Computer Architecture:
Introduction to SPIM Simulator

Date: 2019/10/01

Outline

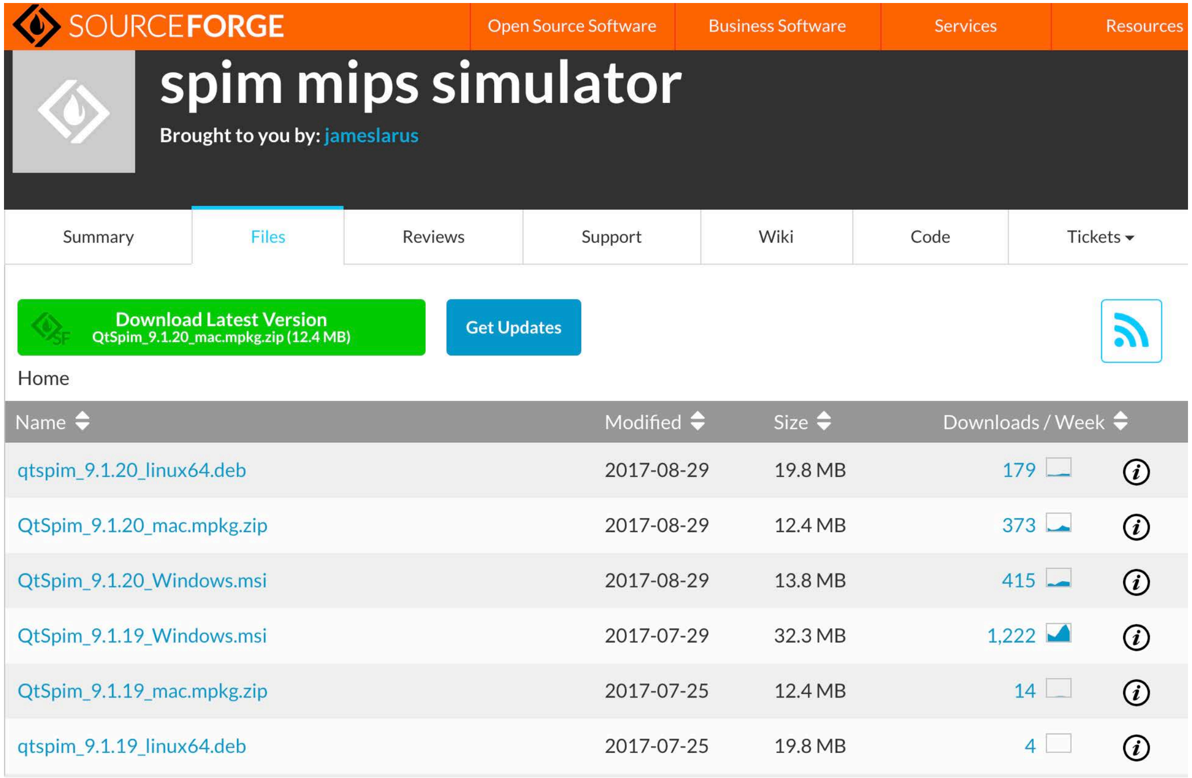
- Introduction
- General layout & SPIM I/O
- Example
- Homework

Introduction

- Developed by James Larus while working as a professor at University of Wisconsin-Madison
- SPIM is a MIPS processor simulator, designed to run assembly language code with this architecture
- Does not simulate caches or memory latency
- Pseudo-Instructions
  - Extend the instruction set for convenience
- Provides a few operating-system-like services
- More information:
  - http://pages.cs.wisc.edu/~larus/spim.html

