ARQUITECTURA DE COMPUTADORES
BOLETÍN DE LABORATORIO 1: EL PROCESADOR

PARTE 1: CONTENIDO DE LAS SESIONES

1. ARQUITECTURA DEL JUEGO DE INSTRUCCIONES RISC-V

1.1. CARACTERÍSTICAS DE LA ARQUITECTURA

RISC-V es una arquitectura de conjunto de instrucciones de hardware libre basado en un diseño de tipo RISC. El proyecto comenzó en 2010 en la Universidad de California en Berkeley, pero muchos colaboradores son voluntarios y trabajadores de la industria fuera de la universidad. El conjunto de instrucciones se ha diseñado pensando en implementaciones pequeñas, rápidas y de bajo consumo para el mundo real, pero sin buscar una microarquitectura concreta.

En mayo de 2017, estaba cerrada la versión 2.2 del conjunto de instrucciones del espacio de usuario. El conjunto de instrucciones privilegiadas estaba disponible como borrador en la versión 1.10. Además de la ISA base, existen diversas extensiones para unidades funcionales concretas y/o sistemas empotrados. Estas extensiones pueden verse en la siguiente figura:

Nombre	Descripción	Versión	Estado[más bajo-alfa 1]
Base			
RV32I	Instrucción de Entero de la base Puso, 32-bits	2.0	✓ **Cerrada**
RV32E	Conjunto de instrucciones de base entera(embedded), 32-bits, 16 registros	1.9	✗ **Abierta**
RV64I	Conjunto de instrucciones de base entera, 64-bits	2.0[44]	
RV128I	Conjunto de instrucciones de base entera, 128-bits	1.7[45]	
Extensión			
M	Extensión estándar para Multiplicación de Entero y División	2.0	✓ **Cerrada**
Un	Extensión estándar para Instrucciones Atómicas	2.0	✗ **Abierta**
F	Extensión estándar para punto flotante de precisión simple	2.0	✓ **Cerrada**
D	Extensión estándar para punto flotante de precisión doble	2.0	✗ **Abierta**
G	Abreviatura para la base y extensiones anteriores		
Q	Extensión estándar para punto flotante de precisión cuádruple	2.0	✓ **Cerrada**
L	Extensión estándar para punto flotante decimal	0.0	✓ **Cerrada**
C	Extensión estándar para instrucciones comprimidas	2.0	
B	Extensión estándar para manipulación de bits	0.36	
J	Extensión estándar para lenguajes traducidos dinámicamente	0.0	✓ **Cerrada**
T	Extensión estándar para Memoria Transaccional	0.0	✗ **Abierta**
P	Extensión estándar para Empaquetado-SIMD Instrucciones	0.1	✓ **Cerrada**
V	Extensión estándar para Operaciones de Vector	0.2	✗ **Abierta**
N	Extensión estándar para interrupciones de nivel de usuario	1.1	✗ **Abierta**

El simulador VisualRV32 integra el juego de instrucciones base para procesadores de 32 bits (RV32I) y algunas extensiones como la de multiplicación y división (M); pero es un simulador en continuo desarrollo y, presumiblemente, integrará más extensiones en el futuro.

Algunas características del procesador concreto implementado por el simulador son:

- Procesador tipo RISC (arquitectura de carga y almacenamiento) de 32 bits
- Buses de datos y de direcciones de 32 bits.
- Almacenamiento en memoria en *Little Endian*.
- 32 registros de propósito general de 32 bits (el Anexo 1 muestra la nomenclatura y las características de cada registro)
- Coprocesador de coma flotante (identificado como coprocesador 1) con un banco de 32 registros de 32 bits (*figura 1*). Estos registros pueden:
  - Contener números en coma flotante de simple precisión (32 bits).
  - O agruparse por pares para almacenar números en coma flotante de doble precisión (64 bits).

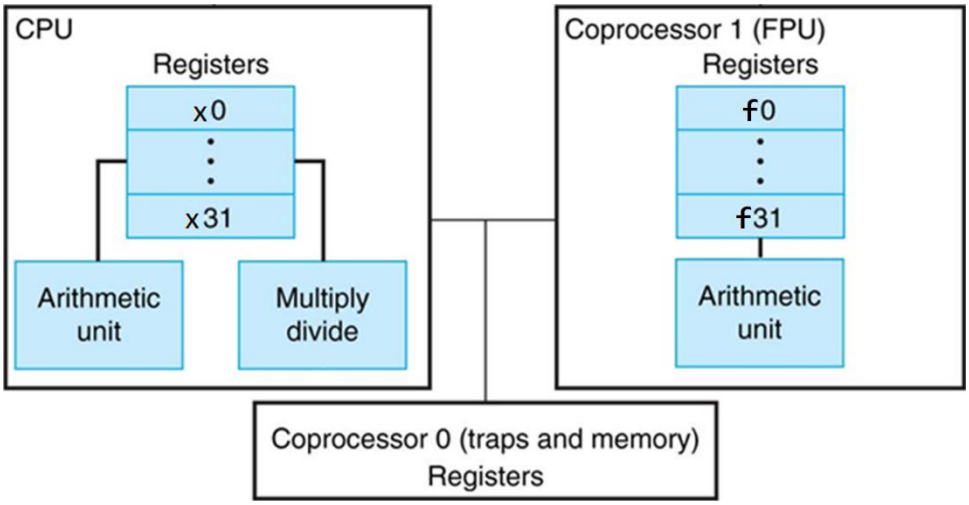


Figura 1. Unidades de procesamiento

1.2. MODOS DE DIRECCIONAMIENTO.

Los modos de direccionamiento hacen alusión a las diferentes nomenclaturas para especificar un operando dentro de una instrucción en lenguaje ensamblador. En las arquitecturas RISC, cada tipo de instrucción sólo admite un conjunto reducido de modos de direccionamientos. A continuación, se muestran todos los modos de direccionamientos de una arquitectura tipo RISC con ejemplos de algunas instrucciones que los usan.

A. Registro. El operando está en el registro (especificado en el anexo 2 como *reg* para registros de enteros y *freg* para registros en FP). Para especificar los registros de enteros en ensamblador puede utilizarse dos nomenclaturas. El símbolo 'x' seguido del número del registro…

Ej. *add x1,x2,x3* # x1 ← x2 + x3

B. Inmediato. El operando es una constante almacenada en la propia instrucción (especificado como *imm* en la lista de instrucciones del Anexo 2).

Ej. *addi x1,x2,* ***2000*** # x1 ← x2 + 2000

C. Registro base más desplazamiento. Direccionamiento a memoria en el que la dirección efectiva viene determinada por el contenido de un registro (registro base) más una constante (desplazamiento). Su sintaxis es *inm(reg)*, siendo *reg* el registro base e *inm* el desplazamiento.

Ej. *lw x1,* ***24(x2)*** # x1 ← Mem([x2]+24)
sb x5, ***20(x6)*** # Mem([x6]+20) ← x5

Cabe recordar que este modo de direccionamiento sólo se usa en instrucciones de acceso a memoria (cargas y almacenamientos) por lo que nunca puede ser utilizado para realizar operaciones ALU en memoria.

D. Relativo al Contador de Programa (PC). Modo de direccionamiento empleado en todos los tipos de saltos (como instrucciones *jal, beq*, *bne*) en el que la dirección destino de salto se obtiene sumando una constante en complemento A2 al registro contador de programa (PC).

A efectos prácticos, este modo queda oculto en los entornos de programación mediante el uso de etiquetas.

Ej. *beq x1,x2* ***sigue***
…
sigue: …

1.3. INSTRUCCIONES:

La estructura general de una instrucción RISC es:

<etiqueta> <instrucción> <operandos> <comentarios>

A. Etiqueta: identifica simbólicamente la dirección de memoria donde se encuentra la instrucción. Resulta de interés especialmente en las instrucciones de salto.

Las etiquetas pueden contener:
- Cualquier secuencia de letras, números y guion bajo ('_')
- No pueden comenzar por un número.
- Debe finalizar con dos puntos (':')

Ej. *repite:* *_bucle:* *bucle_ppal:* ~~*3error:*~~

B. Comentarios: añade información útil al usuario pero irrelevante para el compilador (el compilador lo elimina durante la compilación). Los comentarios deben comenzar con almohadilla ('#') y finalizar con el fin de línea (esto es, son comentarios de línea). Este campo es opcional.

Ej. *add x1,x2,x3* ***# calcula el total***

C. Operandos: el número de operandos depende del tipo de instrucción y su número puede oscilar entre 0 y 3. Salvo excepciones, el primero de ellos actúa como destino y el resto como fuente.

1.4. REPERTORIO DE INSTRUCCIONES

El Anexo 2 muestra el juego de instrucciones soportado por el simulador. A continuación, se muestra una clasificación de las instrucciones atendiendo al tipo de operación que realizan:

A. Instrucciones ALU: realizan operaciones aritméticas y lógicas (siempre de 32 bits). Estas instrucciones suelen emplear dos o tres operandos según sea una operación unaria o binaria, y los modos de direccionamiento registro e inmediato con ciertas restricciones pero en ningún caso operandos en memoria.

Para una misma operación (por ejemplo, *sumar*) suele haber varias instrucciones que realicen la misma operación pero con ciertas variaciones:

- Con o sin signo: Las instrucciones sin signo emplean el sufijo u (de *unsigned*).
- Entre registros o con una contante: Las que emplean una constante utilizan el sufijo i (de *immediate*)

Por ejemplo, para la suma se dispone de las instrucciones:

add x1,x2,x3 # x1 ← x2 + x3
addi x1,x2,25 # x1 ← x2 + 25 La constante extiende el signo hasta 32 bits

Pero hay muchas operaciones ALU que no tienen todas las variantes:

- En las operaciones lógicas (OR, XOR, AND) no existen versiones sin signo, pues no tiene sentido en operaciones lógicas. En este caso, las versiones que operan con constantes de menos de 32 bits (ORI, XORI, ANDI) extienden con ceros hasta 32 bits.
- En multiplicaciones y divisiones no existe el direccionamiento inmediato por razones de implementación.

La instrucción de suma conjuntamente con el registro x0 (que siempre vale 0) pueden utilizarse para inicializar el contenido de los registros a una determinada constante o para realizar transferencias entre registros.

Ej. *addi x10,x0,365* # x10 ← 365

add x10,x0,x15 # x10 ← x15

También se dispone de instrucciones de comparación de operandos. Estas instrucciones establecen el registro destino a "1" (verdadero) o "0" (falso) en función del resultado de la comparación de dos operandos fuente. Estas instrucciones suelen utilizarse conjuntamente con las de salto condicional para definir cualquier condición de salto (se verá más adelante). Por ejemplo, la instrucción de comparación "menor que" *slt* (del inglés: *set if less than*) sería:

Ej. *slt x10,x11,x12* # if (x11<x12) x10=1; else x10=0;

De ésta también hay diferentes versiones según compare con o sin signo, o si compara dos registros o un registro con un inmediato (*slt, sltu, slti, sltiu*)

Finalmente, el procesador también dispone de instrucciones de desplazamiento. Tales instrucciones pueden ser a su vez lógicas o aritméticas (ésta última sólo utilizada en desplazamientos a la derecha).

Ej. *slli x1,x2,5* #Desplaza x2 a la izquierda 5 bits insertando ceros por la derecha.
sll x1,x2,x3 #Como el anterior pero el nº de desplazamiento viene dado por x3
sra x1,x2,x3 #Desplaza x2 a la derecha un nº de bits dados por x3

B. Instrucciones de carga y almacenamiento*:* son instrucciones para la transferencia de datos entre registro y memoria. Las instrucciones de carga (*load*) leen de memoria y almacenan en un registro y las de almacenamiento (*store*) leen de un registro y lo almacena en memoria. Todas ellas utilizan dos operandos, uno de ellos emplea el modo de direccionamiento a registro *reg* y otro el modo de direccionamiento a memoria mediante registro base más desplazamiento *inm(reg)*.

Tanto de carga como de almacenamiento hay múltiples versiones según el tamaño de la transferencia de datos. El tamaño del dato puede ser de 8, 16 o 32, por lo que se tienen los siguientes especificadores de longitud de acceso:

Word: transferencias de 32 bits (una palabra)
HalfWord: " de 16 bits (media palabra)
Byte: " de 8 bits (un byte).

Además, las instrucciones de carga de menos de 32 bits pueden ser con o sin signo según se extiendan en el registro destino con el signo o con ceros (versión sin signo)

Algunos ejemplos:

lw x1, 32(x2)	**(load word)** Carga en x1 una palabra de 32 bits leída de la dirección de memoria x2+32
lbu x1, 32(x2)	**(load byte unsigned)** Lee byte leído de la dirección de memoria x2+32, extiende sin signo el byte a 32 bits y lo guarda en x1
sh x1, 32(x2)	**(store halfword)** Guarda a partir de la dirección x2+32 los 16 bits más bajos del registro x1

Los accesos a memoria deben estar alineados al tamaño de la transferencia, esto es, los accesos de tipo word deben estar alineados a 32 bits o los de tipo half a 16 bits.

C. Instrucciones de salto condicional: son instrucciones que saltan (modifican el flujo de ejecución del programa) si se cumple cierta condición entre dos registros o con el cero. Las más utilizadas son:

beq x1,x2,etiqueta # si x1 es igual a x2 salta a etiqueta
bne x1,x2,etiqueta # si x1 es distinto de x2 salta a etiqueta
bge x1,x2, etiqueta # si x1 es mayor o igual que x2 salta a etiqueta
blt x1,x2,etiqueta # si x1 es menor estricto que x2 salta a etiqueta

Existen otras para comparaciones de desigualdad con cero: *bgez, bgtz, blez, bltz*

Y otras más que son *pseudoinstrucciones* formadas por una instrucción de comparación tipo *set* y una instrucción de salto condicional.

Ej. *ble x1,x2,etiqueta* → *bge x2,x1,e*tiqueta

D. Instrucciones de salto incondicional: aquellas en las que el salto no está sometido a condiciones. En RISC-V, estas instrucciones saltan y enlazan (*jump and link*), almacenando en un registro la dirección actual antes de saltar. El modo de direccionamiento sigue siendo relativo al PC pero, habitualmente, con un inmediato de mayor tamaño.

Ej. *jal* x1,*etiqueta* # Salto a etiqueta con salva de la dirección de retorno en x1

E. Otras instrucciones:

la: (*load address*) *pseudoinstrucción* que almacena la dirección de memoria referenciada por una etiqueta en un registro.

Ej.
```
.data
        valores:  .word  10,-20,30,40
.text
        la x1, valores     # carga en x1 la dirección apuntada por valores
```

li: (*load immediate*) *pseudoinstrucción* que guarda una constante de 32 bits (*inm32*) en un registro.

Ej. *li x1, 0xAABBCCDD*

1.5. ESTRUCTURA DE UN PROGRAMA EN ENSAMBLADOR PARA RISC-V

En esencia, un programa en ensamblador se encarga de describir qué debe cargarse en la sección de la memoria reservada para instrucciones (programa a ejecutar) y qué datos (con sus respectivos valores) tendrá almacenada inicialmente la parte de la memoria reservada para datos. A cada una de estas sesiones se las denomina segmento; teniendo entonces dos segmentos:

- El segmento de código (dirección de memoria donde se cargarán las instrucciones). Se marca inicialmente con la directiva ".text".
- El segmento de datos (establecer los valores iniciales de datos almacenados en memoria). Se marca inicialmente con la directiva ".data".

Además, los compiladores disponen de un conjunto de directivas que permiten configurar ciertas características y acciones del proceso de compilación. El Anexo 3 muestra una tabla con las directivas del compilador más empleadas. A continuación, se muestra una estructura típica de un programa en ensamblador con un ejemplo:

Ej:
```
.data
        valores:    .word 23, 65, -320, 125, 0xFF, -10, 299, 322, 237,-12
        res:        .space 4
.text
        addi x1,x0,10
        add x10,x0,x0
        la x2, valores   # carga la dirección
sigue:  lw x5, 0(x2)
        add x10,x10,x5
        addi x2,x2,4
        addi x1,x1,-1
        bne x1,x0, sigue
        sw x10, 0(x2)
```

2. FUNCIONAMIENTO DEL SIMULADOR VISUALRISCV

2.1. HERRAMIENTAS BÁSICAS

VisualRiscV32 es un simulador de RISC-V para Windows que permite fundamentalmente visualizar la ejecución segmentada, la memoria y los registros del procesador durante la simulación de un programa en ensamblador. La aplicación se encuentra disponible en la plataforma de enseñanza virtual y no precisa instalación alguna. En esta sesión de laboratorio sólo se introducirán aquellas secciones del simulador necesarias para el desarrollo de la práctica.

El simulador se compone fundamentalmente de siete ventanas que el usuario puede mantener de forma independiente de la ventana principal, acoplarla a la ventana principal agrupadas en forma de pestañas o subdividiendo el área de otras ya acopladas. Al iniciar la aplicación las ventanas se encuentran distribuidas como muestra la Figura 2.

