PRÁCTICA DE LABORATORIO 1

Gabriel Blanco C.I:31.346.296 Andrés Roche C.I:30.656.948

7 de Julio de 2025

Pregunta 1

¿Cómo se implementa la recursividad en MIPS32?

Para implementar la recursividad en MIPS32, se utilizan principalmente las etiquetas (‘nombre:‘) para marcar direcciones de memoria y la instrucción ‘jal‘ (jump and link) para realizar llamadas a subrutinas.

La implementación sigue la estructura clásica de los algoritmos recursivos: se define un caso base, se manejan los argumentos, se gestionan los valores de retorno y se establece el cuerpo del código recursivo.

A continuación, se presenta un ejemplo de la función factorial en C para ilustrar el concepto:

```c
int fact(int n){
  // caso base
  if(n==1)
    return 1;
  else
    // resto del codigo
    return fact(n-1) * n;
}
```

Listing 1: Función Factorial en C

Aquí un ejemplo de la función factorial implementada en MIPS32:

```
factorial:
  # Caso base
  beq $a0, $zero, caso_base

  # Preparar la pila
  addi $sp, $sp, -8
  sw $ra, 4($sp)
  sw $a0, 0($sp)

  # (n - 1)
  addi $a0, $a0, -1

  # factorial(n - 1)
  jal factorial

  # Restaurar registros de la pila
  lw $a0, 0($sp)
  lw $ra, 4($sp)
  addi $sp, $sp, 8

  # factorial(n - 1) * n
  mul $v0, $v0, $a0

  jr $ra

caso_base:
  li $v0, 1
```

```
28    jr $ra
```

Listing 2: Función Factorial en MIPS32

Como se observa, la implementación en MIPS32 incluye un caso base manejado por la instrucción 'beq', la preparación de la pila para almacenar el registro de retorno ('$ra') y los argumentos ('$a0'), la disminución del argumento para la siguiente llamada ('n-1' en '$a0'), el uso de '$v0' para el valor de retorno, y la llamada recursiva mediante 'jal factorial'.

¿Qué papel cumple la pila ($sp)?

El uso de la pila es una parte fundamental y obligatoria en MIPS para la implementación de funciones, a diferencia de lenguajes de alto nivel donde su gestión es implícita.

```
1  # Preparar la pila
2    addi $sp, $sp, -8
3    sw $ra, 4($sp)
4    sw $a0, 0($sp)
5
6  # Sacar los datos
7    lw $a0, 0($sp)
8    lw $ra, 4($sp)
9    addi $sp, $sp, 8
```

Listing 3: Uso de la pila en MIPS32

Mediante la pila, se almacenan datos cruciales como parámetros, variables locales y direcciones de retorno de cada llamada a función. Esto es vital para que la función pueda volver al punto correcto en la ejecución de la función llamante.

Pregunta 2

¿Qué riesgos de desbordamiento existen?

Existen principalmente dos tipos de desbordamiento que pueden surgir en este contexto:

1. **Desbordamiento de Pila (Stack Overflow)**: Ocurre cuando la pila se llena, impidiendo que se puedan asignar más marcos de pila para nuevas llamadas a funciones o almacenamiento de datos.
2. **Desbordamiento Aritmético (Arithmetic Overflow)**: Sucede cuando el resultado de una operación aritmética excede el rango máximo que puede almacenar el tipo de dato asignado. Por ejemplo, en un entero de 32 bits, que puede almacenar hasta $2,147,483,647$, un resultado mayor a este valor causaría un desbordamiento.

¿Cómo mitigarlos?

Para el Desbordamiento de Pila

1. **Conversión a Algoritmos Iterativos**: Transformar un algoritmo recursivo en iterativo elimina la necesidad de múltiples llamadas a la pila. Sin embargo, esto puede hacer que algunos algoritmos sean menos legibles.
2. **Programación Dinámica (Memoización)**: Consiste en almacenar los resultados de llamadas a funciones en una caché (por ejemplo, un diccionario o un arreglo). Antes de realizar un cálculo, se verifica si el resultado ya existe en la caché. Si es así, se devuelve; de lo contrario, se calcula y se guarda. Aunque no elimina directamente la profundidad de la recursión, reduce drásticamente las llamadas redundantes, lo que puede ayudar a mantener la pila más pequeña. No obstante, si la entrada es muy grande, la profundidad de la pila aún podría ser un problema.
3. **Optimización de Cola (Tail Call Optimization - TCO)**: Algunos compiladores o intérpretes inteligentes pueden optimizar las llamadas de cola para que no consuman un nuevo marco de pila. (Cabe destacar que MIPS32 generalmente no soporta TCO de forma nativa).
4. **Aumentar el Tamaño de la Pila**: Es posible configurar un tamaño de pila mayor, aunque esta solución depende del sistema operativo y no siempre es viable o deseable.

