

**EL PROGRAMA UNIVERSITARIO DE LA IMAGINACIÓN**

**Laboratorio RVfpga 15**

## **Riesgos de datos**

# INTRODUCCIÓN

En este laboratorio nos ocupamos de **los riesgos de datos** . Como explicaron Hennessy y Patterson en su sexta edición de "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 por ejecutando secuencialmente instrucciones en un procesador no segmentado. Suponga que la instrucción *i* es seguida por la instrucción *j* en el programa y ambas instrucciones usan el registro *x.* Pueden ocurrir tres tipos de riesgos de datos entre *i* y *j* :

* ***lectura después de escritura* (RAW):** este es el tipo de riesgo más común. Ocurre cuando la instrucción *j* lee el registro *x* antes de que la instrucción *i* escriba el registro *x* . Por lo tanto, la instrucción *j* usaría el valor incorrecto de *x* .
* ***escritura después de leer* (WAR)** : los peligros de WAR ocurren cuando la instrucción *j* escribe *x* y la instrucción *i* lee *x* , y la instrucción *j* se reordena para que ocurra antes de *i* . Por lo tanto, la instrucción *i* lee el valor incorrecto de *x* . Este peligro solo ocurre cuando se reordenan las instrucciones, lo que rara vez ocurre en SweRV EH1; específicamente, los peligros WAR nunca ocurren en SweRV EH1.
* ***Write After Write* (WAW):** Los peligros WAW ocurren cuando las instrucciones se reordenan y la instrucción *j* escribe *x* antes de que la instrucción *i* escriba *x* . Este peligro solo ocurre cuando se reordenan las instrucciones, lo que rara vez ocurre en SweRV EH1; sin embargo, en el caso de cargas que no bloqueen, podría ocurrir un peligro WAW, como analizaremos más adelante en esta práctica de laboratorio.

En las siguientes secciones, analizamos cómo se resuelven los riesgos de datos RAW en el procesador SweRV EH1, y luego describimos tareas y ejercicios relacionados con los riesgos RAW. También describimos un ejercicio que analiza una situación cuando ocurre un peligro WAW.

**NOTA:** Antes de analizar la lógica de peligros de datos de SweRV EH1, recomendamos leer la Sección 7.5 en DDCARV sobre cómo se resuelven los peligros en el procesador canalizado. Los riesgos de datos, específicamente, se analizan en la Sección 7.5.3 . Aunque el procesador segmentado que se muestra en el libro es más simple que SweRV EH1, los riesgos de datos se resuelven de manera similar en ambos procesadores.

# SOLUCIÓN DE PELIGROS DE DATOS CON REENVÍO EN LA ETAPA DE DESCODIFICACIÓN

Como se explica en la Sección 7.5.3 de DDCARV, algunos riesgos de datos RAW se pueden resolver mediante el envío (también denominado derivación) de un resultado de una instrucción que se ejecuta en una etapa de canalización avanzada a una instrucción dependiente que se ejecuta en una etapa de canalización anterior. Esto requiere agregar multiplexores frente a las unidades funcionales (ALU, multiplicador, adddor que calcula la dirección efectiva en DC1, etc.) para seleccionar sus operandos del archivo de registro o de las etapas posteriores.

La figura 1 amplía la etapa de decodificación que se muestra en la figura 4 de la práctica de laboratorio 11 con los valores de omisión. La lógica de reenvío produce un bypass (es decir, reenviado) para cada uno de los dos operandos de origen en cada una de las vías:

* **Vía-0:**
  + Primer operando de entrada: i0\_rs1\_bypass\_data\_d[31:0]
  + Segundo operando de entrada: i0\_rs2\_bypass\_data\_d[31:0]
* **Vía-1:**
  + Primer operando de entrada: i1\_rs1\_bypass\_data\_d[31:0]
  + Segundo operando de entrada: i1\_rs2\_bypass\_data\_d[31:0]

Estas cuatro entradas se distribuyen a los multiplexores 3:1 y 4:1 que determinan los operandos de entrada para cada una de las rutas de canalización de las etapas de ejecución. En aras de la claridad en la Figura 1 , las señales están conectadas por nombre . Las entradas a la lógica de reenvío son los resultados producidos por instrucciones de programas anteriores que están más avanzadas en la tubería, como veremos a continuación.



Figura 1 . Bypass de entradas a las Unidades Funcionales.

Existen muchas rutas de reenvío en el procesador SweRV EH1; en esta sección nos centramos en una ruta específica y la analizamos en detalle . Luego, en las tareas y ejercicios, inspeccionarás otros casos. Analizamos la situación de dos instrucciones AL dependientes que se ejecutan simultáneamente y cómo se resuelven los riesgos de datos RAW. Como hicimos en las Prácticas 12 y 13, comenzamos con un estudio básico (Sección 2.A) y luego pasamos a un análisis avanzado (Sección 2.B). Puede optar por completar solo la sección básica o completar ambas secciones.

Trabajaremos con el ejemplo que se muestra en la Figura 2 , que ejecuta dos instrucciones de add contenidas dentro de un ciclo que se repite para iteraciones 0xFFFF. La primera instrucción de add escribe un valor en t4 y la segunda instrucción de add usa t4 como su segundo operando de entrada. Se inserta una instrucción de adición independiente ( agregar t6, t6, -1 ), que es la instrucción que actualiza el índice del ciclo) entre las dos instrucciones de agregar para obligar a las instrucciones de agregar dependientes a usar la misma forma que el procesador.

|  |
| --- |
| .globl Test\_Asamblea  .text Test\_Asamblea:  li t3, 0x3 li t4, 0x2li t5, 0x1li t6, 0xFFFF  REPETIR:  INSERT\_NOPS\_8  **add t4, t4, t5** # t4 = t4 + t5 (t4 = 2 + 1)  add t6, t6, -1  **add t3, t3, t4** # t3 = t3 + t4 (t3 = 3 + 3)  INSERT\_NOPS\_9  li t3, 0x3  li t4, 0x2 li t5, 0x1 bne t6, cero, REPETIR # Repite el ciclo  .final |

Figura 2 . Peligro de datos RAW entre dos instrucciones de adición

carpeta *[RVfpgaPath]/RVfpga/Labs /Lab15/DataHazards\_AL-AL* proporciona el proyecto PlatformIO para que pueda analizar, simular y modificar el programa como desee. Abra el proyecto en PlatformIO, constrúyalo y abra el archivo de desensamblaje (disponible en *[RVfpgaPath]/RVfpga/Labs /Lab15/DataHazards\_AL-AL/.pio/build/swervolf\_nexys/firmware.dis* ) verá que los dos agregan instrucciones que estamos analizando se ubican en las direcciones 0x000001A0 y 0x000001A8:

**0x000001a0: 01ee8eb3 agregar t4,t4,t5**

**0x000001a4: ffff8f93 agregar6,t6,-1**

**0x000001a8: 01de0e33 agregar t3,t3,t4**

1. **Análisis básico de un riesgo de datos RAW entre instrucciones AL**

En el ejemplo que estamos analizando, la segunda instrucción de add ( add t3,t3, t4 ) necesita usar el resultado de la primera instrucción de add ( add t4 , t4,t5 ) como su segundo operando de entrada. Este resultado está disponible en la etapa EX1, desde donde puede pasarse a la etapa Decode y ser utilizado por la segunda instrucción de adición . En nuestro ejemplo ( Figura 2 ), todas las iteraciones son iguales y t4 es 2 inicialmente y 3 después de la primera add. Este último valor (3) es el que debe usar la segunda add como su segundo operando de entrada, y no el valor leído del Archivo de Registro (que es 2 hasta que la primera instrucción de add llega a la etapa Writeback y la actualiza).

