

**EL PROGRAMA UNIVERSITARIO DE LA IMAGINACIÓN**

**Referencia SweRV EH1**

## **Jerarquía, Módulos, Señales y Tipos**

Este documento proporciona instrucciones adicionales sobre los siguientes temas:

* Sección 1: **Estudio Sigasi**
* Sección 2: **Configuración del procesador SweRV EH1**
* Sección 3: **Jerarquía de módulos del Sistema RVfpga y sus señales más relevantes**
* Sección 4: **Principales estructuras/tipos para agrupar bits de control**
* Sección 5: **Instrucciones comprimidas de RISC-V**
* Sección 6: Puntos de **referencia reales**

# 1. ESTUDIO SIGASI

Sigasi Studio mejora la productividad del diseñador al ayudar a escribir, inspeccionar y modificar diseños de circuitos digitales de la manera más intuitiva. Esta herramienta comprende el contexto del diseño. Las funciones avanzadas, como el autocompletado inteligente y la refactorización de código, hacen que el diseño de VHDL, Verilog y SystemVerilog sea más fácil y eficiente.

Sigasi Studio requiere una tarifa para obtener una licencia y poder usarla profesionalmente. Afortunadamente, existe una licencia gratuita con fines educativos que puede obtener fácilmente en: <https://www.sigasi.com/try-form-edu/>. Una vez que complete sus datos y su licencia sea aprobada, recibirá un correo electrónico con las instrucciones y un enlace para descargar ( <https://www.sigasi.com/download/>, ver Figura 1 ), instalar y usar Estudio Sigasi. El software está disponible para Windows, Linux y MacOS.

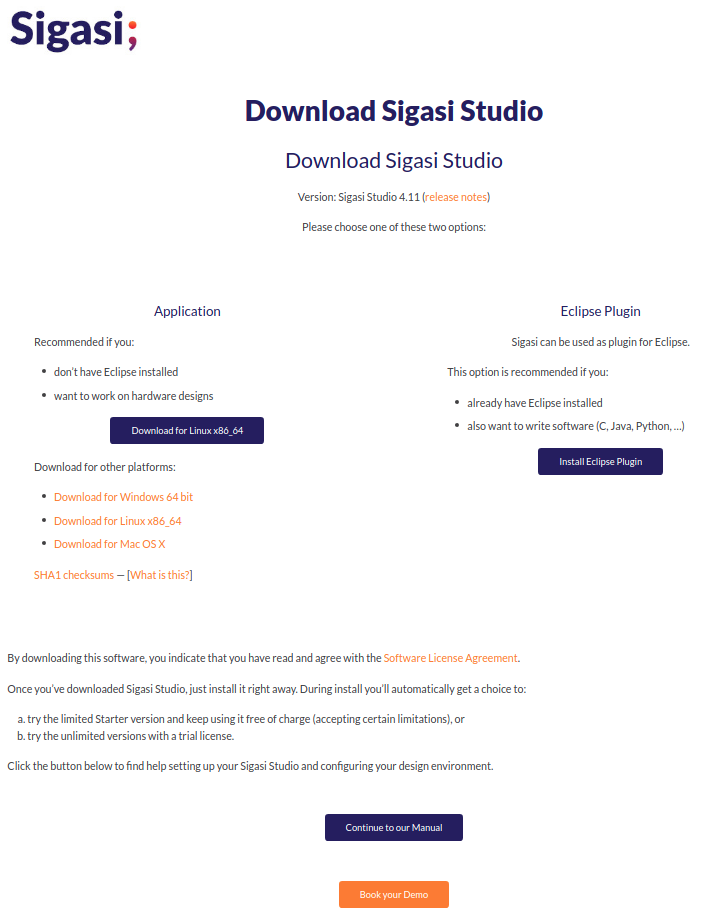


Figura 1 . Enlace para descargar, instalar y usar Sigasi Studio

Una vez que haya instalado Sigasi Studio en su sistema, puede comenzar a usarlo para inspeccionar RVfpga. En el siguiente enlace, hace dos años, Hendrik Eeckhaut publicó instrucciones para crear y configurar un proyecto para SweRV EH1: <https://insights.sigasi.com/tech/swerv_riscv/>. Usando esa información como punto de partida, a continuación proporcionamos instrucciones completas para crear y configurar un proyecto para RVfpga.

1. Cree una copia del *directorio [RVfpgaPath]/RVfpga/src* y asígnele el nombre *[RVfpgaPath]/RVfpga/src\_SigasiStudio*
2. Abra Sigasi Studio yendo al directorio descargado y haciendo doble clic en el archivo *sigasi\_internal* (vea la Figura 2 ).

