

**EL PROGRAMA UNIVERSITARIO DE LA IMAGINACIÓN**

**Laboratorio RVfpga 11**

## **, organización y rendimiento de SweRV EH1**

# 1. INTRODUCCIÓN

En los primeros 10 laboratorios RVfpga (Laboratorios 1–10) presentamos la arquitectura RISC-V y cómo comunicar el SweRV EH1 Core usando varios periféricos. En los siguientes diez laboratorios (Laboratorios 11 a 20), nos sumergiremos en el nivel de la microarquitectura y analizaremos cómo funciona internamente el procesador SweRV EH1 y cómo funciona la jerarquía de caché/memoria.

**ESTUDIO SIGASI:** En estos laboratorios vamos a tratar con un extenso proyecto de Verilog: el SweRV EH1 Core RTL. Una forma de analizar los distintos módulos y señales es utilizar un editor típico como Sublime Text ( <https://www.sublimetext.com/>), que ofrece interesantes funcionalidades para navegar por un proyecto, inspeccionar los archivos, buscar cadenas, etc. Sin embargo, existen alternativas más adecuadas y específicas, como **Sigasi Studio** ( <https://www.sigasi.com/>), que recomendamos encarecidamente. Un documento complementario, **RVfpga\_ SweRVref.docx** , muestra, entre otras cosas, cómo instalar y utilizar Sigasi Studio (Sección 1 del documento RVfpga\_ SweRVref).

Como se explica en la Guía de inicio (GSG) de RVfpga, SweRV EH1 es un procesador en orden superescalar de 9 etapas y 2 vías de 32 bits. La Figura 1 muestra una vista de alto nivel de la microarquitectura SweRV EH1. SweRV EH1 admite extensiones de enteros (I), instrucciones comprimidas (C) y multiplicación y división de enteros (M) de RISC-V . Su impresionante alto rendimiento por MHz (4,9 CM/MHz) se logra gracias a la inclusión de varias técnicas de microarquitectura, desde las más básicas y comunes, como pipelining y caché de instrucciones, hasta otras técnicas más específicas y avanzadas, como superscalar. ejecución, cargas y divisiones sin bloqueo, dos ALU secundarias que permiten que las instrucciones Aritmético-Lógicas se repitan cuando sea necesario debido a riesgos de datos (consulte el Laboratorio 15 para obtener más detalles), cargas y almacenamientos no alineados, memorias de borrador tanto para instrucciones como para datos, y predicción de rama. Todas estas técnicas serán ampliamente analizadas en estos laboratorios.

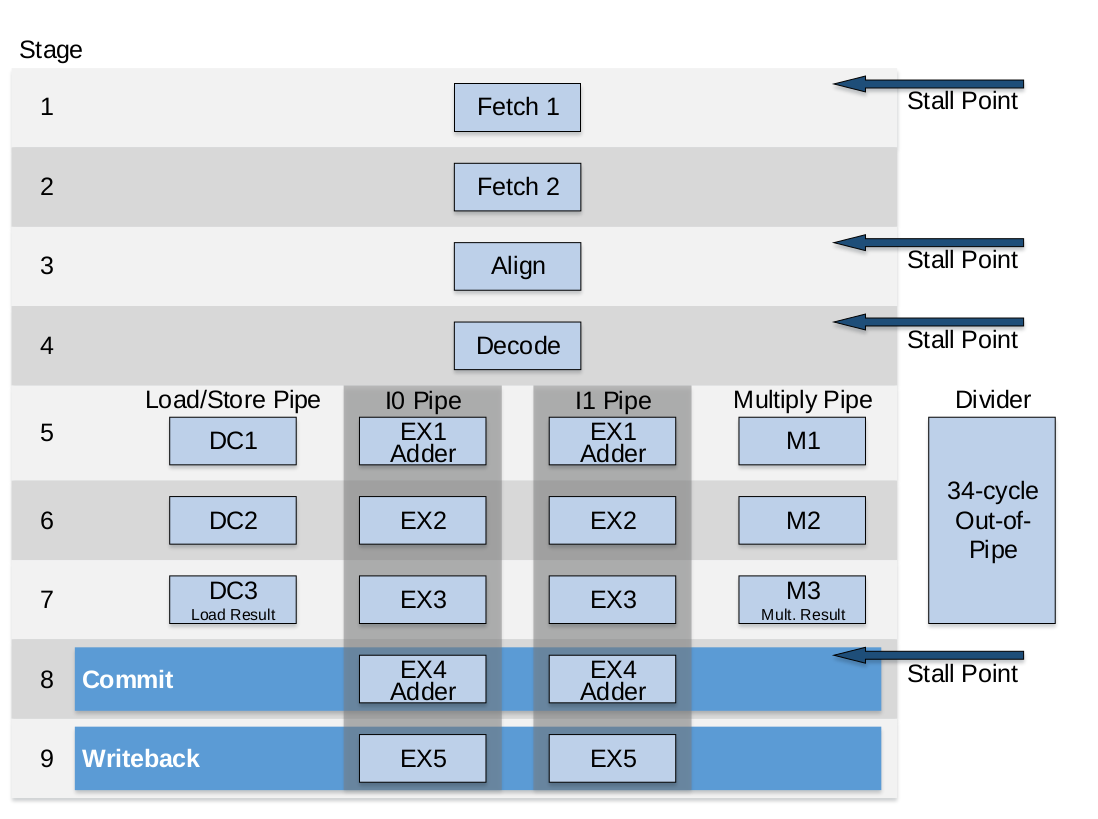


Figura 1 . Microarquitectura de núcleo SweRV EH1

(figura de <https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf>)

**NOTA:** Antes de comenzar este conjunto de prácticas de laboratorio, le recomendamos que lea detenidamente los capítulos 7 y 8 del libro de texto *Diseño digital y arquitectura de computadoras: Edición RISC-V* de S. Harris y D. Harris (Morgan Kaufmann © 2021). Algunos de los contenidos de estos laboratorios están inspirados en ese libro. Nos referiremos al libro como DDCARV.

La mayoría de los laboratorios se dividen en dos partes: una sección de fundamentos seguida de una sección avanzada. Además, dada la alta complejidad de algunas partes de un procesador real como el SweRV EH1, algunos detalles se trasladan al apéndice de un laboratorio determinado. De esta forma, los usuarios pueden optar por completar solo la sección fundamental, completar tanto la sección fundamental como la avanzada, o incluso profundizar en los apéndices y comprender las partes más complejas del procesador.

Las prácticas de laboratorio 11 a 20 comienzan con una explicación teórica de los conceptos y luego ilustran los conceptos usando figuras y una simulación Verilator de un programa de ejemplo. Estos son programas de juguete que solo pretenden ilustrar el concepto. También proporcionamos ejercicios para profundizar la comprensión y la experiencia con los conceptos descritos.

Uno puede completar solo un subconjunto de los laboratorios, según el objetivo y la profundidad del curso. Los conceptos de canalización, organización de memoria y microarquitectura avanzada/jerarquía de memoria se tratan en los siguientes laboratorios:

* **Canalización:** Labs 11, 12, 14, 15 y primera parte de 16 (instrucciones de bifurcación)
* **Memoria:** Laboratorios 11, 13 y 19
* **Microarquitectura avanzada y jerarquía de memoria:** Labs 17, 18, 20 y segunda parte de 16 (predictor de rama)

En este laboratorio (Lab 11), comenzamos a analizar el procesador SweRV EH1. Específicamente:

* **Sección 2** describe la organización de Verilog RTL y los detalles de cada etapa de canalización.
* **Seccion 3** muestra cómo utilizar los contadores de rendimiento para analizar el rendimiento del procesador.

