

# Introduction to SPIM

Computer Architecture 2017

2017/9/27

# Outline

- Introduction
- General Layout, MIPS Instruction and SPIM I/O
- Programming Example
- Homework

# Introduction to SPIM Simulator

- **Spim** is a self-contained simulator that runs **MIPS32** programs
- Developed by **James R. Larus**, Computer Science Department, University of Wisconsin-Madison
- **It only runs assembly code** but not executable binary program
- Homepage
  - <http://spimsimulator.sourceforge.net/>
  - [http://spimsimulator.sourceforge.net/HP\\_AppA.pdf](http://spimsimulator.sourceforge.net/HP_AppA.pdf)







# Install QtSpim

- Download from this webpage
  - <http://sourceforge.net/projects/spimsimulator/files/>

Looking for the latest version? [Download QtSpim\\_9.1.19\\_Windows.msi \(32.3 MB\)](#)

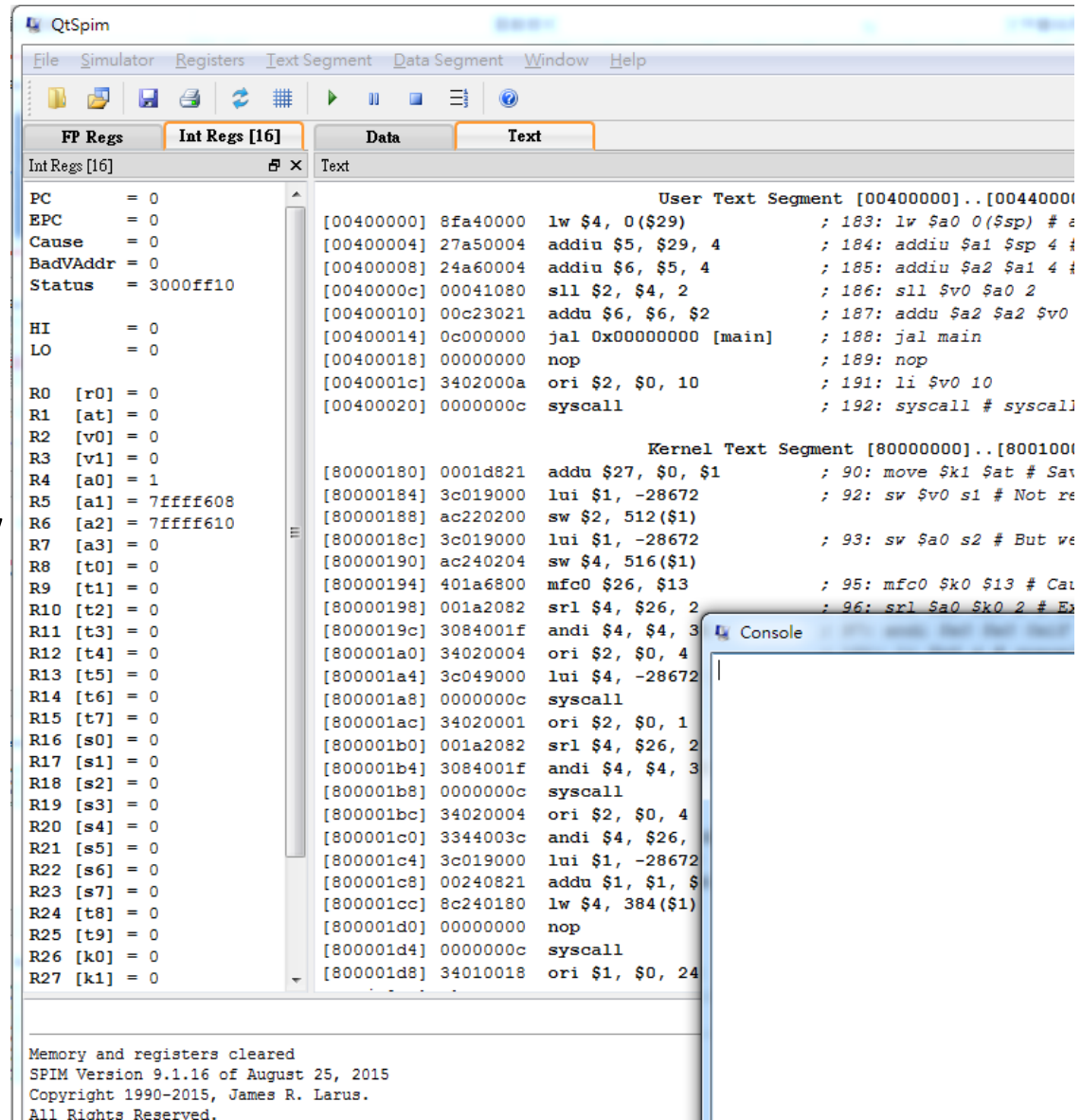
Home



Name ▾	Modified ▾	Size ▾	Downloads / Week ▾
<a href="#">qtspim_9.1.20_linux64.deb</a>	<a href="#">2017-08-29</a>	19.8 MB	247  
<a href="#">QtSpim_9.1.20_mac.mpkg.zip</a>	<a href="#">2017-08-29</a>	12.4 MB	526  
<a href="#">QtSpim_9.1.20_Windows.msi</a>	<a href="#">2017-08-29</a>	13.8 MB	1,048  

