Cycle Level SPIM

Anne Rogers
Scott Rosenberg
Department of Computer Science
Princeton University
Princeton, New Jersey 08544

July 2, 1993

1 Introduction

The original SPIM was written by Jim Larus at the University of Wisconsin at Madison. It's purpose is to simulate the function of the MIPS R2000. Cycle Level SPIM (clspim) is an outgrowth of Larus's original code, built to simulate the pipeline architecture of the MIPS machine. Under its new cycle level mode, SPIM simulates both the control and floating point processor pipelines. The control processor comes complete with simulation of the R2000's on chip TLB and multiply/divide units. In addition, SPIM's exception handling has been expanded to work with the new pipeline simulation and to work in conjuction with its new signal handling capabilities. System calls, processed by the R2000 as traps or exceptions, have also been updated. Finally, the new SPIM contains two caches for instruction and data references. This note outlines the effects of all these changes in SPIM and the new commands that have been implemented in spim, clspim, and xlspim, the Xwindows version. Cycle level mode in SPIM can be turned on at runtime with the `-cycle` flag or after startup with the set cycle command in clspim or the mode button in xlspim.

2 The Pipelines

Both the control and floating point processors's pipelines are processed in reverse order during each cycle. This, combined with bypassing, is done so that values being derived from the execution, memory, and write back stages are available to other instructions in the pipeline during the same cycle.

The following is an outline of what takes place at each of the five stages of the control processor pipeline:

- **WB** - Write Back

  pertinent registers are written by the instruction unless it had caused an exception at some earlier stage in the pipeline

- **MEM** - Memory

  values are fetched from memory; if the TLB is on and the page is not in memory, an exception is flagged - the data will not be loaded or stored until SPIM has a chance to bring in the page and restart the pipeline; if the data cache is on and the address is in one of the cache's lines, no stall occurs; otherwise, a stall may occur as the memory bus fetches or stores the information; results are sent along the memory bypass

- **EX** - Execution

  all calculations are processed here; results are sent along the execution bypass

- **ID** - Instruction Decode

  instructions are interpreted to find their opcode and register references; multiply and divide instructions are passed to the MDU; branch instructions set nPC; instructions referencing registers that are going to be set by the MDU or floating point unit stall; if the instruction in this stage is a branch, SPIM marks the incoming instruction (in IF) as being in a branch delay slot

- **IF** - Instruction Fetch

  the instruction located at the program counter (PC) is fetched; if the TLB is on and this instruction is not on a page in memory, an exception occurs; otherwise, if the instruction cache is on and the PC is in one of the cache's lines, the instruction is brought in; when the line is not in the cache, a stall occurs; PC is set to nPC

The floating point unit is dependent upon the control processor because all instructions must first be fetched by the control processor. It is only at the decode stage that the control processor (also called coprocessor 0) recognizes that an instruction should be passed on to the floating point unit (coprocessor 1) or any other external processor.

The following is a summary of the floating point pipeline:

- **FWB** - Floating Point Write Back

  up to two instructions may be pending in this stage; values are will be written to the floating point unit's own register set

- **FEX3** - Floating Point Execution (Stage 3)

  up to four instructions may be pending in this stage; an instruction may be caught in this phase for numerous cycles

- **FEX2, FEX1** - Floating Point Execution (Stages 2 and 1)

  the beginning stages of floating point execution; instructions have just come off of the control processor

The status of the pipelines and bypass values can be viewed while running clspim in `-noquiet` mode or when stepping through simulated code using the `step` command. The `info pipeline` command will also print this information. When running xlspim, the pipeline can be viewed by clicking on the `pipeline` button. It is important to note that when viewing the pipelines, you are witnessing the state of the machine as it is **about** to execute. By stepping or continuing from your current point, you are telling SPIM to execute whatever is currently in the pipeline.

3 Exceptions

The MIPS R2000 specifies twelve types of internal exceptions. SPIM may pick up on any one of these exceptions at any point during the pipeline. Once an instruction has caused an exception, the pipeline after it is killed, that is, no more instructions will be fetched until the exception is handled. The actual processing of an exception occurs after the bad instruction has reached the write back stage. Before handling the exception, SPIM sets four bits in the Cause register corresponding to the exception's code and sets the EPC (Execution Program Counter) register to the address of the bad instruction or to the address of the instruction preceding if it was in a branch delay slot. SPIM then takes appropriate action to remedy the exception, be it the execution of a system call, paging of memory, stopping for a breakpoint, or posting of a signal. In the case of a properly executed system call, SPIM, as with UNIX, calculates a startup address so that the system call instruction is not executed again. A TLB or breakpoint exception, on the other hand, will start the program up at the EPC. Exceptions such as bus errors, illegal instructions, bad system calls, overflows, and unimplemented CPU's, can be translated into signals, and are thus, posted on the simulated program's pending signal list.

External exceptions, including those that might occur in the MIPS's floating point unit, are not currently simulated in SPIM.

To see how many exceptions and of what type have been handled, one can type `info syscalls` in clspim or pull down the `info` button to select `syscalls` in xlspim.

