APUNTES, MANUALES, PRESENTACIONES

Universidad de Jaén

Escuela Politécnica Superior de Jaén

Arquitectura de computadores: prácticas con RISC-V

Francisco Charte Ojeda

Fecha: 09/2024

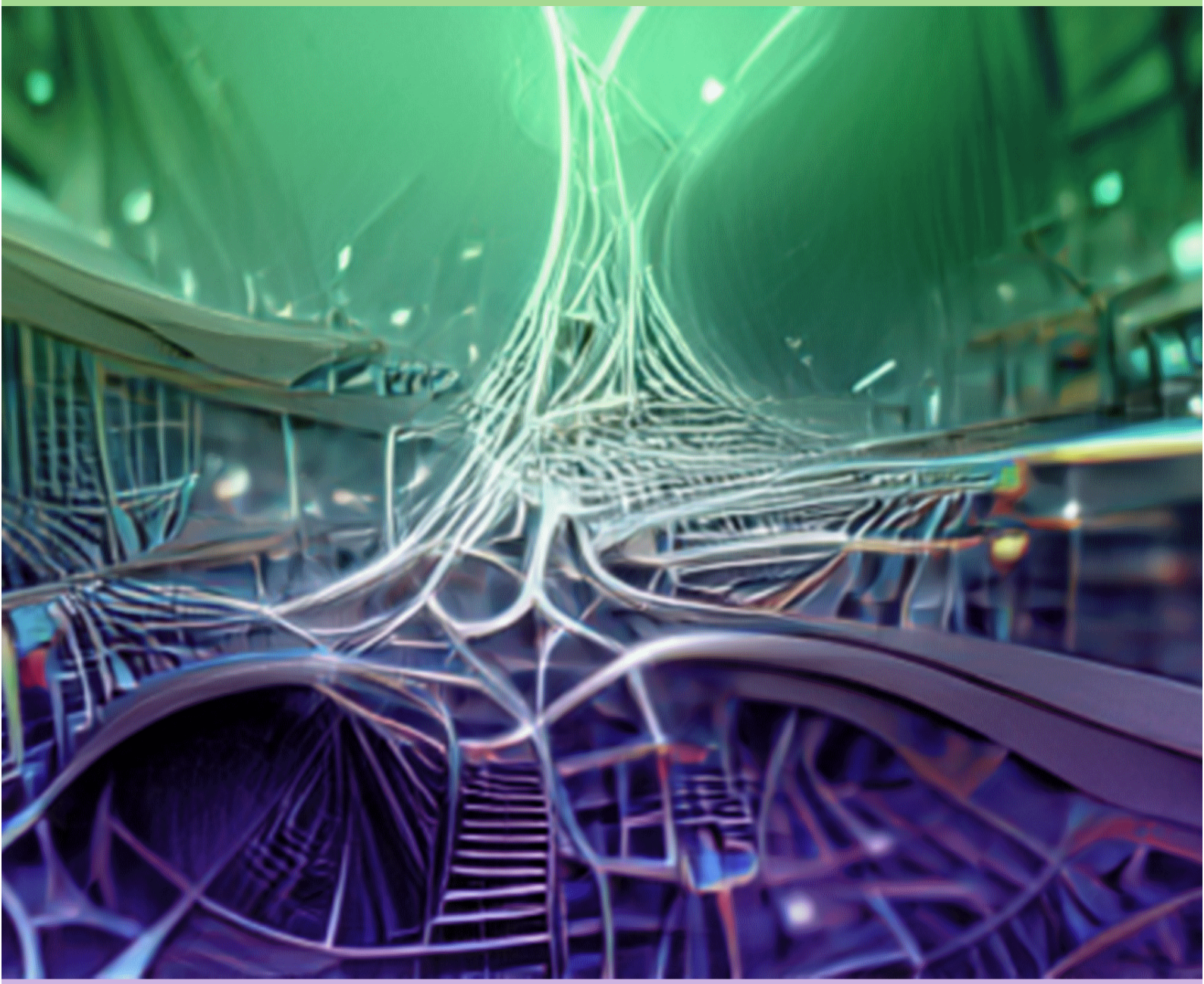
ASIGNATURA: ARQUITECTURA DE COMPUTADORES

CREA

Universidad de Jaén

Arquitectura de computadores

Prácticas con RISC-V



Francisco Charte Ojeda

Universidad de Jaén

Arquitectura y Tecnología de Computadores
Departamento de Informática
Escuela Politécnica Superior de Jaén

Grado en Ingeniería Informática - 2º curso

Arquitectura de computadores
Prácticas con RISC-V

Francisco Charte Ojeda
<fcharte@ujaen.es>

https://fcharte.com

Jaén, 2 de septiembre de 2024 (última actualización)

Arquitectura de computadores
Prácticas con RISC-V

Grado en Ingeniería Informática - 2º curso

Francisco Charte Ojeda
<fcharte@ujaen.es>

Departamento de Informática

Escuela Politécnica Superior

Universidad de Jaén

Arquitectura de computadores
Prácticas con RISC-V

Economía lingüística
A lo largo de este documento el autor ha procurado emplear lenguaje inclusivo siempre que ha sido posible pero, por economía lingüística, ha de saberse que los sustantivos y determinantes masculinos no se emplean solo para referirse a personas de ese sexo, sino que han de entenderse como genéricos y, por tanto, que designan a todas las personas sin distinción de sexo. Así, para referirse a un individuo del colectivo del *profesorado* se usa la forma *el profesor* que hay que hacer extensiva a *el profesor o la profesora*.

Copyright
Francisco Charte Ojeda, 2024

Colophon
This document was typeset with the help of KOMA-Script and LaTeX using the kaobook class.

Ponte manos a la obra sin miedo a equivocarte y a fallar tantas veces como sea preciso. De los errores cometidos siempre se aprende mucho más que de no hacer nada.

Índice general

Índice de figuras

Índice de tablas

Índice de listados de código

Introducción a RISC-V y Ripes 1

1.1. Planificación

- **Objetivo.** Familiarizarnos con la sintaxis del lenguaje ensamblador de la **ISA RISC-V** y sus registros, usándola para escribir programas sencillos. Aprender a usar la herramienta Ripes.
- **Temporización:**
  - **Trabajo en laboratorio**: introducción a la arquitectura de RISC-V, el conjunto de registros y su funcionamiento básico. Instrucciones de carga, almacenamiento y aritméticas. Uso de Ripes para escribir un programa sencillo y aprender a usar las funciones que permiten ejecutar un programa, examinar memoria y registros, etc. Resolución de dudas.
  - **Trabajo autónomo**: experimentación con la ISA y Ripes. Ejercicios de programas simples.
  - **Validación en laboratorio**: test y ejercicios evaluables.
- **Herramientas.** La herramienta fundamental de trabajo durante todas las prácticas será el emulador Ripes[1]. Este puede ejecutarse en Windows, macOS y GNU/Linux de manera indistinta.

1: Ripes es un proyecto de código abierto disponible en el repositorio de Github `https://github.com/mortbopet/Ripes`.

1.2. La ISA de RISC-V

Se denomina ISA[2] a la interfaz que determina cómo mediante software se controla el hardware de un microprocesador. La ISA de RISC-V [1] nos permitirá escribir programas para cualquier CPU que implemente dicha ISA.

2: *Instruction Set Architecture*

[1]: Waterman (2016), *Design of the RISC-V instruction set architecture*

Forman parte de la ISA aspectos como los siguientes:

- El **banco de registros** visibles a los programas.
- Los **modos de direccionamiento** que facilitan el acceso a memoria.
- El **conjunto de instrucciones** que ofrece para realizar distintas operaciones.

En estas prácticas usaremos la ISA de RISC-V denominada **RV32IM**: *RISC-V 32 bits Integer and Multiplication*, en su versión 2.0. El ancho de palabra, y de las instrucciones, es de 32 bits y se contemplan instrucciones de aritmética entera, incluyendo la multiplicación y división.

Banco de registros

Los registros son pequeñas porciones de memoria internas a la CPU. Su finalidad es mantener temporalmente los datos con que se opera y los resultados que producen las operaciones sobre ellos. En RV32I contamos con **32 registros** de propósito general. Cada uno de ellos tiene un tamaño de **32 bits**. La denominación genérica para estos registros es `xN`, siendo `N` un número entre 0 y 31.

A pesar de la denominación de **propósito general**, lo cierto es que algunos de esos 32 registros se emplean para fines específicos según una convención para el desarrollo de compiladores para RISC-V. La Tabla 1.1 denota esos usos estandarizados y los nombres alternativos (*alias*) con los que es posible acceder a los registros.

En asignaturas previas de la titulación, como *Fundamentos de arquitectura de computadores*, probablemente hayas conocido otras ISA, como la del Intel 8085, que tienen conjuntos de registros diferentes, con un acumulador llamado `A` y seis registros de propósito general denominados `B`, `C`, `D`, `E`, `H` y `L`, todos ellos con un tamaño de 8 bits. Otras asignaturas emplean la ISA del Intel 8086/8088, con registros de 16 bits denominados `AX`, `BX`, `CX`, etc.

Además de estos, la arquitectura también cuenta con el registro `pc` (*Program Counter*[3]) —que contiene la dirección de la instrucción a ejecutar— así como otros registros de uso específico que, por el momento, no nos resultan de interés.

3: Al igual que los de propósito general, el `pc` también es un registro de 32 bits en la ISA RV32I.

Tabla 1.1. Conjunto de registros de propósito general en RISC-V. La primera columna corresponde al nombre real de cada registro, la segunda al *alias* que puede usarse para referirse a cada uno de ellos y en la tercera se facilita una descripción breve de su finalidad.
La denominación *registro preservado* significa que el contenido de ese registro no será modificado al realizar una llamada a una función, por lo que su valor se preserva.
Un *frame* es una estructura de datos que se crea habitualmente al llamar a una función, con un espacio reservado para almacenar parámetros y otra información útil como la dirección de retorno.

Nombre	Alias	Uso
`x0`	`zero`	Siempre contiene el valor 0
`x1`	`ra`	Dirección de retorno
`x2`	`sp`	Puntero de pila
`x3`	`gp`	Puntero global
`x4`	`tp`	Puntero de hilo
`x5-7`	`t0-2`	Valor temporal
`x8`	`s0/fp`	Registro preservado / Puntero de *frame*
`x9`	`s1`	Registro preservado
`x10-11`	`a0-1`	Parámetro de función / Valor de retorno
`x12-17`	`a2-7`	Parámetro de función
`x18-27`	`s2-11`	Registro preservado
`x28-31`	`t3-6`	Valor temporal

Conjunto de instrucciones

Además de su banco de registros, para escribir programas necesitaremos conocer también las instrucciones con que cuenta RISC-V [2]. Este conjunto cambia según las extensiones de la ISA que implemente el procesador. Para RV32I es bastante reducido[4] al constar de **solo 40 instrucciones** distintas, con variantes según de dónde se tomen los operandos. A estas, RV32IM añade varias más para el cálculo de multiplicaciones y divisiones.

La Tabla 1.2 resume el conjunto completo de instrucciones dividiéndolo en varios grupos y según que ambos operandos estén en registros o uno de ellos sea un valor inmediato. Conoceremos, en esta y las prácticas posteriores, cómo usar muchas de estas instrucciones.

[2]: Waterman et al. (2017), *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*

4: En comparación, las últimas generaciones de microprocesadores x64 (familias Intel Core y AMD Ryzen) cuentan con un conjunto de unas 1500 instrucciones.

Tabla 1.2. Conjunto de instrucciones de la ISA RV32IM.

Grupo ↓ Tipo →	Registro	Inmediato
Aritmético	`add, sub, mul, div, rem`	`addi, lui, auipc`
Lógico	`and, or, xor`	`andi, ori, xori`
Desplazamiento	`sll, srl, sra`	`slli, srli, srai`
Comparación	`slt, sltu`	`slti, sltiu`
Carga		`lb, lbu, lh, lhu, lw, lwu, ld`
Almacenamiento		`sb, sh, sw, sd`
Salto		`beq, bne, bge, bgeu, blt, bltu, jal, jalr`
Otras	`ecall, ebreak`	

Seudoinstrucciones

La filosofía de los procesadores RISC, en términos generales, se basa en una ISA con pocas instrucciones encargadas de facilitar operaciones básicas[5]. Esto hace que la implementación de dicha ISA, en un microprocesador, sea relativamente sencilla, al tiempo que también se simplifican técnicas que conoceremos más adelante como la segmentación del cauce de ejecución.

5: La carga de un dato desde memoria a un registro, un cálculo aritmético sobre dos datos que están en registros, el almacenamiento del valor contenido en un registro en una determinada posición de memoria, etc.

El inconveniente de esa filosofía es que se precisan más instrucciones ensamblador para realizar ciertas tareas, mientras que en un diseño CISC existirían instrucciones complejas para llevarlas a cabo. RISC-V contempla el uso de seudoinstrucciones que, desde la perspectiva del programador, no son más que instrucciones algo más complejas. Esas seudoinstrucciones son procesadas por el ensamblador[6], no por el propio microprocesador, de forma que cada una de ellas genera una o múltiples instrucciones nativas.

6: Aquí nos referimos al programa (el *assembler*) que traduce el código ensamblador (*assembly code*) a código máquina RISC-V.

A medida que escribamos código RISC-V conoceremos varias de las seudoinstrucciones de uso común. Será la herramienta con la que trabajemos, el emulador Ripes, el que se encargue de traducirlas a las instrucciones que correspondan.

1.3. Hardware RISC-V

En la actualidad los procesadores RISC-V se emplean principalmente en sistemas embebidos (o empotrados) y en supercomputadores, pero no en ordenadores personales[7] como pueden ser los portátiles o los clásicos equipos de sobremesa. Estos recurren, sobre todo, a CPU con arquitectura x86, x64 o ARM.

7: No obstante, ya hay algunos productos como el portátil DC-ROMA con un procesador RISC-V de 64 bits y 8 núcleos.

En el campo de los sistemas embebidos contamos con el microcontrolador ESP32-C3 mostrado en la Figura 1.1. Este cuenta con un núcleo RISC-V de 32 bits a 160 Mhz, WiFi, Bluetooth y 400 KB de memoria RAM. El fabricante ofrece herramientas de código abierto que permiten desarrollar programas para este sistema.

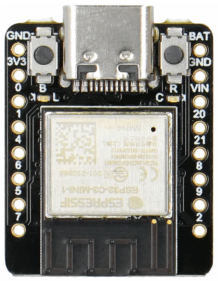


Figura 1.1. Placa ESP32 con un núcleo RISC-V de 32 bits.

Un escalón por encima del anterior sería el equipo VisionFive 2, un SBC (*Single Board Computer*, Ordenador en una sola placa) que dispone de un procesador StarFive mucho más potente. Esto le permite ejecutar sistemas operativos diseñados para RISC-V, como es el caso de GNU/Linux, y facilitar el desarrollo de software para esta ISA de manera mucho más general que con un microcontrolador. Como se aprecia en la Figura 1.2, una vez montado en su caja no difiere de cualquier ordenador personal en cuanto a entradas, salidas y capacidad se refiere.

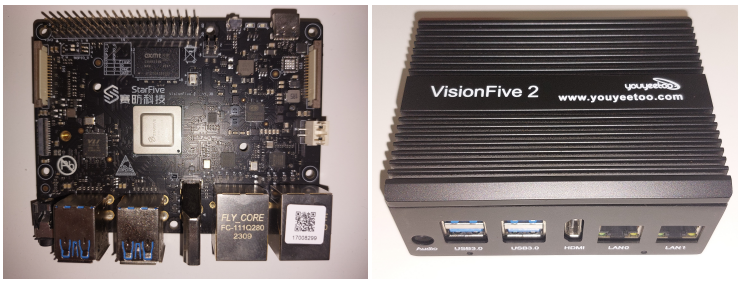


Figura 1.2. Microordenador VisionFive 2 con procesador RISC-V (placa a la izqda., montado a la dcha.). Cuenta con cuatro núcleos StarFive que implementan la API RV64GC y operan a 1.5 GHz e integra también una GPU.

El microprocesador StarFive implementa la ISA RV64GC. Esto implica que cuenta con una arquitectura de 64 bits y dispone de todas las extensiones de propósito general: enteros, multiplicación, punto flotante, operaciones atómicas y compresión de instrucciones.

En el campo de la supercomputación (HPC, *High-Performance Computing*), existen múltiples fabricantes que ofrecen CPU con miles de núcleos RISC-V, así como desarrollos en curso de superordenadores como el del Centro de Supercomputación de Barcelona.

1.4. La herramienta Ripes

Como acaba de describirse, hay disponibles distintos tipos de hardware con procesadores RISC-V. No obstante, en general no son opciones que estén a disposición de todas las personas. Por esa razón recurriremos a un software de

simulación. Este nos facilitará la edición y ejecución de programas para la ISA RV32IM y RV64IM, permitiendo examinar el contenido de los registros, memoria y algunos dispositivos de entrada/salida, algo difícil de hacer en hardware real.

De entre los simuladores RISC-V disponibles elegiremos Ripes [3], un entorno de trabajo que aúna editor, ensamblador, visor del cauce de ejecución, visor de memoria, emulación de dispositivos de E/S, etc.

[3]: Petersen (2021), «Ripes: A Visual Computer Architecture Simulator»

Descarga e instalación

La última versión de este simulador siempre puede encontrarse en el apartado *Releases* del repositorio Github (véase la Figura 1.3) donde lo mantienen sus desarrolladores. Debemos descargar el paquete que corresponda a nuestro sistema operativo y proceder a su instalación.

La versión que utilizaremos en el laboratorio de prácticas está disponible para su descarga desde la plataforma de docencia virtual PLATEA.

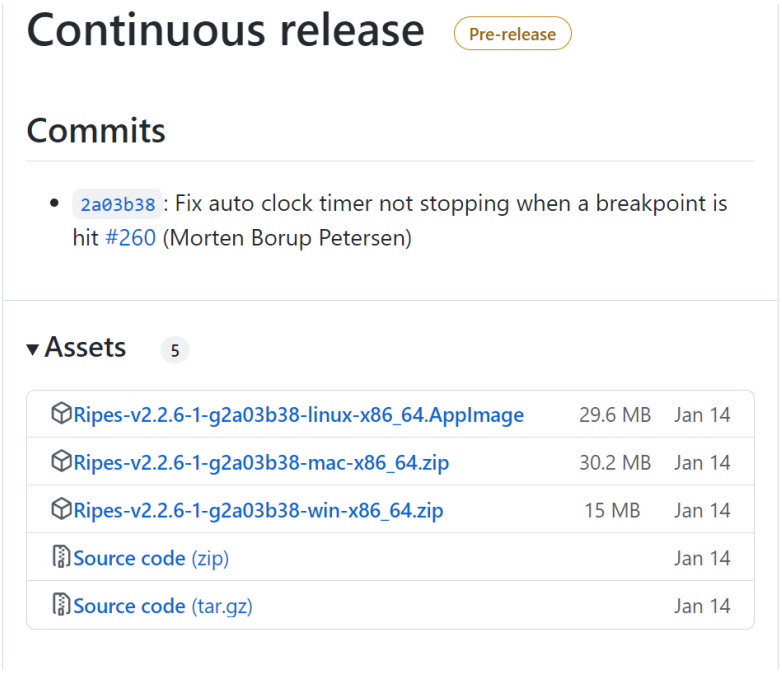


Figura 1.3. Descargamos el software Ripes para nuestro sistema operativo.

En Windows y macOS no hay más que descomprimir el contenido del archivo descargado en una carpeta. En ella encontraremos el ejecutable, por lo que basta con hacer doble clic o escribir **ripes** en la consola para ejecutarlo. En GNU/Linux el archivo descargado es el propio ejecutable, pero antes de iniciarlo con un doble clic puede ser preciso abrir el menú contextual, como en la Figura 1.4, y activar el permiso de ejecución o indicar que se debe confiar en el ejecutable.[8]

8: Una alternativa a la instalación de este software en nuestro equipo consiste en acceder a la herramienta desde el navegador web, a través del URL `https://ripes.me`. Debemos tener en cuenta, sin embargo, que esta versión no dispone de opciones para guardar y recuperar nuestro código. No obstante, es una forma cómoda y rápida de familiarizarnos con los elementos de la interfaz de usuario que se describirán en los siguientes apartados.

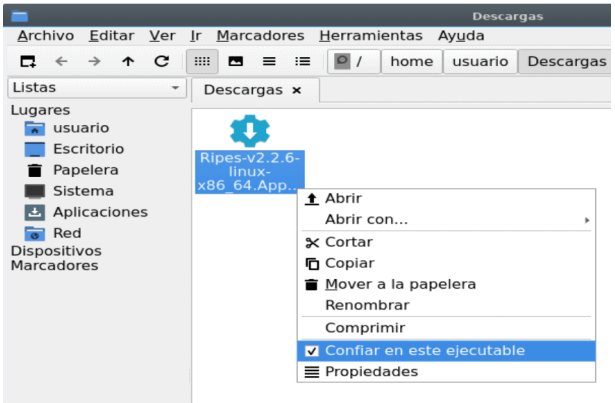


Figura 1.4. Configuración para poder ejecutar Ripes en GNU/Linux.

La interfaz de Ripes

La interfaz de usuario de Ripes dispone de varias vistas distintas. Al ejecutarlo por primera vez es habitual que la que se abra sea **Processor**, como se aprecia en la Figura 1.5. En ella se muestra el cauce de ejecución en la parte central. El panel de la derecha muestra la lista de registros de propósito general, descrita en el siguiente apartado, mientras que la parte inferior facilita información sobre el proceso de ejecución del programa.

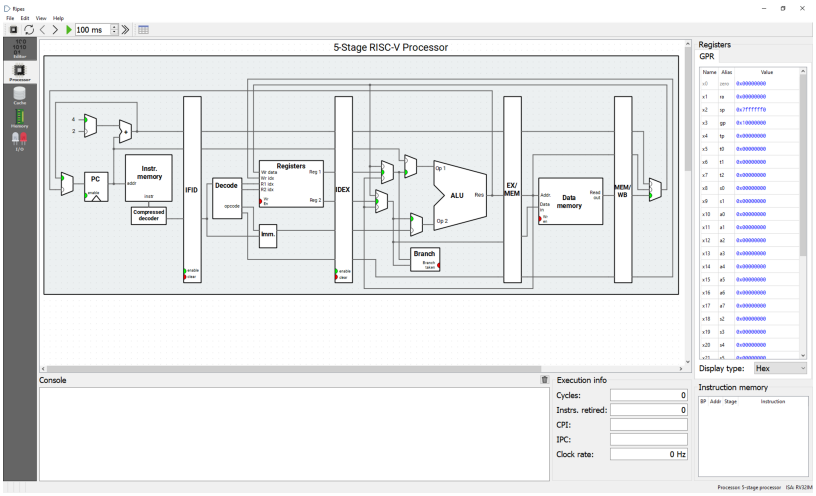


Figura 1.5. Vista por defecto de la GUI de Ripes tras iniciarse. Los botones situados en el margen izquierdo, con un color de fondo más oscuro, son los encargados de cambiar entre las vistas ofrecidas.

Los botones situados en la barra lateral izquierda, con un fondo más oscuro, son los encargados de cambiar entre las vistas existentes[9]. Para comenzar nos interesa la vista en la que editaremos el código. La abrimos con un clic en el primer botón del grupo mencionado, con el título **Editor**.

9: Conoceremos los detalles de cada una de ellas a medida que sea preciso.

Editor de registros

En las dos vistas mencionadas tendremos en el margen derecho el editor de registros, con el nombre **GPR** (*General Purpose Registers*) en la parte superior (véase la Figura 1.6). Se trata de una lista con el nombre, alias y valor actual de cada uno de los registros de propósito general.

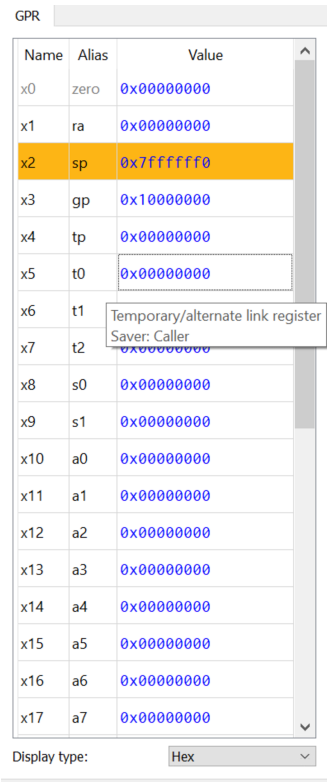


Figura 1.6. El editor de registros.

Sobre este editor es posible llevar a cabo distintas acciones, a saber:

- **Ayuda:** se obtiene una descripción de cualquier registro al situar el puntero del ratón sobre su nombre.
- **Formato:** con la lista desplegable situada en la parte inferior se elige el formato de visualización.
- **Notificación:** durante la ejecución de un programa se notifica la escritura en cualquier registro cambiando el color de fondo con el que aparece en el editor. En la Figura 1.6 se aprecia con dicho formato resaltado el registro `sp`[10] que Ripes inicializa con un valor adecuado.
- **Modificación:** con un doble clic sobre el valor se permite cambiar su contenido.

Al escribir un programa en ensamblador somos libres de emplear el nombre original de cada registro o bien su alias, sin que esto afecte en forma alguna al resultado. Es habitual optar por la segunda opción ya que los alias tienen un sentido semántico del que carecen los nombres reales.

10: El registro `sp` actúa como puntero de pila. RISC-V no cuenta con un registro específico para esta función, como ocurre con otras arquitecturas de procesador, por lo que se emplea uno de los GPR para ello.

Editor de código

Tras cambiar a la vista del **Editor** de código, con el botón antes indicado, estamos en disposición de introducir órdenes en ensamblador RISC-V en las que emplearemos una instrucción de las enumeradas en la Tabla 1.2 y uno o más registros de los mencionados en la Tabla 1.1.

Sumar dos números

Comencemos por algo muy sencillo: alojar dos valores en sendos registros y sumarlos. El resultado lo mostraremos en la consola de Ripes. El código necesario para ello es el siguiente:

Listado 1.1: Cargar dos valores en sendos registros y sumarlos.

```
1  addi a1, zero, 3    # a1 = 3
2  addi a2, zero, 7    # a2 = 7
3  add a0, a1, a2      # a0 = a1 + a2
4  addi a7, zero, 1
5  ecall               # Llamar a función 1 (imprimir entero en a0)
```

Como es habitual en procesadores RISC, no existe una instrucción para cargar en un registro un valor inmediato, de ahí que se emplee la instrucción `addi` con el registro `zero` y el valor que interese. Tampoco se contempla la copia de un registro a otro, para lo cual se puede emplear la seudoinstrucción `mv rd, valor` que produce dicho resultado.

A medida que introducimos estas instrucciones en el editor, en el panel de la derecha aparecerá el código máquina que generan (en binario) o bien su versión desensamblada, según la opción que esté marcada en la parte superior.

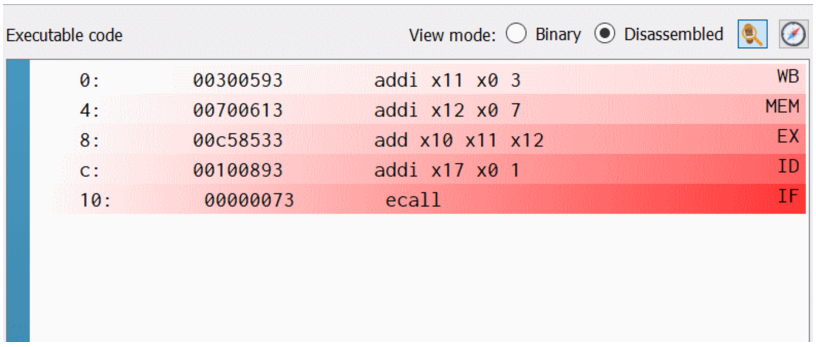


Figura 1.7. Panel con el código en versión ejecutable. Cada fila corresponde a una de las instrucciones introducidas en el editor. Las columnas, de izquierda a derecha, indican 1) la dirección de memoria en la que se almacena el código máquina de la instrucción, 2) dicho código máquina en formato hexadecimal, 3) la instrucción real que ejecutará el microprocesador y 4) la etapa de ejecución en que se encuentra (asumiendo que se tiene activa una configuración de procesador segmentado, algo que conoceremos después).

Controles de ejecución

Una vez que se ha escrito (o cargado) un programa en el editor de Ripes, y asumiendo que no contenga errores, podrá procederse a su ejecución. Con este fin empleamos los botones situados en la parte superior, concretamente los señalados en la Figura 1.8. Los de uso más habitual son ›, para ejecutar un ciclo del programa; ‹, que deshace el último ciclo ejecutado, y » que ejecuta el programa hasta su finalización.

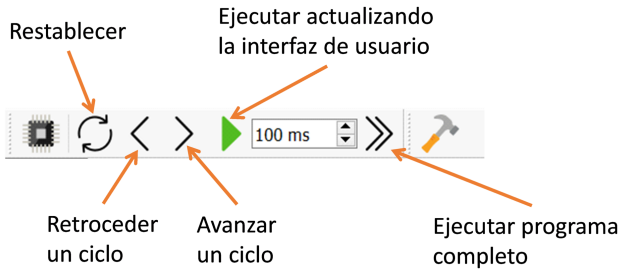


Figura 1.8. Controles de ejecución de Ripes. La ejecución también se restablece, de manera automática, si se modifica el código en el editor.

En lugar de los botones también podemos recurrir a atajos de teclado, por ejemplo **F5**, **F4** y **F8**, para realizar las mismas acciones antes mencionadas.

La consola

La parte inferior de la vista **Editor** de Ripes está ocupada por un panel con el título **Console**. Su finalidad es mostrar información enviada por el programa a la salida estándar. Al ejecutar el anterior ejemplo debería aparecer en dicho panel el resultado de la suma que se aloja en el registro `a0`.

El modo de invocar a un servicio del sistema en RISC-V es mediante la instrucción `ecall`. Esta espera que se entregue en el registro `a7` el servicio a ejecutar. Cada servicio está identificado con un código numérico[11] y precisará una lista de argumentos a medida. Hay servicios que devuelven resultados y otros que no.

Mediante el servicio `1`, valor asignado a `a7` en nuestro programa, se envía a la consola el número entero almacenado en el registro `a0` que, en este caso concreto, es el resultado de la suma producida por la instrucción `add a0, a1, a2`.

11: Usa la opción **Help→System calls** para abrir una ventana en la que se detallan los códigos de servicio implementados en Ripes y los parámetros de entrada y salida.

Guardar y cargar programas

Como es habitual, las opciones para almacenar el código de un programa en disco, guardándolo en un archivo que habitualmente tendrá extensión `.s`, o bien para cargar un programa guardado con anterioridad las encontramos en el menú **File**. Esta será la vía para guardar los ejercicios en archivos y entregarlos a través de la plataforma de docencia virtual.

En el mencionado menú también encontramos la opción **Load Example→Assembly**, mediante la cual es posible abrir diferentes programas de ejemplo provistos por el propio software.

1.5. Instrucciones aritméticas

Además de la operación de suma, RISC-V cuenta con instrucciones que facilitan la realización de otros cálculos aritméticos:

- `sub destino, minuendo, sustraendo`: resta `sustraendo` de `minuendo` y deja el resultado en `destino`.
- `mul destino, multiplicando, multiplicador`: multiplica `multiplicando` por el `multiplicador`.
- `div destino, dividendo, divisor`: efectúa la división entera de `dividendo` entre `divisor` y devuelve el cociente.
- `rem destino, dividendo, divisor`: efectúa la división entera de `dividendo` entre `divisor` y devuelve el resto.

Observamos que hay dos instrucciones de suma —`add` y `addi`— que difieren en el tipo del tercer argumento que toman. En realidad es una sola instrucción que admite dos **modos de direccionamiento**, aspecto que tratamos en la Sección 1.7. El resto de instrucciones aritméticas no cuentan con la segunda variante que toma como parámetro un valor constante.

Manos al teclado

Parte del programa anterior y modifica la instrucción que suma `a1` y `a2` para realizar una a una las operaciones que se enumeran aquí, ejecutando el programa con `F8` para ver en la consola el resultado de cada caso.

Todos los operandos han de ser registros y el primero de ellos siempre actúa como destino del resultado.

La instrucción `addi` es la vía para asignar un cierto valor a un registro, para lo cual se suma al registro `zero`: `addi a0, zero, 7` asignaría el valor 7 a `a0`. Este uso es tan común que existe una seudoinstrucción —`li rd, imm`— que genera la anterior instrucción.

1.6. Carga y almacenamiento de datos

En la mayoría de casos los datos con los que opera un programa están almacenados en memoria separados del código. Es decir, no aparecen como valores constantes (o literales) en las propias instrucciones del programa. Asimismo, lo habitual es que los resultados obtenidos se alojen también en memoria.

Al programar en C/C++ y otros lenguajes de alto nivel leemos y escribimos de memoria a través de la variables. En ensamblador operaremos con direcciones de memoria, a las que se asignan nombres en forma de etiquetas, por lo que el mecanismo será análogo.

El procedimiento habitual para acceder a una posición de memoria, ya sea para leer o escribir, estriba en asignar a un registro dicha dirección y usarlo, a **modo de puntero**, para cargar (leer) o almacenar (escribir) un valor.

Espacio de direccionamiento y mapa de memoria

La ISA RV32IM contempla un espacio de direccionamiento de 32 bits. Esto implica que hay disponibles 2^{32} posiciones de memoria[12], cada una de las cuales tiene 1 byte de capacidad. En ese espacio hay que almacenar el código del programa a ejecutar, los datos sobre los que operará el programa, reservar una zona para dispositivos de entrada/salida, etc.

12: Las direcciones de memoria suelen notarse en base hexadecimal. Ya que cada dígito headecimal representa 4 bits, se precisan 8 posiciones en total para una dirección de 32 bits, de forma que la primera dirección (la más baja) es `0x00000000` y la última es `0xFFFFFFFF`.

Según el equipamiento en el que se integre el procesador RISC-V, ese espacio de direccionamiento se estructurará de una forma u otra. En el simulador Ripes el mapa de memoria es el representado en la Figura 1.9. En el lateral izquierdo se ha dividido el espacio total de 4GB en 16 porciones iguales de 256MB. En el margen derecho se indican las direcciones de inicio, y de fin en su caso, de las secciones.

La configuración del mapa de memoria de Ripes puede ajustarse con los parámetros del cuadro de diálogo al que da paso la opción Edit→Settings.

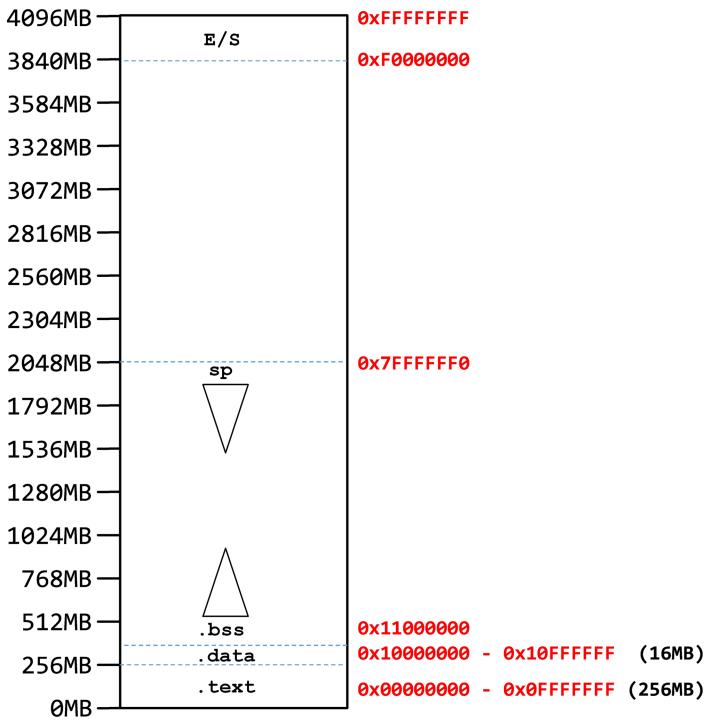


Figura 1.9. Mapa de memoria del emulador Ripes con las distintas secciones en que se divide y sus tamaños.

La sección `.text` contendrá el código del programa. En las secciones `.data` y `.bss` se alojan los datos inicializados y sin inicializar, respectivamente. El registro `sp` apunta a la parte superior de la pila y, a medida que se alojan datos en esta, crece hacia abajo. La zona marcada como **E/S** es el espacio reservado para la entrada/salida mapeada que trataremos después.

Leer y escribir datos en memoria

La lectura de un dato que está almacenado en memoria conlleva asignar la dirección de memoria a un registro y, a continuación, usarlo como puntero para leer el dato en sí mismo. Es una operación que se simplifica gracias a la seudoinstrucción[13] `lw rd, símbolo`, en la que `rd` sería el registro de destino (donde se guardará el valor) y `símbolo` la etiqueta dispuesta delante de dicho dato al declararlo.

13: Esta generará dos instrucciones reales, las puedes observar en el panel derecho de la vista Editor de Ripes, que analizaremos después.

La instrucción complementaria a **lw** es **sw**. Esta tiene por finalidad almacenar el contenido de un registro en la dirección de memoria que se especifique. A diferencia de **lw**, sin embargo, **sw** no es una seudoinstrucción sino una instrucción real. Para usarla emplearemos la sintaxis **sw** `ro, desp, rdir`, en la que `ro` es el registro cuyo valor quiere escribirse en memoria, `rdir` el registro que actuará como puntero y `desp` un desplazamiento a aplicar sobre la dirección contenida en `rdir`.

Antes de poder usar **sw**, por tanto, será preciso asignar al registro `rdir` la dirección de memoria que interese. Con este fin se usa la seudoinstrucción **la** `rdir, símbolo`. Como otras seudoinstrucciones, esta producirá dos instrucciones reales que aparecerán en el panel derecho. Serán estas las que obtengan la dirección que corresponda a `símbolo` y la almacene en el registro `rdir`.

Sobre la base de las instrucciones **lw**, **sw** y **la** vamos a escribir un programa que cargue dos valores de una zona de memoria inicializada, los sume y almacene el resultado en una zona de memoria sin inicializar.

