XSPIM tutorial

Compiled from http://www-courses.cs.uiuc.edu/~cs232/assignments/mp0/mp0.html

Writing an assembly program

1. *Open your favorite text editor and type 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:

1. `li` is short for `load immediate`; that means put a constant into a register (in this case the constant `4` in the register `$v0`).
2. `la` is short for `load address`; that means put an address into a register (in this case the address of the string `silly_str` in the register `$a0`).
3. `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.
4. `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 xspim

Because the wizards at EWS have magically put `~cs232/MachineType/bin` in your path, you should be able to run xspim by simply typing:

```
    xspim &
```

For simplicity, you should do this in the directory in which you've saved your assembly file. If for some reason your path isn't set you can type the slightly more long winded: `~cs232/SunOS/bin/xspim &` or `~cs232/Linux/bin/xspim &`, depending on your platform.

It is important to note that SPIM behaves slightly differently on the two platforms, because x86 is little endian and Sun is big endian. We'll discuss this difference in class; I'm merely pointing it out here so that you aren't surprised if things don't look exactly like in the tutorial.

Loading and running a program

1. *Click on the load button*. That will bring up a dialog box.

2. *Type in the name of the file* (`test.s`) *and then click the button labeled* `assembly file`. You'll see the contents of the the `Text Segments` and `Data Segments` window update themselves.

3. *Click the* `run` *button to run the program*. It will pop up a window indicating the PC to begin execution (`0x00400000`) and the assembly file to execute `test.s`. SPIM should fill in these fields correctly and you will only need to click `ok`. If you typed in the program correctly, the `SPIM console` should pop up and print the message `My first MIPS program`. You might have noticed that some of the register values at the top of the main xspim window changed also. 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.

To run the test program again, you need to return the machine back to its initial state. This can be done with the reload button:

1. *Click the reload button and drag down to select* `assembly file`. The PC register (in the upper left corner) should have reverted back to `0x00400000`.

The first line in the `Text Segments` window should also be highlighted. Each line in the text segment window describes one instruction; the four fields (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`). The first instruction is highlighted because it is at the address currently held in the program counter (PC) register, and, thus, is the next instruction to be executed.

The first 9 instructions (`0x00400000-0x00400020`) shown in the `Text Segments` window don't actually come from your assembly program. The `test.s` code begins at address `0x00400024`.

2. *Click on the step button.* A window will pop up. Again, we will accept the default parameters (single instruction stepping and our `test.s` program).

3. *Click on the `step` button in the dialog window to execute the first instruction.* This first instruction is a `lw` or `load word` instruction that loads a value into register $4. What value was loaded? _______________

This value was loaded from the top of the stack (a 0 offset from the stack pointer register $sp). What is the contents of the $sp register? ____________

Looking at the window labeled `Data Segments`, you can confirm that the correct value was read from memory:

4. *Find the address under the word STACK.* To the right of this address there are up to four 32-bit data words; the leftmost one is at the address in brackets, each position to the right adds 4 to the address. What value is stored at 4($sp)? ___________________

Executing the first instruction also changed the contents of the PC register. What is the new value of the PC? __________________

5. *Click the step button (in the dialog) two more times so that instruction* `0x0040000c` *is highlighted.* Instruction `0x00400008` is of type `addiu` or `ADD Immediate Unsigned`, which means add a constant (in this case 4) to a register (R5) and put it in another register (R6). What value is written into R6? ___________________

Single stepping can become tedious quickly, so:

6. *Set a breakpoint on address* `0x00400010`*, by clicking the* `breakpoints` *button, typing the address into the dialog box, and clicking* `add`.

7. *Click the* `continue` *button in the step dialog box to skip some of the scaffolding code.* When xspim stops, it pops up a dialog box; click on `abort command` to dismiss it. Also,

the `Text Segments` window is highlighting the instruction at which we set the break point, which it has converted to `break $1`. SPIM's debugging support (like gdb or any other debugger) implements breakpoints by converting instructions into illegal instructions (in this case the binary representation `0x00000000`); it stores the original instruction elsewhere so that it is not lost. If you want SPIM to make the original instruction visible again, you can turn off the breakpoint (click breakpoints, type the address in the box, and click delete) before stepping to the next instruction. In any case, single step to the `jal` instruction.

The `jal` or `jump and link` instruction 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. What value is written to register R31?

____

Finally, we made it to the code you typed in. In the text segment window, 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).

8. *Step one instruction to verify that register R2 is written with the value 4.*

Similarly, 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 R4. Looking at the `Data Segments` window, in the DATA section, you can see address `0x10010000`. 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.

The next instruction is the `syscall`:

9. *Step across that instruction and verify that the string is written to the console window.*

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.

10. *Step across the last instruction.* Did anything happen to the $ra register's contents?

____

Modifying the program

1. *Return to your text editor. Replace the contents of* `silly_str` *with your name.* It should look something like:

```
    silly_str: .asciiz "Joe User\n"
```

2. *Save the file, and reload it using the* `reload` *button. Click on the reload button and drag down until* `assembly file` *item is highlighted, then release the button.*

You should see the contents of the `Data Segments` window change (as well as the re-initializing of the register and the `Text Segments` window). Any breakpoints that were set have been deleted.

3. *Verify this by clicking on the* `breakpoints` *button and selecting the* `list` *option*. In the message window at the bottom of xspim's main window you should see the message:

```
    No breakpoints set
```

4. *Run the program to see your name written into the console window*.