Practica 2

Adrian Cordero

22 de febrero de 2026

1. Introducción

El estudio de la arquitectura de computadores requiere comprender la estrecha relación entre la eficiencia de un algoritmo y la estructura del procesador que lo ejecuta. En este contexto, la arquitectura MIPS32 (Microprocessor without Interlocked Pipelined Stages) se presenta como un modelo fundamental basado en la filosofía RISC (Reduced Instruction Set Computer), diseñada para maximizar la velocidad de ejecución mediante un conjunto de instrucciones simplificado y un uso intensivo de registros.

El presente informe analiza la implementación y el comportamiento de dos estrategias de ordenamiento: QuickSort, basado en el paradigma de "divide y vencerász recursividad, y BubbleSort, un modelo iterativo de comparación directa. Sin embargo, el análisis no se limita únicamente a la lógica algorítmica; se centra primordialmente en cómo estas instrucciones interactúan con el camino de datos y la segmentación (Pipeline) del procesador.

El concepto de Pipeline es el núcleo de este estudio. Esta técnica permite que el procesador MIPS32 ejecute múltiples instrucciones de forma simultánea, dividiendo el ciclo de vida de cada una en cinco etapas: Búsqueda (IF), Decodificación (ID), Ejecución (EX), Memoria (MEM) y Escritura (WB). Durante la ejecución de los algoritmos de ordenamiento, el flujo de este pipeline se ve constantemente desafiado por riesgos de control (saltos condicionales y llamadas a funciones) y riesgos de datos (dependencia de valores entre instrucciones).

A través del uso del simulador MARS, se busca observar cómo el manejo de registros, la gestión de la pila (stack) y la predicción de saltos determinan el rendimiento real de un programa. Esta transición del código de alto nivel al lenguaje ensamblador permitirá identificar por qué ciertos algoritmos, aunque teóricamente óptimos, deben ser cuidadosamente orquestados para no saturar las unidades funcionales del procesador y aprovechar al máximo la arquitectura segmentada de 32 bits.

2. Diferencias entre Registros $t y $s

Característica	Registros Temporales ($t0–$t9)	Registros Guardados ($s0–$s7)
Persistencia	Su valor puede cambiar tras un **jal**. No se garantiza su integridad.	Su valor debe ser el mismo antes y después de una llamada a función.
Persistencia	Cálculos rápidos, índices de bucles inmediatos y valores intermedios.	Variables importantes que deben sobrevivir a lo largo de toda una subrutina.

En los archivos se ve claramente cómo se aplica esta distinción de dos formas diferentes:

2.1. En BubbleSort: Uso de preservación de registros $s

En tu función BubbleSort, utilizas registros **$s0** y **$s1** para almacenar la base del arreglo y el tamaño.

- Como **BubbleSort** usa estos registros para su propia lógica, debe asegurarse de que, cuando la función termine y regrese al main, esos registros tengan el valor que tenían antes. Al final, los restauras con **lw $s0**, **4($sp)**.

2.2. En QuickSort: Uso de registros $t y recursividad

En la función QuickSort, utilizas mayoritariamente registros temporales (**$t0**, **$t1**, **$t2**, etc.) para los punteros **i**, **j** y el **pivote**.

- Para evitar que los datos se corrompan, guardas los argumentos **$a0**, **$a1** y **$a2** en la pila. Aunque usas **$t** para la lógica interna del **do-while**, nota que después de cada llamada recursiva (jal), el código vuelve a cargar los valores necesarios desde la pila (ej. **lw $a0**, **0($sp)**) porque no puede confiar en que los registros mantengan su valor.

3. Diferencias existen entre los registros $a0–$a3, $v0–$v1, $ra y cómo se aplicó esta a la practica

Registro	Nombre	Función Principal
$a0 – $a3	**Arguments**	Se usan para pasar parámetros a una función (la "entrada").
$v0 – $v1	**Values**	Se usan para devolver resultados al finalizar la función (la "salida") y para configurar Syscalls.
$ra	**Return Address**	Almacena la dirección de retorno (a dónde debe saltar el programa cuando la función termine).

3.1. Aplicación en la Práctica

a. **Paso de Parámetros ($a0 - $a3)**

En ambos códigos, usas estos registros como el "puente"de comunicación entre el main y las subrutinas:

- En QuickSort, el main carga la dirección del arreglo en $a0, el índice izquierdo en $a1 (0) y el derecho en $a2 (n-1).

b. Valores de Retorno y Syscalls ($v0 - $v1)

En los algoritmos, $v0 tiene un rol dual muy claro:

- para decirle al simulador qué operación realizar. Por ejemplo, **li $v0, 1** para imprimir un entero o **li $v0, 10** para finalizar el programa.

c. Dirección de Retorno ($ra) Este es el registro más crítico.

