MIPS 体系结构和汇编语言快速入门

译者：Sonic Fu, Northeastern University, Boston, MA, USA

译者按：有修改，无删减，初学必读。学习笔记，抛砖引玉！网上有一个老版本，不如此版全面。

英文原版： http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm#IOSystemCalls

本文分 3 部分： 1、寄存器 2、程序结构框架 3、编写汇编程序

概要：数据类型和文法

- 数据类型：字节,byte 占用(8bit), halfword 占 2 byte= 16bit), word 占用（4byte = 32bit）
- 一个字符需要一个 Byte 的空间；
- 一个整数需要 1 个 Word（4 Byte）的空间；
- MIPS 结构的每条指令长度都是 32bit

寄存器

- MIPS 体系架构有 32 个通用寄存器。在汇编程序中，可以用编号 $0 到 $31 来表示；
- 也可以用寄存器的名字来进行表示， 例如： $sp, $t1, $ra….
- 有两个特殊的寄存器 Lo, Hi, 用来保存乘法/除法的运算结果；此 2 寄存器不能直接寻址，只能用特殊的指令：mfhi 和 mflo 来 aceess 其中的内容。
  （含义：mfhi = move from Hi, mflo = Move from Low.）
- 堆栈（Stack）的增长方向是: 从内存的高地址方向，向低地址方向；

表格 1：寄存器的编号名称及分类

编号	寄存器名称	寄存器描述
0	Zero	第 0 号寄存器，其值始终为 0
1	$at	(**A**ssembler **T**emporary) 是 Assembler 保留的寄存器
2 ~ 3	$v0 ~ $v1	(**v**alues)保存表达式或函数返回的结果
4-7	$a0 - $a3	(**a**rguments) 作为函数的前四个入参。在子函数调用的过程中不会被保留。
8-15	$t0 - $t7	(**t**emporaries) Caller saved if needed. Subroutines can use without saving.供汇编程序使用的临时寄存器。在子函数调用的过程中不会被保留。
16-23	$s0 - $s7	(**s**aved values) - Callee saved. A subroutine using one of these must save original and restore it before exiting. 在子函数调用的过程中会被保留。
24-25	$t8 - $t9	(**t**emporaries) Caller saved if needed. Subroutines can use without saving.供汇编程序使用的临时寄存器。在子函数调用的过程中不会被保留。这是对 $t0 - $t7 的补充。
26-27	$k0 - $k1	保留，仅供中断(interrupt/trap)处理函数使用.
28	$gp	**g**lobal **p**ointer. 全局指针。Points to the middle of the 64K block of memory in the static data segment.指向固态数据块内存的 64K 的块的中间。
29	$sp	**s**tack **p**ointer 堆栈指针， 指向堆栈的栈顶。
30	$s8/$fp	**s**aved value / **f**rame **p**ointer 保存的值/帧指针 其中的值在函数调用的过程中会被保留
31	$ra	**r**eturn **a**ddress 返回地址

汇编程序结构框架

汇编源程序代码本质上是文本文件。由 数据声明、代码段 两部分组成。汇编程序文件应该以.s 为后缀，以在 Spim 软件中进行模拟。（实际上 ASM 也行。）

数据声明部分

在源代码中，数据声明部分以 .data 开始。声明了在代码中使用的变量的名字。同时，也在主存（RAM）中创建了对应的空间。

程序代码部分

在源代码中，程序代码部分以 .text 开始。这部分包含了由指令构成的程序功能代码。代码以 main: 函数开始。main 的结束点应该调用 exit system call，参见后文有关 system call 的介绍。

程序的注释部分

使用#符号进行注释。每行以#引导的部分都被视作注释。

一个 MIPS 汇编程序框架：

```
# Comment giving name of program and description of function
# Template.s
# Bare-bones outline of MIPS assembly language program
.data              # variable declarations follow this line
                   # ...
.text              # instructions follow this line

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

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

编写 MIPS 汇编程序：

Content：

PartI: **数据的声明**

Part II： **数据的装载和保存（Load/Store 指令）**

Part III： **寻址**

Part IV： **算术运算指令：Arithmetic Instructions**

Part V **程序控制指令：Control Instructions**

Part VI: **系统调用和 I/O 操作（SPIM 仿真）**

PartI:数据的声明

格式：

```
name:      storage_type    value(s)
```

创建一个以 name 为变量名称，values 通常为初始值，storage_type 代表存储类型。
注意：变量名后要跟一个:冒号

```
example

var1:      .word   3       # create a single integer:
                           #variable with initial value 3

array1:    .byte   'a','b' # create a 2-element character
                           # array with elements initialized:
                           # to  a  and  b

array2:    .space  40      # allocate 40 consecutive bytes,
                           # with storage uninitialized
                           # could be used as a 40-element
                           # character array, or a
                           # 10-element integer array;
                           # a comment should indicate it.

