SOFTWARE SOLUTIONS FOR SINGLE INSTRUCTION ISSUE, IN ORDER PROCESSORS.

Vaneet Aggarwal

Department of Electrical Engineering,

Indian Institute of Technology,

Kanpur, India.

July 27,2004

Project Leader

Sylvain Aguirre

Embedded Information Systems Institute,

University of Applied Science (EIVD),

Yverdon, Switzerland.

Acknowledgements

I would like to express my gratitude towards **Prof. Daniel Mlynek** for giving me this opportunity of doing my internship in EPFL. His suggestions were very beneficial in doing this project.

I would like to thank **Mr. Sylvain Aguirre**, my project supervisor for helping me in understanding the project, discussing the problems, presentations, encouraging me from time to time and editing this report. His comments on various topics were very helpful.

Thanks to **Ms. Aline Gruaz** for the administrative formalities. The support extended by the members of LTS3 was very encouraging.

Table Of Contents

1. Introduction

In my project, I have made some software tools to help VLSI designers take decision early in the design flow(simulator, profiler) and to optimize the input assembly code to the processor(compiler optimizer).

1.1 Processor Simulators

Processor simulators represent to great extent the actual processors. So, the output you get from the simulator is expected if you give the same input to the processor if the simulator is well-made. Some simulators modelling the MIPS R3000 processor [1][2] are:

1. **SPIM**: spim is a self-contained simulator that will run MIPS32 assembly language programs. It reads and immediately executes assembly language code for this processor. spim provides a simple debugger and minimal set of operating system services. spim does not execute binary (compiled) programs. Spim can not run programs compiled for recent SGI processors. MIPS compilers also generate a number of assembler directives that **spim** cannot process. Due to many such disadvantages, we did not use SPIM[8].
2. **MPS**: This is lesser used than VMIPS.
3. **VR3000**: No libraries are available and it is not benchmark compatible
4. **VMIPS**: We can use full cross compiler toolchain with it, there is good documentation, it is updated regularly, and is written in C++, hence is easy to understand and modify. The drawback is that it is not benchmark compatible.

1.2 Virtual Operating System

To allow the user to be able to use any of the library functions that he likes in his program, we need to make a virtual operating system for VMIPS. This will enable VLSI designer to compare his product with the state of the art using standard benchmarks(SPEC[10],EEMBC[11],CommBench[12] etc.). We can run any application, and can execute programs conveniently and efficiently with the help of virtual operating system.

1.3 Profiler

A profiler can help the VLSI designers to take decisions early in their design flow, to find issues, solutions and innovations, to increase performance of the system or to reduce its cost. The profiler can be used to extract relevant information from the user programs and helps characterize various applications. With the relevant informations, VLSI designer can think of a way to improve his design, can see the improvements in

design by the change, and the fallacies in his design, and can adjust some parameters of design by simulating. We had at start a simulator for MIPS R3000 (VMIPS) and the aim of profiler was to add statistics to VMIPS at the execution time.

1.4 Compiler Optimizer

When the user gives some input C file to the compiler, gcc and gas try to do some optimization on the input code. Compiler optimizer is an attempt to optimize machine code so that lesser clock cycles are executed. There are many compiler options with –O3 and –Ofast as the best options in terms of optimizatons. These try to bring about many loop optimizations, jump optimizations, Common Subexpression Elimination, register allocations. And the assembler (gas) tries to do pipeline reorganization to remove pipeline hazards. Our compiler optimizer is based on pipeline reorganization. The aim is to make a compiler optimizer to reduce further the delay slots which are not filled by the assembler by making a standalone tool so that user can, at his pleasure, use it or not.

1.5 Benchmarks

Benchmark is a standard by which something can be measured or judged. Some of the standard benchmarks for processors are:

1. **SPEC Benchmarks**: SPEC CPU2000 provides a comparative measure of integer and/or floating point compute intensive performance. In SPEC CPU2000, Benchmark programs are developed from actual end-user applications as opposed to being synthetic benchmarks. Multiple vendors use the suite and support it. SPEC CPU2000 is highly portable.

2. **EEMBC Benchmarks**: EEMBC benchmarks reflect real-world applications and the demands that embedded systems encounter in these environments. The result is a collection of "algorithms" and "applications" organized into benchmark suites targeting telecommunications, networking, digital media, Java, automotive/industrial, consumer, and office equipment products. An additional suite of algorithms specifically targets the capabilities of 8- and 16-bit microcontrollers.

3. **CommBench**: CommBench is a benchmark for use in evaluating and designing telecommunications network processors. The benchmark applications focus on small, computationally intense program kernels typical of the network processor environment. CommBench has been successfully used for the evaluation of requirements in network processor systems

2. MIPS R3000 Architecture:

MIPS is the most elegant among the effective RISC architectures; even the competition thinks so, as evidenced by the strong MIPS influence to be seen in later architectures like DEC's Alpha and HP's Precision. Elegance by itself does not get you far in the competitive marketplace, but MIPS microprocessors have usually managed to be among the fastest of each generation by remaining among the simplest [2].

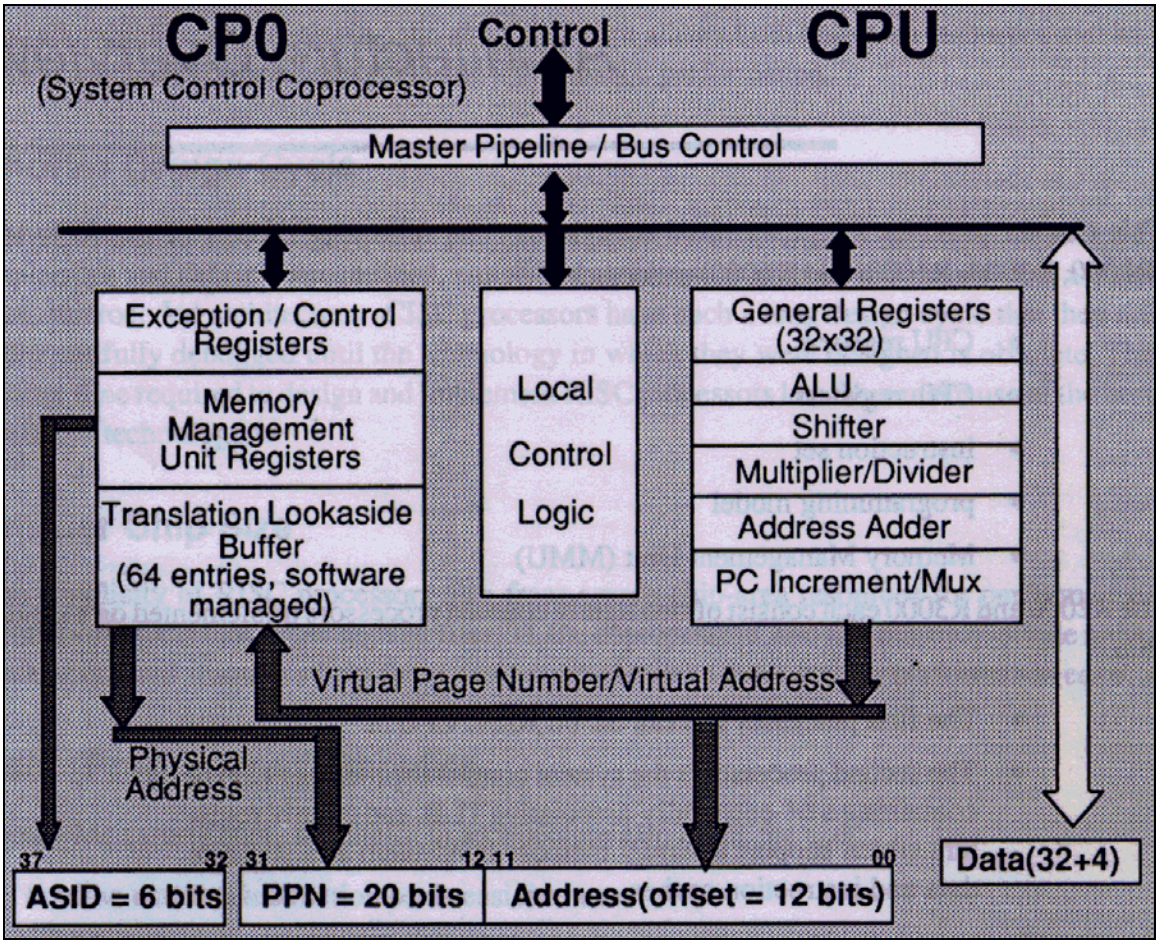


Fig 2.1: Functional Block Diagram of MIPS R3000[2]

2.1 Execution Pipeline

Reduced (or **regular**) **instruction set computer** (or **Computing**) **(RISC)**, is a computer CPU design philosophy that favors a smaller and simpler set of instructions that all take about the same amount of time to execute. The original MIPS SPARC and Motorola 88000 CPUs were classic scalar RISC pipelines. Later, Hennessey and Patterson invented yet another classic RISC, the DLX, for use in their textbook [4]. Each of these designs fetch and attempt to execute one instruction per cycle.

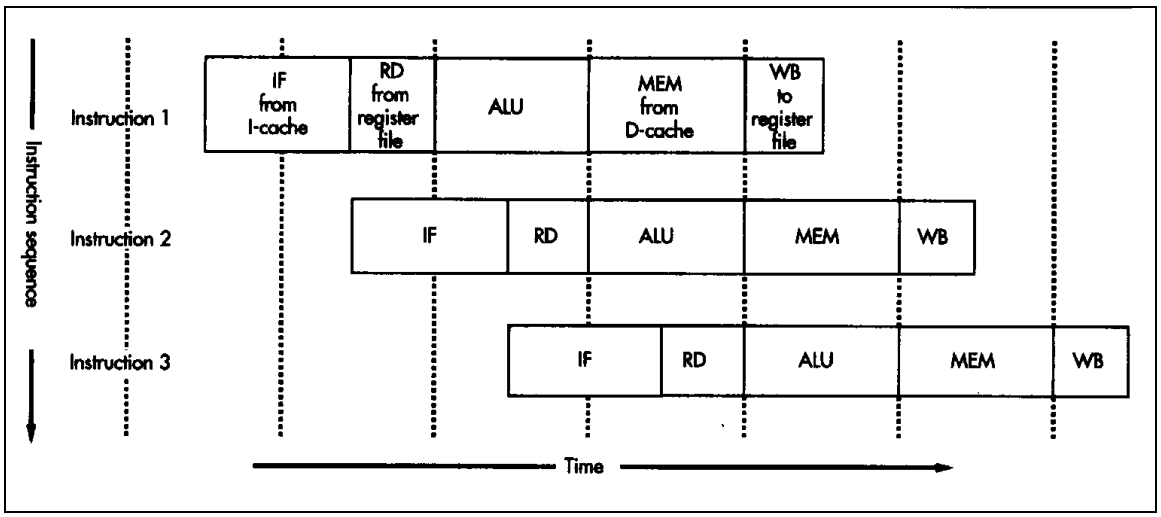


Fig 2.2: MIPS Five-Stage Pipeline

Each design has a five stage execution pipeline. During operation, each pipeline stage work on one instruction at a time. Each pipestage takes a fixed amount of time(some actions take half a clock, and some a clock cycle so that MIPS five-stage pipeline actually occupies only four clock cycles as in Fig 2.2) Each of these stages consists of an initial set of flip-flops, and combinatorial logic which operates on the outputs of those flops. The five stages are:

1. Instruction Fetch: The Instruction Cache on these machines have a latency of one cycle. During the Instruction Fetch stage, a 32 bit instruction is fetched from the cache. At the same time the instruction is fetched, these machines compute the address of the next instruction by incrementing the address of the instruction just fetched. This computation always results in a wrong instruction being executed in the case of a taken branch, jump, or exception.

2. Decode: All MIPS and SPARC instructions have at most two register inputs. During the decode stage, these two register names are identified within the instruction, and the two registers named are read from the register file. In the MIPS design, the register file has 32 entries. At the same time the register file is read, instruction issue logic in this stage determines if the pipeline is ready to execute the instruction in this stage. If not, the issue logic will cause both the Instruction Fetch stage and the Decode stage to stall. On a stall cycle, the stages will prevent their initial flops from accepting new bits. If the instruction decoded is a branch or jump, the target address of the branch or jump can be computed in parallel with reading the register file. The branch condition is also determined in parallel, and if the branch is taken or if the instruction is a jump, the target address is the next cycle's instruction fetch pointer.

3. Execute: Instructions on these simple RISC machines can be divided into three latency classes:

1)Single cycle latency. Add, subtract, compare, and logical operations: During the execute stage, the two arguments are fed to a simple ALU, which generates the result by the end of the execute stage.

2)Two cycle latency. All loads from memory: During the execute stage, the ALU adds the two arguments (a register and a constant offset) to produce a virtual address by the end of the cycle.

3)Many cycle latency. Integer multiply and divide and all floating-point operations: During the execute stage, the operands to these operations are fed to the multi-cycle multiply/divide unit. The rest of the pipeline is free to continue execution while the multiply/divide unit do its work (if there are no data hazards). To avoid complicating the writeback stage and issue logic, multicycle instruction write their results to a separate set of registers. The MIPS CPU does not have integer multiply/divide unit and floating point operations. The multiply unit is independent of rest of the CPU, with its own special output registers.

4. Access: During this stage, single cycle latency instructions simply have their results forwarded to the next stage. This forwarding ensures that both single and two cycle instructions always write their results in the same stage of the pipeline, so that just one write port to the register file can be used, and it is always available. There have been a wide variety of data cache organizations. By far the simplest is direct mapped and virtually tagged, which is what will be assumed for the following explanation. The cache has two SRAMs, one storing data, the other storing tags. During a load, the SRAMs are read in parallel during the access stage. The single tag read is compared with the virtual address specified by the load. If the two are equal, then the datum we are looking for is resident in the cache, and has just been read from the data SRAM. The load is a hit, and the load instruction can complete the writeback stage normally. If the tag and virtual address are not equal, then the line is not in the cache. The CPU pipeline must suspend operation while a state machine reads the required data from memory into the cache, and optionally writes any dirty data evicted from the cache back into memory.

   During a store, the tag SRAM is read to determine if the store is a hit or miss, and if a miss, the same state machine brings the datum into the cache. The store data cannot be written to the cache during the access stage because the processor does not yet know if the correct line is resident. Instead, the store data is written to the cache during the *next* store instruction. In the interim, the store data is held in a Store Data Queue, which, in a classic RISC pipeline is just a single 32 bit register. For reference, the virtual address written is held in the Store Address Queue, again just a single 32 bit register. On more complicated pipelines, the queues actually have multiple hardware registers and variable length. There is one more complication. A load immediately after a store could reference the same memory address, in which case the data must come from the Store Data Queue

rather than from the cache's data SRAM. So, during a load, the load address is also checked against the Store Address Queue. If there is a match, the data from the Store Data Queue is forwarded to the writeback stage, rather than the data from the data SRAM. This operation does not change the flow of the load operation through the pipeline.

5. Writeback: During this stage, both single cycle and two cycle instructions write their results into the register file.

The pipeline tuning has some strange effects as delayed branches, late data from load, and multiply/divide effects. As explained before in instruction fetch, the instruction after the branch is executed, as the branch target is not available so early. This is also clear from Fig 2.3. Similarly for load, there has to be a gap of one clock cycle when the contents of loaded register are available. This can also be seen in Fig 2.4. In MIPS, result of multiply/divide are stored inside multiply unit and instructions mfhi,mflo retrieve the results into specified general registers. Multiply result registers are interlocked. An attempt to read out the results before multiplication is complete results in the CPU being stopped until the operation completes. If an mfhi or mflo instruction is interrupted by some kind of exception before it reaches write-back stage of the pipeline, it will be aborted with the intention of restarting it. However, a subsequent multiply instruction will overwrite hi and lo registers, so that re-execution of mfhi will be wrong. So, it is recommended that multiply must not be started within two instructions of an mfhi/mflo.

