Introduction to SPIM

Computer Architecture 2018

2018/10/3

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

Install QtSpim

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

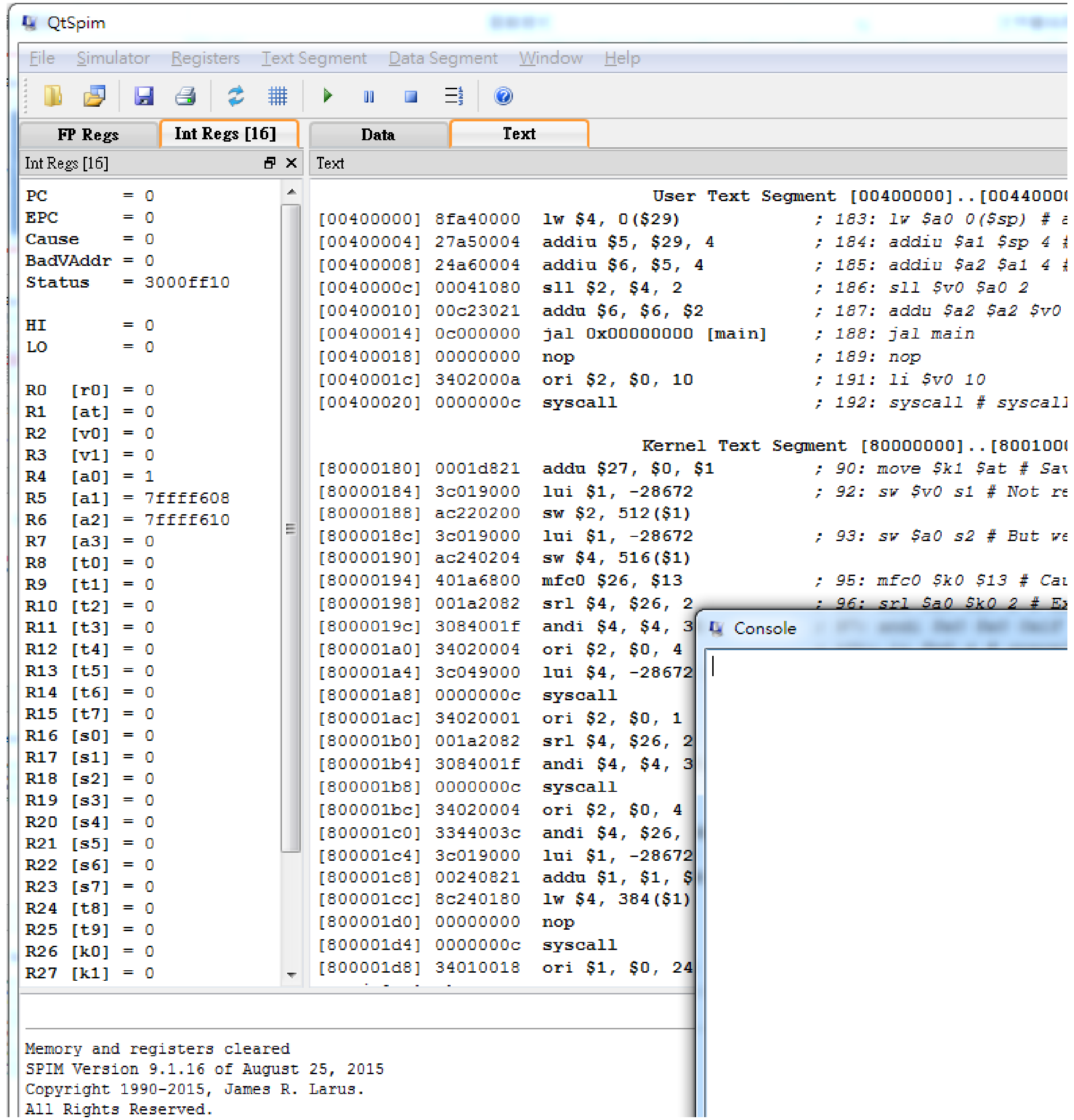
Looking for the latest version? Download QtSpim_9.1.19_Windows.msi (32.3 MB)

Home

Name	Modified	Size	Downloads / Week
qtspim_9.1.20_linux64.deb	2017-08-29	19.8 MB	247
QtSpim_9.1.20_mac.mpkg.zip	2017-08-29	12.4 MB	526
QtSpim_9.1.20_Windows.msi	2017-08-29	13.8 MB	1,048

