Arquitectura y Lenguaje Ensamblador MIPS

Rafael Ignacio Zurita *Rodolfo del Castillo*
rafa@fi.uncoma.edu.ar *rdc@fi.uncoma.edu.ar*

Resumen

Este documento describe el modelo de programación MIPS (registros, ISA). Tambien incluye algunas notas y expansiones del ensamblador al conjunto de instrucciones (pseudo instrucciones, directivas).

> *Con el nombre de MIPS (siglas de Microprocessor without Interlocked Pipeline Stages) se conoce a toda una familia de microprocesadores de arquitectura RISC desarrollados por MIPS Technologies. Los diseños del MIPS son utilizados en la línea de productos informáticos de SGI; en muchos sistemas embebidos; en dispositivos para Windows CE; routers Cisco; y videoconsolas como la Nintendo 64 o las Sony PlayStation, PlayStation 2 y PlayStation Portable. Más recientemente, la NASA usó uno de ellos en la sonda New Horizons1. Fuente: https://es.wikipedia.org/wiki/MIPS_(procesador)*

2.1 MIPS Registros

El registro 0 ($0 o $zero) mantiene siempre el valor cero y no puede ser cambiado. Consecuentemente, cualquier instrucción que utilice el registro cero utiliza la constante cero sin tener que especificar un literal. Esta es una innovación, debido a que provee una extensión significativa al conjunto de instrucciones de la arquitectura (ISA), sin el costo de procesar un codigo de operación. Por otro lado se piere un registro, ya que no puede utilizarse para almacenar otro valor.

Registro	Función	Nombre en MIPS
0	constante cero	$0 o $zero
1	reservado para el ensamblador	$at
2 - 3	resultados de una función	$v0 - $v1
4 - 7	argmentos	$a0 - $a3
8 - 15	valores temporales	$t0 - $t7
16 - 23	valores preservados	$s0 - $s7
24 - 25	valores temporales	$t8 - $t9
26 - 27	reservados para el kernel	$k0 - $k1
28	puntero global	$gp
29	puntero de pila	$sp
30	puntero de marco de pila	$fp
31	dirección de retorno	$ra

De los 32 registros, unicamente el registro $0 y $31 son dedicados exclusivamente (como parte el hardware y del ISA). Los registro $2 a $30 pueden ser utilizados como registros de propósito general, sin restricción. De cualquier manera, debido a que el ensamblador de MIPS realiza un fuerte uso de pseudo instrucciones (y la traducción de pseudo instrucciones requiere de un registro auxiliar) el registro $1 está reservado para ser utilizado por el ensamblador. Esto puede parecer extraño, pero como muchas pseudo instrucciones son traducidas a multiples instrucciones, frecuentemente se requiere de un registro temporario. $1 ($at) es este registro. Dedicando el registro $at al ensamblador permite que el programador no tenga que preocuparse de que el ensamblador inadvertidamente modifique alguno de sus registros utilizados.

2.2 Operaciones de Carga y Almacenamiento

Como un clasico procesador RISC, las operaciones de acceso a memoria que MIPS soporta son unicamente cargar (load) y almacenar (store). Estas instrucciones pueden operar con valores de 8, 16 y 32 bits (byte, half word, y word en la terminología de MIPS). Las instruciones son:

```
sw    Store word            lw    Load word
sh    Store half word       lh    Load half word
sb    Store byte            lb    Load byte
```

El unico modo de direccionamiento a memoria soportado por MIPS es el modo de direccionamiento indirecto por registro con desplazamiento. Instrucciones tipicas de carga y almacenamiento pueden ser:

```
MIPS assembly         Operation

lw $r2,4($r3)         Load $r2 from memory pointed at by r3 + 4
sw $6 ,8($r4)         Store $r6 in memory pointed at by $r4 + 8
```

MIPS, como otros procesadores (por ejemplo ARM) no tienen una instrucción simple para copiar datos de registro a registro. De cualquier manera, una pseudo instrucción llamada move existen en los ensambladores MIPS. La operación se traduce como un addu (sumar sin signo). Por ejemplo:

```
Pseudo operation Action                          Real MIPS code

move $3,$2       copies reg. $2 to reg. $3       addu $2,$0,$2
move $2,$0       clears reg. $2 because $0 = 0   addu $2,$0,$0
```

Los registros pueden ser cargados con un literal. MIPS presenta un campo de 16 bit literal, y especifica instrucciones especificas (mnemotécnico) que trabajan con literales. Por ejemplo:

```
li  $4,0x1234  load register $4 with  the 16-bit value 0001001000110100 and zero-fill to 32 bits.
```

Aquí, el mnemotécnico li (cargar literal) indica la naturaliza del operando.