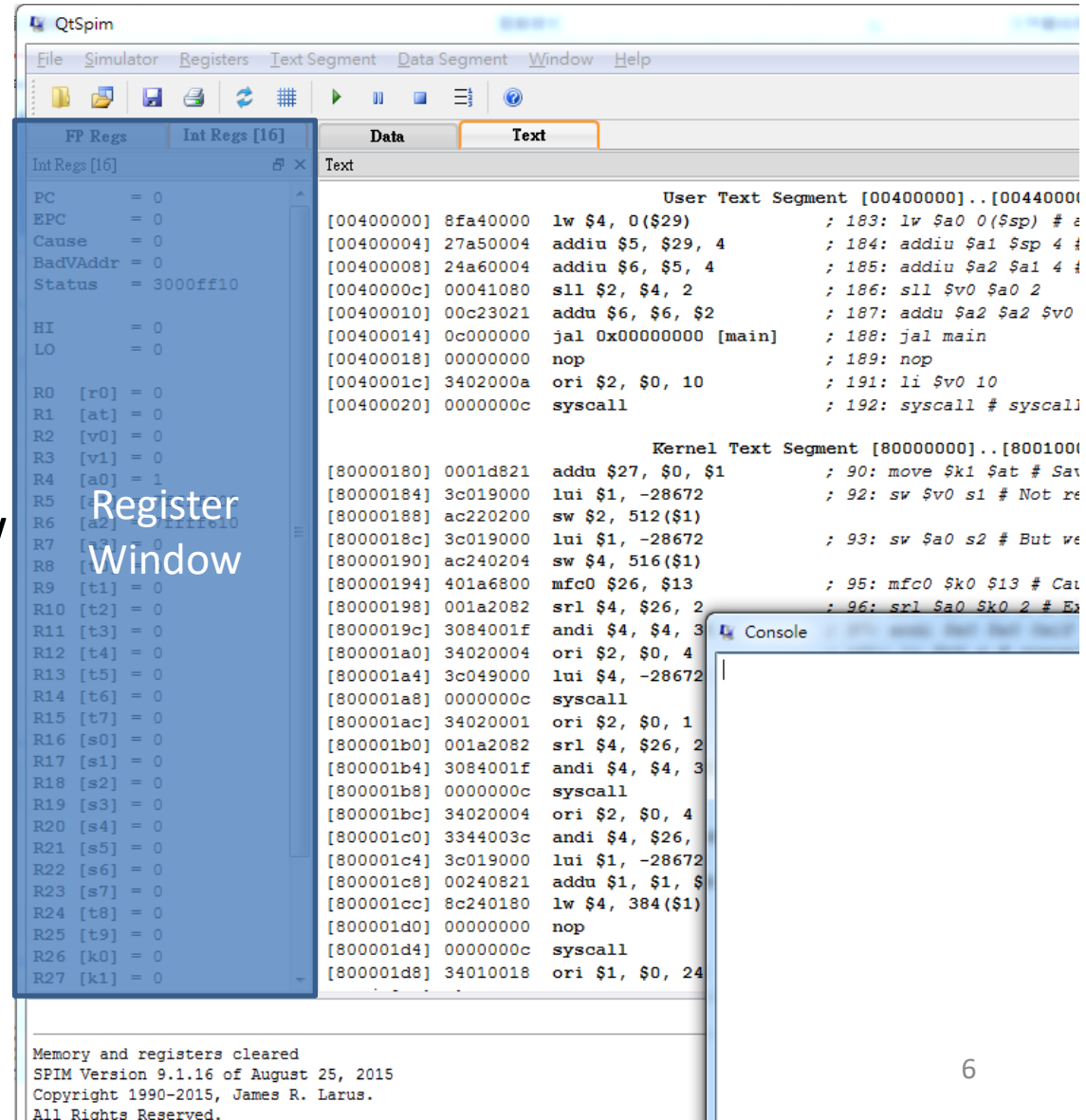
# QtSpim Window

- Register Window
  - shows the values of all registers in the MIPS CPU and FPU
- Text Segment Window
  - shows instructions
- Data Segment Window
  - shows the data loaded into the program's memory and the data of the program's stack
- Message Window
- Console Window



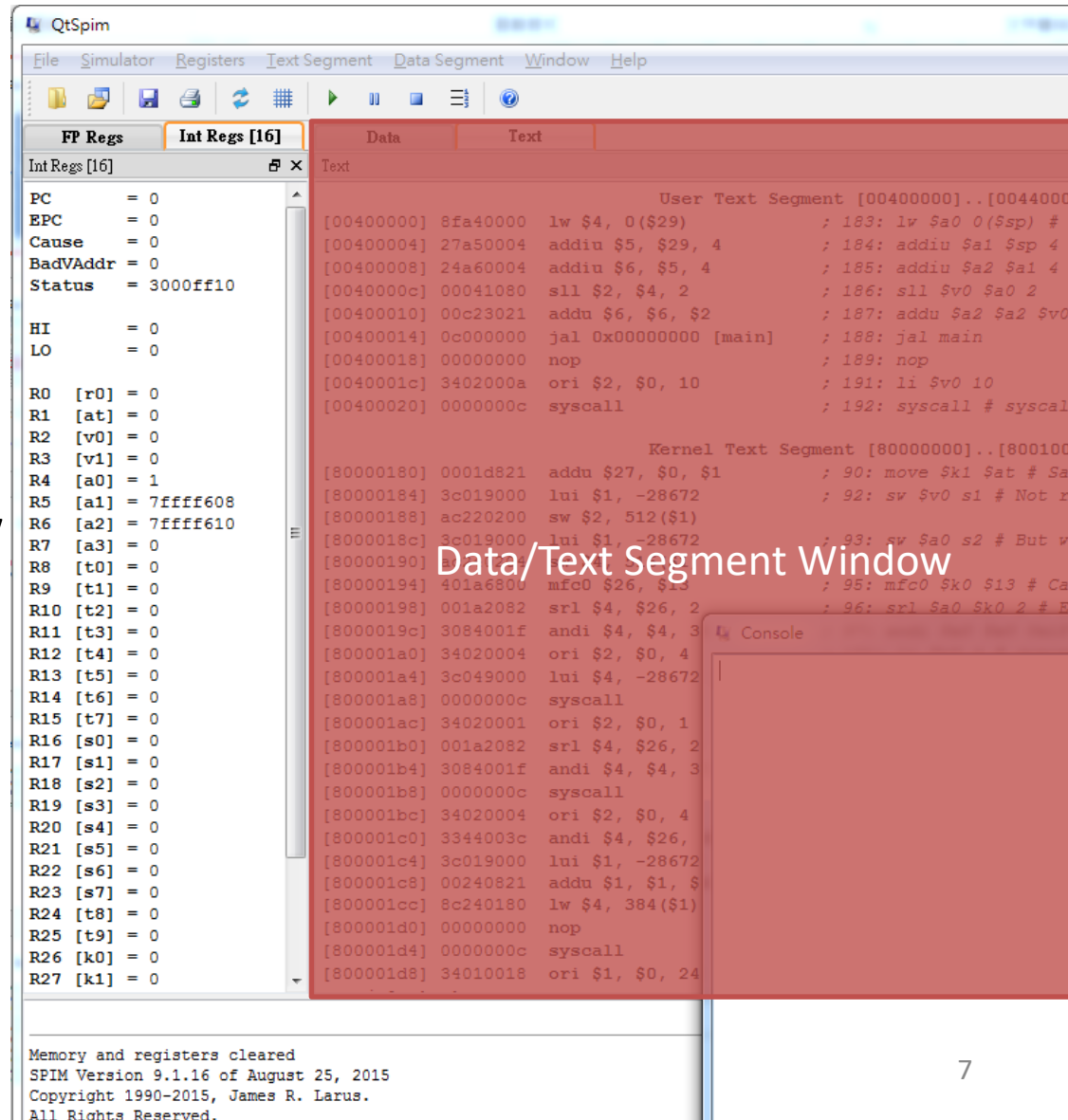
# QtSpim Window

- Register Window
  - shows the values of all registers in the MIPS CPU and FPU
- Text Segment Window
  - shows instructions
- Data Segment Window
  - shows the data loaded into the program's memory and the data of the program's stack
- Message Window
- Console Window