Installation

- Download
  - http://sourceforge.net/projects/spimsimulator/files/

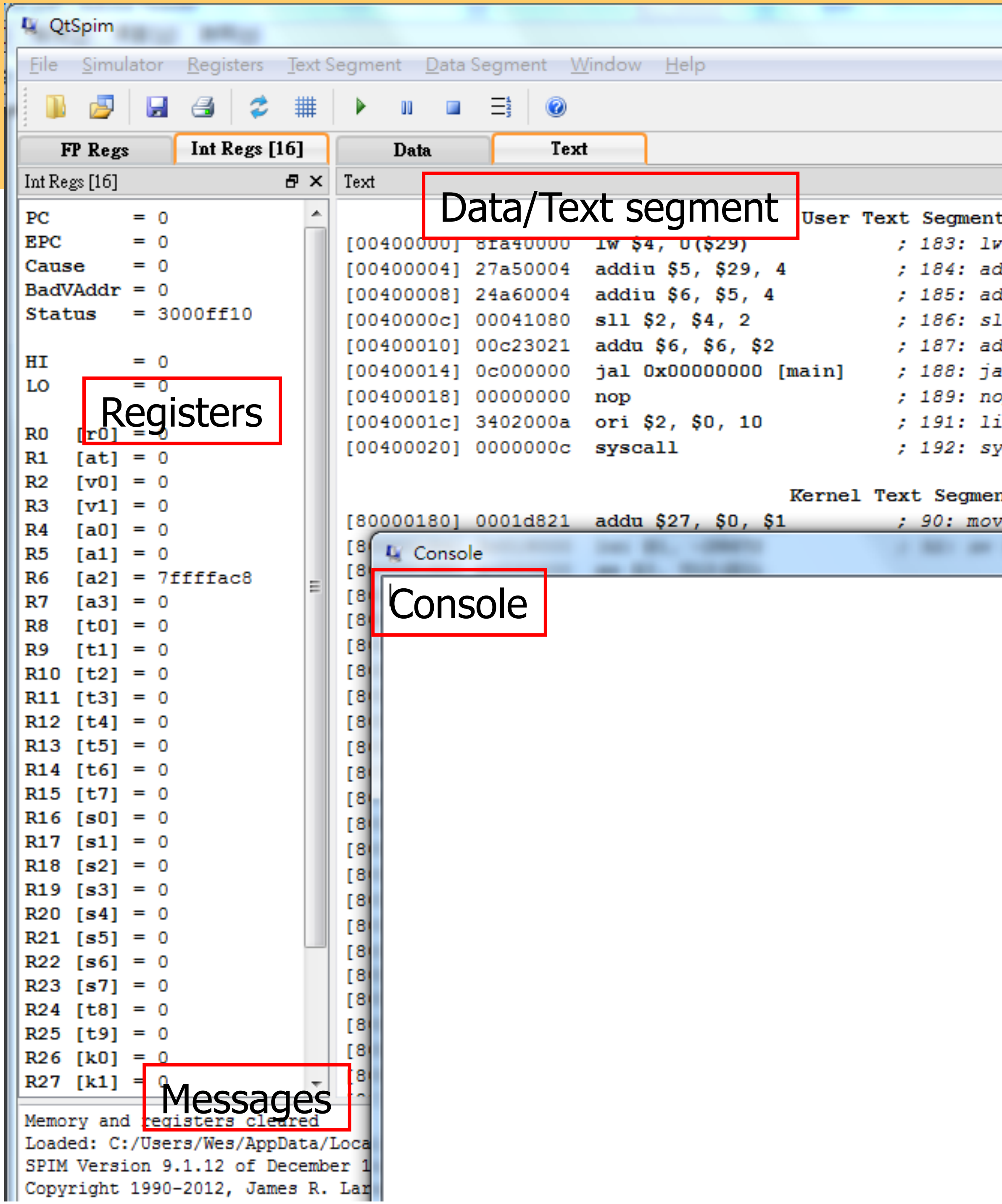


Name	Modified	Size	Downloads / Week
qtspim_9.1.20_linux64.deb	2017-08-29	19.8 MB	179
QtSpim_9.1.20_mac.mpkg.zip	2017-08-29	12.4 MB	373
QtSpim_9.1.20_Windows.msi	2017-08-29	13.8 MB	415
QtSpim_9.1.19_Windows.msi	2017-07-29	32.3 MB	1,222
QtSpim_9.1.19_mac.mpkg.zip	2017-07-25	12.4 MB	14
qtspim_9.1.19_linux64.deb	2017-07-25	19.8 MB	4

