

**EL PROGRAMA UNIVERSITARIO DE LA IMAGINACIÓN**

**Laboratorio RVfpga 14**

## **Riesgos estructurales**

# INTRODUCCIÓN

En los siguientes tres laboratorios, Laboratorios 14-16, analizamos los **peligros de las tuberías** . Como lo explicaron D. Patterson y J. Hennesy en el Capítulo 4, Sección 5 de su reciente libro RISC-V (Computer Organization and Design RISC-V Edition, Patterson & Hennessy, © Morgan Kaufmann 2017) [PaHe]: Existen situaciones en la canalización cuando la siguiente instrucción no puede ejecutarse en el siguiente ciclo de reloj. Estos eventos se denominan peligros. Se producen tres tipos diferentes de peligros: **peligros estructurales, peligros** de **datos** y peligros de **control** .

Como explican Patterson y Hennessy en [PaHe], **los riesgos estructurales** ocurren cuando una instrucción no puede ejecutarse porque el hardware no admite la combinación de instrucciones que están configuradas para ejecutarse. En este laboratorio, analizamos los peligros estructurales en el procesador SweRV EH1.

En el Laboratorio 15 analizamos el segundo tipo de peligro, **los peligros de datos** , en el procesador SweRV EH1. Tal como lo explicaron Hennessy y Patterson en la sexta edición de su libro "Arquitectura informática: un enfoque cuantitativo" [ HePa ]: los riesgos de datos ocurren cuando la canalización cambia el orden de los accesos de lectura/escritura a los operandos de modo que el orden difiere del orden visto mediante la ejecución secuencial de instrucciones en un procesador no canalizado.

Finalmente, el tercer tipo de peligro se llama **peligro de control** . Como explican S. Harris y D. Harris en la Sección 7.5.3 de su libro " *Diseño digital y arquitectura informática: Edición RISC-V* " (que llamamos DDCARV) , se produce un *riesgo de control* cuando la decisión de qué instrucción buscar a continuación no se ha realizado en el momento en que tiene lugar la recuperación. En el Laboratorio 16 analizamos los riesgos de control en el procesador SweRV EH1.

# RIESGOS ESTRUCTURALES EN SweRV EH1

En esta sección, ilustramos dos casos de riesgos estructurales que pueden ocurrir en el procesador SweRV EH1. Cada uno se resuelve de una manera diferente, lo que da como resultado una compensación de rendimiento-costo diferente.

Para ilustrar la primera situación, creamos un ejemplo en la Sección 2.A basado en la instrucción de multiplicación de enteros ( mul ). Al mismo tiempo, describimos la ejecución de esta instrucción en SweRV EH1, que aún no hemos analizado en laboratorios anteriores. Recuerde de las Secciones 3 y 4 de la GSG que esta instrucción pertenece a la Extensión RISC-V M (Extensión estándar para multiplicación y división de enteros), que es compatible con SweRV EH1. Para ejecutar esta instrucción, el procesador SweRV EH1 implementa una unidad de multiplicación multiciclo canalizada (es decir, un multiplicador que está canalizado y necesita más de un ciclo para calcular el resultado) en la canalización Multiply (consulte la Figura 4 de la práctica de laboratorio 11), dividida en tres etapas: M1, M2 y M3.

Específicamente, en este ejemplo, dos instrucciones mul llegan a la etapa de decodificación en el mismo ciclo. Debido a que SweRV EH1 solo tiene una unidad de multiplicación, se produce un riesgo estructural, ya que el procesador "no admite la combinación de instrucciones que están configuradas para ejecutarse" [PaHe] . En un procesador que utilice un multiplicador multiciclo no canalizado, la segunda instrucción mul tendría que esperar hasta que la primera terminara su ejecución, lo que tendría un impacto importante en el rendimiento. Sin embargo, como se indicó anteriormente, el multiplicador utilizado en SweRV EH1 está canalizado, por lo que la segunda instrucción mul solo se retrasa un ciclo y comienza a ejecutarse tan pronto como la primera instrucción de multiplicación finaliza la primera etapa de la multiplicación (M1) y procede a la segunda etapa (M2). Esta solución tiene un impacto moderado en el costo del hardware (una estructura con tubería es más costosa que una sin tubería), pero resuelve el riesgo estructural con un bajo impacto en el rendimiento (solo un ciclo).

En el segundo ejemplo (Sección 2.B), tres instrucciones llegan a la etapa Writeback en el mismo ciclo, una de ellas es una carga sin bloqueo ejecutada varios ciclos antes . En principio, debido a que SweRV EH1 es un núcleo superescalar de 2 vías, no sería posible completar tres instrucciones en el mismo ciclo; sin embargo, como mostramos en el laboratorio 11, el archivo de registro de SweRV EH1 tiene un tercer puerto de escritura, lo que evita el riesgo estructural en esta situación. Esta solución tiene un alto impacto en el costo del hardware debido al puerto de archivo de registro adicional, pero resuelve este riesgo estructural sin pérdida de rendimiento.

**div simultáneas en la etapa de decodificación:** además de estos dos ejemplos, en el apéndice al final de esta práctica de laboratorio ilustramos otro ejemplo basado en instrucciones de división. Aunque este ejemplo no ilustra estrictamente un riesgo estructural, sigue siendo muy interesante y le recomendamos que lo analice también.

