SPIM Tutorial

The purpose of this tutorial is to get you acquainted with the SPIM simulator that we will be using for MP2.

Preparation

Please read Sections 2.1-2.3 and 2.9 (5th edition) of your text book. The rest of this will make so much more sense then. For more information on SPIM, refer to the SPIM documentation at "mp2-spim.pdf". SPIM has been installed on **cpeg323.ece.udel.edu** machines.

Writing an assembly program

1. *Open your favorite text editor and type (or cut and paste) the following*:

```
# comments are delimited by hash marks

.data
silly_str: .asciiz "My first MIPS program\n"

.text

main:
   li $v0, 4              # load the value "4" into register $v0
   la $a0, silly_str      # load the address of "silly_str" into register $a0
   syscall                # perform the "print_string" system call ($v0 = 4)
   jr $ra                 # return to the calling procedure
```

The above program has two parts.

- First is the data segment, tagged with the `.data` directive. The data segment is used to allocate storage and initialize global variables. The above program allocates a single variable `silly_str`. The `.asciiz` directive indicates that this variable is an ASCII string that should be terminated with a zero (that's what the z means). This statement will cause the assembler to allocate 23 bytes of space (one for each character and one more for a terminating zero) for the variable and load it with the ASCII values for the characters, followed by a zero.

- Second is the text segment, indicated by the `.text` directive. This is where we put the instructions we want the processor to execute. In the above program, there is a single function, which is called `main`. The name `main` is special; it will be the first function of our program that gets called. Our `main` function prints `silly_str` to the console and then returns. This is accomplished in four instructions:

    - `li` is short for `load immediate`; that means put a constant into a register (in this case the constant `4` in the register `$v0`).
    - `la` is short for `load address`; that means put an address into a register (in this case the address of the string `silly_str` in the register `$a0`).
    - `syscall` is short for `system call`; SPIM provides a number of operating system services that aren't really a part of MIPS assembly language, but are useful for playing with little assembly programs. We indicate to SPIM which system call to perform by putting a particular number in register `$v0`. System call number 4 is `print_string` which interprets the contents of register `$a0` as the address of a null-terminated string (i.e., a string that ends with a zero) and copies the string to the console.
    - `jr` is short for `jump register`; this instruction performs the return for us. As we will see shortly, the piece of code that calls our `main` function puts a return address into

register `$ra` (the `return address` register). The jump register command sets the processor's program counter (PC) to the contents of register `$ra`, returning execution to the calling function.

2. *Save the file as* `test.s`. The `.s` extension is typically used for assembly files.

Start spim

To start the command line version just enter

"***spim***"

at the command prompt. *(Or you may need to type "/usr/loca/spim/bin/spim" if you use hplab machines. If you still get errors, please send a help request at eecis.udel.edu.)* You will see a copyright notice and SPIM's user prompt:

```
SPIM Version 7.5 of August 14, 2009
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
(spim)
```

Loading and running a program

To load a program, you need to use a "***load***" command, and enclose the file name in double quotes. Once the program is loaded, you run it with the "***run***" command. If you typed in the program correctly, the SPIM console should pop up and print the message My first MIPS program.

```
(spim) load "test.s"
(spim) run
My first MIPS program
```

We'll now look closer at what SPIM is doing.

Stepping through the program

SPIM provides a number of features to support debugging that you will find useful while developing your assembly programs. The two most important are stepping and breakpoints. Stepping allows you to look at the effect of your program instruction by instruction. Breakpoints allow you to stop the execution just before a particular instruction is executed.

Stepping: To execute a program one line at a time, use the "***step***" command. Each time you step, SPIM will display a line of your program and execute the displayed instruction. Each line describes one instruction with the four fields, which (from left to right) are: the instruction address, the binary representation of the instruction, the machine instruction, and the assembly code statement that was translated into the instruction. The last two columns differ in two ways: 1) register names ($sp) have been translated into register numbers ($29), and 2) some assembly instructions (e.g., `li`) don't actually exist on the hardware so must be translated into instructions that the machine implements (e.g., `ori`). In particular, the four lines of assembly code in test.s are translated into 10 machine instructions.

