



计算机组成原理与接口技术（实验） ——基于MIPS架构

Oct, 2016

实验1 MIPS汇编程序设计（第8周）

杨明
华中科技大学电子信息与通信学院
myang@hust.edu.cn



▶ 实验目的

▶ 实验任务及要求

▶ 汇编程序结构

- CPU回顾
- 存储器回顾
- 汇编程序框架

▶ QtSpim汇编软件

- QtSpim简介
- Qtspim系统功能调用
- QtSpim使用示例

- ▶ 熟悉常见的MIPS汇编指令
- ▶ 掌握MIPS汇编程序设计
- ▶ 了解MIPS汇编语言与机器语言之间的对应关系
- ▶ 了解C语言语句与汇编指令之间的关系
- ▶ 掌握QtSpim的调试技术
- ▶ 掌握程序的内存映像

► 实验目的

► 实验任务及要求

► 汇编程序结构

- CPU回顾
- 存储器回顾
- 汇编程序框架

► QtSpim汇编软件

- QtSpim简介
- Qtspim系统功能调用
- QtSpim使用示例

实验任务及要求

- ▶ 任务：用汇编程序实现以下伪代码，要求采用移位指令实现乘除法运算。

```
int main()
{
    int K, Y ;
    int Z[50] ;
    Y = 56;
    for(K=0;K<50;K++)
        Z[K] = Y - 16 * ( K / 4 + 210 ) ;
}
```

- ▶ 要求

- 完成汇编语言程序设计、调试、测试全过程
- 指出用户程序的内存映像，包括代码段和数据段
- 完成软件实验报告，下次上课时交（手写or打印均可，封面签名，禁止抄袭）
- （选做：把Z[K]在屏幕上显示出来）



实验报告要求

- 1. 实验要求
- 2. 汇编源程序设计思路（算法）、源代码及注释
- 3. 详细的调试、测试过程（可以截图并附加说明）
- 4. 程序内存映像（包括数据段和代码段）
 - 最后数据结果Z[K]（K=0~49）的内存映像需验收
- 5. 心得体会

数据段内存映像需要画成这种格式！

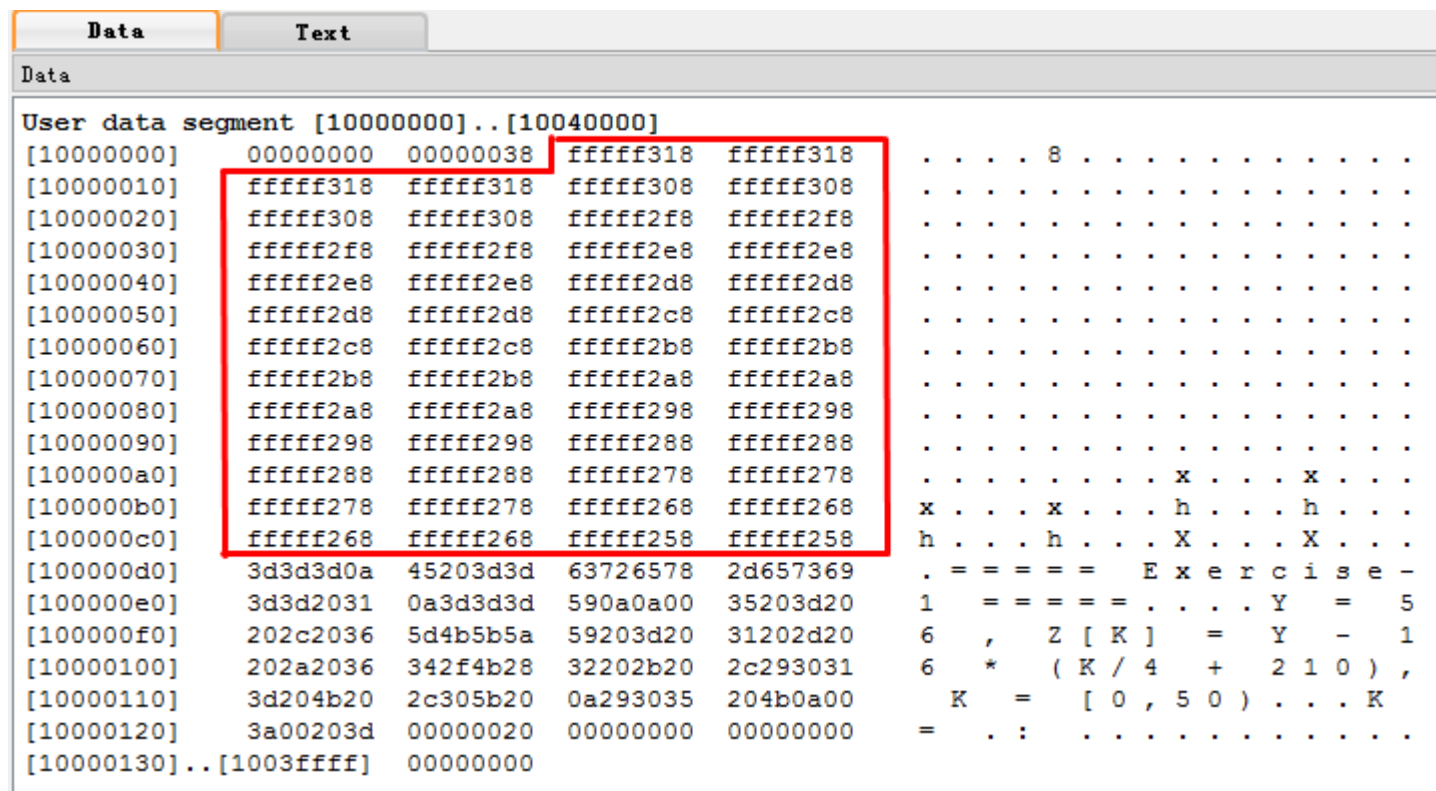
变量名	地址	数据	定义值
<u>str</u>	0x 1000 0000	0x64	"d"
	0x 1000 0001	0x63	"c"
	0x 1000 0002	0x62	"b"
	0x 1000 0003	0x61	"a"
<u>strn</u>	0x 1000 0004	0x44	"D"
	0x 1000 0005	0x43	"C"
	0x 1000 0006	0x42	"B"
	0x 1000 0007	0x41	"A"
	0x 1000 0008	0x00	"\0"
	0x 1000 0009	0x47	"G"
	0x 1000 000A	0x46	"F"
	0x 1000 000B	0x45	"E"
b0	0x 1000 000C	0x04	4
	0x 1000 000D	0x03	3
	0x 1000 000E	0x02	2
	0x 1000 000F	0x01	1
	0x 1000 0010	0x00	
	0x 1000 0011	0x00	
	0x 1000 0012	0x00	
	0x 1000 0013	0x05	5

代码段内存映像需要画成这种格式！

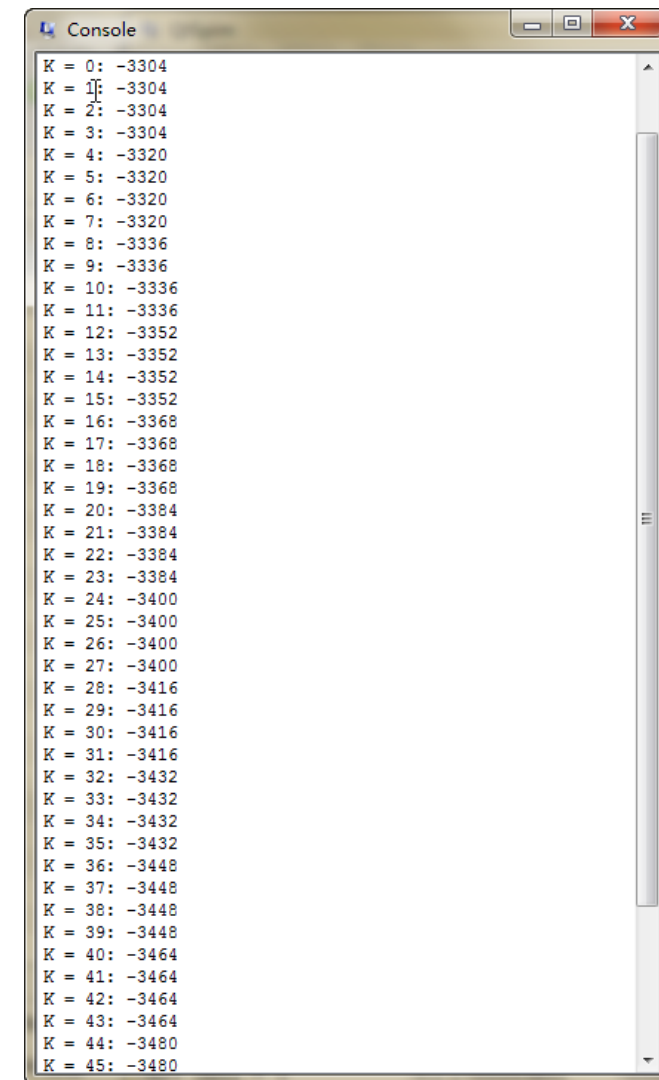
内存地址 (16进制)	汇编指令	机器码 (16进制)
00400024	Ori \$2,\$0,1	34020001
...
...

► QtSpim软件

- 数据段内存
- Z[0]-Z[9]在控制台的输出



```
Data
User data segment [10000000]..[10040000]
[10000000] 00000000 00000038 fffff318 fffff318 . . . 8 . . . . .
[10000010] fffff318 fffff318 fffff308 fffff308 . . . . .
[10000020] fffff308 fffff308 fffff2f8 fffff2f8 . . . . .
[10000030] fffff2f8 fffff2f8 fffff2e8 fffff2e8 . . . . .
[10000040] fffff2e8 fffff2e8 fffff2d8 fffff2d8 . . . . .
[10000050] fffff2d8 fffff2d8 fffff2c8 fffff2c8 . . . . .
[10000060] fffff2c8 fffff2c8 fffff2b8 fffff2b8 . . . . .
[10000070] fffff2b8 fffff2b8 fffff2a8 fffff2a8 . . . . .
[10000080] fffff2a8 fffff2a8 fffff298 fffff298 . . . . .
[10000090] fffff298 fffff298 fffff288 fffff288 . . . . .
[100000a0] fffff288 fffff288 fffff278 fffff278 . . . . . x . . . x . . .
[100000b0] fffff278 fffff278 fffff268 fffff268 x . . . x . . . h . . . h . . .
[100000c0] fffff268 fffff268 fffff258 fffff258 h . . . h . . . X . . . X . . .
[100000d0] 3d3d3d0a 45203d3d 63726578 2d657369 . = = = = E x e r c i s e -
[100000e0] 3d3d2031 0a3d3d3d 590a0a00 35203d20 1 = = = = . . . Y = 5
[100000f0] 202c2036 5d4b5b5a 59203d20 31202d20 6 , Z [ K ] = Y - 1
[10000100] 202a2036 342f4b28 32202b20 2c293031 6 * ( K / 4 + 2 1 0 ) ,
[10000110] 3d204b20 2c305b20 0a293035 204b0a00 K = [ 0 , 5 0 ) . . . K
[10000120] 3a00203d 00000020 00000000 00000000 = . : . . . . .
[10000130]..[1003ffff] 00000000
```



```
Console
K = 0: -3304
K = 1: -3304
K = 2: -3304
K = 3: -3304
K = 4: -3320
K = 5: -3320
K = 6: -3320
K = 7: -3320
K = 8: -3336
K = 9: -3336
K = 10: -3336
K = 11: -3336
K = 12: -3352
K = 13: -3352
K = 14: -3352
K = 15: -3352
K = 16: -3368
K = 17: -3368
K = 18: -3368
K = 19: -3368
K = 20: -3384
K = 21: -3384
K = 22: -3384
K = 23: -3384
K = 24: -3400
K = 25: -3400
K = 26: -3400
K = 27: -3400
K = 28: -3416
K = 29: -3416
K = 30: -3416
K = 31: -3416
K = 32: -3432
K = 33: -3432
K = 34: -3432
K = 35: -3432
K = 36: -3448
K = 37: -3448
K = 38: -3448
K = 39: -3448
K = 40: -3464
K = 41: -3464
K = 42: -3464
K = 43: -3464
K = 44: -3480
K = 45: -3480
```

