

PRÁCTICA 5

Pila, Subrutinas

Objetivos Familiarizarse con los conceptos que rodean a la escritura de subrutinas en una arquitectura RISC. Uso normalizado de los registros, pasaje de parámetros y retorno de resultados, generación y manejo de la pila y anidamiento de subrutinas.

Parte 1: Pila y Subrutinas.

1. Comprendiendo la primer subrutina: potencia ★

Muchas instrucciones que normalmente forman parte del repertorio de un procesador con arquitectura CISC no existen en el MIPS64. En particular, el soporte para la invocación a subrutinas es mucho más simple que el provisto en la arquitectura x86 (pero no por ello menos potente). El siguiente programa muestra un ejemplo de invocación a una subrutina llamada **potencia**:

<pre> .data base: .word 5 exponente: .word 4 result: .word 0 .code ld \$a0, base(\$zero) ld \$a1, exponente(\$zero) jal potencia sd \$v0, result(\$zero) halt </pre>	<pre> potencia: daddi \$v0, \$zero, 1 lazo: bnez \$a1, terminar daddi \$a1, \$a1, -1 dmul \$v0, \$v0, \$a0 j lazo terminar: jr \$ra </pre>
--	---

- ¿Qué hace el programa? ¿Cómo está estructurado el código del mismo?
- ¿Qué acciones produce la instrucción **jal**? ¿Y la instrucción **jr**?
- ¿Qué valor se almacena en el registro **\$ra**? ¿Qué función cumplen los registros **\$a0** y **\$a1**? ¿Y el registro **\$v0**? ¿Qué valores posibles puede recibir en **\$a0** y **\$a1** la subrutina **potencia**?
- Supongamos que el WinMIPS no posee la instrucción **dmul**. ¿Qué sucede si la subrutina **potencia** necesita invocar a otra subrutina para realizar la multiplicación en lugar de usar la instrucción **dmul**? ¿Cómo sabe cada una de las subrutinas a qué dirección de memoria debe retornar?
- Escriba un programa que utilice **potencia**. En el programa principal se solicitará el ingreso de la base y del exponente (ambos enteros) y se deberá utilizar la subrutina **potencia** para calcular el resultado pedido. Muestre el resultado numérico de la operación en pantalla.
- Escriba un programa que lea un exponente **x** y calcule $2^x + 3^x$ utilizando dos llamadas a **potencia**. Muestre en pantalla el resultado. ¿Funciona correctamente? Si no lo hace, revise su implementación del programa. ¿Qué sucede cuando realiza una segunda llamada a **potencia**? **Pista:** Como caso de prueba, intente calcular $2^3 + 3^3 = 8 + 27 = 35$.

2. Salvado de registros ★

Los siguientes programas tienen errores en el uso de la convención de registros. Indicar qué registros cuál es el error y cómo se podría arreglar el problema en cada caso.

<p>A)</p> <pre> .code daddi \$t0, \$0, 5 daddi \$t1, \$0, 7 jal subrutina sd \$t2, variable (\$0) halt subrutina: daddi \$t4, \$0, 2 dmul \$t0, \$t0, \$t4 dmul \$t1, \$t1, \$t4 dadd \$t2, \$t1, \$t0 jr \$ra </pre>	<p>B)</p> <pre> .code daddi \$a0, \$0, tabla jal subrutina daddi \$t0, \$0, 10 daddi \$t1, \$0, 0 loop: bnez \$t0, fin ld \$t2, 0(\$a0) dadd \$t1, \$t1, \$t2 daddi \$t0, \$t0, -1 dadd \$a0, \$a0, 8 j loop fin: halt </pre>
<p>C)</p> <pre> .code daddi \$a0, \$0, 5 daddi \$a1, \$0, 7 jal subrutina dmul \$t2, \$a0, \$v0 sd \$t2, variable (\$0) halt </pre>	<p>D)</p> <pre> .code daddi \$t0, \$0, 10 # dimension daddi \$t1, \$0, 0 # contador daddi \$t2, \$0, 0 # desplazamiento loop: bnez \$t0, fin ld \$a0, tabla (\$t2) jal espar bnez \$v0, seguir daddi \$t1, \$t1, 1 seguir: daddi \$t2, \$t2, 1 daddi \$t0, \$t0, -1 dadd \$t2, \$t2, 8 j loop sd \$t2, resultado(\$0) fin: halt </pre>

