UNIVERSIDAD DE CARABOBO

FACULTAD EXPERIMENTAL DE CIENCIAS Y TECNOLOGIA

NAGUANAGUA, CARABOBO

PRACTICA 2

ESTUDIANTES:

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

PROFESOR:

JOSE CANACHE

Pregunta 1: ¿Qué diferencias existen entre registros temporales y registros guardados? ¿Y cómo se aplicó esta distinción en la práctica?

Los registros temporales se utilizan para almacenar valores intermedios, como resultados de comparaciones, cálculos aritméticos o direcciones momentáneas. Su contenido no se preserva después de una llamada a una subrutina, por lo que pueden ser sobrescritos libremente. Esto los hace adecuados para operaciones de corto alcance dentro de bucles o bloques de código.

Por otro lado, los registros guardados están destinados a mantener valores que deben conservarse durante toda la ejecución de una función o sección del programa. Según la convención MIPS, si una subrutina utiliza registros guardados, debe respaldarlos en la pila antes de modificarlos y restaurarlos al finalizar.

En la práctica realizada, ambos algoritmos de ordenamiento fueron implementados de forma iterativa, por lo que no se empleó recursión ni llamadas anidadas profundas:

En el Quicksort iterativo, los registros guardados se usaron para almacenar información que debía mantenerse a lo largo del algoritmo, como la dirección base del arreglo, los límites de los subarreglos y el estado del algoritmo mientras se procesaban las particiones de forma iterativa. Esto permitió conservar estos valores sin riesgo de ser sobrescritos.

En el Bubble Sort, debido a su estructura simple y completamente iterativa, se utilizaron principalmente registros temporales para manejar contadores de bucles, comparaciones entre elementos y operaciones de intercambio. No fue necesario preservar información por largos periodos, por lo que el uso de registros guardados fue limitado.

Pregunta 2: ¿Qué diferencias existen entre los registros $a0–$a3, $v0–$v1 y $ra, y cómo se aplicó esta distinción en la práctica?

En MIPS32, los registros $a0–$a3 se utilizan para pasar argumentos a una subrutina, como direcciones de arreglos, tamaños o índices necesarios para ejecutar un algoritmo. Estos registros permiten enviar hasta cuatro parámetros sin usar memoria.

Los registros $v0–$v1 se emplean para retornar valores desde una función al programa principal, siguiendo la convención estándar de MIPS.

El registro $ra guarda la dirección de retorno cuando se usa la instrucción jal, lo que permite que el programa continúe su ejecución correctamente al finalizar una subrutina mediante jr $ra.

En la práctica, tanto en Quicksort iterativo como en Bubble Sort, $a0–$a3 se usaron para pasar los datos necesarios a las rutinas de ordenamiento, $v0–$v1 para devolver resultados simples cuando fue necesario, y $ra para asegurar el retorno correcto después de ejecutar subrutinas auxiliares. Esto ayudó a mantener el código ordenado y acorde a la convención de MIPS32.

Pregunta 3: ¿Cómo afecta el uso de registros frente a memoria en el rendimiento de los algoritmos de ordenamiento implementados?

El uso de registros en lugar de memoria tiene un impacto directo en el rendimiento de los algoritmos de ordenamiento, ya que los registros son mucho más rápidos de acceder que la memoria principal. En MIPS32, las operaciones entre registros se ejecutan en menos ciclos que aquellas que requieren instrucciones de carga (lw) y almacenamiento (sw).

En los algoritmos implementados, tanto Quicksort iterativo como Bubble Sort, se usaron registros para manejar contadores, índices y valores temporales durante las comparaciones. Esto permitió reducir la cantidad de accesos a memoria y hacer que el código fuera más eficiente.

Cuando fue necesario acceder al arreglo, sí se tuvo que usar memoria, pero mantener la mayor cantidad posible de datos en registros evitó accesos repetidos y costosos.

Pregunta 4: ¿Qué impacto tiene el uso de estructuras de control (bucles anidados, saltos) en la eficiencia de los algoritmos en MIPS32?

