PRÁCTICA DE LABORATORIO 2: INFORME

Jose Contin V29.947.026
Luis Romero V26.729.561

Julio 2025

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

La diferencia principal entre los registros temporales ($t) y los guardados ($s) se resume en una convención sobre quién es responsable de cuidar un valor: el que llama a una función (caller) o la función que es llamada (callee). Los registros **$t** son "volátiles"; si nuestra función principal necesita un valor después de llamar a otra, es nuestra responsabilidad guardarlo en la pila antes. La función llamada puede usar los registros $t como quiera sin preocuparse.

En cambio, los registros **$s** son "seguros". Si una función los usa, está obligada a guardar su valor original al principio y restaurarlo antes de terminar. Esto da la confianza de que los valores en $s se mantendrán intactos.

En nuestra práctica, como implementamos el Bubble Sort dentro de una única función principal sin llamar a subrutinas, no tuvimos que preocuparnos por este problema. Por eso, usamos casi exclusivamente los registros temporales **$t** para todo: contadores de los bucles (i y j), valores del arreglo que estábamos comparando, etc. Fue más sencillo y directo para nuestro caso. Si hubiéramos creado una función 'swap' separada, habríamos tenido que pensar en usar registros $s o guardar los $t en la pila.

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

Estos registros son el kit de herramientas fundamental para que las funciones puedan comunicarse entre sí.

- **$a0–$a3 (Argumentos):** Son como los parámetros de una función en un lenguaje de alto nivel. Los usamos para pasarle la información inicial a nuestra rutina de ordenamiento. Específicamente, pusimos la dirección de memoria donde empezaba nuestro arreglo en **$a0** y el número de elementos del arreglo en **$a1**.
- **$v0–$v1 (Valores de Retorno):** Son para que una función devuelva un resultado. En nuestro caso, la función de ordenamiento no devolvía un valor (modificaba el arreglo directamente), pero usamos **$v0** constantemente para las llamadas al sistema ('syscall'), por ejemplo, para indicarle a MARS que queríamos imprimir un texto (código 4) o terminar el programa (código 10).
- **$ra (Dirección de Retorno):** Este registro es como una miga de pan que nos permite encontrar el camino de vuelta. Cuando usamos 'jal' para saltar a nuestra rutina, MIPS guarda automáticamente en $ra la dirección de la siguiente instrucción. Al final de la rutina, un 'jr $ra' nos devolvía exactamente a donde nos habíamos quedado. Fue crucial para no perder el flujo del programa.

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

La diferencia es abismal. Acceder a un registro es prácticamente instantáneo para la CPU (un ciclo de reloj), mientras que ir a buscar algo a la memoria RAM es como hacer un viaje largo: toma muchísimos ciclos. Abusar de los accesos a memoria es la forma más segura de hacer un programa lento.

En nuestros algoritmos, el arreglo de números tenía que estar en la memoria, no había otra opción. Pero la clave para el rendimiento fue cargar los valores que usábamos repetidamente dentro de los bucles a los registros. Por ejemplo, en vez de comparar dos elementos del arreglo en memoria, los cargábamos con 'lw' a dos registros temporales (ej. $t2 y $t3) y hacíamos la comparación entre ellos. Todas las operaciones (cálculos de dirección, comparaciones) se hicieron dentro de los registros, que es la zona rápida. Solo accedimos a memoria cuando era estrictamente necesario: para leer un elemento ('lw') o para escribirlo de vuelta ('sw') tras un intercambio.

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

Las estructuras de control como los bucles y los 'if' se traducen en MIPS a saltos condicionales ('beq', 'bne'). Estos saltos, aunque esenciales, pueden afectar la eficiencia de un procesador moderno con pipeline. Básicamente, cuando el procesador encuentra un salto, no sabe de inmediato si debe seguir con la siguiente instrucción o saltar a otra parte del código. Tiene que esperar a que se resuelva la condición.

Esta duda crea una "burbuja" o un "atasco" en la cadena de montaje (el pipeline), haciendo que se pierdan ciclos de reloj. Los procesadores intentan adivinar (predicción de saltos), y en bucles como los nuestros suelen acertar (el salto al final del bucle se toma casi siempre). Sin embargo, en el Bubble Sort, que tiene bucles anidados con un 'if' dentro, hay muchísimos saltos. Cada uno de ellos es un pequeño riesgo para el rendimiento, y sumados, hacen que el programa no sea tan rápido como podría ser teóricamente.