El documento complementario ( **RVfpga\_ SweRVref.docx** ) describe:

* **Sección 1** : Uso de Sigasi Studio.
* **Sección 2** : Configuración del procesador SweRV EH1.
* **Sección 3** : Jerarquía de módulos del sistema RVfpga y sus señales más relevantes
* **Sección 4** : Estructuras y tipos para agrupar bits de control
* **Sección 5** : instrucciones comprimidas RISC-V
* **Sección 6** : Puntos de referencia reales

Después de este enfoque inicial, ampliamos este análisis en las prácticas 12-20 a varias unidades de procesador. Específicamente:

* **El laboratorio 12** se centra en instrucciones **aritmético-lógicas** profundizando en las etapas de decodificación, EX1/EX2/EX3 y reescritura.
* **Laboratorio 13** describe **instrucciones de memoria** (cargas y almacenamiento) enfocándose en las etapas DC1/DC2/DC3.
* **Lab 14** discute **riesgos estructurales** centrándose en la instrucción de multiplicación canalizada de 3 ciclos y en un caso específico relacionado con cargas sin bloqueo. El laboratorio también analiza la instrucción de división no segmentada de 34 ciclos en un apéndice.
* **Lab 15** analiza **los riesgos de datos** al describir las rutas de derivación del procesador.
* **laboratorio 16** describe **los riesgos de control** , las instrucciones de bifurcación y el predictor de bifurcación, para lo cual nos centraremos en las etapas Fetch 1 y Fetch 2 del procesador SweRV EH1.
* Mientras que en las prácticas de laboratorio anteriores solo se usa una forma de procesador en la mayoría de los casos, la **práctica de laboratorio 17** describe procesadores superescalares de 2 vías, como SweRV EH1.
* **Lab 18** es un laboratorio práctico en el que agregará nuevas instrucciones y contadores de hardware al núcleo SweRV EH1.
* **Las prácticas de laboratorio 19** y **20** se centran en las diversas memorias de baja latencia disponibles en el procesador: la memoria caché de instrucciones (I$) y las memorias de datos e instrucciones estrechamente acopladas (ICCM y DCCM).

# UNA PRIMERA APROXIMACIÓN A LA MICROARQUITECTURA SweRV EH1

El procesador descrito en DDCARV tiene 5 etapas de canalización, que se denominan etapas *Fetch, Decode, Execute, Memory* y *Writeback* . En contraste, la tubería SweRV EH1 se divide en 9 etapas ( Figura 1 ): la *Fetch1, Fetch2, Alinear, Decodificar, EX1* / *DC1* / *M1, EX2* / *DC2* / *M2, EX3* / *DC3* / *M3, Confirmar* ,y etapas de *reescritura* . Al comparar los dos procesadores, algunas etapas son equivalentes, como las etapas Decode y Writeback. Pero SweRV EH1 agrega rutas paralelas (cargar/almacenar frente a enteros frente a tuberías de multiplicación), divide algunas etapas en varias etapas ( *Fetch* son 2 etapas y *Execute* son 3 etapas) y agrega etapas (las etapas *Commit* y *Align* ).

El resto de esta sección describe la organización de Verilog RTL y los detalles de cada etapa de canalización. La Sección A describe la jerarquía de los módulos Verilog de SweRV EH1. Las secciones B y C analizan la microarquitectura de SweRV EH1 etapa por etapa. Finalmente, la Sección D proporciona un ejemplo práctico de las explicaciones teóricas dadas en las Secciones B y C.

**CONFIGURACIÓN DEL PROCESADOR SWERV EH1:** Muchas de las estructuras y características del procesador SweRV EH1 se pueden configurar o habilitar/deshabilitar. El documento complementario, RVfpga\_ SweRVref.docx , explica estas diferentes opciones en la Sección 2, que usará con frecuencia en las prácticas de laboratorio 12-20.

1. **Jerarquía de los módulos Verilog de SweRV EH1**

La figura 2 muestra la jerarquía de los principales módulos Verilog (algunos módulos no están incluidos en la figura) que componen el procesador SweRV EH1. Esta figura amplía la Figura 29 del GSG, donde mostramos la jerarquía de los módulos Verilog que componen el Sistema RVfpga. Estos módulos se encuentran en archivos con el mismo nombre en: *directorio [RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex* .

El módulo **mem** instancia las estructuras que conforman la jerarquía de memoria del procesador SweRV EH1: ICCM, DCCM e I$. El módulo **swerv** es la CPU general; instancia los módulos que componen el procesador SweRV EH1: Unidad de obtención de instrucciones ( **ifu** ), Unidad de decodificación ( **dec** ), Unidad de ejecución ( **exu ), Unidad** de carga/almacenamiento ( **lsu** )…



figura 2\_ Módulos principales de SweRV EH1

**PRINCIPALES SEÑALES DEL NÚCLEO SweRV EH1:** El documento complementario, RVfpga\_ SweRVref.docx , proporciona, en la Sección 3, las principales señales de entrada/salida hacia/desde los módulos del procesador SweRV EH1. Puede usarlo como referencia mientras completa los laboratorios 11-20.

1. **Fases Fetch (FC1 y FC2) y Align**

En esta sección, analizamos las tres primeras etapas del pipeline: las dos etapas Fetch (FC1 y FC2) y la etapa Align del pipeline SweRV EH1. La Figura 3 ilustra una vista muy simplificada de estas etapas.



Figura 3 . Vista simplificada de las etapas FC1, FC2 y Align. Tenga en cuenta que el ICCM está sombreado, lo que indica que está deshabilitado en nuestro sistema RVfpga.

1. **Etapas Fetch (FC1 y FC2)**

En cada ciclo, la etapa Fetch se encarga de **leer las instrucciones de la Memoria de Instrucciones** . En nuestra configuración, la Memoria de Instrucciones está compuesta por un ICCM (implementado en el módulo **ifu\_iccm\_mem** ), un Caché de Instrucciones (I$, implementado en el módulo **ifu\_ic\_mem** ) y la Memoria Externa DDR. Tanto el I$ como el ICCM se controlan desde un controlador de memoria unificado ( **ifu\_mem\_ctl** ), mientras que la memoria externa se controla desde el controlador Lite DRAM. En nuestro sistema RVfpga predeterminado, el ICCM está deshabilitado, pero puede incluirlo fácilmente como se explica en el Laboratorio 20.

Como se muestra en la Figura 3 , la dirección de instrucción (llamada dirección de búsqueda, ifc\_fetch\_addr\_f1 ) se calcula en la primera etapa de búsqueda ( **FC1** ), como se analizará más adelante en la práctica de laboratorio 16. Esta dirección se proporciona al controlador de memoria de instrucciones (implementado en el módulo **ifu\_mem\_ctl** ): fetch\_addr\_f1 = ifu\_fetch\_addr\_f1 .

Las señales suelen tener un prefijo correspondiente a la unidad de la que forman parte. Por ejemplo, " ifu " significa Unidad de obtención de instrucciones. Las señales agregan la etapa con la que están asociadas. Por ejemplo, “ f1 ” indica la etapa FC1.

La instrucción se lee durante la segunda etapa de obtención ( **FC2** ) desde la memoria principal (es decir, la memoria externa DDR) o el ICCM. Si la dirección de la instrucción está dentro del rango de direcciones de la memoria principal, el I$ proporciona la instrucción. Ante un fallo de I$, la tubería debe detenerse hasta que la memoria externa proporcione la instrucción a través del bus AXI, lo que lleva varios ciclos. Si la dirección de la instrucción está dentro del rango de direcciones ICCM, la instrucción se proporciona con baja latencia desde el ICCM a través de un multiplexor implementado dentro del módulo **ifu\_ic\_mem** .