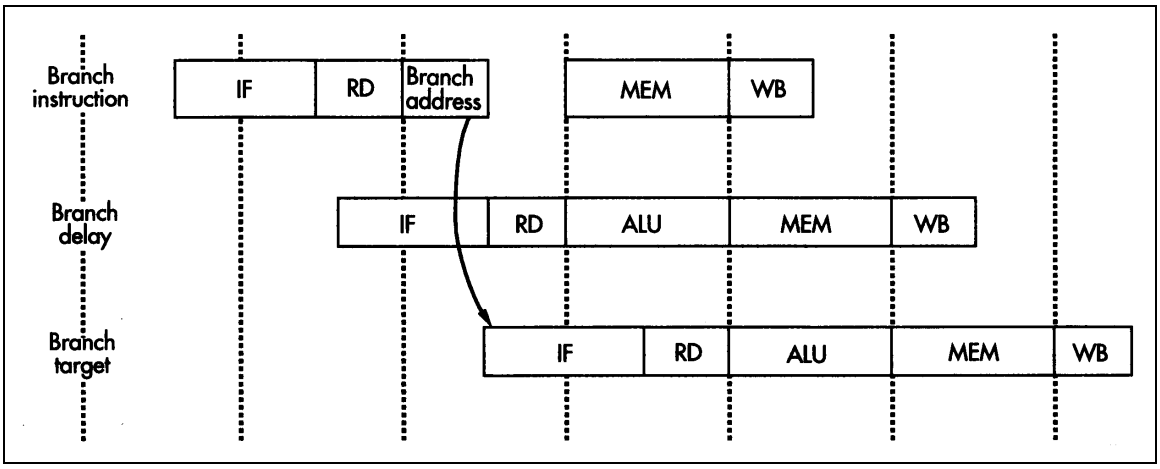


Fig2.3: Branch Delay Slot

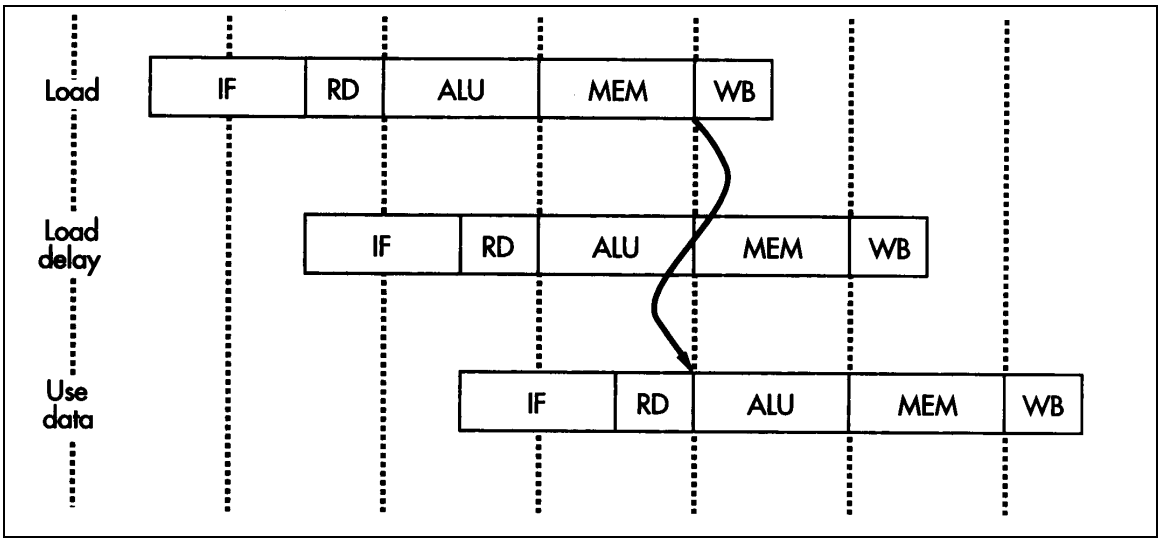


Fig2.4: Load Delay Slot

2.2 Data Types and Literals

The data types are:

- byte, halfword (2 bytes), word (4 bytes)
- a character requires 1 byte of storage
- an integer requires 1 word (4 bytes) of storage

The literals:

- numbers entered as it is e.g. 4
- characters enclosed in single quotes e.g. 'b'
- strings enclosed in double quotes e.g. "A string"

2.3 Addressing

The MIPS R3000 architecture has two address spaces: a virtual address space, consisting of all the addresses that can be used in the program, and a physical address space, consisting of all the addresses that can be sent out on the address bus.

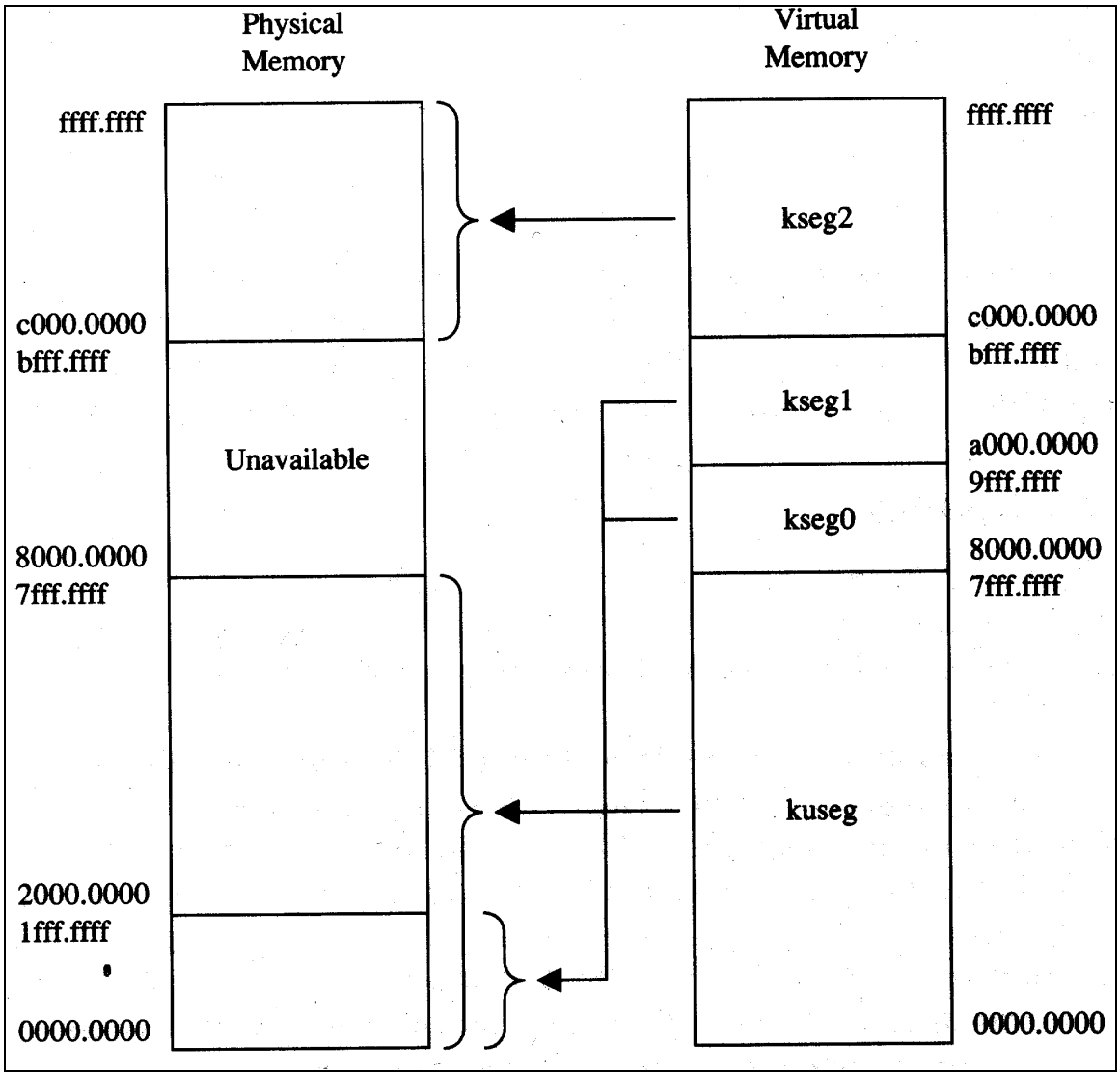


Fig 2.5: Virtual to Physical Address Mapping

The virtual address space (4 GB) is partitioned into four fixed segments: kuseg, kseg0, kseg1, and kseg2. Thus virtual address consists of segment number (the three most significant bits of the address) and an offset within the segment. In the translation of virtual address to the physical address, the 12 least significant bits of the virtual address are unchanged, so that in implementations using a memory management unit (MMU) and a translational lookaside buffer (TLB), segments can be further partitioned into 4K pages, with addresses consisting of a virtual page and an offset within the page.

2.4 Segments

The MIPS divides its address space into several regions that have hardwired properties. These are:

- kseg2, TLB-mapped cacheable kernel space (1024 MB)

- kseg1, direct-mapped non-cacheable kernel space (512 MB)
- kseg0, direct-mapped cacheable kernel space (512 MB)
- kuseg, TLB-mapped cacheable user or kernel space (2048 MB)

Direct mapped cache is a cache where the cache location for a given address is determined from the middle address bits. If the cache line size is 2^n then the bottom n address bits correspond to an offset within a cache entry. If the cache can hold 2^m entries then the next m address bits give the cache location. The remaining top address bits are stored as a "tag" along with the entry. The scheme can suffer from many addresses "colliding" to the same slot, thus causing the cache line to be repeatedly evicted, even though there may be empty slots that aren't being used, or being used with less frequency. TLB-mapped are mapped by Translation Lookaside Buffer with software refill.

Table 2.1: The memory map

Address	**Segment**	**Special properties**
0xffffffff	kseg2	Address bits 31,30,29 are 1,1,*
0xc0000000		
0xbfffffff	kseg1	Address bits 31,30,29 are 1,0,*
0xbfc00180		Exception address if BEV set.
0xbfc00100		UTLB exception address if BEV set.
0xbfc00000		Execution begins here after processor reset.
0xa0000000		
0x9fffffff	kseg0	Address bits 31,30,29 are 1,0,0
0x80000080		Exception address if BEV not set.
0x80000000		UTLB exception address if BEV not set.
0x7fffffff	kuseg	Address bits 31,30,29 are 0,*,*
0x00000000		

kuseg, accessible in both user and kernel mode, is designed to be used by user-mode programs, while also providing accessibility to this address space in kernel mode.kseg1 is designed for access to peripheral devices and for code that requires non-cacheable access, including initialization code that is executed before the caches have been flushed. The processor fetches instructions from this segment following reset, because neither the cache nor the TLB will have been initialized, and this is the only “safe” area from which to execute. kseg2 is designed for those parts of the OS that will use virtual memory. kseg0 is for all other parts of the OS. kseg0 and kseg1 map to same physical addresses. kuseg and kseg2 are mapped via the TLB(if there is one). When no TLB is present, the mapping of these segments is somewhat vendor dependent.

2.5 Modes of Operation:

The MIPS processor has two modes of operation, ***User Mode*** and ***Kernel Mode***, as selected by Current Kernel/User Mode bit (KUc) in the Status Register. The two modes of operation dertermine the addresses, registers, and instructions available to a program.

Kernel-mode programs are permitted to use all addresses, registers, and instructions. User-mode programs, on the other hand, may use only the virtual addresses 0 through 7fffffff(2 GB), and an attempt to access any other segment causes an Address Error Exception. A user-mode program´s use of coprocessor instructions and registers can be restricted by clearing the appropriate Coprocessor Usability bit in the Status Register.

The processor enters kernel mode when the system is reset, when an exception occurs, or when the Status Register´s Previous Kernel/User Mode bit is set by a user-mode program permitted to access coprocessor 0(an unlikely occurrence).

The processor switches to user mode when the restore from exception instruction is executed and the Status Register´s Previous Kernel/User Mode bit is set, or when the Current Kernel/User Mode bit is cleared explicitly by a kernel mode program(not recommended). **The work done is in kernel mode.**

2.6 Registers

There are 32 general-purpose registers and 3 special registers on the MIPS R3000 itself. There are also up to 32 registers each on up to four coprocessors. For CS161 purposes, there is only one coprocessor, coprocessor 0, which is the "system coprocessor"; it takes care of exceptions and virtual memory issues.

Table 2.2: Register Descriptions

Register	**Symbolic name**	**Description**
General registers		
$0	z0, ZERO	Always contains 0, no matter what's written to it.
$1	AT	Assembler temporary..
$2	v0	Used for computations; function return value is placed here.
$3	v1	Used for computations; upper word of 64-bit return value is placed here.
$4	a0	Argument 0. First function argument goes here.
$5	a1	Argument 1. Second function argument goes here.
$6	a2	Argument 2. Third function argument goes here.
$7	a3	Argument 3. Fourth function argument goes here. Also used as

		a flag value on system call return.
$8	t0	General-purpose temporary register.
$9	t1	General-purpose temporary register.
$10	t2	General-purpose temporary register.
$11	t3	General-purpose temporary register.
$12	t4	General-purpose temporary register.
$13	t5	General-purpose temporary register.
$14	t6	General-purpose temporary register.
$15	t7	General-purpose temporary register.
$16	s0	General-purpose saved register.
$17	s1	General-purpose saved register.
$18	s2	General-purpose saved register.
$19	s3	General-purpose saved register.
$20	s4	General-purpose saved register.
$21	s5	General-purpose saved register.
$22	s6	General-purpose saved register.
$23	s7	General-purpose saved register.
$24	t8	General-purpose temporary register.
$25	t9	General-purpose temporary register.
$26	k0	Kernel scratch register.
$27	k1	Kernel scratch register.
$28	gp	Global pointer. Constant for any given process.
$29	sp	Stack pointer.
$30	s8	Saved register #8 - conventionally, but not always, a frame pointer.
$31	ra	Return address of function.
Special registers		
HI	-	High-order word of 64-bit multiply result, or remainder of divide result.
LO	-	Low-order word of 64-bit multiply result, or quotient of divide result.
PC	-	Program counter.
Coprocessor 0		
cop0 $0	c0_index	TLB entry index register.
cop0 $1	c0_random	TLB randomized access register.
cop0 $2	c0_entrylo	Low-order word of "current" TLB entry.

cop0 $4	c0_context	Page-table lookup address.
cop0 $8	c0_vaddr	Virtual address associated with certain exceptions.
cop0 $10	c0_entryhi	High-order word of "current" TLB entry.
cop0 $0	c0_status	Processor status register.
cop0 $13	c0_cause	Exceptions cause register.
cop0 $14	c0_epc	PC at which exception occurred.

Any of the 32 general-purpose registers can be used in any instruction that takes register operands. The special registers are accessed using special instructions; the coprocessor registers can be accessed by using special coprocessor instructions to move their values to general registers and back.

Register $31 is the "link register". Most of the instructions for calling subroutines are hardwired to store the return address into this register. (The jalr instruction is, for some reason, an exception as in this you can give some other value of link register.)

2.7 Exceptions

Exceptions are conditions, which alter the normal sequence of instructions, causing the processor to transfer control to a predefined location in memory, the so-called exception vector. In the MIPS1 base architecture there is a single exception vector, the General Exception Vector, whose virtual address depends on the setting of the Status Register's Bootstrap Exception Vector (BEV) bit, as shown in Table 2.3 and described later in this chapter.

The MIPS architecture recognizes seventeen exceptions: eight external interrupts (six hardware interrupts and two software interrupts), and nine program exception conditions (sometimes referred to as 'traps'). The type of exception is encoded in the Exception Code (ExcCode) field of the Cause Register, as shown in Table 2.4.

Table 2.3 General Exception Vector Addresses

	BEV = 1 (Reset State)	BEV = 0
Virtual Address	Oxbfc0.0180 (kseg1)	0x8000.0080 (kseg0)
Physical Address	0x1fc0.0180	0x0000.0080

Table 2.4 ExcCode Field in Cause Register

ExcCode Value	Assembler Mnemonic	Exception Type
0	EXC_INT	External interrupt
1	-	Reserved
2	-	Reserved
3	-	Reserved
4	EXC_ADEL	Address error (load or instruction fetch)
5	EXC_ADES	Address error (data store)
6	EXC_IBE	Bus error (instruction fetch)
7	EXC_DBE	Bus error (data load or store)
8	EXC_SYS	Syscall instruction
9	EXC_BP	Breakpoint
10	EXC_RI	Reserved instruction
11	EXC_CPU	Coprocessor Unusable
12	EXC_OVF	Arithmetic overflow
13-15		Not used.

When an exception occurs, the following events take place:

1. The currently executing instruction is aborted, as well as any instructions in the pipeline that have already begun executing.
2. In the Status Register, the Previous Kernel/User Mode and Previous Interrupt Enable bits are copied into the Old Mode and Old Interrupt Enable bits respectively, and the Current Mode and Current Interrupt Enable bits are copied into the Previous Mode and Previous Interrupt Enable bits.
3. The Current Interrupt Enable bit is cleared, which disables all interrupts.
4. The Current Kernel/User Mode bit is cleared, which places the processor in Kernel Mode.
5. If the instruction executing when the exception occurred is in the delay slot of a branch, the Branch Delay (BD) bit in the Cause Register is set.
6. The Exception Program Counter Register (EPC) is set with the address at which the program can be correctly restarted. If the instruction causing the exception is in the delay slot of a branch (BD=1), the EPC is written with the address of the preceding branch or jump instruction. Otherwise, it is written with the address of the instruction that caused the exception, or in the case of an interrupt, with the address of the next instruction to be executed.
7. The Exception Code (ExcCode) field of the Cause Register is written with a number, between 0 and 15, that encodes the type of exception (refer to Table 2.4).
8. If the exception is a Coprocessor Unusable exception, the Cause Register's Coprocessor Error (CE) field is set with the referenced Coprocessor Unit number.
9. If the exception is an Address Error, the address associated with the illegal access is written to the BadVAddr register.

10. The processor then jumps to the General Exception Vector, whose address depends on the setting of the BEV bit: When BEV = 1, the General Exception Vector is mapped to a noncacheable kseg1 address; when BEV = 0, it is mapped to a cacheable kseg0 address (see Table 2.3).

When the exception handler has completed, it uses the address in the EPC register as the return address in a jump, and then executes a Return From Exception (rfe) instruction in the jump's delay slot. The rfe instruction restores the Current and Previous Mode and Interrupt Enable bits to their contents prior to the interrupt, leaving the Old bits unchanged.

3. VMIPS, A MIPS R3000 Simulator

3.1 Overview

VMIPS is a virtual machine simulator based around a MIPS R3000 RISC CPU core. It is an open-source project written in C++ and which is distributed under the GNU General Public License. VMIPS, being a virtual machine simulator, does not require any special hardware. It has been tested under Intel-based PCs running FreeBSD and Linux, and a patch has been developed for compatibility with CompaQ Tru64 Unix on 64-bit Alpha hardware. Eventually, we expect to put vmips through more rigorous portability testing. vmips is not useful without a full set of MIPS-targeted cross-compilation tools, however, and so the build process assumes their existence on your host system. Since VMIPS is based on an already-existing architecture, it is relatively easy to find tools to build programs that will run on VMIPS. Since VMIPS is based on RISC architecture, its primitive machine-language commands are all fairly simple to understand.

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

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

3.2 Steps to Execute Programs with VMIPS

Step 0. If VMIPS is installed on your system, you can start building programs with it right away. Otherwise, you (or your system administrator) will have to compile VMIPS first; see the appendix on Installation (A6).