- **En BubbleSort:** Como es una función, el **$ra** se guarda al inicio y se restaura al final para poder hacer **jr $ra.**
- **En QuickSort:** Aquí es vital. Al ser recursivo, cada **jal QuickSort** cambia el valor de **$ra**. Si no hubieras incluido **sw $ra, 12($sp)** al

principio, el programa entraría en un bucle infinito porque perdería el camino de regreso al **main**.

4. Como afecta el uso de registros frente a memoria en el rendimiento de los algoritmos de ordenamiento implementados

4.1. Análisis del Impacto de los Algoritmos

a. En **BubbleSort** (El impacto del acceso repetitivo)

El **BubbleSort** que proporcionado es "pesado.[en] memoria porque:

- En cada iteración del bucle interno (for_j), realizas dos cargas (lw) y, si hay intercambio, dos escrituras (sw).
- Como BubbleSort tiene una complejidad de $O(n^2)$, el número de accesos a memoria crece drásticamente con el tamaño del arreglo.

b. En QuickSort (El impacto de la Pila/Stack)

QuickSort es más eficiente en lógica ($O(n \log n)$ en el caso promedio), pero introduce un uso intensivo de memoria de otro tipo: la Pila.

- Cada llamada recursiva ejecuta: addi $sp, $sp, -16 seguido de cuatro sw.
- Aunque **QuickSort** hace menos comparaciones de elementos, gasta tiempo de CPU gestionando la memoria de la pila para salvar los registros **$ra** y **$a0**.

4.2. Técnicas de Optimización Aplicadas

En los códigos se aplico (quizás sin saberlo) técnicas para mitigar este impacto:

- **Carga Local:** En el swap de **QuickSort**, primero cargas los valores en **$t4** y **$t5**, y luego haces el intercambio.
- Uso de Punteros: En lugar de recalcular la dirección base cada vez, se calcula el offset (sll y add) y se mantiene en un registro temporal para el acceso inmediato.

5. Impacto del uso de estructuras de control en la eficiencia de los algoritmos en MIPS32

El impacto de las estructuras de control en MIPS32 es crítico porque, a diferencia de los lenguajes de alto nivel donde el compilador hace gran parte del trabajo, en ensamblador tú controlas directamente el Contador de Programa (PC) y el flujo de la tubería (Pipeline) de ejecución.

5.1. El Costo de los Saltos y el Pipeline

IPS32 utiliza una arquitectura de pipeline (segmentación) de 5 etapas. Cuando ejecutas un salto (j, jal, jr) o una rama condicional (beq, bne), el procesador a veces no sabe cuál es la siguiente instrucción hasta que se evalúa la condición.

- **Hazard de Control:** Si el procesador predice mal un salto, debe "limpiar"las instrucciones que ya habían entrado al pipeline, lo que genera ciclos perdidos
- **Delay Slot:** En arquitecturas MIPS reales, la instrucción inmediatamente después de un salto se ejecuta siempre (aunque el salto se tome).

5.2. Bucles Anidados (Caso: BubbleSort)

En el código de BubbleSort, hay un bucle for_j dentro de un for_i. El impacto en la eficiencia es masivo por dos razones:

- **Multiplicación de Instrucciones:** Cada instrucción dentro del bucle interno se ejecuta n^2 veces. Por ejemplo, el cálculo del offset (sll \$t4, \$t1, 2) se repite constantemente
- **Ramificaciones Frecuentes:** Al final de cada bucle interno, hay un salto de regreso (j for_j). Esto mantiene al procesador saltando constantemente a direcciones de memoria anteriores, lo que puede afectar la caché de instrucciones.

5.3. Saltos Condicionales vs. Incondicionales

Los algoritmos implementaste ambos:

- **Saltos Condicionales (ble, bge):** Son más costosos para el hardware porque dependen de una comparación previa. En tu QuickSort, las condiciones bge \$t4, \$t2, $while_1$ deciden si el bucle continúa o se rompe.
- **Saltos Incondicionales (j, jr):** Son más eficientes porque el destino es fijo, pero aun así interrumpen el flujo lineal de las instrucciones.

5.4. Eficiencia en la Implementación de QuickSort

A diferencia de BubbleSort, QuickSort usa recursividad (saltos jal).