La Figura 3 ilustra el flujo de las instrucciones del ejemplo de la Figura 2 a través de la canalización SweRV EH1 para una iteración aleatoria del bucle. En el ciclo *i* , el valor calculado en la etapa EX1 de la tubería I0 debe enviarse a la instrucción que se encuentra en la etapa de decodificación de Way-0, debido al riesgo de datos RAW entre las dos instrucciones adicionales bajo análisis.



Figura 3 . Ejecución del código de ejemplo de la Figura 2 . El reenvío se realiza en el ciclo *i* .

La figura 4 ilustra las etapas SweRV EH1 Way-0 Decode y EX1 durante el ciclo *i* de la figura 3 . En este ciclo, la primera instrucción de add ( agregar t4,t4,t5 ) está en la etapa EX1 y la segunda instrucción de add ( agregar t3,t3,t4 ) está en la etapa de decodificación. Como se muestra en la figura, el resultado de la primera instrucción de add se pasa por alto a la etapa de decodificación, es seleccionado por la lógica de reenvío (como analizaremos en detalle en la siguiente sección) y se usa como el segundo operando de entrada para la segunda instrucción de adición .

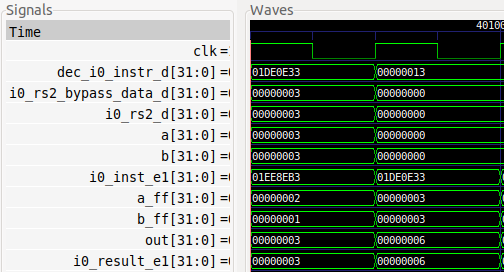


Figura 4 . Resultado reenviado desde EX1 a Decode (segundo operando) de Way 0

Finalmente, la Figura 5 muestra la simulación del programa de la Figura 2 durante los ciclos *i* e *i+1* de la figura 3 .

**add t4,t4,t5**

**add t3,t3,t4**



**i**

**i+1**

**EX1 stage**

**Decode stage**

Figura 5 . Simulación del código de ejemplo de la Figura 2

**TAREA :** Replicar la simulación de la Figura 5 en su propia computadora. Puede usar el archivo *.tcl* proporcionado en: *[RVfpgaPath]/RVfpga/Labs /Lab15/DataHazards\_AL-AL/test\_Basic.tcl* .

Analice la simulación de la Figura 5 y el diagrama de la Figura 4 al mismo tiempo.

* Instrucción agregar t4, t4, t5 (0x01ee8eb3) :
  + En el ciclo *i* , esta instrucción está en la etapa EX1 del Pipe I0 ( i0\_inst\_e1 = 0x01ee8eb3 ). Calcula la siguiente adición en la ALU:

a\_ff (2) + b\_ff (1) = fuera (3)

El resultado de la add se proporciona como entrada a la lógica de reenvío en la etapa de decodificación, como se muestra en la figura 4 .

* Instrucción agregar t3, t3, t4 (0x01de0e33) :
  + En el ciclo *i* , esta instrucción está en la etapa Decode de Way-0 ( dec\_i0\_instr\_d = 0x01de0e33 ). La lógica de reenvío conecta i0\_result\_e1 con i0\_rs2\_bypass\_data\_d . Los dos multiplexores 3:1 seleccionan los operandos de entrada para la add que será calculada en el siguiente ciclo (ciclo *i+1* ) en la etapa EX1 de la Tubería I0; específicamente:

a = 3 (del archivo de registro)

b = 3 (desde la salida ALU en la etapa EX1 de la tubería I0, a través de la lógica de reenvío, señal i0\_rs2\_bypass\_data\_d )

* + En el ciclo *i+1* , esta instrucción está en la etapa EX1 del Pipe I0 ( i0\_inst\_e1 = 0x01de0e33 ). Calcula la siguiente adición en la ALU:

a\_ff (3) + b\_ff (3) = fuera (6)

**TAREA :** Eliminar todas las instrucciones nop del ejemplo de la Figura 2 . Dibuje una figura similar a la Figura 3 para dos iteraciones consecutivas del bucle, luego analice y confirme que la figura es correcta comparándola con una simulación de Verilator, y finalmente calcule el IPC usando los contadores de rendimiento mientras ejecuta el programa en el tablero.

**TAREA :** En el ejemplo de la Figura 2 , elimine todas las instrucciones nop y mueva la instrucción add t6,t6,-1 después de add t3,t3,t4 instrucción, y luego vuelva a examinar el programa tanto en la simulación como en el tablero. En este programa reordenado, las dos instrucciones de adición dependientes ( add t4,t4,t5 y add t3,t3,t4 ) llegan a la etapa de decodificación en el mismo ciclo, y esto tiene un impacto en el rendimiento. Explique el impacto de estos cambios, utilizando tanto la simulación como la ejecución en el tablero.

Pruebe situaciones similares en las que reemplace la instrucción de adición dependiente por otras instrucciones dependientes, como:

- add t4,t4,t5

**múltiple** t3, t3, t4

- add t4,t4,t5

**div** t3, t3, t4

- add t4,t4,t5

**lw** t3, 0(t4)

1. **Análisis avanzado de un riesgo de datos RAW entre instrucciones AL(OPCIONAL AICC, No es necesario hacer nada a partir de aquí en AICC)**
2. **explicación teórica**

La Figura 6 amplía los diagramas de la Figura 1 y la Figura 4 agregando un multiplexor 10:1 (rodeado por un cuadrado azul en la Figura 6 ) que produce la señal i0\_rs2\_bypass\_data\_d , que en nuestro ejemplo de la Figura 2 proporciona el segundo operando de entrada para la segunda instrucción de add ( add t3, t3, **t4** ). Este multiplexor 10:1 se implementa dentro del cuadro Lógica de reenvío que se muestra en la Figura 1 y la Figura 4 .

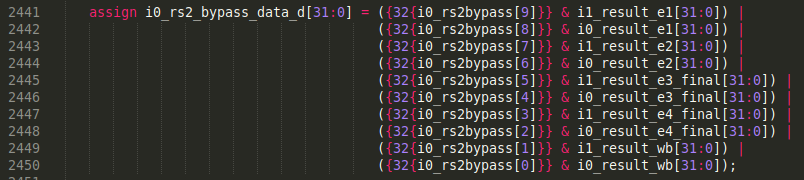
La figura también muestra las conexiones de entrada de este multiplexor 10:1. El valor anulado puede provenir de una instrucción que se ejecuta a través de la vía 0 o la vía 1. Por lo tanto, se necesitan cinco rutas de reenvío por vía. Específicamente, las entradas al multiplexor 10:1 pueden provenir de cualquiera de las etapas posteriores (EX1, EX2, EX3, Commit y Writeback) de Way 0 o Way 1. En aras de la simplicidad, conectamos las cinco entradas provenientes de Vía 0 usando cables, mientras que las 5 entradas provenientes de Vía 1 están conectadas por nombre.

