

# Introduction to SPIM Simulator

Jayaraj P B

---

# SPIM Simulator

- SPIM is a software simulator that runs programs written for MIPS processors
- SPIM's name is just MIPS spelled backwards
- SPIM can read and immediately execute MIPS assembly language files or MIPS executable files
- SPIM contains a debugger and provides a few operating system-like services

# MIPS Processors

- MIPS is a **load-store architecture**, which means that **only load and store instructions access memory**
- **Computation instructions** operate only on values in **registers**

# MIPS Registers

Name	Number	Usage
\$zero	0	constant 0
\$at	1	reserved for assembler
\$v0~\$v1	2~3	return value of a function
\$a0~\$a3	4~7	arguments
\$t0~\$t7	8~15	temporary (not preserved across call)
\$s0~\$s7	16~23	saved temporary (preserved across call)
\$t8~\$t9	24~25	temporary (not preserved across call)
\$k0~\$k1	26~27	reserved for OS kernel
\$gp	28	pointer to global area
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address

# MIPS: Software Conventions for Registers

0	zero	constant 0
1	at	reserved for assembler
2	v0	results from callee
3	v1	returned to caller
4	a0	arguments to callee
5	a1	from caller: caller saves
6	a2	
7	a3	
8	t0	temporary
...		
15	t7	

16	s0	callee saves
...		
23	s7	
24	t8	temporary (cont'd)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	pointer to global area
29	sp	stack pointer
30	fp	frame pointer
31	ra	return Address
		caller saves

# Pseudoinstructions

- Here's a list of useful pseudo-instructions.
- **mov \$t0, \$t1**: Copy contents of register t1 to register t0.
- **li \$s0, immed**: Load immediate into to register s0.
  - The way this is translated depends on whether **immed** is 16 bits or 32 bits.
- **la \$s0, addr**: Load address into to register s0.
- **lw \$t0, address**: Load a word at address into register t0
- Similar pseudo-instructions exist for **sw**, etc.

# Pseudoinstructions

- **Translating Some Pseudoinstructions**
- **mov \$t0, \$s0**                      addi \$t0, \$s0, 0
- **li \$rs, small**                      addi \$rs, \$zero, small
- **li \$rs, big**                          lui \$rs, upper(big) ori \$rs, \$rs, lower(big)
- **la \$rs, big**                          \$rs, upper(big) ori \$rs, \$rs, lower(big)
- where **small** means a quantity that can be represented using 16 bits, and **big** means a 32 bit quantity. **upper( big )** is the upper 16 bits of a 32 bit quantity. **lower( big )** is the lower 16 bits of the 32 bit quantity.
- **upper( big )** and **lower(big)** are not real instructions. If you were to do the translation, you'd have to break it up yourself to figure out those quantities.

