# Arquitectura del Computador
# Práctica de Laboratorio 2

Angel Navas y Jose Natera
*Departamento de Computación*
*Universidad de Carabobo*
Barbula, Carabobo

***Abstract*—Familiarizar al estudiante con la sintaxis del lenguaje ensamblador *MIPS32* y su ejecución en el entorno *MARS*, resolviendo problemas clásicos de programación.**

***Index Terms*—Computer Architecture, Assembly Language, Recursion, Stack management, Work Environment, Sorting Algoritmhs, Complexity Analysis**

## I. Informe

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

La arquitectura *MIPS* define que los registros temporales, identificados como `$t0` a `$t9`, son responsabilidad del llamador . Si una función está usando un valor en `$t0` y decide llamar a otra subrutina, debe asumir que esa subrutina podría sobrescribir el valor de `$t0` para sus propios cálculos. Por lo tanto, si el programador necesita ese dato después de la llamada, debe guardarlo manualmente en la pila (stack) antes de saltar.

Por el contrario, los registros guardados, denominados `$s0` a `$s7`, son responsabilidad del llamado . Existe una promesa implícita: cuando una función termina y devuelve el control, los registros s deben tener exactamente los mismos valores que tenían antes de que la función empezara. Si la función necesita usar `$s0` para trabajar, primero debe copiar el valor original en la pila y, justo antes de finalizar, restaurarlo.

a) *Aplicación en la práctica:*

En el desarrollo real, esta distinción se aplica para optimizar el rendimiento y minimizar el acceso a la memoria *RAM*, que es mucho más lenta que los registros. Cuando un programador escribe una función pequeña que no llama a ninguna otra, utiliza preferiblemente los registros temporales. Como no hay llamadas salientes, no hay riesgo de que nadie le cambie sus datos, y se ahorra el trabajo de escribir y leer en la pila, lo que acelera la ejecución. En cambio, si se está programando un bucle principal o una función que coordina muchas otras llamadas, se utilizan los registros no temporales. Esto permite mantener variables importantes protegidas a lo largo de múltiples saltos a otras funciones, delegando la responsabilidad de «limpiar» y «restaurar» a las funciones secundarias solo si estas deciden usar esos registros específicos.

### *B. ¿Qué diferencias existen entre los registros `$v1`, `$ra` y cómo se aplicó esta distinción en la práctica?*

`$a0`–`$a3`, `$v0`– Los registros `$a0`-`$a3` son la vía de entrada. Se utilizan para pasar parámetros a una función. Si una subrutina requiere, por ejemplo, sumar dos números, estos valores se colocan en `$a0` y `$a1` antes de realizar el salto. Esto estandariza la comunicación: cualquier función sabe exactamente dónde buscar los datos que necesita para operar sin tener que rastrear la memoria. Los registros `$v0` y `$v1` son la vía de salida. Una vez que la función termina su tarea, deposita el resultado en `$v0` (y `$v1` si el dato es de 64 bits). El programa principal sabe que, inmediatamente después de que una función termina, el resultado de esa operación estará disponible en estos registros específicos. El registro `$ra` es quizás el más crítico para la supervivencia del flujo del programa. Cuando se ejecuta una instrucción de salto a una función como `jal`, el procesador guarda automáticamente la dirección de la siguiente instrucción en `$ra`. Esto funciona como un rastro que le permite a la función saber a qué punto exacto del código debe volver cuando termine de ejecutarse. En la práctica, esta estructura permite la creación de código modular y bibliotecas. Al existir un acuerdo sobre el uso de estos registros, un programador puede usar una función escrita por otra persona sin necesidad de leer su código fuente; solo necesita saber qué poner en los registros a y qué esperar en los registros v. Un desafío común ocurre en las funciones anidadas. Como solo existe un registro `$ra`, si la Función A llama a la Función B, el valor de retorno original hacia el programa principal se perdería al ser sobrescrito por la dirección de retorno de B. Para resolver esto, la práctica estándar dicta que debe guardar el contenido de `$ra` en la pila al iniciar y restaurarlo antes de salir. De este modo, se pueden encadenar cientos de llamadas a funciones manteniendo siempre el camino de regreso intacto.