实验结果

► Mars软件

- 数据段内存
- Z[0]-Z[9]在控制台的输出

Data Segment								
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10000000	0x00000000	0x00000038	0xffffffff318	0xffffffff318	0xffffffff318	0xffffffff318	0xffffffff308	0xffffffff308
0x10000020	0xffffffff308	0xffffffff308	0xffffffff2f8	0xffffffff2f8	0xffffffff2f8	0xffffffff2f8	0xffffffff2e8	0xffffffff2e8
0x10000040	0xffffffff2e8	0xffffffff2e8	0xffffffff2d8	0xffffffff2d8	0xffffffff2d8	0xffffffff2d8	0xffffffff2c8	0xffffffff2c8
0x10000060	0xffffffff2c8	0xffffffff2c8	0xffffffff2b8	0xffffffff2b8	0xffffffff2b8	0xffffffff2b8	0xffffffff2a8	0xffffffff2a8
0x10000080	0xffffffff2a8	0xffffffff2a8	0xffffffff298	0xffffffff298	0xffffffff298	0xffffffff298	0xffffffff288	0xffffffff288
0x100000a0	0xffffffff288	0xffffffff288	0xffffffff278	0xffffffff278	0xffffffff278	0xffffffff278	0xffffffff268	0xffffffff268
0x100000c0	0xffffffff268	0xffffffff268	0xffffffff258	0xffffffff258	0x3d3d3da	0x45203d3d	0x63726578	0x2d657369
0x100000e0	0x3d3d2031	0x0a3d3d3d	0x590a0a00	0x35203d20	0x202c2036	0x5d4b5b5a	0x59203d20	0x31202d20
0x10000100	0x202a2036	0x342f4b28	0x32202b20	0x2c293031	0x3d204b20	0x2c305b20	0x0a293035	0x204b0a00
0x10000120	0x3a00203d	0x00000020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Mars Messages	Run I/O
===== Exercise-1 =====	
K = 0: -3304	
K = 1: -3304	
K = 2: -3304	
K = 3: -3304	
K = 4: -3320	
K = 5: -3320	
K = 6: -3320	
K = 7: -3320	
K = 8: -3336	
K = 9: -3336	
K = 10: -3336	
K = 11: -3336	
K = 12: -3352	
K = 13: -3352	
K = 14: -3352	
K = 15: -3352	
K = 16: -3368	
K = 17: -3368	
K = 18: -3368	
K = 19: -3368	
K = 20: -3384	
K = 21: -3384	
K = 22: -3384	
K = 23: -3384	
K = 24: -3400	
K = 25: -3400	
K = 26: -3400	
K = 27: -3400	

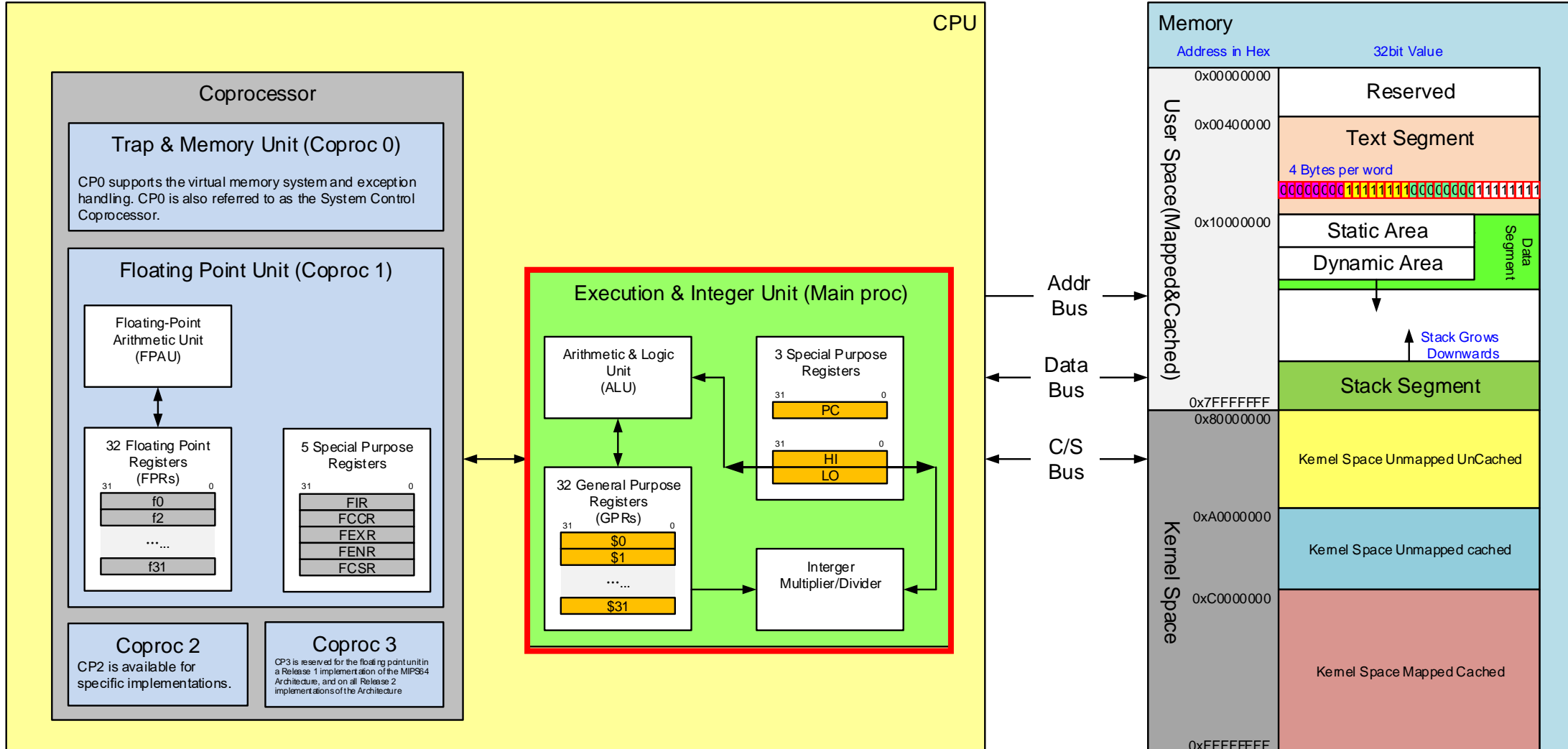
Clear

Agenda

- ▶ 实验目的
- ▶ 实验任务及要求
- ▶ 汇编程序结构
 - CPU寄存器回顾
 - 存储器回顾
 - 汇编程序框架
- ▶ QtSpim汇编软件
 - QtSpim简介
 - Qtspim系统功能调用
 - QtSpim使用示例



► 回顾：CPU和存储器



CPU寄存器回顾

► CPU：通过执行指令，完成运算、控制

- 通用寄存器（32个）

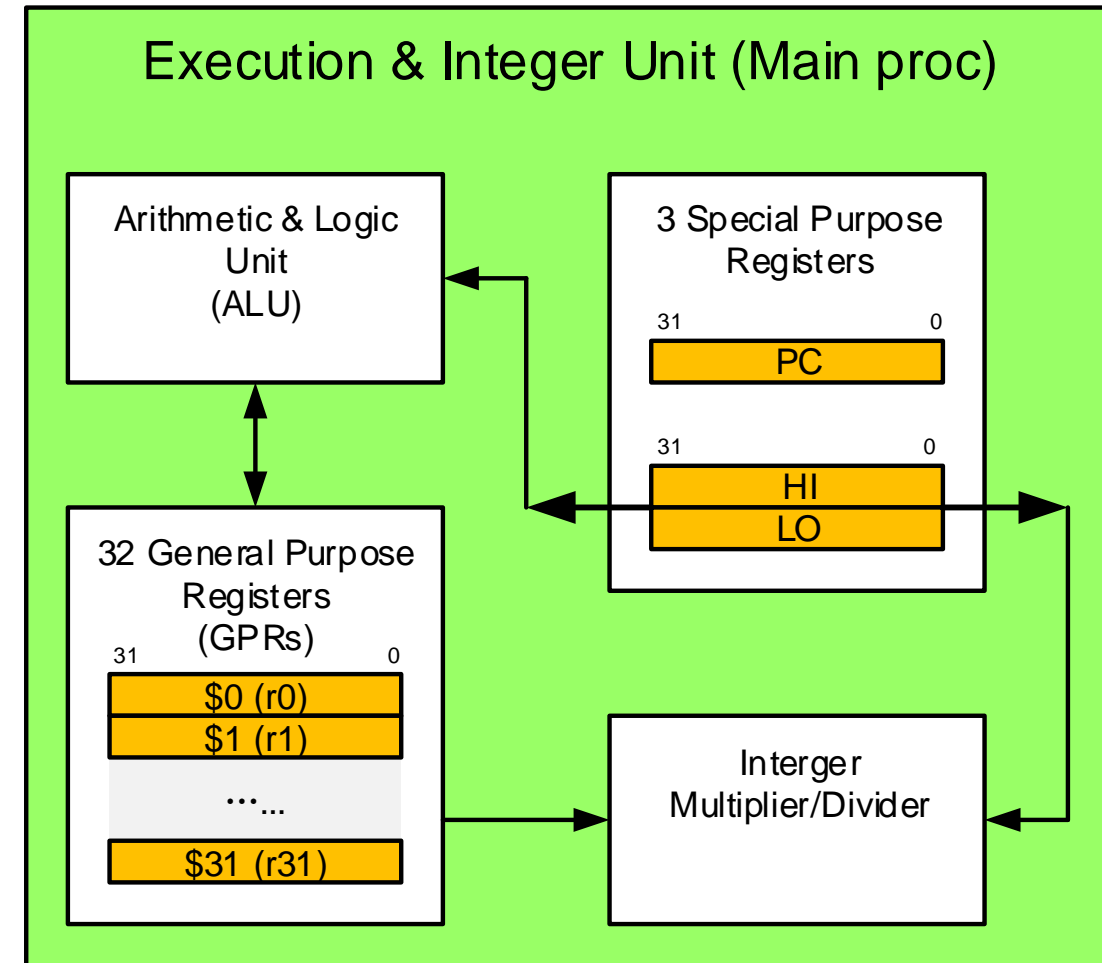
- \$0
- \$1
- ...
- \$31

- 特殊寄存器（3个）

- PC
- HI
- LO

- MIPS32中都是32位

- MIPS没有状态寄存器。CPU或内部都不包含任何用户程序计算结果的状态信息。



► CPU：通过执行指令，完成运算、控制

- 通用寄存器（32个）
 - \$0 ~ \$31
 - 汇编器既可以采用数字0-31加前缀\$的方式来表示：\$0-\$31；也可以采用名字加前缀\$的方式来表示：\$zero、\$at、\$v0、\$t0，此时汇编器会将名字转为对应的编号值。

【汇编例子】：

```
addu    $v0, $zero, $zero    # v0 = 0
addu    $v0, $0, $0          # v0 = 0
                                # 以上两指令功能相同

addu    $a0, $a0, $a1        # a0 = a0 + a1
addu    $4, $4, $5           # a0 = a0 + a1
                                # 以上两指令功能相同
```

在使用寄存器的时候，要尽量用这些约定名或助记符，而不直接引用寄存器编号

\$0 = \$zero	\$16 = \$s0
\$1 = \$at	\$17 = \$s1
\$2 = \$v0	\$18 = \$s2
\$3 = \$v1	\$19 = \$s3
\$4 = \$a0	\$20 = \$s4
\$5 = \$a1	\$21 = \$s5
\$6 = \$a2	\$22 = \$s6
\$7 = \$a3	\$23 = \$s7
\$8 = \$t0	\$24 = \$t8
\$9 = \$t1	\$25 = \$t9
\$10 = \$t2	\$26 = \$k0
\$11 = \$t3	\$27 = \$k1
\$12 = \$t4	\$28 = \$gp
\$13 = \$t5	\$29 = \$sp
\$14 = \$t6	\$30 = \$fp
\$15 = \$t7	\$31 = \$ra



- ▶ CPU：通过执行指令，完成运算、控制
 - 通用寄存器（32个）
 - 汇编器既可以采用数字0-31加前缀\$的方式来表示：\$0-\$31；也可以采用名字加前缀\$的方式来表示：\$zero、\$at、\$v0、\$t0，此时汇编器会将名字转为对应的编号值。

寄存器	编号	助记符	用法	在函数调用时是否需要保存
\$0	0	\$zero	永远为0	n/a
\$1	1	\$at	用做汇编器的暂时变量	
\$2 ~ \$3	2 ~ 3	\$v0 ~ \$v1	子函数调用返回结果	no
\$4 ~ \$7	4 ~ 7	\$a0 ~ \$a3	子函数调用的参数	no
\$8 ~ \$15	8 ~ 15	\$t0 ~ \$t7	暂时变量，被子函数使用时不需要保存与恢复	no
\$24 ~ \$25	24 ~ 25	\$t8 ~ \$t9		
\$16 ~ \$23	16 ~ 23	\$s0 ~ \$s7	子函数寄存器变量。子函数如果要用到这些寄存器，必须在使用前保存并在返回之前恢复寄存器的值，从而使这些寄存器的值对于调用函数没有变化	yes
\$26 ~ \$27	26 ~ 27	\$k0 ~ \$k1	OS内核使用，通常被中断或异常处理程序使用	
\$28	28	\$gp	全局指针。可用于方便的存取static和extern变量	yes
\$29	29	\$sp	堆栈指针	yes
\$30	30	\$fp	框架指针	yes
\$31	31	\$ra	子函数的返回地址	yes

t0-t9, s0-s7寄存器在汇编编程时使用较多。



CPU寄存器回顾

► CPU：通过执行指令，完成运算、控制

• 通用寄存器（32个）

▪ \$zero / \$0

- Register \$0 is always zero.
- Any value written to \$0 is discarded.