string1    .asciiz "Print this.\n"        #declare a string
```

Part II：数据的装载和保存（Load/Store 指令）

- 主存（RAM）的存取 access 只能用 load / store 指令来完成。
- 所有其他的指令都使用的是寄存器作为操作数。

i. load 指令:

```
lw         register_destination, RAM_source
                                  # copy word (4 bytes) at
                                  # source_RAM location
                                  # to destination register.
                                  # load word -> lw

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
                                    # load byte -> lb

li          register_destination, value
                                    #load immediate value into
                                    #destination register
                                    #load immediate --> li
```

ii. store 指令

```
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
```

举个例子：

```
.data
var1:       .word   23              # declare storage for var1;
                                    #initial value is 23

.text
__start:
lw          $t0, var1               # load contents of RAM location
                                    # into register $t0:
                                    # $t0 = var1

li          $t1, 5                  # $t1 = 5   ("load immediate")
sw          $t1, var1               # store contents of register $t1
                                    # into RAM:  var1 = $t1 done
done
```

Part III：寻址 ：

MIPS 系统结构只能用 load/store 相关指令来实现寻址操作，包含 3 中寻址方式：
装载地址：load address，相当于直接寻址，把数据地址直接载入寄存器。

间接寻址：indirect addressing，间接寻址，把寄存器内容作为地址
基线寻址/索引寻址：based or indexed addressing，相对寻址，利用补偿值(offset)寻址。

直接寻址/装载地址：load address:

```
la          $t0, var1
```

把 var1 在主存（RAM）中的地址拷贝到寄存器 t0 中。var1 也可以是程序中定义的一个子程序标签的地址。

间接寻址：indirect addressing:

```
lw          $t2, ($t0)
```

主存中有一个字的地址存在 t0 中，按这个地址找到那个字，把字拷贝到寄存器 t2 中。

```
sw          $t2, ($t0)
```

把 t2 中的字存入 t0 中的地址指向的主存位置。

基线寻址/索引寻址：based or indexed addressing:

```
lw          $t2, 4($t0)
```

把 t0 中地址+4 所得的地址所对应的主存中的字载入寄存器 t2 中，4 为包含在 t0 中的地址的偏移量。

```
sw          $t2, -12($t0)           # offset can be negative
```

把 t2 中的内容存入 t0 中的地址-12 所得的地址所对应的主存中，存入一个字，占用 4 字节，消耗 4 个内存号，可见，地址偏移量可以是负值。

注意：基线寻址在以下场合特别有用：
1、数组：从基址出发，通过使用偏移量，存取数组元素。
2、堆栈：利用从堆栈指针或者框架指针的偏移量来存取元素。

举个例子：

```
#example
.data
array1:    .space  12              #  declare 12 bytes of storage
                                   # to hold array of 3 integers
.text
__start:
la          $t0, array1            #  load base address of array
                                   # into register $t0
li          $t1, 5                 #  $t1 = 5   ("load immediate")
```

```
sw          $t1, ($t0)              #  first array element set to 5;
                                    # indirect addressing
li          $t1, 13          #   $t1 = 13
sw          $t1, 4($t0)      # second array element set to 13

li          $t1, -7          #   $t1 = -7
sw          $t1, 8($t0)      # third array element set to -7

done
```

Part IV 算术运算指令：Arithmetic Instructions

- 算数运算指令的所有操作数都是寄存器，不能直接使用 RAM 地址或间接寻址。
- 操作数的大小都为 Word （4-Byte）

```
add         $t0,$t1,$t2      #  $t0 = $t1 + $t2;   add as signed
                             # (2's complement) integers
sub         $t2,$t3,$t4      #  $t2 = $t3 Ð $t4
addi        $t2,$t3, 5       #  $t2 = $t3 + 5;   "add immediate"
                             # (no sub immediate)
addu        $t1,$t6,$t7      #  $t1 = $t6 + $t7;
addu        $t1,$t6,5        #  $t1 = $t6 + 5;
                             #  add as unsigned integers
subu        $t1,$t6,$t7      #  $t1 = $t6 - $t7;
subu        $t1,$t6,5        #  $t1 = $t6 - 5
                             #  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
```

Part V 程序控制指令：Control Instructions

1. 分支指令（Branches）

条件分支的比较机制已经内建在指令中

```
b          target  #  unconditional branch to program label target
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
```

```
beqz       $t0, lab # Branch to lab if $t0 = 0.
bnez       $t0, lab # Branch to lab if $t0 != 0.
bgez       $t0, lab # Branch to lab if $t0 >= 0.
bgtz       $t0, lab # Branch to lab if $t0 > 0.
blez       $t0, lab # Branch to lab if $t0 <= 0.
bltz       $t0, lab # Branch to lab if $t0 < 0.
```