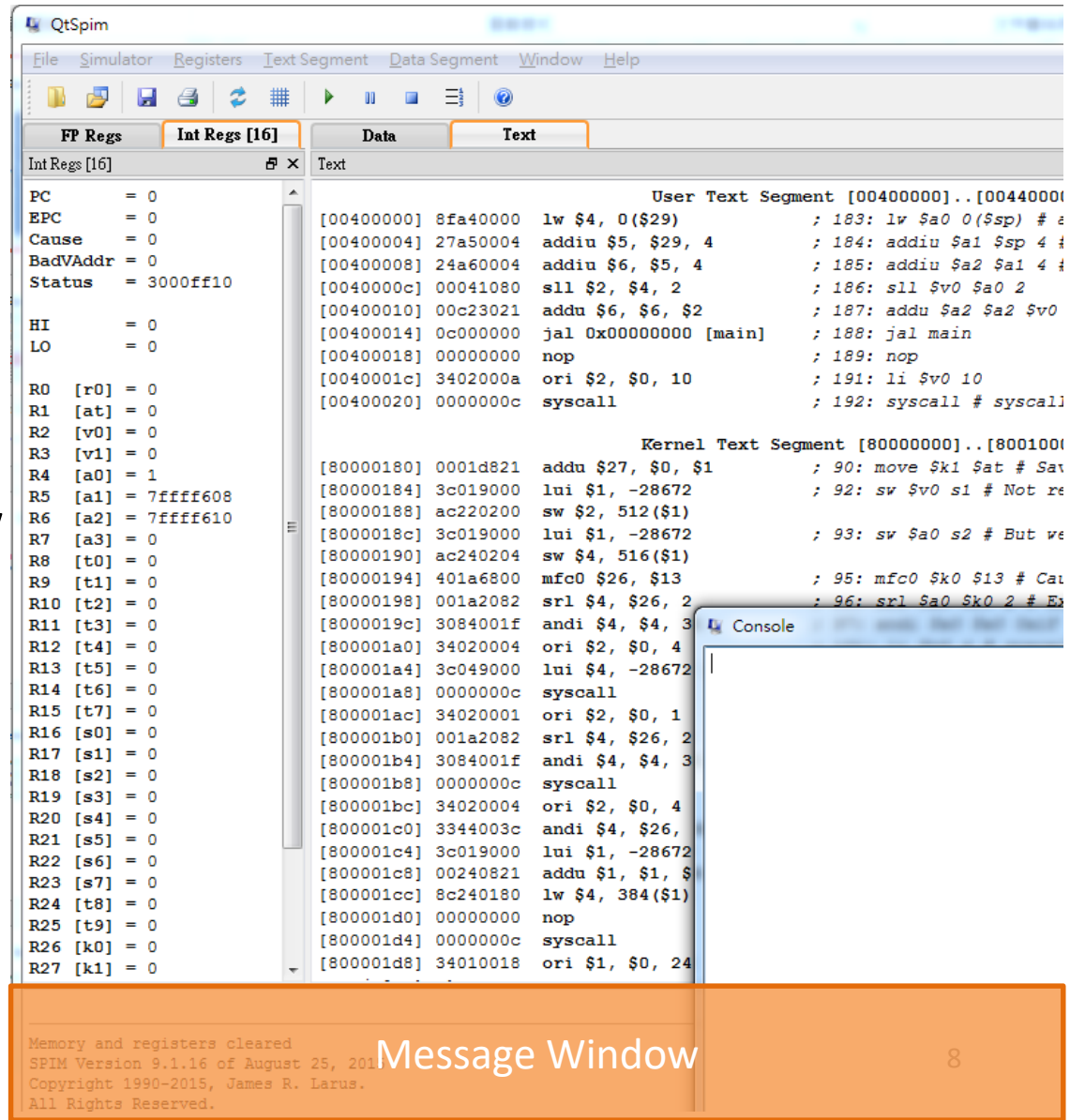
# QtSpim Window

- Register Window
  - shows the values of all registers in the MIPS CPU and FPU
- Text Segment Window
  - shows instructions
- Data Segment Window
  - shows the data loaded into the program's memory and the data of the program's stack
- Message Window
- Console Window



# QtSpim Window

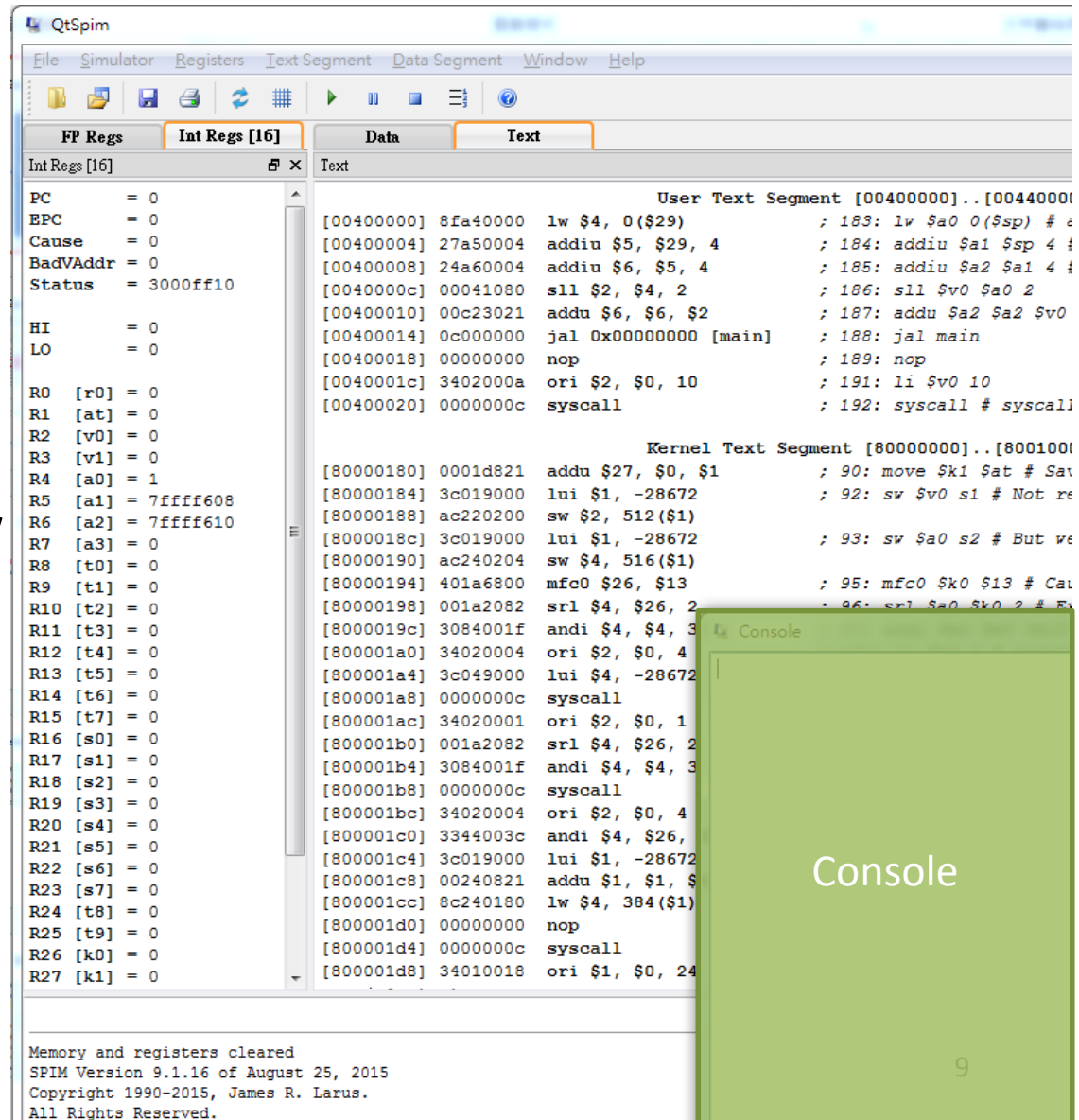
- Register Window
  - shows the values of all registers in the MIPS CPU and FPU
- Text Segment Window
  - shows instructions
- Data Segment Window
  - shows the data loaded into the program's memory and the data of the program's stack
- Message Window
- Console Window



