Informe de Laboratorio 2

Michelle Sayas y Brandon Zanotti

July 2025

1 ¿Qué diferencias existen entre registros temporales ($t0–$t9) y registros guardados ($s0–$s7) y cómo se aplicó esta distinción en la práctica?

En la arquitectura MIPS32, la distinción entre registros temporales ($t0-$t9) y registros guardados ($s0-$s7) es fundamental para la convención de llamadas a procedimientos, que establece quién es responsable de preservar el valor de un registro durante una llamada a función. Los registros temporales son "caller-saved" (salvados por el llamador), lo que significa que la función que realiza la llamada debe guardar su contenido si necesita que persista después de que la función llamada retorne, ya que esta última tiene la libertad de modificarlos sin necesidad de restaurarlos. En contraste, los registros guardados son "callee-saved" (salvados por el llamado), implicando que la función a la que se llama es la responsable de almacenar su valor original en la pila al inicio si planea usarlo, y restaurarlo antes de retornar, asegurando así que el llamador recupere el registro con su contenido intacto. Esta convención es crucial para la modularidad, la eficiencia y la correcta interacción entre las funciones en un programa.

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

En MIPS32, los registros $a0-$a3, $v0-$v1 y $ra desempeñan roles específicos dentro de la *convención de llamadas a procedimientos, facilitando el paso de datos entre funciones de manera estandarizada. Los registros $a0-$a3 (argumentos) se utilizan para pasar los primeros cuatro parámetros enteros a una función; son "caller-saved", lo que significa que el llamador no espera que la función llamada preserve sus valores, así que si el llamador los necesita después del retorno, debe guardarlos. Por otro lado, los registros $v0-$v1 (valores de retorno) están designados para que la función llamada almacene los resultados de su operación, con $v0$ típicamente conteniendo el valor principal y $v1$ un segundo valor si es necesario. Finalmente, el registro $ra (dirección de retorno)

es crucial para el flujo de control, ya que automáticamente almacena la dirección de la instrucción siguiente a la llamada jal (jump and link), permitiendo que la función llamada sepa a dónde regresar una vez que finaliza su ejecución, y es responsabilidad del callee guardarlo en la pila si va a realizar más llamadas anidadas para evitar que se sobrescriba. En la práctica, esta distinción permite que los compiladores y los programadores de ensamblador escriban código modular y predecible, donde las funciones pueden interactuar eficientemente sabiendo dónde encontrar sus entradas, dónde colocar sus salidas y cómo volver al punto de llamada.

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

El uso de registros en los algoritmos de ordenamiento como Insertion Sort y Bubble Sort impacta directamente en su rendimiento, principalmente debido a la velocidad de acceso. Los registros son el medio de almacenamiento más rápido de la CPU, superando significativamente la memoria principal (RAM) en latencia. Al mantener variables críticas como índices de bucle (ejem $s0, $s1 en Insertion Sort; $t0, $t1 en Bubble Sort) y valores temporales para intercambios (ejem, $s2 en Insertion Sort; $t2, $t3 en Bubble Sort) directamente en estos registros, los algoritmos minimizan el tiempo que la CPU espera por los datos.

Esta estrategia es fundamental en los bucles intensivos que caracterizan a estos algoritmos. Cada vez que se accede o modifica un dato, el uso de registros reduce drásticamente las lentas operaciones de carga (lw) y almacenamiento (sw) a la memoria. Por ejemplo, en Insertion Sort, la manipulación de aux y los índices i y j se beneficia enormemente de estar en registros, al igual que en Bubble Sort, donde las comparaciones e intercambios de elementos del arreglo (arr[j] y arr[j+1]) se realizan de forma más eficiente al mantener los valores en registros antes de escribir de vuelta a la memoria.

Aunque los registros son rápidos, su número limitado lleva a la necesidad de usar la pila en memoria para guardar y restaurar registros (como $ra y los registros $sX) al inicio y final de las funciones. Esta sobrecarga de memoria es un costo aceptable que permite la modularidad y la corrección de las funciones, mientras que el cuerpo principal del algoritmo se ejecuta con la máxima eficiencia posible gracias al uso predominante de los registros. En síntesis, la priorización del uso de registros mejora considerablemente el rendimiento de estos algoritmos al reducir la latencia y la cantidad de accesos a memoria.

