Estructura de computadores

Práctica 3

Laboratorio: Simulaciones en el WinDLXV

Carlos Gálvez Reguera

INDICE

INTRODUCCIÓN

La presente práctica tiene como objetivo comprender de manera aplicada el funcionamiento de los procesadores segmentados, a través de la simulación de programas en las herramientas WinDLX y DLXV. A lo largo de los distintos ejercicios se analiza cómo afectan las dependencias entre instrucciones al rendimiento del pipeline, así como las distintas estrategias que se pueden aplicar para mitigarlas.

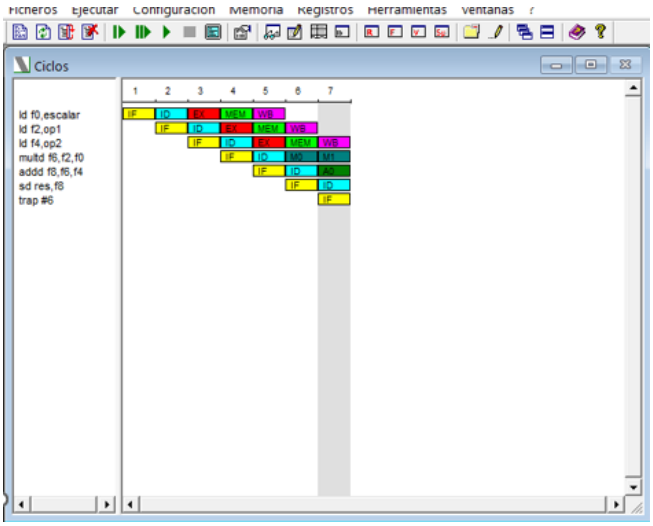
En la primera parte del laboratorio se parte de un programa sencillo que permite observar claramente el impacto de los conflictos tipo RAW (Read After Write) en el cauce de ejecución. A partir de ahí, se experimenta con diferentes técnicas de optimización —como la inserción de instrucciones de espera (nop), la reorganización del código y el desenrollado de bucles— para mejorar el aprovechamiento del procesador.

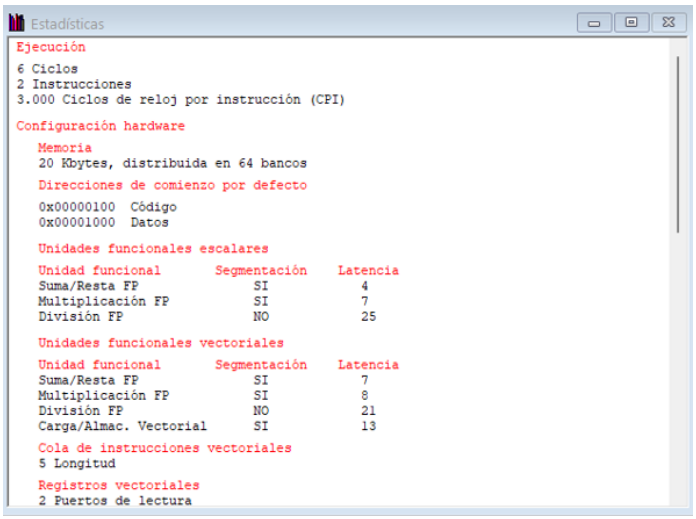
La segunda parte se centra en la arquitectura vectorial del procesador DLXV. En este caso, se comparan tres modos de ejecución: sin mejoras, con encadenamiento vectorial y con adelanto de resultados. Aunque el programa utilizado es breve, las diferencias en la estructura del pipeline permiten apreciar cómo cada técnica contribuye a una ejecución más eficiente.

Más allá de los resultados numéricos, esta práctica sirve para afianzar los conceptos clave de la segmentación y entender cómo pequeñas decisiones a nivel de código pueden tener un impacto considerable en el rendimiento, sobre todo en sistemas reales donde los volúmenes de datos y las instrucciones se multiplican.

.