1. **instrucciones mul simultáneas en la etapa de decodificación**

El RISC-V M Extension incluye, entre otras, la instrucción mul . Esta instrucción realiza la multiplicación de rs1 por rs2 y coloca los bits inferiores en el registro de destino ( rd ). La instrucción en lenguaje máquina para mul es la siguiente (ver Apéndice B de [ DDCARV ]):

0000001 | rs2 | rs1 | 000 | rd | 0110011

**TAREA (OPCIONAL):** puede realizar un estudio similar para la instrucción mul como el realizado en el laboratorio 12 para las instrucciones aritmético-lógicas: ver el flujo de la instrucción a través de las etapas de la canalización, analizar los bits de control (recuerde de la Sección 4 de SweRVref que hay un tipo de estructura específico para la instrucción mul llamada mul\_pkt\_t , y hay una señal definida en el módulo **dec\_decode\_ctl** llamada mul\_p ), etc.

La unidad de multiplicación se implementa en el módulo **exu\_mul\_ctl** ( *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/exu/exu\_mul\_ctl.sv* ). Como se mencionó anteriormente, esta unidad está canalizada y requiere 3 ciclos para calcular el resultado. El uso de un multiplicador canalizado, a diferencia de uno no canalizado, reduce la pérdida de rendimiento debido a riesgos estructurales.

**TAREA (OPCIONAL):** Inspeccione el código Verilog de **exu\_mul\_ctl** y vea cómo se calcula la multiplicación. Recuerde que RISC-V incluye 4 instrucciones de multiplicación ( mul , mulh , mulhsu y mulhu ), y todas ellas deben ser soportadas por el hardware.

Como ejercicio opcional, puede reemplazar la unidad de multiplicación con su propia unidad o una de Internet.

El ejemplo de la Figura 1 ejecuta dos instrucciones mul contenidas dentro de un bucle que se repite para iteraciones 0xFFFF (es decir, 65.535 en decimal). Las instrucciones múltiples están resaltadas en rojo en la figura. En este caso, las instrucciones mul están rodeadas por varias instrucciones nop para aislar cada iteración entre sí. Como de costumbre, el programa no hace nada útil y solo pretende ilustrar *los peligros estructurales* debido a las instrucciones múltiples .

|  |
| --- |
| .globl Test\_Assembly Test\_Assembly:  li t2, 0xFFFF  li t3, 0x3 li t4, 0x2li t5, 0x2li t6, 0x2  REPETIR: beq t2, zero, OUT # ¿Permanecer en el bucle? INSERT\_NOPS\_9 **mul t0, t3, t4** # t0 = t3 \* t4 **múltiple t1, t5, t6** # t1 = t5 \* t6 INSERTAR\_NOPS\_9 agregar t2, t2, -1 agregar t0, cero, cero agregar t1, cero, cero j REPETIR:  .final |

Figura 1 . Ejemplo con dos instrucciones mul consecutivas

carpeta *[RVfpgaPath]/RVfpga/Labs /Lab14/MUL\_Instruction* proporciona el proyecto PlatformIO para que pueda analizar, simular y modificar el programa como desee. La estructura del proyecto se basa en la proporcionada en el Lab 11 para el uso de los contadores de rendimiento: contiene un archivo *.c* que inicializa, detiene e imprime el valor de los contadores deseados y un archivo *.S* que contiene el programa ensamblador que queremos probar (en este caso, el bucle con las dos instrucciones mul ) y cuál se invoca desde el archivo *.c .*

Abra el proyecto en PlatformIO, constrúyalo y abra el archivo de desensamblado (disponible en *[RVfpgaPath]/RVfpga/Labs /Lab14/MUL\_Instruction/.pio/build/swervolf\_nexys/firmware.dis* ). Observe que las instrucciones mul se colocan en las direcciones 0x000001e8 y 0x000001ec.

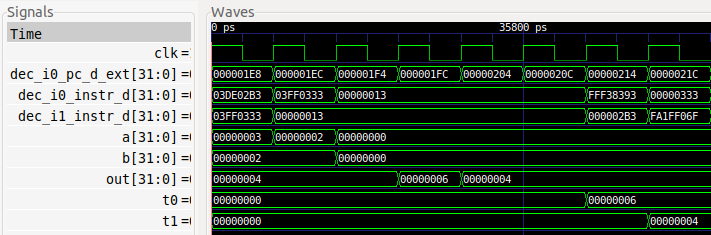
**0x000001e8: 03de02b3 multi0,t3,t4**

**0x000001ec: 03ff0333 multi1,t5,t6**

**TAREA :** Verificar que este par de 32 bits ( 0x03de02b3 y 0x03ff0333 ) correspondan a las instrucciones mul t0,t3,t4 y mul t1,t5,t6 en la arquitectura RISC-V.

La Figura 2 muestra la simulación del programa de la Figura 1 en la segunda iteración del bucle .