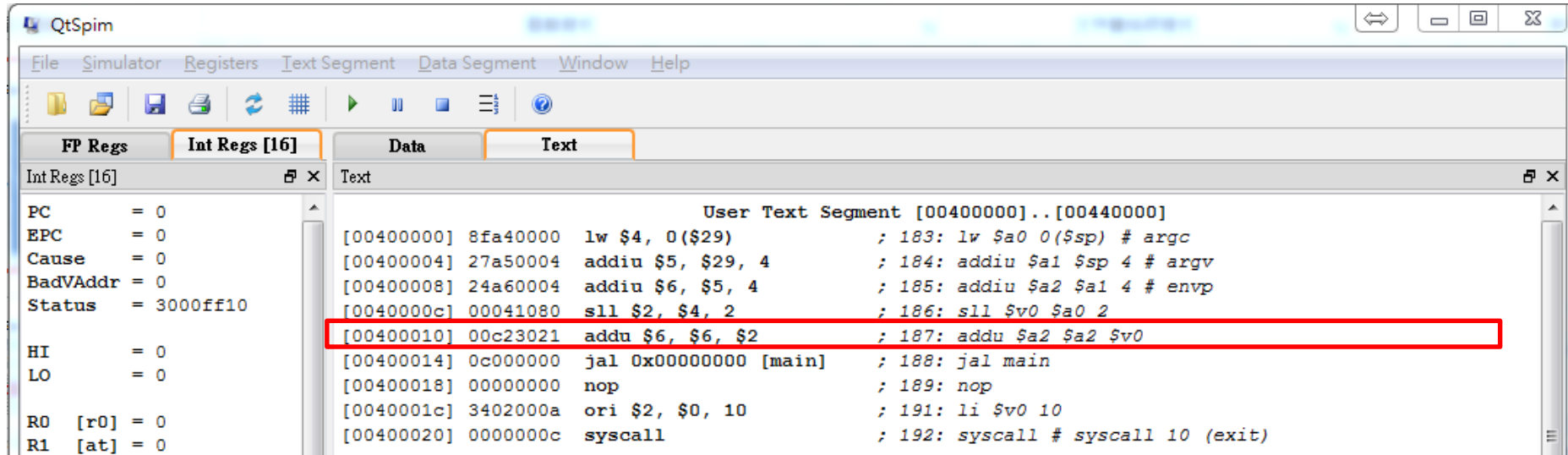


# QtSpim Window

- Register Window
  - shows the values of all registers in the MIPS CPU and FPU
- Text Segment Window
  - shows instructions
- Data Segment Window
  - shows the data loaded into the program's memory and the data of the program's stack
- Message Window
- Console Window



# QtSpim Window



[00400010] 00c23021 addu \$6, \$6, \$2 ; 187: addu \$a2 \$a3 \$v0

memory address of instruction

Instruction's mnemonic description

Source code in assembly file

Instruction's numerical encoding

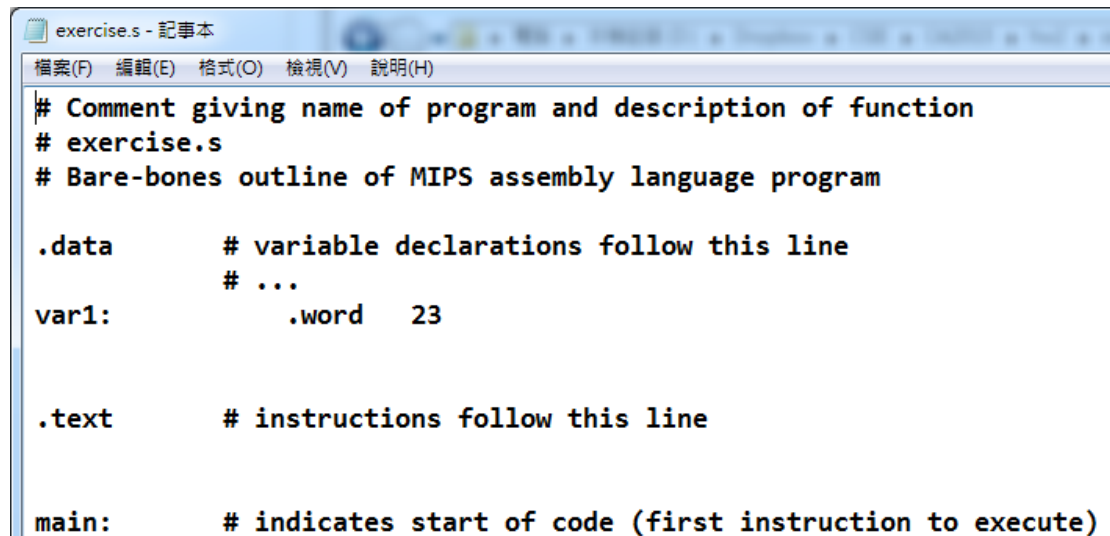
Line number in assembly file

# Outline

- Introduction
- Program structure, MIPS Instructions and SPIM I/O
- Programming Example
- Homework

# Program Structure

- Plain text file with **data declarations**, **program code** (name of file should end in suffix `.s` to be used with SPIM simulator)
- **Data declarations** start with **`.data`** directive
  - Allocated in memory (DRAM)
  - Variables used in program
- **Program code** starts with **`.text`** directive
  - Starting point (**main**)
- **Comments**
  - **`# anything you want`**



```
exercise.s - 記事本
檔案(F) 編輯(E) 格式(O) 檢視(V) 說明(H)
# Comment giving name of program and description of function
# exercise.s
# Bare-bones outline of MIPS assembly language program

.data          # variable declarations follow this line
               # ...
var1:          .word    23

.text          # instructions follow this line

main:          # indicates start of code (first instruction to execute)
```

# Data declarations

- **.word, .half** - 32/16 bit integer
- **.byte** - 8 bit integer (similar to 'char' type in C)
- **.ascii, .asciiz** - string (asciiz is null terminated)
  - Strings are enclosed in double-quotes("")
  - Special characters in strings follow the C convention
  - newline(\n), tab(\t), quote(\")
- **.double, .float** - floating point
- **Format**
  - name:      storage\_type      value(s)
    - Create storage for variable of specified type with given name and specified value
    - Value(s) usually gives initial value(s); for storage type **.space**, gives number of spaces to be allocated (bytes)
    - For example,    `var1:                    .word    23`