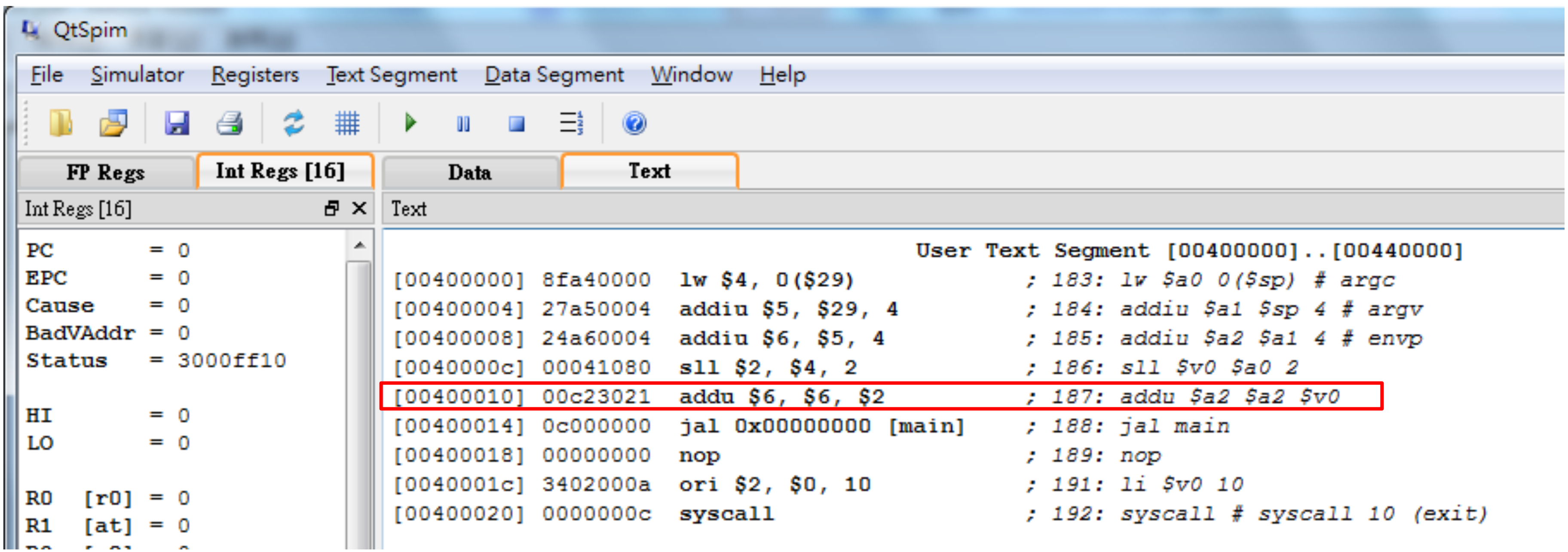
國立臺灣大學 National Taiwan University

QtSpim

- Registers
  - Shows the values of all registers
- Data segment
  - shows the data loaded into the program's memory and the program's stack
- Text segment
  - Shows instructions
- Messages
- Console



QtSpim



[00400010]	00c23021	**addu $6, $6, $2** ;	*187: addu $a2 $a2 $v0*

- memory address of instruction
- Instruction`s numerical encoding
- Instruction`s mnemonic description
- Line number in assembly file
- Source code in assembly file

General Layout

- Data definitions start with .data directive
- Code definition starts with .text directive
- Usually have a bunch of subroutine definitions and a “main”

Data Type

- .word, .half - 32/16 bit integer
- .byte - 8 bit integer (similar to ‘char’ type in C)
- .double, .float - floating point
- .ascii, .asciiz - string (asciiz is null terminated)
  - Strings are enclosed in double-quotas(”)
  - Special characters in strings follow the C convention
- newline(\n), tab(\t), quote(\”)

- A-47 Assembler syntax

Pseudo-Instructions

- Pseudo instructions are instructions that exist in the SPIM assembler, but not designed in MIPS processors
- When machine code is generated, the pseudo instructions are converted to real instructions

Pseudo instruction		Real instruction
move $5, $3	→	add $5, $3, $0
neg $8, $9	→	sub $8, $0, $9
li $8, 44	→	addi $8, $0, 44 *or* ori $8, $0, 44

Pseudo-Instructions

Pseudo-instruction	Expansion	Comment
`blt $3, $4, dest`	`slt $1, $3, $4` `bne $1, $0, dest`	
`bge $3, $4, dest`	`slt $1, $3, $4` `beq $1, $0, dest`	`$3 >= $4` is the opposite of `$3 < $4`
`bgt $3, $4, dest`	`slt $1, $4, $3` `bne $1, $0, dest`	`$3 > $4` same as `$4 < $3`
`ble $3, $4, dest`	`slt $1, $4, $3` `beq $1, $0, dest`	`$3 <= $4` is the opposite of `$3 > $4`