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

La diferencia de rendimiento entre usar registros o memoria es un factor determinante entre un algoritmo eficiente y uno lento. Esto se debe a que el acceso a un registro ocurre en un solo ciclo de reloj, mientras que el acceso a la memoria (`lw` y `sw`) puede tardar más instrucciones.

a) *El impacto en Quicksort:*

Quicksort es un algoritmo recursivo que se basa en la partición de elementos alrededor de un pivote. En la práctica, el rendimiento de Quicksort en *MIPS* depende de cómo se gestionan los registros durante las llamadas recursivas. Dado que Quicksort realiza múltiples llamadas a sí mismo, el programador debe decidir qué valores mantener en los registros guardados (`$s0`–`$s7`). Si se guardan los límites del sub-arreglo (izquierda y derecha) en registros `$s`, se evitan lecturas constantes a la pila en cada iteración del bucle de partición. Sin embargo, debido a la naturaleza recursiva, cada nivel de la recursión debe salvar esos registros `$s` y el registro `$ra` en la pila. El costo de Quicksort es, por tanto, un equilibrio: se gana velocidad usando registros para las comparaciones rápidas del pivote, pero se paga un precio en accesos a memoria cada vez que se profundiza en la recursión.

b) *El impacto en Heapsort:*

Heapsort, en este caso comparte con Quicksort el hecho de tener que guardar parte de los registros en la pila, como consecuencia de tener una implementacion recursiva del procedimiento *Heapify* (aunque este ultimo se puede cambiar a una versión iterativa), sin embargo, gracias a la naturaleza de la estructura *Heap* que le da su nombre, su complejidad espacial es de orden $O(\log n)$ en el peor caso, esto viene del hecho de que el procedimiento *Heapify* se llama a si mismo hasta alcanzar el fondo del *Heap*, que, dado un arreglo de $n$ elementos, tendra una profundidad de $\log_2 n$, lo que sera proporcional al numero de llamadas recursivas y por propiedades del analisis asintotico, se reduce a $\log n$. A diferencia de Quicksort que en el peor caso, tendra que hacer tantas llamadas como elementos tenga el vector, lo que se reduce a una complejidad espacial de orden $O(n)$

c) *Comparativa técnica de rendimiento:*

La principal diferencia en la práctica se resume en estos puntos:

- Localidad de datos: Quicksort tiene una mejor localidad de referencia, lo

que significa que los datos que carga de memoria a los registros suelen estar cerca unos de otros, aprovechando el cache del procesador *MIPS*.

- Gestión de la Pila: Quicksort tiene un uso más intensivo de la memoria

para gestionar la pila de recursión , mientras que Heapsort es más plano y depende más de mover datos entre el arreglo y los registros temporales.

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

a) *El costo de los saltos y el «Branch Delay Slot»:*

*MIPS* utiliza una técnica llamada segmentación (pipelining), donde el procesador procesa varias instrucciones a la vez en diferentes etapas. Cuando el código encuentra una instrucción de salto (como `beq`, `bne` o j), el procesador no sabe con certeza cuál es la siguiente instrucción hasta que el salto se evalúa. Históricamente, *MIPS* implementó el Branch Delay Slot: la instrucción que sigue inmediatamente a un salto siempre se ejecuta, independientemente de si el salto se toma o no. Si un programador no llena ese espacio con una instrucción útil, debe colocar un nop , lo que desperdicia un ciclo de reloj y reduce la eficiencia del algoritmo.

b) *Bucles anidados y la latencia de memoria:*

En algoritmos de ordenamiento o procesamiento de matrices, los bucles anidados multiplican el impacto de las instrucciones de control. Por cada iteración del bucle interno, se ejecutan varias instrucciones de salto y comparación.

- Sobrecarga de control: En un bucle anidado, las instrucciones que

gestionan el índice (incrementar el contador, comparar con el límite y saltar al inicio) pueden representar hasta el 30% o 40% del tiempo total de ejecución.

- Predicción de saltos: Los procesadores *MIPS* más avanzados intentan

predecir si un salto se tomará. En un bucle largo, la predicción es buena, pero en estructuras de control complejas (como los «if» dentro de un Quicksort), los errores de predicción vacían el pipeline, obligando al procesador a detenerse y reiniciar la carga de instrucciones.

