Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

The target platform for the Decaf and Espresso compilers is a simulator, called SPIM, that has been developed by Jim Larus from the University of Wisconsin-Madison. The remainder of this handout describes the SPIM interface. Handout 18 describes the machine being simulated and its assembly instructions. Most of the following text comes directly from the original document written by Jim Larus, which is in the directory /mit/6.035/doc/spim-6.2.[1]

1 SPIM

SPIM S20 is a simulator that runs programs for the MIPS R2000/R3000 RISC computers.[2] SPIM can read and immediately execute files containing assembly language or MIPS executable files. SPIM is a self-contained system for running these programs and contains a debugger and interface to a few operating system services.

The architecture of the MIPS computers is simple and regular, which makes it easy to learn and understand. The processor contains 32 general-purpose registers and a well-designed instruction set that make it a propitious target for generating code in a compiler.

2 SPIM Interface

SPIM provides both a simple terminal and an X-window interface. Both provide equivalent functionality, but the X interface is generally easier to use and more informative. This document describes all the features supported by both interfaces; most of them are not useful for 6.035, but are included for completeness.

`spim`, the terminal version, and `xspim`, the X version, have the following command-line options:

`-bare`
: Simulate a bare MIPS machine without pseudoinstructions or the additional addressing modes provided by the assembler. Implies `-quiet`.

`-asm`
: Simulate the virtual MIPS machine provided by the assembler. This is the default.

[1] "SPIM S20: A MIPS R2000 Simulator", by James R. Larus, larus@cs.wisc.edu, Computer Sciences Department, University of Wisconsin–Madison, 1210 West Dayton Street, Madison, WI 53706, USA, 608-262-9519. Copyright 1990, 1991 by James R. Larus. This document may be copied without royalties, so long as this copyright notice remains on it.

[2] For a description of the real machines, see Gerry Kane and Joe Heinrich, *MIPS RISC Architecture,* Prentice Hall, 1992.

`-notrap`
Do not load the standard trap handler. This trap handler has two functions that must be assumed by the user's program. First, it handles traps. When a trap occurs, SPIM jumps to location 0x80000080, which should contain code to service the exception. Second, this file contains startup code that invokes the routine `main`. Without the trap handler, execution begins at the instruction labeled `__start`.

`-trap`
Load the standard trap handler. This is the default.

`-trap_file`
Specifies a different file to load as the trap handler.

`-noquiet`
Print a message when an exception occurs. This is the default.

`-quiet`
Do not print a message at an exception.

`-nomapped_io`
Disable the memory-mapped IO facility (see R2000 handout).

`-mapped_io`
Enable the memory-mapped IO facility (see R2000 handout). Programs that use SPIM syscalls (see R2000 handout) to read from the terminal should not also use memory-mapped IO.

`-file`
Load and execute the assembly code in the file.

`-execute`
Load and execute the code in the MIPS executable file *a.out*.

`-s seg size` Sets the initial size of memory segment *seg* to be *size* bytes. The memory segments are named: `text`, `data`, `stack`, `ktext`, and `kdata`. For example, the pair of arguments `-sdata 2000000` starts the user data segment at 2,000,000 bytes.

`-lseg size` Sets the limit on how large a memory segment *seg* can grow to be *size* bytes. The memory segments that can grow are: `data`, `stack`, and `kdata`.

`-l library`
Read in the specified shared library file. Used for external callouts, see section 5.

`-extern`
Turn off internal SPIM memory remapping, allowing use of external memory references. Used for external callouts, see section 5.

3 Terminal Interface

The terminal interface (`spim`) provides the following commands:

`exit`
Exit the simulator.

`read "file"`
Read *file* of assembly language commands into SPIM's memory. If the file has already been read into SPIM, the system should be cleared (see `reinitialize`, below) or global symbols will be multiply defined.

`load "file"`
Synonym for `read`.

`execute "a.out"`
Read the MIPS executable file *a.out* into SPIM's memory.

`run <addr>`
Start running a program. If the optional address *addr* is provided, the program starts at that address. Otherwise, the program starts at the global symbol `__start`, which is defined by the default trap handler to call the routine at the global symbol `main` with the usual MIPS calling convention.

`step <N>`
Step the program for N (default: 1) instructions. Print instructions as they execute.

`continue`
Continue program execution without stepping.

`print $N`
Print register N.

`print $fN`
Print floating point register N.

