Práctica 18. La pila.

1. La pila

La pila es una zona de memoria que se usa para almacenar información temporalmente de nuestros programas. Un esquema de la memoria del procesador MIPS se muestra en la siguiente figura.

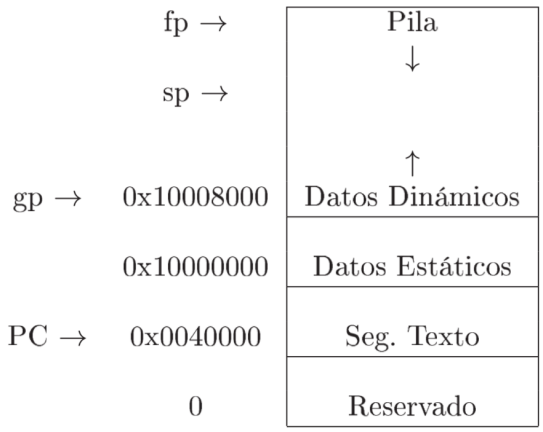

Fig. 1. Estructura de la memoria del procesador MIPS

Una pila es una estructura de memoria de tipo **LIFO (*Last In, First Out*)**, es decir, el último dato que se almacena es el primero que se obtiene al realizar una operación de extracción.

Asociada a cualquier estructura de tipo pila existen dos operaciones básicas:

- **Apilar**: transferir datos hacia la pila (o si se prefiere, almacenar datos en la pila).
- **Desapilar**: transferir datos desde la pila (o "sacar" datos de la pila).

En las máquinas MIPS el segmento de pila es una zona de memoria que "crece" de direcciones superiores a direcciones inferiores, como se puede ver en la figura 1. En el simulador lo hace a partir de la dirección `0x7FFFF1DF`. A medida que se van apilando datos se van ocupando direcciones más bajas. Esto contrasta, por ejemplo, con el funcionamiento del segmento de datos, donde cada dato que se almacena lo hace en direcciones crecientes.

Para gestionar el segmento de pila se dispone de un registro que no se ha utilizado hasta ahora. Se trata del registro `$sp` (*stack pointer* o puntero de pila), que se encontrará en cada momento apuntando a la última dirección en uso dentro del segmento de la pila.

Las operaciones de apilar y desapilar se realizarán de la forma siguiente:

- **Apilar:**
  1. Se actualiza el puntero de pila restándole una cantidad igual al número de bytes del dato que se quiere apilar; es decir, "se reserva sitio" en la pila.
  2. Se almacena el dato a partir de la dirección indicada por el puntero de pila mediante el modo de direccionamiento deseado.
- **Desapilar:**
  1. Se carga el dato a partir de la dirección indicada por el puntero de pila en el registro o registros deseados. Se tendrá en cuenta el tamaño del dato y se empleará un modo de direccionamiento adecuado.
  2. Se actualiza el puntero de pila sumándole un entero igual al número de bytes del dato que ha sido leído, de modo que apunte así al dato siguiente.

Ejemplo de apilar y desapilar un byte, suponiendo que el dato a apilar está en $t0 y que cuando desapilamos lo guardamos en $t1:

```
…
#Apilar
addi $sp, $sp, -1
sb $t0, ($sp)
…
#Desapilar
lb $t1, ($sp)
addi $sp, $sp, 1
…
```

Ejemplo de apilar y desapilar un half, suponiendo que el dato a apilar está en $t0 y que cuando desapilamos lo guardamos en $t1:

```
…
#Apilar
addi $sp, $sp, -2
sh $t0, ($sp)
…
#Desapilar
lh $t1, ($sp)
addi $sp, $sp, 2
…
```

Ejemplo de apilar y desapilar un word, suponiendo que el dato a apilar está en $t0 y que cuando desapilamos lo guardamos en $t1:

```
…
#Apilar
addi $sp, $sp, -4
sw $t0, ($sp)
…
#Desapilar
lw $t1, ($sp)
addi $sp, $sp, 4
```

La pila es un segmento de la memoria, por ello, a la hora de almacenar o extraer datos de la pila, al igual que ocurre al declarar datos en el segmento de datos, hay que tener en cuenta que los datos han de estar alineados. Por tanto, para facilitar su uso, si trabajamos con datos de distintos tamaños apilaremos/desapilaremos siempre con el tamaño del más grande.

