UNIVERSIDAD DE CARABOBO

FACULTAD EXPERIMENTAL DE CIENCIAS Y TECNOLOGIA

NAGUANAGUA, CARABOBO

PRACTICA 1

ESTUDIANTES:

FABIAN PEÑA C.I: 31.994.692
SANTIAGO ESCALONA C.I: 31.768.925

PROFESOR:

JOSE CANACHE

1)¿Como se implementa la recursividad en MIPS32? ¿Que papel cumple la pila?

La recursividad en MIPS32 se implementa mediante el uso de llamadas a funciones, donde una función se invoca a sí misma utilizando la instrucción jal (jump and link). Esta instrucción permite saltar a la dirección de la función y, al mismo tiempo, guardar la dirección de retorno en el registro $ra.

Una vez que se entra a la función (ya sea en la primera llamada o en una llamada recursiva), es dentro de la función donde se reserva espacio en la pila. Para ello, se ajusta el registro $sp y se almacenan en la pila los registros que deben preservarse, como $ra, los parámetros o cualquier registro que la función vaya a modificar y que sea necesario recuperar al retornar.

Cada llamada recursiva crea su propio marco de pila (stack frame), el cual contiene la información necesaria para esa llamada específica. Al finalizar la función, los valores guardados se restauran desde la pila y el puntero $sp vuelve a su valor anterior, permitiendo retornar correctamente a la llamada previa mediante la instrucción jr $ra.

En este proceso, la pila cumple un papel muy importante, ya que permite mantener el estado independiente de cada llamada recursiva, asegurando que el flujo de ejecución y los datos de cada nivel de la recursión no se pierdan ni se sobrescriban.

2.¿Que riesgos de desbordamientos existen?¿Como mitigarlos?

En la implementación recursiva del algoritmo de paridad en MIPS32, el principal riesgo de desbordamiento es el desbordamiento de la pila (stack overflow). Esto ocurre debido a que cada llamada recursiva a la función, reserva espacio en la pila para almacenar el valor del argumento ($a0) y la dirección de retorno ($ra). Dado que el algoritmo reduce el valor de n en una unidad por cada llamada, el número de llamadas recursivas es proporcional al valor de entrada, lo que puede provocar un uso excesivo de la pila si n es grande.

Para mitigar este riesgo, es recomendable limitar el tamaño del valor de entrada, garantizar que el espacio reservado en la pila se libere correctamente al finalizar cada llamada recursiva —como se hace en esta implementación— o preferir una solución iterativa cuando se trabaja con valores grandes.

En cuanto a la versión iterativa del algoritmo, el riesgo de desbordamiento de pila es prácticamente inexistente, ya que no se realizan llamadas recursivas ni se crean múltiples marcos de pila. Además, el riesgo de desbordamiento aritmético es bajo debido a la simplicidad de las operaciones utilizadas.

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

En la implementación recursiva, el uso de memoria es mayor debido a la utilización de la pila (stack). Cada llamada recursiva a la función requiere reservar espacio en la pila para almacenar información importante, como el valor del argumento ($a0) y la dirección de retorno ($ra). En este caso, por cada llamada se consumen 8 bytes de memoria en la pila. Dado que el número de llamadas recursivas es proporcional al valor de entrada n, el consumo de memoria también crece linealmente, lo que puede provocar un desbordamiento de la pila cuando se utilizan valores grandes.

Además,en la implementación recursiva hay que tener cuidado con el manejo de los registros, ya que es necesario preservar el contenido de aquellos que deben mantenerse entre llamadas. Esto obliga a guardar y restaurar registros en la pila, incrementando tanto la complejidad del código como el uso de memoria.

Por otro lado, en la implementación iterativa, el uso de memoria es más eficiente, ya que no se realizan llamadas recursivas ni se crean múltiples marcos de pila. El algoritmo se ejecuta dentro de un solo contexto, utilizando un número fijo de registros durante toda la ejecución. Como resultado, el consumo de memoria es constante y no depende del tamaño del valor de entrada.