國立臺灣大學 National Taiwan University

SPIM I/O

- There is also a small number of system call commands to communicate with the console window of the SPIM simulator
- A program loads the system call code into register \$v0 and arguments into registers \$a0–\$a3 (or \$f12 for floating-point values)
- System calls that return values put their results in register \$v0 (or \$f0 for floating-point results)

System Call

Service	System call code	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		
print_char	11	$a0 = char	
read_char	12		char (in $v0)
open	13	$a0 = filename (string), $a1 = flags, $a2 = mode	file descriptor (in $a0)
read	14	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars read (in $a0)
write	15	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars written (in $a0)
close	16	$a0 = file descriptor	
exit2	17	$a0 = result	

Fig. A.9.1

```
move $a0, $s0
li $v0, 1
syscall
# print result
```

```
li $v0, 5
syscall
# read an integer
# result in $v0
   from console
```

```
li $v0, 10
syscall
# exit program
```

國立臺灣大學 National Taiwan University

Example

- Fibonacci Recurrence

$$\text{fib}(n) = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{otherwise} \end{cases}$$

- C program

```
int fib(int n)
{
   if (n <= 1)
      return n;
   else
      return fib(n-1) + fib(n-2);
}
```

Example (Fibonacci Recurrence)

```
.data
str1: .asciiz  "Input an integer: \n"
str2: .asciiz  "\nResult: "

.text
main:
    li $v0, 4
    la $a0, str1
    syscall
```

print srting

```
    li $v0, 5
    syscall
```

read integer from user and store in $v0

```
    move $a0, $v0
    jal fib
    move $t0, $v0
```

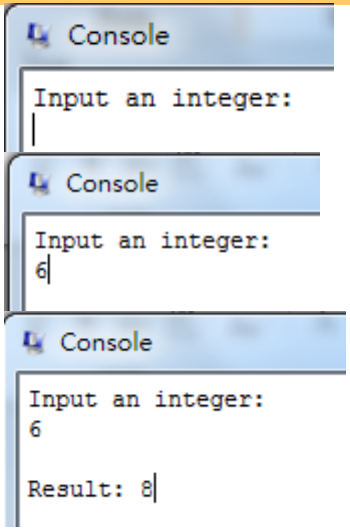
```
li $v0, 4
la $a0, str2
syscall
```

```
move $a0, $t0
li $v0, 1
syscall
```

print integer

```
li $v0, 10
syscall
```

Exit program



National Taiwan University

Example (Fibonacci Recurrence)

```
fib:
    bgt $a0, 1, recurse
    move $v0, $a0
    jr $ra
```

```
if (n <= 1)
  return n;
```

```
recurse:
    sub $sp, $sp, 12
    sw $ra, 0($sp)
    sw $a0, 4($sp)
```

```
    addi $a0, $a0, -1
    jal fib
    sw $v0, 8($sp)
```

```
    lw $a0, 4($sp)
    addi $a0, $a0, -2
    jal fib
```

```
    lw $v1, 8($sp)
    add $v0, $v0, $v1
```

```
lw $ra, 0($sp)
addi $sp, $sp, 12
jr$ra
```

```
int fib(int n)
{
  if (n <= 1)
    return n;
  else
    return fib(n-1) + fib(n-2)
}
```

Register View:

For each call,

input is in $a0

output is in $v0

how to handle result of f(n-1) and f(n-2)?

memory stack layout for a frame

SP' = SP-12

for ra
for n
for f(n-1)

SP

Example (Fibonacci Recurrence)

```
if (n <= 1)
  return n;
```

```
fib:
   bgt $a0, 1, recurse
   move $v0, $a0
   jr $ra
```

```
recurse:
   sub $sp, $sp, 12
   sw $ra, 0($sp)
   sw $a0, 4($sp)
```

First save $ra and the argument $a0. An extra word is allocated on the stack to save the result of fib(n-1).

```
   addi $a0, $a0, -1
   jal fib
   sw $v0, 8($sp)
```

The argument n is already in $a0, so we can decrement it and then "jal fib" to implement the fib(n-1) call. The result is put into the stack.

```
   lw $a0, 4($sp)
   addi $a0, $a0, -2
   jal fib
```

Retrieve n, and then call fib(n-2).

