

Objetivos de la sesión

En esta sesión se introduce el simulador pyMIPS64 y se utilizará para analizar la microarquitectura segmentada en 5 etapas de la CPU MIPS64 presentada en la parte teórica de la asignatura. Los principales objetivos de esta sesión son:

- Familiarizarse con el uso del simulador pyMIPS64.
- Introducir el ensamblador empleado por el simulador.
- Comprender el funcionamiento de programas escritos en ensamblador para MIPS64.

Conocimientos y materiales necesarios

Para poder realizar esta sesión, el alumno debe:

- Revisar la teoría correspondiente al tema dedicado a la CPU. Es importante tener claras las ideas teóricas sobre la arquitectura del juego de instrucciones MIPS64.
- Durante la sesión se plantearán una serie de preguntas que deben responderse en el correspondiente [cuestionario](#) en el campus virtual. Puede abrirse el cuestionario en otra pestaña del navegador pinchando en el enlace mientras se mantiene pulsada la tecla `Ctrl`.

Para llevar a cabo la práctica es necesario disponer del simulador pyMIPS64, que actualmente corre sólo en Windows, por lo que es necesaria una máquina con dicho sistema operativo.

1. Introducción al simulador pyMIPS64

Como has visto en las clases teóricas, MIPS64 es una arquitectura RISC de 64 bits con un juego de instrucciones muy simple. En las prácticas de la asignatura vamos a utilizar el simulador pyMIPS64, que permite simular la ejecución de programas escritos en ensamblador para MIPS64, para estudiar el comportamiento del *pipeline* de la CPU estudiada en las clases teóricas.

- Descarga el [simulador pyMIPS64](#) en tu equipo.
- No es necesario instalarlo, basta con descomprimir el archivo `pyMIPS64.zip` descargado. Dentro la carpeta `pymips64` descomprimida encontrarás un archivo `pyMIPS64.exe` con el ejecutable de la aplicación.
- Para facilitar el uso del simulador crea un acceso directo a dicho archivo ejecutable y muévelo al escritorio. De forma opcional, se le puede asociar el icono de la aplicación haciendo click con el botón derecho del ratón sobre el acceso directo, seleccionando **Propiedades** > **Cambiar icono** y eligiendo como icono el que se encuentra en el directorio `pyMIPS64\Icons`.
- Para probar el simulador son necesarios los archivos de esta sesión que debes descargar del enlace [archivos](#) y descomprimir en tu directorio de trabajo.

Con esto, ya estamos en disposición de probar el simulador.

- Ejecuta el simulador utilizando el acceso directo. Durante el arranque automáticamente se comprueba si existe alguna actualización. En ese caso se recomienda aceptar la actualización, para lo cual debes cerrar la ventana actual del simulador y pulsar el botón **Aceptar** en el diálogo.
- Abre el archivo de ejemplo `2-1example.asm` pinchando sobre el icono de la barra de herramientas.
- Realiza una simulación completa pinchando sobre el icono . El aspecto del simulador es el mostrado en la [figura 1](#) tras ajustar el tamaño de los paneles de la forma habitual, pinchando con el botón derecho del ratón sobre la separación entre ellos y arrastrando.

A continuación se describen los elementos de la interfaz gráfica del simulador mostrados en la [figura 1](#).

Figura 1. Interfaz del simulador pyMIPS64

En primer lugar, se observan un área central y tres paneles, aunque dependiendo del archivo a simular y la configuración del simulador pueden aparecer algunos más:

- **Source code.** Muestra la sección de código del programa junto con la dirección a partir de la que se almacena cada instrucción. Las instrucciones se cargan en memoria a partir de la dirección 800h. Debe tenerse en cuenta que las direcciones de memoria son de 64 bits, por lo que requieren 16 dígitos hexadecimales. Para abreviar, la secuencia 0 . . . 0 indica una secuencia de dígitos todos ceros. Durante la simulación se muestra además la etapa que ha completado la instrucción en el ciclo actual. Por ejemplo, al final de la simulación la última instrucción del programa ha terminado la etapa WB.
- **Register changes.** Permite observar el estado de los registros durante la simulación. Puesto que en la mayor parte de nuestros programas sólo se modifican unos pocos registros, resulta conveniente mostrar sólo los registros que han sido escritos, por lo que aquellos que no aparecen permanecen en el estado inicial con todos sus bits a cero. Cuando uno de los registros mostrados aparece en rojo significa que ha sido escrito en el último ciclo de reloj, justo lo que hace la instrucción `and r13, r14, r15` sobre el registro `r13` en el último ciclo de reloj.
- **Simulation statistics.** Proporciona estadísticas sobre la simulación. En la figura se muestran estadísticas de rendimiento (*performance*) y sobre detenciones (*stalls*). El tipo de información mostrada depende del archivo ensamblador simulado y de la configuración del simulador. Por ejemplo, en la figura se indica que se han simulado 8 ciclos, se han retirado (terminado) 4 instrucciones y que por lo tanto el CPI (Ciclos Por Instrucción) es $8/4 = 2$.
- **Área principal.** Situada en la parte superior derecha de la figura, no tiene barra de título ni controles de ventana. Muestra la evolución del *pipeline* durante la ejecución del programa. Tiene tres partes:
 - Instrucciones ejecutadas. Aparecen numeradas en la parte izquierda.
 - Ciclos de reloj. Aparecen numerados en la parte superior.
 - Cronograma de etapas. Aparece en la parte central.

Los paneles se pueden hacer flotantes pinchando sobre su icono, independizándose de la ventana principal. También se pueden arrastrar para situarlos en otra zona de la ventana principal, o incluso solaparlos con otros paneles creando pestañas. La [figura 2](#) muestra un ejemplo.

Figura 2. Ejemplo de redimensionado y reubicación de los paneles

En la parte superior de la interfaz se encuentra una barra de herramientas. Si acercas el puntero del ratón a cualquiera de los iconos de la misma aparece una descripción emergente que incluye el atajo de teclado asociado. A continuación se describen brevemente:

- Permite seleccionar un archivo fuente ensamblador. En el próximo apartado se describe la sintaxis de estos archivos fuente. Observa en la figura anterior cómo el nombre del archivo abierto aparece en la barra de estado en la parte inferior de la interfaz.
- Abre una ventana de diálogo en la cual se puede configurar la microarquitectura a simular. En las clases de teoría se presentan diferentes alternativas que pueden estudiarse con la ayuda del simulador.
- Retrocede un ciclo de reloj en la simulación.
- Avanza un ciclo de reloj en la simulación.
- Retrocede el número de ciclos de reloj necesarios hasta la terminación de una instrucción, es decir, justo hasta que una instrucción completó la etapa WB.
- Avanza el número de ciclos de reloj necesarios hasta la terminación de una instrucción, es decir, justo hasta que una nueva instrucción completa la etapa WB.

- Retrocede la simulación hasta el principio.
 - Avanza la simulación hasta el final.
 - Permite seleccionar los paneles que se muestran y se ocultan. Se pueden ocultar todos los paneles, pero no el área principal que muestra el cronograma del *pipeline*. Los paneles también se pueden ocultar pinchando sobre el aspa en la esquina superior derecha del panel.
 - Muestra los créditos del simulador y permite mostrar/ocultar información adicional en los cronogramas del *pipeline* (si está disponible).
- Con el archivo de ejemplo `2-1example.asm` abierto, prueba la funcionalidad descrita.

2. Ensamblador del simulador pyMIPS64

Los archivos ensamblador empleados por el simulador tienen extensión `.asm` y definen el programa a simular. Las instrucciones soportadas son básicamente el subconjunto de instrucciones MIPS64 empleado en las clases de teoría, mostrado en el [apéndice](#).

Un programa en el ensamblador del simulador pyMIPS64 tiene hasta tres secciones:

- Sección de código. Definida a partir de la directiva `.code`. Esta sección incluye las instrucciones del programa y es opcional. Se recomienda indentar el código de tal forma que se liberen las primeras columnas para definir etiquetas como destinos de las instrucciones de salto. El programa empezará a ejecutarse por la primera instrucción de la sección de código.
- Sección de datos. Definida a partir de la directiva `.data`. Esta sección incluye las definiciones de variables del programa y también es opcional.
- Sección de microarquitectura. Definida a partir de la directiva `.microarch`. Esta sección es opcional y la emplearemos preferentemente en los exámenes. Contiene la definición de la microarquitectura del simulador empleada exclusivamente en la simulación del archivo fuente. Una vez se abre otro archivo fuente esa microarquitectura se pierde y se simula con la microarquitectura original.

Todas aquellas líneas que aparezcan antes de cualquiera de las directivas anteriores se supone pertenecen a la sección de código, por lo que es posible un archivo fuente con sólo instrucciones y sin directivas. Si editas el archivo `2-1example.asm` verás que ese es su caso.

La sección de datos puede definir variables enteras de tamaño 8 bits con la directiva `byte`, enteras de 16 bits con la directiva `hword`, enteras de 32 bits con la directiva `word` y 64 bits con la directiva `dword` para valores enteros, o `double` para valores reales. Para definir *arrays* de enteros o reales los valores se separan por comas. Incluye además una forma abreviada de definir *arrays* añadiendo el sufijo `n` a las directivas anteriores junto con dos valores; el primero indica el número de elementos y el segundo el valor a almacenar. Por ejemplo, `wordn 20, 10h` se corresponde con un *array* de 20 palabras inicializadas a `10h`.

Los valores enteros se expresan por defecto en decimal, aunque también es posible expresarlos en hexadecimal añadiendo el sufijo `h`. Opcionalmente pueden llevar un signo `-` para los enteros negativos. En el caso de los enteros de tipo `byte` que deban almacenar el código ASCII de un carácter es posible definirlos de la forma habitual, con el carácter entre comillas simples. Si se escribe más de un carácter entre las comillas simples se define una cadena de caracteres que no incluye el carácter nulo de terminación.

A continuación se muestra un ejemplo de programa que incluye muchas de las características descritas.

```
1 .data
2 intval:   word 10h                ; Decimal integer 16
3 array:    dword 6,9,-12,92,100,2,-3,1 ; Array of 64-bit integer numbers
4 string1:  byte 'Hello', 0         ; A null terminated string
5 string2:  byte 'H', 'e', 'l', 'l', 'o', 0 ; The same string
6 bytearray: byten 4, 15h          ; The same as byte 15h, 15h, 15h, 15h
```

```

7 floatval: double 0.5 ; IEEE-754 double value
8
9 .code
10 ld    r1, array(r0)    ; r1 = array[0] = 6
11 dadd  r2, r1, r1        ; r2 = r1+r1 = 12
12 l.d   f1, floatval(r0) ; f1 = 0.5
13 mul.d f2, f1, f1        ; f2 = f1*f1 = 0.25

```

- Abre el archivo fuente `2-lassembler.asm` en el simulador. La [figura 3](#) muestra el aspecto del simulador tras cargar el fichero.

Figura 3. Programa con diferentes definiciones de variables

- Observa por un lado el panel `Data memory` con los datos de memoria. Los datos se almacenan en memoria en formato *little endian* a partir de la dirección de memoria 400h.
- Observa por otro lado el panel `Source code`, el cual muestra la posición de las instrucciones en memoria a partir de la dirección 800h.
- La variable `intval` tiene un tamaño de 32 bits al estar definida con la directiva `word`, es decir, 4 bytes. La variable se almacena a partir de la dirección de memoria 400h. Recuerda que estas direcciones son de 64 bits pero para abreviar no representamos los dígitos más significativos a cero. ¿Cuál es la dirección más alta que ocupa en memoria la variable `intval`? Responde en el [cuestionario](#): pregunta 1.
- El primer elemento de la variable `array` comienza en la dirección 408h. ¿Por qué no ocupa la dirección siguiente a la última de la variable `intval`? Responde en el [cuestionario](#): pregunta 2.
- Simula hasta el final del programa. El simulador tendrá un aspecto similar al de la [figura 4](#).

Figura 4. Simulación de programa con sección de datos

3. Configuración de la microarquitectura

En las clases expositivas se presentan diferentes alternativas microarquitectónicas con el objetivo de estudiar la mejora del rendimiento de la CPU. El simulador permite configurar la microarquitectura con la que se simula la ejecución del programa. Por ejemplo, por defecto la unidad de ejecución que se emplea es monociclo y se denomina `EXA`. No obstante, es posible configurar instrucciones para que se ejecuten sobre las unidades multiciclo `EXB` y `EXC`. La unidad de ejecución monociclo, `EXA`, aparece en el cronograma del *pipeline* de la figura anterior.

- Pincha sobre el icono de la barra de herramientas para abrir el diálogo de configuración de la microarquitectura. Aparece una ventana como la mostrada en la [figura 5](#).

Figura 5. Configuración de microarquitectura

La configuración con la mayor parte de las opciones desactivadas se corresponde con la microarquitectura básica, es decir, sin ningún tipo de mejora. Sólo aparecen dos casillas activas, que sirven para que el simulador no genere excepciones y termine la simulación cuando se producen divisiones por cero o accesos inválidos a la memoria de datos.

El botón **Reset to defaults** restituye el estado por defecto de todas las opciones.

- Ahora vamos a cambiar la microarquitectura para que la instrucción `mul.d` se ejecute en una unidad multiciclo no segmentada de 4 ciclos de reloj. Pincha sobre la instrucción `mul.d` dentro del cuadro `Single cycle ins` y arrástrala hasta el cuadro `Non-pipelined multicycle ins`. Cambia además el número de ciclos de la unidad de ejecución multiciclo no segmentada, `EXB`, para que sea 4 ciclos. El diálogo de configuración tendrá el aspecto de la [figura 6](#). Pulsa a continuación el botón **OK** para guardar la nueva microarquitectura.

Figura 6. Instrucción de multiplicación de reales multiciclo

- Simula hasta el final y observa el uso de la unidad multiciclo `EXB` para la realización de la multiplicación de reales.

Cuando el cronograma tiene un tamaño que no cabe en el panel correspondiente, además de las barras de desplazamiento correspondientes, aparece una pequeña barra de herramientas debajo de las instrucciones del cronograma. Aparece resaltada en la [figura 7](#).

Figura 7. Barra de herramientas de navegación del cronograma

Con esta barra se puede navegar de una forma más rápida y cómoda que usando las barras de desplazamiento. Es particularmente útil en el caso de cronogramas muy grandes. Tiene estos controles:

- Mueve un ciclo de reloj adelante con desplazamiento vertical automático.
- Mueve 10 ciclos de reloj adelante con desplazamiento vertical automático.
- Desplazamiento horizontal y vertical hasta el final del cronograma.
- Mueve un ciclo de reloj atrás con desplazamiento vertical automático.
- Mueve 10 ciclos de reloj atrás con desplazamiento vertical automático.
- Desplazamiento horizontal y vertical hasta el inicio del cronograma.

4. Simulación de un programa complejo

En este apartado se trabajará con un programa que calcula el máximo de una lista de números enteros. Puedes ver su código a continuación.

```

1  .microarch
2      out_of_order_retirement    = true
3      forwarding                  = false
4      early_branch_evaluation     = true
5      register_renaming          = false
6      branch_prediction          = always_not_taken
7      exb_no_pipelined_cycles    = 1
8      exb_instructions           = []
9      exc_pipelined_cycles       = 1
10     exc_instructions            = []
11     ignore_div_exceptions       = false
12     ignore_data_mem_exceptions = false
13
14 .data
15     array: dword 6,9,12,92,100,2,3,1 ; array of numbers
16     count: dword 8                  ; item count
17     max:   dword 0                  ; maximum
18
19 .code
20 main:
21     xor     r8, r8, r8              ; temporary maximum (0 is the minimum)

```

```

22    xor    r9, r9, r9      ; memory index of the current item
23    ld     r10, count(r0) ; remaining items
24 loop:
25    ld     r11, array(r9)  ; current item
26    daddi  r10, r10, -1    ; count--
27    daddi  r9, r9, 8       ; move index to next item (64 bits)
28    slt    r2, r8, r11     ; comparison [r2 = r8 < r11 ? 1 : 0]
29    movn   r8, r11, r2     ; update temporary max. [if (r2 != 0) r8 = r11]
30    beqz   r10, end        ; end of the array
31    j loop
32 end:
33    sd     r8, max(r0)     ; maximum

```

Debes tratar de entender su funcionamiento. Recuerda que la directiva `.microarch` define la microarquitectura a emplear, por lo que no es necesario modificarla en el simulador.