El uso de estructuras de control, como los bucles anidados y los saltos, influye directamente en la eficiencia de los algoritmos en MIPS32, ya que cada salto implica una instrucción adicional que debe ejecutarse y evaluarse.

En el caso de Bubble Sort, los bucles anidados generan una gran cantidad de saltos, ya que el algoritmo compara repetidamente los elementos del arreglo. Esto hace que el número de instrucciones ejecutadas sea alto y, por lo tanto, el algoritmo sea menos eficiente.

En Quicksort iterativo, aunque también se utilizan saltos y bucles, la cantidad de iteraciones suele ser menor, ya que el arreglo se divide en partes más pequeñas. Esto reduce el número total de comparaciones y saltos, mejorando el rendimiento en comparación con Bubble Sort.

Pregunta 5: ¿Cuáles son las diferencias de complejidad computacional entre el algoritmo Quicksort y el algoritmo alternativo (Bubble Sort)? ¿Qué implicaciones tiene esto para la implementación en MIPS32?

La principal diferencia entre Quicksort y Bubble Sort está en su complejidad computacional. Quicksort tiene, en promedio, una complejidad de O(n log n), mientras que Bubble Sort tiene una complejidad de $O(n^2)$, lo que significa que Bubble Sort realiza muchas más comparaciones cuando el tamaño del arreglo aumenta.

En la práctica, esto se nota claramente en la ejecución. Bubble Sort utiliza bucles anidados, por lo que el número de iteraciones y saltos crece rápidamente, haciendo que el algoritmo sea más lento, especialmente para arreglos grandes. En cambio, Quicksort, incluso en su versión iterativa, reduce el número de comparaciones al dividir el arreglo en subarreglos más pequeños.

En un entorno MIPS32, estas diferencias tienen un impacto importante, ya que cada comparación, salto y acceso a memoria implica la ejecución de varias instrucciones. Al tener una complejidad menor, Quicksort ejecuta menos instrucciones en total, lo que se traduce en un mejor rendimiento. Bubble Sort, aunque es más sencillo de implementar, genera más saltos y accesos repetidos a memoria, lo que lo hace menos eficiente.

Pregunta 6: ¿Cuáles son las fases del ciclo de ejecución de instrucciones en la arquitectura MIPS32 (camino de datos)? ¿En qué consisten?

En la arquitectura MIPS32, la ejecución de una instrucción se divide en varias fases dentro del camino de datos, las cuales permiten que la instrucción se procese de forma ordenada.

La primera fase es la búsqueda de la instrucción (Instruction Fetch), donde el procesador obtiene la instrucción desde la memoria usando el contador de programa (PC).

Luego sigue la decodificación de la instrucción (Instruction Decode), en la que se identifica qué tipo de instrucción es y se leen los registros necesarios para su ejecución.

La tercera fase es la ejecución (Execute), donde la ALU realiza las operaciones aritméticas o lógicas, o calcula direcciones de memoria y condiciones para saltos.

Después viene la fase de acceso a memoria (Memory Access), que solo se utiliza si la instrucción requiere leer o escribir datos en memoria, como en los casos de lw o sw.

Por último, está la fase de escritura de resultados (Write Back), donde el resultado de la operación se guarda en el registro correspondiente.

Pregunta 7: ¿Qué tipo de instrucciones se usaron predominantemente en la práctica (R, I, J) y por qué?

En la práctica se usaron principalmente instrucciones tipo R e I.

Las instrucciones R se usaron para operaciones aritméticas y lógicas entre registros, como sumas, restas y comparaciones dentro de los bucles de Quicksort y Bubble Sort. Son útiles porque trabajan directamente con los registros y no requieren acceso a memoria, lo que las hace rápidas.

Las instrucciones I se usaron para acceso a memoria (lw, sw) y para control de flujo simple como comparaciones con saltos condicionales (beq, bne). Son necesarias para leer o escribir los elementos del arreglo y manejar los índices dentro de los bucles.