La instrucción li es, de hecho, una pseudo instrucción. El ensamblador de MIPS traduce la instrucción li $4, 0x1234 en una instrucción ori $4, $r0, 0x1234. La operación OR lógico entre r0 (conteniendo cero) y un literal da como resultado el literal, el cual es copiado al registro destino.

2.2 Un ejemplo sencillo

Cargamos dos registros, los sumamos, y cargamos un tercer registro.

```
 .text                  #start of program
main: li      $t1,0x1234     #load register r9 with 0x00001234
      li      $t2,0xAC       #load register r10 with 0x000000AC
      addu    $t3,$t2,$t1    #add r9 to r10 and put the result in r10
      li      $t4,0xFFEE     #load register r12 with 0x0000FFEE
```

2.2 Cargando valores de 32-bit en MIPS

Como MIPS puede manejar unicamente constantes de 16-bit se necesitan al menos dos instrucciones para ensamblar una constante de 32-bit. Lo que se necesita hacer es obtener los 16-bit de orden superior, desplazar 16 lugares a la izquierda este valor, y concatenar el resultado con la constante de 16-bit de orden inferior. Afortunadamente, esta secuencia se simplifica por el uso de la instrucción lui (load upper immediate). Esta instrucción carga un literal de 16-bit y desplaza este a la izquierda 16 lugares, por lo tanto, lui $t0, 0x1234 tiene el efecto de cargar $t0 con el valor 0x12340000.

De esta manera, el uso de la instrucción de carga de inmediato (li) permite cargar un

literal de 32-bit traduciendo li (pseudo instrucción) a dos instrucciones reales.

```
li $t0, 0x12345678 se traduce a:
```

```
lui $t0, 0x1234
ori $t0, $t0, 0x5678
```

2.3 Intrucciones de procesamiento de datos (unidad aritmética lógica)

MIPS tiene un conjunto convencional de instrucciones para realizar operaciones de procesamiento de datos, y utiliza un formato de tres operandos registros:

```
add   $t2,$t1,$t0   # [t2] ¬ [t1] + [t0]
addu  $t2,$t1,$t0   # [t2] ¬ [t1] + [t0] ignore overflow
addi  $t2,$t1,N     # [t2] ¬ [t1] + N
addiu $t2,$t1,N     # [t2] ¬ [t1] + N ignore overflow
sub   $t2,$t1,$t0   # [t2] ¬ [t1] - [t0]
subu  $t2,$t1,$t0   # [t2] ¬ [t1] - [t0]
subi  $t2,$t1,N     # [t2] ¬ [t1] - N
subiu $t2,$t1,N     # [t2] ¬ [t1] - N
mul   $t1,$t0       # [hi,lo] ¬ [t1] * [t0] 32-bit x 32-bit
mulu  $t1,$t0       # [hi,lo] ¬ [t1] * [t0] 32-bit x 32-bit unsigned
div   $t1,$t0       # [hi,lo] ¬ [t1] / [t0]
divu  $t1,$t0       # [hi,lo] ¬ [t1] / [t0]
and   $t2,$t1,$t0   # [t2] ¬ [t1] . [T0]
andi  $t2,$t1,$t0   # [t2] ¬ [t1] . N
or    $t2,$t1,$t0   # [t2] ¬ [t1] + [t0]
ori   $t2,$t1,$t0   # [t2] ¬ [t1] + N
nor   $t2,$t1,$t0   # [t2] ¬ [t1] + [t0]
xor   $t2,$t1,$t0   # [t2] ¬ [t1] Å [t0]
xor   $t2,$t1,N     # [t2] ¬ [t1] Å N
not   $t2,$t1       # [t2] ¬ [t1]
```