### *E. ¿Cuáles son las diferencias de complejidad computacional entre el algoritmo Quicksort y el algoritmo alternativo? ¿Qué implicaciones tiene esto para la implementación en un entorno MIPS32?*

Ambos algoritmos son excelentes en la computación, pero se comportan de forma distinta en casos extremos:

- Heapsort: Es el modelo de la estabilidad. Su complejidad es siempre $O(n \log n)$ en el mejor, promedio y peor de los casos.
- Quicksort: Es mas rápido pero con un punto débil. En promedio es $O(n \log n)$ y suele ser más rápido que Heapsort en la práctica, pero tiene un

peor caso $O(n^2)$ si el pivote se elige mal.

a) *Implicaciones en el entorno MIPS32:*

Al implementar estos algoritmos en ensamblador *MIPS*, la complejidad se traduce en dos puntos principales:

1) El Costo de la Partición vs. El Costo del Heap

El bucle interno de Quicksort es extremadamente ligero: son comparaciones rápidas usando registros temporales y saltos condicionales.

En cambio, Heapsort requiere calcular constantemente las posiciones de los hijos.

Aunque en *MIPS* esto se optimiza con instrucciones de desplazamiento lógico a la izquierda (sll), sigue siendo una carga aritmética mayor que la simple partición de Quicksort.

2) Cache y Jerarquía de Memoria

Quicksort recorre el arreglo de forma lineal durante la partición, lo que mantiene los datos en la memoria cache del procesador. Heapsort, al saltar entre padres e hijos en el árbol, provoca constantes «fallos de cache», obligando al procesador a esperar a la memoria *RAM* lenta.

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

1) Búsqueda de Instrucción (IF - Instruction Fetch)

En esta fase, el procesador utiliza el Program Counter (PC), que es el registro que contiene la dirección de memoria de la próxima instrucción. Se extrae el código binario de la memoria de instrucciones y se coloca en el registro de instrucción. Al mismo tiempo, el PC se incrementa en 4 para apuntar a la siguiente línea de código.

2) Decodificación y Lectura de Registros (ID - Instruction Decode)

Aquí el procesador lee la instrucción para saber qué debe hacer . Durante esta etapa, se accede al banco de registros para obtener los valores de los operandos fuente. Por ejemplo, si la instrucción es una suma de `$t1` y `$t2`, los valores almacenados en esos registros se preparan para el siguiente paso.

3) Ejecución o Cálculo de Dirección (EX - Execute)

Es el momento donde interviene la ALU (Unidad Aritmético-Lógica). Dependiendo de la instrucción, la ALU realiza una operación matemática o lógica . En el caso de instrucciones de memoria como `lw` o `sw`, la ALU calcula la dirección de memoria efectiva sumando un desplazamiento al registro base.

4) Acceso a Memoria (MEM - Memory Access)

Esta fase solo es activa para instrucciones que interactúan con la memoria *RAM*.

- Si es un Load, el procesador lee un dato de la memoria.
- Si es un Store, el procesador escribe un dato en la memoria. Si la

instrucción es puramente aritmética (como una suma entre registros), esta etapa no realiza ninguna acción relevante, pero la instrucción debe pasar por ella para mantener la sincronía del pipeline.

5) Escritura en Registro (WB - Write Back)

La etapa final consiste en escribir el resultado obtenido de vuelta en el banco de registros. Si fue una operación aritmética, el resultado de la ALU se guarda en el registro destino (por ejemplo, `$t0`). Si fue una carga de memoria, el valor traído de la *RAM* se deposita en el registro correspondiente. Una vez completado este paso, la instrucción se considera terminada.