Tres multiplexores 10:1 adicionales dentro de la lógica de reenvío calculan los otros tres operandos de origen: señales i0\_rs1\_bypass\_data\_d , i1\_rs1\_bypass\_data\_d e i1\_rs2\_bypass\_data\_d . Sin embargo, no los mostramos en la figura ya que no se utilizan en el ejemplo que analizamos en esta sección ( Figura 2 ). Los cuatro multiplexores se pueden encontrar en las líneas 2429-2473 del módulo **dec\_decode\_ctl** .



Figura 6 . Tubería I0 que incluye la lógica de reenvío utilizada para la segunda fuente de entrada de la ALU

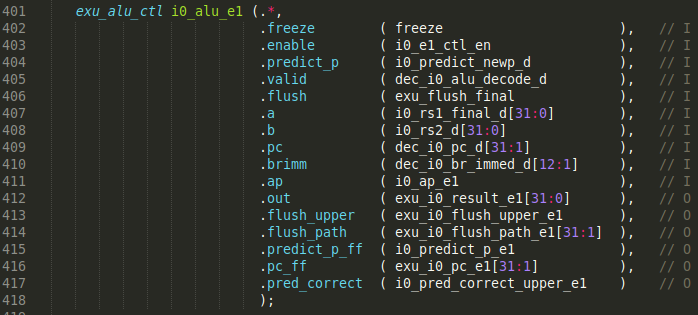




(Módulo **dec\_decode\_ctl** )



(Módulo **exu** )



(Módulo **exu** )

Figura 7 . Multiplexores 10:1 y 3:1 resaltados en la Figura 6

La figura 7 amplía los dos multiplexores (multiplexores 10:1 y 3:1) de la figura 6 que calculan el segundo operando de entrada para la ALU de tubería I0 ( b ). La figura muestra tanto un diagrama de bloques como el código Verilog donde se implementan estos multiplexores en los módulos **dec\_decode\_ctl** y **exu** .

**NOTA:** Los dos multiplexores de la Figura 7 también existen en el procesador de DDCARV. Los datos se reenvían a la etapa de ejecución en ese procesador y existen menos rutas de reenvío porque no es superescalar y tiene una canalización más corta. Puede analizar las rutas de reenvío en la Figura 7.55 de DDCARV.

A continuación, analizamos las entradas, las salidas y la señal de control de los dos multiplexores que se muestran en la Figura 7 .

**Multiplexor 10:1** :

**Salida:** La salida del multiplexor 10:1 es i0\_rs2\_bypass\_data\_d[31:0] . Esta señal contiene el valor que se debe reenviar (omitir) a la instrucción en la etapa de decodificación.

**Entradas:** Las entradas al multiplexor 10:1 son el resultado de instrucciones previas en el programa que se ejecutan en etapas posteriores (EX1, EX2, EX3, Commit o Writeback). Cinco de estas señales provienen de la tubería I0 (como se muestra en la Figura 6 ) y las otras cinco señales provienen de la tubería I1 (no se muestra en la Figura 6 ), ya que la instrucción en la etapa de decodificación podría depender potencialmente de una instrucción que se ejecuta en cualquier de los dos caminos.

**Señal de control:** La señal de control ( i0\_rs2bypass[9:0] ) selecciona qué entrada se conecta con la salida del multiplexor. Está formado por 10 bits, siendo como máximo uno de ellos alto al mismo tiempo (todos pueden ser cero si no hay riesgo de datos). El multiplexor funciona de la siguiente manera:

* Si i0\_rs2bypass[9] == 1🡪 i0\_rs2\_bypass\_data\_d = i1\_resultado\_e1
* Si i0\_rs2bypass[8] == 1🡪 i0\_rs2\_bypass\_data\_d = i0\_resultado\_e1
* Si i0\_rs2bypass[7] == 1🡪 i0\_rs2\_bypass\_data\_d = i1\_resultado\_e2
* Si i0\_rs2bypass[6] == 1🡪 i0\_rs2\_bypass\_data\_d = i0\_resultado\_e2
* Si i0\_rs2bypass[5] == 1🡪 i0\_rs2\_bypass\_data\_d = i1\_resultado\_e3\_final
* Si i0\_rs2bypass[4] == 1🡪 i0\_rs2\_bypass\_data\_d = i0\_resultado\_e3\_final
* Si i0\_rs2bypass[3] == 1🡪 i0\_rs2\_bypass\_data\_d = i1\_resultado\_e4\_final
* Si i0\_rs2bypass[2] == 1🡪 i0\_rs2\_bypass\_data\_d = i0\_resultado\_e4\_final
* Si i0\_rs2bypass[1] == 1🡪 i0\_rs2\_bypass\_data\_d = i1\_resultado\_wb
* Si i0\_rs2bypass[0] == 1🡪 i0\_rs2\_bypass\_data\_d = i0\_resultado\_wb

Para entender cómo se calcula esta señal de control de 10 bits, explicamos el cálculo de la señal i0\_rs2bypass[8] , que es la que sube en nuestro ejemplo de la Figura 2 para el bypass add - add .

* Si i0\_rs2bypass[8] es 1, el valor anulado seleccionado es i0\_result\_e1 , que es el resultado de la ejecución de la instrucción en la etapa EX1 de la tubería I0 (consulte la Figura 6 ).
* Para que EX1 envíe datos a la etapa de decodificación (ambos en la tubería I0), deben ocurrir las siguientes condiciones (consulte la Sección 4 del documento SweRVref para revisar las señales de control):
* El segundo operando de entrada de la instrucción en la etapa de decodificación se lee del archivo de registro y no se lee del registro cero. En la Unidad de Control SweRV EH1 , esto ocurre cuando dec\_i0\_rs2\_en\_d es 1. El código Verilog correspondiente es:

dec\_i0\_rs2\_en\_d = i0\_dp.rs2 & (i0r.rs2[4:0] != 5'd0);

* La instrucción en la etapa EX1 de la Tubería I0 es válida:

e1d.i0v == 1

* El registro destino de la instrucción en la etapa EX1 (del tubo I0) y el segundo registro fuente de la instrucción en la etapa Decode (de la Vía 0) son iguales:

e1d.i0rd[4:0] == i0r.rs2[4:0]

* La instrucción en la etapa EX1 (de la Tubería I0) es una operación ALU:

i0\_rs2\_class\_d.alu == 1

* Teniendo todo esto en cuenta, podemos concluir que:

**i0\_rs2bypass[8] =**

**(i0\_dp.rs2 & (i0r.rs2[4:0] != 5'd0)) &**

**e1d.i0v &**

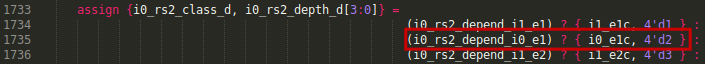
**( e1d.i0rd[4:0] == i0r.rs2[4:0] ) &**

**i0\_rs2\_clase\_d.alu ;**

**TAREA :** Compare las ecuaciones anteriores con las explicadas para el procesador segmentado de DDCARV .

**TAREA :** Analizar el código Verilog para explicar cómo se realiza el cálculo de la ecuación anterior. Debe inspeccionar las siguientes líneas del módulo **dec\_decode\_ctl** .