- Abre el archivo `2-1max.asm` en el simulador.
- Abre el diálogo de microarquitectura y comprueba que la microarquitectura coincide con la especificada después de la directiva `.microarch`. No la cambies, pues en ese caso los cambios introducidos tendrían prioridad sobre los especificados en el archivo fuente.
- Simula instrucción a instrucción al menos dos iteraciones del bucle trazando la ejecución del programa y comprobando los cambios en los registros.
- Una vez termine el programa, ¿qué posiciones de memoria se escribirán? ¿Con qué valores? Responde en el [cuestionario](#): pregunta 3
- Simula hasta el final y comprueba si tu respuesta fue correcta.
- Prueba los botones de navegación del cronograma.

5. Ejercicios

Como ejercicios propuestos se recomienda llevar los ejemplos de teoría al simulador, escribiendo el archivo ensamblador necesario y configurando la microarquitectura. El simulador puede ser una herramienta muy útil para asimilar los conceptos teóricos.

6. Apéndice. Juego de instrucciones del simulador pyMIPS64

El simulador pyMIPS64 soporta el siguiente subconjunto de instrucciones de la arquitectura MIPS64. Se corresponde con las instrucciones empleadas en la parte teórica de la asignatura.

Tabla 1. Juego de instrucciones de pyMIPS64

Instrucción	Descripción
<code>and rd, rs1, rs2</code>	AND lógico
<code>or rd, rs1, rs2</code>	OR lógico
<code>xor rd, rs1, rs2</code>	XOR lógico
<code>slt rd, rs1, rs2</code>	Comparación entera con signo (menor que)
<code>movz rd, rs1, rs2</code>	Copia condicional de registro (si 0)
<code>movn rd, rs1, rs2</code>	Copia condicional de registro (si no 0)
<code>dadd rd, rs1, rs2</code>	Suma de dobles palabras, generando excepción si hay desbordamiento
<code>dsub rd, rs1, rs2</code>	Resta de dobles palabras, generando excepción si hay desbordamiento
<code>dmul rd, rs1, rs2</code>	Multiplicación de dobles palabras
<code>ddiv rd, rs1, rs2</code>	División de dobles palabras

Tabla 1. Juego de instrucciones de pyMIPS64

Instrucción	Descripción
add.d fd, fs1, fs2	Suma de reales de precisión doble
sub.d fd, fs1, fs2	Resta de reales de precisión doble
mul.d fd, fs1, fs2	Multiplicación de reales de precisión doble
div.d fd, fs1, fs2	División de reales de precisión doble
nop	Instrucción nula
lb rd, imm16(rs)	Carga de byte con extensión de signo
lbu rd, imm16(rs)	Carga de byte sin extensión de signo
lh rd, imm16(rs)	Carga de media palabra con extensión de signo
lhu rd, imm16(rs)	Carga de media palabra sin extensión de signo
lw rd, imm16(rs)	Carga de palabra con extensión de signo
lwu rd, imm16(rs)	Carga de palabra sin extensión de signo
ld rd, imm16(rs)	Carga de doble palabra
l.d rd, imm16(rs)	Carga de real de presión doble
sb rs, imm16(ri)	Almacenamiento de byte
sh rs, imm16(ri)	Almacenamiento de media palabra
sw rs, imm16(ri)	Almacenamiento de palabra
sd rs, imm16(ri)	Almacenamiento de doble palabra
s.d fs, imm16(ri)	Almacenamiento de real de presión doble
beqz rs, label	Salto si igual a 0
bnez rs, label	Salto si distinto de 0
beq rs1, rs2, label	Salto si iguales
bne rs1, rs2, label	Salto si distintos
j label	Salto incondicional

Objetivos de la sesión

En esta sesión se estudia el funcionamiento del *pipeline* en la microarquitectura segmentada en 5 etapas de los procesadores MIPS64. Los principales objetivos de esta sesión son:

- Consolidar los conceptos sobre el *pipeline* y los riesgos que producen roturas en el cauce de ejecución segmentado.
- Conocer algunas de las técnicas que permiten minimizar las roturas del cauce segmentado.

Conocimientos y materiales necesarios

Para poder realizar esta sesión, el alumno debe:

- Acudir al laboratorio con los apuntes de teoría correspondientes al tema dedicado a la CPU. Es importante además tener claras las ideas teóricas sobre el *pipeline*.
- Durante la sesión se plantearán una serie de preguntas que puedes responder en el correspondiente [cuestionario](#) en el campus virtual. Puedes abrir el cuestionario en otra pestaña del navegador pinchando en el enlace mientras mantienes pulsada la tecla `Ctrl`.

1. *Pipeline* MIPS64

- Descarga los [ficheros necesarios para la práctica](#) y descomprímelos en tu carpeta de trabajo.

Como sabes, la microarquitectura segmentada MIPS64 vista en clase organiza el cauce de ejecución en 5 etapas:

1. **IF**: búsqueda del código de instrucción e incremento del contador de programa.
2. **ID**: decodificación de la instrucción, lectura de registros y realización de saltos incondicionales.
3. **EX**: ejecución y cálculo de direcciones efectivas.
4. **MEM**: acceso a memoria y modificación del contador de programa en saltos condicionales.
5. **WB**: escritura diferida.

La ejecución segmentada de instrucciones aumenta la productividad de la CPU. En una situación ideal, el número de instrucciones por segundo que es capaz de ejecutar una CPU segmentada es mayor en un factor equivalente al número de etapas respecto a la versión secuencial (no segmentada) de la misma CPU. No obstante, existen riesgos en la segmentación que impiden que se alcance este máximo ideal. En las siguientes secciones veremos los distintos tipos de riesgos que pueden producirse en un cauce segmentado.

1.1. Funcionamiento básico del *pipeline*

Utilizaremos un pequeño programa para ilustrar el funcionamiento del cauce segmentado. Este programa, cuyo listado se muestra a continuación, realiza dos sumas de cuatro enteros almacenados en memoria. Al final, almacena ambos resultados en sendas posiciones de memoria.

```
1      .data
2
3  var1: word 14
4  var2: word 24
5  var3: word 5
6  var4: word 4
7  res1: word 0
8  res2: word 0
9
10     .code
```



```

11
12 main:
13     ld     r10, var1(r0)    ; load var1 in r10
14     ld     r20, var2(r0)    ; load var2 in r20
15     ld     r11, var3(r0)    ;...
16     ld     r21, var4(r0)
17     nop
18     dadd   r1, r20, r10     ; r1 <- r20 + r10
19     dadd   r2, r21, r11
20     nop
21     sd     r1, res1(r0)     ; store r1 in res1
22     sd     r2, res2(r0)

```

- Comprueba que el fichero `2-2ideal.asm` tiene el contenido arriba mostrado.
- Analiza el código fuente e intenta comprender su funcionamiento. La instrucción `ld r10, var1(r0)` carga en el registro `r10` el valor de `var1`. En este caso el registro `r0` se utiliza como registro base con valor cero. La instrucción `dadd` es la instrucción de suma para cantidades de 64 bits. La instrucción `sd r1, res1(r0)` almacena el contenido de `r1` en la variable `res1`. Observa que el orden del operando destino y fuente cambia en las operaciones de carga y almacenamiento.
- Suponiendo que existe una versión secuencial de la microarquitectura MIPS64, es decir, las 5 etapas en las que se divide el cauce de ejecución se ejecutan secuencialmente y no en paralelo, ¿cuántos ciclos serían necesarios para ejecutar el programa? Responde en el [cuestionario](#): pregunta 1.
- Si esta CPU trabajase a una frecuencia de reloj de 400 MHz, ¿cuánto tiempo invertiría en ejecutar cada instrucción? Responde en el [cuestionario](#): pregunta 2. ¿Cuál sería el tiempo necesario para completar el programa? Responde en el [cuestionario](#): pregunta 3.
- ¿Cuál sería la productividad teórica medida en MIPS que puede alcanzar esta CPU? Responde en el [cuestionario](#): pregunta 4.
- Considerando ahora el cauce segmentado donde las 5 etapas trabajan en paralelo, ¿cuál es la productividad máxima teórica de esta CPU en MIPS? Responde en el [cuestionario](#): pregunta 5.
- Si se supone que ninguna de las instrucciones genera una detención, ¿cuál sería el número de ciclos necesarios para ejecutar el fragmento de código anterior? Responde en el [cuestionario](#): pregunta 6.

Vamos a comprobar las respuestas utilizando el simulador.

- Arranca el simulador.
- Carga en el simulador el fichero `2-2ideal.asm`.
- El simulador ya está listo para comenzar la simulación. Fíjate en la ventana principal y en la ventana de código fuente para ver la evolución de la simulación.
- Como puedes observar, tras la carga del archivo aparece el código fuente. Utiliza el botón para ejecutar el primer ciclo de la primera instrucción. Observa como se indica tanto en la ventana de código como en la ventana principal, que la primera instrucción entra en el cauce en el primer ciclo para ejecutar la etapa `IF`, mostrada en amarillo.
- Pulsa de nuevo el botón para avanzar un nuevo ciclo de reloj. Verás que la instrucción `ld r10, var1(r0)` pasa a la segunda etapa (`ID`), mientras que una nueva instrucción entra en el cauce.
- Pulsa el botón otras tres veces. Deberías estar simulando el quinto ciclo. Durante el quinto ciclo todas las etapas del cauce segmentado están trabajando. Se ha superado un transitorio inicial desde que el cauce de ejecución estaba completamente vacío hasta que está completamente lleno. Este transitorio se produce cuando se reinicia la CPU. Al final del ciclo 5 terminará de ejecutarse la primera instrucción.
- Pulsa una vez más el botón . La primera instrucción ha terminado y se ejecuta en este ciclo la última etapa de la segunda instrucción.

- Continúa la ejecución hasta el final pulsando el botón . Tras la ejecución de la última instrucción el simulador detendrá la ejecución. En ese momento el programa habrá terminado. ¿Cuántos ciclos se han consumido en la ejecución del programa? ¿Coincide con tu respuesta anterior?

Como has podido comprobar, el *pipeline* mejora significativamente el rendimiento. No obstante, hay situaciones en las que esa mejora ideal de rendimiento no se alcanza. En lo que resta de práctica se estudiarán los riesgos de la segmentación que impiden llegar a ese rendimiento máximo ideal.

2. Riesgos de la segmentación

Se conocen como riesgos de la segmentación aquellas situaciones que pueden impedir el funcionamiento en paralelo de las diferentes etapas en las que se divide un cauce de ejecución segmentado. En esos casos, la ejecución en paralelo se interrumpe momentáneamente y se dice que se produce una detención, teniendo que esperar unas etapas por otras, lo que ocasiona que la productividad final de un cauce segmentado se reduzca y sea inferior a la teórica.

En las clases de teoría se han visto algunas situaciones que producen detenciones de la segmentación. Se han ejemplificado sobre un diseño muy simple del cauce segmentado MIPS64 dividido en 5 etapas. Bajo estas condiciones, las detenciones en el cauce segmentado se producían por dos motivos: una dependencia de datos entre una o varias instrucciones dentro del cauce o un cambio en el flujo de ejecución del programa. A continuación se describen los dos escenarios anteriores junto con los riesgos estructurales, que no aparecen en el cauce segmentado simplificado (que solo soporta operaciones con enteros y un subconjunto de instrucciones MIPS64).

2.1. Riesgos de datos

Los riesgos de datos son aquellos que se producen porque una instrucción depende del resultado de otra instrucción que todavía sigue en el cauce. Vamos a verlo con un ejemplo.

```

1
2      .data
3
4  var: word 25
5  res: word 0
6
7      .code
8
9  main:
10     daddi r10, r0, 5    ; r10 <- 5
11     nop
12     ld     r20, var(r0) ; load var in r20
13     dadd   r1, r20, r10 ; r1 <- r20 + r10
14     nop
15     nop
16     sd     r1, res(r0)  ; store r1 in res

```

- Comprueba que el fichero 2-2data.asm tiene el contenido arriba mostrado. Analiza brevemente el código e intenta comprender su funcionamiento. Fíjate que es un programa muy simple, que realiza la suma de un entero con una constante almacenando el resultado en memoria.
- Si no existiesen detenciones en la segmentación, ¿cuántos ciclos de reloj invertiría la CPU en ejecutar el programa? Responde en el [cuestionario](#): pregunta 7.
- Carga el programa en el simulador.
- Asegúrate de que no esté activada la opción de menú **Microarchitecture** > **Forwarding**.
- Pulsa el botón cinco veces, hasta que se muestre la quinta etapa de la primera instrucción.
- Fíjate que en este ciclo (etapa ID de la instrucción dadd r1, r20, r10) se leen los enteros almacenados en los registros r20 y r10. Sin embargo, si bien el registro r10 ya está cargado con

el valor inmediato al final del ciclo 5 (el valor se escribe realmente en la primera mitad del ciclo en la etapa WB y se lee en la segunda mitad en la etapa ID), debe esperarse a que el registro `r20` tenga el valor que le corresponde, que ocurrirá en la etapa WB de la instrucción `ld r20, var(r0)`. Por este motivo, se produce una detención en la segmentación por dependencia de datos, y se refleja en el simulador con un borde negro en la etapa detenida. Pulsa el botón y compruébalo.

- El valor que contendrá el registro `r20` tras el acceso a la memoria se escribe en la etapa WB, por lo que la instrucción de suma debe esperar a que la instrucción de carga llegue a esa etapa. Pulsa el botón dos veces y compruébalo.
- Continúa la ejecución hasta el final pulsando el botón . Comprueba que comprendes lo que sucede en el *pipeline* en cada ciclo. ¿Cuántos ciclos ha invertido la CPU realmente en ejecutar el programa? Responde en el [cuestionario](#): pregunta 8. ¿Cuál es la aceleración obtenida en la ejecución del código anterior respecto a una ejecución ideal sin detenciones en el cauce segmentado? Responde en el [cuestionario](#): pregunta 9. ¿Y respecto a una hipotética CPU MIPS64 secuencial? Responde en el [cuestionario](#): pregunta 10.

El valor que contendrá el registro `r20` tras el acceso a la memoria se conoce tras la finalización de la etapa MEM, pues el valor leído de la memoria ya está disponible. Solo tras la finalización de la etapa de escritura (WB) este valor estará realmente almacenado en el registro destino. Sin embargo, la microarquitectura MIPS64 incorpora una funcionalidad que permite utilizar este valor como operando de una instrucción a pesar de que no esté todavía guardado en el registro destino, con lo que se evita otro ciclo de detención en el cauce. A esta técnica se la conoce como adelantamiento o *forwarding* y se estudiará en la siguiente práctica. Por el momento, supondremos que no se utilizan este tipo de técnicas.