# MIPS Instructions (**Load / Store Instructions**)

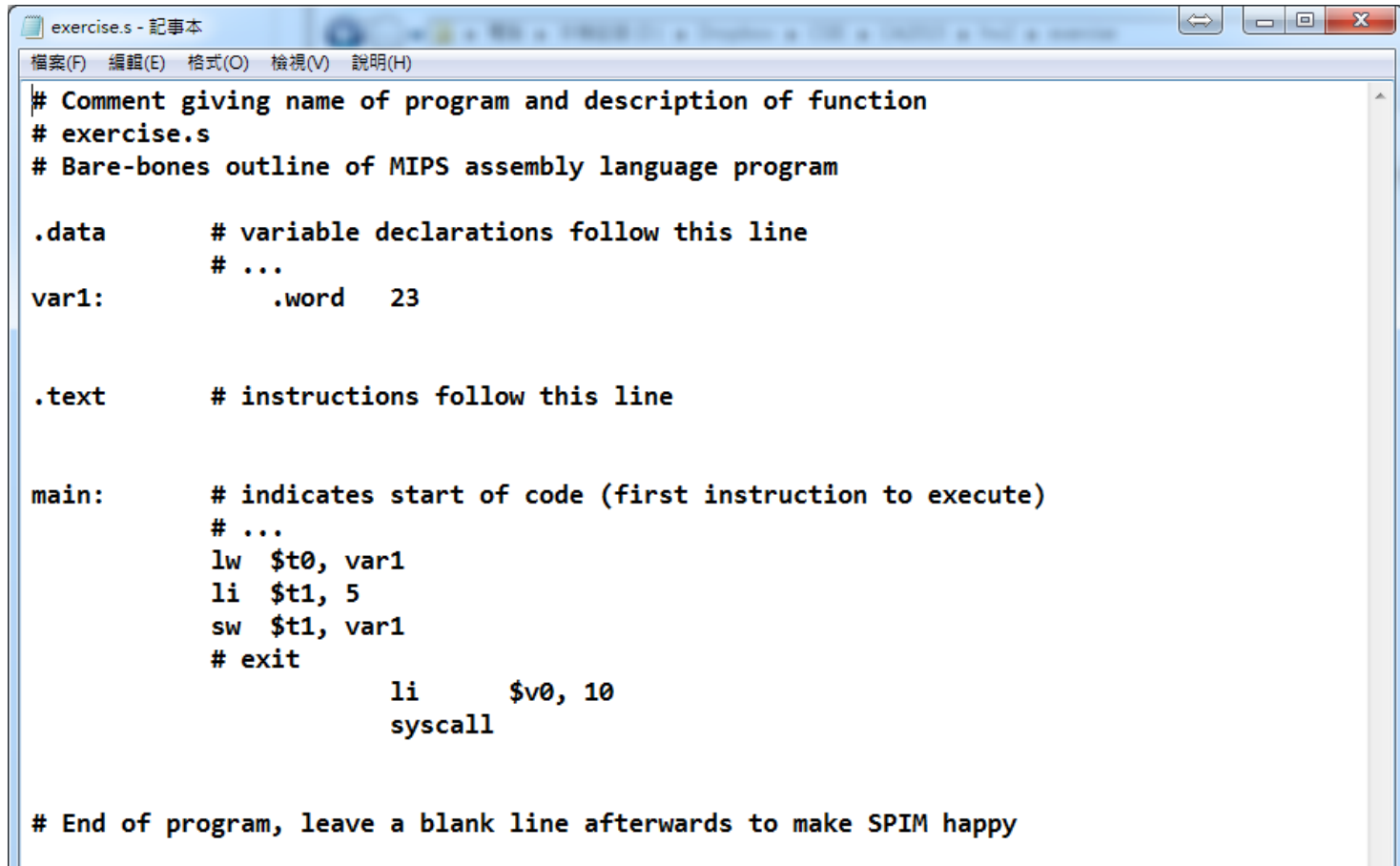
- RAM access only allowed with load and store instructions
  - All other instructions use register operands
- **Load**
  - **lw** **register\_destination, RAM\_source**
    - Copy **word** (4 bytes) at source RAM location to destination register
  - **lb** **register\_destination, RAM\_source**
    - Copy **byte** at source RAM location to low-order byte of destination register, and sign-e.g.tend to higher-order bytes

# MIPS Instructions (**Load / Store Instructions**)

- RAM access only allowed with load and store instructions
  - All other instructions use register operands
- **Store**
  - **sw** **register\_source, RAM\_destination**
    - Store **word** in source register into RAM destination
  - **sb** **register\_source, RAM\_destination**
    - Store **byte** (low-order) in source register into RAM destination
- **load immediate**
  - **li** **register\_destination, value**
    - load **immediate value** into destination register

# MIPS Instructions (Load / Store Instructions)

- Example



```
exercise.s - 記事本
檔案(F) 編輯(E) 格式(O) 檢視(V) 說明(H)

# Comment giving name of program and description of function
# exercise.s
# Bare-bones outline of MIPS assembly language program

.data          # variable declarations follow this line
               # ...
var1:          .word    23

.text          # instructions follow this line

main:          # indicates start of code (first instruction to execute)
               # ...
               lw  $t0, var1
               li  $t1, 5
               sw  $t1, var1
               # exit
               li   $v0, 10
               syscall

# End of program, leave a blank line afterwards to make SPIM happy
```



# MIPS Instructions (Load / Store Instructions)

- Example

```
exercise.s - 記事本
檔案(F) 編輯(E) 格式(O) 檢視(V) 說明(H)
# Comment giving name of program and description of function
# exercise.s
# Bare-bones outline of MIPS assembly language program

.data      # variable declarations follow this line
# ...
var1:      .word   23

.text      # instructions follow this line

main:      # indicates start of code (first instruction to execute)
# ...
lw  $t0, var1
li  $t1, 5
sw  $t1, var1
# exit
      li    $v0, 10
      syscall

# End of program, leave a blank line afterwards to make SPIM happy
```

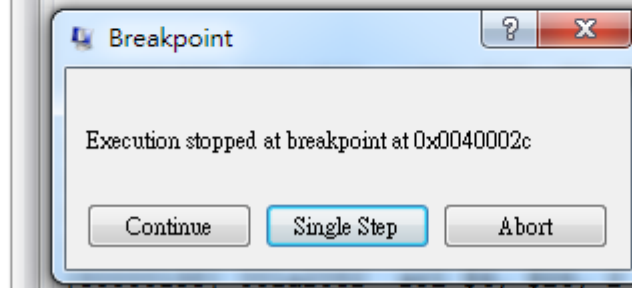
Data	Text
Data	
User data segment [10000000]..[10040000]	
[10000000]..[1000ffff]	00000000
[10010000]	00000017 00000000 00000000 00000000
[10010010]..[1003ffff]	00000000

# MIPS Instructions (Load / Store Instructions)

- Example
  - lw \$t0, var1

```
R0 [r0] = 0
R1 [at] = 10010000
R2 [v0] = 4
R3 [v1] = 0
R4 [a0] = 1
R5 [a1] = 7ffff618
R6 [a2] = 7ffff620
R7 [a3] = 0
R8 [t0] = 17
R9 [t1] = 0
R10 [t2] = 0
R11 [t3] = 0
R12 [t4] = 0
R13 [t5] = 0
R14 [t6] = 0
R15 [t7] = 0
R16 [s0] = 0
R17 [s1] = 0
```

```
[00400010] 3402000a ori $2, $0, 10 ; 191: li $v0, 10
[00400020] 0000000c syscall ; 192: syscall # syscall 10 (exit)
[00400024] 3c011001 lui $1, 4097 ; 14: lw $t0, var1
[00400028] 8c280000 lw $8, 0($1)
[0040002c] 34090005 ori $9, $0, 5 ; 15: li $t1, 5
[00400030] 3c011001 lui $1, 4097 ; 16: sw $t1, var1
[00400034] ac290000 sw $9, 0($1)
[00400038] 3402000a ori $2, $0, 10 ; 18: li $v0, 10
[0040003c] 0000000c syscall ; 19: syscall
```



```
Kernel Text Segment [80000000]..[80010000]
; 90: move $k1 $at # Save $at
; 92: sw $v0 $1 # Not re-entrant and
; 93: sw $a0 $2 # But we need to use
; 95: mfc0 $k0 $13 # Cause register
; 96: srl $a0 $k0 2 # Extract ExcCo
```

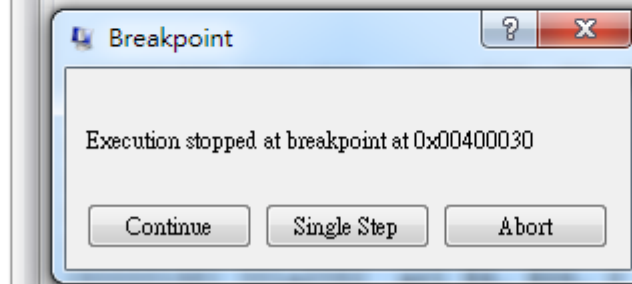
# MIPS Instructions (Load / Store Instructions)