El siguiente ejemplo ilustra el funcionamiento de la pila:

2. Las subrutinas y la pila

Cuando se utilizan subrutinas, en caso de necesitar más de cuatro argumentos (los cuales irían por los registros `$a0` al `$a3`) o de devolver más de dos resultados (los cuales irían por `$v0` y `$v1`) hace falta utilizar la estructura de memoria conocida como *pila*.

El procedimiento será:

- La función que llama a la subrutina, antes de la instrucción `jal` pasará los argumentos por `$a0` al `$a3` y el resto los **apilará** en la pila.
- La subrutina, **desapilará** los argumentos de la pila (y el resto estará en los registros del `$a0` al `$a3`) y hará los cálculos.
- La subrutina almacenará los resultados en los registros `$v0` y `$v1` y si hay más, los **apilará** en la pila.
- La función que llamó a la subrutina (después de la instrucción `jal`) **desapilará** los resultados de la pila y utilizará los que vienen por los registros `$v0` y `$v1`.

Listado 1: Ejemplo de llamada a una subrutina con cinco argumentos.

```
      .data
      .globl main
      .text
main:
      li $a0, 1          # argumento 1
      li $a1, 2          # argumento 2
      li $a2, 3          # argumento 3
      li $a3, 4          # argumento 4
                         # argumento 5 (lo tenemos que apilar)
      addi $sp, $sp, -1 # hacemos "sitio" en la pila decrementando $sp
      li $t0, 5          # ponemos en $t0 el dato que queremos apilar
      sb $t0, ($sp)      # almacenamos $t0 en el espacio reservado

      jal subrutina # el flujo del programa pasa a la subrutina

      li $v0, 10  # cuando la subrutina termina, continúa en este punto
      syscall     # fin del programa

subrutina:
      # utilizamos $a0 - $a3 con normalidad
      ...

      # cuando necesitemos el quinto argumento, lo desapilamos:
      lb $t0, ($sp)    # cargamos de la pila el último byte apilado
      addi $sp, $sp, 1 # actualizamos el puntero de pila

      ...

      jr $ra   # volvemos a la instrucción siguiente a la de llamada
```

3. Subrutinas anidadas y recursivas

Se denominan subrutinas anidadas aquellas que contienen llamadas a otras subrutinas. Por ejemplo, una subrutina que indica si un número es primo que a su vez contiene una llamada a otra subrutina que sirve para indicar cuántos divisores tiene el número que se le pasa como parámetro.

Un caso especial de subrutinas anidadas son las subrutinas recursivas, que se llaman a sí mismas. Por ejemplo, para calcular el factorial de un número *n* se puede llamar a la propia subrutina con el parámetro *n-1*. Siempre tiene que haber un caso trivial que es el que garantiza que las llamadas no sean infinitas.

Si se quiere que una subrutina llame a otra (o bien que se llame a sí misma), mediante la instrucción `jal`, existe el problema de que automáticamente se modifica el contenido del registro `$ra`. Así, se pierde cualquier valor que este registro pudiera tener.

Si el programa principal llama a una subrutina A, que a su vez llama a otra subrutina B, se puede volver desde B a A mediante `jr $ra`, pero al haberse sobreescrito el contenido de `$ra` en la segunda llamada, será imposible volver desde A al programa principal.

La idea es que, antes de realizar una llamada desde una subrutina a otra se salve el contenido de `$ra` en la pila. Así, las direcciones de retorno de las distintas llamadas anidadas/recursivas quedarán almacenadas a medida que se vayan produciendo. Para realizar los retornos, se desapilan las direcciones de retorno y se salta a éstas.