2.2. Riesgos de control

Los riesgos de control se producen en situaciones de cambios en el flujo de ejecución tales como llamadas a función, saltos o excepciones. Bajo estas condiciones, se interrumpe el flujo de ejecución secuencial, lo que impide a la CPU ejecutar las etapas iniciales de las siguientes instrucciones hasta que se resuelva la dirección de destino. Vamos a ilustrarlo con un ejemplo.

```

1
2 .data
3
4 .code
5 main:
6     xor    r10, r10, r10
7     daddi  r20, r0, 12
8     daddi  r21, r0, 3
9     daddi  r22, r0, 2
10    j      dest          ; jump (unconditional)
11    nop
12    xor    r20, r20, r20
13 dest:
14    dsub   r20, r20, r21
    dadd   r22, r22, r21

```

- Comprueba que el fichero `2-2contr-j.asm` tiene el contenido arriba mostrado y analiza su código. Se trata de un programa muy simple que contiene una instrucción de salto incondicional, `j dest`, que rompe el flujo de ejecución secuencial y avanza el contador de programa hasta la instrucción `dsub`.
- Si no existiesen detenciones en la segmentación, ¿cuántos ciclos de reloj invertiría la CPU en ejecutar el programa? Responde en el [cuestionario](#): pregunta 11.
- Vamos a verificarlo con el simulador. Carga el programa `2-2contr-j.asm`.

- Observa la ventana de código. La columna de la izquierda es el desplazamiento desde el comienzo de la sección de código a partir de la que se encuentra cada instrucción. Recuerda que cada instrucción ocupa en memoria 4 posiciones.
- Pulsa el botón de forma repetida hasta que la instrucción `j dest` entre en la primera etapa (`IF`), es decir, hasta el ciclo 5. Al final de este ciclo se acaba de ejecutar la primera etapa de la instrucción, que consiste en la búsqueda en memoria del código de la instrucción.
- Fíjate en la vista del diagrama de ciclos de reloj y consulta el código fuente del programa. Pulsa de nuevo el botón , verás que comienza a ejecutarse la etapa `ID` de la instrucción `j dest`, mientras que la instrucción `nop` pasa a la etapa `IF`. Esto es así porque el *pipeline* asume una ejecución secuencial de instrucciones. Sólo al final de esta etapa (ciclo 6) es cuando la CPU decodifica la instrucción `j dest` y detecta un salto en el flujo de ejecución.

Recuerda que los saltos incondicionales se toman en la etapa `ID`.

- Pulsa de nuevo el botón . La CPU ha identificado la instrucción de salto y además ha modificado el contador de programa con el valor de destino del salto, por lo que se interrumpe la ejecución de la instrucción `nop` (vacía la etapa `IF`) y pasa a ejecutar la primera etapa de la instrucción destino del salto, es decir, `dsub r20, r20, r21`.
- Continúa la ejecución hasta el final pulsando el botón . Comprueba que comprendes lo que sucede en el *pipeline* en cada ciclo. ¿Cuántos ciclos ha invertido la CPU realmente en ejecutar el programa? Responde en el [cuestionario](#): pregunta 12. ¿Cuál es la aceleración obtenida en la ejecución del código anterior respecto a una ejecución ideal sin detenciones en el cauce segmentado? Responde en el [cuestionario](#): pregunta 13. ¿Y respecto a una hipotética CPU MIPS64 secuencial? Responde en el [cuestionario](#): pregunta 14.

Vamos a considerar ahora el caso de los saltos condicionales. El siguiente programa muestra un ejemplo de bucle controlado por un salto condicional.

```

1
2     .code
3
4 main:
5     daddi r8, r0, 2      ; 2 iterations
6     xor   r10, r10, r10
7     nop
8 loop:
9     daddi r8, r8, -1
10    daddi r21, r0, 3
11    daddi r22, r0, 2
12    bnez  r8, loop      ; branch if r8 != 0
    xor   r20, r20, r20

```

- Comprueba que el fichero `2-2contr-bnez.asm` tiene el contenido arriba mostrado y analiza su código. Se trata de un programa que incluye un bucle de tipo *do-while* que itera dos veces. El bucle está controlado por la instrucción `bnez r8, loop`, que realiza un salto hacia atrás en el caso de que el registro `r8` sea distinto de cero.
- Si no existiesen detenciones en la segmentación, ¿cuántos ciclos de reloj invertiría la CPU en ejecutar el programa? Responde en el [cuestionario](#): pregunta 15.
- Vamos a verificarlo con el simulador. Carga el programa `2-2contr-bnez.asm`.
- Pulsa el botón de forma repetida hasta que la instrucción `bnez r8, loop` entre en la primera etapa (`IF`), es decir, hasta el ciclo 7. Al final de este ciclo se acaba de ejecutar la primera etapa de la instrucción, que consiste en la búsqueda en memoria del código de la instrucción.
- Fíjate en la vista del diagrama de ciclos de reloj y consulta el código fuente del programa. Pulsa de nuevo el botón , verás que comienza a ejecutarse la etapa `ID` de la instrucción `bnez r8, loop`, mientras que la instrucción `xor` pasa a la etapa `IF`.

- Pulsa de nuevo el botón tres veces. Como el valor de `r8` es distinto de cero, el salto se toma. Esto se calcula en la etapa `EX` y se efectúa en la etapa `MEM` al utilizar evaluación normal de saltos. Verás que el efecto es mayor que con el salto incondicional.

Vamos a ejecutar ahora el programa anterior utilizando la técnica de evaluación agresiva de saltos.

- Habilita la opción de evaluación agresiva de saltos en el menú **Microarchitecture > Early branch evaluation**
- Vuelve a ejecutar la simulación hasta que la instrucción `bnez r8, loop` entre en la primera etapa (`IF`), es decir, hasta el ciclo 7.
- Pulsa de nuevo el botón, verás que comienza a ejecutarse la etapa `ID` de la instrucción `bnez r8, loop`, mientras que la instrucción `xor` pasa a la etapa `IF`.
- Pulsa de nuevo el botón. Al igual que en la ejecución anterior, el salto se toma, pero en este caso la decisión se toma en la etapa `ID` al utilizar evaluación agresiva de saltos. Verás que el efecto, ahora, es el mismo que con el salto incondicional. De esta forma se evita la penalización que supone tomar la decisión de saltar en etapas tardías del *pipeline*, ya que no es necesario descartar un elevado número de instrucciones.
- Pulsa repetidamente el botón de nuevo hasta que la instrucción de salto condicional vuelva a la etapa `ID`. En este caso, el salto no se toma, pues `r8` vale cero.
- Pulsa dos veces más el botón. Como el simulador no implementa, en este momento, ningún predictor saltos, el *pipeline* se detiene un ciclo en la etapa `IF` de la instrucción `xor` para después continuar ejecutándola.
- Continúa la ejecución hasta el final pulsando el botón. Comprueba que comprendes lo que sucede en el *pipeline* en cada ciclo. ¿Cuántos ciclos ha invertido la CPU realmente en ejecutar el programa? Responde en el [cuestionario](#): pregunta 16. ¿Cuál es la aceleración obtenida en la ejecución del código anterior respecto a una ejecución ideal sin detenciones en el cauce segmentado? Responde en el [cuestionario](#): pregunta 17. ¿Y respecto a una hipotética CPU MIPS64 secuencial? Responde en el [cuestionario](#): pregunta 18.

2.3. Riesgos estructurales

Los riesgos estructurales son aquellos que se producen porque varias de las instrucciones del cauce intentan utilizar un recurso que no está replicado. La implementación del cauce de ejecución MIPS64 simplificada impide que se produzcan detenciones por riesgos estructurales, ya que cada etapa utiliza recursos independientes del resto.

Esto no es así cuando se utilizan unidades funcionales que requieren varios ciclos. Un ejemplo es la ejecución de operaciones aritméticas sobre números en punto flotante sobre la unidad de punto flotante. El simulador nos permite especificar cuántos ciclos necesita la CPU para realizar una operación de punto flotante. En la práctica, definir un número de ciclos mayor que uno para las operaciones de punto flotante equivale a que cada operación de punto flotante requiere de varias etapas, pues habitualmente las unidades de punto flotante también se segmentan.

Se ilustrarán los riesgos estructurales de la segmentación utilizando un programa de ejemplo que trabaja con números en punto flotante, ya que en este caso sí se pueden producir detenciones por este tipo de riesgos.

```

1 .data
2     var1:    double 3.5
3     var2:    double 1.7
4     var3:    double 2.8
5     var4:    double 0.4
6     res1:    double 0
7     res2:    double 0
8
```

```

9  .code
10 main:
11     l.d    f10, var1(r0)
12     l.d    f20, var2(r0)
13     l.d    f11, var3(r0)
14     l.d    f21, var4(r0)
15     nop
16     add.d  f0, f20, f10 ; floating-point addition f0 <- f20 + f10
17     add.d  f1, f21, f11
18     nop
19     nop
20     s.d    f0, res1(r0) ; store f0 in res1
21     s.d    f1, res2(r0)

```

- Confirma que el fichero `2-2struc.asm` tiene el contenido arriba mostrado. Analiza brevemente el código e intenta comprender su funcionamiento.
- Si no existiesen detenciones en la segmentación, y suponiendo que la unidad de suma de números en punto flotante necesita el mismo tiempo para completarse que el resto de etapas, ¿cuántos ciclos de reloj invertiría la CPU en ejecutar el programa? Responde en el [cuestionario](#): pregunta 19.
- Carga el programa en el simulador.
- Configura la latencia de la unidad de suma de punto flotante (el número de etapas en la suma de punto flotante). Para ello, incluye en el menú **Microarchitecture > Non-pipelined multicycle ins** la instrucción `add.d`. Indica que la unidad de punto flotante tarda dos ciclos (EXB cycles con valor 2).

Como puedes ver, el simulador permite utilizar unidades de ejecución multiciclo segmentadas y no segmentadas para las diferentes instrucciones.

- Avanza la ejecución hasta que se muestre la etapa `ID` de la instrucción `add.d f0, f20, f10` en la ventana principal. En este momento la instrucción se ha traído desde la memoria y se ha decodificado, por lo que se va a proceder a realizar la operación de suma utilizando la unidad de suma de punto flotante.
- Pulsa el botón `Next`. Comprobarás que se comienza la ejecución del primero de los dos ciclos de reloj que necesita la unidad de suma de punto flotante, al mismo tiempo que la siguiente operación de suma se decodifica y está lista para pasar a la etapa de suma flotante. Fíjate en la ventana de código fuente, podrás ver sobre qué instrucciones están trabajando cada una de las etapas del cauce segmentado.
- Pulsa el botón `Next`. Se lleva a cabo el segundo ciclo de la etapa de suma de la primera instrucción, mientras que la siguiente instrucción queda detenida en un ciclo de detención estructural.
- Pulsa el botón `Next`. Se termina la etapa de ejecución de la primera instrucción de suma, la segunda instrucción de suma en punto flotante puede avanzar a la primera etapa de la unidad de punto flotante.
- Pulsa una vez más el botón `Next`. Se termina la etapa de ejecución de la segunda instrucción de suma. Además simultáneamente se ejecuta la etapa `EX` de la instrucción `nop` que viene a continuación.
- En el siguiente ciclo tanto la instrucción `nop` como la segunda instrucción `add.d` deberían pasar a la etapa `MEM` pero como esta etapa no está replicada, la instrucción `nop` debe esperar produciéndose otra detención en la segmentación por un riesgo estructural. Pulsa el botón `Next` y compruébalo.
- Continúa la ejecución hasta el final pulsando el botón `Next`. Comprueba que comprendes lo que sucede en el *pipeline* en cada ciclo. ¿Cuántos ciclos ha invertido la CPU realmente en ejecutar el programa? Responde en el [cuestionario](#): pregunta 20. ¿Cuál es la aceleración obtenida en la ejecución del código anterior respecto a una ejecución ideal sin detenciones en el cauce segmentado? Responde en el [cuestionario](#): pregunta 21. ¿Y respecto a una hipotética CPU MIPS64 secuencial? Responde en el [cuestionario](#): pregunta 22.

Archivos de la práctica

En tu carpeta de trabajo debes tener los archivos `2-2ideal.asm`, `2-2data.asm`, `2-2contr-j.asm`, `2-2contr-bnez.asm` y `2-2struc.asm`.

Ejercicios

Habrás observado que en todos los programas de la sesión aparecen instrucciones `nop`. Esto es así para evitar riesgos adicionales que dificultasen la comprensión de cada tipo de riesgo en cada uno de los apartados. Prueba a eliminar las instrucciones `nop` de los programas y a simularlos de nuevo. Observarás que aparecen nuevos riesgos que antes no aparecían al introducir instrucciones *inútiles* en la ejecución del código *útil* de los programas. ¿Coinciden los resultados con los obtenidos anteriormente?

Objetivos de la sesión

En esta sesión se introducen conceptos avanzados de la segmentación encaminados a reducir las detenciones por riesgos de datos y las detenciones por riesgos de control. Todos ellos se ilustran sobre el simulador pyMIPS64.

Conocimientos y materiales necesarios

Para poder realizar esta sesión, el alumno debe:

- Repasar los apuntes de teoría correspondientes al tema dedicado a la CPU. Es importante además tener claras las ideas teóricas sobre el *pipeline*.
- Durante la sesión se plantearán una serie de preguntas que puedes responder en el correspondiente [cuestionario](#) del campus virtual. Puedes abrir el cuestionario en otra pestaña del navegador pinchando en el enlace mientras mantienes pulsada la tecla `Ctrl`.

1. Tipos de detenciones

El *pipeline* de la microarquitectura MIPS64 propuesta puede sufrir diferentes tipos de detenciones. La representación de las mismas en el simulador se muestra entre paréntesis:

- Dependencia de datos RAW (RAW). Una instrucción debe detenerse mientras sus operandos fuente no están disponibles.
- Dependencia de datos WAW (WAW). Dos instrucciones que escriben en el mismo registro deben terminar en orden. Las detenciones de este tipo sólo pueden producirse cuando se dispone de unidades de ejecución multiciclo.
- Estructurales (STR). Dos instrucciones requieren simultáneamente un mismo elemento hardware. Este elemento hardware puede ser una unidad de ejecución multiciclo no segmentada, o la etapa MEM por la que compiten dos instrucciones listas para ser terminadas. Las detenciones de este tipo sólo pueden producirse cuando se dispone de unidades de ejecución multiciclo.
- Control (CNT). Cuando se decodifica una instrucción de salto el *pipeline* debe detenerse hasta que dicho salto es evaluado.
- Terminación en orden (IOT). Cuando las instrucciones deben terminar en orden, tal como exige la gestión de excepciones precisas, puede ser necesario detener el *pipeline* en un punto para garantizar esta terminación en orden. Las detenciones de este tipo sólo pueden producirse cuando se dispone de unidades de ejecución multiciclo.

En el apartado siguiente se practica con técnicas de mejora del rendimiento de la CPU basadas en la reducción o eliminación de las detenciones por riesgos de datos RAW y WAW. El final de la sesión práctica se centra en las técnicas de reducción o eliminación de detenciones por riesgos de control.

2. Reducción de las detenciones por dependencias de datos

De los tres tipos de dependencias de datos: RAW (*Read After Write*), WAR (*Write After Read*) y WAW (*Write After Write*), sólo las dependencias RAW pueden generar detenciones en la versión más simple del *pipeline* visto en clase. Éstas ocurren cuando una instrucción utiliza como operando fuente el operando destino de una instrucción anterior, por lo que no puede ejecutarse hasta que dicho operando destino haya sido escrito. En el caso de la microarquitectura MIPS64 estudiada, esta dependencia sólo es posible con operandos de registro; los operandos de memoria no pueden producir esta dependencia.

A continuación, veremos dos técnicas que afectan a las detenciones producidas por las dependencias de datos RAW: el reordenamiento de instrucciones y las rutas de reenvío. Finalmente, se trabajará con el

renombrado de registros para la reducción, o incluso eliminación, de las detenciones por dependencias de datos WAW.