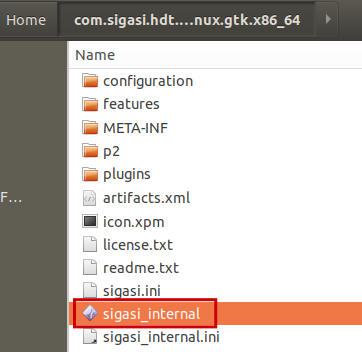


Figura 2 . Estudio Sigasi Abierto

1. En la ventana de Sigasi Studio, haga clic en Archivo → Importar… Se abrirá una nueva ventana que le pedirá que seleccione el tipo de proyecto que desea agregar a su sistema. Elija "Importar un proyecto Verilog (de sistema)" y haga clic en Siguiente ( Figura 3 ).

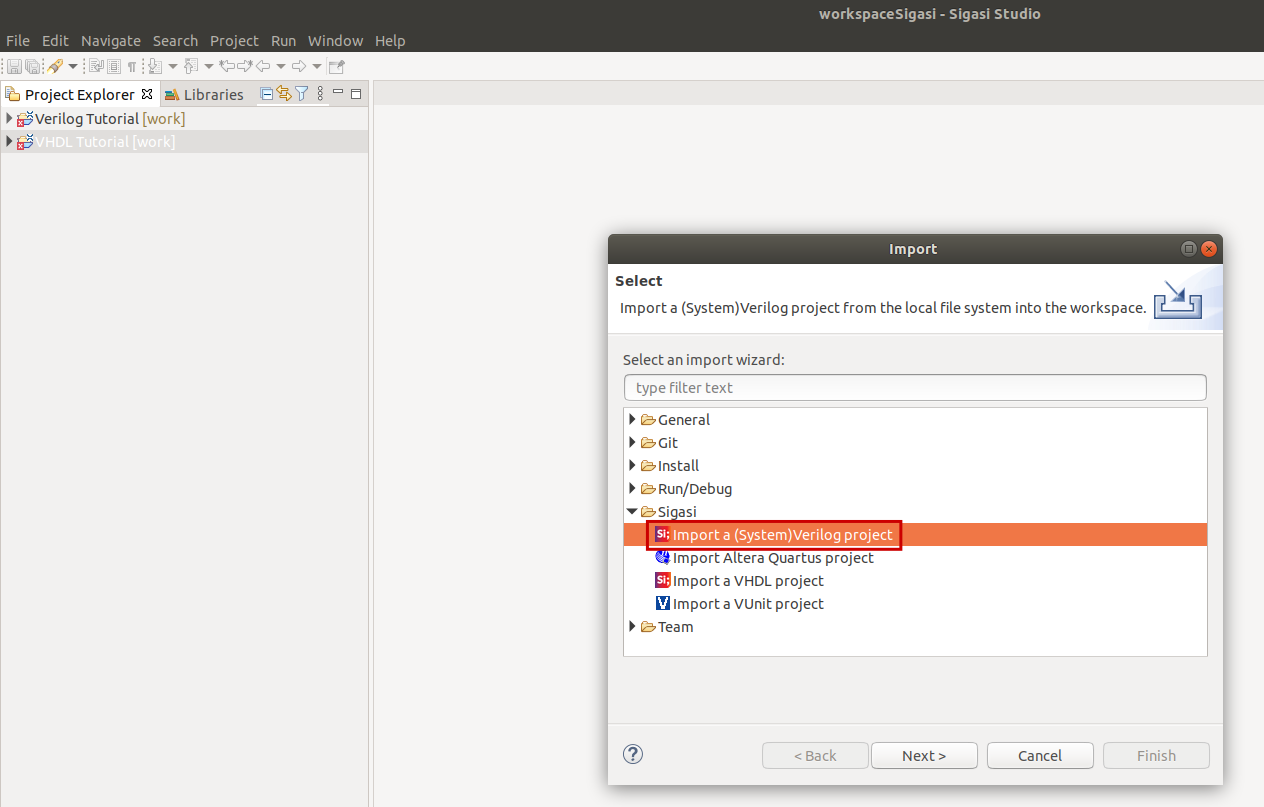


Figura 3 . Importar el proyecto RVfpga

1. Ahora haga clic en "Examinar..." y navegue y seleccione el directorio *src\_SigasiStudio* y haga clic en Abrir (vea la Figura 4 ) y luego haga clic en Finalizar.