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

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

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

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

3.3 An Example

Let's assume you have VMIPS already compiled, and that you have some setup code in `setup.s', and a standalone program (i.e., not one meant to run under an operating system) in `hello.c'.
First assemble the setup code.
vmipstool --assemble -o setup.o setup.s
Compile your program:
vmipstool --compile -c hello.c
Then, link your program and the setup code together to produce an executable:
vmipstool --link -o hello setup.o hello.o
Build a ROM image from the executable:
vmipstool --make-rom hello hello.rom
Run the program.
vmips hello.rom

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

3.4 Building Programs

3.4.1 Source Languages

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

3.4.2 ROM Programs

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

3.4.3 Default Setup Code

This section describes the default VMIPS setup code. It also describes the minimal set of things you need to do before you can run C code from the ROM, since that is the intended purpose of the default VMIPS setup code. Start by clearing out registers and TLB entries. Set yourself up a stack pointer ($sp). Usually this can just be some number of megabytes above the end of your code's data segment. You can get the address of the end of your code's data segment from your linker script. Set up your globals pointer ($gp), if your code uses global data. You can get the right address from your linker script. If you have writable data in ROM, your C code probably doesn't realize that it's in ROM, and it will want to write to it. You should copy the writable data to RAM. There is code to do this in the canned setup code provided with VMIPS. Finally, your setup code should finish by calling the entry point of your C code. Usually this will have a name like entry; using the name main is not recommended, because many versions of GCC assume that they can call standard C runtime setup functions (such as are normally found in `crt0.o') from the beginning of main. You may or may not want this. When the C code returns, you will probably want to halt the machine; the default way to do this is by executing a break instruction.

3.4.5 Linking

You want the text section of your program to start with the setup code, so link in the setup code first - that is, put the name of the object file containing the setup code first on the linker command line. You want the setup code to start at 0xbfc00000, which is the MIPS reset exception vector. In practical terms, when VMIPS starts up, it will reset. When VMIPS resets, it jumps to 0xbfc00000, which is the beginning of your setup code.

3.4.6 Common Errors in Compilation

If the linker complains about not being able to find the symbol _gp_disp, you should turn on the GCC option `-mno-abicalls'. _gp_disp is used by the SGI N32 ABI for MIPS ELF. One reliable reference source claims, “_gp_disp is a reserved symbol defined by the linker to be the distance between the lui instruction and the context pointer." The GNU linkers currently in use do not appear to support this function. If you get lots of R_MIPS_GPREL16 relocation failures from the linker, there are two workarounds: either

combine all the files together first with `ld -x -r -o bigfile.o <all your files>' and then use `vmipstool --link' on `bigfile.o', or compile with `-G 0' in your CFLAGS.

4. A Virtual Operating System for VMIPS

4.1 Overview

VMIPS is an emulator with no Virtual OS of its own. So, we can't use any of the library function in C programs while using VMIPS. For that, we need Virtual Operating System so that we can validate VMIPS, and more precisely the full toolchain, make VMIPS benchmark compatible to compare features and performance of other architectures.

We want that the Virtual OS functions be implemented in the MIPS processor model. Any function in the user code can be implemented using the basic operators (add, subtract, multiply, divide, etc.) and read, write, open, close, lseek, get_mem_info etc. functions. For that purpose, we need that VMIPS should be able to handle read, write, open, close etc. As these functions are machine dependent, so we do not have the implementation of these functions in standard libraries. The operating system provides its services through a set of ***system calls***, which are in effect functions within the operating system that may be called by the user program. On a particular system, the routines of the standard library have to be written in terms of the facilities provided by the host system. Since, we do not want VMIPS to be dependent on host OS on which we are simulating, we must use the host machine functions to simulate so that our emulation is not system-dependent. These system functions must be implemented as system calls, and we must make a call to the host system when we find such instructions. This is because VMIPS has no idea how to access host machine's disc. So, we need some special handling of some basic system calls via the syscall instruction.

So, for making the Virtual Operating system we need to write a setup code. This code can also be used for embedded system based on MIPS1 ISA. So, this is essentially an embedded system program. The MIPS1 ISA includes the R2000 and R3000 designed by MIPS Computer Systems, and all the derived processors from the licensed semiconductor vendors. The software conventions defined by MIPS assembler and compiler is to be followed. These conventions are important to adhere to when you are writing assembly language routines that will be called from C, or if you call C routines from your assembly language program.

The register names and definitions, coprocessor 0 and 1 register definitions and bit names are defined in file asm_regnames.h. The assembler preprocesses all files using standard C preprocessor, and uses this mechanism to implement file inclusion and macro definitions.

Executable files for the MIPS processor consist of three main sections: .text, .data, and .bss, used to hold code, initialized global data, and uninitialized global data, respectively. To support the use of the Global Data Pointer Register (called gp), two additional sections are defined: .sdata and .sbss. These sections are used for initialized and uninitialized data that will be referenced relative to gp. Separate sections are defined

for those because the gp-referencing mechanism is limited to a 64 KB area of memory, and thus not all variables in a large program can be placed in these sections. The default for most tools is to place all the variables of 8 bytes or lower in .sdata or .sbss, although this can usually be overridden by your program's data if space permits. It is also common for compilers to put various kinds of constant data in additional sections. For example, constant data that is read-only is often put in a section called .rdata .

From the point of view of register-usage conventions, there are basically three programming environments: initialization, subroutine, and exception handler.

The initialization environment is the least restricted in that there is no program state to be saved, and all registers are available for use. In the subroutine environment, register usage is more restricted because the caller's state has to be saved and some of the registers are needed for passing arguments and results. Exception handlers are similar to subroutines, except that register usage is even more restricted, because almost all the registers may already be in use by the interrupted program. In fact, only k0 and k1 can be used as temporaries, and then only while interrupts are disabled.

When a procedure is called, the first four scalar arguments are passed in a0-a3 and any additional arguments are passed in the stack. Floating-point arguments are passed in $f12-$f14 and in the stack. Scalar values are returned from procedures in v0 and v1. Actually, ints, chars, shorts, longs, and pointers are returned only in v0. Floating-point values are returned in $f0-$f3.

Registers that can be used by the procedure without being saved are a0-a3, t0-t9, v0 and v1 for scalar values, and $f0-$f19 for floating point values. Keep in mind when using these registers that they are not preserved across calls. These are referred to as "temporaries". These do not have to be saved before they are used.

Registers that can be used as temporaries and whose values are preserved across calls are s0-s7, and s8 if it has not been allocated for use as the frame pointer; for floating-point values, registers $f20-$f30. These registers must be saved before they are used and then restored before exiting the function. These are referred to as "saved" registers.

The global data pointer, gp permits 64K are of memory-resident data to be accessed with a single instruction. The label _gp is initialized by the linker with a 32-bit base address that, when combined with 16-bit offset in a load or store instruction, accesses a range of 64K. gp relative addressing is used for data in the .sdata and .sbss sections.

4.2 Embedded system program.

The setup code is saved under filename setup1.s. The three steps involved (initialization, subroutine and exception handler) have been explained before (Section 4.1). The tasks performed in them are explained here:

4.2.1 Initialization:

Initialization code typically performs the following tasks:

4.2.1.1 Initializes the memory.

```
.set noat
move $1, $0
.set at
move $2, $0
move $3, $0
move $4, $0
move $5, $0
move $6, $0
move $7, $0
move $8, $0
move $9, $0
move $10, $0
move $11, $0
move $12, $0
move $13, $0
move $14, $0
move $15, $0
move $16, $0
move $17, $0
move $18, $0
move $19, $0
move $20, $0
move $21, $0
move $22, $0
move $23, $0
move $24, $0
move $25, $0
move $26, $0
move $27, $0
move $28, $0
move $29, $0
move $30, $0
mtc0 zero, $4
mtc0 zero, $8
mtc0 zero, $14
```

4.2.1.2 Clears the .bss and .sbss sections.

We used only .text, so didn’t do this step.

4.2.1.3 Flush the cache

We didn't do this here as we are working only in non-cacheable, direct mapped kseg1 segment. We are working only in this segment as we want to work in Kernel mode due to restrictions of user mode as explained in Section 2.5. kseg1 is the "only" safe area from which program can execute after reset. Further details on segments are there in Section 2.4,2.5.

4.2.1.4 Copy program data from ROM to RAM.

This has to be done because C permits the values of variables to be modified during execution (of course, the variables have to start in ROM so their initial value is preserved).

```
la t1, _copystart               /* t1 = beginning source address for copy */
la t2, _copyend
addiu t2, t2, 4         /* t2 = one word past ending source address */
move t3, gp                     /* t3 = beginning dest address */
1:
lw t4, 0(t1)                    /* load t4 from ROM */
sw t4, 0(t3)                    /* store it in RAM */
addiu t1, t1, 4         /* increment both pointers */
addiu t3, t3, 4
bne t1, t2, 1b          /* if we're not finished, loop. */
nop
```

4.2.1.5 Initializes the stack pointer and the global data pointer.

```
li sp, INIT_STACK_BASE
la gp, _gp
```

4.2.1.6 Initializes I/O devices.

We need not do this step as we have no I/O devices to use.

4.2.1.7 Enables interrupts

This is for clearing possible old exceptions.

```
li t2, NTLBENTRIES /* t2 = TLB entry number */
li t3, 0x00000fc0       /* t3 = (VPN 0x0, ASID 0x3f) */
1:
addiu t2, t2, -1        /* Decrement TLB entry number */
sll t1, t2, 8           /* Shift entry number into Index field position */
mtc0 t1, $0             /* set Index */
mtc0 zero, $2           /* clear EntryLo */
mtc0 t3, $10            /* set EntryHi */
```

```
tlbwi                           /* write TLB[Index] with (EntryHi, EntryLo) */
bnez t2, 1b             /* Go back if we're not done yet. */
nop
mtc0 zero, $10        /* clear EntryHi (sets effective ASID=0x0) */
```

4.2.1.8 Switches from non-cacheable to cacheable space.

Our code executes in non-cacheable space. So, we do not need this step.

4.2.2 Subroutine:

The subroutines that can be created must pertain to the format shown below.

Format:

```
.globl <function name>
.ent <function name>
<function name>:
function body
.end <function name>
```

An **example** of subroutine with function name as write and generating a syscall with register contents of v0 as 7 is:

```
.globl write
.ent    write
write:
addiu $2,$0,7
syscall
j       $31
.end write
```

4.2.3 Exception Handling:

To return to the interrupted program, we read the value from the EPC Register into k0, insert the required nop (the mfc0 instruction requires 2 cycles), and then do a jump indirect on k0. The rfe in the delay slot restores the status register.

```
.set noreorder
mfc0    k0,EPC
nop
j       k0
rfe
```

```
.set reorder
```

4.3 Implementing other functions in terms of basic functions

Example:

```
int getchar (void)
{
        char ch;
        int rv;

        rv = read (0, &ch, 1);
        if (rv != 1)
          {
            return -1;
          }
        else
          {
            if ((modes & ICRNL) && (ch == '\r'))
              ch = '\n';
            if (modes & ECHO)
              putchar (ch);//To display the character read on screen also
            return ch;
          }
}

int putchar (unsigned char ch)
{
        if ((modes & ONLCR) && (ch == '\n'))
          putchar ('\r');
        return ((write (1, &ch, 1) < 1) ? -1 : 0);
}
```

4.4 Implementation of the Virtual OS

I have made a Virtual Operating Systems using the library functions. The main part of the code is in setup1.s, which you can see in the appendix (A5). The code has been implemented as explained in 4.2. Now, when the library files (libc.a, libidt.a etc.) are used, you find that they jump to particular locations inside the code (at 0x30 for open, 0x38 for read, 0x40 for write, 0x50 for close, 0x58 for inbyte, 0x60 for outbyte, 0x1b8 for get_mem_info etc. with respect to the start of .text). You can find these numbers when you see the disassembled net object file. So, you have to do something there so as to handle the code. I take the example of write function. What I have done is

```
.org 0x40
.set noreorder
```

j writea//As write function is there in the library, which makes this line to execute, we must give some different name from write here, and you must take care that this name do not clash with the names inside user programs. If that is the case, edit this function name.
nop
.set reorder
which jumps to writea function which is :
.globl writea
.ent writea
writea:
addiu $2,$0,7
syscall
j $31
.end writea

Now, this function is a system call, which is implemented in VMIPS, where I take the arguments from a0, a1, and a2(In case of basic types like long, int etc, the arguments are in these registers, and in case of structure or string, the pointer to the virtual address where they are stored are in the registers) and do the operation. When a procedure is called, the first four arguments are passed in a0-a3 and any additional arguments are passed in the stack. Floating-point arguments are passed in $f12-$f14 and in the stack (We did not deal with floating points as MIPS R3000 do not support floating points). Scalar values are returned from procedures in v0 and v1. Actually, int, char, shorts, long, and pointers are returned only in v0. Floating-point values are returned in $f0-$f3. The algorithm of code handling this case in VMIPS is as:

int write(int fd, const void *buf, int count);// The first argument is a file descriptor, 2nd argument is a character array in your program where the data is to come from. The third argument is the number of bytes to be transferred.

So, fd is in reg[REG_A0], the pointer in reg[REG_A1] and count in reg[REG_A2], and we have to put result in reg[REG_V0]. During emulating syscall, put the following algorithm in the code.
case 7://Syscall No. for write
char buf[reg[REG_A2]];
numberofwords=reg[REG_A2];
*buf='\0';
virt=reg[REG_A1];//Virtual Address
phys = cpzero->address_trans(virt, DATALOAD, &cacheable, this*);//Convert Virtual Address to Physical Address*
Now get the contents of buf from memory. Once you fetch the word using
word=mem->fetch_word(phys, DATALOAD, cacheable, this);
you get the 4 bytes in reverse format, go on incrementing the physical address and keep on getting characters and form the string till we have got the count number of bytes or the end of string is observed.
reg[REG_V0]=write(reg[REG_A0],buf,reg[REG_A2]);

```
*Call write of host machine.*
return;
```

Consider one more example, times function: clock_t times(struct tms *buf); which requires from the system to fill the structure and return a value which is of type clock_t. We get the pointer to structure in register a0, and have to find where are the pointers to other members of the structure. This is one of the simple example in which the various arguments take space of four bytes, so keep on adding the address by four and store the next element in the memory. You can check the values stored are in correct position by using your test program and checking the values of same variables inside emulation function and in the test program. Sometimes, you may need to fetch the memory position where to store a particular value of structure.

```
phys=cpzero->address_trans(reg[REG_A0], DATASTORE, &cacheable,
this);
tms start;
reg[REG_V0]=times(&start);
mem->store_word(phys,start.tms_utime , cacheable, this); // Store
start.tms_utime at the physical address location phys
mem->store_word(phys+4,start.tms_stime , cacheable, this);
mem->store_word(phys+8,start.tms_cutime , cacheable, this);
mem->store_word(phys+12,start.tms_cstime , cacheable, this);
```

Similiarly, all the system call functions are implemented. But there are many function made without a call to the system because they do not need it. You must not implement a function as a system call if the function is not defined as the underlying OS function in the libraries you are using. We need system call only in read, write, open, close, lseek, gettimeofday, times like functions. But we need to implement some other functions also like inbyte, outbyte, get_mem_info, etc. The get_mem_info is implemented as:

```
.org 0x1b8
j get_mem_info1
nop

.globl  get_mem_info1
.ent    get_mem_info1
.set    noreorder
get_mem_info1:
subu    sp,sp,0x18      /* create some stack space */
sd      ra,0x00(sp)     /* stack return address */
sd      $30,0x08(sp)     /* stack frame-pointer */
move    $30,sp          /* take a copy of the stack pointer */
li      v0,0x10000000 /*256 MB*//* Remember to put the same number in
the memsize option inside VMIPS*/
nop
sw      v0,0(a0)
```

```
li a2,0x4000 /*16 KB*/
sw a2,4(a0)
sw a2,8(a0)
move    sp,$30          /* recover stack pointer */
ld      ra,0x00(sp)     /* recover return address */
ld      $30,0x08(sp)    /* recover frame-pointer */
j       ra              /* return to the caller */
addu    sp,sp,0x18      /* restore stack pointer {DELAY SLOT} */
.set    reorder
.end    get_mem_info1
```

The get_mem_info jumps to 0x1b8, and we jump to get_mem_info1 from there. We give it default value of amount of memory available as 256 MB, and Instruction and Data Cache sizes as 16 KB.

Similiarly for outbyte function (This outputs a byte to the screen) , which jumps at 0x60, we jump to outbytea. Outbytea will call write (1, (unsigned long)&c, 1) as:

```
addiu   sp,sp,-32       /* create some stack space */
sb      a0,16(sp)       /*Store a0 contents*/
li      a0,1            /*1st argument is 1*/
addiu   a1,sp,16        /*2nd argument */
sw      ra,24(sp)
jal     write
li      a2,1            /*3rd argument*/
lw      ra,24(sp)
nop
jr      ra
addiu   sp,sp,32        /*After returning, restore stack pointer and jump to
return address.*/
```

To link the files, remember that you want the text section of your program to start with the setup code, so link in the setup code first - that is, put the name of the object file containing the setup code first on the linker command line. The setup code is to start at 0xbfc00000, which is the MIPS reset exception vector, which is in non-cacheable direct mapped kseg1 space in which we are working. In practical terms, when VMIPS starts up, it will reset. When VMIPS resets, it jumps to 0xbfc00000, which is the beginning of your setup code. So, the code at this position must call to the initialization routine and after that there must be a call to the user program. Also, the exception routine must be there at the positions where the pc jumps to in case of exception which are 0x100 and 0x180 relative to the starting.