1. Primera parte del laboratorio



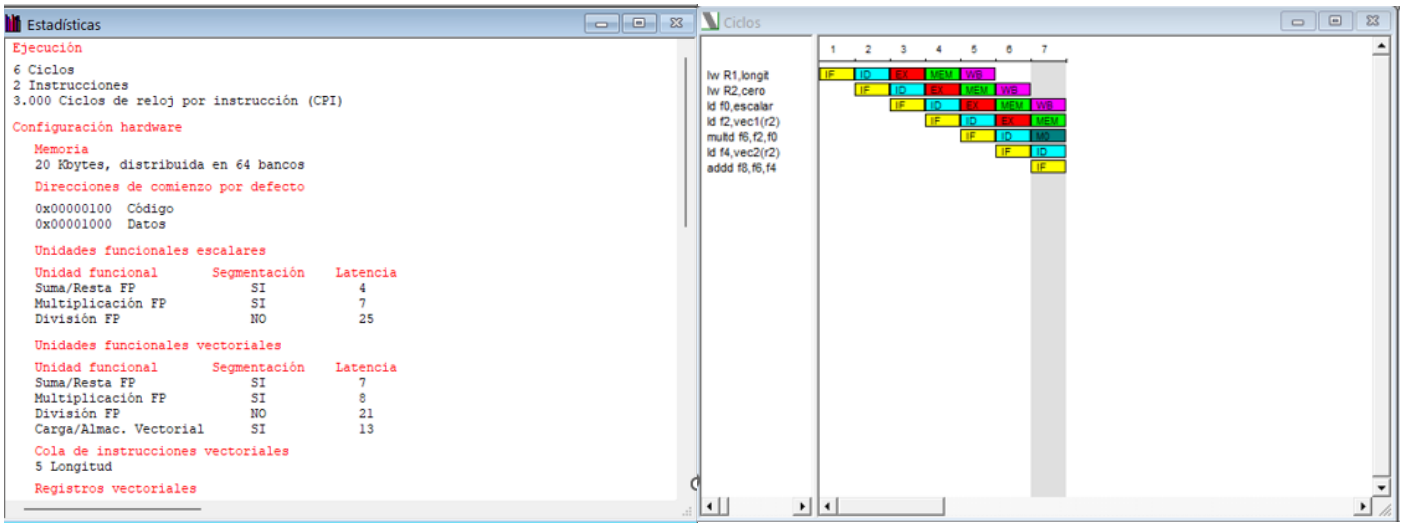


En esta primera simulación se ha trabajado con un programa que realiza una suma acumulativa de datos almacenados en memoria, utilizando un bucle. A pesar de su simplicidad, este ejemplo resulta especialmente útil para analizar el comportamiento del pipeline en la arquitectura DLX, ya que permite identificar con claridad los efectos de las dependencias entre instrucciones.

Al observar el cronograma de ejecución, se aprecia un cierto solapamiento entre instrucciones, pero también aparece una pausa significativa justo después de la instrucción lw. Esta interrupción se produce por una dependencia de tipo RAW (Read After Write): el valor que se carga desde memoria es utilizado de inmediato en una instrucción add, lo que obliga al procesador a esperar a que el dato esté disponible antes de continuar. El resultado es una burbuja en el pipeline que reduce la eficiencia de ejecución.

Las estadísticas obtenidas lo dejan claro: se han necesitado 6 ciclos para completar solo 2 instrucciones útiles, con un CPI de 3. Aunque el ejemplo es muy breve, pone de manifiesto cómo incluso en programas sencillos este tipo de conflictos puede tener un impacto notable en el rendimiento.

En los siguientes ejercicios se abordarán distintas estrategias para mitigar estos parones, como la reordenación de instrucciones o el desenrollado del bucle, con el objetivo de optimizar la ejecución y aprovechar mejor el funcionamiento del procesador segmentado.



En esta segunda versión del programa se mantiene la misma operación de suma acumulativa, pero se introduce una diferencia clave: la inserción de una instrucción nop justo después de la carga de memoria con lw. Esta modificación tiene un propósito claro: dar margen al procesador para que el dato recién cargado esté disponible antes de ser utilizado por la siguiente instrucción, evitando así el conflicto de tipo RAW observado anteriormente.

El cronograma refleja una mejora evidente en la fluidez del pipeline. Gracias a la instrucción de espera, el flujo entre etapas es más ordenado y no se produce la burbuja que interrumpía la ejecución en la versión anterior. El procesador puede avanzar de forma más estable, sin necesidad de detenerse por dependencias inmediatas.

A pesar de ello, las estadísticas se mantienen prácticamente iguales: se siguen necesitando 6 ciclos para ejecutar 2 instrucciones útiles, con un CPI de 3. La mejora, por tanto, no se traduce en una diferencia numérica significativa en este ejemplo concreto, pero sí en una ejecución más limpia y predecible.