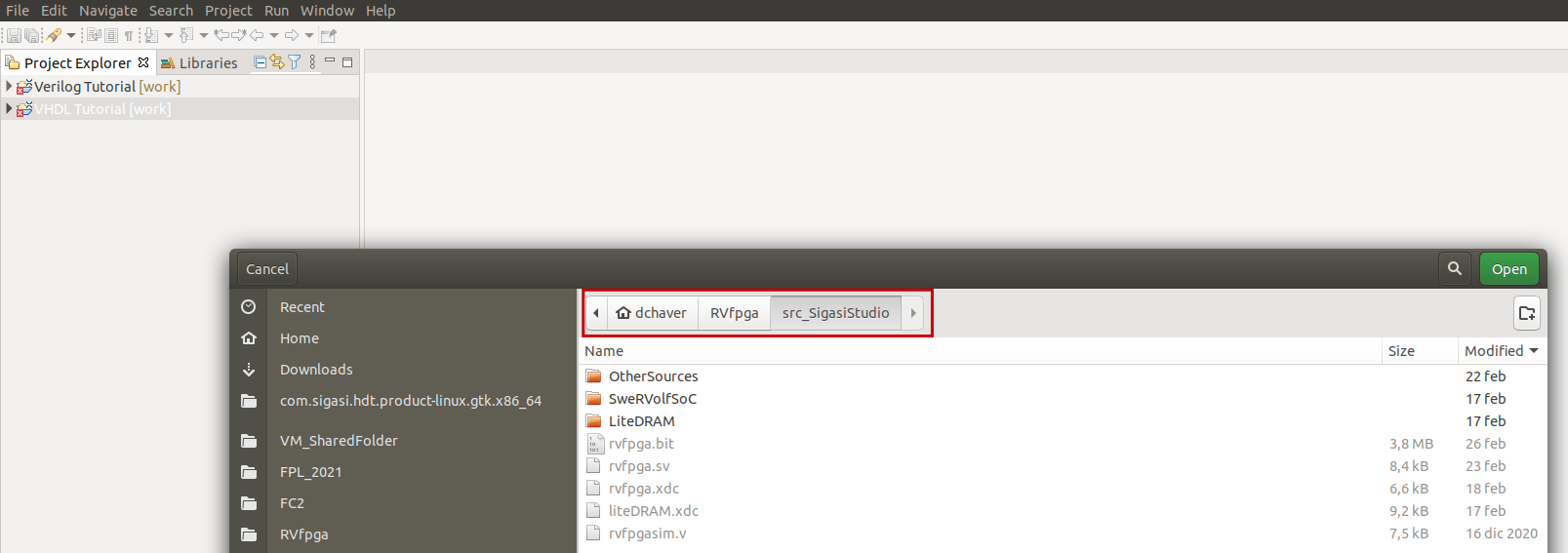


Figura 4 . Abra el directorio fuente de RVfpga

1. El proyecto se abrirá con muchos errores (consulte la Figura 5 ), la mayoría debido a la falta de muchos archivos de inclusión en la configuración del proyecto.

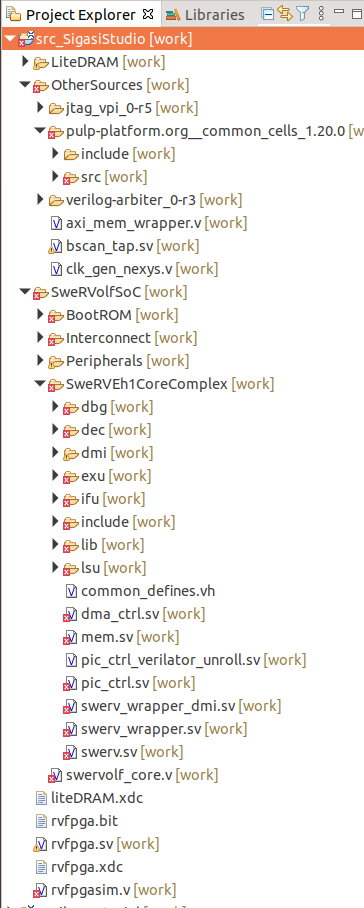


Figura 5 . Errores iniciales en el Proyecto RVfpga Sigasi Studio.

1. En el Explorador de proyectos, haga clic con el botón derecho en el proyecto *src\_SigasiStudio* y abra la ventana Propiedades (consulte la Figura 6 ).