There can be 2 approaches to the making of the OS. The 1st one has been explained. Briefly, link all the library files and implement the system call functions and some other functions that you need. The second one is for the person who uses embedded system with a smaller list of functions. For such user, you can implement these functions easily with the help of keyboard and data registers, and the basic open, close etc.

functions. I have implemented many functions in this pattern (Appendix A8). The use of these is that if the user code uses only the functions I have implemented, he can use all my functions, he need not link any library, and the advantage will be the net code will be about 10-20 times shorter which is a very good use of it. If the user wants to run the embedded processor with only a certain list of functions, it is better that he makes those functions by himself without using library for getting smaller code. This is because, in library, they have implemented a lot of functions, so you call to one function and that will call to some other function, and then some other. In the library codes, they try to optimize all the functions, and not some specific functions, which will be the case in implemented your own functions. So, there is a trade-off in both approaches. If you use 1st, you can use any function you need, and if you use 2nd, you get smaller clock cycles to execute.

Clearly for the two approaches, you need different assembly setup files. So. I have made setup1.s for the 1st approach, which has the following syscall numbers 5, 6, 7, 8, 26, 27, 28, 34, 35 for its operations (opena, reada, writea, closea, lseek, fstat, isatty, gettimeofday, times). The syscall handling functions inside VMIPS for other syscall numbers are extra and can be removed if you are using only this approach but must be kept as such if you want to be able to use both approaches. For 2nd approach, there is setup2.s file in the same directory.

4.5 Some of the common problems

1. Error: Attempt to map two VMIPS components to the same memory area: This is just a warning if you use more than 16 MB memory by –o memsize option. So, don't worry about it. Your program will work well on embedded system if you have the memory size given on command line and the simulation runs as the simulation do not stop after this error message. If you want to get away with the message as you have more than 16 MB RAM, do the following:

   When you start VMIPS you will see several diagnostic printouts intended to help debugging precisely this situation. For example:

   ```
   $ ./vmips boot.rom
   Big-Endian host processor detected.
   Mapping ROM image (boot.rom, 5120 words) to physical address 0x1fc00000
   Mapping RAM module (host=0x1008000, 16384KB) to physical address 0x0
   Mapping Halt device to physical address 0x01010024
   Mapping Clock device to physical address 0x01010000
   Connected IRQ7 to the Clock device
   Mapping SPIM console to physical address 0x02000000
   ```

   This display tells you (among other things) that there are memory-mapped devices configured starting at physical addresses 0x01010000 and 0x01010024. Unless you were to move them or disable them, you won't be able to configure

more than about 16MB of memory,because RAM may not overlap memory-mapped devices. So, edit the address locations so as to allow more size between them if you need.

It is easy to disable the halt device using the -o noclockdevice and –o nohaltdevice options. Moving them requires changing the CLOCK_ADDR and HALT_ADDR #defines in clockreg.h and haltreg.h respectively, and recompiling.

2. If it takes a long time to build a ROM or the ROM file fills the disk

   The likely problem is that you have a section or two, which are not accounted for in the linker script (ld.script). The trick with putting programs that have writable static data into a ROM is that you have to copy the writable data to RAM before you can write to it, and typically you'd like to make this process transparent to the program -- that is, it is handled entirely by the setup code. The setup code assumes any part of the .data section may need to be written to, and copies it all to RAM. The linker script in sample_code/ld.script fills in the addresses of all data objects as they are loaded into RAM by setup.S, not as they are in the ROM. Since the beginning of RAM at 0xa0000000 is ~500 megs or so from the beginning of ROM (0xbfc00000), if there are data objects in sections that the linker script doesn't know about, you will end up with a very large rom file, because objcopy assumes that you will need a rom file that starts with 0xa0000000 and ends at 0xbfc…. The solution is to make sure that all LMAs in the executable are sane with respect to the `loadaddr' variable in your `.vmipsrc' (The ROM file is initially loaded into the memory at the address given in option loadaddr, typically 0xbfc00000), usually by adding any new sections you find to either the .text, .data, or .bss section of the linker script if you need them or to remove those misbehaving sections. You can disassemble the ROM file also to see the various sections and add in the linker script the extra section, if you need that.

3. You want to pass arguments to the user main function.

   One possible approach to passing args to the vmips program would be to add a command-line option to vmips called "commandline" that represents the vmips program's command line. Then you could say $ vmips -o commandline="myprog -x 1234 foo.txt" myprog.rom.

   You will need to edit optiontbl.h to add the new option and its default value, and you will need to edit vmips.cc (or wherever is convenient) to

   1) parse the commandline option (whose value is machine->opt->option("commandline")->str) in order to extract argc, argv pair;
   2) copy the argv into vmips's physical memory using store words,
   3) initialize registers $a0 and $a1 as mentioned above giving in $a0 argc, and in $a1, the address of $a1;
   4) edit setup.S to refrain from zeroing out $a0 and $a1 which it do in the initialization part.

Another approach to passing args to the vmips program would be to edit the setup.S code and, right before where it says "jal FUNCTION" down at the very end, just add a call to something that sets $a0 to the argc value and $a1 to the argv pointer. You can write this to read arguments that the user types at the terminal, for example.

4. In function, undefined reference to '__gccmain'

   You need to change the function name main to main1 if you are linking with my arg2.c file and else entry in place of main function in your files (Appendix A2).

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

   If you have a main() function in your code, GCC will try to call __main or some other kind of setup function even if you use `-ffreestanding'. A simple workaround is to call the entry function entry instead of main.

4.6 To run programs

Step 0. If VMIPS is installed on your system, you can start building programs with it right away. Otherwise, you (or your system administrator) will have to compile VMIPS first; see the appendix on Installation (A6).

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

Step 2. Link your program with any support setup code (setup.s file). VMIPS comes with a linker script for simple standalone programs, which you can run with `vmipstool --link', or you can write your own linker script. Link also the library files which will be there in the cross compiler. Make sure that on the linker line you write name of setup code file first.

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

Step 4. Start the simulator using `vmips ROMFILE', where `ROMFILE' is the name of your ROM image. Your program should run to completion, and if you are using my setup

code, the simulator should halt when it hits the first break instruction, which should happen right after your entry function returns.

4.7 The files in the VMIPS I have changed for this work

As mentioned above, the setup code has been changed; the VMIPS handling of syscall has been completely changed in cpu.cc. Previously, when there was a syscall instruction, the VMIPS code returned Exception. Now, the syscall is handled. Also, the code in vmipstool.cc has been fully changed. The approach of threads which VMIPS uses has been changed to a better approach, and the misbehaving .lit8 section has been removed during objcopy as .lit8 is for 64-bit floating point constants which are not required for MIPS R3000 simulator (in vmipstool.cc in the objcopy command added –remove-section=.lit8). A lot of new files are created for implementation of various functions like lib.c, flib.c, read.c, arg2.c etc. These contain implementation of many functions using basic functions. (Appendix A8)

5. Profiler

5.1 Overview

Profiler is an automated configurable tool that scans the executed program's instructions and guides you through the process of defining systems that you want to monitor.

Profiler is built to extract the information from the user programs so that we know what are the main weaknesses in the hardware/software, which are to be taken care of to improve the efficiency of the hardware. With the profiler, we can take VLSI decisions early in the design flow. We can find issues, solutions and innovations and finding solutions by software with lesser cost involved.

5.2 Work done

I implemented some statistics in VMIPS. Some statistics are standard (like the first six) and are enough to characterize an application but the profiler can provide much more the accurate information according to the user needs. Many other statistics are a part of Xpresso Project and the goal is to find the weaknesses in the compiler chain and processor architecture in order to improve performance. The details of the statistics are:

5.2.1 Number of load instructions:

There are 7 types of load instructions, namely load byte, load byte unsigned, load halfword, load halfword unsigned, load word, load word left, load word right. For these 7 types of instructions, 7 variables are used to count the number of that particular category of load instructions. These seven variables num_lbinstrs, num_lbuinstrs, num_lhinstrs, num_lhuinstrs, num_lwinstrs, num_lwlinstrs, num_lwrinstrs respectively all of type unsigned 64 bit integers. Then the variable totalload in vmips.cc sums all these variables. When we perform some load instruction, the count relating to that particular load instruction is incremented.

5.2.2 Number of store instructions

The variable used is numstoreinstrs. This is defined in cpu.h, incremented when there is any store instruction executed in cpu.cc and used for giving output to file or not in vmips.cc. The options are configurable. The options are defined in optiontbl.h and the variables to identify the options are defined in cpu.h and vmips.h, and they are used in cpu.cc and vmips.cc files. With these options, we can decide what all statistics to print and so the calculations will be oriented to printing those statistics.

5.2.3 Number of branch instructions that are taken

The variable used is numbds. This is defined in cpu.h, incremented when there is any branch is taken in cpu.cc and used for giving output in vmips.cc

5.2.4 Number of branch instructions that are not taken

The variable used is branchesnottaken. This is defined in cpu.h, incremented when there is any branch is not taken in cpu.cc and used for giving output in vmips.cc

5.2.5 Number of branch instructions

This is the sum of the variables in part 3 and part 4, and so no different variable is defined for this.

5.2.6 Number of jump instructions

The variable used is numjds. This is defined in cpu.h, incremented when there is any jump instruction executed in cpu.cc and used for giving output in vmips.cc

5.2.7 Number of J instructions

The variable used is numjinstrs. This is defined in cpu.h, incremented when there is any J instruction executed in cpu.cc and used for giving output in vmips.cc

5.2.8 Number of JAL instructions

The variable used is numjalinstrs. This is defined in cpu.h, incremented when there is any JAL instruction executed in cpu.cc and used for giving output in vmips.cc

5.2.9 Number of load delay slots not filled

The variable used is numlds. This is defined in cpu.h, incremented when there is any load instruction followed by a NOP operation is executed in cpu.cc and used for giving output in vmips.cc. The NOP instruction is found out when there is SLL R0 R0 0, means R0 is shifted left by 0 and the result is put in R0. So, we need to check whether the present instruction is NOP, then if previous was a load instruction, then it is load delay slot that is not filled. To know the previous instruction, a variable pr is introduced. This variable stores the previous instruction was load, store, branch, jump or some other. For load, pr is set to “L” before leaving the function, and similarly “S” for store, “B” for branch, “J” for jump and so on. So, when we find a particular instruction as NOP (by detecting SLL R0 R0 0), if pr was “L”, it is a load delay slot.

5.2.10 Number of branch delay slots not filled

The variable used is numbdsnotfilled. This is defined in cpu.h, incremented when any branch instruction followed by NOP is executed in cpu.cc and used for giving output in vmips.cc. Again pr variable is used as explained before.

5.2.11 Number of jump delay slots not filled

The variable used is numjdsnotfilled. This is defined in cpu.h, incremented when any branch instruction followed by NOP is executed in cpu.cc and used for giving output in vmips.cc. Again pr variable is used as explained before.

5.2.12 Number of consecutive lwl-lwr instructions loading the same register

The aim of calculating this statistic is to calculate how many times we can change lwl and lwr instructions into a lw. This will reduce the number of clock cycles executed.

The variable used is consecutivelwrlwl. The other variables used are pr1 and lwcheck. pr1 is similar to pr, pr helps to distinguish between different types of instructions into load, branch etc. It distinguishes the instructions in the same category, eg- lwl and lwr instruction,backward branch and forward branch instruction.lwcheck is used when there is lwl or lwr instruction, lwcheck is put equal to the register in which contents are loaded.If we reach lwl after lwr (can be checked by seeing the value of pr1) and if that loaded the same register as the present instruction is doing(can be checked by comparing rt(instr) with lwcheck), consecutivelwrlwl is incremented in cpu.cc.

5.2.13 Check if consecutive branch/jump instructions occur

The aim of calculating this is to check the jump optimization of the compiler as if there is a jump and on the target is branch or jump then that could better have been reduced to a single branch or jump.

The variable used is consecutivebj. This is defined in cpu.h, incremented when present instructions being executed is branch or jump and previous was also branch or jump (can be known by pr) in cpu.cc and used for giving output in vmips.cc.

5.2.14 Number of backward going branches

The branch predictor can be made in a way that all backward going branches are taken and the forward going branches are not taken, as is the case with the Xpresso. For this, to calculate percentage of correct prediction, we need to calculate how many branches were backward going and how many forward going and how many of each was taken. This is why these four statistics.

The variable used is backbranch. This is defined in cpu.h, incremented when the target of the present branch instruction being executed is less than the present program counter in cpu.cc and used for giving output in vmips.cc.

5.2.15 Number of forward going branches

This is the difference between total branches found before and backward going branches found before. So, no new variable is defined.

5.2.16 Number of backward going branches taken

The variable used is backbranchtaken. This is defined in cpu.h, incremented in cpu.cc when the target of the present branch instruction being executed is less than the present program counter and the branch is taken and used for giving output in vmips.cc.

5.2.17 Number of forward going branches taken

This is the difference between total branches taken found before and backward going branches taken found before. So, no new variable is defined.

5.2.18 Number of backward going branch delay slots not filled

This statistic is also needed if the branch predictor has to do differently with forward and backward branches.

As explained before, pr1 distinguishes between backward branch and forward branch also. So, if we are presently executing a NOP operation and the previous was a back branch, this is incremented.

5.2.19 Number of forward going branch delay slots not filled

This is the difference between total branch delay slots not filled and the backward going branch delay slots not filed.

5.2.20 Number of consecutive loads

This statistic is designed as if the processor uses 2 separate memories so that one is accessed for odd parity and other for even parity, we can do both the operations together. So, to know how much we gain by dividing the memory access, we need this statistic. The purpose of next five statistics is also the same.

If the previous instruction was a load and present is also load and accessed physical memory with different parity, then there is a parity change. For this, I defined a variable parity (which is either zero or four depending on the physical address/4 being even or odd). As we want that 3 loads together with 1^{st} and 2^{nd} in different parity and 2^{nd}

and 3rd with different parity must be counted as 1 consecutive and not 2 consecutive and so on. So, I made another boolean variable flagload. Flagload is true initially. If the present is load, I check if previous is load, then if prevprev is load, then I invert the flagload else make it true. Now, if flagload is true, I increment the counter consecutiveload which gives consecutive loads.

5.2.21 Number of consecutive stores

If the previous instruction was a store and present is also store and accessed physical memory with different parity, then there is a parity change. For this, I defined a variable parity (which is either zero or four depending on the physical address/4 being even or odd). I made another boolean variable flagstore. Flagstore is true initially. If the present is store, I check if the previous is store, then if prevprev is store, then I invert the flagstore else make it true. Now, if flagstore is true, I increment the counter consecutivestore which gives consecutive stores.

5.2.22 Number of consecutive memory operations

If the previous instruction was a memory operation and present is also memory operation and access physical memory with different parity, then there is a parity change. For this, I defined a variable parity (which is either zero or four depending on the physical address/4 being even or odd). I made another boolean variable flagmem. Flagmem is true initially. If the present is a memory operation, I check if the previous is also a memory operation, then if prevprev is also memory operation, then I invert the flagmem else make it true. Now, if flagmem is true, I increment the counter consecutivemem which gives consecutive memory operations.

5.2.23 Number of consecutive loads that have operands with different parity

If the present instruction is a load and also is the previous instruction, I check if the parity of the two is different. I have a boolean variable flagtestl which is false initially. If the parity is different and flagtestl is false, I set it to true and increment the counter numloadparity. Otherwise, it is set false.

```
if(opcode(instr)>= 32)&&(opcode(instr) <=38)&&(opcode(previnstr)>=32)&&(opcode
(previnstr)<=38)&& ( ((reg[rs(instr)] + s_immed(instr))%8) !=((reg[rs(previnstr)]+
s_immed(previnstr))%8) ) &&(!flagtestl)
{
      numloadparity++;
      flagtestl=true;
}
else flagtestl=false;
```

5.2.24 Number of consecutive stores that have operands with different parity

If the present instruction is a store and also is the previous instruction, I check if the parity of the two is different. I have a boolean variable flagtests which is false initially. If the parity is different and flagtests is false, I set it to true and increment the counter numstoreparity. Otherwise, it is set false.

5.2.25 Number of consecutive memory operations that have operands with different parity

If the present instruction is a memory operation and also is the previous instruction, I check if the parity of the two is different. I have a boolean variable flagtestm which is false initially. If the parity is different and flagtestm is false, I set it to true and increment the counter nummemparity. Otherwise, it is set false.

5.2.26 Number of consecutive loads using the same register and have different parity

The aim of this statistic is that if you have access to different memories for odd and even parity, you can do these two load instructions in one instruction. If they use the same register, this can be known statistically and we can do it by software without the need of hardware.

If the present instruction is a load and also is the previous instruction, I check if the parity of the two is different and they use the same register. I have a boolean variable flagtestload which is false initially. If the parity is different, both use same register and flagtestload is false, I set it to true and increment the counter load_num_parity. Otherwise, it is set false.

```
if((opcode(instr)>=32)&&(opcode(instr)<=38)&&(opcode(previnstr)>=32)&&(opcode(previnstr)<=38)) && (rs(instr)==rs(previnstr)) && ((s_immed(instr)%8)!=(s_immed(previnstr)%8)) && (!flagtestload)
{
        load_num_parity++;
        flagtestload=true;
}
else flagtestload=false;
```

5.2.27 How many instructions following the delay slot need to read the loaded register.

The aim of this statistic is that if in your design, the load delay slots increase to two, you will be requiring how many nops you have to insert, and that is equal to the number reading the loaded register after one instruction inbetween.

For this, I made a variable prevprev for storing previous to previous instruction, and a variable prevloadedregister and loadedregister to store the value loaded in this instruction and in the previous instruction. These values are updated in each instruction. Also, during execution of each instruction, we check if the previous to previous (prevprev) is load, and then if the prevloadedregister is being used now, and decide whether to increment the associated counter numreadlr which is also defined and used as other variables.