**TAREA :** escriba ecuaciones (similares a la anterior) para otros bits de control de i0\_rs2bypass[9:0] , i0\_rs1bypass[9:0] , i1\_rs2bypass[9:0] e i1\_rs1bypass[9:0] .

**Multiplexor 3:1** :

**Salida:** La salida del multiplexor 3:1 es i0\_rs2\_d[31:0] . Esta señal se envía a la segunda entrada ( b ) de la ALU en la Vía 0.

**Entradas:** Las entradas al multiplexor 3:1 son:

* El valor leído del archivo de registro ( gpr\_i0\_rs2\_d ).
* El valor Inmediato ( dec\_i0\_immed\_d ), obtenido de la instrucción.
* El valor desviado de etapas posteriores ( i0\_rs2\_bypass\_data\_d ) obtenido del multiplexor 10:1 descrito anteriormente.

**Señal de control:** la señal de control al multiplexor 3:1 ( dec\_i0\_rs2\_bypass\_en\_d ) selecciona:

* El valor omitido de etapas posteriores ( i0\_rs2\_bypass\_data\_d ), si dec\_i0\_rs2\_bypass\_en\_d ==1
* O el valor proveniente del Archivo de Registro o el Inmediato ( gpr\_i0\_rs2\_d y dec\_i0\_immed\_d , respectivamente ), si dec\_i0\_rs2\_bypass\_en\_d == 0. Puede parecer extraño que la misma señal seleccione dos entradas; sin embargo, la señal que no debe seleccionarse (ya sea gpr\_i0\_rs2\_d o dec\_i0\_immed\_d ) se fuerza a cero en el código Verilog.

La señal de selección del multiplexor 3:1 ( dec\_i0\_rs2\_bypass\_en\_d ) se calcula simplemente como el OR lógico de la señal de control de 10 bits del multiplexor 10:1:

asignar dec\_i0\_rs2\_bypass\_en\_d = |i0\_rs2bypass[9:0];

Así, siempre que el segundo operando de entrada de una instrucción dependa del resultado de una instrucción anterior que aún se está ejecutando (es decir, cualquiera de los 10 bits de la señal i0\_rs2bypass[9:0] es 1), dec\_i0\_rs2\_bypass\_en\_d == 1 y se obtiene el operando a través del reenvío. Por el contrario, si no depende de ninguna instrucción anterior, dec\_i0\_rs2\_bypass\_en\_d == 0 y el operando proviene del Archivo de registro o del Inmediato.

1. **Experimento**

La Figura 8 muestra la simulación del programa de la Figura 2 en una iteración aleatoria del bucle. El ciclo *i* de la figura 3 se indica en la parte superior de la figura.

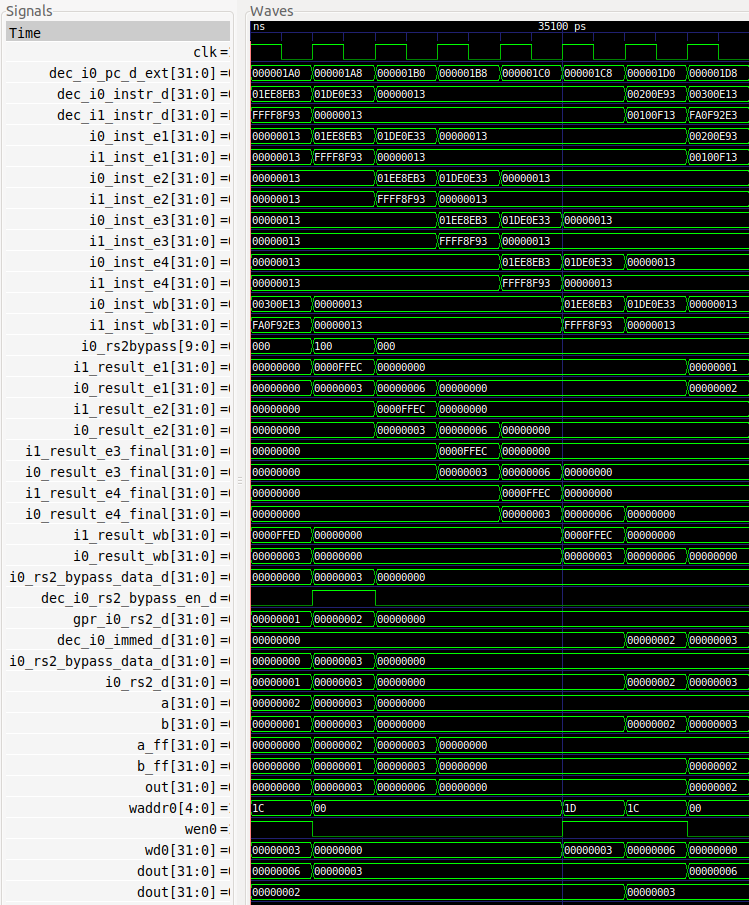
Las señales en la parte superior (Trace Signals) se incluyen para ayudar a rastrear las instrucciones a medida que avanzan a través de la canalización. Tenga en cuenta que estas señales ya se utilizaron en laboratorios anteriores. El significado de cada señal en la Vía 0 es el siguiente (lo mismo se aplica a la Vía 1 simplemente sustituyendo i0 por i1 en los nombres de las señales):

* dec\_i0\_instr\_d 🡪instrucción en la etapa de decodificación
* i0\_inst\_e1 🡪instrucción en la etapa EX1
* i0\_inst\_e2 🡪instrucción en la etapa EX2
* i0\_inst\_e3 🡪instrucción en la etapa EX3
* i0\_inst\_e4 🡪instrucción en la etapa Commit
* i0\_inst\_wb 🡪instrucción en la etapa Writeback

Debajo de Trace Signals, se muestran las principales señales de cada multiplexor analizado anteriormente. Cada multiplexor está rodeado por dos líneas azules, mientras que la señal de control, las entradas y la salida de cada multiplexor están separadas por líneas rojas.

**add t4,t4,t5**

**add t3,t3,t4**



**DECODE**

**WB stage**

**EX1 stage**

**Decode stage**

**COMMIT**

**3-1 MUX**

**10-1 MUX**

**Trace**

**Signals**

**Output**

**3 Inputs**

**Control**

**Output**

**10 Inputs**

**Control**

**EX3**

**WB**

**EX2**

**EX1**

**Cycle i**

Figura 8 . Simulación del programa de la Figura 2 en una iteración aleatoria del ciclo

La figura 9 muestra las etapas de decodificación y EX1 durante la ejecución del programa de la figura 2 en el ciclo *i* (como se define en la figura 8 ).