- **Impacto Negativo:** El uso de jal obliga a manipular la pila ($sp), lo que añade instrucciones de carga y almacenamiento (sw/lw) solo para gestionar el control del programa.
- **Impacto Positivo:** Al dividir el problema (Divide y Vencerás), el número total de veces que se ejecutan los saltos de comparación es mucho menor ($O(n \log n)$), compensando con creces el costo de gestionar la pila.

6. Diferencias de complejidad computacional entre el algoritmo Quicksort y el algoritmo alternativo y sus implicaciones en el entorno MIPS32

6.1. Comparativa de Complejidad Computacional

Algoritmo	Caso Promedio	Peor Caso	Memoria Extra (Espacio
QuickSort	$O(n \log n)$	$O(n^2)$	$O(\log n)$ (Pila de recursión)
BubbleSort	$O(n^2)$	$O(n^2)$	$O(1)$ (In-place)

- **QuickSort:** Es un algoritmo de "divide y vencerás". En cada paso, reduce drásticamente el número de comparaciones necesarias para la siguiente etapa.
- **BubbleSort:** Es un algoritmo de comparación directa. Incluso en su versión optimizada (con el flag swapped), sigue siendo ineficiente para arreglos grandes porque su estructura de bucles anidados lo obliga a procesar casi todos los pares de elementos.

6.2. Implicaciones en un Entorno MIPS32

La diferencia de complejidad no solo se traduce en tiempo, sino en cómo el procesador MIPS32 gestiona sus recursos finitos:

a. Gestión de la Pila (Stack) y Memoria

- **QuickSort:** Al ser recursivo, cada llamada genera un Stack Frame. En MIPS, esto implica múltiples instrucciones sw (save word) para preservar $ra, $a0, etc. Si el arreglo fuera extremadamente grande y el pivote fuera malo, QuickSort podría causar un Stack Overflow.

- **BubbleSort:** No usa la pila para la lógica principal (solo guarda registros al inicio). Es más seguro en términos de memoria, aunque sea más lento.

b. Localidad de Datos y Caché

- **QuickSort:** Al saltar entre particiones (izquierda y derecha), tiene una localidad de datos menos predecible, lo que podría generar más cache misses en arquitecturas MIPS avanzadas.
- **BubbleSort:** Tiene una excelente localidad espacial. Accede a los elementos del arreglo uno tras otro ($v[j]$ y $v[j+1]$). En un procesador con caché, esto es eficiente porque los datos adyacentes ya están cargados.

c. Cantidad de Instrucciones Ejecutadas (Instruction Count)

En MIPS32, la métrica clave es el número de instrucciones.

- **QuickSort:** aunque cada iteración es más compleja (debido al manejo de la pila), el número total de comparaciones es de aproximadamente $10,000$. La ganancia en velocidad es masiva a pesar del "overhead"de la recursión.
- **BubbleSort:** el contador de instrucciones crece de forma cuadrática. Para un arreglo de 1000 elementos, MIPS ejecutaría aproximadamente $1,000,000$ de iteraciones del bucle interno.

7. Fases del ciclo de ejecución de instrucciones en la arquitectura MIPS32

1. **IF (Instruction Fetch) Búsqueda de Instrucción:**
   En esta primera fase, el procesador obtiene la instrucción desde la memoria.

   - Se utiliza el valor del Program Counter (PC) para buscar la dirección de la siguiente instrucción en la memoria de instrucciones.
   - El PC se incrementa automáticamente en 4 (porque cada instrucción MIPS mide 32 bits / 4 bytes) para apuntar a la siguiente instrucción.

2. **ID (Instruction Decode) - Decodificación y Lectura de Registros**
   Aquí el procesador prepara los datos.

   - La Unidad de Control identifica el código de operación (opcode). Simultáneamente, se accede al Banco de Registros para leer los valores de los registros fuente (por ejemplo, en add $t0, $t1, $t2, se leen $t1 y $t2).

- Si la instrucción tiene un valor inmediato (como el -16 en tu addi $sp, $sp, -16), se extiende a 32 bits.

3. **EX (Execute) - Ejecución o Cálculo de Dirección**
   Es el corazón del procesamiento aritmético ALU.
   - La ALU (Unidad Aritmético Lógica) realiza la operación solicitada.
4. **MEM (Memory Access) - Acceso a Memoria**
   Esta fase solo es activada por instrucciones de carga o almacenamiento.
5. **WB (Write Back) - Escritura en Registros**
   El ciclo termina guardando el resultado.
   - El valor calculado en la fase EX o el valor leído en la fase MEM se escribe de vuelta en el registro destino del banco de registros (por ejemplo, en el registro $t0).

