Departamento de Sistemas Telemáticos y Computación (GSyC)

Convenio de llamada a subrutina

Katia Leal Algara
katia.leal@urjc.es
http://gsyc.escet.urjc.es/~katia/

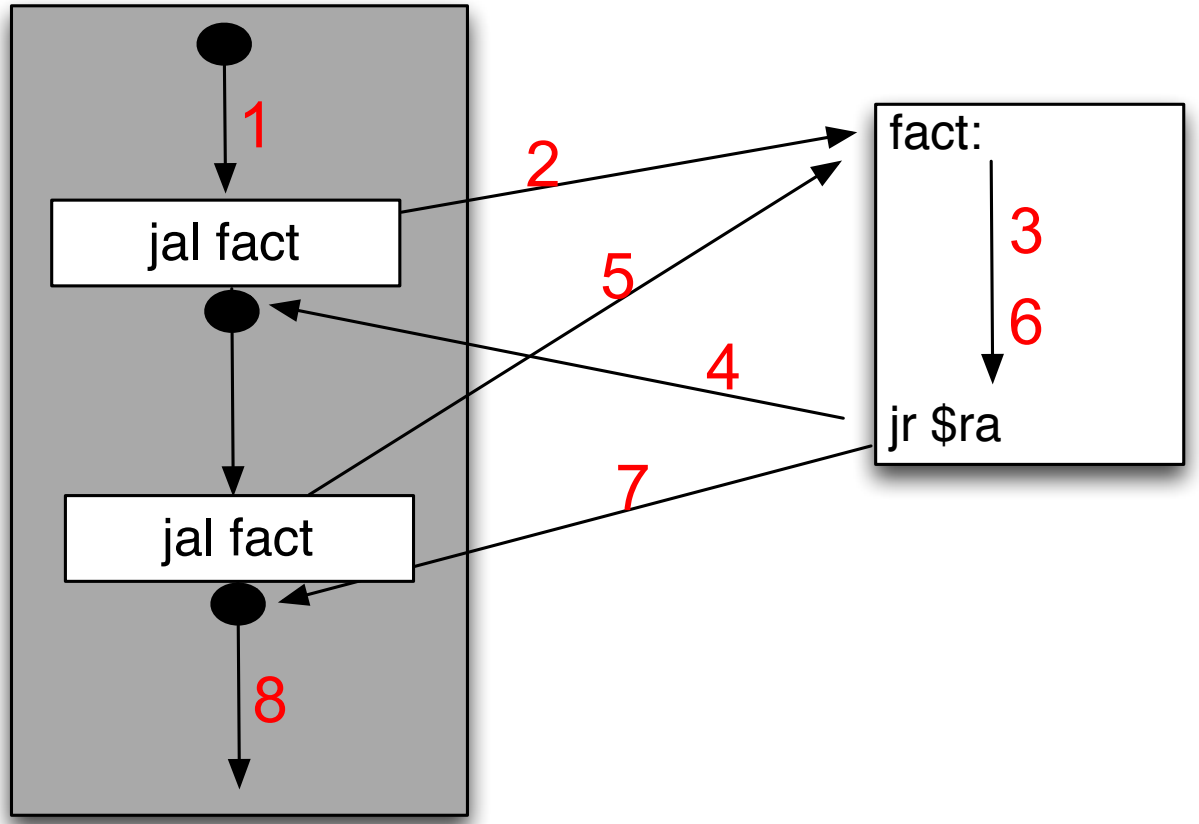
Concepto de procedimiento o función

- Las ***subrutinas*** son el equivalente que ofrecen las instrucciones máquina a los *procedimientos y funciones* de los lenguajes de alto nivel
- En C, una función proporciona varias abstracciones:
  - Mapeo de parámetros actuales a parámetros formales
  - Alojamiento e inicialización del almacenamiento local temporal. Esto es importante en lenguajes que permiten **recursión**: cada llamada a la misma función debe obtener su propia copia de cualquier variable local para evitar que los datos se sobrescriban entre las distintas llamadas

Contexto de una función

- El contexto de una función es toda aquella información que describe el estado de una función durante su ejecución:
  - parámetros actuales
  - valor de las variables locales
  - instrucción que está siendo ejecutada
- Para un programa en lenguaje ensamblador MIPS, el contexto de una función consiste en los valores de todos los registros a los que se hace referencia desde la función
- La mayoría de las arquitecturas, incluida la de MIPS, utiliza el segmento de memoria denominado ***pila*** como medio para salvar y restaurar el contexto de una subrutina

Invocación



- Seguimiento del flujo de ejecución de un programa con llamadas a subrutinas
- ¿Qué valores va tomando el registro PC?

Instrucciones para la llamada y el retorno de subrutina

- **`jal`** `label`: *llamada a subrutina*
  - Salto incondicional a la instrucción objetivo apuntada por la etiqueta *label*. Carga el registro PC con dicha dirección
  - Además, se guarda la dirección de la siguiente instrucción al `jal` en el registro `$ra`
- **`jr`** `$ra`: *retorno de subrutina*
  - Salto incondicional a la instrucción cuya dirección está almacenada en el registro `$ra`. Carga el registro PC con dicha dirección.

- IMPORTANTE: para los programas a realizar sólo se necesitarán estas dos instrucciones tipo J.

