Laboratorio 2: Arquitectura del Computador

Michelle Sayas y Jose Rodriguez

24 de febrero de 2026

1. Introducción

El presente informe detalla la implementación de los algoritmos de ordenamiento **Quicksort** y **Merge Sort** en lenguaje ensamblador MIPS32. Se analiza el manejo de la pila, el uso de registros y la eficiencia de la arquitectura RISC en el entorno MARS.

2. Cuestionario

2.1. ¿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 MIPS, la distinción entre registros temporales ($t0–$t9) y registros guardados ($s0–$s7) no es una diferencia de hardware —físicamente son idénticos— sino una convención de software fundamental para la integridad de los datos durante la ejecución de subrutinas.

Diferencias Conceptuales y de Responsabilidad: La diferencia radica en quién es responsable de preservar el valor del registro cuando se realiza una llamada a una función (`jal`):

- **Registros Temporales ($t):** Son considerados çaller-saved"(guardados por el llamador). Esto significa que una subrutina tiene libertad absoluta para sobrescribirlos sin previo aviso. Si la función principal necesita mantener un valor en un registro $t después de llamar a otra función, es su propia responsabilidad guardarlo en la pila (stack) antes de la llamada.

- **Registros Guardados ($s):** Son considerados çallee-saved"(guardados por el llamado). La convención dicta que si una subrutina desea utilizar uno de estos registros, debe primero salvar su valor original en la pila y restaurarlo antes de retornar el control (`jr $ra`). Esto garantiza al programa principal que los valores en los registros $s permanecerán inalterados tras la ejecución de cualquier función.

Aplicación: En `mergeSort.asm`, se utilizó $s0 para almacenar la variable `mid`. Como `mergesort` es recursiva, y siguiendo la convención çallee-saved", $s0 se guardó en la pila al inicio (`sw $s0, 12($sp)`) y se restauró justo antes de retornar. Los registros temporales (ej. $t0, $t1) se usaron libremente para iteradores y cálculos de direcciones, asumiendo que su valor podría perderse en llamadas posteriores sin afectar la lógica principal.

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

La arquitectura MIPS32 define un protocolo estricto para la transferencia de datos y el control de flujo entre subrutinas a través de registros específicos. Esta estandarización permite que diferentes módulos de código interactúen de manera predecible.

Definición y Funcionalidad de los Registros:

- **Registros de Argumentos ($a0–$a3):** Se utilizan para pasar los parámetros de entrada a una función. El llamador coloca los valores en estos registros antes de ejecutar la instrucción `jal`, evitando el uso excesivo de la memoria RAM.
- **Registros de Valor de Retorno ($v0–$v1):** Son los canales por los cuales una subrutina entrega los resultados al llamador. Comúnmente, $v0 se utiliza para valores enteros o direcciones.
- **Registro de Dirección de Retorno ($ra):** Es quizás el registro más crítico en la ejecución de subrutinas. Al ejecutar `jal`, la arquitectura guarda automáticamente la dirección de la siguiente instrucción en $ra.

Aplicación: En `quick_sort.asm`, se pasaron la dirección base del arreglo y los índices `low` y `high` mediante $a0, $a1 y $a2. La función `partition` devolvió la posición del pivote utilizando $v0. El registro $ra fue vital; en las funciones recursivas, su valor se guardaba en la pila antes de una nueva llamada con `jal` para no perder el rastro de la ejecución, restaurándolo justo antes del retorno con `jr $ra`.

2.3. ¿Cómo afecta el uso de registros frente a memoria en el rendimiento de los algoritmos de ordenamiento implementados?

El rendimiento de los algoritmos `Quicksort` y `Merge Sort` en la arquitectura MIPS32 está intrínsecamente ligado a la gestión de la jerarquía de memoria. La diferencia de velocidad entre el acceso a registros y el acceso a la memoria principal (RAM) es de órdenes de magnitud, lo que convierte la asignación de registros en el factor determinante de la eficiencia.