Listado 1.2: Leer dos números de memoria y escribir su suma.

```
1        .data    # Inicio del segmento de datos
2
3    n1: .word 7    # Almacenamos en memoria
4    n2: .word 3     # dos valores enteros
5
6        .bss    # Inicio de datos sin inicializar
7
8    res: .word 0
9
10       .text    # Inicio del segmento de código
11
12   lw a1, n1        # a1 = (n1)
13   lw a2, n2        # a2 = (n2)
14   add a0, a1, a2   # a0 = a1 + a2
15   la t0, res       # t0 -> res
16   sw a0, 0, t0     # (res) = a0
```

CARGAR DOS NÚMEROS DESDE MEMORIA, SUMARLOS Y ALMACENAR EL RESULTADO

Las directivas `.data`, `.bss` y `.text` establecen el punto en el que comienzan las secciones de datos inicializados, datos sin inicializar y código, respectivamente. Cada una de estas secciones o segmentos tienen una dirección de inicio predeterminada. La directiva `.word` reserva espacio para un número entero de 32 bits.

Examinar la memoria en Ripes

Al ejecutar el anterior programa observaremos, en la lista de registros, que **a0** contiene el resultado de la suma y que **t0** ha cambiado su valor para apuntar a la dirección identificada por el símbolo **res**. Para observar el estado de la memoria, y verificar así los datos almacenados en ella, accederemos a la vista **Memory**. Esta se abre con el botón  y tiene el aspecto de la Figura 1.10.

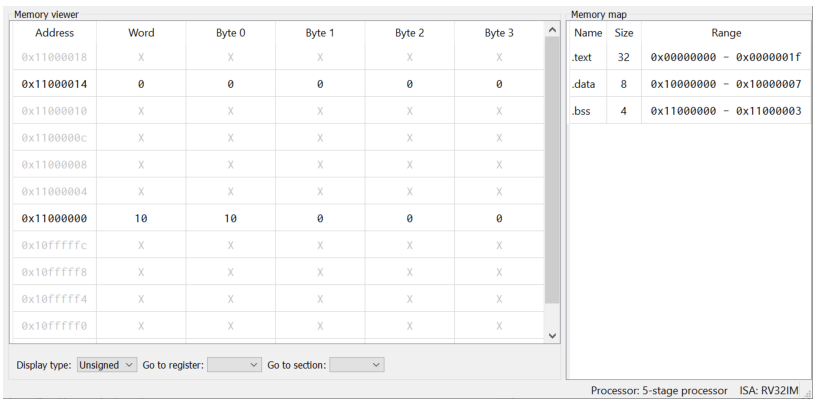


Figura 1.10. Vista de memoria del simulador Ripes. El panel de la izquierda muestra las direcciones de memoria y su contenido. El formato (hexadecimal, decimal, etc.) se elige con las opciones de la primera lista desplegable de la parte inferior. La última facilita el acceso a la sección de memoria que interese. En el panel derecho se facilita el mapa de memoria con la longitud real de las secciones de código y datos.

Dedica un tiempo a experimentar con esta vista, cambiando entre las secciones de memoria para examinar el código máquina del programa, los datos inicializados y sin inicializar en distintos formatos.

1.7. Modos de direccionamiento

La mayor parte de instrucciones ensamblador precisan uno o más operandos: **datos de entrada** sobre los que se efectuará una determinada operación y **el destino** donde quiere almacenarse el resultado. Según el número de dichos operandos y su tipo hablamos de tipos o modos de direccionamiento concretos.

Según la bibliografía consultada es posible encontrar denominaciones diferentes y más o menos modos de direccionamiento para RISC-V. La información aquí recogida se corresponde a la documentación oficial de la especificación para RV32I [2].

Los procesadores RISC suelen ofrecer menos modos de direccionamiento que los CISC, ya que muchas de las operaciones solo pueden efectuarse tras haber cargado los datos en registros. Los modos comunes en RISC-V son los indicados en los siguientes apartados, vinculados a los distintos tipos de instrucciones.

Formato general de las instrucciones RV32I

[4]: Blem et al. (2015), «ISA wars: Understanding the relevance of ISA being RISC or CISC to performance, power, and energy on modern architectures»

La longitud de las instrucciones, en formato de código máquina que es el que puede procesar la CPU, depende del diseño de la ISA [4]. En procesadores CISC es habitual que esa longitud sea variable, de manera que algunas instrucciones ocupan un solo byte, otras dos, tres ... hasta 15 bytes en ocasiones. En contraposición, las arquitecturas RISC promueven el uso de una longitud fija para las instrucciones, ya que esto simplifica de manera considerable la implementación de la ISA.

En RV32IM todas las instrucciones se almacenan en memoria ocupando 32 bits. Este es el código que aparece en la segunda columna del panel **Executable code** de la vista **Editor** de Ripes. Observa que siempre ocupa 8 dígitos hexadecimales y, dado que cada dígito corresponde a 4 bits, por tanto se tienen 32 bits.

14: Registro de destino, registros de origen, valor inmediato, etc.

Los 32 bits de una instrucción tienen que codificar no sola la operación que quiere llevarse a cabo, sino también sus operandos[14]. Según el tipo de instrucción esos 32 bits se estructuran de una forma u otra, dependiendo del número y tipo de operandos. Cada categoría de instrucción, de los cuatro básicos que se describen a continuación, está asociado a un modo de direccionamiento.

La denominación asignada a cada parte de la instrucción es la usada en la especificación oficial de RISC-V.

Instrucciones tipo R

Estas son instrucciones en las que todos los valores que actúan como operandos se encuentran alojados en registros, de ahí que se vincule con el denominado **direccionamiento por registro**. El formato de la instrucción, con los 32 bits de la misma divididos en varios grupos, es el mostrado en la Figura 1.11.

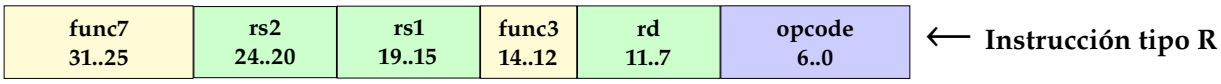


Figura 1.11. Formato de las instrucciones tipo R.

En el panel derecho de la vista **Editor** de Ripes, donde aparece el código máquina, puedes activar la opción **Binary** para observar dicho código en formato binario, lo cual facilita la observación de las distintas partes de la instrucción.

Los tres registros implicados, `rs1`, `rs2` y `rd`, aportan los operandos sobre los que se actuará y el destino en el que se almacenará el resultado, respectivamente. A cada uno le corresponden 5 bits ($2^5 = 32$) ya que hay un total de 32 registros. En todos los formatos de instrucción los registros siempre ocupan las mismas posiciones, lo cual facilita la descodificación por parte del microprocesador.

El código de operación, conocido como *opcode*, siempre ocupa los siete bits de menor peso de la instrucción. Hay grupos de instrucciones que tienen el mismo *opcode* y la operación que realizan viene determinada por un código de función de 10 bits. Estos son los que ocupan las posiciones `func3` y `func7`.

Una instrucción tipo R, y que por tanto emplea direccionamiento por registro, es por ejemplo **`add`** `a0, a1, a2`, así como cualquier otra de las operaciones aritméticas y lógicas. Las instrucciones **`add`**, **`sub`**, **`and`**, **`or`** y **`xor`**, entre otras, tienen asociado un mismo *opcode*, siendo la unión `func7+func3` la que determina la operación a ejecutar.

Instrucciones tipo I y U

Este conjunto de instrucciones está asociado al **direccionamiento inmediato**, caracterizado porque uno de los operandos es un valor inmediato en lugar de un registro, tal y como se aprecia en la Figura 1.12 y la Figura 1.13.

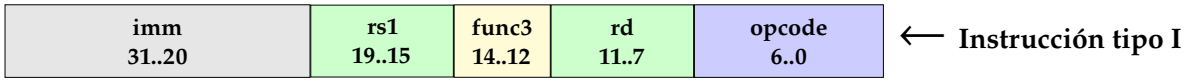


Figura 1.12. Formato de las instrucciones tipo I.

En el tipo I el valor inmediato ocupa los cinco bits que en el tipo R corresponden al `rs2` y también los 7 bits de `func7`, por lo que en total se cuenta con 12 bits para el valor inmediato. Ya conocemos una instrucción de este tipo: **`addi`** `a0, a1, 5`, en la que **`a0`** es el registro de destino (`rd`), **`a1`** el primer operando (`rs1`) y el valor literal **`5`** el segundo (`imm`). Hay varias más similares, todas ellas compartiendo un mismo *opcode*, como son **`andi`**, **`ori`** o **`xori`**.

En RISC-V los números negativos se almacenan en **complemento a 2**. Por tanto, para obtener el negativo de cualquier número se invierten todos sus bits y después se suma 1. Así, y considerando que tenemos 12 bits para representar el valor, el número 5 (**`0000 0000 0101`**) pasaría a ser -5 tras invertir todos los bits (**`1111 1111 1010`**) y a continuación sumar 1 (**`1111 1111 1011`**).

Hay que tener presente que, en general, RISC-V procesa los valores inmediatos como números enteros con signo. El bit de mayor peso denota si el número es positivo o negativo. Esto nos deja 11 bits para la magnitud (el valor absoluto), lo cual sería insuficiente para representar cantidades mayores que 2^{11}. En la práctica esto implica que el rango de valores posibles sería `[-2048, 2047]`.

También son instrucciones de tipo I aquellas que emplean el **direccionamiento indexado**, consistente en tomar el contenido de un registro y sumarle el desplazamiento indicado por el valor inmediato, para obtener así el contenido de la dirección de memoria resultante. Es el caso de la instrucción **`lw`** `a0, 128, t0`, en la que al contenido del registro **`t0`** se le suma el desplazamiento `128` y, a continuación, se lee de `[t0+128]` una palabra para llevarla a **`a0`**.

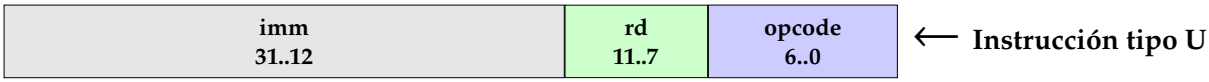


Figura 1.13. Formato de las instrucciones tipo U.

El tipo U se caracteriza porque el valor inmediato tiene una longitud de 20 bits, en lugar de 12, y porque el valor se interpreta como un entero sin signo. Además, solo se cuenta con un registro como operando (`rd`) que actúa como destino de la operación. Una instrucción de este tipo es **`lui`** `t0, valor`, en la que los 20 bits de `valor` se transfieren como los 20 bits de mayor peso[15] del registro **`t0`**, mientras que los 12 bits de menor peso se ponen a 0.

15: Esto equivale a desplazar 12 bits hacia la izquierda el valor o, lo que es lo mismo, multiplicarlo por 4096 (2^{12} = 4096).

La otra instrucción que emplea el formato U es **`auipc`** `rd, imm`. Esta permite usar un tipo de **direccionamiento relativo** en el que uno de los operandos es fijo: el contador de programa (registro **`pc`**). Es un tipo de direccionamiento que permite a los compiladores producir código que puede cargarse en cualquier posición de la memoria, ya que todas las referencias a datos y etiquetas son relativas al **`pc`**.

Instrucciones tipo S

Vinculado al conocido como **direccionamiento relativo**, este conjunto está formado por ciertas instrucciones de salto condicional. Se cuenta con dos operandos de entrada, alojados en sendos registros, sobre los cuales se llevará a cabo una operación según cuyo resultado se modificará el **pc** sumándole el valor inmediato de 12 bits que se obtiene al concatenar `imm1` e `imm2`.

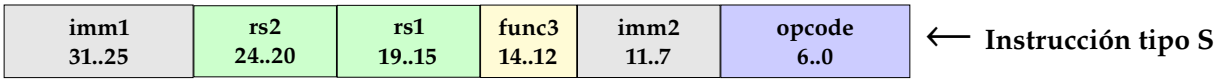


Figura 1.14. Formato de las instrucciones tipo S.

Conoceremos las instrucciones de salto en la siguiente práctica.

Una instrucción que emplea este direccionamiento es **beq** `a0, a2, imm`, que compara el contenido de los registros **a0** y **a2** y, si es igual, modifica el **pc** en la magnitud indicada por el valor inmediato. En consecuencia se produce un *salto* en el flujo de ejecución del programa.

Asignación de una dirección a un registro

Ahora que conocemos la configuración de memoria con la que trabajaremos y los modos de direccionamiento existentes, lo que nos interesa es saber cómo podemos asignar a un registro una cierta dirección de memoria. Completada esa operación, el registro servirá como puntero para acceder a esa dirección y otras adyacentes mediante la suma/resta de un desplazamiento.

lw es una seudoinstrucción cuando se usa solo con dos argumentos: el registro de destino y la dirección del dato a cargar. La instrucción real **lw** precisa tres: el registro de destino, un desplazamiento y un registro con la dirección base.

En el anterior Listado 1.2 hemos usado la seudoinstrucción **lw** para cargar nuestros datos. Esta produce dos instrucciones reales. Si vamos a cargar muchos datos desde memoria una de ellas será reiterativa, ya que la primera instrucción siempre cargará una dirección base en un registro y la segunda aplicará un desplazamiento sobre la misma.

El Listado 1.3 muestra cómo usar las instrucciones **lui** y **addi** para cargar en el registro **t0** —que pretendemos usar como si fuese un puntero— la dirección de 32 bits que corresponde a la etiqueta `par`. Descomponemos la dirección de dicha etiqueta en dos partes, mediante los operadores `%hi` y `%lo`, la primera con los 20 bits de mayor peso y la segunda con los 12 de menor peso. Al sumar ambas partes se compone la dirección que interesa. Esta nos sirve para leer los dos datos, y cualesquiera otros que pudieran existir, con la instrucción **lw**. Observa en el código cómo para leer el primer dato se usa un desplazamiento `0` y en el segundo `4`, dado que cada entero ocupa 4 bytes (32 bits).

Listado 1.3: Uso de una dirección base con distintos desplazamientos para cargar datos

```
1     .data
2
3   par: .word 7, 3 # Dos valores enteros en memoria
4
5         .text
6
7   lui t0, %hi(par)      # t0[31..12] = par[31..12]
8   addi t0, t0, %lo(par) # t0 = t0 + par[11..0]
9   lw a1, 0, t0    # a1 = 0(t0) = par[0]
10  lw a2, 4, t0    # a2 = 4(t0) = par[1]
11  add a0, a1, a2  # a0 = a1 + a2
```

1.8. Ejercicios propuestos

Operaciones con registros

Los ejercicios propuestos a continuación te permitirán familiarizarte con el lenguaje ensamblador de RISC-V y la herramienta Ripes, preparándote para la evaluación de esta primera práctica.

Ejercicio 1.1. Suma los valores inmediatos `3`, `5` y `7` usando solo el registro **`a0`** y muestra el resultado en la consola. En este y los siguientes tres ejercicios usa solo direccionamiento inmediato y por registro.

Ejercicio 1.2. Asigna a **`a0`** el valor `23`, réstale el valor `8` y muestra el resultado por la consola.

Ejercicio 1.3. Muestra por la consola el resultado de multiplicar los números `5` y `7`.

Ejercicio 1.4. Divide el valor **`23`** entre **`4`** y muestra por la consola tanto el cociente como el resto.

Carga y almacenamiento de datos

Ejercicio 1.5. Almacena en memoria tres números y cárgalos uno a uno sumándolos de manera acumulada en el registro **`a0`**. Este deberá ser inicializado a `0` al inicio del programa.

Ejercicio 1.6. Modifica el programa anterior para usar el direccionamiento indexado, de forma que **`t0`** actúe como puntero para leer los tres valores desde memoria.

Ejercicio 1.7. Actualiza el programa del ejercicio previo de forma que tras calcular la suma esta se almacene en memoria, en una dirección previamente reservada.

Ejercicio 1.8. Modifica el programa del **Ejercicio 1.4** para que el cociente y resto se guarden en dos posiciones de memoria consecutivas en lugar de mostrarse por la consola.

Valores de más de 12 bits

Ejercicio 1.9. Carga en el registro **`a0`** el valor `75000`, previamente almacenado en el segmento de datos, y envíalo a la consola.

Ejercicio 1.10. Asigna al registro **`a0`** el valor `75000` sin leerlo de memoria y envíalo a la consola.

Bucles y condicionales en RISC-V 2

2.1. Planificación

- **Objetivo.** Aprender a usar las instrucciones RISC-V para implementar bucles y condicionales. Comunicar la CPU con dispositivos externos a ella mediante operaciones de E/S mapeada.
- **Temporización:**
  - **Trabajo en laboratorio**: descripción de las instrucciones implicadas en la comparación y saltos. Uso de las mismas para realizar ejecución condicional y repetitiva (codificación de bucles). Uso de la pila e instrucciones para la implementación de subrutinas. Entrada/salida mapeada para comunicarse con dispositivos externos como una matriz de ledes. Resolución de dudas.
  - **Trabajo autónomo:** ejercicios de recorrer vectores de valores realizando algún tipo de operación con ellos. Implementación de diversas funciones. Lectura/Escritura de secuencias de valores en los dispositivos de E/S.
  - **Validación en laboratorio**: test y ejercicios evaluables.

2.2. Instrucciones de salto

Un programa que se ejecuta de manera secuencial, de principio a fin, ha de ser necesariamente simple y siempre realizará la misma tarea[1]. Para llevar a cabo funciones más complejas será preciso contar con herramientas de control del flujo de ejecución, a saber:

1: Cada vez que se ejecute procesará las mismas instrucciones aplicadas sobre los mismos datos.

- La posibilidad de ejecutar o dejar de ejecutar un cierto bloque de instrucciones según que se cumpla o no una determinada condición, de forma similar a las instrucciones `if (cond) cierto else falso` de los lenguajes de programación de alto nivel.
- Tener la capacidad de repetir la ejecución de un bloque de instrucciones las veces que se precise, como lo hacen instrucciones del tipo `for` o `while` en lenguajes de alto nivel.

En ambos casos se depende de un mecanismo fundamental en la programación de ordenadores: la alteración del **flujo de ejecución**. Esto implica cambiar de alguna forma el contenido del **contador de programa** para que no se limite a ejecutar una instrucción tras otra en cada ciclo de su funcionamiento que, como sabemos, es el comportamiento por defecto.

En RISC-V el contador de programa es el registro `pc`, cuyo valor se actualiza de forma automática a medida que se ejecutan las instrucciones de un programa.

Aquí entran en escena las conocidas como instrucciones de salto. Estas modifican el contenido del `pc` asignándole un valor que, por regla general, estará simbolizado por una etiqueta dispuesta en el código fuente del programa. Esto nos evitará tener que calcular manualmente las direcciones de las distintas partes de nuestro programa.

Cambiar el contenido del `pc` conlleva ejecutar las instrucciones dispuestas en memoria a partir de la dirección asignada, en lugar de la instrucción siguiente como correspondería. Antes de realizar ese cambio es posible guardar el actual contenido del `pc` en otro registro, por regla general con el objetivo de volver más tarde a ese mismo punto.

De las instrucciones enumeradas en la Tabla 1.2[2] hay un grupo, en total ocho de ellas, a las que habría que sumar varias seudoinstrucciones más con fines similares. Todas ellas se dividen en dos subgrupos: las de salto condicional y las incondicionales. Según este criterio se detallan en los siguientes apartados.

2: Dicha tabla, correspondiente a la primera sesión de prácticas, la encontrarás en la página 2 de este cuaderno.

Saltos condicionales

Esta categoría de instrucciones se caracterizan por llevar a cabo una operación aritmético-lógica, de comparación entre dos valores, según cuyo resultado se alterará o no el contador de programa. Las instrucciones disponibles son:

El valor inmediato que todas estas instrucciones precisan como tercer argumento es un desplazamiento, positivo o negativo, que se sumará al contador de programa, de forma que si la condición se cumple se ejecuta la operación **pc**←**pc** + `imm`. Todas estas instrucciones son del tipo S (véase la Figura 1.14 de la primera práctica) lo cual implica que el desplazamiento está limitado a 12 bits.

- **beq** `rs1, rs2, imm`: salta si `rs1` = `rs2`.
- **bne** `rs1, rs2, imm`: salta si `rs1` ≠ `rs2`.
- **bge** `rs1, rs2, imm`: salta si `rs1` ≥ `rs2`.
- **bgeu** `rs1, rs2, imm`: como la anterior pero interpretando los valores sin signo.
- **blt** `rs1, rs2, imm`: salta si `rs1` < `rs2`.
- **bltu** `rs1, rs2, imm`: como la anterior pero interpretando los valores sin signo.

La igualdad o desigualdad de los dos valores, alojados en sendos registros, es indistinta de si estos se interpretan como números con o sin signo. Al comparar si uno es menor o mayor que el otro, sin embargo, sí hay que tomar en consideración la existencia de signo, de ahí que haya disponibles dos versiones.

Implementación de una estructura condicional

Gracias a estas instrucciones resulta posible implementar condicionales tal y como se aprecia en el Listado 2.1, en el que se ejecuta una instrucción u otra según que el valor en **a2** sea mayor que el de **a1** o no. En la versión en ensamblador faltaría situar la etiqueta `fin`, desde la que continuaría la ejecución del resto del código del programa.

Listado 2.1: Un condicional escrito en ensamblador RISC-V (izqda.) y en lenguaje C.

```
    bge a2, a1, mayora2
    sub a0, a1, a2
    j fin
mayora2:
    sub a0, a2, a1
```

```
    if (a1 > a2)
        a0 = a1 - a2;
    else
        a0 = a2 - a1;
```

Sobre la base de las anteriores instrucciones, RISC-V define múltiples seudoinstrucciones que facilitan saltos con otras condiciones, como **bgt**, **bgtu**, **ble** y **bleu**, que toman los mismos argumentos que las instrucciones previas, y **beqz**, **bnez**, **bgez**, **blez** y **bgtz**, en las que el segundo argumento es siempre el registro **zero**. Introduce estas seudoinstrucciones en el editor de Ripes y observa en el panel derecho el código real que generan.

Implementación de un bucle

De manera análoga podríamos crear una estructura de repetición o bucle, para lo cual nos serviríamos de un salto condicional que volvería al inicio del cuerpo del bucle hasta que deje de cumplirse una cierta relación que, en el caso del Listado 2.2, es que el contenido de **a0** no sea cero.

Listado 2.2: Un bucle escrito en ensamblador RISC-V (izqda.) y en lenguaje C.

```
    li a0, 10
    li a7, 1
repite:
    ecall
    addi a0, a0, -1
    bnez a0, repite
```

```
    for(int a0=10; a0 >= 0; a0--)
        std::cout << a0;
```

La seudoinstrucción **bnez** compara el contenido de **a0** con el registro **zero** y, si no coinciden, ejecuta el salto, de forma que el bucle se repetirá hasta que **a0** contenga el valor `0`.

Saltos incondicionales

Como su propia denominación indica, estas instrucciones cambian el valor del contador de programa siempre que se ejecutan, sin que exista condición previa alguna. Hay dos instrucciones de salto incondicional disponibles:

- **`jal`** `rd, imm`: es una instrucción de tipo U que cuenta con dos operandos. En `rd` se guardará el actual contenido del **`pc`**, tras lo cual este se actualizará sumándole el desplazamiento de 20 bits entregado como segundo argumento.

La estructura de las instrucciones tipo U es la mostrada en la Figura 1.13 (pág. 11) y cuenta con un registro de destino y un valor inmediato de 20 bits.

- **`jalr`** `rd, rs, imm`: es una instrucción de tipo I que cuenta con tres operandos. En `rd` se guardará el actual contenido del **`pc`**, tras lo cual se actualizará asignándole el resultado de la suma **`rs + imm`**. En la práctica esto permite saltar a cualquier punto de un programa, sin las limitaciones de los desplazamientos de 12 o 20 bits respecto a la posición actual del **`pc`**.

La estructura de las instrucción tipo I es la mostrada en la Figura 1.12 (pág. 11) y cuenta con un registro de destino, uno de origen y un valor inmediato de 12 bits.

Como primer parámetro de estas instrucciones suele emplearse el registro **`ra`** (seudónimo de **`x1`**). La seudoinstrucción **`jal`** `imm` produce el código **`jal`** `x1, imm`, de forma que no es preciso indicar de manera explícita el registro de destino. También es posible saltar de manera incondicional a un punto del programa con la seudoinstrucción **`j`** `imm`, caso en el que se genera el código **`jal`** `x0, imm` y, por tanto, el valor actual en el **`pc`** es descartado.

La diferencia fundamental entre **`jal`** y **`jalr`** estriba en que esta última no suma el desplazamiento al **`pc`**, sino que asigna a este el resultado de `rs+imm`. Dado que antes se puede asignar a `rs` cualquier valor[3], esta instrucción hace posible la transferencia de la ejecución a cualquier punto del espacio de direccionamiento, sin las limitaciones de 12 o 20 bits que imponen los desplazamientos relativos.

3: Lo habitual es emplear instrucciones como **`lui`** y **`auipc`**, que ya conocemos, con este fin.

Llamadas a subrutinas

El hecho de que ambas instrucciones de salto incondicional, **`jal`** y **`jalr`**, almacenen el actual valor del **`pc`** en un registro antes de cambiarlo, hace posible el retorno a ese mismo punto[4]. No hay más que usar la instrucción **`jalr`** y cambiar el registro de destino por el de origen, usando el valor `0` como desplazamiento para que la operación real sea **`pc`** ← `rs`.

4: En realidad se toma el valor del **`pc`**, se le suma `4` y a continuación se guarda en el registro de destino, de forma que esa dirección apunta a la instrucción siguiente a la actual.

Las seudoinstrucciones **`call`** y **`ret`** facilitan este tipo de operativa. La primera prepara en el registro **`ra`** la dirección base, que será la del **`pc`**, y salta al punto indicado. La segunda toma de **`ra`** la dirección de retorno, enviando a **`x0`** el actual valor del **`pc`**.

Escritura y llamada a una subrutina

El Listado 2.3 es un programa en el que se ha escrito una subrutina, a partir de la etiqueta `print`, a la que se llama múltiples veces desde el interior de un bucle. Se usan las llamadas del sistema ofrecidas por Ripes para mostrar cadenas de caracteres y números tanto en decimal como en hexadecimal.

```
1     .data
2  dec: .string "Decimal: "
3  hex: .string "Hexadecimal: "
4  sep: .string " - "
5  endl: .string "\n"
6
7     .text
8  inicio:
9     li a1, 9
10    li a2, 20
11
12 bucle:
13    call print
14    addi a1, a1, 1
15    ble a1, a2, bucle
16
17    li a7, 10 # Terminar
18    ecall
19
20 print:
21    la a0, dec
22    li a7, 4
23    ecall
24    mv a0, a1
25    li a7, 1
26    ecall
27    la a0, sep
28    li a7, 4
29    ecall
30    la a0, hex
31    ecall
32    mv a0, a1
33    li a7, 34
34    ecall
35    la a0, endl
36    li a7 4
37    ecall
38    ret
```

Listado 2.3: Programa con una rutina de impresión de números en decimal y hexadecimal.

Prueba el programa en Ripes para observar la salida que produce. Cambia los límites del bucle y experimenta con otras alternativas de impresión.

Dado que la seudoinstrucción **`call`** siempre almacena la dirección de retorno en el registro **`ra`**, de donde la toma la seudoinstrucción **`ret`** para volver a asignarla al **`pc`**, no es posible usar estas seudoinstrucciones para tener subrutinas anidadas.

2.3. Dispositivos de E/S

Una CPU por sí sola no compone un computador. Este precisa memoria, algo que ya sabemos cómo usar en Ripes con las cargas y almacenamientos, y también comunicarse con dispositivos de entrada y salida [5], mediante los cuales el computador interacciona con el usuario.

[5]: Wakerly (1977), «Microprocessor input/output Architecture»

El botón  , en la barra lateral izquierda de Ripes, da paso al panel de la Figura 2.1. En este caso se han agregado al área central tres dispositivos, uno de cada tipo, y la matriz de ledes muestra el resultado de la ejecución de un programa que ilumina la diagonal principal de rojo y la inversa de color verde. Las dimensiones de dicha matriz se han ajustado previamente, no tiene el tamaño asignado por defecto por Ripes a dicho dispositivo.

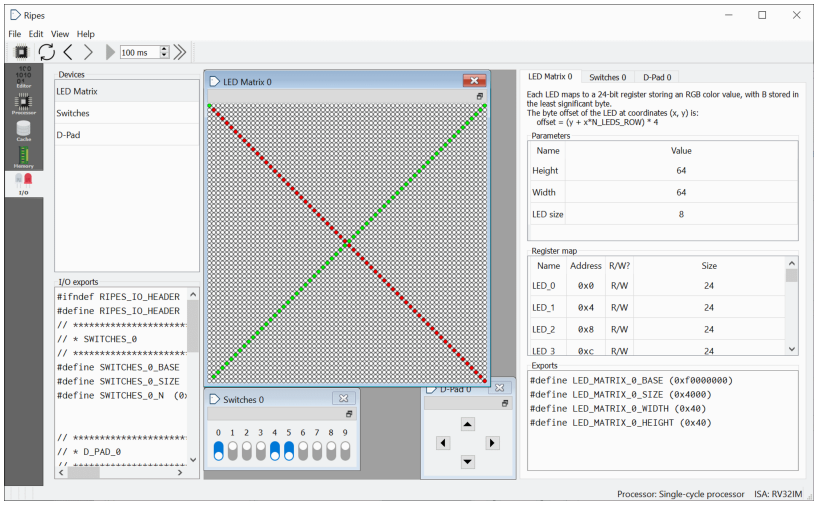


Figura 2.1. Panel con los dispositivos de entrada/salida que nos ofrece Ripes. Este se divide en tres subpaneles. El apartado **Devices** sirve para agregar al área central los dispositivos con los que queramos contar. Hay tres categorías de dispositivos: una matriz de ledes, un grupo de ocho interruptores y un pad con cuatro botones. El subpanel derecho facilita la configuración del dispositivo seleccionado en la parte central.

Configuración de un dispositivo

Ripes permite crear varios dispositivos del mismo tipo, de forma que sería posible contar con varias matrices de ledes para mostrar distintas informaciones. Solo hay que repetir el paso 2 las veces que necesitemos.

Veamos cómo configurar un dispositivo concreto para utilizarlo desde un programa propio. Tras iniciar Ripes reproduce los siguientes pasos:

1. Abre el panel **I/O** con el botón correspondiente. El área central aparece vacía en principio.
2. Haz doble clic sobre el elemento **LED Matrix** del apartado **Devices**. En la zona central aparece una matriz de ledes con una configuración por defecto.
3. En la parte derecha se indica el alto (número de filas) y ancho (número de columnas) de la matriz. Haz clic sobre los valores y cámbialos para tener una pantalla de 64×64 ledes.
4. La ventana que muestra la matriz de ledes tiene un botón, debajo del de cierre, que sirve para fijar la vista del dispositivo. Haz clic sobre él, de forma que la matriz esté siempre visible aunque volvamos al editor de código.

La información sobre los valores de cada dispositivo se facilitan en el panel derecho. Ahí apreciaremos, por ejemplo, cómo determinar la dirección de memoria de un cierto píxel de la matriz de ledes y cómo se interpretan los valores que estos pueden tomar: 24 bits tratados como tres bytes que denotan intensidad de rojo, verde y azul.

De manera análoga configuraríamos otros dispositivos, como el número de interruptores que queremos tener en un dispositivo **Switches**. Por cada uno de ellos se definirá un conjunto de constantes, las vemos en la parte inferior derecha, que llevaremos a nuestro programa para saber, por ejemplo, cuantas filas y columnas tiene nuestra matriz de ledes o cuál es la dirección de memoria a partir de la cual están los valores que determinan su color.

Entrada/salida mapeada

Una vez que conocemos los dispositivos con que cuenta nuestro sistema computador, lo que incluye saber en qué espacio de direcciones están mapeadas las entradas y salidas correspondientes, para acceder desde un programa a un dispositivo no precisamos más que las instrucciones de carga y almacenamiento que conocimos en la primera práctica. Lo primero que necesitamos, por tanto, son las constantes que la propia herramienta Ripes nos facilita en la página **I/O**. En el caso de la matriz de ledes, y asumimos que la configuración se ha establecido de acuerdo a lo indicado en el apartado previo, dichas constantes serían las mostradas en el Listado 2.4.

Según la naturaleza de un dispositivo, en su espacio de E/S se contemplarán solo operaciones de lectura (R), solo escrituras (W) o ambas (R/W).

```
#define LED_MATRIX_0_BASE (0xf0000000)
#define LED_MATRIX_0_SIZE (0x4000)
#define LED_MATRIX_0_WIDTH (0x40)
#define LED_MATRIX_0_HEIGHT (0x40)
```

Listado 2.4: Constantes asociadas al dispositivo matriz de ledes.

No es necesario copiar estas constantes en nuestro código. El programa las leerá directamente de la página **I/O**.

La constante `LED_MATRIX_0_BASE` corresponde a la dirección de memoria donde comienza el espacio de E/S de este dispositivo. Cada led ocupa 4 bytes, de los cuales 3 identifican la intensidad RGB. Esto significa que si escribimos en esa dirección inicial el valor `0xff0000` haremos que el primer led —el de la esquina superior izquierda— se ilumine con color rojo. De manera análoga, al escribir en la dirección `LED_MATRIX_0_BASE + 4` cambiaríamos el color del segundo led, el que está a la derecha del anterior.

Generar patrón con ledes de colores

El programa del Listado 2.5 demuestra cómo escribir en la matriz de ledes. El objetivo es cambiar el color de todos ellos y formar un patrón como el que se aprecia en la Figura 2.2. Almacenamos en **`t0`** la dirección del primer led. En **`t1`** calculamos cuál sería la del último. En el interior del bucle escribimos en la posición **`t0`** el contenido de **`a0`**, que va cambiando en cada iteración, tras lo cual avanzamos al siguiente led hasta llegar al último.

```
li t0, LED_MATRIX_0_BASE
li t1, LED_MATRIX_0_SIZE
add t1, t1, t0
li a0, 0

bucle:

    sw a0, 0, t0
    addi a0, a0, 0x7ff
    addi t0, t0, 4

    bgt t1, t0, bucle
```

Listado 2.5: Llenar la matriz de ledes con diferentes colores.

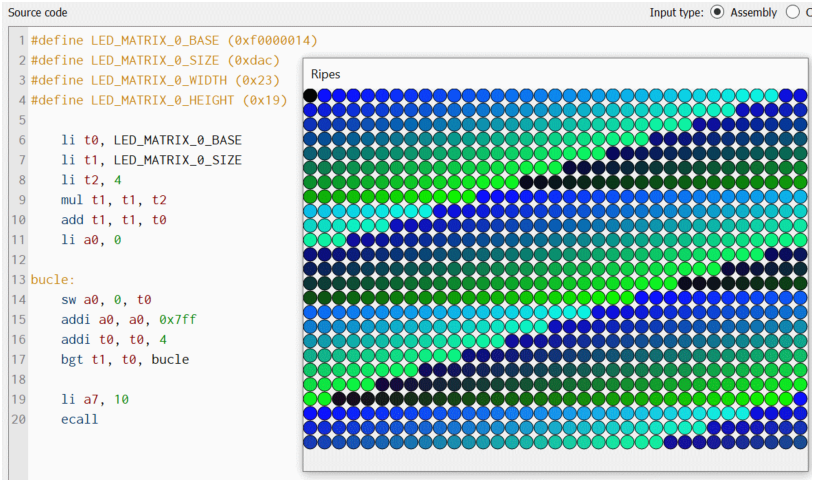


Figura 2.2. El programa y el resultado que produce en la matriz de ledes.

En lugar de incrementar **`t0`** de `4` en `4`, con lo que se consigue avanzar al siguiente led, podríamos sumar cualquier otro valor. Con `LED_MATRIX_0_WIDTH * 4`, por ejemplo, se saltaría hasta la misma posición horizontal de la siguiente fila. Solo se precisan algunos cálculos para dibujar líneas rectas, verticales u horizontales y, a partir de ellas, cuadrados, rectángulos, etc.

2.4. Almacenamiento temporal de datos en la pila

El número de registros con que cuenta cualquier CPU es limitado, de ahí que sea preciso recurrir a la memoria para almacenar los datos. Hay ocasiones en que esa necesidad es temporal: se precisa guardar en memoria el contenido de algunos registros para recuperarlos con posterioridad. Esto es algo habitual cuando, por ejemplo, se invoca a una función con un conjunto de parámetros. En estos casos se recurre a la pila, cuya situación en el mapa de memoria se representó en la Figura 1.9.

El registro **sp** contiene siempre la dirección de la parte alta de la pila —la de la primera posición libre— en una estructura que crece en memoria hacia abajo a medida que en ella se guardan datos.

Cuando se escribe una función en un lenguaje de alto nivel, como podría ser la del Listado 2.6, el compilador se ocupa de generar en cada llamada a la misma lo que se conoce como un *stack frame*. En cada llamada a `factorial()` hay que guardar en la pila la lista de parámetros recibidos, en este caso solo el valor entero `n`, y la dirección de retorno de la llamada. En una implementación recursiva, como esta que se propone, esa operación se efectuará múltiples veces hasta llegar al caso de parada, momento en el que van extrayéndose de la pila y devolviendo a los registros sus valores originales.

Listado 2.6: Función factorial escrita en lenguaje C que usa recursividad.

```c
int factorial(int n) {
    if (n == 1)
        return n;
    else
        return n * factorial(n -1);
}
```

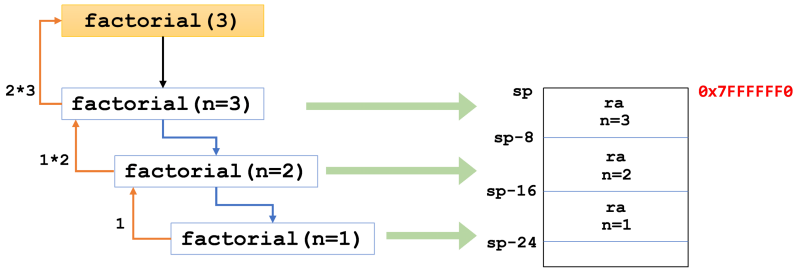


Figura 2.3. Evolución del contenido de la pila durante las llamadas recursivas a la función. En la primera llamada se guarda en la parte alta de la pila la dirección de retorno —registro **ra**— y el valor actual del parámetro de entrada. Esta operación se repite en cada una de las llamadas, de forma que en la pila van conservándose los datos correspondiente al ámbito de cada ejecución de la función. Al llegar al caso base se recuperarían de la pila los datos en sentido inverso, de forma que se devuelve el valor que en ese momento tenía el parámetro y se restablece en **ra** la dirección de retorno.