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

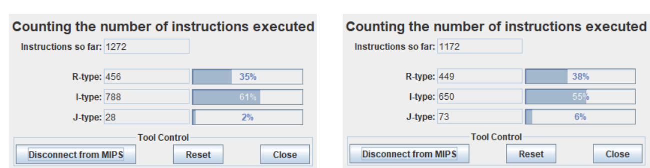
a) *Quicksort:*

Las instrucciones de tipo I (Immediate) son las más numerosas, representando el 55% del código total. Este grupo incluye operaciones de carga de memoria como `lw`, almacenamiento con `sw`, saltos condicionales como `bge` y `bgt`, y la carga de valores inmediatos mediante li y la pseudoinstrucción la. Las instrucciones de tipo R equivalentes al 38% del programa, abarcando operaciones aritméticas entre registros como mul, addu, move y el salto de retorno jr. Finalmente, las instrucciones de tipo J son las menos frecuentes, con solo 9 apariciones que representan el 6% del total, limitándose a saltos incondicionales de tipo j y llamadas a subrutinas mediante `jal`. El predominio de las instrucciones de tipo I sobre las demás se debe principalmente a la naturaleza de la tarea realizada, la cual implica un acceso constante a la memoria *RAM* para manipular el vector y gestionar la pila de recursión. Cada vez que el algoritmo realiza una comparación o un intercambio, debe emplear instrucciones `lw` y `sw`, las cuales pertenecen al formato I. Además, el flujo de control del Quicksort depende de múltiples bifurcaciones condicionales para determinar los límites de las particiones, lo que eleva significativamente el conteo de este tipo de instrucción en comparación con los saltos directos o las operaciones puramente matemáticas entre registros.

b) *Heapsort:*

Igual que en la anterior el tipo de instrucción que predomina es el Tipo I que representan el 61% del código total, este grupo abarca las operaciones de carga y almacenamiento en memoria como `lw` y `sw`, las comparaciones condicionales como `bge`, `ble`, `bgez`, `bgtz` y `beq`, y la manipulación de valores inmediatos, seguidas por las de Tipo R con un 35% y finalmente las de Tipo J con un 2% . La razón por la cual el Tipo I predomina de manera tan marcada sobre las demás radica en la naturaleza recursiva y de gestión de memoria del algoritmo Heapsort. Cada vez que se llama a las funciones heapsort o heapify, el programa debe realizar múltiples operaciones de guardado y recuperación en la pila para preservar los registros de retorno y los argumentos, lo que dispara el uso de `lw` y `sw`. Además, la lógica del algoritmo requiere constantes verificaciones de límites del arreglo y comparaciones entre los nodos padres e hijos para mantener la propiedad del *Max-Heap*, lo cual se traduce en una alta densidad de instrucciones de salto condicional, todas pertenecientes al formato de tipo inmediato. Finalmente, el uso de las instrucciones Tipo R queda relegado principalmente a los cálculos aritméticos necesarios para hallar los índices de los hijos o las direcciones de memoria específicas, mientras que las instrucciones Tipo J solo se activan en los puntos de transferencia de control hacia las subrutinas. El flujo de control complejo, sumado a la interacción constante con la memoria *RAM* para el intercambio de elementos del arreglo, consolida a las instrucciones de Tipo I como el pilar estructural de este programa en lenguaje ensamblador.

TABLE I
NUMERO DE INSTRUCCIONES DE LA IMPLEMENTACION DE HEAPSORT EN CONTRASTE A LA DE QUICKSORT, CONTABILIZANDO LAS INSTRUCCIONES PARA MOSTRAR EL VECTOR