Figura 9 . Etapas de decodificación y EX1 durante la ejecución del programa de ejemplo de la Figura 2 en el Ciclo *i* (como se define en la Figura 8 )

**TAREA :** replicar la simulación de la Figura 8 en su propia computadora. Puede usar el archivo *.tcl* proporcionado en: *[RVfpgaPath]/RVfpga/Labs /Lab15/DataHazards\_AL-AL/test\_Advanced.tcl* .

Analice la simulación de la Figura 8 y el diagrama de la Figura 9 al mismo tiempo.

* **Señales de seguimiento que se muestran en la Figura 8** :
  + En el ciclo *i* , la segunda instrucción de add se ejecuta en la etapa de decodificación de la Vía 0 ( dec\_i0\_instr\_d = 0x01DE0E33), y la primera instrucción de add se ejecuta en la etapa EX1 de la tubería I0 ( i0\_inst\_e1 = 0x01EE8EB3).
  + En el ciclo *i+1* , la segunda adición avanza a la etapa EX1 de la tubería I0 ( i0\_inst\_e1 = 0x01DE0E33) y la primera adición avanza a la etapa EX2 de la tubería I0 ( i0\_inst\_e2 = 0x01EE8EB3).
* **10 :1 Multiplexor** : En el Ciclo *i* , la señal i0\_rs2bypass[9:0] = 0x100 (es decir , i0\_rs2bypass[8] = 1), por lo tanto, la salida se conecta con el valor proveniente de la etapa EX1 de la tubería I0 (consulte la Figura 9 ) :

i0\_rs2\_bypass\_data\_d = i0\_result\_e1 = 0x00000003

* **Multiplexor 3:1** : En el Ciclo *i* , la señal dec\_i0\_rs2\_bypass\_en\_d = 1, por lo que la salida se conecta con el valor proveniente de la lógica de derivación (ver Figura 9 ):

i0\_rs2\_d = i0\_rs2\_bypass\_data\_d = 0x00000003

* **Etapa EX1 que se muestra en la Figura 8** :
  + En el ciclo *i* , la primera instrucción de add calcula la add en la ALU de tubería I0: a\_ff (2) + b\_ff (1) = out (3).
  + En el ciclo *i+1* , la segunda instrucción de add calcula la add en la ALU de tubería I0: a\_ff (3) + b\_ff (3) = out (6).

**TAREA :** Para el programa de la Figura 2 , realice el mismo análisis que en la Figura 8 para situaciones en las que las dos instrucciones dependientes se colocan a diferentes distancias entre sí. Puede controlar la distancia cambiando el número de nops entre las dos instrucciones de adición dependientes.

Además, cree otros ejemplos donde el primer operando de entrada sea el que recibe los datos de reenvío.

También puede crear otros ejemplos en los que las dos instrucciones de adición se ejecuten a través de la tubería I1 y confirmar que el comportamiento es el mismo.

Finalmente, sustituya la instrucción de adición dependiente ( add t3,t3,t4 ) por otras instrucciones dependientes que se ejecutan a través de otras tuberías y analice los resultados de la simulación. Por ejemplo, en lugar de la segunda instrucción de add , podría incluir una de las siguientes instrucciones:

- lw t3, (t4) (forzar que el valor de lectura provenga del DCCM como se explica en la práctica de laboratorio 13)

- mu t3, t3, t4

- división t3, t3, t4

# SOLUCIÓN DE PELIGROS DE DATOS CON REENVÍO EN LA ETAPA DE COMPROMISO

Una situación más delicada ocurre cuando una instrucción depende de una instrucción anterior que necesita varios ciclos para obtener el resultado (es decir, una operación de varios ciclos), como una instrucción lw , una instrucción mul , una instrucción div , etc. En esta sección analizamos una situación concreta que se puede dar en la ejecución de una instrucción lw y una instrucción add dependiente , y dejamos como ejercicio el análisis de otras instrucciones y situaciones.

Como se explicó en la Práctica 13, una instrucción lw necesita tres ciclos (etapas DC1, DC2 y DC3) para obtener su resultado cuando se utiliza la memoria DCCM de baja latencia. Este es el escenario utilizado en esta sección. (Como también analizamos en las Prácticas de laboratorio 13 y 14, se incurre en un retraso mayor cuando se usa la memoria DDR2 externa ; los efectos de esta latencia de memoria más grande sobre los riesgos de datos se dejan como ejercicio).

Si la instrucción lw se ejecuta tres o más ciclos antes que la instrucción de adición dependiente , el riesgo se resuelve como se explica en la Sección 2. En este caso, los mismos multiplexores 10:1 y 3:1 descritos en esa sección se utilizan para reenviar los datos. leído por la instrucción de carga a la instrucción subsiguiente que depende de ella.

**APÉNDICE A:** El apéndice al final del documento incluye un ejemplo de un peligro de lw - add RAW data que se maneja como se explica en la Sección 2.

add dependiente , el riesgo se resuelve de una manera diferente a la descrita en la Sección 2. El problema ahora es que cuando la instrucción de add llega a la etapa EX1, el valor leído por la instrucción lw es no disponible aún.

En el procesador segmentado explicado en DDCARV, se introducen burbujas en este caso, que hacen esperar a la instrucción dependiente y solo utilizan el valor leído cuando está disponible. Esto requiere poco hardware adicional, pero afecta el rendimiento. Por lo tanto, SweRV EH1 permite que la instrucción dependiente continúe a través de la canalización y luego vuelva a calcular la operación en la etapa de confirmación, si es necesario debido a una dependencia de datos.

Específicamente, SweRV EH1 agrega una ALU adicional (la ALU secundaria) en la etapa de compromiso de cada vía. Esta ALU recalcula la operación aritmético-lógica con las entradas adecuadas cuando es necesario. Por lo tanto, no se pierden ciclos debido a la parada, pero a costa de agregar dos ALU adicionales (una por vía), así como señales de control y lógica adicionales. Figura 10 ilustra la implementación de esta ALU secundaria en la etapa de compromiso de Way 0 (la ALU está rodeada por un cuadrado azul), así como la lógica de reenvío agregada en la etapa EX3 para el segundo operando de entrada (esta lógica está rodeada por un cuadrado rojo) . (En la Figura 4 del Laboratorio 11, estas dos ALU adicionales y las rutas de reenvío no se incluyeron por simplicidad).



Figura 10 . ALU secundaria en la etapa de compromiso de Way 0

**TAREA** : Agregue lógica a la Figura 10 para producir el primer operando de entrada ( a ) de la ALU secundaria en la tubería I0.