Pasos llamada a subrutina (1)

1. El *llamante* debe:
   a) Poner los argumentos de entrada de la subrutina en los registros **`$a0-$a3`**. Si hay más de 4 parámetros, se deben pasar por la pila
   b) Salvar los registros **`$a0-$a3`** y **`$t0-$t9`** cuyos valores se quieran preservar entre llamadas. Si dichos registros no contienen valores que queramos preservar, no hay que guardarlos en la pila.

      OJO: no vale usar en las subrutinas otros registros temporales distintos a los del llamante con la intención de no tener que guardarlos en pila.
   c) Ejecutar la instrucción `jal`, la cual salta a la primera instrucción de la subrutina y almacena la dirección de retorno en el registro `$ra`.

Nota: el llamante no debe guardar en la pila los registros estáticos, esto es, los registros $s0-$s7 pues estos se deben preservar entre llamadas. Cualquier *llamado* que quiera usarlos, antes debe guardarlos en la pila.

Pasos llamada a subrutina (2)

2. El *llamado* (la subrutina) debe:
   a) Crear un marco de pila, restando el tamaño del marco del puntero de pila (`$sp`). La pila crece hacia abajo, de dirección mayor a dirección menor. El tamaño mínimo del marco de pila es de 32 bytes
   b) Salvar los registros **`$s0-$s7, $fp`** y **`$ra`** que vayan a ser modificados por la subrutina. Si no se modifican, no es necesario guardarlos.
   c) Establecer el valor del puntero de marco de pila (`$fp`) sumando al puntero de pila (`$sp`) el tamaño del marco de pila menos 4. De esta manera apuntamos a la *cima del marco de pila* y podemos escribir una palabra sin salirnos del marco pila. Cada subrutina debe usar su registro `$fp` para moverse por su marco de pila.
   d) El segmento de memoria pila recibe este nombre porque se comporta como la estructura de datos ***pila***. Es decir, los datos más recientes se almacenan en la cima de la pila y los menos utilizados, los últimos en salir, en la parte baja de la pila. De ahí que los registros `$fp` y `$ra` se almacenen en la parte baja de la pila.

Pasos llamada a subrutina (3)

3. El *llamado* (la subrutina) ejecuta el cuerpo de la función. Para retornar de la función:
   a) Si devuelve un valor, lo debe meter en el registro **`$v0`**
   b) Restaura los registros que hubiera salvado en la pila (`$s0-$s7, $fp, $ra`)
   c) Libera el marco de pila añadiendo el tamaño del marco al puntero de pila
   d) Retornar saltando con **`jr`** a la dirección indicada en el registro **`$ra`**

Pasos llamada a subrutina (4)

4. Para restablecer el estado anterior a la llamada a la subrutina, el *llamante* debe:
   a) Restaurar los registros (`$a0-$a3, $t0-$t9`) que guardó en la pila antes de la llamada a la subrutina
   b) Si espera un valor de retorno, lo debe extraer del registro `$v0`

Paso de parámetros

- Tipo de parámetros:
  - **Por valor**: se pasa el valor del parámetro
  - **Por referencia**: se pasa la dirección de memoria donde esté almacenado el parámetro
- Lugar donde se van a pasar los parámetros:
  - Registros (`$a0-$a3`)
  - Pila
- El ensamblador del MIPS establece el siguiente convenio para realizar el paso de parámetros:
  - Los 4 primeros parámetros de entrada se pasarán a través de los registros (`$a0-$a3`). A partir del quinto parámetro se pasará a través de la pila
  - Los valores de retorno se devuelve en los registros `$v0` y `$v1`, el resto a través de la pila

Ejemplo: cálculo del factorial del número 10

Factorial en C

```
main ()
{
   printf ("The factorial of 10 is %d\n", fact (10));
}
int fact (int n)
{
   if (n < 1)
     return (1);
   else
     return (n * fact (n - 1));
}
```

Factorial en ensamblador (main)

```
## Arquitectura de Computadores curso 2016-2017
## fact.asm -- A program to compute the factorial of 10
## main--
## Registers used:
##      $a0         - syscall parameter -- the number to print
##      $v0         - syscall parameter and return value

          .data
msg:      .asciiz "The factorial of 10 is: "

          .text
main:     la $a0, msg
          li $v0, 4           # Load syscall print-string into $v0
          syscall             # Make the syscall

          li $a0,3            # Put argument (3) in $a0
          jal fact            # Call factorial function

          move $a0, $v0       # Move fact result in $a0
          li $v0, 1           # Load syscall print-int into $v0
          syscall             # Make the syscall

          li $v0, 10                  # Load syscall exit into $v0
          syscall             # Make the syscall
```

Factorial en ensamblador (fact)

```
## fact
## Registers used:
##      $a0
fact:
          subu $sp, $sp, 32    # Stack frame is 32 bytes long
          sw $ra, 20($sp)      # Save return address
          sw $fp, 16($sp)      # Save frame pointer
          addiu $fp, $sp, 28   # Set up frame pointer
          sw $a0, 0($fp)       # Save argument (n)

          lw  $v0, 0($fp)
          bgtz $v0, fact_recur # fact(n-1)
          li   $v0, 1
          b return_fact
fact_recur:
          lw $v1, 0($fp)       # Load n
          subu $v0, $v1, 1     # Compute n - 1
          move $a0, $v0        # Move value to $a0
          jal fact

          lw $v1, 0($fp)       # Load n
          mul $v0, $v0, $v1    # Compute fact(n-1) * n
return_fact:
          lw $ra, 20($sp)
          lw $fp, 16($sp)
          addiu $sp, $sp, 32
          jr $ra
```