# Arithmetic Instructions

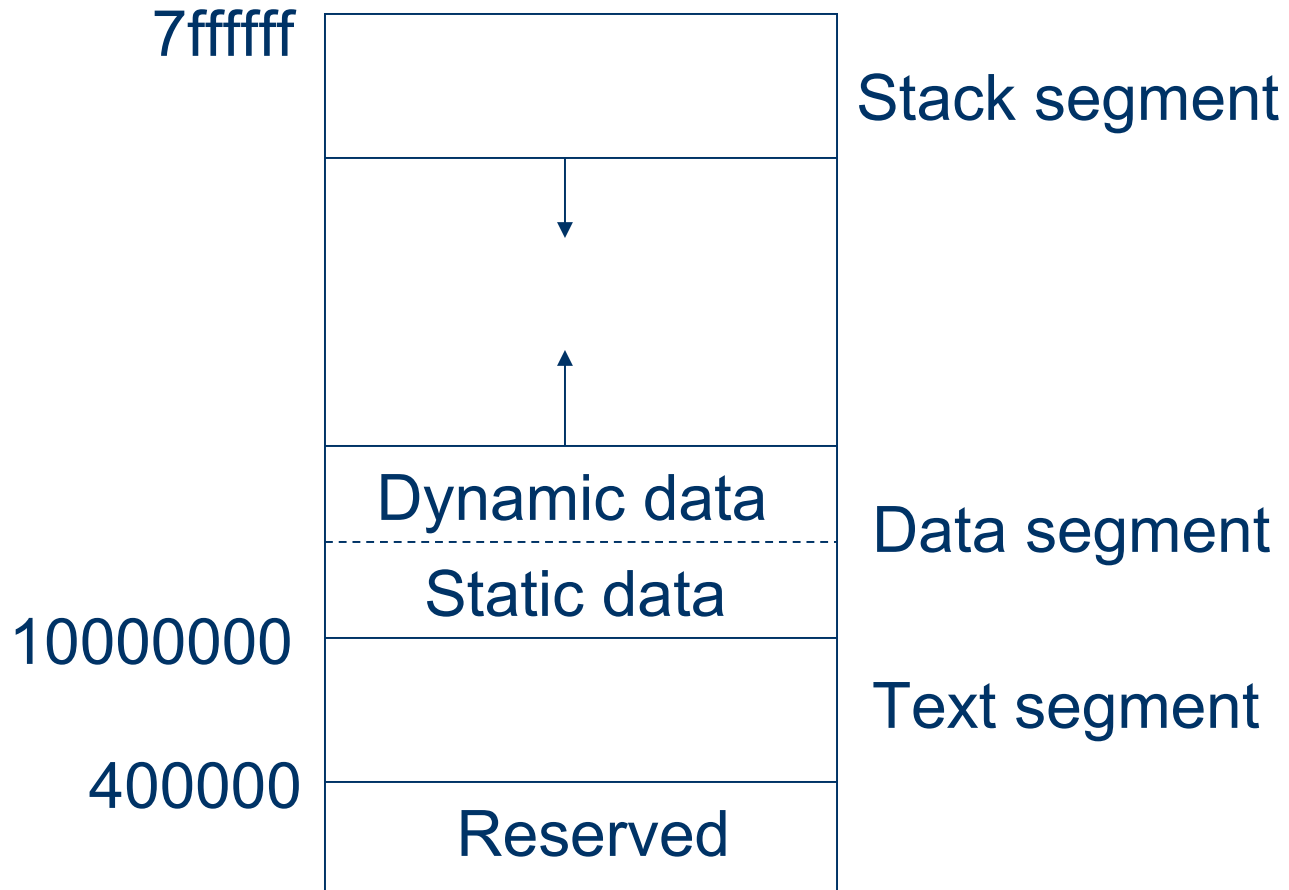
add	rd, rs, rt	$rd \leftarrow rs + rt$
sub	rd, rs, rt	$rd \leftarrow rs - rt$
mul	rd, rs, rt	$rd \leftarrow rs * rt$
div	rd, rs, rt	$rd \leftarrow rs / rt$
rem	rd, rs, rt	$rd \leftarrow rs \% rt$
neg	rd, rs	$rd \leftarrow - rs$



# Branch Instructions

beq	rs, rt, label	branch to label if $rs == rt$
bne	rs, rt, label	branch to label if $rs != rt$
bgt	rs, rt, label	branch to label if $rs > rt$
bge	rs, rt, label	branch to label if $rs \geq rt$
blt	rs, rt, label	branch to label if $rs < rt$
ble	rs, rt, label	branch to label if $rs \leq rt$
b	label	branch to label

# Memory Layout



# MIPS Assembler Directives

- Top-level Directives:

## **.text**

- indicates that following items are stored in the user text segment, typically instructions

## **.data**

- indicates that following data items are stored in the data segment

## **.globl** sym

- declare that symbol sym is global and can be referenced from other files

# MIPS Assembler Directives

- Common Data Definitions:

- **.word** w1, ..., wn
  - store n 32-bit quantities in successive memory words
- **.half** h1, ..., hn
  - store n 16-bit quantities in successive memory halfwords
- **.byte** b1, ..., bn
  - store n 8-bit quantities in successive memory bytes
- **.ascii** str
  - store the string in memory but do not null-terminate it
    - strings are represented in double-quotes “str”
    - special characters, eg. \n, \t, follow C convention
- **.asciiz** str
  - store the string in memory and null-terminate it

# MIPS Assembler Directive

## S

- Common Data Definitions:

- **.float** f1, ..., fn
  - store n floating point single precision numbers in successive memory locations
- **.double** d1, ..., dn
  - store n floating point double precision numbers in successive memory locations
- **.space** n
  - reserves n successive bytes of space
- **.align** n
  - align the next datum on a  $2^n$  byte boundary.
  - For example, **.align 2** aligns next value on a word boundary.
  - **.align 0** turns off automatic alignment of **.half**, **.word**, etc. till next **.data** directive