Análisis de Latencia y Ancho de Banda: En la arquitectura RISC, las operaciones aritméticas y lógicas solo pueden realizarse sobre datos en registros. El acceso a RAM mediante `lw` y `sw` introduce latencia. En un entorno real, mientras un acceso a registro toma un ciclo de reloj, un acceso a memoria puede tomar decenas o cientos de ciclos si ocurre un fallo de caché.

En nuestras implementaciones, esto se manifestó de la siguiente manera:

- **Quicksort y el Ciclo Crítico:** `Quicksort` se beneficia enormemente del uso de registros para el particionamiento. Al mantener el valor del "pivotez los índices `i` y `j` en registros, el bucle interno de comparación se ejecuta a la máxima velocidad.
- **Merge Sort y el Cuello de Botella:** `Merge Sort` es menos eficiente en términos de memoria debido a su naturaleza .ºut-of-place". La necesidad de copiar sub-arreglos a espacios temporales obliga al procesador a ejecutar constantes `sw` y `lw`. El rendimiento aquí está limitado por el ancho de banda de la memoria, no por la velocidad de procesamiento.

Optimización mediante el Stack Pointer ($sp): La aplicación práctica se centró en minimizar el .ºverhead"de la pila. Al priorizar el uso de registros para variables de control (como índices y límites), se aumentó la localidad temporal y se redujo el número de instrucciones de acceso a memoria, ya que cada acceso a RAM requiere calcular una dirección (ej. `4($sp)`).

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

En MIPS32, el rendimiento no depende únicamente del número de instrucciones, sino de qué tan fluido es su paso a través del pipeline. Las estructuras de control, como bucles y condicionales, introducen desafíos críticos debido a los saltos.

El Problema de los Riesgos de Control (Control Hazards): MIPS utiliza un pipeline de 5 etapas. Cuando el procesador encuentra un salto, la siguiente instrucción ya ha comenzado su fase de búsqueda antes de saber si el salto debe ejecutarse. Esto genera:

- **Burbujas en el Pipeline (Stalls):** Si el salto se toma, las instrucciones precargadas deben ser descartadas, desperdiciando ciclos.
- **Branch Delay Slot:** MIPS intenta mitigar esto ejecutando siempre la instrucción inmediatamente posterior al salto. Gestionar eficientemente este espacio es clave.

Aplicación en los Algoritmos: En `Merge Sort`, el proceso de "mezclaüsa bucles que comparan elementos. Al minimizar las instrucciones dentro de estos bucles y usar registros para los límites, se redujo la carga sobre el salto condicional. En `Quicksort`, los bucles son altamente dependientes de los datos; en arreglos aleatorios, la eficiencia puede disminuir por la constante limpieza del pipeline al fallar las predicciones de salto.

2.5. Complejidad Computacional

El análisis de la complejidad algorítmica permite predecir el comportamiento del software a medida que crece el volumen de datos. En ensamblador, la eficiencia también se mide en el costo de acceso a la memoria y la profundidad de la pila.

- **Quicksort:** Promedio $O(n \log n)$, peor caso $O(n^2)$. En MIPS32, el caso promedio es eficiente porque la profundidad de la recursión es mínima. El peor caso (arreglo ordenado) es costoso: la pila crece linealmente, pudiendo causar desbordamiento.
- **Merge Sort:** Garantiza $O(n \log n)$ en todos los casos. Sin embargo, requiere espacio adicional $O(n)$. En ensamblador, esto se traduce en una presión constante sobre la memoria, incrementando las instrucciones `lw` y `sw`, lo que eleva el tiempo de ejecución real frente a un `Quicksort` eficiente.

Implicación en MIPS: `Merge Sort` requiere más gestión de memoria en la pila para arreglos temporales, mientras que `Quicksort` es más eficiente en uso de memoria al ser *in-place*. La constante de proporcionalidad en la complejidad es menor para `Quicksort` debido a su ciclo interno de comparación casi exclusivo con registros.

