Análisis Comparativo: Practica 1- Fibonacci recursivo e iterativo

Orlando Bonilla 30869545 - Enyerber Flores 27157754

08 de Julio

De forma Recursiva

```
.data
entrada: .asciiz "Ingrese n: "
result: .asciiz "Fibonacci(n) = "

.text
.globl main

main:

    li $v0, 4
    la $a0, entrada
    syscall

    li $v0, 5
    syscall
    move $a0, $v0


    jal fibo_recursivo

    move $t0, $v0

    li $v0, 4
    la $a0, result
    syscall

    li $v0, 1
    move $a0, $t0
    syscall

    li $v0, 10
    syscall

fibo_recursivo:

    beqz $a0, fibo

    li $v0, 1
    beq $a0, 1, f_fibo

    addi $sp, $sp, -12
    sw $ra, 0($sp)
    sw $a0, 4($sp)
```

```
    addi $a0, $a0, -1
    jal fibo_recursivo
    sw $v0, 8($sp)

    lw $a0, 4($sp)
    addi $a0, $a0, -2
    jal fibo_recursivo


    lw $t0, 8($sp)
    add $v0, $t0, $v0

    lw $ra, 0($sp)
    addi $sp, $sp, 12

    j f_fibo

fibo:
    li $v0, 0
    j f_fibo

f_fibo:
    jr $ra
```

De forma Iterativa

```
.data
entrada: .asciiz "Ingrese n: "
result: .asciiz "Fibonacci(n) = "

.text
.globl main

main:
    li $v0, 4
    la $a0, entrada
    syscall

    li $v0, 5
    syscall
    move $a0, $v0

    jal fibo

    move $t0, $v0
    li $v0, 4
    la $a0, result
    syscall
    li $v0, 1
    move $a0, $t0
    syscall
    li $v0, 10
    syscall

fibo:
    li $v0, 1
    ble $a0, 1, f_fibo
```

```
    li $t0, 0
    li $t1, 1
    li $t2, 2

ciclo:
    add $t3, $t0, $t1
    move $t0, $t1
    move $t1, $t3
    addi $t2, $t2, 1
    ble $t2, $a0, ciclo

    move $v0, $t1

f_fibo:
    jr $ra
```

Respuestas

1. Implementación de recursividad en MIPS32

La recursividad en MIPS32 se implementa mediante llamadas a la misma función usando `jal`. La pila (`$sp`) juega un papel fundamental en este proceso, ya que permite almacenar el contexto de cada llamada recursiva.

Antes de cada llamada recursiva, el programa debe guardar en la pila: - La dirección de retorno (`$ra`) para saber dónde continuar la ejecución - Los valores actuales de los registros que se modificarán - Cualquier dato temporal necesario para cálculos posteriores

Después de completar la llamada recursiva, el programa restaura estos valores desde la pila antes de continuar con la ejecución. Este mecanismo permite que cada instancia de la función mantenga su propio contexto independiente.

2. Riesgos de desbordamiento y mitigación

El principal riesgo en la implementación recursiva es el desbordamiento de pila (stack overflow). Este ocurre cuando el número de llamadas recursivas excede la capacidad de memoria asignada a la pila.

Cada llamada recursiva consume aproximadamente 12 bytes de espacio en la pila. Para valores grandes de n (n » 30), el consumo de memoria puede volverse significativo y provocar un desbordamiento.

Para mitigar este riesgo:
1. Utilizar la implementación iterativa para valores de n mayores a 25.
2. Aumentar el tamaño de la pila en MARS mediante `Settings → Memory Configuration`.
3. Implementar técnicas de optimización como memoization para evitar cálculos redundantes.
4. Considerar el uso de registros preservados cuando sea posible.
5. Limitar la profundidad recursiva mediante comprobaciones adicionales.