# System Calls

- SPIM provides a small set of operating system-like services through the **system call** (**syscall**) instruction
- To request a service, a program loads the system call code into register **\$v0** and arguments into registers **\$a0~\$a3**
- System calls that return values put their results in register **\$v0**

# System Call Code

Service	Code (put in \$v0)	Arguments	Result
print_int	1	\$a0=integer	
print_float	2	\$f12=float	
print_double	3	\$f12=double	
print_string	4	\$a0=addr. of string	
read_int	5		int 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	addr in \$v0
exit	10		

# QtSPIM

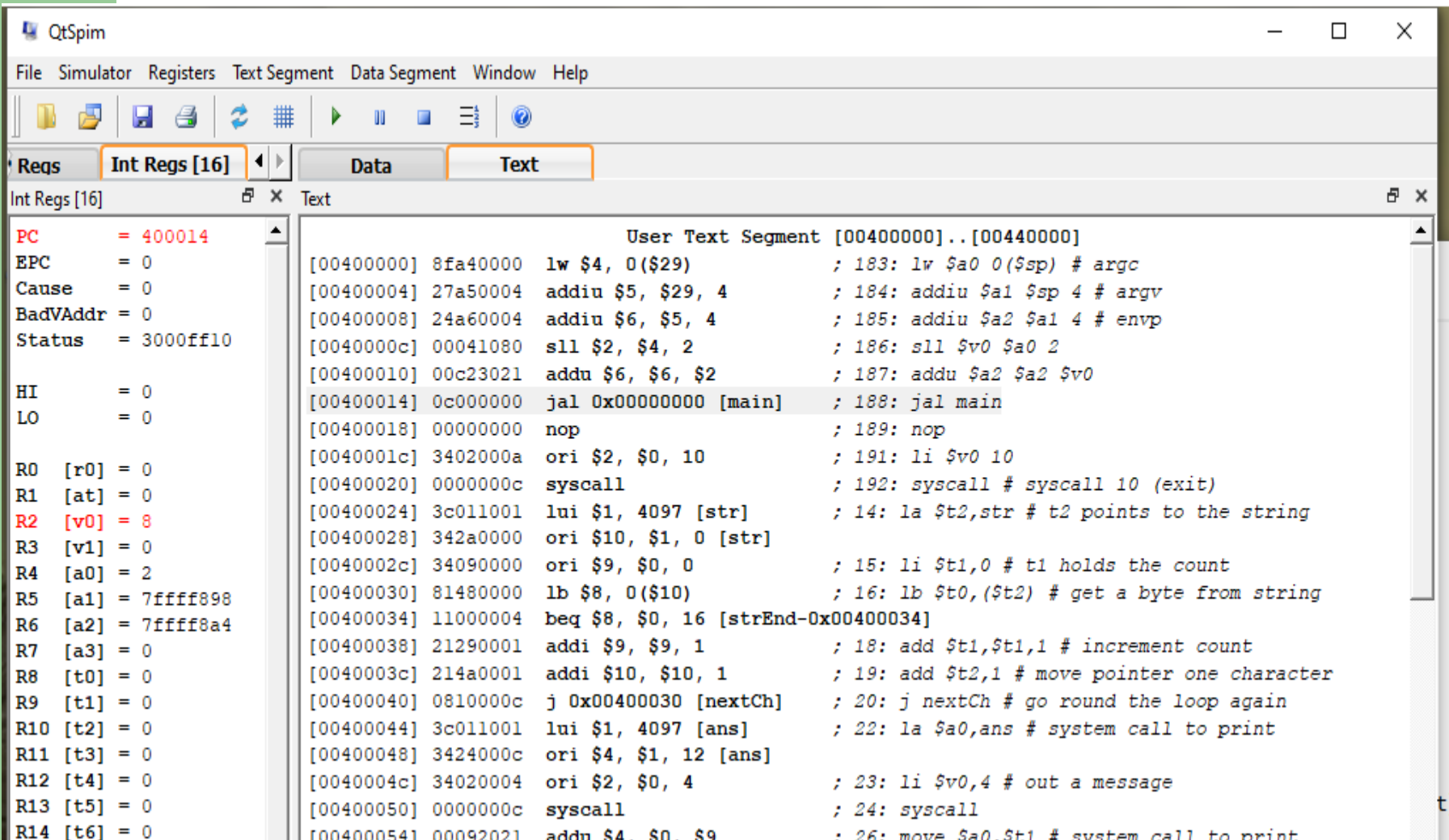
- QtSpim is software that will help you to simulate the execution of MIPS assembly programs.
- It does a context and syntax check while loading an assembly program.
- In addition, it adds in necessary overhead instructions as needed, and updates register and memory content as each instruction is executed.
- Download the source from the SourceForge.org link at:  
<http://pages.cs.wisc.edu/~larus/spim.html>
- Alternatively, you can go directly to:  
<http://sourceforge.net/projects/spimsimulator/files/>
- Versions for Windows, Linux, and Macs are all available