Implementar una función recursiva

Un compilador de C produce el código ensamblador preciso para ir ajustando el registro **sp**, almacenar y extraer datos de la pila. Ese es un código que tendremos que escribir de forma manual para conseguir la mencionada funcionalidad. El Listado 2.7 muestra cómo implementar la función anterior en RISC-V.

Listado 2.7: Código RISC-V que implementa la función factorial de forma recursiva.

```
# Entrada: a0 -> Número del que se quiere calcular el factorial
# Salida: a1 -> Resultado
factorial:
    addi sp, sp, -8 # Reservar 8 bytes de espacio en la pila
    sw ra, 0(sp)    # Para guardar la dirección de retorno
    sw a0, 4(sp)    # y el actual valor de a0

    addi a0, a0, -1
    bnez a0, recur
    li a1, 1          # return 1 if a0 = 0
    j finrecur        # else a0 * factorial(a0-1)

recur:
    call factorial    # Llamada recursiva

finrecur:
    lw a0, 4(sp)      # Recuperar el anterior valor de a0
    mul a1, a1, a0    # y multiplicar a1 por a0
    lw ra, 0(sp)      # Recuperar dirección de retorno
    addi sp, sp, 8    # Devolver espacio en la pila
    ret    # Retornar
```

Introduce el código de esta función en Ripes. Delante del mismo tendrás que inicializar el registro **a0** con el valor que quieras, llamar a la función con **call** y, por último, mostrar por consola el contenido de **a1**. Hay que evitar que el flujo de ejecución entre en la función sin haberla invocado con **call**, caso en el que la pila no contendría un valor de retorno válido. Para ello usa la instrucción **ecall** con la función adecuada para finalizar el programa.

Las tres primeras líneas de código son las encargadas de crear el *stack frame* y guardar en la pila la dirección de retorno y valor actual del parámetro de entrada. Después se actualiza ese parámetro y, con un condicional, se determina si se ha alcanzado el caso base. De no ser así, se produce una nueva llamada a la misma función. Al llegar al caso base se recupera de la pila el parámetro, se realiza la operación que corresponde —en este caso una multiplicación— y, por último, se extrae la dirección de retorno y se reajusta la pila.

2.5. Otras instrucciones de la arquitectura RV32IM

El conjunto de instrucciones que conocemos hasta el momento —aritméticas básicas, carga y almacenamiento en memoria, saltos condicionales y saltos incondicionales— son suficientes para escribir la gran mayoría de programas. Sin embargo, esto no significa que contar con instrucciones adicionales no pueda facilitarnos la tarea en ciertas ocasiones. En este apartado se enumeran buena parte de ellas.

Instrucciones lógicas

Las operaciones lógicas entre bits son de uso muy habitual al programar a bajo nivel, ya que con ellas es posible activar un bit concreto, desactivarlo, invertirlo, comprobar si está activo o no, etc. Las instrucciones disponibles para ello son las siguientes:

- **`and`** `rd, rs1, rs2` y **`andi`** `rd, rs1, imm`: efectúa un *AND* bit a bit entre los operandos `rs1` y `rs2` —la primera— o bien `rs1` e `imm` —la segunda—, dejando el resultado en `rd`.
- **`or`** `rd, rs1, rs2` y **`ori`** `rd, rs1, imm`: efectúa un *OR* bit a bit entre los dos operandos y guarda el resultado en `rd`.
- **`xor`** `rd, rs1, rs2` y **`xori`** `rd, rs1, imm`: efectúa un *XOR* bit a bit entre los dos operandos y guarda el resultado en `rd`.

¿Cómo comprobarías si el valor almacenado en un registro es par o impar? Bastaría con realizar una operación lógica *AND* con el valor adecuado. Todos los números impares tienen el bit de menor peso a 1, mientras que en los pares está a 0. Con **`andi`** `t0, a0, 1` tendríamos en **`t0`** el valor 0 si el número es par o 1 en caso contrario, por lo que con un salto condicional **`beqz`** podríamos discernir entre ambas posibilidades.

A estas hay que añadir un grupo de instrucciones que también operan al nivel de bit, pero en este caso con el objetivo de producir desplazamientos en un sentido u otro:

- **`sll`** `rd, rs1, rs2` y **`slli`** `rd, rs1, imm`: desplaza el valor contenido en `rs1` hacia la izquierda, tantos bits como indique `rs2` —la primera— o `imm` —la segunda—, dejando el resultado en `rd`.
- **`srl`** `rd, rs1, rs2` y **`srli`** `rd, rs1, imm`: desplaza el valor contenido en `rs1` hacia la derecha, tantos bits como indique `rs2` o `imm`, y guarda el resultado en `rd`.
- **`sra`** `rd, rs1, rs2` y **`srai`** `rd, rs1, imm`: desplaza el valor contenido en `rs1` hacia la derecha, tantos bits como indique `rs2` o `imm`, tomando en consideración el bit de signo. El resultado se almacena en `rd`.

Las instrucciones **`sll`** y **`srl`** llevan a cabo un desplazamiento **lógico** de los bits, tratándolos a todos por igual, mientras que **`sra`** trata de forma especial el bit de signo por lo que se dice que efectúa un **desplazamiento aritmético** en lugar de lógico.

El desplazamiento de un valor en un bit a la izquierda equivale a multiplicar dicho valor por dos. De forma análoga, desplazarlo a la derecha resulta en una división por dos. Cambiando el número de bits es posible multiplicar/dividir por cualquier potencia de dos sin necesidad de recurrir a las instrucciones **`mul`** y **`div`**. Esto resulta interesante porque el desplazamiento de bits es mucho más eficiente.

Instrucciones de comparación

En el apartado Saltos condicionales (pág. 16) hemos conocido instrucciones que verifican si se cumple una cierta relación entre dos registros y, en consecuencia, alteran el flujo de ejecución[5]. Además de esas, en RV32IM disponemos de cuatro instrucciones de comparación adicionales. La diferencia es que estas no alteran el valor del **`pc`**, sino que se limitan a almacenar un 1 o un 0 en un registro de destino según que el primer operando sea menor que el segundo o no, respectivamente.

5: Muchas arquitecturas de procesador no disponen de instrucciones así, sino que es preciso primero comparar dos operandos y obtener un señalizador, un valor que suele ser 0 o 1, en base al cual se decide con una segunda instrucción si se salta o no.

Realmente es una sola instrucción con cuatro variantes diferentes: **`slt`** `rd, rs1, rs2`, **`sltu`** `rd, rs1, rs2`, **`slti`** `rd, rs1, imm` y **`sltiu`** `rd, rs1, imm`. En todos los casos se comprueba si el `rs1` es menor que `rs2` o el valor inmediato `imm`. Las dos primeras operan con números con signo, mientras que las dos últimas lo hacen con números sin signo.

Estas instrucciones son útiles cuando el resultado de la comparación no necesariamente implica cambiar el flujo de ejecución. Imaginemos que tenemos una lista de calificaciones y, recorriéndolas con un bucle, queremos contar cuántas de ellas son inferiores a 5.

Contar número de valores menores a uno dado

En un práctica posterior, dedicada a los riesgos de control en la ejecución en un cauce segmentado, apreciaremos la importancia de evitar las instrucciones de salto siempre que sea posible.

Podríamos hacerlo con instrucciones de salto, para comparar si el valor cargado desde memoria es inferior a 5 y, en consecuencia, desviar la ejecución a una instrucción que incremente un contador. La alternativa sería la mostrada en el Listado 2.8, en la que no existe dicho salto condicional.

Listado 2.8: Contar el número de valores que son inferiores a uno dado.

```
1     .data
2  notas: .word 7, 5, 6, 3, 8, 2, 4, 6, 4, 9
3
4     .text
5     la t0, notas
6     li t1, 10
7     li a0, 0
8
9  bucle:
10
11    lw a2, 0, t0
12    slti a1, a2, 5
13    add a0, a0, a1
14    addi t1, t1, -1
15    addi t0, t0, 4
16    bnez t1, bucle
```

Otras seudoinstrucciones

En esta y la práctica anterior hemos conocido la mayoría de seudoinstrucciones contempladas en la ISA RV32IM. En realidad solo hay tres más de uso relativamente común:

- **not** `rd, rs`: calcula el complemento a uno[6] de `rs` y lo almacena en `rd`.
- **neg** `rd, rs`: calcula el complemento a dos[7] de `rs` y lo almacena en `rd`.
- **nop**: no realiza ningún trabajo útil, simplemente consume uno o más ciclos de reloj.

6: Recuerda que el complemento a uno de un valor se obtiene invirtiendo el estado de todos sus bits.
7: El complemento a dos se genera como el complemento a uno más 1.

2.6. Ejercicios propuestos

Los ejercicios propuestos a continuación te permitirán familiarizarte con las instrucciones condicionales y de salto, poniéndolas en práctica en diversos casos prácticos y preparándote para la evaluación de esta segunda práctica.

Codificar bucles

Ejercicio 2.1. Sumar los valores 1 a 1000, acumulando la suma en el registro **a0**, y mostrar el resultado por la consola.

Ejercicio 2.2. Sumar los valores impares entre 1 y 1000 en el registro **a0** y mostrar el resultado por la consola.

Ejercicio 2.3. Mostrar por la consola los números 40 a 122 y el carácter que corresponde a cada uno de esos códigos para conseguir un resultado similar al de la imagen inferior. Consulta las llamadas del sistema de Ripes para saber cómo imprimir el carácter.

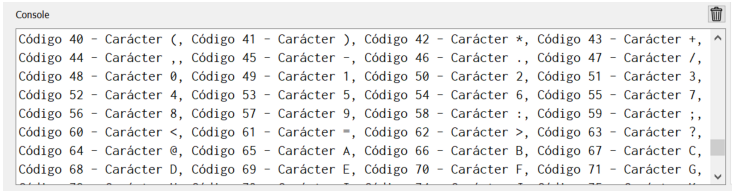


Figura 2.4. Vista parcial de la tabla de códigos y caracteres generada por el programa.

Ejercicio 2.4. Imprime por la consola las tablas de multiplicar de los números a 1 a 10 con un aspecto similar al de la imagen inferior.

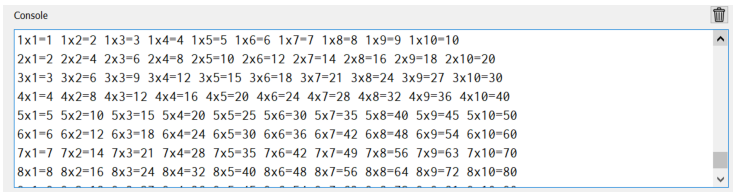


Figura 2.5. Vista parcial de las tablas de multiplicar mostradas por el programa.

Ejercicio 2.5. Almacenar en memoria un vector con los valores `7, 4, 23, 12, 6, 20, 17, 8, 3, 10` y mostrar por consola la media aritmética. Para ello se ha de calcular su suma y también contar cuántos valores hay en el vector.

Bucles y condicionales

Ejercicio 2.6. Almacenar en memoria un vector con los valores `7, 4, 23, 12, 6, 20, 17, 8, 3, 10` y escribir un programa que muestre por consola la suma de aquellos que ocupan posiciones impares, asumiendo que al primer `7` le corresponde la posición 1 (impar).

Ejercicio 2.7. Modifica el programa del ejercicio previo para que sume todos aquellos valores del vector que sean números impares, independientemente de la posición que ocupen en el vector, y muestre el resultado por la consola.

Ejercicio 2.8. Usando el mismo vector del ejercicio 2.6, escribe un programa para buscar el menor y el mayor valor almacenándolos en **`a0`** y **`a1`**, respectivamente. Muestra ambos valores por la consola.

Ejercicio 2.9. Almacenar en memoria el vector de valores `1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16`. Escribir un programa que lo trate como una matriz de 4×4, como en el listado de la derecha, y calcule la suma de cada una de las filas y las muestre por consola

```
.word  1,  2,  3,  4
.word  5,  6,  7,  8
.word  9, 10, 11, 12
.word 13, 14, 15, 16
```

Ejercicio 2.10. Modificar el programa anterior para que calcule y muestre la suma de cada una de las columnas de la matriz.

Ejercicio 2.11. Almacenar en memoria el vector de valores `6, 1, 6, 7, 2, 3, 6, -3, 6, 1, -2, 3, -5, 2, 6`. Escribir un programa que cargue el primer valor del vector, cuente cuántas veces aparece en él y muestre el conteo por consola.

Ejercicio 2.12. Usa el mismo vector del ejercicio previo pero cambia el programa para que cuente cuántos valores negativos hay y lo muestre por consola.

Implementar funciones

Ejercicio 2.13. Usa el vector de valores del ejercicio 2.9 interpretado como una matriz de 4×4. Escribe una función que tome como entrada una fila en **`a1`** y una columna en **`a2`** y devuelva en **`a0`** el valor contenido en esa posición de la matriz. La función solo podrá alterar el contenido de los registros temporales **`t0`**-**`t6`**. Llama a la función para mostrar por consola los valores en las posiciones `(0,0)`, `(1,1)`, `(2,2)` y `(3,3)`.

Ejercicio 2.14. Completa el programa anterior con una función que tome como parámetros la fila en **`a1`**, la columna en **`a2`**, el valor en **`a0`** y muestre por consola una mensaje del tipo `El valor en la fila a1 columna a2 es a0`. Úsala para mostrar los cuatro elementos antes indicados. Como en el caso anterior, la función solo deberá alterar registros temporales. Desde el programa llama primero a la función que obtiene el valor en **`a0`** y posteriormente a esta para mostrarlo en la consola.

Ejercicio 2.15. Escribe una función que tome como entrada —en el registro **`a0`**— un valor numérico y que muestre por consola la secuencia desde 0 hasta **`a0`**. Para ello la función operará de forma recursiva, de manera que ella misma se llamará con **`a0`** `-1` una vez tras otra hasta alcanzar el caso base: que **`a0`** sea igual a `0`. Utiliza la pila para guardar tanto la dirección de retorno como los valores que tomará **`a0`** en cada llamada. Por último, llama a dicha función desde la parte inicial del programa probando con distintos valores.

Ejercicio 2.16. El máximo común divisor (MCD) de dos números se calcula de forma muy sencilla mediante un proceso recursivo como el que muestra el programa en lenguaje C del margen derecho. Escribe una implementación análoga de forma que los registros **`a1`** y **`a2`** contengan los dos números de entrada y **`a0`** el MCD encontrado que, en último término, sería 1 si ambos números no comparten ningún otro factor. Llama desde el cuerpo del programa a la función con distintos valores para verificar su funcionamiento.

```c
int mcd(int a1, int a2) {
    int a = a2 % a1, b = a1;

    if (a == 0)
        return b;
    else
        return mcd(a, b);
}
```

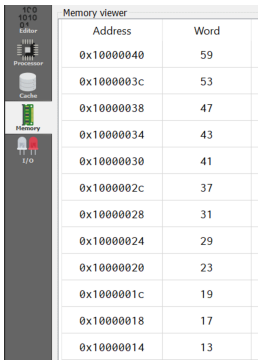


Figura 2.6. Estado de la memoria tras ejecutar el programa.

Ejercicio 2.17. Escribe un programa en el que inicialmente se tiene en memoria el vector `primos: .word 2, 3, 5, 7, 11, 13` con los seis primeros números primos. El programa deberá completar la tabla, agregando los primos adicionales que encuentre hasta llegar al 100. Para ello escribe una función que tomando como entrada un número —en el registro **a1**— y el número de primos que se tienen actualmente en el vector —en el registro **a2**— verifique si dicho número es primo o no. Para ello no hay más que comprobar si el número en **a1** es divisible por cualquiera de los valores existentes en el vector. La función devolverá el valor `1` en el registro **a0** si el número es primo o `0` en caso contrario. Llama a la función desde un bucle en el cuerpo principal del programa, guardando cada nuevo primo en la tabla e incrementando **a2**. Usa el panel **Memory** para comprobar el vector en memoria, como se aprecia en la Figura 2.6. La función que comprueba si un número es primo puede escribirse tanto de forma iterativa como recursiva. ¿Qué versión es más compleja? ¿Qué versión es más eficiente?

Ejercicio 2.18. Escribe una función que, sin llevar a cabo ninguna operación funcional, provoque una espera de aproximadamente un segundo. Desde el cuerpo principal del programa muestra por consola la secuencia de valores 1 a 10 con una espera, de aproximadamente un segundo, entre un número y el siguiente.

Entrada y salida

Ejercicio 2.19. Prepara una función que, haciendo uso de las constantes `LED_MATRIX_0_BASE` y `LED_MATRIX_0_SIZE` que se generan al agregar una matriz de ledes en el panel **I/O**, ponga todos los ledes en blanco. El color blanco se consigue con el valor `0xffffff`, en el que se da la máxima intensidad a los tres colores básicos en el esquema RGB. Desde el cuerpo del programa llama a la función y verifica que toda la matriz se queda en blanco.

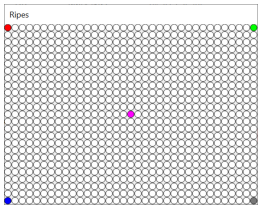


Figura 2.7. Cinco píxeles activados en la matriz.

Ejercicio 2.20. Escribe una función `pixel` que tome como entrada las coordenadas (`X, Y`) de un punto en los registros **a0** y **a1**, junto con un valor de color en **a2**, y active el led indicado de la matriz con ese valor. Llama desde el cuerpo del programa primero a la función del ejercicio previo, para *limpiar* la matriz, y después a la nueva función cinco veces para conseguir un resultado como el de la Figura 2.7, en el que se han activado con distintos colores los píxeles de las cuatro esquinas y el central. Asume que la esquina superior izquierda es la (`0, 0`), por lo que la opuesta sería (`LED_MATRIX_0_WIDTH - 1, LED_MATRIX_0_HEIGHT - 1`). Agrega al programa la función del ejercicio 2.18 y haz un `call espera` antes de cada `call pixel`, de forma que los ledes se activen de manera secuencial uno tras otro.

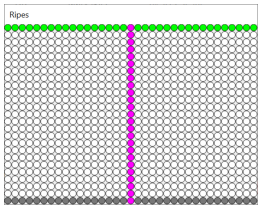


Figura 2.8. Tres líneas dibujadas en la matriz.

Ejercicio 2.21. Escribe una función `linea` que, mediante llamadas a la anterior `pixel`, dibuje una línea horizontal o vertical. Las coordenadas del primer punto se recibirán en **a0** y **a1** y las del segundo en **a3** y **a4**. Si **a0** y **a3** son iguales la línea será vertical e irá desde la fila **a1** a la **a4**. De manera análoga, si **a1** y **a4** son iguales entonces la línea será horizontal e irá desde la columna **a0** a la **a3**. Por tanto, la función `linea`, una vez hecha esa verificación, será en esencia un bucle en el que se llame repetidamente a la función `pixel` para ir dando a cada led el color almacenado en **a2**. Desde el cuerpo del programa llama a la nueva función tres veces para conseguir un resultado similar al de la Figura 2.8.

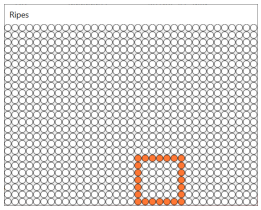


Figura 2.9. Un recuadro de color naranja.

Ejercicio 2.22. Escribe una función `recuadro` que, a partir de las coordenadas recibidas en (**a0**, **a1**) y (**a3**, **a4**) invoque cuatro veces a la función `linea` del ejercicio anterior, a fin de dibujar las dos líneas verticales y las dos horizontales que dibujen un recuadro. Desde el cuerpo del programa llama a la función para conseguir un resultado como el de la Figura 2.9. Prueba a introducir en el cuerpo del programa un bucle que vaya sumando 1 a cada uno de los cuatro registros que determinan las coordenadas del recuadro, combinando las llamadas a las funciones `recuadro` y la que *limpia* la matriz de ledes. ¿Cuál es el resultado? Puedes introducir también una llamada a la función `espera` entre el dibujo de un recuadro y su borrado a fin de ralentizar el efecto de animación.

Segmentación del cauce y riesgos de datos 3

3.1. Planificación

- **Objetivo.** Identificar los riesgos de datos e introducir las soluciones hardware/software apropiadas.
- **Temporización:**
  - **Trabajo en laboratorio**: uso de la configuración RISC-V sin detector de riesgos de datos de Ripes para identificar los problemas que implica la ejecución segmentada de instrucciones con dependencias de datos. Planificación estática para evitar esos riesgos (introducción manual de burbujas). Uso de la configuración con detector de riesgos para analizar la inclusión automática de detenciones en la ejecución y proponer potenciales soluciones: reordenamiento de código o caminos de *bypass*. Resolución de dudas.
  - **Trabajo autónomo:** ejercicios de detección de riesgos de datos y búsqueda de soluciones.
  - **Validación en laboratorio**: test y ejercicios evaluables.

3.2. Configuración de procesador

En las dos primeras prácticas no nos hemos preocupado por la arquitectura interna de nuestro procesador, la forma en que implementa la ISA de RISC-V, y simplemente hemos asumido que es capaz de ejecutar el conjunto de instrucciones que hemos conocido. No obstante, una misma ISA puede ser implementada de muy diversas maneras con el objetivo de mejorar el rendimiento.

Cualquier programa para RISC-V, sin precisar modificación alguna, verá reducido el número de ciclos al ser ejecutado en una implementación con cauce segmentado.

Una de las técnicas más habituales es la **segmentación del cauce de ejecución**, cuyos aspectos teóricos ya hemos estudiado. En esta práctica nos ocuparemos de evaluar qué implicaciones tiene esta técnica en la ejecución de nuestros programas, tanto positivos —la mejora en rendimiento— como negativos.

El emulador Ripes ofrece varias implementaciones diferentes de la ISA RV32IM. Hasta ahora hemos empleado la que se establece por defecto, sin prestarle más atención. Ahora aprenderemos a elegir una de ellas según lo que queramos experimentar.

Selección de la configuración

El botón  , situado en el extremo izquierdo de la barra de botones superior de la interfaz de usuario de Ripes, es el encargado de abrir el cuadro de diálogo mostrado en la Figura 3.1. En la rama **32-bit** del panel izquierdo hay disponibles seis opciones, de las cuales por defecto suele estar elegida **5-stage processor**. Esta corresponde a una arquitectura con un cauce segmentado de cinco etapas[1].

1: Recuerda que las etapas son IF (captación de instrucción), ID (descodificación), EX (ejecución), MEM (acceso a memoria) y WB (escritura en banco de registros).

La primera configuración, llamada **Single-cycle processor**, es la tomada como punto de referencia para medir la mejora de rendimiento. En ella cada instrucción se ejecuta en un solo ciclo de reloj, algo que en la práctica no es posible porque las instrucciones han de ser traídas desde la memoria al procesador, descodificadas, han de esperar a que los operandos estén disponibles y, finalmente, se ejecutarán,

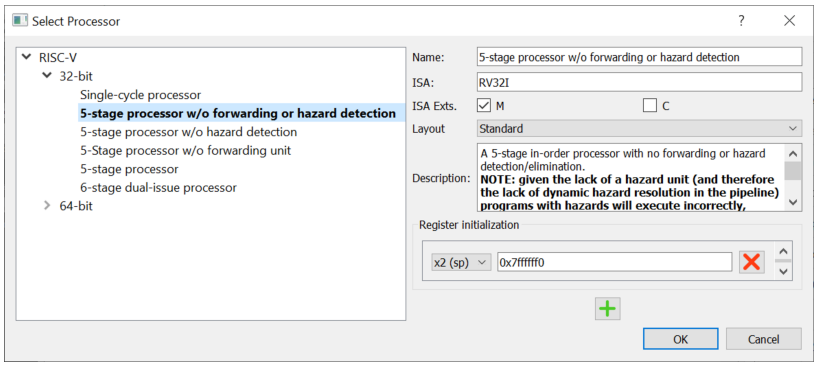


Figura 3.1. El panel izquierdo ofrece seis configuraciones distintas del procesador en versión de 32 bits y otras tantas para 64 bits. Los campos de la parte derecha permiten activar extensiones de la ISA RV32I básica, así como fijar la inicialización de ciertos registros.

tras lo cual todavía suele ser preciso tiempo para escribir los resultados en memoria o en un registro. Al ejecutar cualquier programa con esta configuración obtendremos el mínimo número de ciclos que teóricamente se podría alcanzar.

Vista del cauce de ejecución

En la barra lateral de Ripes tenemos un botón **Processor** que da paso a la vista de la Figura 3.2. Una vez introducido el programa, en la vista **Editor** como hemos venido haciendo hasta ahora, cambiaremos a esta nueva vista que nos permite examinar el cauce de ejecución, el estado de los registros, la lista de instrucciones en el orden en que serán ejecutadas y un resumen estadístico de la ejecución.

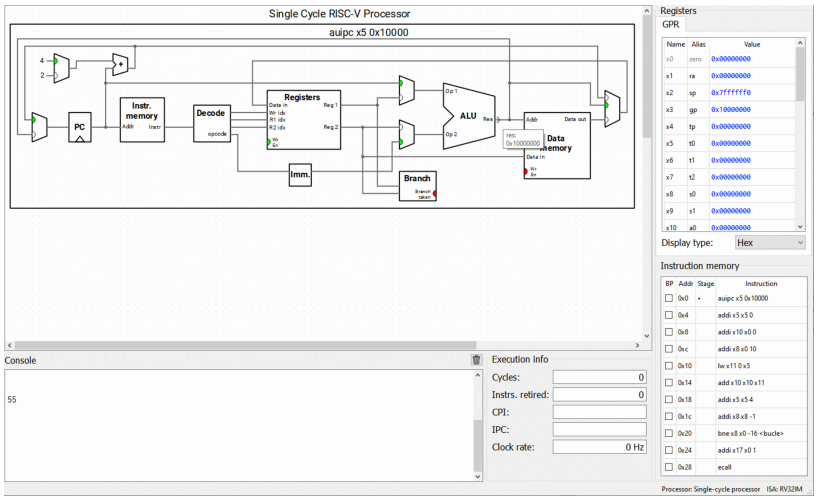


Figura 3.2. Vista del cauce de ejecución de la configuración **Single-cycle**. Solo una instrucción por ciclo entra en el cauce de ejecución. Situando el puntero del ratón sobre cualquier entrada/salida se mostrará su contenido en una etiqueta emergente. Un clic sobre cualquier línea de conexión la iluminará para mostrar las partes del procesador que comunica. En el margen derecho se muestran los registros en la mitad superior y las instrucciones del programa en ejecución en la inferior. En la parte inferior de la interfaz encontramos la consola y las estadísticas de ejecución: número de instrucciones completadas, ciclos totales, ciclos por instrucción e instrucciones ejecutadas por ciclo.

El área que muestra el cauce de ejecución cambiará al elegir una configuración distinta de procesador (véase la Figura 3.3), si bien el resto de elementos permanecerán igual.

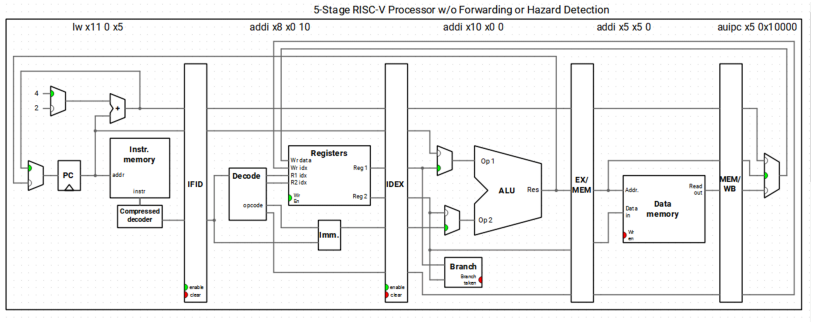


Figura 3.3. La vista del cauce de ejecución cambiarán según la configuración de procesador que elijamos. En este caso es un cauce segmentado con cinco etapas, en el que se aprecia el desacoplamiento entre etapas mediante los bancos de registros intermedios o registros de *pipeline*.

3.3. Determinar mejoras de rendimiento

Para medir las mejoras de rendimiento que obtendríamos al ejecutar un cierto programa, ya sea tras introducir cambios en la configuración del procesador o mediante ajustes en su código, usaremos las fórmulas que ya conocemos por haberlas estudiado en las sesiones de teoría. La unidad de medida que usaremos será el número de ciclos que tarda el programa en completar su tarea. Tomaremos, por tanto, el valor indicado por Ripes en el apartado **Cycles** tras cada ejecución del programa.

También podría usarse una medida de tiempo, pero un programa ineficiente, que emplee más ciclos que otro, podría tardar menos en un procesador que opere a una frecuencia mayor.

Decidida la unidad para medir el rendimiento, lo segundo que necesitamos es una evaluación a tomar como punto referencia, el valor de partida con el que comparar los resultados obtenidos tras las hipotéticas mejoras. En estas prácticas esa referencia nos la dará la configuración **Single-cycle processor**.

Ciclos de un programa en una configuración sin segmentación

Realicemos nuestros cálculos sobre un programa concreto: el del Listado 3.1. Este recorre un vector que contiene diez números para calcular su suma y mostrarla por la consola. Introdúcelo en el editor de Ripes, ejecútalo y verifica que el resultado es correcto: en la consola debería aparecer el valor `55`. Observa el uso que se hace de los registros: **`t0`** como puntero que avanza por el vector desde su primer elemento, **`a0`** como acumulador con el valor 0 inicial y **`s0`** como contador del bucle.

```
1     .data
2
3  v: .word 1, 2, 3, 4, 5
4     .word 6, 7, 8, 9, 10
5
6     .text
7
8     la t0, v
9     li a0, 0
10    li s0, 10
11
12 bucle:
13
14    lw a1, 0, t0
15    add a0, a0, a1
16
17    addi t0, t0, 4
18    addi s0, s0, -1
19    bnez s0, bucle
20
21    li a7, 1
22    ecall
```

Listado 3.1: Programa que suma los valores de un vector almacenado en memoria.

Selecciona la configuración de procesador **Single-cycle processor**, abre la vista **Processor** y ejecuta de nuevo el programa. Esto te permitirá ver el número de ciclos que informa el simulador, en el apartado **Cycles** del panel situado en la parte inferior derecha de la interfaz.

Esta es una configuración *ideal* en la que se asume que la ejecución de cada instrucción se realiza en un solo ciclo, pero la realidad es que en un procesador sin segmentar el número de ciclos que tarda en procesarse cada instrucción es variable, según lo que se tarde en cada fase de ese proceso: IF, ID, EX, MEM y WB. Como aproximación podemos asumir que se empleará un ciclo por fase, por lo que una instrucción tardaría 5 ciclos en ejecutarse.

Para el programa anterior el número de ciclos será 56. Multiplicamos este valor por 5 y obtenemos el número aproximado de ciclos que tardaría en ejecutarse el programa en un procesador sin segmentación: unos **280 ciclos**.

Ciclos del programa en otras configuraciones

Una vez que se dispone de la medida de referencia, 280 en el ejemplo previo, no hay más que obtener el número de ciclos para las demás configuraciones a analizar y aplicar la fórmula teórica que nos interese —ganancia, rendimiento, etc.— para decidir cuál de ellas sería la mejor.

Esta ganancia teórica sería para un número de ciclos alto, en el que podría despreciarse la latencia inicial —hasta que el cauce de ejecución está lleno con las cinco primeras instrucciones— y otros aspectos que afectarían al rendimiento.

Al ejecutar un programa en una arquitectura con cauce segmentado de cinco etapas se obtendría, en un caso perfecto, una ganancia teórica del 500 %, es decir, se emplearía una quinta parte de ciclos. En la práctica la segmentación induce un conjunto de problemas que es necesario abordar y que, a la postre, reducen esa ganancia teórica de manera ostensible. En la siguiente sección trataremos una de las categorías de problemas de la segmentación: los riesgos de datos.

3.4. Riesgos de datos

Al ejecutar múltiples instrucciones en paralelo, hasta cinco en un cauce segmentado con dicho número de etapas, es fácil que aparezcan **dependencias de datos**. Es un aspecto que ya hemos estudiado desde una perspectiva teórica. Sus efectos prácticos, la manera en que afectarán a la ejecución de un programa en principio correcto, los vamos a descubrir con el mismo programa del Listado 3.1.

Cambia la configuración del procesador, según lo indicado en la Subsección 3.2, y elige **5-STAGE PROCESSOR W/O FORWARDING OR HAZARD DETECTION**. A continuación ejecuta de nuevo el programa. El código de este no ha cambiado respecto a la ejecución previa, pero sin embargo ahora genera un error (véase la Figura 3.4). El cauce de ejecución, mostrado en la parte superior de dicha imagen, denota el instante en que se ha producido el fallo.

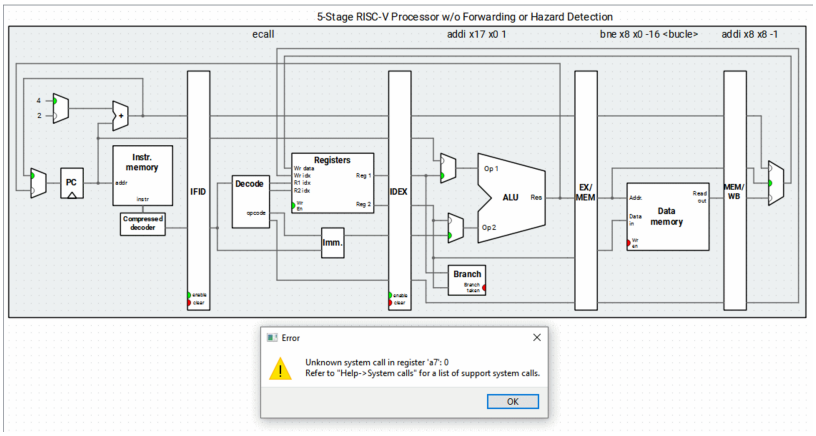


Figura 3.4. Error al ejecutar el programa en un cauce segmentado. La CPU procesa varias instrucciones en paralelo sin efectuar ningún tipo de comprobación, por lo que surgen situaciones inesperadas.

Problemas de la segmentación

Al escribir un programa que se ejecutará en un procesador con cauce segmentado, sin más circuitería de control, hemos de tener presentes las dependencias de datos a fin de evitar problemas. En la Figura 3.5 se ha diferenciado con un color distinto cada etapa para hacerlas más evidentes. En la parte superior se indica la instrucción que estaba en cada fase en el momento en que se produjo el error. Este hacía referencia al uso de una llamada de sistema desconocida y al registro `a7`[2].

2: Recuerda que la denominación `a7` es un sinónimo para el nombre real del registro que, en este caso, es `x17`.

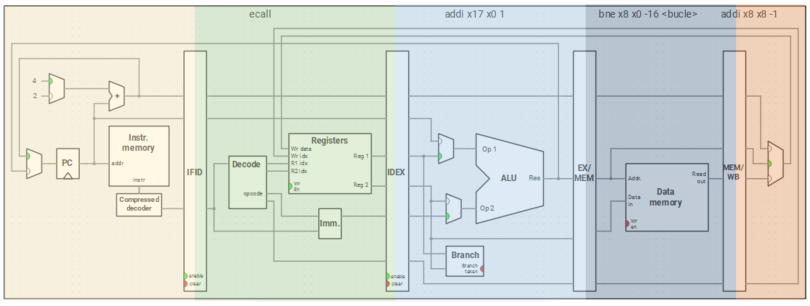


Figura 3.5. Examinamos en el cauce de ejecución la etapa en que se encuentra cada instrucción para identificar las dependencias de datos que darán potencialmente problemas.

La instrucción `ecall`, encargada de realizar llamadas al sistema, se encontraba en la etapa de descodificación en el momento del error. En el ciclo siguiente se ejecutaría la llamada al sistema, instante en el que se espera que `a7` contenga el número de servicio al que se desea invocar. El problema es que la instrucción que realiza la asignación de ese número todavía se encuentra en la fase de ejecución, por lo que faltan aun dos ciclos para que llegue a la etapa WB que, como sabemos, es el momento en que se escriben los resultados en el banco de registros.

¿Qué valor contiene el registro **a7** cuando la instrucción **ecall** llega a la etapa de descodificación? Puesto que la ejecución se ha detenido a causa del error, no hay más que observar la ventana **Registers** para responder a dicha pregunta. El valor es `0` y no existe un servicio con ese código. Ya sabemos por qué se ha producido el error. El siguiente paso sería encontrar una solución a este problema. Antes, no obstante, vamos a conocer otras opciones que nos ofrece Ripes para observar y controlar la ejecución del programa.

Etapas de ejecución y cronograma

Aunque a medida que se ejecuta un programa es posible interaccionar con la vista del cauce de ejecución, y de esta forma observar no solo la instrucción que hay en cada etapa sino también las entradas y salidas de cada bloque lógico de la CPU, hay vías adicionales para conocer las etapas por las que van pasando las instrucciones o por las que han pasado ya.

Si estamos en la vista **Editor**, que es con la que habíamos trabajado fundamentalmente en las dos prácticas anteriores, las instrucciones que se encuentran en el cauce de ejecución aparecen destacadas con distintas tonalidades de rojo. La propia etapa se indica en el margen derecho de esa barra de color, tal y como se aprecia en la mitad izquierda de la Figura 3.6. Si nos encontramos en la vista **Processor**, con la que trabajaremos casi siempre en esta práctica, la etapa de cada instrucción se indica también en el panel **Instruction memory** mostrado en la mitad derecha de la misma imagen.

La indicación de etapas es dinámica a medida que se ejecuta el programa paso a paso, tarea para la cual los atajos de teclado **F5** —ciclo siguiente— y **F4** —ciclo anterior— son el medio más cómodo de avanzar y retroceder.