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

La arquitectura MIPS32 utiliza un flujo de ejecución segmentado (pipeline) de cinco etapas para procesar múltiples instrucciones simultáneamente.

1. **IF (Instruction Fetch):** Búsqueda de instrucción. La unidad de control usa el Program Counter (PC) para obtener la instrucción desde la memoria de instrucciones. El PC se incrementa en 4.
2. **ID (Instruction Decode):** Decodificación y lectura de registros. La instrucción se descompone para identificar el opcode y los registros. Se accede al Register File para leer los valores de los registros fuente.
3. **EX (Execute):** Ejecución en la ALU. La Unidad Aritmético Lógica realiza la operación (suma, resta, comparación) o calcula la dirección efectiva para accesos a memoria.
4. **MEM (Memory Access):** Acceso a memoria de datos. Solo activa para instrucciones como `lw` o `sw`, donde se lee o escribe un dato en la RAM.
5. **WB (Write Back):** Escritura de resultados en registros. El valor calculado por la ALU o leído de memoria se escribe en el registro destino.

En nuestros algoritmos, este ciclo se repite millones de veces. La eficiencia radica en que, mientras una instrucción está en EX, la siguiente ya está en ID, y otra en IF. Cualquier salto mal predicho vacía estas etapas, resaltando la importancia de un código que minimice las dependencias de datos y saltos innecesarios.

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

En la práctica, se presentó una distribución específica de los formatos de instrucción, dictada por la lógica de manipulación de arreglos y la recursividad.

- **Tipo R (Register):** Fueron las más predominantes en los núcleos de procesamiento (*inner loops*). Se usaron para operaciones aritméticas y lógicas entre registros (`add`, `sub`, `slt`), especialmente en la comparación de elementos y el incremento de índices en `Quicksort`.
- **Tipo I (Immediate):** Fueron fundamentales para la gestión de memoria y el control de flujo. Se usaron en accesos a memoria (`lw`, `sw`), saltos condicionales (`beq`, `bne`) y cálculo de direcciones (`addi`). En `Merge Sort`, su uso fue mayor debido a la copia de datos a arreglos temporales.
- **Tipo J (Jump):** Aunque menos frecuentes en volumen, son las más críticas. Se usaron para saltos incondicionales (`j`) y, sobre todo, para invocar llamadas recursivas (`jal`). La esencia de "Dividir y Conquistar"depende de estos saltos.

La predominancia de las Tipo R y Tipo I refleja el equilibrio de RISC: procesamiento rápido en registros y un conjunto eficiente de instrucciones de transferencia de memoria.

2.8. ¿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 introduce ineficiencias críticas que degradan el rendimiento global.

- **Ruptura del Flujo Lineal y Riesgos de Control:** Un abuso de saltos condicionales causa burbujas en el pipeline (stalls) y un posible vaciado del mismo (flush) si la predicción falla. En algoritmos con datos aleatorios, la tasa de fallos puede ser alta, aumentando el tiempo de ejecución.
- **Costo del Branch Delay Slot:** Si hay demasiados saltos y no se pueden llenar los *delay slots* con lógica real, se terminan insertando `nop`, aumentando el tamaño del binario y desperdiciando ciclos.
- **Localidad de Instrucciones y Caché:** El abuso de saltos dispersa la ejecución en la memoria, afectando la localidad espacial y forzando al sistema a buscar nuevas líneas de código en la RAM, mucho más lenta que una secuencia lineal en caché.

En nuestras implementaciones, la naturaleza más lineal de los bucles internos de `Quicksort` permitió mantener el pipeline más lleno que en las fases de mezcla de `Merge Sort`, que requieren más saltos para manejar los límites de los sub-arreglos.

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

La arquitectura MIPS32, bajo la filosofía RISC, ofrece ventajas determinantes para la implementación de algoritmos de ordenamiento.