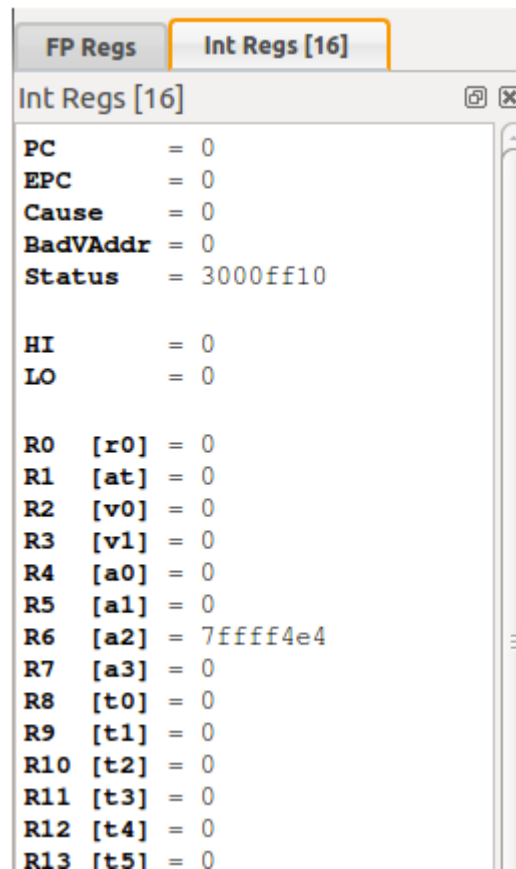
# QtSPIM

- QtSPIM window is divided into different sections:
  1. The *Register tabs* display the content of all registers.
  2. Buttons across the top are used to load and run a simulation
    - Functionality is described in Figure 2.
  3. The *Text tab* displays the MIPS instructions loaded into memory to be executed.
    - From left-to-right, the memory address of an instruction, the contents of the address in hex, the actual MIPS instructions where register numbers are used, the MIPS assembly that you wrote, and any comments you made in your code are displayed.
  4. The *Data tab* displays memory addresses and their values in the data and stack segments of the memory.
  5. The *Information Console* lists the actions performed by the simulator.