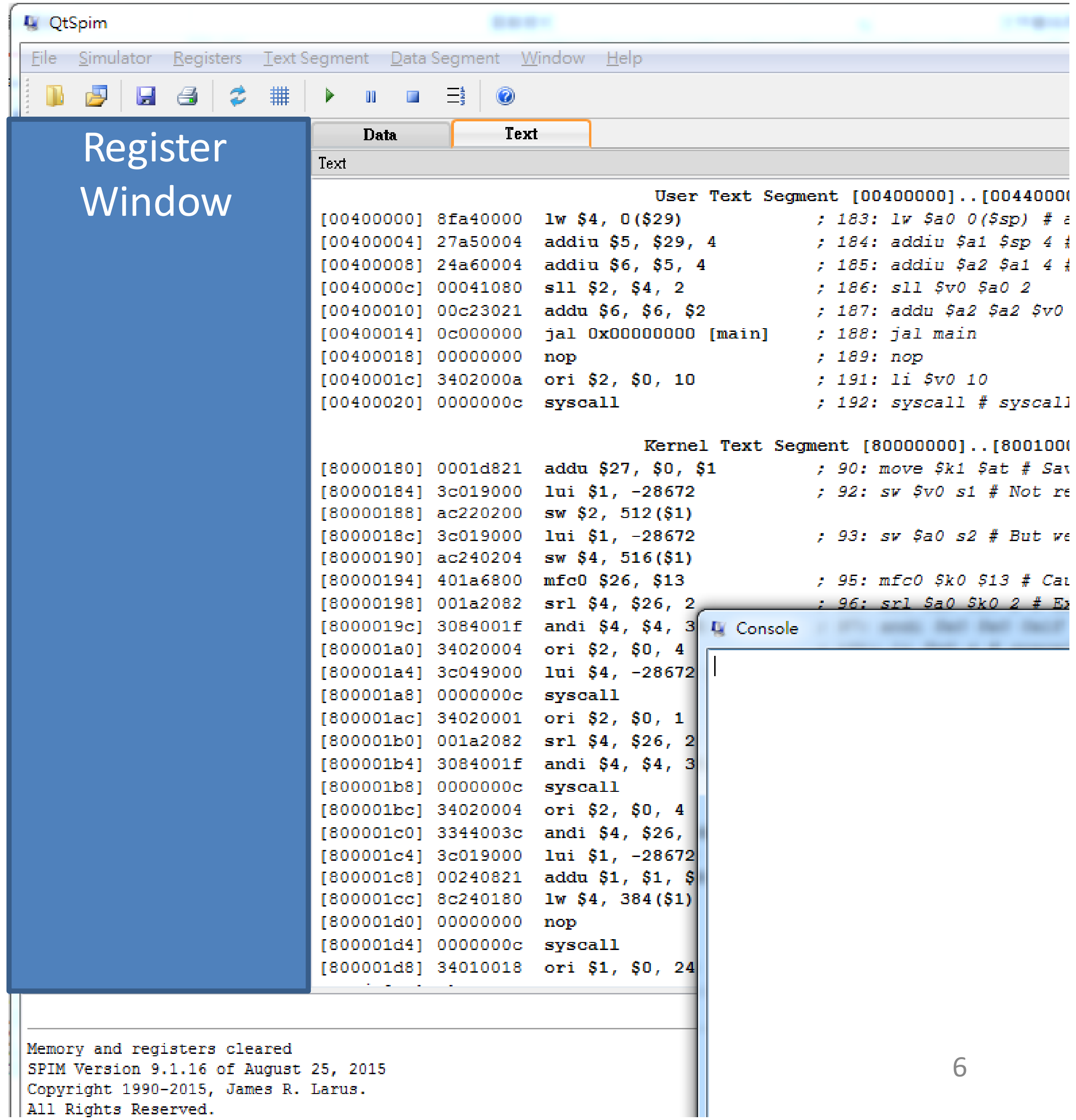
QtSpim Window

- 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



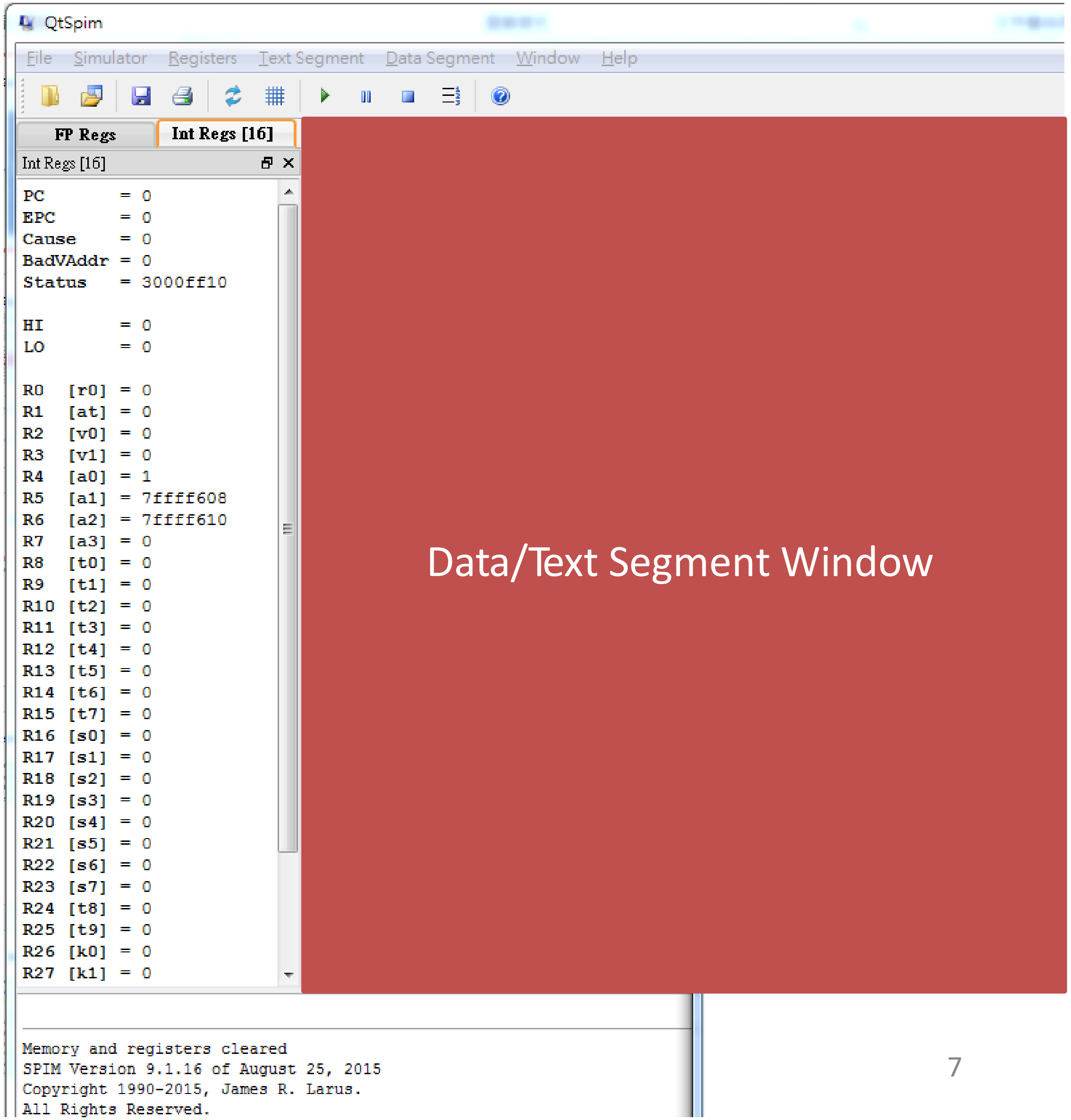
QtSpim Window

- 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



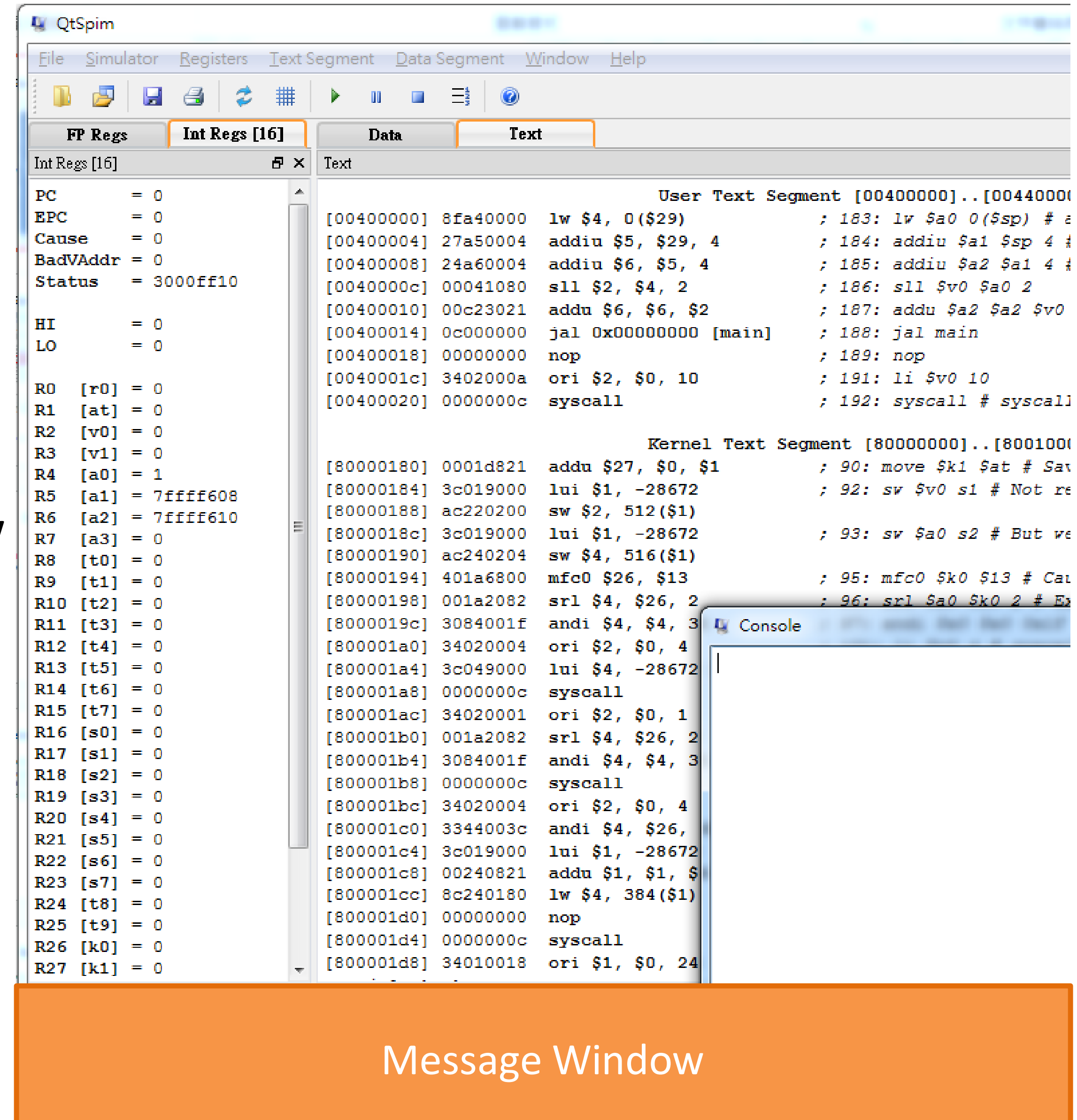
QtSpim Window

- 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