3. Uso de la pila ★

En WinMIPS no existen las instrucciones **PUSH** y **POP**. Por ese motivo, deben implementarse utilizando otras instrucciones existentes. No solo eso, sino que el registro **\$P** es en realidad un registro usual, **r29**, que con la convención se puede llamar por otro nombre, **\$sp**. El siguiente programa debería intercambiar los valores de **\$t0** y **\$t1** utilizando la pila. No obstante, así como está no va a funcionar porque **push** y **pop** no son instrucciones válidas. Implementar la funcionalidad que tendrían estas operaciones utilizando instrucciones **daddi**, **sd** y **ld** para que el programa funcione correctamente. Recordar que los registros ocupan 8 bytes, y por ende el **push** y el **pop** deberán modificar a **\$sp** con ese valor.

```
#v0: devuelve 1 si a0 es par y 0 dlc
#a0: número entero cualquiera
.code
daddi $sp, $0, 0x400
daddi $t0, $0, 5
daddi $t1, $0, 8
push $t0
push $t1
pop $t0
pop $t1
halt
```

4. Variantes de la subrutina potencia ★★

La versión anterior de **potencia** utiliza pasaje por registros y por valor. Escribir distintas versiones de la subrutina **potencia** en base términos del pasaje de parámetros

- a) **Referencia y Registro** Pasando los parámetros por referencia desde el programa principal a través de registros, y devolviendo el resultado a través de un registro por valor.
- b) **Valor y Pila** Pasando los parámetros por valor desde el programa principal a través de la pila, y devolviendo el resultado a través de un registro por valor.
- c) **Referencia y Pila** Pasando los parámetros por referencia desde el programa principal a través de la pila, y devolviendo el resultado a través de un registro por valor.

5. Salvado de registros en subrutinas anidadas ★★

Las siguientes subrutinas anidadas funcionan, pero tienen errores en el uso de la convención de los registros, en especial con respecto a cuáles tienen que salvarse y cuáles no, y también cuándo y en qué caso debe hacerse. Indicar los errores y corregir el código para que las subrutinas usen la convención correctamente.

<p>A) #v0: devuelve 1 si a0 es impar y 0 dlc #a0: número entero cualquiera esimpar: andi \$v0, \$a0, 1 jr \$ra</p> <p>#v0: devuelve 1 si a0 es par y 0 dlc #a0: número entero cualquiera espar: jal esimpar #truco: espar = 1 - esimpar daddi \$s0, \$0, 1 dsub \$v0, \$s0, \$v0 jr \$ra</p>	<p>C) #v0: volumen de un cubo #a0: long del lado lado del cubo vol: daddi \$sp, \$sp, -16 sd \$ra, 0(\$sp) sd \$s0, 8(\$sp) dadd \$s0, \$0, \$a0 dmul \$s0, \$a0, \$a0 dmul \$s0, \$s0, \$a0 daddi \$v0, \$s0, 0 ld \$ra, 0(\$sp) ld \$s0, 8(\$sp) daddi \$sp, \$sp, 16 jr \$ra</p>
<p>B) #v0: devuelve la cantidad de bits 0 que tiene un número de 64 bits #a0: número entero cualquiera cant0: daddi \$t0, \$0, 0 daddi \$t1, \$0, 64</p> <p>loop: jal espar dadd \$t0, \$t0, \$v0 #desplazo a la derecha #para quitar el último bit dsrl \$a0, \$a0, 1 daddi \$t1, \$t1, -1 bnez \$t1, loop jr \$ra</p>	<p>#v0: diferencia de volumen de los cubos #a0: long del lado del cubo más grande #a1: long del lado del cubo más chico diffvol: jal vol daddi \$t0, \$v0, 0 daddi \$a0, \$a1, 0 jal vol dsub \$v0, \$t0, \$v0 jr \$ra</p>

6. Subrutinas anidadas ★★

a) **Factorial:** Implemente la subrutina **factorial**, que dado un número **N** devuelve **factorial(N) = N! = N * (N-1) * (N-2) * ... * 2 * 1**. Por ejemplo, el factorial de 4 es $4! = 4 * 3 * 2 * 1$. Recordá también que el factorial de 0 también existe, y es **factorial(0) = 0! = 1**

b) **Número combinatorio:** Utilizando **factorial**, implementa la subrutina **comb** que calcula el **número combinatorio** (también llamado **coeficiente binomial**) $\text{comb}(m,n) = m! / (n! * (m-n)!)$. Asumir que $n > m$.