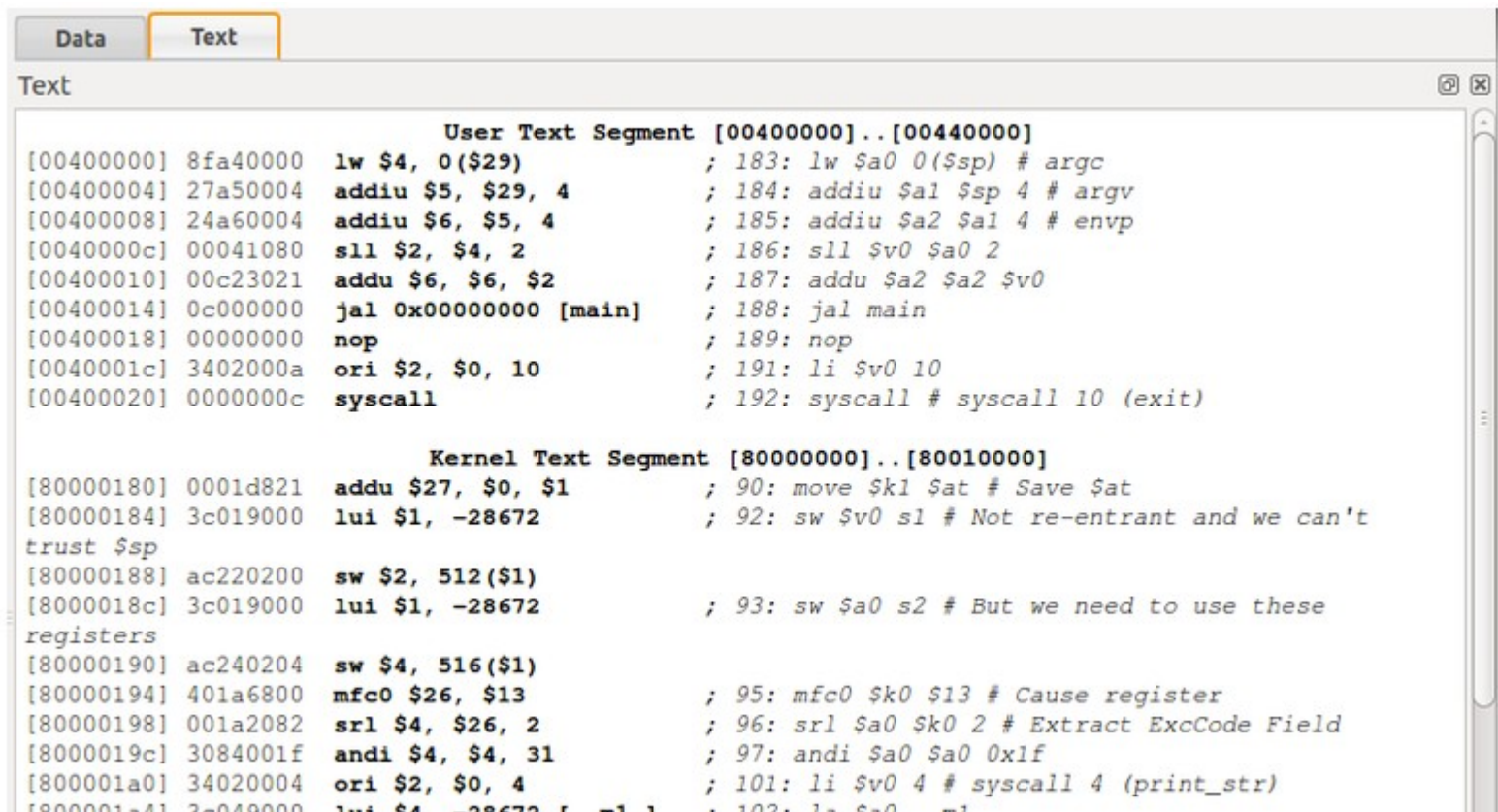
# QtSpim simulator - screenshot



# Register panel



# Memory panel: text panel



The screenshot shows a debugger window with two tabs: 'Data' and 'Text'. The 'Text' tab is selected, displaying assembly code for two segments: 'User Text Segment' and 'Kernel Text Segment'.

**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] 0c000000 jal 0x00000000 [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)
```

**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 can't
trust $sp
[80000188] ac220200 sw $2, 512($1)
[8000018c] 3c019000 lui $1, -28672 ; 93: sw $a0 $2 # But we need to use these
registers
[80000190] ac240204 sw $4, 516($1)
[80000194] 401a6800 mfc0 $26, $13 ; 95: mfc0 $k0 $13 # Cause register
[80000198] 001a2082 srl $4, $26, 2 ; 96: srl $a0 $k0 2 # Extract ExcCode Field
[8000019c] 3084001f andi $4, $4, 31 ; 97: andi $a0 $a0 0x1f
[800001a0] 34020004 ori $2, $0, 4 ; 101: li $v0 4 # syscall 4 (print_str)
[800001a4] 3c019000 lui $1, -28672 ; 102: li $v0 4
```