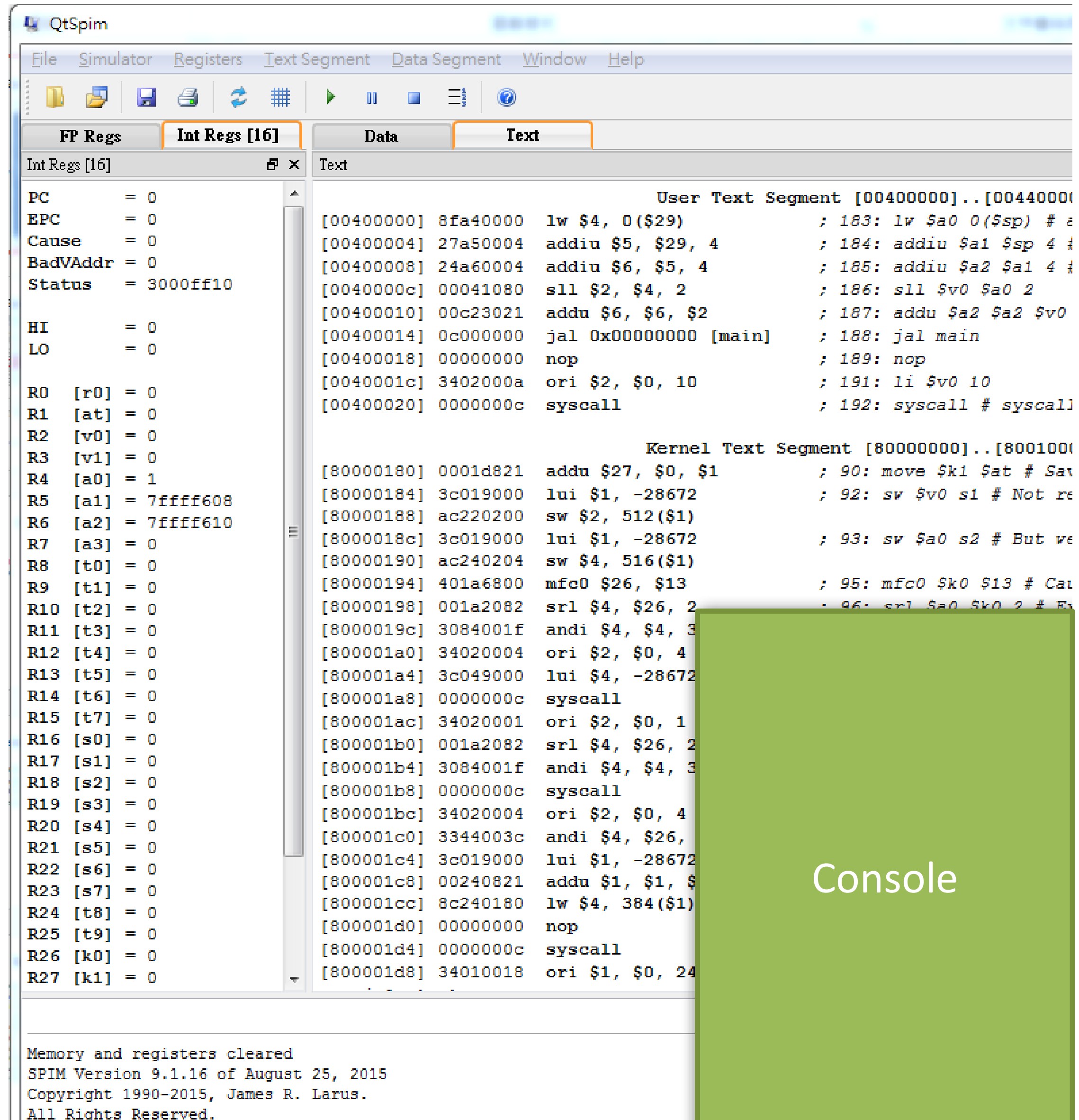
QtSpim Window

- 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

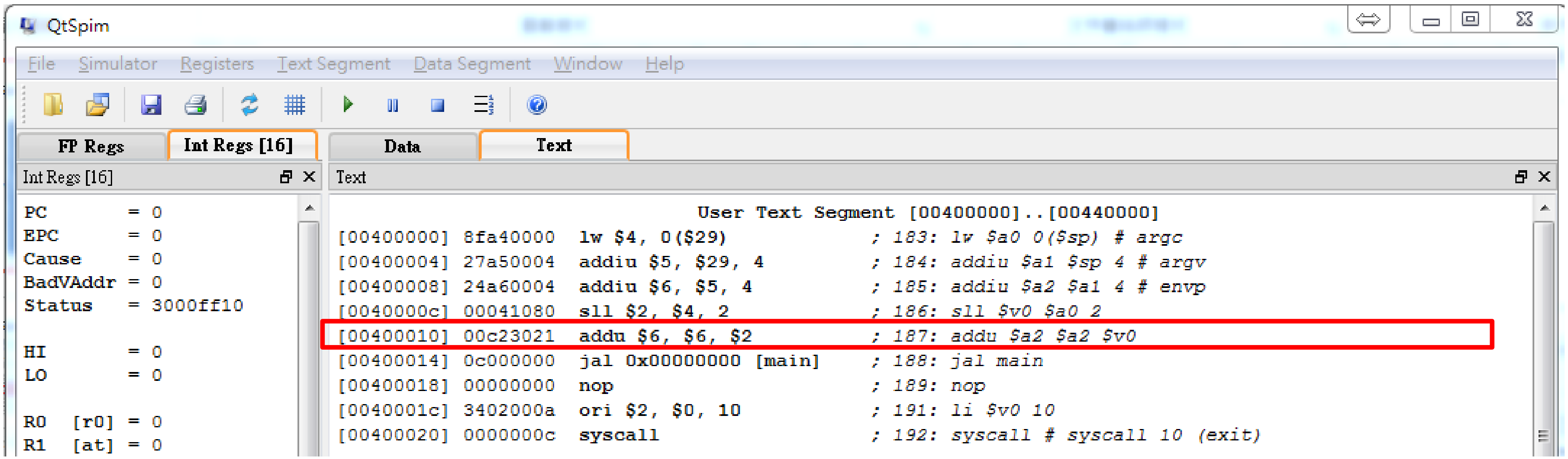


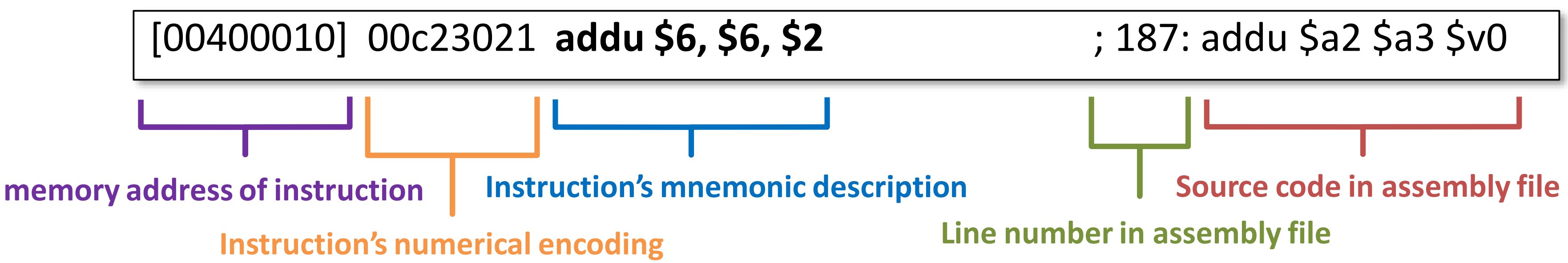
QtSpim Window

- 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



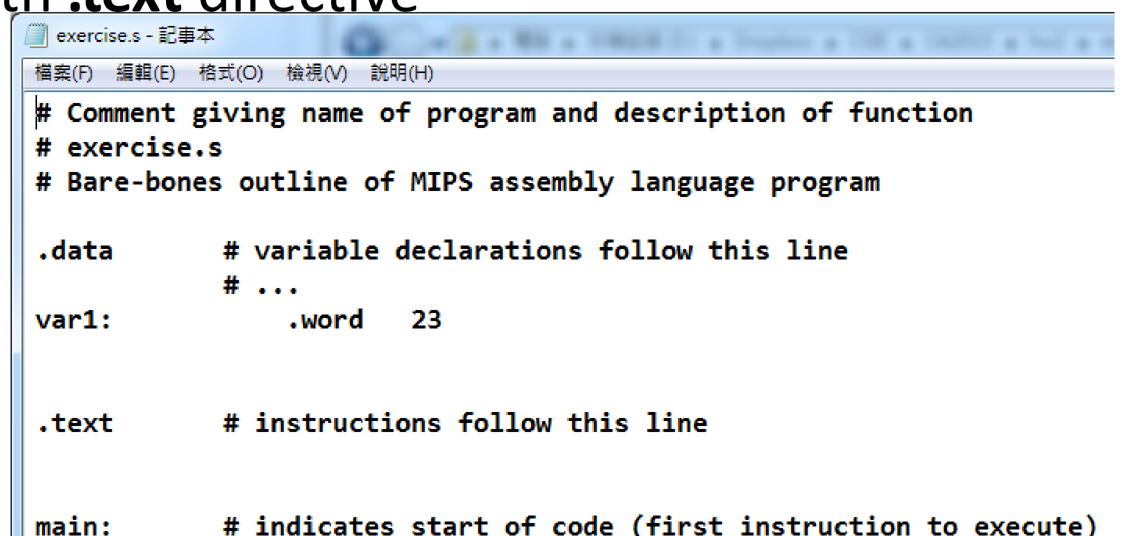


Outline

- 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**




```
# 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)
```

http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm

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-quotas(")
  - 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`