`print addr`
Print the contents of memory at address *addr*.

`print_sym`
Print the contents of the symbol table, i.e., the addresses of the global (but not local) symbols.

`reinitialize`
Clear the memory and registers.

`breakpoint addr`
Set a breakpoint at address *addr*. *addr* can be either a memory address or symbolic label.

`delete addr`
Delete all breakpoints at address *addr*.

`list`
List all breakpoints.

`.`
Rest of line is an assembly instruction that is stored in memory.

`<nl>`
: A newline reexecutes previous command.

`?`
: Print a help message.

Most commands can be abbreviated to their unique prefix e.g., `ex`, `re`, `l`, `ru`, `s`, `p`. More dangerous commands, such as `reinitialize`, require a longer prefix.

4 X-Window Interface

The X version of SPIM, `xspim`, looks different, but should operate in the same manner as `spim`. The X window has five panes (see Figure 1). The top pane displays the contents of the registers. It is continually updated, except while a program is running.

The next pane contains the buttons that control the simulator:

quit
Exit from the simulator.

load
Read a source or executable file into memory.

run
Start the program running.

step
Single-step through a program.

clear
Reinitialize registers or memory.

set value
Set the value in a register or memory location.

print
Print the value in a register or memory location.

breakpoint
Set or delete a breakpoint or list all breakpoints.

help
Print a help message.

terminal
Raise or hide the console window.

mode
Set SPIM operating modes.

