
1

Getting Started With SPIM

Objectives

After this lab you will be able to:

- Load MIPS programs (assembly language) and execute them
- Examine memory locations
- Examine registers
- Execute programs step by step
- Set/remove breakpoints

Introduction

SPIM S20 is a simulator that can run programs for the MIPS R2000 and R3000 architectures. The simulator may load and execute assembly language programs.

The process through which a source file (assembly language) is translated to an executable file contains two stages:

- assembling (implemented by the *assembler*)
- linking (implemented by the *linker*)

An executable file must be loaded before the CPU can actually run that program. This is done by a *loader*. Figure 1.1 shows the relation between the processes involved in translation and loading and various types of files they manipulate.

The assembler performs the translation of an assembly language module into machine code. Note that a program may have several modules, each of them a part of the program. This is usually the case when you build an application from several files.

The output of the assembler is an object module for each source module. Object modules contain machine code. The translation of a module is not complete if the module uses a symbol (a label) that is defined in a different module or is part of a library.

The linker enters now the picture. It's main goal is to resolve *external references*. In other words the linker

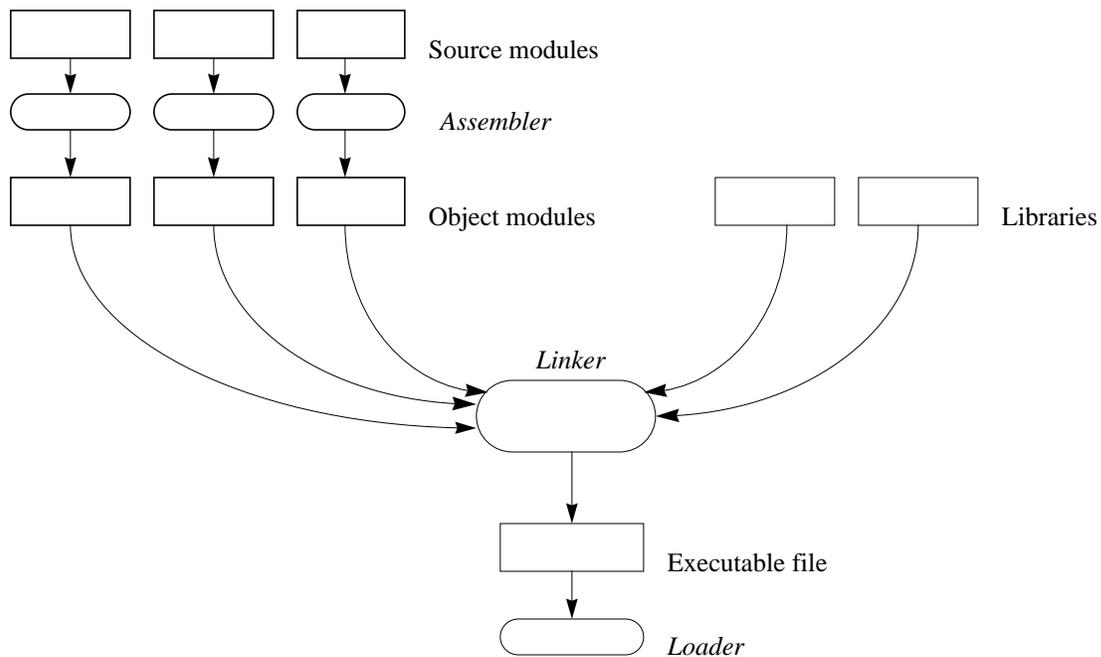


Figure 1.1 The translation process

will match a symbol used in a source module with its definition found in a different module or in a library. The output of the linker is an executable file.

The loader may be as simple as a program that copies the executable in the memory at the appropriate location(s), or it may be more involved if loading occurs at different locations at different times. This second case is called *relocation* and it involves the translation of some addresses as to correspond with the actual loading address.

File Names

File suffixes indicate the type of the file. Files that contain assembly code have the suffix “.s” or “.asm”. Compilers use the first convention.

Files with object programs have the suffix “.o” and executables don’t usually have any suffix.

Translation in SPIM

The process of translation in SPIM is transparent to the user. This means that you don’t have to deal with an assembler, a linker and a loader as separate programs. Provided you have written a correct assembly language program, the only thing you have to do is to start the simulator and then indicate what program you want to execute. The whole process of translation is hidden.

Laboratory 1: Prelab

Date _____ Section _____

Name _____

Step 1

Using a text editor, enter the program P.1. The sharp sign (#) starts a comment, a name followed by a colon (:) is a label, and names that start with a period (.) are assembler directives.