Para el Desbordamiento Aritmético

1. **Verificar los Límites Antes de la Operación**: Implementar comprobaciones para asegurar que el resultado de una operación no excederá el rango del tipo de dato antes de realizarla. Esto implica establecer límites para las funciones o subrutinas.

2. **Utilizar un Tipo de Dato Más Grande**: En lenguajes de alto nivel, se puede cambiar el tipo de dato (por ejemplo, de 'int' a 'long int') para aumentar el rango de valores almacenables. En MIPS32, sin embargo, no existe una forma directa de "aumentar" el valor de un 'int' a un tipo de mayor capacidad directamente.

Pregunta 3

¿Qué diferencias encontraste entre una implementación iterativa y una recursiva en cuanto al uso de memoria y registros?

Algunas de las diferencias que tiene la implementación iterativa vs recursiva:

Gestión de la Pila

- **Implementación Iterativa**: El uso de la pila es mínimo, limitándose principalmente a almacenar la dirección de retorno ($ra) de la llamada inicial a la función o subrutina.

- **Implementación Recursiva**: El consumo de la pila es significativo, ya que requiere el almacenamiento de la dirección de retorno ($ra) y los parámetros de cada llamada ($a0) en marcos de pila individuales, lo que se acumula con la profundidad de la recursión.

Riesgo de Desbordamiento de Pila (Stack Overflow)

- **Implementación Iterativa**: El riesgo de desbordamiento de pila es muy bajo, siendo virtualmente nulo para algoritmos comunes como la secuencia de Fibonacci.

- **Implementación Recursiva**: Presenta un alto riesgo de desbordamiento de pila, especialmente cuando se trabaja con valores de n grandes, debido a la acumulación de marcos de pila con cada llamada recursiva.

Utilización de Registros

- **Implementación Iterativa**: Tiende a hacer mayor uso de variables temporales almacenadas directamente en los registros del procesador para la manipulación de datos durante la ejecución.

- **Implementación Recursiva**: Depende en mayor medida de la pila para el almacenamiento de valores temporales y el contexto de cada invocación de la función, reduciendo la dependencia directa de los registros del procesador para variables temporales a largo plazo.

Pregunta 4

¿Qué diferencias encontraste entre los ejemplos académicos del libro y un ejercicio completo y operativo en MIPS32?

Fibonacci iterativo

- **Implementación del libro (estándar):** Se enfoca en **calcular y devolver únicamente el valor del n-ésimo término** de Fibonacci (ej., F(n) en `$v0`). Es ideal cuando solo necesitas un valor específico.

- **Nuestra Implementación:** Tiene como objetivo **imprimir todos los valores de la secuencia** hasta el n-ésimo término (parámetro `$a0`). Es más útil para visualización o depuración, mostrando la progresión completa.

—

Fibonacci Recursivo

- **Casos Base:** El **Código 1** usa una definición no estándar (F(n)=1 para n¡3), mientras que el **Código 2** utiliza la definición estándar y correcta (F(n)=n para n≤1).
- **Manejo de Registros:** El **Código 1** guarda y restaura explícitamente el registro `$s1` en la pila porque lo usa para almacenar el resultado de una llamada recursiva, lo cual es una buena práctica para registros "callee-saved". El **Código 2**, en cambio, guarda directamente el resultado de la llamada recursiva en la pila, evitando la necesidad de manipular un registro `$s`.
- **Flujo del Caso Base:** El **Código 1** salta directamente al epílogo de la función al alcanzar un caso base. El **Código 2** salta a una sección específica (`base_case`) que gestiona explícitamente la restauración de la pila y el registro de retorno.

Pregunta 5

Elaborar un tutorial de la ejecución paso a paso en MARS.

1 Sección `.data`

La sección `.data` se utiliza para declarar y reservar espacio para las variables y cadenas de caracteres que el programa usará.

- `prompt`: Es una cadena de texto que se mostrará al usuario para pedirle que ingrese un número. La directiva `.asciiz` indica que es una cadena ASCII terminada en un carácter nulo (cero).
- `result`: Es otra cadena de texto usada para mostrar antes del resultado final del cálculo.
- `newline`: Es una cadena que contiene solo un salto de línea (\n), utilizada para formatear la salida en la consola.