Relación con los algoritmos de ordenamiento

Fase	Ejemplo (lw $t4, 0($t6))
IF	Se busca el código binario de la instrucción lw
ID	Se identifica que es un load y se lee el valor del registro base $t6
EX	La ALU suma el contenido de $t6 con el inmediato 0 para obtener la dirección real
MEM	Se accede a la RAM en la dirección calculada para extraer el valor del arreglo
WB	El valor obtenido de la RAM se guarda finalmente en el registro $t4

8. Tipos de instrucciones utilizadas

8.1. Instrucciones Tipo I (Immediate)

Son las que incluyen un valor constante de 16 bits dentro de la instrucción.

- **Ejemplos:**
  - lw
  - sw
  - addi
  - beq
  - bne

    - ble
- **Acceso a Memoria:** Todo ordenamiento requiere mover datos entre la RAM y los registros (lw / sw).
    - Control de Bucles
    - Gestión de la Pila

8.2. Instrucciones Tipo R (Register)

Son aquellas donde todos los operandos son registros.

- **Ejemplos:**
    - add
    - sub
    - sll
    - sra
    - move
    - jr
- **Cálculo de Direcciones:** Para acceder a array[i], usaste sll (shift left logical) para multiplicar el índice por 4 y add para sumarlo a la dirección base.
- **Retorno de Funciones:** La instrucción jr $ra es de tipo R y es la que permite que tus funciones QuickSort y BubbleSort regresen al punto de origen.

8.3. Instrucciones Tipo J (Jump)

Son instrucciones de salto incondicional con una dirección de destino de 26 bits.

- **Ejemplos:**
    - j
    - jal
- **Llamadas a Funciones:** jal QuickSort y jal BubbleSort son fundamentales para modularizar el código.
- **Saltos de Bucle:** El j for_i al final de los ciclos para repetir la ejecución de forma incondicional.

Tipo	Predominio	Razón en la práctica
I	Muy Alto	Debido a la enorme cantidad de accesos a memoria (lw/sw) y constantes para manejar los índices y la pila.
R	Alto	Necesario para la aritmética de punteros (multiplicar el índice por 4) y para realizar los cálculos lógicos entre registros.
J	Bajo	Limitado a las llamadas de función iniciales y los saltos incondicionales de cierre de bucles.

9. Como se ve afectado el rendimiento si se abusa del uso de instrucciones de salto en lugar de las lineales

- Ruptura del Pipeline y Burbujas (Stalls) MIPS32 está diseñado para ejecutar una instrucción en cada ciclo de reloj mediante el solapamiento de fases (IF, ID, EX, MEM, WB).
- Fallos en la Predicción de Saltos. Los procesadores modernos intentan .ªdivinar"si un beq o bne se cumplirá.
- En la arquitectura MIPS original, la instrucción que sigue inmediatamente a un salto (j, beq, bne) se ejecuta siempre, sin importar si el salto se toma o no.
- El rendimiento depende de que las instrucciones estén cerca unas de otras en la memoria. altos Frecuentes: Obligan al "puntero de instrucciones"(PC) a saltar a direcciones lejanas. Esto puede causar un Cache Miss (el procesador no encuentra la instrucción en la memoria rápida y debe ir a la RAM lenta), lo que detiene el sistema por decenas de ciclos.

10. Ventajas ofrece el modelo RISC de MIPS en la implementación de algoritmos

El modelo RISC (Reduced Instruction Set Computer) de MIPS ofrece ventajas estratégicas para los algoritmos de ordenamiento, centrándose en la simplicidad para ganar velocidad.

- Formato de Instrucciones de Tamaño Fijo, , todas las instrucciones miden exactamente 32 bits.
- Arquitectura Carga-Almacenamiento, MIPS no permite realizar operaciones aritméticas directamente sobre la memoria; todo debe pasar por registros.
- Gran Banco de Registros de Propósito General MIPS ofrece 32 registros de 32 bits.
- Ciclo de Ejecución Simple y Predecible (Un ciclo por instrucción), La filosofía RISC apunta a que cada instrucción realice una tarea pequeña que pueda completarse en un solo ciclo de reloj (gracias al Pipeline).