P.1:

```

        .data 0x10000000
msg1:   .asciiz "Please enter an integer number: "

        .text
        .globl main

# Inside main there are some calls (syscall) which will change the
# value in $31 ($ra) which initially contains the return address
# from main. This needs to be saved.

main:   addu $s0, $ra, $0      # save $31 in $16
        li $v0, 4            # system call for print_str
        la $a0, msg1         # address of string to print
        syscall

# now get an integer from the user
        li $v0, 5            # system call for read_int
        syscall              # the integer placed in $v0

# do some computation here with the number
        addu $t0, $v0, $0    # move the number in $t0
        sll $t0, $t0, 2      # last digit of your SSN instead of 2

# print the result
        li $v0, 1            # system call for print_int
        addu $a0, $t0, $0    # move number to print in $a0
        syscall

# restore now the return address in $ra and return from main
        addu $ra, $0, $s0    # return address back in $31
        jr $ra              # return from main

```

Before you continue make sure you have entered the last digit of your SSN instead of 2 in the instruction `sll $t0, $t0, 2`

Let's note here, before we move on, that we save the content of register `$ra` into another register. The register `$ra` (used for the call/return mechanism) has to be saved when you enter main, only if you either call system routines (using `syscall`) or if you call your own routines (there will be a laboratory dedicated to this topic). Saving `$ra` in `$s0` (or in any other register for that matter) only works if

- there is only one call level (in other words there are no recursive calls of the routine)
- the routines you are calling do not modify the register you use for saving

Step 2

Save the file under the name `lab1.1.asm`. Save the file in the same directory where the simulator itself is. Otherwise you will have to change the search path such that the system will be able to execute the simulator no matter what the current working directory is.

Here we use the `.asm` extension for the file name as to differentiate between hand written code and compiler generated assembly code. A compiler would use the `.s` extension for the file containing the assembly code.

Step 3

Start the SPIM simulator by typing `spim` at the prompt. You will see a copyright message, followed by a message indicating that the trap handler has been loaded.

In case you get an error message that says something like "spim: command not found", then you must make sure the directory where `spim` is located is in your search path.

Step 4

At the `(spim)` prompt type

```
load "lab1.1.asm"
```

If you have any error messages go back to Step 1 and make sure you have not made any mistakes when typing the program. If there is no error message, then your program has been translated and you can run it.

Step 5

At the `(spim)` prompt type

```
run
```

to have the program execute. You will be prompted for an integer number; after you enter it, the program will print a result and exit. You know the program has finished to execute since the simulator returns to the `(spim)` prompt.

You can run the program again either by typing `run` at the prompt or by simply pressing the Enter key (which re-executes the last command).

Step 6

You now try to figure out what program `lab1.1.asm` does. Run it several times with various input data. Use

both positive and negative integers. Fill out the following table:

Test cases for *lab1.1.asm*

Input number	Output number

Step 7

What is the formula that describes the relation between the output and the input?

Laboratory 1: Inlab

Date _____ Section _____

Name _____

Using SPIM to Learn About the MIPS Architecture

Using the simulator you will peek into the memory and into various general purpose registers. You will also execute a program step by step. Stepping may be very useful for debugging. Setting breakpoints in a program is another valuable debugging aide: you will be playing with these too.

Step 1

Start the spim simulator and load the program *lab1.1.asm*

Step 2

Type `print_sym` at the (`spim`) prompt. You will see a listing of all global symbols. Global symbols are those that are preceded by the assembler directive `‘.globl’`. For each symbol the address in memory where the labeled instruction is stored, is also printed.

Symbol	Address

Exit the simulator.

Step 3

Modify *lab1.1.asm* as follows: replace the first line after the line labeled `‘main’` with a line that reads

```
label1:    li $v0, 4           # system call for print_int
```

Save the program as *lab1.2.asm*. The only difference between the two programs is the label `‘label1’`

Step 4

Start the spim simulator, load the program *lab1.2.asm* and print the list of global symbols.

Symbol	Address

As you can see there is no difference between the listing you obtain at this step and the one at Step 2. The reason is that 'labell' is a *local* symbol. Local symbols are visible only within the module in which they are defined. A *global* symbol is visible inside and outside the module where it is defined. A global symbol can therefore be referenced from other modules.

Step 5

We now know where the program is stored in memory. It is the address returned by `print_sym` for the symbol 'main'. Let's call it *main_address*. To see what is stored in memory starting with that address do

```
print main_address
```

at the prompt. The address returned by `print_sym` is in hexadecimal so make sure you don't forget the 0x when you type it. The `print` command prints a line that contains (in this order):

- the address in memory
- the hexadecimal representation of the instruction
- the native representation of instructions (no symbolic names for registers)¹
- the textual instruction as it appears in the source file

Q 1:

What is the size of an instruction (in bytes)?

Instruction size =

Step 6

Use the `print` command, starting with the address of the symbol '`__start`' and fill the table below

Label	Address	Native instruction	Source instruction

1. More about this when we discuss synthetic instructions.

Label	Address (PC)	Native instruction	Source instruction

Q 2:

Why does this table differ from the table you got at step 6?

Step 8

Load again *lab1.2.asm*. You will get an error message indicating that some label(s) have multiple definitions. This happens because the program *lab1.2.asm* has already been loaded. If there is a need to reload a program, then the way to do it is

```
reinit
load <program_name>
```

reinit will clear the memory and the registers. Make sure the name of the program you want to load is between double quotes.