La operación de multiplicación es una multiplicación de 32-bit x 32-bit real, la cual crea un producto de 64-bit. MIPS utiliza dos registros especiales para almacenar el resultado, HI y LO. HI almacena los 32-bit del resultado del producto de orden superior, y LO los 32-bit de orden inferior. Para poder acceder a estos registros existen dos instrucciones dedicadas para transferir los dos valores a un registro del usuario:

```
mfhi $t0    # transfer the high-order 32 bits of the product register to $t0
mflo $t1    # transfer the low-order 32 bits of the product register to $t1
```

2.3 Instrucciones de desplazamiento

En principio hay 16 tipos de operaciones de desplazamiento (aritmeticas, lógicas, circulares, y circular a través de carry, x2 a la izquierda o derecha x2 para estático y dinámico). La mayoría de los procesadores MIPS no implementan el conjunto completo (aunque el procesador de arquitectura 68000 CISC es uno que casi contiene todos los tipos). En realidad, no es necesario todos porque se pueden sintentizar un tipo de desplazamiento usando otro existente. MIPS tiene un numero muy modesto de instrucciones de desplazamiento:

```
Instruction         Action

sll  $t1,$t2,4      shift left logical 4 places
srl  $t1,$t2,8      shift right logical 8 places
sra  $t1,$t2,1      shift right arithmetically 1 place
srlv $t1,$t2,$t3    shift right logical by the number of place in $t3
sllv $t1,$t2,$t3    shift left logical by the number of place in $t3
srav $t1,$t2,$t3    shift right arithmetically by the number of place in $t3
```

2.3 Ejemplo de un programa con operaciones aritmeticas

Suponga que se debe calcular F = (A^2 + B + C) x 32 + 4, donde A, B y D son valores de 32 bits consecutivos en memoria. Tambien asumiremos que el resultado se puede almacenar

dentro de 32 bits (y por lo tanto, no tenemos que considerar aritmetica extendida). El programa a continuación utiliza la convención de los ensambladores tipica de MIPS. El area de datos, definida por .data, define e inicializa variabes y reserva espacio.

```
       .data                    # start of data area
A:     .word   2                # define 32-bit variable A and initialize to 2 (ARM DCW)
B:     .word   3                # offset of B is 4 bytes from A
C:     .word   4                # offset of D is 8 bytes from A
D:     .space  4                # define 32-bit variable D and reserve 4 bytes (ARM DS)

       .text                    # start of program
main:  la      $t1,A            # load register t1 with the address of A
      lw      $t2,($t1)        # load register t2 with the value of A
      mult    $t2,$t2          # calculate A * A with 64-bit result in HI:LO
      mflo    $t2              # get low-order 32 bits of product  from LO in $t2
      lw      $t3,4($t1)       # get B
      add     $t2,$t2,$t3      # calculate A*A + B
      lw      $t3,8($t1)       # get C
      add     $t2,$t2,$t3      # calculate A*A + B + C
      sll     $t2,$t2,5        # calculate (A*A + B + C) * 32
      addi    $t2,$t2,4        # calculate (A*A + B + C) * 32 + 4
      sw      $t2,12($t1)      # save result in D
      li      $v0,10           # load register r2 (v$0) with the terminate message
      syscall                  # call the OS to carry out the function specified by r2
```

En este ejemplo se ha declarado la variable A, para tener un puntero en $t1 hacia A, y luego, las demas variables son accedidas utilizando desplazamientos a partir de A. Por ejemplo, la variable B es accedida con 4($t1). Aunque las etiquetas B, C y D están declaradas tambien en el area de datos, estas no son necesarias porque finalmente estos nombres no son utilizados en el resto del código. De cualquier manera, si se utilizaran estas variables haría mas facil la lectura del programa.

2.4 Bifurcaciones y flujo de control

Hasta ahora hemos analizado las instrucciones de carga y almacenamiento, y operaciones aritmeticas y logicas. El próximo paso es introducir operaciones condicionales, las cuales permiten construir flujos de control del estilo if…then…else, y while (x < 4) { hacer }.