2.1. Reordenamiento de instrucciones

Una posible forma de reducir el impacto de las detenciones por dependencias RAW es depositando en el compilador la responsabilidad de minimizar el número de dependencias entre instrucciones cercanas, que ocasionarían el mayor impacto. Esto se consigue generando código donde las dependencias sobre registros están separadas varias instrucciones.

Vamos a verlo con el programa `2-3reordering.asm` de ejemplo. Este programa incluye una dependencia de datos de tipo RAW que provoca una detención en el *pipeline*.

```
1 .code
2 main:
3     dadd  r6, r4, r2
4     daddi r1, r2, 5
5     dsub  r8, r3, r2
6     xor   r1, r5, r3
7     ld    r8, 120(r1)
8     and   r8, r4, r1
```

- Descarga los [archivos de la práctica](#).
- Abre el simulador y comprueba que no tiene habilitada ninguna mejora en la microarquitectura.
- ¿Qué instrucciones tienen dependencias de datos RAW? ¿De qué otras instrucciones dependen? ¿Qué registros crean esas dependencias? Responde en el [cuestionario](#): pregunta 1.
- Dibuja sobre el papel o sobre una hoja de cálculo la evolución del cauce durante la ejecución del programa. ¿Cuántos ciclos de reloj requiere para su ejecución? Responde en el [cuestionario](#): pregunta 2.
- Compruébalo con el simulador.
- ¿Cómo reordenarías el código del programa sin modificar su semántica, es decir, que siga haciendo lo mismo, para eliminar la detención por el riesgo RAW con el menor número de instrucciones recolocadas? Responde en el [cuestionario](#): pregunta 3.
- Para comprobar que desaparece la detención modifica el programa `2-3reordering.asm`, guárdalo con el nombre `2-3reordering2.asm` y simula su ejecución en el simulador.

2.2. Rutas de reenvío (*forwarding*)

El principal problema del reordenamiento de instrucciones es que la responsabilidad recae en el compilador. Este debería conocer la microarquitectura subyacente y ser capaz de encontrar instrucciones independientes para introducirlas entre dos instrucciones dependientes, lo cual no es siempre posible.

Una solución más eficaz y transparente al software, ya que se implementa por hardware, es el uso de rutas de reenvío. Vamos a estudiarlas sobre un ejemplo.

El programa `2-3raw.asm`, mostrado a continuación, contiene varias instrucciones con dependencias de datos RAW.

```
1 .data
2 var1: word 34
3
4 .code
5 main:
6     ori   r1, r0, 4
7     daddi r2, r1, 1
8     ori   r3, r1, 7
```

```

9      ld      r4, var1(r1)
10     daddi   r6, r8, 100
11     dadd    r6, r4, r4
12     ori     r5, r4, 8
13     beqz    r5, main

```

Analizaremos la ejecución del programa anterior en dos escenarios: sin rutas de reenvío y con rutas de reenvío activas.

2.2.1. Sin rutas de reenvío

En este caso las detenciones por dependencias de datos RAW incrementan el tiempo de ejecución del programa.

- ¿Qué instrucciones tienen dependencia de datos RAW? ¿De qué instrucciones dependen? ¿Qué registro crea esa dependencia? Responde en el [cuestionario](#): pregunta 4.
- Dibuja sobre el papel o sobre una hoja de cálculo la evolución del cauce durante la ejecución del programa. Debes suponer que la microarquitectura implementa la evaluación *agresiva de saltos*. ¿Cuántos ciclos de reloj requiere para su ejecución? Responde en el [cuestionario](#): pregunta 5.
- ¿Cuántos ciclos de detención ha sufrido el programa? Responde en el [cuestionario](#): pregunta 6.
- La suma del número de instrucciones, más el número de ciclos de detención, más el transitorio inicial de 4 ciclos debe coincidir con el número de ciclos del programa. Haz esta comprobación y recuerda que sólo es aplicable a programas que no emplean instrucciones que requieren unidades de ejecución multiciclo.
- ¿Cuál es el CPI resultante? Responde en el [cuestionario](#): pregunta 7.

Vamos a comprobar las respuestas utilizando el simulador.

- Abre el simulador pyMIPS64 y configura la microarquitectura para que la opción de evaluación agresiva de saltos (`Early branch evaluation`) esté activada y la opción de rutas de reenvío (`Forwarding`) esté desactivada. A continuación carga el programa `2-3raw.asm` y lleva a cabo la simulación.
- A partir de los resultados de la simulación comprueba tus respuestas anteriores. Si alguna no coincide debes repasar la simulación.

2.2.2. Con rutas de reenvío

El empleo de rutas de reenvío permite eliminar, salvo en casos puntuales, las detenciones por dependencias de datos RAW en el *pipeline* MIPS64. Debe recordarse que en la microarquitectura MIPS64 vista en las clases teóricas se plantean tres rutas de reenvío:

- Salida `EX` → Entrada `EX`. Permite llevar el valor del registro destino desde la salida de la etapa `EX` a la entrada de la etapa `EX` en el ciclo de reloj siguiente. Se emplea cuando las dos instrucciones dependientes son consecutivas y la primera no es una instrucción de carga.
- Salida `MEM` → Entrada `EX`. Permite llevar el valor del registro destino desde la salida de la etapa `MEM` a la entrada de la etapa `EX` en el ciclo de reloj siguiente. Se emplea en dos casos: cuando la primera instrucción es una instrucción de carga, o bien cuando hay una instrucción entre dos instrucciones dependientes.
- Salida `EX` → Entrada `ID`. Permite llevar el valor del registro destino desde la salida de la etapa `EX` a la entrada de la etapa `ID` en el ciclo de reloj siguiente. Se emplea cuando una instrucción de salto condicional con evaluación *agresiva de saltos* tiene una dependencia RAW con una instrucción anterior que no es de carga.

Vamos a responder a las mismas preguntas anteriores pero ahora suponiendo que las rutas de reenvío están activas.

- Dibuja sobre el papel o sobre una hoja de cálculo la evolución del cauce durante la ejecución del programa suponiendo que la microarquitectura implementa la evaluación *agresiva de saltos* y las rutas de reenvío. ¿Cuántos ciclos de reloj requiere para su ejecución? Responde en el [cuestionario](#): pregunta 8.
- ¿Cuántos ciclos de detención ha sufrido el programa? Responde en el [cuestionario](#): pregunta 9.
- Al no haber instrucciones con unidades de ejecución multiciclo, la suma del número de instrucciones, más el número de ciclos de detención, más el transitorio inicial de 4 ciclos debe coincidir con el número de ciclos del programa. Haz esta comprobación.
- ¿Qué rutas de reenvío se han activado? Cada ruta de reenvío debe especificarse de acuerdo al siguiente ejemplo: Salida MEM de `addi, r8, r9, 10` → Entrada MEM de `xor r5, r9, r6`. Responde en el [cuestionario](#): pregunta 10.
- ¿Cuál es el CPI resultante? Responde en el [cuestionario](#): pregunta 11.

De nuevo vamos a comprobar las respuestas utilizando el simulador.

- Abre el simulador pyMIPS64 y activa ahora las opciones `Forwarding` y `Early branch evaluation` en la microarquitectura. Observarás que la simulación se reinicia.
- Lleva a cabo de nuevo la simulación. A partir de los resultados de la simulación comprueba tus respuestas anteriores. Si alguna no coincide debes comparar la evolución del cauce sobre el papel y la simulación.
- Para comprobar si has respondido correctamente a la pregunta sobre las rutas de reenvío y los registros asociados, puedes activar la opción `Show instruction timeline details` en la configuración, .

2.3. Renombrado de registros

El reciclaje de registros es una técnica empleada por los compiladores para tratar de optimizar el rendimiento de los programas. Un compilador siempre que puede ubica las variables en registros en lugar de la memoria, al ser el acceso a los registros mucho más rápido. Por esta razón, cuando el compilador detecta en el programa fuente que a partir de un cierto punto la variable ya no aparece en ninguna instrucción, entiende que el registro sobre el que se almacena está disponible y lo reutiliza para otra variable. La consecuencia es la introducción de dependencias WAW y WAR que podrían evitarse.

El reciclaje de registros en programas complejos es inevitable, pues el número de registros que define la arquitectura del juego de instrucciones es muy limitado. No obstante, empleando la técnica de renombrado de registros se dispone de un número mucho mayor de registros físicos, denominados registros renombrados, que el procesador mapea automáticamente a los registros arquitectónicos.

Vamos a ver los efectos del reciclaje de registros sobre el rendimiento empleando el programa `2-3recycling.asm`. Observa a la derecha los comentarios mostrando el código fuente a partir del cual se obtiene el programa ensamblador.

```
1
2 .code
3
4 ori  r5, r0, 20    ; a = r5 = 20
5 ori  r1, r0, 10    ; y = r1 = 10
6 dmul r3, r1, r1    ; x = y * y = r1 * r1
   dadd r3, r5, r5    ; b = a + a = r5 + r5
```

Este programa incluye una instrucción multiciclo de multiplicación de enteros, la cual requiere un tiempo de ejecución mucho mayor que la instrucción de suma que le sigue. El objetivo es forzar la ejecución fuera de orden para así generar una detención por dependencia de datos WAW. En particular, la multiplicación empleará la unidad multiciclo no segmentada, EXB, la cual se configurará con una latencia de 5 ciclos de reloj. Debe configurarse la microarquitectura con la opción `Out-of-order retirement` para permitir la terminación de instrucciones fuera de orden. Además, para reducir el número de ciclos del programa se supondrán activas las rutas de reenvío.

- ¿Qué instrucciones tienen dependencia de datos WAW o WAR? ¿Qué registro crea esa dependencia? Responde en el [cuestionario](#): pregunta 12.
- Dibuja sobre el papel o sobre una hoja de cálculo la evolución del cauce durante la ejecución del programa. ¿Cuántos ciclos de reloj requiere para su ejecución? Responde en el [cuestionario](#): pregunta 13.
- ¿Qué tipo de detenciones ha sufrido y de qué duración? Responde en el [cuestionario](#): pregunta 14.
- ¿Cuál es el CPI del programa ignorando el transitorio inicial? Responde en el [cuestionario](#): pregunta 15.
- ¿Qué explicación tiene un CPI tan alto y alejado del teórico CPI=1? Responde en el [cuestionario](#): pregunta 16.

Ahora debes comprobar las respuestas anteriores usando el simulador.

- Abre el simulador pyMIPS64. Dentro de la configuración de la microarquitectura y lleva a la unidad multiciclo no segmentada, EXB, la instrucción `dmul`. Configura además la latencia de esa unidad con 5 ciclos de reloj y activa las rutas de reenvío.
- Carga y ejecuta el programa en el simulador.

La detención por dependencia de datos WAW está causada por el reciclaje del registro `r3`, el cual se emplea para almacenar inicialmente la variable `x` y más tarde la variable `b`. Si en lugar de reciclar el registro `r3` se usase un registro diferente para almacenar la variable `b`, la dependencia de datos WAW desaparecería, pero en muchos casos no es posible por el gran número de registros que requieren los programas reales.

La técnica de renombrado de registros se basa en emplear un número de registros físicos mayor que el número de registros arquitectónicos. En el caso del simulador pyMIPS64 la microarquitectura simulada emplea 64 registros físicos de enteros y 64 registros físicos de coma flotante, frente a los 32 registros de cada tipo que tiene la arquitectura. En la microarquitectura empleada, para poder observar la reducción en ciclos de reloj obtenida del renombrado de registros será necesario activar la terminación fuera de orden de las instrucciones.

- Dibuja sobre el papel o sobre una hoja de cálculo la evolución del cauce durante la ejecución del programa, pero ahora suponiendo que la CPU implementa el renombrado de registros. ¿Cuántos ciclos de reloj requiere para su ejecución? Responde en el [cuestionario](#): pregunta 17.
- ¿Cuál es el CPI del programa ignorando el transitorio inicial? Responde en el [cuestionario](#): pregunta 18.
- ¿Qué aceleración ha proporcionado en el programa anterior el empleo del renombrado de registros? Responde en el [cuestionario](#): pregunta 19.
- Para cada una de las instrucciones del programa debes indicar en el cronograma los cambios que experimenta la tabla de renombrado en las etapas `ID` y `WB`. Debes emplear la notación habitual, `rx->rry:n`, donde `rx` es el registro arquitectónico, `rry` el registro físico asociado y `n` el contador de instrucciones pendientes de terminar que usan el registro físico `rry` como registro fuente.
- ¿Qué registros arquitectónicos cambian de registro físico asociado? ¿Qué nuevos registros físicos tienen asociados? Responde en el [cuestionario](#): pregunta 20.

Como de costumbre, vamos a comprobar las respuestas anteriores empleando el simulador.

- Activa las opciones `Register renaming` y `Out-of-order retirement` en el simulador. Observarás que se reinicia la simulación y aparece un nuevo panel de título `Register renaming table`. En ese panel se muestran los cambios que experimenta la tabla de renombrado respecto a su estado inicial, es decir, con los registros arquitectónicos asociados a registros físicos de mismo índice y con todos los contadores a cero.
- Ejecuta ciclo a ciclo el programa en el simulador. Observa cómo el renombrado se realiza en la etapa ID de cada instrucción (salvo cuando está detenida) y a partir de ese momento en el panel `Source code` y en la ventana principal se indica al lado de cada instrucción los registros físicos con los que trabaja.
- Observa el cronograma y los cambios que se producen en la tabla de renombrado en las etapas ID y WB. Comprueba que coinciden con los que has predicho. Los elementos de la tabla marcados en rojo se corresponden con cambios en el último ciclo de reloj. Recuerda además que en cualquier momento puede simular hacia adelante o hacia atrás si algún cambio no te ha quedado claro.
- Al final de la ejecución del programa la tabla de renombrado te indicará los cambios en la asignación de registros físicos a registros arquitectónicos. Comprueba que tu última respuesta fue correcta.
- Activa la opción de configuración del simulador `Show instruction timeline details` y observa la ruta de reenvío activa. Verás que al activar el renombrado de registros las rutas de reenvío se establecen entre los registros físicos.

En el ejemplo mostrado el reciclaje es fácilmente evitable, pues se dispone de muchos registros arquitectónicos sobrantes. Sin embargo, en un programa real, mucho más complejo y grande, podría no haber suficientes registros arquitectónicos. Incluso empleando el renombrado de registros, el número de registros físicos disponible puede no ser suficiente y podrían aparecer detenciones por dependencias de datos WAW, aunque lógicamente en un número menor que si no se emplease esta técnica de mejora del rendimiento.

La configuración empleada en la microarquitectura no es compatible con el soporte de excepciones precisas, pues al estar la opción `Out-of-order retirement` activada se impone la terminación de instrucciones en orden.

- Desactiva la opción `out-of-order retirement` para imponer la terminación de instrucciones en orden.
- Realiza la simulación completa del programa.
- ¿Cuántos ciclos de detención sufre la instrucción `dadd` en su fase de ejecución? ¿A qué son debidos? Responde en el [cuestionario](#): pregunta 21.

Para comprobar la respuesta anterior puedes activar la opción `Show instruction timeline details` en la configuración, el simulador te indicará el tipo de detención sufrida.

3. Reducción de las detenciones por dependencias de control

Las detenciones por dependencias de control se producen cuando aparecen cambios de flujo en la ejecución secuencial de los programas. Estas dependencias suponen la detención del *pipeline* hasta que se determina cuál es la siguiente instrucción a ejecutar. Para el caso de los saltos condicionales, es posible utilizar técnicas de predicción de saltos para ejecutar de forma especulativa una de las dos ramas del salto. Para el caso de los saltos condicionales e incondicionales además se puede guardar la dirección destino de salto calculada para futuras ejecuciones de la instrucción.