La Memoria de Instrucciones del Sistema RVfpga se configura de la siguiente manera (esta configuración se puede modificar, como mostraremos en futuras prácticas):

* Caché de instrucciones de 16 KiB
* 512 KiB ICCM (deshabilitado): rango de direcciones: 0xEE000000 – 0xEE07FFFF
* Memoria externa de 128 MiB: rango de direcciones: 0x00000000 – 0x07FFFFFF

Si el programa no tiene bloqueos (es decir, sin control, datos o peligros estructurales, sin fallos I$, etc.), se leen cuatro instrucciones de 32 bits (128 bits en total) cada dos ciclos: consulte la señal ifu\_fetch\_data[127:0] . Esto es suficiente para mantener la canalización superescalar bidireccional funcionando a su máximo rendimiento de 2 instrucciones por ciclo. Tres búferes ( q0ff , q1ff y q2ff ) pueden almacenar hasta tres de estos paquetes de 128 bits.

1. **Alinear escenario**

La etapa Align, que sigue a las dos etapas Fetch (consulte la Figura 3 ), se implementa en el módulo **ifu\_aln\_ctl** . La etapa Align es responsable de realizar dos tareas principales:

* **Proporcione dos instrucciones de 32 bits por ciclo a la etapa de decodificación** : la etapa de alineación extrae dos instrucciones por ciclo de los paquetes de 128 bits proporcionados por la memoria de instrucciones y que se almacenan temporalmente en los búferes q0ff , q1ff y q2ff . Estas dos instrucciones se asignan a cada una de las dos vías disponibles en SweRV EH1 a través de las señales ifu\_i0\_instr[31:0] (Vía 0) e ifu\_i1\_instr[31:0] (Vía 1), y luego se almacenan en los dos Registros de Instrucción (IR ) implementado en el módulo **dec\_ib\_ctl** .
* **Instrucciones sin comprimir** : la extensión de instrucciones comprimidas (RVC) de RISC-V reduce el tamaño de las instrucciones de punto flotante y entero común a 16 bits al reducir el tamaño de los campos de control, inmediatos y de registro y al aprovechar los registros redundantes o implícitos. Este tamaño de instrucción reducido disminuye el costo, la potencia y la memoria requerida (consulte la Sección 6.6.5 de DDCARV). La etapa Align se descomprime estas instrucciones de 16 bits, cuando sea necesario, antes de pasarlas a la etapa Decode, que solo decodifica instrucciones de 32 bits . Esto lo realiza el módulo **ifu\_compress\_ctl** , que se instancia dentro del alineador (módulo **ifu\_aln\_ctl** ).

**INSTRUCCIONES COMPRIMIDAS:** El documento complementario, RVfpga\_ SweRVref.docx , explica, en la Sección 5, la ejecución de instrucciones comprimidas en SweRV EH1 y propone algunas tareas nuevas.

1. **Etapas de decodificación, ejecución, confirmación y reescritura**

En esta sección analizamos las etapas de decodificación, ejecución, confirmación y reescritura de la canalización SweRV EH1. La figura 4 ilustra una vista simplificada de estas etapas, que ampliaremos en futuros laboratorios.



Figura 4 . Vista simplificada de las etapas Decode, Execution, Commit y WB

1. **Etapa de decodificación**

Los módulos de Verilog para esta etapa están en la carpeta *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec* . En cada ciclo, la etapa de Decodificación es responsable de dos tareas principales:

* **Decodifique las instrucciones y genere las señales de control** : Las señales de control están organizadas en varios tipos, como se define en *el archivo [RVfpgaPath]* /RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv\_types.sv . Cada estructura/tipo está relacionado con una unidad determinada: ALU ( alu\_pkt\_t ), Unidad de multiplicación ( mul\_pkt\_t ), Unidad de división ( div\_pkt\_t ), Registros ( reg\_pkt\_t ), etc.

**ESTRUCTURAS UTILIZADAS PARA LOS BITS DE CONTROL:** El documento complementario, RVfpga\_ SweRVref.docx , amplía, en la Sección 4, la descripción de las principales estructuras/tipos utilizados en el procesador SweRV EH1 para agrupar las señales de control y propone algunas tareas nuevas. En laboratorios posteriores, nos centraremos en los tipos relacionados con la unidad discutida.

La Unidad de Control, implementada en el módulo **dec\_decode\_ctl** , recibe las dos instrucciones de 32 bits captadas, descomprimidas, alineadas y asignadas a cada vía en las etapas anteriores (señales dec\_i0\_instr\_d[31:0] para la Vía 0 y dec\_i1\_instr\_d[31:0] para la Vía 1) y los decodifica, generando las señales de control para cada instrucción. La figura 5 muestra una vista de alto nivel de la unidad de control (módulo **dec\_decode\_ctl** ), que genera señales de control en dos etapas: los primeros dos módulos ( **i0\_dec** e **i1\_dec** ) usan las instrucciones ( i0 e i1 ) para producir señales de control generales ( i0\_dp y i1\_dp , ambos de tipo dec\_pkt\_t ), y luego la segunda unidad ( **decode** ) usa esas señales para generar señales de control para cada ruta de tubería, también denominadas "tuberías" ( i0\_ap , i1\_ap , lsu\_p , mul\_p , etc.).



Figura 5 . Unidad de control

La unidad de control propaga estas señales de control a las etapas posteriores de la canalización mediante registros de canalización (denominados **Registros de canalización de control** en la Figura 4 ), que se colocan entre cada etapa de la canalización.

* **Distribuya las instrucciones a las canalizaciones apropiadas y proporcione los operandos** : como se muestra en la Figura 4 , SweRV EH1 incluye dos canalizaciones Integer (I0 e I1), una canalización Multiply y una canalización Load/Store (L/S). Además, incluye un Divisor de 34 ciclos que se encuentra fuera de la tubería. Una vez que se decodifica cada instrucción, el procesador la envía a una de cuatro canalizaciones separadas:
  + Aritmética-Lógica y de bifurcación se ejecutan en la tubería I0/I1.
  + Las cargas y almacenes se ejecutan en la tubería L/S.
  + Las instrucciones de multiplicación se ejecutan a través de la tubería Multiply.
  + Instrucciones de división ejecutadas a través de la barra divisoria.

Dado que en cada ciclo se decodifican hasta dos instrucciones, una en la Vía 0 y otra en la Vía 1, ambas se programan para su ejecución siempre que sea posible. Por ejemplo, algunas combinaciones posibles son:

* + Se envían dos instrucciones Aritmético-Lógicas independientes a los conductos I0 e I1.
  + Una instrucción Aritmético-Lógica y una instrucción multiplicar ( mul ) se envían a los conductos I0 (o I1) y Multiplicar, respectivamente.
  + Una instrucción de memoria (cargar o almacenar) se ejecuta en la tubería L/S y una instrucción de multiplicación se ejecuta en la tubería Multiply.

Desafortunadamente, existen algunas situaciones (como los peligros, que analizamos en las prácticas 14 a 16) en las que una o las dos instrucciones deben detenerse. Estas situaciones también se determinan en la etapa de decodificación. Por ejemplo:

* + Si se decodifican dos instrucciones mul en el mismo ciclo, el riesgo estructural se resuelve retrasando la segunda instrucción mul en un ciclo (esto se analizará en detalle en la práctica de laboratorio 14).
  + Si se decodifican dos instrucciones Aritmética-Lógica (AL) dependientes en el mismo ciclo, el peligro de los datos SIN PROCESAR se resuelve retrasando la segunda instrucción AL en un ciclo (esto se analizará más a fondo en la Práctica de laboratorio 15).