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 bucles anidados y saltos tiene un impacto directo y significativo en la eficiencia de los algoritmos en MIPS32. En esencia, estas estructuras determinan la cantidad de instrucciones que la CPU debe ejecutar, la frecuencia de los accesos a memoria y la predictibilidad del flujo del programa, factores cruciales para el rendimiento.

Los bucles anidados son particularmente costosos en términos de eficiencia. Un algoritmo con bucles anidados (como Bubble Sort o Insertion Sort) ejecutará el código del bucle interno muchas veces por cada iteración del bucle externo. Esto se traduce en un mayor número de comparaciones, accesos a memoria (cargas y almacenamientos de elementos del arreglo) y operaciones aritméticas (incrementos de índices o punteros), que son las operaciones fundamentales del algoritmo de ordenamiento. Cuantas más veces se repitan estas operaciones, más tiempo tardará el programa en completarse, afectando directamente la eficiencia. Por ejemplo, en el código de Bubble Sort, la InnerLoop se ejecuta n-i-1 veces por cada iteración de OuterLoop, llevando a una complejidad de tiempo de $O(n^2)$ para un arreglo de tamaño n.

Los saltos (jumps y branches), que son los bloques de construcción de los bucles y las condicionales, también influyen en la eficiencia. Cada salto implica que la CPU debe desviar su flujo de ejecución normal. Los saltos condicionales (beqz, blez, bge, ble, bgtz en los ejemplos proporcionados) pueden introducir problemas de predicción de bifurcaciones (branch prediction) en procesadores segmentados. Si el procesador predice incorrectamente el resultado de un salto condicional, tendrá que vaciar su pipeline y cargar las instrucciones correctas, lo que provoca un "stall" o retardo en la ejecución. En bucles que se repiten muchas veces, una mala predicción de bifurcación puede acumularse y ralentizar considerablemente el rendimiento. Además, cada salto incondicional (j) o de retorno (jr) también consume ciclos de CPU.

En resumen, aunque las estructuras de control son indispensables para la lógica de los algoritmos, su uso impacta en la eficiencia al:

4.1 Aumentar el número total de instrucciones ejecutadas, especialmente con bucles anidados.

4.2 Incrementar la frecuencia de accesos a memoria dentro de los bucles, lo cual es más lento que las operaciones de registro.

4.3 Introducir penalizaciones por predicción de bifurcaciones erróneas, especialmente en bucles con condiciones internas que no son fácilmente predecibles para el hardware.

Por lo tanto, al diseñar algoritmos en MIPS32, minimizar las iteraciones de los bucles y optimizar las condiciones de salto para mejorar la predictibilidad de las bifurcaciones son consideraciones clave para lograr una alta eficiencia.

5 ¿Cuáles son las diferencias de complejidad computacional entre el algoritmo Bubble Sorty el algoritmo alternativo? ¿Qué implicaciones tiene esto para la implementación en un entorno MIPS32?

Las complejidades computacionales de Bubble Sort e Insertion Sort son similares en el peor y caso promedio, pero difieren en su rendimiento en el mejor caso, lo que tiene implicaciones importantes para la implementación en MIPS32.

Ambos algoritmos tienen una "complejidad de tiempo de $O(n^2)$ en el peor y caso promedio". Esto significa que a medida que el número de elementos (n) en el arreglo aumenta, el tiempo de ejecución crece cuadráticamente. Para Bubble Sort, esto se debe a sus bucles anidados que realizan comparaciones e intercambios repetidamente en cada pasada, moviendo el elemento más grande (o más pequeño) a su posición correcta en cada iteración del bucle externo. De manera similar, Insertion Sort también emplea bucles anidados, donde el bucle interno desplaza elementos para insertar el elemento actual en su posición ordenada, lo que también resulta en una complejidad cuadrática.