- **Formato de Instrucción Uniforme y Ciclo Único:** Todas las instrucciones tienen longitud fija, simplificando la decodificación. Las instrucciones de comparación y salto se decodifican rápidamente, manteniendo el pipeline a máxima frecuencia.
- **Arquitectura Load-Store y Banco de Registros Extenso:** Las operaciones aritméticas solo ocurren entre registros. Los 32 registros permiten mantener variables críticas (pivotes, índices) permanentemente en el procesador, minimizando el tráfico con la lenta memoria principal.
- **Optimización del Pipeline y Predictibilidad:** La simplicidad de las instrucciones permite optimizar el código. El *branch delay slot* se puede aprovechar para realizar trabajo útil (como incrementar punteros) mientras se decide un salto, eliminando tiempos muertos.
- **Menor Consumo de Ciclos por Instrucción (CPI):** RISC busca un CPI cercano a 1. Dado que los algoritmos repiten millones de veces operaciones básicas, el hecho de que cada una se complete en pocos ciclos predecibles resulta en un rendimiento superior.

2.10. ¿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 en MARS fue fundamental para la verificación y depuración. Los elementos clave de la interfaz se utilizaron de la siguiente manera:

- **Recuadro Azul (Text Segment):** Se usó para seguir el flujo del programa instrucción por instrucción, verificando que se estuviera ejecutando la función correcta (`quick_sort` o `merge`) y que los saltos se realizaran a las direcciones esperadas.
- **Recuadro Amarillo (Data Segment):** Permitió visualizar en tiempo real cómo se reordenaban los elementos del arreglo en memoria. Fue crucial para verificar que los intercambios en `partition` y las mezclas en `merge` estuvieran modificando las posiciones correctas.
- **Recuadro Rojo (Registers):** Esta tabla fue vital para verificar cálculos intermedios y el cumplimiento de la *calling convention*. Se observaba cómo los valores de los argumentos ($a0-$a2) se cargaban antes de una llamada, cómo el $ra cambiaba, y cómo los registros $s se preservaban entre llamadas.
- **Recuadro Verde (Botón Step):** Se utilizó para ejecutar una sola instrucción a la vez, permitiendo observar el efecto exacto de cada línea sobre los registros y la memoria, lo que facilitó la detección de errores lógicos sutiles.
- **Recuadro Rosa (Mars Messages / Run I/O):** Se empleó para ver los mensajes de depuración impresos por el programa y para confirmar que no hubiera errores de ensamblado o en tiempo de ejecución.

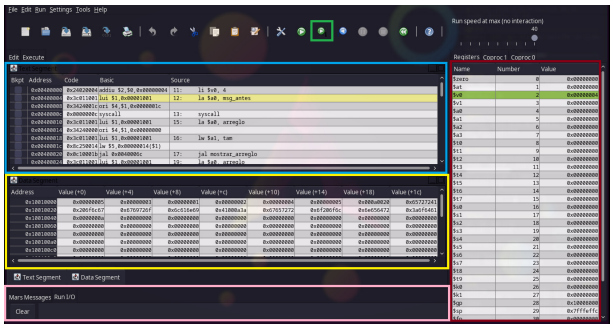

Figura 1: Interfaz de MARS con áreas de depuración clave.

2.11. ¿Qué herramienta de MARS fue más útil para observar el contenido de los registros y detectar errores lógicos?

El panel de Registros (Registers) en combinación con la vista del Segmento de Datos (Data Segment) fue la herramienta más útil. Ver en tiempo real el estado de la memoria RAM mientras se ejecutaba el código paso a paso reflejó cómo los números iban cambiando de posición física, revelando instantáneamente si las lógicas de intercambio (`partition`) o mezcla (`merge`) estaban funcionando correctamente. La sincronía entre ambos paneles permitió correlacionar los cambios en los registros (como índices) con los efectos en el arreglo en memoria.