**mul t1,t5,t6**



**i**

**mul t0,t3,t4**

**i+6**

**i+3**

**i+1**

Figura 2 . Simulación de Verilator del ejemplo de la Figura 1

**TAREA :** replicar la simulación de la Figura 2 en su propia computadora y analizarlo más de cerca. Puede usar el archivo *.tcl [RVfpgaPath]/RVfpga/Labs /Lab14/MUL\_Instruction/test.tcl*

Analice la forma de onda de la Figura 2 . Los valores resaltados en rojo corresponden a diferentes señales relacionadas con las instrucciones mul a medida que atraviesan la tubería.

* **Ciclo i** : Las dos instrucciones mul llegan en el mismo ciclo a la etapa de Decodificación. Un riesgo estructural impide que la segunda instrucción mul ( dec\_i1\_instr\_d = 0x03ff0333) avance a la siguiente etapa, dado que la primera instrucción mul ( dec\_i0\_instr\_d = 0x03de02b3) está programada para esa unidad.
* **Ciclo i+1** : la primera instrucción mul se ejecuta en la primera etapa del multiplicador canalizado (M1), mientras que la segunda instrucción mul espera en la etapa de decodificación.
* **Ciclo i+2** : La primera instrucción mul se ejecuta en la segunda etapa del multiplicador segmentado (M2) y la segunda mul se ejecuta en la primera etapa (M1).
* **Ciclo i+3** : La primera instrucción mul llega a EX3, cuando se produce el resultado de la multiplicación ( out = 0x6 para la primera instrucción mul ).
* **Ciclo i+4** : La segunda instrucción mul llega a EX3, cuando se produce el resultado de la multiplicación ( out = 0x4 para la segunda instrucción mul ).
* **Ciclo i+6** : El archivo de registro se actualiza con el resultado de la primera instrucción mul ( t0 = 0x6).
* **Ciclo i+7** : El archivo de registro se actualiza con el resultado de la segunda instrucción mul ( t1 = 0x4).

La Figura 3 ilustra el flujo de las instrucciones del ejemplo de la Figura 1 a través de la canalización SweRV EH1. **D** representa la etapa Decode, **A** para la etapa Align, **C** para la etapa Commit y **WB** para la etapa Writeback. Cuando se decodifica la primera instrucción mul (ciclo i), la mayoría de las instrucciones posteriores se detienen en su etapa actual (marcada en la figura con el sufijo *st* ) y se insertan burbujas. En el ciclo siguiente (i+1) se reanudan las instrucciones (tenga en cuenta que la segunda instrucción mul se ha movido de la Vía 1 a la Vía 0, y la Vía 1 contiene la siguiente instrucción, que es un nop ). En el ciclo i+2 la primera instrucción mul está en la segunda etapa de ejecución (M2) y la segunda instrucción mul está en la primera etapa de ejecución (M1). En los ciclos i+5 e i+6, las dos instrucciones mul vuelven a escribir su resultado en el archivo de registro, que se puede ver actualizado en la Figura 2 en los ciclos i+6 e i+7.



Figura 3 . Ejecución del código de ejemplo de la Figura 1

**TAREA :** Compare la ilustración de la Figura 3 con la simulación de la Figura 2 centrándose en las dos instrucciones mul . Específicamente, analice cómo se asignan las dos instrucciones a las dos formas en las etapas de alineación y decodificación y cómo avanzan a través de la canalización.

- En el módulo **ifu\_aln\_ctl** (etapa Align) las dos instrucciones se asignan a las siguientes señales siempre que sea posible:

- Vía 0: ifu\_i0\_instr

- Vía 1: ifu\_i1\_instr

- En el módulo **dec\_ib\_ctl ,** las dos instrucciones se almacenan en el búfer de Alinear a Decodificar. Tenga en cuenta que, en algunos casos, las instrucciones pueden detenerse en estos búferes y reasignarse de una manera diferente:

- Vía 0: ifu\_i0\_instr 🡪 dec\_i0\_instr\_d

- Vía 1: ifu\_i1\_instr 🡪 dec\_i1\_instr\_d

- En el módulo **dec\_decode\_ctl** (etapa Decode) se programan las dos instrucciones a los pipes correspondientes siempre que sea posible. Una vez que se envían, continúan a través de las tres etapas de ejecución, la etapa de confirmación y la etapa de reescritura:

- Vía 0: i0\_inst\_e1 🡪 i0\_inst\_e2 🡪i0\_inst\_e3 🡪i0\_inst\_e4 🡪i0\_inst\_wb

- Vía 1: i1\_inst\_e1 🡪 i1\_inst\_e2 🡪i1\_inst\_e3 🡪i1\_inst\_e4 🡪i1\_inst\_wb

Proporcionamos un archivo *.tcl* llamado *[RVfpgaPath]/RVfpga/Labs /Lab14/MUL\_Instruction/test\_AssignmentWays.tcl* que incluye todas estas señales.

**TAREA : elimine** las instrucciones nop incluidas dentro del bucle de la Figura 1 y mida diferentes eventos (ciclos, instrucciones/multiplicaciones comprometidas, etc.) utilizando los contadores de rendimiento disponibles en SweRV EH1, como se explica en el laboratorio 11. ¿Es la cantidad de ciclos la esperada ? después de analizar la simulación de figura 2 ? Justifica tu respuesta.

Ahora reordene el código dentro del ciclo tratando de alcanzar el rendimiento ideal. Justificar los resultados obtenidos en el código original y en el reordenado.