Este tipo de ajustes cobra mayor importancia en programas más extensos, donde los conflictos entre instrucciones son más frecuentes. En ese contexto, introducir pausas controladas o aplicar una reordenación más eficiente del código puede suponer una mejora real en el rendimiento general del sistema.

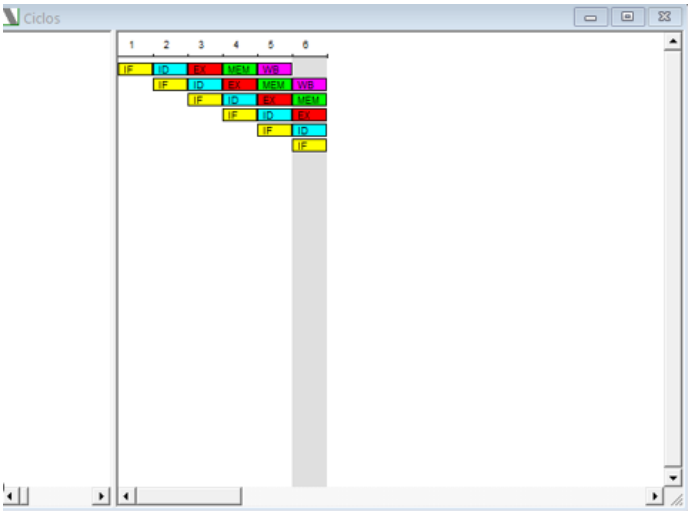
1.1. Código Mejorado

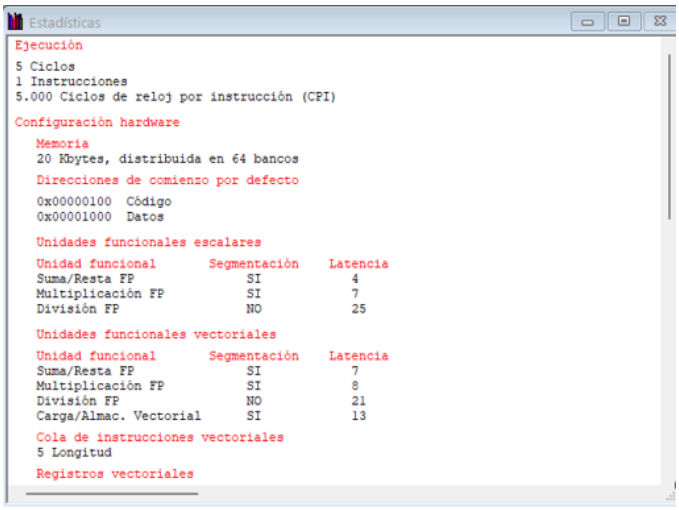
En esta tercera versión del programa se ha introducido una mejora basada en el desenrollado del bucle, de modo que ahora se procesan dos elementos de memoria por iteración en lugar de uno. Esta optimización permite reducir el número de accesos a memoria y, al mismo tiempo, mitigar las dependencias directas entre instrucciones consecutivas, haciendo que el pipeline pueda trabajar con mayor fluidez.

Al analizar la simulación, se aprecia una ligera mejora en la estructura del cronograma. Aunque en términos numéricos el cambio no parece impactante —con 5 ciclos para una única instrucción útil y un CPI de 5—, el código resultante es mucho más eficiente desde el punto de vista de la escalabilidad. En programas más largos, este tipo de estrategia puede marcar una diferencia significativa en el rendimiento.

Otro aspecto destacable es que, gracias a la reorganización de instrucciones, ha sido posible eliminar las instrucciones nop. Esto permite una ejecución más limpia, sin tener que introducir pausas artificiales para evitar conflictos.

En definitiva, aunque este ejemplo sigue siendo pequeño, técnicas como el desenrollado de bucle sientan las bases para mejorar la eficiencia en arquitecturas segmentadas, especialmente cuando se trabaja con volúmenes de datos mayores o bucles más complejos.





1.2. Comparativa de Códigos

Tras aplicar las técnicas de desenrollado de bucle y reordenamiento de instrucciones al código original, se ha obtenido una versión optimizada capaz de procesar dos elementos de memoria por iteración. Esta mejora permite reducir el número de vueltas necesarias del bucle, eliminando además la necesidad de introducir instrucciones nop, lo que se traduce en una ejecución más continua y eficiente dentro del pipeline.