La Figura 11 muestra el código de ejemplo utilizado en esta sección. Ejecuta una instrucción lw seguida inmediatamente por una instrucción de add independiente ( add t6, t6, -1 : que calcula el índice del bucle) y luego una instrucción de add que depende de la carga. La instrucción de add independiente se incluye para forzar tanto la instrucción lw como la instrucción de add dependiente para que se ejecuten a través de la Vía 0. Por lo tanto, la única diferencia con respecto al programa del Apéndice es que las instrucciones lw y add están ahora más cerca; sin embargo, esta pequeña diferencia en el programa se traduce en una gran diferencia en la forma en que se ejecuta, como acabamos de explicar y demostraremos a continuación.

|  |
| --- |
| .globl Test\_Asamblea  .sección .midccm  #.datos  R: .espacio 4  .texto  Asamblea\_de\_prueba:  la t0, A # t0 = addr(A) li t1, 0x1 # t1 = 1 sw t1, (t0) # A[0] = 1 li t1, 0x0li t3, 0x1li t6, 0xFFFF  REPETIR: beq t6, cero, OUT # ¿Permanecer en el bucle? INSERT\_NOPS\_9 **lw t1, (t0)** add t6, t6, -1 **agregue t3, t3, t1** # t3 = t3 + t1 INSERT\_NOPS\_8 li t1, 0x0 li t3, 0x1 agregue t4, t4, 0x1  add t5, t5, 0x1  j REPETIR :  .final |

Figura 11 . Programa que ejecuta un lw , un complemento independiente y un complemento dependiente

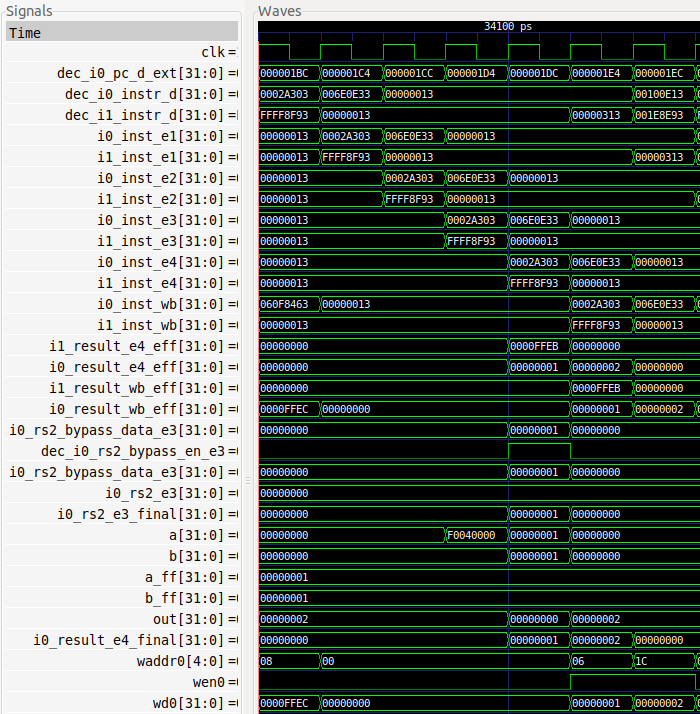
Como de costumbre, la carpeta *[RVfpgaPath]/RVfpga/Labs /Lab15/DataHazards\_Close-LW-AL* proporciona el proyecto PlatformIO para que pueda analizar, simular y modificar el programa como desee. Abra el proyecto en PlatformIO, constrúyalo y abra el archivo de desensamblado (disponible en *[RVfpgaPath]/RVfpga/Labs /Lab15/DataHazards\_ Close -LW-AL/.pio/build/swervolf\_nexys/firmware.dis* ). Observe que las instrucciones lw y add se colocan en las direcciones 0x000001bc y 0x000001c4 .

**0x000001bc: 0002a303 lw t1,0(t0)**

**0x000001c0: ffff8f93 agregar6,t6,-1**

**0x000001c4: 006e0e33 agregar t3,t3,t1**

La Figura 12 muestra la simulación del programa de la Figura 11 en una iteración aleatoria del ciclo. Una vez más, cualquier iteración sería válida excepto la primera, que debe intentar evitar debido a errores en la memoria caché de instrucciones. Como en el ejemplo de la sección anterior, las señales en la parte superior (señales de seguimiento) se incluyen para ayudar a rastrear las instrucciones a medida que avanzan a través de la canalización. Debajo de Trace Signals, se muestran las principales señales de los multiplexores 4:1 y 2:1 y la nueva ALU de la Figura 11 .



**Secondary ALU**

**EX4**

**2-1**

**MUX EX3**

**4-1**

**MUX EX3**

**Trace**

**Signals**

**add t3,t3,t1**

**i**

**i+1**

**lw t1,0(t0)**

Figura 12 . Simulación del programa de la Figura 11 en la tercera iteración del ciclo

La Figura 13 muestra un diagrama de la ejecución del programa de la Figura 11 en la séptima iteración del bucle y para el ciclo *i que* se muestra en la Figura 12 , cuando la instrucción add está en la etapa EX3 y la instrucción lw está en la etapa Commit, y para el ciclo *i+1* , cuando la add la instrucción está en la etapa de compromiso (es decir, EX4) y vuelve a calcular la operación en las entradas adecuadas.





Figura 13 . Diagrama de ejecución del programa de la Figura 11 en la séptima iteración del bucle y para los ciclos *i* e *i+1* de la Figura 12

**TAREA :** Replicar la simulación de la Figura 12 en su propia computadora. Puede utilizar el archivo *.tcl* proporcionado en: *[RVfpgaPath]/RVfpga/Labs /Lab15/DataHazards\_ Close- LW-AL/scriptLoad.tcl*

**TAREA :** Dibuje una figura similar a la Figura 3 para el ejemplo de la Figura 11 .

Analice la forma de onda de la Figura 12 y el diagrama de la Figura 13 al mismo tiempo.

* **Señales de seguimiento que se muestran en la Figura 12 :**
  + En el ciclo *i* , la instrucción add está en la etapa EX3 de Way 0 ( i0\_inst\_e3 = 0x006E0E33), y la instrucción lw está en la etapa Commit de I0 Pipe ( i0\_inst\_e4 = 0x0002A303).
  + En el ciclo *i+1* , la instrucción de add está en la etapa de Commit de la Vía 0 ( i0\_inst\_e4 = 0x006E0E33).
* **4-1 Multiplexor** : En el Ciclo *i* , se selecciona el valor leído por la instrucción de carga, que en este ciclo se encuentra en la etapa de Commit:

i0\_rs2\_bypass\_data\_e3 = i0\_result\_e4\_eff = 0x00000001

* **2-1 Multiplexor** : En el Ciclo *i* , debido a la dependencia entre la carga y la add, se selecciona el valor puenteado ( dec\_i0\_rs2\_bypass\_en\_e3 = 1). Por lo tanto:

i0\_rs2\_e3\_final = i0\_rs2\_bypass\_data\_e3 = 0x00000001

* **Etapa de confirmación ALU** : en el ciclo *i+1* , la add se vuelve a calcular con los valores correctos:

fuera = a\_ff + b\_ff = 0x00000001 + 0x00000001 = 0x00000002

Luego, en el multiplexor 3:1, se selecciona la salida ALU ( exu\_i0\_result\_e4 ). Tenga en cuenta que, si no existe ninguna dependencia, se seleccionaría el valor de la señal i0\_result\_e4 .

