Arquitectura y Lenguaje Ensamblador MIPS

Rafael Ignacio Zurita *(**) Alan Clements (autor original en inglés)*
rafa@fi.uncoma.edu.ar

Resumen

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

Contenido

IMPORTANTE La literatura en microprocesadores MIPS puede ser confusa por varias razones, especialmente para quienes no crecieron utilizando diferentes procesadores MIPS:

Primero, la arquitectura ha evolucionado, y hoy en día existen más instrucciones que las que fueron incorporadas en el diseño original.

Segundo, existen ciertos errores comunes reiterados en el uso consistente de la terminología; el mas significativo siendo el uso de la letra u, la cual tiene diferentes significados en diferentes usos en la arquitectura. Por ejemplo, la

instrucción DIVU tiene un nombre sensato y realiza una división sin signo de números enteros. En cambio, la instrucción ADDU (también terminada en u) realiza una suma sin generar exepciones si se produce un desbordamiento (overflow). Como conclusión, es conveniente recordar que el uso de la letra u final no tiene significado único en MIPS, y debe prestar atención al verdadero significado de esas instrucciones al utilizarlas.

Tercero, MIPS explota la definición del término ISA (instruction set architecture), conjunto de instrucciones de la arquitectura, al límite. El término ISA es usualmente definido como la interfaz del programador con la máquina, a través del lenguaje ensamblador y el código máquina: el ISA incluye los registros, las instrucciones y los modos de direccionamiento. Los diseñadores de ensambladores para MIPS han ampliado el conjunto básico con el uso de pseudoinstrucciones (operaciones que no son parte del ISA oficial pero que el ensamblador traduce a otras instrucciones o grupos de instrucciones). Esta característica puede ser muy confusa para quien se inicia en esta arquitectura, porque a menudo pareciera que MIPS tiene más instrucciones de las que realmente tiene.

2 Introducción

MIPS (siglas de Microprocessor without Interlocked Pipeline Stages) es una familia de microprocesadores de arquitectura RISC, desarrollado inicialmente de manera académica en la universidad de Standford, al inicio de los 80's. El grupo estaba dirigido por John Hennssey. Luego, el diseño fue continuado comercialmente por MIPS Technologies.

La arquitectura MIPS fue utilizada en computadoras Silicom Graphics; 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. Actualmente es utilizada en la mayoría de los routers wireless hogareños, en muchos sistemas embebidos, y tambien en algunas supercomputadoras. Fuente: https://es.wikipedia.org/wiki/MIPS_(procesador)

3 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 código de operación. Por otro lado se pierde 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	argumentos	$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, únicamente 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 múltiples 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.

4 Conjunto de Instrucciones MIPS

4.1 Operaciones de Carga y Almacenamiento

Como un clásico procesador RISC, las operaciones de acceso a memoria que MIPS soporta son únicamente 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 instrucciones son:

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

El único modo de direccionamiento a memoria soportado por MIPS es el modo de direccionamiento indirecto por registro con desplazamiento. Instrucciones típicas 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 ignorando overflow). 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.

4.1.1 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
```

4.1.2 Cargando valores de 32-bit en MIPS

Como MIPS puede manejar únicamente 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
```

4.2 Intrucciones de procesamiento de datos (unidad aritmética lógica)

MIPS tiene un conjunto convencional de instrucciones para realizar operaciones de procesamiento de datos, utilizando 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] xor [t0]
xor   $t2,$t1,N     # [t2] ← [t1] xor 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 almace-

nar 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
```

4.2.1 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
```

4.2.2 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 final-

mente 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.

4.3 Bifurcaciones y flujo de control

Hasta ahora hemos analizado las instrucciones de carga y almacenamiento, y operaciones aritméticas y lógicas. 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 bifurcación 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 diez primeros enteros y almacena el resultado en memoria. El código 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
```

5 Directivas y pseudoinstrucciones del ensamblador

5.1 Directivas

```
.align n
```

Alinea el próximo dato en la próxima dirección disponible múltiplo de 2^n.

Por ejemplo `.align 2` alinea el próximo valor a una dirección de memoria múltiplo de 4 (o lo que es lo mismo, lo alinea a las palabras de la memoria). `.aling 0` desactiva el alineamiento automático de las directivas `.half`, `.word`, `.float`, y `.double` hasta la próxima directiva `.data` o `.kdata`.

```
.ascii str
```

Almacena la cadena de texto string en memoria, sin caracter nulo final.

```
.asciiz str
```

Almacena la cadena de texto string en memoria, finalizando la misma con un caracter nulo.

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

Almacena los n valores b1,...,bn en ubicaciones sucesivas en memoria, utilizando un byte de espacio para cada elemento.