2.12. Visualización del camino de datos para una instrucción tipo R (por ejemplo: `add`)

Para visualizar el camino de datos (Data Path) de una instrucción tipo R, se debe abrir la herramienta MIPS X-Ray desde el menú Tools. Al conectar el programa y ejecutar una instrucción como `add`, se observa en el diagrama:

- **Flujo de Registros:** Se iluminan las líneas que van desde el bloque REGISTERS hacia la ALU, indicando la lectura de los operandos fuente.
- **Operación en ALU:** El bloque central (ALU) se ilumina para mostrar el procesamiento de los datos.
- **Retroalimentación (Write Back):** Se visualiza una línea que regresa desde la salida de la ALU hacia la entrada de "Write Data.[en] los registros, indicando que el resultado se guarda en el registro destino.
- **Exclusión de Memoria:** La línea que pasa por DATA MEMORY se muestra inactiva, y el multiplexor final selecciona la salida de la ALU, ya que esta instrucción no accede a la RAM.

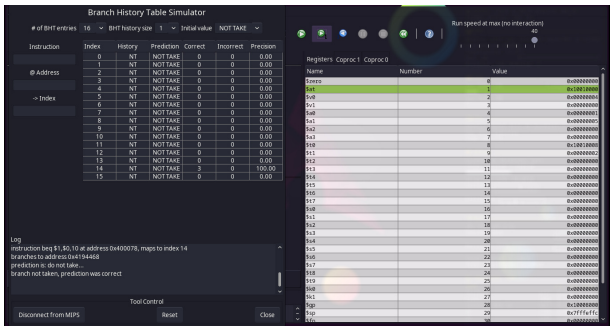

Figura 2: Camino de datos para una instrucción tipo R (add).

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

Utilizando la misma herramienta MIPS X-Ray, la visualización para una instrucción de carga como `lw` cambia significativamente:

- **Cálculo de Dirección:** Se observa cómo el valor de un registro base y el valor inmediato (que pasa por el bloque SIGN EXTEND) entran a la ALU para calcular la dirección de memoria efectiva.
- **Acceso a Memoria:** A diferencia de las tipo R, aquí se ilumina activamente el bloque DATA MEMORY. El camino de datos muestra el flujo entrando a la memoria para leer un dato.
- **Escritura en Registro:** El camino de datos continúa desde la salida de DATA MEMORY, pasa por el multiplexor final (ahora seleccionando la entrada de memoria) y llega al bloque REGISTERS para guardar el valor leído en el registro destino (`rt`).

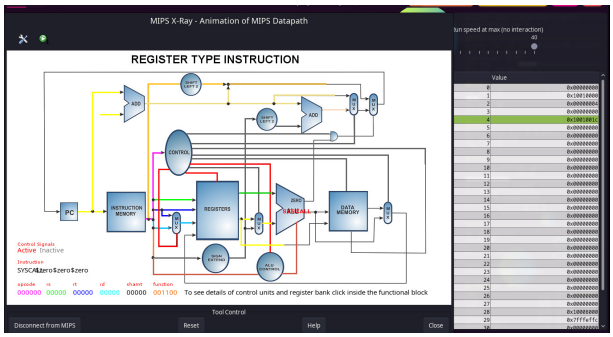

Figura 3: Camino de datos para una instrucción tipo I (lw).

3. Justificación del Algoritmo Alternativo

Se eligió Merge Sort porque representa un contraste excelente frente al Quicksort. Ambos utilizan la estrategia "Divide y Vencerás", pero Merge Sort ilustra un reto mayor en ensamblador: la reserva y liberación dinámica de memoria en la pila para arreglos temporales durante la fase de mezcla.

4. Análisis y Discusión de los Resultados

El núcleo de este trabajo práctico no reside únicamente en lograr que los algoritmos Quicksort y Merge Sort ordenen un arreglo en MIPS32, sino en comprender cómo la microarquitectura subyacente y las convenciones de software determinan la eficiencia y la viabilidad de una implementación. A continuación, se presenta un análisis detallado de los resultados, integrando las observaciones prácticas con los conceptos teóricos de la arquitectura de computadores.