En cuanto al uso de registros, la versión iterativa resulta más sencilla de manejar, ya que no es necesario almacenar ni restaurar la dirección de retorno ni los valores de los registros en la pila. Esto reduce la complejidad del programa y mejora su eficiencia.

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

En relación con los ejemplos presentados en el libro, pude observar que son, en su mayoría, ejemplos básicos, cuyo objetivo principal es enseñar el funcionamiento de instrucciones elementales como beq, addi, así como una introducción al uso de los registros. Si bien estos ejemplos resultan útiles para comprender los fundamentos del lenguaje ensamblador MIPS32, se quedan cortos cuando uno el estudiante, se enfrenta a un ejercicio completo y operativo.

En un ejercicio real y completo en MIPS32, es necesario manejar correctamente aspectos más avanzados, como el uso de la pila, los saltos condicionales, la correcta administración de los registros, el almacenamiento y recuperación de direcciones de memoria, y las llamadas a funciones. Estos elementos no suelen abordarse con suficiente profundidad en los ejemplos del libro, por lo que gran parte del aprendizaje queda a cargo de uno, ya sea mediante el estudio autónomo, la práctica constante o el apoyo de herramientas como la inteligencia artificial.

Al igual ocurre con los ejercicios propuestos al final del capitulo, los cuales generalmente consisten en transcribir pequeñas porciones de código de C a MIPS o en analizar fragmentos cortos de código ensamblador. Aunque estos ejercicios ayudan a reforzar conceptos básicos,

no preparan completamente para desarrollar un programa desde cero.

Cuando uno se enfrenta a un ejercicio completo, como ocurrió durante el parcial, la dificultad aumenta considerablemente, ya que en MIPS32 no se cuenta con la gran cantidad de funciones e instrucciones de alto nivel que existen en lenguajes como C. En ensamblador, el programador debe encargarse de todo el proceso, incluso de tareas que en C resultan sencillas, como acceder a una posición específica de un vector. En MIPS32, estas operaciones deben implementarse manualmente, lo que implica mayor complejidad, más trabajo y un cuidado extremo en el manejo de la memoria y las llamadas a funciones.

Aunque el libro proporciona una base importante para entender MIPS32, tanto los ejemplos como los ejercicios, resultan limitados en comparación con las exigencias de un ejercicio completamente operativo, donde se requiere un dominio más profundo del funcionamiento interno del lenguaje.

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

Para la ejecución y análisis de los programas desarrollados en lenguaje ensamblador MIPS32, tanto en su versión recursiva como iterativa, se utilizó el simulador MARS (MIPS Assembler and Runtime Simulator). A continuación,esta el procedimiento general para ejecutar ambos programas paso a paso y analizar su comportamiento interno.

En primer lugar, se abre el simulador MARS y se carga el archivo correspondiente al programa mediante la opción File → Open. Una vez cargado el código, se ensambla el programa presionando el botón Assemble o la tecla F3, lo cual permite verificar que el código no contenga errores de sintaxis.

Posteriormente, se selecciona el modo de ejecución paso a paso utilizando el botón Step, lo que permite observar detalladamente la ejecución de cada instrucción. Al iniciar la ejecución, el programa muestra un mensaje solicitando al usuario que ingrese un número, el cual es leído mediante una llamada al sistema (syscall) y almacenado en un registro.

En ambas versiones del programa, el valor ingresado se pasa como argumento a la rutina encargada de determinar la paridad del número. En la versión recursiva, la ejecución continúa con múltiples llamadas a la función, lo que puede observarse mediante los cambios en el registro de dirección de retorno ($ra) y en el puntero de pila ($sp), así como en la reserva y liberación de espacio en la pila. En cambio, en la versión iterativa, el flujo del programa permanece dentro de un solo contexto, sin realizar llamadas recursivas ni utilizar la pila de forma intensiva.