http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm

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

http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm

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

http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm

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

- Example

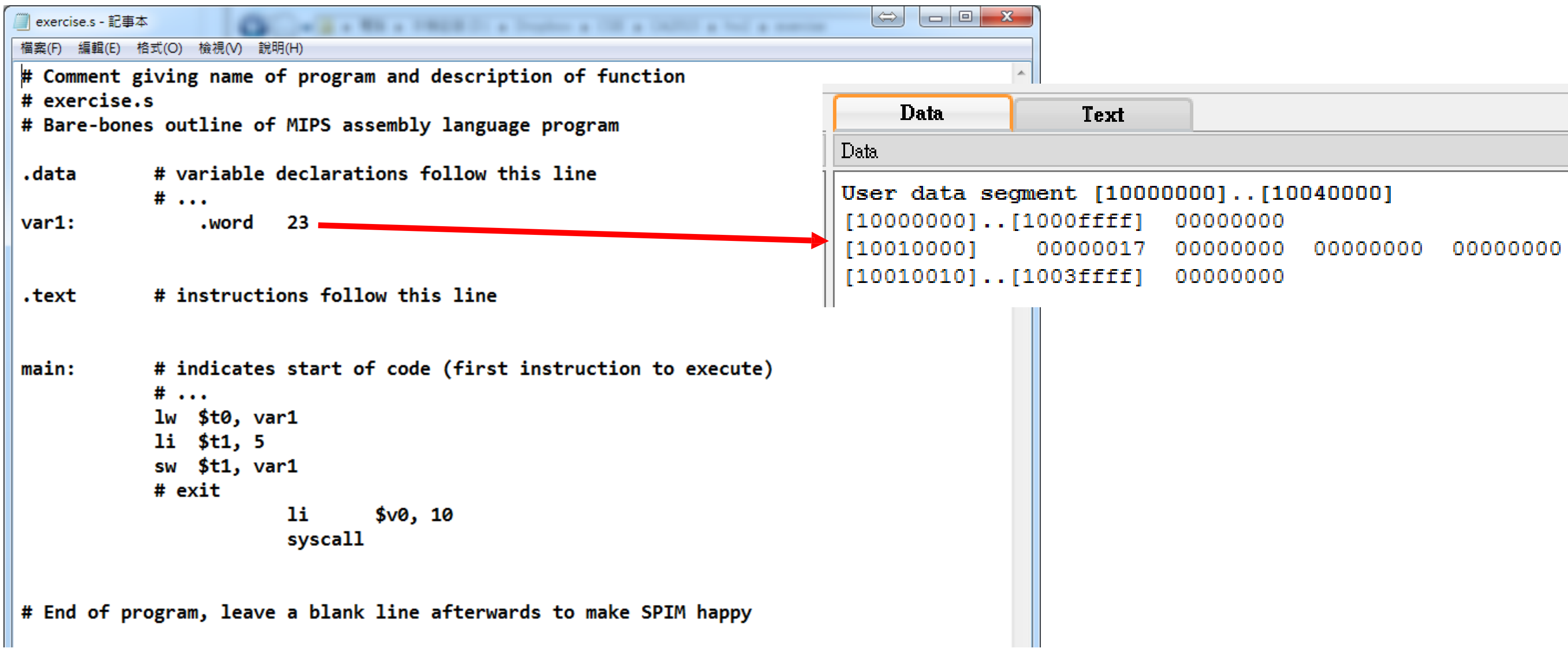
exercise.s - 記事本

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
```

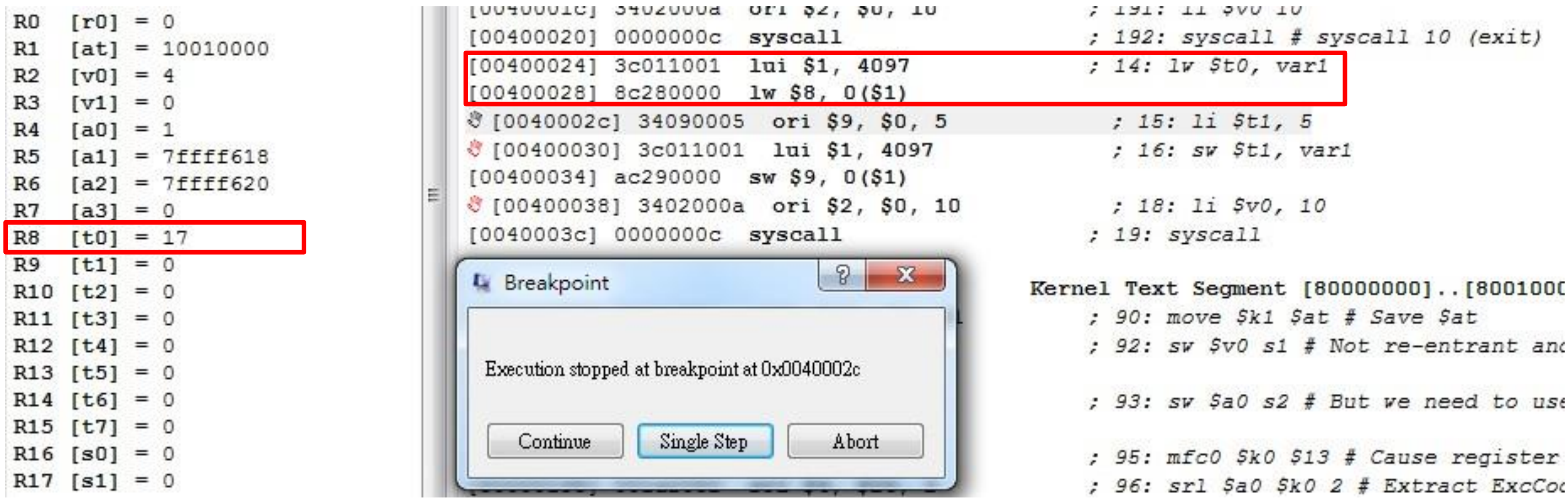
MIPS Instructions (**Load / Store Instructions**)

- Example



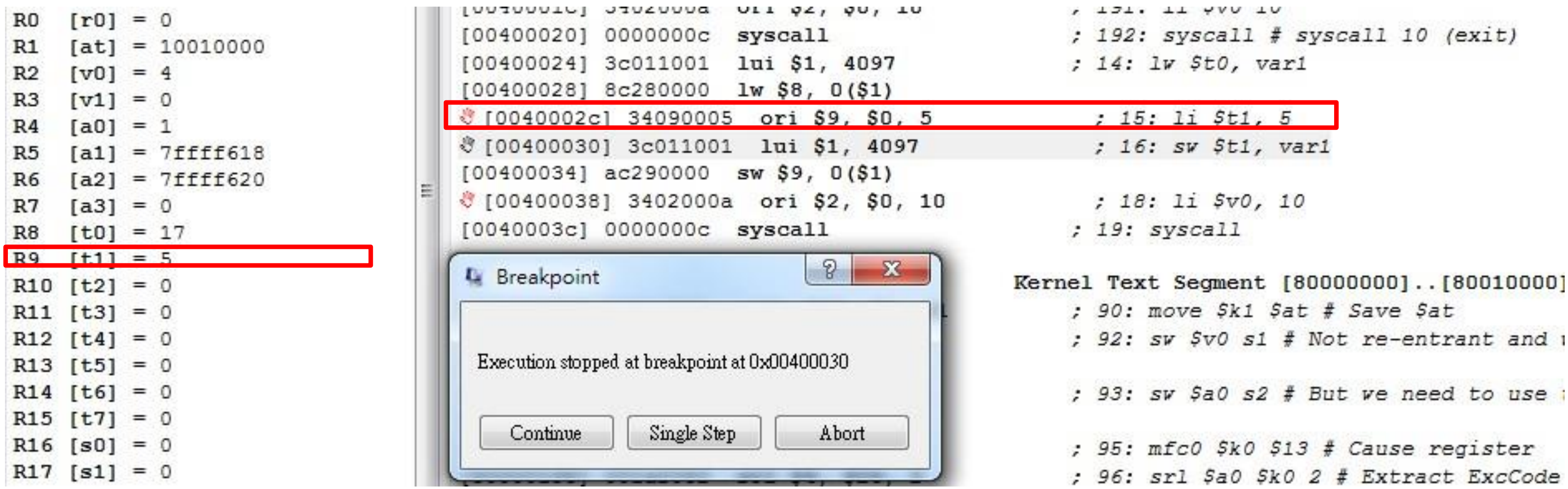
MIPS Instructions (**Load / Store Instructions**)

- Example
  - **lw $t0, var1**



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

- Example
  - **li** **$t1, 5**



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

http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm

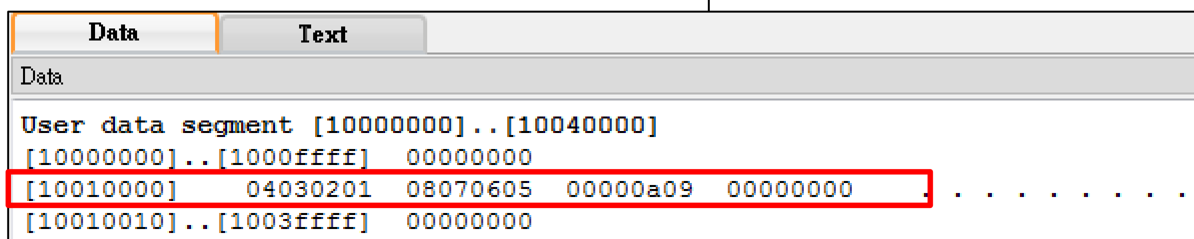
MIPS Instructions (**Indirect and Based Addressing**)

- **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

http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm

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
```