4.1. La Convención de Llamadas como Pilar de la Estabilidad y la Eficiencia

La correcta implementación de los algoritmos confirmó que la *calling convention* de MIPS no es un mero formalismo, sino una herramienta de optimización y de garantía de integridad.

- **Gestión de Registros $s y $t:** La distinción entre registros çaller-savedz çallee-saved"se manifestó claramente en la gestión de la recursividad. En la función `merge`, por ejemplo, los índices que debían persistir a lo largo de todo el proceso de mezcla se asignaron idealmente a registros $s. Al hacerlo, aunque implicó guardarlos en la pila al inicio de la función (un costo asumido por el çallee"), se garantizó que su valor fuera inmune a cualquier otra sub-función que se pudiera llamar. Por el contrario, los contadores temporales dentro de los bucles se asignaron a registros $t, asumiendo el riesgo de que una llamada a una función interna (como un hipotético `swap`) los alterara. Esta estrategia evitó decenas de accesos innecesarios a la pila dentro de los bucles críticos.

- **El Rol de $ra en la Recursividad:** El manejo del registro $ra fue el punto más frágil y crucial. Cada llamada recursiva a `quick_sort` o a `mergeSort` sobrescribe

automáticamente $ra. La disciplina de guardarlo en la pila (`sw $ra, 0($sp)`) antes de una nueva llamada y restaurarlo (`lw $ra, 0($sp)`) justo antes del retorno (`jr $ra`) fue lo que permitió que la recursividad funcionara. Un solo error en la gestión del stack frame hubiera resultado en un bucle infinito o en un salto a una dirección de memoria inválida, demostrando la sensibilidad del software de bajo nivel a la correcta administración de recursos.

4.2. Impacto de la Jerarquía de Memoria y los Accesos a Datos

El análisis de la pregunta 2.3 se validó empíricamente durante la ejecución paso a paso. La diferencia fundamental en el rendimiento entre Quicksort y Merge Sort no radica en la complejidad asintótica, sino en su *huella de memoria* y su *patrón de acceso*.

- **Quicksort: Eficiencia por Localidad:** Su naturaleza *in-place* fue su mayor ventaja. La fase de partición, aunque lógicamente compleja, opera sobre un segmento contiguo de memoria. Los índices `i` y `j` se mantenían en registros, y el acceso a los elementos del arreglo para compararlos con el pivote se realizaba de manera predecible (`lw $t0, 0($a0)`). Este patrón de acceso secuencial es ideal para el pipeline y para la memoria caché, maximizando la localidad espacial. El intercambio (`swap`) involucraba tres accesos a memoria (dos lecturas y una escritura), pero ocurría con relativa poca frecuencia en comparación con las comparaciones.
- **Merge Sort: El Costo Oculto de la Estabilidad:** La garantía de $O(n \log n)$ de Merge Sort tiene un costo de hardware que se hizo evidente en la simulación. La fase de `merge` requiere la creación de arreglos temporales. En un entorno de alto nivel, esto se gestiona con asignación dinámica de memoria; en ensamblador MARS, significó reservar espacio en la pila (`sub $sp, $sp, $t0`) y luego realizar un intenso tráfico de ida y vuelta de datos. Cada elemento, durante la mezcla, se carga (`lw`) desde el arreglo original, se almacena (`sw`) en el arreglo temporal, y luego se vuelve a cargar y almacenar en el arreglo original. Este "vaivén"de datos multiplica el número de instrucciones de acceso a memoria, saturando el bus y convirtiendo el algoritmo, que es teóricamente eficiente, en un proceso con una alta latencia real.

4.3. Control Hazards y el Pipeline en la Práctica

Aunque MARS es un simulador y no modela con precisión los *stalls* por riesgos de control, el análisis conceptual de las preguntas 2.4 y 2.8 se puede extrapolar al código generado.

