The MIPS Pascal compiler is in error by not detecting this fact. |

Section	Symptom and Comments
6.6.5.2-5 **6.6.5.2-6** **6.6.5.2-7** **6.6.5.2-9** **6.6.5.2-10** **6.6.5.2-12** **6.6.5.2-13** **6.6.5.2-14** **6.6.5.2-15** **6.6.6.5-6** **6.6.6.5-7** **6.6.6.5-8**	This test fails to cause an error by applying 'reset' to an undefined file: ``` var f : file of integer; begin reset(f); end. ``` - This is another example of the MIPS Pascal compiler allowing uninitialized variables to be used in expressions. Other errors involving files include: • Allowing a `get` following a `rewrite`. • Allowing a `read` of a type incompatible with the `file` type. • Allowing a `write` of a type incompatible with the `file` type. • Allowing a `get` of a type incompatible with the `file` type. • Allowing a `put` of a type incompatible with the `file` type. • Allowing a `get` past the end of a file. • Allowing a `put` to an undefined buffer variable. • Allowing an `eof` to an undefined `file` variable. • Allowing an `eoln` while `eof` is true. • Allowing an `eoln` to an undefined `file` variable. The only error that *is* detected correctly is: • A `put` on a file not open for writing.

Section	*Symptom and Comments*

`6.6.5.3-6`
`6.6.5.3-7`
`6.6.5.3-8`
`6.6.5.3-9`
`6.6.5.3-10`
`6.6.5.3-11`
`6.6.5.3-13`
`6.6.5.3-14`
`6.6.5.3-16`
`6.6.5.3-17`
`6.6.5.3-21`
	The following example fails to detect the use of a pointer after it has been `disposed`:

```
type
   pointer = ^integer;
var
   p : pointer;
begin
   new(p);
   p^ := 10;
   dispose(p);
   writeln(p^);
end.
```

- -

The MIPS Pascal compiler and run-time is not performing any checks on the validity of pointers, including:

- Allowing a `dispose` on a pointer whose value is currently active as a `var` parameter.

- Allowing a `dispose` on a pointer which is currently being referenced by a `with` statement.

- Allowing the use of a pointer after it has been `disposed`.

- Allowing the use of a pointer that, through assignment, was equal to another pointer that has been `disposed`.

- Allowing a generic `dispose` on a pointer referencing a variant record, or passing different or the wrong number of parameters to the long form of `dispose` (this is merely a portability problem, since the MIPS Pascal compiler uses the generic UNIX memory allocation mechanism).

Section	Symptom and Comments

Section · *Symptom and Comments*

(continued)

- Allowing a reference (either left or right-hand side, or parameter) to the pointer **p^** when **p^** refers to a variant record (i.e., a reference other than to a component of the record). This results in a potentially illegal copying of differing variant record components.

- Allowing the activation of a variant part other than that created by a call to `new(p, c1, c2 ...)`.

All of these failings of the MIPS Pascal compiler are dangerous ones. The first four are classic problems of the C run-time library that should be fixed in a type and range checking language such as Pascal. The last failing presents a severe problem, since only the minimum space is allocated in the call to `new`, and activating a different variant part may write to other, unrelated areas of memory. All of these errors should be fixed.

6.6.5.4-2
6.6.5.4-3
6.6.5.4-4
6.6.5.4-5
6.6.5.4-6
6.6.5.4-7

The MIPS Pascal compiler and run-time fail to detect that the ordinal type parameter to the built-in procedure `pack` is not assignment compatible with the index type of the unpacked array parameter:

```
type
   pak = packed array [ 0 .. 15 ] of boolean;
var
   a: array [ 1 .. 16 ] of boolean;
   z: pak;
   i: 1 .. 16;
begin
   for i := 1 to 16 do
      a[i] := true;
   pack(a, 0, z);
end.
```

- -

The MIPS Pascal compiler is not performing the following checks on arrays:

- Not detecting that the ordinal type parameter to the built-in procedure `pack` (or `unpack`) is not assignment compatible with the index type of the unpacked (packed) array parameter:

- Allowing `pack` (`unpack`) to be called on an array that contains undefined elements.

- Allowing the index of the unpacked (packed) array to be exceeded in a call to `pack` (`unpack`).

The last case is especially nasty, since it implies that the array bounds can be exceeded, writing to an area of memory that may contain other, unrelated information. Since these errors are not detected, spurious program behavior can result. The second error is very difficult and expensive to detect, but the other two errors should be corrected.

6.6.6.4-9

The MIPS Pascal compiler allows the `ord` function to be applied to a pointer.

```
var
   ptr : ^integer;
   i : integer;
begin
   new(ptr);
   i := ord(ptr);
end.
```

- -

Again, the MIPS Pascal compiler is generally fairly poor at checking for assignment compatibility. This is another example of the failure of the compiler to adhere to the Pascal typing rules.

Section	Symptom and Comments
6.6.6.2-4 6.6.6.2-5 6.6.6.2-12 6.6.6.2-13 6.6.6.2-14 6.6.6.3-3 6.6.6.3-4 6.6.6.4-5 6.6.6.4-6 6.6.6.4-7	The MIPS Pascal compiler does not check that the parameters to arithmetic functions are of the correct type: ``` var a : real; begin a:=sqr('4'); end. ``` - The MIPS Pascal compiler is generally fairly poor at checking for assignment and range compatibility. This is one example of the failure of the compiler to adhere to the Pascal range and typing rules. Other failures include: • Allowing a negative number to be passed to the `ln` function. • Allowing a negative number to be passed to the `sqrt` function. • Allowing an undetected (integer) overflow of the `sqr` function. • Allowing a number larger than **maxint** to be passed to `trunc` or `round`. • Allowing the `succ` function on the last value of an ordinal type. • Allowing the `pred` function on the first value of an ordinal type. • Allowing the `chr` function to be used on ordinal types exceeding the range of characters. In all of these cases, no compile time or run-time error is issued. The purpose of the range and type checking inherent in most Pascal compilers is to detect these types of programming errors. By failing to detect these errors, the MIPS Pascal compiler is allowing many potential bugs to creep into programs. It should be noted that these errors pass through even when the **-C** switch is used to enable run-time range checking.
6.7.2.2-8 6.7.2.2-9 6.7.2.2-10 6.7.2.2-11 6.7.2.2-12 6.7.2.2-13 6.7.2.2-16 6.7.2.2-19	The MIPS Pascal compiler does not issue a run-time error when a value larger than **maxint** is printed: ``` var i: integer; function maxie: integer; begin x:= maxint; end; begin i := 100; writeln(' MAXINT + 100 = ',maxie+i); end. ``` - In those cases, in which the condition can be detected at compile time, the MIPS Pascal compiler will report on arithmetic overflow. There appears to be no run-time range checking on any arithmetic operations, including: • Allowing a negative second operand in the `mod` operation. • Allowing a floating-point divide by zero. • Allowing run-time overflow on addition. The run-time will report on an integer division or modulo by zero, but it does so by issuing a break point trap and dumping core. This is unacceptable.

Section	Symptom and Comments
`6.7.2.2-18`	The MIPS Pascal compiler allows operands of other than real or integer to be used in a division operation: ``` var c : char; r, s: real; c := 'A'; s := 1.5; r := c/s; ``` - Since Pascal is a strongly typed language, the MIPS Pascal compiler should check for such blatant violations of type compatibility. Instead, it is following the semantics of C, and considering a character type to be the same as an integer type. This is clearly an error.
`6.7.2.4-9` `6.7.2.5-10`	The MIPS Pascal compiler allows a non ordinal type (i.e., strings or sets) to be the left operand of the `in` operator: ``` var s : set of 0..10; begin s := [3]; if (s in []) or ('HI' in []) then writeln('OOPS'); end. ``` - This is another example of the MIPS Pascal compiler having trouble with type checking and with set operations. A lot of work needs to be done with both of these to bring the compiler up to a workable level.
`6.7.2.5-7`	The MIPS Pascal compiler allows equality and non-equality between different pointer-types: ``` type natural = 0..10; one = ^integer; two = ^natural; var x: one; y: two; begin new(x); x^ := 2; new(y); y^ := 3; if (x <> y) or not (x = y) then writeln('YOW'); end. ``` - Since the range of integers expressed by type `integer` and type `natural` are different, comparisons across these pointer types should be illegal. However, the MIPS Pascal compiler allows them which introduces a serious portability problem and demonstrates a dangerous lack of type checking.

Section	Symptom and Comments
6.9.3.1-2 **6.9.3.1-3** **6.9.3.1-7**	This program deviates from the Standard because it allows output of a non-positive field width: ```pascal var f:text; i:integer; begin rewrite (f); for i:=10 downto -1 do write(f,' ','.':i, 'REP=',i); end. ``` - The MIPS Pascal compiler allows this illegal program, as well as a program which prints a floating point number with a zero field width fraction, to compile and run. While this is a small problem on the MIPS (the program will at least print out *something*), it presents a large portability problem.
6.10-8	This program deviates from the Standard because the *program-parameter* **f** has been subsequently declared as a *function*. ```pascal program t6p10d8(f, output); function f:boolean; begin f := true end; begin writeln('OOPS') end. ``` - In this case, the type of **f** is initially inferred from the program definition. However, it is later defined as a function. When it is referenced, what type is it? In this case, it will be a function, which indicates a lack of the appropriate type checking.

8.3. Bad Code

This section describes samples of incorrect code being generated by the compiler from legal Pascal source code. These examples are the nightmare of every programmer – debugging them is very difficult because as far as the programmer can tell, the source code is perfectly reasonable, although the output of the compiler does not exactly correspond to the input.

These deviations represent serious problems with the compiler. In fact, there may be more examples than the ones shown here. The only reason these were found is because of specific checks put in the test programs to look for such errors, or because the compiler exhibits different behavior with and without the optimizer engaged. In the past, we have been able to generate similar errors by writing intentionally noxious code, or by misusing Pascal. The primary problem lies in the fact that compilers are all too often tested only on good code, and not on incorrect code.

Section	*Symptom and Comments*
`6.1.1-3`	The following code fragment (where `stv` is a `set of [0..9]`) causes an infinite loop at optimization level 2 or above: ``` stv := [1]; repeat with pkr do; until (1) in stv; ``` - The compiler generates the following code: ``` # 140 stv := [1]; li $14, 1073741824 sw $14, -16($6) # 141 repeat $62: # 142 with pkr do; # 143 until (1) in stv; b $62 ``` This plainly loops forever. The reason the compiler generates this code is not obvious, although examining the code generated at optimization level 1 gives us a clue: ``` # 140 stv := [1]; li $9, 1073741824 lw $10, 36($sp) sw $9, -16($10) # 141 repeat $64: # 142 with pkr do; # 143 until (1) in stv; lw $11, 36($sp) lw $12, -16($11) sll $13, $12, 1 bge $13, 0, $64 ``` At this lower level of optimization, the compiler is performing the set-inclusion test. Unfortunately, the test is generated incorrectly. Rather than shifting a single bit to the left (and then comparing the result with the set), the compiler instead is shifting the *set* left and comparing it with zero. The compiler can determine this as a compile-time constant (at the higher optimization level), and it generates an infinite loop.
`6.5.4-1` `6.5.4-2`	The MIPS Pascal compiler allows a pointer which is undefined, or explicitly initialized to **nil,** to be dereferenced, creating a core dump: ``` type rekord = record a : integer; b : boolean end; var pointer : ^rekord; begin pointer:=nil; pointer^.a := 1; end. ``` - Even with value tracking, the compiler is unable to detect this blatant error. In the more subtle case where the value of **pointer** is left uninitialized, the compiler exhibits similar behavior. This is another manifestation of the lack of run-time checking by the compiler, and it should be corrected. At the very least, the run-time should print out a Pascal run-time error message before performing the core dump.

Section	Symptom and Comments
6.6.5.2-8 **6.6.5.2-11**	The following program dumps core on execution: ```pascal var f : file of char; begin get(f); end. ``` - The reason the program dumps core is that the file **f** is undefined when the `get` is performed (i.e., no `reset` was executed). The run-time library should have detected this fact at run-time. Instead, it rather ungracefully terminated execution. At the very least, a run-time error message should have been issued. The program will also dump core if `page` is substituted for `get`.
6.6.5.3-4 **6.6.5.3-5**	The following program dumps core on execution: ```pascal type rekord = record a : integer; b : boolean end; var ptr : ^rekord; begin ptr:=nil; dispose(ptr); end. ``` - The reason the core dump occurs is that **ptr** is `nil`. The run-time should test for illegal values of pointers before executing the dispose operation. This program will also dump core if **ptr** is left undefined (instead of being explicitly set to `nil`). In the latter case, if the variable containing the pointer is uninitialized, but contains (through happenstance) the value of a different pointer, a different dynamic element could be disposed of – a highly undesirous effect. These shortcomings should be corrected and have an error issued from the run-time, rather than have the program dump core.
6.7.1-6	The following example works correctly without the optimizer engaged but fails when optimization level 2 is used: ```pascal n := 2; if [1,2,succ(n)]=[1..3] then c:=c+1; ``` - The reasons for failure result from compile-time value tracking and elimination of redundant code. Specifically, the optimizer knows the values of all of the conditional expressions at compile-time and simply increments `c` for each case where the conditional is true (eliminating the test code in the process). Unfortunately, the optimizer fails to track and recognize the expression `[1,2,succ(n)]=[1..3]` as being true. Examining the assembly output for this fragment, we see that this is another manifestation of the bad code generated for sets:

Section	Symptom and Comments
(continued)	```
# 26   if [1,2,succ(n)]=[1..3] then
  lw $13, 36($sp)
  addu $14, $13, 1
  addu $15, $14, -96
  sltu $24, $15, 32
  not $25, $14
  sll $8, $24, $25
  addu $9, $14, -64
  sltu $10, $9, 32
  sll $11, $10, $25
  or $12, $8, $11
  addu $13, $14, -32
  sltu $15, $13, 32
  sll $24, $15, $25
  or $9, $12, $24
  sltu $10, $14, 32
  sll $8, $10, $25
  or $11, $8, 1610612736
  xor $13, $11, 1879048192
  or $15, $9, $13
  bne $15, 0, $32
  .loc 2 27
# 27   c:=c+1;
  lw $12, 32($sp)
  addu $24, $12, 1
  sw $24, 32($sp)
$32:
```
The two constants 1610612736 and 1879048192 are 0x60000000 and 0x70000000, respectively (which correspond to the sets [1..2] and [1..3], respectively).  The optimizer is performing a correct optimization, given an incorrect source of assembly instructions. |
| **6.7.2.2-4** | The compiler issues the error *"uopt: Warning: multiplication overflow"* on the following example when the optimizer is enabled but issues no error if it is disabled.  In either case, no run-time error is issued.<br><br>```
max:=-(-maxint);
if odd(maxint) then
  i:=(max-((max div 2)+1))*2
```<br>- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -<br><br>The problem here is that the optimizer is of reorganizing the arithmetic expression (while no such reorganization is performed without the optimizer).  This rearrangement causes the arithmetic overflow.  Since the expression was parenthesized specifically to avoid the mathematical overflow, we believe that the compiler is in error. |
| **6.7.2.5-2** | This program fragment does not print **TRUE** as it should:<br><br>```
b := [2,3,4];
c := 3;
if (c in b) then
    writeln('TRUE');
```<br>- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -<br><br>This is another example of the **in** operator generating bad code and having it optimized out to nothingness.  When expressions such as b<>c or b<=c are used, the compiler sometimes also functions incorrectly.  The **in** operator (as well as the equality operator from Example 6.7.1-6) seem to be failing. |

Section	Symptom and Comments
| **6.8.1-6**<br>**6.8.1-7** | The MIPS Pascal compiler allows a `goto` into a `with` statement, with disastrous results. The following program dumps core:<br><br>```pascal
type
    rec = record
            y: integer;
            end;
    ptrec = ^rec;
var
    x: ptrec;
    done: boolean;
begin
    new(x);
    x^.y := 100;
    done := false;
    if done then
        with x^ do
            1:
            begin
            writeln(y);
            y := y + 1
            end;
    if not done then
        begin
        done := true;
        goto 1
        end
end.
```<br><br>- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -<br><br>In this example, the placement of the label is legal, in that it references a simple statement. The `goto` is illegal, however, in that it references an illegal target. In this case, the initialization code for the `with` statement is skipped, and an indirection through an uninitialized register is performed in accessing **x^.y**. This "feature" should be removed from the MIPS Pascal compiler, and only the legal set of `gotos` should be allowed. In general, the MIPS Pascal compiler is implementing the semantics of **C** in allowing this feature. |
| **6.10-10** | The following program causes a core dump:<br><br>```pascal
var
    c : char;
begin
    writeln('Start');
    reset(output);
    read(output,c);
end.
```<br><br>- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -<br><br>This program attempts to reuse **output** as a regular file that can be read from. This attempt is perfectly legal according to the Standard because it is implementation-defined as to whether **output** actually goes to a terminal (all it need [must] do is treat **output** as an ordinary `file`). The MIPS Pascal compiler implementation of **output** classes it as the UNIX stdout file using the standard UNIX file conventions. This breaks the Pascal standard. While few users may take advantage of this aspect of the Standard, there are other ramifications that must be considered. |

Section	Symptom and Comments
-none-	The following code generates the error from the linker: *"Undefined: write_set"*:

```
var s : set of 0..10;

begin
    s := [1,3,5];
    writeln(s);
end.
```

- -

The implementors of the MIPS Pascal compiler library functions have either not implemented the `write_set` operation, or they have failed to include it in the distribution. In any event, `write_set` is not in the library file, and programs which attempt to print out the contents of sets will fail to compile successfully.

8.4. MIPS Extensions to Standard Pascal

According to MIPS, the MIPS Pascal compiler contains the following extensions to the Pascal Standard:

- Allows the use of underscores (_) in variable names.

- Prints alphabetic labels (see test 6.1.6-6 in Section 8.1).

- Allows numbers in a non-decimal radix. Any radix between base 2 and base 36 is permitted. `write` and `writeln` also support arbitrary radix output.

- Predefines three extra data types in the compiler:

    - `double` – double precision floating-point

    - `cardinal` – unsigned integers in the range of 0..4294967295

    - `pointer` – a pointer to any data type

- The MIPS Pascal compiler always does short-circuit boolean evaluation (this is a permitted extension, but dependency on it guarantees non-portability).

- Automatically pads strings with trailing spaces to fill them <to the required length (see test 6.7.1-10 in section 8.1).

- Allows non-ASCII characters in strings, following the UNIX convention of escape character sequences.

- Permits constant expressions in type or array-bound definitions. It also supports the following additional built-in functions:

    - `bitand` – bitwise and

    - `bitor` – bitwise or

    - `bitxor` – bitwise xor

    - `lshift` – logical left shift

    - `rshift` – logical right shift

    - `lbound` – the lower bound of an array (this is odd in that this facility is provided but conformant arrays parameters are not)

    - `hbound` – the higher bound of an array (this is odd in that this facility is provided but conformant arrays parameters are not)

- `first` – the first value of a scalar type

- `last` – the last value of a scalar type

- `sizeof` – the size (in bytes) of a data type

- `min` – the minimum of a set of scalars

- `max` – the maximum of a set of scalars

- `assert` – evaluates a boolean expression and prints a run-time error message

- `date` – the current date in string form

- `time` – the current time of day in string form

- `clock` – the number of milliseconds of CPU time used by the process

- `argv` – returns a specified program argument as passed in from the shell

- Permits ranges as `case` statement constants (see test 6.8.3.5-7 in Section 8.1).

- Includes an `otherwise` clause in the `case` statement.

- Allows a `return` statement to exit a subroutine or function.

- Permits a `continue` and a `break` statement with semantics similar to the `C` version.

- Adds the concept of shared variables and the keyword `external` to facilitate separate compilation.

- Adds variables to have an initialization clause along with their declaration part. This is especially useful for initializing arrays.

- Relaxes the declaration ordering rules. See tests 6.2.1-8, through 6.2.1 -9, and -10 in the section on portability (Section 8.1).

- Allows the `rewrite` and `reset` routines to take an optional filename parameter.

- Allows the `write` and `writeln` routines to work on enumerated types.

- Employs a preprocessor (namely *cpp*) before compilation.

8.5. Local Conclusions

In spite of the large number of specific deviations, the MIPS Pascal compiler is a fairly reasonable compiler which generates very efficient code. The robustness of the compiler is, however, questionable at best. Even with the compiler option -c, which, according to the on-line manual page entry for the Pascal compiler *pc*, is supposed to *"generate code for run-time range checking,"* the range and type checking of the compiler are fairly specious and need to be made much more robust.

It is possible to assign numbers out of their range, to assign one set to another which has no overlapping objects, to generate (without detection) arithmetic overflow and underflow, to index through a deleted pointer, to read past the end of a file, and so on. In short, the MIPS Pascal compiler implements the simple UNIX and C model of a programming language.

We would not dwell so much on the failings of the Pascal compiler were it not for one simple fact: the MIPS common code generator, optimizer, assembler/reorganizer, and the MIPS Pascal compiler itself are all written in this same version of Pascal. Thus, since the compiler does not check for

pointer validity, range overflow, and file validity, unless the programmer performs these checks explicitly, it is entirely possible that all manner of bugs will be lurking in the depths of these programs. MIPS Incorporated has repeatedly asserted that this is not true, but we do not agree. The tests that they have run on their compilers are, by their own admission, a set of programs which are known to function correctly.[60] These programs will only detect that the compiler and utilities function correctly given *correct input*. They in no way test the compilers' behavior given incorrect, or for that matter, merely *different* input. We are willing to give long odds that adding the full complement of range and bounds-checking code to the Pascal compiler will likely turn up at least one hitherto undetected violation of range or boundary limits.

There are numerous examples in the validation suite of the compiler or the run-time crashing while executing suspicious (or in some cases, correct) Pascal source code. While it is unreasonable for the run-time to crash, it is unacceptable for the compiler to *ever* crash, no matter how unreasonable the input. Regrettably, the MIPS Pascal compiler could stand a bit of strengthening in this area.

---

[60]Examples are: the UNIX utility set, their own compilers, the run-time libraries, benchmarks, etc.

9. Unexpected Program Behavior

Figure 9-1 shows a simple assembly program that has four load instructions from two different addresses. Both of the addresses are in the `sdata` psect, and thus all addresses are supposed to be `gp`-relative.