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

Nuestro algoritmo alternativo fue el **Selection Sort**. Sobre el papel, ambos, Bubble Sort y Selection Sort, tienen la misma complejidad temporal: $\mathbf{O(n^2)}$. Esto significa que para entradas grandes, ambos se vuelven lentos de forma similar. Sin embargo, a nivel de implementación en MIPS, hay una diferencia muy interesante.

El **Selection Sort** es más "inteligente" con las escrituras en memoria. En cada pasada del bucle principal, busca el elemento más pequeño y hace **un solo intercambio**. Esto significa que, para un arreglo de n elementos, hará como máximo n-1 intercambios (escrituras).

El **Bubble Sort**, en cambio, puede hacer un intercambio en casi cada comparación en el peor de los casos, lo que lleva a un número de escrituras del orden de $O(n^2)$.

Implicación en MIPS: Un intercambio ('swap') no es una sola instrucción; requiere dos cargas ('lw') y dos escrituras ('sw'). Como las escrituras en memoria son lentas, minimizar su número es una gran optimización. Por lo tanto, aunque ambos son $O(n^2)$, esperamos que el **Selection Sort sea ligeramente más rápido en la práctica** en MARS, simplemente porque "molesta" menos a la memoria.

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

El ciclo de ejecución de MIPS es como una cadena de montaje de 5 pasos por la que pasa cada instrucción. Esto permite que varias instrucciones estén procesándose a la vez.

1. **IF (Búsqueda):** La CPU busca la siguiente instrucción en la memoria, usando la dirección que está en el Program Counter (PC).
2. **ID (Decodificación):** La CPU decodifica la instrucción para entender qué operación tiene que hacer (una suma, una carga, etc.) y lee los valores de los registros que necesita.

3. **EX (Ejecución):** La Unidad Aritmético-Lógica (ALU) hace el cálculo. Si es una suma, suma. Si es una instrucción de memoria, calcula la dirección.
4. **MEM (Memoria):** Si la instrucción era una carga (‘lw‘) o un almacenamiento (‘sw‘), en esta fase se accede a la memoria de datos para leer o escribir. Si no, esta fase no hace nada.
5. **WB (Escritura):** El resultado final (ya sea de la ALU o de la memoria) se escribe de vuelta en el registro destino.

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

Al revisar nuestro código, vimos que las instrucciones de **Tipo I** y **Tipo R** fueron las protagonistas.

- **Tipo I (Inmediato):** Las usamos sin parar. Fueron la base para mover datos (‘lw‘, ‘sw‘), para sumar constantes a nuestros contadores (‘addi‘), y sobre todo, para controlar el flujo con los saltos condicionales (‘beq‘, ‘bne‘). Sin ellas, no podríamos haber construido nuestros bucles.
- **Tipo R (Registro):** Estas fueron las que hacían el ”trabajo pesado” de lógica. Usamos ‘slt‘ (Set on Less Than) para comparar los elementos del arreglo y decidir si había que intercambiarlos. También usamos ‘add‘ para calcular las direcciones de memoria a partir de un índice.
- **Tipo J (Salto):** Las usamos mucho menos. Básicamente, ‘j‘ para saltos incondicionales (como para volver al inicio de un bucle) y ‘jal‘ para la llamada inicial a nuestra función.

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?

Abusar de los saltos, especialmente si no siguen una estructura clara (lo que se conoce como ”código espagueti”), es malo tanto para la persona que lee el código como para el procesador. Para el rendimiento, el problema es que cada salto rompe el flujo secuencial del pipeline. El procesador intenta adivinar a dónde irá el programa, pero si se equivoca, tiene que tirar a la basura todo el trabajo que había adelantado y empezar de nuevo desde la dirección correcta. Cada error de predicción es tiempo perdido. Un código más lineal y predecible permite al procesador trabajar de forma más fluida y eficiente.

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

Al principio, el enfoque RISC de MIPS, con sus instrucciones tan simples, puede parecer limitante. Pero para algoritmos como el ordenamiento, rápidamente vimos sus ventajas:

1. **Claridad:** Nos obligó a descomponer el problema en pasos muy pequeños y concretos. No hay una instrucción mágica de 'swap'; tuvimos que construirla nosotros mismos con cargas y almacenamientos. Esto nos dio un control total.

2. **Arquitectura Load-Store:** La regla de que las operaciones matemáticas solo se hacen entre registros nos forzó a seguir una rutina muy eficiente: cargar datos a registros, operar en ellos (que es muy rápido), y solo al final, guardar el resultado en memoria.

3. **Eficiencia predecible:** Como cada instrucción es simple, es más fácil razonar sobre el rendimiento. Sabíamos que las 'lw' y 'sw' eran las instrucciones lentas, así que nuestro objetivo fue minimizarlas.