**TAREA (OPCIONAL, INTERESANTE):** Carpeta *[RVfpgaPath]/RVfpga/Labs /Lab14/MUL\_Instr\_Accumul\_C-Lang* proporciona el proyecto PlatformIO de un programa C que acumula la resta de dos multiplicaciones dentro de un bucle.

* Analizar el programa C.
* Realice una simulación e inspeccione una iteración aleatoria del ciclo. Tenga en cuenta que el programa C se compila sin optimizaciones.
* Mida diferentes eventos (ciclos, instrucciones/multiplicaciones comprometidas, etc.) utilizando los contadores de rendimiento disponibles en Swe RV EH1, como se explica en el laboratorio 11.

¿Es el número de ciclos el esperado después de analizar la simulación de la Figura 2 ? Justifica tu respuesta.

* Cree un programa análogo en ensamblador RISC-V y compárelo con la versión C. Reordenar las instrucciones tratando de obtener el mejor IPC posible.
* Deshabilite la extensión **M** RISC-V en el programa C y compare los resultados con el programa original. Para ello, modifique la siguiente línea en el archivo *platformio.ini* from :

build\_flags = -Wa,-march=rv32 **ima** -march=rv32 **ima**

A:

build\_flags = -Wa,-march=rv32 **ia** -march=rv32 **ia**

Esto evita el uso de las instrucciones de la extensión M RISC-V y las emula usando otras instrucciones en su lugar.

**TAREA :** Modifique el programa de la Figura 1 , reemplazando las dos instrucciones mul por dos instrucciones lw para el DCCM. Debe observar un peligro estructural análogo al analizado en esta sección y resuelto de manera similar.

1. **Tres instrucciones simultáneas que se ejecutan en la etapa Writeback(OPCIONAL todo el apartado incluyendo sus tareas)**

SweRV EH1 es un procesador superescalar de 2 vías (hemos discutido brevemente esta característica en el GSG y en laboratorios anteriores, y la analizaremos con más detalle en el Laboratorio 17). Esto significa que en este procesador se pueden ejecutar dos instrucciones por ciclo. En una situación en la que llegaran tres instrucciones a la misma etapa en el mismo ciclo, podría ocurrir potencialmente un peligro estructural. Puede parecer que tal situación no es posible dada la estructura de SweRV EH1, sin embargo, hay un caso específico en el que esto puede suceder:

* La memoria DDR2 externa tiene una latencia moderada que obliga a detener las instrucciones de carga. Cuando la carga finalmente recibe sus datos de la memoria, pasa a la etapa de reescritura, donde escribe el valor de lectura en el archivo de registro (supongamos que esta reescritura ocurre en el ciclo ***i*** ) .
* Si las cargas no son bloqueantes (es decir, mientras la carga está esperando que lleguen los datos de la memoria, el procesador continúa ejecutando instrucciones que no dependen de esos datos), puede suceder que otras dos instrucciones lleguen a la etapa Writeback en el ciclo ***i*** y también necesita escribir en el archivo de registro (por ejemplo, dos instrucciones de adición ).
* En esta situación, tres instrucciones estarían tratando de escribir en el archivo de registro en el mismo ciclo (ciclo ***i*** ).

Si el archivo de registro solo tuviera dos puertos de escritura, se produciría un riesgo estructural y una de las tres instrucciones que intentaran escribir tendría que esperar a que el archivo de registro quedara libre. Sin embargo, en SweRV EH1, como mostramos en el laboratorio 11, se implementa un tercer puerto de escritura, lo que permite resolver este riesgo estructural sin paradas y, por lo tanto, sin pérdida de rendimiento.

El ejemplo de la Figura 4 ilustra esta situación. Ejecuta una instrucción lw sin bloqueo seguida de 36 instrucciones de adición contenidas dentro de un bucle que se repite para iteraciones 0xFFFF (es decir, 65.535). La instrucción lw está resaltada en rojo en la figura. Las dos instrucciones add , que llegan a la etapa Writeback en el mismo ciclo que la instrucción lw (ciclo ***i*** ), también están resaltadas. En este caso, no se incluyen las instrucciones de nop . Como de costumbre, el programa no hace nada útil y solo pretende ilustrar el ejemplo de esta sección.

|  |
| --- |
| REPETIR: **lw x28, (x29)** suma x30, x30, -1 suma x1, x1, 1 suma x31, x31, 1 suma x3, x3, 1 suma x4, x4, 1 suma x5, x5, 1 suma x6, x6, 1 suma x7, x7, 1 suma x8, x8, 1 suma x9, x9, 1 suma x10, x10, 1 suma x11, x11, 1 suma x12, x12, 1 suma x13, x13, 1 suma x14, x14, 1 suma x15, x15, 1 suma x16, x16, 1 suma x17, x17, 1 suma x18, x18, 1 suma x19, x19, 1 suma x20, x20, 1 suma x21, x21, 1 suma x22, x22, 1 **suma x23, x23, 1 suma x24, x24, 1** suma x25, x25, 1 suma x26, x26, 1 suma x27, x27, 1 suma x31, x31, 1 suma x3, x3, 1 suma x4, x4, 1 suma x5, x5, 1 sumar x6, x6, 1 sumar x25, x25, 1 sumar x26, x26, 1 sumar x27, x27, 1 bne x30, cero, REPETIR # Repetir el ciclo |

