Prueba 1 Laboratorio en MIPS32

Jesus Herrada CI. 30 647 452

July 7, 2025

Este informe explora la implementación de la recursividad en la arquitectura MIPS32, destacando el rol fundamental de la pila en este proceso. Se analizan los riesgos de desbordamiento asociados y las estrategias para mitigarlos. Asimismo, se compara la eficiencia en el uso de memoria y registros entre implementaciones iterativas y recursivas. Se incluye un tutorial paso a paso para la ejecución en MARS y se justifica la elección de un enfoque (iterativo o recursivo) basado en la eficiencia y claridad del código. Finalmente, se presenta un análisis y discusión de los resultados obtenidos.

1 Introducción

La recursividad es una técnica de programación poderosa donde una función se llama a sí misma para resolver un problema, dividiéndolo en subproblemas más pequeños de la misma naturaleza. En arquitecturas de bajo nivel como MIPS32, la implementación de la recursividad requiere una comprensión profunda del manejo de la pila y los registros del procesador. Este informe detalla cómo se logra la recursividad en MIPS32 y aborda aspectos críticos relacionados con su uso.

2 Implementación de la Recursividad en MIPS32 y el Rol de la Pila

La recursividad en MIPS32 se implementa a través de la gestión eficiente de la **pila del sistema**. Cuando una función se llama a sí misma (o a otra función), se requiere un mecanismo para guardar el estado actual de la función que llama y restaurarlo cuando la función llamada termina. Este es precisamente el papel de la pila.

- **Almacenamiento del Contexto:** Antes de una llamada a una función recursiva, los registros vitales que la función actual necesitará después de la llamada (como la dirección de retorno en `$ra` y los registros `$s0-$s7` que contienen valores guardados) se guardan en la pila. Esto se logra decrementando el puntero de pila (`$sp`) para asignar espacio y luego almacenando los valores.

- **Paso de Parámetros:** Aunque MIPS utiliza los registros `$a0-$a3` para pasar los primeros cuatro argumentos, si hay más argumentos o si los argumentos deben persistir a través de llamadas recursivas anidadas, también se pueden almacenar en la pila.
- **Variables Locales:** Las variables locales de una función que no se ajustan a los registros temporales (`$t0-$t9`) o que deben persistir a través de llamadas a subrutinas también se almacenan en la pila.
- **Puntero de Pila (`$sp`):** El registro `$sp` apunta a la cima de la pila. Para asignar espacio en la pila, `$sp` se decrementa. Para liberar espacio, `$sp` se incrementa. Es crucial que el puntero de pila se mantenga correctamente para evitar corrupción de datos.

El proceso típico para una función recursiva en MIPS es el siguiente:

1. **Prólogo:**
   - Se guarda el registro de la dirección de retorno (`$ra`) en la pila.
   - Se guardan los registros $s (registros guardados por la función que llama) que serán modificados.
   - Se ajusta el puntero de pila (`$sp`) para reservar espacio para variables locales y otros datos.
2. **Cuerpo de la Función:**
   - Se realizan las operaciones de la función, incluyendo la llamada recursiva si es necesario.
   - Los argumentos para la llamada recursiva se colocan en los registros `$a0-$a3` o en la pila.
   - Se realiza la llamada con `jal` (jump and link).
3. **Epílogo:**
   - Se restauran los registros $s que fueron guardados.
   - Se restauran el registro de la dirección de retorno (`$ra`).
   - Se ajusta el puntero de pila (`$sp`) para liberar el espacio asignado.
   - Se regresa al llamador con `jr $ra`.

3 Riesgos de Desbordamiento y Mitigación

El principal riesgo al usar recursividad es el **desbordamiento de pila (stack overflow)**. Esto ocurre cuando el número de llamadas recursivas es tan grande que la pila se queda sin espacio en la memoria asignada. Cada llamada recursiva consume una porción de la pila para almacenar la dirección de retorno, los parámetros y las variables locales. Si el problema no tiene una condición de

parada adecuada o si la profundidad de la recursión es excesiva, la pila crecerá indefinidamente hasta consumir toda la memoria disponible, lo que resultará en un fallo del programa.

Mitigación de Riesgos:

1. **Condición Base Adecuada:** Asegurarse de que toda función recursiva tenga una condición base claramente definida y alcanzable que detenga la recursión. Sin una condición base, la recursión sería infinita y garantizada a desbordar la pila.