Veamos un esquema de llamada a subrutinas anidadas:

Listado 2: Esquema de subrutinas anidadas.

```
      .data
      .globl main
      .text
main:
      # llamamos a la subrutina_1, que a su vez contendrá una llamada
      # a la subrutina_2. Resolveremos el retorno usando la pila.
      jal subrutina_1

      li $v0, 10
      syscall

subrutina_1:

      ...

      # antes de realizar una llamada anidada, tomamos la precaución
      # de apilar la dirección de retorno de subrutina_1, para así
      # poder volver más adelante al programa principal

      addi $sp, $sp, -4
      sw $ra, ($sp)

      jal subrutina_2

      ...

      # antes de volver al programa principal, desapilamos la dirección
      # de retorno previamente almacenada
      lw $ra, ($sp)
      addi $sp, $sp, 4

      jr $ra # volvemos al main

subrutina_2:

      ...

      jr $ra # volvemos a la subrutina_1, al ser un anidamiento simple
             # no hemos necesitado apilar la dirección de retorno a la
             # subrutina_1
```

4. Otros usos de la pila

Debido a su característica LIFO (*Last In, First Out*), la pila es útil, por ejemplo, para invertir el orden de los elementos de un vector o una cadena.

Listado 3: Ejemplo de uso de la pila para invertir una cadena.

```
      .data
cad: .asciiz "Introduzca una cadena: "
cad2: .asciiz "La cadena invertida es: "
cadleida:   .space 100
cadinv:     .space 100
      .globl main
      .text
main:
      li $v0, 4
      la $a0, cad
```

```
      syscall

      li $v0, 8
      la $a0, cadleida
      li $a1, 100
      syscall

      #Apilo el 0 en la pila para saber cuál es el principio de la pila
      addi $sp, $sp, -1
      sb $zero, ($sp)

      la $t0, cadleida #Puntero a cadena leida
      la $t2, cadinv   #Puntero a cadena invertida

      #Mientras no llegue al \0, leo una letra y la apilo
bucle:
      lb $t1, ($t0)
      beq $t1, $zero, desapilo
      addi $sp, $sp, -1
      sb $t1, ($sp)
      addi $t0, $t0, 1 #El puntero apunta a la siguiente letra
      j bucle
desapilo:
      #Mientras no desapile un 0
      lb $t1, ($sp)
      addi $sp, $sp, 1
      beq $t1, $zero, fin
      sb $t1, ($t2) #EScribo la letra en la cadena
      addi $t2, $t2, 1 #El puntero apunta al siguiente hueco
      j desapilo

fin:
      li $v0, 4
      la $a0, cad2
      syscall


      li $v0, 4
      la $a0, cadinv
      syscall

      li $v0, 10
      syscall
```

5. Ejercicios

El objeto de esta práctica es familiarizarse con el manejo de la pila. Se pide:

1. Realizar un programa que solicite al usuario 6 números enteros, y utilizando los registros $a0, $a1, $a2, $a3 y la pila, llame a una subrutina que devuelva la suma y el producto de todos los números y la resta del 3er número menos el 5º número**.**

2. Realice un programa que pida un número por teclado (se debe controlar si el número insertado es positivo, en caso contrario, se mostrará un mensaje de error) e incluya una subrutina que indique si el número de divisores de un número es par o impar. Para calcular los divisores, dicha subrutina llamará a otra subrutina que calcula el número de divisores del número.

Aclaraciones:
- subrutina 1, devuelve 0 si es impar o 1 si es par
- subrutina 2, devuelve el número de divisores del número insertado

3. Realice un programa que lea 10 enteros y los almacene en memoria (reservar espacio con la directiva `space`) y a continuación, que guarde en memoria el vector al revés (en otra zona de memoria reservada con la directiva `space`). Utiliza la pila para cambiar el orden de los elementos del vector. NOTA: haz 3 versiones distintas del programa considerando enteros de tipo byte, half o word.

4. Teniendo el siguiente segmento de datos:

```
      .data
datos1:
      .byte 14, 23
      .align 2
      .word 47
      .align 1
      .half 9, 12, 15
      .align 2
datos2:
      .space 4
      .byte 'c','o'
```

a. Sumar el word con valor 47, con el half de valor 15 y guardarlo en el .space 4 como dato word, utilizando la etiqueta datos1.

b. Guardar el carácter 'd' y el carácter 'p' en la posición del half con valor 12, utilizando la etiqueta datos2. NOTA: 'd' es el carácter siguiente a la letra 'c' y 'p' es el carácter siguiente a la letra 'p'.