En este apartado se emplearán dos predictores, el predictor *siempre no tomado* y el predictor dinámico de dos bits. Se compararán los resultados obtenidos con una configuración que no emplea predicción. En todos los casos se empleará evaluación *agresiva de saltos* y rutas de reenvío para reducir la longitud de los cronogramas del *pipeline*.

Partimos del programa `2-3branch.asm`, mostrado a continuación.

```
1
2 .code
3     ori r1, r0, 2 ; r1 = 2
4 loop:
5     beqz r1, endloop ; Branch on equal to zero
6     daddi r1, r1, -1 ; r1 = r1 - 1
7     j loop
8 endloop:
    xor r1, r1, r1 ; r1 = 0
```

- Abre en el simulador el archivo de programa `2-3branch.asm`.
- Configura adecuadamente la microarquitectura de la CPU en el simulador. Para evitar errores restaura la microarquitectura por defecto pulsando el botón **Reset to defaults** y a continuación activa las opciones de evaluación *agresiva de saltos* y rutas de reenvío.
- Realiza ahora una simulación completa. ¿Cuántos ciclos de reloj requiere la ejecución del programa anterior? Responde en el [cuestionario](#): pregunta 22.
- Identifica las detenciones de control sufridas, indicando los ciclos detención de cada una de ellas y las causas de las mismas. Responde en el [cuestionario](#): pregunta 23.
- Activa la opción `Show instruction timeline details` en la configuración del simulador. la información mostrada te permitirá comprobar si has identificado correctamente las detenciones de control.

3.1. Predictor *siempre no tomado*

En este apartado vamos a simular el mismo programa anterior, pero ahora empleando el predictor *siempre no tomado*.

- Configura `Always-not-taken` en la opción `Branch prediction` de la microarquitectura. El resto de opciones deben ser las mismas que las del apartado anterior.
- Realiza una simulación completa. ¿Cuántos ciclos de reloj requiere la ejecución del programa anterior? Responde en el [cuestionario](#): pregunta 24. ¿Qué aceleración ha introducido el predictor con respecto al caso de no emplear predicción? Responde en el [cuestionario](#): pregunta 25.
- ¿Cuántos aciertos y cuántos fallos ha experimentado el predictor? Recuerda que no existe predicción en los saltos incondicionales. Responde en el [cuestionario](#): pregunta 26.
- Comprueba tu respuesta activando la opción `Show instruction timeline details` en la configuración del simulador. En cada etapa `ID` de la instrucción de salto condicional se muestra la predicción realizada y si se ha producido un acierto o fallo de predicción. La predicción realizada es siempre *Not Taken* (NT) y puede existir un acierto (HIT), o un fallo (MISS).
- ¿Cuántos ciclos de detención han sido causados por la instrucción de salto incondicional? Responde en el [cuestionario](#): pregunta 27.

3.2. Predictor dinámico de dos bits

En esta ocasión simularemos el mismo programa anterior, pero ahora empleando el predictor dinámico de dos bits.

- Configura `Two-bit dynamic` en la opción `Branch prediction` de la microarquitectura. El resto de opciones no deben modificarse.

- Realiza una simulación completa. ¿Cuántos ciclos de reloj requiere la ejecución del programa anterior? Responde en el [cuestionario](#): pregunta 28. ¿Qué aceleración ha introducido el predictor con respecto al caso de no emplear predicción? Responde en el [cuestionario](#): pregunta 29.
- ¿Cuántos aciertos y cuántos fallos ha experimentado el predictor? Recuerda que no existe predicción en los saltos incondicionales. Responde en el [cuestionario](#): pregunta 30.
- Comprueba tu respuesta activando la opción `Show instruction timeline details` en la configuración del simulador. En cada etapa `ID` de la instrucción de salto condicional se muestra la predicción realizada y si se ha producido un acierto o fallo de predicción. La predicción realizada puede ser *Not Taken* (`NT`) o *Taken* (`TK`) y puede existir un acierto (`HIT`), o un fallo (`MISS`).
- ¿Cuántos ciclos de detención han sido causados por la instrucción de salto incondicional? Responde en el [cuestionario](#): pregunta 31.

El predictor dinámico emplea una tabla de predicción. En la etapa `IF` realiza una predicción y/o obtención de la dirección destino de salto basada en el estado de la tabla, mientras que en la etapa `ID` (o `MEM` para los saltos con condicionales con evaluación en etapa `MEM`) se realiza la evaluación.

- Lleva la simulación al comienzo, pulsando sobre el icono o usando el atajo correspondiente.
- Observa que al estar activo el predictor de saltos dinámico ha aparecido un nuevo panel, de título `Branch prediction table`. Este panel muestra el estado de la tabla de predicción de saltos durante la simulación. Quizás tengas que redimensionarlo para visualizarlo correctamente. Cada entrada de la tabla tendrá tres campos: dirección (`Address`), dirección destino del salto (`Destination`) y los dos bits de estado (`Status bits`).
- Simula ciclo de reloj a ciclo de reloj hasta completar el ciclo de reloj 10. Durante el proceso analiza la predicción que se hace en la etapa `IF` cada vez que se ejecuta la instrucción de salto condicional. Asimismo, analiza el cambio que se produce en la tabla de traducción cada vez que la etapa `ID` procesa una instrucción de salto condicional o incondicional.
- Una vez se simula el ciclo 10 observarás que la instrucción de salto incondicional no produce una detención de control. ¿Cuál es la razón? Responde en el [cuestionario](#): pregunta 32.
- Simula un ciclo más hasta completar el ciclo de reloj 11. Se produce un fallo de predicción. ¿Cuál es la razón? Responde en el [cuestionario](#): pregunta 33.

4. Ejercicios

El compilador puede jugar un papel fundamental a la hora de generar código para evitar detenciones por riesgos de control.

Las estructuras de control que más dependencias de control introducen son los bucles. Estos están gobernados por una condición que debe ser evaluada en cada iteración, por lo que se produce al menos una dependencia de control cada vez que el bucle itera.

Para evitarlo, cuando se activan las opciones de optimización del compilador, éste puede hacer uso de una optimización denominada desenrollado de bucles. Veámoslo con un ejemplo.

```
1
2 for (i = 0; i != 16; i++)
3 {
4     a = a + c;
5 }
```

Aunque el bucle mostrado podría sustituirse por la expresión `a = 16 * c`, se opta por un ejemplo sencillo para facilitar la comprensión.

4.1. Sin desenrollado de bucles

El código que generaría el compilador podría ser el siguiente:

```
1
2      .code
3
4  main:
5      xor      r8, r8, r8
6      daddi    r9, r0, 16
7  for:
8      beq      r8, r9, end
9      dadd     r10, r10, r11 ; a = a + c
10     daddi    r8, r8, 1
11     j        for
12 end:
      nop
```

- Edita el fichero `2-3loop.asm` para que tenga el contenido mostrado arriba.
- Abre el simulador pyMIPS64, carga el fichero `2-3loop.asm` y configúralo el simulador con las rutas de reenvío activas, evaluación *agresiva de saltos* y el predictor dinámico de dos bits. A continuación ejecuta el programa. ¿Cuántos ciclos necesita para ejecutarse? Responde en el [cuestionario](#): pregunta 34.

4.2. Con desenrollado de bucles

El desenrollado de bucles busca disminuir el número de veces que se comprueba la condición del bucle. Para ello, se replica el cuerpo del mismo en un número divisor del número total de iteraciones. En el ejemplo, se podría replicar el cuerpo del bucle 4 veces, ya que 4 es divisor del total de 16 iteraciones.

```
1
2      .code
3
4  main:
5      xor      r8, r8, r8
6      daddi    r9, r0, 16
7  for:
8      beq      r8, r9, end
9      dadd     r10, r10, r11 ; a = a + c
10
11     dadd     r10, r10, r11 ; a = a + c
12
13     dadd     r10, r10, r11 ; a = a + c
14
15     dadd     r10, r10, r11 ; a = a + c
16     daddi    r8, r8, 4
17     j        for
18 end:
      nop
```

Fíjate cómo la variable de control del bucle debe ser incrementada por el número de veces que se replica el cuerpo del bucle.

- Edita el fichero `2-3loop-unroll.asm` para que tenga el contenido mostrado arriba.
- Abre el simulador pyMIPS64, carga el fichero `2-3loop-unroll.asm`, configura el simulador con las mismas opciones que el caso anterior y simula el programa. ¿Cuántos ciclos necesita para ejecutarse? Responde en el [cuestionario](#): pregunta 35.
- ¿Cuál es la aceleración de la versión con el bucle desenrollado respecto de la versión inicial? Responde en el [cuestionario](#): pregunta 36.

Como ves, el desenrollado de bucles constituye una optimización que puede utilizar el compilador para acelerar la ejecución de programas. Como contrapartida, los programas serán más grandes en memoria, ya que estarán formados por más instrucciones.

Objetivos de la sesión

Esta sesión trata aspectos relacionados con el soporte por parte de la CPU a los sistemas operativos multitarea y a la virtualización.

En una primera parte de la sesión se muestran los principales mecanismos de protección que dan soporte a los sistemas operativos multitarea. Incluyen al menos los siguientes tipos de protección:

- Niveles de privilegio. Evitan que las tareas puedan ejecutar instrucciones privilegiadas. El sistema operativo, en cambio, puede ejecutar cualquier instrucción del juego de instrucciones, incluyendo las privilegiadas.
- Protección de memoria. Evita que una tarea pueda acceder al área de memoria de otra tarea o al área de memoria del sistema operativo.

Para ilustrar los mecanismos de protección anteriores se realizarán varios programas sobre un sistema operativo Linux.

Finalmente, se tratan algunos de los conceptos de virtualización explicados en teoría.

Conocimientos y materiales necesarios

Para poder realizar esta sesión, el alumno debe:

- Revisar los apuntes de teoría correspondientes al apartado dedicado al soporte de la CPU a los sistemas operativos multitarea y a la virtualización.
- Durante la sesión se plantearán una serie de preguntas que deben responderse en el correspondiente [cuestionario](#) en el campus virtual. Puedes abrir el cuestionario en otra pestaña del navegador pinchando en el enlace mientras mantienes pulsada la tecla `Ctrl`.

La sesión práctica se realizará sobre la máquina virtual Linux de la asignatura.

1. La protección en el sistema operativo Linux

Para probar la protección de los sistemas operativos emplearemos la máquina virtual Linux de la asignatura.

Algunas pruebas que realizaremos en Linux pueden exigir ejecutar determinadas instrucciones en ensamblador. En ese caso incluiremos instrucciones ensamblador dentro del programa C. La sintaxis es análoga a la del ejemplo siguiente:

```
1
2 #include <stdio.h>
3 int main()
4 {
5     printf("Hello\n");
6
7     __asm__
8     (
9         "mov %ax, 0\n\t"
10        "add %ax, %cx"
11    );
12 }
```

Observa cómo se pone cada instrucción ensamblador en una línea y todas las líneas salvo la última se terminan con `\n\t`. Aunque no se aprecia bien en el código de ejemplo, justo antes y después de la palabra `asm` hay dos guiones bajos consecutivos.

- Arranca la máquina virtual Linux.
- Cambia al directorio de tu repositorio y sincronízalo con el repositorio de Bitbucket para actualizar los cambios que se hubiesen realizado con anterioridad.

```
$> git pull
```

- Descarga los ficheros necesarios para la práctica y descomprímelos con estas órdenes:
- ```
$> wget http://rigel.atc.uniovi.es/grado/2ac/files/sesion2-4.tar.gz
```

```
$> tar xvfz sesion2-4.tar.gz
```
- Añade los ficheros al índice git y confirma los cambios con estas órdenes:
- ```
$> git add sesion2-4
```



```
$> git commit
```
- Cámbiate al directorio `sesion2-4`.

1.1. Comprobación de los niveles de privilegio

Los niveles de privilegio impiden que las tareas puedan ejecutar determinadas instrucciones de la arquitectura. Si intentan hacerlo la instrucción correspondiente da lugar a una excepción.

En los sistemas Linux, y Unix en general, cuando un programa termina por una excepción se puede generar un archivo de nombre `core` dentro del mismo directorio en el que se encuentra el ejecutable. Dicho archivo contiene una imagen de la memoria del programa y de sus registros, lo que permite el análisis *post mortem* de las causas del problema. Sin embargo, en algunos sistemas dicho archivo no se genera por defecto, pues muchos usuarios no lo usan.

- Ejecuta la orden `ulimit -c unlimited` para forzar la generación del archivo `core` en caso de excepción.

Pasemos a comprobar la existencia de niveles de privilegio.

- Edita el archivo `2-4priv.c` proporcionado para visualizar su contenido.
- Compila el programa anterior y ejecútalo. Verás que Linux indica que ha habido un error de ejecución ¿Qué ha ocurrido? ¿Cuál es la razón? Responde en el [cuestionario](#): pregunta 1.
- Para comprobar la respuesta vamos a ejecutar el depurador `gdb` que es el habitual de los sistemas Linux. Ejecuta la siguiente orden.

```
$> gdb 2-4priv core
```

Una vez ejecutada la orden anterior el depurador nos muestra la causa de la excepción y nos proporciona una interfaz de comandos propia que permite introducir *breakpoints*, visualizar el contenido de la memoria y registros, etc. En la última línea observarás que hace referencia a la función `main()`, lo que no da muchas pistas sobre la instrucción causante de la excepción. La razón es que el archivo ejecutable no contiene información de depuración.

- Sal del depurador con la orden `quit`.
- Borra el archivo `core` y compila ahora el programa `2-4priv.c` usando la opción `-g`, la cual añade dentro del archivo ejecutable información para la depuración.

Debes borrar siempre el archivo `core`, pues si ya existe **NO SE SOBREScribe**.

- Ejecuta de nuevo el programa y analiza la causa de la excepción empleando el depurador `gdb`. Ahora te indicará una línea en el archivo fuente. Observarás que no hace referencia exactamente a la instrucción en ensamblador, sino al bloque ensamblador. La explicación es que el bloque en su conjunto se interpreta como una única instrucción de alto nivel.
- Para ver la instrucción ensamblador que produce la excepción teclea `layout asm` en `gdb`. Verás que se abre una ventana con la traducción a ensamblador del código del programa y que el programa está detenido sobre la instrucción concreta que produjo la excepción, en este caso `cli`.

Si quieres volver a ver el código del programa en lenguaje C, teclea `layout prev` en `gdb`.

- Sal de la interfaz de comandos de `gdb` y borra el archivo `core`.

Como habrás podido observar, el sistema operativo Linux impide la ejecución de la instrucción de deshabilitación de interrupciones. Si no fuese así, cualquier tarea que se ejecutase sobre Linux podría provocar un mal funcionamiento del sistema. Para que Linux pueda proporcionar esta protección la CPU debe dar el soporte necesario.

A continuación vamos a comprobar la protección de memoria de Linux.

1.2. Comprobación de la protección de memoria

La memoria está protegida en Linux, de tal forma que una tarea sólo puede acceder a las áreas de memoria que tiene asignadas.

- Edita el archivo `2-4mem1.c` proporcionado para visualizar su contenido.
- Compila el programa `2-4mem1.c` con la opción de depuración y a continuación ejecútalo. ¿Qué ha ocurrido? ¿Cuál es la razón? Responde en el [cuestionario](#): pregunta 2.
- Para comprobar la respuesta ejecuta el depurador con la orden `gdb 2-4mem1 core`. ¿Qué instrucción da lugar a la excepción? Responde en el [cuestionario](#): pregunta 3.
- Recuerda salir del depurador con la orden `quit` y borrar el archivo `core` generado.
- ¿Por qué crees que la instrucción que asigna la dirección de memoria al puntero `p` no genera una excepción? Responde en el [cuestionario](#): pregunta 4

A partir de la prueba anterior hemos comprobado que la memoria está protegida en Linux, gracias al soporte proporcionado por el procesador. Cuando una tarea intenta acceder a una dirección que no le pertenece se produce una excepción.