2. **Optimización de Cola (Tail Recursion Optimization - TRO):** En algunos lenguajes de programación y compiladores, la recursión de cola puede ser optimizada para no consumir espacio adicional en la pila. Una llamada recursiva es de cola si es la última operación en la función antes de retornar. MIPS no tiene soporte directo para TRO a nivel de instrucción, pero los compiladores avanzados pueden convertir llamadas recursivas de cola en iteraciones.

3. **Limitar la Profundidad de Recursión:** Si es posible, imponer un límite máximo a la profundidad de la recursión. Si se excede este límite, el programa puede manejar la situación de manera elegante (por ejemplo, lanzando una excepción o retornando un error) en lugar de fallar por desbordamiento de pila.

4. **Convertir a Iteración:** Para muchos problemas, una solución recursiva puede ser reescrita como una solución iterativa utilizando bucles. Las soluciones iterativas generalmente no sufren de desbordamiento de pila ya que no apilan múltiples marcos de función. Esta es la estrategia más robusta en entornos con recursos de pila limitados.

5. **Aumentar el Tamaño de la Pila:** En algunos sistemas operativos o entornos de ejecución, es posible configurar un tamaño de pila más grande para el programa. Sin embargo, esta no es una solución escalable y solo pospone el problema si la recursión es muy profunda.

6. **Manejo Explícito de Estructuras de Datos (simulando una pila):** Para problemas que inherentemente requieren una estructura de tipo pila (como el recorrido de árboles), pero que son propensos a desbordamiento con recursión implícita, se puede implementar explícitamente una pila usando un arreglo o lista enlazada. Esto da al programador un control más granular sobre el uso de memoria.

4 Diferencias entre Implementación Iterativa y Recursiva

La elección entre una implementación iterativa y una recursiva tiene implicaciones significativas en el uso de memoria y registros en MIPS32.

Uso de Memoria (Pila):

- **Recursiva:** Cada llamada recursiva crea un nuevo marco de pila que almacena la dirección de retorno, los parámetros de la función y las variables locales. Esto puede consumir una cantidad considerable de memoria de la pila, especialmente para problemas con una gran profundidad de recursión. El uso de memoria es proporcional a la profundidad de la recursión.
- **Iterativa:** Las implementaciones iterativas generalmente utilizan una cantidad constante o predecible de memoria, ya que no se acumulan marcos de pila con cada "iteración" del problema. El bucle simplemente reutiliza los mismos registros y variables. El uso de memoria es, en la mayoría de los casos, mucho menor y más predecible.

Uso de Registros:

- **Recursiva:** Las funciones recursivas a menudo necesitan guardar y restaurar registros en la pila (especialmente `$ra` y los registros $s) para preservar el estado de la función que llama. Esto implica un overhead de instrucciones de carga y almacenamiento, lo que puede afectar el rendimiento.
- **Iterativa:** Las implementaciones iterativas tienden a hacer un uso más eficiente de los registros, ya que no hay necesidad de guardar y restaurar el contexto de llamadas anidadas. Los registros pueden ser reutilizados dentro del bucle sin la sobrecarga de la pila. Esto puede resultar en un código más rápido debido a menos accesos a memoria.

En resumen, las implementaciones iterativas suelen ser más eficientes en términos de uso de memoria y registros en MIPS32, ya que evitan la sobrecarga de la gestión de la pila asociada con cada llamada recursiva. Sin embargo, la recursividad puede ofrecer una mayor claridad para ciertos problemas.

5 Diferencia entre los ejemplos academicos del libro y un ejercicio completo y operativo en mips32

si bien los ejemplos proporcionados por el libro son bastantes completos , existen algunas diferencias con los ejercicios completos y operativos usados en mips32:

- **Declaracion de variables y directivas (etiquetas globales):** La mayoria de los ejercicios o código que proporciona de como implementar mips son solo fragmentos puntuales de dicho código , que esta claro pues el objetivo es centrarse en como se implementa solamente esa fracción del código sin embargo, esto puede ser confuso en ciertas situaciones si se es nuevo en el lenguaje pues no se tiene idea desde donde es invocada la función, cuales son sus parámetros de retorno y de salida , aunque estos se especifiquen en comentarios antes de iniciar el fragmento de código. fuera

de esto los ejercicios que se muestran en el libro son bastante practicos y con un buen enfoque, aunque seria bueno que el libro implementase en ciertas ocasiones un codigo completo desde la declaración de variables con .data y luego la ejecución del código desde el main y de alli ir describiendo el resto del codigo.