—

2 Sección `.text` y `main`

La sección `.text` contiene el código ejecutable de tu programa. La directiva `.globl main` hace que la etiqueta `main` sea global, indicando el punto de entrada principal del programa donde comienza la ejecución.

Solicitar Entrada al Usuario

Esta parte del código es responsable de interactuar con el usuario para obtener el valor de n.

- Se carga el valor **4** en el registro `$v0` para indicar que se desea realizar una llamada al sistema para **imprimir una cadena**.
- Luego, se carga la dirección de la cadena `prompt` en el registro `$a0`, que es el registro estándar para pasar argumentos a las llamadas al sistema.
- Finalmente, se ejecuta la instrucción `syscall` para mostrar el mensaje "Ingrese n para calcular fib(n): " en la consola.

Leer la Entrada del Usuario

Después de solicitar la entrada, el programa debe leer el número que el usuario ingresa.

- Se carga el valor **5** en `$v0` para especificar que la siguiente llamada al sistema será para **leer un entero**.
- Se ejecuta `syscall`. El entero ingresado por el usuario es automáticamente almacenado por el sistema operativo en el registro `$v0`.
- Para preparar este valor como argumento para la función Fibonacci, su contenido se **mueve** de `$v0` a `$a0`.

Validar Entrada

Es importante asegurar que la entrada del usuario sea válida para evitar comportamientos inesperados en la función recursiva.

- Se utiliza la instrucción `bltz $a0, invalido`. Esto significa saltar si es menor que cero. Si el valor de n (que está en `$a0`) es negativo, el programa salta directamente a la etiqueta `invalido`, evitando el cálculo de Fibonacci para valores no definidos en esta implementación.

Calcular `fib(n)`

Si la entrada es válida, se procede a calcular el número de Fibonacci.

- Se realiza una llamada a la subrutina `fibonacci` utilizando la instrucción `jal fibonacci`. Esta instrucción no solo salta a la función, sino que también guarda la dirección de la siguiente instrucción (después del `jal`) en el registro `$ra` (return address). Esto es crucial para que la función `fibonacci` sepa a dónde debe regresar una vez que complete su tarea.
- Después de que la función `fibonacci` retorna, el resultado del cálculo (que por convención se encuentra en `$v0`) se **mueve** al registro `$s0`. Este es un registro de tipo "callee-saved", lo que significa que su valor se preserva a través de llamadas a funciones, asegurando que el resultado no se pierda antes de ser mostrado.

Mostrar Resultado

Una vez que el número de Fibonacci ha sido calculado, se muestra al usuario.

- Se repite el patrón de llamadas al sistema para imprimir. Primero, se carga **4** en `$v0` y la dirección de la cadena `result` en `$a0` para imprimir "El resultado es: ".
- Luego, se carga **1** en `$v0` y el resultado almacenado en `$s0` se mueve a `$a0` para **imprimir el número entero** calculado.
- Finalmente, se carga **4** en `$v0` y la dirección de `newline` en `$a0` para imprimir un salto de línea, mejorando la legibilidad de la salida.

Salir del Programa y Etiqueta `invalido`

El programa debe terminar su ejecución de manera ordenada.

- Se carga el valor **10** en `$v0` para invocar la llamada al sistema de **terminar el programa**.
- La etiqueta `invalido` es el destino del salto condicional `bltz`. Si la entrada inicial fue negativa, el programa salta directamente aquí y termina sin realizar ningún cálculo, manejando así la validación básica.

—

3 Función `fibonacci` (La Recursión)

Esta es la función central que implementa la lógica recursiva para calcular el número de Fibonacci. El manejo de la pila es fundamental aquí para que la recursión funcione correctamente.

Configuración del Marco de Pila

Cada vez que la función `fibonacci` es llamada, se prepara un "marco de pila" para guardar su contexto.

- La instrucción `addi $sp, $sp, -12` ajusta el puntero de la pila (`$sp`), restando 12 bytes. Esto reserva espacio en la pila para tres "palabras" (cada una de 4 bytes) que se utilizarán para guardar el contexto de esta llamada a la función.

- Se guarda el contenido del registro `$ra` (dirección de retorno) en la pila en la posición 8($*sp*) (`sw $ra, 8($sp)`). Esto es crucial para la recursión: cada llamada a `fibonacci` necesita saber a dónde regresar después de completarse.
- Se guarda el argumento actual n (que está en `$a0`) en la pila en la posición 4($*sp*) (`sw $a0, 4($sp)`). Este valor de n es necesario para calcular $n-2$ después de que se complete la llamada recursiva para $n-1$.