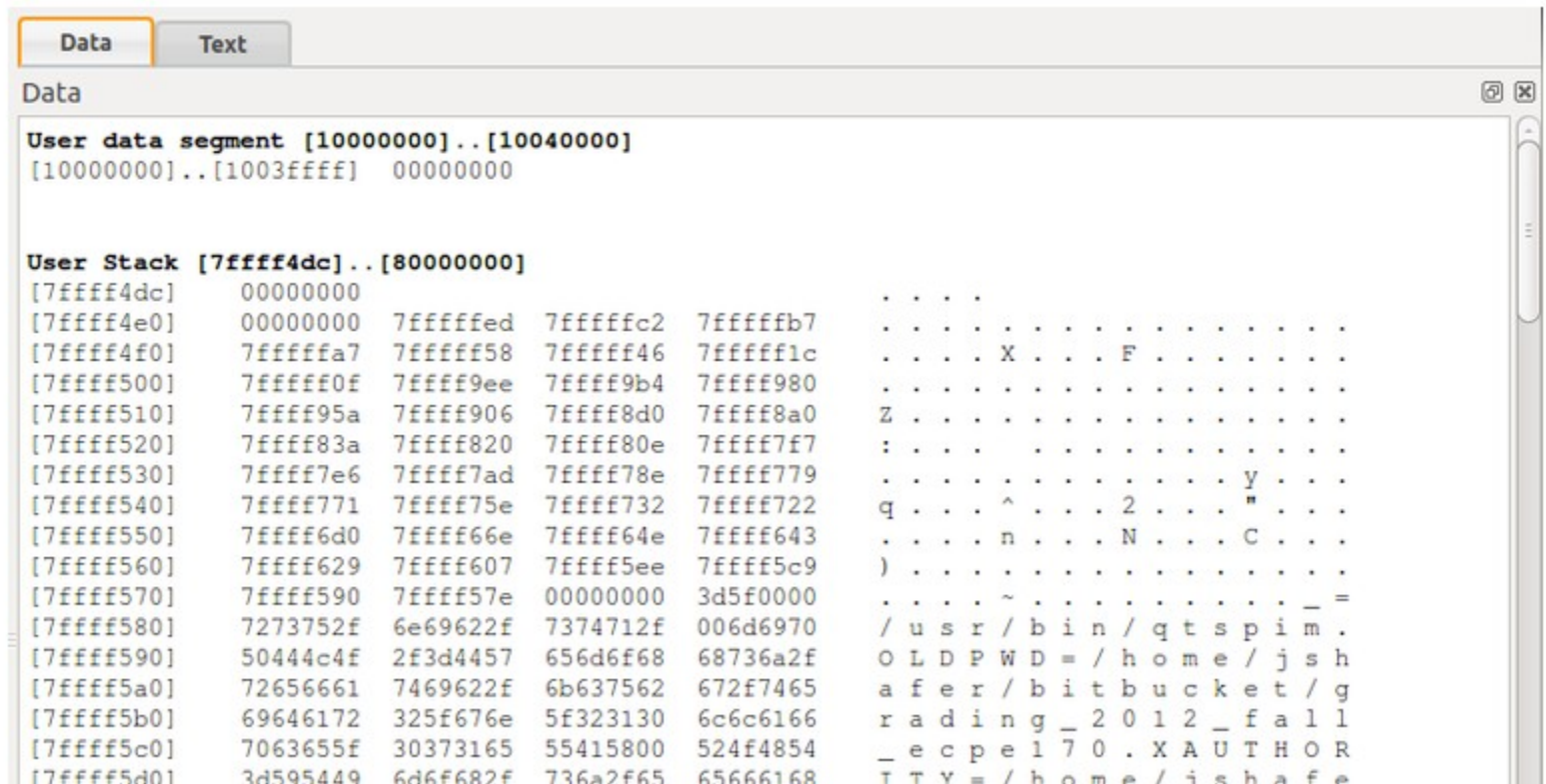
# Message panel

## Messages Panel

The Messages panel displays messages from QtSPIM to the user.

```
Memory and registers cleared  
Loaded: /tmp/qt_temp.MT3159  
SPIM Version 9.1.6 of February 4, 2012  
Copyright 1990-2012, James R. Larus.  
All Rights Reserved.
```

# Memory panel :data segment



The screenshot shows a debugger's memory panel with two tabs: 'Data' and 'Text'. The 'Data' tab is selected, displaying the 'User data segment' and 'User Stack'.