MIPS tiene una bifurcación incondicional, como la mayoría de los procesadores. Su formato en lenguaje ensamblador es b destino, donde destino es una etiqueta. La dirección destino es amacenada en la instrucción, como un literal de 16-bit. Debido a que las direcciones de las instrucciones están siempre alineadas (a direcciones múltiplos de 4), el literal de 16-bit se utiliza para especificar los bits b17 - b02 de una dirección, y los bits b01 y b00 son cero. El modo de direccionamiento es relativo al contador de programa, por lo tanto el literal se suma al contenido del contador de programa, como un desplazamiento con signo, para permitir bifurcaciones de 128K-byte hacia delante o detrás desde el valor del PC actual.

Las bifurcaciones condicionales en MIPS no son dependientes de un set o flag o código de condición. Esta es dependiente de una operación definida, que es una comparación de registros. Por ejemplo:

```
beq $t0, $t1, destino # bifurcar si [t0] = [t1]
```

Todas las instrucciones de bifurcacion utilizan un desplazamiento de 16-bit, que es un valor extendido a 18-bit con signo, y que es sumado al contador de programa para generar la dirección destino.

La bifurcación correspondiente a "no igual" es:

```
bne  $t0,$t1,target   #branch to target is [t0] != [t1]
```

Considere el siguiente ejemplo en donde una repetitiva suma los iez primeros enteros y almacena el resultado en memoria. El codigo es:

```
       .data                          # start of data area
sum:  .space   4                    # define 32-bit variable for the result
      .text                         # start of program
main: la      $t0,sum               # load register t0 with the address of the result
      li      $t1,1                 # we are going to add 10 integers starting with 1
      li      $t2,10                # 10 to count
      li      $t3,0                 # clear the sum in t3
next: add     $t3,$t3,$t1           # add the next increment
      addi    $t1,$t1,1             # add 1 to the next increment
      bne     $t1,$t2,next          # are we there yet? If not repeat
      sw      $t3,($t0)             # if we are, store sum in memory
      li      $v0,10                # and stop
      syscall                       #
     .end     main
```

Hay pocas instrucciones de bifurcaciones, que es lo tradicional. Otros tipos de instrucciones con desigualdades deben ser sintetizadas:

```
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 ³ $t1
```

2.5 Apendice: directivas y pseudoinstrucciones del ensamblador

```
.align n
```

Align the next datum on a 2n byte boundary. For example, .align 2 aligns the next value on a word boundary. .align 0 turns off automatic alignment of .half, .word, .float, and .double directives until the next .data or .kdata directive.

```
.ascii str
```

Store the string in memory, but do not null-terminate it.

```
.asciiz str
```

Store the string in memory and null-terminate it.

```
.byte b1,..., bn
```

Store the n values in successive bytes of memory.

```
.data
```

The following data items should be stored in the data segment. If the optional argument addr is present, the items are stored beginning at address addr.

```
.double d1,..., dn
```

Store the n floating point double precision numbers in successive memory locations.

```
.extern sym size
```

Declare that the datum stored at sym is size bytes large and is a global symbol. This directive enables the assembler to store the datum in a portion of the data segment that is efficiently accessed via register $gp.

```
.float f1,..., fn
```

Store the n floating point single precision numbers in successive memory locations.

```
.globl sym
```

Declare that symbol sym is global and can be referenced from other files.

```
.half h1,..., hn
```

Store the n 16-bit quantities in successive memory halfwords.

```
.space n
```

Allocate n bytes of space in the current segment (which must be the data segment in SPIM).

```
.text
```

The next items are put in the user text segment. In SPIM, these items may only be instructions or words (see the .word directive below). If the optional argument addr is present, the items are stored beginning at address addr.

```
.word w1,..., wn
```

Store the n 32-bit quantities in successive memory words. SPIM does not distinguish various parts of the data segment (.data, .rdata and .sdata).

Acknowledgments

El profesor (retirado) Alan Clements es el autor original de artículo. Gentilmente, nos otorgó permiso para la traducción y distribución del documento.

Availability

PEDCO

References