Universidad Rey Juan Carlos

Arquitectura de Computadores/Arquitectura de Sistemas Audiovisuales II

Práctica 4: Llamada a subrutina

Katia Leal Algara

A. Versión optimizada factorial: algoritmo recursivo.

Implementa una versión optimizada de la subrutina *fact* que encontrarás en la última transparencia del "Convenio de Llamada a Subrutina". Se trata de reducir el número total de instrucciones, sobre todo, el número de instrucciones de acceso a memoria. Por ejemplo, cuando se cumple la condición de parada no es necesario crear pila, puesto que esa parte del código se limita a retornar un 1. Si no creamos pila, tampoco tenemos que liberarla, y con ello, nos ahorramos un buen número de instrucciones y de instrucciones de acceso a memoria.

B. Cálculo de la secuencia de Fibonacci: algoritmo recursivo

La secuencia de Fibonacci tiene la siguiente definición recursiva: sea F(n) el n-ésimo elemento (donde n >= 0) en la secuencia:

- Si n < 2, entonces F(n) = 1 (caso base)
- En otro caso, F(n) = F(n - 1) + F(n - 2) (caso recursivo)

Siguiendo el convenio de llamada a subrutina del MIPS, implementa un programa en ensamblador que pida un número por el terminar y que calcule, mediante una función recursiva, el valor de la secuencia de Fibonacci correspondiente a dicho número. En la medida de lo posible, se debe implementar el algoritmo más óptimo salvando el menor número de registros posible. De esta manera, conseguiremos que el cambio de contexto sea lo más ligero posible y que, por lo tanto, el tiempo de ejecución sea menor.

C. Cálculo de la secuencia de Fibonacci: algoritmo iterativo

Implementa un programa en ensamblador que realice el cálculo de la secuencia de Fibonacci de un número introducido por el terminal, pero en este caso se debe utilizar una subrutina que realice el cálculo con un algoritmo iterativo.

Compara el tiempo de ejecución de esta versión con la versión recursiva. Para

hacerlo, debes ejecutar en el terminal los siguientes comandos:

```
epsilon05:~$ time java -jar /usr/local/mars/Mars_4_2.jar recursivo.asm
epsilon05:~$ time java -jar /usr/local/mars/Mars_4_2.jar iterativo.asm
```

D. atoi-4.asm

Modifica el programa atoi-4.asm de tal manera que incorpore una función atoi que se comporte de la misma manera que la función de librería de C. La función debe cumplir con el convenio de llamada a subrutina de MIPS. Si no puede convertir la cadena porque contiene caracteres no numéricos, debe retornar un 0. En caso de desbordamiento, devuelve el número calculado hasta el momento.

E. palindromo.asm

Modifica el programa palíndromo.asm para que incluya al menos dos funciones:

- test_palindrome(char *) devuelve un 1 en caso de que la cadena sea un palíndromo y 0 en caso contrario.
- main() lee una línea de texto de la consola, por medio de la función test_palindrome() determina si es palíndroma y por último imprime el mensaje correspondiente.

F. Número primos

Escribe un programa que imprima los 100 primeros números primos. Un número primo es un número natural mayor que 1 que tiene únicamente dos divisores distintos: él mismo y el 1. Debes implementar dos rutinas:

- test_prime(n) devuelve 1 en caso de que el número sea primo y 0 en caso contrario.
- main() itera sobre números enteros comprobando si son primos. Imprime los 100 primeros números que sean primos.

G. Torre de Hanoi

Implementa un programa ensamblador recursivo que resuelva el problema de las Torres de Hanoi. El juego, en su forma más tradicional, consiste en tres

varillas verticales. En una de las varillas se apila un número indeterminado de discos que determinará la complejidad de la solución, por regla general se consideran ocho discos. Los discos se apilan sobre una varilla en tamaño decreciente. No hay dos discos iguales, y todos ellos están apilados de mayor a menor radio en una de las varillas, quedando las otras dos varillas vacantes. El juego consiste en pasar todos los discos de la varilla ocupada (es decir la que posee la torre) a una de las otras varillas vacantes. Para realizar este objetivo, es necesario seguir tres simples reglas:

1. Sólo se puede mover un disco cada vez.
2. Un disco de mayor tamaño no puede descansar sobre uno más pequeño que él mismo.
3. Sólo puedes desplazar el disco que se encuentre arriba en cada varilla.

Existen diversas formas de realizar la solución final, todas ellas siguiendo estrategias diversas. A continuación se muestra un programa en C que se puede utilizar como ayuda para escribir el programa en ensamblador:

```
/* move n smallest disks from start to finish using extra */
void hanoi(int n, int start, int finish, int extra)
{
      if(n != 0)
      {
          hanoi(n-1, start, extra, finish);
          print_string("Move disk");
          print_int(n);
          print_string("from peg");
          print_int(start);
          print_string("to peg");
          print_int(finish);
          print_string(".\n");
          hanoi(n-1, extra, finish, start);
       }
}

main()
{
    int n;
    print_string("Introduce el número de discos (1-8)>");
    n = read_int();
    hanoi(n, 1, 2, 3);
    return 0;
}
```