```
        .sdata
        .align 2
x:      .word 1

        .text
L:
        lw $2,x
        la $3,x
        lw $4,y
        la $5,y

        .sdata
        .align 2
y:      .word 1
```

Figure 9-1: Assembly Code that Triggers `gp`-Relative Bug

The MIPS assembler/reorganizer is supposed to take assembly language programs and translate them into MIPS M/500 native instructions, potentially changing some instruction sequences into others. One of the instruction sequences that it is supposed to modify is the load-class instruction. If the source of the load is at a `gp`-relative address, then the assembler reorganizer should make the load be `gp`-relative. If not, then the assembler/reorganizer should make the load be from a 32-bit address. The advantage to the `gp`-relative load is that it requires only one instruction, while the 32-bit address load requires two.

In the source code in figure 9-1, all of the address references are properly `gp`-relative, and each should be translated into a single MIPS M/500 instruction. However, as can be seen in figure 9-2, this is not the case.

```
0x0:    8f828010        lw      v0,-32752(gp)
0x4:    27838010        addiu   v1,gp,-32752
0x8:    3c010000        lui     at,0x0
0xc:    8c24001c        lw      a0,28(at)
0x10:   2425001c        addiu   a1,at,28
0x14:   00000000        nop
```

Figure 9-2: MIPS M/500 Code from Figure 9-1

Both of the references to the variable **x** are encoded as a `gp`-relative reference, whereas both references to the variable **y** are not. The only difference between **x** and **y** is that **y** is a forward reference. This behavior is reminiscent of an early 1 or 1.5 pass assembler and should not be present in a modern 2 pass assembler.

10. Conclusions

The RISC Evaluation Project set out to answer two questions:

1. Taking hardware and system software together, is a machine built using RISC principles a feasible competitor to a CISC machine?

2. How well do the actual hardware and software of a specific RISC system (in this case, the MIPS M/500) compare with those of a specific CISC system (in this case, the VAX)?

The first question can be answered only qualitatively, in terms of one's instinct or opinion. The second question can be addressed quantitatively by analyses of benchmarks, instruction-set usage patterns, and other data.

This report documents in detail our answer to the second question, presenting both the data themselves and, where appropriate, the evaluation methods we employed. Our conclusions, culled from the body of the report, are:[61]

- The particular machine studied, the MIPS M/500, conforms closely to the CORE ISA definition and can fairly be classed as a RISC class machine.

- The overall performance of the hardware is very impressive, about 8 million machine instructions per second.

- When code in high-level languages is run, this hardware performance yields objective benchmark and application performance of about five times that of a VAX-11/780 running UNIX 4.3 BSD, this being the unofficial "one MIP" machine.

- This level of performance was consistent across a wide variety of benchmarks and applications. Although we stress that benchmark statistics without the accompanying evaluation are next to useless, we also observed that the MIPS M/500 benchmarked at 2408 Whetstones (FORTRAN single precision), and at 14184 Dhrystones (register and non-register).

In attempting to answer the first question, we made the following observations, which we again emphasize are qualitative rather than quantitative:

- Hand coding of small benchmarks can still provide major improvement over compiler-generated code. Nevertheless, compilers for the RISC machine performed, overall, much better than those for the CISC machine.

- The compiler-generated code shows substantially more effective usage of the RISC instructions and addressing modes, with no serious inefficiencies caused by omitted instructions and addressing modes. This finding, especially, bears out the claims made on behalf of RISC machines.

- Targeting a compiler to a RISC machine does not seem much harder than targeting one to a CISC machine. Different tasks have to be done, but the overall amount of work is about the same. However, we believe that the compiler should also perform any object-code reorganization that may be required, rather than leaving this to a separate program.

---

[61]More detailed conclusions can be found in the sections entitled Local Conclusions. These are sections 3.3 (assembly language reorganization), 4.1.6, LC.WHET, 4.2.3, and 4.3.2 (benchmarking), 6.2.8, and 6.3.4 (compiler utilization of the instruction set), 8.5 (Pascal compiler conformance), and Appendix Section C.7 (conformance to the CORE ISA). The reader is urged to read these sections for more information.

- Fewer actual instructions are required by a CISC machine to perform the same function as a RISC machine – an expected phenomenon. The ratio of the number of bytes required to represent these instructions (a much more valid measure) is far closer to one than is the ratio of instruction counts. With memory costs decreasing as they are, the greater processing power of the RISC architecture far outweighs the slightly increased memory use.

We also formed some conclusions about the assessment process itself, which are perhaps of general applicability:

- It is not easy to disentangle the effects of hardware, operating system, file system, compilers, and languages. The investigator must be prepared to recognize tiny anomalies, track down vague clues, run down blind alleys, and perform a large number of experiments differing only in minute detail.

- One must be very specific about what one is measuring. The same benchmark in two languages may yield quite different numbers; the same program run twice may give different timings; two compilers for the same language may show radically different code patterns for the same idioms.

- *The purpose of computing is insight, not numbers.* No datum is useful unless it can be *explained*; no explanation is useful unless it serves to illuminate an issue or progress an argument. If there is a "bottom line" in benchmarking, it is that you must understand what you are doing and why you are doing it.

Finally, it seems appropriate to reiterate the main conclusion of this investigation:

> There may not always be a right choice and a wrong choice in the
> RISC versus CISC debate. However, in all the areas we examined,
> the *RISC* design was *never* the wrong architectural choice.

Bibliography

[Am2900 87]   *Am29000 Streamlined Instruction Processor User's Manual*
                   Advanced Micro Devices, Sunnyvale, CA, 1987.

[Barbacci 78]   M. R. Barbacci, W. E. Burr, S. H. Fuller, D. P. Siewiorek.
                   Evaluation of Alternative Computer Architectures.
                   Technical Report CMU-CS-77-EACA, Carnegie Mellon University Computer
                        Science Department, February, 1978.

[Bell 86]   C. Gordon Bell.
                   RISC: Back to the future?
                   Datamation 32(11), June, 1986.

[Buchholz 69]   W. Buchholz.
                   A Synthetic Job for Measuring System Performance.
                   IBM System Journal (4), 1969.

[Ciechanowicz 86] Z. J. Ciechanowicz and Brian Wichmann.
                   A Reader's Guide to Pascal Compiler Validation Reports.
                   Technical Report DITC 24/83, National Physics Laboratory, Teddington, Mid-
                        dlesex TW11 0LW, UK, 1986.

[Cook 82]   R.P. Cook and I. Lee.
                   A Contextual Analysis of Pascal Programs.
                   Software Practices and Experiences 12(2):195-203, February, 1982.

[CORE 87]   Robert Firth.
                   CORE Set of Assembly Language Instructions for MIPS-Based MicroProcessors.
                   Technical Report Maintained Under Contract RADC F19628-85-C-0003, Software
                        Engineering Institute, 1987.
                   Originally prepared by Thomas Gross of Carnegie Mellon University.

[Curnow 76]   H. J. Curnow and B. A. Wichmann.
                   A Synthetic Benchmark.
                   Computer Journal 19(1):43-49, February, 1976.

[DEC 72]   *DecSystem-10 Assembly Language Handbook*
                   Digital Equipment Corporation, Maynard, MA, 1972.

[DePrycker 82]   M. DePrycker.
                   On the Development of a Measurement System for High-Level Language
                        Program Statistics.
                   IEEE Transactions on Computing 9:883-891, September, 1982.

[Fleming 86]   P. J. Fleming and J. J. Wallace.
                   How Not To Lie With Statistics: The Correct Way to Summarize Benchmark
                        Results.
                   Communications ACM 29(3):218-221, March, 1986.

[Himelstein 87]   Mark Himelstein et al.
                   Cross-Module Optimization: Its Implementations and Benefits.
                   In *Usenix Conference Proceedings.* June, 1987.

[Hinnat 84]   David F. Hinnat.
                   Benchmarking UNIX Systems.
                   BYTE 9(8), August, 1984.

[Jensen 85]   Kathleen Jensen and Niklaus Wirth.
                   Pascal - User Manual and Report.
                   Springer Verlag, New York, 1985.

[Kernighan 70]   Brian Kernighan and Dennis Ritchie.
                   The C Programming Language.
                   Prentice-Hall, Englewood Cliffs, NJ, 1970.

[McDonell 87]   Ken McDonell.
                   Taking Performance Analysis out of the "Stone" Age.
                   In *Usenix Conference Proceedings.* June, 1987.

[Milutinovic 87]   Veljko Milutinovic et al.
                   Architecture/Compiler Synergism in GaAs Computer Systems.
                   IEEE Computer 20(5):72-93, May, 1987.

[MIPS 86a]   *Assembly Language Programmer's Guide.*
                   Mips Computer Systems, Inc., Sunnyvale, CA, 1986.

[MIPS 86b]   *Language Programmer's Guide.*
                   Mips Computer Systems, Inc., Sunnyvale, CA, 1986.

[Pascal 82]   *Specification for Computer Programming Language Pascal.*
                   British Standards Institution, 2 Park Street, London W1A 2BS England, 1982.

[Patterson 85]   David Patterson.
                   Reduced Instruction Set Computers.
                   Communications ACM 28(1):8-21, January, 1985.

[RISC 86]   .
                   How to recognize a RISC.
                   Mini-Micro Systems 9(13), November, 1986.

[Serlin 86]   Omri Serlin.
                   MIPS, Dhrystones, and Other Tales.
                   Datamation 32(11), June, 1986.

[SPARC 87]   *The SPARC^TM Architecture Manual*
                   Sun Microsystems, Inc., Mountain View, CA, 1987.

[Tannenbaum 78]   Andrew S. Tannenbaum.
                   Implications for Structured Programming for Machine Architecture.
                   Communications ACM 21(3):237-246, March, 1978.

[Tennent 85]   R. D. Tennent.
                   A Comparison of the ANSI and ISO Pascal Standards.
                   Software - Practice and Experience 15(8):821-822, August, 1985.

[Unix 79]   *UNIX Assembler Reference Manual*
                   AT&T Bell Laboratories, Holmdale, NJ, 1979.

[Weicker 84]   Reinhold P. Weicker.
                   Dhrystone: A Synthetic Systems Programming Benchmark.
                   Communications ACM 27(10):1013-1030, October, 1984.

[Wichmann 76]   Brian Wichmann.
                   Ackermann's Function: A Study in the Efficiency of Calling Procedures.
                   BIT 16:103-110, 1976.

[Wichmann 77]   Brian Wichmann.
                 How To Call Procedures, or Second Thoughts on Ackermann's Function.
                 Software Practice and Experience 7, 1977.

[Wichmann 82]   Brian Wichmann.
                 Latest Results from the Procedure Calling Test, Ackermann's Function.
                 Technical Report DITC 3/82, National Physics Laboratory, Teddington, Middlesex
                     TW11 0LW, UK, 1982.

[Wichmann 83]   Brian Wichmann and Z. J. Ciechanowicz (editors).
                 Pascal Compiler Validation.
                 John Wiley & Sons, Chichester, UK, 1983.

[Zeigler 83]   S. F. Zeigler and R. P. Weicker.
                 Ada Language Statistics for the iMAX 432 Operating System.
                 Ada Letters 2(6):63-67, May, 1983.

Appendix A: Overview of MIPS Instruction Set Translation

Table A-1 lists the correspondences between the MIPS high-level instruction set names for the registers and the MIPS M/500 machine instruction equivalents. Both names can be accessed by the user (see Chapters 1 and 7 of *"The Mips Assembly Language Programmer's Guide"* [MIPS 86a] for more details).

Register Name(s)	Equivalent Name(s)	Register Name(s)	Equivalent Name(s)
$0	zero	$16	s0
$at	at	$17	s1
$2	v0	$18	s2
$3	v1	$19	s3
$4	a0	$20	s4
$5	a1	$21	s5
$6	a2	$22	s6
$7	a3	$23	s7
$8	t0	$24	t8
$9	t1	$25	t9
$10	t2	$26 or $kt0	k0
$11	t3	$27 or $kt1	k1
$12	t4	$28 or $gp	gp
$13	t5	$29 or $sp	sp
$14	t6	$30 or $fp	fp or s8
$15	t7	$31	ra

Table A-1: MIPS M/500 High- and Low-Level Equivalent Register Names

What follows is a table of all of the MIPS assembly language instructions followed by the corresponding MIPS M/500 native instructions that are generated by the assembler reorganizer. We have attempted to cover all of the possible operand combinations allowed by each instruction. These modes are typically two operand (dest/src1, src2), three operand (dest, src1, src2), three operand with one immediate value (including a small integer, a large integer, and a large integer power of two), and three operand with one zero value (expressed as both an immediate value and as the zero register).

In all cases, the machine language output has been assembled relative to a base address of 0, so that all branches are based at the beginning of the code fragment. Each instruction takes up four bytes, so a branch to address 0x1c will transfer to the eighth instruction (counting from zero). Also, the large constant value 2097152 is 0x20000 (a convenient large power of two that exceeds the immediate operand size of the MIPS M/500). The constant values greater than 2097152 are used as non-even-multiples of two for comparison purposes.

The main table is designed to parallel the instruction order listed in Chapter 5 of the *MIPS Assembly Language Programmer's Guide [MIPS 86a]*. An alphabetic cross reference can be found in table A-2 at the end of this appendix section.

Assembler Input	*Machine Language Output*	*Comments*
`la    $4,($5)`	`addiu   a0,a1,0`	All of the addressing modes are available to all of the load instructions, but some do not make sense, in which case, loading the address of a indexed register is the same as taking the value of the register.
`la    $4,24`	`li      a0,24`	
`la    $4,2097156`	`lui     at,0x20` `addiu   a0,at,4`	Since the MIPS M/500 can only store a 16-bit address in a 32-bit instruction, the upper 16-bits of an address must be loaded in a separate instruction (the `lui`).
`la    $4,24($5)`	`addiu   a0,a1,24`	In this case, the address of a based address is the value in the base register plus the value of the offest.

`la    $4,2097156($5)`	`lui     at,0x20`
	`addu    at,at,a1`
	`addiu   a0,at,4`
		In this case, too, the address of a based address is the value in the base register plus the value of the offset. However, the addition must be done in two stages, because of to the limitation of the 16-bit immediate field.

`la    $4,BEGIN`	`lui     at,0` `addiu   a0,at,0`	Loading the address of a global variable (that is relocatable) requires that the upper 16-bits always be loaded, with the linker filling in the correct value (since it cannot be determined at assembly time what the value of the upper 16 bits will be).
`la    $4,BEGIN+24`	`lui     at,0` `addiu   a0,at,24`	

`la    $4,BEGIN($5)`	`lui     at,0`
	`addu    at,at,a1`
	`addiu   a0,at,0`
		What appears to be a superfluous `addiu` in this sequence is actually needed. The sequence of events here is to load the upper 16-bits of the address of `BEGIN`, then add in (i.e., index off of) register `a1`, then add in the lower 16-bits of the address of `BEGIN` (which will be relocated to some other address at link time).

`la    $4,BEGIN+24($5)`	`lui     at,0`
	`addu    at,at,a1`
	`addiu   a0,at,24`	

Assembler Input	*Machine Language Output*	*Comments*
lb $4,($5)	`lb` `a0,0(a1)` `nop`	
lb $4,24	`lb` `a0,24(zero)`	An absolute address is expressed as a based address off of the zero register.

lb $4,2097156	`lui` `at,0x20`
	`lb` `a0,4(at)`
	`nop`
		When the absolute address exceeds 16-bits, it is calculated in two stages, using `at` as a temporary register.

lb $4,24($5)	`lb` `a0,24(a1)`	

lb $4,2097156($5)	`lui` `at,0x20`
	`addu` `at,at,a1`
	`lb` `a0,4(at)`	

lb $4,BEGIN	`lui` `at,0` `lb` `a0,0(at)`	Relocatable addresses are unknown at assembly time, so their full 32 bits must be planned for by the assembler.
lb $4,BEGIN+24	`lui` `at,0` `lb` `a0,24(at)`	

lb $4,BEGIN($5)	`lui` `at,0`
	`addu` `at,at,a1`
	`lb` `a0,0(at)`	

lb $4,BEGIN+24($5)	`lui` `at,0`
	`addu` `at,at,a1`
	`lb` `a0,24(at)`
	`nop`	

lbu $4,($5)	`lbu` `a0,0(a1)` `nop`	The `lbu` instruction follows a format identical to the `lb` instruction.
lbu $4,24	`lbu` `a0,24(zero)`	

lbu $4,2097156	`lui` `at,0x20`
	`lbu` `a0,4(at)`
	`nop`	

lbu $4,24($5)	`lbu` `a0,24(a1)`	

lbu $4,2097156($5)	`lui` `at,0x20`
	`addu` `at,at,a1`
	`lbu` `a0,4(at)`	

lbu $4,BEGIN	`lui` `at,0` `lbu` `a0,0(at)`	
lbu $4,BEGIN+24	`lui` `at,0` `lbu` `a0,24(at)`	

lbu $4,BEGIN($5)	`lui` `at,0`
	`addu` `at,at,a1`
	`lbu` `a0,0(at)`	

lbu $4,BEGIN+24($5)	`lui` `at,0`
	`addu` `at,at,a1`
	`lbu` `a0,24(at)`
	`nop`	

Assembler Input		*Machine Language Output*		*Comments*
lh	$4,($5)	**lh** **nop**	**a0,0(a1)**	The lh instruction follows a format identical to the lb instruction.
lh	$4,24	**lh**	**a0,24(zero)**	
lh	$4,2097156	**lui** **lh** **nop**	**at,0x20** **a0,4(at)**	
lh	$4,24($5)	**lh**	**a0,24(a1)**	
lh	$4,2097156($5)	**lui** **addu** **lh**	**at,0x20** **at,at,a1** **a0,4(at)**	
lh	$4,BEGIN	**lui** **lh**	**at,0** **a0,0(at)**	
lh	$4,BEGIN+24	**lui** **lh**	**at,0** **a0,24(at)**	
lh	$4,BEGIN($5)	**lui** **addu** **lh**	**at,0** **at,at,a1** **a0,0(at)**	
lh	$4,BEGIN+24($5)	**lui** **addu** **lh** **nop**	**at,0** **at,at,a1** **a0,24(at)**	
lhu	$4,($5)	**lhu** **nop**	**a0,0(a1)**	The lhu instruction follows a format identical to the lb instruction.
lhu	$4,24	**lhu**	**a0,24(zero)**	
lhu	$4,2097156	**lui** **lhu** **nop**	**at,0x20** **a0,4(at)**	
lhu	$4,24($5)	**lhu**	**a0,24(a1)**	
lhu	$4,2097156($5)	**lui** **addu** **lhu**	**at,0x20** **at,at,a1** **a0,4(at)**	
lhu	$4,BEGIN	**lui** **lhu**	**at,0** **a0,0(at)**	
lhu	$4,BEGIN+24	**lui** **lhu**	**at,0** **a0,24(at)**	
lhu	$4,BEGIN($5)	**lui** **addu** **lhu**	**at,0** **at,at,a1** **a0,0(at)**	
lhu	$4,BEGIN+24($5)	**lui** **addu** **lhu** **nop**	**at,0** **at,at,a1** **a0,24(at)**	
lw	$4,($5)	**lw** **nop**	**a0,0(a1)**	The lw instruction follows a format identical to the lb instruction.

Assembler Input	*Machine Language Output*	*Comments*
lw $4,24	`lw      a0,24(zero)`	

lw $4,2097156	`lui     at,0x20`
	`lw      a0,4(at)`
	`nop`	

lw $4,24($5)	`lw      a0,24(a1)`	

lw $4,2097156($5)	`lui     at,0x20`
	`addu    at,at,a1`
	`lw      a0,4(at)`	

lw $4,BEGIN	`lui     at,0` `lw      a0,0(at)`	
lw $4,BEGIN+24	`lui     at,0` `lw      a0,24(at)`	

lw $4,BEGIN($5)	`lui     at,0`
	`addu    at,at,a1`
	`lw      a0,0(at)`	

lw $4,BEGIN+24($5)	`lui     at,0`
	`addu    at,at,a1`
	`lw      a0,24(at)`
	`nop`	

lwl $4,($5)	`lwl     a0,a1,0` `nop`	The `lwl` instruction follows a format identical to the `lb` instruction.
lwl $4,24	`lwl     a0,zero,24`	

lwl $4,2097156	`lui     at,0x20`
	`lwl     a0,at,4`
	`nop`	

lwl $4,24($5)	`lwl     a0,a1,24`	

lwl $4,2097156($5)	`lui     at,0x20`
	`addu    at,at,a1`
	`lwl     a0,at,4`	

lwl $4,BEGIN	`lui     at,0` `lwl     a0,at,0`	
lwl $4,BEGIN+24	`lui     at,0` `lwl     a0,at,24`	

lwl $4,BEGIN($5)	`lui     at,0`
	`addu    at,at,a1`
	`lwl     a0,at,0`	

lwl $4,BEGIN+24($5)	`lui     at,0`
	`addu    at,at,a1`
	`lwl     a0,at,24`
	`nop`	

lwr $4,($5)	`lwr     a0,a1,0` `nop`	The `lwr` instruction follows a format identical to the `lb` instruction.
lwr $4,24	`lwr     a0,zero,24`	

Assembler Input	Machine Language Output	Comments

`lwr    $4,2097156`	`lui    at,0x20`
	`lwr    a0,at,4`
	`nop`	

`lwr    $4,24($5)`	`lwr    a0,a1,24`	

`lwr    $4,2097156($5)`	`lui    at,0x20`
	`addu   at,at,a1`
	`lwr    a0,at,4`	

`lwr    $4,BEGIN`	`lui    at,0` `lwr    a0,at,0`	
`lwr    $4,BEGIN+24`	`lui    at,0` `lwr    a0,at,24`	

`lwr    $4,BEGIN($5)`	`lui    at,0`
	`addu   at,at,a1`
	`lwr    a0,at,0`	

`lwr    $4,BEGIN+24($5)`	`lui    at,0`
	`addu   at,at,a1`
	`lwr    a0,at,24`
	`nop`	

`ld     $4,($5)`	`lw     a0,0(a1)` `lw     a1,4(a1)`	The `ld` instruction does not exist on the Mips M/500 and is implemented with two `lw` instructions.
`ld     $4,24`		The assembler generates no code for this instruction, and issues no warning message. We cannot find any reason why this should be the case.

`ld     $4,2097160`	`lui    at,0x20`
	`lw     a1,12(at)`
	`lw     a0,8(at)`
	`nop`
		The implementation of this instruction is clever. Since the full 32 bits of the absolute address need to be loaded, the assembler reorganizer loads the high-order 16 bits with the `lui` instruction, and this accounts for the low-order 16 bits in the offsets presented to the `lw` instructions.

`ld     $4,24($5)`	`lw     a0,24(a1)` `lw     a1,28(a1)`	

`ld     $4,2097156($5)`	`lui    at,0x20`
	`addu   at,at,a1`
	`lw     a1,8(at)`
	`lw     a0,4(at)`
	`nop`	

`ld     $4,BEGIN`	`lui    at,0`
	`lw     a1,4(at)`
	`lw     a0,0(at)`
	`nop`	

`ld     $4,BEGIN+24`	`lui    at,0`
	`lw     a1,28(at)`
	`lw     a0,24(at)`
	`nop`	

Assembler Input	Machine Language Output	Comments

ld $4,BEGIN($5)	`lui` `at,0`
	`addu` `at,at,a1`
	`lw` `a1,4(at)`
	`lw` `a0,0(at)`
	`nop`	

ld $4,BEGIN+24($5)	`lui` `at,0`
	`addu` `at,at,a1`
	`lw` `a1,28(at)`
	`lw` `a0,24(at)`
	`nop`	

ulh $4,($5)	`lb` `a0,0(a1)`
	`lbu` `at,1(a1)`
	`sll` `a0,a0,8`
	`or` `a0,a0,at`
		The `ulh` instruction loads a halfword irrespective of the alignment of the source address. It must therefore load the bytes of the halfword independently and shift-and-or the results to the destination. Thus a simple MIPS instruction is expanded to 400% of its original size.

ulh $4,24	`lb` `a0,24(zero)`
	`lbu` `at,25(zero)`
	`sll` `a0,a0,8`
	`or` `a0,a0,at`
		This is suboptimal code, since the assembler can determine that the absolute expression `24` is halfword-aligned. This should simply emit a `lh` instruction.

ulh $4,2097156	`lui` `at,0x20`
	`addiu` `at,at,4`
	`lb` `a0,0(at)`
	`lbu` `at,1(at)`
	`sll` `a0,a0,8`
	`or` `a0,a0,at`
		Suboptimal code (see above)

ulh $4,24($5)	`lb` `a0,24(a1)`
	`lbu` `at,25(a1)`
	`sll` `a0,a0,8`
	`or` `a0,a0,at`	

ulh $4,2097156($5)	`lui` `at,0x20`
	`addu` `at,at,a1`
	`addiu` `at,at,4`
	`lb` `a0,0(at)`
	`lbu` `at,1(at)`
	`sll` `a0,a0,8`
	`or` `a0,a0,at`	

ulh $4,BEGIN	`lui` `at,0`
	`lb` `a0,0(at)`
	`lbu` `at,1(at)`
	`sll` `a0,a0,8`
	`or` `a0,a0,at`	

ulh $4,BEGIN+24	`lui` `at,0`
	`lb` `a0,24(at)`
	`lbu` `at,25(at)`
	`sll` `a0,a0,8`
	`or` `a0,a0,at`	

Assembler Input	Machine Language Output	Comments

ulh $4,BEGIN($5)	`lui   at,0`
	`addu  at,at,a1`
	`lb    a0,0(at)`
	`lbu   at,1(at)`
	`sll   a0,a0,8`
	`or    a0,a0,at`	

ulh $4,BEGIN+24($5)	`lui   at,0`
	`addu  at,at,a1`
	`lb    a0,24(at)`
	`lbu   at,25(at)`
	`sll   a0,a0,8`
	`or    a0,a0,at`	

ulhu $4,($5)	`lbu   a0,0(a1)`
	`lbu   at,1(a1)`
	`sll   a0,a0,8`
	`or    a0,a0,at`
		The `ulhu` instruction follows a format identical to the `ulh` instruction, except that is uses `lbu` instructions instead of `lb` instructions.

ulhu $4,24	`lbu   a0,24(zero)`
	`lbu   at,25(zero)`
	`sll   a0,a0,8`
	`or    a0,a0,at`	

ulhu $4,2097156	`lui   at,0x20`
	`addiu at,at,4`
	`lbu   a0,0(at)`
	`lbu   at,1(at)`
	`sll   a0,a0,8`
	`or    a0,a0,at`	

ulhu $4,24($5)	`lbu   a0,24(a1)`
	`lbu   at,25(a1)`
	`sll   a0,a0,8`
	`or    a0,a0,at`	

ulhu $4,2097156($5)	`lui   at,0x20`
	`addu  at,at,a1`
	`addiu at,at,4`
	`lbu   a0,0(at)`
	`lbu   at,1(at)`
	`sll   a0,a0,8`
	`or    a0,a0,at`	

ulhu $4,BEGIN	`lui   at,0`
	`lbu   a0,0(at)`
	`lbu   at,1(at)`
	`sll   a0,a0,8`
	`or    a0,a0,at`	

ulhu $4,BEGIN+24	`lui   at,0`
	`lbu   a0,24(at)`
	`lbu   at,25(at)`
	`sll   a0,a0,8`
	`or    a0,a0,at`	

Assembler Input	*Machine Language Output*		*Comments*

ulhu $4,BEGIN($5)	`lui`
	`addu`
	`lbu`
	`lbu`
	`sll`
	`or`
		`at,0`
		`at,at,a1`
		`a0,0(at)`
		`at,1(at)`
		`a0,a0,8`
		`a0,a0,at`	

ulhu $4,BEGIN+24($5)	`lui`
	`addu`
	`lbu`
	`lbu`
	`sll`
	`or`
		`at,0`
		`at,at,a1`
		`a0,24(at)`
		`at,25(at)`
		`a0,a0,8`
		`a0,a0,at`	

ulw $4,($5)	`lwl`
	`lwr`
	`nop`
		`a0,a1,0`
		`a0,a1,3`
			Although the expansion for this instruction appears wrong, it is correct. The `ulw` instruction is supposed to load a word from memory irrespective of its byte alignment. If the source address is word-aligned, then the `lwl` and `lwr` instructions will load the same memory address twice. If, however, the source address is not word aligned, the two instructions will each load a part of the source word.

ulw $4,24	`lwl` `lwr`	`a0,zero,24` `a0,zero,27`	This is suboptimal code, since the assembler can determine that the absolute expression `24` is word aligned. This should simply emit an `lw` instruction.

ulw $4,2097156	`lui`
	`addiu`
	`lwl`
	`lwr`
	`nop`
		`at,0x20`
		`at,at,4`
		`a0,at,0`
		`a0,at,3`
			Suboptimal code (see above)

ulw $4,24($5)	`lwl` `lwr`	`a0,a1,24` `a0,a1,27`	

ulw $4,2097156($5)	`lui`
	`addu`
	`addiu`
	`lwl`
	`lwr`
		`at,0x20`
		`at,at,a1`
		`at,at,4`
		`a0,at,0`
		`a0,at,3`	

ulw $4,BEGIN	`lui`
	`lwl`
	`lwr`
		`at,0`
		`a0,at,0`
		`a0,at,3`	

ulw $4,BEGIN+24	`lui`
	`lwl`
	`lwr`
		`at,0`
		`a0,at,24`
		`a0,at,27`	

ulw $4,BEGIN($5)	`lui`
	`addu`
	`lwl`
	`lwr`
		`at,0`
		`at,at,a1`
		`a0,at,0`
		`a0,at,3`	