Casos Base de la Recursión

La recursión debe tener condiciones de parada, conocidas como casos base, para evitar un bucle infinito.

- Se carga el valor **1** en un registro temporal `$t0`.
- La instrucción `ble $a0, $t0, base`$_{c}ase$("$branchiflessthanorequal$")$compruebasielvalorden`n`$(en$`$a0`$)esmenoroigu$
  - $fib(0) = 0$
  - $fib(1) = 1$

Calcular `fib(n-1)`

Si no es un caso base, la función se llama a sí misma para calcular el término anterior.

- Se decrementa el valor de n en `$a0` en **1** (`addi $a0, $a0, -1`) para preparar el argumento para la llamada recursiva a $fib(n-1)$.
- Se realiza la llamada recursiva a `fibonacci` con `jal fibonacci`. Cuando esta llamada retorna, el resultado de $fib(n-1)$ estará en `$v0`.
- El resultado de $fib(n-1)$ (que está en `$v0`) se **guarda** en la pila en la posición 0($*sp*) (`sw $v0, 0($sp)`). Esto es necesario porque la siguiente llamada recursiva ('fib(n-2)') utilizará `$v0` para su propio resultado, y el valor de $fib(n-1)$ se necesita posteriormente para la suma final.

Calcular `fib(n-2)`

Después de calcular $fib(n-1)$, se procede con $fib(n-2)$.

- Se **recupera** el valor original de n de la pila (`lw $a0, 4($sp)`). Es crucial hacer esto porque `$a0` fue modificado en el paso anterior para la llamada a $fib(n-1)$.
- Se decrementa este n original en **2** (`addi $a0, $a0, -2`) para preparar el argumento para la llamada a $fib(n-2)$.
- Se realiza otra llamada recursiva a `fibonacci` con `jal fibonacci`. Al retornar, el resultado de $fib(n-2)$ estará en `$v0`.

Sumar Resultados

Una vez que ambos resultados recursivos están disponibles, se combinan.

- Se **carga** el valor de $fib(n-1)$ que se había guardado en la pila en el registro temporal `$t0` (`lw $t0, 0($sp)`).
- Se realiza la **suma** de los dos términos: `$t0` (que contiene $fib(n-1)$) y `$v0` (que ahora contiene $fib(n-2)$). El resultado de esta suma, que es $fib(n)$, se guarda en `$v0`, el registro estándar para el valor de retorno de la función.

Restaurar Contexto y Retornar

Antes de salir de la función, se debe restaurar el estado original de los registros y la pila.

- Se **recupera** la dirección de retorno original (`$ra`) de la pila (`lw $ra, 8($sp)`). Esto garantiza que, al finalizar la función actual, el control regrese a la instrucción correcta en la función que la llamó.
- Se ajusta el puntero de la pila (`addi $sp, $sp, 12`), sumando 12 bytes. Esto **libera** el espacio que se había reservado al inicio de la función.
- La instrucción `jr $ra` ("jump register") salta a la dirección de memoria almacenada en `$ra`, devolviendo el control al punto en el código desde donde se llamó a esta instancia de `fibonacci`.

Etiqueta `base_case`

Esta sección maneja los casos base de la recursión de Fibonacci.

- Para los valores de $n = 0$ o $n = 1$, el resultado de $fib(n)$ es simplemente n mismo. Por lo tanto, el valor de n (que se encuentra en `$a0`) se **mueve** directamente a `$v0` (`move $v0, $a0`), que es el registro de retorno.
- Se utiliza `jr $ra` para retornar. En este caso, no hay necesidad de ajustar la pila porque el espacio para el marco de pila se reservó después de la comprobación del caso base; solo las llamadas recursivas que necesitan más espacio y guardar el contexto lo hacen.

Pregunta 6

Justificar la elección del enfoque (iterativo o recursivo) según eficiencia y claridad en MIPS.

La elección de implementar el cálculo de Fibonacci de forma **recursiva** en MIPS32, en lugar de una solución iterativa, se hizo con un propósito pedagógico muy claro: demostrar explícitamente cómo funciona la **recursión** en un lenguaje de bajo nivel y, lo que es más importante, cómo se gestiona el uso de la **pila**.