- Example

– li \$t1, 5

R0 [r0] = 0  
R1 [at] = 10010000  
R2 [v0] = 4  
R3 [v1] = 0  
R4 [a0] = 1  
R5 [a1] = 7ffff618  
R6 [a2] = 7ffff620  
R7 [a3] = 0  
R8 [t0] = 17  
**R9 [t1] = 5**  
R10 [t2] = 0  
R11 [t3] = 0  
R12 [t4] = 0  
R13 [t5] = 0  
R14 [t6] = 0  
R15 [t7] = 0  
R16 [s0] = 0  
R17 [s1] = 0

```
[00400010] 3f02000a ori $2, $0, 10 ; 151: li $v0, 10  
[00400020] 0000000c syscall ; 192: syscall # syscall 10 (exit)  
[00400024] 3c011001 lui $1, 4097 ; 14: lw $t0, var1  
[00400028] 8c280000 lw $8, 0($1)  
[0040002c] 34090005 ori $9, $0, 5 ; 15: li $t1, 5  
[00400030] 3c011001 lui $1, 4097 ; 16: sw $t1, var1  
[00400034] ac290000 sw $9, 0($1)  
[00400038] 3402000a ori $2, $0, 10 ; 18: li $v0, 10  
[0040003c] 0000000c syscall ; 19: syscall
```



Kernel Text Segment [80000000]..[80010000]  
; 90: move \$k1 \$at # Save \$at  
; 92: sw \$v0 \$1 # Not re-entrant and  
; 93: sw \$a0 \$2 # But we need to use  
; 95: mfc0 \$k0 \$13 # Cause register  
; 96: srl \$a0 \$k0 2 # Extract ExcCode

# MIPS Instructions (Load / Store Instructions)

- Example
  - **sw**            **\$t1, var1**

```
[00400024] 3c011001 lui $1, 4097           ; 14: lw $t0, var1
[00400028] 8c280000 lw $8, 0($1)
[0040002c] 8f8a8000 lw $10, -32768($28)      ; 15: lw $t2, -0x8000($gp)
[00400030] 34090005 ori $9, $0, 5           ; 16: li $t1, 5
[00400034] 3c011001 lui $1, 4097           ; 17: sw $t1, var1
[00400038] ac290000 sw $9, 0($1)
[0040003c] 3402000a ori $2, $0, 10          ; 19: li $v0, 10
[00400040] 0000000c syscall                ; 20: syscall
```

Data	Text		
Data			
User data segment [10000000]..[10040000]			
[10000000]..[1000ffff]	00000000		
[10010000]	00000005	00000000	00000000 00000000
[10010010]..[1003ffff]	00000000		

# MIPS Instructions (Indirect and Based Addressing)

- **Load address**

- **la**            **\$t0, var1**

- Copy RAM address of var1 (presumably a label defined in the program) into register \$t0

- **Indirect addressing**

- **lw**            **\$t2, (\$t0)**

- load word at RAM address contained in \$t0 into \$t2

- **sw**            **\$t2, (\$t0)**

- store word in register \$t2 into RAM at address contained in \$t0

# MIPS Instructions (Indirect and Based Addressing)

- **Based or indexed addressing:**
  - **lw            \$t2, 4(\$t0)**
    - load word at RAM address (\$t0+4) into register \$t2
    - "4" gives offset from address in register \$t0
  - **sw            \$t2, -12(\$t0)**
    - store word in register \$t2 into RAM at address (\$t0 - 12)
    - negative offsets are fine

# MIPS Instructions (Indirect and Based Addressing)

```
.data          # variable declarations follow this line
               # ...
array1: .space    10

.text          # instructions follow this line

main:         # indicates start of code (first instruction to execute)
               # ...
    la      $t0, array1
    li      $t2, 10
    li      $t1, 1

loop:
    sb      $t1, ($t0)
    addi    $t0, $t0, 1
    addi    $t1, $t1, 1
    ble     $t1, $t2, loop
    # exit

exit:
    li      $v0, 10
    syscall
```

Data	Text
Data	
User data segment [10000000]..[10040000]	
[10000000]..[1000ffff]	00000000
[10010000]	04030201 08070605 00000a09 00000000 . . . . .
[10010010]..[1003ffff]	00000000

**Note:** Based addressing is especially useful for:

- Arrays
  - Access elements as offset from base address
- Stacks
  - Easy to access elements at offset from stack pointer or frame pointer

# MIPS Instructions (Arithmetic Instructions)

- Operand size is **word** (4 bytes)

<b>add \$t0,\$t1,\$t2</b>	<b>\$t0 = \$t1 + \$t2; add as signed (2's complement) integers</b>
<b>sub \$t2,\$t3,\$t4</b>	<b>\$t2 = \$t3 - \$t4</b>
<b>addi \$t2,\$t3, 5</b>	<b>\$t2 = \$t3 + 5; "add immediate" (no sub immediate)</b>
<b>addu \$t1,\$t6,\$t7</b>	<b>\$t1 = \$t6 + \$t7; add as unsigned integers</b>
<b>subu \$t1,\$t6,\$t7</b>	<b>\$t1 = \$t6 + \$t7; subtract as unsigned integers</b>
<b>mult \$t3,\$t4</b>	<b>multiply 32-bit quantities in \$t3 and \$t4, and store 64-bit result in special registers Lo and Hi: (Hi,Lo) = \$t3 * \$t4</b>
<b>div \$t5,\$t6</b>	<b>Lo = \$t5 / \$t6 (integer quotient) Hi = \$t5 mod \$t6 (remainder)</b>
<b>mfhi \$t0</b>	<b>move quantity in special register Hi to \$t0: \$t0 = Hi</b>
<b>mflo \$t1</b>	<b>move quantity in special register Lo to \$t1: \$t1 = Lo used to get at result of product or quotient</b>
<b>move \$t2,\$t3</b>	<b>\$t2 = \$t3</b>



# MIPS Instructions (Control Structures)

- **Branches**

<b>beq \$t0,\$t1,target</b>	<b>branch to target if \$t0 = \$t1</b>
<b>blt \$t0,\$t1,target</b>	<b>branch to target if \$t0 &lt; \$t1</b>
<b>ble \$t0,\$t1,target</b>	<b>branch to target if \$t0 &lt;= \$t1</b>
<b>bgt \$t0,\$t1,target</b>	<b>branch to target if \$t0 &gt; \$t1</b>
<b>bge \$t0,\$t1,target</b>	<b>branch to target if \$t0 &gt;= \$t1</b>
<b>bne \$t0,\$t1,target</b>	<b>branch to target if \$t0 &lt;&gt; \$t1</b>

- **Jumps**

<b>j target</b>	<b>unconditional jump to program label target</b>
<b>jr \$t3</b>	<b>jump to address contained in \$t3 ("jump register")</b>

# MIPS Instructions (Control Structures)

- **Control flow in MIPS**
  - Subroutine/function Calls
  - A, B & C functions

1. Someone calls A
2. A calls B
3. B calls C
4. C returns to B
5. B returns to A
6. A returns

# Control flow in C

- Invoking a function changes the control flow of a program **twice**.
  - **Calling** the function
  - **Returning** from the function
- In this example the main function calls fact twice, and fact returns twice—but to different locations in main.
- Each time fact is called, the CPU has to remember the appropriate return address.

```
int main()
{
    ...
    t1= fact(8);
    t2= fact(3);
    t3= t1+t2;
    ...
}

int fact(int a0)
{
    int t1, v0 = 1;
    for(t1 = a0; t1 > 1; t1--)
        v0 = v0 * t1;
    return v0;
}
```

# Control flow in MIPS

- MIPS uses the jump-and-link instruction **jal** to call functions.
  - The jal saves the return address (the address of the next instruction) in the dedicated register \$ra, before jumping to the function.
  - jal is the only MIPS instruction that can access the value of the program counter, so it can store the return address PC+4 in \$ra.