Assembler Input	*Machine Language Output*		*Comments*
ulw $4,BEGIN+24($5)	**lui** **addu** **lwl** **lwr** **nop**	**at,0** **at,at,a1** **a0,at,24** **a0,at,27**	
li $4,24	**li**	**a0,24**	The `li` instruction simply loads an immediate value.
li $4,2097156	**lui** **ori**	**a0,0x20** **a0,a0,0x4**	If the source of the `li` instruction is larger than 16 bits, the assembler breaks it up into two instructions.
lui $4,24	**lui**	**a0,0x18**	The `li` instruction simply loads an immediate value.
sb $4,($5)	**sb**	**a0,0(a1)**	The `sb` instruction follows a format identical to the `lb` instruction.
sb $4,24	**sb**	**a0,24(zero)**	
sb $4,2097156	**lui** **sb**	**at,0x20** **a0,4(at)**	
sb $4,24($5)	**sb**	**a0,24(a1)**	
sb $4,2097156($5)	**lui** **addu** **sb**	**at,0x20** **at,at,a1** **a0,4(at)**	
sb $4,BEGIN	**lui** **sb**	**at,0** **a0,0(at)**	
sb $4,BEGIN+24	**lui** **sb**	**at,0** **a0,24(at)**	
sb $4,BEGIN($5)	**lui** **addu** **sb**	**at,0** **at,at,a1** **a0,0(at)**	
sb $4,BEGIN+24($5)	**lui** **addu** **sb**	**at,0** **at,at,a1** **a0,24(at)**	
sd $4,($5)	**sw** **sw**	**a0,0(a1)** **a1,4(a1)**	The `sd` instruction does not exist on the MIPS M/500 and is implemented with two `sw` instructions. The `sd` instruction follows a format identical to the `ld` instruction.
sd $4,24			The assembler generates no code for this instruction and issues no warning message. We cannot find any reason why this should be the case.
sd $4,2097160	**lui** **sw** **sw**	**at,0x20** **a0,8(at)** **a1,12(at)**	
sd $4,24($5)	**sw** **sw**	**a0,24(a1)** **a1,28(a1)**	

Assembler Input	Machine Language Output		Comments
sd $4,2097156($5)	lui	at,0x20	
	addu	at,at,a1	
	sw	a0,4(at)	
	sw	a1,8(at)	
sd $4,BEGIN	lui	at,0	
	sw	a0,0(at)	
	sw	a1,4(at)	
sd $4,BEGIN+24	lui	at,0	
	sw	a0,24(at)	
	sw	a1,28(at)	
sd $4,BEGIN($5)	lui	at,0	
	addu	at,at,a1	
	sw	a0,0(at)	
	sw	a1,4(at)	
sd $4,BEGIN+24($5)	lui	at,0	
	addu	at,at,a1	
	sw	a0,24(at)	
	sw	a1,28(at)	
sh $4,($5)	sh	a0,0(a1)	The sh instruction follows a format identical to the lh instruction.
sh $4,24	sh	a0,24(zero)	
sh $4,2097156	lui	at,0x20	
	sh	a0,4(at)	
sh $4,24($5)	sh	a0,24(a1)	
sh $4,2097156($5)	lui	at,0x20	
	addu	at,at,a1	
	sh	a0,4(at)	
sh $4,BEGIN	lui	at,0	
	sh	a0,0(at)	
sh $4,BEGIN+24	lui	at,0	
	sh	a0,24(at)	
sh $4,BEGIN($5)	lui	at,0	
	addu	at,at,a1	
	sh	a0,0(at)	
sh $4,BEGIN+24($5)	lui	at,0	
	addu	at,at,a1	
	sh	a0,24(at)	
swl $4,($5)	swl	a0,a1,0	The swl instruction follows a format identical to the lwl instruction.
swl $4,24	swl	a0,zero,24	
swl $4,2097156	lui	at,0x20	
	swl	a0,at,4	
swl $4,24($5)	swl	a0,a1,24	

Assembler Input		Machine Language Output		Comments
swl	$4,2097156($5)	lui	at,0x20	
		addu	at,at,a1	
		swl	a0,at,4	
swl	$4,BEGIN	lui	at,0	
		swl	a0,at,0	
swl	$4,BEGIN+24	lui	at,0	
		swl	a0,at,24	
swl	$4,BEGIN($5)	lui	at,0	
		addu	at,at,a1	
		swl	a0,at,0	
swl	$4,BEGIN+24($5)	lui	at,0	
		addu	at,at,a1	
		swl	a0,at,24	
swr	$4,($5)	swr	a0,a1,0	The swr instruction follows a format identical to the lwl instruction.
swr	$4,24	swr	a0,zero,24	
swr	$4,2097156	lui	at,0x20	
		swr	a0,at,4	
swr	$4,24($5)	swr	a0,a1,24	
swr	$4,2097156($5)	lui	at,0x20	
		addu	at,at,a1	
		swr	a0,at,4	
swr	$4,BEGIN	lui	at,0	
		swr	a0,at,0	
swr	$4,BEGIN+24	lui	at,0	
		swr	a0,at,24	
swr	$4,BEGIN($5)	lui	at,0	
		addu	at,at,a1	
		swr	a0,at,0	
swr	$4,BEGIN+24($5)	lui	at,0	
		addu	at,at,a1	
		swr	a0,at,24	
sw	$4,($5)	sw	a0,0(a1)	The sw instruction follows a format identical to the lw instruction.
sw	$4,24	sw	a0,24(zero)	
sw	$4,2097156	lui	at,0x20	
		sw	a0,4(at)	
sw	$4,24($5)	sw	a0,24(a1)	
sw	$4,2097156($5)	lui	at,0x20	
		addu	at,at,a1	
		sw	a0,4(at)	
sw	$4,BEGIN	lui	at,0	
		sw	a0,0(at)	

Assembler Input		*Machine Language Output*		*Comments*
sw	$4,BEGIN+24	lui	at,0	
		sw	a0,24(at)	
sw	$4,BEGIN($5)	lui	at,0	
		addu	at,at,a1	
		sw	a0,0(at)	
sw	$4,BEGIN+24($5)	lui	at,0	
		addu	at,at,a1	
		sw	a0,24(at)	
ush	$4,($5)	sb	a0,1(a1)	The ush instruction follows a format identical to the ulh instruction, except that the ush instruction uses the store-shift-store method.
		srl	at,a0,8	
		sb	at,0(a1)	
ush	$4,24	sb	a0,25(zero)	This is suboptimal code, since the assembler can determine that the absolute expression 24 is halfword-aligned. This should simply emit an sh instruction.
		srl	at,a0,8	
		sb	at,24(zero)	
ush	$4,2097156	lui	at,0x20	Suboptimal code (see above and below)
		addiu	at,at,4	
		sb	a0,1(at)	
		srl	a0,a0,8	
		sb	at,0(at)	
		lbu	at,1(at)	
		sll	a0,a0,8	
		or	a0,a0,at	
ush	$4,24($5)	sb	a0,25(a1)	
		srl	at,a0,8	
		sb	at,24(a1)	
ush	$4,2097156($5)	lui	at,0x20	This code is a classic example of a reason not to dedicate a single temporary register to an assembler/reorganizer, and an argument for putting reorganization into the compiler. This instruction uses at as a temporary register in the calculation of the destination address. However, since the single temporary register is in use for that purpose, it must destructively shift a0 to the right to perform both sb instructions. It must then re-shift a0 to the left, and re-load the previously stored value to reconstruct the original value in a0. If this value is never used again, three instructions are wasted (and as it is, a single MIPS instruction gets expanded to nine times its original size). For a discussion of this and other deleterious effects of the reorganizer, see Chapter 7.
		addu	at,at,a1	
		addiu	at,at,4	
		sb	a0,1(at)	
		srl	a0,a0,8	
		sb	at,0(at)	
		lbu	at,1(at)	
		sll	a0,a0,8	
		or	a0,a0,at	

Assembler Input		_Machine Language Output_		_Comments_
ush	$4,BEGIN	lui sb srl sb lbu sll or	at,0 a0,1(at) a0,a0,8 at,0(at) at,1(at) a0,a0,8 a0,a0,at	Suboptimal, see above.
ush	$4,BEGIN+24	lui sb srl sb lbu sll or	at,0 a0,25(at) a0,a0,8 at,24(at) at,25(at) a0,a0,8 a0,a0,at	Suboptimal, see above.
ush	$4,BEGIN($5)	lui addu sb srl sb lbu sll or	at,0 at,at,a1 a0,1(at) a0,a0,8 at,0(at) at,1(at) a0,a0,8 a0,a0,at	Suboptimal, see above.
ush	$4,BEGIN+24($5)	lui addu sb srl sb lbu sll or	at,0 at,at,a1 a0,25(at) a0,a0,8 at,24(at) at,25(at) a0,a0,8 a0,a0,at	Suboptimal, see above.
usw	$4,($5)	swl swr	a0,a1,0 a0,a1,3	The usw instruction follows a format identical to the ulw instruction.
usw	$4,24	swl swr	a0,zero,24 a0,zero,27	
usw	$4,2097156	lui addiu swl swr	at,0x20 at,at,4 a0,at,0 a0,at,3	
usw	$4,24($5)	swl swr	a0,a1,24 a0,a1,27	
usw	$4,2097156($5)	lui addu addiu swl swr	at,0x20 at,at,a1 at,at,4 a0,at,0 a0,at,3	
usw	$4,BEGIN	lui swl swr	at,0 a0,at,0 a0,at,3	

Assembler Input		*Machine Language Output*		*Comments*
usw	$4,BEGIN+24	lui	at,0	
		swl	a0,at,24	
		swr	a0,at,27	
usw	$4,BEGIN($5)	lui	at,0	
		addu	at,at,a1	
		swl	a0,at,0	
		swr	a0,at,3	
usw	$4,BEGIN+24($5)	lui	at,0	
		addu	at,at,a1	
		swl	a0,at,24	
		swr	a0,at,27	
		nop		
abs	$4	bgez	a0,0xc	The MIPS high-level assembler has an abs instruction, but there is no corresponding instruction in the machine language. Instead, the assembler reorganizer translates the abs instruction into a test, branch, and negate triplet. This causes a 3:1 increase in execution time for this instruction. Statistically, this increase is of small significance, since the abs instruction is rarely used in compiled code.
		nop		
		sub	a0,zero,a0	
abs	$4,$5	bgez	a1,0xc	As shown in Figure 3-4 on page 8, the move instruction has been shifted down to fill the nop after the bgez instruction. The move is always executed, whether or not the branch is taken.
		move	a0,a1	
		sub	a0,zero,a1	
abs	$4,$0	bgez	zero,0xc	The absolute value of zero is obviously zero, so that while this code expansion is correct, it would be more reasonable to change it to move a0,zero.
		move	a0,zero	
		sub	a0,zero,zero	
neg	$4	sub	a0,zero,a0	The MIPS M/500 does not have a negate instruction but performs this operation by subtracting the number from zero. Depending on whether a signed or unsigned negate is desired, a sub or subu instruction is used. Since the cycle count for this operation is still 1, there is no sacrifice in execution speed.
neg	$4,$5	sub	a0,zero,a1	
neg	$4,$0	sub	a0,zero,zero	The negative of 0 is still 0. This instruction could be replaced with move a0,zero, although its current form is no more expensive to execute.
negu	$4	subu	a0,zero,a0	
negu	$4,$5	subu	a0,zero,a1	

Assembler Input	*Machine Language Output*	*Comments*
negu $4,$0	**subu a0,zero,zero**	The negative of 0 is still 0. This instruction could be replaced with move a0,zero, although its current form is no more expensive to execute.
not $4	**nor a0,a0,zero**	The MIPS M/500 does not have a complement instruction but performs this operation by executing a nor with 0. Since the cycle count for this instruction is still 1, there is no sacrifice in execution speed.
not $4,$5	**nor a0,a1,zero**	
not $0	**nor zero,zero,zero**	The zero register as a destination is meaningless. This instruction should be elided or replaced with a nop instruction.
not $4,$0	**nor a0,zero,zero**	This expansion makes sense, especially when considered as the fastest way to load a register full of ones.
add $4,$5	**add a0,a0,a1**	
add $4,$5,$6	**add a0,a1,a2**	
add $4,$5,$0	**add a0,a1,zero**	
add $4,$5,0	**addi a0,a1,0**	
add $4,$0	**add a0,a0,zero**	This instruction sequence does nothing, and should be elided by the assembler reorganizer.
add $4,0	**addi a0,a0,0**	This instruction sequence does nothing, and should be elided by the assembler reorganizer.
add $4,$0,$5	**add a0,zero,a1**	This instruction could be replaced by a move a0,a1. However, performing the add incurs no extra expense.
add $4,$5,15	**addi a0,a1,15**	
add $4,$5,2097153	**lui at,0x20** **ori at,at,0x1** **add a0,a1,at**	The MIPS M/500 native instruction set limits the size of immediate operands to 16 bits. Therefore, when a large constant value is needed, it is loaded in 16-bit halves. The lui instruction loads the upper half of the register (clearing the lower half), while the ori instruction OR's in the lower half.
add $4,$5,2097152	**lui at,0x20** **add a0,a1,at**	When an immediate operand is larger than 16 bits long, but the bottom 16 bits are zeroes, the assembler reorganizer never generates the ori instruction.
addu $4,$5	**addu a0,a0,a1**	
addu $4,$5,$6	**addu a0,a1,a2**	

Assembler Input	Machine Language Output	Comments
addu $4,$5,$0	**move** a0,a1	There is no actual move instruction on the MIPS M/500. Instead, the assembler allows it as a pseudo-instruction, and encodes it as an addu with zero. The disassembler also knows of this mapping, which accounts for the translation shown here.
addu $4,$5,0	**addiu** a0,a1,0	
addu $4,$0	**move** a0,a0	This instruction sequence clearly does nothing and should be elided by the assembler reorganizer.
addu $4,0	**addiu** a0,a0,0	This instruction sequence does nothing and should be elided by the assembler reorganizer.
addu $4,$0,$5	**addu** a0,zero,a1	This instruction could be replaced by a move a0,a1. However, performing the add incurs no extra expense.
addu $4,$5,15	**addiu** a0,a1,15	
addu $4,$5,2097153	**lui** at,0x20 **ori** at,at,0x1 **addu** a0,a1,at	
and $4,$5	**and** a0,a0,a1	
and $4,$5,$6	**and** a0,a1,a2	
and $4,$5,$0	**and** a0,a1,zero	
and $4,$5,0	**andi** a0,a1,0	
and $4,$0	**and** a0,a0,zero	This could be replaced by move a0,zero. Keeping the and instruction, however, incurs no extra expense.
and $4,0	**andi** a0,a0,0	This could be replaced by move a0,zero. Keeping the andi instruction, however, incurs no extra expense. Notice, however, how the assembler reorganizer again treats the constant value 0 and the zero register differently.
and $4,$0,$5	**and** a0,zero,a1	This could also be replaced by move a0,zero. Keeping the and instruction, however, incurs no extra expense.
and $4,$5,15	**andi** a0,a1,0xf	
and $4,$5,2097153	**lui** at,0x20 **ori** at,at,0x1 **and** a0,a1,at	

Assembler Input	Machine Language Output	Comments
`and $4,$5,2097152`	`lui    at,0x20` `and    a0,a1,at`	

`div    $4,$5`	`div    a0,a1`
	`bne    a1,zero,0x10`
	`nop`
	`break  7`
	`li     at,-1`
	`bne    a1,at,0x28`
	`lui    at,0x8000`
	`bne    a0,at,0x28`
	`nop`
	`break  6`
	`mflo   a0`
	`nop`
	`nop`
		This expansion is a little complicated in that the overflow checking advertized in the documentation is done at run-time by the software and not by the MIPS M/500 `div` instruction. The first test is for division by zero, with a branch to the `break 7` if this is the case. The second test is for division of the largest negative number by -1 (effectively taking the absolute value of the largest negative number). Since there is one more negative number than positive number in two's complement arithmetic, this would be an overflow condition, so the code tests for it and branches to the `break 6` if this is the case.

`div    $4,$5,$6`	`div    a1,a2`
	`bne    a2,zero,0x10`
	`nop`
	`break  7`
	`li     at,-1`
	`bne    a2,at,0x28`
	`lui    at,0x8000`
	`bne    a1,at,0x28`
	`nop`
	`break  6`
	`mflo   a0`
	`nop`
	`nop`	

`div    $4,$5,$0`	`div    a1,zero`
	`bne    zero,zero,0x10`
	`nop`
	`break  7`
	`li     at,-1`
	`bne    zero,at,0x28`
	`lui    at,0x8000`
	`bne    a1,at,0x28`
	`nop`
	`break  6`
	`mflo   a0`
	`nop`
	`nop`
		Even though this instruction is performing a divide by zero (by using the zero register, which always contains the constant value 0), the assembler reorganizer does not issue an error message. The error will still be detected at run-time, though, so this translation is legal, though suboptimal.

`div    $4,$5,0`	`break  7`	The assembler reorganizer here correctly detects a divide by zero and simply generates a `break 7` instruction (which traps to an error handler at run-time), rather than actually generating a sequence of instructions that will divide by zero.

Assembler Input	Machine Language Output	Comments

`div $4,$0,$5`	`div   zero,a1`
	`bne   a1,zero,0x64`
	`nop`
	`break 7`
	`li    at,-1`
	`bne   a1,at,0x7c`
	`lui   at,0x8000`
	`bne   zero,at,0x7c`
	`nop`
	`break 6`
	`mflo  a0`
	`nop`
	`nop`
		The assembler reorganizer fails to recognize that a dividend of zero always results in a quotient of zero, unless the divisor is also zero. The code here could be correspondingly shortened and sped up (through the elimination of the `div` instruction).

`div $4,$0`	`div   a0,zero`
	`bne   zero,zero,0x98`
	`nop`
	`break 7`
	`li    at,-1`
	`bne   zero,at,0xb0`
	`lui   at,0x8000`
	`bne   a0,at,0xb0`
	`nop`
	`break 6`
	`mflo  a0`
	`nop`
	`nop`
		Even though this instruction is performing a divide by zero (by using the zero register, which always contains the constant value 0), the assembler reorganizer does not issue an error message. The error will still be detected at run-time, so this translation is legal, though suboptimal.

`div $4,0`	`break 7`	The assembler reorganizer here correctly detects a divide by zero and simply generates a `break 7` instruction (which traps to an error handler at run-time), rather than actually generating a sequence of instructions that will divide by 0.

`div $4,$5,15`	`li    at,15`
	`div   a1,at`
	`mflo  a0`
	`nop`
	`nop`	

Assembler Input	*Machine Language Output*		*Comments*
div $4,$5,2097152	bgez	a1,0x10	Notice that division by a power of two is accomplished by simply arithmetically shifting the source register to the right. If the source register is negative, then it is decremented by 1 prior to shifting to insure correct results (without the decrementation, -5 >> 1 yields -3, although -5 / 2 = -2). Notice also that the effects of the move instruction are canceled if the branch is not taken (remember that the move executes before the bgez completes), but that the move instruction is necessary if the branch *is* taken. Contrast this behavior with that of the divu instruction on page 163.
	move	at,a1	
	lui	at,0x20	
	addiu	at,a1,-1	
	sra	a0,at,21	
div $4,$5,2097153	lui	at,0x20	
	ori	at,at,0x1	
	div	a1,at	
	mflo	a0	
	nop		
	nop		
divu $4,$5	divu	a0,a1	
	bne	a1,zero,0x10	
	nop		
	break	7	
	mflo	a0	
	nop		
	nop		
divu $4,$5,$6	divu	a1,a2	
	bne	a2,zero,0x10	
	nop		
	break	7	
	mflo	a0	
	nop		
	nop		
divu $4,$5,$0	divu	a1,zero	Even though this instruction is performing a divide by zero (by using the zero register, which always contains the constant value 0), the assembler reorganizer does not issue an error message. The error will still be detected at run-time, so this translation is legal, though suboptimal.
	bne	zero,zero,0x10	
	nop		
	break	7	
	mflo	a0	
	nop		
	nop		
divu $4,$5,0	break	7	The assembler reorganizer here correctly detects a divide by zero, and simply generates a break 7 instruction (which traps to an error handler at run-time), rather than actually generating a sequence of instructions that will divide by 0.

Assembler Input	*Machine Language Output*		*Comments*
divu $4,$0,$5	**divu** **bne** **nop** **break** **mflo** **nop** **nop**	**zero,a1** **a1,zero,0xd0** **7** **a0**	The assembler reorganizer fails to recognize that a dividend of zero always results in a quotient of zero, unless the divisor is also zero. The code here could be correspondingly shortened and sped up (through the elimination of the div instruction).
divu $4,$0	**divu** **bne** **nop** **break** **mflo** **nop** **nop**	**a0,zero** **zero,zero,0xec** **7** **a0**	Even though this instruction is performing a divide by zero (by using the zero register, which always contains the constant value 0), the assembler reorganizer does not issue an error message. The error will still be detected at run-time, so this translation is legal, though suboptimal.
divu $4,0	**break**	**7**	The assembler reorganizer here correctly detects a divide by zero and simply generates a break 7 instruction (which traps to an error handler at run-time), rather than actually generating a sequence of instructions that will divide by 0.
divu $4,$5,15	**li** **divu** **mflo** **nop** **nop**	**at,15** **a1,at** **a0**	
divu $4,$5,2097152	**srl**	**a0,a1,21**	Notice that division by a power of two is accomplished by shifting the source to the right. There is no check for negative numbers here as there was with the div instruction on page 162. This is because the divu instruction is designed to operate only on unsigned (i.e., positive) numbers.
divu $4,$5,2097153	**lui** **ori** **divu** **mflo** **nop** **nop**	**at,0x20** **at,at,0x1** **a1,at** **a0**	
xor $4,$5	**xor**	**a0,a0,a1**	
xor $4,$5,$6	**xor**	**a0,a1,a2**	
xor $4,$5,$0	**xor**	**a0,a1,zero**	This instruction sequence is equivalent to move a0,a1. However, since both instructions take a single cycle to execute, there is no penalty at run-time.

Assembler Input	Machine Language Output	Comments
`xor    $4,$5,0`	**`xori   a0,a1,0`**	This instruction sequence is equivalent to `move a0,a1`. However, since both instructions take a single cycle to execute, there is no penalty at run-time.
`xor    $4,$0,$5`	**`xor    a0,zero,a1`**	
`xor    $4,$0`	**`xor    a0,a0,zero`**	This instruction complements `a0`, and could also have been written as `nor a0,a0,zero`.
`xor    $4,0`	**`xori   a0,a0,0`**	
`xor    $4,$5,15`	**`xori   a0,a1,0xf`**	
`xor    $4,$5,2097153`	**`lui    at,0x20`** **`ori    at,at,0x1`** **`xor    a0,a1,at`**	
`mul    $4,$5`	**`multu  a0,a1`** **`mflo   a0`** **`nop`** **`nop`**	
`mul    $4,$5,$6`	**`multu  a1,a2`** **`mflo   a0`** **`nop`** **`nop`**	
`mul    $4,$5,$0`	**`multu  a1,zero`** **`mflo   a0`** **`nop`** **`nop`**	While the assembler reorganizer is smart enough to recognize that a multiply by a constant value zero produces a zero result, it does not correctly handle the case of multiplication by the zero register, and instead causes the multiplication to be needlessly executed. This is the case for all types of multiply instructions.
`mul    $4,$5,0`	**`move   a0,zero`**	
`mul    $4,$0,$5`	**`multu  zero,a1`** **`mflo   a0`** **`nop`** **`nop`**	The assembler reorganizer should code this as `move a0,zero`, instead of consuming many cycles performing a multiplication by zero.
`mul    $4,$0`	**`multu  a0,zero`** **`mflo   a0`** **`nop`** **`nop`**	The assembler reorganizer should code this as `move a0,zero`, instead of consuming many cycles performing a multiplication by zero.
`mul    $4,0`	**`move   a0,zero`**	
`mul    $4,$5,15`	**`sll    a0,a1,4`** **`subu   a0,a0,a1`**	Multiplication by a constant is converted into a sequence of shifts and adds (or subtracts). See Section 3.2.1 for more details.

Assembler Input		*Machine Language Output*		*Comments*
mul	$4,$5,2097152	**sll**	**a0,a1,21**	The mul instruction is substantially faster than the mulo instruction (see page 166), since it does not have to check for overflow (the sll instruction used here does not register a numeric overflow).
mul	$4,$5,2097153	**sll** **addu**	**a0,a1,21** **a0,a0,a1**	The mul instruction is substantially faster than the mulo instruction (see page 166), since it does not have to check for overflow (the sll instruction used here does not register a numeric overflow).
mulo	$4,$5	**mult** **mflo** **sra** **mfhi** **beq** **mflo** **break** **nop**	**a0,a1** **a0** **a0,a0,31** **at** **a0,at,0x1c** **a0** **6**	
mulo	$4,$5,$6	**mult** **mflo** **sra** **mfhi** **beq** **mflo** **break** **nop**	**a1,a2** **a0** **a0,a0,31** **at** **a0,at,0x1c** **a0** **6**	
mulo	$4,$5,$0	**mult** **mflo** **sra** **mfhi** **beq** **mflo** **break** **nop**	**a1,zero** **a0** **a0,a0,31** **at** **a0,at,0x1c** **a0** **6**	While the assembler reorganizer is smart enough to recognize that a multiply by a constant value zero produces a zero result, it does not correctly handle the case of multiplication by the zero register, and instead causes the multiplication to be needlessly executed. This is the case for all types of multiply instructions.
mulo	$4,$5,0	**move**	**a0,zero**	
mulo	$4,$0,$5	**mult** **mflo** **sra** **mfhi** **beq** **mflo** **break** **nop**	**zero,a1** **a0** **a0,a0,31** **at** **a0,at,0x1c** **a0** **6**	The assembler reorganizer should code this as move a0,zero, instead of consuming many cycles performing a multiplication by zero.

Assembler Input	*Machine Language Output*		*Comments*
`mulo    $4,$5,15`	`add`	`a0,a1,a1`	Note that this sequence of instructions allows for the overflow checking described in the documentation (since the `add` instruction can signal an overflow condition). Contrast this with the multiplication by a constant using the `mul` instruction on page 165. Also, see Section 3.2.1 for a more detailed analysis of multiplication instruction expansion.
	`add`	`a0,a0,a1`	
	`add`	`a0,a0,a0`	
	`add`	`a0,a0,a1`	
	`add`	`a0,a0,a0`	
	`add`	`a0,a0,a1`	
`mulo    $4,$5,2097153`	`lui`	`at,0x20`	
	`ori`	`at,at,0x1`	
	`mult`	`a1,at`	
	`mflo`	`a0`	
	`sra`	`a0,a0,31`	
	`mfhi`	`at`	
	`beq`	`a0,at,0x24`	
	`mflo`	`a0`	
	`break`	`6`	
	`nop`		
`mulou   $4,$5`	`multu`	`a0,a1`	
	`mfhi`	`at`	
	`beq`	`at,zero,0x14`	
	`mflo`	`a0`	
	`break`	`6`	
	`nop`		
`mulou   $4,$5,$6`	`multu`	`a1,a2`	
	`mfhi`	`at`	
	`beq`	`at,zero,0x14`	
	`mflo`	`a0`	
	`break`	`6`	
	`nop`		
`mulou   $4,$5,$0`	`multu`	`a1,zero`	While the assembler reorganizer is smart enough to recognize that a multiply by a constant value zero produces a zero result, it does not correctly handle the case of multiplication by the zero register, and instead causes the multiplication to be needlessly executed. This is the case for all types of multiply instructions.
	`mfhi`	`at`	
	`beq`	`at,zero,0x14`	
	`mflo`	`a0`	
	`break`	`6`	
	`nop`		
`mulou   $4,$5,0`	`move`	`a0,zero`	
`mulou   $4,$0,$5`	`multu`	`zero,a1`	The assembler reorganizer should code this as `move a0,zero`, instead of consuming many cycles performing a multiplication by zero.
	`mfhi`	`at`	
	`beq`	`at,zero,0x14`	
	`mflo`	`a0`	
	`break`	`6`	
	`nop`		

Assembler Input	*Machine Language Output*		*Comments*
mulou $4,$0	multu	a0,zero	The assembler reorganizer should code this as move a0,zero, instead of consuming many cycles performing a multiplication by zero.
	mfhi	at	
	beq	at,zero,0x14	
	mflo	a0	
	break	6	
	nop		
mulou $4,0	move	a0,zero	
mulou $4,$5,15	li	at,15	
	multu	a1,at	
	mfhi	at	
	beq	at,zero,0x18	
	mflo	a0	
	break	6	
	nop		
mulou $4,$5,2097153	lui	at,0x20	
	ori	at,at,0x1	
	multu	a1,at	
	mfhi	at	
	beq	at,zero,0x1c	
	mflo	a0	
	break	6	
	nop		
nor $4,$5	nor	a0,a0,a1	
nor $4,$5,$6	nor	a0,a1,a2	
nor $4,$5,$0	nor	a0,a1,zero	
nor $4,$5,0	ori	a0,a1,0	The assembler reorganizer fails to recognize the special case of a nor with a constant value 0, and generates one extra instruction here. The correct behavior would be to simply perform a nor a0,a1,zero.
	nor	a0,a0,zero	
nor $4,$4	nor	a0,a0,a0	
nor $4,$0,$5	nor	a0,zero,a1	
nor $4,$0	nor	a0,a0,zero	
nor $4,0	ori	a0,a0,0	The assembler reorganizer fails to recognize the special case of a nor with a constant value 0, and generates one extra instruction here. The correct behavior would be to simply perform a nor a0,a0,zero.
	nor	a0,a0,zero	

Assembler Input	Machine Language Output	Comments
`nor $4,$5,15`	`ori   a0,a1,0xf` `nor   a0,a0,zero`	The assembler reorganizer breaks the simple `nor` instruction into two instructions (an `ori` and a `nor`). Since the MIPS M/500 native instruction set has a `nor` in its repertoire, we can conclude that either the assembler reorganizer is making a mistake here or that the native instruction set is not orthogonal, and that the `nor` instruction cannot be executed with an immediate operand.

`nor $4,$5,2097153`	`lui   at,0x20`
	`ori   at,at,0x1`
	`nor   a0,a1,at`	

`or $4,$5`	`or    a0,a0,a1`	
`or $4,$5,$6`	`or    a0,a1,a2`	
`or $4,$5,$0`	`or    a0,a1,zero`	An `or` with zero could easily be be translated into `move a0,a1`, but since there is no additional overhead in not doing that, the assembler reorganizer is behaving appropriately. Where the source and destination registers are identical, the `or` can be deleted entirely in this case, the assembler reorganizer fails to recognize this shortcut.
`or $4,$5,0`	`ori   a0,a1,0`	
`or $4,$0,$5`	`or    a0,zero,a1`	This could also be translated into `move a0,a1`, with no greater or lesser run-time expense.
`or $4,$0`	`or    a0,a0,zero`	This instruction does nothing and should be elided by the assembler reorganizer.