4 System Calls

System calls in the new cycle level SPIM are processed as exceptions or traps, as the MIPS R2000 instruction set dictates. When a system call instruction enters the pipeline, it is noted by SPIM as an exception. It will be handled, as with all other exceptions, when it reaches the write back stage of the pipeline. The system call, whose value should already have been put in register \$v0, can be one of three types: 1.) unimplemented; 2.) special; or, 3.) relayable to UNIX. (See the file cl-mips-syscall.c for the table which tells each system call's type). Some system calls like fork could not be implemented in SPIM. Other calls, like brk, sbrk, open, close, exit, and several others, are handled by SPIM because they deal with simulated data space or file descriptors. Some calls, however, (and this depends on the architecture where SPIM is running) can be sent on to UNIX. Currently, only a SPIM that was compiled on a MIPS machine is capable of sending system calls on to UNIX. When compiled on a non-MIPS machine, SPIM's old set of pseudo-system calls applies.

The clspim command `info syscall` will tell the frequency of each of the system calls in a program. The xlspim `info` button's `syscalls` option does the same.

5 Signals

Signals can reach a simulated program in one of two ways. The first, as alluded to earlier, is by way of an exception. Bus errors, bad system calls, illegal instructions, and CPU errors all get translated into signals and may be posted into the simulated program's pending signal list. The second means by which a signal can reach a program is through SPIM's own signal catching. If a signal is caught off of UNIX by SPIM and deemed pertinent to the simulated program, it can be passed on.

Whether a signal is actually passed on to a program is determined by a table in cl-except.c. In clspim, one can alter this table using the handle command. As in gdb, the command line

```
handle <signal name or number> <flag>
```

tells SPIM to apply the flag to the given signal. Valid flags are: pass, nopass, print, noprint, stop, and nostop. The print commands tell SPIM to print a message when it sees the signal. The stop commands tell SPIM to halt execution if the signal occurs. Currently, no parallel exists in xlspim. To change SPIM's default method of signal handling (which can be viewed at any time with clspim's `info signals` command or xlspim's `info` button's `signals` option), one would have to change the table in cl-except.c and recompile.

In line with its new signal handling, several system calls previously meaningless can now be used by the simulated program. These are: sigvec, sigreturn, sigsetmask, and sigblock. Library routines like `sigvec()` and `signal()` make use of these system calls. (The system call sigreturn is usually used by a segment of trampoline code that is included automatically with these library routines. It should therefore probably not be used explicitly. The creators of 4.3BSD UNIX apparently meant it this way – no library routine exists for sigreturn.) When one of these system calls is made, SPIM internally takes note of the new handler or new mask being implemented. When and if that signal occurs, SPIM posts a signal into the program's pending signal list. Each time the simulated program has an exception to be processed, that list is checked. If a signal is pending, SPIM checks to see if the signal should be passed to the program, then if it is being masked, and then if a handler has been specified. If a handler exists, SPIM saves the state of the machine on the stack and jumps to a segment of code (actually in the sigvec library routine) that serves as a trampoline to the specified handler. The call eventually comes back from the handler, a sigreturn occurs, and regular execution resumes. When no handler has been specified and a signal is passed to the program, SPIM's default action is to halt execution.

Signal handling in SPIM was designed to function similarly to 4.3BSD UNIX as described in "The Design and Implementation of the 4.3BSD UNIX Operating System", by Leffler, McKusick, Karels, and Quarterman.

6 Virtual Memory - SPIM's new TLB

When the Translation Lookaside Buffer is on, SPIM executes a remapping of virtual address space (as the simulated program sees it) to physical paged address space. SPIM's TLB contains 64 registers and the simulated hardware to map the upper twenty bits of a virtual address to a physical page. Each page is therefore, by default, four kilobytes (12 bits). When a memory access refers to an address that is not mapped to one of the 64 pages in the TLB, an exception occurs. Upon processing that exception, SPIM replaces one of the TLB's registers with the new page number and resumes execution.

The TLB can be turned on in clspim with the `-tlb` flag at runtime or by toggling it on with the set tlb command. In xlspim, the TLB can be turned on at runtime with the `-tlb` flag or by pulling down the mode button after startup.

The TLB defaults to on in cycle level mode for SPIM unless the `-notlb` flag is given at runtime or the tlb has been explicitly toggled off. When the TLB is OFF, SPIM acts as if all pages are accessible and will not flag any TLB exceptions.

7 Instruction and Data Caches

Cycle level SPIM comes with two identical direct mapped caches, with 512 lines at 32 bytes per line. When an instruction fetch is made and the instruction cache is on, SPIM checks to see if the current address matches the tag in the cache line to which it maps. If the line is valid, no stall occurs. Otherwise, the instruction must be fetched from memory and a stall occurs while the memory bus works to load the instruction. The data cache functions similarly to the instruction cache with the difference lying in the fact that it has the added concern of stores. The data cache is write through, no allocate, which means that a store writes memory whether the address it is referencing is in the cache or not. The write buffer is six requests deep. The read buffer is one request deep and always takes priority unless the bus is busy or the load address conflicts with an address in the write buffer.

Caching can be turned on or off at runtime for either clspim or xlspim using the `-icache`, `-noicache`, `-dcache`, and `-nodcache` flags. In interactive clspim, the `set icache` and `set dcache` commands toggle the caches on and off. In xlspim, the `mode` button's `icache` and `dcache` options toggle the caches. Also, in xlspim the caches's tags and valid bits can be viewed using the new `cache` button.

Two important points to note: First, to say a cache is OFF in SPIM is equivalent to saying that a memory reference will not stall. It does not mean that all references will directly access the bus. Second, when the TLB is on, tags written to the caches represent the upper bits of a physical address. When the TLB is off, the tags are indicative of a virtual address.