```
.data
```

Los siguientes ítems son considerados datos y serán almacenados en memoria en el segmento de datos del programa. Si se agrega el argumento opcional addr entonces los items son almacenados en memoria comenzando en la dirección addr.

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

Almacena los n valores d1,...,dn en ubicaciones sucesivas en memoria, utilizando el formato de punto flotante IEEE-754 doble precisión (8 bytes).

```
.extern sym size
```

Declara que el dato guardado en `sym` tiene un tamaño `size` y es un símbolo global. Esta directiva le permite al ensamblador guardar al dato en una porción del segmento de dato que puede ser eficientemente accedido a través del registro `$gp`

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

Almacena los n valores f1,...,fn en ubicaciones sucesivas en memoria, utilizando el formato de punto flotante IEEE-754 simple precisión (4 bytes).

```
.globl sym
```

Declara que el símbolo sym es global, y puede ser referenciado desde otro archivos (por ejemplo, los símbolos `main` y `__start` deberían ser declarados como globales para que el sistema pueda iniciar la ejecución del programa principal).

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

Almacena las n cantidades de 16-bits en ubicaciones sucesivas en memoria, utilizando medias palabras (2 bytes) para cada elemento. Ensamblando con GNU as las medias palabras quedan alineadas a direcciones múltiplo de 2.

```
.space n
```

Reserva n bytes de espacio en el segmento actual (en memoria).

```
.text
```

Los siguientes ítems son considerados instrucciones y serán almacenados en memoria en el segmento de texto (código) del programa. Si se agrega el argumento opcional `addr` entonces los ítems son almacenados en memoria comenzando en la dirección `addr`.

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

Almacena los n valores w1,...,wn de 32-bits en ubicaciones sucesivas en memoria, utilizando el espacio de una palabra (4 bytes) para cada elemento. Ensamblando con GNU as estas palabras quedan alineadas a direcciones múltiplo de 4.

5.2 Pseudo-instrucciones