【汇编例子】：

```
add    $s0, $0, $0      # s0 = 0
add    $t0, $0, $0      # t0 = 0
ori     $s6, $0, 0x1000
```

▪ \$at / \$1

- 汇编器保留，由汇编器生成的复合指令使用。
- 当必须明确地使用它(如存入数据或从异常处理中恢复寄存器值)时，有一个汇编directive可以阻止编译器隐式地使用它(这样会有一些汇编宏指令不能用)。

Register	Number	Name	Usage
\$0	0	\$zero	always equal to 0 (forced by hardware)
\$1	1	\$at	assembler temporary; used by the assembler
\$2 ~ \$3	2 ~ 3	\$v0 ~ \$v1	return value from a function call
\$4 ~ \$7	4 ~ 7	\$a0 ~ \$a3	arguments (first four parameters) for a function call
\$8 ~ \$15	8 ~ 15	\$t0 ~ \$t7	temporary values (need not be preserved)
\$16 ~ \$23	16 ~ 23	\$s0 ~ \$s7	saved registers (preserved across call)
\$24 ~ \$25	24 ~ 25	\$t8 ~ \$t9	more temporary values
\$26 ~ \$27	26 ~ 27	\$k0 ~ \$k1	reserved for OS kernel
\$28	28	\$gp	global pointer (points to global data)
\$29	29	\$sp	stack pointer (points to top of stack)
\$30	30	\$fp	frame pointer (points to stack frame)
\$31	31	\$ra	return address (used by jal for function call)

► CPU：通过执行指令，完成运算、控制

- 通用寄存器（32个）
 - \$v0 ~ \$v1 / \$2 ~ \$3
 - 用来存放一个子程序(函数)的非浮点运算的结果或返回值。如果这两个寄存器不够存放需要返回的值，编译器将会通过内存来完成。
 - \$a0 ~ \$a3 / \$4 ~ \$7
 - 用来传递子函数调用时前4个非浮点参数。

【汇编例子】：

```
...  
sw    $a0, 0($fp)    #保存参数  
lw    $v0, 0($fp)    #装入a0
```

Register	Number	Name	Usage
\$0	0	\$zero	always equal to 0 (forced by hardware)
\$1	1	\$at	assembler temporary; used by the assembler
\$2 ~ \$3	2 ~ 3	\$v0 ~ \$v1	return value from a function call
\$4 ~ \$7	4 ~ 7	\$a0 ~ \$a3	arguments (first four parameters) for a function call
\$8 ~ \$15	8 ~ 15	\$t0 ~ \$t7	temporary values (need not be preserved)
\$16 ~ \$23	16 ~ 23	\$s0 ~ \$s7	saved registers (preserved across call)
\$24 ~ \$25	24 ~ 25	\$t8 ~ \$t9	more temporary values
\$26 ~ \$27	26 ~ 27	\$k0 ~ \$k1	reserved for OS kernel
\$28	28	\$gp	global pointer (points to global data)
\$29	29	\$sp	stack pointer (points to top of stack)
\$30	30	\$fp	frame pointer (points to stack frame)
\$31	31	\$ra	return address (used by jal for function call)

【汇编例子】：

```
...  
li    $v0, 1          #返回1  
j      L1              #跳转到代码并返回  
...  
L1:                                #结果保存到寄存器$ v0  
sw    $ra, 20($sp)    #保存寄存器$ ra  
sw    $fp, 16($sp)    #保存寄存器$ fp
```



► CPU：通过执行指令，完成运算、控制

• 通用寄存器（32个）

- \$t0 ~ t9 / \$8 ~ \$15、\$24 ~ \$25
 - 依照约定，一个子函数可以不用保存并随便的使用这些寄存器。
 - 在作表达式计算时，这些寄存器是非常好的暂时变量。
 - 注意：当调用一个子函数时，这些寄存器中的值有可能被子函数破坏掉。如果想保持不被改变，需要通过堆栈保存、恢复。

【汇编例子】：

```
...  
addu    $t0, $t0, $t1      #保存参数
```

Register	Number	Name	Usage
\$0	0	\$zero	always equal to 0 (forced by hardware)
\$1	1	\$at	assembler temporary; used by the assembler
\$2 ~ \$3	2 ~ 3	\$v0 ~ \$v1	return value from a function call
\$4 ~ \$7	4 ~ 7	\$a0 ~ \$a3	arguments (first four parameters) for a function call
\$8 ~ \$15	8 ~ 15	\$t0 ~ \$t7	temporary values (need not be preserved)
\$16 ~ \$23	16 ~ 23	\$s0 ~ \$s7	saved registers (preserved across call)
\$24 ~ \$25	24 ~ 25	\$t8 ~ \$t9	more temporary values
\$26 ~ \$27	26 ~ 27	\$k0 ~ \$k1	reserved for OS kernel
\$28	28	\$gp	global pointer (points to global data)
\$29	29	\$sp	stack pointer (points to top of stack)
\$30	30	\$fp	frame pointer (points to stack frame)
\$31	31	\$ra	return address (used by jal for function call)



► CPU：通过执行指令，完成运算、控制

- 通用寄存器（32个）

 - \$s0 ~ \$s7 / \$16 ~ \$23

 - 依照约定，子函数必须保证当函数返回时这些寄存器的内容必须恢复到函数调用以前的值，
 - 或者在子函数里不用这些寄存器或把它们保存在堆栈上并在函数退出时恢复。
 - 这种约定使得这些寄存器非常适合作为寄存器变量、或存放一些在函数调用期间必须保存的原来的值。

【汇编例子】：

```
...  
add    $s1, $0, $0  
add    $s2, $0, $s1  
addiu  $s3, $0, 1  
lw     $s4, 0($a0)
```

Register	Number	Name	Usage
\$0	0	\$zero	always equal to 0 (forced by hardware)
\$1	1	\$at	assembler temporary; used by the assembler
\$2 ~ \$3	2 ~ 3	\$v0 ~ \$v1	return value from a function call
\$4 ~ \$7	4 ~ 7	\$a0 ~ \$a3	arguments (first four parameters) for a function call
\$8 ~ \$15	8 ~ 15	\$t0 ~ \$t7	temporary values (need not be preserved)
\$16 ~ \$23	16 ~ 23	\$s0 ~ \$s7	saved registers (preserved across call)
\$24 ~ \$25	24 ~ 25	\$t8 ~ \$t9	more temporary values
\$26 ~ \$27	26 ~ 27	\$k0 ~ \$k1	reserved for OS kernel
\$28	28	\$gp	global pointer (points to global data)
\$29	29	\$sp	stack pointer (points to top of stack)
\$30	30	\$fp	frame pointer (points to stack frame)
\$31	31	\$ra	return address (used by jal for function call)



► CPU：通过执行指令，完成运算、控制

- 通用寄存器（32个）

 - \$k0 ~ \$k1 / \$26 ~ \$27

 - 被OS的异常或中断处理程序使用。被使用后不会恢复原来的值。因此它们很少在别的地方被使用。

 - \$gp / \$28

 - 如果存在一个全局指针，它将指向运行时决定的静态数据(static data)区域的一个位置。这意味着，利用gp作基指针，在gp指针32K左右的数据存取，系统只需要一条指令就可完成。
 - 如果没有全局指针，存取一个静态数据区域的值需要两条指令：
 - 一条是获取有由编译器和loader决定好的32位的地址常量。
 - 另外一条是对数据的真正存取。
 - 为了使用\$gp, 编译器在编译时刻必须知道一个数据是否在\$gp的64K范围之内。
 - 并不是所有的编译和运行系统支持gp的使用。

Register	Number	Name	Usage
\$0	0	\$zero	always equal to 0 (forced by hardware)
\$1	1	\$at	assembler temporary; used by the assembler
\$2 ~ \$3	2 ~ 3	\$v0 ~ \$v1	return value from a function call
\$4 ~ \$7	4 ~ 7	\$a0 ~ \$a3	arguments (first four parameters) for a function call
\$8 ~ \$15	8 ~ 15	\$t0 ~ \$t7	temporary values (need not be preserved)
\$16 ~ \$23	16 ~ 23	\$s0 ~ \$s7	saved registers (preserved across call)
\$24 ~ \$25	24 ~ 25	\$t8 ~ \$t9	more temporary values
\$26 ~ \$27	26 ~ 27	\$k0 ~ \$k1	reserved for OS kernel
\$28	28	\$gp	global pointer (points to global data)
\$29	29	\$sp	stack pointer (points to top of stack)
\$30	30	\$fp	frame pointer (points to stack frame)
\$31	31	\$ra	return address (used by jal for function call)



► CPU：通过执行指令，完成运算、控制

- 通用寄存器（32个）

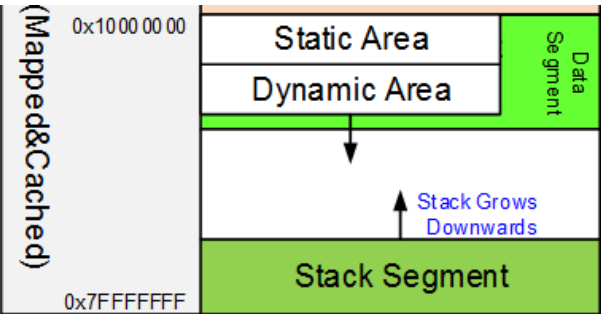
 - \$sp / \$29

 - MIPS通常只在子函数进入和退出的时刻才调整堆栈的指针。这通过被调用的子函数来实现。

 - MIPS的堆栈延伸方向：

 - 入栈：往小地址

 - 出栈：往大地址



 - MIPS没有专门的入栈、出栈指令，而是通过存储器访问指令 sw 和 lw 来完成，另外，堆栈指针也不会自动修改，需要用户通过算术指令修改。

Register	Number	Name	Usage
\$0	0	\$zero	always equal to 0 (forced by hardware)
\$1	1	\$at	assembler temporary; used by the assembler
\$2 ~ \$3	2 ~ 3	\$v0 ~ \$v1	return value from a function call
\$4 ~ \$7	4 ~ 7	\$a0 ~ \$a3	arguments (first four parameters) for a function call
\$8 ~ \$15	8 ~ 15	\$t0 ~ \$t7	temporary values (need not be preserved)
\$16 ~ \$23	16 ~ 23	\$s0 ~ \$s7	saved registers (preserved across call)
\$24 ~ \$25	24 ~ 25	\$t8 ~ \$t9	more temporary values
\$26 ~ \$27	26 ~ 27	\$k0 ~ \$k1	reserved for OS kernel
\$28	28	\$gp	global pointer (points to global data)
\$29	29	\$sp	stack pointer (points to top of stack)
\$30	30	\$fp	frame pointer (points to stack frame)
\$31	31	\$ra	return address (used by jal for function call)

【汇编例子】：

```
.text
main:
    sw    $ra, 0($sp)      #入栈，保存返回地址
    sw    $s0, 4($sp)      #s0入栈保存
    subu   $sp, $sp, 8      #修改堆栈指针，往小地址
    # addi   $sp, $sp, -8    #修改堆栈指针的另一种方法
```



► CPU：通过执行指令，完成运算、控制

- 通用寄存器（32个）

 - \$fp / \$30

 - 另外的约定名是s8
 - fp作为框架指针，可以被过程用来记录堆栈的情况，在一个过程中变量相对于框架指针的偏移量是不变的。一些编程语言显式的支持这一点。汇编程序员经常会利用fp的这个用法。C语言的库函数alloca()就是利用了fp来动态调整堆栈。
 - 注意：如果堆栈的底部在编译时刻不能被决定，你就不能通过\$sp 来存取堆栈变量，因此\$fp被初始化为一个相对于该函数堆栈的一个常量的位置。这种用法对其他函数可以是不可见的。

Register	Number	Name	Usage
\$0	0	\$zero	always equal to 0 (forced by hardware)
\$1	1	\$at	assembler temporary; used by the assembler
\$2 ~ \$3	2 ~ 3	\$v0 ~ \$v1	return value from a function call
\$4 ~ \$7	4 ~ 7	\$a0 ~ \$a3	arguments (first four parameters) for a function call
\$8 ~ \$15	8 ~ 15	\$t0 ~ \$t7	temporary values (need not be preserved)
\$16 ~ \$23	16 ~ 23	\$s0 ~ \$s7	saved registers (preserved across call)
\$24 ~ \$25	24 ~ 25	\$t8 ~ \$t9	more temporary values
\$26 ~ \$27	26 ~ 27	\$k0 ~ \$k1	reserved for OS kernel
\$28	28	\$gp	global pointer (points to global data)
\$29	29	\$sp	stack pointer (points to top of stack)
\$30	30	\$fp	frame pointer (points to stack frame)
\$31	31	\$ra	return address (used by jal for function call)



- ▶ CPU：通过执行指令，完成运算、控制
 - 通用寄存器（32个）
 - \$ra / \$31
 - 永远存放着正常函数调用指令(jal)的返回地址；
 - 当调用任何一个子函数时，返回地址存放在ra寄存器中，因此通常一个子程序的最后一个指令是：jr ra.