5.2.28 How many instructions following the load and nop pattern need to read the loaded register.

The aim of this statistic is to test the pipeline optimization of the compiler because the compiler-assembler must not insert nop if the next instruction is not reading the loaded register.

I made a record of seven previous instructions (actually didn't do for this purpose, did it for optimization statistics which will come later in the next chapter). These can be accessed from the step() function in cpu.cc where every instruction comes in and is directed to some place. These variables are updated and used in this function. The variables are instr, previnstr, prevprevinstr, prev3instr, prev4instr, prev5instr, prev6instr and prev7instr. These are the 32 bit instructions. From these I detect if the prevprevinstr was load, and previnstr was nop, then if the instr is using the loaded register of prevprevinstr, then the counter numnopreadlr is incremented.

5.2.29 Check whether the branch instructions and the instruction that filled its delay slot can be swapped

The aim of this instruction was to check the method by which the assembler works for branch delay slots. It checks the previous instruction. If that can be swapped with the branch, then it does that. So, the result is expected to be true. But it is false, the reason being that there are many macros, which expand in noreorder section with branch and in its delay slot a statement, which can't always, be swapped.

For this, I make a Boolean variable canbeswapped, and two unsigned int variables branchrs and branchrt, storing the two registers which branch used. In case, it used one, both are same. Now I initially put canbeswapped to true; when I find a case when those used registers is being written in the next instruction, I make it false, and output comes false. If such case never arises, output comes true.

5.2.30 Number of branches that have the same target

I made a vector vectbranch containing vectors of unsigned 64 bit integers. When a branch is encountered in the program, I check if the vectbranch is empty, I add a vector of two elements 1st the target of branch and second present program counter of branch to

the vectbranch. If it is not empty, I search through all vectors present, if the 1st element of each is the present target or not. If I find a target, I replace that vector by new vector containing one more element (the present pc) in the same position if it was not there presently otherwise I add a new vector of 2 elements (target and present pc) to the vectbranch. By doing this, I have made a structure, vector of vectors, which is a type of double array, in which the 1st column is the targets, and the rows are the program counters of the branches which had the target as the 1st element of the row. This structure is made and changed according to this procedure in cpu.cc file.

This statistic is obtained in vmips.cc file after the execution of the whole program. I go through each vector inside vectbranch and the number of elements of the vector-1 is the number of branches that pointed to the target, which is the first element of this vector.

5.2.31 Check whether a branch target filled a branch delay slot

This also uses the previous implementation of vectbranch. In vmips.cc, I scan the double array for the target and for each target, I go through each element of double array (leaving the target elements) and see if the target-branch pc =4, when it is 4, I say that branch target is filled in delay slot.

5.2.32 CPU time in clock cycles

This is required to compare the same program processed differently as to what approach on the program is better.

For this there is a variable num_instrs. This variable is incremented in vmips.cc in the step() function so that total executed instructions are counted. As we are working on processor with one instruction per clock cycle, so we get the number of clock cycles same as this number

5.2.33 NOP number

This is required to see the performance of assembler and the compiler.

I have used a variable numnops, which is incremented each time we get a nop in cpu.cc.

5.2.34 CPI of load, store, branch and jump operations

CPI is the clock cycles per instruction. We are using processor with one instruction per clock cycle, but due to pipeline hazards, we need to sometimes put nop after jump, branch or load. This increases the CPI from 1. This can be calculated as 1+(A delay slots not filled)/(Total number of A instructions) where A=load, branch or jump as the case is. As the store does not require nop after that, so its CPI is 1.

5.2.35 Overall CPI

Overall CPI is the overall clock cycles per instruction. You get total clock cycles, and the waste nop instructions, hence you know the useful instructions executed. The ratio of the two is the overall CPI.

5.2.36 Customizable displays and instructions to execute.

The aim of customization is to provide user possibility to get some statistics and not other. He can choose to execute only a fixed number of instructions of the program and so on. So, all these have been provided as command line options. The first option is statistic. If you disable this option, you will not get any statistic. For any statistic, you have first to turn on this option. By default, it is false. Then I have categorized the statistics in four categories- Instruction Count, Memory Instructions, Branch-Jump Instructions and MFMT Instructions (MF/MT HI/LO/C0). These four have four different options. Default they are all on. If you turn on the statistic, you get all these displayed on the file name given by option on command line (default results.txt). If you don't want a particular, you have to specify that on the command line or change its default value in optiontbl.h. The options I have nade are:

1. meminstrs: This is a Boolean option. If this option is set, it will display the memory instruction statistics.

2. brjuinstrs: This is a Boolean option. If this option is set, it will display the branch jump instruction statistics.

3. outfile: This requires a string input which is the name of output statistic file.

4. inst_number: This requires an integer as argument, which tells number of instructions to execute.

5. fastfwd This requires an integer as argument, which tells the starting instruction number from which inst_number will be calculated.

6. statistic: This is a Boolean option. You have to set this option to get statistics.

7. mfmt This is a Boolean option. If this option is set, it will display the statistics regarding mfhi,mflo,mtc0,mfc0 etc. instructions.

6. Compiler Optimizer

6.1 Overview:

Compilers are not nearly as clever as one might think. The compiler writer's first responsibility is to ensure that the generated code does precisely what the language semantics say it should; and that is hard enough. It has for long time been accepted good practice that cunning improvements be made as a sequence of transformations to an internal representation of the program, with each individual optimisation leaving a transformed but equivalent program. The optimizer I have made is based on this approach. This approach has the merit of allowing programs to be built either including or leaving out a particular optimisation step, thus containing the complexity of the compiler debugging job. Most compilers do the following. Occasionally, the assembler may perform some of these too.

1. **Common Subexpression Elimination** (CSE): This detects when the code is doing same work twice. It gives compiler the ability to globally optimize across the function. Most memory-reference optimization is actually done by CSE. The enemy of CSE is unpredictable flow of control: the conditional branch. CSE can really only operate inside basic blocks.

2. **Jump Optimization**: This removes redundant tests and jumps like jumps to jumps, jumps around unreachable code, redundant conditional jumps etc.

3. **Loop Optimization**: This studies loops in your code, starting with the inner one. Subexpressions that depend on variables that don't change inside the loop can be pre-computed before the loop starts. Expressions that depend in some simple way on a loop variable can be simplified. Loops can be unrolled, allowing the work of two or few iterations of the loop to be performed in line.

4**. Function Inlining**: Some small functions can be usefully expanded in a line, like a macro, rather than calling them. Some compilers may recognize the inline keyword to allow the programmer to specify which functions' invocations can be replaced by inlined code.

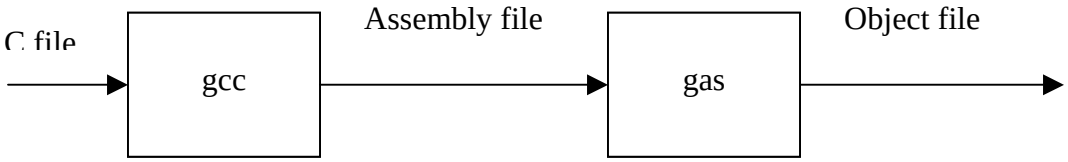
5. **Register Allocation**: Register allocator's job is to minimize the amount of work done in shuffling the data in and out of the registers; it does this by maintaining some variables in registers for all or part of a function's run time.

6. **Pipeline Reorganization**: The compiler or assembler can sometimes move the logical instruction flow around so as to make good use of the branch and load delay slots. This can be done only late in the compilation process.

6.2 Design Flow for the Optimizer

We have decided to work at the micro-architecture level (in other terms, at the assembler level) and allow any compilation option. However, the optimizer has been developed under the use of most efficient option in terms of speed (i.e. O3). The profiling part is important so that we can test the various optimization techniques and choose the best before actually implementing it. I planned to optimize by reorganizing the instructions inside the pipeline. In this approach, I try to find an instruction, which could be put in a delay slot even if the assembler is not able to do it. What the gas assembler does in case of delay slots is that it checks load with the next instruction. If the next reads the loaded register, it puts a NOP in-between. In case of branch and jumps, it checks the previous instruction, if it can be swapped with branch, if not it inserts a NOP.

Now the problem is where do I put my optimizer in the compiler design flow. The design flow is as follows:



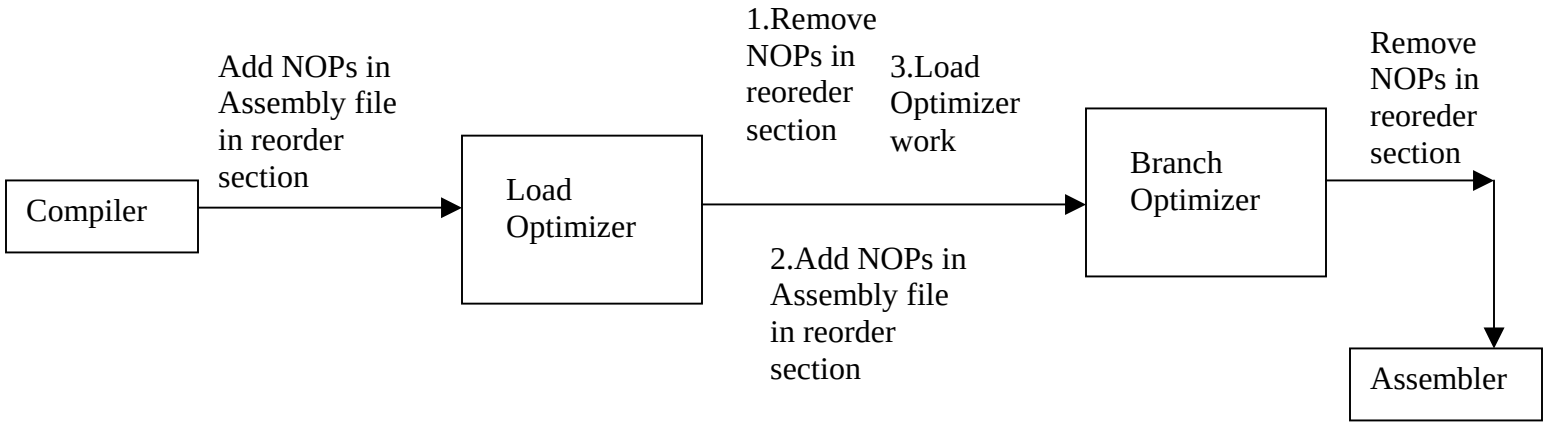
The input to the cross compiler is the C file, which goes through gcc to give assembly file. The assembly file then goes to the assembler (gas), which converts the assembly file to an object file. The object files can be linked together to make an executable. Then you can execute it on your simulator, eg. For VMIPS, make a rom file from this executable and then run it. In this flow, the gcc tries to do all the optimisations except the pipeline reorganization, which is done by the assembler. The pipeline reorganization that the assembler does has been explained before. We want to improve on the reorganization. So, the possible paths for implementing the optimizer can be

1. To go inside the assembler code and get one more output assembly file with all the work it has done incorporated, and then optimize this and then again give the file to the assembler as a sort of feedback.
2. The place where the assembler decided to put the NOPs by checking the dependencies is in append_insn() function in the assembler code in tc-mips.c in the config directory inside the assembler. The approach can be to check more dependencies and change the order of instructions and not to put NOP if some other instruction can be put.
3. Make a standalone tool in-between the gcc and gas.

We worked with a standalone tool as this has many advantages. The user may want that he wants his executions very fast. He does not care about the optimisations. That is why also the compiler options –O2 –O3 –Ofast etc. has been given as options that you can use them for optimisation, but if you use them, program takes more time to compile. So, with a standalone tool, the user has the benefit that he can choose whether to use the optimizer or not. The major advantage lies in the fact that if there is a bug in the

code, you can detect where actually the bug is if you have standalone tools. You do remove standalone tools one by one and find the tool that did not perform well. If you implement the optimizer code inside the assembler itself, it will become difficult to debug and to change the assembler for some other architecture.

With the profiler, we found that the delay slots of loads and branch-jumps are the most important to handle. By handling other delay slots (of mfhi,mflo etc.) , you get almost no optimisations due to lesser instructions executed. What I planned is to make the optimizer consisting of different standalone tools: The load optimizer, the branch and jump optimizer so that it will be easy to debug by enabling one at a time and then together. The load optimizer will first put all NOPs after jumps, branches, loads, mfhi, mflo, mfc0, mtc0 instructions in the reorder section and then will try to remove NOPs and move the instructions around for doing that and in the end I remove all NOPs in the reorder section to let assembler do so that if I made a mistake, it will still correct it. The second stage is the branch and jump optimizer. This will put NOP after jump, branch,load etc. as previous. But as this stage is after load, this will first remove NOPs after load, in which the next instruction does not use the loaded register to incorporate the load optimisation that will be done now by the assembler (Load Optimizer work). Although this involves a bit of repetition here, but that is the price we have to pay if we want to keep the code not complex and easy to debug. Then we optimize for branch and jumps and again remove NOPs in the reorder section. I chose perl language for the optimizer.



6.3 Load Optimizer

Profiler for Load Optimizer:

I first started profiler for optimising the load. I tried to check two instructions after the NOP and two instructions below the load if any can be put in the delay slot. To implement this statistic, as I also mentioned in the previous chapter, I create a database of 8 instructions, so that in each instruction, I can see what all instructions were executed till 7 instructions ago. The names of variables are instr, previnstr, prevprevinstr, prev3instr ,

prev4instr, prev5instr, prev6instr and prev7instr. I check if the prev4instr is a load and prev3instr is a NOP, then I check if prevprevinstr (instruction after NOP) could be put inside NOP, if not, we check if previnstr (2 instructions after NOP) can be put in NOP, and similiarly continue for prev5instr (instruction before NOP) and prev6instr (2 instructions before NOP). I have for checking dependencies, made all equations as userd() returns the written register, and users() and usert() the two used registers. For prevprevinstr, I check if this instruction reads the loaded register. For previnstr, I check if previnstr and prevprevinstr can be swapped, and if prevprevinstr uses loaded register, and previnstr or prevprevinstr must not be jump or branch, and if prevprevinstr is load, then the present instruction must not use its loaded register. For the prev5instr, I check if this and prev4instr (load) have any dependency, prev5instr or prev6instr must not be branch or jump, and if prev5instr or prev6instr is a load, then the instruction that will now come in its respective delay slot must not use the respective loaded register. For the prev6instr also, we check dependency of prev6instr with prev5instr and prev6instr with prev4instr, and prev5instr, prev6instr or prev7instr must not be a branch or jump, and if prev6instr or prev7instr is a load, then the instruction that will now come in its respective delay slot must not use the respective loaded register.

Load Optimizer:

The implementation of this was on the same algorithm as the profiler. First of all I inserted the NOP after load, branch and jumps. Then I made an array filecontent with the lines of the file not containing blank lines and the lines with # at the beginning. From this array, I made seven more arrays, which identify the instruction. With these, I am able to categorize line (into label, load, nops, branch-jumps, labels, macros, etc.), say if it is instruction or not, get the written and the used registers, say whether the instruction is in reorder or noreorder section, in macro or nomacro section. But I can’t reconstruct the line from these seven properties as is obvious. These seven are just for helping in decision making of whether a particular instruction can be put in the delay slot.

I go through the $index number of lines and check if the present instruction is a load, and if so we get the index of the load instruction in all arrays as $i. Then I search for next instruction. If it is not a NOP, we have nothing to do. If it is a nop, we save the index of nop in $j. We continue forward, if we find an instruction ($k) before any .end or any label, then we check its dependency with load. In case of no dependency, we remove the NOP. If we are not able to remove the NOP and we have not yet encountered .end or a label which may be target of some branch or jump (in case we encounter them, we will continue backward instructions and not forward) we get the next instruction ($l). Now, if we can find next instruction ($m), we see that if $l can be put in nop with different dependency checks than when we were not able to find $m, due to Labels or .end and with the dependency checks, we detect the sections and try to remove the NOP efficiently. In some cases, we did not do anything like if anything other than .set reorder/ noreorder/ macro/ nomacro comes in-between; we leave the NOP unchanged to have a safer solution (So, an improvement can be done). Similar is done for one more instruction ahead. Then if till now also, we are not able to fill the delay slot, we move backward from $i to see if the instructions before that could be put in the delay slot. The

similar steps are done for previous instructions using the same algorithm as explained in the profiling part. With the program, you can easily check for more instructions for above and below and do the work. I have also extended the program for one more instruction forward so that anyone using the tool can easily see and extend the tool to his need. So, the scope of load optimizer is 3 instructions forward and 2 instructions backward. Many functions have been made in general so that anyone can easily extend this. After this, I write the lines in the output file except the lines which have NOP in reorder section.

6.4 Branch-Jump Optimizer

Profiler for Branch-Jump Optimizer:

For branch-jump optimizer also, I first made the profiler. The same variables for the instructions are used here also. I check if the previous instruction is a Nop and the instruction before it(prevprevinstr) is a jump or branch, I first check if prev3instr can be put in the nop, for that I check dependency of prev3instr with prevprevinstr and then check that prev3instr or prev4instr must not be branch or jump. Then, if prev4instr is a load, then the prevprevinstr must be able to be put inside the delay slot. If the prev3instr is itself a load, we can put it in the delay slot if the present executed instruction can be inside its delay slot but during static case of making the optimizer, we have to find the target to make possible decision for putting this load in the branch delay slot. If it is a load that we have thought in execution stage that we can put inside branch/jump delay slot by its dependency with present instruction, a counter is incremented to see the possibility of improvement that is maximum possible if we try to find the target of branch and deciding whether the load can be put in the delay slot.