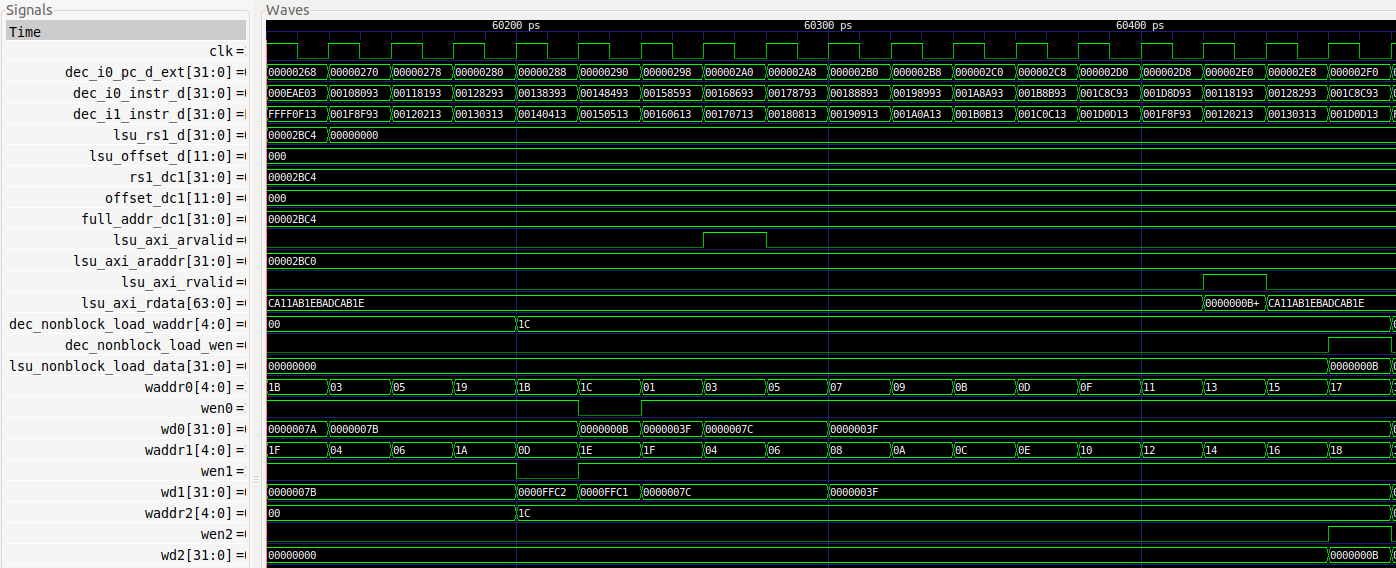
Figura 4 . Ejemplo de una instrucción lw sin bloqueo seguida de 36 instrucciones AL

carpeta *[RVfpgaPath]/RVfpga/Labs /Lab14/LW\_Instruction\_ExtMemory* proporciona el proyecto PlatformIO para que pueda analizar, simular y modificar el programa como desee. La estructura del proyecto se basa en la proporcionada en el Lab 11 para el uso de los contadores de rendimiento: contiene un archivo *.c* que inicializa, detiene e imprime el valor de los contadores deseados y un archivo *.S* que contiene el programa ensamblador que queremos probar (en este caso, el bucle con la instrucción lw que no bloquea ) y que se invoca desde el archivo *.c .*

Como se muestra en la Figura 5 , los datos de 32 bits obtenidos en el módulo **lsu\_bus\_intf** (interfaz de bus) se proporcionan al archivo de registro a través de la señal lsu\_nonblock\_load\_data[31:0] . Además, las señales de control que le indican al archivo de registro dónde escribir esos datos y cuándo escribirlos, que se generaron en la etapa de decodificación y se propagaron a través de los registros de tubería, se proporcionan al archivo de registro a través de las señales dec\_nonblock\_load\_waddr[4:0] y dec\_nonblock\_load\_wen respectivamente. Estas tres señales ingresan al archivo de registro a través del tercer puerto de escritura disponible en esta estructura ( waddr2 , wen2 y wd2 ), como se ilustra en la figura. Recuerde que en la Figura 6 del Laboratorio 11 ilustramos el archivo de registro en detalle.



Figura 5 . Instrucción de carga sin bloqueo que accede a la memoria externa



**lw x28, (x29)**

**add x23, x23, 1**

**add x24, x24, 1**

**i-17**

**Delay due to accessing External Memory. Independent instructions keep executing.**

**i**

**i-5**

**3 register file**

**Write Ports**

**Decode**

**AXI Bus**

**DC1**

Figura 6 . Simulación de Verilator para el ejemplo de la Figura 4

**Three simultaneous writes to the Register File:**

* **lw writes register x28 (0x1C)**
* **add writes register x23 (0x17)**
* **add writes register x24 (0x18)**



Figura 7 . Ejecución del código de ejemplo de la Figura 4

La Figura 6 y la Figura 7 muestran la simulación de Verilator para el programa de la Figura 4 y un diagrama que ilustra la ejecución de este programa para una iteración aleatoria del ciclo.

