Programando sobre MIPS64

Escribir un programa en cualquier lenguaje involucra estructurar el código de forma que éste sea modular, extensible, simplede mantener y reutilizable. En assembly, eso involucra dividir el programa en funciones o subrutinas, de forma de evitar repetir el mismo código en más de un lugar.

En MIPS64, el soporte para la invocación a subrutinas es mucho más reducido que el que provee la arquitectura x86 (como ocurre en este y en otros aspectos más cuando se compara una arquitectura RISC contra otra CISC). No hay un manejo implícito de la pila en la invocación a las subrutinas (como ocurre con las instrucciones CALL y RET del x86) y tampoco hay instrucciones explícitas para apilar y desapilar valores en la pila (cosa que hacen las instrucciones PUSH y POP del x86).

El siguiente programa muestra un ejemplo de invocación a una subrutina escrita en MIPS64:

```
          .data
result:   .word  0

          .text
          daddi r4, r0, 10
          daddi r5, r0, 20
          jal   sumar              # Se llama a la subrutina "sumar"
          sd    r2, result(r0)
          halt

sumar:    dadd  r2, r4, r5         # subrutina "sumar"
          jr    r31                # Retorna al punto donde se llamó a "sumar"
```

La instrucción encargada de realizar la invocación a la subrutina `sumar` es la instrucción **`jal`** (*jump and link*). Esta instrucción toma un único argumento que es la dirección a donde continuar con la ejecución (`sumar`).
La instrucción **`jal`** realiza estas dos acciones:

- Guarda en el registro `r31` la dirección de la instrucción siguiente a la instrucción **`jal`** (en este caso, sería la dirección de la instrucción **`sd`**).
- Pone en el registro PC la dirección indicada por la etiqueta `sumar` (con lo que la próxima instrucción a ejecutar va a ser la instrucción **`dadd`**, que es la primera instrucción de la subrutina `sumar`).

Para retornar de la subrutina, basta con saltar a la dirección contenida en el registro `r31`, que es lo que hace la instrucción **`jr`** `r31` en el ejemplo. Así, la ejecución continuará en la instrucción siguiente a la instrucción **`jal`**.

Si se desean pasar parámetros a la subrutina, basta con definir qué valores deben ser guardados y en qué registros, de forma que tanto la subrutina como el programa que la invoca estén de acuerdo en donde se dejan y se toman los parámetros en cuestión. Lo mismo ocurre con los valores que la subrutina pueda devolver.

El inconveniente que presenta este mecanismo tan sencillo es que si el programa principal invoca a una subrutina (dejando guardada la dirección de retorno en el registro `r31`) y dentro de esa subrutina se necesite invocar a una segunda subrutina, el segundo **`jal`** sobrescribiría el valor contenido en el registro `r31` con la nueva dirección de retorno (que sería la dirección de instrucción siguiente al **`jal`** dentro de la primera subrutina). También puede ocurrir que la segunda subrutina utilice registros que la primera subrutina estaba usando para almacenar valores antes de que se produjera la invocación y que esperaría que se conserven luego de regresar de la misma.

Para simplificar la escritura de subrutinas, se estableció una convención que define tanto el mecanismo para la invocación a las subrutinas así como la posibilidad de usar una pila para almacenar temporalmente valores durante la invocación a una subrutina. A diferencia de una arquitectura x86, en MIPS no existen instrucciones que manipulan explícitamente la pila (no hay instrucciones del tipo PUSH y POP, por ejemplo). Sin embargo (y al igual que en otros casos similares), que no existan instrucciones para un propósito específico no significa que esas funciones no puedan realizarse usando instrucciones más simples y generales.

Convención para el uso de los registros:

El procesador MIPS64 posee 32 registros, de 64 bits cada uno, llamados `r0` a `r31` (también llamados `$0` a `$31`). Sin embargo, resulta más conveniente para los programadores darles nombres más significativos a esos registros. La siguiente tabla muestra la convención empleada para nombrar a los 32 registros mencionados:

Registros	Nombres	Usos	Preservado
r0	$zero	Siempre tiene el valor 0 y no se puede cambiar.	
r1	$at	*Assembler Temporary* – Reservado para ser usado por el ensamblador.	
r2-r3	$v0-$v1	Valores de retorno de la subrutina llamada.	
r4-r7	$a0-$a3	Argumentos pasados a la subrutina llamada.	
r8-r15	$t0-$t7	Registros temporarios. No son conservados en el llamado a subrutinas.	
r16-r23	$s0-$s7	Registros salvados durante el llamado a subrutinas.	**X**
r24-r25	$t8-$t9	Registros temporarios. No son conservados en el llamado a subrutinas.	
r26-r27	$k0-$k1	Para uso del kernel del sistema operativo.	
r28	$gp	*Global Pointer* – Puntero a la zona de la memoria estática del programa.	**X**
r29	$sp	*Stack Pointer* – Puntero al tope de la pila.	**X**
r30	$fp	*Frame Pointer* – Puntero al marco actual de la pila.	**X**
r31	$ra	*Return Address* – Dirección de retorno en un llamado a una subrutina.	**X**

La tabla anterior establece una convención a seguir al momento de escribir subrutinas, de forma que cualquier subrutina escrita por un programador (o generada por un compilador de algún lenguaje de más alto nivel) pueda ser usada junto con otras escritas por otros. Los registros marcados con una '**X**' en la última columna de la tabla deben ser preservados durante la invocación a una subrutina. Esto quiere decir que si la subrutina los va a usar, debe asegurarse de salvar sus valores antes de alterarlos para así poder restaurarlos antes de retornar (de ahí los nombres `$s0`-`$s7`, la 's' es de ***S**aved Register*).

Los registros `$s0` al `$s7` son usados normalmente para cumplir el rol de variables locales a la subrutina: existen desde el principio al final de la subrutina y su valor es preservado durante invocaciones a otras subrutinas. Al contrario, los registros `$t0` al `$t9` son usados para almacenar valores temporarios y probablemente sean modificados al invocar a otras subrutinas. A pesar de esto, resultan útiles para almacenar valores auxiliares dentro de cálculos complicados.

El programa anterior quedaría entonces escrito de la siguiente manera:

```
          .data
result:   .word  0

          .text
          daddi $a0, $0, 10
          daddi $a1, $0, 20
          jal   sumar
          sd    $v0, result($0)
          halt

sumar:    dadd  $v0, $a0, $a1
          jr    $ra
```

El segundo programa es exactamente igual al primero. Hablar de `$a0`, `r4` o `$4` es lo mismo (son nombres distintos para el mismo registro). Sin embargo, a la persona que lea el segundo programa le resultará mucho más sencillo interpretar que cuando se llame a la subrutina "`sumar`" en el programa principal, se le estará pasando como primer argumento el valor 10 en el registro `$a0` y como segundo argumento el valor 20 en el registro `$a1`. Además, esperará que al retornar de esa subrutina, el resultado esté en `$v0`.

De esta manera, siguiendo la convención propuesta, todas las subrutinas usarían los mismos registros para recibir parámetros y para retornar valores. Además, se establece que una subrutina debe asegurarse que al regresar de su invocación, los registros `$s0` a `$s7` van a tener los valores que tenían antes de la invocación.

Esto no se aplica a los registros `$t0` a `$t9`, por lo que el código que llama a una subrutina debe asegurarse de preservarlo en caso que los llegara a necesitar luego de la llamada. Tampoco se aplica a los registros `$a0` a `$a3` ni a los registros `$v0` y `$v1`, que pueden llegar a tener valores diferentes antes y después de la invocación.

Los ejemplos a continuación muestran la diferencia en el uso de los registros `$s0` a `$s7` con respecto a `$t0` a `$t9`. Se debe tener presente que se está suponiendo que las variables locales `a`, `b`, `c`, `d` y `e` de un lenguaje de alto nivel se corresponden respectivamente con los registros `$s0`, `$s1`, `$s2`, `$s3` y `$s4` de MIPS64.

La siguiente línea escrita en un lenguaje de alto nivel:	Puede traducirse en las siguientes instrucciones de MIPS64:

`e := a + b + c - d`	`dadd $t0, $s0, $s1`
	`dadd $t0, $t0, $s2`
	`dsub $s4, $t0, $s3`

En cambio, la línea:	Puede traducirse en las siguientes instrucciones:

`e := (a + b) - (c + d)`	`dadd $t0, $s0, $s1`
	`dadd $t1, $s2, $s3`
	`dsub $s4, $t0, $t1`

La garantía de preservar los valores originales también se extiende al registro `$ra`, que contiene la dirección de retorno luego de una llamada a una subrutina. De esta manera, una subrutina puede llamar a una segunda subrutina y, cuando la segunda retorne, la primera aún tendría disponible la dirección a la cual deberá regresar (ya que el registro `$ra` fue preservado durante la segunda llamada).