Además de programar las instrucciones, las tuberías deben estar provistas de los operandos correspondientes. Para ello, varios multiplexores 3:1 y 4:1 (ver Figura 4 ) seleccionan entre los posibles operandos y los propagan a las siguientes etapas utilizando registros de tubería. Estos multiplexores se implementan en las líneas 279-328 del módulo **exu** (aunque los multiplexores están dentro del módulo **exu** , operan en la etapa de decodificación). Sus operandos de entrada pueden provenir de varios lugares:

* + **Lógica de derivación** : la mayoría de las dependencias de datos se resuelven en la etapa de decodificación por medio de la derivación, como analizaremos en el laboratorio 15. Las entradas provenientes de la lógica de derivación no están etiquetadas en los multiplexores 3: 1 y 4: 1 de la Figura 4 para el En aras de la simplicidad, solo se muestran cables en blanco.
  + **Inmediato** : algunas instrucciones RISC-V utilizan el modo de direccionamiento inmediato, en el que el operando se proporciona directamente desde los bits de instrucción. Las entradas provenientes del Inmediato no se muestran en los multiplexores 3:1 y 4:1 de la Figura 4 – solo se muestra un cable de entrada en blanco) .
  + **Archivo de registro** : el archivo de registro disponible en el procesador SweRV EH1 ( Figura 6 ) tiene 4 puertos de lectura y 3 puertos de escritura (tenga en cuenta que el tercer puerto de escritura se ignora en el archivo de registro incluido en la Figura 4 , ya que solo se usa para situaciones específicas que analizará en futuros laboratorios ). Estos puertos de lectura/escritura permiten la ejecución de dos instrucciones por ciclo. Las entradas provenientes del Archivo de Registro se muestran en los multiplexores 3:1 y 4:1 de la Figura 4 utilizando sólo los nombres de las señales. Las conexiones con el archivo de registro no se muestran por motivos de simplicidad.

Cada puerto de lectura/escritura tiene una dirección de 5 bits ( raddr0 … raddr3 , waddr0 … waddr2 ), así como una señal de habilitación de 1 bit ( rden0 … rden3 , wen0 … wen2 ) que no se muestra en la Figura 4 . Los puertos de escritura también tienen una entrada de datos de escritura de 32 bits ( wd0 … wd2 ) y los puertos de lectura tienen una salida de datos de lectura de 32 bits ( rd0 … rd3 ). El archivo de registro contiene 32 registros de 32 bits, llamados x0-x31, con x0 conectado a 0.



Figura 6 . Archivo de registro disponible en SweRV EH1

**TAREA (OPCIONAL) :** El Archivo de Registro está implementado en el módulo **dec\_gpr\_ctl** y esinstanciado en el módulo **dec** (ver Figura 7 ). Analice tanto el código Verilog como la simulación de las señales principales del módulo **dec\_gpr\_ctl** (disponible en el archivo *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec\_gpr\_ctl.sv* ), para entender cómo funciona. Tenga en cuenta que el procesador SweRV EH1 permite la inclusión de varios Archivos de Registro, pero la configuración utilizada en el Sistema RVfpga solo usa un Archivo de Registro (ver línea 402 del archivo *dec.sv* : localparam GPR\_BANKS = 1; ).

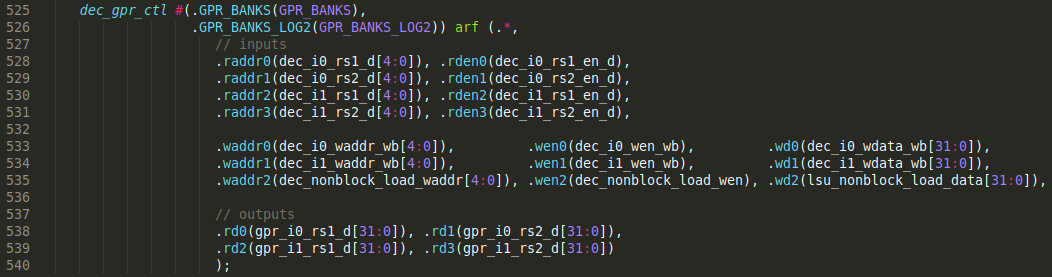


Figura 7 . Registro Instanciación de archivos dentro del módulo dec

1. **Etapas de ejecución**

En esta subsección, analizamos versiones simplificadas de las tuberías disponibles en SweRV EH1: dos tuberías enteras ( tubería I0 y tubería I1 ), una tubería de multiplicación , una tubería de carga/almacenamiento y un divisor de 34 ciclos sin tubería .

Tubos I0/I1 : Los dos tubos enteros se muestran en azul en la Figura 4 . Se dividen en tres etapas denominadas EX1, EX2 y EX3. Cada uno de estos dos conductos incluye una ALU de latencia de 1 ciclo en EX1 , que es capaz de realizar operaciones aritméticas como *suma* o *resta* , así como operaciones lógicas como *y* o *o* . Las etapas EX2 y EX3 realizan pocas tareas pero son necesarias para sincronizar las instrucciones AL con los otros tipos de instrucciones (como cargas, almacenamientos, multiplicaciones, etc.) que requieren tres ciclos para computar sus operaciones. En el laboratorio 12 analizaremos las tuberías I0/I1 con más detalle.

Tubería de multiplicación : La tubería de multiplicación se muestra en rojo en la Figura 4 . Se divide en tres etapas: M1 , M2 y M3 . Esta tubería incluye un multiplicador de 3 ciclos capaz de realizar multiplicaciones enteras. En el laboratorio 14 analizaremos la tubería Multiply con más detalle.

carga/almacenamiento (L/S) : La tubería L/S se muestra en verde en la Figura 4 . En el Laboratorio 13 exploramos esta ruta de canalización en profundidad. Tanto las instrucciones de carga como las de almacenamiento se ejecutan a través de la tubería L/S. Incluye 3 etapas:

* DC1 : En la primera etapa, la Unidad Sumadora calcula la dirección sumando la dirección base del registro y el desplazamiento inmediato.
* DC2 : en la segunda etapa, las instrucciones de carga leen la memoria utilizando la dirección calculada en DC1. Si la dirección se asigna al DCCM, la latencia de acceso es solo de 1 ciclo y la canalización continúa sin paradas. Sin embargo, si el acceso se asigna a la memoria principal, es posible que la canalización deba detenerse durante varios ciclos, según el uso de cargas bloqueantes/no bloqueantes y la existencia de dependencias, como analizaremos en futuros laboratorios.
* DC3 : en la tercera etapa, los datos se alinean y fusionan (por ejemplo, si todavía se está ejecutando un almacenamiento anterior en la misma dirección, es posible que los datos de ese almacenamiento deban reenviarse a la carga). En esta etapa, las instrucciones de almacenamiento comienzan a escribir en la memoria, lo que continuará durante varios ciclos. Si la escritura se asigna al DCCM, tanto los datos como la dirección se almacenan en el búfer de almacenamiento antes de enviarse al DCCM, como analizamos en la práctica de laboratorio 13; si la escritura se asigna a la Memoria Principal, tanto los datos como la dirección se envían a la Memoria Externa a través del bus AXI (el controlador Lite DRAM administra los accesos a esta memoria).

Divisor : El divisor se muestra en blanco en la Figura 4 . Es una unidad no canalizada que requiere hasta 34 ciclos para calcular su resultado. Lab 14 analiza el divisor con más detalle.

Dos multiplexores 3:1 : Al final de la tercera etapa de ejecución (EX3/DC3/M3), como se ilustra en la Figura 4 , el resultado de las instrucciones se selecciona de la tubería adecuada (I0/I1, MUL o L/S ) utilizando dos multiplexores 3:1, uno para cada vía. Estos multiplexores están ubicados en el módulo dec\_decode\_ctl . En la Figura 8 se muestra el multiplexor superior, asociado con la Vía 0 . Las tres entradas de este multiplexor son:

1. **Resultado de la tubería I0:** i0\_resultado\_e3 . Lab 12 analiza este camino.
2. **Resultado de tubería L/S:** lsu\_resultado\_dc3 . Lab 13 analiza este camino.
3. **Resultado de la tubería de multiplicación:** exu\_mul\_resultado\_e3 . Lab 14 analiza este camino.