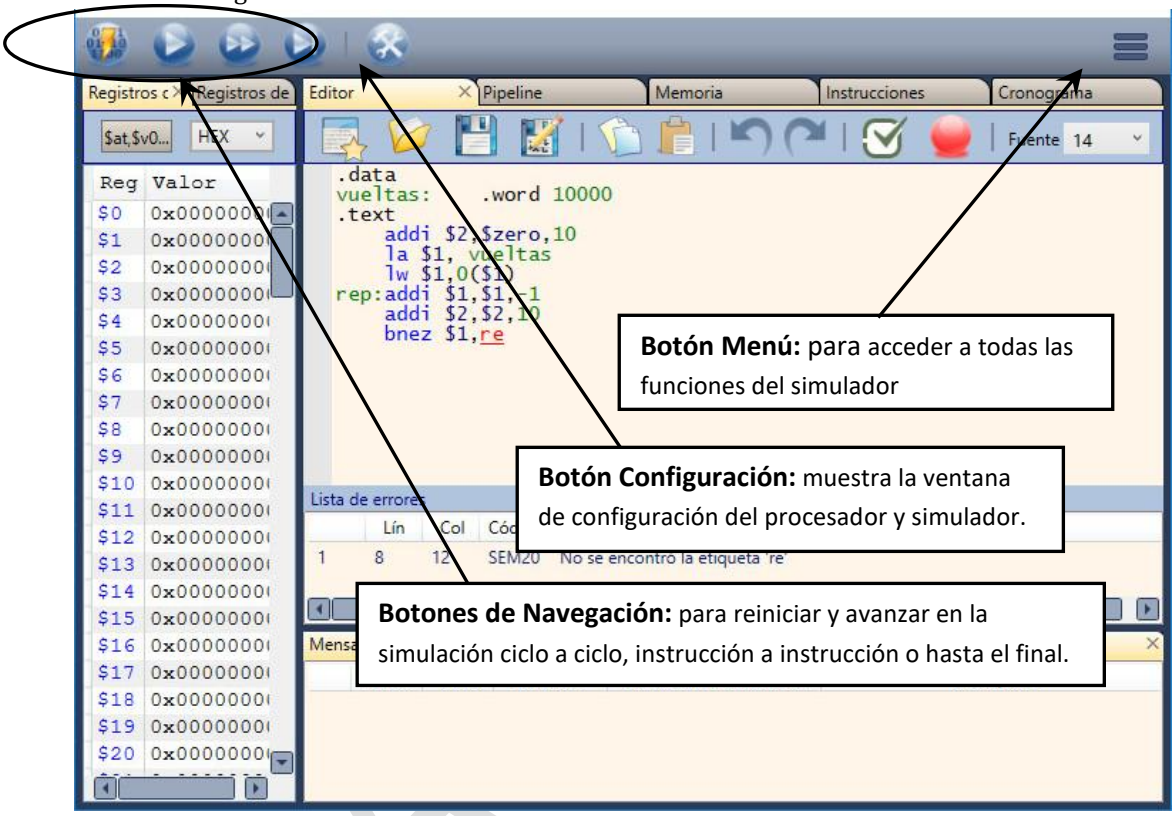


Figura 2. Visión general del simulador

A. Ventana *Editor*: utilizada principalmente para cargar, editar y compilar el código en ensamblador a simular. Los programas se guardan con la extensión *.asm*

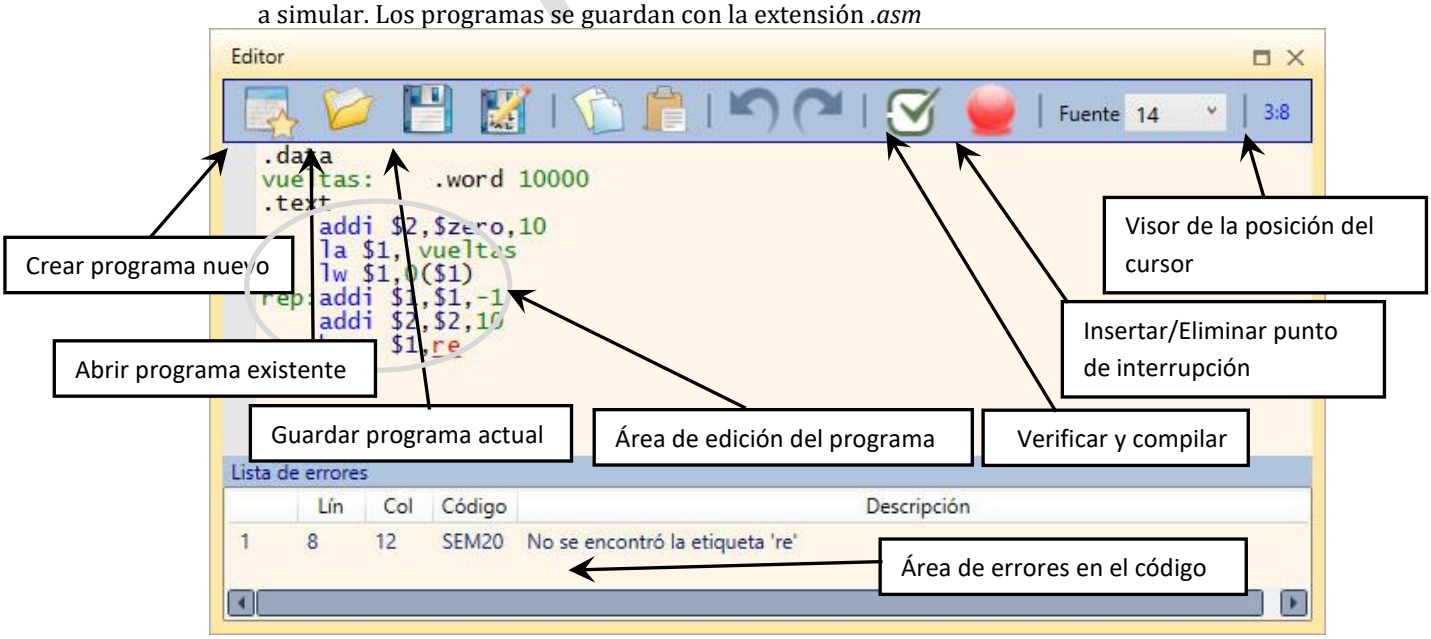


Figura 3. Ventana Editor

El botón ***Verificar y Compilar*** detecta los errores del código, si la opción *Autocompilar* de la ventana de configuración está activa (por defecto lo está) los errores se mostrarán automáticamente en la ventana *mensajes*. Esta ventana permite además establecer los puntos de interrupción (*breakpoints*) antes de comenzar la simulación. Al comenzar la simulación, los puntos de interrupción se transfieren a la ventana *Instrucciones*.

B. Ventana *Mensajes*: lista los errores del código durante su compilación y las excepciones generadas por el procesador durante la ejecución.

C. Ventana *Memoria*: muestra y permite editar el contenido de la memoria tanto del segmento de código como del segmento de datos (la región destinada al segmento del *kernel* no es accesible). Cuando se escribe en memoria, bien sea por el usuario o el procesador, la ventana muestra en rojo el último dato que ha sido modificado.

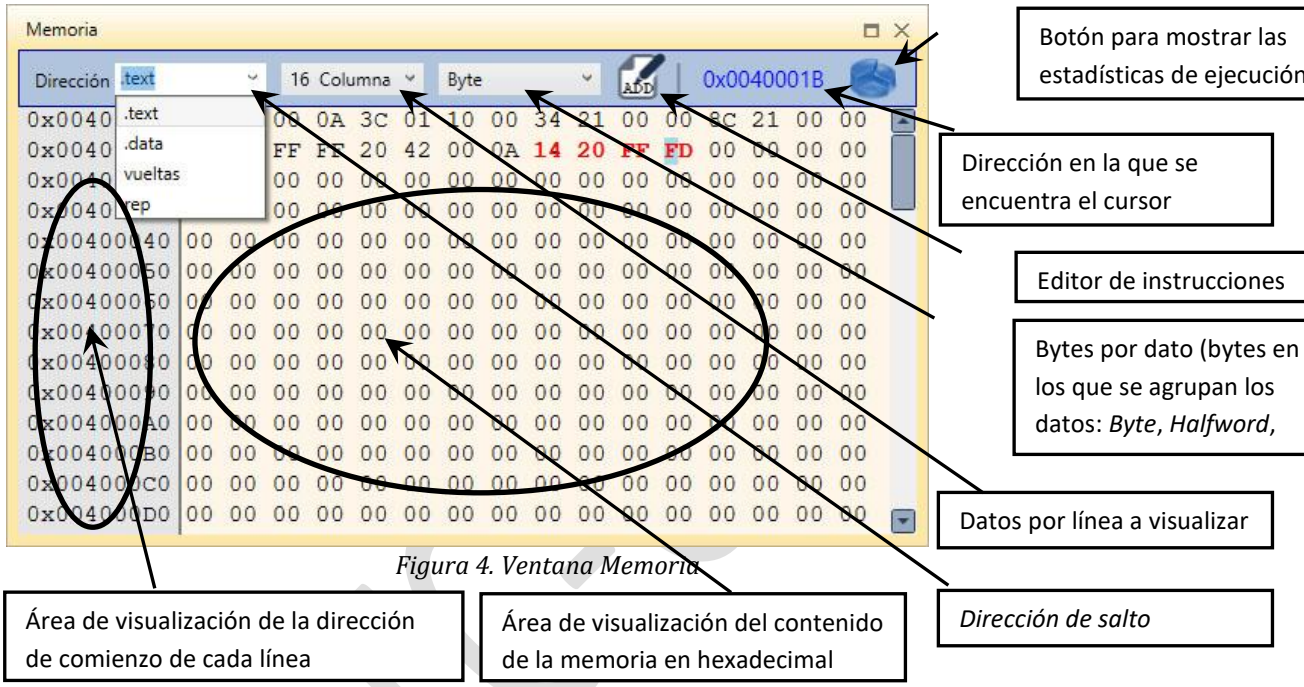


Figura 4. Ventana Memoria

El elemento *Dirección de salto* que aparece en la *figura 4* permite al usuario acceder rápidamente a una zona determinada de la memoria. En ella puede indicarse:

- Una dirección de memoria en hexadecimal. Ej. *0x00400000*
- El nombre de un segmento. Por ejemplo, para ir al comienzo del segmento de código hay que escribir *.text* o para el de datos especificar *.data*
- El nombre de una variable definida en el programa. En el ejemplo de la *figura 4 vueltas* o *rep* .

D. Ventanas de *Registros*: consiste en dos ventanas para visualizar y modificar el contenido de los registros del banco de registros de enteros y los del banco de coma flotante.

Figura 5. Ventanas de Registros

E. Ventana *Instrucciones*: muestra la codificación y la interpretación de las instrucciones almacenadas en el segmento de código así como las fases en las que se encuentra cada instrucción. Esta ventana permite además editar tales instrucciones y establecer puntos de interrupción durante la ejecución.

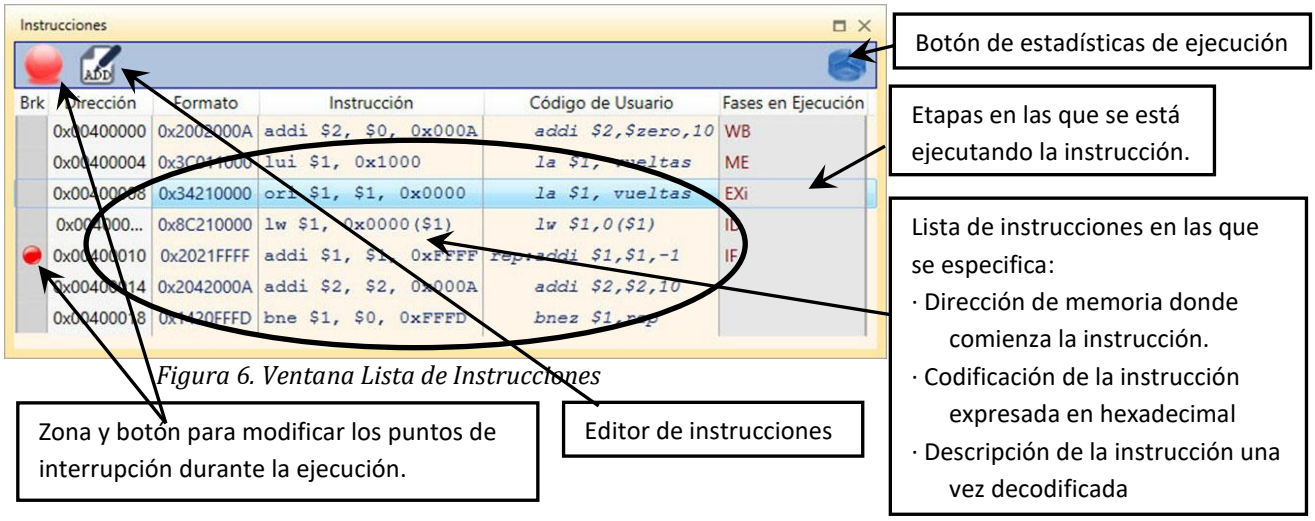


Figura 6. Ventana Lista de Instrucciones

F. Ventana *Pipeline*: representa un esquema de las etapas del procesador y muestra en cada ciclo de ejecución las instrucciones que se encuentran en cada etapa y su estado (si la instrucción ha ejecutado la fase, qué tipo de bloqueo se ha producido, etc.).

G. Ventana *Cronograma*: muestra un cronograma de ejecución, el estado de cada etapa al final de cada ciclo.

A nivel avanzado, puesto que el comportamiento del procesador de VisualRISCV32 configurable, es necesario establecer las propiedades del procesador según las características u optimizaciones que queremos que estén presentes. Es por ello por lo que debemos verificar la configuración para interpretar correctamente los resultados incluso tras arrancar el simulador ya que el simulador carga por defecto la última configuración utilizada en anteriores sesiones.

Para configurar el procesador que simula (en realidad todo el computador) hay 2 posibilidades:

- Abrir la ventana propiedades a través del botón  o directamente pulsando F12 y acceder, para nuestro caso, a la sección "Procesador".
  En esta misma ventana puede reestablecer la configuración por defecto en la sección 'configuración'. Esta opción resulta útil en caso duda o si desconoce los cambios que hayan podido ser realizados respecto a la configuración original.
- También puede configurarse el sistema total o parcialmente a través de directivas de configuración dentro de la sección de configuración (*.config*) del programa fuente.
  Esta solución puede resultar especialmente útil cuando se pretende realizar varias simulaciones con diferentes configuraciones.
  Tenga en cuenta que la configuración mediante directivas es **predominante** por lo que, antes de iniciar la simulación, el simulador tomará como base la configuración existente en la ventana 'Propiedades' y aplicará los cambios especificados mediante directivas.
  Algunas directivas de configuración que podrían ser de interés para la práctica son:

proc (single | multi | pipelined) forwarding? delayed?	Simula un procesador secuencial de ciclo único (*single-cycle*), secuencial de ciclo múltiple (*multi-cycle*) o segmentado (*pipelined*). En caso de ser segmentado puede opcionalmente activarse los adelantamientos (forwarding) o los saltos retardados (delayed).
forwarding on | off	Activa/desactiva los adelantamientos. (optimización aplicable a procesadores segmentados para reducir los bloqueos por dependencia de datos RAW).
delayed on | off	Activa/desactiva los saltos retardados (optimización de procesadores segmentados para reducir los bloqueos de control)

exm NUMCICLOS [pipelined]	Especifica el número de ciclos de la unidad aritmética de la multiplicación y si la unidad está segmentada (con 'pipelined' se especifica que está segmentada).
exd NUMCICLOS [pipelined]	Como la directiva anterior pero aplicable a la unidad aritmética de la división.

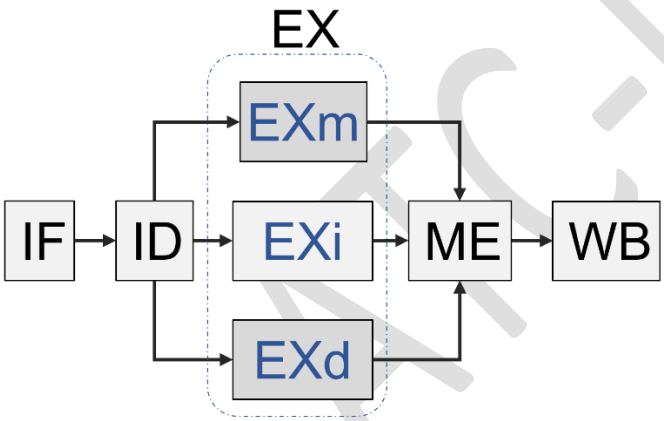
2.2. CAMINO DE DATOS

Según se ha visto en teoría, el diseño de nuestra arquitectura dispone de una ALU en la que se realiza toda operación aritmética o lógica. En la práctica un procesador puede disponer de otras unidades aritméticas para realizar el cálculo de operaciones específicas como puede ser la multiplicación, la división u operaciones en coma flotante.

Puesto que puede haber varias unidades aritméticas, resulta necesario añadir a nuestra arquitectura nuevas rutas de datos que encaminen los operandos hacia la unidad aritmética adecuada y que posteriormente transfiera el resultado a la siguiente etapa.

En el caso de VisualRISCV32 podemos distinguir actualmente 3 unidades aritméticas: la unidad de la multiplicación (EXm) utilizada exclusivamente por las instrucciones de multiplicación (mul, mulh, mulhu, mulhsu), la unidad de la división (EXd) empleadas por las instrucciones relacionadas con la división (div, divu y remu) y por supuesto, la unidad aritmético-lógica para enteros básica (EX) que se encarga del resto de operaciones (incluyendo el cálculo de direcciones para el acceso a memoria) y que se identifica con la ALU estudiada en clase de teoría.

Estos 3 componentes pertenecen a la etapa EX, por lo que la ruta de datos básica del RISCV queda como se aprecia en la siguiente figura:



- **Unidad de enteros principal (*EXi*):** encargada de las cargas, almacenamientos, saltos y las operaciones ALU enteras excepto las de multiplicación y división.
- **Unidad para multiplicación de enteros (*EXm*)**
- **Unidad para la división de enteros (*EXd*)**

Figura 7. Pipeline con las unidades funcionales independientes

En la ventana Pipeline del simulador podemos ver esta ruta de datos básica que mostrará la instrucción que se encuentra en cada etapa y que será de interés para estudiar el secuenciamiento de instrucciones en un procesador segmentado. La siguiente figura muestra una imagen de la ventana Pipeline:

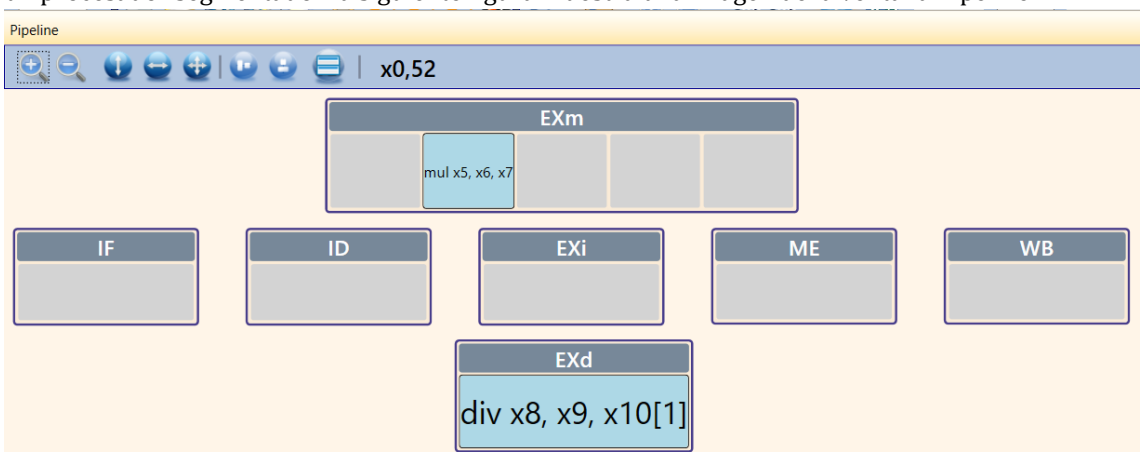


Figura 8. Ventana Pipeline del simulador que muestra una instrucción de multiplicación y división en ejecución

Puesto que las unidades aritméticas *EXm* y *EXd* realizan operaciones más complejas suelen precisar de varios ciclos de reloj para realizar el cálculo (parámetro configurable). Esto supone que las instrucciones que utilizan alguna de estas unidades requerirán varios ciclos en su fase *EX*.