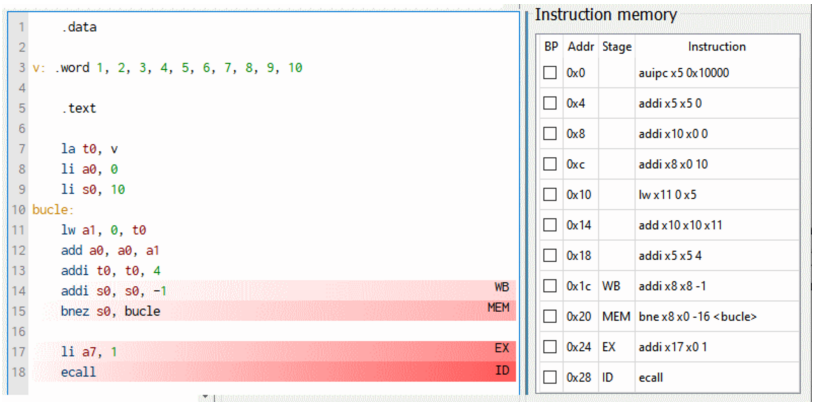


Figura 3.6. También podemos observar la etapa en que se encuentra cada instrucción recurriendo al panel **Instruction memory**, en la vista del cauce segmentado del procesador, o en el propio editor de código, en el que cada línea que está en el cauce aparece con un color diferente y una indicación en el margen derecho de la etapa donde está.

La señalización de las etapas, ya sea en el editor o en la vista del cauce de ejecución, se corresponde con el estado actual (que depende del valor del **pc** en ese instante) y cambia a medida que se avanza o retrocede. La ventana **Pipeline diagram** (véase la Figura 3.7) nos ofrece un registro cronológico de la ejecución, incluso cuando el programa ya ha llegado a su fin.

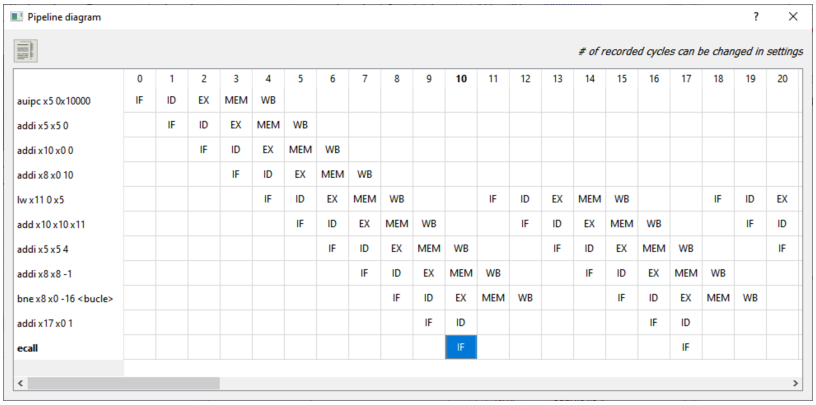
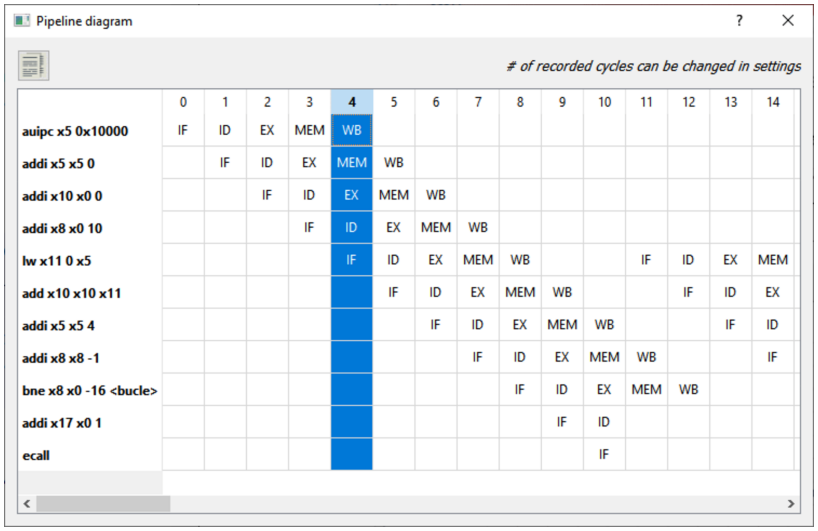


Figura 3.7. La ventana **Pipeline diagram**, que se abre con el botón situado en la parte superior de la vista **Processor**, facilita un cronograma con el registro de toda la ejecución hasta el punto actual, lo cual nos permite saber la etapa en que se encontraba cada instrucción en cada ciclo.
Importante: por defecto el cronograma solo muestra los primeros 100 ciclos de ejecución del programa. Para cambiar esta configuración usamos la opción **Edit>Preferences**, a continuación hacemos clic en la sección **Environment** de la ventana que se abre y, por último, ajustamos el parámetro **Max. pipeline diagram cycles** incrementándolo hasta el valor que necesitemos.

Análisis de la ejecución en el cronograma

El cronograma es una herramienta fundamental a la hora de descubrir las dependencias de datos que surgen durante la ejecución del programa, primer paso para ponerles solución y conseguir que funcione de manera correcta. Los ciclos se numeran desde 0 y se distribuyen de izquierda a derecha según el orden en que ha ido produciéndose la ejecución. Veamos qué ocurre durante los primeros cinco ciclos según la parte inicial del cronograma —visible en la Figura 3.8— tras haber lanzado la ejecución del código anterior.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
auipc x5 0x10000	IF	ID	EX	MEM	WB										
addi x5 x5 0		IF	ID	EX	MEM	WB									
addi x10 x0 0			IF	ID	EX	MEM	WB								
addi x8 x0 10				IF	ID	EX	MEM	WB							
lw x11 0 x5					IF	ID	EX	MEM	WB			IF	ID	EX	MEM
add x10 x10 x11						IF	ID	EX	MEM	WB			IF	ID	EX
addi x5 x5 4							IF	ID	EX	MEM	WB			IF	ID
addi x8 x8 -1								IF	ID	EX	MEM	WB			IF
bne x8 x0 -16 <bucle>									IF	ID	EX	MEM	WB		
addi x17 x0 1										IF	ID				
ecall											IF				

Figura 3.8. Momento en el que se ocupan las cinco etapas del cauce de ejecución. A partir de ese ciclo la CPU operaría a pleno rendimiento, tras la latencia inicial hasta que se cargan las primeras cinco instrucciones del programa.

Ciclo 0. Se capta la primera instrucción. El resto de etapas por el momento no tienen trabajo que hacer.

Ciclo 1. La instrucción anterior pasa a la etapa de descodificación y, al mismo tiempo, se capta la segunda instrucción del programa.

Ciclo 2. Pasa a captación la tercera instrucción, la segunda se descodifica y la primera alcanza la fase de ejecución.

Ciclo 3. La primera instrucción alcanza la etapa de memoria, aunque en la práctica no tenga nada que hacer en ella, mientras las demás avanzan por el cauce y se capta la cuarta instrucción.

Ciclo 4. Al captarse la quinta instrucción las cinco etapas del cauce están ocupadas. A partir de este momento, y asumiendo un caso ideal en el que no hubiese ningún tipo de riesgo, la CPU irá finalizando una instrucción por ciclo.

Las dos primeras instrucciones que vemos en el cronograma, `auipc` y `addi` con el registro `x5` como destino, son resultado de la seudoinstrucción `la` con la que guardamos en el registro `t0` la dirección del vector de valores.

Al observar estos primeros ciclos estamos en disposición de detectar ciertos fallos. Fíjate en qué ocurre en el ciclo 2 con la instrucción `addi`. Se encuentra en la etapa ID, momento en el que hay que leer los operandos, en este caso `x5`, del banco de registros. En el ciclo 3 ese operando y el valor inmediato 0 se introducirán en la ALU para hacer la suma. El problema estriba en que el contenido del registro `x5`, establecido por la instrucción `auipc`, no será escrito hasta el ciclo 4, momento en que esa instrucción llega a la etapa WB. Por tanto, la operación aritmética de la segunda instrucción no se efectuará con los valores correctos.

El hecho de que una instrucción se capte pero luego no pase a descodificarse ni ejecutarse es producto de los saltos existentes en el programa y dan lugar a los denominados **riesgos de control**, de los cuales nos ocuparemos en la práctica siguiente.

Volveremos sobre el problema que plantea la ejecución de esas dos instrucciones, y otras dependencias de datos existentes en nuestro código y que también produciría fallos, en un momento. Ahora centrémonos en la instrucción `ecall` que genera el error al intentar ejecutar el programa. Como se aprecia en el cronograma, cuya parte final aparece en la Figura 3.9, `ecall` es captada múltiples veces a lo largo de la ejecución del programa. Sin embargo, no es hasta el ciclo 81 cuando pasa por primera vez a la etapa de descodificación.

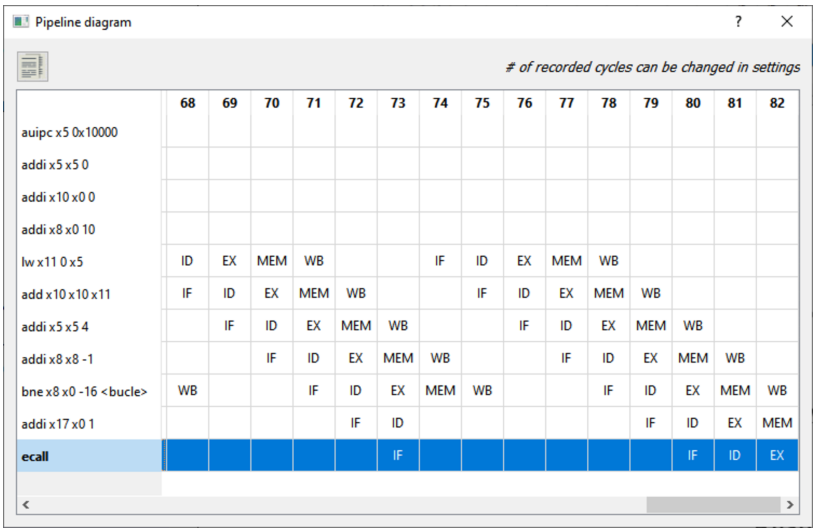


Figura 3.9. Etapas por las que pasa la instrucción `ecall` durante la ejecución. Hasta que no se llega al ciclo 80 esta instrucción no pasa de la etapa de captación. En el ciclo 81 se descodifica, como resultado de que el salto condicional deja de cumplirse, momento en que se detecta la inexistencia del servicio solicitado y se produce el fallo.

En el ciclo 82, que no ha llegado a ejecutarse a consecuencia del error, la instrucción `addi x17 x0 1` que se encarga de seleccionar el servicio al que debe invocarse, todavía está en la etapa MEM. Por tanto no es posible ejecutar la orden `ecall`, que ya está en la etapa de ejecución. Habrá que retrasar la llegada a esa etapa hasta que el registro `x17` —alias de `a7`— haya alcanzado la etapa WB.

Introducción manual de paradas

La solución al problema encontrado en nuestro programa pasa por la introducción de tantos ciclos de parada —también conocidos como burbujas o *stalls*— como se precisen para que el registro respecto al cual se tiene una dependencia adquiera su valor correcto. En este caso concreto necesitamos que cuando `ecall` llegue a la etapa ID, momento en que se leen del banco de registros los argumentos, la instrucción `li a7, 1` ya se haya ejecutado.

Para introducir una detención de un ciclo recurriremos a la seudoinstrucción `nop` (*No OPeration*) que, al ensamblarse, produce una instrucción `addi x0, x0, 0`. Dado que `x0` es inmutable, siempre contiene el valor 0, esa instrucción no tiene efecto alguno, pero deberá pasar por todas las etapas de ejecución. Es una manera *artificial* pero simple de introducir en el cauce de ejecución tantos ciclos de parada como se precisen.

La introducción de una burbuja afectará, como es lógico, a todas las instrucciones que aparezcan tras ella, no solo a la inmediatamente posterior.

Tomamos las dos instrucciones implicadas, `li` y `ecall` en este caso, y fijamos la etapa en la que la segunda depende del valor de la primera: ID. Tendríamos que rellenar las etapas EX, MEM y WB para que la primera haya terminado, por lo que introducimos tres ciclos de parada tal y como se aprecia en el listado al margen.

```
1    li a7, 1        # Finalizada
2    nop             # WB
3    nop             # MEM
4    nop             # EX
5    ecall           # ID
```

Tras hacer el cambio anterior verifica que el programa se ejecuta sin producir error alguno y, de hecho, al terminar muestra por consola un resultado. Este, sin embargo, dista mucho de ser correcto. Ello se debe a que en el código del programa existen aun varios riesgos de datos sin resolver, debidos a las siguientes dependencias:

- La línea 15 del Listado 3.1 precisa el contenido de `a1` para sumarlo a `a0`, pero hasta que la instrucción `lw` de la línea 14 no lea ese valor desde memoria no estará disponible. Habrá que introducir los ciclos de parada necesarios.
- La línea 18 precisa el contenido de `s0`, a fin de comprobar si es 0, pero el valor de ese registro es modificado por la instrucción previa y el cambio no será efectivo hasta que alcance la etapa WB. De nuevo, es preciso añadir ciclos de parada.

```
bucle:
    lw a1, 0, t0
    nop
    nop
    add a0, a0, a1
    addi t0, t0, 4
    addi s0, s0, -1
    nop
    nop
    bnez s0, bucle
```

El cuerpo del bucle del programa, tras agregar las instrucciones **nop** imprescindibles, quedaría como se aprecia al margen. A pesar de estas correcciones el programa continuará sin producir el resultado esperado. Todavía queda un riesgo de datos en el código y, en principio, no es posible un **nop** entre las instrucciones implicadas. Dedica unos minutos a examinar el código ensamblador, el que aparece en el panel derecho de Ripes, e intenta identificar el punto en cuestión y cuál sería la solución.

La primera orden de nuestro programa es **la** `t0, v`, con la cual alojamos en el registro **t0** la dirección marcada por la etiqueta `v`. Sabemos que se trata de una seudoinstrucción que genera dos instrucciones, ya que una dirección de 32 bits —que es lo que representa el símbolo `v`— no puede ser cargada en un registro en un solo paso. De ahí que se asignen primero sus 20 bits de mayor peso[3] y después, con una operación de suma inmediata, se completen los 12 bits restantes. Como es lógico, este segundo paso no debería ejecutarse hasta en tanto no se haya completado el primero, ya que de lo contrario el valor resultante no será correcto.

3: Revisa el apartado sobre modos de direccionamiento de la primera práctica para recordar cómo funcionan instrucciones como **auipc** y **lui**, esenciales en esta tarea.

Suma de un vector de números con detenciones

Dado que no es posible introducir un **nop** entre las dos instrucciones que produce **la**, la alternativa es no usar esta seudoinstrucción sino cargar la dirección del vector en los dos pasos ya citados, lo que sí nos da opción a generar esos ciclos de espera. La versión correcta del programa, adecuada para su ejecución en un cauce segmentado con cinco etapas, sería la del Listado 3.2.

Listado 3.2: El programa tras introducir los cambios necesarios para evitar los riesgos de datos.

```

    .data

v: .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

    .text

    lui t0, %hi(v)
    nop
    nop

    addi t0, t0, %lo(v)
    li a0, 0
    li s0, 10


bucle:

    lw a1, 0, t0
    nop
    nop

    add a0, a0, a1
    addi t0, t0, 4
    addi s0, s0, -1
    nop
    nop

    bnez s0, bucle

    li a7, 1
    nop
    nop
    nop
    ecall
```

A pesar de todo esta versión del programa, al ejecutarse en un cauce segmentado, reduce de manera considerable los 280 ciclos que tardaría en una CPU secuencial antes de introducir los **nop**. El resultado es una ganancia significativa en rendimiento.

Ahora, al ejecutar el programa, el resultado sí será correcto. No obstante, el programa ocupa más memoria de la que precisaba originalmente, ya que hemos introducido nueve instrucciones **nop**. Esto también tiene consecuencias en el tiempo que tarda el programa en ejecutarse: 123 ciclos para esta versión, con 1.22 ciclos por instrucción (CPI). Este último valor se aleja del objetivo —$CPI = 1$— que se alcanzaría con un cauce sin dependencias de datos y otros problemas que abordaremos en la siguiente práctica.

Detección automática de riesgos

Introducir de forma manual las detenciones precisas en un programa no es algo práctico. Incluso para un caso muy sencillo, como es el que estamos empleando, nos ha llevado un tiempo encontrar los puntos donde eran necesarias las paradas. Es un trabajo que podría realizar de manera automática un compilador, pero es todavía más eficiente dejar esta tarea al propio hardware.

Abre el cuadro de diálogo **Select Processor** de Ripes y elige la opción **5-Stage processor w/o forwarding unit**. Esta implementación de la ISA RV32IM cuenta con los elementos precisos para detectar los riesgos de datos, que generan errores como los que hemos tenido ocasión de experimentar, e introducir las detecciones adecuadas sin que esto se traduzca en más instrucciones en el programa.

Ahora carga la versión original del programa, la del Listado 3.1, y ejecuta los primeros ciclos con **F5** mientras examinas el cauce de ejecución. Observa en la Figura 3.10 cómo justo antes de que la primera instrucción `addi` llegue a la etapa de ejecución se han insertado dos ciclos de espera, señalados en rojo en la parte superior. Esto soluciona el fallo que detectábamos la primera vez que ejecutamos este código en un cauce segmentado.

Aunque en el cauce aparece la indicación `nop` cuando el hardware inserta una detención, en realidad no se ejecuta instrucción alguna como sí ocurría al agregar de forma manual esa instrucción. Este es un aspecto que influirá en los cálculos de rendimiento de la CPU.

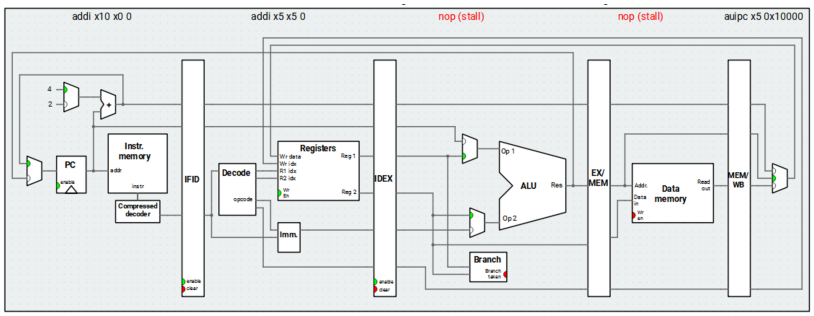


Figura 3.10. Con la nueva configuración la CPU dispone de la circuitería precisa para detectar los riesgos de datos y evitarlos, para lo cual se introducen de manera automática las detenciones que se precisen en el cauce de ejecución.

Ejecuta ahora el programa completo. El número de ciclos que tarda con esta nueva configuración es casi el mismo que nuestra versión con detenciones manuales: 122 ciclos. Sin embargo, el CPI ha subido hasta 2.18. La razón la encontramos en el número de instrucciones efectivamente ejecutadas, dato que ha pasado desde las 101 en la versión con instrucciones `nop` respecto a 56 ahora. Claro, todos los `nop` que se ejecutaban en cada iteración del bucle ahora no están ahí, las detenciones las genera el hardware cuando detecta un riesgo de datos, sin incrementar de manera artificial la cantidad de instrucciones.

Adelantamiento de datos

La mayor parte de riesgos de datos se producen como consecuencia de que el valor resultante de una cierta operación no ha sido todavía escrito en el registro de destino. Esa operación tiene lugar en la etapa WB, como ya sabemos. Esto no significa, sin embargo, que ese valor no se conozca con antelación. El resultado de ejecutar `addi a0, zero, 5` se generará en la etapa EX, cuando en la ALU se introduzca el valor 0 como primer operando, el valor 5 como segundo y se genere la salida. Esta se almacena temporalmente en los registros intermedios que hay entre una etapa y la siguiente.

¿Qué ocurriría si creásemos una conexión, entre el banco de registros intermedio que hay entre las etapas EX y MEM, para que la salida que acaba de generar la ALU se reconduzca como primer operando de entrada de ese mismo elemento? Sería una ruta hacia atrás, en lugar de hacia adelante, que permitiría usar el resultado producido por una operación a pesar de que todavía no haya sido escrito en el registro de destino. Son los llamados **caminos de adelantamiento** o bien caminos de *bypass*.

Contar con este mecanismo evitaría tener que introducir ciclos de espera cuando se detectan riesgos de datos, por lo que contribuiría a mejorar el rendimiento de la CPU. A cambio haría el diseño de esta algo más complejo, desde una perspectiva electrónica, ya que es preciso detectar cuándo una etapa precisa un operando que todavía no ha sido escrito en el banco de registros pero ya está disponible en otra etapa posterior.

Para activar los adelantamientos en Ripes elegiremos la configuración de procesador **5-stage processor**. Es una implementación que no solo detectará de manera automática los riesgos de datos sino que, siempre que sea posible, los evitará anticipando los resultados conocidos como operandos de entrada. Hecho este cambio de configuración, ejecutamos los primeros ciclos del programa hasta llegar al momento en que la instrucción `auipc x5 0x10000` es ejecutada y, por tanto, pasa a la etapa MEM. Es el instante cuyo detalle aparece en la Figura 3.11.

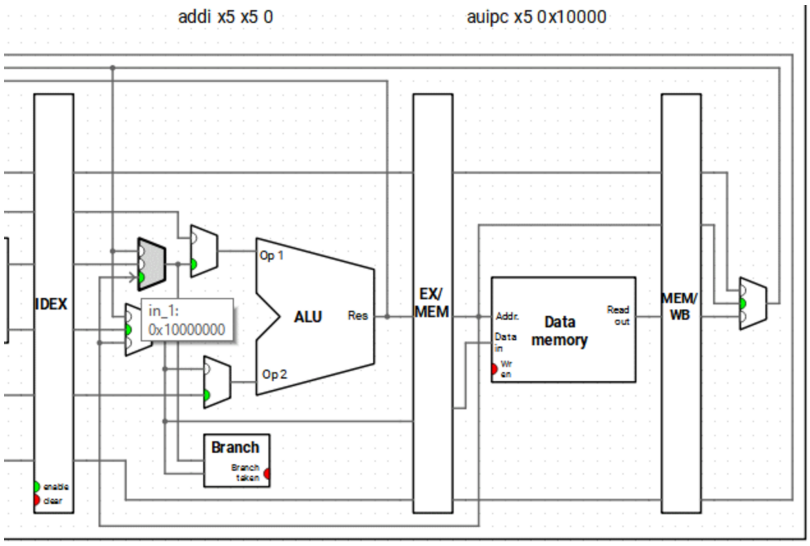


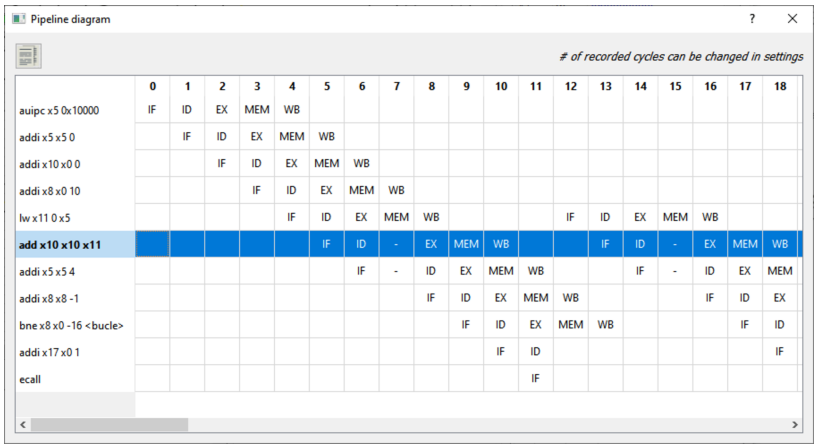
Figura 3.11. Con los adelantamientos activados, la etapa en la que se precisa el operando, en este caso el registro `x5` en la etapa EX, puede obtener su valor sin esperar a que se escriba en el banco de registros. Observa el detalle de la etiqueta del multiplexor de la etapa EX que muestra el valor que se está recibiendo desde la etapa posterior.

La siguiente instrucción, que ha alcanzado la etapa EX, depende del resultado que la anterior haya almacenado en `x5`. Si situamos el puntero del ratón sobre la entrada del multiplexor que hay junto a la entrada **Op 1** de la ALU, concretamente la que está en color verde, en una etiqueta se indicará el valor que llega en ese ciclo por el camino de *bypass*. Es el que se almacenará después en `x5` pero, si comparas esta imagen con la Figura 3.10, comprobarás que el procesador no ha introducido ninguna espera entre una instrucción y la siguiente. El adelantamiento hace posible ejecutar cada instrucción en el momento que corresponde, sin demoras.

La ejecución con esta configuración precisará 90 ciclos, contra los 122 anteriores, y el CPI bajará hasta 1.61.

Reordenación de código

Los caminos de adelantamiento son un avance significativo en el hardware de una CPU, al reducir los ciclos en los que no se efectúa trabajo útil, pero no son una solución universal al problema de las detenciones por riesgos de datos. Es algo que comprobaremos si, tras ejecutar el programa con la configuración anterior, abrimos la ventana **Pipeline diagram** y buscamos qué ocurre entre las instrucciones `lw` y `add` dispuestas al inicio del bucle. Observamos (véase la Figura 3.12) que se ha introducido un ciclo de espera entre ambas.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
auipc x5 0x10000	IF	ID	EX	MEM	WB														
addi x5 x5 0		IF	ID	EX	MEM	WB													
addi x10 x0 0			IF	ID	EX	MEM	WB												
addi x8 x0 10				IF	ID	EX	MEM	WB											
lw x11 0 x5					IF	ID	EX	MEM	WB				IF	ID	EX	MEM	WB		
add x10 x10 x11						IF	ID	-	EX	MEM	WB			IF	ID	-	EX	MEM	WB
addi x5 x5 4							IF	-	ID	EX	MEM	WB			IF	-	ID	EX	MEM
addi x8 x8 -1									IF	ID	EX	MEM	WB				IF	ID	EX
bne x8 x0 -16 <bucle>										IF	ID	EX	MEM	WB				IF	ID
addi x17 x0 1											IF	ID							IF
ecall												IF							

Figura 3.12. Los caminos de adelantamiento no pueden eliminar todas las detenciones. En la ventana del cronograma se aprecia cómo al usar la instrucción `lw` el valor no está disponible en la etapa de ejecución, como ocurre con las instrucciones aritméticas, sino que hay que esperar hasta que se complete la transferencia desde memoria en la etapa MEM, de ahí que sean todavía precisas algunas detenciones.

Apreciaremos mejor el detalle del problema si examinamos el propio cauce de ejecución y prestamos atención a la instrucción `lw` a medida que avanza por el mismo. Cuando llega a la etapa EX, en la que se calculará la dirección desde la que se transferirá un dato[4], la instrucción `add` está en la etapa ID. Para pasar a EX necesita sus operandos, pero uno de ellos todavía no es conocido. Hasta que

4: Esta dependerá de la posición del vector que marque el registro `t0` en cada iteración.

la instrucción `lw` no alcance la etapa MEM no se hará efectiva la lectura desde memoria. Por esta razón la instrucción `add` ha de esperar un ciclo. Efectuada la lectura, y a pesar de que el valor obtenido de memoria todavía no se haya escrito en el banco de registros, ya sí es posible usar un camino de *bypass* —resaltado en amarillo en la Figura 3.13— y, de esta forma, no introducir un segundo ciclo de espera.

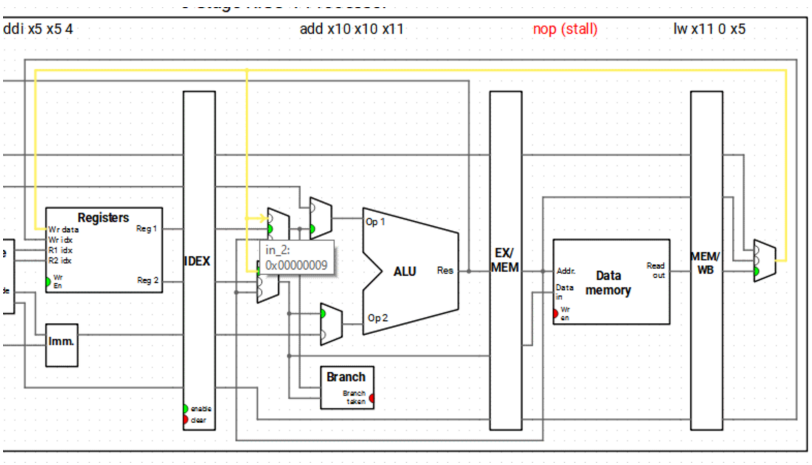


Figura 3.13. El resultado en `x11` no lo produce la ALU, sino que es leído de memoria y, por tanto, no estará disponible hasta que se haya ejecutado la etapa MEM.

La optimización del código de programas que presentan estos problemas, en forma de riesgos de datos no resolubles mediante adelantamiento, pasa por llevar a cabo una reordenación del código. La idea básica es la siguiente: separar aquellas instrucciones que tienen dependencias, intercalando entre ellas otras que puedan ser ejecutadas en orden distinto al que tenían originalmente en el programa sin que el resultado final de este se vea afectado.

La eliminación de dependencias de datos mediante reordenación es un proceso iterativo. Partimos del código en su versión original y repetimos los siguientes pasos:

1. Identificamos un par de instrucciones $d1$ y $d2$ entre las que existe una dependencia que provocará detenciones.
2. Buscamos una instrucción I, anterior a $d1$ o posterior a $d2$, que pueda intercalarse entre ellas sin alterar el resultado final.
3. Efectuamos el cambio, ejecutamos el programa y verificamos que el resultado que genera sigue siendo correcto.
4. Si quedan todavía ciclos de parada por riesgos de datos volvemos al paso 1.

Código reordenado para evitar riesgos de datos

Tomamos el código del programa original, mostrado en la mitad izquierda del Listado 3.3, y aplicamos el método anterior las veces necesarias, de forma que los riesgos de datos se reduzcan de forma paulatina. De esta forma llegaríamos a la versión del programa de la parte derecha. Los cambios introducidos a cada paso han sido los siguientes:

- Se ha intercalado la instrucción de la línea 15 entre la 13 y 14, de forma que la carga del dato de la memoria tenga tiempo de ejecutarse.
- La instrucción en la línea 17, el salto condicional, necesita que la operación aritmética sobre `s0` se ejecute en la ALU, razón por la que hemos tomado la línea 16 y la hemos adelantado.
- Al llevar la instrucción de la línea 16 a la 14 ha aparecido un nuevo riesgo de datos, provocado por la proximidad de la instrucción `addi` a la instrucción `li` con la que se inicializaba `s0`. Por esa razón se han intercambiado las líneas 8 y 9.

Tras los cambios ya no quedan riesgos de datos en el programa. Incluso aunque desactivásemos los caminos de adelantamiento los ciclos de parada precisos serían mínimos. Esta versión del programa se ejecuta en 80 ciclos y consigue un CPI de 1.43, por lo que es un 53 % más rápido que la versión original.

Listado 3.3: El programa original (izquierda) y tras reordenar las instrucciones para evitar riesgos de datos. En esta última versión se han marcado con comentarios los cambios de posición en aquellas líneas que se han intercambiado.

```
    .data

v:  .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

    .text

    la t0, v
    li a0, 0
    li s0, 10

bucle:

    lw a1, 0, t0
    add a0, a0, a1
    addi t0, t0, 4
    addi s0, s0, -1
    bnez s0, bucle

    li a7, 1
    ecall
```

```
    .data

v: .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

    .text

    la t0, v
    li s0, 10   # 9->8
    li a0, 0    # 8->9

bucle:

    lw a1, 0, t0
    addi s0, s0, -1  # 16->14
    addi t0, t0, 4
    add a0, a0, a1   # 14->16
    bnez s0, bucle

    li a7, 1
    ecall
```

3.5. Comparativa de rendimientos

A lo largo de esta práctica se han hecho múltiples ejecuciones en las que hemos combinado tres versiones distintas del código y varias configuraciones diferentes de la CPU. Lo lógico es resumir toda esta información a fin de facilitar las comparaciones. Esto podemos hacerlo en una representación tabular, como la de la Tabla 3.1, o bien con una representación gráfica en la que también incluyamos los valores, como se aprecia en la Figura 3.14.

Tabla 3.1. Comparación del número de ciclos y CPI conseguidos con cada versión del programa. La columna más a la derecha indica cuánto más rápida es una versión respecto a la original.

Versión	Ciclos	CPI	Mejora
Original con detección de riesgos	122	2.18	-
Con caminos de adelantamiento	90	1.61	36 %
Reordenado sin caminos de adelantamiento	82	1.46	48 %
Reordenado con caminos de adelantamiento	80	1.43	53 %

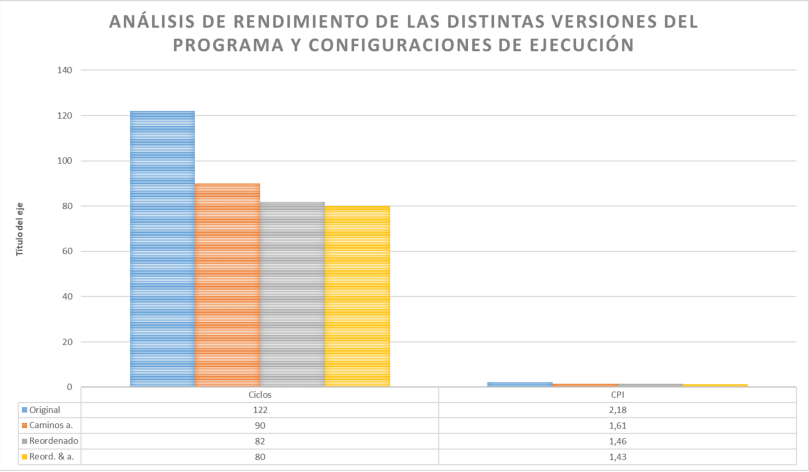


Figura 3.14. Gráfica con los ciclos, CPI y mejora de cada configuración.

También resultaría de interés incluir en la tabla/gráfica el número de detenciones que se produzcan en cada configuración, información que obtendríamos directamente desde el cronograma de Ripes. Lo ideal es que la versión final, aquella en la que hemos reordenado el código y activado los caminos de adelantamiento, no tenga ninguna detención por riesgos de datos. Según el código original del programa esto puede ser más o menos fácil de conseguir.

3.6. Ejercicios propuestos

Los ejercicios propuestos a continuación te permitirán poner en práctica las técnicas que acaban de describirse, así como analizar las mejoras en rendimiento que se consiguen con las distintas configuraciones de la CPU que ofrece Ripes. En dichos ejercicios siempre emplearemos las configuraciones de procesador **Single-cycle processor** (C1), **5-Stage processor w/o forwarding unit** (C2) y **5-stage processor** (C3).

Ejercicio 3.1. Toma el código del ejercicio 2.5 de la práctica anterior y lleva a cabo el siguiente trabajo:

a) Ejecuta el código en la configuración C1 y calcula cuál sería el número de ciclos en un procesador sin segmentación del cauce de ejecución. El CPI será igual a 1.
b) Ejecuta el código en la configuración C2, anota el número de ciclos y CPI. Calcula la mejora en rendimiento respecto a la configuración anterior.
c) Repite el paso anterior pero con la configuración C3.
d) Identifica las detenciones que se estén produciendo en el programa y reordena el código para evitarlas. Anota los ciclos y CPI de esta nueva versión en la configuración C3 y calcula la mejora de rendimiento.

Puedes usar una o más de las fórmulas aprendidas en teoría para calcular la ganancia de rendimiento de cada nueva ejecución. Asimismo, también puedes incluir la mejora respecto a la versión inicial, no solo respecto a la anterior.

Ejercicio 3.2. Toma el código del ejercicio 2.6 de la práctica anterior y repite los pasos indicados para el 3.1.

Ejercicio 3.3. Toma el código del ejercicio 2.11 de la práctica anterior y repite los pasos indicados para el 3.1.

Puedes realizar el trabajo indicado sobre cualquier otro ejercicio cuyo código hayas escrito, con el objetivo de obtener una versión optimizada, sin detenciones por riesgos de datos, y un análisis de los ciclos, CPI y mejora de las distintas configuraciones.

Ejercicio 3.4. Toma el código del ejercicio 2.1 de la práctica anterior. Activa la configuración **5-stage processor w/o forwarding or hazard unit**. Modifica el código introduciendo las instrucciones `nop` que garanticen una correcta ejecución del programa.

Ejercicio 3.5. Toma el código del ejercicio 2.2 de la práctica anterior y repite el trabajo indicado para el 3.4.

Segmentación del cauce y riesgos de control 4

4.1. Planificación

- **Objetivo.** Identificar los riesgos de control que implican las instrucciones de salto y reducir el impacto que conllevan en el rendimiento.
- **Temporización:**
  - **Trabajo en laboratorio**: análisis de la influencia de los saltos en la ejecución del programa. Riesgos de control en una ejecución segmentada. Desenrollado de bucles como solución para reducir ese impacto. Resolución de dudas.
  - **Trabajo autónomo:** ejercicios de detección de riesgos de control y búsqueda de soluciones.
  - **Validación en laboratorio**: test y ejercicios evaluables.

4.2. Configuración de procesador

En la práctica anterior conocimos las distintas configuraciones de CPU que nos ofrece el software Ripes, así como el impacto que tienen en la gestión de los riesgos de datos.

Ni la unidad de detección de riesgos de datos ni los caminos de adelantamiento, herramientas ambas esenciales para asegurar un correcto funcionamiento de los programas y minimizar los ciclos de espera, afectan en modo alguno a las instrucciones de salto. En consecuencia, los riesgos de control generados por estas instrucciones estarán presentes sin que importe qué configuración tengamos activa.

Por esta razón, en los ejercicios de la presente práctica siempre seleccionaremos la configuración **5-stage processor**, en la que se tienen activadas las unidades de detección de riesgos de datos y también los caminos de *bypass*. El objetivo es poder centrarnos en los riesgos de control, sin que los de datos nos afecten de manera significativa.

4.3. Instrucciones de salto

En el apartado 2.2 (véase la segunda práctica, en la pág. 15) conocimos las instrucciones de salto que nos ofrece la ISA de RISC-V, tanto condicionales como incondicionales. También se describieron algunas seudoinstrucciones, como `call` y `ret`, que al fin y al cabo se traducen a instrucciones de salto incondicional, por lo que funcionan como tales.