Para poder mantener esa garantía, es necesario contar con algún lugar donde almacenar los valores que originalmente tienen los registros antes de que la llamada a una subrutina los modifique. ¿Cómo es posible hacer esto si todos los registros tienen funciones asignadas? La respuesta es almacenar esos valores en la memoria principal y la estructura utilizada típicamente para hacerlo es una pila.

Convención para el uso de la pila:

En el MIPS64, no hay instrucciones específicas para manipular y mantener la pila. Sin embargo, existe un registro que por convención todas las subrutinas usarán como puntero al tope de la pila. Ese registro es el `r29`, también conocido como `$sp` (*stack pointer*).

Dado que la pila será implementada por el programa, se tiene la libertad de elegir como trabajar con la misma. En particular, se puede definir una pila que al estar vacía apunte al tope de la memoria y cada vez que se apile un valor, el puntero al tope de la pila se decrementa. También puede implementarse una pila que comience apuntado a una dirección de memoria dada y cada vez que se apile un valor, el puntero al tope de la pila se incremente. Salvo que se lo indique explícitamente, se asumirá siempre el primer caso, en donde el puntero al tope de la pila se decrementa al apilar valores.

Suponiendo que al comienzo del programa el registro `$sp` fue inicializado adecuadamente, es posible apilar un valor en el tope de la pila usando el siguiente par de instrucciones:

```
daddi $sp, $sp, -8
sd    $t1, 0($sp)
```

El efecto neto conseguido es el equivalente a la instrucción **`push`** `$t1` del x86 (imaginando que `$t1` fuera un nombre de registro válido en esa arquitectura). Al registro `$sp` se le resta 8 porque esa es la cantidad de bytes que ocupa el registro `$t1` y porque además la arquitectura impone la restricción que todas las palabras deben estar en la memoria alineadas en direcciones múltiplo de 8.

De forma análoga, para desapilar un valor del tope de la pila se usaría:

```
        ld    $t2, 0($sp)
        daddi $sp, $sp, 8
```

Nuevamente, el efecto neto conseguido es el equivalente a la instrucción **pop** `$t2` del x86.

Toda subrutina asumirá que el registro `$sp` apunta al tope de la pila y deberá realizar las acciones necesarias para modificar ese registro acorde a las necesidades que la subrutina tenga. Generalmente, lo primero que debe hacer es almacenar en la pila todos los registros que la subrutina va a usar y que se sabe que debe preservar. En particular, toda subrutina que invoque a una segunda subrutina debe resguardar el valor contenido en el registro `$ra`. Por otro lado, lo último que toda subrutina debe hacer es recuperar todos los valores almacenados de los registros resguardados y dejar el tope de la pila en el mismo lugar en que lo encontró.

Así, toda subrutina se dividirá siempre en **tres partes**: el prólogo, el cuerpo y el epílogo. En el prólogo, se resguarda en la pila todos los registros que deban ser preservados; en el cuerpo, se escribirá el código propio de la subrutina y, finalmente, en el epílogo, se deberá deshacer todo lo hecho en el prólogo.

Entonces, el esqueleto de toda subrutina será:

```
subrutina:
prólogo:    daddi $sp, $sp, -tamaño_frame       ; Reserva espacio en la pila
            sd    $ra, 0($sp)                   ; Guarda la dirección de retorno
            sd    $s0, 8($sp)                   ; Guarda el registro $s0
            sd    $s1, 16($sp)                  ; Guarda el registro $s1
            :
            :
cuerpo:     … instrucciones …                   ; Cuerpo de la subrutina
            :
            :
epílogo:    ld    $ra, 0($sp)                   ; Recupera la dirección de retorno
            ld    $s0, 8($sp)                   ; Recupera el registro $s0
            ld    $s1, 16($sp)                  ; Recupera el registro $s1
            :
            :
            daddi $sp, $sp, tamaño_frame        ; Restaura el tope de la pila
            jr    $ra                           ; Retorna
```

Notar que en lugar de modificar el registro `$sp` cada vez que se desea apilar un valor, resulta más conveniente calcular de antemano cuantos registros se van a apilar durante la ejecución de la subrutina y en un solo paso modificar el registro `$sp` para hacer el lugar suficiente como para almacenarlos. Eso es lo que se hace al comienzo del prólogo y al final del epílogo con la instrucción **`daddi`**.