Por ejemplo, una instrucción de multiplicación en el simulador requiere por defecto 5 ciclos por lo que su ejecución requiere 9 ciclos como se muestra en la siguiente figura:

	1	2	3	4	5	6	7	8	9
mul x5, x6, x7	IF	ID	EXm0	EXm1	EXm2	EXm3	EXm4	ME	WB

Figura 9. Cronograma de una instrucción de multiplicación de 5 ciclos en EXm (larga duración)

Además, estas unidades aritméticas que requiere varios ciclos pueden ser, a su vez, secuenciales si sólo pueden realizar una operación aritmética a la vez, o segmentadas si son capaces de realizar varias operaciones aritméticas en paralelo (cada operación en curso deberá estar en un ciclo diferente). Por defecto, la multiplicación está segmentada y la división es secuencial, aunque es configurable.

A las instrucciones que requieren unidades aritméticas con varios ciclos de ejecución se las denomina instrucciones de larga duración.

2.3. RUTA DE DATOS DETALLADA DEL SIMULADOR

El simulador VisualRISCV32 muestra una posible interpretación de la arquitectura interna del procesador RISC-V a nivel de transferencia de registros (RT). En su diseño, se detallan las unidades funcionales y el camino de datos y cómo éstos están siendo utilizado por cada una de las instrucciones ejecutadas mostrando los valores que van tomando durante la simulación.

A partir de este diseño, el alumno podrá interpretar de forma exhaustiva el funcionamiento interno del procesador RISCV y facilitar la compresión de las diferentes optimizaciones que serán aplicadas al modelo básico del procesador en posteriores sesiones de laboratorio.

Para acceder a la ventana que contiene el diseño del circuito con la ruta de datos acceda a *Menú* → *Ventanas* → *Ruta de Datos* o bien pulsando Ctrl + R.

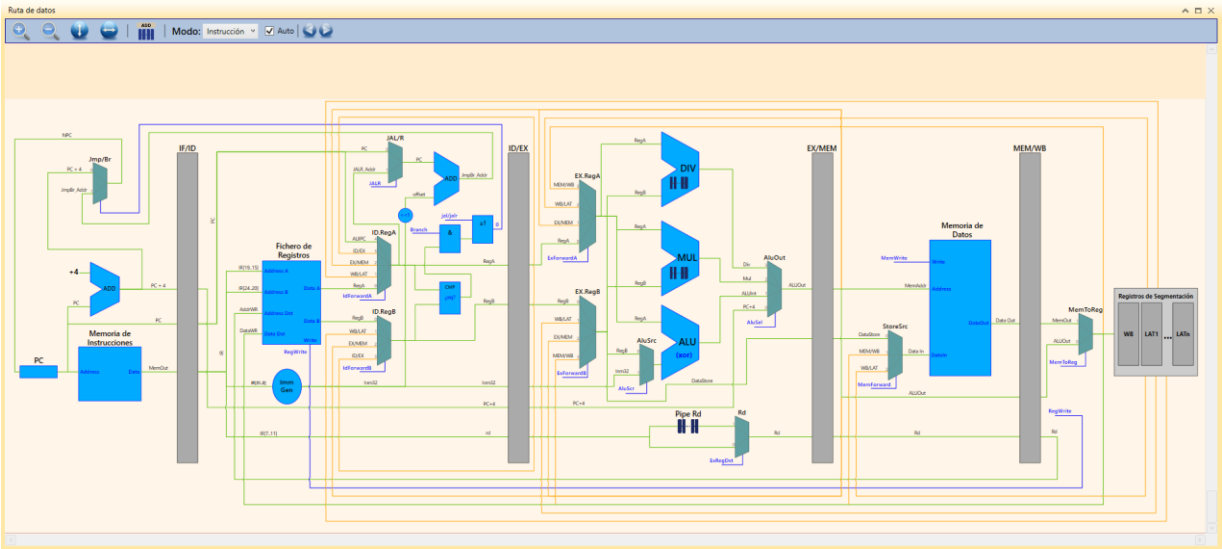


Figura 9. Ventana 'Ruta de Datos'

La figura anterior muestra una instantánea de la ventana Ruta de Datos. Además del propio circuito mostrado en la parte inferior de la ventana, se distinguen 2 zonas:

a) **La barra de control:** La barra de control ubicada en la parte superior y contiene entre otros, los elementos necesarios para ajustar el zoom, modificar el comportamiento del diseño y botones de navegación.

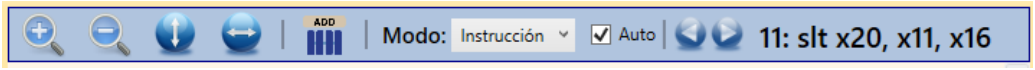


Figura 10. Barra de control de la ventana 'Ruta de Datos'

De todos ellos, resulta de interés para la práctica el selector de *Modo*. Este selector permite alternar entre las siguientes opciones:

- **Modo Ciclo:** en este modo, el diseño de la ruta de datos mostrará el estado del circuito en el ciclo seleccionado actualmente en la ventana cronograma. En el caso de simular el procesador segmentado, mostrará los detalles de la instrucción que se encuentra en cada etapa para el ciclo seleccionado. Como ejemplo, al seleccionar en la ventana de cronograma la etapa EX de la instrucción, la ruta de datos mostrará el estado del procesador en el ciclo 5, detallando el comportamiento de la instrucción 1 en WB, la instrucción 2 en ME, etc.

  La información relativa al formato, decodificación, comportamiento o estado de la instrucción que se encuentra en cada etapa se describe en la zona de instrucciones que más adelante se describe.
- **Modo Instrucción:** Todas las etapas mostrarán la ejecución completa de la instrucción seleccionada en el cronograma. En el caso de la figura que aparece a continuación, mostrará los detalles de la ruta para la instrucción 3.

Por comodidad, puede utilizar los botones de navegación  para pasar a la instrucción anterior o la siguiente sin necesidad de seleccionarla en el cronograma. La instrucción seleccionada se mostrará en la parte derecha de la barra de control.

	1	2	3	4	5	6
1: [65536] addi x10, x0, 10	IF	ID	EX	ME	WB	
2: [65540] ori x5, x0, 5		IF	ID	EX	ME	WB
3: [65544] addi x11, x0, 256			IF	ID	EX	ME
4: [65548] slli x11, x21, 8				IF	ID	EX
5: [65552] addi x30, x0, 0					IF	ID
6: [65556] lb x12, 0(x11)						IF

Figura 11. Ventana Cronograma. Selección de la instrucción 3 o el ciclo 5 según el modo de visualización empleado.

Por último, la zona de control integra la opción "Auto" que selecciona automáticamente el último ciclo ejecutado cuando está activa. Esta opción resulta especialmente útil para observar la evolución de la ruta de datos a medida que va sucediendo la ejecución de las instrucciones, pues permite apreciar cómo se transmiten las señales de unos componentes a otros a medidas que las instrucciones avanzan por el pipeline.

b) **La zona de instrucciones:** se encuentra en el área intermedia y está reservada para mostrar la instrucción que está en ejecución en cada una de las etapas, así como información relativa a ésta. Esta zona tiene sentido cuando se utiliza el modo "Ciclo" ya que, en modo instrucción, todas las etapas están ejecutando la misma instrucción.

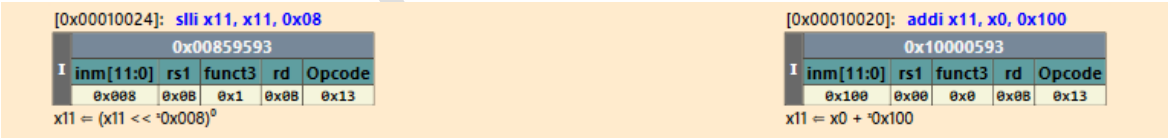


Figura 12. Ventana Ruta de Datos

3. CÁLCULO DEL RENDIMIENTO MEDIANTE EL CRONOGRAMA

La forma más habitual de medir el rendimiento de un procesador es mediante el cálculo del tiempo de CPU, esto es, el tiempo que tarda en ejecutar un programa sin tener en cuenta los accesos a la E/S, otros procesos, etc.

Según hemos visto en clase de teoría, el cálculo del tiempo de ejecución de un procesador puede obtenerse de forma teórica mediante la expresión:

$$TiempoCPU_{seg} = NumInstruc \cdot CPI \cdot \tau$$

Donde cada uno de los factores que forman la expresión dependerá del tipo de procesador (de los 3 procesadores básicos estudiados) además de otros elementos como puede ser el compilador.

Otra forma de obtener el tiempo de ejecución consiste en interpretar el cronograma de ejecución. De esta forma podemos obtener este tiempo en ciclos (a falta del aplicar el periodo de reloj) y que será la que abordemos en esta sección con ayuda del simulador.

El simulador dispone de una ventana donde se muestra el cronograma de ejecución (Ctrl+G) para poder extraer esta información. Esta ventana será muy utilizada en esta práctica y muy especialmente en prácticas posteriores por lo que resulta conveniente que se familiarice con ella.

En el caso de procesadores secuenciales, el cálculo del tiempo de ejecución en ciclos mediante el cronograma es muy sencillo y no supone un reto, pues basta con obtener el ciclo en el que se ejecuta la última etapa (WB) de la última instrucción sobre el cronograma.

El cálculo de este tiempo en un procesador segmentado resulta algo más complejo pues, en primer lugar, la ejecución de una instrucción comienza antes de finalizar las anteriores, por lo que habrá varias instrucciones en ejecución en un mismo ciclo de reloj.

De lo anterior se deduce que, al comienzo de la ejecución de nuestro programa, aún se están ejecutando instrucciones anteriores que no pertenecen a nuestro programa (o simplemente se está produciendo el llenado del pipeline) y que antes de finalizar la ejecución de la última instrucción de nuestro programa ya habrá comenzado la ejecución de instrucciones posteriores no pertenecientes a nuestro programa. Dicho de otra forma, durante la ejecución de las instrucciones de nuestro programa habrá también otras instrucciones que están finalizando su ejecución o que comenzarán su ejecución.

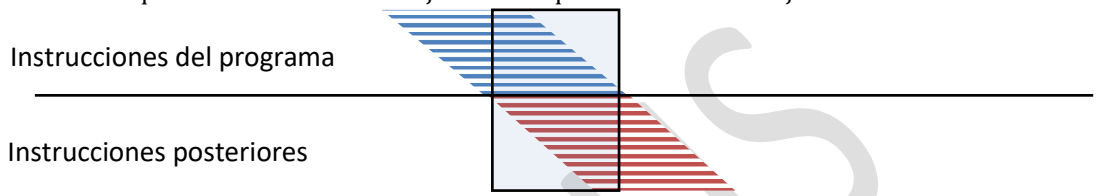


Figura 13. Ejecución segmentada de las instrucciones del programa y posteriores.

Una forma de medir este tiempo en procesadores segmentados a partir del cronograma consiste en contabilizar **los ciclos transcurridos hasta la etapa IF (ejecutada) de la primera instrucción posterior al programa menos 1**. Esta forma de medir el tiempo, aunque no es infalible, es válida en la mayoría de los casos.

Por ejemplo. En el cronograma de la siguiente figura, el tiempo de ejecución del procesador en ciclos es 7, pues la primera instrucción no perteneciente al programa ejecuta su etapa IF en el ciclo 8, que al restarle 1, obtenemos 7 y que coincide en este caso con el número de instrucciones ejecutadas.

	1	2	3	4	5	6	7	8
1: [65536] addi x5, x0, 1	IF	ID	EX	ME	WB			
2: [65540] addi x0, x0, 0		IF	ID	EX	ME	WB		
3: [65544] addi x0, x0, 0			IF	ID	EX	ME	WB	
4: [65548] addi x5, x0, -1				IF	ID	EX	ME	WB
5: [65552] addi x0, x0, 0					IF	ID	EX	ME
6: [65556] addi x0, x0, 0						IF	ID	EX
7: [65560] addi x5, x0, -12							IF	ID
[65564] (Fuera de código)								IF

Figura 14. Cronograma de un procesador segmentado. Ciclo de referencia para el cálculo del tiempo de ejecución

4. INSTRUCCIONES DE LARGA DURACIÓN

Como se ha visto anteriormente, el computador dispone de varias unidades de cálculo, una integrada en el procesador y otras alojadas en un coprocesador independiente, cada una de ellas dedicadas a realizar ciertos tipos de operaciones. El simulador VisualRISCV32 simula una implementación RISC-V con las siguientes unidades:

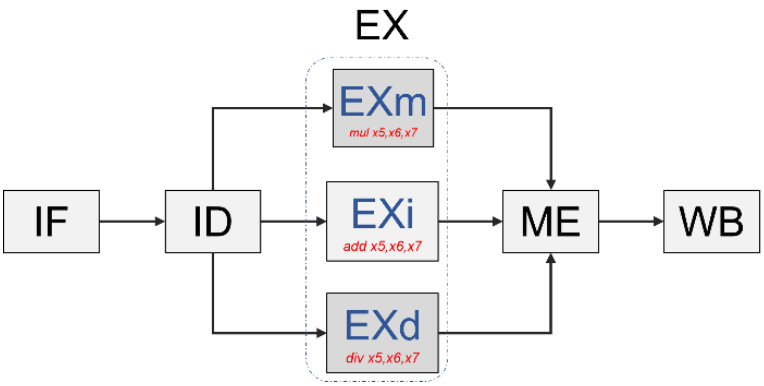


Figura 15. Pipeline con las unidades funcionales independientes

- **Unidad de enteros principal (*EXi*):** encargada de las cargas, almacenamientos, saltos y las operaciones ALU enteras excepto las de multiplicación y división.
- **Unidad para multiplicación de enteros (*EXm*)**
- **Unidad para la división de enteros (*EXd*)**

Las etapas *EXm* y *EXd* realizan operaciones complejas por lo que suelen precisar de varios ciclos de reloj para realizar el cálculo. Esto supone que las instrucciones que utilicen alguna de estas unidades más complejas requerirán varios ciclos en su fase *EX*.

Puesto que la fase *EX* dispone de varias unidades funcionales independientes (*figura 15*) y que algunas de estas unidades son multiciclo, puede darse el caso de que varias instrucciones puedan estar ejecutando su fase *EX* simultáneamente. Véase como ejemplo el cronograma de la *figura 16* en el que hay 3 instrucciones en EX durante el ciclo 5:

	1	2	3	4	5	6	7	8	9	10
mul x5, x6, x7	IF	ID	EXm0	EXm1	EXm2	EXm3	EXm4	ME	WB	
div x8, x9, x10		IF	ID	EXd	EXd	EXd	EXd	EXd	EXd	EXd
add x11, x12, x13			IF	ID	EX	ME	WB			

Figura 16. Ejecución de una instrucción de multiplicación de 5 ciclos en EXm(segmentada), una de división que requiere de 8 ciclos en EXd (no segmentada) y una instrucción add que ejecuta EX en la ALU convencional.

Además, estas unidades de cálculo multiciclo pueden estar a su vez segmentadas, esto es, que el cálculo de la operación puede estar dividido en varias subetapas (de un ciclo cada una). Esta división en varias subetapas segmentadas va a permitir que puedan realizarse varias operaciones en paralelo en la misma unidad funcional siempre que estén en subetapas diferentes. Este hecho permitirá iniciar un nuevo cálculo sin haber terminado la anterior pudiéndose incluso llegar a finalizar una operación compleja en cada ciclo de reloj.

Por ejemplo, la etapa de *EXm* de la *figura 17* está segmentada (este hecho se identifica en el simulador porque cada subetapa muestra un número que la identifica: subetapa *EXm0*, *EXm1*...) lo que permite que la instrucción ***mul x14,x15,x16*** pueda comenzar su etapa *EXm* (ciclo 6) incluso antes de que haya finalizado la etapa *EXm* de la multiplicación anterior ***mul x5,x6,x7*** (ciclo 7). De esta forma, ambas instrucciones de multiplicación pueden ejecutar simultáneamente su etapa *EXm* siempre que se no se encuentren en la misma subetapa de *EXm*.

En cambio, según el cronograma de la *figura 17*, la unidad funcional de la división *EXd* no está segmentada pues no se identifican subetapas mediante un número, sino que *EXd* es una sólo etapa de varios ciclos. Al no estar segmentada *EXd*, toda instrucción de división debe esperar a que la etapa *EXd* esté completamente libre antes de iniciar dicha etapa. Este hecho puede comprobarse en el cronograma de la *figura 17*, pues la instrucción ***div x17,x18,x19*** queda bloqueada en ID a la espera de que ***div x8,x9,x10*** abandone *EXd* al final del ciclo 11.

Este bloqueo en ID es considerado de tipo estructural pues debe interpretarse como una falta de recursos en la etapa *EXd* no segmentada para simultanear en el mismo ciclo las operaciones requeridas, a diferencia de las etapas segmentadas.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
mul x5, x6, x7	IF	ID	EXm0	EXm1	EXm2	EXm3	EXm4	ME	WB							
div x8, x9, x10		IF	ID	EXd	EXd	EXd	EXd	EXd	EXd	EXd	EXd	ME	WB			
add x11, x12, x13			IF	ID	EX	ME	WB									
mul x14, x15, x16				IF	ID	EXm0	EXm1	EXm2	EXm3	EXm4	ME	WB				
div x17, x18, x19					IF	ID	ID	ID	ID	ID	ID	EXd	EXd	EXd	EXd	EXd
add x20, x21, x22						IF	IF	IF	IF	IF	IF	ID	EX	ME	WB	

Figura 17. Ejecución de instrucciones de multiplicación en la unidad EXm segmentada e instrucciones de división en EXd no segmentada. Las instrucciones que utilizan una unidad funcional no segmentada deben esperar a que la unidad quede libre para poder utilizarla, lo que produce bloqueos estructurales.