Branch-Jump Optimizer:

For the optimizer, the assembly program is passed through a perl script which adds NOPs after load, branch, jump, mflo,mfho,mtc0,mfc0 . This script is the same as is used as the first step of the load optimizer. Then I made an array filecontent with the lines of the file not containing blank lines and the lines with # at the beginning. From this array, I made seven more arrays, which identify the instruction. With these, I am able to categorize line (into label, load, nops, branch-jumps, labels, macros, etc.), say if it is instruction or not, get the written and the used registers, say whether the instruction is in reorder or noreorder section, in macro or nomacro section. These seven arrays are helpful in decision making of whether a particular instruction can be put in the delay slot. Then I remove NOPs after load, in case the instruction after NOP do not use the loaded register to account for the load optimisation that will be done by the assembler using the similar procedure as explained in Load Optimizer.

After this, when I find a pattern Branch or Jump followed by NOP, I get branch/jump line number in $i and NOP line no. in $j. Then I search for the next instruction after NOP in $k, and instructions before load in $h,$g,$f,$e with $h as instruction just before the branch. If the instruction at $k is a NOP, I remove the line and

redo the process on this branch. Then I check which instruction can be put in the delay slot not considering loads can be put in the delay slot. After I have found such instruction, I do handle it taking into account the reorder and noreorder sections. If the branch is in reorder section, just put the instruction before branch and remove the NOP if it is in no-reorder section. If branch is in noreorder section, put the instruction to be put in delay slot and the branch in reorder section. Doing this, we use the assembler to put the instruction in the delay slot as we are going to assemble the file, and we have found the instruction that could be put in the delay slot by the assembler and let the assembler put the instruction in the delay slot. After the work of branch-jump optimizer is over, we remove the NOPs in the reorder section and give the file to the assembler.

7. Benchmarks

Recently, it has become popular to put together collections of benchmarks to try to measure the performance of the processors with a variety of applications. Of course, such suites are only as good as the constituent individual benchmarks. Nonetheless, a key advantage of such suites is that the weakness of one is lessened by the presence of the other benchmarks. Benchmarks for embedded systems are in far more nascent state than those for either desktop or server environments [4]. I have used the benchmarks in my project to validate O.S, Profiler and Compiler Optimizer and the complete flow. I used two benchmarks- SPEC CPU2000 (Integer Benchmarks) and EEMBC. Both work completely with Simulator, Profiler, Operating System and the Compiler Optimizer. VMIPS, and more precisely complete toolchain with all the tools made have been validated with the benchmarks.

7.1 SPEC CPU2000

The SPEC benchmarks are real programs, modified for portability and to minimize the role of I/O in overall benchmark performance. The **integer** benchmarks vary from part of a C compiler to a VLSI place-and-route tool to a graphics application. The floating-point benchmarks were not used, as MIPS R3000 does not support floating points. The approach in which VMIPS, and all the tools were used to run the SPEC benchmarks is given below. You need to adjust the directories while using it according to your files and the OBJ values. Launch as gmake –f makefile clean link make-rom exe after copying the read.c, arg2.c and errno.c files in the SPEC benchmark directory

Makefile: (for MCF Program of SPEC CPU2000-INT)

```
CFLAGS=-S -O3 -mno-gpopt –G 0

OBJ=  mcf.o mcfutil.o readmin.o implicit.o pstart.o output.o \
          treeup.o pbla.o pflowup.o psimplex.o pbeampp.o read.o arg2.o errno.o
```

(All the object files needed. This also uses 3 files read.o arg2.o errno.o which are in VMIPS directory and you need to copy these inside the same directory (Appendix A2))

```
.c.o:  #Compiles all the C files in the program directory
        vmips-1-1-3/vmipstool --compile $(CFLAGS) $< -o tret34.s
        perl t2.perl   tret34.s out2.s
```

(All the hard-coded names are arbitrary. If you are using any of these in your program, do make a change here. (See Appendix A2))

```
        perl t3.perl out2.s out3.s 1
        rm tret34.s
        rm out2.s
        perl t5.pl out3.s out4.s
        rm out3.s
        perl t3.perl out4.s out3.s 1
```

```
		rm out4.s
		perl tj.pl out3.s out4.s (These run the optimizer as is explained in the appendix A2)
		/home/aggarwal/TOOLS/vmips-1.1.3/vmipstool --assemble –G 0 -o $@ out4.s
		rm out4.s
		ln -s -f `pwd`/$@ vmips-1.1.3

	setup.o:
		vmips-1.1.3/vmipstool --assemble –G 0 -o setup.o vmips-1.1.3/setup1.s
		ln -s -f `pwd`/*.o /home/aggarwal/TOOLS/vmips-1.1.3

	link: setup.o ${OBJ}
		cd vmips-1.1.3/; vmipstool --link -o mcfp setup.o arg2.o ${OBJ} read.o errno.o ../mipstoolchain/mipsel-elf/lib/libc.a ../mipstoolchain/mipsel-elf/lib/libidt.a

	make-rom:
		vmips-1.1.3/vmipstool --make-rom vmips-1.1.3/mcfp mcfperlj.rom
	clean:
		rm *.o
		rm mcfperlj.rom
		cd vmips-1.1.3/; rm ${OBJ}
		cd vmips-1.1.3/; rm mcfp
	exe:
		vmips-1.1.3/vmips -o noinstdump -o romfile=mcfperlj.rom -o outfile=resultprintnet2.txt -o statistic (Running VMIPS as explained in appendix A2)
```

7.2 EEMBC

For those embedded applications that can be characterized well by kernel performance, the best-standardized set of benchmarks appears to be the EDN Embedded Microprocessor Benchmark Consortium (or EEMBC pronounced “embassy”). Although many embedded applications are sensitive to the performance of small kernels, remember that often the overall performance of the entire application is also critical. Thus, for many applications, the EEMBC benchmarks can be used to partially access performance. The script for using EEMBC with the full toolchain, including all the tools made by me is there in the appendix A3. EEMBC benchmarks involve some floating-point calculations for time calculations and displays, which can be avoided by changing one option (FLOAT_SUPPORT) in EEMBC (See Appendix A3). As MIPS R3000 does not support floating-points, the information that is avoided can be extrapolated using profiler. But, the time given by profiler is time for it to execute the program, which includes many other calculations also, and not between given two instants inside the user program, a profiler time result is higher. Number of Iterations is given by running EEMBC, and the time by profiler. You can get time per iteration and iterations per second by using them.

8. Conclusions

This chapter discusses the achievements, and the way this work can be extended and used.

8.1 Virtual Operating System

The Virtual Operating System works on all the tested programs including SPEC 2000 benchmarks and EEMBC Benchmarks. With the help of Virtual Operating system, user can use in his assembly fine any library function now and can simulate his design. My virtual operating system will work for all instructions supported by the cross-compiler toolchain, and not only on the instructions supported by MIPS R3000.

8.2 Profiler

The profiler for programs running on MIPS R3000 processor has been implemented and can easily be extended.

8.3 Compiler Optimizer

The compiler optimizer is fully implemented, and my compiler optimizer tries to find instruction for delay slot more efficiently than the assembler. The optimisations achieved differ from program to program and rough estimate of the optimisations possible if you use the tools can be got from the profiler. Actually, load optimisation seems efficient to have the most important impact.

8.4 Benchmarks

Both benchmarks (SPEC CPU2000-INT and EEMBC) work completely with Simulator, Profiler, Operating System and the Compiler Optimizer. VMIPS, and more precisely complete toolchain with all the tools made have been validated with the benchmarks. Some programs use the instructions not supported by MIPS R3000 (like the CP1 instructions eg. lwc1, swc1 etc.). The simulator in the case displays the warning, and it is not recommended to use the optimizer on such programs.

8.5 Future Work

The Virtual Operating system can be easily extended for other processors using different ISAs but the processors derived from processors using MIPS1 ISA can use the

same operating system. The profiler algorithm can be easily adapted to other processor simulators. For compiler optimizer, I have optimized using Pipeline Reorganization. This can be increased by considering the targets of branches and enabling putting loads sometimes in branch-jump delay slots identifying possible targets and checking dependency of load with all (1 or 2 as the case may be depending on the instruction like j, jal you need to check only with target and for branches, the next instruction and the target both) possible next instructions that can be executed (this is not applicable for jr and jalr as for jr and jalr, you can not know the target in static case). This statistic is also provided in the profiler. Since the statistic was too low on the tested programs, I did not implement it. If in some specific application, this statistic is high, one can implement this. Also, one can optimize using other approaches besides pipeline reorganization (Section 8.1). Compiler Optimizer can be extended for other architectures as well. For that, you need to add in the list of the available function definitions the definitions of written and used registers of the new instructions, and that alone will work(Appendix A9).

Appendix

A1: Cross Compiler Installation Flow

This document describes how to built a MIPS cross compiler toolchain, i.e. a cross compiler (GCC), an assembler (GAS), a linker (GLD), a library (newlib), various utilities (Binutils) and a debugger (GDB). [13]

A1.1 WHAT WE NEED

Compiler : GCC 2.95.2, file gcc-2.95.2.tar.gz
Assembler : GAS 2.13.1/GLD 2.13.1, file binutils-2.13.tar.gz)
Debugger : GDB 6.0, file gdb-46.0.tar.gz
Library : Newlib 1.12.0, file ftp://sourceware.cygnus.com:/pub/newlib/

A1.2 INFORMATION ABOUT MY WORK ENVIRONMENT

Operating system : solaris 2.8
Host processor type : sparc-sun-solaris2.8
C++ and C compiler make and version : 2.95.2 2

A1.3 INSTALLATION FLOW

The toolchain is created in two passes : the first pass creates a bootstrap compiler because it doesn't contain the header files for the targeted architecture and the second pass builds the cross compiler by using the complete set of target-specific header files elaborating during the previous step . Global installation scheme:

1. Binutils
2. GCC
3. Newlib
4. Binutils
5. GCC
6. Newlib
7. GDB

Flow optimization : this scheme may be improve by doing 1. Binutils (--with-headers), 2. GCC (--wihtout-headers), 3. Newlib, 4. GGC (with-headers), 5. GDB.

A.1.3.1 Initialization

1. Put the binutils bin directory in your PATH. For instance, set PATH = (/home/aggarwal/TOOLS/mipstoolchain/ bin $PATH) in your .cshrc file or setenv

PATH /home/aggarwal/TOOLS/mipstoolchain/ bin:${PATH} in a terminal (for a tcshell environnment) .

2. Set the CC environment variable to gcc, as setenv CC gcc

3. Unpack all the sources (tar xvfz *.tar.gz) in your home directory, say in /home/aggarwal/TOOLS

4. chose the place where the cross compiler is going to be installed (ex. /home/aggarwal/TOOLS/mipstoolchain)

5. mkdir build-binutils build-gcc build-newlib build-gdb in home directory , say in /home/aggarwal/TOOLS

6. Check that make is in your PATH or add it (setenv PATH /usr/ccs/bin:${PATH}) The best way to be sure to have the right setting is to build an appropriate .cshrc file for this last point and the two first one.

A.1.3.2 Configure, build and install binutils

1. cd build-binutils
2. ../binutils-2.13/configure --target=mipsel-elf --prefix=/home/aggarwal/TOOLS/mipstoolchain --without-headers --with-newlib --enable-languages=c –v
3. gmake all
4. gmake install

A.1.3.3 Configure, build and install gcc without headers

gcc will be configure as a bootstrap compiler : it will not take into account the target-specific library and header files thanks to the configure option --without-headers. The bootstrap compiler will allow to build the specific library and header files for the targeted architecture during the newlib build process.

1. cd ../build-gcc
2. ../gcc-2.95.2/configure --target=mipsel-elf --prefix=/home/aggarwal/TOOLS/mipstoolchain --with-gnu-as --with-gnu-ld --without-headers --with-newlib --enable-languages=c –v
3. gmake LANGUAGES=c all-gcc
4. gmake LANGUAGES=c install-gcc

A.1.3.4 Configure, build and install newlib

Specific-library and header files construction:

1. cd ../build-newlib

2. ../newlib-1.12.0/configure --target=mipsel-elf --prefix=/home/aggarwal/TOOLS/mipstoolchain --with-gnu-as --with-gnu-ld --enable-languages=c -v
3. gmake CFLAGS=" -O3 -G 0" all
4. gmake CFLAGS=" -O3 -G 0" install

A.1.3.5 Configure, build and install gcc without headers

gcc will be configure as a cross compiler : it will take into account the target-specific library and header les thanks to the configure option --with-headers.

1. cd ../build-gcc
2. ../gcc-2.95.2/configure --target=mipsel-elf --prefix=/home/aggarwal/TOOLS/mipstoolchain --with-gnu-as --with-gnu-ld --with-headers --with-newlib --enable-languages=c –v
3. gmake LANGUAGES=c all-gcc
4. gmake LANGUAGES=c install-gcc

A.1.3.6 Configure, build and install gdb

1. cd ../build-gdb
2. ../gdb-6.0/configure --target=mipsel-elf --prefix=/home/aggarwal/TOOLS/mipstoolchain --with-gnu-as --with-gnu-ld --enable-languages=c –v
3. gmake all install

A2: Complete VMIPS Flow to Run Programs with Profiler, Operating System and Optimizer

Profiler: This can be used in the execution command line to VMIPS. You launch –o statistic and you get all the statistics and you can choose which all you want also from the command lines.

Operating System: For operating system, you need to include the required libraries in the link step and also the files read.c made by me, the errno.c from newlib. Your main function has to be defined entry in case you are using only these two files. In this case, you will not be able to provide the arguments to your function. In case you need to pass arguments, use the arg2.c file and put the arguments in that file and then you can customize the function name in your own file, you can change main to any other name as you have to specify in arg2.c file, say main1. But, this must not clash with main, entry or any function inside the program or the libraries. arg2.c has in it the implementation of entry which calls the main1 function with arguments. So, if you use this, the main of your programs should be replaced with main1. The example contents of arg2.c file are:

```
int entry(void)
{
    //For MCF program of SPEC CPU2000
    char * out[2];
    out[0]="vmips";
    out[1]="../data/test/input/inp.in";
    main1(2,out);
}
```

Optimizer: This is a set of perl programs on the assembly file generated. So, you need to make an assembly file from C file. Then launch the optimizer to give the output assembly file. Then, assemble the assembly file, which is output of assembler to the required object file. Then, collect all output object files to make a big object file, from which make a rom image and execute with VMIPS. The tasks performed by different optimizer files are:

t2.perl: Removes all C and C++ style comments. This requires two arguments - the input file and the output file.
t3.perl: This requires 3 arguments, input file, output file and delay slots. The program puts delay slot times nop after every load, jump, branch in reorder section and 2* delay slot times nop after mtc0,mfhi,mflo,mfc0 in reorder section
t5.pl: Removes blank lines and the lines starting with # and do load optimisation. This requires two arguments - the input file and the output file.
tj.pl: Removes blank lines and the lines starting with # and do branch-jump optimisation. This requires two arguments - the input file and the output file.

So, the flow of optimizer is:
perl t2.perl <inassemblyfile> see1.s

perl t3.perl see1.s see2.s 1
perl t5.pl see2.s see1.s
perl t3.perl see1.s see2.s 1
perl tj.pl see2.s < outassemblyfile>

So, the **complete flow** can be given as:

1. Assemble setup code

vmipstool --assemble –o setup.o setup1.s

2. Compile your program and also the read.c, arg2.c and errno.c files

vmipstool --compile –S –O3 –o <inassemblyfile> <in C file>
perl t2.perl <inassemblyfile> see1.s
perl t3.perl see1.s see2.s 1
perl t5.pl see2.s see1.s
perl t3.perl see1.s see2.s 1
perl tj.pl see2.s < outassemblyfile>
vmipstool --assemble –o <outobjectfile> < outassemblyfile >

3. Link all files with setup code the first on the line and all the libraries needed by your program

vmipstool –link –o <net object file> setup.o <All Object files> -lc –lidt –lm

4. Build a ROM image from the executable:

vmipstool –make-rom <net object file> <rom file>

5. Run the program with statistics

vmips <rom file> -o statistic

You can see all the options to VMIPS and their default values in optiontbl.h file.

Example: (Input file is test.c)
./vmipstool --assemble -o setup.o setup1.s
./vmipstool --compile -S -O3 test.c –o optimize.s
perl t2.perl optimize.s see1.s
perl t3.perl see1.s see2.s 1
perl t5.pl see2.s see1.s
perl t3.perl see1.s see2.s 1
perl tj.pl see2.s optimize.s
vmipstool --assemble –o test.o optimize.s
./vmipstool --compile -S -O3 read.c –o optimize.s
perl t2.perl optimize.s see1.s
perl t3.perl see1.s see2.s 1
perl t5.pl see2.s see1.s
perl t3.perl see1.s see2.s 1
perl tj.pl see2.s optimize.s
vmipstool --assemble –o read.o optimize.s

vmipstool --compile -S -O3 arg2.c –o optimize.s
perl t2.perl optimize.s see1.s
perl t3.perl see1.s see2.s 1
perl t5.pl see2.s see1.s
perl t3.perl see1.s see2.s 1
perl tj.pl see2.s optimize.s
vmipstool --assemble –o arg2.o optimize.s
./vmipstool --compile -S -O3 ../newlib-1.12.0/newlib/libc/errno/errno.c c –o optimize.s
perl t2.perl optimize.s see1.s
perl t3.perl see1.s see2.s 1
perl t5.pl see2.s see1.s
perl t3.perl see1.s see2.s 1
perl tj.pl see2.s optimize.s
vmipstool --assemble –o errno.o optimize.s
./vmipstool --link -o netout setup.o arg2.o test.o read.o errno.o -lc –lidt
#You can also used the compiled libraries, Appendix A4
rm hello.rom
./vmipstool --make-rom netout hello.rom
./vmips -o noinstdump -o romfile=hello.rom -o outfile=resultprintnet2.txt

So, the **complete flow** if you are using second approach for OS can be given as:

1. Assemble setup code
vmipstool --assemble –o setup.o setup2.s

2. Compile your program including scan.c, lib.c, flib.c files.
vmipstool --compile –S –O3 –o <inassemblyfile> <in C file>
perl t2.perl <inassemblyfile> see1.s
perl t3.perl see1.s see2.s 1
perl t5.pl see2.s see1.s
perl t3.perl see1.s see2.s 1
perl tj.pl see2.s < outassemblyfile>
vmipstool --assemble –o <outobjectfile> < outassemblyfile >

3. Link all files with setup code the first on the line
vmipstool –link –o <net object file> setup.o <All Object files> scan.o lib.o flib.o read.o

4. Build a ROM image from the executable:
vmipstool –make-rom <net object file> <rom file>

5. Run the program with statistics
vmips <rom file> -o statistic
You can see all the options to VMIPS and their default values in optiontbl.h file.