El tamaño del *frame*, o la porción de la pila que usará una subrutina en cada llamado, lo calcularemos como:

`tamaño_frame` = 8 bytes x (número de registros a guardar + número de variables locales a la subrutina)

Son 8 bytes ya que en la arquitectura MIPS64, cada registro almacenado es de 64 bits = 8 bytes.

Una vez que se hizo lugar en el tope de la pila, es posible usar $sp como registro base e indicar un desplazamiento como valor inmediato, usando un modo de direccionamiento relativo a la base desplazada. Así, `0($sp)` hace referencia al tope de la pila, `8($sp)` al valor anterior al tope, `16($sp)` al valor anterior a este último, etc. De esa forma, se puede saber que en `0($sp)` se encuentra almacenada la dirección de retorno, en `64($sp)`, por ejemplo, se encuentra una variable local, etc.

Dentro de una subrutina puede ser necesario apilar y desapilar valores arbitrariamente, causando que el registro `$sp` se modifique a lo largo de la ejecución de la subrutina. En ese caso, ya no se puede usar al registro `$sp` como base

para el *frame*, ya que no siempre apuntará a la misma dirección si el tope de la pila se desplaza.

Para remediar esta situación, se suele usar al registro `$fp` como base para el *frame*, de forma que una vez que se le asigne en el prólogo la dirección de comienzo del *frame*, `$fp` no se modifique hasta que la subrutina termine, usándolo como base para direccionar a las variables y registros almacenados en la pila. En otras palabras, usaríamos `0($fp)` en lugar de `0($sp)`, `32($fp)` en lugar de `32($sp)`, etc. Esto implicaría que el valor del registro `$fp` deberá ser resguardado en la pila antes de asignarle la dirección de comienzo del *frame*.

Cabe aclarar que si una subrutina no modifica los registros `$s0` a `$s7`, no es necesario guardarlos. De la misma forma, si una subrutina no va a llamar a otra subrutina, no es necesario que guarde el valor contenido en `$ra`. De hecho, una subrutina sencilla puede no necesitar modificar ni el registro `$sp` ni el registro `$fp` en lo absoluto.

De esta manera, es posible economizar la cantidad de instrucciones que forman parte del prólogo y del epílogo, permitiendo una total libertad en cuanto a la elección de la estrategia a usar para acceder a la pila, a las variables locales y a los registros que se usarán durante la ejecución de la subrutina.

El siguiente ejemplo sencillo convierte una cadena de caracteres a caracteres en mayúsculas:

```
            .data
cadena:     .asciiz "Caza"
            .text
            DADDI $sp, $0, 0x400       ; La pila comienza en el tope de la memoria de datos
            DADDI $a0, $0, cadena      ; Guarda como primer argumento para upcaseStr
            JAL   upcaseStr
            HALT

        ; Pasar una cadena a mayuscula
        ; Parametros: - $a0 -> inicio de cadena
        ; Se utiliza la pila para guardar:
        ;      - $ra -> porque se invoca a otra subrutina
        ;      - $s0 -> para guardar la dirección de inicio de la cadena y recorrerla

upcaseStr:  DADDI $sp, $sp, -16        ; Reserva lugar en la pila -> 2 x 8
            SD    $ra, 0($sp)
            SD    $s0, 8($sp)
            DADD  $s0, $a0, $zero      ; copia la dirección de inicio de la cadena
upcaseStrLOOP: LBU   $a0, 0($s0)       ; recupera el caracter actual y lo pone como
                                       ; argumento para upcase
            BEQ   $a0, $zero, upcaseStrFIN   ; Si es el fin de la cadena, termina
            JAL   upcase
            SB    $v0, 0($s0)          ; Guarda el caracter procesado en la cadena
            DADDI $s0, $s0, 1          ; avanza al siguiente caracter
            J     upcaseStrLOOP
upcaseStrFIN: LD    $ra, 0($sp)
            LD    $s0, 8($sp)
            DADDI $sp, $sp, 16
            JR    $ra

        ; Pasar un caracter a mayúscula.
        ; Parámetros: - $a0 -> caracter
        ;             - $v0 -> caracter en mayúscula
        ; No se utiliza la pila porque no se usan registros que deban ser salvados

upcase:     DADD  $v0, $a0, $zero
            SLTI  $t0, $v0, 0x61       ; compara con 'a' minúscula
            BNEZ  $t0, salir           ; no es un caracter en minúscula
            SLTI  $t0, $v0, 0x7B       ; compara con el caracter siguiente a 'z' minúscula (z=7AH)
            BEQZ  $t0, salir           ; no es un caracter en minúscula
            DADDI $v0, $v0, -0x20      ; pasa a minúscula (pone a '0' el 6to bit)
salir:      JR    $ra                  ; retorna al programa principal
```