La *figura 18* muestra el estado del pipeline en el ciclo 7 según el cronograma de la figura anterior. En éste se puede apreciar las 2 instrucciones de multiplicación en *EXm* y la instrucción ***div x17,x18,x19*** en ID a la espera de que ***div x8,x9,x10*** finalice la etapa *EXd* no segmentada.

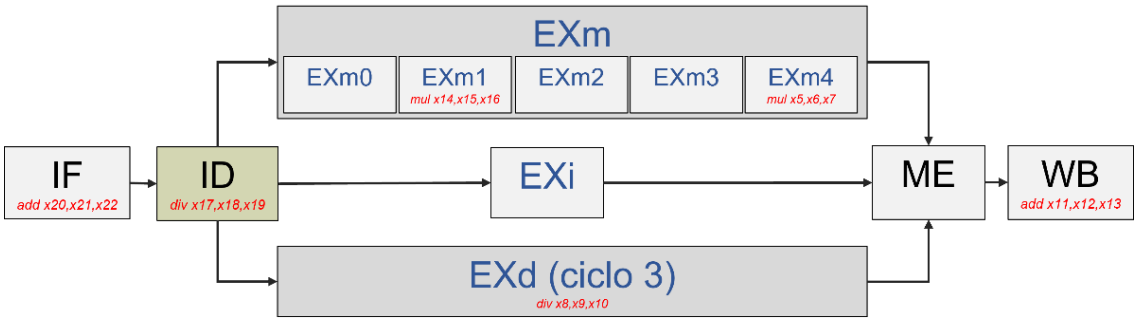


Figura 18. Estado del pipeline en el ciclo 7 de la figura 3. La etapa ID está marcada para representar el bloqueo estructural.

4.1. BLOQUEOS DEBIDOS A INSTRUCCIONES DE LARGA DURACIÓN

Se ha visto que hay instrucciones cuya ejecución de su fase EX requiere varios ciclos. La ejecución de estas instrucciones, denominadas instrucciones de larga duración, va a tener una serie de consecuencias sobre el rendimiento:

a) Incremento de los bloqueos de datos por dependencias RAW

Según se ha visto en teoría, los bloqueos por dependencia *RAW* (*read after write*) se producen para evitar que una instrucción posterior (instrucción dependiente) lea un operando antes de que otra anterior haya terminado de calcularlo. El aumento de este tipo de bloqueos en instrucciones de larga duración se debe a que los datos van a estar disponibles en ciclos más tardíos. El uso de adelantamientos (forwarding) sólo consigue reducir ligeramente estos ciclos de bloqueo. La siguiente figura muestra este tipo de bloqueo producido por una multiplicación:

	1	2	3	4	5	6	7	8	9	10	11	12
mul x5, x6, x7	IF	ID	EXm0	EXm1	EXm2	EXm3	EXm4	ME	WB			
add x8, x5, x9		IF	ID	~~*EX~~	~~*EX~~	~~*EX~~	~~*EX~~	EX	ME	WB		
sub x10, x8, x12			IF	~~ID~~	~~ID~~	~~ID~~	~~ID~~	ID	EX	ME	WB	
or x10, x10, x12				~~IF~~	~~IF~~	~~IF~~	~~IF~~	IF	ID	EX	ME	WB

Figura 19. Ejemplo de bloqueos RAW producidos por una instrucción de larga duración. Los bloqueos RAW aparecen en rojo, tachado y con un asterisco a su izquierda para simplificar su lectura.

El número de ciclos de bloqueos RAW que se produce en una instrucción que es dependiente de otra de larga duración inmediatamente anterior, será igual al número de ciclos que precisa la instrucción de larga duración en su etapa *EX* menos 1. Utilizando como ejemplo el cronograma de la *figura 19*, el número de ciclos de bloqueo de la instrucción add será de 4 puesto que depende de una instrucción mul inmediatamente anterior cuya etapa *EXm* precisa de 5 ciclos.

b) Aparición de bloqueos de datos por dependencias WAW

Los bloqueos por dependencia de salida o *WAW* (*write after write*) evitan que se realicen las escrituras sobre un registro en un orden incorrecto.

En los procesadores con instrucciones de larga duración, al existir etapas *EX* con distinto número de ciclos, una instrucción de larga duración podría realizar su etapa *WB* más tarde que otras instrucciones posteriores de menor duración, lo que provoca que se altere el orden natural de la etapa *WB* en ambas instrucciones.

Si ambas instrucciones tienen dependencias de salida debido a que comparten el mismo registro de destino, este cambio de orden en las etapas *WB* no debería producirse pues alteraría el orden de las escrituras en el registro que comparte produciéndose una ejecución incorrecta. Es por

ello por lo que la instrucción posterior es bloqueada en su etapa *EX* hasta que la instrucción anterior avance a *ME* lo que garantiza que no se produzca el cambio de orden.

La *figura 20* muestra 3 instrucciones con dependencia de salida, lo que exige que ninguna realice su etapa *WB* antes que otra. Puesto que la instrucción mul precisa 9 ciclos hasta su etapa WB y add sólo 5 ciclos, esta última es bloqueada en *EX* para evitar que realice su etapa *WB* antes que la instrucción mul.

	1	2	3	4	5	6	7	8	9	10	11
mul x5, x6, x7	IF	ID	EXm0	EXm1	EXm2	EXm3	EXm4	ME	WB		
add x5, x8, x9		IF	ID	~~EX*~~	~~EX*~~	~~EX*~~	~~EX*~~	EX	ME	WB	
sub x5, x10, x11			IF	~~ID~~	~~ID~~	~~ID~~	~~ID~~	ID	EX	ME	WB

Figura 20. Ejemplo de bloqueos WAW. Los bloqueos WAW se muestran en verde, tachado y con un asterisco a su derecha para una mayor legibilidad.

c) Bloqueos estructurales

Aunque en clase de teoría los bloqueos estructurales del procesador fueron resueltos, al considerar la existencia de instrucciones de larga duración y de varias unidades funcionales en la etapa *EX* que concurren en una misma etapa (véase *figura 15*), es posible que se produzcan bloqueos estructurales.

Con carácter general, habrá 2 situaciones en las que se puedan producir bloqueos estructurales en este procesador. El primer caso ya fue descrito al comienzo de este estudio teórico con relación a la *figura 17*, en el que una instrucción que ejecuta su etapa *EX* en una unidad no segmentada (como es el caso de ***div x17,x18,x19*** en la *figura 17*) puede bloquearse en *ID* a la espera de ser liberada la etapa EX no segmentada que precisa. Fíjese que esta situación no se produce por la existencia de distintas etapas *EX* (*EXm*, *EXd*…), puesto que podría consistir simplemente en una etapa *EX* multiciclo no segmentada común a todas.

El segundo caso en el que pueden producirse bloqueos estructurales sí está relacionado con la existencia de varias etapas *EX* que bifurcan el pipeline y que posteriormente desembocan en la misma etapa, en este procesador, en la etapa *ME*. En esta arquitectura, si a varias instrucciones les corresponde avanzar desde la etapa *EX* hacia la única etapa *ME* existente, evidentemente sólo una de ellas podrá pasar a *ME* (la instrucción anterior a todas) por lo que el resto de las instrucciones deberán ser bloqueadas hasta que *ME* esté disponible. Como anteriormente, este bloqueo se considera estructural al interpretarse que se produce por falta de recursos en *ME* para atender simultáneamente a todas las instrucciones que les corresponde avanzar a dicha etapa.

La siguiente figura muestra un ejemplo de este tipo de bloqueos en el que a las instrucciones *mul* y *and* les corresponde abandonar la etapa *EX* simultáneamente en el ciclo 7, pero sólo *mul* podrá avanzar a *ME* obligando a que *and* se bloquee.

	1	2	3	4	5	6	7	8
mul x5, x6, x7	IF	ID	EXm0	EXm1	EXm2	EXm3	EXm4	ME
add x8, x9, x10		IF	ID	EX	ME	WB		
sub x11, x12, x13			IF	ID	EX	ME	WB	
or x14, x15, x16				IF	ID	EX	ME	WB
and x17, x18, x19					IF	ID	EX^	EX

Figura 21. Ejemplo de bloqueo estructural. En el ciclo 7, la instrucción mul *va a avanzar a la fase ME, lo que impide que lo haga* and*. Los bloqueos estructurales aparecen en amarillo, tachado y con un acento circunflejo a su derecha para una mayor legibilidad.*

4.2. BLOQUEOS QUE NO DETIENEN LA CADENA

Otro fenómeno producido como consecuencia de la bifurcación de la etapa *EX* hacia diferentes unidades funcionales es la aparición de etapas bloqueadas (generalmente bloqueos de datos) que no provocan la detención del pipeline. Esta situación se produce cuando una instrucción está bloqueada en una etapa *EX*, pero las instrucciones posteriores e independientes pueden seguir avanzando de etapa al no utilizar la etapa *EX* bloqueada.

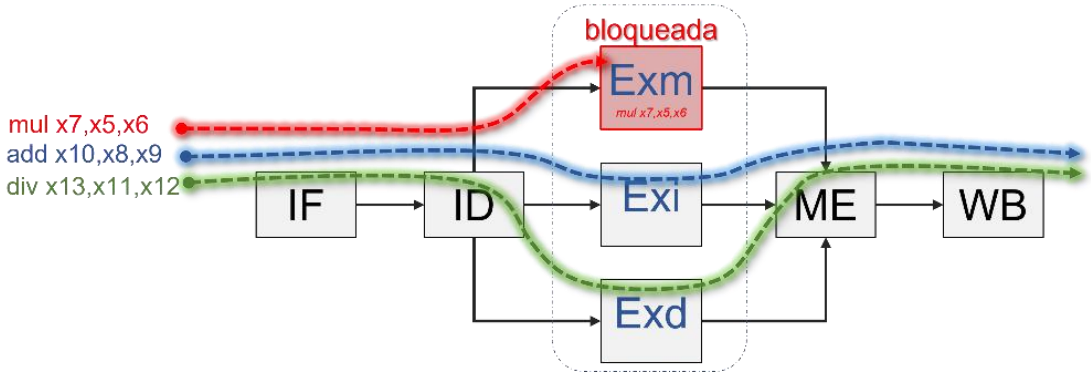


Figura 22. Ejemplo de avance de instrucciones por el pipeline a pesar del bloqueo.

La figura anterior ilustra un ejemplo de este tipo, en el que la instrucción de multiplicación queda bloqueada en *EXm* lo que no impide que otras instrucciones independientes que no utilicen *EXm* puedan continuar su ejecución.

Puesto que el pipeline no es detenido, estos bloqueos no deben ser tenidos en cuenta para calcular el tiempo de ejecución pues procesador sigue emitiendo una nueva instrucción en cada ciclo. Como puede verse en el cronograma de la siguiente figura, los bloqueos de la instrucción ***mul x7,x5,x6*** en los ciclos del 4 al 6 no han generado un atasco en las instrucciones add, div, sub y and por lo que no aparecen tachadas en el cronograma (en la vertical del bloqueo de mul).

Una situación similar ocurre en el ciclo 7, el bloqueo de mul no impide que add y div continúen, pero en este caso sí se produce la detención de la cadena por un bloqueo estructural de sub al coincidir en la etapa *MEM* con ***mul x5,x4,x3*** en el siguiente ciclo, por lo que la detención del pipeline debe asociarse al bloqueo estructural y no al bloqueo de datos.

	1	2	3	4	5	6	7	8	9	10	11	12
mul x5, x4, x3	IF	ID	EXm0	EXm1	EXm2	EXm3	EXm4	ME	WB			
mul x7, x5, x6		IF	ID	~~*EXm0~~	~~*EXm0~~	~~*EXm0~~	~~*EXm0~~	EXm0	EXm1	EXm2	EXm3	EXm4
add x10, x8, x9			IF	ID	EX	ME	WB					
div x13, x11, x12				IF	ID	EXd	EXd	EXd	EXd	EXd	EXd	EXd
sub x16, x14, x15					IF	ID	~~EX^~~	EX	ME	WB		
and x19, x17, x17						IF	~~ID~~	ID	EX	ME	WB	
or x21, x20, x21							~~IF~~	IF	ID	EX	ME	WB
and x24, x22, x23									IF	ID	EX	ME
(Fuera de código)										IF		

Datos RAW:	0 (+4)
Datos WAW:	0
Control:	0
Estructural:	
Etapa ocupada:	1
Memoria ocupada:	0
Acceso a Mem. Ppal.:	0
Ciclos de bloqueo:	1

Figura 23. Cronograma que muestra bloqueos de datos que no detienen el pipeline y un bloque de datos enmascarado por un bloqueo estructural. A la derecha aparece la tabla resumen del simulador con los bloqueos producidos.

El simulador muestra una tabla resumen de los bloqueos que se producen pulsando en el botón  . Como se aprecia en la parte derecha de la *figura 23*, esta tabla muestra los distintos tipos de bloqueos que se han producido durante la ejecución, distinguiendo 2 tipos de bloqueos estructurales: bloqueos por etapa ocupada y por memoria ocupada. El caso aquí estudiado corresponde a bloqueos estructurales por etapa ocupada, siendo los bloqueos estructurales por memoria ocupada objeto de estudio en el tema de jerarquía de memoria.

La tabla resumen de los bloqueos distingue también los bloqueos que detienen y no detienen la cadena. Los que no detienen la cadena aparecen entre paréntesis y, como es evidente, no deben ser contabilizados en el cálculo teórico del tiempo de CPU ni del CPI.

En el ejemplo de la figura anterior, el cálculo teórico del tiempo de ejecución es:

$$t = Ciclos_{instrucciones} + Ciclos_{bloqueos} = 8 \text{ ciclos de las instrucciones} + 1 \text{ ciclo del bloqueo estructural} = 9 \text{ ciclos.}$$

Puede comprobarse el cálculo midiendo el tiempo de ejecución directamente sobre el cronograma. Como se indicó en una práctica anterior, se contabiliza hasta el IF (válido) de la primera instrucción que no pertenece al programa menos 1. En el caso de la *figura 23*, la etapa IF de la primera instrucción fuera del código está en el ciclo 10, que restando 1, se obtiene los 9 ciclos calculados teóricamente.

5. OPTIMIZACIONES DEL CÓDIGO

5.1. REORDENACIÓN DE INSTRUCCIONES

Una forma de eliminar los riesgos por dependencia de datos RAW consiste en separar durante la ejecución las instrucciones dependientes hasta que el riesgo desaparezca. Como se ha visto en clase, la forma más inmediata de conseguirlo es bloquear la instrucción dependiente (la que entra en ejecución más tarde). La *figura 24* muestra un ejemplo de ejecución sin *forwarding* en el que la instrucción sub se detiene mientras add continúa avanzando durante 2 ciclos para alcanzar la etapa *WB* que hace desaparecer el riesgo por dependencia.

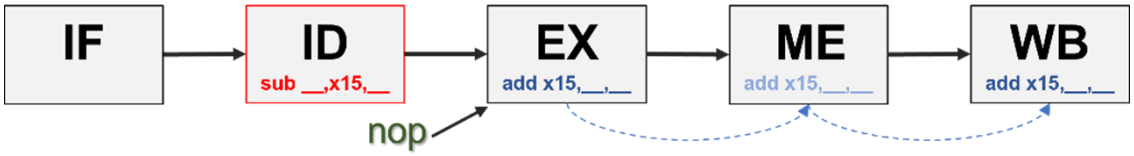


Figura 24. Ejecución de dos instrucciones en un procesador sin forwarding. La instrucción sub permanece bloqueada hasta que add alcance WB.

Los bloqueos de datos impiden la ejecución de otras instrucciones lo que provoca que el CPI aumente y por tanto que el rendimiento disminuya. Estos bloqueos provocan la inyección de una instrucción *nop* por cada ciclo de bloqueo en la etapa siguiente a la bloqueada (etapa *EX* en la *figura 24*). Las *nop* inyectadas están ocupando recursos del procesador y no realizan ninguna operación útil desde el punto de vista del programador.

	1	2	3	4	5	6
add x15, x16, x17	IF	ID	EX	ME	WB	
sub x5, x15, x7		IF	~~ID~~	~~ID~~	ID	EX
slli x21, x21, 2			~~IF~~	~~IF~~	IF	ID
addi x25, x25, 4						IF

a) Sin instrucciones entre *add* y *sub*. Provoca 2 ciclos de bloqueo

	1	2	3	4	5	6
add x15, x16, x17	IF	ID	EX	ME	WB	
slli x21, x21, 2		IF	ID	EX	ME	WB
sub x5, x15, x7			IF	~~ID~~	ID	EX
addi x25, x25, 4				~~IF~~	IF	ID

b) *sll* ha sido introducida entre *add* y *sub*. Reduce los ciclos de bloqueo a 1.

	1	2	3	4	5	6
add x15, x16, x17	IF	ID	EX	ME	WB	
slli x21, x21, 2		IF	ID	EX	ME	WB
addi x25, x25, 4			IF	ID	EX	ME
sub x5, x15, x7				IF	ID	EX

c) Dos instrucciones entre *add* y *sub*. Elimina todos los ciclos de bloqueo.

Figura 25. Ejemplo de reducción de ciclos de bloqueo entre add *y* sub *en un procesador sin forwarding mediante la inserción de otras instrucciones del programa (reordenación de instrucciones).*

Una forma de reducir este impacto negativo en el rendimiento consiste en separarlas instrucciones que sufren riesgos por dependencia con otras instrucciones del programa. Al separarlas con estas otras instrucciones, se reducirá o incluso se eliminarán los ciclos de bloqueos antes producidos ya que las instrucciones *nop* inyectadas han sido sustituidas por otras que sí realizan operaciones útiles para el programa (véase el ejemplo de la *figura 25*).

Como es lógico, al reordenar las instrucciones la semántica del programa no debe cambiar (no debe alterarse el comportamiento del programa), para que esto no ocurra es muy importante que la instrucción que va a moverse no tenga dependencia de datos de ningún tipo con las instrucciones que va "encontrando a su paso". Véase el ejemplo de la *figura 26*, el cual se basa en el código de la *figura 25* pero en el que *slli* y *addi* poseen ahora dependencias RAW y WAW respectivamente con el registro *x5* de la instrucción *sub*.

add x15,x16,x17
sub x5,x15,x7
*slli x21,**x5**,2*
*addi **x5**,x25,4*

⇒

add x15,x16,x17
~~*slli x21,**x5**,2*~~
~~*addi **x5**,x25,4*~~
sub x5,x15,x7

Figura 26. Ejemplo de una incorrecta reordenación al no tenerse en cuenta las dependencias.