**TAREA :** En el ejemplo anterior, analizar cómo se obtiene el primer operando para la instrucción add t3, t3, t1 ( t3 ) . Puede utilizar el archivo *.tcl* proporcionado en: *[RVfpgaPath]/RVfpga/Labs /Lab15/DataHazards\_ Close- LW-AL/scriptLoad\_FirstOperand.tcl*

**TAREA :** elimine las instrucciones nop del ejemplo de la Figura 11 y obtenga el IPC utilizando los contadores HW.

**TAREA :** deshabilite la ALU secundaria como se explica en el laboratorio 11 y analice el ejemplo de la Figura 11 tanto con una simulación de Verilator como con una ejecución en el tablero.

**TAREA :** En el ejemplo de la Figura 11 , mueva la instrucción add t6,t6,-1 después de add t3,t3,t 1 instrucción y vuelva a examinar el programa tanto en la simulación como en el tablero.

# EJERCICIOS

1. Modifique el programa utilizado en la Sección 3 agregando una instrucción aritmético-lógica adicional que depende del resultado de la instrucción de add . Por ejemplo, puede reemplazar el ciclo de la Figura 11 con el siguiente código, donde se ha incluido una nueva instrucción AND ( **y t3, t4, t3** ), y donde hemos reordenado ligeramente el código avanzando con la instrucción **agregar t5, t5, 0x1** :

REPETIR:

beq t6, cero, FUERA

INSERTAR\_NOPS\_ 9

lw t1, (t0)

add t6, t6, -1

add t3, t3, t1

**add t5, t5, 0x1**

**y t3, t4, t3**

INSERT\_NOPS\_8

li t1, 0x0

li t3, 0x1

agregar t4, t4, 0x1

j REPETIR

FUERA:

Analice la simulación de Verilator y explique cómo se manejan los riesgos de datos para la nueva instrucción AL. Luego elimine todas las instrucciones nop y analice los resultados proporcionados por los contadores HW.

1. Analice la misma situación que la descrita en la Sección 3 para una instrucción mul seguida de una instrucción de add que usa el resultado de la multiplicación. En el programa de la Figura 11 , simplemente puede sustituir el lw por un mul que escribe en el registro t1 .
2. Analice una situación con una instrucción lw seguida de una instrucción mul que depende del valor leído por la carga. En el programa de la Figura 11 , simplemente puede sustituir la instrucción de adición dependiente por una instrucción mul .
3. ( *El siguiente ejercicio se basa en los ejercicios 4.18* , *4.19* , *4.20* y *4.26 de [PaHe].* )

Suponga que ejecutó el siguiente código en una versión del procesador SweRV EH1 que no maneja los riesgos de datos (es decir, el programador es responsable de abordar los riesgos de datos insertando nops cuando sea necesario). Agregue nops al código para que se ejecute correctamente.

add x11, x12, 5

add x13, x11, x12

add x14, x11, 15

add x15, x13, x12

Luego, cree secuencias de al menos tres fragmentos de código ensamblador que muestren diferentes tipos de riesgos de datos SIN PROCESAR. El tipo de dependencia de datos RAW se identifica por la etapa que produce el resultado y la siguiente instrucción que consume el resultado.

Para cada secuencia, ¿cuántos nops deberían insertarse y dónde, para permitir que su código se ejecute correctamente en un procesador SweRV EH1 sin reenvío ni detección de peligros? ¿Cuál es el CPI si usamos el reenvío disponible en SweRV EH1 y no insertamos nops?

1. En el programa de la Sección 2.C del Laboratorio 14 (disponible en *[RVfpgaPath]/RVfpga/Labs /Lab14/LW\_Instruction\_ExtMemory* ), reemplace la instrucción add **x1** , x1, 1 con add **x28** , x1, 1 . Esto introduce un peligro WAW entre la instrucción de adición modificada y la carga sin bloqueo al comienzo del bucle ( lw x28, (x29) ). Analiza en simulación cómo se maneja este peligro en SweRV EH1, para lo cual puedes mirar el valor de la señal wen2 en el Archivo de Registro. Intente comprender cómo se calcula esta señal en la Unidad de control (módulo **dec** ).
2. En el programa de la Sección 2.C del Laboratorio 14 (disponible en *[RVfpgaPath]/RVfpga/Labs /Lab14/LW\_Instruction\_ExtMemory* ), reemplace la instrucción add x1 , **x1** , 1 con add x1, **x28** , 1 . Esto introduce un riesgo RAW entre la instrucción de adición modificada y la carga sin bloqueo al comienzo del bucle ( lw x28, (x29) ). Analizar en simulación cómo se maneja este peligro en SweRV EH1.
3. En el programa de la Sección 2.C del Laboratorio 14 (disponible en *[RVfpgaPath]/RVfpga/Labs /Lab14/LW\_Instruction\_ExtMemory* ), reemplace la instrucción add x1 , **x1** , 1 con add x1, **x28** , 1 y la instrucción add **x7** , x7 , 1 con addr **x28** , x7 , 1 . Esto hace que ocurra un peligro tanto RAW como WAW. Analice en simulación cómo se manejan estos dos peligros en SweRV EH1.
4. **Reenvío de tienda a carga**

Esta es una situación muy interesante que no hemos analizado en este laboratorio y que usted analizará en este ejercicio. Cuando una tienda seguida de una carga accede a la misma dirección, los datos se pueden reenviar desde la tienda a la carga dentro del núcleo y se puede evitar la lectura de la memoria externa DDR, ahorrando tiempo y energía.

La lógica que implementa este reenvío está incluida en la LSU, y específicamente en los módulos **lsu\_bus\_intf** y **lsu\_bus\_buffer** , que debe inspeccionar en este ejercicio.

El proyecto PlatformIO de *[RVfpgaPath]/RVfpga/Labs /Lab15/Sw-Lw-Forwarding* ilustra un reenvío de almacenamiento y carga. Se proporciona un script *.tcl* en esa carpeta, que puede usar para analizar una iteración aleatoria del bucle y comprender cómo se lleva a cabo el reenvío de carga de almacenamiento en SweRV EH1.

# APÉNDICE A (OPCIONAL)

En este apéndice, incluimos un ejemplo de un riesgo de lw - agregar datos RAW que se maneja como se explica en la Sección 2. La Figura 14 muestra el código de ejemplo utilizado en este apéndice. Ejecuta una instrucción lw seguida de 5 instrucciones nop y una instrucción add que depende de la carga. Las instrucciones intermedias nop se incluyen para separar las dos instrucciones dependientes.

|  |
| --- |
| .globl Test\_Asamblea  .sección .midccm #.datosA: .espacio 4  .text Test\_Asamblea:  # El registro t3 también se llama registro 28 (x28) la t0, A # t0 = addr(A) li t1, 0x1 # t1 = 1 sw t1, (t0) # A[0] = 1 li t1, 0x0li t3, 0x1li t6, 0xFFFF  REPETIR: beq t6, cero, OUT # ¿Permanecer en el bucle? INSERT\_NOPS\_8 **lw t1, (t0)** INSERT\_NOPS\_5 **add t3, t3, t1** # t3 = t3 + t1 INSERT\_NOPS\_8 li t1, 0x0 li t3, 0x1 add t6, t6, -1 j REPETIR :  .final |

Figura 14 . Programa que ejecuta lw , 5 nops y una instrucción de add dependiente

Como de costumbre, f old [ RVfpgaPath *]/RVfpga/Labs /Lab15/DataHazards\_FarAway-LW-AL* proporciona el proyecto PlatformIO para que pueda analizar, simular y modificar el programa como desee. Abra el proyecto, constrúyalo y abra el archivo de desensamblado (disponible en *[ RVfpgaPath ]/RVfpga/Labs/Lab15/DataHazards\_FarAway-LW-AL/.pio/build/swervolf\_nexys/firmware.dis )* . Observe que las instrucciones lw y add se colocan en las direcciones 0x000001b0 y 0x000001c8 .