**TAREA :** Replicar la simulación de la Figura 6 en su propia computadora. Utilice el archivo *test\_NonBlocking.tcl* (proporcionado en *[RVfpgaPath]/RVfpga/Labs/Lab14/LW\_Instruction\_ExtMemory* ). *Acercar* ( ) varias veces y pasar a 60120ps.

Analice la forma de onda de la Figura 6 y el diagrama de la Figura 7 .

* **Ciclo i-17** : La instrucción lw está en la etapa de decodificación.
* **Ciclo i-16** : La dirección de memoria efectiva se computa y se envía a la Memoria Externa a través del Bus AXI. La latencia de la Memoria Externa obliga a la instrucción de carga a esperar varios ciclos para que los datos lleguen al núcleo.
* **Ciclo i-5** : Se decodifican las dos instrucciones de adición en conflicto .
* **Ciclo i** : La instrucción lw y las dos instrucciones add en conflicto pasan a la etapa Writeback, donde todas deben escribir el archivo de registro. Esto es posible gracias a los tres puertos de escritura disponibles en el archivo de registro de SweRV EH1. Tenga en cuenta que los números de registro se muestran en hexadecimal en la simulación. x23 , x24 y x28 (registros 0x17, 0x18 y 0x1c) se están escribiendo.

**TAREA:** Compare la simulación que se muestra en la Figura 6 (carga sin bloqueo) con la simulación que se muestra en la Figura 14 de la práctica de laboratorio 13 (carga con bloqueo). Agregue todas las señales necesarias para la comparación.

**TAREA :** Compare la ilustración de la Figura 7 con la simulación de la Figura 6 que ha replicado en su propia computadora. Agregue señales para ampliar la simulación y profundizar la comprensión, según lo desee.

**TAREA :** Mida diferentes eventos (ciclos, instrucciones/cargas comprometidas, etc.) utilizando los contadores de rendimiento disponibles en SweRV EH1, como se explica en la práctica de laboratorio 11. ¿Es el número de ciclos el esperado después de analizar la simulación de la Figura 6 ? Justifica tu respuesta.

Compare estos resultados con los obtenidos cuando las cargas se configuran como cargas de bloqueo.

# EJERCICIOS

1. **(OPCIONAL, INTERESANTE)** Analice, tanto en simulación como en el tablero, el riesgo estructural que ocurre entre dos instrucciones de memoria consecutivas (puede analizar cualquier combinación de dos instrucciones de memoria consecutivas, como cargas y almacenamiento) que llegan al cauce L/S en el mismo ciclo. Puede usar el proyecto PlatformIO proporcionado en: *[RVfpgaPath]/RVfpga/Labs /Lab14/ TwoConsecutiveLW\_Instructions* .
2. **(OPCIONAL, INTERESANTE)** ( *El siguiente ejercicio se basa en el ejercicio 4.22 del libro “Computer Organization and Design – RISC-V Edition”, de Patterson & Hennessy ([PaHe]).* )

Considere el fragmento del ensamblaje RISC-V a continuación:

interruptor x29, 12(x16)

largo x29, 8(x16)

sub x17, x15, x14

beqz x17, etiqueta

suma x15, x11, x14

subx15, x30, x14

Supongamos que modificamos el procesador SweRV EH1 para que tenga una sola memoria (que maneja tanto instrucciones como datos). En este caso, habrá un riesgo estructural cada vez que un programa necesite obtener una instrucción durante el mismo ciclo en el que otra instrucción accede a los datos.

* 1. Dibuje un diagrama de tubería para mostrar dónde se detendrá el código anterior en esta versión imaginaria del procesador SweRV EH1.
  2. En general, ¿es posible reducir el número de paradas/nops reordenando el código?
  3. ¿Este riesgo estructural debe ser manejado en hardware? Hemos visto que los riesgos de datos se pueden eliminar agregando nops al código. ¿Puedes hacer lo mismo con este riesgo estructural? Si es así, explique cómo. Si no, explica por qué no.

# APÉNDICE A(OPCIONAL, interesante solo sí tiene tiempo e interés ) – DOS INSTRUCCIONES DIV SIMULTÁNEAS EN LA ETAPA DE DESCODIFICACIÓN

Este ejemplo adicional se basa en la instrucción de división de enteros ( div ). Al igual que la instrucción mul , la instrucción div pertenece a la extensión RISC-V M (extensión estándar para multiplicación y división de enteros), que es compatible con SweRV EH1.

La instrucción div realiza la división de enteros con signo de rs1 por rs2 y almacena el resultado en rd . La instrucción en lenguaje máquina para div es (consulte el Apéndice B de [ DDCARV ]):

0000001 | rs2 | rs1 | 100 | rd | 0110011

**TAREA :** puede realizar un estudio similar para la instrucción div como el realizado en el laboratorio 12 para las instrucciones aritmético-lógicas: ver el flujo de la instrucción a través de las etapas de la canalización, analizar los bits de control (recuerde de la Sección 4 de SweRVref que hay un tipo de estructura específico para la instrucción div llamada div\_pkt\_t , y hay una señal definida en el módulo **dec\_decode\_ctl** llamada div\_p ), etc.