11. Ejecución paso a paso en MARS para verificar la correcta ejecución del algoritmo

En amnbos algoritmos se ejecutaron paso a paso prestando especial atencion en las instruciones de posicionamiento en los arreglos, cambio de valores como:

- swap
- asignaciones (li,la,move,addi)
- incremento de contadores
- saltos condicionales
- saltos incondicionales
- recuperacion de datos guardados en la pila
- recuperacion de la return address
- una subrutina para visualizar el arreglo por terminal

12. Herramienta de MARS fue más útil para observar el contenido de los registros y detectar errores lógicos

El resaltado de ejecución (o Instruction Trace) en MARS fue fundamental para la depuración, ya que permitió mapear visualmente el flujo de control. Al observar el comportamiento del resaltado en las estructuras de bucle del Quick-Sort, se pudo verificar que las condiciones de salida se alcanzaran efectivamente, evitando estados de ciclo infinito mediante la observación directa de los saltos de retorno

13. Visualizar en MARS el camino de datos para una instrucción tipo R y I

Para visualizar el camino de datos de una instrucción tipo R en MARS, la herramienta específica se llama MIPS X-Ray. Esta es una de las funciones más avanzadas del simulador para entender qué ocurre físicamente dentro del procesador.

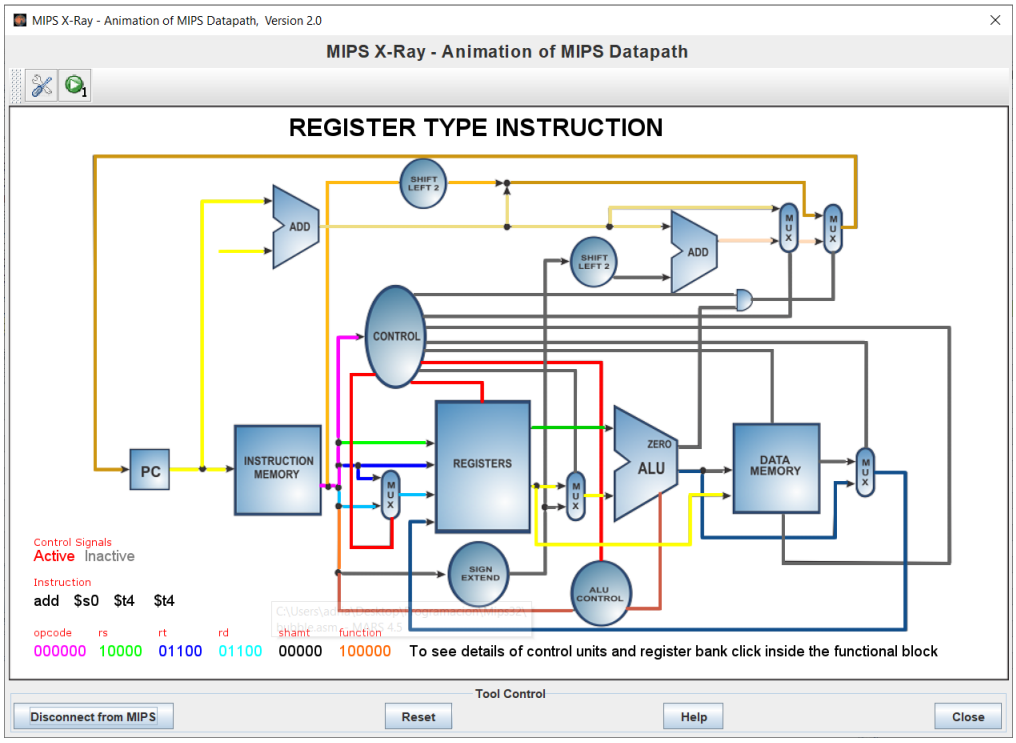

Figura 1: Tipo R: add

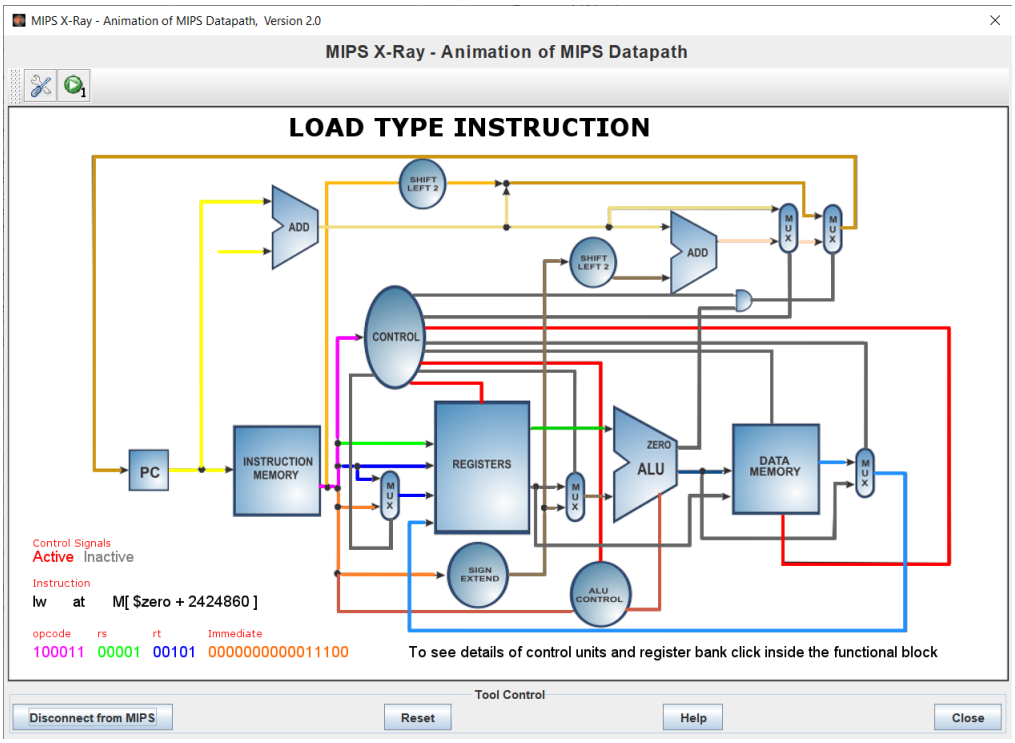

Figura 2: Tipo I: lw

14. Justificación la elección del algoritmo alternativo

la justificación de elegir BubbleSort como algoritmo alternativo (frente a uno más complejo como QuickSort) debe basarse en la simplicidad arquitectónica, el uso de recursos y la facilidad de depuración en un entorno de bajo nivel como MIPS32

- A diferencia de QuickSort, que es recursivo y requiere una gestión constante de la pila para salvar $ra, $a0 y registros temporales, BubbleSort es iterativo.
- Baja Complejidad de Implementación
- Ya estaba implementado como activadaden el curso.
- Es el primer algoritmo de ordenamiento que por lo general se aprende.

15. Análisis y Discusión de los Resultados

15.1. Eficiencia Algorítmica en MIPS32