【汇编例子】：

```
...
addi    $a0, $0, 3
jal     A          #调用子函数
move    $s1, $a0   #伪指令
...
A :                # 函数A
...
jr $ra
...
```

Register	Number	Name	Usage
\$0	0	\$zero	always equal to 0 (forced by hardware)
\$1	1	\$at	assembler temporary; used by the assembler
\$2 ~ \$3	2 ~ 3	\$v0 ~ \$v1	return value from a function call
\$4 ~ \$7	4 ~ 7	\$a0 ~ \$a3	arguments (first four parameters) for a function call
\$8 ~ \$15	8 ~ 15	\$t0 ~ \$t7	temporary values (need not be preserved)
\$16 ~ \$23	16 ~ 23	\$s0 ~ \$s7	saved registers (preserved across call)
\$24 ~ \$25	24 ~ 25	\$t8 ~ \$t9	more temporary values
\$26 ~ \$27	26 ~ 27	\$k0 ~ \$k1	reserved for OS kernel
\$28	28	\$gp	global pointer (points to global data)
\$29	29	\$sp	stack pointer (points to top of stack)
\$30	30	\$fp	frame pointer (points to stack frame)
\$31	31	\$ra	return address (used by jal for function call)

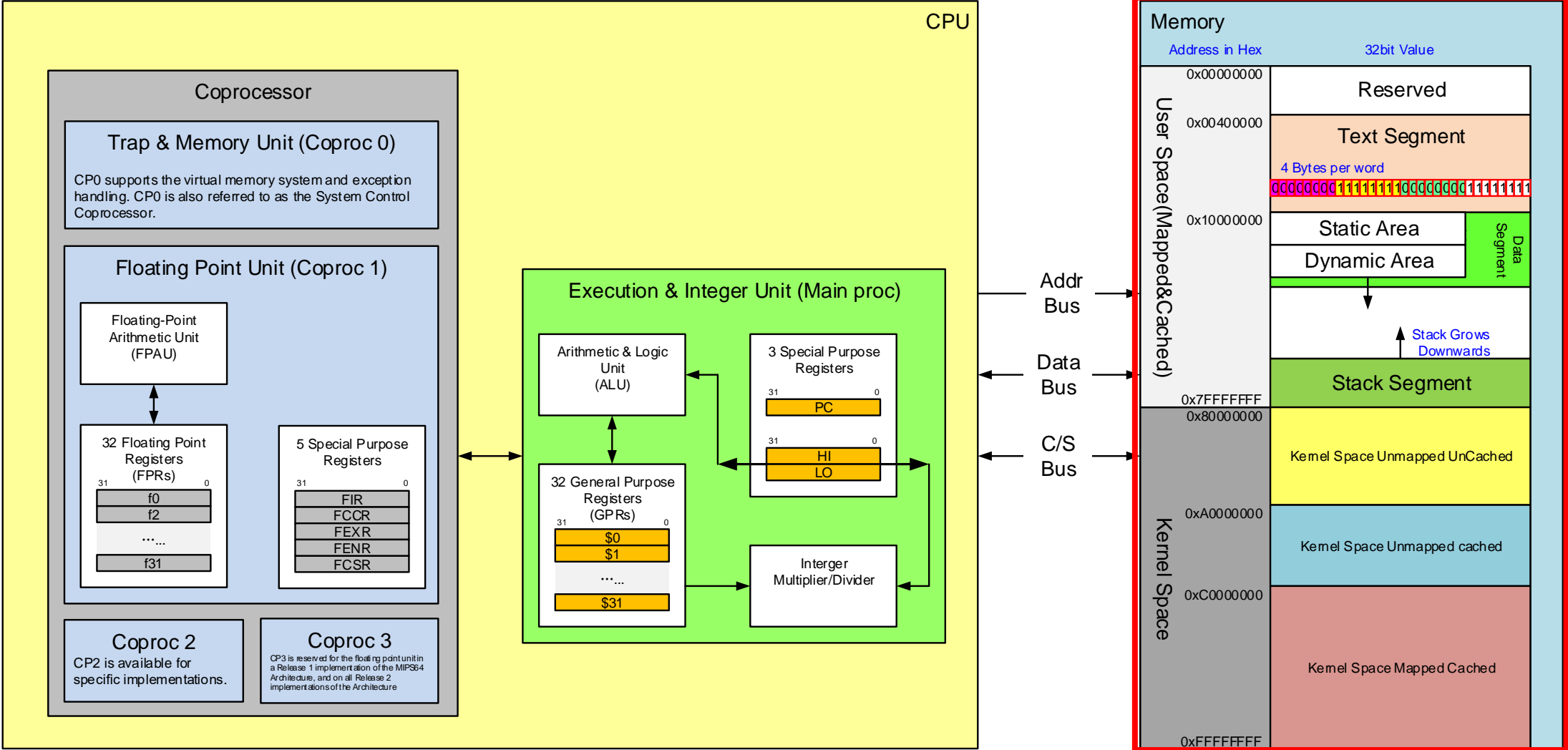
【汇编例子】：

```
.text
main:
    sw    $ra, 0($sp)      #保存返回地址
    subu  $sp, $sp, 4      #修改堆栈指针
#   addi  $sp, $sp, -8     #修改堆栈指针的另一种方法
```

- 子函数如果还要调用其他的子函数，必须保存ra的值，通常通过堆栈。



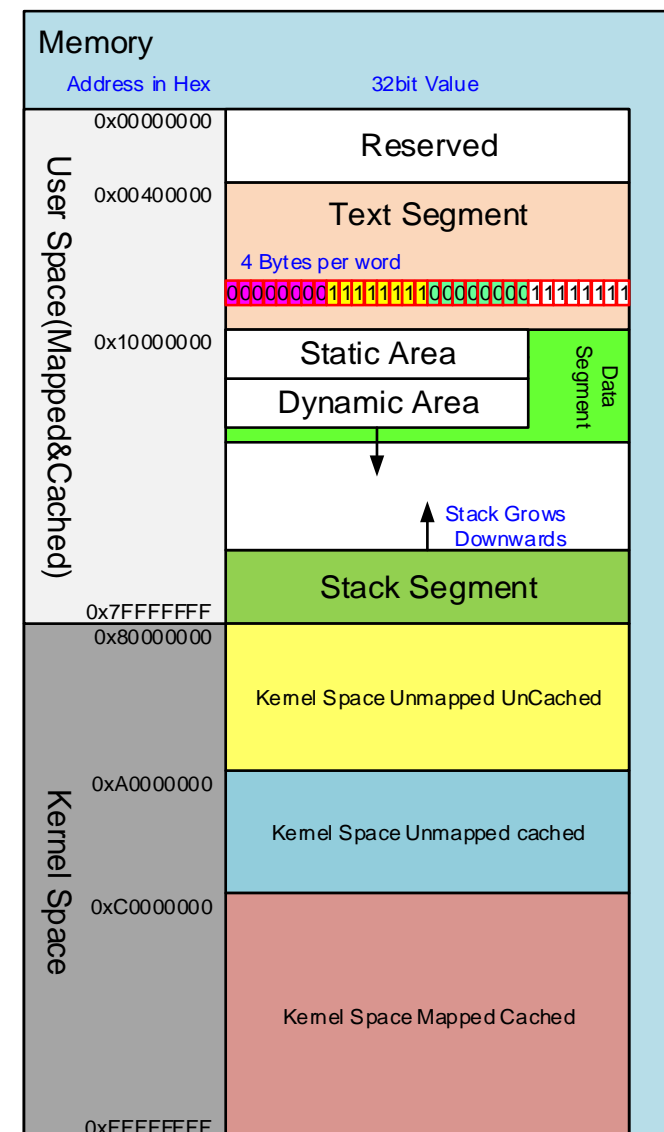
▶ 回顾：CPU和存储器



存储器回顾

► Memory：存储指令机器码、数据

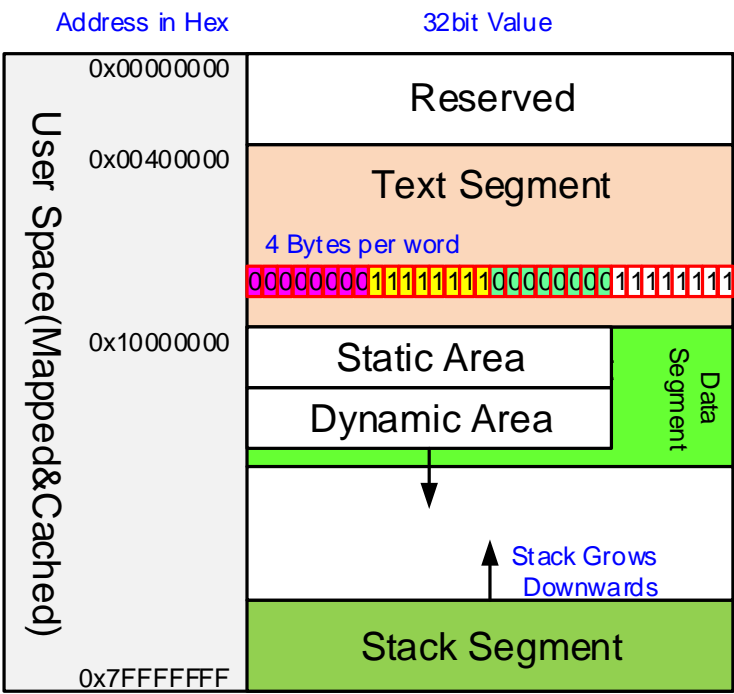
- 每四个字节构成一个字，MIPS中使用字寻址，相邻数据字的字地址相差4，总共有 2^{30} 个存储字：
 - 存储器[0]、存储器[2]、...、存储器[$2^{30}-1$]
- MIPS只能使用数据传送指令访问存储器。
- 存储器分段：在32位MIPS体系结构下，最多可寻址4GB地址空间。这4GB空间的分配是怎样的呢？
 - 内核空间（2GB：0x8000 0000 ~ 0xFFFF FFFF）
 - 用户空间（2GB：0x0000 0000 ~ 0x7FFF FFFF）
 - 代码段（0x0040 0000开始）
 - 数据段
 - 静态数据（0x1000 0000开始）
 - 动态数据：静态后是由C中的malloc分配的动态数据，向上增长
 - 堆栈段（0x7FFF FFFF开始）
 - 从顶端开始，对栈指针初始化为7ffffff，并向向下向数据段增长



► 汇编程序框架

【汇编例子】MIPS汇编程序框架

```
# Title:                               Filename:
# Author:                             Date:
# Description:
# Input:
# Output:
##### Data segment #####
.data
...
##### Code segment #####
.text
.globl main                          # 声明全局变量main
main:                               # main program entry
...
li $v0, 10                          # Exit program
syscall
```



If a program is loaded into QtSpim, its **.text** segment is automatically placed at **0x00400000**, its **.data** segment at **0x10000000**



► 框架中的伪指令

- **.data** directive
 - Defines the data segment of a program containing data.
 - The program's variables should be defined under this directive.
 - Assembler will allocate and initialize the storage of variables.
- **.text** directive
 - Defines the code segment of a program containing instructions.
- **.globl** directive
 - Declares a symbol as global.
 - Global symbols can be referenced from other files.
 - We use this directive to declare main procedure of a program.

