Laboratory 1: Inlab

Date	10/09/2014	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
g_eoth	0x00400024
g_start	0x00400000

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
g_eoth	0x00400024
g_start	0x00400000

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 'label1' 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 = 32

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
[00400000]	8fa40000	lw $4, 0($29)	183: lw $a0 0($sp) # argc
[00400004]	27a50004	addiu $5, $29, 4	184: addiu $a1 $sp 4 # argv
[00400008]	24a60004	addiu $6, $5, 4	185: addiu $a2 $a1 4 # envp
[0040000c]	00041080	sll $2, $4, 2	186: sll $v0 $a0 2

1. More about this when we discuss synthetic instructions.

Label	Address	Native instruction	Source instruction
[00400010]	00c23021	addu $6, $6, $2	187: addu $a2 $a2 $v0
[00400014]	0c100009	jal 0x00400024 [main]	188: jal main
[00400018]	00000000	nop	189: nop
[0040001c]	3402000a	ori $2, $0, 10	191: li $v0 10
[00400020]	0000000c	syscall	192: syscall # syscall 10 (exit)
[00400024]	03e08021	addu $16, $31, $0	8: addu $s0, $ra, $0
[00400028]	34020004	ori $2, $0, 4	9: li $v0, 4
[0040002c]	3c041000	lui $4, 4096 [msg1]	11: la $a0, msg1
[00400030]	0000000c	syscall	12: syscall
[00400034]	34020005	ori $2, $0, 5	14: li $v0, 5

Step 7

The `step <N>` command allows the user to execute a program step by step. If the optional argument <N> is missing, then the simulator will print the instruction, execute it and stop. The user can then see (using the `print` command) how a specific instruction has modified registers, memory, etc. If the optional argument <N> is present, then the simulator will stop after executing N instructions. For example, `step 3` will tell the simulator to execute three instructions and then stop.

The format of line(s) printed by the `step` command is the same as the format for the `print` command. The first field is the address of the executed instruction (the Program Counter).

Use `step 1` (or simply `step`) to fill out the following table

Label	Address (PC)	Native instruction	Source instruction
[00400014]	0c100009	jal 0x00400024 [main] ;	188: jal main
[0040003c]	00404021	addu $8, $2, $0 ;	22: addu $t0, $v0, $0
[00400040]	00084080	sll $8, $8, 2 ;	23: sll $t0, $t0, 2
[00400044]	34020001	ori $2, $0, 1 ;	25: li $v0, 1
[00400048]	01002021	addu $4, $8, $0 ;	26: addu $a0, $t0, $0
[0040004c]	0000000c	syscall ;	27: syscall
[00400050]	0010f821	addu $31, $0, $16 ;	33: addu $ra, $0, $s0 # return address back in $31
[00400054]	03e00008	jr $31 ;	34: jr $ra # return from main
[00400018]	00000000	nop ;	189: nop
[0040001c]	3402000a	ori $2, $0, 10 ;	191: li $v0 10
[00400020]	0000000c	syscall ;	192: syscall # syscall 10 (exit)
[00400004]	27a50004	addiu $5, $29, 4 ;	184: addiu $a1 $sp 4 # argv
[00400008]	24a60004	addiu $6, $5, 4 ;	185: addiu $a2 $a1 4 # envp

Label	Address (PC)	Native instruction	Source instruction
[0040000c]	00041080	sll $2, $4, 2 ;	186: sll $v0 $a0 2

Q 2:

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

Because this table follows the processes that how the program runs. The table that got at step 6 shows how the program stored in the memory.

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**	0	0	0

Register number	Register name	Before the syscall	After the syscall	Changed
1	**`$at`**	0	0	0
2	**`$v0`**	4	a	6
3	**`$v1`**	0	0	0
4	**`$a0`**	0	0	0
5	**`$a1`**	7ffffe70	7ffffe70	0
6	**`$a2`**	7ffffe78	7ffffe78	0
7	**`$a3`**	0	0	0
8	**`$t0`**	0	0	0
9	**`$t1`**	0	0	0
10	**`$t2`**	0	0	0
11	**`$t3`**	0	0	0
12	**`$t4`**	0	0	0
13	**`$t5`**	0	0	0
14	**`$t6`**	0	0	0
15	**`$t7`**	0	0	0
16	**`$s0`**	400018	400018	0
17	**`$s1`**	0	0	0
18	**`$s2`**	0	0	0
19	**`$s3`**	0	0	0
20	**`$s4`**	0	0	0
21	**`$s5`**	0	0	0
22	**`$s6`**	0	0	0
23	**`$s7`**	0	0	0
24	**`$t8`**	0	0	0
25	**`$t9`**	0	0	0
26	**`$k0`**	0	0	0
27	**`$k1`**	0	0	0
28	**`$gp`**	10008000	10008000	0
29	**`$sp`**	7ffffe6c	7ffffe6c	0
30	**`$fp`**	0	0	0
31	**`$ra`**	400018	400018	0

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.

No.There is only one experience that cannot used as a common sense. Different registers have different jobs, and what the syscall do with are not certain.

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?

The words that will displayed on screen.

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.

When the program runs to the second loop.

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 precison 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 accomodate 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.