Organización y Arquitectura de las Computadoras

Práctica 07: Convención de llamadas a subrutinas

Bonilla Reyes Dafne 319089660

Medina Guzmán Sergio 314332428

1. Objetivo

El alumno se familiarizará con los principios detrás de una invocación a una subrutina, analizando el flujo de las instrucciones y los recursos compartidos, haciendo énfasis en la interacción entre el hardware y el software.

2. Procedimiento

Escribe las rutinas necesarias para cada uno de los ejercicios en lenguaje ensamblador de MIPS, usa la convención de llamadas a subrutinas y agrega una rutina main en donde se carguen de la memoria los argumentos en los registros adecuados y se llame a la subrutina que resuelva el ejercicio. Cada ejercicio debe encontrarse en un archivo de código fuente distinto. No olvides documentar el código.

3. Ejercicios

1. Completa el código de la figura 2 agregando los preámbulos, invocaciones, conclusiones y retornos faltantes de las rutinas main, mist 0 y mist 1. Los segmentos del código dado no pueden ser modificados. Responde en la documentación del código: ¿Qué hace la rutina?

```
        .data
a:      .word   5
b:      .word   4
        .text
main:   # Preambulo main
        # Invocacion de mist_1
        # Retorno de mist_1
        # Conclusion main
# mist_1 recibe como argumentos $a0 y $a1
mist_1: # Preambulo mist_1
        move    $s0, $a0
        move    $t0, $a1
        li      $t1, 1
loop_1: beqz$s0, end_1
        # Invocación de mist_0
        move    $a0, $t0        # Se pasa el argumento $a0
        move    $a1, $t1        # Se pasa el argumento $a1
        # Retorno de mist_0
        move    $t1, $v0
        subi    $s0, $s0, 1
        j       loop_1
end_1:  # Conclusion mist_1
        move    $v0, $t1        # Se retorna el resultado en $v0
# mist_0 recibe como argumentos $a0 y $a1
mist_0: # Preambulo mist_0
        mult    $a0, $a1
        # Conclusion mist_0
        mflo    $v0             # Se retorna el resultado en $v0
```

Para la realización de este ejercicio tomamos varias cosas en cuenta. Expliquemos cada parte de este:

- `data:` En la parte de `.data` solo agregamos una cadena para imprimir el resultado. Además, se agregó la línea de `.globl main` para preservar la sintaxis del código en ensamblador.
- `main:` Aquí eliminamos el comentario de preámbulo, por lo que lo único que se carga en esta parte son los argumentos en los registros `$a0` y `$a1`. A continuación, invocamos al método `mist_1` y guardamos el resultado en `$a1` para que cuando pasemos como argumento de `syscall` a `$a0` este no se modifique. Finalmente, imprimimos.
- `mist_1:` Guardamos espacio en la pila para el registro de retorno y para `$s0`, que funcionará como una especie de contador para el número de iteraciones en el loop.
- `loop_1:` Asignamos un posible caso en el que `b` es 0. Esto considerando que se ingresaran los argumentos por terminal, aunque al final dejamos el programa funcional solo para los valores preestablecidos. Aquí, llamamos a `mist_0` y restamos 1 al contador `$s0`.
- `end_1:` Cuando `$s0` llegue a 0, pasaremos a esta parte, en donde cargamos todo en la pila y la limpiamos. Aquí terminaría la subrutina.
- `mist_0:` En esta parte solo multiplicamos los argumentos y regresamos el resultado.

Finalmente, es importante mencionar que gran parte de la estructura de este ejercicio fue diseñada considerando el ejemplo `Ejemplo4-Practica7` de las notas del profesor.

2. Escribe una rutina que calcule recursivamente el coeficiente binomial de n en k utilizando la identidad de Pascal:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

para todo $k, n \in \mathbb{N}^+$ con caso base $\binom{n}{0} = 1$, con $n \geq 0$ y $\binom{0}{k} = 0$ con $k > 0$. Se recomienda que intenten primero hacer esta función en su lenguaje de programación preferido antes de pasarlo a MIPS, para que entiendan la estructura y funcionamiento del programa. Si hacen esto, añadan una captura de su código en su reporte y comenten que es lo que hace cada línea.