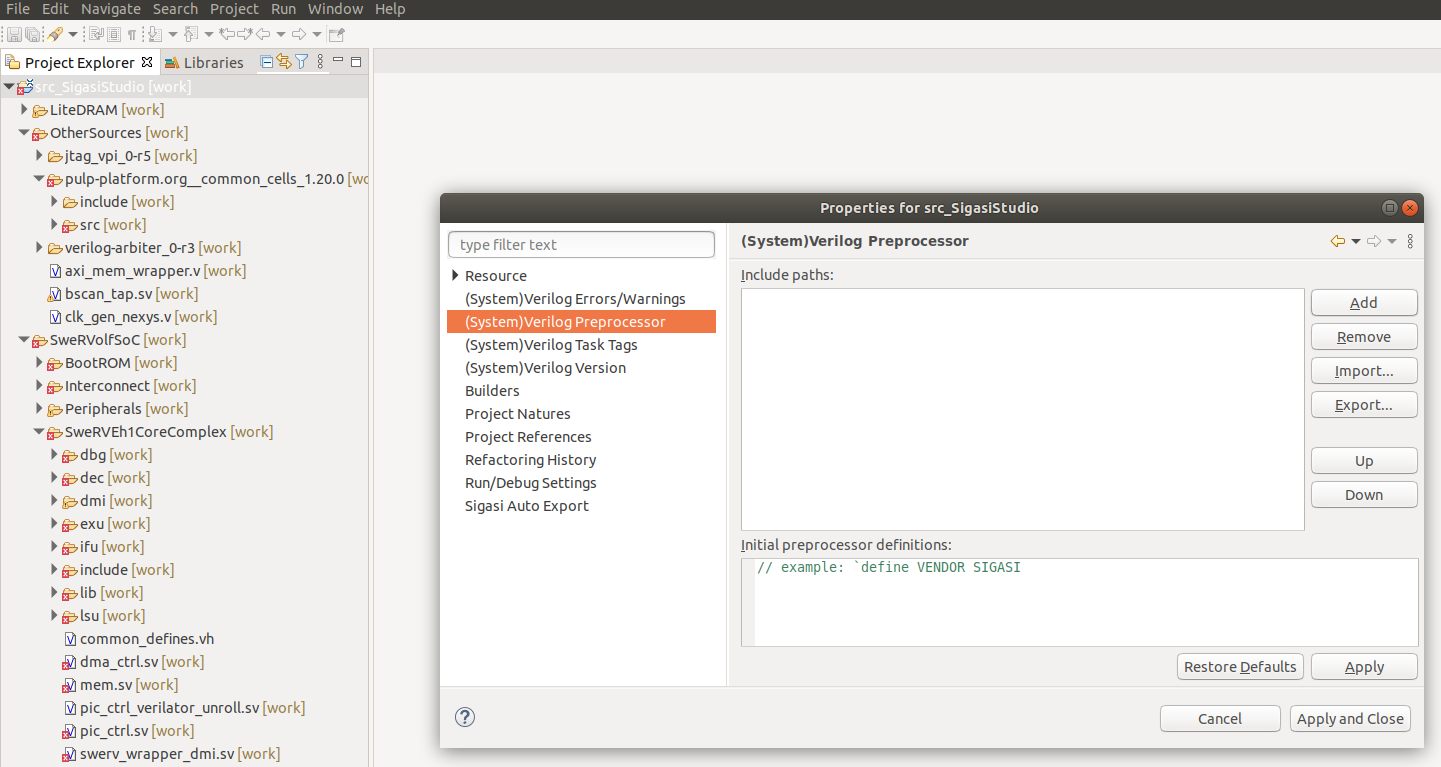


Figura 6 . propiedades del proyecto.

1. En la ventana Propiedades ( Figura 6 ), seleccione el "Preprocesador (Sistema) Verilog" y agregue las siguientes rutas de inclusión (haciendo clic en el botón Agregar a la derecha):
   * *[RVfpgaPath]/ RVfpga/src\_SigasiStudio/SweRVolfSoC/SweRVEh1CoreComplex/include*
   * *[RVfpgaPath]/RVfpga/src\_SigasiStudio/OtherSources/pulp-platform.org\_\_common\_cells\_1.20.0/include*
   * *[RVfpgaPath]/RVfpga/src\_SigasiStudio/SweRVolfSoC/Interconnect/AxiInterconnect/pulp-platform.org\_\_axi\_0.25.0/include*
   * *[RVfpgaPath]/RVfpga/src\_SigasiStudio/SweRVolfSoC/Interconexión/AxiInterconexión*
   * *[RVfpgaPath]/RVfpga/src\_SigasiStudio/SweRVolfSoC/Interconnect/WishboneInterconnect*

Una vez que se hayan agregado los cinco directorios, haga clic en el botón Aplicar.

Luego, en la misma ventana, en el cuadro inferior (Definiciones iniciales del preprocesador), ingrese la siguiente línea: `include "common\_defines.vh" . Haga clic en el botón Aplicar y cerrar.

La figura 7 muestra el estado final.