Esta limitación es extensible también a las dependencias de control, esto es, la instrucción que va a ser movida no debe "atravesar" a priori una instrucción de salto (ni etiquetas que sean destino de saltos) ya que la ejecución o no de la instrucción que ha sido movida dejaría de estar condicionada por el salto. Por tanto, los saltos y etiquetas van a actuar como "barreras" que delimitan bloques de instrucciones, de forma que las instrucciones de un determinado bloque sólo podrán moverse dentro del bloque al que pertenece (*figura 27*).

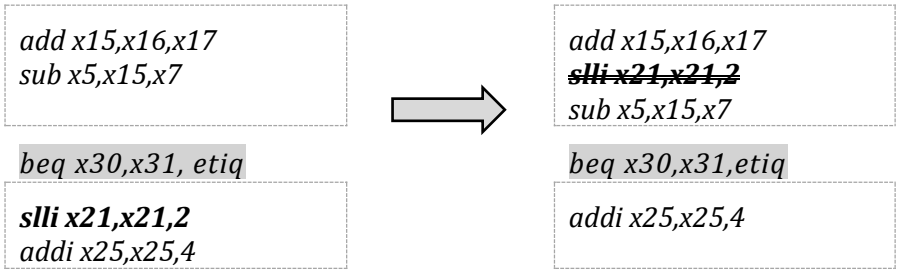
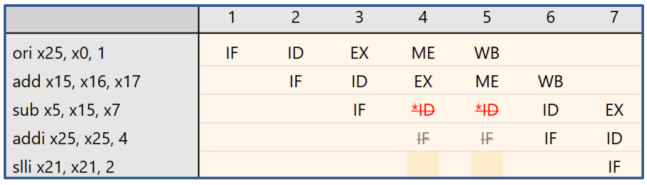


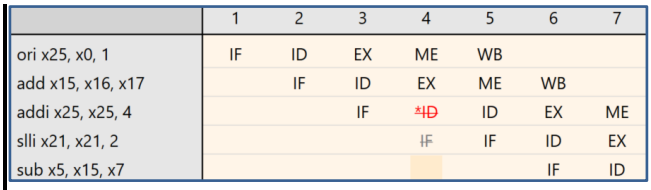
Figura 27. Ejemplo de una incorrecta reordenación al moverse más allá de una instrucción de salto

Además, debe tenerse en cuenta que la reordenación de instrucciones puede provocar nuevos bloqueos pues, al separar instrucciones dependientes puede que se aproximen a otras dependientes que antes no se bloqueaban (véase el ejemplo de *figura 28*).



	1	2	3	4	5	6	7
ori x25, x0, 1	IF	ID	EX	ME	WB		
add x15, x16, x17		IF	ID	EX	ME	WB	
sub x5, x15, x7			IF	~~*ID~~	~~*ID~~	ID	EX
addi x25, x25, 4				~~IF~~	~~IF~~	IF	ID
slli x21, x21, 2							IF

Sin instrucciones entre *add* y *sub*. Provoca 2 ciclos de bloqueo (similar al caso de la figura 25a)

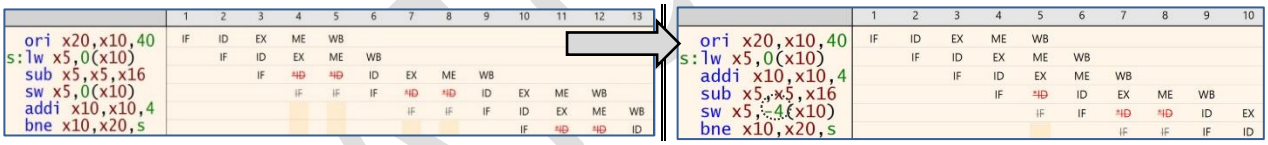


	1	2	3	4	5	6	7
ori x25, x0, 1	IF	ID	EX	ME	WB		
add x15, x16, x17		IF	ID	EX	ME	WB	
addi x25, x25, 4			IF	~~*ID~~	ID	EX	ME
slli x21, x21, 2				~~IF~~	IF	ID	EX
sub x5, x15, x7						IF	ID

Al introducir dos instrucciones entre *add* y *sub* se eliminan los bloqueos de *sub* pero aparece uno nuevo al aproximar *addi* a *ori* (que se soluciona intercambiando *addi* con *slli*).

Figura 28. Ejemplo en un procesador sin forwarding de cómo la reordenación puede provocar que aparezcan nuevos bloqueos. Eliminar los bloqueos entre sub y add provoca un bloqueo entre addi y ori.

Por último, hay situaciones en las que es posible reordenar instrucciones, aunque tengan dependencias. El caso más habitual se encuentra en las instrucciones de carga y almacenamiento cuando las dependencias se producen con los registros dedicados al cálculo de la dirección. Puesto que en estas instrucciones la dirección de memoria se especifica mediante el modo de direccionamiento registro base más desplazamiento, ese desplazamiento puede modificarse para que la reordenación no altere la semántica del programa.



	1	2	3	4	5	6	7	8	9	10	11	12	13
ori x20,x10,40	IF	ID	EX	ME	WB								
s:lw x5,0(x10)		IF	ID	EX	ME	WB							
sub x5,x5,x16			IF	~~*ID~~	~~*ID~~	ID	EX	ME	WB				
sw x5,0(x10)				~~IF~~	~~IF~~	~~IF~~	~~*ID~~	~~*ID~~	ID	EX	ME	WB	
addi x10,x10,4							~~IF~~	~~IF~~	~~IF~~	ID	EX	ME	WB
bne x10,x20,s										~~IF~~	~~*ID~~	~~*ID~~	ID

	1	2	3	4	5	6	7	8	9	10
ori x20,x10,40	IF	ID	EX	ME	WB					
s:lw x5,0(x10)		IF	ID	EX	ME	WB				
addi x10,x10,4			IF	ID	EX	ME	WB			
sub x5,x5,x16				~~IF~~	~~*ID~~	ID	EX	ME	WB	
sw x5,-4(x10)					~~IF~~	~~IF~~	~~*ID~~	~~*ID~~	ID	EX
bne x10,x20,s							~~IF~~	~~IF~~	~~IF~~	ID

Figura 29. Ejemplo de reordenación de una instrucción con dependencia

5.2. DESENROLLADO DE BUCLES

Como se ha indicado en el apartado anterior, la reordenación de instrucciones sólo puede realizarse dentro del bloque de instrucciones delimitado entre los saltos y etiquetas. Cuando estos bloques contienen muy pocas instrucciones la reordenación suele resultar poco eficiente pues el número de bloqueos que puede eliminar es bajo.

En el caso de los bucles, existe una técnica denominada ***desenrollado de bucles*** destinada a aumentar el número de instrucciones independientes en el cuerpo de un bucle. Este aumento de instrucciones permite que la técnica de reordenación de instrucciones pueda reducir aún más ciclos de bloqueo de datos. El desenrollado de bucles posee además otras ventajas que se especificarán más adelante.

El procedimiento para desenrollar un bucle consiste básicamente en aumentar el número de instrucciones del bloque a base de duplicar el cuerpo del bucle cierto número de veces y ajustar las instrucciones para que la semántica del programa no cambie.

La *figura 30* muestra un ejemplo básico de un bucle desenrollado una vez en lenguaje de alto nivel. En este ejemplo, la instrucción *for* reajusta el número de iteraciones incrementando la variable i en 2 y el índice de los vectores de la instrucción duplicada han sido modificados para que realicen la iteración i+1 del bucle.

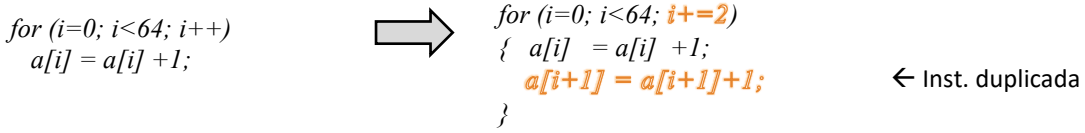


Figura 30. Ejemplo de desenrollado de bucle en lenguaje de alto nivel.

Para obtener el bucle desenrollado en lenguaje ensamblador, se podría traducir el programa desenrollado de alto nivel (como el de la figura anterior) a ensamblador, o bien desenrollar directamente el bucle en ensamblador. El procedimiento para desenrollar un bucle directamente en ensamblador se divide en una serie de pasos que se describen a continuación con ayuda del ejemplo de la *figura 31*.

1.- Identificar las instrucciones útiles y de *overhead* del bucle. Las instrucciones de *overhead* son las destinadas al control del bucle, por ejemplo, las encargadas de actualizar los registros que controlan el número de iteraciones o las que actualizan las posiciones de memoria a las que acceder en cada iteración (véase la *figura 31-a*).

2.- Duplicar las instrucciones útiles. El bloque de instrucciones útiles debe ser duplicado tantas veces como se vaya a desenrollar. En el ejemplo de la *figura 31-b* aparece las instrucciones útiles duplicadas una sola vez.

3.- Ajustar las instrucciones útiles duplicadas. Cada bloque de instrucciones duplicado deberá realizar el cálculo equivalente al que se realiza en iteraciones posteriores en la versión sin desenrollar. Por lo general sólo hay que:

- Modificar los desplazamientos de las instrucciones de lectura y escritura en memoria para acceder a la posición de memoria correcta. En el ejemplo de la *figura 30*, los accesos al vector en la instrucción duplicada han cambiado su índice a i+1. En el ejemplo de la *figura 31-b*, las instrucciones *lw* y *sw* del bloque de instrucciones duplicadas incrementan su desplazamiento en 4 (pues los datos almacenados en memoria son de 4 bytes) para acceder al dato siguiente al que especifica el registro *x20*.

- Renombrar los registros utilizados exclusivamente dentro de cada bloque de instrucciones útiles por otros no usados con el objeto de eliminar las dependencias *WAR*. Este paso no es necesario en el desenrollado de bucles pero es fundamental para reordenar posteriormente las instrucciones adecuadamente. En la *figura 31-c*, el registro *x10* de las instrucciones útiles duplicadas ha sido renombrado a *x11* antes de reordenar. Si no hubiese sido renombrado, la reordenación no hubiese sido posible ya que el segundo *lw* habría "machacado" el dato leído por el primer *lw* antes de que *addi* pudiera utilizarlo.

4.- Ajustar las instrucciones de *overhead*. Las instrucciones de *overhead* se encargan de actualizar los registros para adecuarse a la ejecución de la siguiente iteración. Puesto que cada iteración de la versión desenrollada equivale a varias iteraciones del código no desenrollado, los registros tendrán que actualizarse con valores diferentes. En el ejemplo de la *figura 31-b*, en la instrucción *addi x5,x5,2* el registro x5 (que se corresponde con la variable i del código de la figura 7) es incrementado en 2 en vez de en 1 como ocurría en la versión desenrollada (*figura 31-a*) pues se encarga de controlar la iteración. Por otro lado, la instrucción *addi x20,x20,8* que actualiza el registro *x20* (encargado de mantener la dirección de memoria a la que se accede) es incrementado en 8 en vez de en 4 pues en cada iteración se opera con 2 valores de memoria cada uno de ellos de 4 bytes.

5.- Reordenar las instrucciones. La reordenación de las instrucciones útiles es muy sencilla ya que, al poseer cada bloque instrucciones independientes de las que hay en otros bloques, pueden intercalarse como aparece en la *figura 31-c*. Las instrucciones de *overhead* pueden reordenarse también para reducir los bloqueos que no hayan sido completamente resueltos con instrucciones útiles.

```
        addi x31,x0,x4
        addi x5,x0,0
s:      lw   x10,0(x20)   ┐ Bloque de
        addi x10,x10,1    ┘ Inst. Útiles
        sw   x10,0(x20)   ┐
        addi x5,x5,1      ┘ Inst. de
        addi x20,x20,4      Overhead
        bne  x5,x31, s
```

```
        addi x31,x0,64
        addi x5,x0,0
s:      lw   x10,0(x20)
        addi x10,x10,1
        sw   x10,0(x20)

        lw   x10,4(x20)   ┐ Bloque de
        addi x10,x10,1    ┘ Instr. útiles
        sw   x10,4(x20)     duplicadas

        addi x5,x5,2
        addi x20,x20,8
        bne  x5,x31, s
```

```
        addi x31,x0,64
        addi x5,x0,0
s:      lw   x10,0(x20)
        lw   x11,4(x20)

        addi x10,x10,1
        addi x11,x11,1

        sw   x10,0(x20)
        sw   x11,4(x20)

        addi x5,x5,2
        addi x20,x20,8
        bne  x5,x31, s
```

a) Código sin desenrollar de la figura 7 en ensamblador. | *b) Código desenrollado en ensamblador.* | *c) Código desenrollado y reordenado*

Figura 31. Ejemplo del proceso de desenrollado de bucles en ensamblador para un RISC-V con forwarding. En el caso de que fuese un procesador RISC-V sin forwarding, el código de la figura 31-c podría reordenar también las instrucciones de overhead para eliminar más bloqueos de datos.

El desenrollado de bucles permite reducir el tiempo de ejecución (aumentar el rendimiento) pues consigue:

1.- Reducir los bloqueos de datos al posibilitar una reordenación de instrucciones más efectiva.
2.- Reducir los bloqueos de control ya que el número de saltos es inferior (menos iteraciones).
3.- Reducir el número de instrucciones ejecutadas pues se procesan menos instrucciones de *overhead*.

5.3. SALTOS RETARDADOS

Como se ha visto en clases de teoría, los saltos retardados es una técnica orientada a reducir los bloqueos de control. Para llevar a cabo esta técnica es necesario que el procesador y el compilador (o el programador si fuese el caso) actúen conjuntamente. El procesador ejecutará siempre, se tome o no el salto, la instrucción posterior a la instrucción de salto en el código fuente por lo que no cancela su ejecución. Será tarea del compilador mover de forma adecuada una instrucción del código justo después de la instrucción de salto (hueco del salto) teniendo en cuenta que dicha instrucción nunca va a ser cancelada. Mediante esta técnica, en vez de provocar un ciclo de bloqueo de control en los saltos tomados por la cancelación de la instrucción posterior al salto, el procesador ejecutará dicha instrucción posterior al salto y que corresponderá a la instrucción útil que el compilador haya ubicado en el hueco del salto.

Al igual que ocurría en la reordenación de instrucciones vista en el apartado anterior, para mover una instrucción al hueco del salto habrá que tener en cuenta las dependencias RAW, WAR y WAW con los registros del resto de instrucciones para no alterar el comportamiento del programa. En la elección de la instrucción a mover es preferible que la instrucción proceda del bloque de instrucciones anterior a la instrucción de salto pues las instrucciones de dicho bloque siempre van a ejecutarse. En caso de que no fuese posible optar por una anterior a la instrucción de salto, podrá seleccionarse una instrucción procedente de una de las ramas del salto (de la rama tomada o de la no tomada) siempre que su ejecución sea inocua para el comportamiento del programa en caso de que se ejecutara la rama contraria (véase la transparencia sobre salto retardado del tema del procesador segmentado/paralelismo a nivel de instrucciones).

En un procesador RISC-V con saltos retardados todos los saltos son retardados por lo que el hueco del salto debe ser rellenado siempre para asegurar que la ejecución del programa sea correcta. En caso de no encontrar una instrucción adecuada con la que rellenar el hueco del salto, podrá utilizarse una instrucción nop como última alternativa. Esta opción debe utilizarse como último recurso pues la ejecución de estas instrucciones nop actúan en realidad como un bloqueo (sino que introducido vía software) tanto para la rama tomada como no tomada. De hecho, en el cálculo del rendimiento, tales instrucciones nop no deberían contabilizar como instrucciones del programa.

```
1    ori x20,x10,40
2  s: lw x5,0(x10)
3    addi x10,x10,4
4    sub x5,x5,x16
5    sw x5,-4(x10)
6    bne x10,x20, s
```

*a) Código para un procesador **sin** salto retardado*

```
   ori x20,x10,40
s: lw x5,0(x10)
   addi x10,x10,4
   sub x5,x5,x16
   bne x10,x20, s
   sw x5,-4(x10)
```

*b) Código para un procesador **con** salto retardado*

Figura 32. Ejemplo de salto retardado

Los cronogramas de la *figura 33* muestran ejemplos de ejecución de los códigos de la figura 7 en su versión sin y con salto retardado. Según el código de la *figura 32-a)*, la instrucción 1 no puede utilizarse para rellenar el hueco de salto por encontrarse antes de la etiqueta 's' entre otros motivos, la instrucción 2 tampoco por su dependencia RAW y WAW con la instrucción 4, la instrucción 3 por su dependencia RAW con la instrucción de salto, la instrucción 4 por su dependencia RAW con la 5, en cambio sw sí puede moverse al hueco del salto por ser independiente de la instrucción de salto.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ori x20, x10, 40	IF	ID	EX	ME	WB										
lw x5, 0(x10)		IF	ID	EX	ME	WB									
addi x10, x10, 4			IF	ID	EX	ME	WB								
sub x5, x5, x16				IF	*ID	ID	EX	ME	WB						
sw x5, -4(x10)					IF	IF	*ID	*ID	ID	EX	ME	WB			
bne x10, x20, -16							IF	IF	IF	ID	EX	ME	WB		
(Sin decodificar)										IF					
lw x5, 0(x10)											IF	ID	EX	ME	WB

Figura 33-a. Cronograma del código de la figura 32-a), suponiendo un procesador sin salto retardado

	1	2	3	4	5	6	7	8	9	10	11	12	13
ori x20, x10, 40	IF	ID	EX	ME	WB								
lw x5, 0(x10)		IF	ID	EX	ME	WB							
addi x10, x10, 4			IF	ID	EX	ME	WB						
sub x5, x5, x16				IF	*ID	ID	EX	ME	WB				
bne x10, x20, -12					IF	IF	ID	EX	ME	WB			
sw x5, -4(x10)							IF	*ID	ID	EX	ME	WB	
lw x5, 0(x10)								IF	IF	ID	EX	ME	WB

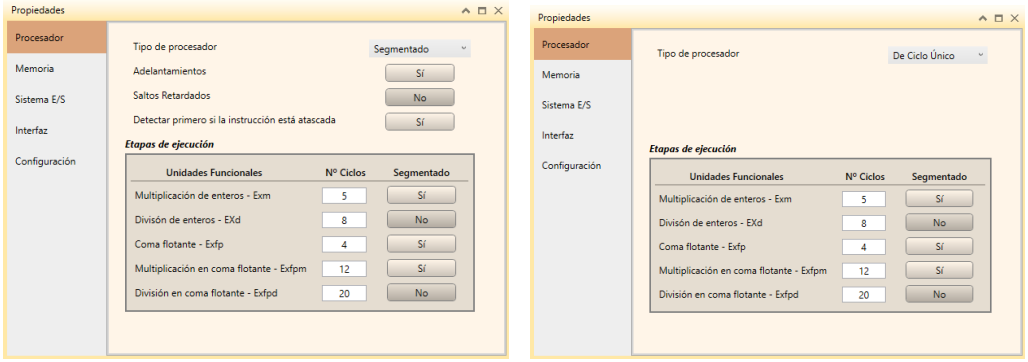
Figura 33-b. Cronograma del código de la figura 32-b) suponiendo un procesador con salto retardado

Con el código de la *figura 32-b)* para un procesador con saltos retardados se consigue eliminar los 9 bloqueos de control (el salto se toma 9 veces) mediante los saltos retardados y 10 ciclos de bloqueo de datos (uno por iteración) entre addi y sw ya que ambos se separan un ciclo al rellenar el hueco del salto con sw (véase el cronograma de la *figura 33-b*).