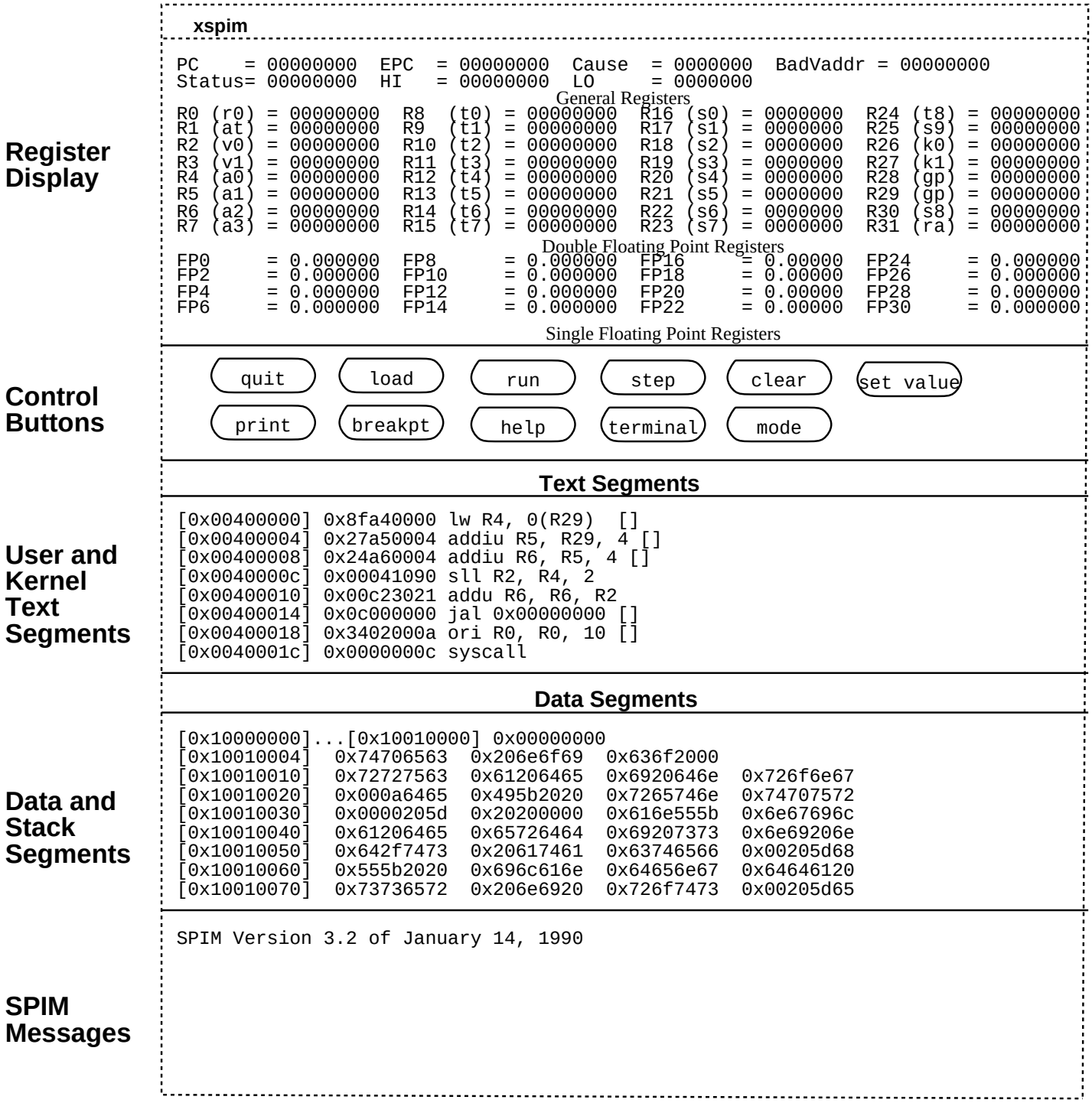

Figure 1: X-window interface to SPIM.

The next two panes display the memory contents. The top one shows instructions from the user and kernel text segments.[3] The first few instructions in the text segment are startup code (`__start`) that loads `argc` and `argv` into registers and invokes the `main` routine.

The lower of these two panes displays the data and stack segments. Both panes are updated as a program executes.

The bottom pane is used to display messages from the simulator. It does not display output from an executing program. When a program reads or writes, its IO appears in a separate window, called the Console, which pops up when needed.

5 Callouts and Shared Libraries

SPIM supports callouts by loading in external libraries at runtime. The `-l` *libname* command line argument to SPIM specifies a library that contains external functions. SPIM loads in the specified libraries, and then searches them for functions used in callout instructions. SPIM automatically loads `libc.so`, the standard C library, and has the compatibility library builtin. `libc` contains many commonly-used functions such as `printf` (for I/O) and `malloc` (for memory allocation). You may also create your own libraries and load them into SPIM. Creating shared libraries is very operating-system dependent however, and isn't required for the course. We'll provide some tips on doing that in a future handout.

Due to the way SPIM simulates memory accesses, you can run SPIM in one of two modes. In the default mode, if your assembly program tries to access memory that it has not allocated, SPIM will raise an exception, which typically reports the error condition to the console. In this mode, SPIM allocates approximately 1 MB of memory at startup for the program. If your program allocates more than this, SPIM will continue to work, but the `callout` instruction will fail and likely cause a coredump. Additionally, in standard mode SPIM does not support calling functions that return pointers to dynamically allocated memory.[4] `malloc()` is the most obvious example of this, but SPIM contains an internal version of `malloc()` that avoids this problem. An example of a function that will break is `strdup()`, which returns a newly-allocated copy of a string.

To allow programs to allocate more than 1 MB of memory, or to use functions that allocate memory themselves, you can specify the `-extern` command-line option to SPIM. In this mode, programs can allocate as much memory as the host computer will allow, and functions that allocate their own memory will work. The tradeoff is that SPIM can no longer detect illegal memory accesses. Programs that access illegal memory will either exhibit undefined behavior, or will cause the entire simulator to crash and dump core. You probably want to avoid running the simulator in this mode until you're sure that your program is correct, but it allows certain programs to work that otherwise would break. Be aware that this mode of operation is still under development. The options to specify and tune this mode will likely change, and there may be some bugs. Generally it won't be necessary to use the external memory mode when running typical programs in SPIM.

[3] These instructions are real—not pseudo—MIPS instructions. SPIM translates assembler pseudoinstructions to 1–3 MIPS instructions before storing the program in memory. Each source instruction appears as a comment on the first instruction to which it is translated.

[4] If this doesn't make sense, don't worry. It comes about because of the way SPIM internally builds a virtual memory layout regardless of SPIM's load address, and it's impossible to fix up externally returned data to match these addresses.

6 Surprising Features

Although SPIM faithfully simulates the MIPS computer, it is a simulator and certain things are not identical to the actual computer. The most obvious differences are that instruction timing and the memory systems are not identical. SPIM does not simulate caches or memory latency, nor does it accurately reflect the delays for floating point operations or multiplies and divides.

Another surprise (which occurs on the real machine as well) is that a pseudoinstruction expands into several machine instructions. When single-stepping or examining memory, the instructions that you see are slightly different from the source program. The correspondence between the two sets of instructions is fairly simple since SPIM does not reorganize the instructions to fill delay slots.