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

La ejecución paso a paso (F7) fue nuestra herramienta de depuración principal. Sin ella, habría sido casi imposible encontrar los errores. La usamos para:

- **Seguir los contadores:** En cada paso, mirábamos los registros que usábamos como contadores ('$t0$', 't1') para asegurarnos de que los bucles se ejecutaban el número correcto de veces.

- **Verificar la lógica:** Nos deteníamos justo antes de una comparación ('slt') y un salto ('beq'). Mirábamos los valores en los registros y predecíamos si el salto debía ocurrir. Luego, dábamos un paso más para confirmar si nuestra lógica era correcta. Así encontramos varios errores en las condiciones.

- **Observar los intercambios:** Durante un 'swap', íbamos paso a paso viendo cómo un valor se cargaba de la memoria a un registro, y luego se escribía en otra posición. Mirábamos tanto los registros como la ventana de datos de MARS para confirmar que el intercambio se hacía bien.

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

Sin lugar a dudas, la ventana de **"Registers"**. La característica que la hace tan poderosa es que **resalta en verde el registro que acaba de cambiar**. Esto es un salvavidas. Cuando ejecutábamos una instrucción, mirábamos inmediatamente qué registro se ponía verde. Si no era el que esperábamos, sabíamos al instante que algo estaba mal en esa línea de código. Nos permitió cazar errores tontos, como escribir en '$t1$'*cuandoqueríamosescribiren*'t2', de forma muy visual y rápida.

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

MARS tiene una herramienta increíble para esto en **Tools ¿ Data Path Visualization**. Después de conectarla, seleccionamos una instrucción 'add' en nuestro código y le dimos a ejecutar paso a paso en el visualizador. Fue genial ver la teoría en acción: se iluminaba el camino que seguían los datos desde dos registros fuente, pasando por la ALU donde se realizaba la suma, y el resultado volvía para escribirse en el registro destino. Hizo que el diagrama abstracto del libro de texto cobrara vida.

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

Usando la misma herramienta, **Data Path Visualization**, pero seleccionando una instrucción 'lw'. El camino que se iluminaba era diferente y muy instructivo. Se veía cómo el valor de un registro (la dirección base del arreglo) y el valor inmediato (el desplazamiento) entraban en la ALU para calcular la dirección final. Luego, esa dirección se enviaba a la Memoria de Datos, que se iluminaba mientras "buscaba" el valor. Finalmente, el dato viajaba desde la memoria de vuelta al banco de registros para ser guardado. Visualizar esto nos ayudó a entender por qué las instrucciones de memoria son más complejas y lentas.

14 Justificar la elección del algoritmo alternativo

Elegimos el **Selection Sort** como alternativa al Bubble Sort porque, aunque tienen la misma complejidad teórica $O(n^2)$, su estrategia es muy diferente y nos permitía explorar un matiz importante del rendimiento. La idea era comparar dos algoritmos "igual de lentos" en teoría, pero que se comportan de forma distinta a bajo nivel. Como mencionamos, el Selection Sort minimiza las escrituras en memoria. Queríamos ver si esta diferencia, que es invisible en el análisis de complejidad, se podía razonar a nivel de instrucciones MIPS, enfocándonos en el costo de las operaciones 'sw'. Fue una elección didáctica para profundizar en cómo la implementación real importa.

15 Análisis y Discusión de los Resultados

Esta práctica fue una experiencia fundamental para entender cómo "piensa" realmente una computadora. Traducir un algoritmo que parece simple en un lenguaje de alto nivel, como Bubble Sort, a ensamblador MIPS fue un reto que nos obligó a gestionar manualmente cada detalle: cada contador, cada dirección de memoria, cada salto. El principal desafío fue, sin duda, la implementación correcta de los bucles anidados y las condiciones, donde un pequeño error en un 'beq' o en un cálculo de dirección podía llevar a bucles infinitos o a corromper datos.

El simulador MARS fue nuestro mejor aliado. La ejecución paso a paso y la visualización de registros nos permitieron "ver" el algoritmo en acción y cazar errores que habrían sido imposibles de encontrar de otra manera. La discusión sobre Bubble Sort vs. Selection Sort nos llevó a una conclusión importante: la eficiencia no solo depende de la complejidad $O(n)$, sino también del tipo y

número de operaciones que se realizan, especialmente las que involucran accesos a la lenta memoria principal.

Al final, no solo logramos que los algoritmos funcionaran, sino que ganamos una intuición mucho más profunda sobre la arquitectura de computadores, el funcionamiento del pipeline y la importancia de programar pensando en el hardware subyacente.