Este segundo ejemplo muestra como se realiza la inicialización de la pila, como se pasan parámetros a subrutinas, como se reciben los resultados y como respetar la convención para las llamadas a subrutinas. El ejemplo cuenta cuantos palíndromos hay en un texto dado. Un palíndromo es una palabra que se lee igual hacia adelante que hacia atrás.

```
        .data
texto:  .asciiz  "Nos sometemos a problemas si se usa un radar para reconocer ese oso azul"

        .text
        daddi   $sp, $0, 0x400          ; La pila comienza en el tope de la memoria de datos
        daddi   $a0, $0, texto
        jal     contar_palindromos
        halt

; Cuenta cuantos palíndromos hay en un texto dado. Recibe en $a0 la dirección del comienzo
; del texto (terminado en 00). Devuelve en $v0 la cantidad de palíndromos encontrados.
contar_palindromos:
        daddi   $sp, $sp, -48
        sd      $ra,  0($sp)
        sd      $s0,  8($sp)
        sd      $s1, 16($sp)
        sd      $s2, 24($sp)
        sd      $s3, 32($sp)
        sd      $s4, 40($sp)
        dadd    $s0, $a0, $0            ; $s0 apunta al comienzo de una palabra
        dadd    $s1, $a0, $0            ; $s1 apuntará al final de la palabra
        daddi   $s2, $0, 32             ; $s2 contiene el código ASCII del espacio en blanco
        dadd    $s4, $0, $0             ; $s4 cuenta cuantos palíndromos se encontraron
while1: lbu     $s3, 0($s1)
while2: daddi   $s1, $s1, 1
        dadd    $a0, $s0, $0
        daddi   $a1, $s1, -2
        beqz    $s3, procesa            ; Si se acabó la cadena, procesa la última palabra
        bne     $s3, $s2, while1        ; Si no es un espacio, busca el siguiente carácter
procesa:jal     es_palindromo           ; Si es palíndromo devuelve 1, sino 0.
        dadd    $s4, $s4, $v0           ; Suma el resultado al contador de palíndromos
        beqz    $s3, terminar           ; Si se acabó la cadena, termina
saltea_blancos:
        dadd    $s0, $s1, $0
        lbu     $s3, 1($s1)
        daddi   $s1, $s1, 1
        beqz    $s3, terminar
        bne     $s3, $s2, while2
        j       saltea_blancos
terminar:
        dadd    $v0, $s4, $0
        ld      $ra,  0($sp)
        ld      $s0,  8($sp)
        ld      $s1, 16($sp)
        ld      $s2, 24($sp)
        ld      $s3, 32($sp)
        ld      $s4, 40($sp)
        daddi   $sp, $sp, 48
        jr      $ra

; Determina si una palabra es PALÍNDROMO. Recibe en $a0 la dirección del comienzo de la palabra y
; en $a1 la dirección del final de la palabra. Devuelve 1 en $v0 si es palíndromo o 0 si no lo es.
es_palindromo:
        dadd    $v0, $0, $0
lazo:   lbu     $t0, 0($a0)
        daddi   $a0, $a0, 1
        lbu     $t1, 0($a1)
        daddi   $a1, $a1, -1
        slt     $t2, $a1, $a0
        bne     $t0, $t1, no_es
        beqz    $t2, lazo
        daddi   $v0, $0, 1
no_es:  jr      $ra
```