Las excepciones son muy beneficiosas durante el proceso de depuración de programas pues permiten detectar problemas que de otra manera serían muy difíciles de encontrar.

- Edita un programa de nombre `2-4mem2.c` con el siguiente código.

```
1
2 #include <stdio.h>
3
4 int main()
5 {
6     int *p = NULL; /* Pointers should be initialized to NULL */
7
8     printf("%d\n", *p);
9 }
```

La constante `NULL` se emplea para inicializar punteros en C y habitualmente toma el valor cero.

- ¿Qué instrucción crees que dará lugar a una excepción? Responde en el [cuestionario](#): pregunta 5.
- Compila y ejecuta el programa anterior. Utiliza el `gdb` para identificar la instrucción que da lugar a una excepción. ¿Coincide con la que habías predicho? Recuerda salir del depurador con la orden `quit` y borrar el archivo `core` generado.

La inicialización de punteros a `NULL` permite detectar cuando se escribe de forma incorrecta en un puntero no inicializado, lo cual es un error bastante común cuando se programa en C.

2. Virtualización en x86-64

Los ordenadores de la sala de prácticas utilizan la arquitectura x86-64 (a veces llamada x64), que es una evolución de 64 bits sobre la arquitectura de 32 bits desarrollada por Intel y denominada habitualmente x86. La arquitectura x86 no estaba pensada para ser virtualizada y no cumplía los requisitos de Popek y Goldberg para una virtualización eficiente. En concreto, había instrucciones sensibles que no eran privilegiadas.

El funcionamiento habitual de la arquitectura x86 define distintos niveles de privilegio a los que denomina anillos. Habitualmente, el núcleo del sistema operativo funciona en el anillo 0, que es el de más privilegio, y las aplicaciones, en el anillo 3, con menor privilegio. Los primeros programas de virtualización movieron el sistema operativo al anillo 1 y ejecutaban el hipervisor o Virtual Machine Monitor (VMM) en el nivel 0. Sin embargo, esto no bastaba para implementar la técnica de *trap-and-emulate* debido a las instrucciones sensibles que no provocaban una excepción cuando se ejecutaban en el anillo 1, con lo que había que utilizar también traducción binaria para lidiar con estas instrucciones, lo que reducía el rendimiento.

Para evitar este problema, tanto AMD como Intel introdujeron unas extensiones de soporte a la virtualización, denominadas respectivamente AMD-V e Intel VT-x, que añadían un nuevo modo de operación de la CPU, llamado *Guest Mode* en AMD y *VMX Root Mode* en Intel. Este modo tiene más privilegio que el anillo 0 y permite que el hipervisor o Virtual Machine Monitor (VMM) se ejecute en este nivel (a veces llamado anillo -1) controlando los sistemas operativos que se ejecutan en el anillo 0.

En las prácticas se utiliza VirtualBox como VMM. ¿Qué tipo de VMM (1 o 2) es VirtualBox y por qué? Responde en el [cuestionario](#): pregunta 6.

Vamos a analizar algunas opciones de VirtualBox.

- Abre VirtualBox y para tu máquina virtual Linux vete a la pestaña Aceleración dentro de **Configuración > Sistema**. ¿Se está utilizando VT-x/AMD-V? Responde en el [cuestionario](#): pregunta 7.

Como verás, en esa ventana hay una opción para seleccionar el tipo de interfaz de paravirtualización. Como has estudiado en clase, la paravirtualización consiste en modificar el sistema operativo para que incluya llamadas a funciones del hipervisor. Por lo tanto, estas funciones son distintas dependiendo del tipo de hipervisor que se esté utilizando. Los sistemas operativos modernos suelen ofrecer estas funciones de paravirtualización, pero cada sistema operativo ofrece llamadas a sólo algunos tipos de hipervisores.

VirtualBox puede aprovecharse de algunas funciones de paravirtualización para mejorar aún más el rendimiento. Entre las opciones que ofrece VirtualBox están:

- Ninguna: no utiliza estas extensiones.
- Predeterminada: selecciona la opción en función del sistema operativo invitado seleccionado durante la creación de la máquina virtual.

- Heredada: es para máquinas virtuales creadas con versiones antiguas de VirtualBox.
- Mínima: simplemente le indica al sistema operativo invitado que se está ejecutando dentro de un hipervisor y le da cierta información sobre la frecuencia de algunos elementos del sistema. Es necesaria cuando el sistema operativo invitado es Mac OS X.
- Hyper-V: utiliza las llamadas al hipervisor Hyper-V creado por Microsoft.
- KVM: utiliza las llamadas al hipervisor KVM (Kernel-based Virtual Machine) desarrollado para virtualizar sistemas Linux.

Los sistemas Linux tienen, lógicamente, extensiones para hacer llamadas al hipervisor KVM pero también para Hyper-V, ya que Microsoft aportó al núcleo de Linux el código necesario para que funcionasen mejor los invitados Linux en los anfitriones Windows, algo que utiliza, por ejemplo, en Azure, su nube de máquinas virtuales, que permite ejecutar máquinas virtuales bajo demanda en servidores de Microsoft.

La opción por defecto en la sección de paravirtualización es `Predeterminada`. Vamos a comprobar que, como estamos ejecutando un sistema invitado Linux, eso se traduce en que se utiliza KVM.

- La orden `dmesg` permite obtener los mensajes que genera el núcleo. Durante el inicio del sistema, el núcleo indica si se está utilizando paravirtualización, así que vamos a buscar algún mensaje relativo a la paravirtualización con esta orden:

```
$> dmesg | grep paravirt
```

Como verás, se generan dos líneas que te indican que se está utilizando KVM. ¿Qué dice la primera línea? Responde en el [cuestionario](#): pregunta 8.

2.1. Número de procesadores

Una de las ventajas de la virtualización es que se pueden repartir los recursos de una máquina física entre varias máquinas virtuales de manera controlada. Uno de los recursos más fáciles de repartir es la CPU en sistemas multinúcleo, ya que se puede limitar el número de núcleos que utiliza la máquina virtual. Vamos a comprobar cómo afecta esto al rendimiento.

- En primer lugar, vamos a comprobar cuántos núcleos tiene el procesador físico. Abre el administrador de tareas de Windows pulsando `Ctrl+Shift+Esc`. Si no te aparecen pestañas, pulsa en el botón `Más detalles`. Escoge la pestaña `Rendimiento` y en la sección de CPU, mira cuántos núcleos te indica. Responde en el [cuestionario](#): pregunta 9.
- En VirtualBox, vete a la sección `Procesador` dentro del menú **Configuración > Sistema** de la máquina virtual. Si no están asignados dos procesadores, debes cambiar el valor, pero para ello tendrás que detener antes la máquina virtual y arrancarla después.
- Vamos a comprobar que ese número de núcleos es el que se ve en la máquina virtual. Ejecuta la orden `nproc --all`. Comprueba que te sale lo mismo que en la respuesta anterior.
- Abre Visual Studio y crea un proyecto de consola de Visual C++ de tipo `Aplicación de consola Win32` llamado `CpuBurn`. En la función `main` del programa copia este código:

```
for (;;)
{
    // Infinite loop to burn CPU
}
```

La idea es que tenemos un bucle infinito vacío simplemente para ocupar una CPU.

- Ejecuta el programa `CpuBurn` y asegúrate, mirando el administrador de tareas, de que uno de los núcleos está ocupado mientras lo estás ejecutando.
- Mientras se está ejecutando `CpuBurn`, ejecuta en la máquina virtual el benchmark `bench_fp_mt`, creado en la sesión 1.2, mientras observas qué ocurre con el reparto de la CPU en la pestaña

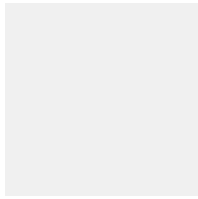
Procesos del administrador de tareas. ¿Qué porcentaje de CPU ocupa VirtualBox y qué porcentaje el proceso CpuBurn? Responde en el [cuestionario](#): pregunta 10

- Cierra la máquina virtual, cambia el número de núcleos de la máquina virtual a 1 y vuelve a arrancarla.
- Mientras sigue ejecutándose CpuBurn, vuelve a ejecutar el benchmark bench_fp_mt mientras observas qué ocurre con el reparto de la CPU en la pestaña Procesos del administrador de tareas. ¿Qué porcentaje de CPU ocupa ahora VirtualBox y qué porcentaje el proceso CpuBurn? Responde en el [cuestionario](#): pregunta 11
- Cierra el programa CpuBurn. Limpia el proyecto, cierra el Visual Studio, comprime el directorio del proyecto con el nombre 2-4CpuBurn.zip y añade el archivo resultante al repositorio.

Pregunta 1

Finalizado

Sin calificar



Marcar pregunta

Enunciado de la pregunta

¿Cuál es la dirección de memoria más alta que ocupa la variable **intval** ? Responde en hexadecimal sin incluir prefijos o sufijos y omitiendo los 0 por la izquierda.

Respuesta:

404

Retroalimentación

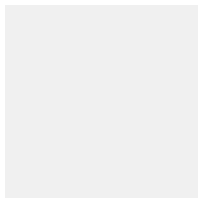
Un dato de tipo palabra requiere requiere 32 bits en memoria, esto es, 4 bytes. Dado que la CPU direcciona al byte, esto significa 4 direcciones. Por tanto, la dirección más alta es la dirección donde comienza la variable más 3.

La respuesta correcta es: 403

Pregunta 2

Finalizado

Sin calificar



Marcar pregunta

Enunciado de la pregunta

¿Por qué la variable **array** no ocupa la dirección siguiente a la última de la variable **intval** ?

Debe tomar como primera posición una que sea múltiplo de 8.

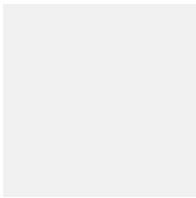
Retroalimentación

Esto se debe a que la variable **array** apunta a un vector de dobles palabras. Dado que los accesos a dobles palabras deben estar alineados a 8 y la dirección más alta de **intval** no lo está, el vector comienza desde la dirección alineada a 8 más baja después de **intval**.

Pregunta 3

Finalizado

Sin calificar



Marcar pregunta

Enunciado de la pregunta

Una vez que el programa termina, se escriben Respuesta

desde la dirección de memoria Respuesta

448

h con el valor Respuesta

64

en decimal.

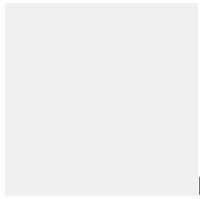
Retroalimentación

La variable **max** será escrita para guardar el máximo de la lista. Dado que el máximo es 100 y que se ha utilizado el tipo doble palabra para los elementos de la lista, se escribirán 8 bytes desde la posición 448h.

Pregunta 1

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

Suponiendo que existe una versión no segmentada de la microarquitectura MIPS64, es decir, las 5 etapas en las que se divide el cauce de ejecución se ejecutan secuencialmente y no en paralelo, ¿cuántos ciclos serían necesarios para ejecutar el programa 2-2ideal.s?

Respuesta:

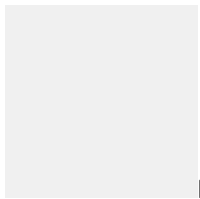
Retroalimentación

La respuesta correcta es: 50

Pregunta 2

Incorrecta

Puntúa 0,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

Si una CPU MIPS64 no segmentada trabajase a una frecuencia de reloj de 400 MHz, ¿cuánto tiempo invertiría en ejecutar cada instrucción? Indica la unidad.

Respuesta:

Retroalimentación

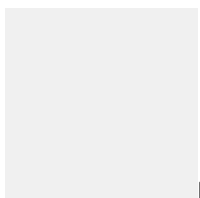
La unidad no es correcta

La respuesta correcta es: 12,5 ns

Pregunta 3

Incorrecta

Puntúa 0,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

Si una CPU MIPS64 no segmentada trabajase a una frecuencia de reloj de 400 MHz, ¿cuánto tiempo invertiría en ejecutar el programa 2-2ideal.asm? Indica la unidad.

Respuesta:

Retroalimentación

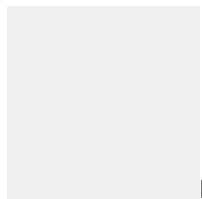
La unidad no es correcta

La respuesta correcta es: 125 ns

Pregunta 4

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

Si una CPU MIPS64 no segmentada trabajase a una frecuencia de reloj de 400 MHz, ¿cuál sería la productividad teórica medida en MIPS que puede alcanzar?

Respuesta:

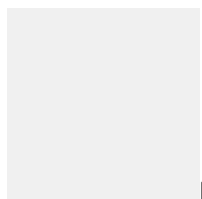
Retroalimentación

La respuesta correcta es: 80

Pregunta 5

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

¿Cuál es la productividad máxima teórica en MIPS que puede alcanzar la CPU MIPS64 que trabaja a 400 MHz?

Respuesta:

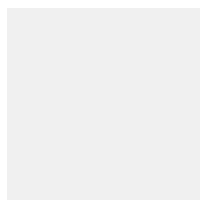
Retroalimentación

La respuesta correcta es: 400

Pregunta 6

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

[Enunciado de la pregunta](#)

Si se supone que ninguna de las instrucciones genera una detención, ¿cuál sería el número de ciclos necesarios para ejecutar el programa 2-2ideal.asm por parte de la CPU MIPS64?

Respuesta:

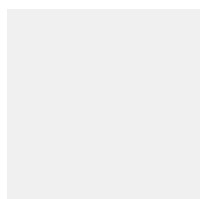
[Retroalimentación](#)

La respuesta correcta es: 14

Pregunta 7

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

[Enunciado de la pregunta](#)

¿Cuántos ciclos de reloj invertiría la CPU MIPS64 en ejecutar el programa 2-2data.asm suponiendo que no se produjesen detenciones por riesgos de la segmentación?

Respuesta:

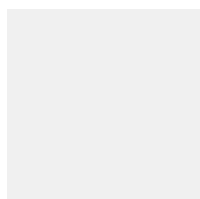
[Retroalimentación](#)

La respuesta correcta es: 11

Pregunta 8

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

[Enunciado de la pregunta](#)

¿Cuántos ciclos de reloj invierte la CPU MIPS64 en ejecutar el programa 2-2data.asm?

Respuesta:

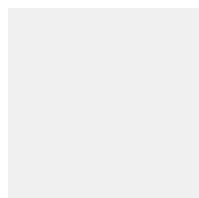
[Retroalimentación](#)

La respuesta correcta es: 13

Pregunta 9

Incorrecta

Puntúa 0,00 sobre 1,00



Marcar pregunta

[Enunciado de la pregunta](#)

¿Cuál es la aceleración obtenida en la ejecución del programa 2-2data.asm por parte de la CPU MIPS64 con respecto a una hipotética ejecución ideal sin detenciones en el cauce?

Respuesta:

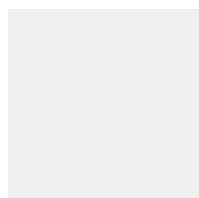
[Retroalimentación](#)

La respuesta correcta es: 0,846

Pregunta 10

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

[Enunciado de la pregunta](#)

¿Cuál es la aceleración obtenida en la ejecución del programa 2-2data.asm por parte de la CPU MIPS64 con respecto a la ejecución en una hipotética CPU MIPS64 no segmentada?

Respuesta:

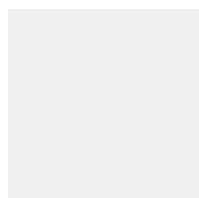
[Retroalimentación](#)

La respuesta correcta es: 2,69

Pregunta 11

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

¿Cuántos ciclos de reloj invertiría la CPU MIPS64 en ejecutar el programa 2-2contr-j.asm suponiendo que no se produjesen detenciones por riesgos de la segmentación?