```
Pseudo-instrucción   Significado

move $t0, $t4        move: copia el contenido de t4 en t0
la $t0, etiqueta     load address: carga en t0 la dirección de etiqueta
li $t0, 0x8003FAA2   load immediate: carga en t0 la constante
abs $t0, $t4         absolute value: t0 = valor absoluto de t4
neg $t0, $t4         negate: calcula el opuesto de t4 y lo guarda en t0
mult $t0, $t4, $5    multiply: multiplica t4 por t5 y guarda el resultado en t0
div $t0, $t4, $t5    divide: divide t4 por t5 y guarda el resultado en t0
rem $t0, $t4, $t5    remainder: divide t4 por t5 y guarda el resto en t0
sgt $t0, $t4, $t5    set greater than: si t4 > t5 entonces t0=1, sino t0=0
sle $t0, $t4, $t5    set less or equal: si t4 <= t5 entonces t0=1, sino t0=0
sge $t0, $t4, $t5    set greater or equal: si t4 >= t5 entonces t0=1, sino t0=0
rol $t0, $t4, $t5    rotate left: rotar a la izquierda t4 por t5 lugares
ror $t0, $t4, $t5    rotate right: rotar a la derecha t4 por t5 lugares
not $t0              not: invertir los bits de t0
ld $t0, 4($t5)       load doubleword
sd $t0, 4($t5)       store doubleword
blt $t0, $t5, etiq   branch less than: si t0 < t5 bifurca la ejecucion a etiqueta
bgt $t0, $t5, etiq   branch greater than: si t0 > t5 bifurca la ejecucion a etiqueta
ble $t0, $t5, etiq   branch less or equal: si t0 <= t5 bifurca la ejecucion a etiqueta
bge $t0, $t5, etiq   branch greater or equal: si t0 >= t5 bifurca la ejecucion a etiqueta
```

6 Convención de llamada a procedimientos

La convención de llamadas a procedimientos o funciones es un esquema de implementación de bajo nivel para determinar de qué manera las subrutinas reciben parámetros de su "llamador" y devuelven un resultado.
Diferentes arquitecturas tienen diferentes implementaciones. El hardware implementa algunas tareas de estas implementaciones, y las demás son convenciones llevadas a cabo en software (por los compiladores). Incluso, pueden existir diferentes convenciones de uso de registros para una misma arquitectura, lo que puede llevar a confusiones.

En MIPS, la convención de llamadas a procedimientos rige principalmente el uso de los registros de propósito general. La convención predeterminada es la utilizada por el compilador GCC, llamada O32. La convención presentada en esta sección es levemente mas sencilla que la de GCC, pero compatible. [a]

---

[a] La convención que utilizamos es en realidad la que presenta el libro de Patterson y Henessy, para estar en sintonía con la bibliografía de la materia. La versión del libro es la convención utilizada por GCC durante el desarrollo de ese libro.

Llamada a procedimientos o funciones

Para llamar a una subrutina o procedimiento se utiliza la instrucción jal (jump and link). La misma resguarda en el registro ra (31) la dirección de retorno, y modifica el registro pc con la dirección de la primera instrucción del procedimiento invocado. Para retornar al "invocador" se utiliza la instrucción jr ra.

Los registros t son temporales. Si se utilizan antes de invocar a un procedimiento (jal) se los debe resguardar en la pila. Cuando el procedimiento invocado finalizó se les recupera su valor anterior al jal desde la pila.

Los registros s son mantenidos. Si un procedimiento invocado los utiliza debe resguardar, en la pila, el contenido original de los mismos antes de modificarlos. Antes de que el procedimiento invocado finalice (jr ra) debe recuperar desde la pila los valores de los registros s originales. De esta manera, si el invocador utilizaba los mismos registros s, los valores son mantenidos.

Los registros a0, a1, a2, y a3 se utilizan para el pasaje de los cuatro primeros argumentos a un procedimiento. Los demás argumentos se deben pasar utilizando la pila "actual" del procedimiento invocador. Los registros v0 y v1 se utilizan para devolver resultados desde un procedimiento.

El registro ra mantiene la dirección de retorno. Si un procedimiento debe invocar a otro procedimiento (procedimientos anidados o recursivos) debe resguardar antes su valor (antes de invocar con jal al nuevo procedimiento). Luego de que el procedimiento invocado ha finalizado, el valor de ra es recuperado de la pila. Con este mecanismo es posible preservar las direcciones de retorno en funciones anidadas, que en otro caso serían sobreescritas por la ejecución repetida de la instrucción jal.

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	argumentos	$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

Gestión de la pila

La administración de la pila se realiza por software (por convención) de la siguiente manera. La pila crece a direcciones más bajas, y el puntero de pila (registro sp) debe siempre apuntar a una dirección alineada con doble palabra (múltiplo de 8). Cuando una función o procedimiento debe utilizar la pila le resta al registro sp la cantidad de bytes que necesite, de esta manera reserva espacio en la pila para su uso. El tamaño mínimo de la pila es de 24 bytes, para poder colocar ahí el contenido de los argumentos (registros a0..a3) y para almacenar ra. No es necesario resguardar en pila estos registros, pero la reserva de espacio mínima de 24 bytes debe implementarse por convención.

La dirección contenida en sp es un espacio libre en memoria, es decir, no debe ser utilizada para resguardar ningún valor.

El registro fp (frame pointer) mantiene la dirección de memoria más alta del seg-

mento de pila actual.

Cuando el procedimiento que utilizó pila está por retornar a su llamador debe restablecer el valor original de sp, sumando la misma cantidad de bytes que sustrajo al crear espacio de pila.

Ejemplo de código en C

```
void swap(int vector[], int n)
{

        int t = vector[n];
        vector[n] = vector[n+1];
        vector[n+1] = t;

}


void pares(int vector[], int cantidad)
{
        int i;
        for (i=0; i < cantidad; i = i + 2)
                if (vector[i] < vector[i+1])
                        swap(vector, i);
}


int v[10] = {1, 2, 4, 3, 5, 6, 8, 7, 9, 10};

void main()
{
        pares(v, 10);

        /* resto de codigo de main */
}
```

Traducción del código en C a lenguaje ensamblador MIPS

```
# Segmento de DATOS
        .data
memoria:
v: .word 1, 2, 4, 3, 5, 6, 8, 7, 9, 10


# Segmento de CODIGO
# main
        .text
        .globl __start
        .globl main
__start:
main:

        addiu   $sp,$sp,-24     # Reserva espacio en el segmento pila
        sw      $ra,20($sp)     # Resguarda ra
        sw      $fp,4($sp)      # Resguarda fp
        addi    $fp, $sp, 20    # Nuevo fp

        la      $a0, v          # argumento 0: direccion de v
        li      $a1, 10         # argumento 1: cantidad de elementos
        jal     pares           # Invocacion al procedimiento pares

        # resto de codigo de main

        lw      $fp,4($sp)      # Restablece fp
        lw      $ra,20($sp)     # Restablece ra
        addiu   $sp,$sp,24      # Libera el espacio utilizado del segmento pila