4. Preguntas

1. ¿Cuál es la diferencia entre las instrucciones **jal** y **jr**?

   Mientras que ambas son instrucciones de salto **jal** (Jump and link), almacena en el registro $ra la dirección de retorno actual y esto es útil al invocar subrutinas, pues podemos regresar al punto desde el cual se invicaron dichas subrutinas; por otro lago, **jr** (jump register) simplemente salta a una dirección sin almacenar la dirección de retorno y es útil para implementar un bucle o saltar a una sección específica del programa.

2. ¿Qué utilidad tiene el registro $ra? ¿Se puede prescindir de él?

   El registro $ra nos permite almacenar de manera sencilla una dirección de retorno de una llamada de una función; a pesar de que podríamos prescindir de este registro y hacer saltos continuos dentro del programa, esto involucraría llevar un seguimiento manual y propenso a errores del flujo del programa, mientras que haciendo uso del registro $ra podemos olvidarnos de este problema con el flujo del programa haciendo saltos "manuales" con **j** o con **jr**.

3. ¿Qué utilidad tiene el registro $fp? ¿Se puede prescindir de él?

   Funciona para apuntar al inicio del marco de pila actual, lo que permite a las funciones acceder a su propio marco de pila eficientemente y de forma segura, sin tener que conocer la dirección exacta de la pila en la memoria; este registro facilita el rastreo de las llamadas a las funciones y con ello la depuración de nuestros programas; no podemos prescindir de él, pues garantiza una ejecución correcta de las funciones y la preservación del marco de pila.

4. ¿Cómo es que estas convenciones aseguran que la **subrutina invocada** (callee) no sobreescriba los registros de la **rutina invocadora** (caller)?

   Se asegura que no ocurran tales errores ya que la subrutina invocada se encargará de preservar los valores originales de los registros $s0-$s7, $fp y $ra de la rutina invocadora, almacenándolos en la pila para que se restauren al terminar de ejecutarse la subrutina; de esta manera la subrutina invocada podrá hacer uso de los registros como sea necesario sin perder la información ni el estado original de los mismos. Finalmente, la convención de retorno nos asegura que el control de la ejecución sea devuelto a la rutina invocadora, para una correcta continuación de la ejecución del programa.

5. Definimos como **subrutina nodo** a una subrutina que realiza una o más invocaciones a otras subrutinas y como **subrutina hoja** a una subrutina que no realiza llamadas a otras subrutinas.

   - ¿Cuál es el tamaño mínimo que puede tener un marco para una subrutina nodo? ¿Bajo que condiciones ocurre?

     Cómo hemos visto en los ejercicios de esta práctica, el tamaño mínimo del marco para una subrutina nodo dependerá de 4 cosas principalmente:

     1. La cantidad de registros que se necesitan guardar
     2. La cantidad de parámetros
     3. Variables locales que se utilizan
     4. Complejidad de la subrutina

     El marco para este tipo de subrutinas deberá contener al menos la información necesaria para guardar el estado de los registros que se utilizan y la dirección de retorno de la subrutina, teniendo en cuenta la cantidad de registros que se deben preservar antes de realizar llamadas a otras subrutinas, y la cantidad de espacio que se debe reservar en la pila para las variables locales y los parámetros de la subrutina, además de si se utilizarán o no registros temporales adicionales para almacenar valores intermedios.

     Por lo que podemos decir que el tamaño está condicionado a la implementación de la arquitectura MIPS y del conjunto de instrucciones utilizado en el código de la subrutina.[1]

   - ¿Cuál es el tamaño mínimo que puede tener un marco para una subrutina hoja? ¿Bajo que condiciones ocurre?