## jal fact

- To transfer control back to the caller, the function just has to jump to the address that was stored in \$ra.

## jr \$ra

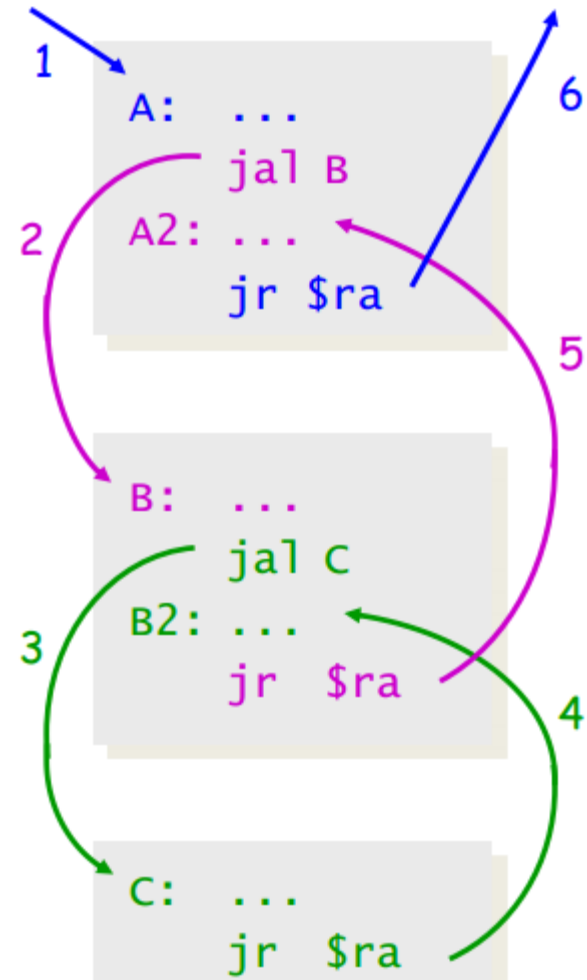
**Note: return address stored in register \$ra; if subroutine will call other subroutines, or is recursive, return address should be copied from \$ra onto stack to preserve it, since jal always places return address in this register and hence will overwrite previous value**

# Function calls and stacks

- Notice function calls and returns occur in a stack-like order: the most recently called function is the first one to return.

1. Someone calls A
2. A calls B
3. B calls C
4. C returns to B
5. B returns to A
6. A returns

- Here, for example, C must return to B before B can return to A.



# Register

Register name	Number	Usage
\$zero	0	constant 0
\$at	1	reserved for assembler
\$v0	2	expression evaluation and results of a function
\$v1	3	expression evaluation and results of a function
\$a0	4	argument 1
\$a1	5	argument 2
\$a2	6	argument 3
\$a3	7	argument 4
\$t0	8	temporary (not preserved across call)
\$t1	9	temporary (not preserved across call)
\$t2	10	temporary (not preserved across call)
\$t3	11	temporary (not preserved across call)
\$t4	12	temporary (not preserved across call)
\$t5	13	temporary (not preserved across call)
\$t6	14	temporary (not preserved across call)
\$t7	15	temporary (not preserved across call)

Results  
(\$v0, \$v1)

Function parameters  
(\$a0, \$a1, \$a2, \$a3)

→ The usage description of these registers are just “convention”. They are physically the same.

# Register

\$s0	16	saved temporary (preserved across call)
\$s1	17	saved temporary (preserved across call)
\$s2	18	saved temporary (preserved across call)
\$s3	19	saved temporary (preserved across call)
\$s4	20	saved temporary (preserved across call)
\$s5	21	saved temporary (preserved across call)
\$s6	22	saved temporary (preserved across call)
\$s7	23	saved temporary (preserved across call)
\$t8	24	temporary (not preserved across call)
\$t9	25	temporary (not preserved across call)
\$k0	26	reserved for OS kernel
\$k1	27	reserved for OS kernel
\$gp	28	pointer to global area
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address (used by function call)

# SPIM I/O

- **SPIM** provides a small set of operating system-like services through the system call instruction.
- 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 \$a0)
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	

```
move $a0, $s1
li $v0, 1
syscall
# print the result to
console
```

```
li $v0, 5
syscall
# read a integer into
$v0
```

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

# Pseudo Instructions

- When machine code is generated, the pseudo instructions are converted to real instructions

`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`

`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`

# Outline

- Introduction
- General Layout, MIPS Instruction and SPIM I/O
- Programming Example
- Homework

# Example (Fibonacci Recurrence)

- Definition

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

- This is easy converse to a C program

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

# Example (Fibonacci Recurrence)

```
.data  
.text  
.globl main
```

```
main:
```

```
li $v0, 5  
syscall  
move $s0, $v0
```

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

```
li $v0, 1  
syscall
```