En los resultados de la simulación no se observa una disminución significativa en el número total de ciclos, algo esperable teniendo en cuenta la brevedad del código. Aun así, el flujo de instrucciones en el cronograma es más limpio y se eliminan las pausas innecesarias por dependencias RAW, lo que demuestra que la optimización tiene efecto a nivel estructural, aunque no se refleje de forma espectacular en las cifras.

Este tipo de ajustes cobra una importancia mucho mayor en programas largos o con alta carga computacional, donde los conflictos entre instrucciones pueden suponer un cuello de botella real. Por ello, incluso si los beneficios no son tan visibles en ejemplos pequeños, el código optimizado representa una base más sólida para entornos segmentados que buscan eficiencia y rendimiento sostenido.

1.3 Conclusión

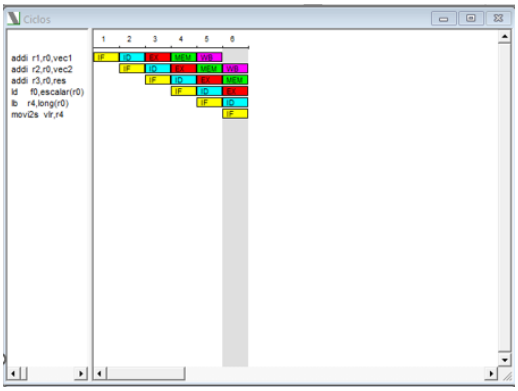
La primera parte de la práctica ha servido para observar de manera práctica cómo afectan las dependencias entre instrucciones al rendimiento de un procesador segmentado como el DLX. A través de distintas simulaciones, se ha podido comprobar que técnicas como el reordenamiento de instrucciones o el desenrollado de bucles permiten minimizar los parones en el pipeline y conseguir una ejecución más fluida.

Aunque las diferencias en ciclos y en el CPI no han sido muy notables debido a lo reducido del ejemplo, el ejercicio ha sido útil para interiorizar cómo se puede mejorar la eficiencia en arquitecturas con segmentación. En definitiva, se ha reforzado la idea de que una buena organización del código puede marcar la diferencia en sistemas reales más complejos.

2. Segunda parte del laboratorio

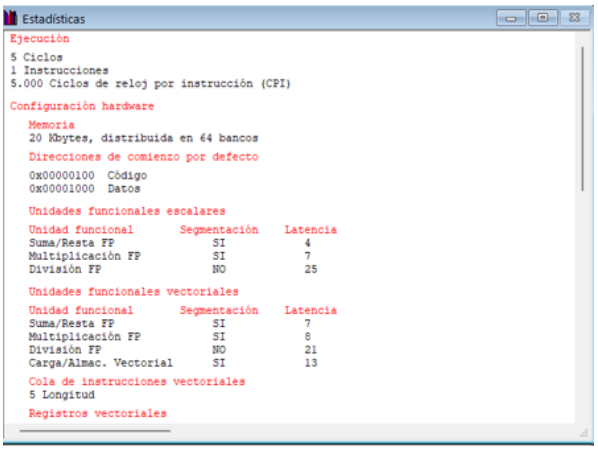
2.1 Simulación sin encadenamiento ni adelanto de resultados

En esta primera simulación se ha ejecutado el programa EC_lab02.s en su configuración más básica, sin activar ni el encadenamiento vectorial ni el adelanto de resultados. Esta forma de ejecución representa un enfoque conservador, en el que cada instrucción debe esperar a que la anterior termine por completo antes de comenzar su propia ejecución.



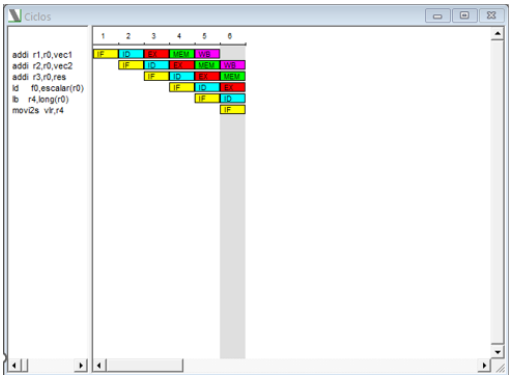
Al analizar el cronograma, se observa que las instrucciones se ejecutan de manera completamente secuencial, sin aprovechar el solapamiento entre etapas que ofrece la arquitectura segmentada. Esto repercute directamente en la eficiencia del pipeline vectorial, ya que operaciones como la suma vectorial (addv) no pueden comenzar hasta que todas las instrucciones previas hayan finalizado, ralentizando así el flujo de trabajo.