```
bgezal     $t0, lab #If $t0 >= 0, then put the address of the next
                    #instruction into $ra and branch to lab.
bgtzal     $t0, lab #If $t0 > 0, then put the address of the next
                    #instruction into $ra and branch to lab.
bltzal     $t0, lab #If $t0 < 0, then put the address of the next
                    #instruction into $ra and branch to lab.
```

2. 跳转指令(Jumps)

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

3. 子程序调用指令

子程序调用指令的实质是跳转并链接（Jump and Link），它把当前程序计数器的值保留到$ra 中，以备跳回）：

跳转到子程序：

```
jal        sub_label       #  "jump and link", preserve pc to $ra
```

sub_label 为子程序的标签，如 LOOP， SUB_ROUTINE

从子程序返回：

```
jr        $ra     #  "jump register" jump as the value of $ra
```

返回到$ra 中储存的的返回地址对应的位置， $ra 中的返回地址由 jal 指令保存。

注意，返回地址存放在$ra 寄存器中。如果子程序调用了下一级子程序，或者是递归调用，此时需要将返回地址保存在堆栈中，因为每执行一次 jal 指令就会覆盖$ra 中的返回地址。

Part VI: 系统调用和 I/O 操作（SPIM 仿真）

系统调用是指调用操作系统的特定子程序。
系统调用用来在仿真器的窗口中打印或者读入字符串 string, 并可显示程序是否结束。
用 syscall 指令进行对系统子程序的调用。
本操作首先支持$v0 and $a0-$a1 中的相对值
调用以后的返回值（如果存在）会保存在$v0 中。

表二：系统调用的功能：

Service	Code in $v0	Arguments	Results
print_int	1		
print_float	2		
print_double	3		
print_string	4		
read_int	5		integer returned in $v0
read_float	6		float returned in $v0
read_double	7		double returned in $v0
read_string	8	$a0 = memory address of string input buffer $a1 = length of string buffer (n)	
sbrk	9	$a0 = amount	address in $v0
exit	10		

The print_string service expects the address to start a null-terminated character string. The directive .asciiz creates a null-terminated character string.
打印字符串的功能认为起始地址为一个空终止符串。声明字符串使用的**.asciiz** 指示符会建立一个空终止符串。

The read_int, read_float and read_double services read an entire line of input up to and including the newline character.
读入整形，读入浮点型和读入双精度的功能会读取一整行，包含换行符。

The read_string service has the same semantices as the UNIX library routine fgets.
It reads up to n-1 characters into a buffer and terminates the string with a null character.
If fewer than n-1 characters are in the current line, it reads up to and including the newline and terminates the string with a null character.
读入字符串的功能和 UNIX 库中 fgets 函数的语法相同。他会读入 n-1 个字符到缓存，然后以空字符结尾。如果少于 n-1 的字符，它会读到结尾并包含换行符，并以空字符结尾。

The sbrk service returns the address to a block of memory containing n additional bytes. This would be used for dynamic memory allocation.
sbrk 功能返回一个包含有 n 个附加字节的存储区的地址，这回被用于动态内存分配。

exit 功能用于停止程序运行。

e.g. Print out integer value contained in register $t2
例：打印在$t2 中的整数的值

```
li         $v0, 1          # load appropriate system call
                           # code into register $v0;
                           #code for printing integer is 1
move       $a0, $t2        # move integer to be printed
                           # into $a0:  $a0 = $t2
syscall                    # call operating system to
                           # perform operation
```

```
#e.g.   Read integer value, store in RAM location with label
# int_value (presumably declared in data section)

li         $v0, 5                  # load appropriate system call
                                  # code into register $v0;
                                   # code for reading integer is
                                   #5
syscall                            # call operating system to
                                   # perform operation
sw         $v0, int_value          # value read from keyboard
                                   # returned in register $v0;
                                   # store this in desired location
```

e.g. Print out string (useful for prompts)

```
.data
string1             .asciiz "Print this.\n"        # declaration
                                                   #for string variable,
                                                   # .asciiz directive makes
                                                   # string null terminated

.text
main:       li      $v0, 4                         # load appropriate system call
                                                   #code into register $v0;
                                                   # code for printing string is 4
la          $a0,    string1                        # load address of string to be
                                                   # printed into $a0
syscall                                            # call operating system to
                                                   # perform print operation
```

e.g. To indicate end of program, use exit system call; thus last lines of program should be:

```
li          $v0, 10          # system call code for exit = 10
syscall                      # call operating sys
```

附： ASCII Code Table:

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com