- **Explicación por cada instrucción:** Uno de los principales puntos fuertes que tiene el libro a la hora de explicar algún ejercicio o fragmento de código es que comenta y explica cada ejecucion , como : acceso a direcciones de memoria, asignación , desplazamiento de bytes , carga de valores y muchas otras, lo cual es una gran ventaja por que permite tener una idea de que es lo que hace dicha instrucción de mips y aunque en ocasiones no paresca relevante , debido al conocimiento previo de como funcionan dichas instrucciones en otro lenguaje como C , la verdad es que es un buen punto de partida para familiarizarse con el lenguaje y entnder como se ejecuta internamente cada instrucción.

6 Tutorial de Ejecución Paso a Paso en MARS

MARS (MIPS Assembler and Runtime Simulator) es una excelente herramienta para entender la ejecución de código MIPS, incluyendo la recursividad.

1. **Abrir MARS:** Inicia la aplicación MARS.
2. **Crear un nuevo archivo:** Hacer click en "file"y crear una nueva hoja en blanco y guarda "save as".
3. **Implementar el código:**
4. **Conceptos básicos:** La secuencia de Fibonacci es: .F(0)=0 .F(1)=1 .F(n)= F(n-1) + F(n-2) para n >1

```
.data
# No se definen datos específicos aquí, pero se podría añadir
# si se quieren imprimir mensajes o el resultado de forma más elaborada.

.text
.globl main

main:
# --- Ejemplo de uso: Calcular fibonacci(10) ---
li $a0, 10       # Cargar el valor de 'n' (10 en este caso) en $a0
jal fibonacci    # Llamar a la función fibonacci. El resultado estará en $v0

# --- Imprimir el resultado (ejemplo para SPIM) ---
# Si estás usando SPIM, puedes descomentar las siguientes líneas para ver el res
# li $v0, 1        # Servicio de sistema para imprimir un entero
```

```
# move $a0, $v0    # Mover el resultado de $v0 a $a0 (argumento para la impresió
# syscall          # Ejecutar la llamada al sistema (imprime el valor de $a0)

# --- Salir del programa (para SPIM) ---
li $v0, 10       # Servicio de sistema para salir del programa
syscall          # Ejecutar la llamada al sistema


# ------------------------------------------------------------------
# fibonacci(n)
#
# Calcula el n-ésimo número de Fibonacci de forma iterativa.
#
# Argumentos:
#   $a0: n (el índice del número de Fibonacci a calcular)
#
# Retorno:
#   $v0: El n-ésimo número de Fibonacci
#
# Registros usados y salvados (callee-saved):
#   $s0 (para 'a')
#   $s1 (para 'b')
#   $ra (dirección de retorno)
# ------------------------------------------------------------------
fibonacci:
# Prológo de la función: Guardar registros en la pila
# Reservar espacio para $ra, $s0, $s1 (3 palabras = 12 bytes)
addi $sp, $sp, -12
sw $ra, 8($sp)      # Guardar la dirección de retorno
sw $s0, 4($sp)      # Guardar el registro $s0 (que usaremos para 'a')
sw $s1, 0($sp)      # Guardar el registro $s1 (que usaremos para 'b')

# --- Manejo de casos base: if n <= 1 ---
# $a0 contiene 'n'
addi $t0, $zero, 1  # $t0 = 1
sle $t1, $a0, $t0   # $t1 = 1 si n <= 1, 0 en caso contrario (Set Less Equal)
bne $t1, $zero, handle_base_case # Si $t1 es distinto de cero (n <= 1), saltar a

# --- Inicialización de variables para el bucle ---
# a = 0
# b = 1
add $s0, $zero, $zero # $s0 = 0 (representa 'a')
addi $s1, $zero, 1    # $s1 = 1 (representa 'b')

# Inicialización del contador del bucle: i = 2
addi $t1, $zero, 2    # $t1 = 2 (representa 'i')
```

```
# --- Bucle principal: for i from 2 to n ---
loop_start:
# Condición del bucle: i <= n
# Si i > n, salir del bucle. Esto es equivalente a preguntar si n < i.
slt $t2, $a0, $t1   # $t2 = 1 si $a0 (n) < $t1 (i), 0 en caso contrario (Set Les
bne $t2, $zero, loop_end # Si $t2 es distinto de cero (n < i), salta a loop_end

# Cuerpo del bucle:
# temp = a + b
add $t0, $s0, $s1   # $t0 = $s0 + $s1  ('temp' = 'a' + 'b')

# a = b
move $s0, $s1       # $s0 = $s1        ('a' = 'b')

# b = temp
move $s1, $t0       # $s1 = $t0        ('b' = 'temp')

# i++
addi $t1, $t1, 1    # $t1 = $t1 + 1    ('i'++)

j loop_start        # Saltar al inicio del bucle para la siguiente iteración

loop_end:
# El resultado final está en 'b' ($s1)
move $v0, $s1       # Mover el resultado a $v0 (registro de valor de retorno)

# Epílogo de la función: Restaurar registros y regresar
lw $ra, 8($sp)      # Restaurar la dirección de retorno
lw $s0, 4($sp)      # Restaurar $s0
lw $s1, 0($sp)      # Restaurar $s1
addi $sp, $sp, 12   # Ajustar el puntero de pila (liberar espacio)
jr $ra              # Regresar al llamador

# --- Manejador del caso base ---
handle_base_case:
# Si n <= 1, el resultado de Fibonacci es n mismo
move $v0, $a0       # Mover n (que está en $a0) a $v0

# Epílogo de la función para el caso base: Restaurar registros y regresar
lw $ra, 8($sp)      # Restaurar la dirección de retorno
lw $s0, 4($sp)      # Restaurar $s0
lw $s1, 0($sp)      # Restaurar $s1
addi $sp, $sp, 12   # Ajustar el puntero de pila
jr $ra              # Regresar al llamador
```