     El tamaño mínimo del marco para una subrutina hoja dependerá solo de la cantidad de registros que se necesiten preservar en la pila antes de que se realicen operaciones y la cantidad de espacio que se necesite reservar para las variables locales de la subrutina, ya que al ser de tipo hoja, no hace llamadas a otras subrutinas, volviendo innecesario el tener que guardar la dirección de retorno, por lo que en la mayoría de los casos, el tamaño del marco para una subrutina hoja será menor que el de una subrutina nodo.[2]

     De esta manera, podemos decir que el tamaño mínimo del marco para una subrutina hoja dependerá de 4 cosas:

     1. La cantidad de registros que se necesitan preservar
     2. La complejidad de la subrutina
     3. La cantidad de espacio que se necesita reservar para las variables locales

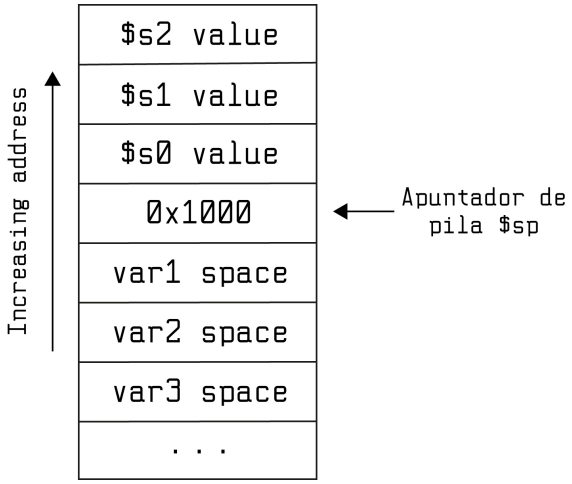
     Por lo tanto, podemos decir que el tamaño mínimo del marco para una subrutina hoja está condicionado a las necesidades específicas de la subrutina y debe ser determinado por el programador en función de los requisitos de la implementación de la arquitectura MIPS.[3]

6. Considera el siguiente pseudocódigo de la figura 3. En donde a[5] es un arreglo de tamaño 5 y "..." son otras acciones que realiza la rutina, además, supón que en la función B se realizan cambios en los registros $s0, $s1 y $s2. Bosqueja la pila de marcos después del preámbulo de la función B.

   En general, cuando se llama a una función la pila de marcos en MIPS reserva un bloque de memoria en la pila para almacenar los registros que la función modifica y el espacio para las variables locales de la función.

```
funcion_A(a, b)
    a[5]
    ...
    funcion_B(a, b, arreglo[0], arreglo[1], arreglo[2])
    ...
```

Teniendo esto en cuenta, el preámbulo de la función B se encargará de guardar en la pila los valores de los registros $s0, $s1 y $s2 que se van a modificar en la función, así como el espacio para las variables locales que se utilizarán. Por lo tanto, después del preámbulo de la función B, la pila de marcos se verá de la siguiente manera:[4]



Cada *varx space* con $x \in \{1, 2, \ldots, i\}$ tal que i es el número de variables locales de la función, representa, como su nombre lo indica, un bloque de memoria reservado para las variables locales de la función, y los valores de los registros $s0, $s1 y $s2 se guardan en la pila en ese orden, comenzando desde la dirección de memoria más alta.

La cantidad de espacio reservado para las variables locales dependerá del número y tamaño de las variables que se utilicen en la función B.

Por otro lado, veamos que el apuntador de pila $sp señala el espacio reservado para la dirección del arreglo. Es importante mencionar que la dirección del arreglo almacenada en la pila de marcos es solo la dirección base del arreglo. Para acceder a los elementos del arreglo, se necesitará un desplazamiento adicional basado en el tamaño de los elementos del arreglo. Por ejemplo, si el arreglo es de tipo *int*, cada elemento tendrá un tamaño de 4 bytes, por lo que para acceder al elemento i-ésimo del arreglo, se debe sumar un desplazamiento de $i \cdot 4$ a la dirección base del arreglo.

Referencias

[1] Britton, R. (2003). *MIPS assembly language programming.* Prentice Hall.

[2] Marut, C. (1992). *Assembly Language Programming and Organization of the IBM PC.* McGraw-Hill.

[3] Patterson, D. A., & Hennessy, J. L. (2018). *Computer organization and design: The hardware/software interface.* Morgan Kaufmann.

[4] MIPS Stack