【汇编例子】MIPS汇编程序框架

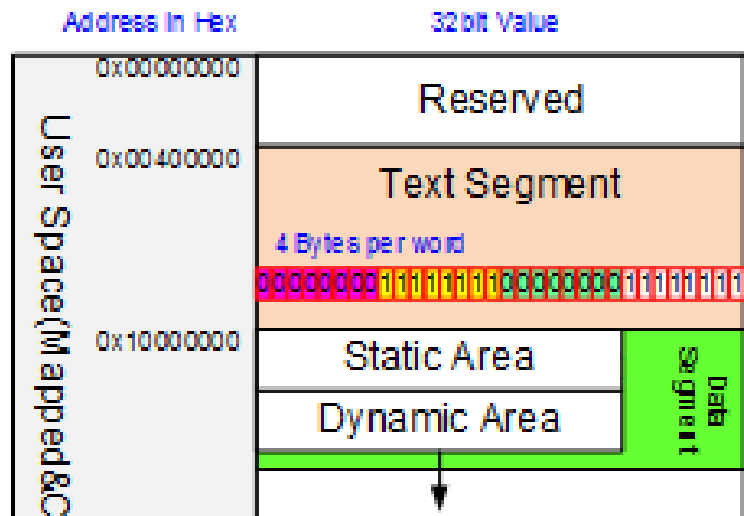
```
# Title:                               Filename:
# Author:                             Date:
# Description:
# Input:
# Output:
##### Data segment #####
.data
...
##### Code segment #####
.text
.globl main                            # 声明全局变量main
main:                                 # main program entry
...
li $v0, 10                             # Exit program
syscall
```

► 框架中的伪指令

• 数据段 (.data指定) 的数据定义

- 格式 : [name:] **directive** initializer [, initializer] ...
- 例 : var1: **.WORD** 10(初始值)
- data directive

- **.byte** : The program's variables should Stores the list of values as 8-bit bytes (用8位的字节存储列表的值)
- **.half** : Stores the list as 16-bit values **aligned on half-word boundary**
- **.word** : Stores the list as 32-bit values **aligned on a word boundary**
- **.float** : Stores the listed values as single-precision floating point
- **.double** : Stores the listed values as double-precision floating point
- **.ascii** : Allocates a sequence of bytes for an ASCII string (为ASCII字符串 分配字节序列)
- **.asciiz** : Same as .ASCII directive, but adds a **NULL char (0x00)** at end of string(Strings are null-terminated, as in the C programming language)
- **.space** : Allocates space of n uninitialized bytes in the data segment(在数据段中, 分配n个未初始化的字节空间)



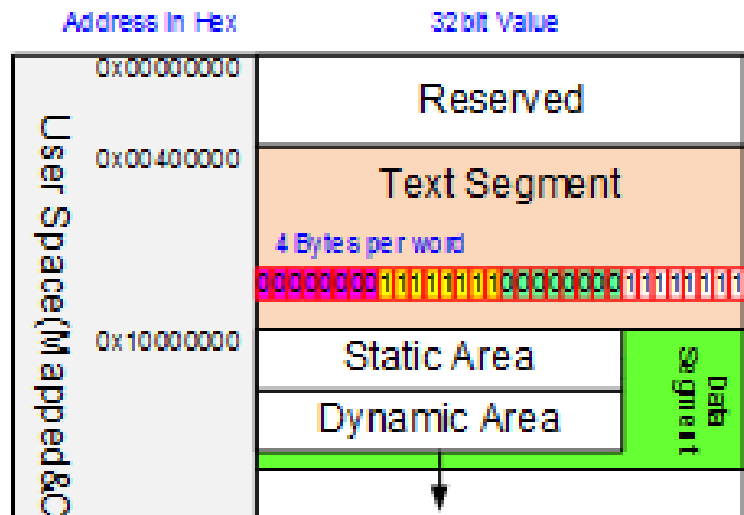
► 框架中的伪指令

【汇编例子】数据定义举例

Data segment

.data

```
var1:    .byte      'A', 'E', 127, -1, '1', '2', '3'
var2:    .half      -10, 0xfffe
var3:    .word      0x12345678:10
var4:    .float      12.3, -0.1
var5:    .double     12.3
str1:    .ascii     "A String\n"
str2:    .asciiz     "NULL Terminated String"
array:   .space      10
```



每四个字节，按照
Little-endian方式存储

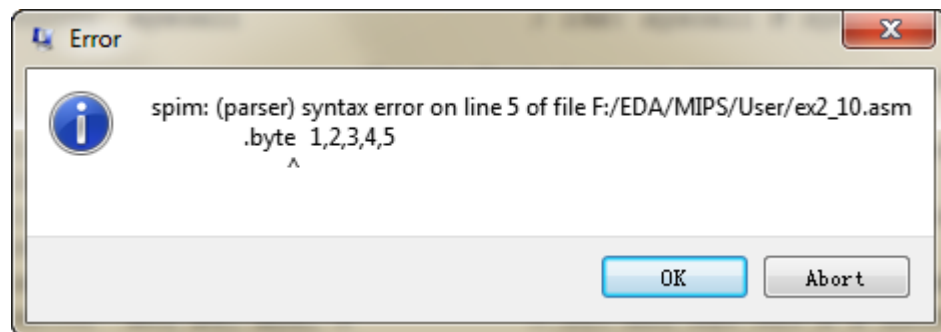
MIPS processors can operate with **either big-endian or little-endian** byte order. SPIM operates with both byte orders. SPIM's byte order **is the same as the byte order of the underlying machine that runs the simulator**. For example, **on a Intel 80x86**, SPIM is **little-endian**, while on a Macintosh or Sun SPARC, SPIM is bigendian.

Data	Text
Data	
User data segment [10000000]..[10040000]	
[10000000]..[1000ffff] 00000000	
[10010000] ff7f4541 00333231 fffefff6 12345678	A E . . 1 2 3 x V 4 .
[10010010] 12345678 12345678 12345678 12345678	x V 4 . x V 4 . x V 4 . x V 4 .
[10010020] 12345678 12345678 12345678 12345678	x V 4 . x V 4 . x V 4 . x V 4 .
[10010030] 12345678 4144cccd bdcccccd 00000000	x V 4 . . . D A
[10010040] 9999999a 40289999 74532041 676e6972 (@ A S t r i n g
[10010050] 4c554e0a 6554204c 6e696d72 64657461	. N U L L T e r m i n a t e d
[10010060] 72745320 00676e69 00000000 00000000	S t r i n g
[10010070]..[1003ffff] 00000000	


- ▶ 实验目的
- ▶ 实验任务及要求
- ▶ 汇编程序结构
 - CPU回顾
 - 存储器回顾
 - 汇编程序框架
- ▶ QtSpim汇编软件
 - QtSpim简介
 - Qtspim系统功能调用
 - QtSpim使用示例

► QtSpim软件

- 运行在windows操作系统下的支持MIPS32指令集的MIPS微处理器仿真器，具备调试、运行MIPS32汇编指令程序的功能，可直接打开.s、.asm、.txt汇编源文件
- 软件下载：<http://sourceforge.net/projects/spimsimulator/files/>
 - 建议QtSpim_9.1.17或QtSpim_9.1.12版本
 - 不建议使用QtSpim_9.1.16（莫名错误）























- 不支持直接装载二进制文件
- 不支持在线编辑，需要用另外的文字编辑软件，如UltraEdit等编辑汇编源文件。

 **spim mips simulator**
Brought to you by: jameslarus

Summary Files Reviews Support Wiki Code Tickets ▾

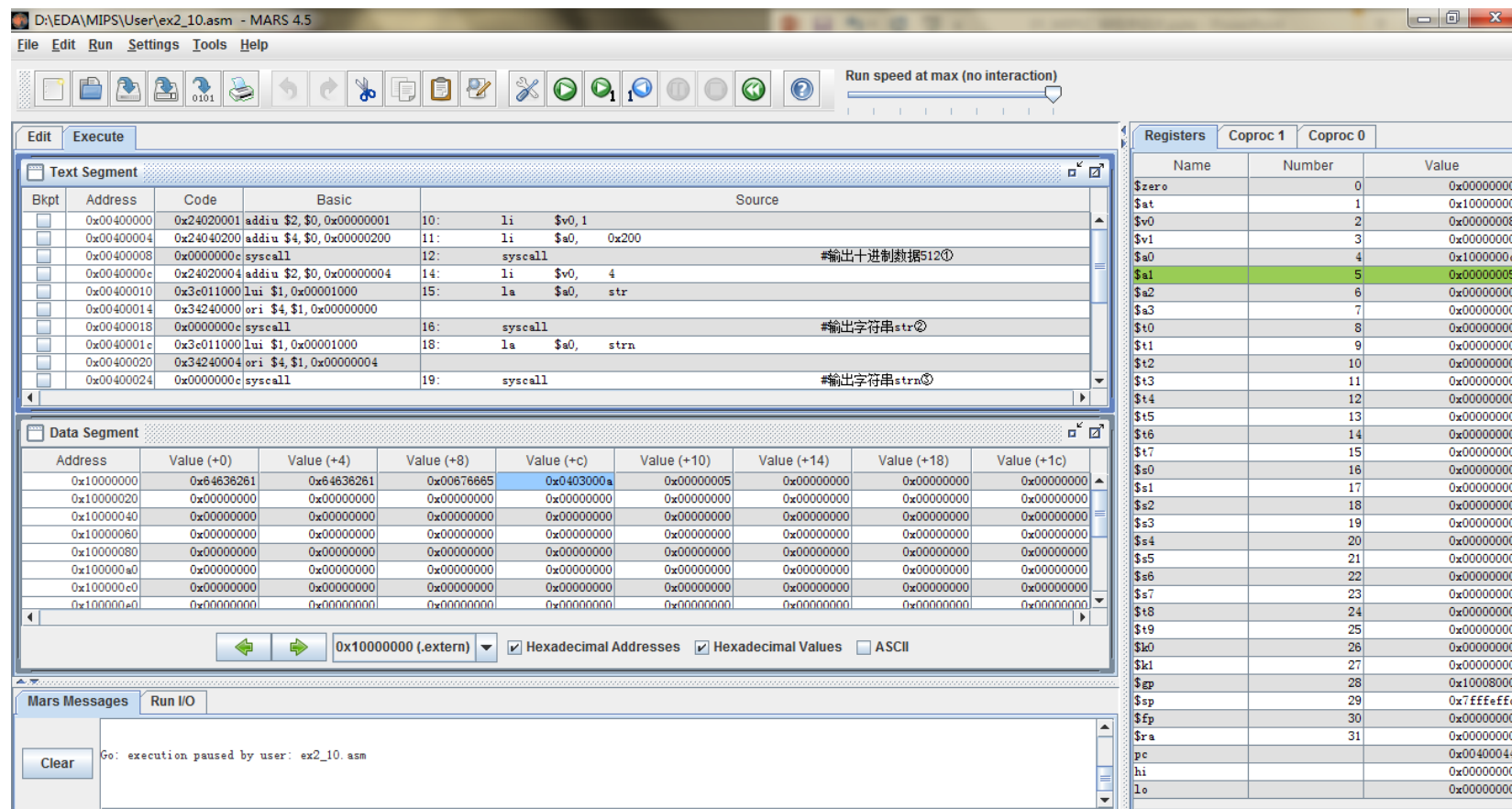
Looking for the latest version? [Download QtSpim_9.1.17_Windows.exe \(34.3 MB\)](#)

Home

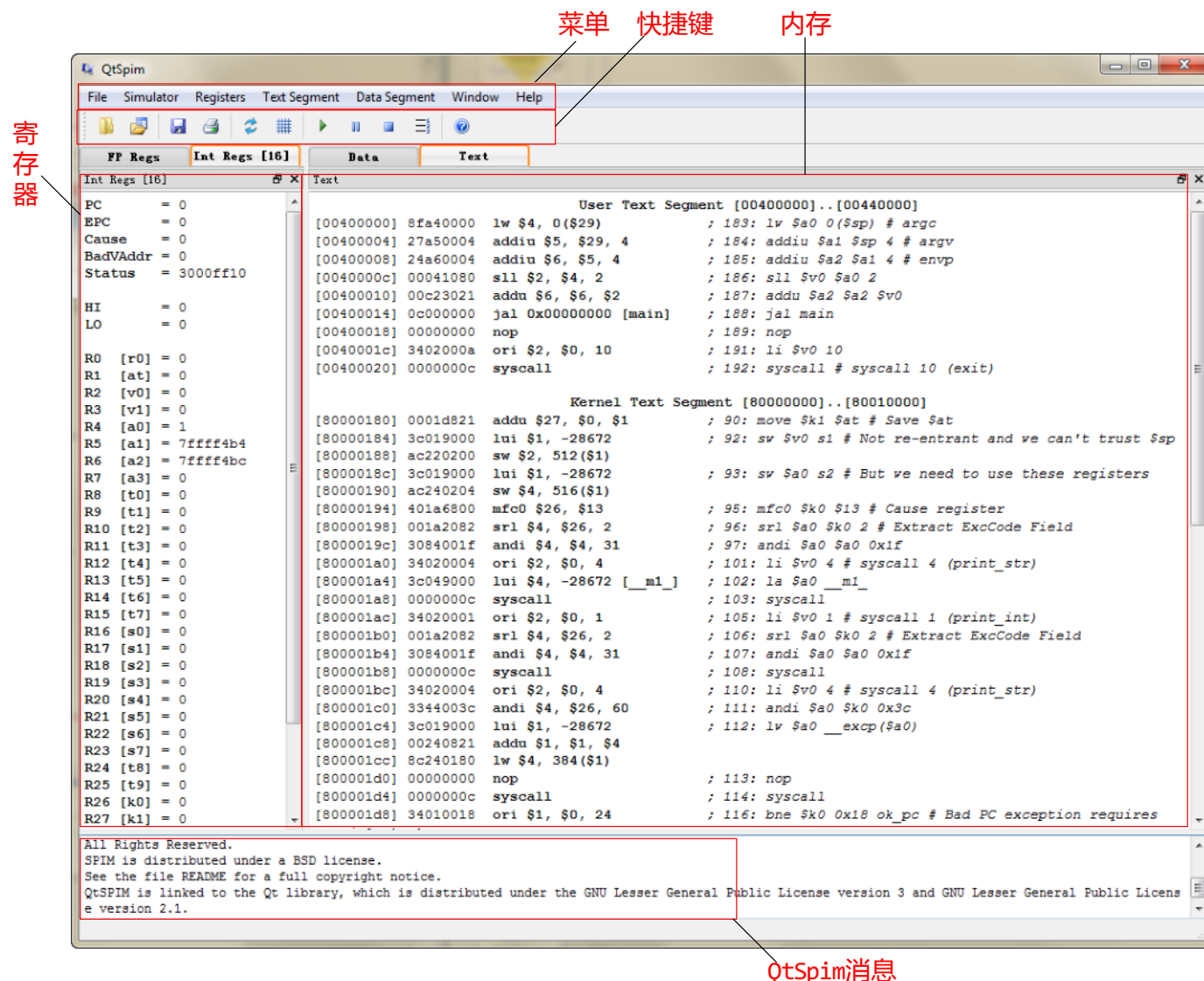
Name ▴	Modified ▴	Size ▴	Downloads / Week ▴
QtSpim_9.1.17_Windows.exe	2015-12-30	34.3 MB	1,249  
qtspim_9.1.17_linux32.deb	2015-12-30	30.8 MB	75  
qtspim_9.1.17_linux64.deb	2015-12-30	29.1 MB	226  
QtSpim_9.1.17_mac.mpkg.zip	2015-12-30	28.9 MB	639  
qtspim_9.1.16_linux32.deb	2015-08-26	29.7 MB	1  
QtSpim_9.1.16_mac.mpkg.zip	2015-08-26	28.7 MB	36  
qtspim_9.1.16_linux64.deb	2015-08-26	28.1 MB	4  
QtSpim_9.1.16_Windows.exe	2015-08-26	34.3 MB	101  
QtSpim_9.1.15_mac.mpkg.zip	2015-04-26	28.7 MB	8  
QtSpim_9.1.13_mac.mpkg.zip	2014-02-05	33.1 MB	2  