### H. *¿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 las instrucciones de salto en MIPS32 impacta negativamente en el rendimiento debido a la ruptura de la secuencia natural del procesador. Cuando el código es lineal, el hardware puede predecir con total certeza qué instrucción sigue a la anterior, manteniendo el flujo de trabajo al máximo. Al introducir saltos constantes como j , `beq` o `bne`, se presentan tres problemas técnicos principales: Vaciado del Segmentado (Pipeline Stall) El procesador MIPS32 intenta ejecutar cinco instrucciones a la vez en diferentes etapas. Cuando aparece un salto condicional, el procesador no sabe si debe seguir la secuencia o saltar a otra dirección hasta que la instrucción llega a la etapa de ejecución (EX). Si la predicción falla, el procesador debe limpiar las instrucciones que ya habían entrado al túnel de ejecución y empezar de cero en la nueva dirección. Esto genera burbujas o ciclos muertos donde el procesador no hace nada útil. Desaprovechamiento del Branch Delay Slot Como mencionamos antes, *MIPS* ejecuta la instrucción que sigue inmediatamente a un salto pase lo que pase. Si el código está saturado de saltos desordenados, al programador le resulta muy difícil colocar instrucciones útiles en esos huecos. El resultado es un código lleno de instrucciones nop, que ocupan espacio en la memoria y consumen tiempo de ciclo sin procesar datos reales. Fragmentación de la Memoria Cache Las instrucciones se cargan desde la memoria *RAM* a una memoria cache ultra rápida en bloques. Las estructuras lineales permiten que el procesador traiga un bloque de código y lo ejecute por completo. El uso excesivo de saltos obliga al procesador a saltar a direcciones de memoria lejanas que probablemente no están en la cache. Esto provoca un fallo de cache (cache miss), deteniendo al procesador durante cientos de ciclos mientras se busca la nueva instrucción en la *RAM* lenta. En la práctica, un algoritmo con muchos saltos puede tardar el doble que una versión desenrollada o lineal, aunque ambos realicen la misma cantidad de operaciones matemáticas.

### I. *¿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 ventajas estructurales que impactan directamente en la velocidad de ejecución de algoritmos de ordenamiento. A diferencia de las arquitecturas CISC, donde una sola instrucción puede realizar múltiples tareas complejas, *MIPS* se basa en la simplicidad y la predictibilidad. La principal ventaja de RISC es que casi todas las instrucciones (sumas, comparaciones, movimientos) tienen un tamaño fijo de 32 bits y se ejecutan en un tiempo uniforme. En algoritmos como Quicksort, donde el bucle interno realiza millones de comparaciones, esta uniformidad permite que el procesador mantenga un ritmo constante. El programador puede calcular con precisión cuántos ciclos de reloj tomará cada iteración, facilitando la optimización fina del código. Las instrucciones de cálculo solo operan sobre registros. En la práctica de un Heapsort, esto obliga a cargar los elementos del arreglo en registros antes de compararlos. Aunque parezca un paso extra, esto reduce la complejidad del hardware y permite que el procesador alcance frecuencias de reloj mucho más altas. Al trabajar solo con registros, se elimina la latencia de esperar a la memoria *RAM* en medio de una operación matemática crítica. *MIPS* ofrece 32 registros, una cifra elevada comparada con arquitecturas antiguas. Para un algoritmo de ordenamiento, esto es una mina de oro. Permite mantener en «memoria ultrarrápida» no solo los datos que se están comparando, sino también los índices del bucle, los límites del arreglo (izquierda y derecha en Quicksort) y el valor del pivote. Al minimizar las veces que el procesador debe «ir y venir» a la memoria *RAM*, el rendimiento aumenta exponencialmente. Debido a que las instrucciones RISC son simples y uniformes, el procesador puede aplicar la técnica de segmentación de forma muy eficiente. Mientras una instrucción está guardando un resultado de una comparación anterior en un registro, la siguiente ya está comparando otros dos valores y una tercera está siendo buscada en la memoria. Esta línea de montaje permite que, en condiciones ideales, el procesador complete una instrucción por cada ciclo de reloj, algo fundamental para procesar grandes volúmenes de datos en ordenamientos masivos. Para un compilador, la sencillez de RISC facilita la reordenación de instrucciones. El compilador puede identificar huecos (como los Branch Delay Slots tras un salto) y rellenarlos con operaciones útiles del algoritmo. En algoritmos lineales o con bucles claros, esta capacidad de optimización mecánica permite extraer el máximo provecho del hardware sin necesidad de lógica compleja en el silicio.