Respuesta:

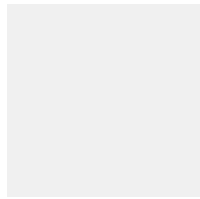
Retroalimentación

La respuesta correcta es: 11

Pregunta 12

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

¿Cuántos ciclos de reloj invierte la CPU MIPS64 en ejecutar el programa 2-2contr-j.asm?

Respuesta:

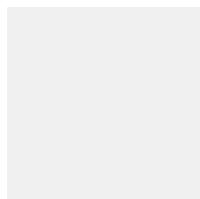
Retroalimentación

La respuesta correcta es: 12

Pregunta 13

Incorrecta

Puntúa 0,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

¿Cuál es la aceleración obtenida en la ejecución del programa 2-2contr-j.asm por parte de la CPU MIPS64 con respecto a una hipotética ejecución ideal sin detenciones en el cauce?

Respuesta:

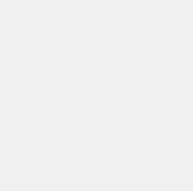
Retroalimentación

La respuesta correcta es: 0,917

Pregunta 14

Incorrecta

Puntúa 0,00 sobre 1,00



Marcar pregunta

[Enunciado de la pregunta](#)

¿Cuál es la aceleración obtenida en la ejecución del programa 2-2contr-j.s por parte de la CPU MIPS64 con respecto a una hipotética CPU MIPS64 no segmentada?

Respuesta:

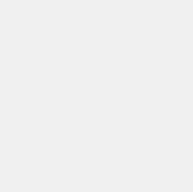
[Retroalimentación](#)

La respuesta correcta es: 2,917

Pregunta 15

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

[Enunciado de la pregunta](#)

¿Cuántos ciclos de reloj invertiría la CPU MIPS64 en ejecutar el programa 2-2contr-bnez.asm suponiendo que no se produjesen detenciones por riesgos de la segmentación?

Respuesta:

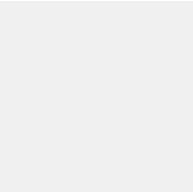
[Retroalimentación](#)

La respuesta correcta es: 16

Pregunta 16

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

[Enunciado de la pregunta](#)

¿Cuántos ciclos de reloj invierte la CPU MIPS64 en ejecutar el programa 2-2contr-bnez.asm?

Respuesta:

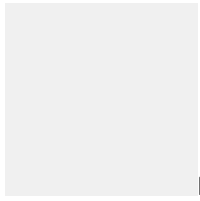
[Retroalimentación](#)

La respuesta correcta es: 18

Pregunta 17

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

[Enunciado de la pregunta](#)

¿Cuál es la aceleración obtenida en la ejecución del programa 2-2contr-bnez.asm por parte de la CPU MIPS64 con respecto a una hipotética ejecución ideal sin detenciones en el cauce?

Respuesta:

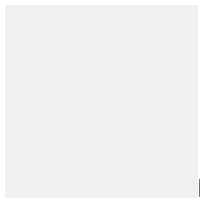
[Retroalimentación](#)

La respuesta correcta es: 0,89

Pregunta 18

Incorrecta

Puntúa 0,00 sobre 1,00



Marcar pregunta

[Enunciado de la pregunta](#)

¿Cuál es la aceleración obtenida en la ejecución del programa 2-2contr-bnez.asm por parte de la CPU MIPS64 con respecto a una hipotética CPU MIPS64 no segmentada?

Respuesta:

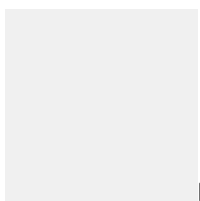
[Retroalimentación](#)

La respuesta correcta es: 3,33

Pregunta 19

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

[Enunciado de la pregunta](#)

¿Cuántos ciclos de reloj invertiría la CPU MIPS64 en ejecutar el programa 2-2struc.asm suponiendo que no se produjesen detenciones por riesgos de la segmentación?

Respuesta:

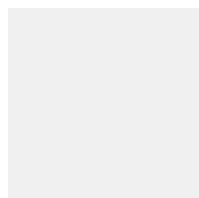
[Retroalimentación](#)

La respuesta correcta es: 15

Pregunta 20

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

[Enunciado de la pregunta](#)

¿Cuántos ciclos de reloj invierte la CPU MIPS64 en ejecutar el programa 2-2struc.asm?

Respuesta:

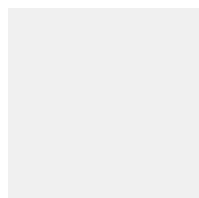
[Retroalimentación](#)

La respuesta correcta es: 17

Pregunta 21

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

[Enunciado de la pregunta](#)

¿Cuál es la aceleración obtenida en la ejecución del programa 2-2struc.asm por parte de la CPU MIPS64 con respecto a una hipotética ejecución ideal sin detenciones en el cauce?

Respuesta:

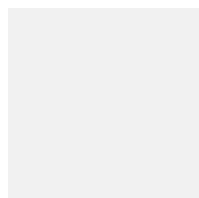
[Retroalimentación](#)

La respuesta correcta es: 0,88

Pregunta 22

Incorrecta

Puntúa 0,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

¿Cuál es la aceleración obtenida en la ejecución del programa 2-2struc.asm por parte de la CPU MIPS64 con respecto a una hipotética CPU MIPS64 no segmentada?

Respuesta:

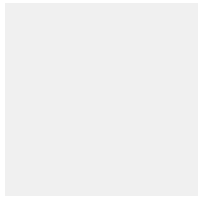
Retroalimentación

La respuesta correcta es: 3,35

Pregunta 1

Finalizado

Puntúa como 1,00



Marcar pregunta

Enunciado de la pregunta

¿Qué instrucciones tienen dependencias de datos RAW en 2-3reordering.asm? ¿De qué instrucciones dependen? ¿Qué registros crean esas dependencias?

7 y 8 dependen de 6. R1

Retroalimentación

La instrucción ld r8, 120(r1) depende de la instrucción daddi r1, r2, 5 a través del registro r1. Sin embargo, esta dependencia no produce detenciones.

La instrucción ld r8, 120(r1) depende de la instrucción xor r1, r5, r3 a través del registro r1.

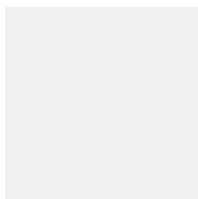
La instrucción and r8, r4, r1 depende de la instrucción daddi r1, r2, 5 a través del registro r1. Sin embargo, esta dependencia no produce detenciones.

La instrucción and r8, r4, r1 depende de la instrucción xor r1, r5, r3 a través del registro r1. Sin embargo, esta dependencia no produce detenciones.

Pregunta 2

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

¿Cuántos ciclos de reloj requiere el programa 2-3reordering.asm para su ejecución?

Respuesta:

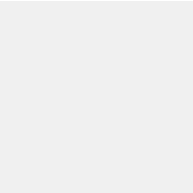
Retroalimentación

La respuesta correcta es: 12

Pregunta 3

Finalizado

Puntúa como 1,00



Marcar pregunta

Enunciado de la pregunta

¿Cómo reordenarías el código del programa 2-3reordering.asm sin modificar su semántica, es decir, que siga haciendo lo mismo, para eliminar la detención por el riesgo RAW con el menor número de instrucciones recolocadas?

Nuevo Orden:

1, 2, 4, 6, 5, 3, 6, 7

Retroalimentación

Dos opciones:

main:

```
daddi r1, r2, 5
xor r1, r5, r3
dadd r6, r4, r2
dsub r8, r3, r2
ld r8, 120(r1)
and r8, r4, r1
```

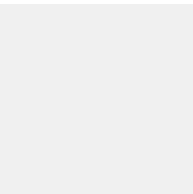
main:

```
daddi r1, r2, 5
xor r1, r5, r3
dsub r8, r3, r2
dadd r6, r4, r2
ld r8, 120(r1)
and r8, r4, r1
```

Pregunta 4

Finalizado

Puntúa como 1,00



Marcar pregunta

Enunciado de la pregunta

¿Qué instrucciones tienen dependencia de datos RAW? ¿De qué instrucciones dependen? ¿Qué registro crea esa dependencia?

7, 8 y 9 dependen de 6. R1

Retroalimentación

La instrucción daddi r2, r1, 1 depende de la instrucción ori r1, r0, 2 a través del registro r1.

La instrucción ori r3, r1, 7 depende de la instrucción ori r1, r0, 2 a través del registro r1.

La instrucción dadd r6, r4, r4 depende de la instrucción ld r4, var1(r1) a través de r4.

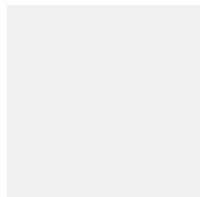
La instrucción ori r5, r4, 8 depende de la instrucción ld r4, var1(r1) a través de r4

La instrucción beqz r5, main depende de la instrucción ori r5, r4, 8 a través de r5

Pregunta 5

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

¿Cuántos ciclos de reloj requiere para su ejecución?

Respuesta:

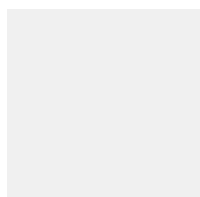
Retroalimentación

La respuesta correcta es: 17

Pregunta 6

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

¿Cuántos ciclos de detención ha sufrido el programa?

Respuesta:

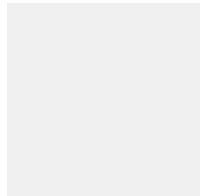
[Retroalimentación](#)

La respuesta correcta es: 5

Pregunta 7

Incorrecta

Puntúa 0,00 sobre 1,00



Marcar pregunta

[Enunciado de la pregunta](#)

¿Cuál es el CPI resultante?

Respuesta:

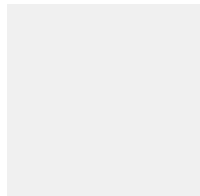
[Retroalimentación](#)

La respuesta correcta es: 2,125

Pregunta 8

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

[Enunciado de la pregunta](#)

¿Cuántos ciclos de reloj requiere para su ejecución?

Respuesta:

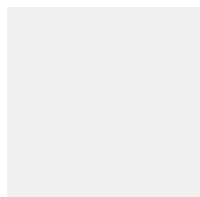
[Retroalimentación](#)

La respuesta correcta es: 13

Pregunta 9

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

¿Cuántos ciclos de detención ha sufrido el programa?

Respuesta:

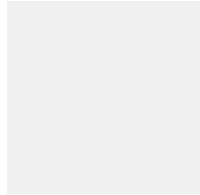
Retroalimentación

La respuesta correcta es: 1

Pregunta 10

Finalizado

Puntúa como 1,00



Marcar pregunta

Enunciado de la pregunta

¿Qué rutas de reenvío se han activado? Cada ruta de reenvío debe especificarse de acuerdo al siguiente ejemplo: Salida MEM de addi, r8, r9, 10 -> Entrada MEM de xor r5, r9, r6

Salida EX de ori r1, r0, 4 -> Entrada EX de daddi r2, r1, 1

Salida MEM de ori r1, r0, 4 -> Entrada EX de ori r3, r1, 7

Salida MEM de ld r4, var1(r1) -> Entrada EX de dadd r6, r4, r4

Salida EX de ori r5, r4, 8 -> Entrada ID de beqz r5, main

Retroalimentación

Hay que tener en cuenta que algunas dependencias RAW no requieren reenvío pues las instrucciones dependientes están muy separadas.

Salida EX ori r1, r0, 2 -> Entrada EX daddi r2, r1, 1

Salida MEM ori r1, r0, 2 -> Entrada EX ori r3, r1, 7

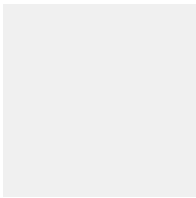
Salida MEM ld r4, var1(r1) -> Entrada EX dadd r6, r4, r4

Salida EX ori r5, r4, 8 -> Entrada ID beqz r5, main

Pregunta 11

Incorrecta

Puntúa 0,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

¿Cuál es el CPI resultante?

Respuesta:

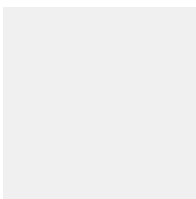
Retroalimentación

La respuesta correcta es: 1,625

Pregunta 12

Finalizado

Puntúa como 1,00



Marcar pregunta

Enunciado de la pregunta

¿Qué instrucciones tienen dependencia de datos WAW o WAR? ¿Qué registro crea esa dependencia?

Instrucción 5 con 4 (R1) y 6 con 3 (R5)

Retroalimentación

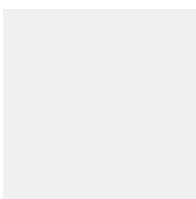
Para las dependencias WAW hay que fijarse en instrucciones que escriban sobre el mismo registro. Para las dependencias WAR hay que fijarse en instrucciones que lean en un registro que más tarde es escrito por otra instrucción.

La instrucción dadd r3, r5, r5 tiene una dependencia WAW de la instrucción dmul r3, r1, r1 a través del registro r3.

Pregunta 13

Incorrecta

Puntúa 0,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

¿Cuántos ciclos de reloj requiere para su ejecución?

Respuesta:

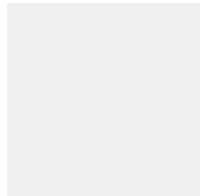
Retroalimentación

La respuesta correcta es: 12

Pregunta 14

Finalizado

Puntúa como 1,00



Marcar pregunta

Enunciado de la pregunta

¿Qué tipo de detenciones ha sufrido y de qué duración?

Detención de 4 ciclos por dependencia WAW

Retroalimentación

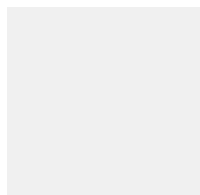
La dependencia estructural es debida a que las instrucciones `dmul r3, r1, r1` y `dadd r5, r5, r5` intentan acceder a la vez a la etapa MEM y ésta no está replicada.

Una detención por dependencia de datos WAW de 4 ciclos de duración (la Str es en realidad parte de la WAW).

Pregunta 15

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

¿Cuál es el CPI del programa ignorando el transitorio inicial?

Respuesta:

Retroalimentación

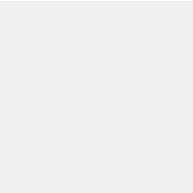
Se obtiene dividiendo el número total de ciclos (12 ciclos) menos el transitorio inicial (4 ciclos) = 8, entre el número de instrucciones ejecutadas, 4.

La respuesta correcta es: 2

Pregunta 16

Finalizado

Puntúa como 1,00



Marcar pregunta

Enunciado de la pregunta

¿Qué explicación tiene un CPI tan alto y alejado del teórico $CPI=1$?

Al existir la orden `dmul` inmediatamente antes de la `dadd`, hace que `dadd` tenga que esperar varios ciclos para que `R3` tenga finalmente el valor debido.

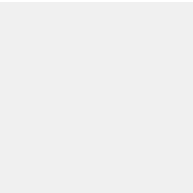
Retroalimentación

La detención `WAW` de cuatro ciclos y la instrucción `dmul` de 5 ciclos

Pregunta 17

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

¿Cuántos ciclos de reloj requiere para su ejecución?

Respuesta:

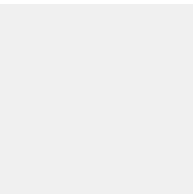
Retroalimentación

La respuesta correcta es: 11

Pregunta 18

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

¿Cuál es el CPI del programa ignorando el transitorio inicial?