7. Subrutinas de Strings ★★

Implemente subrutinas típicas para manipular strings, de modo de tener construir una pequeña librería de código reutilizable.

a) **longitud:** Recibe en \$a0 la dirección de un string y retorna su longitud en \$v0

b) **contiene:** Recibe en \$a0 la dirección de un string y en \$a1 un carácter (código ascii) y devuelve en \$v0 1 si el string contiene el carácter \$a1 y 0 de lo contrario.

c) **es_vocal:** Determina si un carácter es vocal o no, ya sea mayúscula o minúscula. La rutina debe recibir el carácter y debe retornar el valor 1 si es una vocal ó 0 en caso contrario. Para implementar **es_vocal**, utilizar la subrutina **contiene**, de modo que para preguntar si un carácter es una vocal, se pregunte si un string con todas las vocales posibles contiene a este carácter.

d) **cant_vocales** Usando la subrutina escrita en el ejercicio anterior, **cant_vocales** recibe una cadena terminada en cero y devuelve la cantidad de vocales que tiene esa cadena.

e) **comparar:** Recibe como parámetros las direcciones del comienzo de dos cadenas terminadas en cero y

retorna la posición en la que las dos cadenas difieren. En caso de que las dos cadenas sean idénticas, debe retornar -1.

8. Subrutinas con vectores ★★

a) **suma** Escribir una subrutina que reciba como argumento en \$a0 la dirección de una tabla de números enteros de 64 bits, y en \$a1 la cantidad de números de la tabla. La subrutina debe devolver en \$v0 la suma de los números de la tabla

b) **positivos** Idem a), pero la subrutina debe contar la cantidad de números positivos. Para ello, implementar y usar otra subrutina llamada **es_positivo**, que reciba un número en \$a0 y devuelva en \$v0 1 si es positivo y 0 de lo contrario.

9. Subrutina recursiva ★★★

En un ejercicio anterior se implementó la subrutina **factorial** de forma iterativa. Implementar ahora la subrutina pero de forma **recursiva**.

La definición recursiva de **factorial** es:

Caso base: **factorial**(0) = 1

Caso recursivo: **factorial**(n) = n * **factorial**(n-1)

En términos de pseudocódigo, su implementación es:

```
subrutina factorial(n):  
  if n = 0:  
    retornar 1  
  else:  
    m = factorial(n-1)  
    retornar n * m
```

a) Implemente la subrutina **factorial** definida en forma recursiva.

Pista 1: El caso base puede codificarse directamente

Pista 2: En el caso recursivo hay una llamada a otra subrutina, con lo cual deberá preservar el registro \$ra

Pista 3: En el caso recursivo deberá primero llamar a **factorial** de forma recursiva, pero si el valor **n** original está guardado en un registro temporal, se perderá ese valor.

b) ¿Es posible escribir la subrutina **factorial** sin utilizar una pila? Justifique.

Parte 2: Ejercicios tipo parcial

1. Lectura y procesamiento de números ★★★

Implementar una subrutina **INGRESAR_NUMERO**. La misma deberá solicitar el ingreso por teclado de un número entero del 1 al 9. Si el número ingresado es un número válido entre 1 y 9 la subrutina deberá imprimir por pantalla el número ingresado y retornar dicho valor. En caso contrario, la subrutina deberá imprimir por pantalla "Debe ingresar un número" y devolver el valor 0. Para ello, implementar y usar una subrutina **ENTRE** que reciba un número N y otros dos números B y A, y devuelva 1 si $B < N < A$ o 0 de lo contrario.

Usando la subrutina INGRESAR_NUMERO implementar un programa que invoque a dicha subrutina y genere una tabla llamada NUMEROS con los valores ingresados. La generación de la tabla finaliza cuando la suma de los resultados obtenidos sea mayor o igual a el valor almacenado en la dirección MAX.

Al finalizar la generación de la tabla, deberá invocar a la subrutina PROCESAR_NUMEROS, que debe recibir como parámetro la dirección de la tabla NUMEROS y la cantidad de elementos y contar la cantidad de números impares ingresados. Se debe mostrar por pantalla el valor calculado, con el texto "Cantidad de Valores Impares: " y el valor. Para ello, utilizar la subrutina ES_IMPAR codificada anteriormente.

2. Colores alternativos ★★★

Escribir un programa que imprime alternativamente un punto rojo y uno azul en la pantalla gráfica, y llena toda la pantalla de esta forma.

Para ello, implementar una subrutina **fila_alternativa** que recibe un número de fila y dos colores, y llena toda la fila con pixeles de los dos colores de forma alternativa. Utilizando **fila_alternativa**, escriba un programa que pinte toda la pantalla de rojo y azul, de forma tal que en la primera fila se comience con el color rojo, en la 2da con azul, y así sucesivamente.