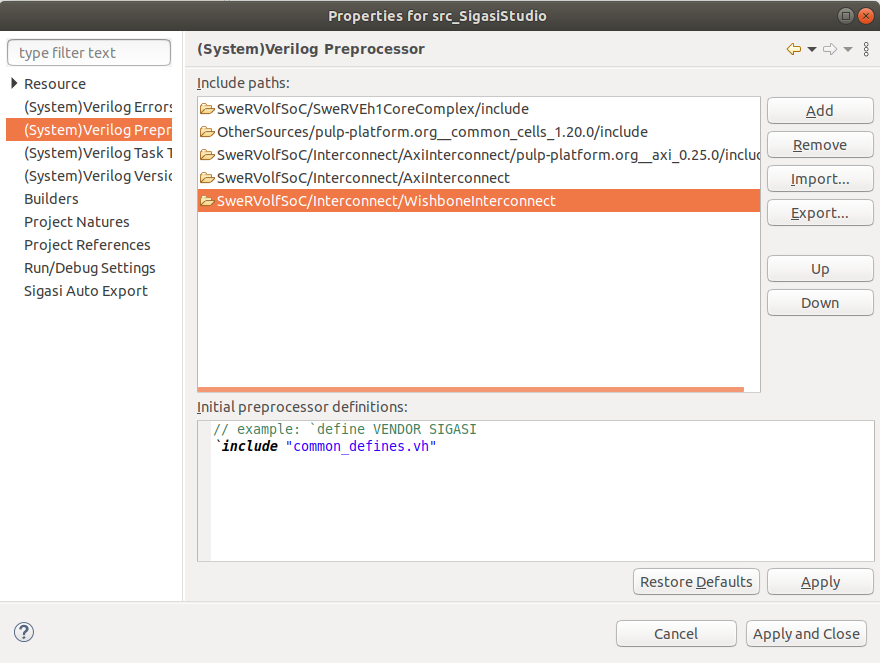


Figura 7 . Incluir directorios y archivos

1. Finalmente, elimine el archivo *[RVfpgaPath]/ RVfpga/src\_SigasiStudio/SweRVolfSoC/BootROM/sw/boot\_main.vh* , que no necesitamos para nuestro proyecto y da algunos errores. Puede eliminarlo en su Explorador de archivos o dentro de Sigasi Studio.

Todos los errores deberían haber desaparecido después de estos pasos y solo deberían quedar algunas advertencias que puedes ignorar.

Puede comenzar a usar Sigasi Studio para inspeccionar el SoC RVfpga. A modo de prueba, mostramos a continuación algunas funcionalidades de la herramienta:

1. En el menú superior, abra Ventana → Mostrar vista → Diagrama de bloques, que abre una nueva ventana en la parte derecha de la herramienta que le permite navegar gráficamente por el módulo.
2. En este laboratorio analizamos instrucciones aritméticas y lógicas. Estas instrucciones se ejecutan en la ALU, que se implementa dentro del módulo **exu\_alu\_ctl** . Abra ese módulo haciendo doble clic en él en la ventana Project Explore. Debería ver lo que mostramos en la Figura 8 .