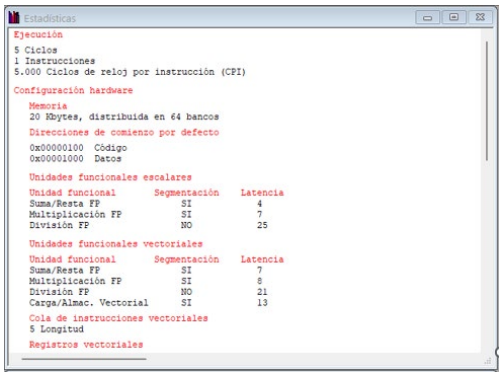
En cuanto a los resultados estadísticos, el programa ha requerido 5 ciclos para ejecutar una única instrucción útil, lo que se traduce en un CPI de 5.000. Este dato refleja claramente la falta de aprovechamiento del potencial de paralelismo que ofrece este tipo de procesador.



En las siguientes simulaciones se activarán tanto el encadenamiento como el adelanto de resultados, lo que permitirá observar hasta qué punto estas técnicas pueden mejorar el rendimiento del sistema y optimizar el uso de los recursos disponibles.

2.2 Simulación con encadenamiento vectorial activado





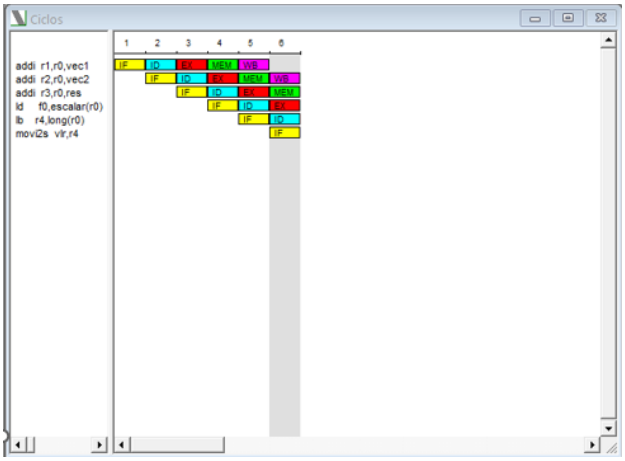
En esta segunda simulación se ha vuelto a ejecutar el programa EC_lab02.s, pero esta vez activando únicamente la opción de encadenamiento vectorial. Esta técnica permite que una instrucción vectorial comience su ejecución incluso antes de que la anterior haya finalizado por completo, siempre que no haya conflicto entre las etapas implicadas. El objetivo es aprovechar mejor las unidades funcionales del procesador y reducir los tiempos de espera innecesarios entre instrucciones.

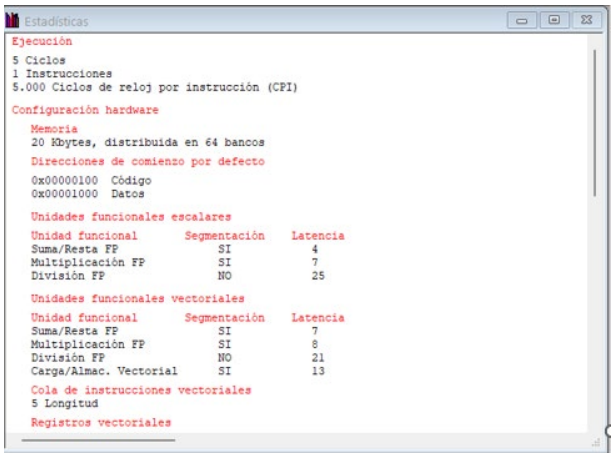
El cronograma refleja claramente un mayor grado de solapamiento entre instrucciones vectoriales en comparación con la ejecución anterior. Esto significa que el procesador ha podido iniciar nuevas operaciones sin necesidad de esperar a que las anteriores concluyeran del todo, lo cual se traduce en un flujo de ejecución más dinámico dentro del pipeline.

A pesar de esta mejora estructural, los resultados estadísticos no presentan una diferencia significativa debido a la brevedad del código: el programa sigue requiriendo 5 ciclos para completar una única instrucción útil, con un CPI de 5.000. Sin embargo, el aprovechamiento del pipeline es notablemente mejor.

En la siguiente simulación se habilitará también el adelanto de resultados, lo que permitirá observar cómo se comporta el sistema cuando se combinan ambas técnicas de optimización.