`or $4,0`	`ori   a0,a0,0`	This instruction also does nothing, and should be elided by the assembler reorganizer.
`or $4,$5,15`	`ori   a0,a1,0xf`	

`or $4,$5,2097153`	`lui   at,0x20`
	`ori   at,at,0x1`
	`or    a0,a1,at`
		The `ori` instruction is to load in the lower half of the constant 2097153. The `or` instruction performs the actual work.

Assembler Input	Machine Language Output	Comments

`rem    $4,$5`	`div    a0,a1`
	`bne    a1,zero,0x10`
	`nop`
	`break  7`
	`li     at,-1`
	`bne    a1,at,0x28`
	`lui    at,0x8000`
	`bne    a0,at,0x28`
	`nop`
	`break  6`
	`mfhi   a0`
	`nop`
	`nop`	

`rem    $4,$5,$6`	`div    a1,a2`
	`bne    a2,zero,0x10`
	`nop`
	`break  7`
	`li     at,-1`
	`bne    a2,at,0x28`
	`lui    at,0x8000`
	`bne    a1,at,0x28`
	`nop`
	`break  6`
	`mfhi   a0`
	`nop`
	`nop`	

`rem    $4,$5,$0`	`div    a1,zero`
	`bne    zero,zero,0x10`
	`nop`
	`break  7`
	`li     at,-1`
	`bne    zero,at,0x28`
	`lui    at,0x8000`
	`bne    a1,at,0x28`
	`nop`
	`break  6`
	`mfhi   a0 nop`
	`nop`
		Even though this instruction is performing a divide by zero (by using the zero register, which always contains the constant value 0), the assembler reorganizer does not issue an error message. The error will still be detected at run-time, so this translation is legal, though suboptimal.

`rem    $4,$5,0`	`break  7`	The assembler reorganizer here correctly detects a divide by zero and simply generates a `break 7` instruction (which traps to an error handler at run-time), rather than actually generating a sequence of instructions that will divide by 0.

Assembler Input	*Machine Language Output*		*Comments*

rem $4,$0,$5	`div`
	`bne`
	`nop`
	`break`
	`li`
	`bne`
	`lui`
	`bne`
	`nop`
	`break`
	`mfhi`
	`nop`
	`nop`
		`zero,a1`
		`a1,zero,0x10`
		
		`7`
		`at,-1`
		`a1,at,0x28`
		`at,0x8000`
		`zero,at,0x28`
		
		`6`
		`a0`
			This instruction should be recoded much more simply, since a division does not need to be performed when the dividend of a remainder operation is zero.

rem $4,$0	`div`
	`bne`
	`nop`
	`break`
	`li`
	`bne`
	`lui`
	`bne`
	`nop`
	`break`
	`mfhi`
	`nop`
	`nop`
		`a0,zero`
		`zero,zero,0x10`
		
		`7`
		`at,-1`
		`zero,at,0x28`
		`at,0x8000`
		`a0,at,0x28`
		
		`6`
		`a0`
			This instruction should be recoded much more simply, since a division does not need to be performed when the dividend of a remainder operation is zero.

rem $4,0	`break`	`7`	The assembler reorganizer here correctly detects a divide by zero and simply generates a `break 7` instruction (which traps to an error handler at run-time), rather than actually generating a sequence of instructions that will divide by 0.

rem $4,$5,15	`li`
	`div`
	`mfhi`
	`nop`
	`nop`
		`at,15`
		`a1,at`
		`a0`	

rem $4,$5,2097152	`lui`
	`addiu`
	`bgez`
	`and`
	`beq`
	`addiu`
	`subu`
		`at,0x20`
		`at,at,-1`
		`a1,0x1c`
		`a0,a1,at`
		`a0,zero,0x1c`
		`at,at,1`
		`a0,a0,at`	

rem $4,$5,2097153	`lui`
	`ori`
	`div`
	`mfhu`
	`nop`
	`nop`
		`at,0x20`
		`at,at,0x1`
		`a1,at`
		`a0`	

Assembler Input	*Machine Language Output*		*Comments*
`remu    $4,$5`	`divu`	`a0,a1`	
	`bne`	`a1,zero,0x10`	
	`nop`		
	`break`	`7`	
	`mfhi`	`a0`	
	`nop`		
	`nop`		
`remu    $4,$5,$6`	`divu`	`a1,a2`	
	`bne`	`a2,zero,0x10`	
	`nop`		
	`break`	`7`	
	`mfhi`	`a0`	
	`nop`		
	`nop`		
`remu    $4,$5,$0`	`divu`	`a1,zero`	Even though this instruction is performing a divide by zero (by using the zero register, which always contains the constant value 0), the assembler reorganizer does not issue an error message.
	`bne`	`zero,zero,0x10`	
	`nop`		
	`break`	`7`	
	`mfhi`	`a0`	
	`nop`		
	`nop`		
`remu    $4,$5,0`	`break`	`7`	The assembler reorganizer here correctly detects a divide by zero and simply generates a `break 7` instruction (which traps to an error handler at run-time), rather than actually generating a sequence of instructions that will divide by 0.
`remu    $4,$0,$5`	`divu`	`zero,a1`	This instruction should be recoded much more simply, since a division does not need to be performed when the dividend of a remainder operation is zero.
	`bne`	`a1,zero,0x10`	
	`nop`		
	`break`	`7`	
	`mfhi`	`a0`	
	`nop`		
	`nop`		
`remu    $4,$0`	`divu`	`a0,zero`	This instruction should be recoded much more simply, since a division does not need to be performed when the dividend of a remainder operation is zero.
	`bne`	`zero,zero,0x10`	
	`nop`		
	`break`	`7`	
	`mfhi`	`a0`	
	`nop`		
	`nop`		
`remu    $4,0`	`break`	`7`	The assembler reorganizer here correctly detects a divide by zero and simply generates a `break 7` instruction (which traps to an error handler at run-time), rather than actually generating a sequence of instructions that will divide by 0.

Assembler Input	Machine Language Output	Comments

`remu    $4,$5,15`	`li      at,15`
	`divu    a1,at`
	`mfhi    a0`
	`nop`
	`nop`	

`remu    $4,$5,2097152`	`lui     at,0x20`
	`addiu   at,at,-1`
	`and     a0,a1,at`	

`remu    $4,$5,2097153`	`lui     at,0x20`
	`ori     at,at,0x1`
	`divu    a1,at`
	`mfhi    a0`
	`nop`
	`nop`	

`rol     $4,$5`	`subu    at,zero,a1`
	`srlv    at,a0,at`
	`sllv    a0,a0,a1`
	`or      a0,a0,at`
		The MIPS M/500 native instruction set does not have a rotate instruction. What the assembler reorganizer does is to rotate the source register both right and left and merge the result into the destination register. For a `rol` instruction, the source word is logically (not arithmetically) rotated right by the *negative* of the rotation amount. Since the native instruction set specifies that the shift amount is taken *modulo* 32, this translates into a shift right by the correct number of bits. The same register is then rotated left by the specified amount, and the results are merged together with an `or` instruction.

`rol     $4,$5,$6`	`subu    at,zero,a2`
	`srlv    at,a1,at`
	`sllv    a0,a1,a2`
	`or      a0,a0,at`	

`rol     $4,$5,$0`	`subu    at,zero,zero`
	`srlv    at,a1,at`
	`sllv    a0,a1,zero`
	`or      a0,a0,at`
		The assembler reorganizer should translate this instruction to a `move $4,$5`, since a rotation by zero bits is no rotation at all. Instead, it incorrectly generates the superfluous rotation code.

`rol     $4,0`		This instruction does not assemble at all and generates the assembler run-time error *"(fimmed >= 0) and (fimmed <= 31)"* from *../as1emit.p*, line 588. The correct action would be to ignore this instruction. MIPS Inc. claims that this bug is fixed in a newer release of the assembler.

Assembler Input	*Machine Language Output*		*Comments*
rol $4,$5,0			This instruction does not assemble at all and generates the assembler run-time error *"(fimmed >= 0) and (fimmed <= 31)"* from *../as1emit.p*, line 588. The correct action would be to ignore this instruction.
rol $4,$0,$5	**subu**	**at,zero,a1**	This instruction should be recoded as move a0,zero, since rotating zero by any number of bits (especially zero bits) still yields zero.
	srlv	**at,zero,at**	
	sllv	**a0,zero,a1**	
	or	**a0,a0,at**	
rol $4,$0	**subu**	**at,zero,zero**	This instruction should be recoded as move a0,zero, since rotating zero by any number of bits (especially zero bits) still yields zero.
	srlv	**at,a0,at**	
	sllv	**a0,a0,zero**	
	or	**a0,a0,at**	
rol $4,$5,15	**sll**	**at,a1,15**	
	srl	**a0,a1,17**	
	or	**a0,a0,at**	
rol $4,$5,2097153			This instruction generates the assembly error *"Shift amount not 0..31"*. While this is reasonable enough, the documentation maintains that shift amounts outside the range of 0..31 are taken *modulo 32* before shifting, thus implying that this line of code would be legal.
ror $4,$5	**subu**	**at,zero,a1**	See note for rol instruction.
	sllv	**at,a0,at**	
	srlv	**a0,a0,a1**	
	or	**a0,a0,at**	
ror $4,$5,$6	**subu**	**at,zero,a2**	
	sllv	**at,a1,at**	
	srlv	**a0,a1,a2**	
	or	**a0,a0,at**	
ror $4,$5,$0	**subu**	**at,zero,zero**	The assembler reorganizer should translate this instruction to a move $4,$5, since a rotation by zero bits is no rotation at all. Instead it incorrectly generates the superfluous rotation code.
	sllv	**at,a1,at**	
	srlv	**a0,a1,zero**	
	or	**a0,a0,at**	
ror $4,0			This instruction does not assemble at all and generates the assembler run-time error *"(fimmed >= 0) and (fimmed <= 31)"* from *../as1emit.p*, line 588. The correct action would be to ignore this instruction.
ror $4,$5,0			This instruction does not assemble at all generates the assembler run-time error *"(fimmed >= 0) and (fimmed <= 31)"* from *../as1emit.p*, line 588. The correct action would be to ignore this instruction.

Assembler Input		*Machine Language Output*		*Comments*
ror	$4,$0,$5	subu sllv srlv or	at,zero,a1 at,zero,at a0,zero,a1 a0,a0,at	This instruction should be recoded as move a0,zero, since rotating zero by any number of bits (especially zero bits) still yields zero.
ror	$4,$0	subu sllv srlv or	at,zero,zero at,a0,at a0,a0,zero a0,a0,at	This instruction should be recoded as move a0,zero, since rotating zero by any number of bits (especially zero bits) still yields zero.
ror	$4,$5,15	srl sll or	at,a1,15 a0,a1,17 a0,a0,at	
ror	$4,$5,2097153			This instruction generates the assembly error *"Shift amount not 0..31."* While this is reasonable enough, the documentation maintains that shift amounts outside of the range of 0..31 are taken *modulo 32* before shifting, thus implying that this line of code would be legal.
seq	$4,$5	xor sltiu	a0,a0,a1 a0,a0,1	The MIPS M/500 native instruction set does not have an seq instruction, so it is faked with two other instructions, effectively doubling the execution time of this opcode.
seq	$4,$5,$6	xor sltiu	a0,a1,a2 a0,a0,1	
seq	$4,$5,$0	xor sltiu	a0,a1,zero a0,a0,1	The assembler reorganizer once again misses the fact that the zero register is functionally equivalent to the constant value zero.
seq	$4,$5,0	sltiu	a0,a1,1	
seq	$4,$0	xor sltiu	a0,a0,zero a0,a0,1	The assembler reorganizer once again misses the fact that the zero register is functionally equivalent to the constant value zero.
seq	$4,0	sltiu	a0,a0,1	
seq	$4,$0,$5	xor sltiu	a0,zero,a1 a0,a0,1	
seq	$4,$5,15	xori sltiu	a0,a1,0xf a0,a0,1	
seq	$4,$5,2097153	lui ori xor sltiu	at,0x20 at,at,0x1 a0,a1,at a0,a0,1	
slt	$4,$5	slt	a0,a0,a1	
slt	$4,$5,$6	slt	a0,a1,a2	
slt	$4,$5,$0	slt	a0,a1,zero	

Assembler Input	Machine Language Output	Comments
slt $4,$5,0	**slti a0,a1,0**	
slt $4,$0	**slt a0,a0,zero**	
slt $4,0	**slti a0,a0,0**	
slt $4,$0,$5	**slt a0,zero,a1**	
slt $4,$5,15	**slti a0,a1,15**	
slt $4,$5,2097153	**lui at,0x20** **ori at,at,0x1** **slt a0,a1,at**	
sltu $4,$5	**sltu a0,a0,a1**	
sltu $4,$5,$6	**sltu a0,a1,a2**	
sltu $4,$5,$0	**sltu a0,a1,zero**	
sltu $4,$5,0	**sltiu a0,a1,0**	
sltu $4,$0	**sltu a0,a0,zero**	
sltu $4,0	**sltiu a0,a0,0**	
sltu $4,$0,$5	**sltu a0,zero,a1**	
sltu $4,$5,15	**sltiu a0,a1,15**	
sltu $4,$5,2097153	**lui at,0x20** **ori at,at,0x1** **sltu a0,a1,at**	
sle $4,$5	**slt a0,a1,a0** **xori a0,a0,0x1**	The MIPS M/500 native instruction set does not have an sle instruction, so it is faked with two other instructions, effectively doubling the execution time of this opcode. We would like to point out that other architectures usually require several instructions to set condition codes and test them. The scheme that MIPS uses is actually better, in spite of the occasional code expansion.
sle $4,$5,$6	**slt a0,a2,a1** **xori a0,a0,0x1**	
sle $4,$5,$0	**slt a0,zero,a1** **xori a0,a0,0x1**	The assembler reorganizer once again misses the fact that the zero register is functionally equivalent to the constant value zero.
sle $4,$5,0	**slti a0,a1,1**	
sle $4,$0	**slt a0,zero,a0** **xori a0,a0,0x1**	The assembler reorganizer once again misses the fact that the zero register is functionally equivalent to the constant value zero.
sle $4,0	**slti a0,a0,1**	
sle $4,$0,$5	**slt a0,a1,zero** **xori a0,a0,0x1**	

Assembler Input		*Machine Language Output*		*Comments*
sle	$4,$5,15	**slti**	**a0,a1,16**	
sle	$4,$5,2097153	**lui** **ori** **slt**	**at,0x20** **at,at,0x2** **a0,a1,at**	
sleu	$4,$5	**sltu** **xori**	**a0,a1,a0** **a0,a0,0x1**	The Mips M/500 native instruction set does not have an sleu instruction, so it is faked with two other instructions, effectively doubling the execution time of this opcode.
sleu	$4,$5,$6	**sltu** **xori**	**a0,a2,a1** **a0,a0,0x1**	
sleu	$4,$5,$0	**sltu** **xori**	**a0,zero,a1** **a0,a0,0x1**	The assembler reorganizer once again misses the fact that the zero register is functionally equivalent to the constant value zero.
sleu	$4,$5,0	**sltiu**	**a0,a1,1**	
sleu	$4,$0	**sltu** **xori**	**a0,zero,a0** **a0,a0,0x1**	The assembler reorganizer once again misses the fact that the zero register is functionally equivalent to the constant value zero.
sleu	$4,0	**sltiu**	**a0,a0,1**	
sleu	$4,$0,$5	**sltu** **xori**	**a0,a1,zero** **a0,a0,0x1**	
sleu	$4,$5,15	**sltiu**	**a0,a1,16**	
sleu	$4,$5,2097153	**lui** **ori** **sltu**	**at,0x20** **at,at,0x2** **a0,a1,at**	
sgt	$4,$5	**slt**	**a0,a1,a0**	The Mips M/500 native instruction set does not have an sgt instruction, so it is faked with an slt instruction with reversed operands at no extra cost.
sgt	$4,$5,$6	**slt**	**a0,a2,a1**	
sgt	$4,$5,$0	**slt**	**a0,zero,a1**	
sgt	$4,$5,0	**slt**	**a0,zero,a1**	
sgt	$4,$0	**slt**	**a0,zero,a0**	
sgt	$4,0	**slt**	**a0,zero,a0**	
sgt	$4,$0,$5	**slt**	**a0,a1,zero**	
sgt	$4,$5,15	**li** **slt**	**at,15** **a0,at,a1**	
sgt	$4,$5,2097153	**lui** **ori** **slt**	**at,0x20** **at,at,0x1** **a0,at,a1**	

Assembler Input	Machine Language Output	Comments
sgtu $4,$5	**sltu** **a0,a1,a0**	The MIPS M/500 native instruction set does not have an sgtu instruction, so it is faked with an sltu instruction with reversed operands at no extra cost.
sgtu $4,$5,$6	**sltu** **a0,a2,a1**	
sgtu $4,$5,$0	**sltu** **a0,zero,a1**	
sgtu $4,$5,0	**sltu** **a0,zero,a1**	
sgtu $4,$0	**sltu** **a0,zero,a0**	
sgtu $4,0	**sltu** **a0,zero,a0**	
sgtu $4,$0,$5	**sltu** **a0,a1,zero**	
sgtu $4,$5,15	**li** **at,15** **sltu** **a0,at,a1**	
sgtu $4,$5,2097153	**lui** **at,0x20** **ori** **at,at,0x1** **sltu** **a0,at,a1**	
sge $4,$5	**slt** **a0,a0,a1** **xori** **a0,a0,0x1**	The MIPS M/500 native instruction set does not have an sge instruction, so it is faked with two other instructions, effectively doubling the execution time of this opcode.
sge $4,$5,$6	**slt** **a0,a1,a2** **xori** **a0,a0,0x1**	
sge $4,$5,$0	**slt** **a0,a1,zero** **xori** **a0,a0,0x1**	
sge $4,$5,0	**slti** **a0,a1,0** **xori** **a0,a0,0x1**	
sge $4,$0	**slt** **a0,a0,zero** **xori** **a0,a0,0x1**	
sge $4,0	**slti** **a0,a0,0** **xori** **a0,a0,0x1**	
sge $4,$0,$5	**slt** **a0,zero,a1** **xori** **a0,a0,0x1**	
sge $4,$5,15	**slti** **a0,a1,15** **xori** **a0,a0,0x1**	
sge $4,$5,2097153	**lui** **at,0x20** **ori** **at,at,0x1** **slt** **a0,a1,at** **xori** **a0,a0,0x1**	
sgeu $4,$5	**sltu** **a0,a0,a1** **xori** **a0,a0,0x1**	The MIPS M/500 native instruction set does not have an sgeu instruction, so it is faked with two other instructions, effectively doubling the execution time of this opcode.

Assembler Input		*Machine Language Output*		*Comments*
sgeu	$4,$5,$6	sltu	a0,a1,a2	
		xori	a0,a0,0x1	
sgeu	$4,$5,$0	sltu	a0,a1,zero	All unsigned numbers are greater than 0, so this instruction should simply expand to xori a0,a0,0x1. Instead, it is expanded to a code sequence that, while functionally correct, takes twice as long to execute.
		xori	a0,a0,0x1	
sgeu	$4,$5,0	sltiu	a0,a1,0	Suboptimal code (see above).
		xori	a0,a0,0x1	
sgeu	$4,$0	sltu	a0,a0,zero	Suboptimal code (see above).
		xori	a0,a0,0x1	
sgeu	$4,0	sltiu	a0,a0,0	Suboptimal code (see above).
		xori	a0,a0,0x1	
sgeu	$4,$0,$5	sltu	a0,zero,a1	Suboptimal code (see above).
		xori	a0,a0,0x1	
sgeu	$4,$5,15	sltiu	a0,a1,15	
		xori	a0,a0,0x1	
sgeu	$4,$5,2097153	lui	at,0x20	
		ori	at,at,0x1	
		sltu	a0,a1,at	
		xori	a0,a0,0x1	
sne	$4,$5	xor	a0,a0,a1	Not only does the MIPS M/500 native instruction set does not have an sne instruction (so that it fakes it with three other instructions, effectively tripling the execution time of this opcode) but it generates the wrong code sequence! What should be generated is xor a0,a0,a1 followed by sltu a0,zero,a0, which takes only two cycles to execute.
		sltiu	a0,a0,1	
		xori	a0,a0,0x1	
sne	$4,$5,$6	xor	a0,a1,a2	Suboptimal code (see above).
		sltiu	a0,a0,1	
		xori	a0,a0,0x1	
sne	$4,$5,$0	xor	a0,a1,zero	The assembler reorganizer once again misses the fact that the zero register is functionally equivalent to the constant value zero. It is also generating suboptimal code (see above).
		sltiu	a0,a0,1	
		xori	a0,a0,0x1	
sne	$4,$5,0	sltiu	a0,a1,1	Suboptimal code (see above).
		xori	a0,a0,0x1	
sne	$4,$0	xor	a0,a0,zero	Suboptimal code (see above).
		sltiu	a0,a0,1	
		xori	a0,a0,0x1	
sne	$4,0	sltiu	a0,a0,1	Suboptimal code (see above).
		xori	a0,a0,0x1	

Assembler Input		*Machine Language Output*		*Comments*
sne	$4,$0,$5	**xor**	**a0,zero,a1**	Suboptimal code (see above).
		sltiu	**a0,a0,1**	
		xori	**a0,a0,0x1**	
sne	$4,$5,15	**xori**	**a0,a1,0xf**	Suboptimal code (see above).
		sltiu	**a0,a0,1**	
		xori	**a0,a0,0x1**	
sne	$4,$5,2097153	**lui**	**at,0x20**	Suboptimal code (see above).
		ori	**at,at,0x1**	
		xor	**a0,a1,at**	
		sltiu	**a0,a0,1**	
		xori	**a0,a0,0x1**	
sll	$4,$5	**sllv**	**a0,a0,a1**	
sll	$4,$5,$6	**sllv**	**a0,a1,a2**	
sll	$4,$5,$0	**sllv**	**a0,a1,zero**	This instruction could be substituted with a simple move instruction, since a shift of zero bits is no shift at all. However, since both instructions take one cycle, there is no extra incurred expense.
sll	$4,$5,0	**sll**	**a0,a1,0**	
sll	$4,$0	**sllv**	**a0,a0,zero**	
sll	$4,0	**sll**	**a0,a0,0**	
sll	$4,$0,$5	**sllv**	**a0,zero,a1**	This instruction should be recoded as move a0,zero, since rotating zero by any number of bits (especially zero) still yields zero.
sll	$4,$5,15	**sll**	**a0,a1,15**	
sll	$4,$5,2097153			This instruction generates the assembly error *"Shift amount not 0..31."* While this is reasonable enough, the documentation maintains that shift amounts outside of the range of 0..31 are taken *modulo 32* before shifting, thus implying that this line of code would be legal.
sra	$4,$5	**srav**	**a0,a0,a1**	
sra	$4,$5,$6	**srav**	**a0,a1,a2**	
sra	$4,$5,$0	**srav**	**a0,a1,zero**	This instruction could be substituted with a simple move instruction, since a shift of zero bits is no shift at all. However, since both instructions take one cycle, there is no extra incurred expense.
sra	$4,$5,0	**sra**	**a0,a1,0**	
sra	$4,$0	**srav**	**a0,a0,zero**	
sra	$4,0	**sra**	**a0,a0,0**	

Assembler Input		*Machine Language Output*		*Comments*
sra	$4,$0,$5	**srav**	**a0,zero,a1**	This instruction should be recoded as move a0,zero, since rotating zero by any number of bits (especially zero) still yields zero.
sra	$4,$5,15	**sra**	**a0,a1,15**	
sra	$4,$5,2097153			This instruction generates the assembly error *"Shift amount not 0..31."* While this is reasonable enough, the documentation maintains that that shift amounts outside of the range of 0..31 are taken *modulo 32* before shifting, thus implying that this line of code would be legal.
srl	$4,$5	**srlv**	**a0,a0,a1**	
srl	$4,$5,$6	**srlv**	**a0,a1,a2**	
srl	$4,$5,$0	**srlv**	**a0,a1,zero**	This instruction could be substituted with a simple move instruction, since a shift of zero bits is no shift at all. However, since both instructions take one cycle, there is no extra incurred expense.
srl	$4,$5,0	**srl**	**a0,a1,0**	
srl	$4,$0	**srlv**	**a0,a0,zero**	
srl	$4,0	**srl**	**a0,a0,0**	
srl	$4,$0,$5	**srlv**	**a0,zero,a1**	This instruction should be recoded as move a0,zero, since rotating zero by any number of bits (especially zero) still yields zero.
srl	$4,$5,15	**srl**	**a0,a1,15**	
srl	$4,$5,2097153			This instruction generates the assembly error *"Shift amount not 0..31."* While this is reasonable enough, the documentation maintains that that shift amounts outside of the range of 0..31 are taken *modulo 32* before shifting, thus implying that this line of code would be legal.
sub	$4,$5	**sub**	**a0,a0,a1**	
sub	$4,$5,$6	**sub**	**a0,a1,a2**	
sub	$4,$5,$0	**sub**	**a0,a1,zero**	This instruction could be substituted with a simple move, since subtracting zero from a number gives that number as a result. However, since both instructions take one cycle, there is no extra incurred expense.

Assembler Input	*Machine Language Output*	*Comments*
sub $4,$5,0	**addi a0,a1,0**	This instruction could be substituted with a simple move, since subtracting zero from a number gives that number as a result. However, since both instructions take one cycle, there is no extra incurred expense.
sub $4,$0	**sub a0,a0,zero**	When both the minuend and the subtrahend of the subtraction are the same, the assembler reorganizer should remove the instruction when the subtrahend is zero. As can be seen, it does not.
sub $4,0	**addi a0,a0,0**	When both the minuend and the subtrahend of the subtraction are the same, the assembler reorganizer should remove the instruction when the subtrahend is zero. As can be seen, it does not.
sub $4,$0,$5	**sub a0,zero,a1**	
sub $4,$5,32767	**addi a0,a1,-32767**	Subtraction of constant values is implemented as the addition of their negative value.
sub $4,$5,32768	**li at,32768** **sub a0,a1,at**	Unfortunately, the assembler reorganizer is not smart enough to recognize that -32768 would be only 16 bits long. What should be generated here is addi a0,a1,-32768.
sub $4,$5,-32767	**addi a0,a1,32767**	The negatives of the values are used for both positive and negative constants.
sub $4,$5,-32768	**li at,-32768** **sub a0,a1,at**	The assembler reorganizer *is* smart enough to know that 32768 is too big for an immediate operand though.
sub $4,$5,15	**addi a0,a1,-15**	The use of the addi instruction instead of the anticipated subi, while entirely legal, suggests a lack of orthogonality of the MIPS M/500 native instruction set. In this case, this is perfectly reasonable (since the MIPS M/500 native architecture is RISC in nature).
sub $4,$5,2097153	**lui at,0x20** **ori at,at,0x1** **sub a0,a1,at**	
subu $4,$5	**subu a0,a0,a1**	
subu $4,$5,$6	**subu a0,a1,a2**	
subu $4,$5,$0	**subu a0,a1,zero**	
subu $4,$5,0	**addiu a0,a1,0**	

Assembler Input	Machine Language Output	Comments
subu $4,$0	**subu a0,a0,zero**	This instruction does nothing and should be elided by the assembler reorganizer.
subu $4,0	**addiu a0,a0,0**	This instruction does nothing and should be elided by the assembler reorganizer.
subu $4,$0,$5	**subu a0,zero,a1**	
subu $4,$5,15	**addiu a0,a1,-15**	
subu $4,$5,2097153	**lui at,0x20** **ori at,at,0x1** **subu a0,a1,at**	
move $4,$5	**move a0,a1**	
mult $4,$5	**mult a0,a1**	
mult $4,$0	**mult a0,zero**	This instruction could be replaced with move a0,zero, but since other instructions may be counting on the contents of the **hi** and **lo** registers afterward, this cannot be done. (The mult instruction is documented as leaving the results of the multiplication in these registers.)
multu $4,$5	**multu a0,a1**	
multu $4,$0	**multu a0,zero**	This instruction could be replaced with move a0,zero, but since other instructions may be counting on the contents of the **hi** and **lo** registers afterwards, this cannot be done. (The multu instruction is documented as leaving the results of the multiplication in these registers.)
b TOP	**b 0** **nop**	The trailing nop instructions that follow each of these condition tests may be filled with an instruction that the assembler reorganizer can move downward.
beq $4,$5,TOP	**beq a0,a1,0** **nop**	
beq $4,0,TOP	**beq a0,zero,0** **nop**	In this case, the assembler reorganizer correctly treats the zero register and the constant value 0 as identical.
beq $4,$0,TOP	**beq a0,zero,0** **nop**	
beq $4,15,TOP	**li at,15** **beq a0,at,0** **nop**	None of the conditional branches supports an immediate operand, so the assembler reorganizer loads the immediate operand into the temporary register at.
beq $4,2097153,TOP	**lui at,0x20** **ori at,at,0x1** **beq a0,at,0** **nop**	

Assembler Input	*Machine Language Output*	*Comments*
bgt $4,$5,TOP	**slt** **at,a1,a0** **bne** **at,zero,0** **nop**	The MIPS M/500 native instruction set does not have a bgt instruction, so it is faked with two other instructions, effectively doubling the execution time of this opcode.
bgt $4,0,TOP	**bgtz** **a0,0** **nop**	In this case, the use of the MIPS M/500 native bgtz instruction keeps the effective execution time in line with the anticipated time.
bgt $4,$0,TOP	**bgtz** **a0,0** **nop**	In this case, the assembler reorganizer correctly treats the zero register and the constant value 0 as identical.
bgt $4,15,TOP	**slti** **at,a0,16** **beq** **at,zero,0** **nop**	None of the conditional branches supports an immediate operand, so the assembler reorganizer loads the immediate operand into the temporary register at.
bgt $4,2097153,TOP	**lui** **at,0x20** **ori** **at,at,0x2** **slt** **at,a0,at** **beq** **at,zero,0** **nop**	
bge $4,$5,TOP	**slt** **at,a0,a1** **beq** **at,zero,0** **nop**	The MIPS M/500 native instruction set does not have a bge instruction, so it is faked with two other instructions, effectively doubling the execution time of this opcode.
bge $4,0,TOP	**bgez** **a0,0** **nop**	In this case, the use of the MIPS M/500 native bgez instruction keeps the effective execution time in line with the anticipated time.
bge $4,$0,TOP	**bgez** **a0,0** **nop**	In this case, the assembler reorganizer correctly treats the zero register and the constant value 0 as identical.
bge $4,15,TOP	**slti** **at,a0,15** **beq** **at,zero,0** **nop**	None of the conditional branches supports an immediate operand, so the assembler reorganizer loads the immediate operand into the temporary register at.
bge $4,2097153,TOP	**lui** **at,0x20** **ori** **at,at,0x1** **slt** **at,a0,at** **beq** **at,zero,0** **nop**	
bgeu $4,$5,TOP	**sltu** **at,a0,a1** **beq** **at,zero,0** **nop**	The MIPS M/500 native instruction set does not have a bgeu instruction, so it is faked with two other instructions, effectively doubling the execution time of this opcode.