Aquí vamos a analizar cómo se procesan las instrucciones de salto en el cauce segmentado de ejecución de RISC-V. Para ello recurriremos a la misma vista **Processor** del programa Ripes que empleábamos en la práctica anterior.

Usaremos de nuevo el código del programa que suma los elementos de un vector, reproducido en el Listado 4.1, al que hemos agregado un salto incondicional y dos instrucciones `nop` justo antes de la etiqueta `bucle`. De esta forma contamos tanto con saltos condicionales como no condicionales. Este último no afectará al resultado producido por el programa, dado que se limita a saltar las dos instrucciones `nop` intercaladas antes del cuerpo del bucle.

Listado 4.1: Programa que suma los valores de un vector almacenado en memoria tras agregar un salto incondicional que no afecta al resultado final.

```
1     .data
2
3  v: .word 1, 2, 3, 4, 5
4     .word 6, 7, 8, 9, 10
5
6     .text
7
8     la t0, v
9     li a0, 0
10    li s0, 10
11    j bucle
12    nop
13    nop
14
15 bucle:
16
17    lw a1, 0, t0
18    add a0, a0, a1
19
20    addi t0, t0, 4
21    addi s0, s0, -1
22    bnez s0, bucle
23
24    li a7, 1
25    ecall
```

Como sabemos, el registro **pc** es el que indica en cada momento a la CPU la instrucción que ha de captar —durante la etapa IF— de memoria. Nos interesa analizar cómo se actualiza el contenido de dicho registro a lo largo de la ejecución de un programa, tanto al procesar instrucciones aritméticas, de carga, etc., como cuando se tienen instrucciones de salto, ya que estas últimas alteran el contenido del **pc**.

Cómputo de la dirección de salto

Una vez introducido el código del anterior programa en el editor de Ripes, pulsamos **F5** varias veces hasta casi llenar el cauce de ejecución, tal y como se aprecia en la Figura 4.1. A cada ciclo se calculará el nuevo valor del **pc**. Este trabajo se lleva a cabo en la etapa IF de manera automática, sin que el programa intervenga en dicha tarea.

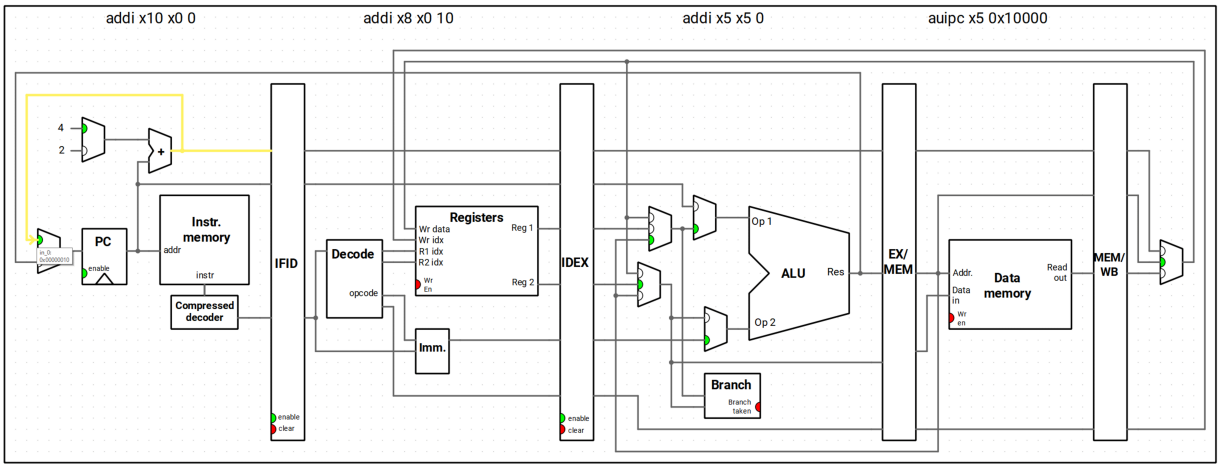


Figura 4.1. El **pc** se actualiza en cada ciclo, por defecto con la salida que genera un sumador —situado en la etapa IF— que incrementa el anterior valor del mismo registro. Observa que la entrada del **pc** proviene de un multiplexor cuya primera entrada (la superior) procede de dicho sumador.

El recuadro señalado como **PC** representa un registro, compuesto de los biestables necesarios para almacenar el estado de los bits en cada momento.

En la parte superior de la etapa IF se aprecia la existencia de un sumador: una pequeña ALU con una primera entrada fija —el valor 4— y una segunda que procede de la salida del propio **pc**. En cada ciclo el contenido del **pc** toma el valor del multiplexor situado a su izquierda, concretamente la entrada marcada en verde de dicho elemento. La línea resaltada en color claro denota la conexión entre el sumador y la entrada al multiplexor.

Usa los comandos asociados a **F5** y **F4** de Ripes para analizar cómo cambia el valor del **pc** a medida que se avanza en el programa. En la parte inferior derecha de la interfaz tienes las instrucciones del programa y la dirección en que están alojadas, en el margen izquierdo. Para todas las instrucciones iniciales del programa el comportamiento siempre será el mismo: el valor del contador de programa se incrementa en 4 para avanzar a la siguiente instrucción, ya que cada una ocupa 4 bytes de memoria.

Veamos ahora qué ocurre cuando en el cauce de ejecución se procesa una instrucción de salto. En concreto la instrucción `j bucle` del Listado 4.1. En realidad **`j`** es una seudoinstrucción que se traduce como **`jal`** `x0, desp`, de manera que a la actual dirección del **`pc`** se le suma `desp` y el anterior valor del **`pc`** se descarta puesto que **`x0`** siempre es cero. El salto es incondicional y, en consecuencia, esa modificación del **`pc`** se efectuará siempre que se ejecute la instrucción.

La actualización del **`pc`** se produce tras haberse calculado la nueva dirección, lo que ocurre en el ciclo en el que la instrucción de salto pasa de la etapa EX a la MEM.

Es en la etapa EX —aquella en la que se encuentra la instrucción **`jal`** en la Figura 4.2— donde se calcula el nuevo valor que se asignará al **`pc`**. Para ello se toma como primer operando el contenido de dicho registro y como segundo el valor inmediato que acompaña a la instrucción. El resultado de esa suma, como se aprecia en la línea de conexión destacada en color claro, vuelve hacia atrás, a la etapa IF.

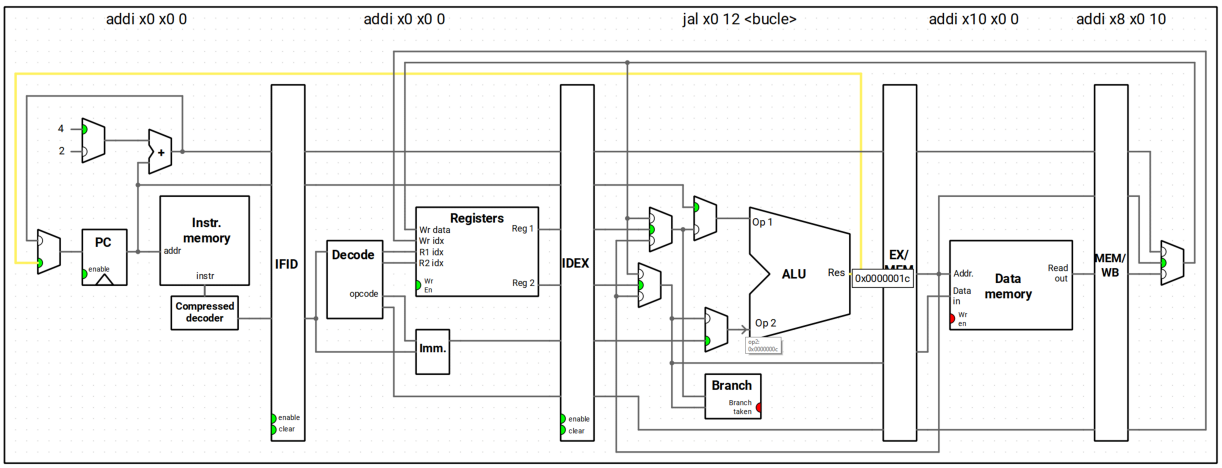


Figura 4.2. La nueva dirección del contador de programa se calcula en la etapa EX y se devuelve como entrada al **`pc`** en el siguiente ciclo de ejecución.

Evaluación de la condición de salto

Las instrucciones de salto condicional operan, en cuanto al cálculo de la dirección que se asignaría al **`pc`**, de la forma que acaba de describirse. La diferencia estriba en que la nueva dirección, obtenida como salida de la ALU, se enviará de vuelta al **`pc`** solo si la condición analizada se cumple. Esta verificación tiene lugar en el elemento **Branch** situado en la parte inferior de la etapa EX y cuyo detalle se muestra en la Figura 4.3.

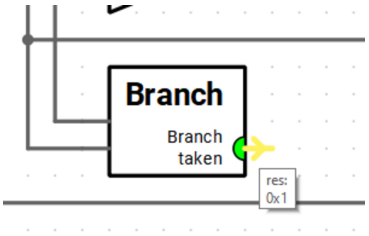


Figura 4.3. En la misma etapa EX se verifica la condición que determina si la salida de la ALU se asignará o no al contador de programa.

Más allá de esta verificación, que controla cuál será la línea de entrada al **`pc`** en el siguiente ciclo, los saltos condicionales no presentan mayores diferencias frente a los incondicionales. Por ello ambos dan lugar a los mismos riesgos de control.

4.4. Riesgos de control

Ahora que sabemos cómo se procesan las instrucciones de control en un cauce segmentado —la dirección se calcula en la etapa EX y se asigna al **`pc`** en la etapa MEM— centrémonos en los riesgos que esto implica durante la ejecución de un programa cualquiera. Para ello, y siempre con el código anterior cargado en Ripes, procedemos a ejecutar varios ciclos y nos detenemos en el momento en que la instrucción **`bne`** alcanzada la citada etapa EX. Es la situación a la que corresponde la imagen mostrada en la Figura 4.4.

En la parte inferior de la etapa EX se observa que la salida del elemento **Branch** es 1, lo que denota que el salto tendrá lugar y, en consecuencia, la salida de la ALU se enviará como nueva entrada del **`pc`**. Sin embargo, fíjate en que en la

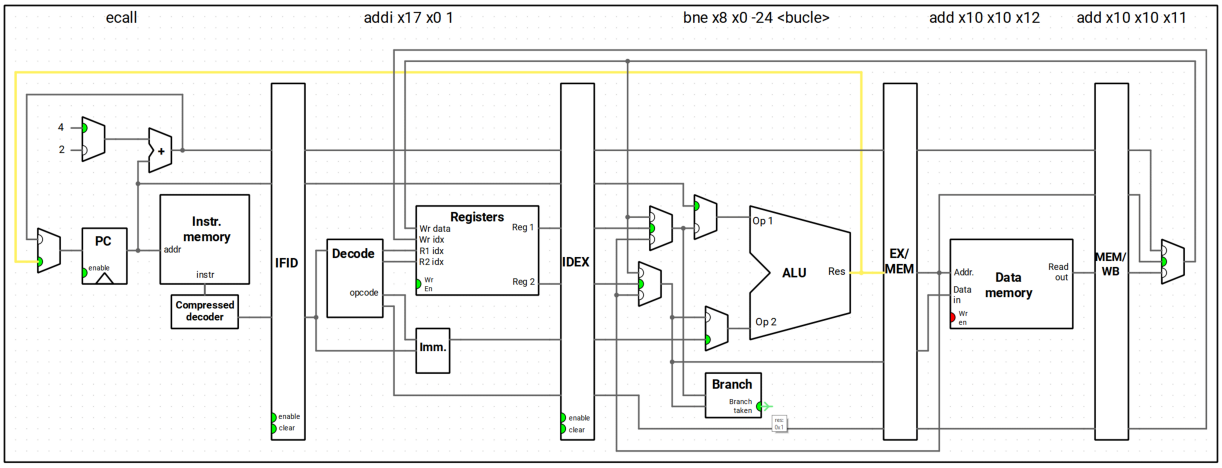


Figura 4.4. La CPU procesa instrucciones que hay tras la de salto, se captan y descodifican, y actualiza el `pc` a cada ciclo sin tener conocimiento de lo que ocurrirá como consecuencia del salto.

Los riesgos de control pueden presentar una configuración distinta según la implementación que se haga de la ISA RISC-V. Es posible, por ejemplo, adelantar el cómputo de la nueva dirección y verificación de si la condición de salto se cumple a la etapa ID, lo cual requeriría circuitería adicional y haría la CPU algo más compleja.

etapa de descodificación ya tenemos la instrucción `addi x17 x0 1` (generada por la seudoinstrucción `li a7, 1`), mientras que la instrucción `ecall`, la última del programa, ya ha sido captada de memoria. No obstante, en esta primera iteración del bucle esas dos instrucciones no habrá que ejecutarlas, ya que la instrucción `bne` devolverá el control a la etiqueta `bucle` para iniciar una nueva iteración.

Volvamos un par de ciclos atrás, al momento en que se capta la instrucción `bne` del programa. Al ciclo siguiente esa instrucción avanzará a la etapa ID. El captador no puede quedar ocioso, captará la instrucción que le indique el `pc` que, por defecto, será la situada 4 bytes más adelante. Otro ciclo más y esa nueva instrucción se transfiere a la etapa ID, una vez que `bne` alcanza la etapa EX, y el captador toma la siguiente instrucción. El funcionamiento de la CPU es el esperable, opera como lo haría con cualquier otra instrucción.

Es en el siguiente ciclo, tras verificar que el salto se produce, cuando surge el problema: en el cauce de ejecución hay sendas instrucciones que no deberían haberse procesado. Es necesario desecharlas y comenzar a captar a partir de la nueva dirección del `pc`. Esto da lugar a la situación mostrada en la Figura 4.5. Observa que las instrucciones `addi` y `ecall` han desaparecido del cauce. La indicación FLUSH en la parte superior de las etapas ID y EX denota que esas instrucciones se han descartado por haberse captado y descodificado de manera errónea.

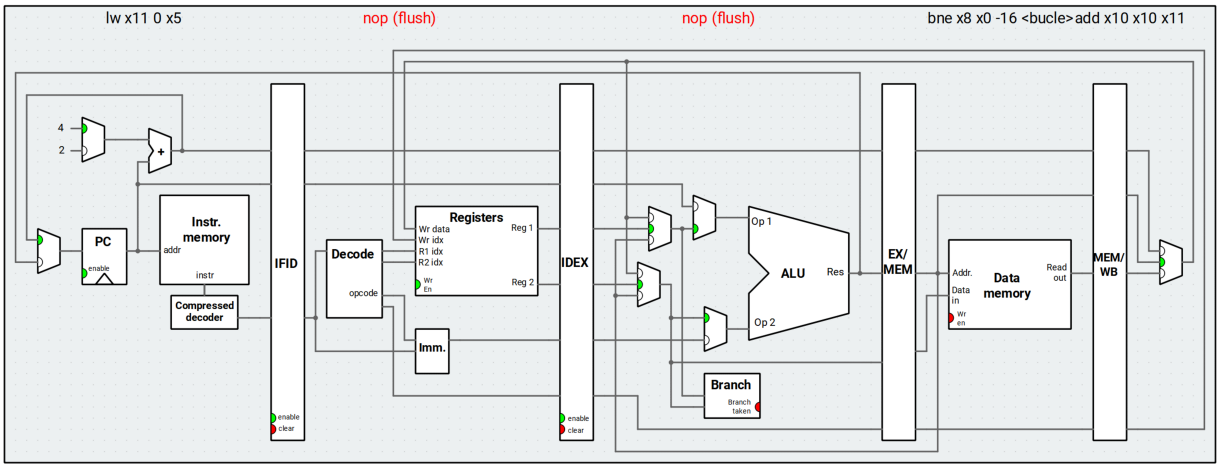


Figura 4.5. Las instrucciones que se encontraban en las etapas IF e ID se descartan, produciendo dos ciclos perdidos en el cauce, y se capta la primera instrucción tras actualizar el contador de programa.

El riesgo de control que acabamos de identificar no implica un mal funcionamiento del programa, ya que la CPU detecta la situación y, de manera automática, saca del cauce aquellas instrucciones que no deberían haberse empezado a procesar. No obstante, sí que produce una pérdida de rendimiento al ejecutar el programa, sobre todo si la escena descrita se produce con mucha frecuencia. Es algo que podemos saber examinando el cronograma de ejecución (véase la Figura 4.6): en nuestro caso son dos ciclos perdidos por cada iteración del bucle.

La cuantificación del número de veces que sucede un riesgo de control nos permitirá estimar el rendimiento perdido, ya que por cada riesgo hay dos ciclos perdidos. Para diez iteraciones, y al tener en cuenta que el último salto no se produce, tendríamos un total de 18 ciclos adicionales a los que teóricamente habría que ejecutar.

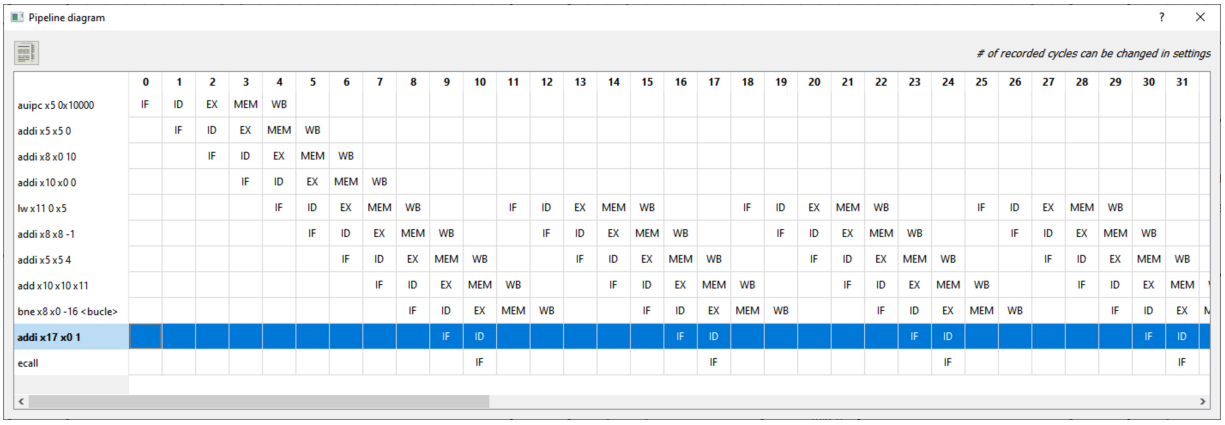


Figura 4.6. En el diagrama de ciclos se aprecia que por cada iteración del bucle hay una instrucción que se capta y descodifica de forma innecesaria, mientras otra se capta y luego también se descarta.

Al llegar a la última iteración, momento en que el componente que verifica la condición dará como salida cero tal y como se aprecia en la Figura 4.7, el **pc** no verá alterado su contenido, de forma que las dos últimas instrucciones, una ya captada y descodificada y la otra recién captada, terminarán su avance por el cauce de ejecución como sería de esperar.

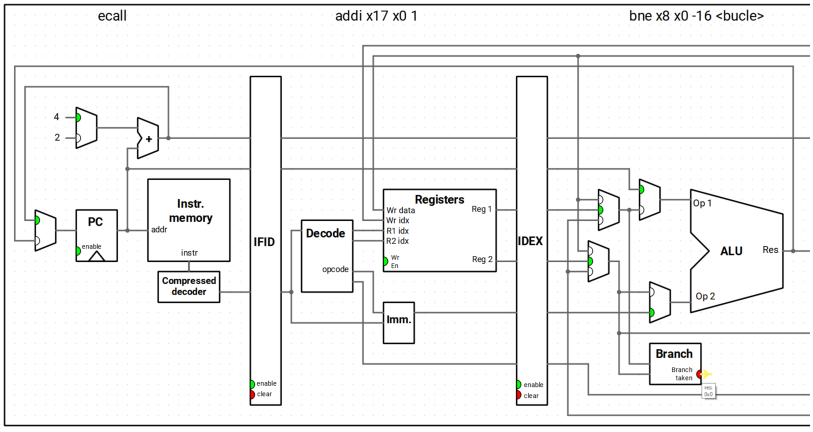


Figura 4.7. En la última iteración del bucle la condición de la instrucción **bne** no se cumple, de manera que el contenido del **pc** seguirá su avance habitual.

Imagina que nuestro sencillo programa tuviese que procesar no diez elementos, sino miles o millones de ellos —algo bastante habitual hoy en día— y que en cada iteración empleásemos dos ciclos más de los estrictamente necesarios. El número total sería considerable y, por tanto, hay espacio para introducir mejoras en el rendimiento si se consigue reducir el número de riesgos de control.

4.5. Mejoras hardware para minimizar los riesgos de control

En la práctica previa experimentamos diversas soluciones para reducir los riesgos de datos. Algunas de ellas eran mejoras en el hardware, con circuitería capaz de detectarlos y, si fuese preciso, reducir su impacto merced a los caminos de adelantamiento.

Las CPU cuentan asimismo con medios para detectar los riesgos provocados por saltos, de forma que nunca lleguen a completarse instrucciones que no deberían ejecutarse. El simulador Ripes dispone de este mecanismo, como hemos comprobado en los apartados previos. Al encontrar una instrucción de salto en el flujo de un programa es posible seguir distintas estrategias. En el caso de Ripes, esta consiste en actualizar el **pc** normalmente y seguir con la captación de nuevas instrucciones. En el momento en que se llega a la etapa de ejecución se determina si dichas instrucciones debían haberse captado o no y, si es preciso, se descartan en el siguiente ciclo.

Una estrategia alternativa sería detener la captación en cuanto entra en el cauce una instrucción de salto, reanudándola una vez que se conoce a ciencia cierta el valor que tendrá el **pc** una vez ejecutada. No obstante, esta estrategia no aporta beneficio alguno en cuanto al número de ciclos perdidos.

En todas las CPU modernas se cuenta con un elemento adicional: el **predictor de saltos**. Este se basa en un registro de lo que ha ocurrido al procesar cada una de las instrucciones de salto del programa en casos anteriores, por una parte, y una máquina de estados que determina si, al encontrarla de nuevo, lo probable es que el salto se produzca o no, por otra. Según esa predicción, el **pc** se actualiza —con la dirección de la siguiente instrucción según el flujo normal o bien con la indicada por el salto— en la etapa de captación, por lo que no se pierde ciclo alguno. Como es lógico, si la predicción hecha por el predictor de saltos resulta fallida la situación será similar a no tener ese elemento, ya que se habrán captado y descodificado instrucciones que habrá que descartar antes de captar las correctas.

Dado que en Ripes no tenemos predictor de saltos, un componente relativamente complejo que requiere bastante circuitería adicional, la única ayuda que nos aporta la CPU es la detección de los riesgos de control, pero no una vía para minimizar el impacto que tienen en el rendimiento.

4.6. ¿Cómo reducir los riesgos de control?

El análisis llevado a cabo nos lleva a concluir que la ejecución de un bucle durante muchas iteraciones conlleva una cierta pérdida de rendimiento. Sin contar con ayuda de la propia CPU, en forma de predictor de saltos, cabe preguntarse cuál sería la vía para reducir el número total de riesgos de control.

Si a más iteraciones del bucle se *pierden* más ciclos, la respuesta lógica sería reducir el número total de repeticiones. Pero, ¿cómo es posible hacer esto sin dejar de procesar la información con la que tenga que operar el programa?

Desenrollado de bucles

Cuando se indica que un programa ha de procesar N elementos, en nuestro caso diez valores enteros, siempre se asume que es necesario codificar un bucle que realice el trabajo sobre uno de los valores y, por tanto, se precisen N iteraciones. No obstante, nada nos impide procesar en cada vuelta del bucle dos elementos, tres o más, en cuyo caso el número de iteraciones se reduciría a $N/2$, $N/3$ o, en general, N/t, siendo t el número de elementos procesados a cada paso.

Lo habitual es tomar un valor t que sea divisor de N, de manera que el número de datos total se pueda dividir en porciones de igual tamaño. Para un vector de 12 elementos t podría ser 2, 3, 4 o 6.

El desenrollado de un bucle se efectúa, en primera instancia, tomando todas las instrucciones que hay en su cuerpo, excepto la del salto final, y repitiéndolas t veces. A pesar de que la longitud del programa —y por tanto su ocupación en memoria— será mayor, el tiempo preciso para realizar el trabajo será menor. Vamos a comprobarlo en la práctica: partimos del código del Listado 4.1, seleccionamos las cuatro instrucciones que hay tras la etiqueta `bucle`, las copiamos al portapapeles y las pegamos debajo, justo antes del salto condicional, de forma que obtenemos la versión mostrada en el Listado 4.2.

```
    .data

v: .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

    .text

    la t0, v
    li a0, 0
    li s0, 10

bucle:
    lw a1, 0, t0
    add a0, a0, a1
    addi t0, t0, 4
    addi s0, s0, -1

    lw a1, 0, t0
    add a0, a0, a1
    addi t0, t0, 4
    addi s0, s0, -1

    bnez s0, bucle

    li a7, 1
    ecall
```

Listado 4.2: Versión del programa con un desenrollado del bucle de factor dos (se procesan dos instrucciones por ciclo).

Dado que en cada iteración del bucle se procesan dos elementos del vector, en lugar de solo uno, la instrucción **bnez** pasa de ejecutarse diez veces a solo cinco. Esto conlleva reducir a la mitad el número de ciclos que se perdían en la versión original del programa.

Desenrollado y reordenado

La técnica de desenrollado que acabamos de conocer ya aporta una mejora en el rendimiento del programa, a pesar de ser muy simple: copiar un código y repetirlo las veces que sea necesario. En realidad, la mejora obtenida será mayor si, tras lo anterior, examinamos el código y evitamos los siguientes aspectos:

- Al duplicar un bloque de código en el que existen riesgos de datos, como puede ser una carga con una operación aritmética a continuación, también estamos duplicando esos riesgos.
- En la versión del programa del Listado 4.2 se observa que hay instrucciones repetidas, como la que resta 1 al contenido del registro **s0** en el que se lleva el control del número de repeticiones, de manera innecesaria. Podría restar 2 a dicho registro, sin más, y ahorraríamos una de las operaciones.

Cuanto mayor sea el factor de desenrollado, tantas más instrucciones habrá en el cuerpo del bucle, lo cual facilita la reordenación para evitar riesgos de datos, algo que no siempre es posible cuando en cada iteración se procesa un único elemento.

Aplicar la reordenación y eliminación de instrucciones redudantes puede forzarnos a emplear algunos registros adicionales, como se aprecia en una nueva versión del programa, la del Listado 4.3, en el que se cargan sendos valores del vector utilizando no solo **a1** sino también **a2** como registros temporales. Además, el avance del puntero **t0** se hace en un solo paso, sumando 8 en lugar de dos veces 4, por lo que en la instrucción de carga hay que indicar un desplazamiento distinto para cada elemento: `0` para leer el primero y `4` para el siguiente.

```
    .data

v: .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

    .text

    la t0, v
    li s0, 10
    li a0, 0

bucle:
    lw a1, 0, t0
    lw a2, 4, t0
    addi s0, s0, -2
    addi t0, t0, 8
    add a0, a0, a1
    add a0, a0, a2
    bnez s0, bucle

    li a7, 1
    ecall
```

Listado 4.3: Versión del programa tras desenrollar y reordenar para reducir los riesgos de datos.

Esta versión del programa no solo es más compacta que la del desenrollado inicial, y por tanto precisa menos memoria para almacenarse, sino que también se ejecutará en bastantes menos ciclos como resultado de eliminar varias instrucciones —que se fusionan en una sola como es la actualización de contador **s0** y del puntero **t0**— y evitar los riesgos de datos que se producían.

Ejecuta las tres versiones del programa y, tras cada ejecución, examina el cronograma para identificar los ciclos perdidos tanto por riesgos de control como de datos. ¿Es posible introducir alguna mejora adicional en el código?

Comparativa de rendimientos

La identificación de los riesgos de control, y su reducción mediante el desenrollado de bucles, es una herramienta más que se añade a nuestra batería de soluciones de optimización del código, en la que ya contábamos con los caminos de adelantamiento y el reordenado de instrucciones. La Tabla 4.1 resume el número de ciclos y CPI para cada una de las versiones del programa empleado en estos ejercicios.

Tabla 4.1. Número de ciclos y CPI de cada una de las versiones del programa.

Versión del programa	Ciclos	CPI
Original (sin desenrollar)	90	1.61
Desenrollado	75	1.47
Desenrollado y reordenado	55	1.34

4.7. Ejercicios propuestos

Los ejercicios propuestos a continuación te permitirán poner en práctica las técnicas que acaban de describirse, así como analizar las mejoras en rendimiento que se consiguen con el desenrollado de bucles y la combinación con el reordenado de código. En dichos ejercicios siempre emplearemos la configuración de procesador **5-stage processor** que nos ofrece Ripes, de forma que siempre tengamos detección de riesgos de datos y caminos de adelantamiento.

Ejercicio 4.1. Almacenar en memoria un vector con los valores `7, 4, 23, 12, 6, 20, 17, 8, 3, 10, 11, 23` y mostrar por consola la media aritmética. Para ello se ha de calcular su suma y también contar cuántos valores hay en el vector, como se indicaba en el ejercicio 2.5. A continuación, realizar las siguientes acciones:

a) Ejecuta el programa y anota el número de ciclos y CPI.
b) Examina el cronograma de ejecución para identificar dónde se producen los riesgos de control.
c) Aplica un desenrollado de factor dos, ejecuta el programa y anota ciclos y CPI.
d) Examina el cronograma de ejecución para identificar dónde aparecen riesgos de datos.
e) Aplica reordenado de código para evitar todos los riesgos de datos posibles, ejecuta el programa y anota ciclos y CPI.
f) Calcula la ganancia de rendimiento obtenida con cada versión del programa respecto a la original.

Ejercicio 4.2. Partiendo del mismo programa del ejercicio 4.1, repetir los pasos indicados pero aplicando un desenrollado del bucle con factor tres.

Ejercicio 4.3. Partiendo del mismo programa del ejercicio 4.1, repetir los pasos indicados pero aplicando un desenrollado del bucle con factor cuatro.

Ejercicio 4.4. Almacenar en memoria un vector con los valores `7, 4, 23, 12, 6, 20, 17, 8, 3, 10, 11, 23` y mostrar por consola la suma de los números impares. A continuación, realizar las siguientes acciones:

a) Ejecuta el programa y anota el número de ciclos y CPI.
b) Examina el cronograma de ejecución para identificar dónde se producen los riesgos de control.
c) Aplica un desenrollado de factor dos, ejecuta el programa y anota ciclos y CPI.
d) Examina el cronograma de ejecución para identificar dónde aparecen riesgos de datos.
e) Aplica reordenado de código para evitar todos los riesgos de datos posibles, ejecuta el programa y anota ciclos y CPI.

f) Calcula la ganancia de rendimiento obtenida con cada versión del programa respecto a la original.

Ejercicio 4.5. Partiendo del mismo programa del ejercicio 4.4, repetir los pasos indicados pero aplicando un desenrollado del bucle con factor tres.

Ejercicio 4.6. Partiendo del mismo programa del ejercicio 4.4, repetir los pasos indicados pero aplicando un desenrollado del bucle con factor cuatro.

Arquitectura superescalar 5

5.1. Planificación

- **Objetivo.** Analizar los beneficios en rendimiento de una arquitectura superescalar y las implicaciones para la generación del código.
- **Temporización:**
  - **Trabajo en laboratorio**: descripción de la configuración del cauce dual de Ripes con ejecución por dos vías en paralelo con planificación estática. Cómo escribir el código para optimizar la ejecución y consecuencias de no hacerlo. Resolución de dudas.
  - **Trabajo autónomo:** ejercicios de reordenamiento de código para aprovechar la configuración superescalar con planificación estática.
  - **Validación en laboratorio**: test y ejercicios evaluables.

5.2. Configuración del procesador

En prácticas previas hemos usado varias de las configuraciones de procesador que nos ofrece el simulador Ripes. Entre las versiones de 32 bits encontramos una, en el último lugar según se ve en la Figura 5.1, que se diferencia porque cuenta con un cauce de ejecución con una arquitectura muy distinta. Se activa con la opción **6-stage dual-issue processor**. Al seleccionarla es importante que leamos la descripción que se facilita a la derecha. Esta será clave, como veremos después, para preparar el código de forma que aproveche al máximo este cauce de ejecución superescalar.

En la presente práctica emplearemos esta nueva configuración, así como la **5-stage processor** ya conocida y que nos servirá como punto de referencia para analizar las ganancias de rendimiento.

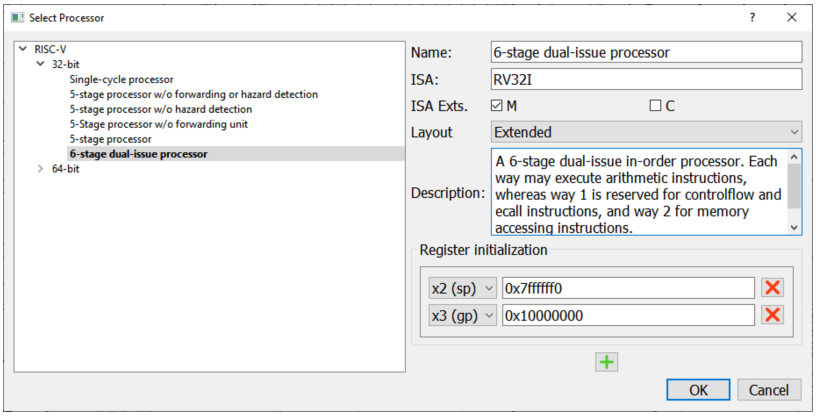


Figura 5.1. Ripes nos ofrece una configuración de CPU con arquitectura superescalar de seis etapas y cauce de ejecución dual con planificación estática.

Esta configuración de Ripes simula un procesador RISC-V superescalar de dos vías —lo cual significa que se procesan dos instrucciones en paralelo— y con planificación estática. Por tanto, las instrucciones se ejecutan en el orden en que aparecen originalmente en el programa. Además, las dos vías de ejecución están especializadas de la siguiente forma:

- **Vía 1:** ejecuta instrucciones aritméticas, de salto y `ecall`.
- **Vía 2:** ejecuta instrucciones aritméticas, de carga y de almacenamiento.

5.3. Arquitectura superescalar

Hablamos de *arquitectura superescalar* para referirnos a las CPU que tienen la capacidad de procesar más de una instrucción de manera simultánea. Para ello, como es lógico, las unidades funcionales asociadas a cada una de las etapas de ejecución han de ajustarse adecuadamente. Por regla general se cuenta con dos alternativas para conseguirlo:

- **Incremento de la capacidad de la unidad:** ciertas unidades pueden realizar su trabajo sobre más de una instrucción en un solo ciclo con un incremento de su capacidad. Un ejemplo de ello es el captador: en lugar de transferir una instrucción desde memoria a la CPU e incrementar el **pc** en 4 bytes, tras un rediseño tiene la capacidad de llevar dos instrucciones en paralelo e incrementar el **pc** en 8 bytes.
- **Duplicación de unidades:** unidades más complejas requieren soluciones también más elaboradas, de las cuales la más común es la replicación de su funcionalidad. Una ALU puede realizar una operación sobre dos operandos en un ciclo de reloj, para hacer lo mismo con dos instrucciones se precisan dos ALU.

En realidad las CPU de los microprocesadores actuales no se limitan a duplicar, sino que cuentan con múltiples unidades funcionales de cada tipo, por ejemplo sumadores, multiplicadores y divisores, de forma que se procesan cuatro, seis o más operaciones de ese tipo en paralelo.

Además de la duplicación de unidades funcionales o su incremento de prestaciones, el diseño superescalar suele demandar asimismo la incorporación de estructuras adicionales que permiten almacenar temporalmente la información necesaria para el procesamiento de n instrucciones en paralelo. También hay que considerar la sincronización entre las etapas que, como puede suponerse, se hace más compleja.

Planificación estática

Una CPU superescalar con planificación estática tiene algunas de sus unidades funcionales mejoradas o replicadas, lo cual le permite procesar más de una instrucción en una misma etapa y ciclo. El orden en que las instrucciones aparecen en el programa es vital para aprovechar esa arquitectura.

La herramienta encargada de disponer las instrucciones en el código ensamblador en el mejor orden para una cierta arquitectura de CPU es el compilador. Las opciones de optimización de este analizan las interacciones entre instrucciones y las unidades funcionales de cada etapa —aparte de los riesgos de datos y control— y generan el mejor código posible.

La planificación estática implica que la CPU procesará las instrucciones **en orden**, según aparecen en el código del programa. En consecuencia, las instrucciones de ese programa han de disponerse de forma que sea posible explotar las posibilidades que ofrece la arquitectura. Por ejemplo, si solo se cuenta con una unidad de transferencia de datos a y desde memoria, en la secuencia de órdenes no deberían aparecer juntas dos instrucciones de ese tipo. De hacerlo, la CPU tendría que demorar una de ellas mediante, como es habitual, una detención o *stall*.

Escribir código ensamblador para una CPU superescalar con planificación estática exige del programador, la persona que escribirá las instrucciones en ensamblador, un conocimiento detallado de la arquitectura para poder aprovechar sus capacidades.

Planificación dinámica

Si la arquitectura superescalar de una CPU cuenta con planificación dinámica, lo cual provoca que el diseño del procesador sea mucho más complejo que en el caso anterior, en cierta medida el aprovechamiento de las unidades funcionales extendidas recae en el propio cauce de ejecución. Una configuración así precisa, además, componentes como los ya estudiados en teoría, entre ellos:

- **Cola de instrucciones:** la estructura de datos donde el captador almacena las instrucciones que transfiere desde la memoria.

- **Ventana de instrucciones:** una o más estructuras de datos[1] con instrucciones que están a la espera de contar con sus operandos y la unidad funcional correspondiente para poder ser ejecutadas.
- **Buffer de reorden:** una o más estructuras de datos que sirven para garantizar que la escritura de resultados, tanto en registros como en memoria, es consistente.
- **Predictor de saltos:** el elemento descrito en la Sección 4.5 del capítulo previo (pág. 43) es indispensable en una planificación dinámica, ya que el coste de los riesgos de control es mucho mayor que en un arquitectura no superescalar.