Step 9

Let's assume you don't want to step through the program. Instead, you want to stop every time *right before* some instruction is executed. This allows you to see what is in memory or in registers right before the instruction is executed.

Set a breakpoint at the second syscall in your program.

```
breakpoint <address>
```

where <address> can be found in the table you filled out at step 7. Now you can run the program, up to the first breakpoint encountered (there is only one at this time).

```
run
```

Use the `print` command to view the registers just before the syscall is executed. For example `print $0` will print the content of register 0. Fill the 'Before the syscall' column of the following table

Register number	Register name	Before the syscall	After the syscall	Changed
0	zero			

Register number	Register name	Before the syscall	After the syscall	Changed
1	\$at			
2	\$v0			
3	\$v1			
4	\$a0			
5	\$a1			
6	\$a2			
7	\$a3			
8	\$t0			
9	\$t1			
10	\$t2			
11	\$t3			
12	\$t4			
13	\$t5			
14	\$t6			
15	\$t7			
16	\$s0			
17	\$s1			
18	\$s2			
19	\$s3			
20	\$s4			
21	\$s5			
22	\$s6			
23	\$s7			
24	\$t8			
25	\$t9			
26	\$k0			
27	\$k1			
28	\$gp			
29	\$sp			
30	\$fp			
31	\$ra			

Step 10

Type `step` at the `(spim)` prompt to have the syscall executed. Before you can do anything else you must

supply an integer. This happens because the program executes a syscall, a call to a system function, in this case one that reads an integer from the keyboard.

Fill out the 'After the syscall' column of the above table. In the column 'Changed', mark with a star registers that have changed.

Q 3:

Some registers have changed during the syscall execution. Can you assume that syscall uses only these registers? Explain.

Q 4:

The first instruction in your program moves the content of register `$ra` to register `$s0`. The content of that register is a memory address. What is stored in memory at that address?

Q 5:

This question is related to the previous one. When will be executed the instruction stored at the address in `$s0`? Indicate the instruction that immediately precedes it in execution.

Step 11

You may set as many breakpoints as you want in a program. If you want to remove them, then you have to do

```
delete <address>
```

where `<address>` is the address at which the breakpoint has been set. If you have more than one and you have forgot where they are, then you can list them.

```
list
```

will produce a listing with all breakpoints you have set.

Remove the breakpoint you have previously set and run the program again to make sure it has been removed.

Laboratory 1: Postlab

Date _____ Section _____

Name _____

Learn More About MIPS

In this exercise you will be using the floating-point registers of MIPS.

Background

For practical reasons, the original definition of the R2000 architecture defined a MIPS processor as composed of

- integer unit (the actual CPU)
- coprocessors

The idea was that the technology just did not allow to integrate everything on a single silicon die. Therefore coprocessors could be separate integrated circuits, or could just be software emulators (i.e. for floating point) if the price was a serious concern. Defining coprocessors neatly separates the architectural definition from the implementation constraints or details. Keep in mind that the same architecture may have several implementations, each using possibly different technologies and having different performance.

SPIM simulates two coprocessors

- coprocessor 0: handles interrupts, exceptions and the virtual memory system
- coprocessor 1: floating point unit (FPU)

The FPU performs operations on

- single precision floating point numbers (32 bit representation); a declaration like `float a=1.5;` in C would reserve space for a variable called `a` which is single precision floating point, and is initialized to 1.5
- double precision floating point numbers (64 bit representation); a declaration like `float a=1.5;` in C would reserve space for a variable called `a` which is double precision floating point, and is initialized to 1.5

The coprocessor has 32 registers, numbered from 0 to 31 (their names are `$f0` to `$f31`). Each register is 32 bit wide. To accommodate doubles registers are grouped together (0 with 1, 2 with 3, ..., 30 with 31). To simplify things, floating point operations use only even numbered registers.

Step 1

Create a program (use *lab1.1.asm* as a model) that reads a float (i.e. single precision number) from the keyboard and then outputs it.

You will need to look at the instruction set to find out what instruction to use for moving a float from one floating point register to another (`addu $f12, $f0, $0` will not work).

Save the program as *lab1.3.asm*. Run the program and fill out the ‘Single precision’ section of the following table (the content of registers after program finished). The input you type at the keyboard when prompted will be the last four digits of your Social Security Number, followed by a period (.), followed by the current year (four digits).

Step 2

Create a program that reads a double (i.e. double precision number) from the keyboard and then outputs it. Save the program as *lab1.4.asm*. Run the program and fill out the ‘Double precision’ section of the following table (the content of registers after program finished). The input you type at the keyboard when prompted will be the same as at Step 1.

Register	Single precision	Double precision
\$f0		
\$f2		
\$f4		
\$f6		
\$f8		
\$f10		
\$f12		
\$f14		
\$f16		
\$f18		
\$f20		
\$f22		
\$f24		
\$f26		
\$f28		
\$f30		

Step 3

Return to your lab instructor copies of *lab1.3.asm* and *lab1.4.asm* together with this postlab description. Ask your lab instructor whether copies of programs must be on paper (hardcopies), e-mail or both.