El algoritmo recursivo para Fibonacci ($fib(n) = fib(n-1) + fib(n-2)$) es un ejemplo canónico de recursión, lo que lo hace ideal para entender este concepto fundamental. Al implementar la recursión en MIPS, el código refleja casi directamente la definición matemática. Se puede ver cómo la función se llama a sí misma con $n-1$ y $n-2$, y cómo esos resultados se combinan. Esto proporciona una representación visual y lógica clara de lo que significa que una función se llame a sí misma.

En MIPS32, cada vez que la función `fibonacci` se llama a sí misma, somos nosotros quienes debemos explícitamente manipular el puntero de la pila (`$sp`) y usar las instrucciones `sw` (store word) y `lw` (load word) para guardar y restaurar los registros importantes (como `$ra` y `$a0`). Este proceso manual **revela directamente el mecanismo de los marcos de pila**.

En resumen, la elección del enfoque recursivo para Fibonacci en MIPS32 se hizo por su capacidad excepcional para ilustrar de forma cristalina los principios de la recursión y la gestión explícita de la pila en la arquitectura MIPS.

Análisis y Discusión de Resultados

Este documento ha explorado en detalle la implementación de la serie de Fibonacci en MIPS32, centrándose particularmente en el enfoque recursivo, y lo ha contrastado con aspectos de la implementación iterativa y la gestión de la memoria a bajo nivel. Los resultados y discusiones clave se pueden agrupar en varias áreas:

La Recursividad como Herramienta Pedagógica

La elección de implementar Fibonacci de forma recursiva, a pesar de sus conocidas desventajas de eficiencia, se justifica plenamente desde una perspectiva educativa. Como se detalla en la Pregunta 6, el

algoritmo recursivo de Fibonacci es un "ejemplo canónico" que permite una comprensión "cristalina" de la recursión en el contexto de MIPS32. La correspondencia directa con la definición matemática facilita la comprensión del flujo de control recursivo, mostrando cómo una función se llama a sí misma para resolver subproblemas más pequeños. Este enfoque pedagógico es invaluable para estudiantes que se inician en la programación de bajo nivel y en la arquitectura de computadoras.

Gestión Explícita de la Pila: Un Aprendizaje Crucial

Un punto central de este análisis es el papel de la pila (`$sp`) en la recursión. A diferencia de los lenguajes de alto nivel donde la gestión de la pila es implícita, el código MIPS32 obliga a una manipulación explícita del `$sp` y de las instrucciones `sw` y `lw`. Esta obligatoriedad revela directamente el mecanismo de los "marcos de pila". Se evidencia cómo cada llamada a función requiere guardar su contexto (dirección de retorno `$ra` y argumentos `$a0`), permitiendo a cada instancia recursiva operar de forma independiente y saber a dónde regresar. Esta visibilidad es una ventaja fundamental para entender la arquitectura subyacente.

El Desbordamiento Aritmético y sus Implicaciones en MIPS

La mención del desbordamiento aritmético es un recordatorio importante de las limitaciones de los tipos de datos a nivel de hardware. El hecho de que en MIPS32 no haya una forma directa de "aumentar" el tipo de dato como en lenguajes de alto nivel subraya la necesidad de verificaciones explícitas de límites o de replantear el algoritmo para evitar exceder los $2^{31} - 1$ (o $2,147,483,647$) para enteros con signo. Esto resalta que, a bajo nivel, la gestión de rangos numéricos es responsabilidad directa del programador.

Variaciones en las Implementaciones de Fibonacci

Las diferencias señaladas entre ejemplos académicos y una implementación "completa y operativa" (Pregunta 4) revelan la flexibilidad y los matices del desarrollo en ensamblador. Desde la elección de los casos base hasta el manejo de registros (`$s1` vs. almacenamiento directo en pila) y el flujo del caso base, se demuestra que no hay una única "mejor" implementación. Cada variación puede optimizar un aspecto diferente (uso de registros, claridad del flujo, etc.), lo que obliga a tomar decisiones de diseño informadas.

Conclusión del Análisis

En síntesis, el estudio del código de Fibonacci en MIPS32 ha servido como una plataforma excelente para comprender la recursión a un nivel fundamental, poniendo de manifiesto la crítica interacción entre el código, la pila y los registros. Si bien ha expuesto las ineficiencias de la recursión directa para este problema, ha proporcionado una valiosa visión de cómo los lenguajes de bajo nivel gestionan procesos que son automáticos en entornos de más alto nivel. Este conocimiento es esencial para el desarrollo de software optimizado y para la depuración de problemas relacionados con la memoria y el rendimiento en sistemas embebidos o críticos.