Para ejecutar esta instrucción, el procesador SweRV EH1 implementa una unidad de división multiciclo de bloqueo no segmentada en el módulo **exu\_div\_ctl** ( *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/exu/exu\_div\_ctl.sv* ). Esta unidad necesita hasta 34 ciclos para calcular el resultado; sin embargo, dependiendo de las entradas, puede ser mucho más pequeño. La unidad de división envía varias señales al procesador ( div\_stall , finish\_early , finish ) para indicar el estado de una instrucción de división.

**TAREA :** Inspeccione el código Verilog de **exu\_div\_ctl** para comprender cómo se calcula la división. Analice también el efecto de las señales div\_stall , finish\_early y finish . Como ejercicio opcional, reemplace la unidad de división con su propia unidad o una de Internet.

El ejemplo de la Figura 8 ejecuta dos instrucciones div contenidas dentro de un bucle que se repite para iteraciones 0xFFFF (es decir, 65.535 en decimal). Las instrucciones div están resaltadas en rojo en la figura. En este caso, a diferencia de muchos otros ejemplos, las instrucciones nop no son necesarias, ya que las instrucciones div ya están aisladas de cualquier otra instrucción debido a la alta latencia de la unidad de división. Como en los programas de juguetes anteriores que hemos usado, el programa no hace nada útil.

|  |
| --- |
| .globl Test\_Assembly Test\_Assembly:  li t2, 0xFFFF  li t3, 0x8000000 li t4, 0x2li t5, 0x2000000li t6, 0x2  REPETIR: **división t0, t3, t4** # t0 = t3 / t4 **división t1, t5, t6** # t1 = t5 / t6 suma t2, t2, -1 suma t0, cero, cero suma t1, cero, cero bne t2, cero, REPETIR # repite el ciclo  .final |

Figura 8 . Ejemplo de dos instrucciones div consecutivas

carpeta *[RVfpgaPath]/RVfpga/Labs /Lab14/DIV\_Instruction* proporciona el proyecto PlatformIO para que pueda analizar, simular y cambiar el programa como desee. La estructura del proyecto es como la utilizada para la instrucción mul y está basada en la incluida en el Laboratorio 11 para el uso de los Contadores de Rendimiento.

Si abre el proyecto en PlatformIO, lo compila y abre el archivo de desensamblaje (disponible en *[RVfpgaPath]/RVfpga/Labs /Lab14/DIV\_Instruction/.pio/build/swervolf\_nexys/firmware.dis* ) verá que las instrucciones div son colocado en las direcciones 0x000001c0 y 0x000001c4.

**0x000001c0: 03de42b3 divt0,t3,t4**

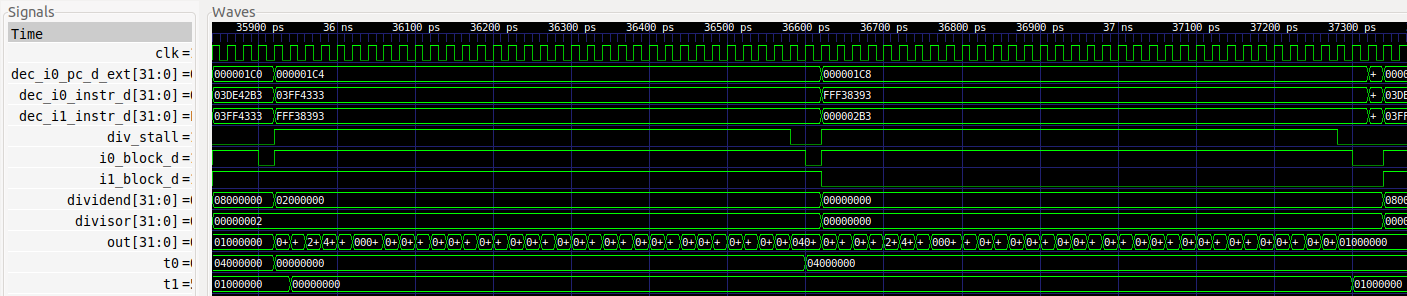
**0x000001c4: 03ff4333 divt1, t5, t6**

**TAREA :** Verificar que este par de 32 bits ( 0x03de42b3 y 0x03ff4333 ) corresponda a las instrucciones div t0,t3,t4 y div t1,t5,t6 en la arquitectura RISC-V.

La Figura 9 muestra la simulación del programa de la Figura 8 en una iteración aleatoria del bucle .

**div t1,t5,t6**

**div t0,t3,t4**



**Execution of** div t1,t5,t6

**Execution of** div t0,t3,t4

Figura 9 . Simulación de Verilator para el ejemplo de la Figura 8

**TAREA :** Replicar la simulación de la Figura 9 en su propio ordenador y analícelo en detalle.

Analice la forma de onda de la Figura 9 . Los valores resaltados en rojo son señales relacionadas con las dos instrucciones div a medida que atraviesan la tubería.

* Las dos instrucciones div llegan a la etapa de decodificación en el mismo ciclo ( dec\_i0\_pc\_d\_ext = 0x000001c0, que es la dirección de instrucción del primer div ). La primera instrucción div (0x03de42b3) está programada para ejecutarse en la unidad de división, por lo que envía el dividendo y el divisor ( dividendo = 0x08000000 y divisor = 0x00000002) a esta unidad. Tenga en cuenta que seleccionamos valores altos para los dividendos para hacer que el tiempo de cálculo de la división sea cercano al máximo (34 ciclos).