PARTE 2: EJERCICIOS DE LABORATORIO

***IMPORTANTE:** Los primeros ejercicios tienen como principal cometido familiarizarnos con el entorno y el lenguaje ensamblador, por ello no se ejecutarán los programas como procesador segmentado, sino como procesador secuencial de Ciclo Único. Para ello, se seleccionará la opción correspondiente en las propiedades del simulador accediendo a la herramienta "Propiedades" (o pulsando "F12"), y cambiaremos el tipo de procesador a "De Ciclo Único".*

Observe la siguiente figura con el antes y el después:



Cambio de tipo de procesador. (izquierda) Antes; (derecha) Después

Ejercicio 0 (guiado). El siguiente programa guarda en memoria el resultado de la resta de dos valores (si el resultado es positivo) o cero (si el resultado es negativo). Los dos operandos también provendrán de memoria.

Posible código en C:

```
op1 = 15;
op2 = 9;
temp = op1-op2;
if(temp>=0)
   res = temp;
else
   res = 0;
…
```

```
.data                   # Segmento de datos (origen y destino de las operaciones)
     op1:  .word 15     # Operando 1 etiquetado como "op1", de tipo word y con valor "15"
     op2:  .word 9      # Operando 2 etiquetado como "op2", de tipo word y con valor "9"
     res: .space 4      # Espacio reservado para el resultado, etiquetado como "res" y
                        # con un tamaño de 4 bytes (ya que el resultado será word).

.text                   # Segmento de código
     la x2, op1         # Cargamos la dirección de memoria donde se encuentra "op1" (no
                        #   su valor), ya que no la sabemos.
                        # La pseudo-instrucción "la" (load address) nos resuelve este
                        #   problema. ¡Cuidado! Son 2 instrucciones en una.
     la x3, op2         # Ídem para "op2"
     la x10, res        # Debemos hacer lo mismo con aquellas posiciones de memoria que
                        #   utilicemos para volcar los resultados.
     lw x4, 0 (x2)      # Ahora sí cargamos el valor de "op1" (contenido de la dirección
                        #   donde se encuentra "op1" en memoria)
     lw x5, 0 (x3)      # Ídem con el valor de "op2"
     sub x6, x4, x5     # Realiza la resta y el resultado lo guarda en el registro x6
     slt x7, x6, x0     # compara el resultado de la resta con el contenido del registro
                        #   x0 (que siempre es '0'). Si el contenido de x6 es menor a '0'
                        #   (o lo que es lo mismo, si es negativo), se asignará el valor
                        #   '1' al registro x7; en caso contrario se le asignará un '0'.
     bne x7, x0, tag    # Si el contenido de x7 no es cero (en cuyo caso el resultado de
                        #    la comparación anterior es que [x6]<0, es decir, que es
                        #    negativo), saltamos a la posición etiquetada con "tag".
                        # En caso contrario (positivo), el programa seguirá su curso
                        # normal (siguiente instrucción).
     sw x6, 0 (x10)     # Esta instrucción solo se ejecutará en caso de que no salte la
                        # instrucción anterior, y significará que el resultado de la
                        # resta no es negativo.
                        # Primera rama del condicional. Se guarda el resultado de la
                        # resta en la dirección donde se encontrada la variable "res".
     jal x0, fin        # Salto incondicional al fin del programa. Es necesario porque,
                        # en caso contrario, se ejecutaría la siguiente instrucción que
                        # es la segunda rama del condicional (y no queremos eso).
tag: sw x0, 0 (x10)     # Comienzo de la segunda rama del condicional. Se guarda el valor
                        # '0' en la dirección donde se encontrada la variable "res".
fin:                    # Etiqueta de fin de programa
```

Ejercicio 1: Pruebe el siguiente programa e intente descubrir qué famosa "operación" realiza. Pruebe con distintos valores para la constante almacenada en la dirección *n*. ¿Por qué hay que restar 2 al registro x10?

```
.data
n:      .half 20
.text
        la x10, n
        lhu x10,0(x10)
        addi  x10,x10,-2
        addi x5,x0,0
        addi x6,x0,1
rep:    add x7,x5,x6
        add x5,x0,x6
        add x6,x0,x7
        addi x10,x10,-1
        bnez x10,rep
```

Ejercicio 2: Analice el siguiente programa:

```
.config
      proc single
.data 0x20000
 x:   .word 0xAABBCCDD,-1,0x10,0x20,0x30
 y:   .word 1,1,2,3,4
 z:   .space 20
.text
      addi x2,x0,5
      la x11, x         #Dir. x
      addi x12,x11,20   #Dir. y
      addi x13,x11,40   #Dir. z
loop: lw x5,0(x11)
      lw x6,0(x12)
      add x7,x5,x6
      sw x7,0(x13)
      addi x11,x11,4
      addi x12,x12,4
      addi x13,x13,4
      addi x2,x2,-1
      bne x2,x0,loop
```

a) ¿Qué valor almacenará exactamente x11, x12 y x13 la primera vez que es asignada en el programa?
b) ¿Cuántas iteraciones ejecuta el bucle?
c) ¿Por qué se incrementa x11, x12 y x13 en 4?
d) Indique el contenido de la memoria a partir de la dirección etiquetada con z (20 posiciones) teniendo en cuenta que RISC-V accede a memoria en orden *Little-Endian*.

Ejercicio 3: Analice el siguiente programa:

```
.config
      proc single
.data 0x20000
 x:   .half 12,34,5,21,57,76,54,32,10,5
 y:   .half 11,22,13, 9,55,66,77,88
 res: .space 2
.text
      add x2,x0,x0
      addi x3,x0,100
      la x11, x
loop: lhu x5,0(x11)
      lhu x6,20(x11)
      add x2,x2,x5
      add x2,x2,x6
      addi x11,x11,2
      blt x2,x3,loop
      la x11, res
      sh x2, 0(x11)
```

a) ¿Cuántas iteraciones ejecuta el bucle?
b) ¿Qué valor es almacenado en memoria y a partir de qué posición de memoria?
c) Añada 2 instrucciones al programa para que sólo sume a *x2* el elemento del vector *x* si éste es impar (utilice la instrucción *andi* para calcular si es par. Resultado: *[x2]*=193)

Ejercicio 4: realice un programa en ensamblador que calcule el número de bits a '1' que hay en un número de 32 bits almacenado en memoria.

Ejercicio 5: Partiendo del siguiente programa en C++

```
char v[10] = {12,23,34,45,56,67,78,89,90,100}; // Las variables char son de 8 bits en C-A2
for (int i=0;i<9;i++)
     v[i] = v[i+1];
```

a) Tradúzcalo a lenguaje ensamblador:
b) Modifíquelo suponiendo que el vector *v* está definido de tipo *unsigned int* (variable de 32 bits sin signo)

Ejercicio 6: Realice un programa que guarde en memoria el sumatorio de los números naturales que se encuentra en el intervalo indicado en memoria.

```
.config
         proc single
.data
intervalo:.byte 9, 16 #Sumar los valores del intervalo
                      #      [9, 16]-> Resultado 100
res:      .space 2    #Espacio para "res" de 2 bytes.
.text
          la x2, intervalo # Carga la dir. de memoria
                           #    de "intervalo".
          la x6, res       # Carga la dir de memoria de
                           #    "res" para el resultado
```

Posible código en C:

```
sum = 0;
inicio = intervalo[0];
fin = intervalo[1];
while(inicio<=fin)
{
   sum = sum + inicio;
   inicio++;
}
res = sum;
```

Ejercicio 7: Realice un programa que, dado un vector de 8 enteros (de 4 bytes cada uno) almacenado en memoria, guarde en "res" la media aritmética de los valores positivos del vector.

```
.config
       proc single
.data
 x:    .word 6, -4, 13, 8, -2, 11, 7, 20
 res:  .space 4
.text
       la x2,x      # Dir inicial del vector (1er elemento)
       la x10,res   # Dir de "res" para el resultado final
       …
```

Posible código en C

```
int i;
aux = 0;
positivos = 0;
for(i=0; i<8; i++)
{
   if(x[i] >= 0)
   {
      aux = aux + x[i];
      positivos++;
   }
}
aux = aux/positivos;
res = aux;
```

Ejercicio 8: Traduzca a ensamblador el siguiente bucle SAXPY (para enteros) en C:

```
short x[8] = {2, 4, 6, 8, 10, 12, 14, 16}; // short: son de 16 bits en C-A2
short y[8] = {64, 32, 16, 8 , 4, 2, 1, 0};
short a = 5; // 'a' se encuentra en memoria.
int i;
for (i=0; i<8; i++)
   y[i] = a*x[i]+y[i];
```

Continúe el programa:

```
.config
      proc single
.data
 x:   .half 2,4,6,8,10,12,14,16
 y:    .half 64,32,16,8,4,2,1,0
 a:    .half 5
.text
```

Ejercicio 9: Realice una programa que guarde de memoria la distancia entre dos puntos del plano P1 (x_1, y_1) y P2 (x_2, y_2) almacenados en memoria (denominada distancia euclídea), sabiendo que, para ello, se debe hacer lo siguiente: $d_E(P1,P2)^2 = (x_2-x_1)^2 + (y_2-y_1)^2$.

```
.config
      proc single
.data
 P1:   .word 4, 5    # (x1,y1)
 P2:   .word 6, 9    # (x2,y2)
 res:  .space 4
.text
      la x1, P1      # Dirección donde comienza P1
       …
```

Posible código en C (paso a paso)

```
x1 = P1[0];
x2 = P2[0];
aux1 = x2-x1;
aux1 = aux1*aux1;
y1 = P1[1];
y2 = P2[1];
aux2 = y2-y1;
aux2 = aux2*aux2;
res = aux1 + aux2;
```

Ejercicio 10: Realice un programa que, dado un vector de 8 enteros (de 4 bytes cada uno) almacenado en memoria, y un valor correspondiente a la media aritmética de dichos elementos, guarde en "res" la desviación típica del vector (valor medio de las diferencias con respecto a la media).

```
.config
     proc single
.data
 x:      .word 6, 4, 13, 8, 3, 11, 7, 20
 med:    .word 9
 res:    .space 4
.text
       la x2,x     # Dir.inicial del vector (1er elemento)
       la x10,res  # Dir de "res" para el resultado final
       la x20,med  # Dir de "med" para la media
       …
```

Posible código en C

```
int i;
aux = 0;
for(i=0; i<8; i++)
{
   if(med > x[i])
      aux = aux + (med - x[i]);
   else
      aux = aux + (x[i] - med);
}
aux = aux/8;
res = aux;
```

Ejercicio 11: Traduzca a ensamblador el siguiente bucle en C:

```
char x[8] = {1, 2, 4, 6, 8, 10, 12, 14}; // char es de 8 bits en C-A2
char  y[8] = {8, 8, 4, 4 , 2, 2, 1, 1};
char  a = 3; // 'a' se encuentra en memoria.
int i;
for (i=0; i<8; i++)
   y[i] = a*y[i]- x[i];
```

Continúe con el programa:

```
.config
       proc single
.data
 x:   .byte 1,2,4,6,8,10,12,14
 y:   .byte 8,8,4,4,2,2,1,1
 a:   .byte 3
.text
      la x5,x         # Dir vector x
      la x6,y         # Dir vector y
      addi x3,x0,8    # x3 -> 8
      la x4,a
      lh x4,0(x4)     # x4 -> a
```

Ejercicio 12: Dado el siguiente código en ensamblador, analice qué hace, cuántas iteraciones ejecuta y qué se almacena en la memoria. Nota: en primer lugar, intente analizarlo en papel y, una vez completado su análisis, verifique sus resultados con los obtenidos por el simulador.

```
.data
  res:     .space 4
  v1:      .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
  v2:      .space 20
.text
          addi x30,x0,0
          addi x31,x0,30
          la x10,v1          # Carga la dirección de memoria de la variable "v1".
          addi x1,x10,-4     # Carga la dirección de memoria de la variable "res".
          addi x20,x10,40    # Carga la dirección de memoria de la variable "v2".
  sigue:  lw x11, 0(x10)
          lw x12, 4(x10)
          add x13,x11,x12
          sw x13,0(x20)
          add x30,x30,x13
          addi x10,x10,8
          addi x20,x20,4
          blt x30,x31,sigue
          sw x30,0(x1)
```

Ejercicio 13: Realice un programa que, dado un vector de 10 enteros (de 2 bytes cada uno) almacenado en memoria, guarde en "res" la media aritmética de los valores del vector.

```
.data
  x:    .half 6, -4, 15, 9, -2, 11, 8, 20, 5, 2
  res:  .space 2     # El resultado debería dar 70
.text
       la x2, x      # Dir.inicial del vector (1erelemento)
       la x10, res   # Dirección de "res" (donde almacenar
                     # el resultado final)
        …
```

Posible código en C

```
aux = 0;
for(i=0; i<10; i++)
{
   aux = aux + x[i];
}
aux = aux/10;
res = aux;
```

Ejercicio 14: Realice un programa que, dado un vector de 16 enteros (de 4 bytes cada uno) almacenado en memoria, guarde en "contP" el número de positivos y en "contN" el número de negativos.

```
.data
 x:     .word  6,-4,13,8,-2,11,7,20,9,-3,-12,5,1,23,-25,8
 contP: .space 4   # Deberá contener el resultado final de 11
 contN: .space 4   # Deberá contener el resultado final de 5
.text
       la x2, x       # Dir. inicial del vector (1er elemento)
       addi x3, x0, 0 # Inicialización del contador de positivos
       addi x4, x0, 0 # Inicialización del contador de negativos
       …
```

Posible código en C

```
auxP = 0;
auxN = 0;
for(i=0; i<16; i++)
{
   if(x[i]<0)
      auxN++;
else
     auxP++;
}
contP = auxP;
contN = auxN;
```

Ejercicio 15: Partiendo del siguiente código:

```
.eqv       sum     x5
.eqv       dirx    x6
.eqv       dfin    x7
.eqv       Kmul    x28
.eqv       Kdiv    x29
```

```
.eqv       res     x30
.eqv       op      x31
.eqv       cond    x10
.config
         proc single
         forwarding off
         exm 5 pipelined
         exd 5 pipelined
.data 0x00020000
 x:         .byte 25,30,40,45
 y:         .space 4
 resultado: .space 1

.text
            addi Kdiv,x0,14      # Divisor
            ori  Kmul,x0,2       # Multiplicador
            addi dirx,x0,0x200
            slli dirx,dirx,8     # Puntero al vector x (0x00020000)
            addi dfin,dirx,4     # Puntero al vector 4 (0x00020004)
            addi sum,x0,0        # Sumatorio
sigue:
            lb   op,0(dirx)      # Leer el valor de x
            mul  op,op,Kmul      # Multiplicar x[i]*Multiplicador
            sb   op,4(dirx)      # Guarda el resultado en y[i]
            add  sum,sum,op      # Actualizar el sumatorio
            addi Kmul,Kmul,1     # Incrementa el multiplicador
            addi dirx,dirx,1     # Siguiente posición del vector
            slt  cond,dirx,dfin  # x10=1 si dirx<dirfin
            beq  cond,x0,end
            jal  x1, sigue
end:
            div  res,sum,Kdiv    # Dividir el sumatorio entre el divisor
            sb   res,4(dirx)     # Guardar el resultado
```

Intente realizar los siguientes apartados **sin el simulador**, suponiendo que las instrucciones de larga duración (mul y div) tienen 5 ciclos de su etapa EX (EXm para mul y EXd para div)

a) Realice una traza de programa anterior (en papel) y rellene la siguiente tabla:

Número de iteraciones	
Nº instrucciones de Multiplicación	
Nº Instrucciones de División	
Nº Instrucciones ALU (excluyendo las anteriores)	
Nº instrucciones de carga (load)	
Nº instrucciones de almacenamiento (store)	
Nº instrucciones de salto	
Nº de instrucciones ejecutadas en total	

b) Calcule de forma teórica el **tiempo de ejecución en ciclos** del programa anterior considerando los siguientes casos:

- **Caso I: Procesador secuencial de ciclo único:**
- **Caso II: Procesador secuencial de ciclo múltiple:**

  Teniendo en cuenta el uso de cada etapa:

ALU	→ IF, ID, EX, WB
$ALU_{larga\_duración}$	→ IF, ID, EX0, EX1, EX2, EX3, EX4, WB
Load	→ IF, ID, EX, ME, WB
Store	→ IF, ID, EX, ME
Salto (Cond e Incond)	→ IF, ID

- **Caso III: Procesador segmentado** (suponiendo un procesador ideal: sin bloqueos)

c) Teniendo en cuenta los resultados obtenidos en el apartado anterior, calcule la mejora (aceleración) de prestaciones que se obtiene en el procesador segmentado respecto a las versiones secuenciales considerando que:

- El procesador secuencial de ciclo único tiene un periodo de reloj de τ = 2ns
- En el procesador secuencial de ciclo múltiple, la duración de las etapas no está perfectamente equilibrada por lo que tiene un periodo de reloj de τ = 200ps
- Debido al impacto de la segmentación, el periodo de reloj del procesador segmentado aumenta un 10% respecto del procesador de ciclo múltiple.