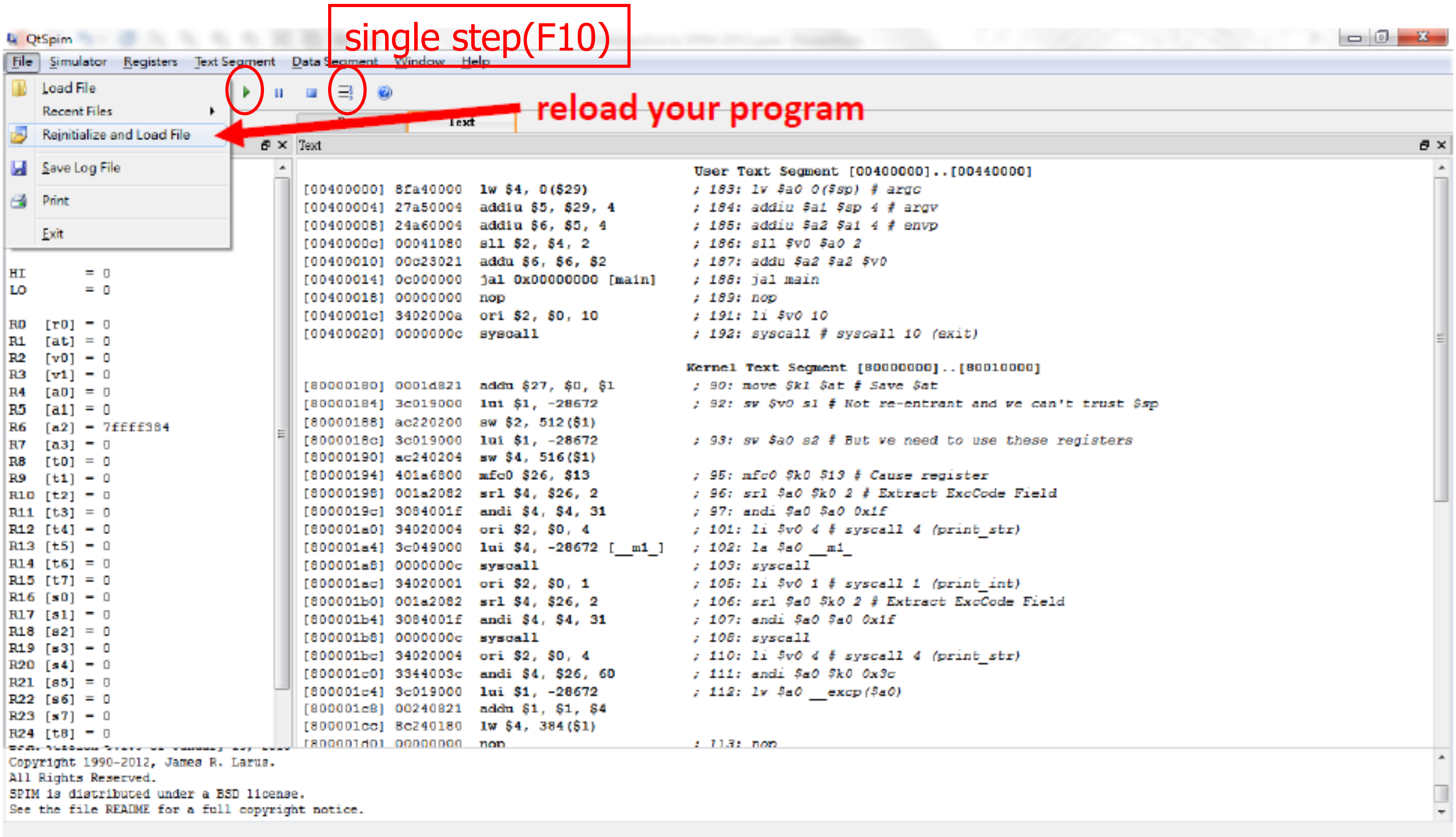
```
   lw $v1, 8($sp)
   add $v0, $v0, $v1
```

The results are summed and put in $v0.

```
lw $ra, 0($sp)
addi $sp, $sp, 12
jr$ra
```

We only need to restore $ra before popping our frame and saying bye-bye.

Load Your Program



References

[1] Chia-Lin Yang. (2013). *SPIM tutorial* [Online] http://eclab.csie.ntu.edu.tw/courses/ca2013/

[2] David A. Patterson and John L. Hennessy, *Computer Organization and Design, Fourth Edition: The Hardware/Software Interface*, 4th ed. Morgan Kaufmann Publishers, 2008

[3] http://en.wikipedia.org/wiki/SPIM

[4] http://pages.cs.wisc.edu/~larus/spim.html