Example: (Input file is test.c)
./vmipstool --assemble -o setup.o setup2.s
./vmipstool --compile -S -O3 test.c –o optimize.s
perl t2.perl optimize.s see1.s
perl t3.perl see1.s see2.s 1
perl t5.pl see2.s see1.s
perl t3.perl see1.s see2.s 1
perl tj.pl see2.s optimize.s
vmipstool --assemble –o test.o optimize.s
./vmipstool --compile -S -O3 read.c –o optimize.s
perl t2.perl optimize.s see1.s
perl t3.perl see1.s see2.s 1
perl t5.pl see2.s see1.s
perl t3.perl see1.s see2.s 1
perl tj.pl see2.s optimize.s
vmipstool --assemble –o read.o optimize.s
vmipstool --compile -S -O3 arg2.c –o optimize.s
perl t2.perl optimize.s see1.s
perl t3.perl see1.s see2.s 1
perl t5.pl see2.s see1.s
perl t3.perl see1.s see2.s 1
perl tj.pl see2.s optimize.s
vmipstool --assemble –o arg2.o optimize.s
vmipstool --compile -S -O3 lib.c –o optimize.s
perl t2.perl optimize.s see1.s
perl t3.perl see1.s see2.s 1
perl t5.pl see2.s see1.s
perl t3.perl see1.s see2.s 1
perl tj.pl see2.s optimize.s
vmipstool --assemble –o lib.o optimize.s
vmipstool --compile -S -O3 flib.c –o optimize.s
perl t2.perl optimize.s see1.s
perl t3.perl see1.s see2.s 1
perl t5.pl see2.s see1.s
perl t3.perl see1.s see2.s 1
perl tj.pl see2.s optimize.s
vmipstool --assemble –o flib.o optimize.s
vmipstool --compile -S -O3 scan.c –o optimize.s
perl t2.perl optimize.s see1.s
perl t3.perl see1.s see2.s 1
perl t5.pl see2.s see1.s
perl t3.perl see1.s see2.s 1
perl tj.pl see2.s optimize.s
vmipstool --assemble –o scan.o optimize.s
./vmipstool --link -o netout setup.o arg2.o test.o read.o lib.o scan.o flib.o

```
./vmipstool --make-rom netout hello.rom
./vmips -o noinstdump -o romfile=hello.rom -o outfile=resultprintnet2.txt
```

In case you have relocations truncated errors, use –G 0 during compilation and assembling (Section 3.4).

A3: Running Scripts for EEMBC with profiler, Virtual OS, and Optimizer.

The script file for EEMBC is given below. Except where mentioned to change, do not change anything, and change to the corresponding directory for other four benchmarks. The script mentioned is for Office Benchmark of EEMBC.

Main Script: scriptofficeoptimization.scr

```
#! /bin/tcsh

#do not forget to do what follows :
#cp /home/aggarwal/TOOLS/mipstoolchain/mipsel-elf/lib/*.a
/home/aggarwal/TOOLS/vmips-1.1.3/

rm -rf OFFICEOPT/* #Change for various benchmarks
cd OFFICEOPT #Change for various benchmarks
cp ../eembc-1.1.zip .
cp ../office-1.1.zip . #Change for various benchmarks
unzip -a eembc-1.1.zip
unzip -a office-1.1.zip #Change for various benchmarks
sed -e 's/8_16-bit automotive consumer networking office telecom/office/' < forall >
tmp.txt #Change for various benchmarks
mv -f tmp.txt forall
gmake distclean
foreach d ( `find . -type f` )
sed -e 's/\bmain\b/main1/g' <$d> t1.txt
mv t1.txt $d
end

cd office #Change for various benchmarks
sed -e 's/RUN\_FLAGS/RUN\_FLAGS\_ROM/' < runx86.mak > tmp.txt
mv -f tmp.txt runx86.mak
sed -e 's/\-c//' < makefile > tmp.txt
mv -f tmp.txt makefile

cd ../util/make
sed -e 's/\-c//' < makefile > tmp.txt
mv -f tmp.txt makefile
cd -

cp ../../t1.pl .
cp ../../com.pl .
cp /home/aggarwal/TOOLS/perl/*.pl . #Optimizer files
cp /home/aggarwal/TOOLS/perl/*.perl . #Optimizer files
perl t1.pl runx86.mak tmp.txt
mv -f tmp.txt runx86.mak
ln -s /home/aggarwal/TOOLS/vmips-1.1.3/* .
```

```
cd ../th/x86/al
sed -e 's/define FLOAT\_SUPPORT (TRUE)/define FLOAT\_SUPPORT (FALSE)/'
< thcfg.h > tmp.txt #To remove floating point support
mv -f tmp.txt thcfg.h
cd ../../../th_lite/x86/al
sed -e 's/define FLOAT\_SUPPORT (TRUE)/define FLOAT\_SUPPORT (FALSE)/'
< thcfg.h > tmp.txt #To remove floating point support
mv -f tmp.txt thcfg.h
cd ../../../
cd util/make/
mv -f vc.mak vc.mak.ori
cp gcc.mak mips.mak
mv -f gcc.mak gcc.mak.ori
sed -e 's/usr/home\/aggarwal\/TOOLS\/mipstoolchain\/mipsel-elf/' <
mips.mak > tmp.txt
mv -f tmp.txt mips.mak
sed -e 's/O2/O3/' < mips.mak > tmp.txt
mv -f tmp.txt mips.mak
sed -e 's/\-Wreturn\-type/ /g' < mips.mak > tmp.txt
mv -f tmp.txt mips.mak
sed -e 's/CC\t\t\= \$(TOOLS)\/bin\/gcc/CC              \= vmipstool
\-\-compile \-S \-G 0/' < mips.mak > tmp.txt
mv -f tmp.txt mips.mak
sed -e 's/AS\t\t\= \$(TOOLS)\/bin\/as/AS              \= vmipstool
\-\-assemble \-G 0/' < mips.mak > tmp.txt
mv -f tmp.txt mips.mak
sed -e 's/LD\t\t\= \$(TOOLS)\/bin\/gcc/LD              \= vmipstool
\-\-link/' < mips.mak > tmp.txt
mv -f tmp.txt mips.mak
sed -e 's/\-lm/setup.o arg2.o read.o errno.o  \-lc \-lidt -lm/' <
mips.mak > tmp.txt
mv -f tmp.txt mips.mak
#sed -e 's/LIBTYPE\t\= \.a/LIBTYPE\t\= \.a \-lc \-lidt -lm/' <
mips.mak > tmp.txt
sed -e 's/LIBTYPE\t\= \.a/LIBTYPE\t\= \.a libc\.a libidt\.a libm\.a /'
< mips.mak > tmp.txt
mv -f tmp.txt mips.mak
sed -e 's/RUN\t\t\t\t\=/RUN\t\t\t\= vmipstool /' < mips.mak > tmp.txt
mv -f tmp.txt mips.mak
sed -e 's/EXE\t\t\= \.exe/EXE\t\t\= \.exe\nROM\t\t\= \.rom/' <
mips.mak > tmp.txt
mv -f tmp.txt mips.mak
sed -e 's/RUN\_FLAGS\t\t\=/RUN\_FLAGS\_ROM\t\t\=
\-\-make\-rom\nSIM\t\t\= vmips\nRUN\_FLAGS\_SIM\t\t\= \-o noinstdump
/' < mips.mak > tmp.txt
mv -f tmp.txt mips.mak
```

```
cd /home/aggarwal/TOOLS/mipstoolchain/mipsel-elf/bin/
ln -s ../../bin/mipsel-elf-size ./size
cd -
cd ../../th
cp -R gcc/ mips
cd mips/
sed -e 's/th\/gcc\/harness/th\/mips\/harness/' < depgen.cml > tmp.txt
mv -f tmp.txt depgen.cml
cd ..
touch harness.h
perl ../util/perl/makerule.pl -cmd mips/depgen.cml
cd ../th_lite/
cp -R gcc/ mips
cd mips/
sed -e 's/th\_lite\/gcc\/harness/th\_lite\/mips\/harness/' <
depgen.cml > tmp.txt
mv -f tmp.txt depgen.cml
cd ..
touch harness.h
perl ../util/perl/makerule.pl -cmd mips/depgen.cml
cd ../util/awk/
cp sizegcc.awk sizemips.awk
cd ../../office #Change for various benchmarks
cp depgen_gcc.cml depgen_mips.cml
sed -e 's/targets\_gcc\.mak/targets\_mips\.mak/' < depgen_mips.cml > tmp.txt
mv -f tmp.txt depgen_mips.cml
./vmipstool --assemble -o setup.o setup1.s
./vmipstool --compile -S -G 0 -O3 read.c
./vmipstool --compile -S -G 0 -O3 arg2.c
./vmipstool --compile -S -G 0 -O3
/home/aggarwal/TOOLS/newlib-1.12.0/newlib/libc/errno/errno.c
perl t3.perl  read.s see1.s 1 #These perl files are explained in Appendix on Complete
VMIPS flow.
perl t5.pl see1.s see2.s
perl t3.perl  see2.s see1.s 1
perl tj.pl see1.s see2.s
vmipstool --assemble -o read.o see2.s
perl t3.perl arg2.s see1.s 1
perl t5.pl see1.s see2.s
perl t3.perl  see2.s see1.s 1
perl tj.pl see1.s see2.s
vmipstool --assemble -o arg2.o see2.s
perl t3.perl  errno.s see1.s 1
perl t5.pl see1.s see2.s
perl t3.perl  see2.s see1.s 1
perl tj.pl see1.s see2.s
```

```
vmipstool --assemble -o errno.o see2.s
gmake harness TOOLCHAIN=mips
perl com.pl targets_mips.mak tmp.txt
mv -f tmp.txt targets_mips.mak
cd ../th/mips/
cp ../../../com.pl .
perl com.pl harness.mak tmp.txt
mv -f tmp.txt harness.mak
cd -
cd ../th_lite/mips/
cp ../../../com.pl .
perl com.pl harness.mak tmp.txt
mv -f tmp.txt harness.mak
cd -
gmake TOOLCHAIN=mips
```

This uses 2 perl files also which are com.pl and t1.pl. These files are:

t1.pl:
This file changes all occurrences of running from the object file to running by making a ROM file and then running ROM file, in other words adapt the EEMBC execution to the VMIPS execution flow.

```
#!/usr/local/bin/perl -w
$oldfile = $ARGV[0];
$newfile = $ARGV[1];
open(OF, $oldfile);
open(NF, ">$newfile");

# read in each line of the file
while ($line = <OF>)
{
    if((($line =~ /RUN\_FLAGS/)))# If line contains RUN_FLAGS
    {
        $p=$line;
        @line=split(/\//,$p);
        @line2=split(/\$/,$line[1]);
        @line3=split(/\>/,$p);

        print NF $line[0];
        print NF "\/";
        print NF $line2[0];
        print NF "\$\(LITE\)\$\(EXE\) \$\(BINBUILD\)\/";
        print NF $line2[0];
        print NF "\$\(LITE\)\$\(ROM\) ";
        print NF "\n\t";
```

```
		print NF "\-\$\(SIM\) \$\(RUN\_FLAGS\_SIM\) \$\(BINBUILD\)\/";
		print NF $line2[0];
		print NF "\$\(LITE\)\$\(ROM\) \-o statistic \-o nobrjuinstrs \-o
nomeminstrs \-o outfile\=\$\(RESULTS\)\/$line2[0]\.txt  \> ";
		print NF $line3[1];
		print NF "\n";

	}
	else
	{
		print NF $line;
	}
}

close(OF);
close(NF);
```

com.pl:

This file changes the direct compilation flow to form an object file to the optimized flow.

```
#!/usr/local/bin/perl -w
$oldfile = $ARGV[0];
$newfile = $ARGV[1];
open(OF, $oldfile);
open(NF, ">$newfile");
# read in each line of the file
while ($line = <OF>)
{
	if((($line =~ /COM/)))
	{
		$p=$line;
		@line=split(/ /,$p);
		$ctrt=0;
		print NF "\t";
		for($ctrt=0;$ctrt<$#line-1;$ctrt++)
		{
			print NF $line[$ctrt];
			print NF " ";
		}
		print NF "-o res.s ";
		print  NF $line[$#line];
		print NF "\n";
		print NF "\tperl t3.perl  res.s see1.s 1\n";
		print NF "\tperl t5.pl see1.s see2.s\n";
		print NF "\tperl t3.perl  see2.s see1.s 1\n";
```

```
            print NF "\tperl tj.pl see1.s see2.s \n";
            print NF "\tvmipstool --assemble –G 0 $line[$#line-1] see2.s\n";
        }
        else
        {
            print NF $line;
        }
}

close(OF);
close(NF);
```

A4: Running Optimizer on the libraries

The optimized flow has also to be used for the libraries. For these, make a new archive, say libcopt.a , and add all the C files from the newlib directory. Launch this code from inside newlib libc directory,say /home/aggarwal/TOOLS/newlib-1.12.0/newlib/libc

```
/home/aggarwal/TOOLS/vmips-1.1.3/vmipstool --compile -S -O3 stdlib/*.c
/home/aggarwal/TOOLS/vmips-1.1.3/vmipstool --compile -S -O3 stdio/*.c
/home/aggarwal/TOOLS/vmips-1.1.3/vmipstool --compile -S -O3 reent/*.c
/home/aggarwal/TOOLS/vmips-1.1.3/vmipstool --compile -S -O3 string/*.c
/home/aggarwal/TOOLS/vmips-1.1.3/vmipstool --compile -S -O3 time/*.c
/home/aggarwal/TOOLS/vmips-1.1.3/vmipstool --compile -S -O3 ctype/*.c
/home/aggarwal/TOOLS/vmips-1.1.3/vmipstool --compile -S -O3 locale/*.c
/home/aggarwal/TOOLS/vmips-1.1.3/vmipstool --compile -S -O3 machine/mips/*.c
cp ../libm/common/*.h .
/home/aggarwal/TOOLS/vmips-1.1.3/vmipstool --compile -S -O3 ../libm/math/*.c -I.
/home/aggarwal/TOOLS/vmips-1.1.3/vmipstool --compile -S -O3 ../libm/common/*.c -I.

cp /home/aggarwal/TOOLS/perl/*.pl .
cp /home/aggarwal/TOOLS/perl/*.perl .
mkdir outs
foreach d(*.s)
perl t2.perl $d see
perl t3.perl  see see1 1
perl t5.pl see1 see2
perl t3.perl  see2 see1 1
perl tj.pl see1 see2
/home/aggarwal/TOOLS/vmips-1.1.3/vmipstool --assemble -o $d".o" see2
mv $d".o" outs
end
rm *.s
rm fdlibm.h
rm *.pl
rm *.perl
cd outs
rm *.a
cp /home/aggarwal/TOOLS/build-newlib/mipsel-elf/newlib/libc/reent/*r.o .
cp /home/aggarwal/TOOLS/build-newlib/mipsel-elf/newlib/libc/stdlib/*r.o .
rm fstatr.s.o openr.s.o sbrkr.s.o timer.s.o getenv_r.s.o putenv_r.s.o setenv_r.s.o unlinkr.s.o
rm closer.s.o linkr.o mblen_r.s.o signalr.s.o execr.s.o lseekr.s.o mbstowcs_r.s.o rand_r.s.o statr.s.o wcstombs_r.s.o
```

```
rm fcntlr.s.o mbtowc_r.s.o readr.s.o strtoll_r.s.o wctomb_r.s.o strtoull_r.s.o writer.s.o
/home/aggarwal/TOOLS/mipstoolchain /bin/mipsel-elf-ar q libcopt.a *.o
cp libcopt.a /home/aggarwal/TOOLS/vmips-1.1.3/
rm *.o
cd –
```

And you will have the library made and you in place of adding libc.a and libm.a, add this library, and if need be, make additions to this library with more files to solve your purpose.