1: Cuando se tiene una ventana de instrucciones por unidad funcional suele denominárseles *estaciones de reserva*.

Apoyándose en estas estructuras de datos, y con la circuitería adicional para tomar decisiones con la información que contienen, la CPU superescalar tiene la capacidad de ejecutar las instrucciones de manera **no ordenada**.

En general, en cuanto una instrucción captada tiene disponibles sus operandos —y la unidad funcional que la debe ejecutar está disponible— se procede a ejecutarla. El resultado que produzca no se lleva directamente al banco de registros o a memoria, a la espera de poder finalizar esa parte manteniendo la consistencia.

5.4. La arquitectura superescalar de Ripes

La configuración superescalar que nos ofrece el simulador Ripes tiene dos vías de ejecución y opera con planificación estática. Por tanto, es capaz de procesar dos instrucciones en paralelo por ciclo en el mejor de los casos, siempre y cuando el orden de esas instrucciones sea el adecuado. Veamos con algo más de detalle los aspectos específicos de esta arquitectura.

Elementos en el cauce de ejecución

Al observar el diagrama del cauce de ejecución (véase la Figura 5.2) de esta arquitectura lo primero que debería llamarnos la atención es que existen seis etapas en él, en lugar de las cinco a las que estábamos habituados hasta el momento. Existe una nueva etapa, situada entre la de descodificación y la de ejecución, denominada II (*Instruction Issue*) o de **emisión de instrucciones**.

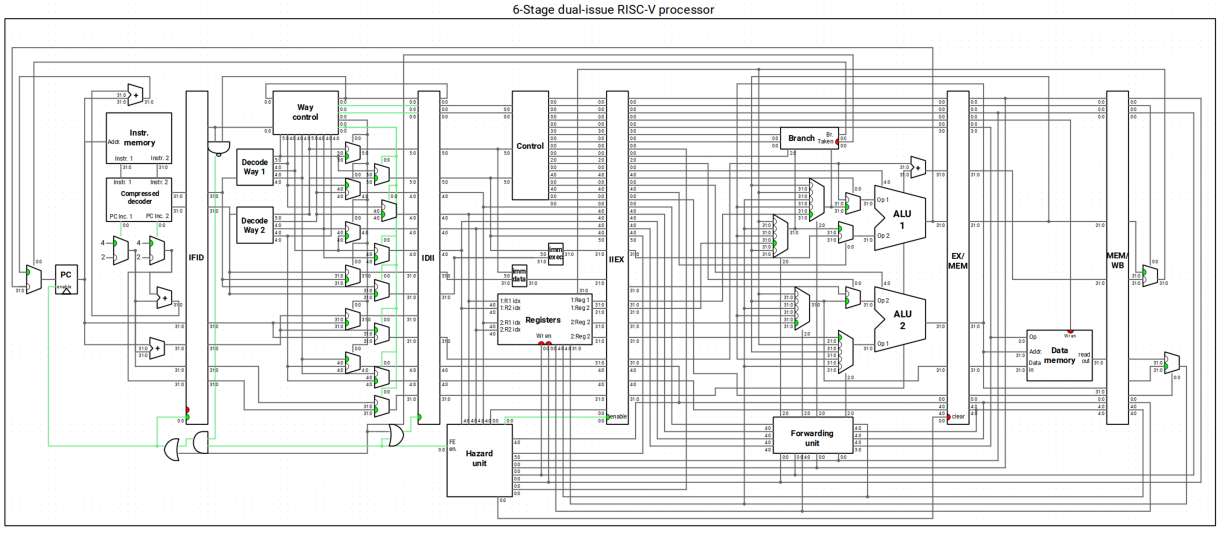


Figura 5.2. Vista del cauce de ejecución segmentado de la configuración superescalar de Ripes.

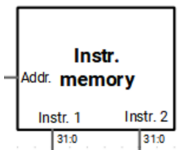


Figura 5.3. Dos buses de 32 bits permiten transferir dos instrucciones desde la memoria cache a la CPU.

La unidad de control de vía puede redirigir la primera instrucción captada al segundo camino y la segunda al primero si fuese necesario —por ejemplo para situar un acceso a memoria en el segundo si apareciese en el primero conjuntamente con una instrucción aritmética— y no existen riesgos que lo impidan.

En general, una instrucción de salto no podrá ser ejecutada en paralelo con ninguna otra por el potencial riesgo de control que ello conlleva, dado que no se cuenta con un predictor de saltos.

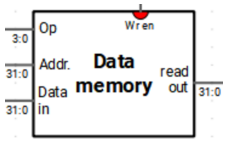


Figura 5.4. La unidad de acceso a memoria solo cuenta con un bus de datos de 32 bits, por lo que únicamente se puede escribir o leer un dato en cada ciclo.

Examinemos con atención el anterior esquema del cauce de ejecución, a fin de anotar las diferencias fundamentales que se observan en cada una de sus etapas respecto al cauce de cinco etapas que ya conocemos.

- **IF:** para captar dos instrucciones por ciclo se precisan dos conjuntos de líneas de 32 bits salientes desde la memoria de instrucciones (véase el detalle de la Figura 5.3 al margen), en lugar de solo una como teníamos hasta ahora. Por otra parte, hay dos valores constantes `4` que, tras pasar por un multiplexor, se dirigen al sumador cuya salida está conectada a la entrada del `pc`, de forma que se actualice el contador de programa en el número de bytes adecuado.
- **ID:** la etapa de descodificación cuenta ahora con dos unidades **Decode**, una para cada vía de ejecución del cauce. De esta forma las dos instrucciones captadas en el ciclo t se descodifican en el ciclo $t + 1$ en un solo paso. La unidad **Way control** determina la vía de ejecución por la que se enviará cada una de estas instrucciones según su tipo.
- **II:** esta nueva etapa —de emisión de instrucciones para su ejecución— cuenta con una unidad de control que no existe en la configuración de cinco etapas, así como un componente ampliado de detección de riesgos. Si no se detecta ningún riesgo las dos instrucciones se envían, una por cada vía del cauce, a la etapa siguiente y se procede con su ejecución. Hay, no obstante, múltiples causas por lo que esto podría no ser así y que solo una de las instrucciones pueda avanzar. Las condiciones a verificar se detallan en la posterior Subsección 5.4 Tipos de riesgos en el cauce superescalar.
- **EX:** la peculiaridad de esta etapa estriba en que cuenta con dos ALU, en lugar de solo una como en las configuraciones de CPU que ya habíamos utilizado. Esto permite ejecutar dos instrucciones aritméticas —o bien una aritmética y otra que requiera el cálculo de una dirección (de salto, de lectura o de escritura en memoria)— en un solo ciclo. Solo existe un componente **Branch** para evaluar la condición de salto, de ahí que no puedan gestionarse dos instrucciones de este tipo de forma simultánea.
- **MEM:** salvo por la existencia de un mayor número de líneas de comunicación entre la etapa previa y la posterior, la etapa de acceso a memoria de este cauce superescalar es muy similar a la estándar. Solo hay un componente de acceso a memoria, con los buses necesarios para seleccionar la dirección, la operación a efectuar (lectura o escritura) y enviar o recibir el dato correspondiente. Esta es la razón de que no sea posible ejecutar de forma simultánea dos transferencias a o desde memoria.
- **WB:** esta última etapa cuenta con un mayor número de líneas de comunicación con el banco de registros, situado en la etapa de emisión de instrucciones, lo cual le permite escribir en un mismo ciclo los resultados a almacenar en sendos registros. Los potenciales riesgos se habrán verificado con anterioridad, de forma que al llegar a este punto las instrucciones pueden finalizarse de manera segura.

Para comprender mejor el funcionamiento de este complejo cauce de ejecución es aconsejable tomar un programa sencillo y ejecutarlo paso a paso, al tiempo que se analiza lo que ocurre en cada una de las etapas. Para ello es también importante conocer los tipos de riesgos que pueden producirse.

Tipos de riesgos en el cauce superescalar

En un cauce superescalar con planificación estática, una CPU que no dispone de circuitería adicional para decidir de forma dinámica qué ejecutar en cada momento según los recursos —unidades funcionales y operandos— disponibles, pueden producirse distintos tipos de riesgos. Algunos de ellos, como los de datos

RAW y control, los conocemos, mientras que otros es la primera vez que los encontramos, caso de los riesgos estructurales y los de datos tipo WAR. En general, los tipos de riesgos que contempla el simulador Ripes son los siguientes:

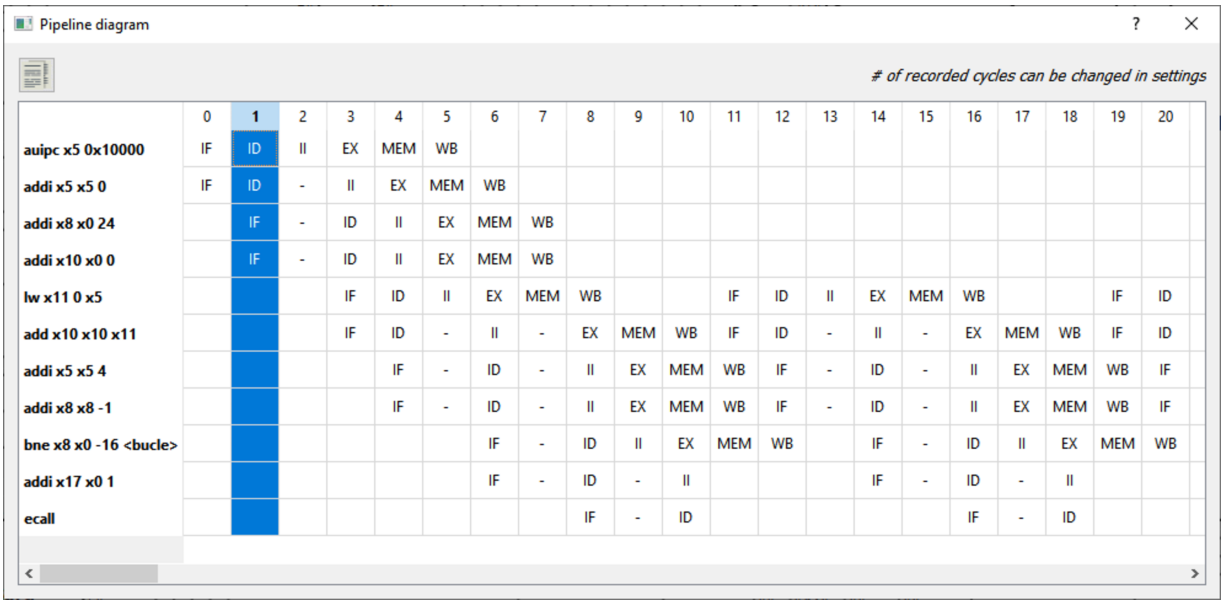
Los detalles sobre cómo Ripes analiza los potenciales riesgos y decide si es posible emitir las dos instrucciones captadas o solo una de ellas los encontramos en el propio código fuente del simulador, en concreto en el módulo rv6s_dual_waycontrol.h.

- **Riesgo de control:** al procesarse una instrucción de salto que viene por la primera vía de descodificación se emite para ejecución solo ella, al tiempo que se genera una detención para la instrucción captada por la segunda vía. De esta forma se evita que llegue a completarse una instrucción que, por el flujo real del programa, no tendría que haberse ejecutado. La detención da tiempo a que la condición del salto se verifique, en un ciclo posterior, y que se descarte la siguiente instrucción si procede.
- **Riesgo estructural:** surge cuando se descodifican de forma conjunta dos instrucciones de acceso a memoria, dos instrucciones de salto o bien dos `ecall`. Ya que solo hay una vía capaz de ejecutar cada uno de estos tipos de operaciones, la arquitectura no está diseñada para procesarlas en paralelo, se emite primero una y luego la otra según el orden original del programa, enviándola por la vía que corresponda a la unidad funcional adecuada.
- **Riesgo RAW/WAR:** aparece cuando hay dos instrucciones de tipo aritmético y una de ellas tiene como destino del resultado un registro que actúa como operando en la otra. Si es la primera la que escribe, el riesgo es RAW, mientras que si es la segunda, sería WAR. Ante esta situación siempre se emite la primera instrucción, según el orden del programa, y un ciclo después la segunda.

Ante ausencia de riesgos las dos instrucciones captadas se emiten y pasan a la etapa de ejecución. Serán dos instrucciones aritméticas sin dependencias de datos entre ellas o bien una combinación de instrucciones que no presenten riesgo estructural: aritmética/acceso a memoria, aritmética/salto, salto/acceso a memoria, etc.

Estructura del cronograma

El cronograma que ofrece Ripes —en la ventana **Pipeline diagram** que ya conocemos— mostrará una estructura diferente de la ejecución del programa, tal y como se aprecia en la Figura 5.5.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
auipc x5 0x10000	IF	ID	II	EX	MEM	WB															
addi x5 x5 0	IF	ID	-	II	EX	MEM	WB														
addi x8 x0 24		IF	-	ID	II	EX	MEM	WB													
addi x10 x0 0		IF	-	ID	II	EX	MEM	WB													
lw x11 0 x5				IF	ID	II	EX	MEM	WB			IF	ID	II	EX	MEM	WB			IF	ID
add x10 x10 x11				IF	ID	-	II	-	EX	MEM	WB	IF	ID	-	II	-	EX	MEM	WB	IF	ID
addi x5 x5 4					IF	-	ID	-	II	EX	MEM	WB	IF	-	ID	-	II	EX	MEM	WB	IF
addi x8 x8 -1					IF	-	ID	-	II	EX	MEM	WB	IF	-	ID	-	II	EX	MEM	WB	IF
bne x8 x0 -16 <bucle>							IF	-	ID	II	EX	MEM	WB		IF	-	ID	II	EX	MEM	WB
addi x17 x0 1							IF	-	ID	-	II				IF	-	ID	-	II		
ecall									IF	-	ID						IF	-	ID		

Figura 5.5. Vista parcial del cronograma de ejecución en la arquitectura superescalar de Ripes.

En lugar de la apariencia en escalera propia de la ejecución de un cauce segmentado, con una instrucción activa en cada etapa/ciclo, aquí nos encontramos en un ciclo dado con dos instrucciones en una misma etapa. En el primer ciclo las dos primeras instrucciones del programa están en la etapa IF. Al ciclo siguiente esas pasan a la etapa ID y se captan dos más. En teoría podrían llegar a tenerse hasta doce instrucciones distintas en el cauce, aunque en la práctica esto se consigue pocas veces.

El cronograma de ejecución nos servirá, como ya lo hizo al tratar los riesgos de datos y de control en las prácticas previas, para identificar las detenciones y reducirlas al mínimo posible, de forma que se maximice la ocupación del cauce y, por tanto, el rendimiento conseguido.

La razón de que el cauce presente múltiples detenciones, representadas con el carácter **-** en el cronograma, se debe a los distintos tipos de riesgos antes enumerados. ¿Por qué en el tercer ciclo no pasan a la etapa II las dos instrucciones ya captadas y descodificadas: **`auipc`** y **`addi`**? Es fácil comprobar que ambas escriben en el registro **`x5`** el resultado que producen. Por ello, y a pesar de que este cauce tiene la capacidad de procesar dos instrucciones aritméticas en paralelo merced a sus dos ALU, es preciso enviar a ejecución la primera y luego, un ciclo después, la segunda, evitando la escritura simultánea en la etapa WB.

5.5. Preparación del código

A partir del conocimiento adquirido sobre el cauce superescalar con planificación estática que nos ofrece el simulador Ripes, veamos ahora cuáles serían sus efectos en la ejecución de un programa y cómo tendríamos que preparar el código para aprovechar sus capacidades lo máximo posible.

Programa inicial

Comenzamos con un programa sencillo, similar al usado en prácticas previas, cuya finalidad es obtener la suma de los valores almacenados en un vector. En este caso la longitud es de 24 elementos. El código inicial de ese programa sería el del Listado 5.1.

Listado 5.1: Código para obtener la suma de un vector con 24 elementos

```
1     .data
2
3  v: .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
4         11, 12, 1, 2, 3, 4, 5, 6, 7, 8,
5         9, 10, 11, 12
6
7     .text
8
9     la t0, v
10    li s0, 24
11    li a0, 0
12
13 bucle:
14    lw a1, 0, t0
15    add a0, a0, a1
16    addi t0, t0, 4
17    addi s0, s0, -1
18    bnez s0, bucle
19
20    li a7, 1
21    ecall
```

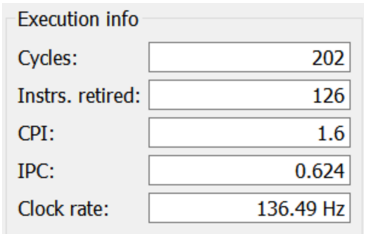


Figura 5.6. Ejecución del programa en el cauce segmentado con cinco etapas.

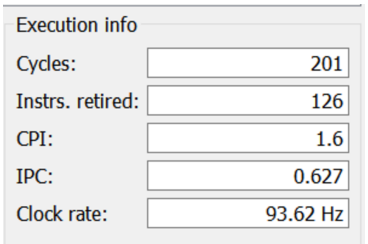


Figura 5.7. Ejecución del programa en el cauce superescalar con seis etapas.

Tras introducir el código en el editor de Ripes, selecciona la configuración de CPU estándar con cauce de cinco etapas, la misma usada en prácticas anteriores, y ejecuta el programa. Esta prueba nos servirá como punto de referencia para comparar las posibles mejoras de rendimiento. El resultado que deberías obtener es el mostrado en la Figura 5.6: 202 ciclos y un CPI de 1.6.

Esperaríamos que el simple hecho de cambiar la configuración de CPU al cauce superescalar nos aportara ya una mejora, puesto que procesa las instrucciones en parejas siempre que no existan riesgos entre ellas. Los caminos de adelantamiento operan como lo hacían en la anterior configuración —en realidad su gestión es más compleja porque los caminos de *bypass* se duplican— y evitarán muchas de las detenciones por riesgos de datos.

Al ejecutar con la nueva configuración, sin embargo, observamos (véase la Figura 5.7) que el rendimiento es casi idéntico: se tardan 201 ciclos y el CPI no ha cambiado. El CPI nos dice que, si el programa procesase muchos más elementos, el número de ciclos total sería similar al de la configuración no superescalar si no cambiamos nada.

En realidad el resultado podría ser incluso peor. Ciertos programas tardarán más en ejecutarse en el cauce superescalar que en el segmentado estándar. La razón es que este último tiene cinco etapas, mientras que el superescalar cuenta con seis. Si el orden de las instrucciones no es el adecuado, y sin una correcta planificación raramente lo será, la emisión será siempre individual, por lo que el número final de ciclos y el CPI serán superiores.

Desenrollado del código

El primer paso en la preparación de nuestro código será, en este caso concreto en el que partimos de un programa relativamente corto, aplicar las técnicas de desenrollado de bucles que ya conocemos. Esto nos abrirá dos vías de mejora:

En este ejercicio aplicaremos un desenrollado de factor cuatro, de forma que en 6 iteraciones —en lugar de 24— se procese el vector completo.

- **Reducir riesgos de control:** los riesgos de control en una arquitectura superescalar siempre tienen un mayor impacto, ya que el número de instrucciones que es preciso descartar si finalmente se produce el salto es mayor: dos por etapa. Si se reduce el número de iteraciones del bucle también disminuirá el de riesgos de control, es algo que verificamos de forma experimental en la práctica previa.
- **Mayor número de instrucciones:** al desenrollar el bucle contaremos con más instrucciones en el cuerpo del mismo. Un bucle con un cuerpo muy pequeño ofrece pocas posibilidades a la hora de reordenar instrucciones, por lo que será difícil realizar una planificación estática de cómo fluirán por el cauce. Con más instrucciones, esa planificación resulta más sencilla.

Identificar riesgos de planificación

Tras el desenrollado, la planificación estática continúa con la identificación de riesgos que impedirían la ejecución concurrente de pares de instrucciones. Con este fin aplicamos el siguiente procedimiento:

1. Agrupamos las instrucciones ejecutables de nuestro programa por parejas de manera que sea fácil identificar las que se captarían conjuntamente. La vía más fácil es dejar una línea en blanco entre cada par de instrucciones.
2. Si en una pareja hay dos instrucciones de acceso a memoria, o dos saltos, hemos identificado un riesgo estructural. Las parejas con dos instrucciones aritméticas solo representan un problema si comparten un operando que sea destino en una de ellas y origen en la otra o destino en ambas. Serían riesgos de tipo RAW y WAR/WAW.
3. Mediante la ventana del cronograma localizamos dónde se están introduciendo detenciones que no correspondan a los dos casos previos, a fin de determinar otros riesgos que existan en el código.
4. La ejecución paso a paso del programa, mientras se mantiene visible el cauce de ejecución, también es una vía para observar cuándo dos instrucciones captadas y descodificadas no se emiten de manera conjunta.

Los riesgos de control también aparecerán tanto en la ventana del cronograma de ejecución como en la vista del cauce, al ejecutar paso a paso, pero estos serán inevitables en la configuración de Ripes que usamos.

Durante este proceso hemos de tener presente que hay ciertas seudoinstrucciones de RISC-V que, como sabemos, producen dos instrucciones reales, y que estas podrían provocar una detención. Para poder separarlas sería preciso cambiar la seudoinstrucciones por las instrucciones correspondientes, lo cual nos permitirá tratarlas de forma individual.

Reordenar las instrucciones

Aplicamos el procedimiento que acaba de describirse a nuestro programa, agrupando las instrucciones en parejas y, donde sea preciso, llevando a cabo una reordenación que, como siempre, no debe afectar al resultado del programa. En

este proceso las herramientas esenciales han sido el cronograma de ejecución y la propia vista del cauce. El resultado final es el código que aparece en el Listado 5.2. Las tres primeras parejas de instrucciones se han señalado explícitamente con un comentario. En total hay ocho parejas.

Listado 5.2: Código para obtener la suma de un vector con 24 elementos tras desenrollar y reordenar

```
    .data

    v:  .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
    11, 12, 1, 2, 3, 4, 5, 6, 7, 8,
    9, 10, 11, 12

    .text

        lui t0, %hi(v)  # Pareja
        li s0, 6        # 1

        li a0, 0        # Pareja
bucle:  lw a1, 0, t0    # 2
        lw a2, 4, t0    # Pareja
        addi s0, s0, -1 # 3

        add a0, a0, a1
        lw a1, 8, t0

        add a0, a0, a2
        lw a2, 12, t0

        add a0, a0, a1
        addi t0, t0, 16

        add a0, a0, a2
        bnez s0, bucle

        li a7, 1
        ecall
```

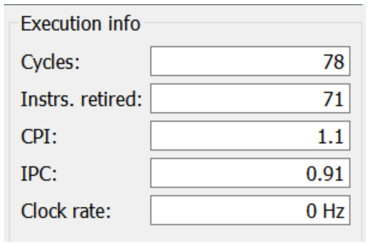


Figura 5.8. Ejecución del programa desenrollado y reordenado en el cauce superescalar con seis etapas.

Observa que se ha sustituido la orden `la t0, v`, que producía dos instrucciones aritméticas —**`auipc`** y **`addi`**— con un riesgo RAW entre ellas, por **`lui`** `t0,%hi(v)` que, en este caso concreto, produce el mismo resultado ya que el identificador `v` está al inicio del segmento de datos.

En el resto del código, tras haber aplicado un desenrollado del bucle tal y como se indicó, solo se ha intercambiado el orden de algunas de las instrucciones para evitar riesgos allí donde aparecían.

Al ejecutar esta nueva versión del programa, el resultado obtenido será el que se aprecia en la Figura 5.8. La mejora de rendimiento es muy destacable y representa un 158 % de ganancia respecto a antes de aplicar la planificación descrita. El CPI ha bajado hasta un 1.1, lo que significa que casi hemos alcanzado el ideal de finalizar una instrucción por ciclo.

En realidad, esta nueva versión del programa todavía se encuentra **lejos del óptimo** que podría alcanzarse. Si lo ejecutamos paso a paso, y observamos el cauce de ejecución en la vista **Processor** de Ripes, observaremos que todavía existen múltiples riesgos. Algunos de ellos se eliminarían con una nueva reordenación. El emparejamiento correcto de instrucciones sin dependencias ni riesgos podría exigir la introducción de alguna instrucción **`nop`**. Tras completar dicho trabajo sería posible bajar el tiempo de ejecución hasta 63 ciclos, con **un CPI de solo 0.818** y una ganancia del 221 % respecto a la versión original del programa.

Tu programa puede fallar a pesar de no tener errores

Precauciones a considerar

A medida que el diseño de una CPU incluye características más avanzadas, como es la ejecución simultánea de múltiples instrucciones, el diseño se hace más complejo y, en consecuencia, las probabilidades de incurrir en errores también se incrementa. En los procesadores más avanzados de los últimos años, con diferentes arquitecturas como x64 o ARM, se han identificado fallos [6] que han dado lugar a distintos tipos de ataques.

[6]: Abu-Ghazaleh et al. (2019), «How the spectre and meltdown hacks really worked»

Listado 5.3: Programa que suma los cinco elementos de un vector

```
    .data

v: .word 45, 24, 25, 57, 23

    .text

    la t0, v
    li s0, 5
    li a0, 0  # Inicializar acumulador de suma
    li a1, 0  # y registro de carga de valores

bucle:
    lw a1, 0, t0
    add a0, a1, a0
    addi s0, s0, -1
    addi t0, t0, 4

    bnez s0,bucle

# Al llegar aquí a0 debería tener el valor 174
```

Analiza el código del Listado 5.3. La primera parte inicializa un puntero, en el registro **t0**, para recorrer el vector de cinco elementos. Asimismo, se da un valor inicial al contador (en **s0**), el acumulador de la suma (en **a0**) y el registro que se usará para ir cargando los valores del vector (en **a1**). En realidad, esta última operación no sería precisa ya que la instrucción **lw** del bucle sobrescribiría ese contenido. No obstante, la ejecución de este programa no produce un resultado correcto cuando se ejecuta en la configuración superescalar de Ripes.

Si ejecutamos este programa en Ripes, tras haber activado la configuración de procesador **5-stage processor** o **Single-cycle processor**, observaremos que, una vez finalizado, el registro **a0** contiene el valor 174. Sin embargo, al cambiar a la configuración **6-Stage Dual-Issue Processor** y ejecutar de nuevo veremos que el resultado ya no es correcto: **a0** no tiene el valor esperado. Puesto que no se ha modificado nada en el código, es evidente que la configuración del procesador ha alterado la forma en que se ejecuta.

Source code — Input type: Assembly / C

```
1     .data
2
3  v: .word 45, 24, 25, 57, 23
4
5     .text
6
7     la t0, v
8     li s0, 5
9     li a0, 0  # Inicializar acumulador de suma
10    li a1, 0  # y registro donde se cargan valores
11
12    bucle:
13        lw a1, 0, t0
14        add a0, a1, a0
15        addi s0, s0, -1
16        addi t0, t0, 4
17
18        bnez s0,bucle
19
20 # Al llegar aquí a0 debería tener el valor 174
```

WB WB EX II II

Figura 5.9. En el editor apreciamos cómo dos instrucciones que escriben en el mismo registro llegan de forma simultánea a la etapa WB.

A fin de identificar el problema, procedemos a ejecutar el programa paso a paso y observamos en el editor y la ventana de registros qué ocurre. El ciclo clave es el mostrado en la Figura 5.9.

Se ha señado en verde la coincidencia de dos instrucciones en la etapa WB del cauce, lo cual no sería un problema si el registro de destino de la operación de escritura no fuese el mismo.

Como puede apreciarse, tenemos dos instrucciones que escriben en el registro **a1** de forma simultánea. Ambas llegan al mismo tiempo a la etapa WB, algo que no debería ocurrir. En la vista del cauce (véase la Figura 5.10) aparecen dichas instrucciones. Para otras posteriores el simulador sí ha identificado dependencias e introducido paradas, pero este riesgo WAW, en el que la instrucción **addi** `x11 x0 0` escribe en **a1** después de **lw** `x11 0 x5` y, por tanto, se pierde el valor cargado desde memoria, no es adecuadamente controlado por el simulador.

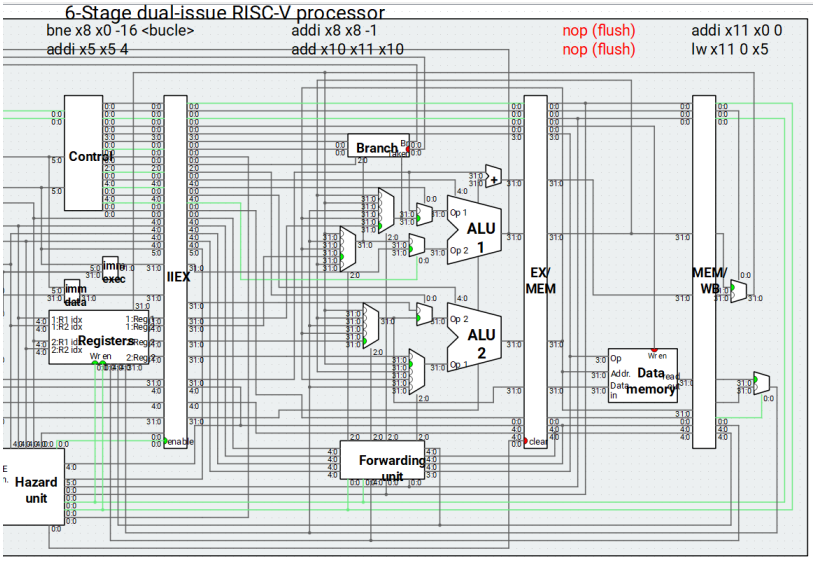


Figura 5.10. En la vista del cauce de ejecución se aprecian las dos líneas de datos que conectan la etapa WB con el banco de registros.

Con esta prueba determinamos que la configuración superescalar de Ripes tiene un fallo, lo que nos lleva a ser precavidos y no asumir que el resultado de un programa será correcto a pesar de que no haya errores en el código.

5.6. Ejercicios propuestos

Los ejercicios propuestos a continuación te permitirán poner en práctica la técnica de planificación estática que acaba de describirse, así como analizar las mejoras en rendimiento que se consiguen al ejecutar el código en un cauce superescalar. En dichos ejercicios siempre emplearemos la configuración de procesador **5-stage processor** que nos ofrece Ripes como punto de referencia, tras lo cual activaremos la opción **6-stage dual-issue processor** para la ganancia de rendimiento.

Ejercicio 5.1. Copia de `https://bit.ly/3pchzlQ` la lista de 100 números enteros aleatorios que aparecen, pégalos en el editor de Ripes y sepáralos con comas, introduciéndolos como un vector en tu programa. A continuación escribe el código necesario para obtener la suma y mostrarla en la consola. A partir de ese código lleva a cabo las siguientes acciones:[2]

2: La lista de números puede introducirse en Ripes sin necesidad de separarlos con comas, basta con anteponer delante de cada línea la palabra clave `.word` como es habitual.

a) Ejecuta el programa con la configuración de cinco etapas no superescalar y anota el número de ciclos y CPI.
b) Cambia a la configuración de seis etapas superescalar, ejecuta de nuevo el programa y anota el número de ciclos y CPI.
c) Aplica un desenrollado de factor cuatro a fin de incrementar el número de instrucciones disponibles en el programa.
d) Empareja las instrucciones y marca aquellos pares en los que surjan riesgos estructurales. Reordena para evitarlos, de forma que no haya parejas de dos instrucciones de acceso a memoria o de salto.
e) Identifica otro tipo de riesgos, como los RAW y WAR/WAW, si es preciso ayudándote del cronograma y la vista del cauce mientras ejecutas paso a paso. Reordena donde sea posible para reducir esos riesgos.
f) Ejecuta la versión final del programa, anota número de ciclos y CPI.
g) Resume en una tabla los ciclos, CPI y ganancia de rendimiento de cada versión respecto a la anterior.

Ejercicio 5.2. Copia de `https://bit.ly/3pchzlQ` una nueva lista de 100 números enteros y prepáralos para usarlos desde un nuevo programa. Este deberá calcular por separado la media aritmética de los valores que ocupan posición impar y la de los que ocupan posiciones pares, mostrando ambas sumas al final por la consola. Completado el programa repite los pasos indicados para el ejercicio 5.1.

Ejercicio 5.3. Copia de `https://bit.ly/3pchzlQ` una nueva lista de 100 números enteros y prepáralos para usarlos desde un nuevo programa. Este deberá calcular por separado la media aritmética de aquellos valores que sean impares, por una parte, y la de los que sean pares, por otra, mostrando ambas sumas al final por la consola. Completado el programa repite los pasos indicados para el ejercicio 5.1.

Ejercicio 5.4. Toma el código de la versión final del programa del ejercicio 5.1 y amplia el desenrollado para procesar cinco elementos del vector por iteración, en lugar de cuatro. Analiza si se producen riesgos adicionales y, si es preciso, reordena de nuevo el código. Ejecuta el programa y compara el número de ciclos y CPI. ¿Merece la pena el desenrollado de factor cinco en lugar de cuatro?

Memoria caché 6

6.1. Planificación

- **Objetivo.** Comprender el impacto que la configuración de memoria caché tiene en el rendimiento de la CPU.
- **Temporización:**
  - **Trabajo en laboratorio**: explicación de las opciones que ofrece Ripes para ajustar la configuración tanto de la caché de datos como de la de instrucciones. Análisis del cambio en rendimiento de un programa según la configuración de memoria caché. Identificación de los parámetros más relevantes. Resolución de dudas.
  - **Trabajo autónomo:** ejercicios de análisis de rendimiento de la ejecución de un programa atendiendo a la configuración de la memoria caché de datos y el patrón de accesos del código.
  - **Validación en laboratorio**: test y ejercicios evaluables.

6.2. Configuración del procesador

El objetivo principal de esta última práctica será estudiar cómo afecta el patrón de accesos a memoria de un cierto código, conjuntamente con la configuración de la memoria caché de datos con la que cuenta la CPU, en el rendimiento obtenido al ejecutar un cierto programa, sin tomar en consideración otros aspectos como los riesgos de datos, de control o estructurales.

La configuración interna del cauce de ejecución no es de interés para este estudio. Por dicha razón seleccionaremos el **Single-cycle processor** para todas las ejecuciones. Las estimaciones de rendimiento se efectuarán con pautas similares a las descritas en la parte teórica, al considerar el número de aciertos, de fallos y la ratio de acierto como medidas básicas. Nuestros datos de referencia para esa tarea serán los siguientes:

- Un acceso a memoria caché, ya sea para leer un dato o para escribirlo —aunque en este último caso dependerá de cómo se configure el modo de escritura en caché—, se efectúa en **un ciclo de reloj**. En consecuencia no se demora el cauce de ejecución, ya sea secuencial, segmentado o superescalar, siempre y cuando se trate de un acierto.
- Un acceso a memoria principal precisaría **100 ciclos de reloj**, por lo que será dos órdenes de magnitud más lento que el trabajo con la caché. Esto implica que el cauce de ejecución se detendrá, total o parcialmente según su configuración, hasta que el dato sea devuelto desde memoria o escrito en ella.
- Un fallo de acceso a la caché implica la comprobación inicial, para saber si el dato se encuentra ya en ella o es preciso transferirlo antes, y luego el acceso a la memoria principal, así como la actualización de la caché con los nuevos datos, por lo que en total tendríamos **102 ciclos de reloj**.

En resumen, un *hit* o acierto conllevaría un ciclo de reloj mientras que un *miss* o fallo conllevaría emplear 100 ciclos más. No hay más que obtener el producto del número de fallos por 100 para saber cuántos ciclos de más se están empleando por los accesos a memoria.

Los modernos microprocesadores cuentan con tres niveles de caché, denominados L1, L2 y L3 según su cercanía a la CPU. El tiempo de acceso a la L1 suele ser de entre 2 y 4 ciclos, mientras que para la L3 precisaría de 70 a 80 ciclos. Aquí trabajaremos con un modelo simplificado que solo cuenta con una L1 muy rápida y luego la memoria principal.

6.3. La vista caché de Ripes

De las vistas ofrecidas por el simulador Ripes, accesibles mediante los botones situados en la barra lateral izquierda, solo nos queda por conocer la denominada **Cache**. Esta da paso a los elementos que se aprecian en la Figura 6.1, cuyo detalle conoceremos en esta sección. En la mencionada figura se tiene un programa cargado del que se han ejecutado varios ciclos a fin de tener algunos datos de acceso a la caché para poder analizarlos.

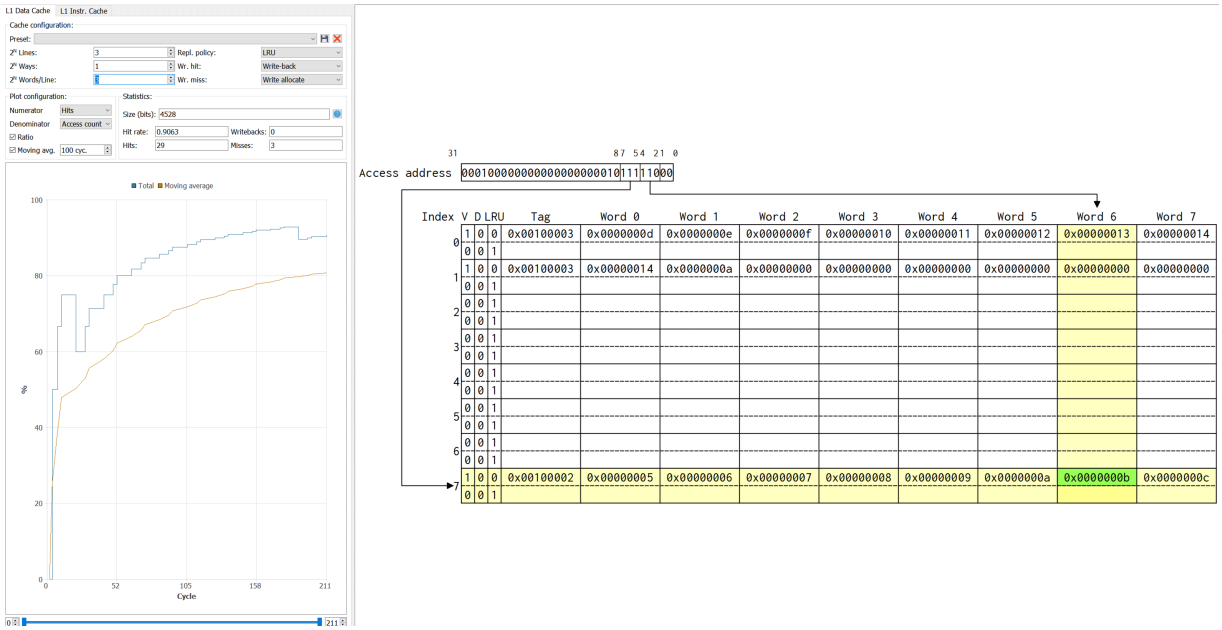


Figura 6.1. Vista **Cache** del simulador Ripes. En la parte superior izquierda se encuentran los controles de configuración de la caché. Debajo los conteos de aciertos, fallos y ratio de aciertos, así como la gráfica obtenida al representarlos. La parte derecha está ocupada por el diagrama de la caché, en este caso la de datos, con la información relativa a un acceso concreto durante la ejecución paso a paso de un programa.