Respuesta:

Retroalimentación

Número de ciclos (11) menos transitorio inicial (4) = 7 ciclos

Número de instrucciones = 4

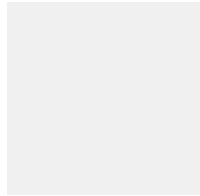
$CPI = 7/4 = 1,75$

La respuesta correcta es: 1,75

Pregunta 19

Incorrecta

Puntúa 0,00 sobre 1,00



Marcar pregunta

[Enunciado de la pregunta](#)

¿Qué aceleración ha experimentado la ejecución del programa al eliminar el reciclaje del registro?

Respuesta:

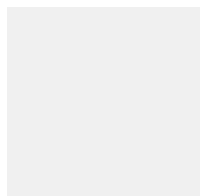
[Retroalimentación](#)

La respuesta correcta es: 1,1428

Pregunta 20

Finalizado

Puntúa como 1,00



Marcar pregunta

[Enunciado de la pregunta](#)

¿Qué registros arquitectónicos cambian de registro físico asociado? ¿Qué nuevos registros físicos tienen asociados?

r1 --> rr33:0

r3 --> rr35:0

r5 --> rr32:0

[Retroalimentación](#)

Instrucción r5, r0, 20

- Etapa ID: r5 --> rr32 0

Instrucción ori r1, r0, 10

- Etapa ID: r1 --> rr33 0
- Etapa WB: r1 --> rr33 1

Instrucción dmul r3, r1, r1

- Etapa ID: r3 --> rr34 0
- Etapa WB: r1 --> rr33 0

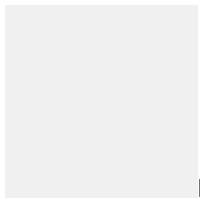
Instrucción dadd r3, r5, r5

- Etapa ID: r3 --> rr35 0
- Etapa ID: r5 --> rr32 1
- Etapa WB: r5 --> rr32 0

Pregunta 21

Finalizado

Puntúa como 1,00



Marcar pregunta

Enunciado de la pregunta

¿Cuántos ciclos de detención sufre la instrucción dadd en su fase de ejecución? ¿A qué son debidos?

dadd sufre 4 ciclos de detención para finalizar después de la orden dmul.

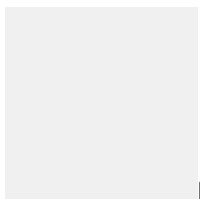
Retroalimentación

Sufre cuatro ciclos debido a una dependencia IOT (In order termination) debido al soporte de excepciones precisas

Pregunta 22

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

¿Cuántos ciclos de reloj requiere la ejecución del programa 2-3branch.asm?

Respuesta:

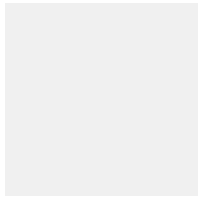
Retroalimentación

La respuesta correcta es: 19

Pregunta 23

Finalizado

Puntúa como 1,00



Marcar pregunta

Enunciado de la pregunta

Identifica las detenciones de control sufridas, indicando los ciclos detención de cada una de ellas y las causas de las mismas.

Ciclo	// Orden	// Causa
3	// beqz r1, endloop	// RAW (r1)
3, 4	// daddi r1, r1, -1	// CNT
7	// xor r1, r1, r1	// CNT
9	// daddi r1, r1, -1	// CNT
14	// daddi r1, r1, -1	// CNT

Retroalimentación

Se produce un total de 5 detenciones de control:

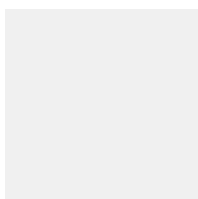
- La instrucción daddi r1, r1, -1 que sufre un ciclo de detención (ciclo 4) la primera vez que se ejecuta, un ciclo de detención la segunda vez que se ejecuta (ciclo 9) y otro ciclo de detención la última vez que se ejecuta (ciclo 14). En total esta instrucción sufre 3 ciclos de detención. Todas ellas son debidas al salto condicional beqz r1, endloop
- La instrucción xor r1, r1, r1 que sufre un ciclo de detención la primera vez que se ejecuta (ciclo 7) y otro ciclo de detención (ciclo 12) la segunda vez que se ejecuta. En total esta instrucción sufre 2 ciclos de detención. Todas ellas son debidas al salto incondicional j loop

A parte de estas detenciones de control también se genera una detención de tipo RAW entre la instrucción beqz r1, endloop y la instrucción ori r1, r0, 2

Pregunta 24

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

¿Cuántos ciclos de reloj requiere la ejecución del programa anterior?

Respuesta:

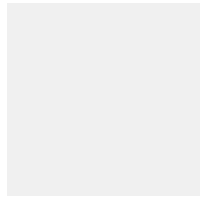
Retroalimentación

La respuesta correcta es: 17

Pregunta 25

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

¿Qué aceleración ha introducido el predictor con respecto al caso de no emplear predicción?

Respuesta:

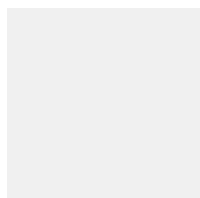
Retroalimentación

La respuesta correcta es: 1,12

Pregunta 26

Incorrecta

Puntúa 0,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

¿Cuántos aciertos y cuántos fallos ha experimentado el predictor?

Respuesta:

Retroalimentación

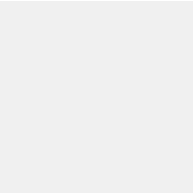
2 aciertos y 1 fallo

La respuesta correcta es: 2 aciertos y 1 fallo

Pregunta 27

Incorrecta

Puntúa 0,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

¿Cuántos ciclos de detención han sido causados por la instrucción de salto incondicional?

Respuesta:

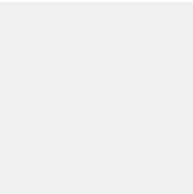
Retroalimentación

La respuesta correcta es: 2

Pregunta 28

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

¿Cuántos ciclos de reloj requiere la ejecución del programa 2-3branch.asm?

Respuesta:

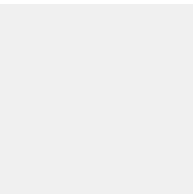
Retroalimentación

La respuesta correcta es: 16

Pregunta 29

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

¿Qué aceleración ha introducido el predictor con respecto al caso de no emplear predicción?

Respuesta:

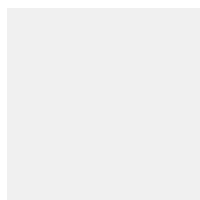
Retroalimentación

La respuesta correcta es: 1,1875

Pregunta 30

Incorrecta

Puntúa 0,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

¿Cuántos aciertos y cuántos fallos ha experimentado el predictor?

Respuesta:

Retroalimentación

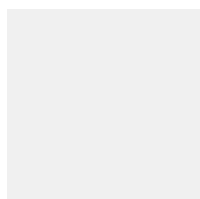
2 aciertos y 1 fallo

La respuesta correcta es: 2 aciertos y 1 fallo

Pregunta 31

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

¿Cuántos ciclos de detención han sido causados por la instrucción de salto incondicional?

Respuesta:

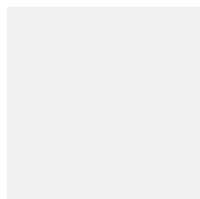
Retroalimentación

La respuesta correcta es: 1

Pregunta 32

Finalizado

Puntúa como 1,00



Marcar pregunta

Enunciado de la pregunta

Una vez se simula el ciclo 10 observarás que la instrucción de salto incondicional no produce una detención de control. ¿Cuál es la razón?

Se debe a que es un salto incondicional y no es el primero.

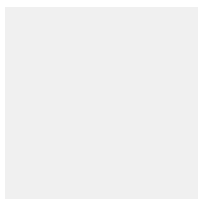
Retroalimentación

Se ha producido un acierto y se corresponde a un salto incondicional, por tanto, la dirección que se escribe en el PC es la que figura como dirección de salto en la tabla de predicción (BHT). Tanto la comprobación de la tabla como la escritura del PC se producen en la etapa IF, por lo que no se produce ninguna detención.

Pregunta 33

Finalizado

Puntúa como 1,00



Marcar pregunta

Enunciado de la pregunta

Simula un ciclo más hasta completar el ciclo de reloj 11. Se produce un fallo de predicción. ¿Cuál es la razón?

Anteriormente ese salto nunca se había tomado, por lo tanto la predicción indica hacia que tampoco se va a tomar, sin embargo, esta vez sí se toma.

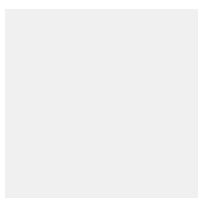
Retroalimentación

La razón es que el bit de estado es 01, lo que se corresponde con weak not taken, es decir se predice que no va a haber salto. Sin embargo, esta predicción es incorrecta, ya que en ese momento r1 vale 0 y por tanto se debe ejecutar el salto.

Pregunta 34

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

¿Cuántos ciclos necesita el programa 2-3loop.asm para ejecutarse?

Respuesta:

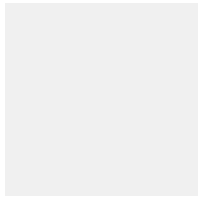
Retroalimentación

La respuesta correcta es: 75

Pregunta 35

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

¿Cuántos ciclos necesita el programa 2-3loop-unroll.asm para ejecutarse?

Respuesta:

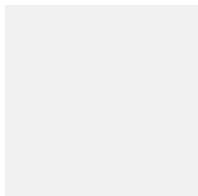
Retroalimentación

La respuesta correcta es: 39

Pregunta 36

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

¿Cuál es la aceleración de la versión con el bucle desenrollado respecto de la versión inicial?

Respuesta:

Retroalimentación

$$\text{Aceleración} = (\text{Ciclos 2-3loop.asm}) / (\text{Ciclos 2-3loop-unroll.asm})$$

La respuesta correcta es: 1,92

Comenzado el lunes, 25 de octubre de 2021, 18:09

Estado Finalizado

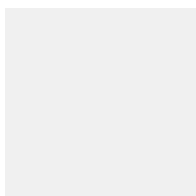
Finalizado en lunes, 25 de octubre de 2021, 18:52

Tiempo empleado 42 minutos 41 segundos

Pregunta 1

Finalizado

Puntúa como 1,00



Marcar pregunta

Enunciado de la pregunta

¿Qué ha ocurrido al intentar deshabilitar las interrupciones en Linux al ejecutar el programa 2-4priv? ¿Cuál es la razón?

Error de segmentación

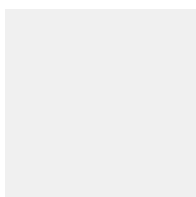
Retroalimentación

Se ha producido una excepción. La razón es que la instrucción de deshabilitar las interrupciones es una instrucción privilegiada y no puede ejecutarla una tarea.

Pregunta 2

Finalizado

Puntúa como 1,00



Marcar pregunta

Enunciado de la pregunta

¿Qué ha ocurrido al ejecutar el programa 2-4mem1? ¿Cuál es la razón?

Error de segmentación

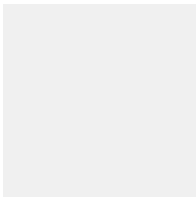
Retroalimentación

Se ha generado una excepción y el programa termina de forma abrupta. La razón es que ha intentado acceder a una dirección que no pertenece a ninguno de los rangos de direcciones que tiene asignados.

Pregunta 3

Incorrecta

Puntúa 0,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

¿Qué instrucción da lugar a la excepción al ejecutar el programa 2-4mem1?

Respuesta:

Retroalimentación

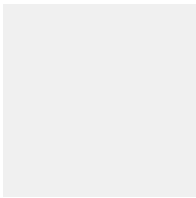
La primera instrucción que escribe en una dirección de memoria no válida.

La respuesta correcta es: `p = 0x1234;`

Pregunta 4

Finalizado

Puntúa como 1,00



Marcar pregunta

Enunciado de la pregunta

¿Por qué crees que la instrucción que asigna la dirección de memoria al puntero *p* en el programa 2-4mem1 no genera una excepción?

Porque esa dirección no está protegida y sí se puede usar.

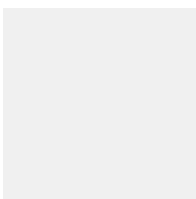
Retroalimentación

La instrucción que modifica *p* escribe sobre una variable del programa, por lo que si estuviese en memoria tendría una dirección válida y por lo tanto no podría dar lugar a excepción por acceso a una dirección no válida.

Pregunta 5

Correcta

Puntúa 1,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

¿Qué instrucción crees que dará lugar a una excepción en el programa 2-4mem2?

Respuesta:

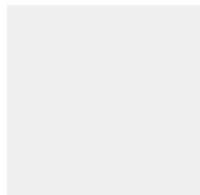
[Retroalimentación](#)

La respuesta correcta es: printf("%d\n", p);

Pregunta 6

Incorrecta

Puntúa 0,00 sobre 1,00



Marcar pregunta

[Enunciado de la pregunta](#)

¿Cuál es el tipo de VMM de VirtualBox?

Seleccione una:



a.

Tipo 1



b.

Tipo 2

[Retroalimentación](#)

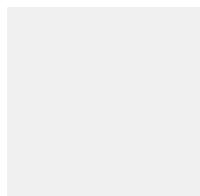
Respuesta incorrecta.

La respuesta correcta es: Tipo 2

Pregunta 7

Incorrecta

Puntúa 0,00 sobre 1,00



Marcar pregunta

[Enunciado de la pregunta](#)

¿Se está utilizando VT-x?

Seleccione una:



Verdadero



Falso

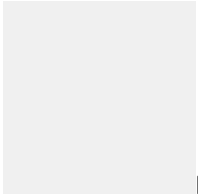
[Retroalimentación](#)

La respuesta correcta es 'Verdadero'

Pregunta 8

Incorrecta

Puntúa 0,00 sobre 1,00



Marcar pregunta

[Enunciado de la pregunta](#)

¿Qué obtienes al ejecutar `dmesg | grep paravirt`?

Respuesta:

Arrancando kernel paravirtualizado en KVM

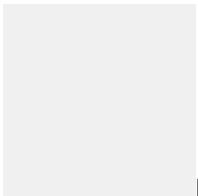
[Retroalimentación](#)

La respuesta correcta es: [0.000000] Booting paravirtualized kernel on KVM

Pregunta 9

Incorrecta

Puntúa 0,00 sobre 1,00



Marcar pregunta

[Enunciado de la pregunta](#)

¿Cuántos núcleos tiene el procesador físico?

Respuesta:

4

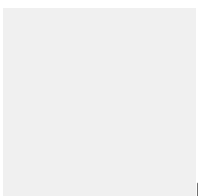
[Retroalimentación](#)

La respuesta correcta es: 2

Pregunta 10

Incorrecta

Puntúa 0,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

¿Qué porcentaje de CPU ocupa VirtualBox y qué porcentaje el proceso CpuBurn, (empleando 2 procesadores)?

Respuesta:

Retroalimentación

El porcentaje que ocupa cada uno depende del número de cores del procesador.

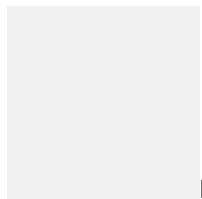
En general ocuparan $1/n^{\text{a}}$ cores $2/n^{\text{o}}$ cores

La respuesta correcta es: 25,5

Pregunta 11

Incorrecta

Puntúa 0,00 sobre 1,00



Marcar pregunta

Enunciado de la pregunta

¿Qué porcentaje de CPU ocupa VirtualBox y qué porcentaje el proceso CpuBurn, (empleando 1 procesador)?

Respuesta:

Retroalimentación

El porcentaje que ocupa cada uno depende del número de cores físicos del procesador:

En general $1/n^{\text{o}}$ cores y $1/n^{\text{o}}$ de cores

La respuesta correcta es: 25,25