        # Finalizar programa (retorna al SO)
```

```
        move $a0, $0
        li      $v0, 4001
        syscall



# Procedimiento pares
pares:
        addiu   $sp,$sp,-40     # Establece un nuevo marco de pila para pares
        sw      $ra,36($sp)     # Resguarda ra
        sw      $fp,4($sp)      # Resguarda fp
        addi    $fp, $sp, 36    # Nuevo fp

        sw      $a0,32($sp)     # Resguarda la direccion de vector
        sw      $a1,28($sp)     # Resguarda cantidad
        sw      $0,8($sp)       # Variable local i de pares

loop_for:
        lw $t1, 8($sp)          # Variable local i

        # .. resto del codigo de pares....

        # Verifica si elemento i de vector es
        # menor al elemento i + 1
        # Si es menor llama a swap para intercambiarlos
        # Si es menor...
        sw    $a0,32($sp)     # argumento 0: direccion de vector
        move  $a1, $t1        # argumento 1: i

        jal   swap            # Invoca a swap

        lw    $t1,8($sp)      # Recupera i
        addiu $t1,$t1,2       # Incrementa variable i en dos y la preserva
        sw    $t1,8($sp)      # nuevamente en la pila.

        # codigo para iterar a loop_for: nuevamente si i < cantidad

        salir_de_pares:
        lw      $ra,36($sp) # Recupera ra, fp y sp
        lw      $fp,4($sp)
        addiu   $sp,$sp,40
        jr      $ra          # Retorna a main



# Procedimiento swap
        .text
swap:
        # codigo de swap

        jr        $ra            # Retorna a pares
```

En la siguiente figura (1) se observa el esquema del segmento de pila para el ejemplo anterior. Se marca con celeste claro el marco de pila para main, y en gris el marco de pila del procedimiento pares.

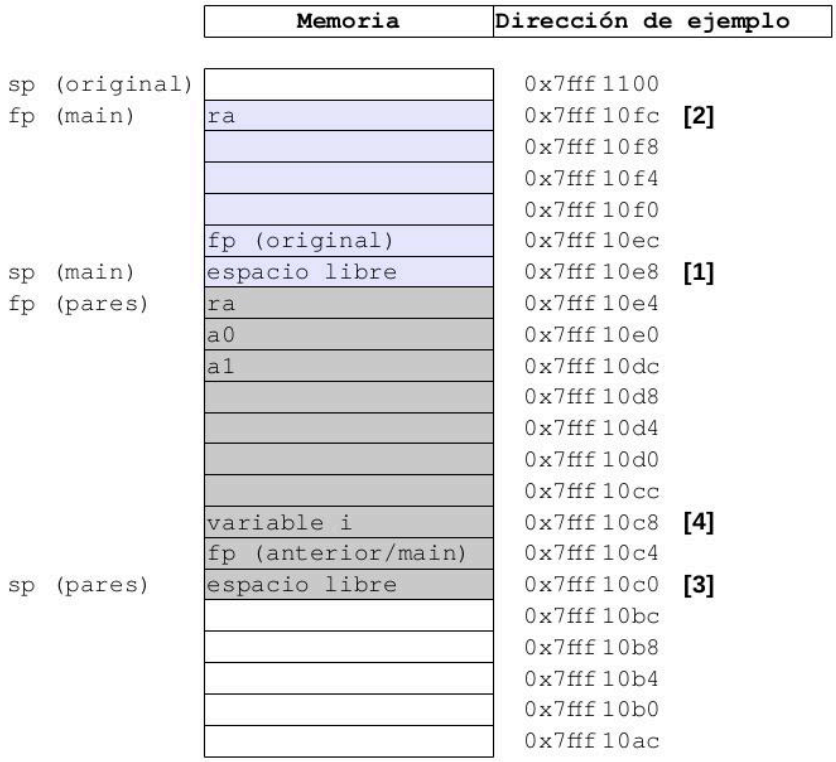

Figure 1: [1] Este es el valor de sp luego de que la CPU ejecuta la instrucción de main: addiu $sp, $sp, -24. [2] Este es el valor de fp luego de que la CPU ejecuta la instrucción de main: addi $fp, $sp, 20. [2] ra es resguardado en la pila a través de la instrucción de main: sw $ra, 20($sp). [3] Este es el nuevo valor de sp luego de que la CPU ejecuta la instrucción del procedimiento pares: addiu $sp, $sp, -40. [4] En esta dirección se mantiene la variable i definida en el procedimiento pares en el código original en C.

Reconocimientos

(**) Al profesor (retirado) Alan Clements, autor original de artículo. Gentilmente, nos otorgó permiso para la traducción y distribución del documento. Fue escrito para las materias de arquitecturas de computadoras, de la universidad de Teesside, Inglaterra.

Revisión: Lic. Rodrigo Cañibano

Availability

PEDCO