► QtSpim软件

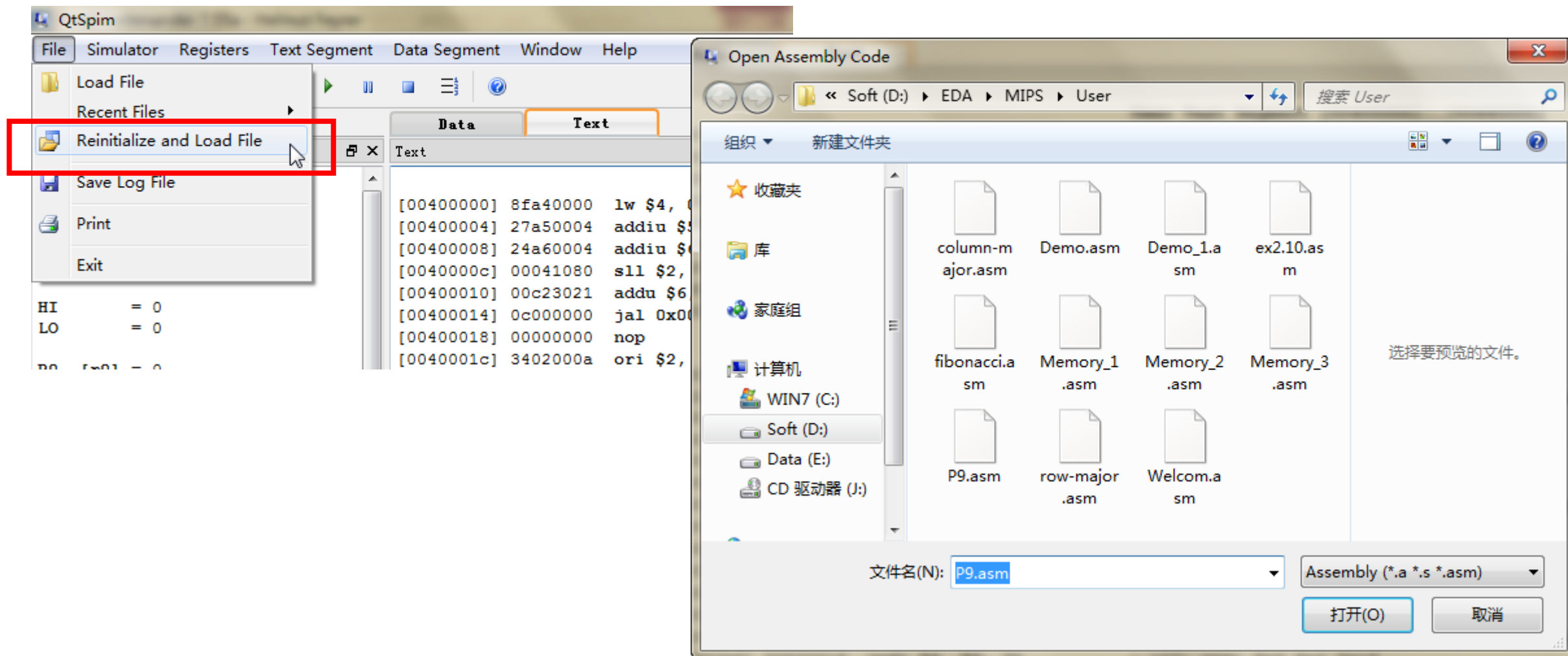
- 软件不支持在线编辑，需要用另外的文字编辑软件，如UltraEdit等。
- 推荐Mars软件
 - Editor
 - MIPS编译器、仿真器
 - 视频：mars安装.exe



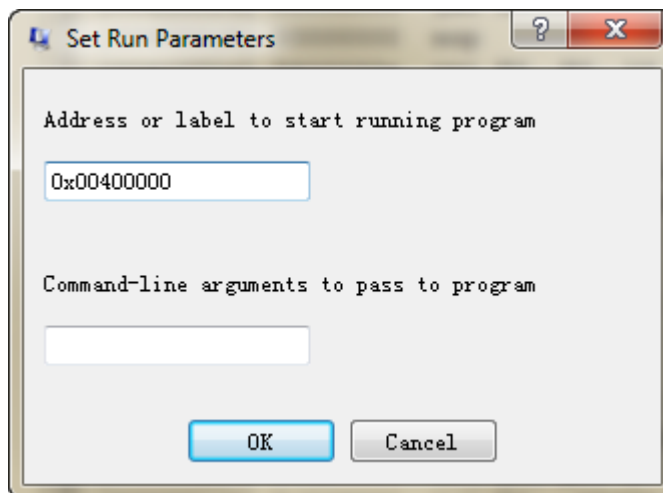
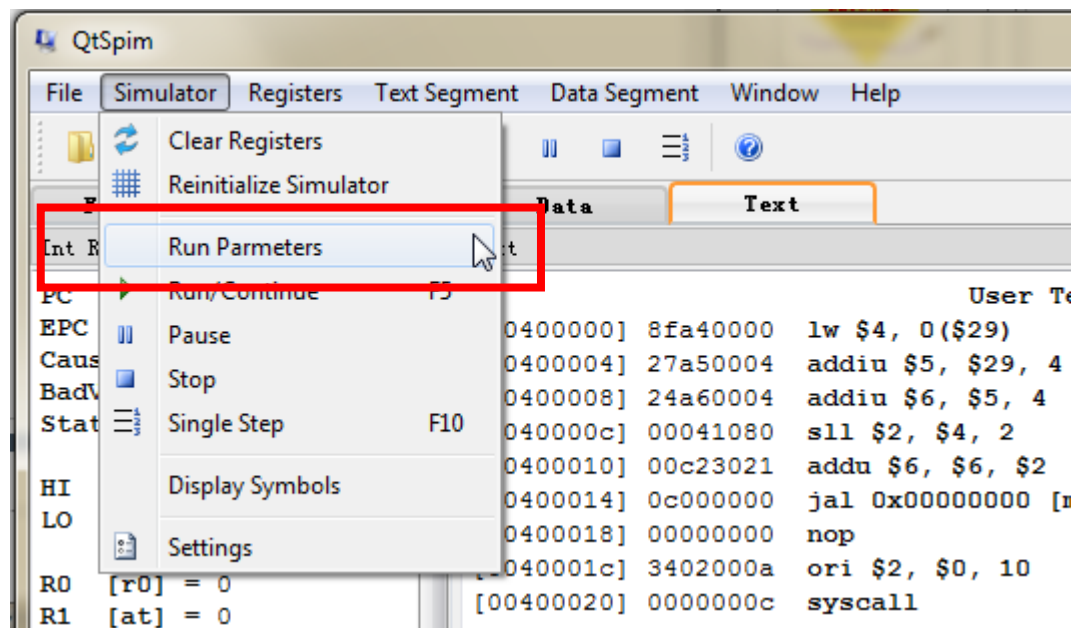
► Qspim界面



► File菜单



► Simulator菜单



► QtSpim系统功能调用

功能描述	功能号 (\$v0)	输入参数	输出参数
显示整数	1	\$a0: 整数值	
显示字符串直到字符串结束符 0	4	\$a0: 字符串首地址	
读入整数	5		\$v0: 输入的整数值
读入字符串	8	\$a0: 内存空间首地址 \$a1: 内存空间长度	
退出	10		

• 步骤

- 将功能号赋给 \$v0 ;
- 设置好入口参数 ;
- 执行 syscall 指令 ;
- 处理输出参数。

【汇编例子】：输出显示256

```
...  
li      $v0, 1          # 送功能号  
li      $a0, 256        # 入口参数  
syscall          # 显示整数256  
...  
li      $v0, 10         # 送功能号  
syscall          # 退出
```

QtSpim系统功能调用

► QtSpim系统功能调用

• 步骤

- 将功能号赋给\$V0 ;
- 设置好入口参数 ;
- 执行 syscall 指令 ;
- 处理输出参数。

【汇编例子】：输出显示字符串 “Hellow world!”

```
.data
Str      .asciiZ      "Hellow world!"
...
li        $V0, 4          # 送功能号
la        $a0, Str        # 入口参数
syscall                   # 显示字符串
```

【思考1】：怎么输出回车换行符 \n ?

【思考2】：.asciiZ改成.ascii，程序运行可能会出现什么情况？为什么？

```
Str      .ascii      "Hellow world!\0"
```

SPIM System Services

Service	Number	Arguments	Return Value
print integer	1	\$a0 (integer)	
print float	2	\$f12 (float)	
print double	3	\$f12 (double)	
print string	4	\$a0 (address of string)	
read integer	5		\$V0 (integer)
read float	6		\$f0 (float)
read double	7		\$f0 (float)
read string	8	\$a0 (address of buffer) \$a1 (bytes allocated for string)	
sbrk	9	\$a0 (integer amount)	
exit	10		

【思考1】：怎么输出回车换行符 \n ?

```
Str      .asciiZ      "\nHellow world!\n"
...
li        $V0, 4          # 送功能号
la        $a0, Str        # 入口参数
syscall                   # 显示字符串
```



常用汇编指令

- 加、减：
add、sub
- 存储器的字读、写：
lw、sw
- 与、或：
and、or
- 相等则转：
beq
- 小于设置：
slt
- 跳转：
j
- 子程序调用、返回：
jal、jr \$ra
- 系统功能调用：
syscall

常用宏指令指令

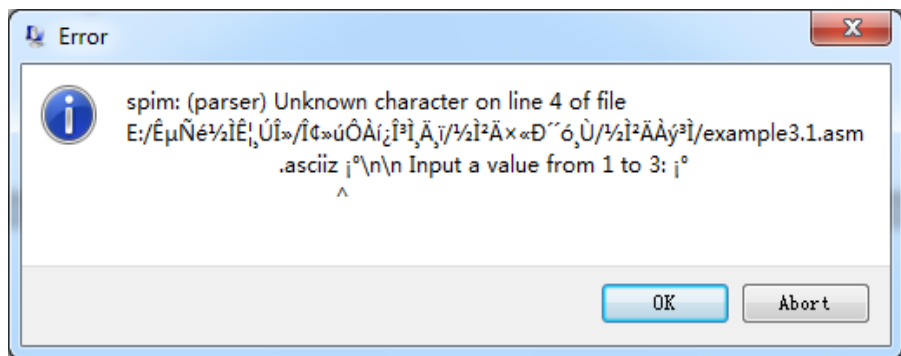
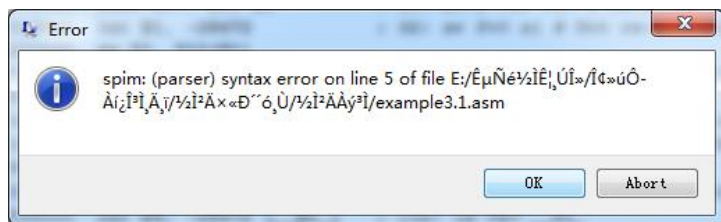
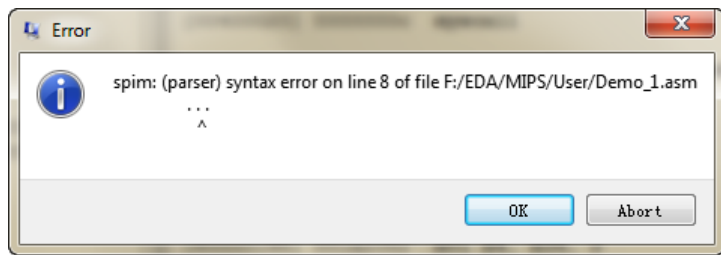
- 给Rd赋值：
li Rd, value
- 取地址到Rd：
la Rd, Label