**User data segment [10000000]..[10040000]**  
[10000000]..[1003ffff] 00000000

**User Stack [7ffff4dc]..[80000000]**

Address	Value	Address	Value	Address	Value	Address	Value
[7ffff4dc]	00000000						
[7ffff4e0]	00000000	7fffffed	7fffffc2	7fffffb7			
[7ffff4f0]	7fffffa7	7fffff58	7fffff46	7fffff1c			
[7ffff500]	7fffff0f	7ffff9ee	7ffff9b4	7ffff980			
[7ffff510]	7ffff95a	7ffff906	7ffff8d0	7ffff8a0			
[7ffff520]	7ffff83a	7ffff820	7ffff80e	7ffff7f7			
[7ffff530]	7ffff7e6	7ffff7ad	7ffff78e	7ffff779			
[7ffff540]	7ffff771	7ffff75e	7ffff732	7ffff722			
[7ffff550]	7ffff6d0	7ffff66e	7ffff64e	7ffff643			
[7ffff560]	7ffff629	7ffff607	7ffff5ee	7ffff5c9			
[7ffff570]	7ffff590	7ffff57e	00000000	3d5f0000			
[7ffff580]	7273752f	6e69622f	7374712f	006d6970			
[7ffff590]	50444c4f	2f3d4457	656d6f68	68736a2f			
[7ffff5a0]	72656661	7469622f	6b637562	672f7465			
[7ffff5b0]	69646172	325f676e	5f323130	6c6c6166			
[7ffff5c0]	7063655f	30373165	55415800	524f4854			
[7ffff5d0]	3d595449	6d6f682f	736a2f65	65666168			

The rightmost column of the stack shows the ASCII representation of the memory values, which includes the string: `/usr/bin/qtspm.OLDPWD=/home/jshafter/bitbucket/g...rad ing_2012_fall..._ecpe170.XAUTHOR...TTY=/home/is h a f e`

# Program template

```
.data
    # data segment
.text
.globl main
main:
    # your code will come here

EXIT: li $v0,10
      syscall
```

# Program for reading a string

```
.data
theString: .space 64
.text
main:
    li    $v0, 8
    la    $a0, theString
    li    $a1, 64
    syscall
    jr    $ra
```



```

        .text
        .globl main

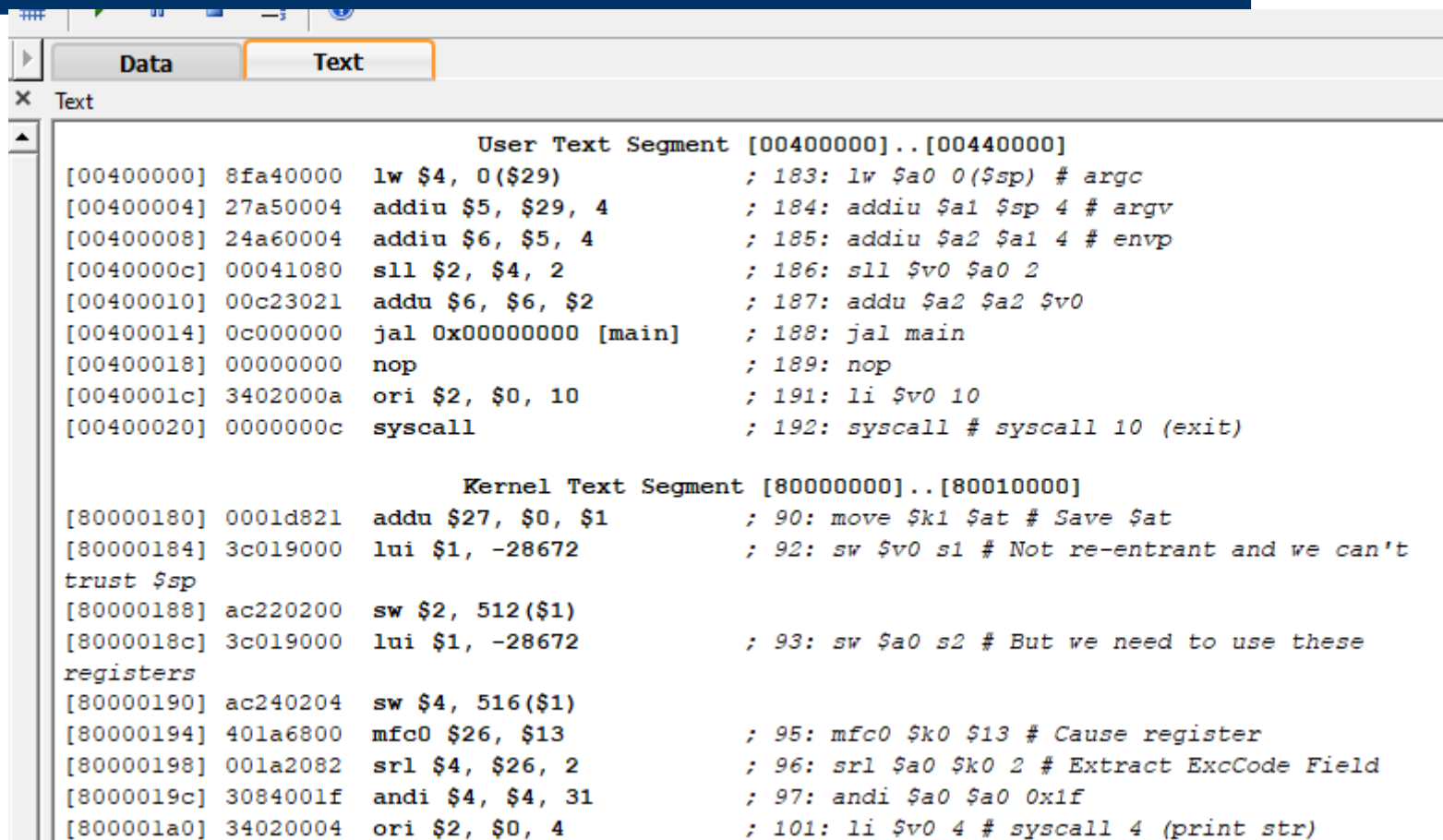
main:
    li $t2, 25           # Load immediate value (25)
    lw $t3, value        # Load the word stored in val
    add $t4, $t2, $t3    # Add
    sub $t5, $t2, $t3    # Subtract
    sw $t5, Z            #Store the answer in Z (decla

    li $v0, 10 # Sets $v0 to "10" to select exit syscall
    syscall # Exit

        .data
value:  .word 12
Z:      .word 0