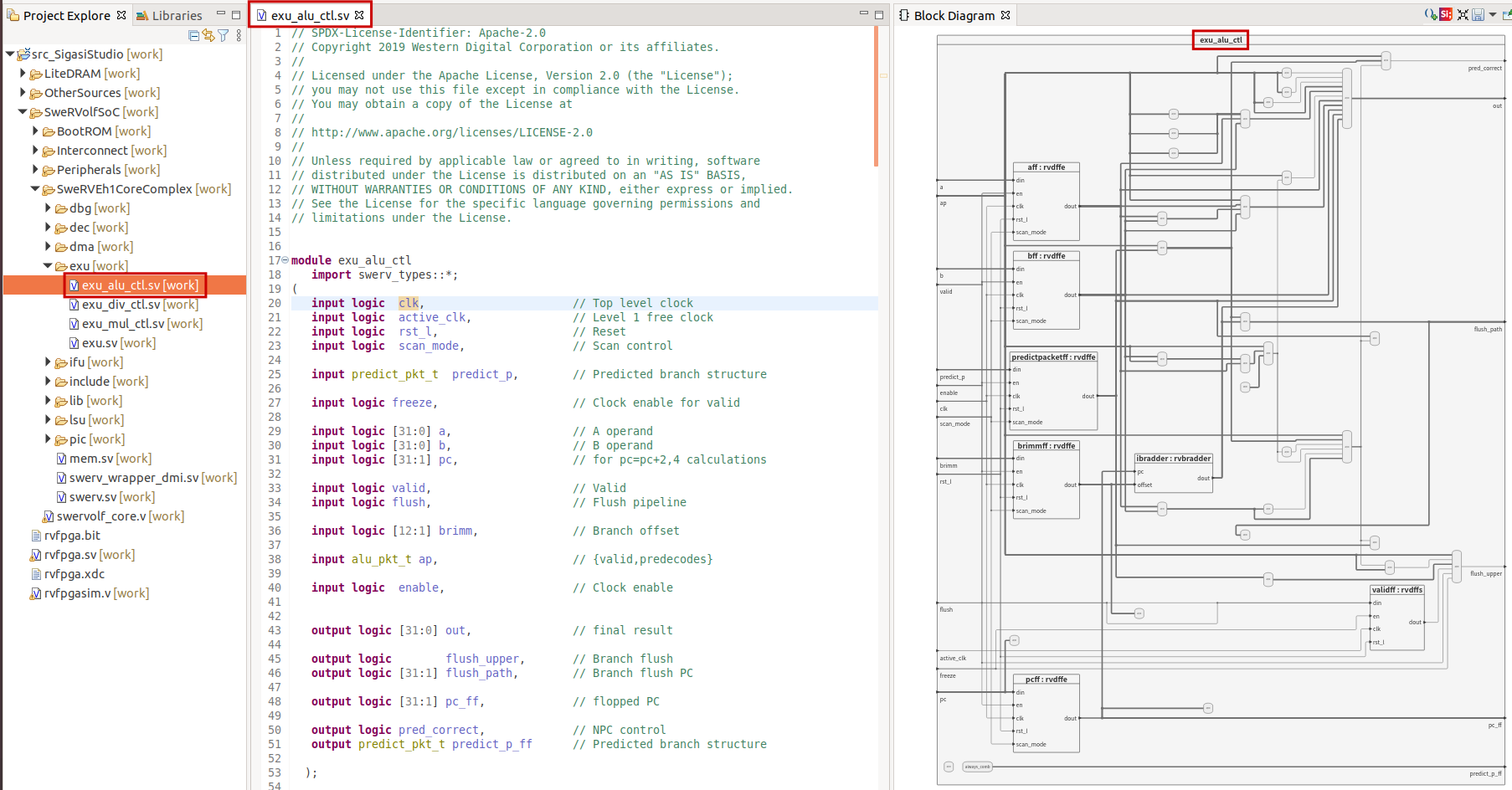


Figura 8 . Archivo *exu\_alu\_ctl.sv* : código Verilog y diagrama de bloques

1. Puede resaltar una señal en el diagrama haciendo clic derecho sobre ella en el código Verilog y seleccionando Mostrar en → Diagrama de bloques. Los cables asociados con la señal se resaltarán en la ventana del Diagrama de bloques, como se muestra en la Figura 9 , donde se resalta el paquete ap .

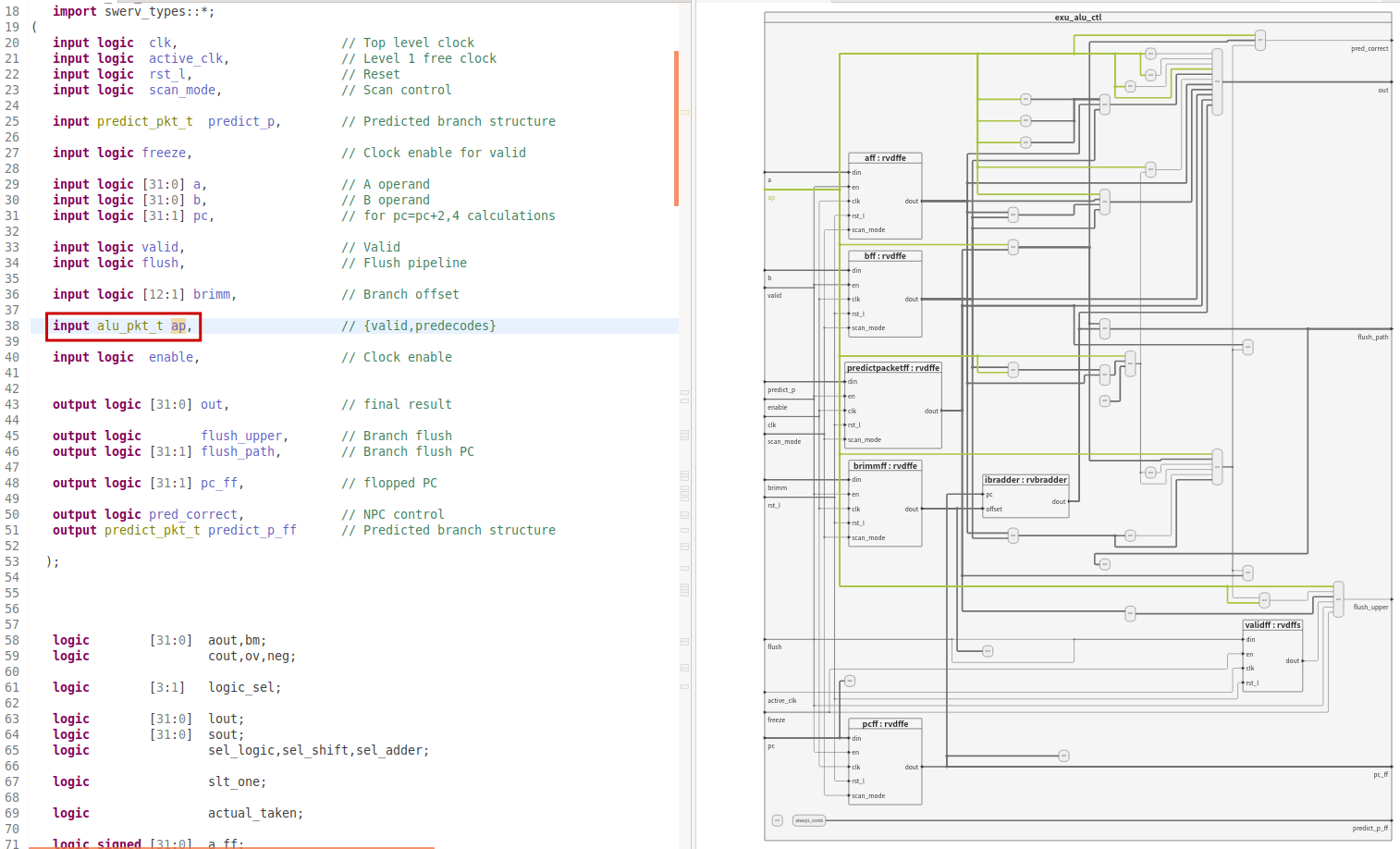


Figura 9 . Resalte la señal ap

1. También puede buscar la implementación de un módulo combinacional en el código Verilog, haciendo doble clic en el módulo en el diagrama de bloques. Por ejemplo, en la Figura 10 , se muestra el módulo que genera la salida de señal .