Assembler Input	Machine Language Output	Comments
`bgeu $4,0,TOP`	`b       0` `nop`	All numbers are greater than or equal to zero in unsigned comparisons, so the assembler reorganizer has correctly translated the conditional branch into an unconditional branch instruction.
`bgeu $4,$0,TOP`	`b       0` `nop`	In this case, the assembler reorganizer correctly treats the zero register and the constant value 0 as identical.

`bgeu $4,15,TOP`	`sltiu   at,a0,15`
	`beq     at,zero,0`
	`nop`
		None of the conditional branches supports an immediate operand, so the assembler reorganizer loads the immediate operand into the temporary register `at`.

`bgeu $4,2097153,TOP`	`lui     at,0x20`
	`ori     at,at,0x1`
	`sltu    at,a0,at`
	`beq     at,zero,0`
	`nop`	

`bgtu $4,$5,TOP`	`sltu    at,a1,a0`
	`bne     at,zero,0`
	`nop`
		The MIPS M/500 native instruction set does not have a `bgtu` instruction, so it is faked with two other instructions, effectively doubling the execution time of this opcode.

`bgtu $4,0,TOP`	`bne     a0,zero,0` `nop`	In unsigned comparisons, all numbers are either greater than or equal to zero. Since we are concerned with numbers that are greater than zero, the assembler reorganizer tests for not equal to zero, which suffices.
`bgtu $4,$0,TOP`	`bne     a0,zero,0` `nop`	In this case, the assembler reorganizer correctly treats the zero register and the constant value 0 as identical.

`bgtu $4,15,TOP`	`sltiu   at,a0,16`
	`beq     at,zero,0`
	`nop`
		None of the conditional branches supports an immediate operand, so the assembler reorganizer loads the immediate operand into the temporary register `at`.

`bgtu $4,2097153,TOP`	`lui     at,0x20`
	`ori     at,at,0x2`
	`sltu    at,a0,at`
	`beq     at,zero,0`
	`nop`	

`blt $4,$5,TOP`	`slt     at,a0,a1`
	`bne     at,zero,0`
	`nop`
		The MIPS M/500 native instruction set does not have a `blt` instruction, so it is faked with two other instructions, effectively doubling the execution time of this opcode.

`blt $4,0,TOP`	`bltz    a0,0` `nop`	In this case, the use of the MIPS M/500 native `bltz` instruction keeps the effective execution time in line with the anticipated time.

Assembler Input	*Machine Language Output*	*Comments*
blt $4,$0,TOP	**bltz a0,0** **nop**	In this case, the assembler reorganizer correctly treats the zero register and the constant value 0 as identical.
blt $4,15,TOP	**slti at,a0,15** **bne at,zero,0** **nop**	None of the conditional branches supports an immediate operand, so the assembler reorganizer loads the immediate operand into the temporary register at.
blt $4,2097153,TOP	**lui at,0x20** **ori at,at,0x1** **slt at,a0,at** **bne at,zero,0** **nop**	
ble $4,$5,TOP	**slt at,a1,a0** **beq at,zero,0** **nop**	The MIPS M/500 native instruction set does not have a ble instruction, so it is faked with two other instructions, effectively doubling the execution time of this opcode.
ble $4,0,TOP	**blez a0,0** **nop**	In this case, the use of the MIPS M/500 native blez instruction keeps the effective execution time in line with the anticipated time.
ble $4,$0,TOP	**blez a0,0** **nop**	In this case, the assembler reorganizer correctly treats the zero register and the constant value 0 as identical.
ble $4,15,TOP	**slti at,a0,16** **bne at,zero,0** **nop**	None of the conditional branches supports an immediate operand, so the assembler reorganizer loads the immediate operand into the temporary register at.
ble $4,2097153,TOP	**lui at,0x20** **ori at,at,0x2** **slt at,a0,at** **bne at,zero,0** **nop**	
bleu $4,$5,TOP	**sltu at,a1,a0** **beq at,zero,0** **nop**	The MIPS M/500 native instruction set does not have a bleu instruction, so it is faked with two other instructions, effectively doubling the execution time of this opcode.
bleu $4,0,TOP	**beq a0,zero,0** **nop**	In unsigned comparisons, all numbers are either greater than or equal to zero. Since we are concerned with numbers that are less than or equal to zero, the assembler reorganizer tests for equal to zero, which suffices.
bleu $4,$0,TOP	**beq a0,zero,0** **nop**	In this case, the assembler reorganizer correctly treats the zero register and the constant value 0 as identical.

Assembler Input	Machine Language Output	Comments

`bleu    $4,15,TOP`	`sltiu    at,a0,16`
	`bne      at,zero,0`
	`nop`
		None of the conditional branches supports an immediate operand, so the assembler reorganizer loads the immediate operand into the temporary register `at`.

`bleu    $4,2097153,TOP`	`lui      at,0x20`
	`ori      at,at,0x2`
	`sltu     at,a0,at`
	`bne      at,zero,0`
	`nop`	

`bltu    $4,$5,TOP`	`sltu     at,a0,a1`
	`bne      at,zero,0`
	`nop`
		The MIPS M/500 native instruction set does not have a `bltu` instruction, so it is faked with two other instructions, effectively doubling the execution time of this opcode.

`bltu    $4,0,TOP`		This instruction generates no code at all. This is correct behavior, since no number may be less than 0 in an unsigned comparison, so the branch can never be taken. If the branch instruction is addressed by a label, and hence possibly the target of a branch, the assembler reorganizer substitutes a `nop` instruction for the `bltu`.
`bltu    $4,$0,TOP`		This instruction generates no code at all. This is correct behavior, since no number may be less than 0 in an unsigned comparison, so the branch can never be taken. If the branch instruction is addressed by a label, and hence possibly the target of a branch, the assembler reorganizer substitutes a `nop` instruction for the `bltu`. In this case also, the assembler reorganizer correctly treats the zero register and the constant value 0 as identical.

`bltu    $4,15,TOP`	`sltiu    at,a0,15`
	`bne      at,zero,0`
	`nop`
		None of the conditional branches supports an immediate operand, so the assembler reorganizer loads the immediate operand into the temporary register `at`.

`bltu    $4,2097153,TOP`	`lui      at,0x20`
	`ori      at,at,0x1`
	`sltu     at,a0,at`
	`bne      at,zero,0`
	`nop`	

`bne     $4,$5,TOP`	`bne      a0,a1,0` `nop`	
`bne     $4,0,TOP`	`bne      a0,zero,0` `nop`	

Assembler Input	*Machine Language Output*	*Comments*
bne $4,$0,TOP	**bne** **a0,zero,0** **nop**	In this case, the assembler reorganizer correctly treats the zero register and the constant value 0 as identical.
bne $4,15,TOP	**li** **at,15** **bne** **a0,at,0** **nop**	None of the conditional branches supports an immediate operand, so the assembler reorganizer loads the immediate operand into the temporary register at.
bne $4,2097153,TOP	**lui** **at,0x20** **ori** **at,at,0x1** **bne** **a0,at,0** **nop**	
bal TOP	**bgezal** **zero,0** **nop**	Apparently, there is no unconditional branch and link instruction in the Mips M/500 native instruction set, so the assembler reorganizer substitutes the conditional bgezal instruction with an always TRUE condition.
bltzal TOP		According to the documentation, this instruction is legal, but when assembled, generates the error *"Register expected: TOP"*. It would seem that neither the bltzal nor the bgezal instruction functions at all.
bgezal $4		According to the documentation, this instruction is legal, but when assembled, generates the error *"label expected"*. It would seem that neither the bltzal nor the bgezal instruction functions at all.
beqz $4,TOP	**beq** **a0,zero,0** **nop**	
bgez $4,TOP	**bgez** **a0,0** **nop**	
bgtz $4,TOP	**bgtz** **a0,0** **nop**	
blez $4,TOP	**blez** **a0,0** **nop**	
bltz $4,TOP	**bltz** **a0,0** **nop**	
bnez $4,TOP	**bne** **a0,zero,0** **nop**	
j TOP	**j** **0** **nop**	
j $4	**jr** **a0** **nop**	
jal TOP	**jal** **0** **nop**	

Assembler Input		*Machine Language Output*		*Comments*
jal	$4	**jalr** **nop**	**a0**	
break	0	**break**	**0**	
rfe		**c0**	**rfe**	
syscall		**syscall**		
mfhi	$4	**mfhi** **nop** **nop**	**a0**	
mthi	$4	**mthi**	**a0**	
mflo	$4	**mflo** **nop** **nop**	**a0**	
mtlo	$4	**mtlo**	**a0**	
lwc0	$4,ADDR	**lui** **lwc0**	**at,0** **a0,at,7592**	
lwc1	$f4,ADDR	**lui** **lwc1**	**at,0** **f4,7592(at)**	
lwc2	$4,ADDR	**lui** **lwc2**	**at,0** **a0,at,7592**	
lwc3	$4,ADDR	**lui** **lwc3**	**at,0** **a0,at,7592**	
swc0	$4,ADDR	**lui** **swc0**	**at,0** **a0,at,7592**	
swc1	$f4,ADDR	**lui** **swc1**	**at,0** **f4,7592(at)**	
swc2	$4,ADDR	**lui** **swc2**	**at,0** **a0,at,7592**	
swc3	$4,ADDR	**lui** **swc3**	**at,0** **a0,at,7592**	
mfc0	$4,$5	**mfc0** **nop**	**a0,c0r5**	Note that c0r5 refers to coprocessor 0 register 5.
mfc1	$4,$f5	**mfc1** **nop**	**a0,f5**	
mfc1.d	$4,$f6	**mtc1** **mtc1** **nop**	**a1,f6** **a0,f7**	This instruction is undocumented in the *Mips Assembly Language Programmers Guide.* It serves to store a double-precision floating-point number from the floating-point co-processor by performing two single-word store instructions.
mfc2	$4,$5	**c2** **nop**	**a0,zero,10240**	
mfc3	$4,$5	**c3** **nop**	**a0,zero,10240**	

Assembler Input		*Machine Language Output*		*Comments*
mtc0	$4,$5	**mtc0** **nop**	**a0,c0r5**	
mtc1	$4,$f5	**mtc1** **nop**	**a0,f5**	
mtc1.d	$4,$f6	**mtc1** **mtc1** **nop**	**a1,f6** **a0,f7**	This instruction is undocumented in the *Mips Assembly Language Programmers Guide*, but is generated by the compilers. It serves to load a double-precision floating-point number into the floating-point co-processor by performing two single-word load instructions.
mtc2	$4,$5	**c2** **nop**	**a0,a0,10240**	
mtc3	$4,$5	**c3** **nop**	**a0,a0,10240**	
bc0f	TOP	**bc0f** **nop**	**0**	
bc1f	TOP	**bc1f** **nop**	**0**	
bc2f	TOP	**c2** **nop**	**zero,t0,0**	
bc3f	TOP	**c3** **nop**	**zero,t0,0**	
bc0t	TOP	**bc0t** **nop**	**0**	
bc1t	TOP	**bc1t** **nop**	**0**	
bc2t	TOP	**c2** **nop**	**at,t0,0**	
bc3t	TOP	**c3** **nop**	**at,t0,0**	
c0	15	**c0**	**c0op15**	
c1	15	**fop0f.s f0,f0,f0**		The disassembler supplied by MIPS (and used to extract the machine-language output) "knows" that co-processor 1 is the floating-point unit, so it interprets c1 as a floating-point instruction. We are not sure exactly what this instruction is, though.
c2	15	**c2**	**zero,s0,15**	
c3	15	**c3**	**zero,s0,15**	
cfc0	$4,$5			This instruction, although documented, is not recognized by the assembler reorganizer as being legal.

Assembler Input		*Machine Language Output*		*Comments*
cfc1	$4,$5	**cfc1**	**a0,f5**	
		nop		
cfc2	$4,$5			This instruction, although documented, is not recognized by the assembler reorganizer as being legal.
cfc3	$4,$5			This instruction, although documented, is not recognized by the assembler reorganizer as being legal.
ctc0	$4,$5			This instruction, although documented, is not recognized by the assembler reorganizer as being legal.
ctc1	$4,$5	**ctc1**	**a0,f5**	
		nop		
ctc2	$4,$5			This instruction, although documented, is not recognized by the assembler reorganizer as being legal.
ctc3	$4,$5			This instruction, although documented, is not recognized by the assembler reorganizer as being legal.
tlbp		**c0**	**tlbp**	
tlbr		**c0**	**tlbr**	
tlbwr		**c0**	**tlbwr**	
tlbwi		**c0**	**tlbwi**	
nop		**nop**		Although undocumented, this instruction's function should be obvious.
l.s	$f2,TOP	**lui**	**at,0**	In this and all floating-point load/store operations, the instructions that are generated use the lwc1 and swc1 instructions. These instructions use a general address expression for their second operand. Therefore, the assembler reorganizer must generate a load instruction for the **at** register, even if the resultant effective address will be a simple constant value.
		lwc1	**f2,0(at)**	
l.d	$f2,TOP	**lui**	**at,0**	Loading a double-precision number requires two lwc1 instructions to load all 64 bits.
		lwc1	**f2,4(at)**	
		lwc1	**f3,0(at)**	
		nop		
s.s	$f2,TOP	**lui**	**at,0**	
		swc1	**f2,0(at)**	
s.d	$f2,TOP	**lui**	**at,0**	Storing a double-precision number requires two lwc1 instructions to store all 64 bits.
		swc1	**f3,0(at)**	
		swc1	**f2,4(at)**	
abs.s	$f2,$f4	**abs.s**	**f2,f4**	

Assembler Input	*Machine Language Output*	*Comments*
abs.d $f2,$f4	**abs.d f2,f4**	
neg.s $f2,$f4	**neg.s f2,f4,f0**	The neg instruction appears to need an extra register.
neg.d $f2,$f4	**neg.d f2,f4,f0**	The neg instruction appears to need an extra register.
add.s $f2,$f4,$f6	**add.s f2,f4,f6**	
add.d $f2,$f4,$f6	**add.d f2,f4,f6**	
sub.s $f2,$f4,$f6	**sub.s f2,f4,f6**	
sub.d $f2,$f4,$f6	**sub.d f2,f4,f6**	
mul.s $f2,$f4,$f6	**mul.s f2,f4,f6**	
mul.d $f2,$f4,$f6	**mul.d f2,f4,f6**	
div.s $f2,$f4,$f6	**div.s f2,f4,f6**	
div.d $f2,$f4,$f6	**div.d f2,f4,f6**	
cvt.s.d $f2,$f4	**cvt.s.d f2,f4**	
cvt.d.s $f2,$f4	**cvt.d.s f2,f4**	
cvt.w.d $f2,$f4	**cvt.w.d f2,f4**	
cvt.d.w $f2,$f4	**cvt.d.w f2,f4**	
cvt.s.w $f2,$f4	**cvt.s.w f2,f4**	
cvt.w.s $f2,$f4	**cvt.w.s f2,f4**	
trunc.w.s $f2,$f4,$4	**cfc1 a0,f31** **cfc1 a0,f31** **nop** **ori at,a0,0x3** **xori at,at,0x2** **ctc1 at,f31** **nop** **cvt.w.s f2,f4** **ctc1 a0,f31** **nop** **nop** **nop**	Truncation appears to be a rather expensive operation (although the documentation does describe these instructions as being "macro" instructions).
trunc.w.d $f2,$f4,$4	**cfc1 a0,f31** **cfc1 a0,f31** **nop** **ori at,a0,0x3** **xori at,at,0x2** **ctc1 at,f31** **nop** **cvt.w.d f2,f4** **ctc1 a0,f31** **nop** **nop** **nop**	Truncation appears to be a rather expensive operation (although the documentation does describe these instructions as being "macro" instructions).

Assembler Input	Machine Language Output	Comments

`round.w.s $f2,$f4,$4`	`cfc1    a0,f31`
	`cfc1    a0,f31`
	`li      at,-4`
	`and     at,at,a0`
	`ctc1    at,f31`
	`nop`
	`cvt.w.s f2,f4`
	`ctc1    a0,f31`
	`nop`
	`nop`
	`nop`
		Rounding appears to be a rather expensive operation (although the documentation does describe these instructions as being "macro" instructions).

`round.w.d $f2,$f4,$4`	`cfc1    a0,f31`
	`cfc1    a0,f31`
	`li      at,-4`
	`and     at,at,a0`
	`ctc1    at,f31`
	`nop`
	`cvt.w.d f2,f4`
	`ctc1    a0,f31`
	`nop`
	`nop`
	`nop`
		Rounding appears to be a rather expensive operation (although the documentation does describe these instructions as being "macro" instructions).

`c.f.s   $f2,$f4`	`c.f.s   f2,f4` `nop`	The trailing `nop` instructions that follow each of these condition tests may be filled with an instruction that the assembler reorganizer can move downward. Note that the MIPS M/500 native instruction set does not have any floating-point conditional branches *per se*, but instead uses the `bc1t` and `bc1f` instructions (page 189) to branch on the condition codes set by these relational operations.
`c.f.d   $f2,$f4`	`c.f.d   f2,f4` `nop`	
`c.un.s  $f2,$f4`	`c.un.s  f2,f4` `nop`	
`c.un.d  $f2,$f4`	`c.un.d  f2,f4` `nop`	
`c.eq.s  $f2,$f4`	`c.eq.s  f2,f4` `nop`	
`c.eq.d  $f2,$f4`	`c.eq.d  f2,f4` `nop`	
`c.ueq.s $f2,$f4`	`c.ueq.s f2,f4` `nop`	
`c.ueq.d $f2,$f4`	`c.ueq.d f2,f4` `nop`	
`c.olt.s $f2,$f4`	`c.olt.s f2,f4` `nop`	

Assembler Input	Machine Language Output	Comments
c.olt.d $f2,$f4	**c.olt.d f2,f4** **nop**	
c.ult.s $f2,$f4	**c.ult.s f2,f4** **nop**	
c.ult.d $f2,$f4	**c.ult.d f2,f4** **nop**	
c.ole.s $f2,$f4	**c.ole.s f2,f4** **nop**	
c.ole.d $f2,$f4	**c.ole.d f2,f4** **nop**	
c.ule.s $f2,$f4	**c.ule.s f2,f4** **nop**	
c.ule.d $f2,$f4	**c.ule.d f2,f4** **nop**	
c.sf.s $f2,$f4	**c.sf.s f2,f4** **nop**	
c.sf.d $f2,$f4	**c.sf.d f2,f4** **nop**	
c.ngle.s $f2,$f4	**c.ngle.s f2,f4** **nop**	
c.ngle.d $f2,$f4	**c.ngle.d f2,f4** **nop**	
c.seq.s $f2,$f4	**c.seq.s f2,f4** **nop**	
c.seq.d $f2,$f4	**c.seq.d f2,f4** **nop**	
c.ngl.s $f2,$f4	**c.ngl.s f2,f4** **nop**	
c.ngl.d $f2,$f4	**c.ngl.d f2,f4** **nop**	
c.lt.s $f2,$f4	**c.lt.s f2,f4** **nop**	
c.lt.d $f2,$f4	**c.lt.d f2,f4** **nop**	
c.nge.s $f2,$f4	**c.nge.s f2,f4** **nop**	
c.nge.d $f2,$f4	**c.nge.d f2,f4** **nop**	
c.le.s $f2,$f4	**c.le.s f2,f4** **nop**	
c.le.d $f2,$f4	**c.le.d f2,f4** **nop**	

Assembler Input	Machine Language Output	Comments
c.ngt.s $f2,$f4	**c.ngt.s f2,f4** **nop**	
c.ngt.d $f2,$f4	**c.ngt.d f2,f4** **nop**	
mov.s $f2,$f4	**mov.s f2,f4**	
mov.d $f2,$f4	**mov.d f2,f4** **nop**	

Table A-2 (on the following page) provides an alphabetic cross reference of MIPS assembler instructions. The previous table was listed in the instruction order presented in Chapter 5 of the *MIPS Assembly Language Reference Manual* [MIPS 86a]. Table A-2 is supplied to provide an easy mechanism for locating the page number on which instructions are first referenced.

Instruction	Page		Instruction	Page		Instruction	Page		Instruction	Page
abs	157		c.ngl.s	193		lb	145		rfe	188
abs.d	191		c.ngle.d	193		lbu	145		rol	172
abs.s	190		c.ngle.s	193		ld	148		ror	173
add	158		c.ngt.d	194		lh	146		round.w.d	192
add.d	191		c.ngt.s	194		lhu	146		round.w.s	192
add.s	191		c.ole.d	193		li	152		s.d	190
addu	158		c.ole.s	193		lui	152		s.s	190
and	159		c.olt.d	193		lw	146		sb	152
b	182		c.olt.s	192		lwc0	188		sd	152
bal	187		c.seq.d	193		lwc1	188		seq	174
bc0f	189		c.seq.s	193		lwc2	188		sge	177
bc0t	189		c.sf.d	193		lwc3	188		sgeu	177
bc1f	189		c.sf.s	193		lwl	147		sgt	176
bc1t	189		c.ueq.d	192		lwr	147		sgtu	177
bc2f	189		c.ueq.s	192		mfc0	188		sh	153
bc2t	189		c.ule.d	193		mfc1	188		sle	175
bc3f	189		c.ule.s	193		mfc1.d	188		sleu	176
bc3t	189		c.ult.d	193		mfc2	188		sll	179
beq	182		c.ult.s	193		mfc3	188		slt	174
beqz	187		c.un.d	192		mfhi	188		sltu	175
bge	183		c.un.s	192		mflo	188		sne	178
bgeu	183		c0	189		mov.s	194		sra	179
bgez	187		c1	189		mov.d	194		srl	180
bgezal	187		c2	189		move	182		sub	180
bgt	183		c3	189		mtc0	189		sub.d	191
bgtu	184		cfc0	189		mtc1	189		sub.s	191
bgtz	187		cfc1	190		mtc1.d	189		subu	181
ble	185		cfc2	190		mtc2	189		sw	154
bleu	185		cfc3	190		mtc3	189		swc0	188
blez	187		ctc0	190		mthi	188		swc1	188
blt	184		ctc1	190		mtlo	188		swc2	188
bltu	186		ctc2	190		mul	164		swc3	188
bltz	187		ctc3	190		mul.d	191		swl	153
bltzal	187		cvt.d.s	191		mul.s	191		swr	154
bne	186		cvt.d.w	191		mulo	165		syscall	188
bnez	187		cvt.s.d	191		mulou	166		tlbp	190
break	188		cvt.s.w	191		mult	182		tlbr	190
c.eq.d	192		cvt.w.d	191		multu	182		tlbwi	190
c.eq.s	192		cvt.w.s	191		neg	157		tlbwr	190
c.f.d	192		div	160		neg.d	191		trunc.w.d	191
c.f.s	192		div.d	191		neg.s	191		trunc.w.s	191
c.le.d	193		div.s	191		negu	157		ulh	149
c.le.s	193		divu	162		nop	190		ulhu	150
c.lt.d	193		j	187		nor	167		ulw	151
c.lt.s	193		jal	187		not	158		ush	155
c.nge.d	193		l.d	190		or	168		usw	156
c.nge.s	193		l.s	190		rem	169		xor	163
c.ngl.d	193		la	144		remu	171			

Table A-2: Alphabetic Cross Reference of MIPS Assembler Instructions

Tables A-3 and A-4 are a list of the actual hardware instructions supported by the MIPS M/500 and its floating-point co-processor, respectively. They are provided to give the reader a feel for the real instruction set architecture, rather than the pseudo-instructions presented by the assembler reorganizer. Please note that the `nop` and `move` instructions are really just special cases of the `addu` instruction.

add	addi	addiu	addu	and	andi
b	bc0f	bc0t	bc1f	bc1t	beq
bgez	bgezal	bgtz	blez	bltz	bne
break	c0	c2	c3	cfc1	ctc1
div	divu	j	jal	jalr	jr
lb	lbu	lh	lhu	li	lui
lw	lwc0	lwc1	lwc2	lwc3	lwl
lwr	mfc0	mfc1	mfhi	mflo	move
mtc0	mtc1	mthi	mtlo	mult	multu
nop	nor	or	ori	sb	sh
sll	sllv	slt	slti	sltiu	sltu
sra	srav	srl	srlv	sub	subu
sw	swc0	swc1	swc2	swc3	swl
	swr	syscall	xor	xori	

Table A-3: Actual MIPS M/500 Instruction Set

abs.d	abs.s	add.d	add.s	c.eq.d	c.eq.s
c.f.d	c.f.s	c.le.d	c.le.s	c.lt.d	c.lt.s
c.nge.d	c.nge.s	c.ngl.d	c.ngl.s	c.ngle.d	c.ngle.s
c.ngt.d	c.ngt.s	c.ole.d	c.ole.s	c.olt.d	c.olt.s
c.seq.d	c.seq.s	c.sf.d	c.sf.s	c.ueq.d	c.ueq.s
c.ule.d	c.ule.s	c.ult.d	c.ult.s	c.un.d	c.un.s
cvt.d.s	cvt.d.w	cvt.s.d	cvt.s.w	cvt.w.d	cvt.w.s
div.d	div.s	mov.d	mov.s	mul.d	mul.s
	neg.d	neg.s	sub.d	sub.s	

Table A-4: MIPS M/500 Floating-Point Co-Processor Instruction Set

Appendix B: Compiler and Assembler Version Information

The following three tables list the version numbers of the compilers, assembler, and linker used to generate all of the information in this report. The version information was obtained by running the three compilers (C, FORTRAN, and Pascal) with the **-v** switch and no source file. The subcomponents of the compilers and libraries are also listed, and are primarily from Berkeley release software. The compiler components were created at MIPS on January 29, 1987, and installed at the Software Engineering Institute on March 20, 1986. All of the test results describe in this document were obtained after that installation date.

B.1. C Compiler

C Compiler Components	
Compiler Component	*Version Number*
/usr/lib/cpp	**Mips Computer Systems Release 1.10c**
/usr/lib/ccom	**Mips Computer Systems Release 1.10g**
/usr/lib/ujoin	**Mips Computer Systems Release 1.10c**
/usr/bin/uld	**Mips Computer Systems Release 1.10h**
/usr/lib/usplit	**Mips Computer Systems Release 1.10c**
/usr/lib/umerge	**Mips Computer Systems Release 1.10b**
ldopen.c	**1.3 2/16/83**
ldclose.c	**1.3 2/16/83**
vldldptr.c	**1.1 1/8/82**
allocldptr.c	**1.2 2/16/83**
freeldptr.c	**1.1 1/7/82**
/usr/lib/uopt	**Mips Computer Systems Release 1.10e**
/usr/lib/ugen	**Mips Computer Systems Release 1.10j**
ldopen.c	**1.3 2/16/83**
ldclose.c	**1.3 2/16/83**
vldldptr.c	**1.1 1/8/82**
allocldptr.c	**1.2 2/16/83**
freeldptr.c	**1.1 1/7/82**
/usr/lib/as0	**Mips Computer Systems Release 1.10f**
/usr/lib/as1	**Mips Computer Systems Release 1.10f**
/usr/lib/crt0.o	*unknown*

C Compiler Components (contd.)	
`/usr/lib/libc.a`	*unknown*
`/usr/bin/ld`	**Mips Computer Systems Release 1.10h**
`cc`	**Mips Computer Systems 1.10**

B.2. Fortran-77 Compiler

Fortran-77 Compiler Components	
Compiler Component	*Version Number*
`/usr/lib/cpp`	**Mips Computer Systems Release 1.10c**
`/usr/lib/fcom`	**Mips Computer Systems Release 1.10h**
`/usr/lib/ujoin`	**Mips Computer Systems Release 1.10c**
`/usr/bin/uld`	**Mips Computer Systems Release 1.10h**
`/usr/lib/usplit`	**Mips Computer Systems Release 1.10c**
`/usr/lib/umerge`	**Mips Computer Systems Release 1.10b**
`ldopen.c`	**1.3 2/16/83**
`ldclose.c`	**1.3 2/16/83**
`vldldptr.c`	**1.1 1/8/82**
`allocldptr.c`	**1.2 2/16/83**
`freeldptr.c`	**1.1 1/7/82**
`/usr/lib/uopt`	**Mips Computer Systems Release 1.10e**
`/usr/lib/ugen`	**Mips Computer Systems Release 1.10j**
`ldopen.c`	**1.3 2/16/83**
`ldclose.c`	**1.3 2/16/83**
`vldldptr.c`	**1.1 1/8/82**
`allocldptr.c`	**1.2 2/16/83**
`freeldptr.c`	**1.1 1/7/82**
`/usr/lib/as0`	**Mips Computer Systems Release 1.10f**
`/usr/lib/as1`	**Mips Computer Systems Release 1.10f**
`/usr/lib/crt0.o`	*unknown*
`/usr/lib/libc.a`	*unknown*
`/usr/lib/libm.a`	**Mips Computer Systems Release 1.10b**
`pow.c`	**4.5 (Berkeley) 8/21/85**
`support.c`	**1.1 (Berkeley) 5/23/85**

Fortran-77 Compiler Components (contd.)	
cbrt.c	**1.1 (Berkeley) 5/23/85**
cabs.c	**1.2 (Berkeley) 8/21/85**
log__L.c	**1.2 (Berkeley) 8/21/85**
log1p.c	**1.3 (Berkeley) 8/21/85**
exp__E.c	**1.2 (Berkeley) 8/21/85**
expm1.c	**1.2 (Berkeley) 8/21/85**
asinh.c	**1.2 (Berkeley) 8/21/85**
acosh.c	**1.2 (Berkeley) 8/21/85**
atanh.c	**1.2 (Berkeley) 8/21/85**
/usr/lib/libF77.a	**Mips Computer Systems Release 1.10c**
/usr/lib/libI77.a	**Mips Computer Systems Release 1.10d**
/usr/lib/libU77.a	*unknown*
/usr/bin/ld	**Mips Computer Systems Release 1.10h**
f77	**Mips Computer Systems 1.10**

B.3. Pascal Compiler

Pascal Compiler Components	
Compiler Component	*Version Number*
/usr/lib/cpp	**Mips Computer Systems Release 1.10c**
/usr/lib/upas	**Mips Computer Systems Release 1.10e**
/usr/lib/ujoin	**Mips Computer Systems Release 1.10c**
/usr/bin/uld	**Mips Computer Systems Release 1.10h**
/usr/lib/usplit	**Mips Computer Systems Release 1.10c**
/usr/lib/umerge	**Mips Computer Systems Release 1.10b**
ldopen.c	**1.3 2/16/83**
ldclose.c	**1.3 2/16/83**
vldldptr.c	**1.1 1/8/82**
allocldptr.c	**1.2 2/16/83**
freeldptr.c	**1.1 1/7/82**
/usr/lib/uopt	**Mips Computer Systems Release 1.10e**
/usr/lib/ugen	**Mips Computer Systems Release 1.10j**
ldopen.c	**1.3 2/16/83**

Pascal Compiler Components (contd.)

ldclose.c	**1.3 2/16/83**
vldldptr.c	**1.1 1/8/82**
allocldptr.c	**1.2 2/16/83**
freeldptr.c	**1.1 1/7/82**
/usr/lib/as0	**Mips Computer Systems Release 1.10f**
/usr/lib/as1	**Mips Computer Systems Release 1.10f**
/usr/lib/crt0.o	*unknown*
/usr/lib/libc.a	*unknown*
/usr/lib/libp.a	**Mips Computer Systems Release 1.10d**
/usr/lib/libm.a	**Mips Computer Systems Release 1.10b**
pow.c	**4.5 (Berkeley) 8/21/85**
support.c	**1.1 (Berkeley) 5/23/85**
cbrt.c	**1.1 (Berkeley) 5/23/85**
cabs.c	**1.2 (Berkeley) 8/21/85**
log__L.c	**1.2 (Berkeley) 8/21/85**
log1p.c	**1.3 (Berkeley) 8/21/85**
exp__E.c	**1.2 (Berkeley) 8/21/85**
expm1.c	**1.2 (Berkeley) 8/21/85**
asinh.c	**1.2 (Berkeley) 8/21/85**
acosh.c	**1.2 (Berkeley) 8/21/85**
atanh.c	**1.2 (Berkeley) 8/21/85**
/usr/bin/ld	**Mips Computer Systems Release 1.10h**
pc	**Mips Computer Systems 1.10**

Appendix C: Conformance with CORE Instruction Set Architecture

The key evidence that the MIPS machine conforms to CORE ISA [CORE 87] is the existence of a translator from the CORE assembler code to the Mips high-level assembler. However, the closeness with which the MIPS M/500 conforms can be established only by a feature analysis, which is given in this appendix. In all cases, *MIPS* refers to the true instruction set of the MIPS M/500 machine, not to the high-level assembler. The latter is superficially closer to CORE, but we think it appropriate to measure conformance in terms of what the machine actually executes.

C.1. Registers

The CORE ISA allows the machine registers to be represented in two ways: by absolute names and by logical *resource* names.

C.1.1. Absolute Registers
The CORE [Section 2.2.1] requires at least 16 integer registers `(0..15)` and 4 double-precision floating-point registers `(f0..f3)`. The MIPS M/500 provides 27 free integer registers and 32 floating-point (or 16 double-precision floating-point) registers, and so conforms.

C.1.2. Logical Registers
The CORE [Figure 2-3] defines sets of logical registers with specific functions. The MIPS assembler conventions define a very similar set, as shown in the following table:

```
CORE        Mips                        Comment
.sp         sp                 stack pointer
.fp         fp                 frame pointer
.lr         ra                 procedure return link
.fr         v0..v1             function result
.gX         v0..v1             expression evaluation
.aX         a0..a3             argument transmission
.tX         t0..t7             temporaries
.sX         s0..s7             locals (saved across calls)
.gp         gp                 global pointer
.fX         f0..f31            floating-point registers
.z          r0                 zero register
```

In all cases, the MIPS provides at least the minimum required number of each resource type.

C.2. Data Types

The CORE [Section 2.1] specifies byte, halfword, and word integer types, and single- and double-precision floating types. The MIPS M/500 provides all these, and in addition has unsigned byte and halfword types.

The CORE [Section 2.1] requires natural alignment[62] for all data types. MIPS recommends observing this requirement, but in fact permits double-word values to be aligned on word boundaries.

---

[62]Natural alignment means the address of any variable of that type must be an exact multiple of the size of the type.

C.2.1. Integer Operations

The CORE [Section 2.2] requires both overflowing and non-overflowing operations. The MIPS M/500 provides both, except that overflow on division is implemented by a software check. This is a permissible deviation.

The CORE [Section 3.1] requires the following integer operations:

```
abs  add  div  mod  mul  neg  rem  sub
```

The MIPS M/500 provides them in the following manner:

- **abs** is implemented by a conditional branch around a negate.
- **div** and **rem** are implemented as one operation yielding both quotient and remainder.
- **neg** is implemented by subtraction from zero.
- The other instructions are implemented as given in CORE.

C.2.2. Logical Operations

The CORE [Section 3.1.1] requires the following logical operations:

```
and  not  or  xor
```

On the MIPS M/500, **not** is implemented by nor with zero, with the other instructions as in the CORE specification.

C.2.3. Shift Operations

The CORE [Section 3.2] defines the following shift operations:

- **sll** (shift left logical)
- **srl** (shift right logical)
- **sra** (shift right arithmetic)
- **rol** (rotate left)
- **ror** (rotate right)

for single-word operands. The MIPS M/500 implements **sll**, **srl**, and **sra** directly. It expands the rotate instructions into three-instruction sequences, which is not unreasonable given that no common high-level language can generate rotates. The CORE [Section 3.2.2] also requires the same operations with double-word operands. The MIPS assembler does *not* provide these operations; they must be constructed out of the single-word forms.

C.3. Load and Store Operations

The CORE [Section 3.3] defines load, store, and load address instructions. The MIPS M/500 provides all these, and in addition two load immediate instructions (**lui** and **li**), which together allow constants of up to 32 bits to be loaded from the instruction stream. The other operand of all load and store instructions is a register in both CORE and the MIPS M/500.

C.3.1. Addressing Modes

The CORE [Section 3.3.1] requires all addressing modes of the following form:

```
relocatable + absolute  (register)
```

with all three components optional. MIPS provides exactly these modes, but requires the relocated offset to be representable as a signed 16-bit quantity. Many static addresses must therefore be constructed by first loading the upper 16 bits into a temporary register; the defects of this process are discussed in Chapter 7.

The CORE [Section 3.1.1] also requires a register-to-register move, which is provided by the MIPS M/500 `move`, `mov.s`, and `mov.d` instructions.

C.4. Control Transfers

C.4.1. Branch and Jump Instructions

The CORE [Section 3.4.1] requires an unconditional branch and the full set of conditional branches. The MIPS M/500 does not provide this. Instead, it uses a combination of the "set" instructions and the branch on zero/non-zero to construct all possible branch idioms. Defects of this process are shown in Appendix A.

The CORE says nothing about the possible *range* of a branch. The MIPS M/500 provides a signed 16-bit word offset, which should be enough for all but the traditional "pathological cases."

The CORE [Section 3.4.2] also requires a general jump instruction to a destination whose value is held in a register. The MIPS M/500 provides exactly this instruction.

C.4.2. Call Instruction

The CORE [Section 3.4.3] requires a call instruction of the following form:

```
cal target, link
```

where the target can be a label or the contents of a register, and the link can be a register or a based address.

The MIPS M/500 provides three instructions, `bal`, `jal`, and `jalr`, according to whether the target is a label, a general address, or a value in a register. In all cases, the return link is stored into `ra`. However, the next instruction after the call is executed immediately, so that instruction should store the link if necessary. The MIPS M/500 also provides a conditional call instruction, `bgezal`.

C.4.3. Trap Instruction

The CORE [Section 3.4.4] requires a trap instruction that transfers control synchronously to an exception handler with a status code in the range 0..255. The MIPS M/500 provides a break instruction with equivalent functionality.

C.5. Floating-Point Instructions

The CORE [Section 3.5] defines a set of floating-point instructions. The MIPS M/500 defines a set of general co-processor instructions, which in the special case of a floating-point co-processor become floating-point instructions.

C.5.1. Floating-Point Load and Store

The CORE [Section 3.5.1] defines load-and-store operations for both floating data types operating between a general address and a floating register.

MIPS defines all these operations at the higher level. However, the double-precision load and store expand into two single-precision loads and stores. This can create further problems with addressability, as discussed in Section 7.1.

The CORE [Section 3.5.1] also defines loads and stores that perform various conversions and roundings. The MIPS M/500 provides all the required conversions, but only with register operands; these CORE instructions therefore expand into a load and a conversion, or a conversion and a store. This is a reasonable simplification (and probably improves instruction timing predictability).

C.5.2. Floating Operations

The CORE [Section 3.5.2] requires the full following IEEE set of operations:

```
add  sub  mul  div  abs  sqrt
```

for both single and double precision operands. Mips provides the following:

```
add  sub  mul  div  abs  neg
```

It does not provide `sqrt`, which must be implemented by a routine call. This is an understandable simplification, but regrettable.

The CORE [Section 3.5] requires only *round to nearest* to be provided. The MIPS M/500 provides all the IEEE rounding modes.

C.5.3. Floating Comparisons

The CORE [Section 3.5.3] requires the usual six conditional branches with floating or double operands. The MIPS M/500 implements them all, and in addition provides detailed control of the action to be taken if the operands are unordered. This is a most useful extension.

C.5.4. Floating Exceptions

The CORE [Section 3.5.4] requires that the following exceptions be recognized:

- division by zero
- invalid operation
- overflow
- underflow

The MIPS M/500 recognizes and handles all of them. It also recognizes, and can trap on, invalid operands, unordered comparisons, and all the interesting errors associated with infinity.

Overall, the MIPS floating-point co-processor provides a creditable implementation of the IEEE standard, which is both more than CORE requires and thoroughly commendable.

C.6. Assembler Directives

The CORE [Appendix I] defines a set of assembler directives that a conforming translator should support. MIPS provides most of these, though with a UNIX bias.

C.6.1. Segments

The CORE [Section I.2] requires the assembler to support named segments, of any of the types (`instruction`, `data`, `common`) with any of the attributes (`read_only`, `absolute`, `relocatable`, `based_global`).

MIPS supports an extended set of UNIX segments:

- **.text** – instruction, read_only, relocatable

- **.rdata** – data, read_only, relocatable

- **.sdata** – data, relocatable, based_global

- **.data** – data, relocatable

- **.sbss** – common, relocatable, based_global

- **.bss** – common, relocatable

Named common segments are generated by the **.lcomm** directive and allocated to the **.bss** or **.sbss** regions depending on the size of the segment.

This is clearly an evolution of the UNIX view of segmentation and is understandable for an assembler intended exclusively for UNIX-based code. However, it is inadequate for code running under other regimes. In particular, the inability to define several based global areas, or to access read-only data through a base pointer, is a serious handicap, as has been discussed in Sections 6.2.3 and 7.6.

C.6.2. Data Directives

The CORE [Section I.5] requires the usual set of directives for generating initialized and uninitialized static data space. Mips provides all of them, as:

```
CORE            Mips                Comment
.align          .align          align next datum
.ascii          .ascii          ascii string
                .asciz          zero-terminated ascii string
.block          .space          reserve uninitialized space
.byte           .byte           byte data
.double         .double         double precision data
.float          .float          single precision data
.half           .half           halfword data
.word           .word           word data
```

Mips also conforms exactly to the syntax of each directive.

C.7. Local Conclusions

The MIPS M/500 instruction set architecture conforms very closely to the CORE ISA standard. The few deviations are small and can be handled by simple macro substitution or peephole translation. Most of them are justified by the additional simplicity they bring (and hence, one presumes, by cost or performance advantages).

The high-level MIPS assembler is even closer to CORE and can take on most of the burden of handling the deviations. The minor problems inherent in this approach have been discussed elsewhere, and they do not bear on the issue of conformance.

The MIPS assembler directives are very close to those required by CORE, except for restrictions on program segmentation that follow from a UNIX bias. We have argued elsewhere that these restrictions are undesirable.

Overall, the MIPS system is a reasonable and accurate realization of the CORE ISA.

Table of Contents

List of Figures

List of Tables