```

# User text segment



The screenshot shows a debugger window with two tabs: 'Data' and 'Text'. The 'Text' tab is selected, displaying assembly code for two segments: 'User Text Segment' and 'Kernel Text Segment'.

**User Text Segment [00400000]..[00440000]**

Address	Hex	Assembly	Comment
[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]	0c000000	jal 0x00000000 [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)

**Kernel Text Segment [80000000]..[80010000]**

Address	Hex	Assembly	Comment
[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 can't trust \$sp
[80000188]	ac220200	sw \$2, 512(\$1)	
[8000018c]	3c019000	lui \$1, -28672	; 93: sw \$a0 \$2 # But we need to use these registers
[80000190]	ac240204	sw \$4, 516(\$1)	
[80000194]	401a6800	mfc0 \$26, \$13	; 95: mfc0 \$k0 \$13 # Cause register
[80000198]	001a2082	srl \$4, \$26, 2	; 96: srl \$a0 \$k0 2 # Extract ExcCode Field
[8000019c]	3084001f	andi \$4, \$4, 31	; 97: andi \$a0 \$a0 0x1f
[800001a0]	34020004	ori \$2, \$0, 4	; 101: li \$v0 4 # syscall 4 (print_str)

# QtSPIM Program Example

- A Simple Program

```
#sample example 'add two numbers'
```

```
.text                                # text section
.globl main                          # call main by SPIM

main:    la $t0, value               # load address 'value' into $t0
         lw $t1, 0($t0)              # load word 0(value) into $t1
         lw $t2, 4($t0)              # load word 4(value) into $t2
         add $t3, $t1, $t2           # add two numbers into $t3
         sw $t3, 8($t0)              # store word $t3 into 8($t0)

.data                                # data section
value:   .word 10, 20, 0             data for addition
#
```

# QtSPIM Example Program

```
## Program adds 10 and 11
```

```
.text                # text section  
.globl    main        # call main by SPIM
```

```
main:
```

```
    ori    $8,$0,0xA    # load "10" into register 8  
    ori    $9,$0,0xB    # load "11" into register 9  
    add     $10,$8,$9    # add registers 8 and 9, put result  
                        # in register 10
```

## QtSPIM Example Program: swap2memoryWords.asm

```
## Program to swap two memory words
```

```
.data                # load data
```

```
.word 7
```

```
.word 3
```

```
.text
```

```
.globl main
```

```
main:
```

```
lui $s0, 0x1001 # load data area start address 0x10010000
```

```
lw  $s1, 0($s0)
```

```
lw  $s2, 4($s0)
```

```
sw  $s2, 0($s0)
```

```
sw  $s1, 4($s0)
```