**0x000001b0: 0002a303 lw t1,0(t0)**

**0x000001b4: 00000013 no**

**0x000001b8: 00000013 no**

**0x000001bc: 00000013 no**

**0x000001c0: 00000013 no**

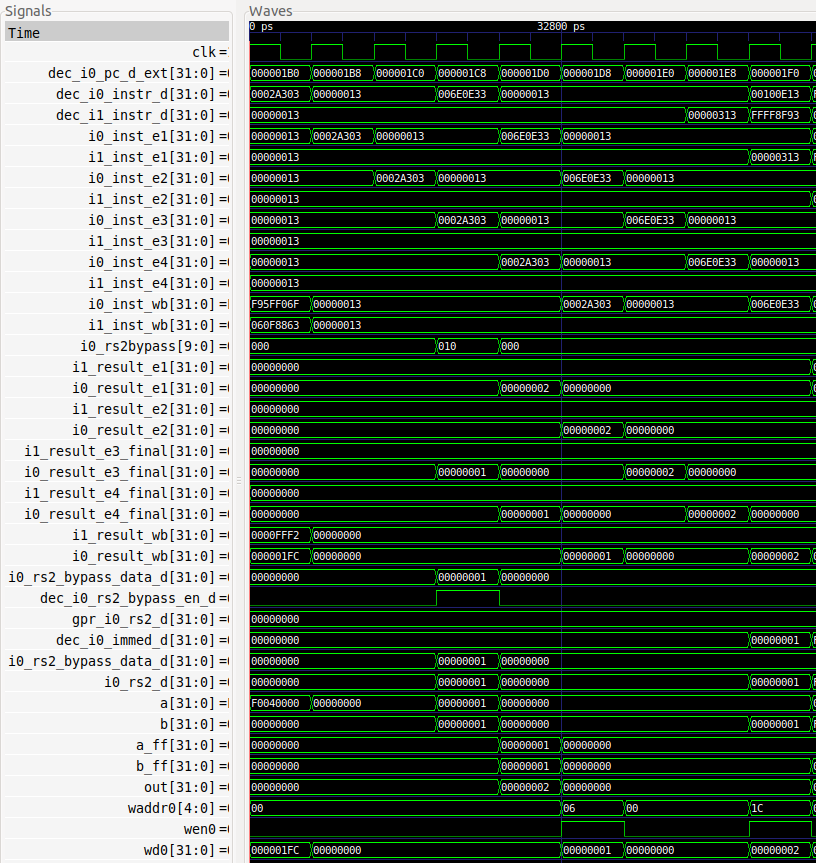
**0x000001c4: 00000013 no**

**0x000001c8: 006e0e33 agregar t3,t3,t1**

La Figura 15 muestra la simulación del programa de la Figura 14 en la tercera iteración del ciclo. Una vez más, cualquier iteración sería válida excepto la primera, que debe intentar evitar debido a errores en la memoria caché de instrucciones. Como en los ejemplos del laboratorio principal, las señales en la parte superior (Trace Signals) ayudan a rastrear las instrucciones a medida que avanzan a través de la canalización. Debajo de Trace Signals, se muestran las principales señales de cada multiplexor. Las señales de cada multiplexor están rodeadas de líneas discontinuas azules. Se ilustran la señal de control, las entradas y la salida de cada multiplexor, como se hizo en el laboratorio principal.

**lw t1,(t0)**

**add t3,t3,t1**



**Cycle i**

**10:1 MUX**

**3:1 MUX**

**Trace**

**Signals**

Figura 15 . Simulación del programa de la Figura 14 en la tercera iteración del ciclo

Figura 16 muestra un diagrama de la ejecución del programa de la figura 14 en el ciclo *i* (como se define en la figura 15 ), cuando la instrucción add está en la etapa de decodificación y la instrucción lw está en DC3.



Figura 16 . Hardware durante la ejecución del programa de la Figura 14 en la tercera iteración del bucle y en el cuarto ciclo que se muestra en la Figura 15

**TAREA :** Replicar la simulación de la Figura 15 en su propia computadora.

Analice la forma de onda de la Figura 15 y el diagrama de la Figura 16 al mismo tiempo.

* **Señales de seguimiento que se muestran en la Figura 15 :**
  + En el ciclo *i* , la instrucción add está en la etapa Decode de Way 0 ( dec\_i0\_instr\_d = 0x006E0E33), y lw está en la etapa DC3 de I0 Pipe ( i0\_inst\_e3 = 0x0002A303).
* **Multiplexor 10:1** : En el ciclo *i* , la señal i0\_rs2bypass[9:0] = 0x010 (es decir , i0\_rs2bypass[4] = 1), por lo que la salida se conecta al valor que proviene de la etapa EX3/DC3 de la tubería I0 (consulte la figura 16 ):

i0\_rs2\_bypass\_data\_d = i0\_result\_e3\_final = 0x00000001

* **Multiplexor 3:1** : En el Ciclo *i* , la señal dec\_i0\_rs2\_bypass\_en\_d = 1, por lo que la salida se conecta al valor proveniente de la lógica de derivación (ver Figura 9 ):

i0\_rs2\_d = i0\_rs2\_bypass\_data\_d = 0x00000001

**TAREA :** Compare cómo se maneja el escenario anterior en SweRV EH1 y en el procesador segmentado de DDCARV .

**TAREA :** Si compara cuidadosamente la Figura 16 y la Figura 6 de la Práctica de laboratorio 13, verá que el valor que lee la instrucción lw en el Archivo de registro de la Figura 6 de la Práctica de laboratorio 13 (señal lsu\_ld\_data\_corr\_dc3[31:0] ) es diferente al valor reenviado por el lw en la Figura 16 (señal lsu\_ld\_data\_dc3[31:0] ). La diferencia entre ambos valores es que el primero ha sido comprobado por la lógica ECC en el módulo **lsu\_ecc** , mientras que el segundo no. Explique por qué no es problemático que el valor enviado por el lw no se verifique en busca de errores.

**TAREA :** En el ejemplo de la Figura 14 , elimine todas las instrucciones nop antes de lw y después de add . No elimine los 5 nops entre las dos instrucciones dependientes. Analice la simulación y luego calcule el IPC con los contadores de rendimiento ejecutando el programa en la placa (puede parecer incómodo mantener las instrucciones nop al medir el IPC, ya que son instrucciones inútiles; sin embargo, el programa en sí es inútil y nuestro único objetivo aquí es analizar los riesgos de datos y comprenderlos).