Sin embargo, en el "mejor caso", donde el arreglo ya está ordenado, Insertion Sort tiene una complejidad de tiempo de $O(n)$, mientras que Bubble Sort sigue siendo $O(n^2)$. Insertion Sort puede lograr esta eficiencia lineal porque, en un arreglo ya ordenado, el bucle interno de inserción solo realiza una comparación por cada elemento y no realiza intercambios significativos, ya que cada elemento ya está en su lugar correcto. En contraste, Bubble Sort siempre realiza un número cuadrático de comparaciones, incluso si el arreglo está ordenado,

ya que sus bucles externos e internos seguirán iterando a través de casi todos los elementos en cada pasada. Las implicaciones para la implementación en un entorno MIPS32 son las siguientes:

1) Rendimiento con Datos Parcialmente Ordenados: Si se espera que los datos de entrada a menudo estén parcial o casi ordenados, *Insertion Sort será significativamente más eficiente en MIPS32* que Bubble Sort. La menor cantidad de operaciones de desplazamiento y comparaciones en el mejor caso se traduce directamente en menos instrucciones MIPS ejecutadas, menos accesos a memoria y una ejecución más rápida. Esto es crucial en MIPS32, donde cada instrucción y acceso a memoria contribuye al tiempo total de ejecución.

2) Operaciones por Iteración: Ambos algoritmos involucran un alto número de operaciones dentro de sus bucles anidados, como cargas (lw), almacenamientos (sw), operaciones aritméticas (addi, sll, add) y saltos condicionales (bge, blez, ble, etc.). Dado que MIPS32 es una arquitectura RISC con un conjunto de instrucciones simple y de ciclo único, la eficiencia se logra minimizando el número total de instrucciones. Aunque ambos son $O(n^2)$, el factor constante (el número de instrucciones por iteración) puede variar ligeramente. Insertion Sort podría tener un factor constante ligeramente menor en el caso promedio debido a su patrón de acceso a la memoria más local (desplazamientos) en comparación con los intercambios más dispersos de Bubble Sort.

3) Uso de Registros y Pila: Ambos algoritmos optimizan el rendimiento utilizando registros para variables críticas (índices, valores temporales) para reducir los accesos a memoria, que son lentos. Sin embargo, la frecuencia de las operaciones de carga y almacenamiento en la pila (para preservar registros como *ray*sX) será proporcional al número de llamadas a funciones o a la necesidad de preservar el estado del llamador/llamado. Dado que ambos son algoritmos iterativos implementados en una sola función, el impacto de la pila se limita a la entrada y salida de la función de ordenamiento.

En resumen, aunque tanto Bubble Sort como Insertion Sort son $O(n^2)$ en el peor y caso promedio, la ventaja de Insertion Sort de ser $O(n)$ en el mejor caso lo hace más adecuado para escenarios donde los datos de entrada pueden estar parcial o completamente ordenados, resultando en una implementación más eficiente en MIPS32 bajo esas circunstancias.

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, el ciclo de ejecución de instrucciones (también conocido como camino de datos o datapath) se divide generalmente en cinco fases

principales, que permiten a la CPU procesar una instrucción completa:

6.1 Fetch (Búsqueda de Instrucción):

En esta fase, la CPU obtiene la instrucción a ejecutar de la memoria. El *Program Counter (PC), un registro especial, contiene la dirección de la siguiente instrucción. La instrucción se lee de la memoria en la dirección indicada por el PC y se almacena en el registro Instruction Register (IR). Simultáneamente, el PC se incrementa para apuntar a la siguiente instrucción (PC = PC + 4, ya que las instrucciones MIPS son de 4 bytes).

6.2 Decode (Decodificación de Instrucción y Lectura de Registros):

La instrucción que está en el IR se decodifica para determinar qué operación debe realizarse (ej., sumar, cargar, almacenar, saltar). Al mismo tiempo, se leen los valores de los registros especificados en la instrucción. MIPS es una arquitectura RISC (Reduced Instruction Set Computer), lo que significa que la mayoría de las instrucciones operan sobre registros. Por lo general, se leen los valores de hasta dos registros fuente especificados por la instrucción.

6.3 Execute (Ejecución de Operación o Cálculo de Dirección):

Esta es la fase donde se realiza la operación principal.

Para instrucciones aritméticas/lógicas (tipo R, como add, sub, and): La ALU (Arithmetic Logic Unit) realiza la operación especificada (suma, resta, AND, OR, etc.) utilizando los valores leídos de los registros en la fase de decodificación.

Para instrucciones de carga/almacenamiento (tipo I, como lw, sw): La ALU calcula la dirección de memoria efectiva sumando el contenido de un registro base con un desplazamiento inmediato (offset).