Figura 8 . Multiplexor 3:1 para seleccionar resultado EX3: diagrama y Verilog

**TAREA (OPCIONAL):** Analice los bits de control del multiplexor de la Figura 8 . Tenga en cuenta que los bits de control están en la señal e3d , que se canalizó a partir de la señal dd , que fue generada en la etapa de decodificación por la unidad de control (consulte RVfpga\_ SweRVref.docx para obtener descripciones de los bits de control).

1. **Etapa de compromiso**

En la etapa de Confirmación, dos multiplexores 3:1, uno por vía, seleccionan el resultado para volver a escribir en el archivo de registro (consulte la Figura 4 ). En la Figura 9 se muestra el multiplexor superior, asociado con la Vía 0 REF \_Ref82184719 \h \\* MERGEFORMAT . Tiene tres entradas:

1. **resultado EX3:** i0\_resultado\_e4 . (La salida del multiplexor 3:1 de EX3).
2. **Datos de lectura corregidos:** lsu\_resultado\_corr\_dc4 . Lab 13 analiza este camino.
3. **Resultado ALU secundario:** exu\_i0\_resultado\_e4 . Estas ALU no se muestran en la Figura 4 por motivos de simplicidad. Como mencionamos anteriormente, permiten que las instrucciones aritméticas y lógicas se repitan cuando sea necesario debido a riesgos de datos (consulte el Laboratorio 15 para obtener más detalles).





Figura 9 . Multiplexor 3:1 para seleccionar el resultado final: diagrama y Verilog

**TAREA (OPCIONAL):** Analice los bits de control del multiplexor de la Figura 9 , que puede encontrar en el módulo **dec\_decode\_ctl** .

1. **Etapa de reescritura**

La etapa final, la etapa Writeback, escribe los resultados en el archivo de registro utilizando los dos primeros puertos de escritura (0 y 1) ilustrados en la Figura 6 (en la práctica de laboratorio 14 veremos cuándo se usa el tercer puerto de escritura, el 2). No todos los ciclos escribirán dos resultados: algunas instrucciones no escriben un registro (es decir, instrucciones de bifurcación, instrucciones de almacenamiento...), y no todos los ciclos ejecutan dos instrucciones. Los identificadores de registro y las señales de habilitación se generaron en la etapa de decodificación y son proporcionados por los registros de canalización de control.

1. **Ejemplo de simulación en Verilator**

En esta sección, ilustramos la simulación de dos instrucciones que se ejecutan en paralelo en la tubería SweRV EH1, mostrando las señales presentadas en las secciones anteriores. Los futuros laboratorios también utilizarán simulaciones de Verilator para visualizar las señales internas del procesador y para ilustrar las explicaciones teóricas.

A continuación, ejecutamos el código de ejemplo que se muestra en la Figura 10 , centrándonos en las instrucciones mul y add (resaltadas en rojo), que forman parte del bucle infinito . La carpeta *[RVfpgaPath]/RVfpga/Labs /Lab11/ExampleProgram* proporciona el proyecto PlatformIO para que pueda analizar, simular y cambiar el programa como desee. Abra el proyecto en PlatformIO y compílelo (recuerde en la Guía de inicio que puede compilar el proyecto haciendo clic en el botón , ubicado en la parte inferior de VSCode ). El archivo de desmontaje (disponible en *[RVfpgaPath]/RVfpga/Labs /Lab11/ExampleProgram /.pio/build/swervolf\_nexys/firmware.dis* ) muestra las direcciones y el código de la máquina. Observe que las dos instrucciones están en las direcciones 0x000000F0 y 0x000000F4:

**0x000000f0: 03de8e33 mult3,t4,t4**

**0x000000f4: 01ff0f33 complemento5,t5,t6**

Estas dos instrucciones están rodeadas de varias instrucciones nop (sin operación) para aislarlas de otras instrucciones y poder analizarlas mejor. La instrucción nop no cambia el estado del sistema. En RISC-V, nop se traduce en addi x0,x0,0 , que se codifica como una instrucción de máquina de 32 bits con el valor de 0x00000013. En este código, definimos varias macros para insertar una cantidad de instrucciones nop (del 1 al 10) en nuestro código (por simplicidad, las definiciones de las macros no se incluyen en la Figura 10 pero se pueden ver en el proyecto PlatformIO) .

Para mayor claridad, deshabilitamos el Branch Predictor y las instrucciones comprimidas, siguiendo el procedimiento que explicamos en la Sección 2 del documento RVfpga\_ SweRVref .

|  |
| --- |
| li x28, 0x1  li x29, 0x2  li x30, 0x4  li x31, 0x1    REPETIR:  **mul x28, x29, x29 # x28 = 2 \* 2 = 4 (iteraciones posteriores: 3\*3=9, 4\*4=16, ...)**  **add x30, x30, x31 # x30 = 4 + 1 = 5 (iteraciones posteriores: 5+1=6, 6+1=7, ...)**  INSERT\_NOPS\_10  add x29, x29, 1 # x29 = x29 + 1  INSERT\_NOPS\_10  beq zero, zero, REPEAT # Repite el ciclo |

Figura 10 . Programa de ejemplo que contiene un mul y agrega instrucciones dentro de un ciclo

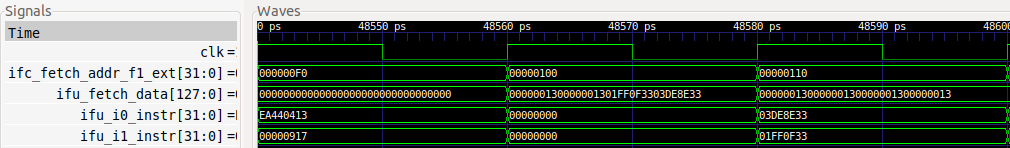
Figuras 11 y 12 muestran las formas de onda de Verilator de las señales del procesador mientras se ejecuta el programa de la Figura 10 . La Figura 11 muestra las señales de las tres primeras etapas de canalización (FC1, FC2 y Align; consulte la Figura 3 ). La Figura 12 muestra las señales de las etapas restantes (ver Figura 4 ) . Dividimos los resultados en dos cifras en aras de la coherencia con la Figura 3 y la figura 4 , pero recuerde que estas dos instrucciones van desde la etapa de alineación (a la derecha de la figura 11 ) hasta la etapa de decodificación (a la izquierda de la figura 12 ).

Las siguientes señales se incluyen en las figuras para rastrear las instrucciones a medida que avanzan a través de la canalización ( ifu para las instrucciones en la etapa de alineación, dec para las instrucciones en la etapa de decodificación, eX para las instrucciones en la etapa X (X = primero, segundo , tercero) Etapa de ejecución, e4 para las instrucciones en la etapa Commit, y wb para las instrucciones en la etapa Writeback) y saber a qué vía están asignadas ( i0 para la Vía 0 e i1 para la Vía 1).