Durante la ejecución paso a paso, es posible observar en la ventana de registros de MARS cómo se modifican valores importantes como los registros de argumentos, resultados y control del programa. Asimismo, en la versión recursiva se puede visualizar claramente el uso de la pila en la ventana de memoria, mientras que en la versión iterativa el consumo de memoria se mantiene constante.

Y por ultimo, al concluir el codigo recursivo o iterativo del número ingresado, el programa retorna el resultado al procedimiento principal, el cual evalúa el valor obtenido y muestra en pantalla si el número es par o impar.

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

Para este informe, se considera que el enfoque iterativo es la elección superior en términos de eficiencia técnica, mientras que el recursivo destaca por su claridad académica al seguir fielmente la definición matemática proporcionada.

Eficiencia: Como se mencionó en la pregunta 3, la versión iterativa es significativamente más eficiente porque mantiene un consumo de memoria constante. Al no requerir la creación de múltiples marcos de pila (stack frames) ni el guardado constante de registros como $ra y $a0, el procesador realiza menos operaciones de acceso a memoria (carga y almacenamiento), lo que acelera la ejecución.

Claridad: El enfoque recursivo ofrece una mayor claridad conceptual, ya que su estructura en MIPS32 es un reflejo directo de la fórmula paridad(n) = 1 - paridad(n-1). Sin embargo, esta claridad se traduce en una mayor complejidad de implementación en ensamblador debido a la gestión manual de la pila que el programador debe realizar.

Conclusión de elección: Aunque el enfoque recursivo es valioso pedagógicamente, para un entorno operativo se justifica la elección del enfoque iterativo. Este evita el riesgo de stack overflow ante valores grandes de $n y optimiza el uso de los recursos limitados del procesador MIPS32.

7. Análisis y discusión de los resultados

A partir de la implementación del algoritmo de paridad en MIPS32, tanto en su versión recursiva como iterativa, se obtuvieron resultados correctos en ambos casos, ya que los dos enfoques permiten determinar adecuadamente si un número es par o impar. Sin embargo, el análisis de la ejecución y del uso de recursos muestra diferencias importantes entre ambas implementaciones.

Durante la ejecución paso a paso en el simulador MARS, se pudo observar que la versión recursiva realiza múltiples llamadas a la función encargada de calcular la paridad, lo que implica un uso intensivo de la pila. En cada llamada se reservó espacio para almacenar registros y la dirección de retorno, lo que incrementa el consumo de memoria de forma proporcional al valor de entrada. Aunque el programa funciona correctamente, este comportamiento evidencia una menor eficiencia y un mayor riesgo de desbordamiento de pila cuando se utilizan valores grandes.

En contraste, la versión iterativa mostró un comportamiento más estable y eficiente. Al ejecutarse dentro de un solo contexto, no requiere llamadas recursivas ni un uso constante de la pila, manteniendo un consumo de memoria fijo y un manejo más simple de los registros. Esto se traduce en una ejecución más rápida y segura, especialmente en un lenguaje de bajo nivel como MIPS32.

Desde el punto de vista conceptual, la versión recursiva resulta más intuitiva para comprender la lógica del problema, ya que sigue una definición clara basada en la reducción progresiva del valor de entrada. No obstante, esta claridad se ve limitada en ensamblador debido a la complejidad adicional del manejo manual de la pila y de los registros, lo cual no ocurre en lenguajes de alto nivel.

En general, los resultados obtenidos permiten concluir que, aunque ambos enfoques son válidos y funcionales, la implementación iterativa es más adecuada para MIPS32 debido a su mayor eficiencia, menor consumo de memoria y simplicidad en la ejecución. La versión recursiva, por su parte, resulta útil como herramienta didáctica para comprender el funcionamiento de la pila y las llamadas a funciones, pero no es la opción más óptima para una implementación práctica en ensamblador.