Para instrucciones de salto/bifurcación (tipo J o I): La ALU puede calcular la dirección del destino del salto.

6.4 Memory Access (Acceso a Memoria):

Esta fase se utiliza solo si la instrucción actual requiere acceso a la memoria de datos (no a la memoria de instrucciones).

Para instrucciones de carga (lw): Se lee un dato de la memoria en la dirección calculada en la fase de ejecución.

Para instrucciones de almacenamiento (sw): Se escribe un dato (tomado de un registro) en la memoria en la dirección calculada en la fase de ejecución.

Para otras instrucciones (aritméticas, lógicas, saltos): Esta fase puede ser inactiva o se puede realizar algún control secundario.

6.5 Write Back (Escritura de Resultado en Registro):

En esta fase final, el resultado de la operación se escribe de vuelta en el banco de registros.

Para instrucciones aritméticas/lógicas: El resultado de la ALU se escribe en el registro destino especificado en la instrucción.

Para instrucciones de carga: El dato leído de la memoria se escribe en el registro destino especificado.

Para otras instrucciones (almacenamiento, saltos): Esta fase puede ser inactiva, ya que no modifican directamente los registros de propósito general con un resultado de operación.

Estas cinco fases son la base del diseño de pipelines en procesadores MIPS, donde se intenta solapar la ejecución de diferentes fases de distintas instrucciones para mejorar el rendimiento global.

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

Se emplearon predominantemente instrucciones de tipo R (Registro) y de tipo I (Inmediato). Las instrucciones de tipo R son fundamentales para la manipulación de datos directamente en los registros del procesador. Esto incluye operaciones aritméticas y lógicas esenciales para el cálculo de índices, la actualización de contadores de bucle y la preparación de direcciones de memoria. Ejemplos de esto se ven en add para sumas y sll para desplazamientos, cruciales para el manejo de los datos del arreglo durante el ordenamiento.

Por otro lado, las instrucciones de tipo I son ampliamente utilizadas para el *acceso a memoria y el control de flujo*. Las instrucciones lw (load word) y sw (store word) son vitales para leer y escribir elementos del arreglo desde y hacia la memoria, ya que los datos del arreglo residen allí. Además, las instrucciones de bifurcación condicional (bge, blez, ble, bgtz), que son de tipo I, son indispensables para implementar la lógica de los bucles (OuterLoop, InnerLoop, forExt, while) y las decisiones de intercambio dentro de ambos algoritmos. Las instrucciones li y addi también son comunes para inicializar valores o ajustar el puntero de pila, siendo de tipo I.

Las instrucciones de tipo J (Jump), aunque cruciales para la estructura general del programa, son menos frecuentes en el núcleo iterativo de los algoritmos de ordenamiento. Se utilizan principalmente para saltos incondicionales (j) que dirigen el flujo de ejecución entre las diferentes secciones del código o para las llamadas a procedimientos (jal) y retornos (jr $ra). Su rol es más bien de navegación entre subrutinas que de procesamiento intensivo de datos dentro de los bucles de ordenamiento.

En síntesis, la alta prevalencia de instrucciones de tipo R e I en estos algoritmos de ordenamiento refleja la naturaleza de la arquitectura MIPS como un sistema "Load-Store", donde la mayoría de las operaciones se realizan sobre registros, y la interacción con la memoria se maneja a través de instrucciones de carga y almacenamiento específicas, complementadas por bifurcaciones para controlar la lógica iterativa y condicional.

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 abuso de instrucciones de salto (como j, beq, bne) en lugar de estructuras de código más lineales afecta negativamente el rendimiento en la arquitectura MIPS32, principalmente por el impacto en la predicción de bifurcaciones (branch prediction) y la eficiencia del pipeline del procesador.

Cuando el procesador ejecuta un flujo de instrucciones, intenta predecir qué camino tomará una bifurcación condicional (beq, bne) antes de que se conozca el resultado de la condición. Si la predicción es correcta, el pipeline puede seguir llenándose con las instrucciones anticipadas, manteniendo una ejecución fluida. Sin embargo, si la predicción es incorrecta, el procesador debe *descartar las instrucciones que ya había cargado en el *pipeline, recuperar el estado anterior y comenzar a cargar las instrucciones correctas desde la nueva dirección de salto. Este proceso, conocido como "pipeline stall" o "flush", introduce un retardo significativo, ya que varios ciclos de reloj se pierden sin realizar trabajo útil. Un número excesivo de saltos impredecibles incrementa la probabilidad de estas penalizaciones.