* Instrucciones ifu\_i0\_instr e ifu\_i1\_instr 🡪en la etapa Align
* Instrucciones dec\_i0\_instr\_d y dec\_i1\_instr\_d 🡪en la etapa Decode
* Instrucciones i0\_inst\_e1 e i1\_inst\_e1 🡪en la etapa EX1
* Instrucciones i0\_inst\_e2 e i1\_inst\_e2 🡪en la etapa EX2
* Instrucciones i0\_inst\_e3 e i1\_inst\_e3 🡪en la etapa EX3
* Instrucciones i0\_inst\_e4 e i1\_inst\_e4 🡪en la etapa Commit
* Instrucciones i0\_inst\_wb e i1\_inst\_wb 🡪en la etapa Writeback

**mul t3,t4,t4**

**add t5,t5,t6**



**ALIGN**

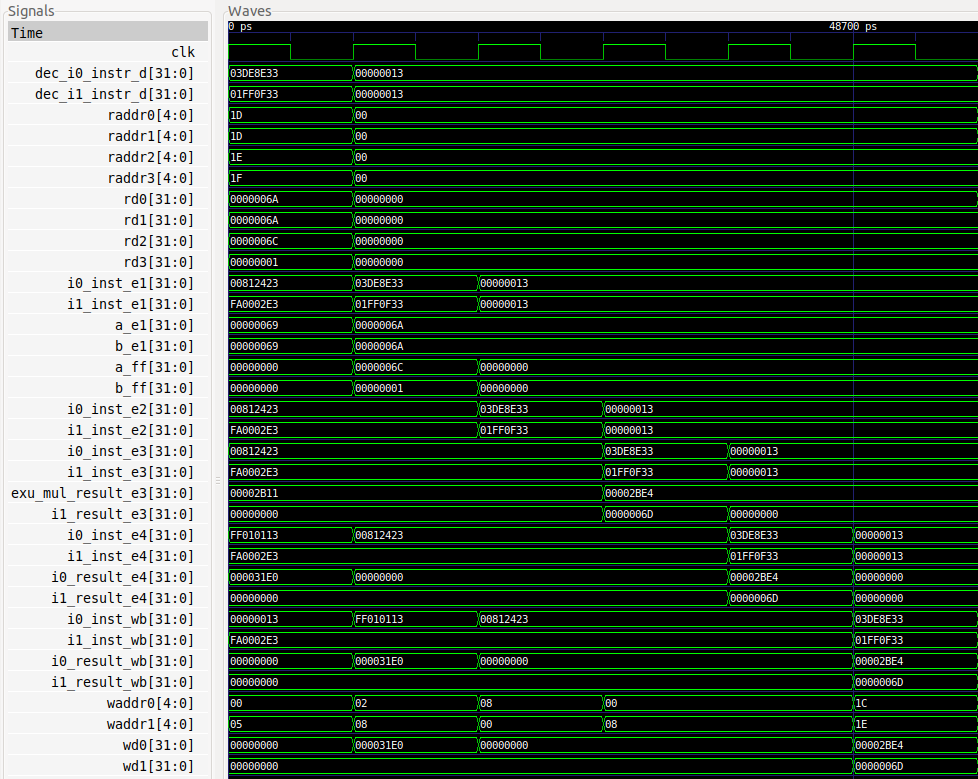
**FC2**

**FC1**

Figura 11 . Simulación de las tres primeras etapas del pipeline: FC1, FC2 y Align

**mul t3,t4,t4**

**add t5,t5,t6**



**WRITEBACK**

**COMMIT**

**EX2/M2**

**EX1/M1**

**EX3/M3**

**DECODE**

Figura 12 . Simulación de las etapas finales: Decode, EX1/M1, EX2/M2, EX3/M3, Commit, Writeback

**TAREA :** Replique la simulación de la Figura 11 y la Figura 12 en su propia computadora siguiendo estos pasos (como se describe en detalle en la Sección 7 de la GSG):

* Si es necesario, genere el binario de simulación ( *Vrvfpgasim* ).
* En PlatformIO, abra el proyecto provisto en: *[RVfpgaPath]/RVfpga/Labs /Lab11/ExampleProgram* .
* Establezca la ruta correcta al binario de simulación RVfpga ( *Vrvfpgasim* ) en el archivo *platformio.ini* .
* Genere el seguimiento de la simulación con Verilator (Generar seguimiento).
* Abra la traza usando GTKWave.
* Utilice los archivos *test\_1.tcl* y *test\_2.tcl* (proporcionados en *[RVfpgaPath]/RVfpga/Labs /Lab11/ExampleProgram* ) para abrir las mismas señales que se muestran en la Figura 11 y la Figura 12 . Para ello, en GTKWave, haga clic en *Archivo → Leer archivo de script Tcl* y seleccione el archivo *test\_1.tcl* o *test\_2.tcl* .
* Haga clic en *Acercar* ( ) varias veces y muévase a 48600ps (o cualquier otra iteración del ciclo, excepto la primera).

Analice la forma de onda de la Figura 11 y la Figura 12 y los diagramas de la Figura 3 y la Figura 4 al mismo tiempo. Las figuras incluyen algunas señales asociadas con cada una de las etapas del oleoducto. Los valores resaltados en rojo corresponden a las dos instrucciones ( mul y add ) a medida que fluyen a través de la tubería.

* **FC1** : En el primer ciclo de la Figura 11 , la señal ifc\_fetch\_addr\_f1\_ext[31:0] (el contador de programa, que se proporciona a la memoria de instrucciones) contiene la dirección de (es decir, *apunta* a) la instrucción mul ( ifc\_fetch\_addr\_f1\_ext = 0x000000F0 ).
* **FC2** : En el segundo ciclo de la Figura 11 , la Memoria de Instrucciones proporciona una nueva señal de 128 bits que incluye las dos instrucciones que estamos analizando en el ejemplo ( se muestra mul en verde y add en rojo):

ifu\_fetch\_data = 0x0000001300000013 01FF0F33 03DE8E33

* **Align** : En el ciclo final de la Figura 11 , las dos instrucciones se extraen de la nueva señal de 128 bits y se distribuyen a las dos vías que incluye SweRV EH1.

ifu\_i0\_instr = 0x03DE8E33 (Vía 0)

ifu\_i1\_instr = 0x01FF0F33 (Vía 1)

* **Decodificación** : en el primer ciclo de la Figura 12 , se decodifican las dos instrucciones, es decir, los valores de registro de las instrucciones se leen del archivo de registro y se generan los bits de control (no se muestran en la figura, pero puede agregar algunos de ellos). como se describe en RVfpga\_ SweRVref .docx). Los operandos (valores de registro) se colocan en rd0 , rd1 , rd2 y rd3 .

rd0 = 0x0000006A

rd1 = 0x0000006A

rd2 = 0x0000006C

rd3 = 0x00000001

* **EX1/M1** , **EX2/M2** , **EX3/M3** y **Commit** : En los siguientes tres ciclos de la Figura 12 , se realizan la suma y la multiplicación. Al final de EX3/M3, los resultados se seleccionan utilizando los dos multiplexores 3:1 y luego se propagan a la etapa de confirmación.

i0\_resultado\_e4 = exu\_mul\_resultado\_e3 = 0x6A \* 0x6A = 0x2BE4

i1\_resultado\_e4 = i1\_resultado\_e3 = 0x6C + 0x01 = 0x6D

* **Writeback** : en el ciclo final de la Figura 12 , los resultados se vuelven a escribir en el archivo de registro.

waddr0 = 0x1C wd0 = 0x2BE4

waddr1 = 0x1E wd1 = 0x6D

# CONTADORES DE HARDWARE EN SweRV EH1

Ahora mostramos cómo usar contadores de rendimiento para analizar el rendimiento del procesador. Los contadores de hardware son un conjunto de registros de propósito especial incluidos en la mayoría de los procesadores actuales para registrar una variedad de métricas, como la cantidad de instrucciones ejecutadas, la cantidad de ciclos ejecutados, el promedio de ciclos de reloj por instrucción (CPI), la cantidad de Instrucción Aciertos/errores de caché, el número de bifurcaciones correctas/incorrectas previstas, etc.

En Labs 12-20 usaremos regularmente los contadores de rendimiento disponibles en SweRV EH1 para medir y comparar las diferentes magnitudes.