Durante las pruebas, se observó una diferencia drástica en el número de instrucciones ejecutadas (Instruction Count). Mientras que QuickSort requiere una configuración inicial compleja (manejo de pila y recursión), su estrategia de "divide y vencerásreduce significativamente el número total de comparaciones en comparación con BubbleSort.

15.2. Gestión de Registros y Memoria

La discusión sobre los registros es fundamental. Se validó que:

- El uso de registros temporales ($t0-$t9) en los bucles internos.
- La preservación de registros guardados ($s0-$s7) y de la dirección de retorno ($ra) en la pila permitió que las funciones fueran reentrantes (especialmente en QuickSort), demostrando la importancia de la convención de llamadas de MIPS.

15.3. Impacto de las Estructuras de Controle

Se analizó que los saltos condicionales son el principal cuello de botella en el pipeline.

- En BubbleSort, la alta frecuencia de ble y j provoca rupturas constantes en el flujo secuencial de instrucciones.
- En QuickSort, el flujo es más errático debido a la recursividad, pero al procesar menos elementos, el impacto total en el pipeline es menor que en una ejecución masiva de BubbleSort.

15.4. Observaciones con Herramientas de MARS

El uso de MIPS X-Ray y el Data Segment permitió confirmar que:

- El intercambio (swap) de datos es la operación más costosa a nivel de hardware, ya que involucra las 5 fases del ciclo de instrucción, especialmente la fase MEM.
- El resaltado de ejecución fue la clave para identificar que el algoritmo de QuickSort convergía correctamente hacia el caso base, evitando desbordamientos de memoria.

Resumen

Los resultados demuestran que en la arquitectura MIPS32, la eficiencia no solo depende del algoritmo elegido, sino de cómo este interactúa con la jerarquía de memoria y el banco de registros. Aunque QuickSort presenta una mayor complejidad de implementación en ensamblador, su optimización en el conteo de instrucciones lo hace superior para arreglos de gran escala. Por otro lado, BubbleSort destaca por su predictibilidad y bajo consumo de memoria volátil (pila), siendo una opción viable para sistemas con restricciones estrictas de hardware.