5. **Ensamblar el Código:** Hacer clic en el botón "**Assemble**" (el martillo azul). Si hay errores de sintaxis, MARS los mostrará en la ventana "Messages".

6. **Vista de la Pila y Registros:** En la parte inferior de la ventana de MARS, asegúrarse de tener las pestañas "**Registers**" y "**Data Segment**" visibles. La pestaña "Data Segment" permitirá ver el contenido de la pila (Stack) en caso de que la función sea recursiva.

7. **Ejecución Paso a Paso:**

   - Hacer clic en el botón "**Step**" (el icono de una flecha hacia abajo). Esto ejecutará una instrucción a la vez.
   - Se puede observa cómo los valores de los registros (`$sp`, `$ra`, `$a0`, `$v0`, etc.) cambian con cada paso.
   - Cuando el programa entra en la función `fibonacci`.

8. **Puntos de Ruptura (Breakpoints):** Para acelerar la depuración, puedes establecer puntos de ruptura haciendo clic en el margen izquierdo junto a la línea de código. Cuando ejecutes el programa, se detendrá en el punto de ruptura, permitiéndote inspeccionar el estado de la máquina.

7 Justificación de la Elección del Enfoque (Iterativo o Recursivo) en MIPS32

La elección entre un enfoque iterativo y uno recursivo en MIPS32 depende de varios factores, principalmente la **eficiencia** y la **claridad** del código.

Eficiencia:

- **Iterativo (Generalmente Preferido en MIPS32):** Las implementaciones iterativas son casi siempre más eficientes en MIPS32 en términos de velocidad y uso de memoria. Esto se debe a que evitan la sobrecarga de la gestión de la pila (guardado y restauración de registros, ajuste del puntero de pila) por cada "iteración" o llamada recursiva. Un bucle simplemente salta a una etiqueta, lo que es mucho más rápido que una secuencia de `jal`, `sw`, `lw` y `jr`. Para problemas que pueden resolverse eficientemente de forma iterativa (como el factorial o la serie de Fibonacci sin memorización), el enfoque iterativo minimiza el número de instrucciones y el uso de memoria.

- **Recursivo:** La recursividad introduce una **sobrecarga significativa** debido a la manipulación de la pila. Cada llamada de función requiere una serie de operaciones para guardar el estado del llamador y restaurarlo al regresar. Esto consume ciclos de CPU y espacio en la pila, lo que puede ser crítico en sistemas embebidos o con memoria limitada. Además, el riesgo de desbordamiento de pila es una preocupación constante. Si un

problema tiene una estructura inherentemente recursiva y la profundidad de la recursión es pequeña y acotada, la recursión puede ser aceptable.