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)

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

http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm

MIPS Instructions (**Arithmetic Instructions**)

- MIPS

mult \$t3,\$t4	**multiply 32-bit quantities in \$t3 and \$t4, and store 64-bit result in special registers Lo and Hi: (Hi,Lo) = \$t3 * \$t4**
div \$t5,\$t6	**Lo = \$t5 / \$t6 (integer quotient) Hi = \$t5 mod \$t6 (remainder)**
mfhi \$t0	**move quantity in special register Hi to \$t0: \$t0 = Hi**
mflo \$t1	**move quantity in special register Lo to \$t1: \$t1 = Lo used to get at result of product or quotient**

- RISC-V

ARITHMETIC CORE INSTRUCTION SET

RV64M Multiply Extension

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)
`mul,mulw`	R	MULtiply (Word)	R[rd] = (R[rs1] * R[rs2])(63:0)
`mulh`	R	MULtiply High	R[rd] = (R[rs1] * R[rs2])(127:64)
`mulhu`	R	MULtiply High Unsigned	R[rd] = (R[rs1] * R[rs2])(127:64)
`mulhsu`	R	MULtiply upper Half Sign/Uns	R[rd] = (R[rs1] * R[rs2])(127:64)
`div,divw`	R	DIVide (Word)	R[rd] = (R[rs1] / R[rs2])
`divu`	R	DIVide Unsigned	R[rd] = (R[rs1] / R[rs2])
`rem,remw`	R	REMainder (Word)	R[rd] = (R[rs1] % R[rs2])
`remu,remuw`	R	REMainder Unsigned (Word)	R[rd] = (R[rs1] % R[rs2])

MIPS Instructions (**Control Structures**)

- **Branches**

beq $t0,$t1,target	**branch to target if $t0 = $t1**
blt $t0,$t1,target	**branch to target if $t0 < $t1**
ble $t0,$t1,target	**branch to target if $t0 <= $t1**
bgt $t0,$t1,target	**branch to target if $t0 > $t1**
bge $t0,$t1,target	**branch to target if $t0 >= $t1**
bne $t0,$t1,target	**branch to target if $t0 <> $t1**

- **Jumps**

j target	**unconditional jump to program label target**
jr $t3	**jump to address contained in $t3 ("jump register")**

http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm

MIPS Instructions (**Control Structures**)

- MIPS

Jump	`j`	J	PC=JumpAddr
Jump And Link	`jal`	J	R[31]=PC+8;PC=JumpAddr
Jump Register	`jr`	R	PC=R[rs]

- RISC-V

`jal`	UJ	Jump & Link	R[rd] = PC+4; PC = PC + {imm, 1b'0}
`jalr`	I	Jump & Link Register	R[rd] = PC+4; PC = R[rs1]+imm
`lb`	I	Load Byte	R[rd] = {56'bM[](7),M[R[rs1]+imm](7:0)}

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

http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm

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

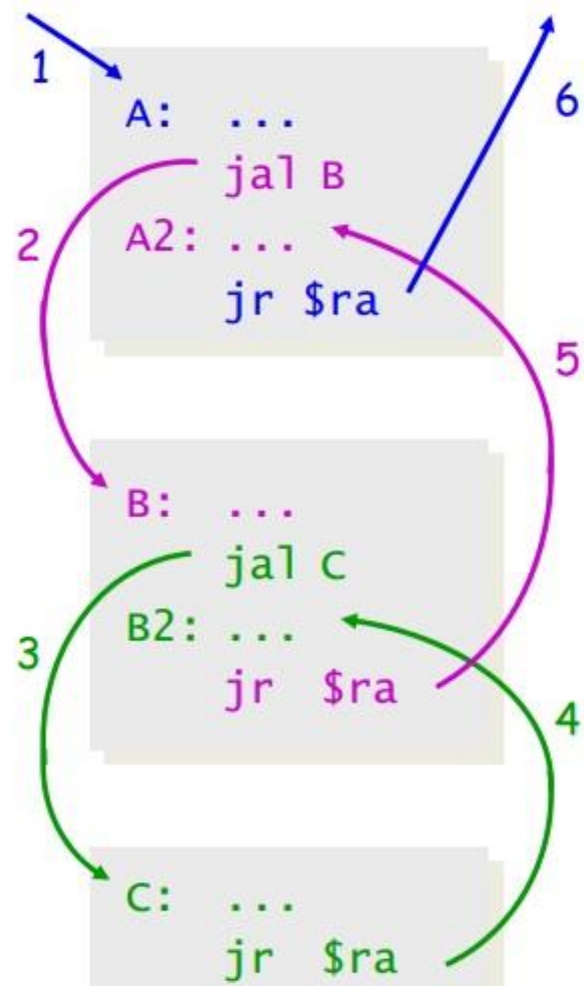
- To transfer control back to the caller, the function just has to jump to the address that was stored in $ra.

jr $ra

Note: return address stored in register $ra; if subroutine will call other subroutines, or is recursive, <u>return address should be copied from $ra onto stack to preserve it,</u> 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 consule
```

```
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

Pseudo instruction	Real instruction(s)	
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

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

- 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).