Las instrucciones J se usaron muy poco o casi nada, porque los bucles y saltos condicionales se resolvieron mayormente con instrucciones tipo I y R.

Pregunta 8: ¿Cómo se ve afectado el rendimiento si se abusa del uso de instrucciones de salto (j, beq, bne) en lugar de usar estructuras lineales?

El uso excesivo de instrucciones de salto (j, beq, bne) puede reducir el rendimiento porque cada salto implica cambiar el flujo de ejecución y, en algunos casos, vaciar o recargar el pipeline del procesador. Esto genera esperas y ciclos extra, especialmente si los saltos son muy frecuentes o impredecibles.

En contraste, las estructuras lineales (ejecución secuencial de instrucciones) permiten que el procesador trabaje de manera continua, usando registros y evitando accesos innecesarios a memoria o saltos.

En los algoritmos de ordenamiento, abusar de saltos en lugar de bucles bien estructurados haría que el código sea más lento y menos eficiente, ya que se ejecutan más instrucciones de control en lugar de operaciones útiles sobre los datos.

Pregunta 9: ¿Qué ventajas ofrece el modelo RISC de MIPS en la implementación de algoritmos básicos como los de ordenamiento?

El modelo RISC de MIPS ofrece instrucciones simples, de un solo ciclo, lo que facilita implementar algoritmos básicos como Quicksort o Bubble Sort. Cada instrucción hace poco, pero rápido, y se ejecuta de manera predecible, lo que mejora la eficiencia.

Además, el uso de registros abundantes permite almacenar contadores, índices y valores temporales sin depender tanto de la memoria, reduciendo accesos lentos a RAM.

En conjunto, estas características hacen que los algoritmos de ordenamiento sean más rápidos, fáciles de depurar y más eficientes en MIPS32, comparado con arquitecturas complejas donde cada instrucción puede tardar varios ciclos.

Pregunta 10: ¿Cómo se usó el modo de ejecución paso a paso (step, step into) en MARS para verificar la correcta ejecución del algoritmo?

En MARS, el modo step permite ejecutar el programa una instrucción a la vez, mientras que step into hace lo mismo pero entrando dentro de las subrutinas. Esto ayuda a verificar que cada instrucción haga lo que esperamos.

Durante la práctica, se usó este modo para revisar paso a paso cómo se movían los índices, cómo se comparaban y se intercambiaban los elementos del arreglo, y cómo cambiaban los registros. Así se pudo confirmar que tanto el Quicksort iterativo como el Bubble Sort funcionaban correctamente antes de ejecutar todo el programa de manera automática.

Pregunta 11: ¿Qué herramienta de MARS fue más útil para observar el contenido de los registros y detectar errores lógicos?

La herramienta más útil fue el panel de Registers ubicado en el costado derecho de la interfaz, junto con el Data Segment. Mientras que el panel de registros permitía monitorear en tiempo real cómo cambiaban los valores de los índices ($a0-$a2) y los punteros temporales durante las particiones de Quicksort, el Data Segment fue crucial para verificar que los intercambios (sw) se estuvieran realizando en las direcciones de memoria correctas. Esta visualización directa permitió identificar errores comunes como el offset incorrecto al acceder a elementos del arreglo o problemas de alineación de memoria (como el error de word boundary) antes de que el programa terminara su ejecución.

Pregunta 12: ¿Cómo puede visualizarse en MARS el camino de datos para una instrucción tipo R?

En MARS, el camino de datos se visualiza a través de la herramienta "Tools ->MIPS X-Ray.º "Data Path Simulator". Para una instrucción tipo R como add $t0, $t1, $t2, el simulador muestra cómo se activan las señales de control para leer dos registros de la fuente, la operación se procesa en la ALU (Unidad Aritmético-Lógica) y el resultado se escribe de vuelta en el registro destino. En esta fase, no se activa la señal de lectura o escritura en memoria, ya que la operación ocurre estrictamente dentro del banco de registros y la ALU.

Pregunta 13: ¿Cómo puede visualizarse en MARS el camino de datos para una instrucción tipo I?