Además, un código con muchos saltos rompe la localidad espacial de las instrucciones. Idealmente, el procesador prefiere ejecutar instrucciones que se encuentran contiguas en memoria, ya que esto permite una carga eficiente desde la caché de instrucciones. Cuando hay muchos saltos, especialmente a direcciones distantes, el procesador tiene que buscar instrucciones en diferentes ubicaciones de memoria, lo que puede resultar en más fallos de caché de instrucciones (instruction cache misses). Un fallo de caché significa que el procesador debe ir

a la memoria principal (mucho más lenta) para obtener la instrucción, lo que introduce latencia adicional y ralentiza la ejecución. Las estructuras lineales, en contraste, minimizan los saltos y favorecen la localidad espacial, permitiendo que el pipeline se mantenga lleno y que la caché de instrucciones sea más efectiva.

En resumen, el abuso de instrucciones de salto reduce el rendimiento al:

1) Aumentar las penalizaciones por predicción de bifurcaciones erróneas, obligando al pipeline a vaciarse y recargarse.

2) Aumentar las penalizaciones por predicción de bifurcaciones erróneas, obligando al pipeline a vaciarse y recargarse.

Esto significa que, aunque los saltos son esenciales para la lógica condicional y los bucles, un diseño de algoritmo que minimice su uso o que los haga más predecibles puede resultar en una ejecución más eficiente en MIPS32.

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

El modelo RISC (Reduced Instruction Set Computer) de MIPS ofrece una serie de ventajas fundamentales que impactan directamente en la eficiencia de la implementación de algoritmos básicos como los de ordenamiento. La piedra angular de MIPS es su conjunto de instrucciones reducido y simple, lo que permite un diseño de hardware más sencillo y, crucialmente, la implementación de un pipeline de instrucciones altamente eficiente y rápido*. Para algoritmos de ordenamiento, que son inherentemente intensivos en el número de operaciones ejecutadas (comparaciones, intercambios y accesos a memoria dentro de bucles anidados), un pipeline fluido minimiza los "stalls" o bloqueos, maximizando la capacidad del procesador para ejecutar múltiples instrucciones simultáneamente en diferentes etapas de su ciclo de vida. Esto se traduce en un procesamiento más rápido de cada operación fundamental.

Una ventaja distintiva de MIPS es su filosofía de ejecución de un ciclo por instrucción para muchas de sus operaciones. Esta uniformidad simplifica el diseño de la unidad de control y permite que los procesadores MIPS operen a frecuencias de reloj más altas. En el contexto de algoritmos de ordenamiento, donde cada suma, carga, almacenamiento o comparación contribuye al tiempo total de ejecución, la capacidad de completar estas operaciones en un solo ciclo de reloj acelera significativamente el rendimiento general del algoritmo.

Otro pilar del diseño RISC de MIPS es la provisión de un amplio banco de registros de propósito general. Esta característica es particularmente beneficiosa

para los algoritmos de ordenamiento. Permite que las variables intermedias que se utilizan con mucha frecuencia, como los índices de bucle, los valores temporales durante los intercambios o los contadores, se mantengan directamente en los registros del CPU. El acceso a los registros es órdenes de magnitud más rápido que el acceso a la memoria principal. Al minimizar la necesidad de realizar costosas operaciones de carga (lw) y almacenamiento (sw) desde y hacia la memoria principal, que son relativamente lentas, se logra una mejora sustancial en la velocidad de ejecución de los bucles intensivos que caracterizan a los algoritmos de ordenamiento.

Finalmente, el modelo "Load/Store" de MIPS simplifica tanto el diseño del hardware como la tarea de los compiladores. En MIPS, solo las instrucciones explícitas lw y sw interactúan directamente con la memoria de datos; todas las demás operaciones aritméticas y lógicas se realizan exclusivamente sobre los datos que ya residen en los registros. Esta clara separación entre el procesamiento y el acceso a la memoria simplifica el control del pipeline y permite que los compiladores generen código máquina altamente optimizado, compacto y eficiente para implementar las operaciones de ordenamiento. Esto asegura que el cuello de botella de la memoria esté bien definido y pueda ser manejado de manera más efectiva.