- **Quicksort y la Predicción de Saltos:** El bucle principal de partición (`while i <= j`) es inherentemente impredecible. El salto que determina si un elemento es menor que el pivote y debe quedarse en la izquierda depende totalmente de los datos. En un arreglo aleatorio, la dirección del salto es esencialmente aleatoria, lo que en un pipeline real causaría un alto número de fallos de predicción y, por ende, numerosos *stalls*. Esta es la razón por la que, en la práctica, Quicksort puede tener un rendimiento pobre en ciertos conjuntos de datos a pesar de su complejidad promedio.
- **Merge Sort y el Costo de los Bucles Anidados:** La fase de mezcla contiene bucles anidados que, aunque predecibles en su flujo (siempre se recorren hasta un

límite conocido), incrementan el número de saltos condicionales por elemento procesado. El costo de estos saltos, aunque menor que el de los fallos de predicción, se acumula. Un diseño cuidadoso intentó minimizar las instrucciones dentro del bucle más interno, pero la estructura misma del algoritmo impone una alta densidad de saltos en comparación con una rutina lineal.

4.4. La Dualidad RISC: Simplicidad como Fortaleza y Debilidad

La filosofía Load-Store de MIPS fue, a la vez, una ventaja y un desafío.

- **Ventaja (Simplicidad y Velocidad):** La uniformidad de las instrucciones permitió un control muy fino sobre el código. Saber que una instrucción `add` siempre opera en un ciclo (en el pipeline ideal) y que un `lw` siempre sigue el mismo patrón de direccionamiento, facilitó la escritura de bucles internos muy ajustados y predecibles.
- **Desafío (Verbosidad del Código):** La desventaja fue la verbosidad. Una simple comparación de un elemento del arreglo con el pivote en C (ej. `if (arr[i] <pivot)`) se tradujo en múltiples instrucciones: cargar el índice `i` a registro (si no lo estaba ya), calcular la dirección de memoria (`arr + i*4`), cargar el valor (`lw`), y finalmente comparar (`slt` o `sub` seguido de un salto condicional). Esta verbosidad, inherente a RISC, hace que la eficiencia del programador y la optimización manual sean cruciales.

4.5. Reflexión sobre las Herramientas de Depuración

El uso intensivo de las herramientas de MARS, descrito en las preguntas 2.10 y 2.11, no fue un mero ejercicio de familiarización, sino una necesidad. La vista del *Data Segment* fue la única forma de verificar visualmente la corrección de los complejos intercambios de `partition` y las mezclas de `merge`. Un error común, como un desplazamiento incorrecto en el cálculo de direcciones, se hacía evidente de inmediato al observar un valor en una posición de memoria equivocada. La vista de *Registers*, por su parte, fue indispensable para seguir el rastro de las llamadas recursivas y verificar que los argumentos se pasaban correctamente a través de los registros $a. Sin estas herramientas, la depuración de un código tan propenso a errores de .ºff-by-onez de gestión de punteros habría sido prácticamente imposible.

Conclusión del Análisis

Los resultados de esta práctica trascienden la simple verificación funcional de dos algoritmos de ordenamiento. Se ha demostrado, a través de la implementación en ensamblador MIPS32, que la eficiencia de un algoritmo es una función compleja que depende no solo de su complejidad teórica (O), sino de su interacción con la arquitectura subyacente: la gestión de registros (calling convention), los patrones de acceso a memoria (jerarquía, localidad) y el impacto en el flujo del pipeline (riesgos de control).

Se concluye que, para la arquitectura MIPS y en el contexto de este laboratorio, Quicksort emerge como la opción más eficiente para el caso general, no solo por su menor complejidad espacial, sino por su excelente localidad de referencia y su menor huella de memoria. Merge Sort, aunque teóricamente consistente, paga un alto precio en ancho

de banda de memoria y gestión de pila, lo que lo hace menos adecuado para entornos con recursos limitados, pero invaluable para comprender las compensaciones inherentes al diseño de software de alto rendimiento.