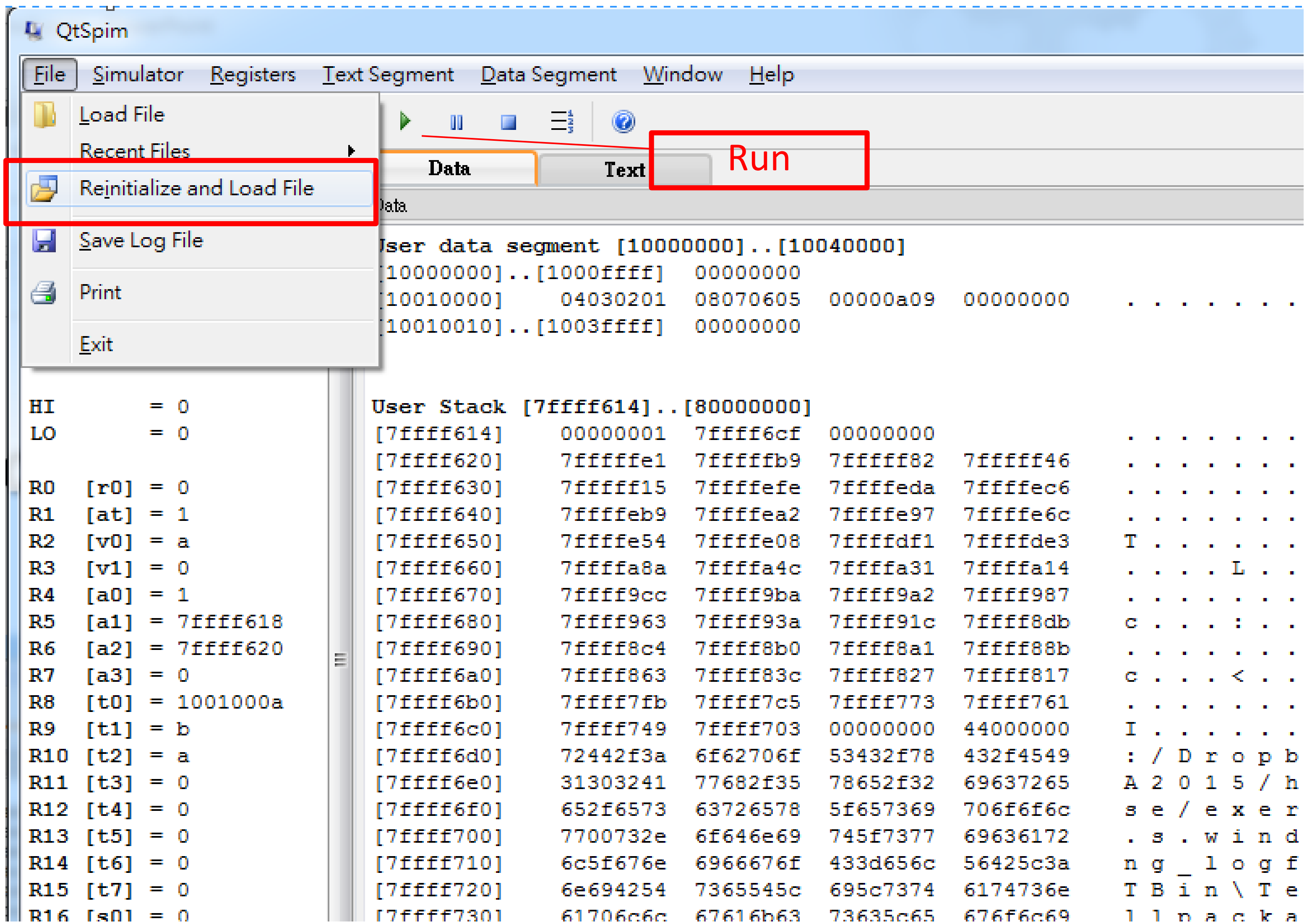
```
    lw $v1, 8($sp)
    add $v0, $v0, $v1
```

The results are summed and put in $v0.

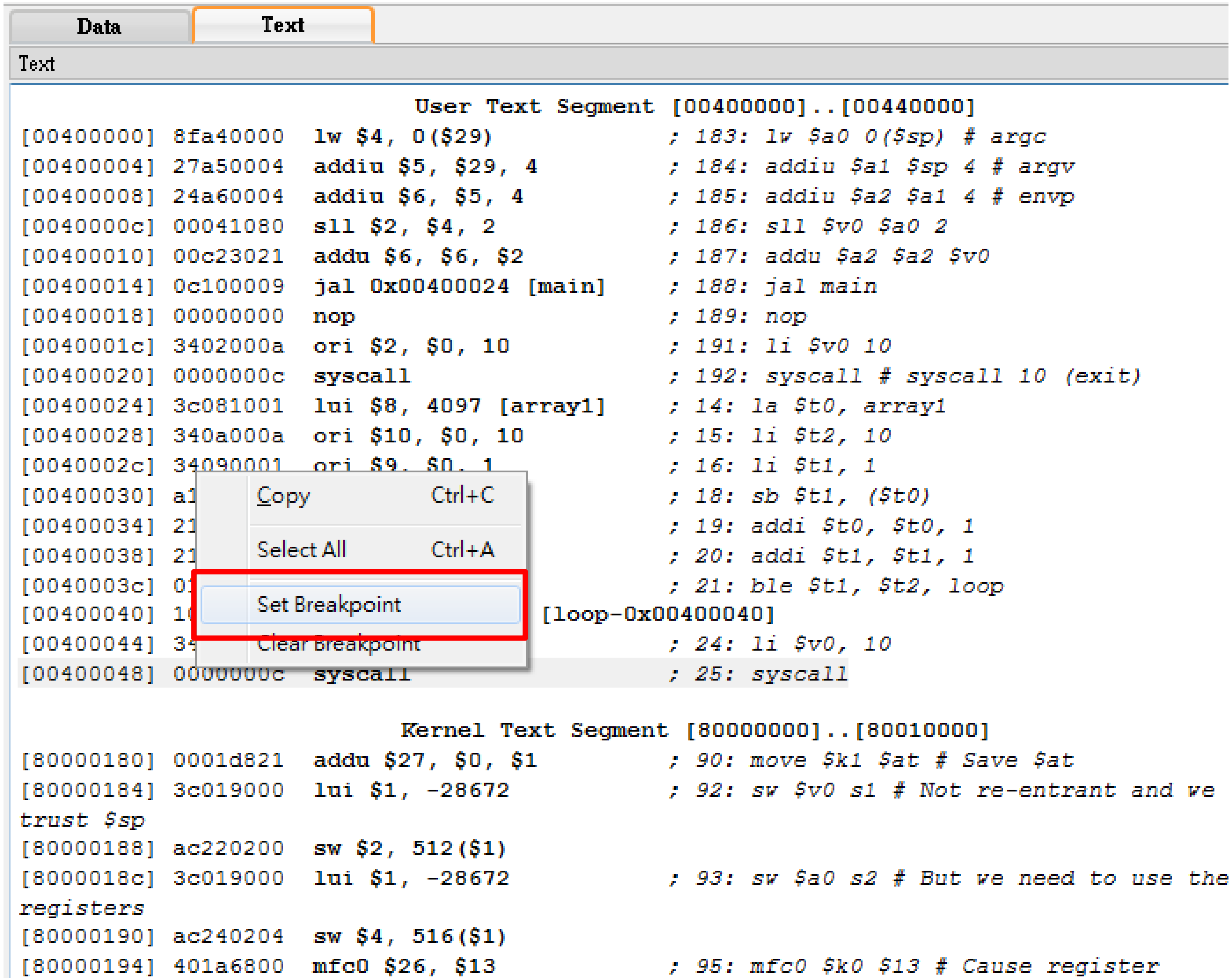
```
    lw $ra, 0($sp)
    addi $sp, $sp, 12
    jr $ra
```