Terminología

Antes de pasar a detallar las distintas partes de la vista **Cache** de Ripes, algo esencial para los ejercicios desarrollados a continuación y propuestos al final, es importante tener en cuenta la terminología que se emplea en este simulador. Los conceptos básicos son los siguientes:

- **Bloque:** la memoria caché contará con un número determinado de bloques de un tamaño fijo. Ese tamaño determinará asimismo en cuántos bloques se estructurará la memoria principal. En Ripes a los bloques se les denomina **Líneas** (**Lines**) y su número se expresa como el exponente de una potencia en base 2. En general, cuando en esta práctica se haga referencia a una línea de caché nos estaremos refiriendo a un bloque.
- **Tamaño de bloque:** es también el exponente de una potencia en base 2 y, en el caso de Ripes, viene expresado como número de palabras de 32 bits en lugar de hacerlo como número de bytes. En la interfaz se indica como **Words/Line** (número de palabras por líneas). Dado que una palabra son 2^2 bytes, este será el factor por el que se multiplicará el tamaño de bloque de Ripes para convertirlo a bytes.
- **Asociatividad:** el nivel de asociatividad de la caché en Ripes se controla con el elemento **Ways** (vías o planos de caché) que, como los anteriores, también es el exponente de una potencia en base 2. Un valor 0 hace que la correspondencia de bloques de memoria principal a caché sea **directa**.

Con el valor 3 en **Words/Line** se tendrían $2^3 = 8$ palabras por bloque o, lo que es lo mismo, bloques de 32 bytes (8 palabras × 4 bytes/palabra=32 bytes). Una memoria principal de 16 Kb se estructuraría en 512 bloques de ese tamaño. Cada transferencia de esa memoria a la caché o viceversa se haría siempre en bloques de 32 bytes, mientras que el acceso desde la CPU a la caché se realizaría, como es habitual, en forma de palabras.

Un valor mayor que 0 generaría una caché **asociativa por conjuntos**. Para tener una memoria **completamente asociativa** el número de líneas debe ser 2^0.

Los tres parámetros anteriores determinarán el tamaño de la memoria caché. Si la configuramos con 2^3 palabras por línea, 2^3 líneas y 2^1 vías tendríamos una caché de $2^{3+3+1+2} = 2^9 = 512$ bytes asociativa por conjuntos. Dado que siempre operamos con exponentes de potencias de 2, el tamaño de la caché siempre tendrá un número de bytes potencia de 2.

Configuración de la caché de datos

La caché de instrucciones es solo de lectura, no se modifica el código durante la ejecución, y los accesos suelen ser siempre secuenciales, salvo cuando se ejecuta un salto, por lo que su configuración es más simple que en el caso de la caché de datos.

La vista **Cache** de Ripes cuenta con dos páginas: **L1 Data Cache** y **L1 Instr. Cache**. Operaremos solo sobre la primera, a fin de configurar la caché de datos como nos convenga. Esto implica no solo fijar su tamaño, en los apartados **Lines**, **Ways** y **Words/Line** ya descritos, sino también las estrategias que se emplearán para la sustitución de bloques y la realización de escrituras en memoria. Con este fin recurriremos a los apartados de la columna derecha de la Figura 6.2.

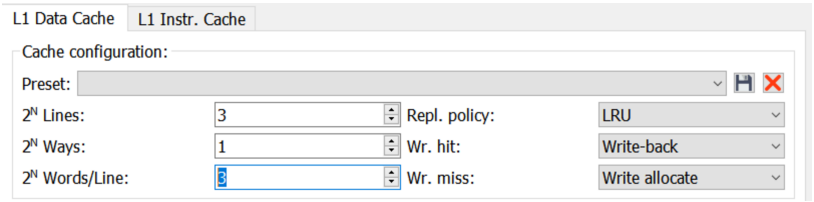


Figura 6.2. Controles para la configuración de la memoria caché ofrecidos por el simulador Ripes.

Cada una de las tres listas desplegables nos permite elegir:

La memoria principal tiene muchos más bloques que la caché, por lo que en ocasiones será preciso *hacer sitio* en esta última para alojar bloques de la primera que sean demandados por el programa.

- **Estrategia de reemplazo:** Ripes contempla dos estrategias de alojamiento de nuevos bloques de memoria principal en la caché. Con la opción **Random** la elección será aleatoria, mientras que con **LRU** (*Least Recently Use*) la nueva línea reemplazará a aquella que hace más tiempo que no se ha usado.
- **Modo de escritura en acierto:** para escribir un dato en un bloque de memoria que está alojado en la caché se puede elegir entre el modo **Write-back** —que escribe dicho dato en la caché y marca el bloque como invalidado— o bien **Write-through** —que realiza la escritura en memoria principal de manera inmediata— usándose la primera opción por defecto.
- **Modo de escritura en fallo:** al escribir un dato en un bloque de memoria que no está alojado en la caché es posible elegir entre los modos **Write allocate** y **No write allocate**. El primero traerá el bloque a la caché, realizando la sustitución que proceda, mientras que el segundo escribe en memoria principal sin llevar a cabo el reemplazo del bloque.

Cada vez que se modifica la configuración Ripes reinicia la CPU, lo cual permite volver a ejecutar el programa y observar los cambios en la efectividad de la caché de forma inmediata.

La lista **Preset**, en la parte superior de la Figura 6.2, nos permite elegir entre tres configuraciones predefinidas. Los botones situados a la derecha también facilitan la creación de nuevas parametrizaciones y su eliminación.

Interpretación del diagrama de memoria

Cada cambio realizado sobre la configuración de la memoria caché reajustará el diagrama que ocupa la parte derecha de la vista. A medida que se ejecuta el programa, con cada ciclo, dicho diagrama nos permite examinar el estado de la caché y saber cuándo se produce un acierto y cuándo un fallo. Por ello necesitamos saber cómo interpretar la información que nos ofrece.

Supongamos que tenemos configurada la memoria caché con 2^3 líneas de un tamaño de 2^3 palabras y 2^1 planos. Esto implica que tendremos 8 columnas —correspondientes a las 8 palabras por línea— y 16 filas que representan las 8

líneas cada una con dos planos, como se aprecia en la Figura 6.3. El índice que aparece en el margen izquierdo denota el número de fila o bloque, desde la fila 0 que está en la parte superior hasta la 7 en la inferior. La línea 1 aparece resaltada porque es a la que se ha accedido en la última instrucción ejecutada. Es una fila con dos planos que, aunque aparecen en el diagrama como dos filas adyacentes, físicamente no tendrían esa disposición. Lo importante es entender que se tiene una memoria asociativa por conjuntos, de forma que para cada fila hay dos vías o planos independientes, en los que podría mapearse un bloque de memoria principal.

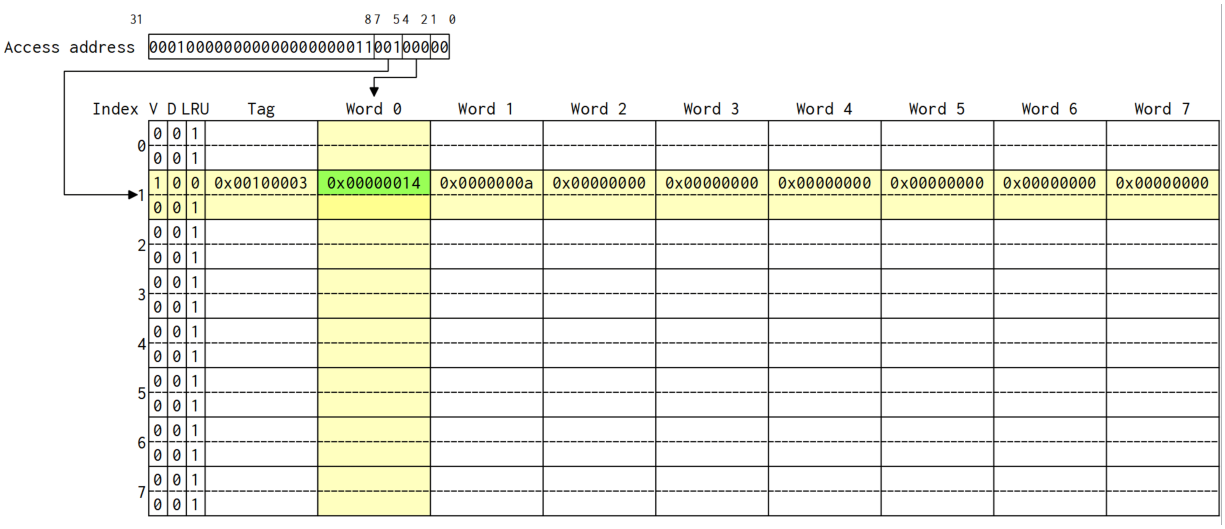


Figura 6.3. En el diagrama de la memoria caché podemos ver su estructura, los indicadores asociados a cada línea, si el acceso en curso ha provocado un acierto o un fallo, etc.

En la parte superior del diagrama se encuentra el apartado **Access address**. Este nos muestra cómo se está extrayendo de la dirección de memoria a la que se quería acceder, la identificación de la fila de caché donde podría estar alojada. En la Figura 6.3 se ha accedido a la dirección `0x10000320`. Los 32 bits de la misma se dividen, de derecha a izquierda, de la siguiente manera:

Para entender cómo funciona este esquema de direccionamiento lo mejor es que tomes la dirección a la que se quiere acceder, `0x10000320` en este caso, y conviertas cada dígito hexadecimal en un grupo de cuatro bits. A continuación, de derecha a izquierda, ve separando los grupos que se indican para obtener los índices de byte, palabra y línea. Lo que queda será la etiqueta.

- **Byte:** los dos bits menos significativos denotan el byte dentro de la palabra. Se precisan dos bits ya que cada palabra tiene cuatro bytes.
- **Palabra:** puesto que cada bloque aloja 8 palabras, se precisan 3 bits para indicar qué palabra es la que interesa dentro de una línea.
- **Línea:** los tres bits siguientes seleccionan una de las 8 filas con que cuenta la caché.
- **Etiqueta:** los restantes 24 bits forman la etiqueta, que permitirá saber si la línea de memoria principal a la que se quiere acceder se encuentra en la caché o no, comparándola con la columna **Tag** de cada uno de los planos de la línea indicada por los tres bits previos.

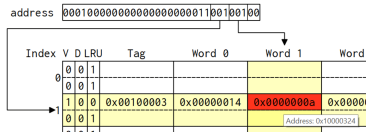


Figura 6.4. Representación de un fallo en el acceso a caché.

Las líneas que parten de la dirección nos indican cuál sería la línea y columna en la que el bloque de memoria principal que contiene esa dirección podría estar alojado en la caché. Al comparar la etiqueta de dicha dirección con la contenida en **Tag** se sabe si es un acierto —el bloque está en la caché— o bien un fallo. En este último caso el recuadro asociado aparecería en rojo (véase la Figura 6.4) en lugar de en verde.

Además del índice y la etiqueta, en el margen izquierdo del diagrama hay otros indicadores (podrían no ser visibles en algunas configuraciones de la caché) cuyo significado es el que se indica a continuación:

- **V**: un bit que denota si la línea contiene un bloque de memoria válido (1) o no (0). Al iniciar la ejecución este bit tiene el valor 0 para todas las líneas. A medida que se trasladan bloques desde memoria principal, va tomando el valor 1.
- **D**: un bit que indica si la línea contiene datos por escribir en memoria (1) —porque se ha efectuado una escritura en modo **Write-back**— o no (0).
- **LRU**: uno o más bits, según el número de planos con que cuenta cada línea. Se usa para saber qué línea ha sido más recientemente usada (0) y toma valores > 0 según el historial de accesos. Al traer una nueva línea desde memoria principal, este campo determinará cuál de los planos de la línea será sustituido: aquel que tenga el mayor valor en este campo.

Cuando el número de planos es 2^0, y por tanto cada línea solo tiene un plano posible, la sustitución por un nuevo bloque traído desde memoria principal es inmediata a partir del índice de fila, razón por la que no existe el campo LRU.

Gráfica de rendimiento y estadísticas

La última parte de esta vista **Cache** está ubicada en la parte inferior izquierda, con un panel de varios campos en los apartados **Plot configuration** —que facilita la configuración de la gráfica de rendimiento— y **Statistics**, por una parte, y la mencionada gráfica bajo ambos, por otra, tal y como se aprecia en la Figura 6.5.

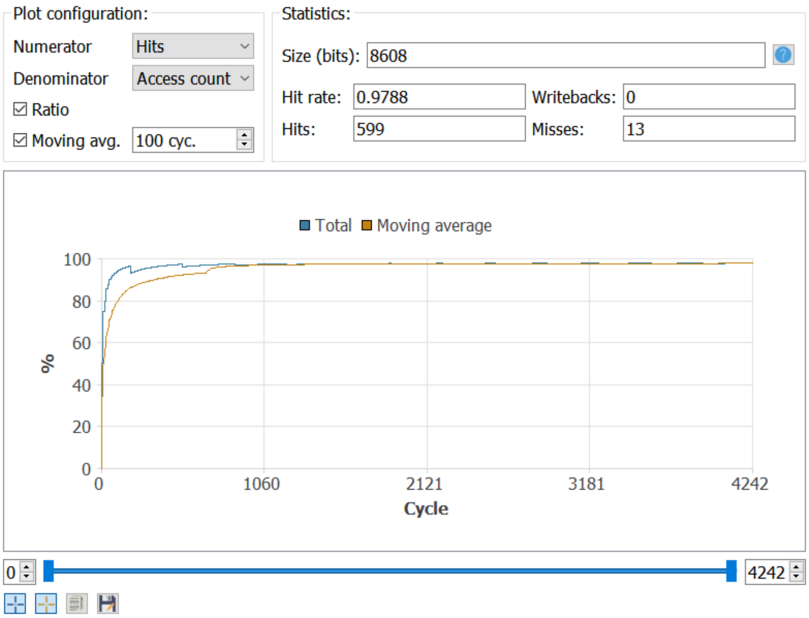


Figura 6.5. La gráfica es configurable con los controles situados en su parte superior y se actualiza ciclo a ciclo.

Los datos que nos interesan son los del bloque **Statistics**, en concreto los siguientes valores:

- **Hit rate**: la ratio de aciertos expresada como un valor en el intervalo $[0, 1]$. En la Figura 6.5 el valor `0.9788` se interpreta como un 97.88 % de aciertos en los accesos a memoria.
- **Hits**: un contador con el número absoluto de accesos que han resultado en aciertos.
- **Writebacks**: contador con el número de escrituras realizadas en la caché.
- **Misses**: un contador con el número absoluto de accesos que han resultado fallidos y, por tanto, han implicado traer desde memoria principal a la caché una nueva línea.

El análisis conjunto del número de aciertos y fallos, que se resume en la ratio de aciertos, es la información a tomar como referencia a la hora de comparar el rendimiento de un programa según su patrón de accesos a memoria y de la configuración de caché de la CPU en que se ejecute.

6.4. La caché en accesos de lectura desde memoria

Ahora que conocemos los medios que nos ofrece el simulador Ripes para ajustar y analizar el comportamiento de la memoria caché, estudiaremos con un caso práctico cuál sería la mejor configuración —de entre varias posibles— así como la influencia que los patrones de acceso a memoria del programa tienen en el rendimiento.

Sumar los elementos de una matriz

Partimos de un programa, el del Listado 6.1, que opera sobre una matriz cuyo número de filas y columnas es configurable a través de dos valores almacenados en la propia memoria. La subrutina `direccion` se encarga de calcular la dirección de cualquier elemento de la matriz. El cuerpo del programa se limita a calcular la suma de los elementos de la matriz, para lo cual recorre todas las columnas de cada fila con dos bucles anidados.

Listado 6.1: Programa que suma los valores contenidos en una matriz

```
    .data
filas: .word 10
v:  .word  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
    .word  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
    .word  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
    .word  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
    .word  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
    .word  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
    .word  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
    .word  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
    .word  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
    .word  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
columnas: .word 20

    .text

    li a0, 0
    lw t1, filas
    lw t2, columnas
bucle:
    call direccion
    lw a1, 0, t0
    add a0, a0, a1
    addi t1, t1, -1
    bnez t1, bucle

    lw t1, filas
    addi t2, t2, -1
    bnez t2, bucle

    li a7, 1
    ecall
    li a7, 10
    ecall

# Calcula la dirección de un elemento de la matriz
# Entradas: t1 -> fila, t2 -> columna, ambas entre 1 y max
# Salidas: t0 -> dirección o bien 0 si la fila > filas en matriz
direccion:
    lw s0, filas
    bgt t1, s0, cero
    lw s0, columnas
    addi s1, t1, -1
    mul t0, s0, s1
    addi s1, t2, -1
    add t0, t0, s1
    li s0, 4
    mul t0, t0, s0
    la s0, v
    add t0, t0, s0
    ret

cero:
    li t0, 0
    ret
```

Rendimiento según la configuración de la caché

¿Qué configuración de memoria caché sería la que ofrecería mejor rendimiento para un programa así? Vamos a considerar dos tamaños de caché posibles: 256 y 512 bytes. Probaremos con tres configuraciones distintas por cada tamaño: una caché con correspondencia directa, otra completamente asociativa y una tercera asociativa por conjuntos. El tamaño de línea/bloque lo mantendremos constante en 32 bytes (2^3 palabras), de forma que habrá que ajustar solo el número de líneas y el de planos para dar lugar a los tres casos mencionados: 256 bytes / 32 bytes por bloque = 8 bloques que se repartirán entre líneas y planos, según el tipo de caché que interese. Para 512 bytes serían 16 bloques.

Introducimos la primera configuración en los ajustes de la caché de Ripes y ejecutamos el programa. Anotamos el número de aciertos, de fallos y la ratio entre ambos. Repetimos la operación para todas las configuraciones. Las anotaciones obtenidas serían las de la Tabla 6.1

Bytes	Líneas	Vías	Palabras	Hits	Misses	Ratio
256	2^3	2^0	2^3	432	190	0.6945
256	2^0	2^3	2^3	428	194	0.6881
256	2^2	2^1	2^3	470	152	0.7556
512	2^4	2^0	2^3	560	62	0.9003
512	2^0	2^4	2^3	587	35	0.9437
512	2^2	2^2	2^3	587	35	0.9437

Tabla 6.1. Resultados obtenidos por las distintas configuraciones de caché al ejecutar el código original del programa mostrado en el Listado 6.1. La diferencia entre el mejor caso y el peor sería de $190 - 35 = 165$ fallos $\rightarrow 165 \times 100 = 16\,500$ ciclos más de ejecución.

Es fácil apreciar que el tamaño de la caché tiene un impacto importante en el rendimiento. La matriz que procesa el programa ocupa 800 bytes, por lo que una caché de 256 bytes ofrece, en el mejor de los casos, un 76 % de tasa de aciertos. Al duplicar el tamaño de la caché, incluso la peor configuración sobrepasa el 90 % de aciertos.

Para una caché pequeña la configuración asociativa por conjuntos es apreciablemente más eficiente que la de correspondencia directa y la completamente asociativa. Para el tamaño más grande, sin embargo, ambas configuraciones asociativas ofrecen el mismo rendimiento. Dado que al nivel del hardware la asociativa por conjuntos es menos compleja —además de más rápida que la completamente asociativa al trabajar con un espacio menor a la hora de buscar la etiqueta— esta sería la elección obvia.

Patrón de accesos a memoria e impacto en la caché

En el apartado anterior siempre se ha ejecutado el mismo programa, por lo que el patrón de accesos a memoria es idéntico. Habría que preguntarse si en ese patrón se rompe en algunos casos el **principio de localidad** en el que se basa el óptimo funcionamiento de una memoria caché.

La elaboración de la traza consiste en ejecutar el programa paso a paso, hasta procesar varios ciclos del cuerpo del bucle, y anotar un índice de orden y la dirección a la que se accede.

Para analizar el patrón de accesos a memoria que sigue nuestro programa hemos de realizar una traza solo de las instrucciones de carga, dado que no realizamos almacenamientos, y de esta forma obtener el orden en que se ejecutan esos accesos y las direcciones de memoria implicadas. El resultado de esta traza —tras tres iteraciones del cuerpo del bucle— es el que aparece en la Figura 6.6.

En la traza tenemos dos accesos iniciales en ciclos consecutivos. Estas lecturas se repiten en las parejas de ciclos (3,4), (6,7), (9,10), etc., en un patrón que se repite tantas veces como iteraciones ejecute el bucle. Está claro que hay una **localidad temporal** en estos dos accesos, ya que siempre se leen esas posiciones en ciclos contiguos. No existe, sin embargo, **localidad espacial**, ya que en el primero de los ciclos siempre se accede a la dirección `0x10000000` y en el segundo a `0x10000324`.

```
    li a0, 0
    lw t1, filas  # Fila        1) 0x10000000
    lw t2, columnas  # Columna  2) 0x10000324
bucle:

    call direccion
    lw a1, 0, t0    5) 0x10000320  8) 0x100002d0  11) 0x10000280
    add a0, a0, a1
    addi t1, t1, -1
    bnez t1, bucle

    lw t1, filas
    addi t2, t2, -1
    bnez t2, bucle

    li a7, 1
    ecall
    li a7, 10
    ecall

direccion:
    lw s0, filas       3,6,9) 0x10000000
    bgt t1, s0, cero
    lw s0, columnas    4,7,10) 0x10000324
    addi s1, t1, -1
```

Figura 6.6. Generamos el patrón de accesos a memoria de nuestro programa mediante una traza de la ejecución de operaciones de carga y las direcciones correspondientes.

Si hacemos los cálculos descritos antes, con independencia de que la caché sea de 256 o 512 bytes, los bloques de memoria que contienen esas direcciones corresponden a líneas distintas. Lo ideal sería que, puesto que siempre se leerán en ciclos consecutivos, ocupasen direcciones adyacentes —en una misma línea de caché— lo cual garantizaría que la segunda carga siempre será un acierto de acceso a caché.

```
1 columnas: .word 20
2 filas: .word 10
```

Conseguir localidad espacial en este caso es muy sencillo: solo hay que cambiar la disposición de `filas` para que esté en una dirección de memoria cercana a `columnas`. Tomamos el primer dato, que está delante de la matriz, y lo llevamos detrás, de forma que estos identificadores queden como se aprecia en el fragmento de código mostrado al margen.

Hecho este cambio, el patrón de accesos a memoria cambiará y se accederá, por cada ciclo del programa, primero a la dirección `0x10000324` y después a la `0x10000320`. ¿Cómo afectará esto al rendimiento al ejecutar el código? No tenemos más que repetir la ejecución con las mismas configuraciones empleadas antes y recoger los datos mostrados en la Tabla 6.2.

Tabla 6.2. Resultados tras mejorar la localidad espacial de dos datos. Se han usado las mismas configuraciones que con el programa original.

Bytes	Líneas	Vías	Palabras	Hits	Misses	Ratio
256	2^3	2^0	2^3	457	165	0.7347
256	2^0	2^3	2^3	421	201	0.6768
256	2^2	2^1	2^3	472	150	0.7588
512	2^4	2^0	2^3	577	45	0.9277
512	2^0	2^4	2^3	591	31	0.9502
512	2^2	2^2	2^3	591	31	0.9502

Si bien hay una mejora general en la ratio de aciertos de acceso a la caché, la configuración que más se beneficia es la de correspondencia directa —sin asociatividad— en ambos tamaños. En la caché asociativa, si el acceso a un elemento de la matriz correspondiese a la línea de caché en la que están ubicados `columnas` y `filas` no sería necesaria la sustitución, porque el nuevo bloque puede llevarse a otro plano de esa misma línea. Sin asociatividad, por el contrario, la

sustitución sería obligatoria y después provocaría una fallo de acceso, la siguiente vez que se carguen los valores de `columnas` y `filas`. Por ello la mejora obtenida para las configuraciones asociativas es bastante menor.

Volvamos a la traza de la Figura 6.6 y observemos ahora el patrón de accesos a memoria en los ciclos 5, 8 y 11. Este patrón se repetirá en los siguientes: cada tres accesos a memoria uno es para obtener un elemento de la matriz. Sin embargo, las direcciones de memoria a las que se accede son relativamente distantes: `0x10000320`, `0x100002d0`, `0x10000280`, etc. A cada paso se va a una posición de memoria que está 80 bytes delante de la recién leída. Dado que cada línea de nuestra caché tiene un tamaño de 32 bytes, está claro que ese nuevo acceso recaerá en una línea distinta.

Como hiciéramos en prácticas previas al desenrollar bucles y reordenar código para evitar riesgos de control y datos, respectivamente, cualquier modificación del código para mejorar la localidad espacial/temporal debe garantizar que el nuevo código es válido y, por tanto, produce el mismo resultado que la versión original.

Con el actual código recorremos los elementos de la matriz por columnas, de forma que se suman todos los elementos de la última columna, luego los de la anterior y así sucesivamente hasta la primera. Por cada columna vamos a la última fila, después a la anterior y en esa dirección hasta alcanzar la primera. Es ese salto de una fila a la anterior, en la misma columna, la que hace que las direcciones tengan una distancia de 80 bytes entre ellas, ya que eso es lo que ocupa cada una de las filas de la matriz.

Ese patrón de accesos puede cambiarse, sin que este caso implique mucho trabajo, a fin de optimizar la localidad espacial y que los accesos sean a posiciones consecutivas. Para ello tendríamos que recorrer los elementos por filas, en lugar de por columnas. Los cambios a realizar son los mostrados en el Listado 6.2. En esencia, se han intercambiado los índices del bucle exterior y del interior, alojados en los registros **`t1`** y **`t2`** respectivamente, por lo que a cada ciclo del bucle exterior es necesario recargar el número columnas en **`t2`** en lugar del número de filas en **`t1`**.

```
1  bucle:
2
3      call direccion
4      lw a1, 0, t0
5      add a0, a0, a1
6      addi t1, t1, -1 # Cambio t1->t2
7      bnez t1, bucle
8      lw t1, filas
9      addi t2, t2, -1 # Cambio t2->t1
10     bnez t2, bucle
```

```
11
12 bucle:
13
14     call direccion
15     lw a1, 0, t0
16     add a0, a0, a1
17     addi t2, t2, -1
18     bnez t2, bucle
19
20     lw t2, columnas
21     addi t1, t1, -1
22     bnez t1, bucle
```

Listado 6.2: Código original (izqda.) y tras introducir los cambios para optimizar la localidad espacial.

Introducido este cambio repetiríamos las ejecuciones con las configuraciones previas para poder comparar las diferencias en rendimiento. Una vez más se observa (véase la Tabla 6.3) que la configuración más beneficiada por la mejora en la localidad temporal es la que corresponde a una caché de correspondencia directa, siendo menores las ganancias en los demás casos.

Bytes	Líneas	Vías	Palabras	Hits	Misses	Ratio
256	2^3	2^0	2^3	541	71	0.8840
256	2^0	2^3	2^3	586	26	0.9575
256	2^2	2^1	2^3	586	26	0.9575
512	2^4	2^0	2^3	571	41	0.9330
512	2^0	2^4	2^3	586	26	0.9575
512	2^2	2^2	2^3	586	26	0.9575

Tabla 6.3. Los resultados mejoran en general tras optimizar la localidad espacial de los accesos a memoria.

Con el código optimizado para facilitar los accesos a posiciones de memoria cercanas, incrementar el tamaño de la caché a 1024 bytes con una configuración 2^3, 2^1 y 2^4, asociativa por conjuntos, permitiría alcanzar los 599 aciertos por 13 fallos, con una ratio del 97.88 % de acierto. Este es un rendimiento similar al de cualquier microprocesador actual con tamaños mayores de caché al ejecutar programas más complejos.

Estrategia de reemplazo para caché asociativa

Hasta el momento nos hemos centrado en ajustar el tamaño de la caché y su estructura: correspondencia directa (CD), completamente asociativa (CA) o asociativa por conjuntos (AC), según el número de filas y planos. Otro aspecto que incide en la tasa de aciertos es la estrategia de reemplazo de unas filas por otras cada vez que es preciso traer a la cache un nuevo bloque desde memoria principal. El simulador Ripes elige por defecto el método LRU. Este suele producir mejores resultados que su alternativa: **Random**.

En la configuración indicada la estrategia LRU logra en torno a un 68 % de tasa de aciertos, mientras que la aleatoria —cuyos resultados cambian ligeramente en cada ejecución por dicha aleatoriedad— oscila entre el 75 % y el 77 %.

Para una memoria caché de pequeño tamaño, con 256 bytes, y un patrón de acceso a memoria que no tenga optimizada la localidad espacial, como ocurría en la versión original del programa usado en el ejercicio, la estrategia **Random**, por el contrario, suele producir mejores resultados que la LRU, por la sencilla razón de que esta última confía en que se respete ese principio de localidad.

Comparativa de rendimientos

Completadas todas las pruebas, tanto de configuración hardware de la caché como de ajustes software en el patrón de accesos a memoria, es posible llevar a cabo una comparativa que nos permite analizar todos los resultados obtenidos y determinar cuál sería la mejor opción. Con este fin puede recurrirse a una gráfica como la de la Figura 6.7.

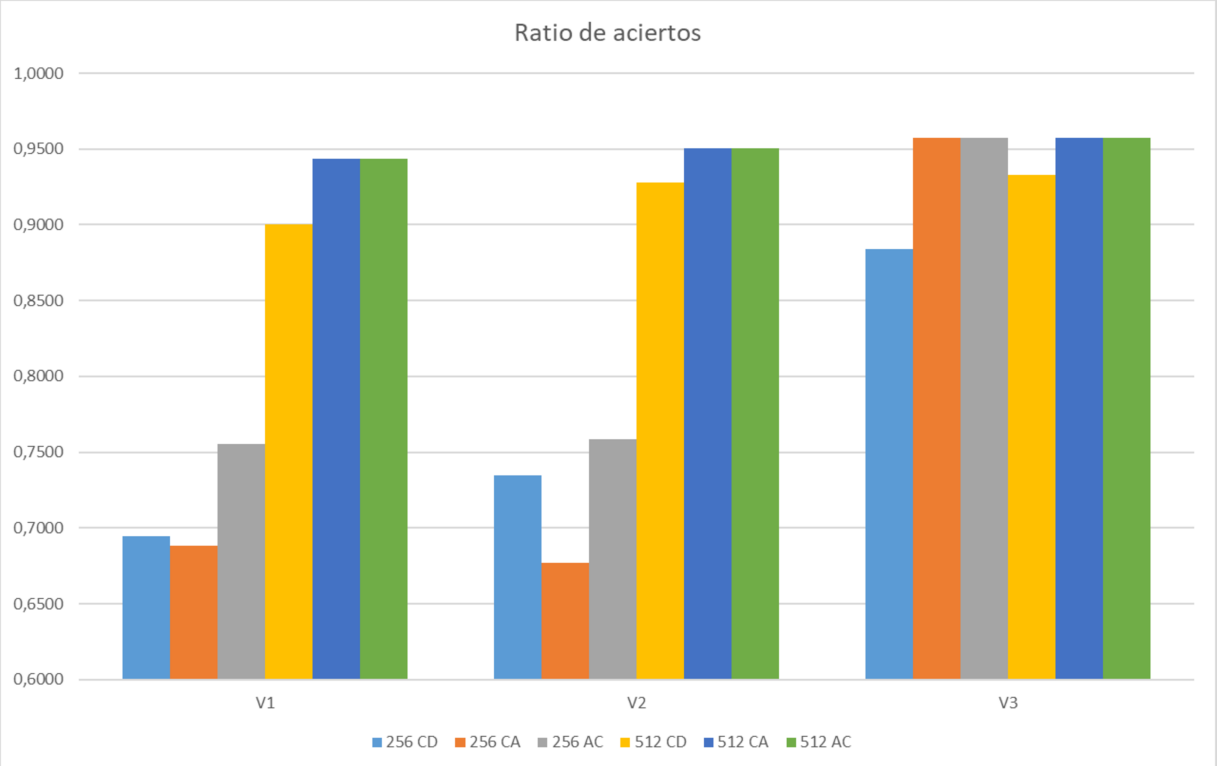


Figura 6.7. Comparamos los resultados de rendimiento de la memoria caché. Hay tres grupos que corresponden a las tres versiones del programa: **V1** es la versión original, **V2** tras realizar el cambio del identificador `filas` y **V3** una vez optimizada la localidad espacial para acceder a los elementos de la matriz. Las seis barras de cada grupo corresponden a las configuraciones de caché probadas.

Una de las conclusiones obvias es que una memoria caché de mayor tamaño, al alojar más filas, siempre mejora la tasa de aciertos. Otra es que la optimización de la localidad espacial afecta de manera importante al rendimiento y, en ocasiones, permite alcanzar una efectividad casi idéntica al de tener una mayor cantidad de caché.

6.5. La caché en accesos de escritura en memoria

El ejercicio anterior solo realiza operaciones de carga desde memoria, por lo que no le afectan los parámetros que configuran las estrategias para gestionar las escrituras. Modificar los apartados **Wr. hit** y **Wr. miss** que nos ofrece Ripes no causará, por tanto, ninguna diferencia en los resultados que hemos obtenido hasta el momento.

Para analizar el efecto que esos parámetros de configuración tendrán en el comportamiento de la memoria caché, por tanto, el primer paso consistirá en contar con un programa que realice escrituras en memoria. En la mayoría de programas esa operación, la escritura en memoria, es mucho menos frecuente que la lectura.

Cambios en el código

Tomamos el programa usado como ejercicio en los apartados previos, cuyo código ya conocemos, y vamos a introducir algunos cambios puntuales dirigidos a realizar escrituras en memoria. El objetivo es que la subrutina `direccion`, que hasta ahora tomaba sus parámetros de entrada y devolvía la salida en ciertos registros, lea sus argumentos desde unas posiciones de memoria concretas y devuelva el resultado de similar forma.

```
fil: .word 0
col: .word 0
dir: .word 0
```

El primer paso será introducir en el segmento de datos tres nuevas etiquetas, asociadas a otras tantas palabras, que serán las que aparecen al margen. Las dos primeras actuarán como parámetros de entrada, de forma que el cuerpo del bucle deberá escribir en esas posiciones de memoria la fila y columna de la matriz que quiere leer. La tercera será la empleada para retornar la dirección que corresponde a ese elemento.

```
bucle:
    la s0, fil
    sw t1, 0, s0
    la s0, col
    sw t2, 0, s0
    call direccion
    lw t0, dir
    lw a1, 0, t0
    add a0, a0, a1
    addi t2, t2, -1
    bnez t2, bucle

    lw t2, columnas
    addi t1, t1, -1
    bnez t1, bucle
```

A continuación habrá que retocar el cuerpo del bucle de manera que, antes de llamar a la subrutina, se guarden en las posiciones de memoria correspondientes la fila y columna de la matriz y, tras la llamada, se recupere la dirección. Hechos estos cambios, esa parte del código quedaría como aparece en el margen.

Por último, deberemos actualizar el código de la propia subrutina `direccion`. Los datos que antes dábamos por supuesto que se encontraban en los registros **`t1`** y **`t2`** ahora hay que leerlos de memoria. Una vez efectuadas las comprobaciones y el cálculo, la dirección que corresponda a ese elemento de la matriz se guardará en memoria para que el cuerpo la recupere.

```
direccion:
    lw s0, filas
    lw s1, fil
    bgt s1, s0, cero
    lw s0, columnas
    addi s1, s1, -1
    mul t0, s0, s1
    lw s1, col
    addi s1, s1, -1
    add t0, t0, s1
    li s0, 4
    mul t0, t0, s0
    la s0, v
    add t0, t0, s0
    la s0, dir
    sw t0, 0, s0
    ret
```

Las configuraciones de escritura en caché —y la manera en que afectarán al rendimiento de ejecución— que probaremos a continuación, asumen que hemos realizado estos cambios en el código del programa.

Rendimiento según la configuración de escritura

Dado que, tras los experimentos llevados a cabo antes, hemos llegado a la conclusión de que una memoria caché de 256 bytes, asociativa por conjuntos y con la estrategia de sustitución LRU ofrecía un buen rendimiento sin un alto coste, será esta la que usemos siempre con la nueva versión del programa. Contar con 512 bytes costaría el doble, en cuanto a hardware se refiere, y la ganancia era leve, por lo que no se justificaría su uso para este programa concreto.

Lo único que cambiaremos, por tanto, será la estrategia de escritura de valores en memoria. Para ello probamos las cuatro combinaciones posibles de configuración, ya que cada apartado tiene dos opciones. Los resultados obtenidos con cada una de ellas son los de la Tabla 6.4.

Tabla 6.4. Influencia de las configuraciones de escritura en caché en el rendimeinto.

Wr. hit	Wr. miss	Wbacks	Hits	Misses	Ratio
Write-through	Write allocate	600	1786	26	0.9857
Write-through	No write allocate	600	1784	28	0.9845
Write-back	Write allocate	0	1786	26	0.9857
Write-back	No write allocate	2	1784	28	0.9845

Al elegir la opción **Write-through** tenemos una estrategia de escritura que consiste en que los almacenamientos se lleven de forma simultánea tanto a la caché como a la memoria principal. Esto consigue que siempre exista coherencia entre ambas memorias. El inconveniente es que implica muchos accesos innecesarios a memoria principal —600 para el programa usado en este ejercicio— en caso de se escriba de manera repetida en las mismas posiciones.