A5: Setup Code.

setup1.s:

```
#include "asm_regnames.h"
#define MEM_BASE              0xa0000000
#define MEM_SIZE              0x100000
#define DATA_START            MEM_BASE + (MEM_SIZE*3/4)
#define INIT_STACK_BASE DATA_START - 4
#define NTLBENTRIES           64
      .data

      j begin
      .text
      .globl __start
      .ent __start
__start:
      j begin
      .end __start
      .org 0x30
      .set noreorder
      j opena
      nop
      j reada
      nop
      j writea
      nop
      nop
      nop
      j closea
      nop
      j inbytea
      nop
      j outbytea
      nop
      .set reorder
      /* Halt on user tlb exceptions. */
      .org 0x100
      break 0x0
      /*Alternate Exception Subroutine*/
      .set noreorder
      mfc0    k0,EPC
      nop
      addiu k0,k0,4
      j       k0
      rfe
```

```
        .set reorder
        /* Halt on exceptions. */
        .org 0x180
        break 0x0
        /*Alternate Exception Subroutine*/
       .set noreorder
        mfc0    k0,EPC
        nop
        j       k0
        rfe
        .set reorder
        .org 0x1b8
        j get_mem_info1
        nop
        addiu $2,$0,9
        syscall
        j       $31
        .org 0x200

#define FUNCTION                entry#You can change the entry keyword from here if
your program uses it
        .globl begin
        .ent begin
begin:
/* Start by clearing everything out. */
        .set noat
        move $1, $0
        .set at
        move $2, $0
        move $3, $0
        move $4, $0
        move $5, $0
        move $6, $0
        move $7, $0
        move $8, $0
        move $9, $0
        move $10, $0
        move $11, $0
        move $12, $0
        move $13, $0
        move $14, $0
        move $15, $0
        move $16, $0
        move $17, $0
        move $18, $0
        move $19, $0
```

```
    move $20, $0
    move $21, $0
    move $22, $0
    move $23, $0
    move $24, $0
    move $25, $0
    move $26, $0
    move $27, $0
    move $28, $0
    move $29, $0
    move $30, $0
    mtc0 zero, $4
    mtc0 zero, $8
    mtc0 zero, $14

    /* Clear out the TLB. */
    li t2, NTLBENTRIES      /* t2 = TLB entry number */
    li t3, 0x00000fc0       /* t3 = (VPN 0x0, ASID 0x3f) */
1:
    addiu t2, t2, -1        /* Decrement TLB entry number */
    sll t1, t2, 8           /* Shift entry number into Index field position */
    mtc0 t1, $0             /* set Index */
    mtc0 zero, $2           /* clear EntryLo */
    mtc0 t3, $10            /* set EntryHi */
    tlbwi                   /* write TLB[Index] with (EntryHi, EntryLo) */
    bnez t2, 1b             /* Go back if we're not done yet. */
    nop
    mtc0 zero, $10          /* clear EntryHi (sets effective ASID=0x0) */

    /* Set up the stack and globals pointer. */
    li sp, INIT_STACK_BASE
    la gp, _gp

    /* Copy writeable data to writeable RAM. */
    la t1, _copystart               /* t1 = beginning source address for copy */
    la t2, _copyend
    addiu t2, t2, 4         /* t2 = one word past ending source address */
    move t3, gp                     /* t3 = beginning dest address */
1:
    lw t4, 0(t1)                    /* load t4 from ROM */
    sw t4, 0(t3)                    /* store it in RAM */
    addiu t1, t1, 4         /* increment both pointers */
    addiu t3, t3, 4
    bne t1, t2, 1b          /* if we're not finished, loop. */
    nop
    /* Call the function. */
```

```
        jal FUNCTION

        /* Wait a minute, wait a minute, stop the execution! */
        break 0x0
        .end begin

      .globl writea
        .ent    writea
writea:
        addiu $2,$0,7
        syscall
        j       $31
        .end writea

        .globl opena
        .ent    opena
opena:
        addiu $2,$0,5
        syscall
        j       $31
        .end opena

        .globl closea
        .ent    closea
closea:
        addiu $2,$0,8
        syscall
        j       $31
        .end closea

        .globl reada
        .ent    reada
reada:
        beq $4,0x0,read1
        addiu $2,$0,6
        syscall
        j       $31
        .end reada

        .globl _exit
        .ent    _exit
_exit:
        break 0x0
        .end _exit

/* The following routine is required by the "sbrk()" function: */
```

```
        .globl  get_mem_info1
        .ent    get_mem_info1
        .set    noreorder
get_mem_info1:
        /*# in:  a0 = pointer to 3 word structure
        # out: void*/
        subu    sp,sp,0x18      /* create some stack space */
        sd      ra,0x00(sp)     /* stack return address */
        sd      $30,0x08(sp)     /* stack frame-pointer */
       /* sd      a0,0x10(sp)     /* stack structure pointer */
        move    $30,sp          /* take a copy of the stack pointer */
        li      v0,0x10000000
       nop
        /*ld      a0,0x10(sp)   /*  # recover structure pointer*/
        sw      v0,0(a0)
        li a2,0x4000
        sw a2,4(a0)
        sw a2,8(a0)
        move    sp,$30          /* recover stack pointer */
        ld      ra,0x00(sp)     /* recover return address */
        ld      $30,0x08(sp)     /* recover frame-pointer */
        j       ra              /* return to the caller */
        addu    sp,sp,0x18      /* restore stack pointer {DELAY SLOT} */
        .set    reorder
        .end    get_mem_info1

        .globl lseek
        .ent lseek
lseek:
        addiu v0,$0,26
        syscall
        j $31
        .end lseek

        .globl fstat
        .ent fstat
fstat:
        addiu v0,$0,27
        syscall
        j $31
        .end fstat

        .globl isatty
        .ent isatty
isatty:
        addiu v0,$0,28
```

```
        syscall
        j $31
        .end isatty

        .globl gettimeofday
        .ent gettimeofday
gettimeofday:
        addiu v0,$0,34
        syscall
        j $31
        .end gettimeofday

        .globl times
        .ent times
times:
        addiu v0,$0,35
        syscall
        j $31
       .end times
```

This code uses a few functions provided in some other C files which are in read.c.

Spimconsreg.h:

```
#ifndef _SPIMCONSREG_H_
#define _SPIMCONSREG_H_

#include "devreg.h"

/* default physical spim base address */
#define SPIM_BASE          0x02000000

/* default virtual (KSEG0) spim address */
#define SPIM_ADDR          0xa2000000

/* register offsets */
#define KEYBOARD_1_CONTROL  0x00
#define KEYBOARD_1_DATA     0x04
#define DISPLAY_1_CONTROL   0x08
#define DISPLAY_1_DATA      0x0C
#define KEYBOARD_2_CONTROL  0x10
#define KEYBOARD_2_DATA     0x14
#define DISPLAY_2_CONTROL   0x18
#define DISPLAY_2_DATA      0x1C
#define CLOCK_CONTROL       0x20

#endif /* _SPIMCONSREG_H_ */
```

read.c :

```
#include "spimconsreg.h"

#define IOBASE 0xa2000000
#define IS_READY(ctrl) (((*(ctrl)) & CTL_RDY) != 0)

volatile long *display_data_reg = (long *)(IOBASE+DISPLAY_1_DATA);
volatile long *display_control_reg = (long *)(IOBASE+DISPLAY_1_CONTROL);
volatile long *keyboard_data_reg = (long *)(IOBASE+KEYBOARD_1_DATA);
volatile long *keyboard_control_reg = (long *)(IOBASE+KEYBOARD_1_CONTROL);

static unsigned char
internal_read_byte (void)//Input byte from screen
{
     char c;
  do
    {
      0;
    }
  while (!IS_READY (keyboard_control_reg));
  c= (char) *keyboard_data_reg;
  write(1,&c,1);
  return c;
}


int
read1 (int fd, unsigned char *buf, int count)//For input from screen
{
  int i;
  char *b = buf;

  if(count>1023)count=count/1024;//This is a hack for scanf as scanf uses this argument with 1024 for one character input, and I assume that if you want a data from user to type, you won't normally need more than 1023 characters.

  if (fd != 0)
    return 0;
  for (i = 0; i < count; i++)
    {
      *b++ = internal_read_byte ();
    }
  return count;
```

```
}
int
outbytea (unsigned char c)//Display a byte
{
  return write(1, (unsigned long)&c, 1);
}

unsigned char inbytea (void)//Get a byte from screen
{
  char c;
  read1( 0, &c, 1);
  return c;
}
```

A6: VMIPS Installation Flow.

1. Extract VMIPS tar file to the desired location.

2. Check that make is in your path, else add it.

3. cd vmips-1.1.3

4. ./configure --with-mips=/home/aggarwal/TOOLS/mipstoolchain --with-mips-endianness=little --with-mips-bfdtarget=elf32-littlemips
   Here, we fit the endianness of emulated processor (elf32-littlemips) with an explicit endianness information (--with-mips-endianness=little). The cross compiler must compile C programs in the same way, i.e. little endian in this case.

5. gmake all-am

6. If the previous step did not give any error, do not continue.
   c++ -DSYSCONFDIR=\"/usr/local/etc\" -I/home/aggarwal/TOOLS/mipstoolchain /include -I./include -g -Wall -fno-strict-aliasing -o vmips cpu.o cpzero.o devicemap.o mapper.o memorymodule.o options.o range.o intctrl.o spimconsole.o stub-dis.o tlbentry.o vmips.o deviceint.o debug.o remotegdb.o clockdev.o error.o clock.o task.o terminalcontroller.o haltdev.o -L/home/aggarwal/TOOLS/mipstoolchain /lib /usr/lib/libopcodes.a ../build-binutils/opcodes/libopcodes.a /usr/lib/libbfd.a -L/usr/src/build/227532-i386/BUILD/binutils-2.13.90.0.18/build-i386-redhat-linux/libiberty/pic -liberty ../build-binutils/bfd/libbfd.a

7. gmake all-am

A7: VMIPS Installation Flow used when you copy VMIPS in other directory, or change Cross Compiler Toolchain, etc.

First method is to reinstall after removing object files, 2nd way is:

1. Check that make is in your path, else add it.

2. cd <the VMIPS Directory>

3. ./configure --with-mips=/home/aggarwal/TOOLS/mipstoolchain --with-mips-endianness=little --with-mips-bfdtarget=elf32-littlemips
   Here, we fit the endianness of emulated processor (elf32-littlemips) with an explicit endianness information (--with-mips-endianness=little). The cross compiler must compile C programs in the same way, i.e. little endian in this case.

4. gmake all-am

5. rm *.o

6. gmake clean-compile

7. gmake vmips

8. gmake all-am

If you are only changing few files in the directory, you need to do only from step 5.

A8: Some function implementations

These implementations are used in the second implementation of the Virtual Operating System (Section 4.4). But if you the first approach also, you can use these functions if you want to reduce the instructions that will be executed on the processor. So, for real applications, in place of printf, what you can do is use printf1 in the code, and put the printf function implementation inside the user code in function name printf1.

```
#include <stdarg.h>
static int
fprint_int (FILE *file,int i, unsigned int radix, unsigned int use_uppercase)
{
  char digit = digit_to_char (i % radix, use_uppercase);
  int remains = (i / radix);

  if (i < 0)
    {
      fputc ('-',file);
      return 1 + fprint_int (file,-i, radix, use_uppercase);
    }
  else if (remains > 0)
    {
      int nprinted = fprint_int (file,remains, radix, use_uppercase);
      fputc (digit,file);
      return 1 + nprinted;
    }
  else
    {
      fputc (digit,file);
      return 1;
    }
}

static int
fprint_unsigned_int (FILE *file,unsigned int i, unsigned int radix,
                     unsigned int use_uppercase)
{
  char digit = digit_to_char (i % radix, use_uppercase);
  int remains = (i / radix);

  if (remains > 0)
    {
      int nprinted = fprint_unsigned_int (file,remains, radix, use_uppercase);
      fputc (digit,file);
      return 1 + nprinted;
```

```
    }
  else
    {
      fputc (digit,file);
      return 1;
    }
}
int
fprintf (FILE *file,const char *fmt, ...)
{
  const char *f = fmt;
  va_list ap;
  int count = 0;
  int nextarg_is_long = 0;
  int i;
  unsigned int u;
  char c;
  char *s;

  va_start (ap, fmt);
  while (f[0])
    {
      if (f[0] != '%')
        {
          fputc (*f,file);
          count += 1;
          f += 1;
        }
      else
        {
          if (f[1] == 'l') { nextarg_is_long++; f++; }
          switch (f[1])
            {
            case 'c':
              c = (char) va_arg (ap, int);
              fputc (c,file);
              count += 1;
              break;
            case 'd':
            case 'i':
              i = va_arg (ap, int);
              count += fprint_int (file,i, 10, 0);
              break;
            case 'o':
              u = (unsigned int) va_arg (ap, int);
              count += fprint_unsigned_int (file,u, 8, 0);
```

```
            break;
          case 'u':
            u = (unsigned int) va_arg (ap, int);
            count += fprint_unsigned_int (file,u, 10, 0);
            break;
          case 'x':
            u = (unsigned int) va_arg (ap, int);
            count += fprint_unsigned_int (file,u, 16, 0);
            break;
          case 'X':
            u = (unsigned int) va_arg (ap, int);
            count += fprint_unsigned_int (file,u, 16, 1);
            break;
          case 's':
            s = (char *) va_arg (ap, char *);
            while (*s)
              {
                fputc (*s++,file);
                count++;
              }
            break;
          default:
            fputc (f[1],file);
            count += 1;
            break;
          }
        f += 2;
       }
    }
  return count;
}


/* fgets:  get at most n chars from iop */
   char *fgets(char *s, int n, FILE *iop)
   {
       register int c;
       register char *cs;

       cs = s;
       while (--n > 0 && (c = fgetc(iop)) != -1)
           if ((*cs++ = c) == '\n')
               break;
       *cs = '\0';
       return ((c == -1) && (cs == s)) ? "" : s;
   }
```

```
  /* fputs:  put string s on file iop */
  int fputs(const char *s, FILE *iop)
  {
      int c;

      while (c = *s++)
          fputc(c, iop);
      return ferror(iop) ? -1 : 0;
  }
char *
 strcat(char *  s, const char *  append)
 {
        char *save = s;

        for (; *s; ++s);
        while ((*s++ = *append++) != 0);
        return(save);
 }
```

On the same lines, I have implemented some other functions like scanf, etc. Some functions like printf, etc. are available inside the VMIPS, which also can be used. You can use these if you need some functionality and that functionality is fully provided by the implementations. These result in shorter number of executed instructions.

A9: To integrate new instructions (like Floating-Point instructions)

First of all for this purpose, you need to adjust VMIPS Simulator so that the respective opcodes are handled well. For this, in the cpu.cc file, there is a JumpTable where the program goes to depending on the opcode. So, according to your opcode, you have to make some new function name and add its definition in cpu.h and implement in cpu.cc. If the opcode is 0 or 1, we enter funct_emulate or regimm_emulate respectively, and these further go to some function after checking the last six digits or the rt respectively. So, that is the place where you have to call your function in that case.

You need to update some quantities in the profiler. So, in the function you have made, add the things like shown below after adjusting them a bit

```
if((prevprev=="L")&&(opt_statistic))
{
     if((prevloadedregister==rs(instr))||(prevloadedregister==rt(instr)))
          //Assuming that rs and rt are used in this instruction, else change it
          //correspondingly
                    numreadlr++;
}
 if(canbeswapped&&(pr=="B")&&(opt_statistic))
   if((rd(instr)==branchrs)||(rd(instr)==branchrt))
          //Assuming that rd  is written in this instruction, else change it
          //correspondingly
          canbeswapped=false
if(opt_statistic)
{
          prevprev=pr;
          prevloadedregister=loadedregister;
          pr="NewInstr";
          pr1=" NewInstr ";
}
```

The operating system will work without any change.

For the optimizer, you need to edit the files t5.pl and tj.pl where I set the values of the arrays of registers used and written, you need to adjust that for the new instructions and the optimizer will work for all the new instructions.

```
for($i=0;$i<$index;$i++)
{
          #print "i\n";
          $p=$filecontent[$i];
          #print $p;
          if($p=~/^(\s)*nop/)                    {
                    $operand[$i]="N";
                    $instruction[$i]="Y";
```

```
		$written[$i]=0;
		$used1[$i]=0;
		$used2[$i]=0;
	}
	……..
	elsif($p=~/^(\s)*newinstr/)
	{
		$operand[$i]="NEW";
		$instruction[$i]="Y";
		$written[$i]=?? //Update acc. instruction
		$used1[$i]=?? //Update acc. instruction
		$used2[$i]=?? //Update acc. instruction
	}
	……..
	else
	{
		$operand[$i]="MACRO";
		@dollararray=DA($i);
		$instruction[$i]="Y";
		$written[$i]=$dollararray[0];
		$used1[$i]=0;
		$used2[$i]=0;
		if($#dollararray>0)
		{
			$used1[$i]=$dollararray[1];
			$used2[$i]=$dollararray[1];
		}
		if($#dollararray>1)
		{
			$used2[$i]=$dollararray[2];
		}
	}
}
```

Thus, all the tools will be adjusted and everything will work in same way for a different architecture. The same approach can be used for any architecture derived from MIPS R3000.

Bibliography

1. **Kane, G., and Heinrich, J.** *MIPS RISC Architecture*. Prentice Hall, 1992.

2. **Sweetman, D.** *See MIPS Run.* Academic Press, 2002.

3. **Farquhar, E., and Bunce, P.** *The MIPS Programmer's Handbook.* San Francisco: Morgan Kauffman Publishers, 1994.

4. **Hennessy, J., and Patterson, D.** *Computer Architecture: A Quantitative Approach*. San Francisco: Morgan Kauffman Publishers, 1996.

5. **Tannenbaum, A.** *Modern Operating Systems*, 2nd edition. Prentice Hall, 2001.

6. **Kernighan, B., and Richie, D.** *The C Programming Language*, 2nd edition. Englewood Cliffs, NJ: Prentice Hall, 1988.

7. Lectures of Operating Systems
   http://cgi.cse.unsw.edu.au/~cs3231/lectures.php

8. SPIM Simulator
   http://www.cs.wisc.edu/~larus/spim.html

9. VMIPS Simulator
   www.dgate.org/vmips

10. SPEC 2000 Benchmark
    http://www.specbench.org/

11. EEMBC benchmark
    www.eembc.org

12. CommBench benchmark.
    http://ccrc.wustl.edu/~wolf/cb/

13. *Full Cross Compiler Installation Flow* by **Sylvain Aguirre,** April, 2003.