### J. *¿Cómo se usó el modo de ejecución paso a paso (Step, Step Into) en* MARS

Para verificar la correcta ejecución del algoritmo? La función Step permite ejecutar una sola instrucción a la vez. En la práctica, esto se utiliza para vigilar cómo cambian los registros temporales t y guardados tras cada línea de código. Al depurar un intercambio en un algoritmo de ordenamiento, el programador usa Step para confirmar que los valores de dos posiciones del arreglo realmente se movieron a los registros y luego regresaron a la memoria en las posiciones invertidas. *MARS* resalta en color los registros que han cambiado su valor en el último paso, facilitando la detección de errores lógicos inmediatos.

a) *Uso de Step Into en subrutinas:*

La distinción con Step Into es crítica cuando el algoritmo está modularizado. Usar Step Into permite «entrar» en esa subrutina y seguir su ejecución línea por línea. Esto es vital para verificar el contrato de registros: asegurar que el registro de dirección de retorno `$ra` se guardó correctamente en la pila y que los argumentos `$a0`-`$a3` contienen los parámetros esperados antes de empezar la lógica interna de la función.

b) *Verificación de la Memoria y el Stack:*

Mientras se avanza paso a paso, la ventana «Data Segment» de *MARS* permite visualizar el arreglo en tiempo real. Para algoritmos de ordenamiento, esto permite ver cómo los elementos se desplazan físicamente por la memoria *RAM* simulada. Además, al ejecutar paso a paso las instrucciones de gestión de pila , se puede verificar que el puntero de pila se mueve correctamente y que no hay solapamientos de datos que puedan corromper el flujo del programa.

c) *Identificación de Branch Delay Slots y Saltos:*

El modo paso a paso es la única forma efectiva de entender el impacto de los saltos. En *MARS*, al ejecutar una instrucción de salto como `beq`, el programador puede observar si la siguiente instrucción (la del delay slot) se ejecuta antes de que el Program Counter (PC) cambie a la dirección de destino. Esto permite validar si se ha colocado un nop necesario o si una instrucción útil está siendo procesada correctamente a pesar del salto.

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

En conjunto con las herramientas de instrucción paso a paso (*Step*, *Step Into*, etc), los dos elementos mas utiles de *MARS* para visualizar el contenido de los registros durante la ejecución del código asi como la detección de errores lógicos (*debugging*), fueron el *Text Segment* y la tabla de registros, por una parte, el *Text Segment* permite ver cual instrucción se va a realizar durante la ejecución del programa, en compañia con esto, la tabla de registros permite ver el contenido de todos y cada uno de los registros de *MIPS32*, permitiendo entender que datos esta manejando el programa, y en conjunto con el *Text Segment*, el como los esta manejando.

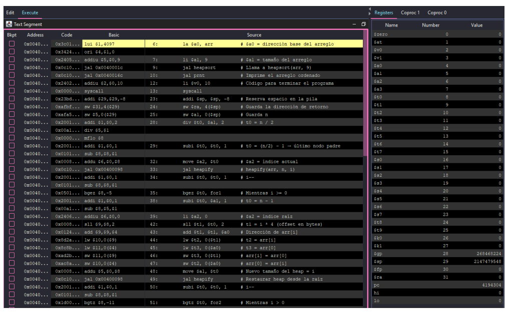


Fig. 1. Captura del Text Segment y la tabla de registros durante la ejecución del algoritmo Heapsort

Es valido además, mencionar las herramientas de *Instruction Counter* y *Instruction Statistics* bajo el apartado de *Tools* (herramientas) que trae *MARS* para ver la cantidad de instrucciónes segun el tipo en cada implementación.

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

*MARS* trae un conjunto de herramientas para el analisis y *debugging* del codigo *MIPS32*, orientados principalmente a que el estudiante pueda entender todas las capas del como funciona un lenguaje ensamblador y la arquitectura de un ordenador, teniendo eso en cuenta, para poder visualizar el camino de datos (*Datapath*) de una instrucción, se puede combinar la herramienta *MIPS X-Ray* bajo el apartado de *Tools* (herramientas) que trae el editor *MARS*, esta herramienta permite ver el flujo de datos de la instrucción que se este ejecutando en ese momento, por lo que, para poder ver el camino de datos de una instrucción de tipo R, necesitamos hacer uso de la herramienta cuando se este ejecutando una instrucción de tipo R, la ventana, ademas, nos indicara el tipo de instrucción que se esta ejecutando en ese momento.