Ejercicio 16: Realice los siguientes apartados **con el simulador** utilizando el código del ejercicio anterior.

a) **Estudio práctico del procesador de ciclo múltiple secuencial**: cargue el programa en el editor y mantenga el simulador configurado como procesador secuencial de ciclo múltiple (directiva ***proc multi*** en la sección *.config*). Posteriormente ejecute el programa hasta el final y acceda a la ventana que muestra el cronograma de ejecución (Ctrl+G).

- A partir del cronograma generado, determine el número de ciclos de ejecución del programa.
- Examinando nuevamente el cronograma, ¿Coincide el tiempo de ejecución en ciclos obtenido en el apartado anterior respecto al tiempo calculado teóricamente en el ejercicio 15 (caso II)?

b) **Estudio práctico del procesador de ciclo segmentado:** configure el simulador para un procesador segmentado básico (directiva ***proc pipelined*** y ***forwarding off*** en la sección *.config*) y ejecute nuevamente el programa hasta el final.

Observe el cronograma generado y estime el número de ciclos que dura la ejecución según éste. Tenga en cuenta el método descrito en las páginas 12 y 13 de este boletín.

Ejercicio 17: Realizaremos un estudio sobre el camino de datos del procesador segmentado usando el código del Ejercicio 15.

Como habrá podido observar, dicho código posee instrucciones de todo tipo salvo alguna excepción (instrucciones *jalr*), lo que va a permitir integrar en nuestro estudio todos los componentes del diseño. En este caso, trabajaremos fundamentalmente con 3 ventanas: el editor pues contiene nuestro código, el cronograma para seleccionar las instrucciones objeto de estudio y la ventana ruta de datos. Para poder trabajar con estas ventanas más fácilmente, se recomienda utilizar la combinación Ctrl+5 o bien Menú → Vistas → Ruta de Datos para adecuar) que presenta una distribución de las ventanas más adecuada para este estudio.

Puesto que vamos a estudiar la ruta de datos del procesador segmentado (sin desvíos), debemos configurar el simulador convenientemente con la directiva ***proc pipelined*** **y** ***forwarding off*** en la sección *.config*. Posteriormente seleccione el modo "Instrucción" en la ventana "Ruta de datos" y ejecute el programa hasta el final. A partir de este momento, puede seleccionar cualquier instrucción en la ventana cronograma para ver su camino de datos completo en la ventana de la ruta de datos. Dedique el tiempo necesario para analizar e interpretar el comportamiento del circuito para cada uno de los tipos de instrucciones que contiene el programa.

A continuación, resuelva los siguientes apartados:

a) En el diseño se distingue una serie de multiplexores. Estos multiplexores permiten unificar la ruta de datos que cada instrucción precisa para su ejecución, reencaminando así los datos hacia las unidades funcionales que precisa su ejecución. Indique para los multiplexores que se especifican a continuación, cuál es su función y a qué tipo de instrucciones involucran:

Multiplexor	Descripción
AluSrc	
AluOut	
StoreSrc	
Jmp/Br	
Jal/R	

b) Sitúese sobre la instrucción de salto condicional con salto tomado (ésta se produce en la última iteración) y analice detenidamente la ruta de datos que activa y los componentes que emplea tanto para calcular la dirección de salto como evaluar la condición de salto. Indique a continuación para qué se utiliza cada componente y qué diferencias observa respecto al diseño estudiado hasta ahora en clase.

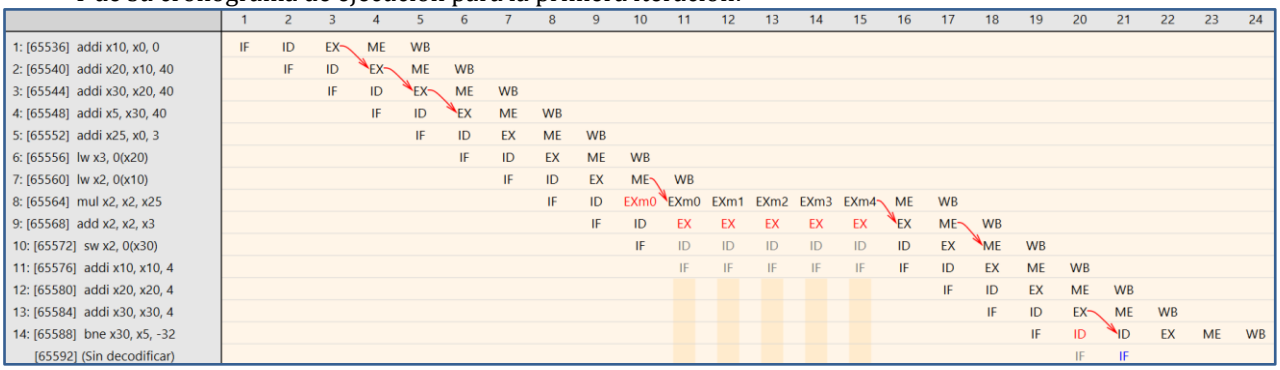
c) Fíjese a continuación, en la ejecución de las instrucciones de multiplicación y división respecto al resto de instrucciones aritméticas. ¿En qué se diferencia su ejecución? ¿Qué problema potencial considera que podría producirse teniendo en cuenta que las instrucciones de multiplicación y división requieren varios ciclos en su ALU correspondiente?

Ejercicio 18: Partiendo de este código para un procesador tipo RISC con bypasses activos:

```
        addi x10, x0, 0      # dir. inicial de "x"
        addi x20, x10, 40    # dir. inicial de "y"
        addi x30, x20, 40    # dir. inicial de "z"
        addi x5, x30, 40     # final de "z"
        addi x25, x0, 3
for:
        lw x3, 0(x20)
        lw x2, 0(x10)
        mul x2, x2, x25
        add x2, x2, x3
```

```
sw x2, 0(x30)
addi x10, x10, 4
addi x20, x20, 4
addi x30, x30, 4
bne x30, x5, for
```

Y de su cronograma de ejecución para la primera iteración:



a) Intente razonar qué sucede con la etapa EXm0 en el ciclo 10. Detalle lo que está sucediendo en ese punto.

b) Aplique las técnicas de optimización que estime oportunas para reducir al máximo los bloqueos. Muestre el código resultante a continuación.

c) Calcule la aceleración de la versión reordenada respecto a la original. Muestre los cálculos realizados.

Ejercicio 19: SIN HACER USO DEL SIMULADOR, represente la primera iteración del cuerpo del cuerpo del bucle siguiente suponiendo un procesador RISC-V **Segmentado** con **bypasses** en el que la unidad funcional de la multiplicación está segmentada y requiere **3 ciclos.** Dado el siguiente código:

Código ensamblador	**Código C**

`addi x10, x0, 0x100  # dir. inicial de "x"`
`addi x20, x0, 0x140  # dir. inicial de "y"`
`addi x30, x20, 40  # final de "y"`
`addi x25, x0, 3`
`for:`
`lw x2, 0(x20)`
`mul x2, x2, x25`
`addi x2, x2, 7`
`sw x2, 0(x10)`
`addi x10, x10, 4`
`addi x20, x20, 4`
`bne x20, x30, for`
	`//"int" equivale a una palabra`
	`int i, x[10], y[10]={…};`
	`for(i=0; i<=9; i++)`
	`{`
	`x[i] = (y[i]*3)+7;`
	`}`

Ejercicio 20: HACIENDO USO DEL SIMULADOR, realice los siguientes apartados suponiendo un procesador RISC-V **Segmentado** con **bypasses** en el que la unidad funcional de la multiplicación está segmentada y requiere **2 ciclos,** y la unidad de división también segmentada y de **3 ciclos.** Dado el siguiente código:

```
.config
    proc pipelined
    forwarding on
    delayed off
    exm 2 pipelined
    exd 3 pipelined

.data
 x:     .half 1,2,3,4,5,6,7,8,9,10
 y:     .half 1,2,3,4,5,6,7,8,9,10
 z:     .space 20
 a:     .half 5
```

```
.text
    la x10, x
    la x20, y
    la x30, z
    la x31,a
    lh x5,0(x31)    # a
for:
    lh x11, 0(x10)
    mul x11,x11,x5
    addi x10, x10, 2
    lh x21, 0(x20)
    div x21,x21,x5
    addi x20, x20, 2
    add x2,x11,x21
    sh x2,0(x30)
    addi x30, x30, 2
    bne x30, x31, for
```

a) Copie el código ensamblador en el simulador y ejecute la simulación para obtener la siguiente información de su ejecución:

	NÚMERO DE CICLOS TOTALES
Ciclos de Bloqueos de datos	

Ciclos de Bloqueos estructurales	
Ciclos de Bloqueos de control	
Nº de Instrucciones	

b) Calcule el CPI del código teniendo en cuenta el número de instrucciones y los ciclos de bloqueos totales proporcionados por el simulador (tenga especial cuidado con los ciclos de bloqueo indicados entre paréntesis, ya que hacen alusión a bloqueos de una rama concreta que no bloquean la cadena):

Ejercicio 21: Partiendo del código del ejercicio anterior, realizaremos algunas modificaciones y razonaremos acerca de los resultados obtenidos.

a) Desactive los bypasses (modifique la directiva "forwarding" a "off") y calcule nuevamente el CPI. Acto seguido, calcule la aceleración del sistema al utilizar bypasses.

b) Volviendo a activar los bypasses, realice las optimizaciones que estime oportunas sobre el código para intentar reducir el número de bloqueos al mínimo posible. Escriba a el código resultante y rellene la tabla.

	NÚMERO DE CICLOS TOTALES
Ciclos de Bloqueos de datos	
Ciclos de Bloqueos estructurales	
Ciclos de Bloqueos de control	
Nº de Instrucciones	

CPI:
Aceleración respecto al Ejercicio 20:

c) Intercambie las posiciones de las instrucciones "mul x11,x11,x5" y "addi x10, x10, 2". De igual forma, intercambie las instrucciones "sh x2,0(x30)" y "addi x30, x30, 2", modificando el valor inmediato de la instrucción de almacenamiento ("0") por "-2". Ejecute nuevamente el programa, compruebe que el resultado es el mismo y calcule el CPI obtenido en esta ocasión. ¿Qué ha sucedido para obtener la mejora?

Ejercicio 22: Con ayuda del simulador VisualRISCV con adelantamientos activos (bypass), desarrolle un fragmento de código en ensamblador que produzca:

a) Bloqueo RAW de 3 ciclos.

b) Bloqueo WAW de 3 ciclos.

Ejercicio 23: SIN HACER USO DEL SIMULADOR, realice los siguientes apartados suponiendo un procesador RISC-V **Segmentado** con bypasses activos**.** Dado el siguiente código:

```
        addi x10,x0,0
        addi x20,x10,40
        addi x30,x10,80
rep:    lw x21,0(x20)
        lw x11,0(x10)
        sub x11,x11,x21
        add x3,x3,x11
        addi x10,x10,4
        addi x20,x20,4
        bne x20,x30,rep
        sw x3, 0(x30)
```

a) Realice el cronograma de ejecución de la **primera iteración** del bucle suponiendo un RISC-V con los saltos retardados **inactivos**. Deberá indicar los bloqueos que se produzcan con el carácter '**–**' y representando mediante flechas los bypasses.

b) Suponiendo los saltos retardados activos, reordene el código hasta obtener CPI ideal y represente el cronograma de la primera iteración.

c) Calcule la aceleración obtenida entre las versiones especificadas en los apartados anteriores, indicando todos los cálculos realizados (bloqueos, instrucciones…).

d) ¿Qué cree sucedería si activamos los saltos retardados utilizando el código original?

Ejercicio 24: HACIENDO USO DEL SIMULADOR, realice los siguientes apartados suponiendo un procesador RISC-V **Segmentado** con **bypasses.** Dado el siguiente código:

```
.config
      proc pipelined
      forwarding on
.data
 x:   .word 2 ,4, -6, 8,-10,12,-14, 16,-18, 20, 0
 y:   .word 1,-2,  3,-4,  5,-6,  7, -8, -9,-10
 z:   .space 4
 i:   .space 4
.text
      la x2,x
      la x3,y
      la x4,z
      add x20,x0,x0      # x20: Acumulador
      add x30,x0,x0      # x30: Contador
rep:  lw x12,0(x2)       # x12 <- x[i]
      lw x13,0(x3)       # x13 <- y[i]
      beq x12, x0, fin   # Salto sin fin vector (elemento = 0)
      addi x30,x30,1     # x30++
      slt x5,x12,x0      # si (x12<0) x12 = -x12
      beq x5,x0,posX
      sub x12,x0,x12
posX: slt x5,x13,x0      # si (x13<0) x13 = -x13
      beq x5,x0,posY
      sub x13,x0,x13
posY: sub x12,x12,x13    # x12 = x12 - x13
      add x20,x20,x12    # acc = acc + x12
      addi x2,x2,4       # i++
      addi x3,x3,4
      j rep
fin:  sw x20,0(x4)       # Acumulador (Valor final 55)
      sw x30,4(x4)       # Contador   (Valor final 10)
```

a) Copie el código ensamblador en el simulador y ejecute la simulación para obtener la siguiente información de su ejecución:

	NÚMERO DE CICLOS TOTALES
Ciclos de Bloqueos de datos	
Ciclos de Bloqueos estructurales	
Ciclos de Bloqueos de control	
Nº de Instrucciones	

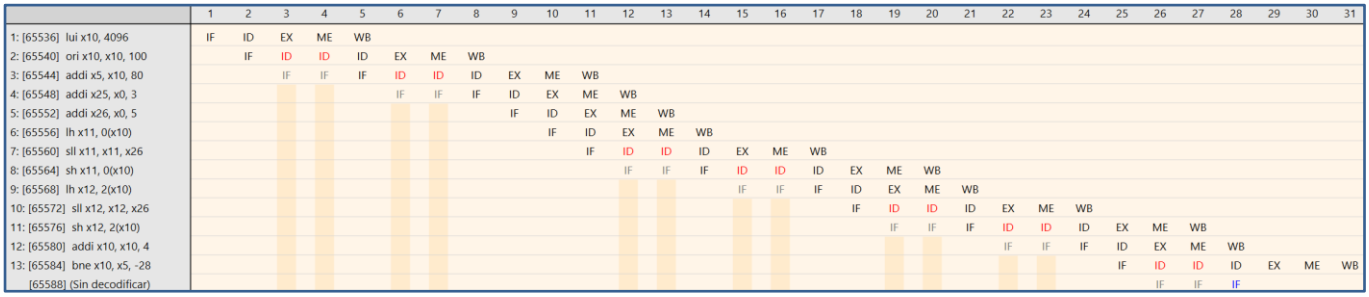
b) Habilite los saltos retardados en el simulador (para ello, pulse el botón "Propiedades" -que tiene como icono un martillo y una llave inglesa cruzadas- o presione la tecla "F12", cambiando la opción "saltos retardados" a "sí") y reordene el código para esta configuración del RISC-V sin hacer uso de la instrucción NOP en el hueco del salto. Para comprobar que el comportamiento no varía, asegúrese de que el valor de los registros x20 y x30 es 55 y 10 (respectivamente) tras la ejecución del código.

c) Calcule su CPI e indique la aceleración con respecto al apartado anterior.

Ejercicio 25: Partiendo de este código para un procesador RISCV sin bypasses activos:

```
      lui x10,0x1000
      ori x10,x10, 100   # dir. inicial de "x"
      addi x5, x10, 80   # final de "x"
      addi x25, x0, 3
      addi x26, x0, 5
for:  lh x11, 0(x10)
      sll x11, x11, x26
      sh x11, 0(x10)
      lh x12, 2(x10)
      sll x12, x12, x26
      sh x12, 2(x10)
      addi x10, x10, 4
      bne x10, x5, for
```

Y de su cronograma de ejecución para la primera iteración:



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
1: [65536] lui x10, 4096	IF	ID	EX	ME	WB																										
2: [65540] ori x10, x10, 100		IF	ID	ID	ID	EX	ME	WB																							
3: [65544] addi x5, x10, 80			IF	IF	IF	ID	ID	ID	EX	ME	WB																				
4: [65548] addi x25, x0, 3						IF	IF	IF	ID	EX	ME	WB																			
5: [65552] addi x26, x0, 5									IF	ID	EX	ME	WB																		
6: [65556] lh x11, 0(x10)										IF	ID	EX	ME	WB																	
7: [65560] sll x11, x11, x26											IF	ID	ID	ID	EX	ME	WB														
8: [65564] sh x11, 0(x10)												IF	IF	IF	ID	ID	ID	EX	ME	WB											
9: [65568] lh x12, 2(x10)															IF	IF	IF	ID	EX	ME	WB										
10: [65572] sll x12, x12, x26																		IF	ID	ID	ID	EX	ME	WB							
11: [65576] sh x12, 2(x10)																			IF	IF	IF	ID	ID	ID	EX	ME	WB				
12: [65580] addi x10, x10, 4																						IF	IF	IF	ID	EX	ME	WB			
13: [65584] bne x10, x5, -28																									IF	ID	ID	ID	EX	ME	WB
[65588] (Sin decodificar)																										IF	IF	IF			

a) Represente su cronograma de ejecución para la primera iteración en el caso de tener los bypasses activos (represente igualmente los bypasses utilizados).

b) Calcule la aceleración de la versión con bypasses respecto a la sin bypasses. Muestre los cálculos realizados.

Ejercicio 26: SIN HACER USO DEL SIMULADOR, realice los siguientes apartados suponiendo un procesador RISC-V **Segmentado** con bypasses activos**.** Dado código del ejercicio anterior:

a) Reordene el código para reducir al máximo los ciclos de bloqueo de datos, manteniendo la semántica del programa. Represente el código reordenado y el cronograma de ejecución de la primera iteración. Además, calcule el CPI de su ejecución completa y la aceleración obtenida respecto al resultado del estudio previo.

b) A continuación, aplique la técnica de salto retardado sobre el código anterior para reducir los ciclos de bloqueo de control. Recuerde que, para aprovechar las ventajas de dicha técnica, tiene que volver a reordenar el código para incluir una instrucción útil en el hueco del salto. Escriba en el siguiente recuadro con qué instrucción rellenaría el hueco del salto (y otras modificaciones si las hubiera), además del CPI resultante y la aceleración respecto al apartado 'a'.

c) ¿Qué cree que sucedería si activamos los saltos retardados utilizando el código del apartado 'a' directamente?