类型	指令	指令举例	含义	备注
算术运算	加法	add \$s1,\$s2,\$s3	\$s1=\$s2+\$s3	三个寄存器操作数
	减法	sub \$s1,\$s2,\$s3	\$s1=\$s2-\$s3	三个寄存器操作数
	加立即数	addi \$s1,\$s2,20	\$s1=\$s2+20	用来加立即数
数据传送	读取字	lw \$s1,20(\$s2)	\$s1=mem[\$s2+20]	从内存读字到寄存器
	存储字	sw \$s1,20(\$s2)	mem[\$s2+20] = \$s1	从寄存器写字到内存
	读取半字	lh \$s1,20(\$s2)	\$s1=mem[\$s2+20]	从内存读半字到寄存器
	读取无符号半字	lhu \$s1,20(\$s2)	\$s1=mem[\$s2+20]	从内存读半字到寄存器
	存储半字	sh \$s1,20(\$s2)	mem[\$s2+20] = \$s1	从寄存器写半字到内存
	读取字节	lb \$s1,20(\$s2)	\$s1=mem[\$s2+20]	从内存读字节到寄存器
	读取无符号字节	lbu \$s1,20(\$s2)	\$s1=mem[\$s2+20]	从内存读字节到寄存器
	存储字节	sb \$s1,20(\$s2)	mem[\$s2+20] = \$s1	从寄存器写字节到内存
	读取链接字	ll \$s1,20(\$s2)	\$s1= mem[\$s2+20]	读字作为原子交换的第一半
	条件存储字	sc \$s1,20(\$s2)	mem[\$s2+20] =s\$1;\$s1=0或1	写字作为原子交换的第二半
	读取立即数到高半字	lui \$s1,20	\$s1=20*2 ¹⁶	读取一个常数到高16位
逻辑操作	与	and \$s1,\$s2,\$s3	\$s1=\$s2&\$s3	三个寄存器，位与
	或	or \$s1,\$s2,\$s3	\$s1=\$s2 \$s3	三个寄存器，位或
	或非	nor \$s1,\$s2,\$s3	\$s1=~(\$s2 \$s3)	三个寄存器，位或非
	与立即数	andi \$s1,\$s2,20	\$s1=\$s2&20	寄存器与立即数位与
	或立即数	ori \$s1,\$s2,20	\$s1=\$s2 20	寄存器与立即数位或
	逻辑左移	sll \$s1,\$s2,10	\$s1=\$s2<<10	左移常数次
	逻辑右移	srl \$s1,\$s2,10	\$s1=\$s2>>10	右移常数次
条件跳转	相等转移	beq \$s1,\$s2,25	If (\$s1=\$s2) goto PC+4+25*4	相等测试，转移
	不相等转移	bne \$s1,\$s2,25	If (\$s1!= \$s2) goto PC+4+25*4	不相等测试，转移
	小于设置	slt \$s1,\$s2,\$s3	If(\$s2<\$s3) \$s1=1 else \$s1=0	比较小于设置\$s1=1
	低于设置	sltu \$s1,\$s2,\$s3	If(\$s2<\$s3) \$s1=1 else \$s1=0	比较低于设置\$s1=1
无条件跳转	小于常数设置	slti \$s1,\$s2,20	If(\$s2<20) \$s1=1 else \$s1=0	和常数比较小于设置\$s1=1
	低于常数设置	sltiu \$s1,\$s2,20	If(\$s2<20) \$s1=1 else \$s1=0	和常数比较低于设置\$s1=1
	直接跳转	j 2500	goto 2500*4	跳转到目标地址
	间接跳转	jr \$ra	goto \$ra	用在分支和子程序返回
系统功能调用	跳转并链接	jal 2500	\$ra=PC+4; goto 2500*4	用在子程序调用
	系统功能调用	syscall		实现人机对话



QtSpim实验示例

- ▶ 编程举例
- ▶ 装载汇编源程序时的错误信息提示



不支持：

- 1) 源文件中有**中文全角**字符；
- 2) 源文件、路径都**不支持汉字**；
- 3) ...

```

1  .data 0x10000000
2  .align 2
3  str: .ascii "abcd"
4  strn: .asciiz "ABCDEFGH"
5  b0: .byte 1,2,3,4,5
6
7  .globl main
8  .text
9  main:
10     li $v0, 1
11     li $a0, 0x200           #输出十进制数据512①
12     syscall
13
14     li $v0, 4
15     la $a0, str
16     syscall                #输出字符串str②
17
18     la $a0, strn
19     syscall                #输出字符串strn③
20
21     li $v0, 5
22     syscall                #输入十进制整数④
23
24     li $v0, 8
25     la $a0, b0
26     li $a1, 5
27     syscall                #输入字符串⑤
28
29     li $v0, 10
30     syscall                #程序结束退出

```



► QtSpim用户程序入口

QtSpim自动添加的一段入口代码：获取入口参数，调用用户程序。

DataText

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] 34020001 ori \$2, \$0, 1 ; 10: li \$v0,1

[00400028] 34040200 ori \$4, \$0, 512 ; 11: li \$a0, 0x200

[0040002c] 0000000c syscall ; 12: syscall # 0x000000000000000051200

[00400030] 34020004 ori \$2, \$0, 4 ; 14: li \$v0, 4

[00400034] 3c041000 lui \$4, 4096 [str] ; 15: la \$a0, str

[00400038] 0000000c syscall ; 16: syscall # 0x0000000000000000str00

[0040003c] 3c011000 lui \$1, 4096 [strn] ; 18: la \$a0, strn

[00400040] 34240004 ori \$4, \$1, 4 [strn]

[00400044] 0000000c syscall ; 19: syscall # 0x0000000000000000strn00

[00400048] 34020005 ori \$2, \$0, 5 ; 21: li \$v0, 5

[0040004c] 0000000c syscall ; 22: syscall # 0x00000000000000000000

[00400050] 34020008 ori \$2, \$0, 8 ; 24: li \$v0, 8

[00400054] 3c011000 lui \$1, 4096 [b0] ; 25: la \$a0, b0

[00400058] 3424000c ori \$4, \$1, 12 [b0]

[0040005c] 34050005 ori \$5, \$0, 5 ; 26: li \$a1, 5

[00400060] 0000000c syscall ; 27: syscall # 0x00000000000000000000

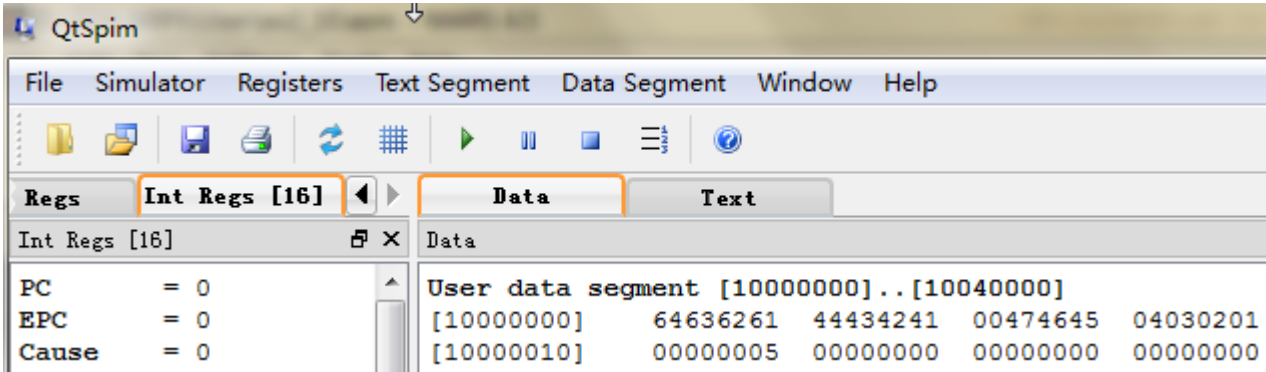
[00400064] 3402000a ori \$2, \$0, 10 ; 29: li \$v0, 10

[00400068] 0000000c syscall ; 30: syscall # 0x00000000000000000000

Kernel Text Segment [80000000]..[80010000]

```
1 .data 0x10000000
2 .align 2
3 str: .ascii "abcd"
4 strn: .asciiz "ABCDEFGG"
5 b0: .byte 1,2,3,4,5
6
7 .globl main
8 .text
9 main:
10     li $v0,1
11     li $a0, 0x200 #输出十进制数据512①
12     syscall
13
14     li $v0, 4
15     la $a0, str #输出字符串str②
16     syscall
17
18     la $a0, strn #输出字符串strn③
19     syscall
20
21     li $v0, 5 #输入十进制整数④
22     syscall
23
24     li $v0, 8
25     la $a0, b0
26     li $a1, 5 #输入字符串⑤
27     syscall
28
29     li $v0, 10 #程序结束退出
30     syscall
```


- ▶ 编程举例
- ▶ 数据段内存映像
 - QtSpim软件仿真结果



```
1  .data 0x10000000
2  .align 2
3  str: .ascii "abcd"
4  strn: .asciiz "ABCDEFGH"
5  b0: .byte 1,2,3,4,5
6
7  .globl main
8  .text
9  main:
10     li $v0, 1
11     li $a0, 0x200
12     syscall #输出十进制数据512①
13
14     li $v0, 4
15     la $a0, str
16     syscall #输出字符串str②
17
18     la $a0, strn
19     syscall #输出字符串strn③
20
21     li $v0, 5
22     syscall #输入十进制整数④
23
24     li $v0, 8
25     la $a0, b0
26     li $a1, 5
27     syscall #输入字符串⑤
28
29     li $v0, 10
30     syscall #程序结束退出
```

- Mars软件仿真结果

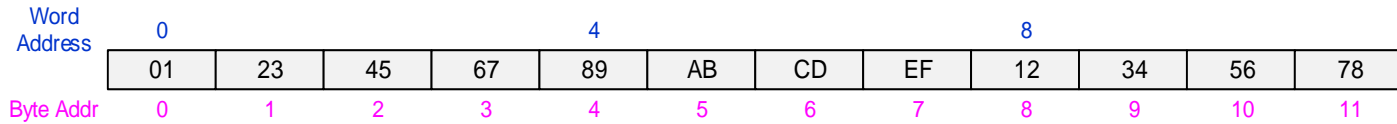
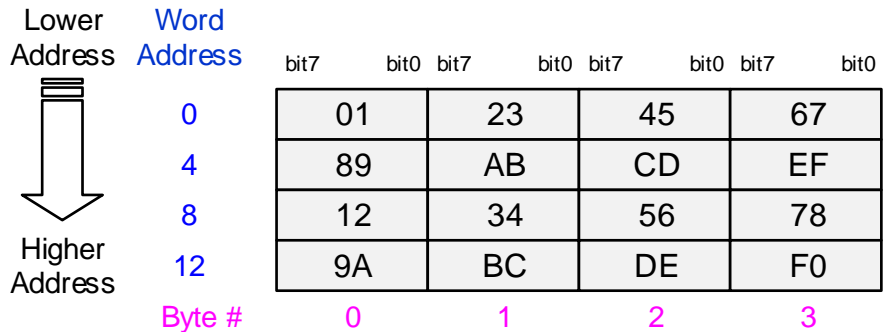
Data Segment						
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)
0x10000000	0x64636261	0x44434241	0x00474645	0x04030201	0x00000005	0x00000000
0x10000020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000



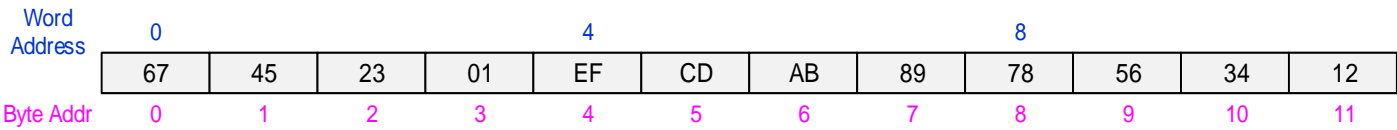
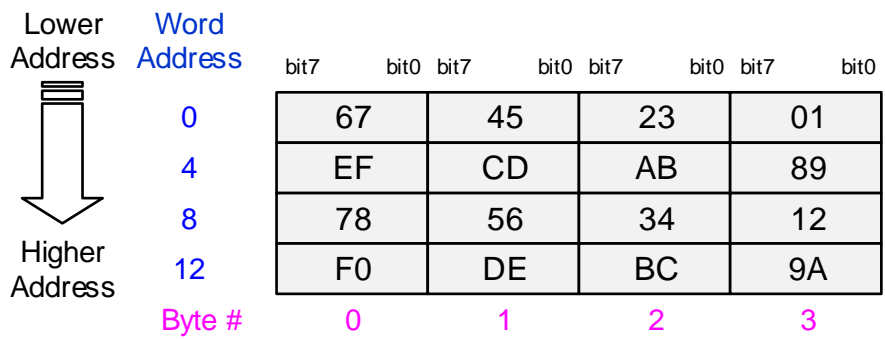
数的存储