**BENCHMARKS REALES:** En la carpeta *[RVfpgaPath]/RVfpga/Labs /Lab20/RealBenchmarks* proporcionamos tres aplicaciones reales (CoreMark, Dhrystone e Image Processing) que utilizará en Lab 20 para probar las diferentes funciones de nuestro procesador SweRV EH1. El documento complementario, RVfpga\_ SweRVref.docx , describe brevemente estas aplicaciones en la Sección 6, y el Laboratorio 20 amplía estas descripciones y propone varias tareas.

1. **Contadores de rendimiento en SweRV EH1**

El Manual de referencia del programador RISC-V SweRV EH1 ( <https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf>) describe las capacidades básicas de monitoreo del rendimiento del hardware de un procesador RISC-V. Se deben implementar los siguientes contadores de rendimiento, que también son registros de control y estado (CSR):

* *mcycle* : número de ciclos de reloj que ha ejecutado el hart (hilo de hardware) desde un tiempo arbitrario en el pasado.
* *minstret* : número de instrucciones que el ciervo ha retirado desde algún tiempo arbitrario en el pasado.
* *mhpmcounter3* – *mhpmcounter31* : 29 otros contadores de eventos. Los CSR del selector de eventos, *mhpmevent3* – *mhpmevent31* , son registros WARL (escribir cualquier valor, leer valores legales) que controlan qué evento hace que se incremente el contador correspondiente. El significado de estos eventos lo define la plataforma, pero el evento 0 está reservado para significar "sin evento".

No es necesario implementar todos los contadores. Es una implementación legal cablear tanto el contador como su correspondiente selector de eventos a 0. Específicamente, en SweRV EH1, solo los contadores de eventos 3 a 6 ( *mhpmcounter3* - *mhpmcounter6* ) y sus correspondientes selectores de eventos ( *mhpmevent3* - *mhpmevent6* ) son funcionales, mientras que los contadores de eventos 7 a 31 ( *mhpmcounter7* - *mhpmcounter31* ) y sus correspondientes selectores de eventos ( *mhpmevent7* - *mhpmevent31* ) están cableados a '0'. La habilitación de estos contadores se controla mediante el bit 0 del registro *mgpmc* (0 = deshabilitar, 1 = habilitar).

El Capítulo 7 del Manual de referencia del programador de SweRV EH1 ( <https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf>) describe en detalle las características y el funcionamiento de los cuatro contadores de rendimiento disponibles en SweRV EH1:

* Cuatro contadores de eventos estándar de 64 bits de ancho
* Selección estándar de eventos separados para cada contador
* Capacidad de control de activación/desactivación de conteo selectivo estándar
* Contador sincronizado activar/desactivar controlabilidad
* Contador de ciclos estándar
* Contador de instrucciones retiradas estándar
* Compatibilidad con registros de temporizador de máquina basados en SoC estándar

La Tabla 7-2 de ese documento enumera los 50 eventos contables disponibles en SweRV EH1, que se resumen en la Tabla 1 .

Tabla 1 . Lista de eventos contables en SweRV EH1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | Reservado | 17 | CSR lectura/escritura | 34 | Ciclos SB/WB estancados |
| 1 | Reloj de ciclos activo | 18 | CSR escribir rd==0 | 35 | Cycles DMA DCCM transacción estancada |
| 2 | Aciertos de I-Cache | 19 | descanso | 36 | Cycles DMA ICCM transacción estancada |
| 3 | I-Cache falla | 20 | Llamada electrónica | 37 | excepciones tomadas |
| 4 | Instrs comprometidos | 21 | Valla | 38 | Interrupciones de temporizador tomadas |
| 5 | Instrs comprometidos 16-b | 22 | Valla.i | 39 | Interrupciones externas tomadas |
| 6 | Instrs comprometidos 32-b | 23 | Mret | 40 | Descargas de TLU |
| 7 | Instrumentos alineados | 24 | Ramas comprometidas | 41 | Descargas de error de rama |
| 8 | Instrumentos decodificados | 25 | Ramas mal predichas | 42 | Transacciones I-bus – instr |
| 9 | Mules comprometidos | 26 | Ramas tomadas | 43 | Transacciones D-bus – ld/st |
| 10 | Divs comprometidos | 27 | Ramas impredecibles | 44 | Transacciones D-bus desalineadas |
| 11 | Cargas comprometidas | 28 | Recuperación de ciclos estancada | 45 | Errores de bus I |
| 12 | Tiendas comprometidas | 29 | Alineador de ciclos atascado | 46 | Errores de bus D |
| 13 | Cargas desalineadas | 30 | Decodificación de ciclos estancada | 47 | Ciclos estancados debido a I-bus ocupado |
| 14 | Tiendas desalineadas | 31 | Postsincronización de ciclos estancada | 48 | Ciclos detenidos debido a D-bus ocupado |
| 15 | Alus se comprometió | 32 | Presincronización de ciclos estancada | 49 | Interrupciones de ciclos deshabilitadas |
| dieciséis | lectura de RSE | 33 | Ciclos congelados | 50 | Interrupciones de ciclos estancadas mientras están deshabilitadas |

1. **Uso de los contadores de rendimiento por medio del Processor Support Package (PSP) de Western Digital**

Usar el sistema de monitoreo de desempeño a nivel de registro sería un poco complejo; afortunadamente, el PSP de WD ( <https://github.com/westerndigitalcorporation/riscv-fw-infrastructure>) incluye varias funciones que brindan un enfoque mucho más simple para monitorear el rendimiento. Si ha instalado PlatformIO siguiendo las instrucciones del GSG, debería encontrar los dos archivos siguientes en su sistema Ubuntu:

* ~/.platformio/packages/framework-wd-riscv-sdk/psp/psp\_performance\_monitor\_eh1.c
* ~/.platformio/packages/framework-wd-riscv-sdk/psp/api\_inc/psp\_performance\_monitor\_eh1.h

**Windows :** la carpeta *.platformio* se encuentra dentro de su carpeta de usuario (C:\Users\<USUARIO>). Tenga en cuenta que es posible que deba habilitar el sistema para ver archivos/carpetas ocultos.

**macOS :** como en Linux, la carpeta *.platformio* se encuentra dentro de su carpeta de inicio ( *~/.platformio* ).

El archivo .c ( *psp\_performance\_monitor\_eh1.c* ) implementa funciones que le permiten hacer cosas como activar/desactivar el monitor de rendimiento de grupo ( pspEnableAllPerformanceMonitor ), emparejar un contador con un evento ( pspPerformanceCounterSet ) u obtener el valor del contador ( pspPerformanceCounterGet ).

El archivo .h ( *psp\_performance\_monitor\_eh1.h* ) proporciona nombres para cada uno de los eventos de la Tabla 1 en: typedef enum pspPerformanceMonitorEvents .

El siguiente ejemplo ( Figura 13 ), proporcionado en *[RVfpgaPath]/RVfpga/Labs /Lab11/HwCounters\_Example* , ilustra el uso de los cuatro contadores de hardware disponibles en SweRV EH1 para medir: *ciclos* , *instrucciones* y *bifurcaciones comprometidas* y *mal previstas* . La función principal :

* Inicializa el UART ( uartInit() )
* Habilita los contadores de hardware ( pspEnableAllPerformanceMonitor(1) )
* Asigna los eventos que se van a medir ( *ciclos* , *instrucciones* y *bifurcaciones comprometidas* y mal previstas ) a cada contador ( *D\_PSP\_COUNTER0* - D\_PSP\_COUNTER3 )
* Lee los contadores ( pspPerformanceCounterGet(D\_PSP\_COUNTER0) )
* Llama a un programa de ensamblaje simple ( Test\_Assembly() ) y lee los contadores nuevamente
* Imprime el valor de cada contador usando la función printfNexys .