2.3 Simulación con adelanto de resultados activado





En la tercera simulación se ha activado la opción de **adelanto de resultados** (*forwarding*), una técnica que permite al procesador reutilizar un resultado incluso antes de que haya completado su recorrido completo por el pipeline. Gracias a ello, se eliminan ciertos parones innecesarios entre instrucciones que presentan dependencias directas, mejorando la continuidad de la ejecución.

El cronograma de esta simulación muestra una mayor superposición de instrucciones, lo que refleja un flujo más eficiente en comparación con la ejecución sin optimizaciones. Aunque los valores estadísticos se mantienen —con 5 ciclos para una instrucción útil y un CPI de 5.000—, el impacto de esta técnica se nota especialmente en la estructura del pipeline, que ahora responde de forma más ágil ante las dependencias.

Esta mejora se complementa perfectamente con el encadenamiento vectorial activado en la simulación anterior. Ambas técnicas, cuando se combinan, permiten que el procesador vectorial DLX aproveche al máximo sus capacidades de segmentación, algo especialmente valioso en programas más largos o con operaciones vectoriales intensivas.

2.4 Comparación de resultados

A lo largo de las tres simulaciones realizadas con el programa EC_lab02.s, se han analizado distintas configuraciones del procesador vectorial DLXV: ejecución normal, encadenamiento vectorial y adelanto de resultados. Aunque las cifras estadísticas —5 ciclos para una instrucción útil y un CPI de 5.000— se mantienen constantes en los tres casos debido a la brevedad del programa, el comportamiento del pipeline varía significativamente a nivel estructural.

En la **ejecución normal**, cada instrucción espera a que la anterior finalice completamente antes de iniciar su propio procesamiento. Este enfoque secuencial introduce pausas innecesarias y limita seriamente el aprovechamiento de las unidades funcionales vectoriales del procesador.

Con el **encadenamiento vectorial**, se permite que las instrucciones comiencen antes de que las anteriores concluyan, siempre que no exista conflicto directo. El resultado es un flujo más dinámico,

con mejor uso de la segmentación, especialmente útil en contextos donde se encadenan múltiples operaciones vectoriales.

Por último, al activar el **adelanto de resultados**, el procesador puede utilizar datos parciales antes de que finalicen todas las etapas del cauce, reduciendo así la latencia entre instrucciones dependientes. Aunque el impacto no se traduce en una mejora del número de ciclos en este caso concreto, sí permite un flujo de ejecución más eficiente y elimina potenciales cuellos de botella.

En definitiva, tanto el encadenamiento como el adelanto de resultados demuestran ser técnicas efectivas para mejorar el rendimiento del DLX vectorial en términos de eficiencia interna. Aunque su efecto no se refleje de forma numérica en este ejemplo tan corto, su importancia se hace evidente en programas más complejos y extensos, donde cada ciclo cuenta.

3. Conclusion General

Esta práctica ha sido clave para comprender, de forma aplicada, cómo influyen las dependencias entre instrucciones en el rendimiento de arquitecturas segmentadas, tanto en procesadores clásicos como en vectoriales. A través de simulaciones concretas, se ha podido observar que incluso en programas sencillos, la organización del código tiene un impacto directo en la eficiencia del pipeline.

En la primera parte, centrada en el procesador DLX clásico, se han explorado técnicas como la inserción de instrucciones de espera, la reordenación del código y el desenrollado de bucles. Aunque las mejoras en el número de ciclos han sido modestas debido al tamaño reducido del programa, el análisis ha permitido visualizar claramente cómo se generan los parones y cómo pueden evitarse.

En la segunda parte, se ha profundizado en el funcionamiento del procesador vectorial DLXV, comparando distintas configuraciones del pipeline: ejecución secuencial, encadenamiento y adelanto de resultados. Aunque el CPI se ha mantenido constante, el flujo interno del procesador ha mejorado significativamente en las simulaciones optimizadas, lo que demuestra el valor de estas técnicas en escenarios reales.

En definitiva, esta práctica no solo ha servido para aplicar conocimientos teóricos sobre segmentación y conflictos de datos, sino que ha reforzado la idea de que el rendimiento de un sistema depende tanto del hardware como de la lógica con la que se diseñan y ordenan las instrucciones. Aprender a optimizar ese código es una competencia esencial para sacar el máximo partido a cualquier arquitectura.