You do not have to type "step" for each new step. If you hit the enter key, SPIM will repeat the last command executed. As you step through the program, you can display register values with the "***print***" command. To see the current value in register $v0, for example, you may enter "print $v0". Moreover, you can use "print PC" to get the address currently held in the program counter (PC) register. The instruction at the address held on PC is the next instruction to be executed. Use "***continue***" to resume running when you are finished stepping through your program.

Breakpoints: You can set breakpoints with the "***breakpoint***" command. To do this, type either an address (e.g. ,0x0040002c) or label (e.g., L1) as an argument (before you can set a breakpoint on a label it must be declared ".global"). To see what breakpoints are set use the "***list***" command and to clear a breakpoint use "***delete***" with the address of the breakpoint to be deleted as the argument. If

breakpoints are set, "running" or "continuing" a program will execute until a breakpoint is reached. You can the step through your program from that point.

Now, it is your turn to learn how to use stepping and breakpoints to go through the program.

- If you want to run the test program again or modify the program without leaving and restarting SPIM, you must return the machine back to its intial state by running the "***reinitialize***" command. Load the assembly code by "**load test.s**".

- Run the "**print_all_regs hex**" command to print all registers in hexadecimal. Please go through every register and look at their initial values.

- Run the "**step**" command. The first instruction is a lw or load word instruction that loads a value into register $4. The value was loaded from the top of the stack (a 0 offset from the stack pointer register $sp). Executing the first instruction also changed the contents of the PC register.

- Run the "**step**" command two more times. Instructions 0x00400004 and 0x00400008 are both of type addiu or Add Immediate Unsigned. In particular, the instruction 0x00400008 means add a constant (in this case 4) to a register ($5) and put it in another register ($6).

Single stepping can become tedious quickly, so:

- Set a breakpoint on address 0x00400010, by running "**breakpoint 0x00400010**".

- Run the "**continue**" command to skip some of the scaffolding code. When spim shows "Breakpoint encountered at 0x00400010"

- Run the "**step**" command two more times. The second instruction (0x00400014) is of type jal, which is used to implement function calls. The jump specifies an address (in this case `0x00400024`) of the next instruction to be executed; this address will be written to the PC register. In addition, a `link` operation is performed to allow the called function to return to this function. This is accomplished by writing PC+4 to the return address ($ra) register.

- Run the "**step**" command one more time. Finally, we made it to the code you typed in. In the displayed line, you can see that SPIM converted the `li` instruction we used into an `ori` instruction. It stands for `(logical) OR immediate`. The `li` instruction is one of the pseudo instructions. It isn't really a MIPS machine instruction, but is provided to simplify assembly programming. If we were to try to load a very large immediate, it would take multiple MIPS instructions to compute the immediate. For a small immediate like 4, SPIM performs a logical OR of the immediate with the $0 register (whose contents is always the value 0).

- Run the "**step**" command one more time. You can see that the `la` pseudo instruction is converted into the MIPS primitive lui or load upper immediate. `lui` is used for setting the upper 16 bits of a register. It turns out that SPIM places `silly_str` at address `0x10010000`, the bottom 16 bits of which are all zeros. As a result SPIM can write the address of `silly_str` into a register with a single instruction that writes (4097 << 16) into register $4. The string in our assembly file has been converted into ASCII and runs from address `0x10010000` to `0x10010016`. Each ASCII character is eight bits, so four are packed together into the 32-bit data words in the data segment.

- Run the "**step**" command once. The instruction is the `syscall`. You will see that the string is written to the console line.

- Run the "**step**" command. The last instruction of our program performs the return. The $ra register should still contain the value placed there by the `jal` instruction. The jump register instruction will copy the contents of the $ra register back to the PC register.

Modifying a program

1. *Return to your text editor. Replace the contents of* `silly_str` *with "Hello World".* It should look something like:
   ```
   silly_str: .asciiz "Hello World\n"
   ```

2. If you did not leave and restart SPIM, you must run the "***reinitialize***" command.
3. Load the assembly code by "**load test.s**".
4. Run the program by "**run**" to see "Hello World" written into the console.

You can use xspim instead of spim. More information about xspim can be found at mp2-xspim.pdf.