10 ¿Cómo se usó el modo de ejecución paso a paso (Step, Step Into) en MARS para verificar la correcta ejecución del algoritmo?

El modo de ejecución paso a paso (Step, Step Into) en el simulador MARS fue una herramienta crucial para verificar la correcta ejecución de los algoritmos de ordenamiento de la siguiente manera:

10.1 Observación del Flujo de Control:

Al ejecutar el código instrucción por instrucción, pudimos ver exactamente cómo el programa navegaba a través de los bucles y las condicionales. Esto nos permitió confirmar que los saltos (j, beq, bne) y las bifurcaciones condicionales (bge, blez, ble, bgtz) estaban tomando el camino esperado según la lógica del algoritmo. Por ejemplo, en Insertion Sort, se pudo verificar si el bucle while se ejecutaba el número correcto de veces o si la condición bge $s2, $t2, whileExit se cumplía o no en el momento esperado.

10.2 Monitoreo del Estado de los Registros:

El panel de registros en MARS fue fundamental. En cada paso, podíamos observar cómo los valores en los registros cambiaban. Esto incluía:

Registros de argumento ($a0$a3): Verificar que los parámetros de entrada (dirección del arreglo, longitud) se pasaban correctamente a las funciones insertionSort o bubbleSort.

Registros temporales ($t0-$t9): Observar cómo se utilizaban para cálculos intermedios y si sus valores se actualizaban como se esperaba (ej., $t0 para la dirección actual del elemento en Insertion Sort, o $t2, $t3 para los elementos a comparar en Bubble Sort).

Registros guardados ($s0-$s7): Confirmar que estos registros se guardaban correctamente en la pila al inicio de las funciones (sw $sX, X($sp)) y se restauraban antes de regresar (lw $sX, X($sp)), asegurando que los valores del llamador no se corrompieran.

Puntero de pila ($sp): Verificar que el puntero de pila se ajustaba correctamente (addi $sp, $sp, -X) para asignar espacio y se restauraba (addi $sp, $sp, X) al salir de las funciones.

Registro de dirección de retorno ($ra): Asegurarse de que$ra contenía la dirección correcta después de una llamada jal y que se restauraba desde la pila cuando la función anidada regresaba.

10.3 Inspección de la Memoria: La vista de la memoria en MARS permitió ver el contenido del arreglo en tiempo real. Esto fue crucial para:

* Verificar los intercambios: Observar si los elementos del arreglo se movían a sus posiciones correctas después de cada iteración de los bucles de ordenamiento. Por ejemplo, en Bubble Sort, confirmar que sw $t2, 4($s0) y sw $t3, 0($s0) realmente intercambiaban los valores en memoria.

Validar las operaciones de carga/almacenamiento: Asegurarse de que lw leía el valor correcto de la dirección esperada y que sw escribía el valor correcto en la posición deseada

10.4 Detección de Errores Lógicos y de Implementación: El paso a paso facilitó la identificación de errores sutiles, como:

* Bucles infinitos: Si un contador no se actualizaba correctamente o una condición de salida no se cumplía.

Condiciones de borde: Probar el algoritmo con arreglos vacíos, arreglos de un solo elemento o arreglos ya ordenados/invertidos para ver cómo se comportaba

el código en esos casos límite.

Errores de indexación o direccionamiento: Identificar si el algoritmo intentaba acceder a una dirección de memoria incorrecta, lo que podría llevar a resultados inesperados o errores de acceso.

En resumen, el modo paso a paso de MARS, junto con la observación de registros y memoria, fue indispensable para una depuración minuciosa, permitiendo verificar que cada instrucción contribuía al comportamiento esperado del algoritmo y que los datos se manipulaban correctamente en todas las fases de la ejecución.

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 registros combinado con el modo de ejecución paso a paso (Step/Step Into).

El panel de registros permitía ver en tiempo real cómo los valores de cada registro (como $a0-$a3, $v0-$v1,$t0-$t9, $s0-$s7, $sp,$ra, etc.) cambiaban con cada instrucción ejecutada. Esta observación directa era crucial para:

Verificar si los parámetros se pasaban correctamente a las funciones.