Con la opción **Write-back** los almacenamientos solo afectan, en principio, a la memoria caché. Cuando una línea de esta tiene escrituras pendientes de llevar a memoria principal, se activa el indicador **D** (*Dirty*) tal y como se aprecia en la Figura 6.8. Aunque durante intervalos de tiempo no haya coherencia entre la caché y memoria principal, lo interesante es que el número de accesos a esta última se reduce de manera drástica.

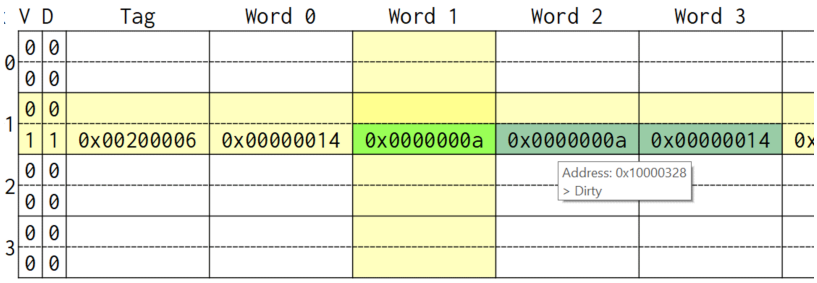


Figura 6.8. El diagrama de Ripes nos indica con un color distinto qué palabras de una línea de la caché tienen cambios pendientes de escribir en la memoria principal. Asimismo, la línea afectada tendrá activo el indicador **D**.

En cuanto a las opciones **Write allocate** y **No write allocate**, para el código de este programa tienen un impacto muy limitado. Con la primera opción se traerá a la memoria caché la línea correspondiente a la dirección de memoria en la que hay que escribir, en caso de que no estuviera allí disponible, mientras que la segunda no lleva a cabo la sustitución, es decir, la escritura se hace en memoria principal sin más.

Dada la disposición de las direcciones de memoria en que nuestro programa realiza la escrituras, situadas justo detrás de la matriz, al llevar filas de esta a la memoria caché también se incluirán las palabras que corresponden a los tres nuevos identificadores. De ahí que la diferencia sea mínima. Con otros patrones de acceso a memoria los resultados podrían ser muy dispares.

6.6. Ejercicios propuestos

Los ejercicios propuestos a continuación nos permitirán analizar el impacto que tienen en el rendimiento tanto la configuración de la memoria caché como el patrón de accesos a memoria de distintos programas. En dichos ejercicios siempre emplearemos la configuración de procesador **Single-cycle processor**.

Ejercicio 6.1. Tenemos en un programa un vector con 40 elementos, como el del listado mostrado al margen, y espacio en otros dos bloques de memoria separados 4 096 bytes (`0x1000`) entre ellos. Escribe un programa que copie en `v2` los elementos de `v1` multiplicados por 2 y, de manera análoga, en `v3` esos valores multiplicados por 3. Usar la vista **Memory** de Ripes para verificar que el resultado que produce el programa es correcto. A continuación, completa los siguientes pasos:

a) Establece las opciones de escritura en caché en sus valores por defecto.

```
1  .data
2  v:  .word  1,2,3,4,5,6,7,8,9,10
3  .word  11,12,13,14,15,16,17,18,19,20
4  .word  21,22,23,24,25,26,27,28,29,30
5  .word  31,32,33,34,35,36,37,38,39,40
6  v2: .word v+0x1000
7  v3: .word v+0x2000
8
```

b) Configura una memoria caché de 256 bytes con estrategia de sustitución por correspondencia directa, ejecuta el programa y toma los datos resultantes.
c) Mantén el tamaño y configura la caché como completamente asociativa, ejecuta el programa y guarda los resultados.
d) Configura ahora el mismo tamaño de memoria como asociativa por conjuntos, ejecuta y anota los resultados. Puedes probar distintos números de vías y de líneas.
e) Analiza los resultados obtenidos y justifica de manera razonada cuál sería la mejor configuración de caché.
f) Cambia la configuración de escritura en caché, prueba las diferentes combinaciones posibles y, para cada una de ellas, repite los pasos *b* a *d*. ¿Qué configuración de escritura funciona mejor y por qué razón?

Ejercicio 6.2. Toma el mismo vector `v1` del ejercicio anterior y escribe un programa que lea cada valor, lo duplique y lo vuelva a escribir en la misma posición. A continuación, reproduce los pasos del ejercicio 6.1 para analizar cómo afectan a ese código distintas configuraciones de caché, siempre con un tamaño de 256 bytes. Explica de manera razonada a qué se debe el comportamiento observado.

Ejercicio 6.3. Tenemos en el segmento de datos una matriz de 10 filas por 8 columnas con los valores mostrados en el listado del margen derecho. Escribir un programa que calcule la suma de cada columna de la matriz y la almacene en posiciones consecutivas a partir del identificador `sumcols`. Usar la vista **Memory** de Ripes para verificar que el resultado que produce el programa es correcto, las sumas deberían ser 460 para la primera columna, 470 para la siguiente y así sucesivamente hasta 530 para la octava. A continuación completa los siguientes pasos:

```
1  .data
2  v:   .word  1,2,3,4,5,6,7,8
3       .word  11,12,13,14,15,16,17,18
4       .word  21,22,23,24,25,26,27,28
5       .word  31,32,33,34,35,36,37,38
6       .word  41,42,43,44,45,46,47,48
7       .word  51,52,53,54,55,56,57,58
8       .word  61,62,63,64,65,66,67,68
9       .word  71,72,73,74,75,76,77,78
10      .word  81,82,83,84,85,86,87,88
11      .word  91,92,93,94,95,96,97,98
12
13 sumcols: .word v+0x2000
14
```

a) Establece las opciones de escritura en caché en sus valores por defecto.
b) Configura una memoria caché de 256 bytes con estrategia de sustitución por correspondencia directa, ejecuta el programa y toma los datos resultantes.
c) Mantén el tamaño y configura la caché como completamente asociativa, ejecuta el programa y guarda los resultados.
d) Configura ahora el mismo tamaño de memoria como asociativa por conjuntos, ejecuta y anota los resultados. Puedes probar distintos números de vías y de líneas.
e) Analiza los resultados obtenidos y justifica de manera razonada cuál sería la mejor configuración de caché.
f) Ejecuta paso a paso varios ciclos del programa manteniendo abierta la vista **Cache** y anota el patrón de accesos a memoria. ¿Es posible mejorar la localidad espacial para reducir el número de fallos en el acceso a caché? En caso afirmativo trata de reescribir el código para ello.

Ejercicio 6.4. Escribe un programa que almacene los números 1 a 1000 en posiciones de memoria consecutivas a partir de la dirección `0x10000000`. Genera los valores en un bucle, contando en un registro desde 1 hasta 1000, escribiendo en cada iteración el valor en la posición adecuada. Ejecuta el programa y usa la vista **Memory** de Ripes para verificar que el resultado que produce es correcto, como se aprecia parcialmente a la derecha. A continuación completa los siguientes pasos:

Memory viewer

Address	Word
0x1000002c	12
0x10000028	11
0x10000024	10
0x10000020	9
0x1000001c	8
0x10000018	7
0x10000014	6
0x10000010	5
0x1000000c	4
0x10000008	3
0x10000004	2
0x10000000	1

a) Elige la opción **No write allocate** del apartado **Wr. miss**.
b) Prueba con distintas configuraciones de memoria caché —correspondencia directa, completamente asociativa, asociativa por conjuntos— en tamaños de 256 o 512 bytes. Para cada una ejecuta el programa y observa la tasa de aciertos. ¿Qué ocurre? ¿A qué se debe ese comportamiento?
c) Fija la opción del paso *a* en el valor **Write allocate** y repite las pruebas indicadas en el paso *b*. Anota los resultados obtenidos con cada ejecución. ¿Cuál es la mejor configuración?
d) Con un límite de 256 bytes, busca la configuración de caché que ofrecería un mejor rendimiento para este programa en concreto.

Soluciones a ejercicios 7

En este apéndice se facilitan posibles soluciones a parte de los ejercicios propuestos al final de cada práctica. Ha de tenerse en cuenta que no existe una única forma de realizar dichos ejercicios. Al contrario, el mismo resultado válido puede conseguirse por distintos medios, por ejemplo al emplear otros registros y combinaciones de instrucciones. Por lo tanto, las facilitadas aquí son solo una de las posibles soluciones para cada caso.

7.1. Introducción a RISC-V y Ripes

Ejercicio 1.2

```
addi a0, zero, 23
addi a1, zero, 8
sub a0, a0, a1
addi a7, zero, 1
ecall
```

Ejercicio 1.3

```
addi a0, zero, 5
addi a1, zero, 7
mul a0, a0, a1
addi a7, zero, 1
ecall
```

Ejercicio 1.4

```
addi a1, zero, 23
addi a2, zero, 4
div a0, a1, a2
addi a7, zero, 1
ecall
rem a0, a1, a2
ecall
```

Ejercicio 1.5

```
    .data
n1: .word 5
n2: .word 7
n3: .word 11

    .text
mv a0, zero
lw a1, n1
add a0, a0, a1
lw a1, n2
add a0, a0, a1
lw a1, n3
add a0, a0, a1
addi a7, zero, 1
ecall
```

Ejercicio 1.6

```
    .data
n1: .word 5
n2: .word 7
n3: .word 11

    .text
```

```
mv a0, zero
la t0, n1
lw a1, 0(t0)
add a0, a0, a1
lw a1, 4(t0)
add a0, a0, a1
lw a1, 8(t0)
add a0, a0, a1
addi a7, zero, 1
ecall
```

Ejercicio 1.7

```
    .data
n1: .word 5
n2: .word 7
n3: .word 11

    .bss
suma: .word 0

    .text
mv a0, zero
la t0, n1
lw a1, 0(t0)
add a0, a0, a1
lw a1, 4(t0)
add a0, a0, a1
lw a1, 8(t0)
add a0, a0, a1
la t1, suma
sw a0, 0, t1
```

Ejercicio 1.8

```
    .bss
cociente: .word 0
resto: .word 0

    .text
addi a1, zero, 23
addi a2, zero, 4
la t0, cociente

div a0, a1, a2
sw a0, 0, t0
rem a0, a1, a2
sw a0, 4, t0
```

Ejercicio 1.9

```
    .data
n: .word 75000

    .text
lw a0, n
addi a7, zero, 1
ecall
```

Ejercicio 1.10

```
    .text
lui a0, 18   # 18 << 12 bits = 73728
addi a0, a0, 1272

addi a7, zero, 1
ecall
```

7.2. Bucles y condicionales en RISC-V

Ejercicio 2.1

```
    li a1, 1000
    li a0, 0

bucle:
    add a0, a0, a1
    addi a1, a1, -1
    bnez a1, bucle

    li a7, 1
    ecall
```

Ejercicio 2.2

```
    li a1, 999
    li a0, 0

bucle:
    add a0, a0, a1
    addi a1, a1, -2
    bgtz a1, bucle

    li a7, 1
    ecall
```

Ejercicio 2.3

```
    .data
    cod: .string "Código "
    car: .string " - Carácter "
    sep: .string ", "

    .text

    li a1, 40

bucle:
    la a0, cod
    li a7, 4
    ecall
    mv a0, a1
    li a7, 1
    ecall
    la a0, car
    li a7, 4
    ecall
    mv a0, a1
    li a7, 11
    ecall
    la a0, sep
    li a7, 4
    ecall

    addi a1, a1, 1
    li a0, 122
    ble a1, a0, bucle
```

Ejercicio 2.4

```
    li a3, 10
    li a1, 1

bucle1:
    li a2, 1

bucle2:
    mv a0, a1
    li a7, 1
    ecall
    li a0, 120
    li a7, 11
    ecall
    mv a0, a2
    li a7, 1
    ecall
    li a0, 61
    li a7, 11
    ecall
    mul a0, a1, a2
    li a7, 1
    ecall
    li a0, 32
    li a7, 11
```

```
    ecall

    addi a2, a2, 1
    ble a2, a3, bucle2

    li a0, 13
    li a7, 11
    ecall

    addi a1, a1, 1
    ble a1, a3, bucle1
```

Ejercicio 2.5

```
   .data
v: .word 7, 4, 23, 12, 6, 20, 17, 8, 3, 10
fin:

    .text

    li a0, 0
    li a1, 0
    la t0, v
    la t1, fin

bucle:
    lw a1, 0, t0
    add a0, a0, a1
    addi a1, a1, 1
    addi t0, t0, 4
    blt t0, t1, bucle

    div a0, a0, a1
    li a7, 1
    ecall
```

Ejercicio 2.6

```
    .data
v: .word 7, 4, 23, 12, 6, 20, 17, 8, 3, 10
fin:

    .text

    li a0, 0
    la t0, v
    la t1, fin

bucle:
    lw a1, 0, t0
    add a0, a0, a1
    addi t0, t0, 8
    blt t0, t1, bucle

    li a7, 1
    ecall
```

Ejercicio 2.7

```
    .data
v: .word 7, 4, 23, 12, 6, 20, 17, 8, 3, 10
fin:

    .text

    li a0, 0
    la t0, v
    la t1, fin

bucle:
    lw a1, 0, t0
    andi a2, a1, 1
    beqz a2, noimpar
    add a0, a0, a1
    noimpar:
    addi t0, t0, 4
    blt t0, t1, bucle

    li a7, 1
    ecall
```

Ejercicio 2.8

```
    .data
may: .string "El mayor es "
men: .string "El menor es "
v: .word 7, 4, 23, 12, 6, 20, 17, 8, 3, 10
fin:

    .text

    lw a0, v
    lw a1, v
    la t0, v
    la t1, fin

bucle:
    lw a2, 0, t0
    bge a2, a0, nomenor
    mv a0, a2

nomenor:
    ble a2, a1, nomayor
    mv a1, a2

nomayor:
    addi t0, t0, 4
    blt t0, t1, bucle

    mv a2, a0
    la a0, men
    li a7, 4
    ecall
    mv a0, a2
    li a7, 1
    ecall
    li a0, 13
    li a7, 11
    ecall
    la a0, may
    li a7, 4
    ecall
    mv a0, a1
    li a7, 1
    ecall
```

Ejercicio 2.9

```
    .data
texto: .string "Suma de la fila "
v:   .word  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16

    .text

    li a0, 0
    li a1, 0
    li a2, 0
    li a3, 0
    la t0, v
    li t1, 4

bucle:
    lw t2, 0, t0
    add a0, a0, t2
    lw t2, 16, t0
    add a1, a1, t2
    lw t2, 32, t0
    add a2, a2, t2
    lw t2, 48, t0
    add a3, a3, t2

    addi t0, t0, 4
    addi t1, t1, -1
    bnez t1, bucle

    li t0, 1
    call muestra

    li t0, 2
    mv a0, a1
    call muestra

    li t0, 3
    mv a0, a2
    call muestra

    li t0, 4
    mv a0, a3
```

```
    call muestra

    li a7, 10
    ecall

muestra:
    mv s0, a0
    la a0, texto
    li a7, 4
    ecall
    mv a0, t0
    li a7, 1
    ecall
    li a0, 61
    li a7, 11
    ecall
    mv a0, s0
    li a7, 1
    ecall
    li a0, 13
    li a7, 11
    ecall

    ret
```

Ejercicio 2.10

```
    .data
texto: .string "Suma de la columna "
v:   .word  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16

    .text

    li a0, 0
    li a1, 0
    li a2, 0
    li a3, 0
    la t0, v
    li t1, 4

bucle:
    lw t2, 0, t0
    add a0, a0, t2
    lw t2, 4, t0
    add a1, a1, t2
    lw t2, 8, t0
    add a2, a2, t2
    lw t2, 12, t0
    add a3, a3, t2

    addi t0, t0, 16
    addi t1, t1, -1
    bnez t1, bucle

    li t0, 1
    call muestra
    li t0, 2
    mv a0, a1
    call muestra
    li t0, 3
    mv a0, a2
    call muestra
    li t0, 4
    mv a0, a3
    call muestra

    li a7, 10
    ecall

muestra:
    mv s0, a0
    la a0, texto
    li a7, 4
    ecall
    mv a0, t0
    li a7, 1
    ecall
    li a0, 61
    li a7, 11
    ecall
    mv a0, s0
    li a7, 1
    ecall
    li a0, 13
    li a7, 11
    ecall

    ret
```

Ejercicio 2.11

```
    .data
v:   .word  6,1,6,7,2,3,6,-3,6,1,-2,3,-5,2,6
fin:

    .text
    li a0, 0
    lw a1, v
    la t0, v
    la t1, fin

bucle:
    lw a2, 0, t0
    bne a2, a1, noigual
    addi a0, a0, 1

noigual:
    addi t0, t0, 4
    blt t0, t1, bucle

    li a7, 1
    ecall
```

Ejercicio 2.12

```
    .data
v:   .word  6,1,6,7,2,3,6,-3,6,1,-2,3,-5,2,6
fin:

    .text

    li a0, 0
    la t0, v
    la t1, fin

bucle:
    lw a2, 0, t0
    slti a1, a2, 0
    add a0, a0, a1
    addi t0, t0, 4
    blt t0, t1, bucle

    li a7, 1
    ecall
```

Ejercicio 2.13

```
    .data
v:   .word  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16

    .text

    li a1, 0
    li a2, 0
    call leev
    call print
    li a1, 1
    li a2, 1
    call leev
    call print
    li a1, 2
    li a2, 2
    call leev
    call print
    li a1, 3
    li a2, 3
    call leev
    call print

    li a7, 10
    ecall

    # Entrada: a1 -> fila, a2 -> columna
    # Salida: a0 <- valor de esta posición
leev:
    la t0, v
    mv t1, a1
    li t2, 4
    mul t1, t1, t2
    add t1, t1, a2
    mul t1, t1, t2
    add t0, t0, t1
    lw a0, 0, t0
```

```
    ret

print:
    li a7, 1
    ecall
    li a7, 11
    li a0, 13
    ecall
    ret
```

Ejercicio 2.14

```
    .data
cad1: .string "El valor en la fila "
cad2: .string " columna "
cad3: .string " es "
v:   .word  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16

    .text

    li a1, 0
    li a2, 0
    call leev
    call print
    li a1, 1
    li a2, 1
    call leev
    call print
    li a1, 2
    li a2, 2
    call leev
    call print
    li a1, 3
    li a2, 3
    call leev
    call print

    li a7, 10
    ecall

    # Entrada: a1 -> fila, a2 -> columna
    # Salida: a0 <- valor de esta posición
leev:
    la t0, v
    mv t1, a1
    li t2, 4
    mul t1, t1, t2
    add t1, t1, a2
    mul t1, t1, t2
    add t0, t0, t1
    lw a0, 0, t0
    ret

    # Entrada: a1 -> fila, a2 -> columna, a0 -> valor
    # Salida: ninguna
print:
    mv t0, a0
    la a0, cad1
    li a7, 4
    ecall
    mv a0, a1
    li a7, 1
    ecall
    la a0, cad2
    li a7, 4
    ecall
    mv a0, a2
    li a7, 1
    ecall
    la a0, cad3
    li a7, 4
    ecall
    mv a0, t0
    li a7, 1
    ecall
    li a7, 11
    li a0, 13
    ecall
    mv a0, t0
    ret
```

Ejercicio 2.15

```
    li a0, 10
    call printr
```

```
    li a7, 10
    ecall

    # Entrada: a0 -> \U'ltimo valor a mostrar
printr:
    addi sp, sp, -8
    sw ra, 0, sp
    sw a0, 4, sp

    beqz a0, finrecur
    addi a0, a0, -1
    call printr

finrecur:
    lw a0, 4, sp
    call printn
    lw ra, 0, sp
    addi sp, sp, 8
    ret

printn:
    li a7, 1
    ecall
    li a7, 11
    li a0, 13
    ecall
    ret
```

Ejercicio 2.16

```
    li a1, 42
    li a2, 70
    call mcd
    li a7, 1
    ecall
    li a7, 10
    ecall

    # Entrada: a1 -> primer número, a2 -> segundo número
    # Salida: a0 <- Máximo común divisor
mcd:
    addi sp, sp, -12
    sw ra, 0, sp
    sw a1, 4, sp
    sw a2, 8, sp

    rem a1, a2, a1
    lw a2, 4, sp
    beqz a1, finmcd

    call mcd
    j finrecur

finmcd:
    mv a0, a2

finrecur:
    lw ra, 0, sp
    addi sp, sp, 12
    ret
```

Ejercicio 2.17 - Recursivo

```
# Rellenar tabla de números primos - Versión recursiva

    .data

primos: .word 2, 3, 5, 7, 11, 13

    .text

    li a2, 6
    li a1, 14
    li a3, 100

bucle:
    call primo
    beqz a0, print

    li t0, 4
    mul t0, a2, t0
    la t1, primos
    add t0, t0, t1
    addi a2, a2, 1
    sw a1, 0, t0
```

```

    mv a0, a1
    li a7, 1
    ecall
    li a0, 13
    li a7, 11
    ecall

print:
    addi a1, a1, 1
    blt a1, a3, bucle

    li a7, 10
    ecall

    # Entrada: a1 -> número a comprobar, a2 -> número de primos en la tabla
    # Salida: a0 <- 1 si el número es primo o 0 en caso contrario, a2 -> número actualizado de
      primos
primo:
    addi sp, sp, -12
    sw ra, 0, sp
    sw a1, 4, sp
    sw a2, 8, sp
    li a0, 0

    addi a2, a2, -1
    li t0, 4
    mul t0, a2, t0
    la t1, primos
    add t0, t0, t1
    lw t0, 0, t0

    rem t1, a1, t0
    beqz t1, finprimo

    lw a2, 8, sp
    addi a2, a2, -1
    beqz a2, esprimo
    call primo
    beqz a0, finprimo

esprimo:
    li a0, 1
    j finrecur

finprimo:
    li a0, 0

finrecur:
    lw a2, 8, sp
    lw ra, 0, sp
    lw a1, 4, sp
    addi sp, sp, 12
    ret
```

Ejercicio 2.17 - Iterativo

```
# Rellenar tabla de números primos - Versión iterativa

    .equ primero 14
    .equ ultimo 100

    .data
primos: .word 2, 3, 5, 7, 11, 13

    .text

    li a2, 6
    li a1, primero
    li a3, ultimo

    bucle:
    call primo
    beqz a0, print

    li t0, 4
    mul t0, a2, t0
    la t1, primos
    add t0, t0, t1
    addi a2, a2, 1
    sw a1, 0, t0

    mv a0, a1
    li a7, 1
    ecall
    li a0, 13
    li a7, 11
    ecall
```

```
32
33 print:
34     addi a1, a1, 1
35     blt a1, a3, bucle
36
37     li a7, 10
38     ecall
39
40     # Entrada: a1 -> número a comprobar, a2 -> número de primos en la tabla
41     # Salida: a0 <- 1 si el número es primo o 0 en caso contrario, a2 -> número actualizado de
         primos
42 primo:
43     li a0, 0
44     la t0, primos
45     mv t1, a2
46
47 bprimo:
48     lw t2, 0, t0
49     rem t2, a1, t2
50     beqz t2, noprimo
51     addi t0, t0, 4
52     addi t1, t1, -1
53     bnez t1, bprimo
54
55     li a0, 1
56     noprimo:
57     ret
```

Ejercicio 2.18

```
1     li a0, 1
2     li a1, 10
3     li a7, 1
4
5     bucle:
6     ecall
7     call espera
8     addi a0, a0, 1
9     ble a0, a1, bucle
10
11    li a7, 10
12    ecall
13
14 espera:
15    lui t0, 0xf  # t0 <- 65535. Según la velocidad del ordenador el valor de t0 tendrá que ser
        mayor o menor
16
17 bespera:
18    nop
19    addi t0, t0, -1
20    bnez t0, bespera
21    ret
```

Ejercicio 2.19

```
1     call limpia
2
3     li a7, 10
4     ecall
5
6  limpia:
7     li t1, LED_MATRIX_0_BASE
8     li t2, LED_MATRIX_0_SIZE
9     add t2, t2, t1
10    li t3, 0xffffff
11 blimpia:
12    sw t3, 0, t1
13    addi t1, t1, 4
14    bne t1, t2, blimpia
15    ret
```

Ejercicio 2.20

```
1     .data
2  puntos: .word 0, 0, 0xff0000, 34, 0, 0x00ff00, 0, 24, 0x0000ff, 34, 24, 0x777777, 17, 12, 0xff00ff
3
4     .text
5
6     call limpia
7     la s0, puntos
8     li a3, 5
9
```

```
    bucle:
    lw a0, 0, s0
    lw a1, 4, s0
    lw a2, 8, s0
    call pixel
    call espera

    addi a3, a3, -1
    addi s0, s0, 12
    bnez a3, bucle

    li a7, 10
    ecall

pixel: # a0: x, a1: y, a2: color
    li t0, LED_MATRIX_0_WIDTH
    mul t0, t0, a1
    add t0, t0, a0
    li t1, 4
    mul t0, t0, t1
    li t1, LED_MATRIX_0_BASE
    add t0, t0, t1
    sw a2, 0, t0
    ret

limpia:
    li t1, LED_MATRIX_0_BASE
    li t2, LED_MATRIX_0_SIZE
    add t2, t2, t1
    li t3, 0xffffff
blimpia:
    sw t3, 0, t1
    addi t1, t1, 4
    bne t1, t2, blimpia
    ret

espera:
    lui t0, 0xf

bespera:
    addi t0, t0, -1
    bnez t0, bespera
    ret
```

Ejercicio 2.21

```
    .data
puntos: .word 0, 0, 34, 0, 0x00ff00, 0, 24, 34, 24, 0x777777, 17, 0, 17, 24, 0xff00ff

    .text

    call limpia
    la s0, puntos
    li s3, 3

    bucle:
    lw a0, 0, s0
    lw a1, 4, s0
    lw a3, 8, s0
    lw a4, 12, s0
    lw a2, 16, s0
    call linea
    call espera

    addi s3, s3, -1
    addi s0, s0, 20
    bnez s3, bucle

    li a7, 10
    ecall

linea: # a0, a1 -> (x1, y1), a3, a4 -> (x2, y2), a2 -> color
    addi sp, sp, -16
    sw ra, 0, sp
    sw a0, 4, sp
    sw a1, 8, sp

    beq a1, a4, lhoriz
    beq a0, a3, lvert

fin:
    lw ra, 0, sp
    lw a0, 4, sp
    lw a1, 8, sp
    addi sp, sp, 16
    ret

lvert:
```

```
    call pixel
    addi a1, a1, 1
    ble a1, a4, lvert
    j fin

lhoriz:
    call pixel
    addi a0, a0, 1
    ble a0, a3, lhoriz
    j fin

pixel: # a0: x, a1: y, a2: color
    li t0, LED_MATRIX_0_WIDTH
    mul t0, t0, a1
    add t0, t0, a0
    li t1, 4
    mul t0, t0, t1
    li t1, LED_MATRIX_0_BASE
    add t0, t0, t1
    sw a2, 0, t0
    ret


limpia:
    li t1, LED_MATRIX_0_BASE
    li t2, LED_MATRIX_0_SIZE
    add t2, t2, t1
    li t3, 0xffffff
blimpia:
    sw t3, 0, t1
    addi t1, t1, 4
    bne t1, t2, blimpia
    ret

espera:
    lui t0, 0xf

bespera:
    addi t0, t0, -1
    bnez t0, bespera
    ret
```

Ejercicio 2.22

```
    li a2, 0xff7733
    li a0, 0
    li a1, 0
    li a3, 6
    li a4, 6
    li a5, 24

    bucle:
    call limpia
    call recuadro
    call espera
    addi a0, a0, 1
    addi a1, a1, 1
    addi a3, a3, 1
    addi a4, a4, 1
    ble a4, a5, bucle

    li a7, 10
    ecall

recuadro: # a0, a1 -> (x1, y1), a3, a4 -> (x2, y2), a2 -> color
    addi sp, sp, -32
    sw ra, 0, sp
    sw a0, 4, sp
    sw a1, 8, sp
    sw a3, 12, sp
    sw a4, 16, sp
    sw a2, 20, sp

    mv a4, a1
    call linea

    lw a4, 16, sp
    mv a1, a4
    call linea

    lw a1, 8, sp
    mv a3, a0
    call linea

    lw a3, 12, sp
    mv a0, a3
    call linea

```

```
finr:
    lw a2, 20, sp
    lw a4, 16, sp
    lw a3, 12, sp
    lw a1, 8, sp
    lw a0, 4, sp
    lw ra, 0, sp
    addi sp, sp, 32
    ret

linea: # a0, a1 -> (x1, y1), a3, a4 -> (x2, y2), a2 -> color
    addi sp, sp, -16
    sw ra, 0, sp
    sw a0, 4, sp
    sw a1, 8, sp

    beq a1, a4, lhoriz
    beq a0, a3, lvert

fin:
    lw ra, 0, sp
    lw a0, 4, sp
    lw a1, 8, sp
    addi sp, sp, 16
    ret

lvert:
    call pixel
    addi a1, a1, 1
    ble a1, a4, lvert
    j fin

lhoriz:
    call pixel
    addi a0, a0, 1
    ble a0, a3, lhoriz
    j fin

pixel: # a0: x, a1: y, a2: color
    li t0, LED_MATRIX_0_WIDTH
    mul t0, t0, a1
    add t0, t0, a0
    li t1, 4
    mul t0, t0, t1
    li t1, LED_MATRIX_0_BASE
    add t0, t0, t1
    sw a2, 0, t0
    ret


limpia:
    li t1, LED_MATRIX_0_BASE
    li t2, LED_MATRIX_0_SIZE
    add t2, t2, t1
    li t3, 0xffffff

blimpia:
    sw t3, 0, t1
    addi t1, t1, 4
    bne t1, t2, blimpia
    ret

espera:
    lui t0, 0x3

bespera:
    addi t0, t0, -1
    bnez t0, bespera
    ret
```

7.3. Segmentación del cauce y riesgos de datos

Ejercicio 3.1

```
    #               Ciclos        CPI        Mejora
    # Sin pipeline    295 (59)       1
    # Pipeline s/cb   127            2.15        132%
    # Pipeline c/cb    93            1.58         37%
    # Pipline reord.   83            1.41         12%

    .data
v: .word 7, 4, 23, 12, 6, 20, 17, 8, 3, 10
fin:

    .text

```

```
    li a0, 0
    li a1, 0
    la t0, v
    la t1, fin

bucle:
    lw a1, 0, t0
    addi t0, t0, 4
    add a0, a0, a1
    addi a1, a1, 1
    blt t0, t1, bucle

    div a0, a0, a1
    li a7, 1
    ecall
```

Ejercicio 3.4

```
    li a1, 1000
    li a0, 0
    nop
    nop

bucle:
    add a0, a0, a1
    nop
    addi a1, a1, -1
    nop
    nop
    bnez a1, bucle

    li a7, 1
    nop
    nop
    nop
    ecall
```

7.4. Segmentación del cauce y riesgos de control

Ejercicio 4.1

```
    .data
v: .word 7,4,23,12,6,20,17,8,3,10,11,23
fin:

    .text

    li a0, 0
    li a1, 0
    la t0, v
    la t1, fin

bucle:
    lw a2, 0, t0
    lw a3, 4, t0
    add a0, a0, a2
    add a0, a0, a3
    addi t0, t0, 8
    addi a1, a1, 2
    blt t0, t1, bucle

    div a0, a0, a1
    li a7, 1
    ecall
```

Ejercicio 4.2

```
    .data
v: .word 7,4,23,12,6,20,17,8,3,10,11,23
fin:

    .text

    li a0, 0
    li a1, 0
    la t0, v
    la t1, fin

bucle:
    lw a2, 0, t0
```

```
    lw a3, 4, t0
    lw a4, 8, t0
    add a0, a0, a2
    add a0, a0, a3
    add a0, a0, a4
    addi t0, t0, 12
    addi a1, a1, 3
    blt t0, t1, bucle

    div a0, a0, a1
    li a7, 1
    ecall
```

Ejercicio 4.3

```
    .data
v: .word 7,4,23,12,6,20,17,8,3,10,11,23
fin:

    .text

    li a0, 0
    li a1, 0
    la t0, v
    la t1, fin

bucle:
    lw a2, 0, t0
    lw a3, 4, t0
    lw a4, 8, t0
    lw a5, 12, t0
    add a0, a0, a2
    add a0, a0, a3
    add a0, a0, a4
    add a0, a0, a5
    addi t0, t0, 16
    addi a1, a1, 4
    blt t0, t1, bucle

    div a0, a0, a1
    li a7, 1
    ecall
```

Ejercicio 4.4

```
    .data
v: .word 7,4,23,12,6,20,17,8,3,10,11,23
fin:

    .text

    li a0, 0
    la t0, v
    la t1, fin

bucle:
    lw a1, 0, t0
    lw a3, 4, t0
    andi a2, a1, 1
    beqz a2, noimpar
    add a0, a0, a1

noimpar:
    andi a2, a3, 1
    addi t0, t0, 8
    beqz a2, noimpar2
    add a0, a0, a3

noimpar2:
    blt t0, t1, bucle

    li a7, 1
    ecall
```

7.5. Arquitectura superescalar

Ejercicio 5.1

```
# Versión final del programa, tras reducir los riesgos posibles
# y ejecutarlo en la configuración superescalar de Ripes

    .data

v:  .word 33    71  72  74  47  62  1   33  2   73  73  57  1   75  39  68  53  81  74  48  14  23
        6   51  89
    .word 3 89  84  24  12  36  28  83  84  35  2   79  84  5   98  71  48  7   6   12  100 81  61
        90  73
    .word 69    2   7   90  82  18  44  18  21  89  46  91  83  40  49  71  66  36  43  50  72  70
        73  17  29
    .word 52    62  67  34  8   78  67  58  78  42  16  1   11  88  3   37  31  55  89  52  23  16
        84  76  88

    .text

    lui t0, %hi(v)
    li s0, 25

    li a0, 0

bucle:
    lw a1, 0, t0
    addi s0, s0, -1

    lw a2, 4, t0
    nop

    lw a3, 8, t0
    add a0, a0, a1

    lw a4, 12, t0
    add a0, a0, a2

    add a0, a0, a3
    addi t0, t0, 16

    add a0, a0, a4
    bnez s0, bucle

    li a7, 1
    ecall

    # 234 ciclos, CPI=0.717
```

7.6. Memoria caché

Ejercicio 6.1

```
    .data
v:  .word  1, 2, 3, 4, 5, 6, 7, 8, 9, 10
    .word  11, 12, 13, 14, 15, 16, 17, 18, 19, 20
    .word  21, 22, 23, 24, 25, 26, 27, 28, 29, 30
    .word  31, 32, 33, 34, 35, 36, 37, 38, 39, 40
v2:  .word v+0x1000
v3:  .word v+0x2000

    .text

    la t0, v
    lw t1, v2
    lw t2, v3
    li t3, 40

bucle:
    lw a0, 0, t0
    add a0, a0, a0
    sw a0, 0, t1
    lw a0, 0, t0
    add a1, a0, a0
    add a1, a1, a0
    sw a1, 0, t2
    addi t0, t0, 4
    addi t1, t1, 4
    addi t2, t2, 4
    addi t3, t3, -1
    bnez t3, bucle
```

Ejercicio 6.2

```
    .data
v:  .word  1, 2, 3, 4, 5, 6, 7, 8, 9, 10
    .word  11, 12, 13, 14, 15, 16, 17, 18, 19, 20
```

```
    .word  21, 22, 23, 24, 25, 26, 27, 28, 29, 30
    .word  31, 32, 33, 34, 35, 36, 37, 38, 39, 40

    .text

    la t0, v
    li t3, 40

bucle:
    lw a0, 0, t0
    add a0, a0, a0
    sw a0, 0, t0
    addi t0, t0, 4
    addi t3, t3, -1
    bnez t3, bucle
```

Ejercicio 6.3

```
    .data
v:  .word  1, 2, 3, 4, 5, 6, 7, 8
    .word  11, 12, 13, 14, 15, 16, 17, 18
    .word  21, 22, 23, 24, 25, 26, 27, 28
    .word  31, 32, 33, 34, 35, 36, 37, 38
    .word  41, 42, 43, 44, 45, 46, 47, 48
    .word  51, 52, 53, 54, 55, 56, 57, 58
    .word  61, 62, 63, 64, 65, 66, 67, 68
    .word  71, 72, 73, 74, 75, 76, 77, 78
    .word  81, 82, 83, 84, 85, 86, 87, 88
    .word  91, 92, 93, 94, 95, 96, 97, 98

sumcols: .word v+0x2000


    .text

    la t0, v
    lw t1, sumcols
    li s0, 0
    li s1, 8
    li s2, 10

bucle:
    add t2, t1, s0
    lw a0, 0, t2
    lw a1, 0, t0
    add a0, a0, a1
    sw a0, 0, t2
    addi t0, t0, 4
    addi s0, s0, 4
    addi s1, s1, -1
    bnez s1, bucle
    li s1, 8
    li s0, 0
    addi s2, s2, -1
    bnez s2, bucle
```

Ejercicio 6.4

```
    lui t0, 0x10000
    li s0, 1
    li s1, 1000

bucle:
    sw s0, 0, t0
    addi t0, t0, 4
    addi s0, s0, 1
    blt s0, s1, bucle

    li a7, 10
    ecall
```

Bibliografía

Referencias en orden de aparición en el texto.

[1] Andrew Shell Waterman. *Design of the RISC-V instruction set architecture*. University of California, Berkeley, 2016.

[2] Andrew Shell Waterman y Krste Asanovic. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. University of California, Berkeley, 2017.

[3] Morten B Petersen. «Ripes: A Visual Computer Architecture Simulator». En: *2021 ACM/IEEE Workshop on Computer Architecture Education (WCAE)*. IEEE. 2021, págs. 1-8.

[4] Emily Blem, Jaikrishnan Menon, Thiruvengadam Vijayaraghavan y Karthikeyan Sankaralingam. «ISA wars: Understanding the relevance of ISA being RISC or CISC to performance, power, and energy on modern architectures». En: *ACM Transactions on Computer Systems (TOCS)* 33.1 (2015), págs. 1-34.

[5] John F Wakerly. «Microprocessor input/output Architecture». En: *Computer* 10.2 (1977), págs. 26-33.

[6] Nael Abu-Ghazaleh, Dmitry Ponomarev y Dmitry Evtyushkin. «How the spectre and meltdown hacks really worked». En: *IEEE Spectrum* 56.3 (2019), págs. 42-49.

Índice alfabético

Universidad de Jaén