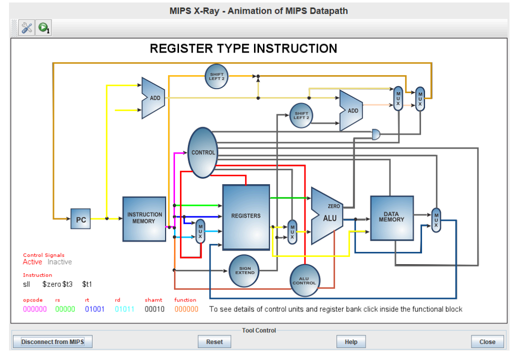



Fig. 2. Captura del camino de datos durante la ejecución de una instrucción de tipo R

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

Similarmente al como se hace con las instrucciones de tipo R, utilizamos la herramienta *MIPS X-Ray* durante la ejecución de una instrucción de tipo I para visualizar su camino de datos.

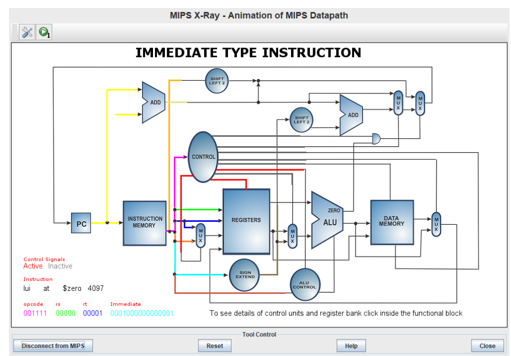



Fig. 3. Captura del camino de datos al final de la ejecución de una instrucción de tipo I

### N. *Justificar la elección del algoritmo alternativo*

La elección de Heapsort con respecto a Quicksort viene dada en función de la situación en la que nos encontremos. Si bien, ambos algoritmos son *in-place* y tienen una complejidad espacial $O(\log n)$ Heapsort tiene una complejidad algoritmica de $O(n \log n)$ en el peor caso, en contraste con el $O(n^2)$ de Quicksort, lo que si bien a priori parece mejor, debido a la simplificación de constantes, en la practica Heapsort solo resulta superior en el ordenamiento de arreglos de un tamaño muy grande, mientras que en promedio, Quicksort terminara siendo mas rapido.

### O. *Análisis y Discusión de los Resultados*

Tras la implementación de los algoritmos Heapsort y Quicksort en el simulador *MARS*, se observa cómo las decisiones de diseño a nivel de ensamblador impactan directamente en el uso de los recursos del procesador MIPS32. El código de Quicksort utiliza macros como SavePila y LoadPila para gestionar la recursión. En cada partición, el algoritmo debe salvar el registro de retorno `$ra` y los límites del sub--arreglo. Si bien Quicksort es generalmente más rápido por su baja carga aritmética, su dependencia de la pila lo hace vulnerable a un mayor consumo de memoria si la profundidad de la recursión aumenta, especialmente al manejar el pivote en el extremo del vector. Por otro lado, el código de Heapsort muestra una estructura de «hundimiento» (heapify) que también es recursiva. Sin embargo, su fase de construcción (for1) e intercambio (for2) es predominantemente iterativa. La gran diferencia radica en que Heapsort protege una mayor cantidad de registros temporales al inicio de heapify, lo que garantiza que las operaciones de cálculo de índices hijos no corrompan los datos de las llamadas superiores. En Heapsort, el uso de mul y sll para calcular los desplazamientos de memoria refleja la carga computacional de navegar por un árbol binario implícito en un arreglo. El algoritmo realiza constantes accesos a memoria para comparar padres e hijos. Quicksort, mediante el uso de macros, simplifica la sintaxis, pero oculta un flujo de datos muy lineal. El bucle while de partición es extremadamente eficiente en *MIPS* porque utiliza comparaciones simples (`bgt`) y mantiene el pivote en un registro fijo durante todo el proceso de comparación, minimizando las lecturas a memoria.

a) *Conclusion final:*

- Quicksort es más ágil para arreglos de tamaño medio gracias a su bucle

de partición simplificado y al uso intensivo de registros temporales para el pivote.

- Heapsort ofrece una robustez superior en cuanto a la gestión de memoria

de la pila, ya que su estructura de control es más predecible y el crecimiento de su pila es logarítmico y controlado.