Retrieve return address and restore the stack pointer

Load your program

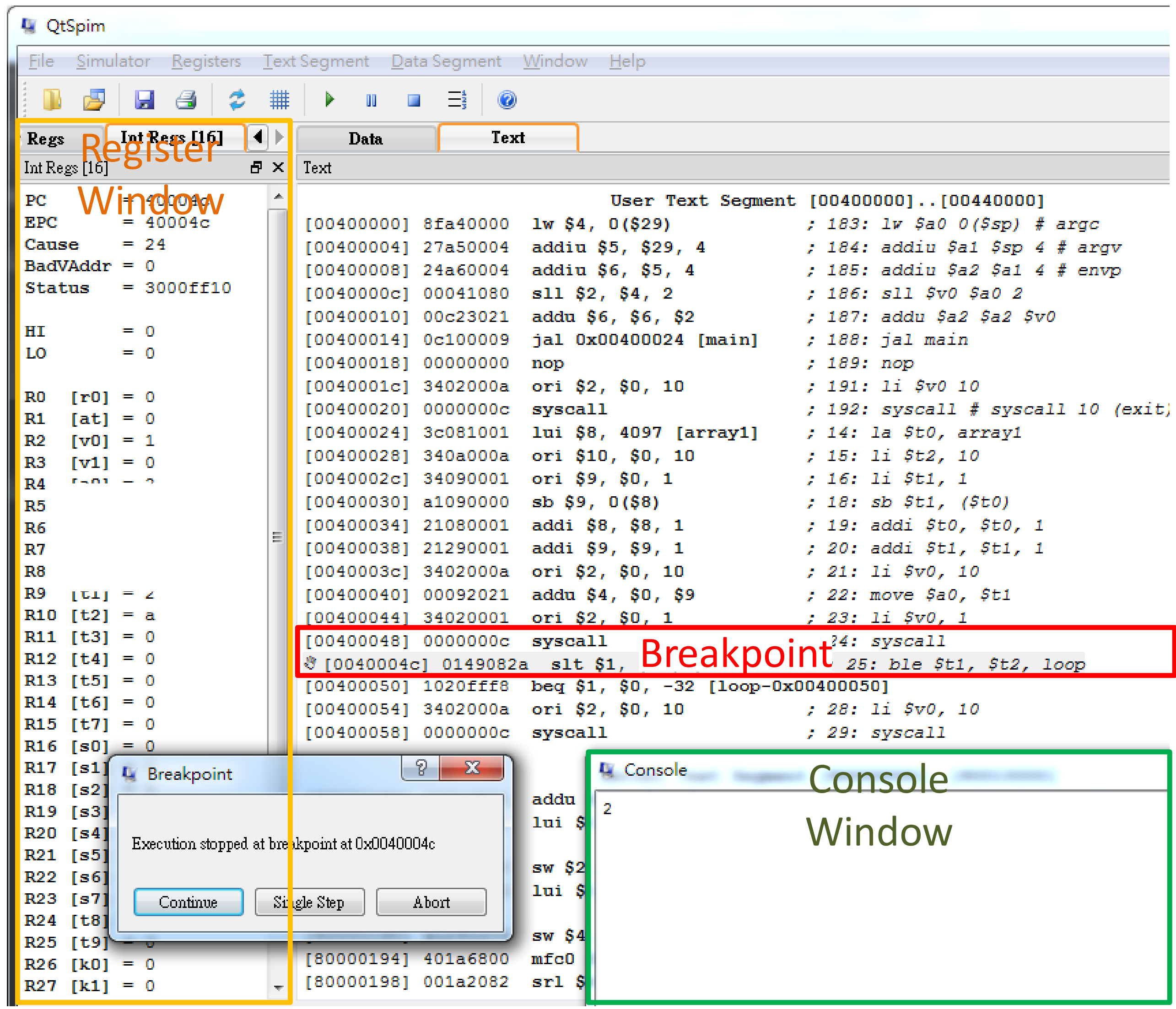


Breakpoint


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 ; 18: sb $t1, ($t0)
[00400034] 21 ; 19: addi $t0, $t0, 1
[00400038] 21 ; 20: addi $t1, $t1, 1
[0040003c] 01 ; 21: ble $t1, $t2, loop
[00400040] 10 [loop-0x00400040]
[00400044] 34 ; 24: li $v0, 10
[00400048] 0000000c syscall ; 25: syscall
Copy Ctrl+C
Select All Ctrl+A
Set Breakpoint
Clear Breakpoint
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 s1 # Not re-entrant and we trust $sp
[80000188] ac220200 sw $2, 512($1)
[8000018c] 3c019000 lui $1, -28672 ; 93: sw $a0 s2 # But we need to use the registers
[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



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

0101

      - $0 \le$ 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:
        .asciiz "input.txt"
    file_out:
        .asciiz "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: 2018/10/16 (Tuesday) midnight (23:59:59)
  - Please upload to NTU COOL

"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

- 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