Dado que la división se está bloqueando en SweRV EH1, cualquier otra instrucción posterior a la div se detiene. Sin embargo, tenga en cuenta que, incluso si la división no tuviera bloqueo, un peligro estructural debido a que solo tiene un divisor haría que la instrucción de la segunda división se detuviera . Como se explicó en la Sección 2, habría otros enfoques para mejorar el rendimiento, como canalizar el divisor o incluir otro. Sin embargo, dado que la división no es una operación frecuente, en este caso prevalece la reducción del costo del hardware.

* La canalización se detiene durante la ejecución de la primera instrucción div (consulte la señal div\_stall = 1 durante el cálculo de la primera división). También puede ver que tanto Way-0 como Way-1 están bloqueados con las señales i0\_block\_d e i1\_block\_d siendo 1. Además, ahora dec\_i0\_pc\_d\_ext = 0x000001c4, que es la dirección de la segunda instrucción de división, que está estancada en la etapa de decodificación.
* La señal de salida de la unidad de división proporciona el resultado después de 34 ciclos, que se escribe en el registro de destino ( *t0* = 0x04000000). Puede ver cómo el valor de salida cambia en cada ciclo a medida que la operación de división se convierte sucesivamente en el resultado final.
* Cuando se obtiene el resultado, se libera el divisor, se permite que continúe la canalización ( div\_stall = 0) y se programa la segunda instrucción div para la unidad de división. Luego, 34 ciclos después, el resultado de la segunda instrucción div se escribe en el archivo de registro ( *t1* = 0x01000000).

Al igual que en la primera instrucción div , todas las instrucciones posteriores a la segunda deben detenerse debido al divisor de bloqueo. En este caso, sin embargo, aquellas instrucciones que no dependen de *t1* podrían continuar, si se tratara de una división sin bloqueo.

La Figura 10 ilustra el flujo de las instrucciones en el ejemplo de la Figura 8 a través de la canalización SweRV EH1. Cuando se decodifica la primera instrucción div (ciclo i), la segunda div y las instrucciones subsiguientes se bloquean en su etapa actual debido a la división de bloqueo de SweRV EH1 y al riesgo estructural en la unidad de división. (Una instrucción detenida está marcada en la figura con el sufijo *-st* ). Luego, 34 ciclos más tarde (ciclo i+34), la primera instrucción div termina de ejecutarse y escribe el resultado nuevamente en el archivo de registro a través del multiplexor 2:1 que se mostró en la Figura 4 del Laboratorio 11. En el ciclo siguiente (i+35), se reanudan las instrucciones subsiguientes. Luego, en el ciclo 36, la segunda instrucción div comienza a ejecutarse y las instrucciones posteriores se bloquean nuevamente debido a la división de bloqueo de SweRV EH1.



Figura 10 . Ejecución del código de ejemplo de la Figura 8 ( el sufijo *–st* indica una instrucción detenida)

**TAREA :** Compare la ilustración de la Figura 10 y la simulación de la Figura 9 que ha replicado en su propia computadora. Agregue señales para ampliar la simulación y profundizar la comprensión, según lo desee.

**TAREA :** Medir diferentes eventos (ciclos, instrucciones/divisiones comprometidas, etc.) utilizando los contadores de rendimiento disponibles en SweRV EH1, como se explica en el laboratorio 11.

¿Es el número de ciclos el esperado después de analizar la simulación de la Figura 9 ? Justifica tu respuesta.

**TAREA :** Pruebe diferentes dividendos y divisores y vea cómo la cantidad de ciclos para calcular el resultado depende de su valor. Vea el experimento tanto en simulación como con los contadores HW.

**TAREA :** Carpeta *[RVfpgaPath]/RVfpga/Labs /Lab14/DIV\_Instr\_Accumul\_C-Lang* proporciona el proyecto PlatformIO de un programa C que acumula la resta de dos divisiones dentro de un bucle.

* Analizar el programa C.
* Realice una simulación e inspeccione una iteración aleatoria del bucle. Tenga en cuenta que el programa C se compila sin optimizaciones.
* Mida diferentes eventos (ciclos, instrucciones/divisiones comprometidas, etc.) utilizando los contadores de rendimiento disponibles en SweRV EH1, como se explica en el laboratorio 11.

¿Es el número de ciclos el esperado después de analizar la simulación de la Figura 9 ? Justifica tu respuesta.

* Cree un programa análogo en ensamblador RISC-V y compárelo con la versión C.
* Deshabilite la extensión **M** RISC-V en el programa C y compare los resultados con el programa original. Para ello, modifique la siguiente línea en el archivo *plataformaio.ini* de:

build\_flags = -Wa,-march=rv32 **ima** -march=rv32 **ima**

A:

build\_flags = -Wa,-march=rv32 **ia** -march=rv32 **ia**

Esto evita el uso de las instrucciones de la extensión RISC-V M y las emula usando otras instrucciones en su lugar.

**TAREA :** En SweRV EH1, las instrucciones div están bloqueando. Modifique el procesador para permitir instrucciones div sin bloqueo.

Luego agregue un segundo divisor al procesador SweRV EH1, de modo que dos instrucciones div del ejemplo de la Figura 8 puedan ejecutarse en paralelo.