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

Read integer from user  
and store in register \$v0

Set argument \$a0

Jump to Label fib and store  
next instruction program counter

Print integer result \$a0

Exit program

# 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)
```

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)**.

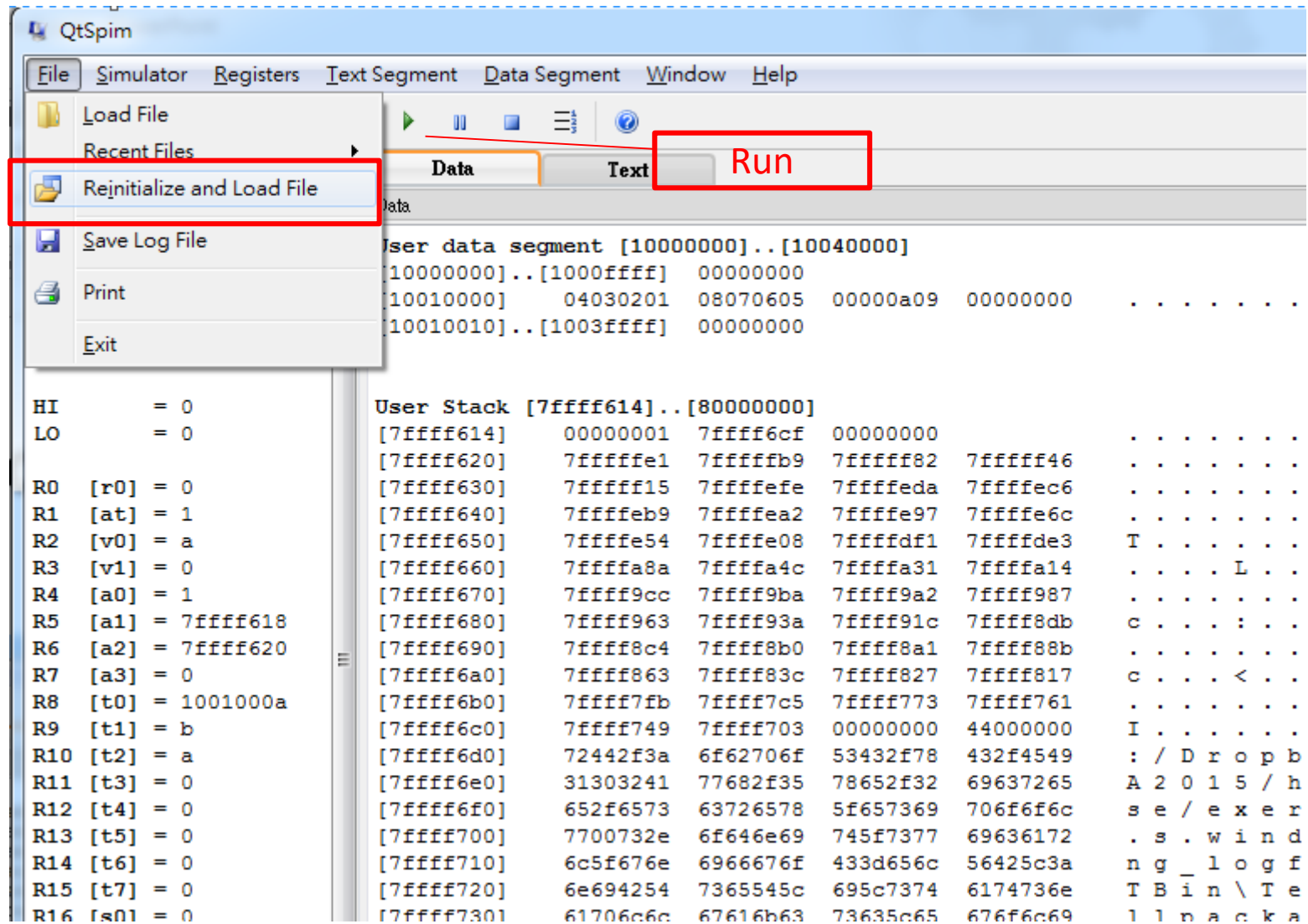
```
lw $v1, 8($sp)  
add $v0, $v0, $v1
```

The results are summed and put in \$v0.

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

Retrieve return address and restore the stack pointer

# Load your program



# Breakpoint

Data	Text
Text	
User Text Segment [00400000]..[00440000]	
[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
[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] 3c081001	lui \$8, 4097 [array1] ; 14: la \$t0, array1
[00400028] 340a000a	ori \$10, \$0, 10 ; 15: li \$t2, 10
[0040002c] 34090001	ori \$9, \$0, 1 ; 16: li \$t1, 1
[00400030] a1	Copy Ctrl+C ; 18: sb \$t1, (\$t0)
[00400034] 21	Select All Ctrl+A ; 19: addi \$t0, \$t0, 1
[00400038] 21	Set Breakpoint ; 20: addi \$t1, \$t1, 1
[0040003c] 01	Clear Breakpoint ; 21: ble \$t1, \$t2, loop
[00400040] 10	[loop-0x00400040]
[00400044] 34	li \$v0, 10 ; 24: li \$v0, 10
[00400048] 0000000c	syscall ; 25: syscall
Kernel Text Segment [80000000]..[80010000]	
[80000180] 0001d821	addu \$27, \$0, \$1 ; 90: move \$k1 \$at # Save \$at
[80000184] 3c019000	lui \$1, -28672 ; 92: sw \$v0 \$1 # Not re-entrant and we
[80000188] ac220200	sw \$2, 512(\$1)
[8000018c] 3c019000	lui \$1, -28672 ; 93: sw \$a0 \$2 # But we need to use the
[80000190] ac240204	sw \$4, 516(\$1)
[80000194] 401a6800	mfc0 \$26, \$13 ; 95: mfc0 \$k0 \$13 # Cause register



# Debugger

- Register Window
- Breakpoint
- System call to console

The screenshot displays the QtSpim MIPS simulator interface. The main window is divided into several panes:

- Register Window (left):** Shows the state of MIPS registers. The PC (Program Counter) is 40004c. The EPC (Exception Program Counter) is also 40004c. The Cause register is 24. The BadVAddr register is 0. The Status register is 3000ff10. The HI and LO registers are 0. The R0-R16 registers are shown with their values. The R17-R27 registers are also shown.
- Text Segment (right):** Displays the assembly code for the User Text Segment. The code starts at address 00400000 and ends at 00440000. The code includes instructions like `lw $4, 0($29)`, `addiu $5, $29, 4`, `addiu $6, $5, 4`, `sll $2, $4, 2`, `addu $6, $6, $2`, `jal 0x00400024 [main]`, `nop`, `ori $2, $0, 10`, `syscall`, `lui $8, 4097 [array1]`, `ori $10, $0, 10`, `ori $9, $0, 1`, `sb $9, 0($8)`, `addi $8, $8, 1`, `addi $9, $9, 1`, `ori $2, $0, 10`, `addu $4, $0, $9`, `ori $2, $0, 1`, `syscall`, `beq $1, $0, -32 [loop-0x00400050]`, `ori $2, $0, 10`, and `syscall`.
- Breakpoint Dialog (bottom center):** A dialog box titled "Breakpoint" with the message "Execution stopped at breakpoint at 0x0040004c". It has buttons for "Continue", "Single Step", and "Abort".
- Console Window (bottom right):** A window titled "Console" showing the output of the program. The output is "2".

The Register Window is highlighted with an orange border. The Breakpoint dialog is highlighted with a red border. The Console window is highlighted with a green border.

# Outline

- Introduction
- General Layout, MIPS Instruction and SPIM I/O
- Programming Example
- Homework

# Homework2

- Simple Calculator
  - Write a MIPS32 assembly program to calculate two integers.
  - Read equation from an input file and output to an output file
  - Support "+", "-", "\*", "/" **integer** operations
  - Output "XXXX" and exit immediately when:
    - Unsupported operator (^, √, ...)
    - Divided by 0
  - You don't need to check if the input number is really an integer. (we won't test "1.1+2.3")

# Homework2

- Simple Calculator

- I/O Formats:

- Input format and an example:

<n1><operator><n2>

02+99

- Input filename "input.txt"
      - $0 \leq n1, n2 < 100, n1, n2 \in \mathbb{Z}$
      - All the number are two-digit
        - » 2(x) 02(o)
    - Output: print the result in a file named "output.txt"

0101

- $0 \leq \text{Output}$
        - Output filename "output.txt"
        - Four-digit positive number or "XXXX"

# Homework2

- Simple Calculator

- Modify from the "sample\_code.s"

- Make sure your program could do the right calculation
      - You should identify whether the operator is "+", "-", "\*" or "/"
    - Make sure your program satisfies the I/O formats
      - You should implement the function of "itoa"
    - Make sure your program read from & dump the result to the correct file before submission
      - "input.txt" && "output.txt"

```
# [TODO] : change the file name/path to access the files
# NOTE : Before you submit the code, make sure these two fields are "input.txt" and "output.txt"
file_in:
    .asciiiz "input.txt"
file_out:
    .asciiiz "output.txt"
```

- Helpful tools in the sample code

- A file reader and writer already exist in the "sample\_code.s"
    - A function that pops outputs (integer) to console to help you debug.

# Homework2

- Submission

- Due: 2016/10/9 (Monday) midnight (23:59:59)

- FTP server will be closed on due.

- FTP:

- IP address: 140.112.31.136

- Port: 21 (default)

- Username: ca

- Password: ca2017\_fall

- Upload your homework to "hw2" directory.

- You should compress the folder in a .zip file

- hw2\_<studentID>[\_v<version>].zip (ex. hw2\_r03922024\_v0.zip) (英文小寫)

- hw2\_<studentID>

- hw2\_<studentID>.s

- readme.txt

"readme.txt":

大概說明一下

1. code 是怎麼實作

2. 編寫的平台(Ex: Windows, Linux or Apple)

主要是批改有問題的時候助教會作為參考