Para una instrucción tipo I como lw $t0, 0($a0), el simulador de camino de datos muestra una ruta extendida que incluye la fase de Memory Access. Se visualiza cómo la ALU calcula la dirección efectiva sumando el registro base y el inmediato, y luego esa dirección se utiliza para extraer un dato de la memoria de datos. Finalmente, se observa la señal de Write Back llevando ese valor desde la memoria hasta el registro destino. A diferencia de las instrucciones tipo R, aquí se evidencia el uso del bus de datos de memoria, lo que permite entender por qué estas instrucciones suelen consumir más ciclos o recursos en comparación con las aritméticas simples.

Pregunta 14: Justificar la elección del algoritmo alternativo.

Se seleccionó el Bubble Sort como algoritmo alternativo debido a su contraste didáctico con Quicksort. Mientras que Quicksort representa la eficiencia mediante la estrategia de "dividir y vencerás"(O(n log n)), Bubble Sort ofrece una estructura puramente iterativa

basada en bucles anidados que es mucho más sencilla de implementar en MIPS32 sin necesidad de recursión o manejo complejo de la pila. Esta elección permitió comparar cómo un algoritmo con complejidad $O(n^2)$ incrementa drásticamente el número de instrucciones ejecutadas y saltos realizados en comparación con el enfoque más optimizado de Quicksort.

Pregunta 15: Análisis y Discusión de los Resultados

El desarrollo de la práctica permitió contrastar dos paradigmas de ordenamiento bajo las restricciones y ventajas de la arquitectura MIPS32. A continuación, se analizan los puntos clave observados durante la ejecución de los algoritmos Quicksort y Bubble Sort:

Eficiencia en el Camino de Datos: Se observó que Quicksort, al emplear la estrategia de "dividir y vencerás", reduce drásticamente la cantidad de instrucciones ejecutadas en comparación con Bubble Sort. Mientras que Bubble Sort satura el camino de datos con instrucciones de carga (lw) y almacenamiento (sw) debido a sus constantes intercambios en memoria, Quicksort optimiza el uso de los registros para mantener los límites de las particiones, lo que se traduce en un menor número de accesos a la memoria principal y, por ende, en un rendimiento superior.

Impacto de los Saltos Condicionales: La ejecución en MARS evidenció que el abuso de estructuras de control (bucles anidados en Bubble Sort) genera un flujo de ejecución con múltiples saltos (beq, bne, j), lo cual es menos eficiente en arquitecturas con pipelining. Quicksort, aunque utiliza recursión (o iteración con particiones), mantiene una estructura de saltos más predecible y limitada a medida que el arreglo se divide, permitiendo que el procesador ejecute más instrucciones útiles por cada salto realizado.

Gestión de Memoria y Alineación: Un hallazgo crítico durante la implementación fue la sensibilidad de MIPS a la alineación de datos. Se confirmó que el uso de la directiva .align 2 es indispensable para evitar excepciones de tiempo de ejecución (word boundary alignment error) al manipular el arreglo con instrucciones de palabra completa. Esto resalta la importancia de comprender cómo la arquitectura organiza la memoria en bloques de 4 bytes para mantener la velocidad de acceso.

Uso Estratégico de Registros: La distinción entre registros temporales ($t) y guardados ($s) fue fundamental para garantizar la integridad de los datos. En Quicksort, el respaldo de registros en la pila (stack) permitió que las direcciones de retorno y los límites de los subarreglos no se perdieran durante las llamadas anidadas, demostrando que una correcta gestión de la pila es el corazón de la eficiencia en algoritmos complejos dentro de MIPS.

Conclusión de Rendimiento: En conclusión, los resultados prácticos validaron la teoría de la complejidad computacional: Quicksort (O(n log n)) superó ampliamente a Bubble Sort ($O(n^2)$). En un entorno RISC, donde cada instrucción cuenta, la capacidad de Quicksort para minimizar el trabajo redundante sobre la memoria y maximizar la lógica en registros lo posiciona como la opción óptima para el procesamiento de datos a gran escala.