3. Diferencias en uso de memoria y registros

La implementación recursiva utiliza intensivamente la pila para almacenar el contexto de cada llamada. Cada invocación guarda el registro de retorno (`$ra`), el valor actual de n (`$a0`) y el resultado parcial, consumiendo 12 bytes por llamada.

Este enfoque resulta en un uso de memoria O(n) que crece linealmente con el tamaño de la entrada. Además, requiere numerosas operaciones de guardado y restauración de registros, lo que aumenta la complejidad de la implementación.

En contraste, la implementación iterativa utiliza un enfoque de bucle que mantiene solo cuatro registros (`$t0`, `$t1`, `$t2`, `$t3`) durante toda su ejecución. Su uso de memoria es constante (O(1)) ya que no depende del tamaño de n y no utiliza la pila para almacenamiento temporal.

La versión iterativa opera directamente sobre registros temporales sin necesidad de preservar contexto entre iteraciones, lo que simplifica el diseño y reduce el número de operaciones de memoria.

4. Diferencias con ejemplos académicos

Los ejemplos académicos típicos presentan versiones simplificadas que difieren de implementaciones reales en varios aspectos clave:

Los libros de texto generalmente muestran pseudocódigo en lugar de código ejecutable completo, omitiendo elementos esenciales para un programa funcional como el manejo de entrada/salida y las instrucciones de terminación.

Además, las implementaciones académicas suelen simplificar el manejo de la pila, ignorando aspectos importantes como la alineación de memoria y la protección completa de registros según las convenciones de llamada estándar de MIPS32.

Nuestra implementación incluye todos los componentes necesarios para un programa completamente funcional en MARS, incluyendo la interfaz de usuario, gestión completa de la pila, protección de registros y manejo adecuado de casos base.

5. Tutorial de Ejecución en MARS

Configuración inicial

1. Iniciar el simulador MARS 2. Crear un nuevo archivo mediante `File → New` 3. Seleccionar y copiar el código deseado (recursivo o iterativo) 4. Pegar el código en el editor de MARS

Ensamblado y ejecución

1. Ensamblar el programa con `Run → Assemble` (F3) 2. Verificar que no haya errores en la consola inferior 3. Ejecutar el programa con `Run → Go` (F5) 4. En la consola, ingresar el valor de n cuando se solicite 5. Observar el resultado del cálculo Fibonacci

6. Justificación del enfoque

El enfoque recursivo ofrece ventajas conceptuales al alinearse directamente con la definición matemática de Fibonacci, resultando en código más compacto y fácil de entender para valores pequeños de n.

Sin embargo, su complejidad exponencial ($O(2^n)$) y consumo lineal de memoria ($O(n)$) lo hacen poco práctico para valores mayores.

El enfoque iterativo presenta complejidad lineal (O(n)) tanto en tiempo como en espacio, con uso constante de memoria (O(1)). Esto elimina el riesgo de desbordamiento de pila y permite calcular valores significativamente mayores con recursos mínimos.

Para MIPS32, donde la gestión manual de memoria y la eficiencia son críticas, el enfoque iterativo es claramente superior para cómputo real. El recursivo mantiene valor pedagógico para enseñar recursión y manejo de pila, pero debe limitarse a valores pequeños de n.

7. Análisis de resultados

Al probar ambos programas en MARS, notamos diferencias claras:

a) Para números pequeños (n=5 o n=10):

-Ambos dan el mismo resultado correcto (5 para n=5, 55 para n=10)

-El recursivo tarda un poco más pero es aceptable

-El iterativo es casi instantáneo

b) Para números medianos (n=15 a n=20):

-El recursivo se vuelve muy lento (varios segundos)

-El iterativo sigue siendo instantáneo

-El recursivo usa mucha memoria de la pila

-El iterativo mantiene el mismo bajo consumo de memoria

c) Para números grandes (n » 30):

- El recursivo falla por desbordamiento de pila

- El iterativo sigue funcionando perfectamente

- Ambos pueden dar error numérico si n » 47 (límite de enteros)