Confirmar que los valores temporales e índices de bucle se actualizaban como se esperaba.

Asegurar que los registros guardados y el puntero de pila se gestionaban correctamente según la convención de llamadas.

Al avanzar instrucción por instrucción con "Step" o "Step Into", se podía seguir el flujo exacto del programa, lo que facilitaba la identificación de cuándo y por qué un registro adquiría un valor incorrecto, o cuándo una bifurcación no tomaba el camino deseado, revelando así errores lógicos y de implementación.

12 ¿Cómo puede visualizarse en MARS el camino de datos para una instrucción tipo R?(por ejemplo: add)

El procesamiento de una instrucción add $t0, $t1, $t2 se realiza en cinco fases distintas, que puedes seguir de forma simulada en MARS:

12.1 Fetch (Búsqueda de Instrucción):

En esta primera fase, el procesador obtiene la instrucción que debe ejecutar. El Program Counter (PC), que es un registro especial que guarda la dirección de la siguiente instrucción, envía esa dirección a la Memoria de Instrucciones. La instrucción se lee de allí y se coloca en el Instruction Register (IR). Simultáneamente, el PC se actualiza automáticamente para apuntar a la siguiente instrucción en el código, preparándose para el siguiente ciclo. En MARS, esto se ve cuando la línea de código actual se resalta y el valor del PC en el panel de registros cambia.

12.2 Decode (Decodificación de Instrucción y Lectura de Registros):

Con la instrucción ya en el IR, la Unidad de Control la decodifica para entender qué operación se va a realizar (en este caso, una suma). Al mismo tiempo, se identifican los registros que contienen los valores de entrada ($t1 y $t2). Sus contenidos son leídos directamente del *Banco de Registros* y se preparan para la siguiente etapa. En MARS, puedes observar cómo los valores de $t1 y $\widehat{2}$ están presentes y listos para ser usados en el panel de registros.

12.3 Execute (Ejecución de Operación

Esta es la fase donde la operación real ocurre. Los valores leídos de $t1 y $t2 se envían a la ALU (Unidad Aritmético Lógica). La ALU realiza la operación especificada por la instrucción, que en este caso es una suma. El resultado de $t1 + $t2 se calcula aquí. Aunque MARS no muestra una representación gráfica de la ALU, esta es la etapa donde se produce el cálculo del nuevo valor.

12.4 Memory Access (Acceso a Memoria):

Para una instrucción add (que es de tipo R), esta fase no implica acceso a la memoria de datos. Las instrucciones de tipo R operan exclusivamente sobre los valores que ya están en los registros. No hay necesidad de leer o escribir información desde o hacia la memoria principal en este punto. Por lo tanto, no observarás ningún cambio en el panel de memoria de datos de MARS durante esta fase para una instrucción add.

12.5 Write Back (Escritura de Resultado en Registro):

Finalmente, el resultado que la ALU calculó en la fase de ejecución se escribe de vuelta en el Banco de Registros. En el caso de add $t0, $t1, $t2, el valor de la suma se almacena en el registro destino especificado, que es $t0. En MARS, esto se visualiza claramente en el panel de registros, donde el valor de $t0 se actualizará con el nuevo resultado y a menudo se resaltará para indicar el cambio.

Siguiendo estos pasos en el modo de ejecución paso a paso de MARS y observando los paneles de registros, texto y, ocasionalmente, memoria, se puede comprender cómo cada instrucción se procesa a través del camino de datos del procesador MIPS.

13 ¿Cómo puede visualizarse en MARS el camino de datos para una instrucción tipo I? (por ejemplo: lw)

Para visualizar el camino de datos de una instrucción tipo I como lw $t0, 8($s0) en MARS, puedes seguir el proceso paso a paso:

13.1 Fetch (Búsqueda de Instrucción):

En esta fase, el procesador obtiene la instrucción lw $t0, 8(s0)$ de la Memoria de Instrucciones. El PC, que es un registro especial que guarda la dirección de la siguiente instrucción, envía esa dirección. El valor del PC en el panel "Registers" mostrará la dirección de esta instrucción y luego se incrementará en 4 para apuntar a la siguiente tarea. La instrucción leída se coloca internamente en el IR.