El montaje\_de\_prueba() función, después de algunas inicializaciones de registro, repite un ciclo 1,000,000 veces; el bucle contiene cinco instrucciones de lógica aritmética (AL) y una rama condicional. El archivo de desensamblado también se muestra al final de la Figura 13 para que sepa el valor de las instrucciones de máquina de 32 bits que componen el cuerpo del bucle.

|  |
| --- |
| **Archivo Prueba.C**  #si está definido(D\_NEXYS\_A7) #incluye <bsp\_printf.h> #incluye <bsp\_mem\_map.h> #incluye <bsp\_version.h>#else PRE\_COMPILED\_MSG("no se definió ninguna plataforma")#endif  #incluye <psp\_api.h>  vacío externo Test\_Assembly (vacío);  int principal (vacío) { int cyc\_beg, cyc\_end; int instr\_beg, instr\_end; int BrCom\_inicio, BrCom\_fin; int BrMis\_beg, BrMis\_end;  /\* Inicializar Uart \*/  uartInit();  **pspEnableAllPerformanceMonitor** (1);  **pspPerformanceCounterSet** (D\_PSP\_COUNTER0, E\_CYCLES\_CLOCKS\_ACTIVE ); **pspPerformanceCounterSet** (D\_PSP\_COUNTER1, E\_INSTR\_COMMITTED\_ALL ); **pspPerformanceCounterSet** (D\_PSP\_COUNTER2, E\_BRANCHES\_COMMITTED ); **pspPerformanceCounterSet** (D\_PSP\_COUNTER3, E\_BRANCHES\_MISPREDICTED );  cyc\_beg = **pspPerformanceCounterGet** (D\_PSP\_COUNTER0); instr\_beg = **pspPerformanceCounterGet** (D\_PSP\_COUNTER1); BrCom\_beg = **pspPerformanceCounterGet** (D\_PSP\_COUNTER2); BrMis\_beg = **pspPerformanceCounterGet** (D\_PSP\_COUNTER3); **Montaje\_de\_prueba()** ;  cyc\_end = **pspPerformanceCounterGet** (D\_PSP\_COUNTER0); instr\_end = **pspPerformanceCounterGet** (D\_PSP\_COUNTER1); BrCom\_end = **pspPerformanceCounterGet** (D\_PSP\_COUNTER2); BrMis\_end = **pspPerformanceCounterGet** (D\_PSP\_COUNTER3);  printfNexys("Ciclos = %d", cyc\_end-cyc\_beg); printfNexys("Instrucciones = %d", instr\_end-instr\_beg); printfNexys("BrCom = %d", BrCom\_end-BrCom\_beg); printfNexys("BrMis = %d", BrMis\_end-BrMis\_beg);  mientras(1);  }  **Archivo Test\_Assembly.S**  .globl Test\_Asamblea  .texto  Asamblea\_de\_prueba:  li t1, 0x1 li t3, 0x3 li t4, 0x4 li t5, 0x5 li t6, 0x6 li a0, 0x0lui a1, 0xF4add a1, a1, 0x240nop  **REPETIR: agregar a0, a0, 1 agregar t3, t3, t1 sub t4, t4, t1 o t5, t5, t1 xor t6, t6, t1 bne a0, a1, REPETIR # Repite el ciclo**  .final  **Archivo firmware.dis**  000001e4 <ensamblaje\_de\_prueba>:  1e4: 00100313 lit1,1  1e8: 00300e13 lit3,3  1ec: 00400e93 lit4,4  1f0: 00500f13 lit5,5  1f4: 00600f93 lit6,6  1f8: 00000513 lia0,0  1fc: 000f45b7 luia1,0xf4  200: 24058593 addia1,a1,576 # f4240 <\_sp+0xf0788>  204: 00000013 no  00000208 <REPETIR>:  208: **00150513** addia0,a0,1  20c: **006e0e33** suma3,t3,t1  210: **406e8eb3** subt4,t4,t1  214: **006f6f33**  o t5,t5,t1  218: **006fcfb3**  xor t6,t6,t1  21c: **feb516e3** bnea0,a1,208 <REPETIR> |

Figura 13 . Test.C, Test\_Assembly.S y firmware.dis

**TAREA :** Ejecute el programa de la Figura 13 en la placa Nexys A7 como se explica en la Guía de inicio. Debería obtener los resultados que se muestran en la Figura 14 para los cuatro eventos medidos. Explique y justifique los resultados.

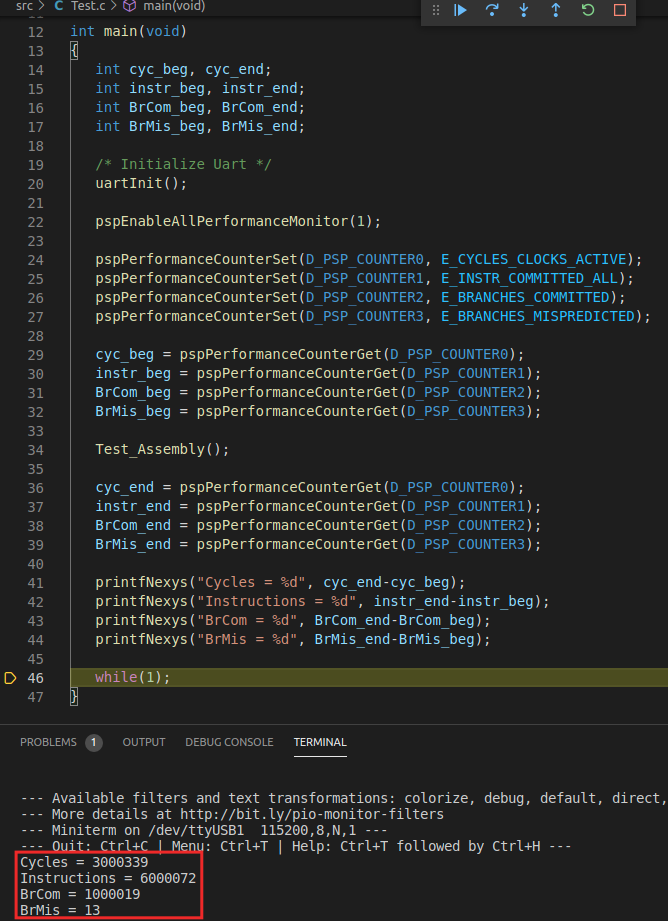


Figura 14 . Ejecución de Test.C

**TAREA (OPCIONAL) :** Medir otros eventos en los Contadores de hardware para el programa de la Figura 13 . Para ello, debe cambiar en el archivo *Test.c* la configuración de los eventos a medir con la función pspPerformanceCounterSet . Tenga en cuenta que los diferentes eventos (que se muestran en la Tabla 1 ) se pueden configurar mediante las macros definidas en el archivo PSP de WD: *.platformio/packages/framework-wd-riscv-sdk/psp/api\_inc/psp\_performance\_monitor\_eh1.h* . Por ejemplo, si desea medir el número de errores de I$ en lugar del número de errores de rama, debe sustituir en la línea del archivo *Test.c* : **pspPerformanceCounterSet** (D\_PSP\_COUNTER3, E\_BRANCHES\_MISPREDICTED );

para la línea: **pspPerformanceCounterSet** (D\_PSP\_COUNTER3, E\_I\_CACHE\_MISSES );

**TAREA (OPCIONAL):** Proponer otros programas en la función Test\_Assembly y comprobar si los diferentes eventos dan los resultados esperados. Puedes probar otras instrucciones como cargas, almacenes, multiplicaciones, divisiones… así como peligros que provoquen atascos de tuberías.