Claridad del Código:

- **Recursivo:** Para ciertos problemas, especialmente aquellos que tienen una definición matemática recursiva natural (como los algoritmos de árbol, búsqueda en profundidad, o la función factorial de forma elegante), la implementación recursiva puede ser significativamente **más clara y concisa**. El código a menudo refleja directamente la definición del problema, lo que facilita su comprensión y verificación. Por ejemplo, el factorial de $n! = n \times (n-1)!$ se traduce directamente en una función recursiva simple.
- **Iterativo:** Si bien es más eficiente, una implementación iterativa para un problema inherentemente recursivo puede requerir el uso de una pila explícita (si se está convirtiendo una recursión no de cola) o puede oscurecer la lógica subyacente del algoritmo, haciéndolo más difícil de leer y mantener. Para problemas que son naturalmente iterativos (como recorrer un arreglo), el código iterativo es, por supuesto, más claro.

Justificación: En el contexto de MIPS32, donde el control de recursos y la eficiencia son primordiales, **el enfoque iterativo es generalmente la elección preferida** a menos que la claridad del código para un problema específico sea abrumadoramente mejor con la recursividad y se tenga la certeza de que la profundidad de la recursión será mínima y no causará desbordamiento de pila. Para aplicaciones de alto rendimiento o sistemas con recursos limitados, la eficiencia del enfoque iterativo es un factor decisivo. La complejidad de implementar y depurar la gestión de pila en código ensamblador para recursiones profundas también inclina la balanza hacia la iteración.

8 Análisis y Discusión de los Resultados

Al analizar las implementaciones recursivas e iterativas en MIPS32, se observan los siguientes resultados:

- **Uso de Recursos:** Las pruebas en MARS confirman que la implementación recursiva de un algoritmo como el factorial consume significativamente más espacio en la pila. Al observar el segmento de datos durante la ejecución paso a paso, se ve cómo `$ra` y posiblemente otros registros se apilan con cada llamada. Esto contrasta con la versión iterativa, que mantiene un uso de pila mínimo y constante.
- **Rendimiento (Ciclos de Instrucción):** Aunque MARS no proporciona directamente un conteo de ciclos detallado para cada instrucción de forma fácil, el mayor número de instrucciones ejecutadas para la gestión de la pila (prolog y epilog de cada llamada recursiva) en la versión recursiva sugiere un rendimiento inferior en comparación con la iterativa. Cada `sw`,

`lw`, `addi $sp`, `jal`, `jr` añade un overhead (uso adicional de recursos en tiempo de ejecución y espacio de memoria).

- **Legibilidad y Depuración:** Para problemas simples como el factorial, la versión recursiva puede parecer inicialmente más "elegante" y legible en un lenguaje de alto nivel. Sin embargo, en ensamblador MIPS, la gestión explícita de la pila en la recursión puede hacer que el código sea más propenso a errores y más difícil de depurar si no se maneja correctamente. Un error en el ajuste del puntero de pila o en el guardado/restaurado de registros puede llevar a fallos sutiles y difíciles de rastrear. La versión iterativa, al ser más lineal, a menudo es más sencilla de depurar a nivel de ensamblador.

- **Robustez:** La implementación iterativa es inherentemente más robusta frente al desbordamiento de pila. A menos que se asigne memoria dinámica en un bucle infinito (un problema diferente), las soluciones iterativas no están sujetas a los límites de tamaño de la pila del sistema. La recursión, por otro lado, siempre lleva el riesgo de desbordamiento si la profundidad máxima no se conoce o no se gestiona adecuadamente.

Conclusión del Análisis: Los resultados prácticos y teóricos refuerzan la idea de que, en un entorno de ensamblador de bajo nivel como MIPS32, la **implementación iterativa es superior en la mayoría de los casos en términos de eficiencia y robustez**. Si bien la recursividad puede ofrecer una mayor claridad conceptual para ciertos algoritmos, su implementación conlleva una sobrecarga significativa en el uso de la pila y el rendimiento. Por lo tanto, para el desarrollo de software en MIPS32, se recomienda priorizar soluciones iterativas siempre que sea posible, reservando la recursividad para problemas donde su claridad sea indispensable y la profundidad de la recursión sea limitada y controlable. La conversión de un algoritmo recursivo a iterativo (si es posible) es una práctica común para optimizar el rendimiento y evitar problemas de desbordamiento de pila en MIPS32.