13.2 Decode (Decodificación de Instrucción y Lectura de Registros):

Una vez que la instrucción está en el IR, la Unidad de Control la decodifica para entender que es una operación de carga (lw). Al mismo tiempo, se identifica el registro base ($s0) cuyo valor es esencial para calcular la dirección de memoria. Los contenidos de este registro son leídos directamente del Banco de Registros y se preparan para la siguiente etapa. En MARS, puedes observar el valor del registro $s0 en el panel "Registers", confirmando que está listo para ser utilizado.

13.3 Execute (Ejecución de Operación - Cálculo de Dirección):

En esta fase, la ALU se utiliza para calcular la dirección de memoria exacta. Para lw $t0, 8($s0), la ALU sumará el contenido del registro $s0 (obtenido en la fase anterior) con el valor inmediato 8. El resultado de esta suma es la dirección de memoria efectiva de donde se obtendrá el dato. Aunque MARS no muestra una representación gráfica de la ALU, internamente el simulador realiza este cálculo.

13.4 Memory Access (Acceso a Memoria):

Esta es la fase distintiva para las instrucciones de carga y almacenamiento. La dirección efectiva calculada en la fase anterior se envía a la Memoria de Datos. El dato que se encuentra almacenado en esa dirección específica se lee de la memoria y se prepara para ser escrito en un registro. En MARS, puedes observar el panel "Data Segment" (o "Mem"). Si la dirección de memoria calculada está en el rango visible, podrás ver el valor del dato en esa ubicación* y cómo se "extrae" de la memoria en este paso.

13.5 Write Back (Escritura de Resultado en Registro):

Finalmente, el dato que fue leído de la Memoria de Datos en la fase anterior se escribe en el Banco de Registros. Para lw$t0, 8($s0), este dato se almacena en el registro destino especificado, que es $t0. En MARS, la visualización es muy clara: el valor del registro $t0 en el panel "Registers" cambiará para mostrar el dato recién cargado de la memoria*, y a menudo el registro se resaltará para indicar que ha sido modificado.

Al avanzar paso a paso (Step o Step Into) en MARS, puedes seguir visualmente cómo la instrucción lw progresa a través de cada una de estas fases, observando los cambios en los paneles correspondientes del simulador.

14 Justificar la elección del algoritmo alternativo

La elección del algoritmo Insertion Sort debido a su eficiencia en casos específicos y su simplicidad de implementación en MIPS32. Aunque su complejidad en el peor y caso promedio is O (n 2), Insertion Sort destaca por su rendimiento de O(n) en el mejor caso, es decir, cuando el arreglo de entrada ya está ordenado o casi ordenado. Esta característica lo hace particularmente adecuado para escenarios donde se espera que los datos de entrada a menudo cumplan esta condición, resultando en un menor número de operaciones y, por ende, una ejecución más rápida en una arquitectura RISC como MIPS32. Además, al ser un algoritmo in-place (no requiere memoria adicional significativa) y estable (mantiene el orden relativo de elementos iguales), su implementación en ensamblador puede ser directa y optimizada, aprovechando el uso eficiente de registros para manejar los índices y los desplazamientos de elementos, tal como se observa en el código proporcionado.

15 Análisis y Discusión de los Resultados

La implementación de algoritmos de ordenamiento en MIPS32 no solo pone en práctica conceptos fundamentales de arquitectura RISC, sino que también permite comprender a fondo cómo el manejo eficiente de registros, memoria y

control de flujo impacta en el rendimiento de un programa. A través del análisis detallado de Bubble Sort e Insertion Sort, se evidenció cómo las decisiones de diseño algorítmico y el seguimiento riguroso de convenciones de llamada pueden optimizar la ejecución en una arquitectura segmentada. Además, el uso del simulador MARS demostró ser clave en la validación y depuración del comportamiento del código, reforzando la importancia de herramientas prácticas en el aprendizaje de bajo nivel. La elección de Insertion Sort se justifica no solo por su eficiencia en casos favorables, sino también por su facilidad de implementación y coherencia con las fortalezas de MIPS32, consolidando este trabajo como un ejercicio integral entre teoría y práctica.