Ejercicio 27: Partimos de un código en ensamblador que convierte un texto almacenado en memoria a mayúsculas haciendo uso de los valores numéricos de la tabla ASCII, y tendremos que aplicar diversas optimizaciones sobre él.

Además, para ayudar a la hora de comprobar la mejoría de las optimizaciones aplicadas, este código incluye una serie de llamadas al sistema (comandos "ecall") que serán explicadas a continuación. Aspectos a tener en cuenta en el código:

1. El programa que convierte a mayúsculas es el contenido entre los comentarios "INICIO DEL CÓDIGO A EVALUAR" y "FIN DEL CÓDIGO A EVALUAR". Por lo tanto, a la hora de aplicar optimizaciones, ÚNICAMENTE se hará sobre el programa contenido entre dichos comentarios, manteniendo el resto del código sin variar.
2. Antes del programa se consulta el número de ciclos y el número de instrucciones que lleva ejecutadas el sistema en ese momento, almacenándolas en x25 y x26, respectivamente.
3. Después del programa se consulta nuevamente el número de ciclos y el número de instrucciones que lleva ejecutadas el sistema en ese momento, y a éstas se les resta las consultadas inicialmente. De esta forma, tenemos el número de ciclos y el número de instrucciones que tarda en ejecutarse nuestro programa.
4. Además, se calcula el CPI por código mediante la división del número de ciclos entre el número de instrucciones. PERO, como el simulador no integra la unidad de punto flotante (no trabaja con números reales), el resultado se multiplica por 100 para obtener un valor en punto fijo con 2 decimales; por ejemplo, CPI 1.0 será representado como 100, CPI 2.32 será representado como 232.
5. El programa muestra por consola tres valores numéricos en este orden: número de ciclos, número de instrucciones, CPI (acceder a la consola a través de "Menú" → "Ventanas" → "Consola").
6. Finalmente, el programa muestra a través de la misma consola la cadena de texto resultante de la ejecución del programa.

¿Para qué tanta cobertura para un programa tan sencillo?

- *Gracias a mostrar la cadena resultante, podrás saber si tus optimizaciones han cambiado la semántica del programa o no. Si el resultado no es la cadena completamente en mayúsculas, el último cambio ha modificado la ejecución del programa y debes descartarlo.*
- *Gracias a mostrar el tiempo de ejecución, podrás conocer inmediatamente si la optimización aplicada ha mejorado o no el rendimiento sin necesidad de realizar los cálculos; de esta forma, se agiliza la resolución del ejercicio.*

```
.config
        proc multi
        forwarding off
        delayed off
.data
```

```
        msg: .asciiz  "En un Lugar de la Mancha, de cuyo Nombre no qUiero acordaRme, no
        ha mucho tiempo que vivIa un hidalgO de los de lAnza en astillerO, adargA
        antigua, rocIn fLaco y Galgo correDor"
.text
        addi x27,x0,100  # Constante para calcular CPI en punto fijo (2 decimales)
        addi x17,x0,31   # Función de obtener ciclos e instrucciones
        ecall            # Llamada al sistema
        add x25,x0,x10   # En x25 tenemos los ciclos utilizados hasta este punto
        add x26,x0,x11   # En x26 tenemos las instrucciones ejecutadas hasta este punto
        ################## INICIO DEL CÓDIGO A EVALUAR
        la x4,msg
        addi x30,x4,0
        addi x20,x0,97
        addi x21,x0,122
rep:    lb x10,0(x4)
        beq x10,x0,fin
        slt x31, x10, x20
        bne x31, x0, sigue
        addi  x10,x10,-32
sigue:  sb x10,0(x4)
        addi x4,x4,1
        jal x0,rep
fin:    add x10,x30,x0
        ################## FIN DEL CÓDIGO A EVALUAR
        addi x17,x0,31    # Función de obtener ciclos e instrucciones
        ecall             # Llamada al sistema
        sub x12,x10,x25   # En X12 tenemos los ciclos transcurridos desde el inicio
        sub x13,x11,x26   # En X13 tenemos las instrucciones ejecutadas desde el inicio
        mul x14,x12,x27   # En X14 tenemos los ciclos multiplicados por 100
        div x14,x14,x13   # En X14 tenemos el CPI en punto fijo (2 decimales)
        addi x17,x0,1     # Función de imprimir número
        add x10,x0,x12    # Imprimimos ciclos
        ecall             # Llamada al sistema
        add x10,x0,x13    # Imprimimos instrucciones
        ecall             # Llamada al sistema
        add x10,x0,x14    # Imprimimos CPI
        ecall             # Llamada al sistema
        la x10,msg        # Seleccionamos la posición de la cadena para imprimirla
        addi x17,x0,4     # Función de imprimir cadena
        ecall             # Llamada al sistema
```

a) Copie el código en el simulador y ejecute la simulación para obtener la siguiente información de su ejecución:

Ciclos de Bloqueos de datos	
Ciclos de Bloqueos de control	
Nº de Ciclos Totales	
Nº de Instrucciones	
CPI	

b) Active la ejecución segmentada mediante la directiva *proc* `pipelined`. Rellene nuevamente una tabla similar con los nuevos datos y calcule la aceleración obtenida respecto al apartado 'a'.

Ciclos de Bloqueos de datos	
Ciclos de Bloqueos de control	
Nº de Ciclos Totales	
Nº de Instrucciones	
CPI	
Aceleración	

c) Active, además de lo anterior, los adelantamientos (bypasses) modificando la directiva *forwarding* a *on*. Rellene nuevamente la tabla con los nuevos datos y calcule la aceleración obtenida respecto al apartado 'b'.

Ciclos de Bloqueos de datos	
Ciclos de Bloqueos de control	
Nº de Ciclos Totales	
Nº de Instrucciones	
CPI	

Aceleración	

d) Lleve a cabo una reordenación del código para reducir los ciclos de bloqueo de datos que pueda. Tras completar la reordenación ejecute el código reordenado y compruebe que la cadena de texto resultante por consola se encuentra, efectivamente, en mayúsculas completamente. Si es el caso, copie a continuación el código reordenado (solo la parte del programa útil, no las llamadas al sistema). Seguidamente, rellene otra vez la tabla con los nuevos datos y calcule la aceleración obtenida respecto al apartado 'c'.

Ciclos de Bloqueos de datos	
Ciclos de Bloqueos de control	
Nº de Ciclos Totales	
Nº de Instrucciones	
CPI	
Aceleración	

e) Finalmente, habilite los saltos retardados en el simulador (modificando la directiva "`delayed`" a "on") y reordene nuevamente rellenando el hueco del salto. Compruebe que su código sigue realizando la misma operación de conversión a mayúsculas. Indique a continuación cómo ha rellenado el hueco del salto. Finalmente, vuelva a rellenar la tabla con los nuevos datos obtenidos en este caso y calcule la aceleración respecto al apartado 'd'. Nota: tenga presente de que, al reordenar con el salto, puede generar nuevos bloqueos RAW (y que, quizás, no puede solventarlos todos).

Ciclos de Bloqueos de datos	
Ciclos de Bloqueos de control	
Nº de Ciclos Totales	
Nº de Instrucciones	
CPI	
Aceleración	

Ejercicio 28: Reordene el siguiente código cuanto sea posible suponiendo un RISC-V con *forwarding* y calcule la aceleración obtenida por la reordenación.

```
.data
 inicio: .word 0x11,0x22,0x33,0x44,0x55,0x66,0x77,0x88,0x99,0xAA
 fin:
.text
       la   x2,fin
       addi x2,x2,-4
       la   x10,inicio
bucle: lw   x1, 0(x2)
       addi x1,x1,-1
       sw   x1,0(x2)
       addi x2,x2,-4
       slt  x8, x2,x10
       beq  x8,x0, bucle
```

Ejercicio 29: Una posible traducción del siguiente bucle SAXPY en C++

```
int X[20] ={3,5,7,9,11,13,15,17,19,21…
int Y[20] ={132,118,104,90,76,62,48,34,20,6…
int A = 7;
for (unsigned int i=0; i < 20;i++)
   X[i]= A*X[i]+Y[i];
```

en lenguaje ensamblador para RISC-V sería:

```
.data
X:     .word 3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39,41
Y:     .word 234,220,206,192,178,164,150,136,122,108,94,80,66,52,38,24,10,-4,-18,-32
A:     .word 7
.text
       la   x2,X
       la   x3,Y
       la   x5,A
       lw   x5,0(x5)
       add  x4,x0,x3    #Para detectar el fin del vector
bucle: lw   x11,0(x3)
       lw   x10,0(x2)
       mul  x10,x10,x5
       add  x10,x10,x11
```

```
        sw   x10,0(x2)
        addi x3,x3,4
        addi x2,x2,4
        bne  x2,x4, bucle
```

Al ejecutar el programa se observa que en memoria todos los elementos del vector X han sido modificados con el valor 0xFF.

a) Suponiendo un **RISC-V con *forwarding***, reordene las instrucciones para reducir los bloqueos cuanto sea posible y calcule el porcentaje de mejora obtenido con la reordenación respecto al código original especificando en el cálculo el número de ciclos de bloqueo y el número de instrucciones. Nota: Puede comprobar el correcto funcionamiento del programa tras la reordenación verificando que todos los elementos del vector X han sido actualizados a 0xFF.

b) A continuación se muestra el bucle del código original desenrollado a 2 iteraciones (esto es, se ha desenrollado una vez)

```
.data
 X:     .word 3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39,41
 Y:     .word 234,220,206,192,178,164,150,136,122,108,94,80,66,52,38,24,10,-4,-18,-32
 A:     .word 7
.text
        la   x2,X
        la   x3,Y
        la   x5,A
        lw   x5,0(x5)
        add  x4,x0,x3    #Para detectar el fin del vector
bucle:  lw  x11,0(x3)
        lw   x10,0(x2)
        mul  x10,x10,x5
        add  x10,x10,x11
        sw   x10,0(x2)

        lw   x13,4(x3)
        lw   x12,4(x2)
        mul  x12,x12,x5
        add  x12,x12,x13
        sw   x12,4(x2)

        addi x2,x2,8
        addi x3,x3,8
        bne  x2,x4, bucle
```

Reordene las instrucciones del bucle desenrollado hasta eliminar todos los bloqueos de datos y vuelva a calcular el porcentaje de mejora respecto al código original de la misma forma que en el apartado anterior. No olvide comprobar el correcto funcionamiento del programa reordenado.

c) Basándose en el código del apartado anterior desenrolle 4 iteraciones (desenrollarlo 2 veces más) y reordénelo nuevamente. Preste especial atención al renombrado de los registros y al reajuste de las instrucciones. Calcule la mejora que se consigue respecto al código obtenido en el apartado anterior (desenrollado 2 de iteraciones y reordenado) ¿A qué se ha debido dicha mejora?

PARTE 3: ANEXOS

ANEXO 1: NOMENCLATURA Y CARACTERÍSTICAS DE LOS REGISTROS DE ENTEROS

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

ANEXO 2: INSTRUCCIONES MÁS UTILIZADAS DEL REPERTORIO DE INSTRUCCIONES RISC-V

(Consulte la lista de instrucciones RISC-V soportadas por el simulador pulsando F1)

	MNEMÓNICO	TIPO	SINTAXIS	DESCRIPCIÓN
	add	Aritmética	add Rd, Rs1, Rs2	Suma de Enteros
	addi	Aritmética	addi Rd, Rs1, Imm12	Suma de Enteros con Inmediato
	div	Aritmética	div Rd, Rs1, Rs2	Cociente de la división entera
	divu	Aritmética	divu Rd, Rs1, Rs2	Cociente de la división entera sin signo
	mul	Aritmética	mul Rd, Rs1, Rs2	Multiplicación con signo. Parte Baja
	mulh	Aritmética	mulh Rd, Rs1, Rs2	Multiplicación con signo. Parte alta
*	neg	Aritmética	neg Rd, Rs1	Negación de Enteros
	rem	Aritmética	rem Rd, Rs1, Rs2	Resto de la división entera
	remu	Aritmética	remu Rd, Rs1, Rs2	Resto de la división entera sin signo
	sub	Aritmética	sub Rd, Rs1, Rs2	Resta de Enteros
	and	Lógica	and Rd, Rs1, Rs2	Operación AND
	andi	Lógica	andi Rd, Rs1, Imm12	Operación AND con inmediato
*	not	Lógica	not Rd, Rs1	Operación NOT (inversión de bit)
	or	Lógica	or Rd, Rs1, Rs2	Operación OR
	ori	Lógica	ori Rd, Rs1, Imm12	Operación OR con inmediato
	sll	Lógica	sll Rd, Rs1, Rs2	Desplazamiento lógico variable a la izquierda
	slli	Lógica	slli Rd, Rs1, Imm5	Desplazamiento lógico a la izquierda con inmediato
	sra	Lógica	sra Rd, Rs1, Rs2	Desplazamiento arimético variable a la derecha
	srai	Lógica	srai Rd, Rs1, Imm5	Desplazamiento aritmético a la derecha con inmediato
	srl	Lógica	srl Rd, Rs1, Rs2	Desplazamiento lógico variable a la derecha
	srli	Lógica	srli Rd, Rs1, Imm5	Desplazamiento lógico a la derecha con inmediato
	xor	Lógica	xor Rd, Rs1, Rs2	Operación XOR
	xori	Lógica	xori Rd, Rs1, Imm12	Operación XOR con inmediato
	lb	Carga	lb Rd, Imm12(Rs1)	Lee de memoria un byte y extiende el signo
	lbu	Carga	lbu Rd, Imm12(Rs1)	Lee de memoria un byte y extiende con ceros
	lh	Carga	lh Rd, Imm12(Rs1)	Lee de memoria media palabra (16 bits) y extiende el signo
	lhu	Carga	lhu Rd, Imm12(Rs1)	Lee de memoria media palabra (16 bits) y extiende con ceros
	lw	Carga	lw Rd, Imm12(Rs1)	Lee de memoria una palabra de 32 bits
	sb	Almacenamiento	sb Rs2, Imm12(Rs1)	Almacena en memoria un Byte
	sh	Almacenamiento	sh Rs2, Imm12(Rs1)	Almacena en memoria media palabra (16 bits)
	sw	Almacenamiento	sw Rs2, Imm12(Rs1)	Almacena en memoria una palabra de 32 bits
	slt	Comparación	slt Rd, Rs1, Rs2	Establece a 1 si menor que
	slti	Comparación	slti Rd, Rs1, Imm12	Establece a 1 si menor que un inmediato

	sltiu	Comparación	sltiu Rd, Rs1, Imm12	Establece a 1 si menor que un inmediato sin signo
	sltu	Comparación	sltu Rd, Rs1, Rs2	Establece a 1 si menor que sin signo
	beq	Salto Cond.	beq Rs1, Rs2, Label	Salta si es igual
*	beqz	Salto Cond.	beqz Reg, Label	Salta si igual a cero
	bge	Salto Cond.	bge Rs1, Rs2, Label	Salta si mayor o igual que
	bgeu	Salto Cond.	bgeu Rs1, Rs2, Label	Salta si mayor o igual que (comparación sin signo)
*	bgez	Salto Cond.	bgez Reg, Label	Salta si es mayor o igual a cero
*	bgt	Salto Cond.	bgt Reg1, Reg2, Label	Salta si es mayor que
*	bgtu	Salto Cond.	bgtu Reg1, Reg2, Label	Salta si es mayor que sin signo
*	bgtz	Salto Cond.	bgtz Reg, Label	Salta si es mayor que cero
*	ble	Salto Cond.	ble Reg1, Reg2, Label	Salta si menor o igual
*	bleu	Salto Cond.	bleu Reg1, Reg2, Label	Salta si menor o igual sin signo
*	blez	Salto Cond.	blez Reg, Label	Salta si es menor o igual a cero
	blt	Salto Cond.	blt Rs1, Rs2, Label	Salta si menor que
	bltu	Salto Cond.	bltu Rs1, Rs2, Label	Salta si menor que (comparación sin signo)
*	bltz	Salto Cond.	bltz Reg, label	Salta si es menor que cero
	bne	Salto Cond.	bne Rs1, Rs2, Label	Salta si no es igual
*	bnez	Salto Cond.	bnez Reg, label	Salta si no es igual a cero
*	j	Salto Incond.	j Label	Salto incondicional relativo al PC sin retorno
	jal	Salto Incond.	jal Rd, Label	Salto incondicional relativo al PC con retorno
	jalr	Salto Incond.	jalr Rd, Imm12(Rs1)	Salto incondicional mediante registro con retorno
*	jr	Salto Incond.	jr Reg	Salto incondicional mediante registro sin retorno
*	ret	Salto Incond.	ret	Retorno de subrutina
*	la	Transferencia	la Rd, Label	Carga la dirección de la etiqueta
*	li	Transferencia	li Rd, Imm32	Carga inmediato
	lui	Transferencia	lui Rd, Imm20	Carga inmediato de 20 bits en la parte superior
*	mv	Transferencia	mv Rd, Rs1	Tranfiere entre registros
*	nop	Otras	nop	Operación nula

* Pseudoinstrucciones (instrucciones sólo definidas en el ensamblador y que se traducen en una o varias instrucciones máquina).

ANEXO 3: DIRECTIVAS MÁS COMUNES DEL COMPILADOR

Directiva	Descripción	Ejemplo
.text <dirección>	Comienzo de una zona de código en el programa. La dirección es opcional y si no se especifica comenzará al principio del segmento de código o a continuación de la última zona de código definida	*.text* *.text 0x004500000*
.data <dirección>	Como el anterior pero para la zona de datos.	*.data* *.data 0x12000000*
.space <nbytes>	Reserva un espacio de <nbytes> bytes en memoria.	*.space 25*
.byte <val1>,<val2>,...	Almacena en memoria una lista de valores de enteros de 8 bits (con o sin signo)	*.byte 20, -128, 255*
.half <val1>,<val2>,...	Igual que el anterior de 16 bits	*.half -32768, 65535*
.word <val1>,<val2>,...	Igual que el anterior de 32 bits	*.word 50, 0xFFFFFFFF*
.ascii <cadena> *.string <cadena>*	Almacena en memoria una cadena de caracteres encerrados entre comillas dobles	*.ascii "Hola mundo"* *.string "Hola mundo"*

.asciiz <cadena> *.stringz <cadena>*	Almacena en memoria una cadena de caracteres siendo el último carácter la marca de fin de cadena (carácter ascii 0)	*.asciiz "Mi mensaje"* *.stringz "Mi mensaje"*