- 0x01234567、0x89ABCDEF、0x12345678、0x9ABCDEF0存入0、4、8、12地址
- **Big-endian**：高对低、低对高；顺序存放；

MIPS R3000 processor capabilities, such as cache size, multiprocessor interface, "big-endian" or "little-endian" byte ordering, can be configured immediately after processor reset



- **Little-endian**：高对高、低对低；如：PC



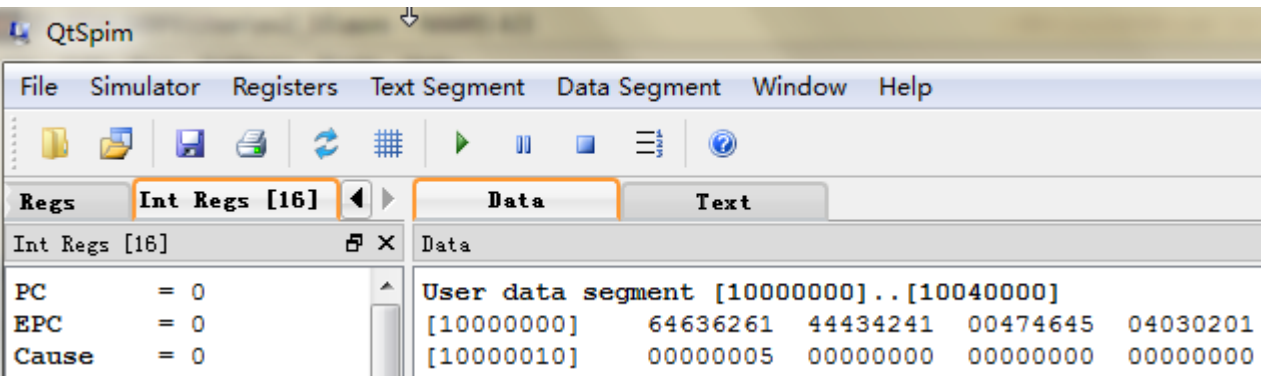
MIPS processors can operate with either *big-endian* or *little-endian* byte order. QtSpim operates with both byte orders. QtSpim's byte order is the same as the byte order of the underlying machine that runs the simulator. For example, on a Intel 80x86, QtSpim is little-endian, while on a Macintosh or Sun SPARC, QtSpim is big-endian.

▶ 数据段内存映像

```
1 .data 0x10000000
2 .align 2
3 str: .ascii "abcd"
4 strn: .asciiz "ABCDEFGH"
5 b0: .byte 1,2,3,4,5
6
```

数据段内存映像需要画成这种格式！

• QtSpim软件仿真结果



• Mars软件仿真结果

Data Segment						
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)
0x10000000	0x64636261	0x44434241	0x00474645	0x04030201	0x00000005	0x00000000
0x10000020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

变量名	地址	数据	定义值
<u>str</u>	0x 1000 0000	0x64	"d"
	0x 1000 0001	0x63	"c"
	0x 1000 0002	0x62	"b"
	0x 1000 0003	0x61	"a"
<u>strn</u>	0x 1000 0004	0x44	"D"
	0x 1000 0005	0x43	"C"
	0x 1000 0006	0x42	"B"
	0x 1000 0007	0x41	"A"
	0x 1000 0008	0x00	"\0"
	0x 1000 0009	0x47	"G"
	0x 1000 000A	0x46	"F"
	0x 1000 000B	0x45	"E"
b0	0x 1000 000C	0x04	4
	0x 1000 000D	0x03	3
	0x 1000 000E	0x02	2
	0x 1000 000F	0x01	1
	0x 1000 0010	0x00	
	0x 1000 0011	0x00	
	0x 1000 0012	0x00	
	0x 1000 0013	0x05	5

► 代码段内存映像

用户代码段的内存映像需要自己来画，不能简单截图！

内存地址 (16进制)	汇编指令	机器码 (16进制)
00400024	Ori \$2,\$0,1	34020001
...
...

系统调用代码

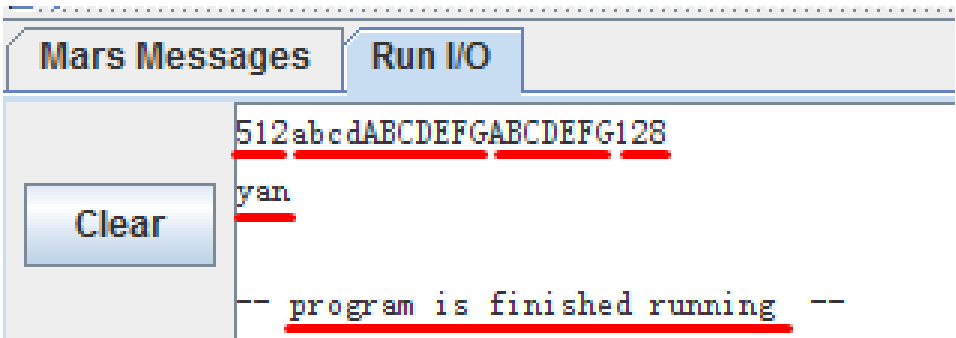
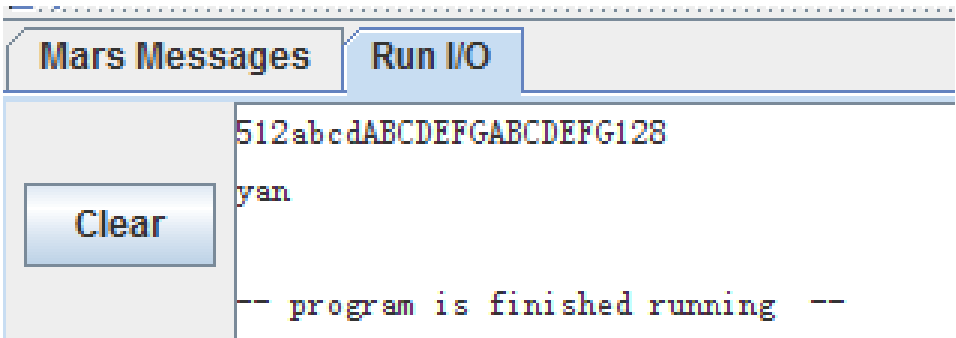
用户程序代码

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]	34020001 ori \$2, \$0, 1 ; 10: li \$v0,1		
[00400028]	34040200 ori \$4, \$0, 512 ; 11: li \$a0, 0x200		
[0040002c]	0000000c syscall ; 12: syscall #0000000000000000512000		
[00400030]	34020004 ori \$2, \$0, 4 ; 14: li \$v0, 4		
[00400034]	3c041000 lui \$4, 4096 [str] ; 15: la \$a0, str		
[00400038]	0000000c syscall ; 16: syscall #00000000_0000str00		
[0040003c]	3c011000 lui \$1, 4096 [strn] ; 18: la \$a0, strn		
[00400040]	34240004 ori \$4, \$1, 4 [strn] ; 19: syscall #00000000_0000strn00		
[00400044]	0000000c syscall ; 21: li \$v0, 5		
[00400048]	34020005 ori \$2, \$0, 5 ; 22: syscall #00000000000000000000		
[0040004c]	0000000c syscall ; 24: li \$v0, 8		
[00400050]	34020008 ori \$2, \$0, 8 ; 25: la \$a0, b0		
[00400054]	3c011000 lui \$1, 4096 [b0] ; 26: li \$a1, 5		
[00400058]	3424000c ori \$4, \$1, 12 [b0] ; 27: syscall #00000000_00000000		
[0040005c]	34050005 ori \$5, \$0, 5 ; 29: li \$v0, 10		
[00400060]	0000000c syscall ; 30: syscall #00000000000000000000		
[00400064]	3402000a ori \$2, \$0, 10		
[00400068]	0000000c syscall		
内存地址	机器码	反汇编后的汇编指令	用户宏汇编代码



▶ 程序运行情况（Step仿真）

▪ Mars软件仿真结果

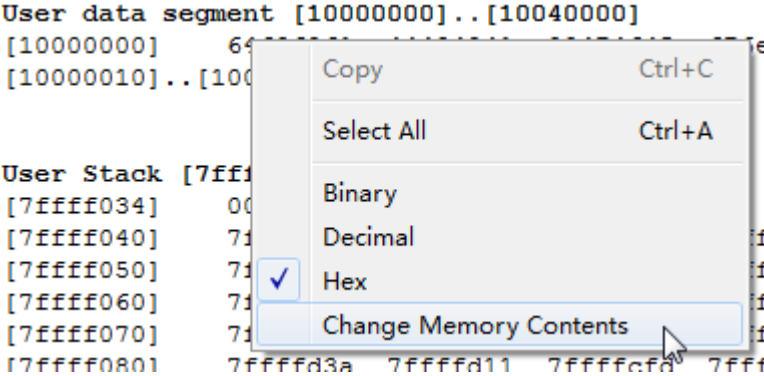
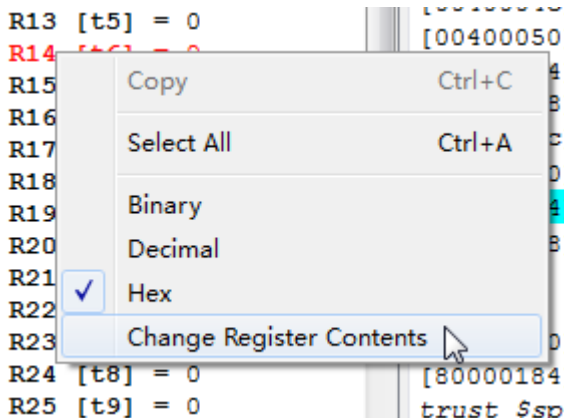
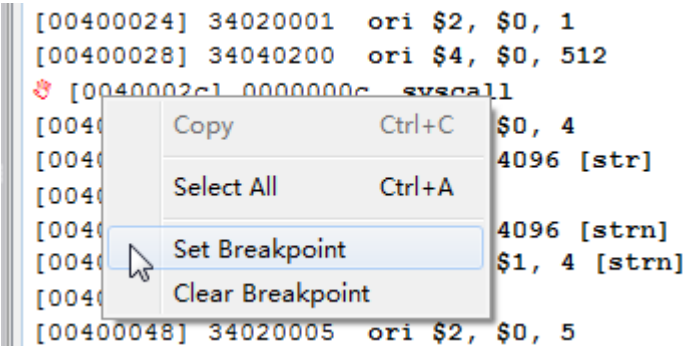
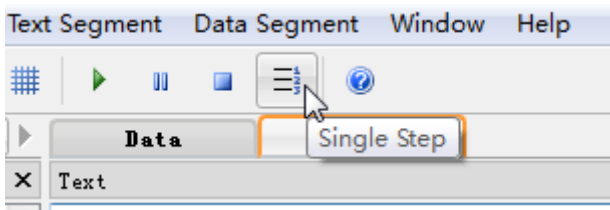
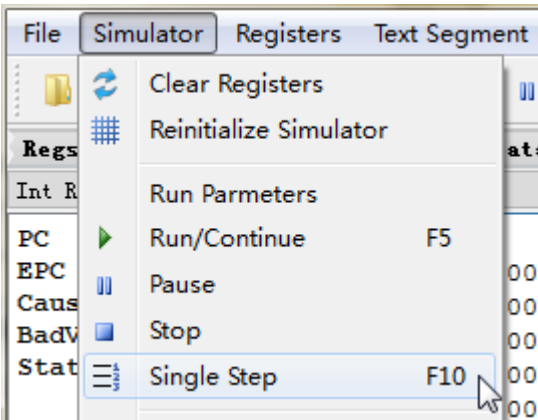


```
1 .data 0x10000000
2 .align 2
3 str: .ascii "abcd"
4 strn: .asciiz "ABCDEFGH"
5 b0: .byte 1,2,3,4,5
6
7 .globl main
8 .text
9 main:
10     li $v0, 1
11     li $a0, 0x200
12     syscall #输出十进制数据512①
13
14     li $v0, 4
15     la $a0, str
16     syscall #输出字符串str②
17
18     la $a0, strn
19     syscall #输出字符串strn③
20
21     li $v0, 5
22     syscall #输入十进制整数④
23
24     li $v0, 8
25     la $a0, b0
26     li $a1, 5
27     syscall #输入字符串⑤
28
29     li $v0, 10
30     syscall #程序结束退出
```



▶ 程序调试技巧

- 程序内存映像查看
 - 数据段内存映像
 - 代码段内存映像
 - 堆栈段内存映像
- Step执行代码
- 断点设置
 - 光标处鼠标右键
 - 快捷菜单
- 修改储存器/寄存器的值
 - 光标处鼠标右键
 - 快捷菜单



Thanks