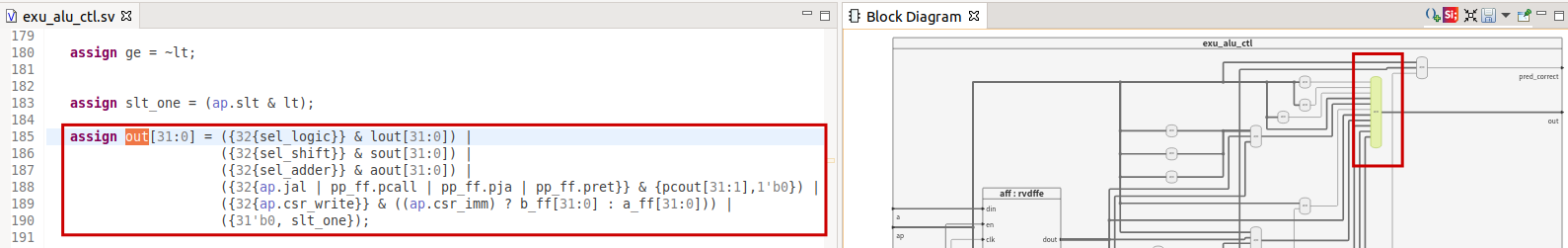


Figura 10 . Resalte el código Verilog para el módulo combinacional que genera la salida de señal

1. Finalmente, abrimos una declaración de módulo en el diagrama de bloques haciendo clic con el botón derecho en la instanciación del módulo en el código Verilog y seleccionando Abrir declaración. La Figura 11 muestra el módulo **rvdffe** , implementado en el archivo *beh\_lib.sv* .

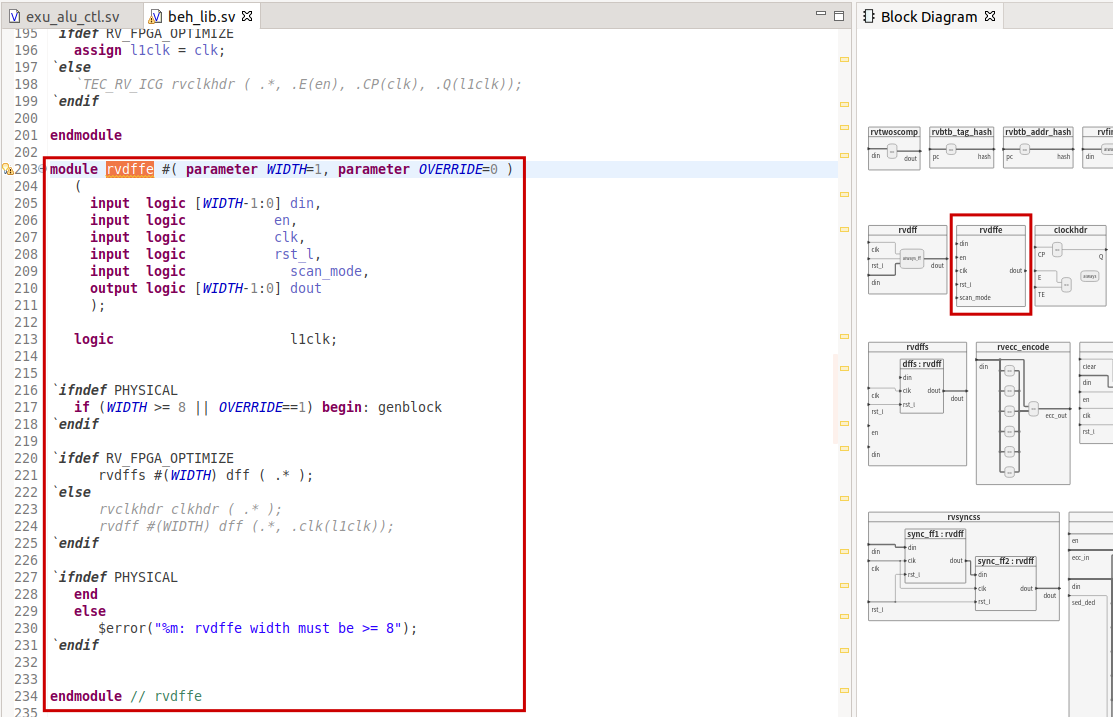


Figura 11 . Modulo rvdffe

# 2. CONFIGURACIÓN DEL PROCESADOR SWERV EH1

1. **Configurar las estructuras principales**

*[RVfpgaPath]/ RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/common\_defines.vh* permite al usuario configurar muchas estructuras del núcleo, como la caché de instrucciones, el ICCM/DCCM, el predictor de rama, etc. Se proporciona una configuración predeterminada en el Sistema RVfpga, que puedes cambiar de dos maneras diferentes:

* Puede editar manualmente los parámetros en el archivo *common\_defines.vh* .
* Puede usar el script *swerv.config* proporcionado por Western Digital con el paquete SweRV EH1. El uso de este script se describe en <https://github.com/chipsalliance/Cores-SweRV/tree/branch1.8>

En RVfpga puede encontrar el script *swerv.config* en: *[RVfpgaPath]/ RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/*

Una vez que haya generado los nuevos archivos de configuración, puede volver a sintetizar el SoC en Vivado como se explica en el laboratorio 1 y obtener el nuevo flujo de bits del sistema RVfpga.

1. **Deshabilitar el uso de instrucciones comprimidas**

En algunos casos, nos puede interesar deshabilitar el uso de instrucciones comprimidas. Para ello, debemos realizar dos cambios en nuestro proyecto PlatformIO:

* Incluya las siguientes líneas nuevas en el archivo *platformio.ini* :

build\_unflags = -Wa,-march=rv32imac -march=rv32imac

build\_flags = -Wa,-march=rv32ima -march=rv32ima

extra\_scripts = extra\_script.py

* Agregue el archivo *extra\_script.py* a las fuentes del proyecto. Este archivo contiene las siguientes líneas:

Importar ("env")

env.Append(

BANDERAS DE ENLACE=[

"-Wa,-marzo=rv32ima",

"-marcha=rv32ima"

]

)

En la mayoría de los ejemplos usados en Labs 11-20, deshabilitaremos el uso de instrucciones comprimidas en aras de la simplicidad.

1. **Habilitar/deshabilitar funciones principales**

La Tabla 10-1 del Manual de referencia del programador de SweRV EH1 ( <https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf>) muestra los bits del registro *mfdc* (en *CSR* 0x7F9). Este registro alberga bits de control de núcleo de bajo nivel para deshabilitar funciones específicas, como la ejecución de problemas dobles o canalizados, el predictor de rama, etc. La Tabla 1 muestra las nueve funciones principales que puede controlar este registro. Al establecer los bits adecuados del registro en 0 o 1, se habilita o deshabilita cada característica principal. Por ejemplo, puede incluir las siguientes dos instrucciones de ensamblaje en su programa de ensamblaje para deshabilitar la ejecución de dos problemas, la ALU secundaria y la ejecución canalizada:

li t2, 0x481 csrrs t1, 0x7F9, t2

Tabla 1 . Registro de control de desactivación de funciones ( *mfdc* : CSR 0x7F9)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31-11 | Reservado | 7 | 0: habilitar ALU secundaria  1: deshabilitar ALU secundaria | 3 | 0: habilita la predicción de sucursales y la pila de direcciones de retorno  1: desactivar la predicción de sucursales y la pila de direcciones de retorno |
| 10 | 0: ejecución de problema dual  1: ejecución de un solo problema | 6 | 0: las tiendas de efectos secundarios están canalizadas  1: las tiendas de efectos secundarios bloquean todas las transacciones de bus subsiguientes hasta  almacenar la respuesta con el valor predeterminado recibido | 2 | 0: habilita la fusión del búfer de escritura  1: desactivar la fusión del búfer de escritura |
| 9 | Reservado | 5 | 0: habilitar cargas/divisiones sin bloqueo  1: deshabilitar cargas/divisiones no bloqueantes | 1 | Reservado |
| 8 | 0: Comprobación ICCM/DCCM ECC habilitada  1: Comprobación ICCM/DCCM ECC deshabilitada | 4 | 0: habilitar división rápida  1: deshabilitar división rápida | 0 | 0: ejecución segmentada  1: ejecución de una sola instrucción |

Usaremos diferentes configuraciones en Labs 11-20 para comparar el rendimiento, los aciertos/errores de I$, los aciertos/errores de Branch Predictor, etc., de SweRV EH1 cuando las diferentes funciones principales están habilitadas/inhabilitadas.

# 3 . PRINCIPALES MÓDULOS Y SEÑALES DEL NÚCLEO SweRV EH1

El sistema RVfpga se ejecuta en el FPGA Artix-7 ubicado en la placa Nexys A7, como se muestra en la Figura 12 . La figura detalla la jerarquía del sistema, incluidos los nombres de los módulos y submódulos de Verilog. El sistema RVfpga consta del núcleo SweRVolf ( ***swervolf\_core*** ), el controlador DRAM ( ***litedram\_top*** ), el módulo de generación de reloj ( ***clk\_gen\_nexys*** ) y algunos módulos de interfaz. El núcleo de SweRVolf, a su vez, consta del procesador SweRV EH1 ( ***swerv\_wrapper\_dmi*** ) y módulos de interfaz adicionales ( ***wb\_intercon*** *,* ***axi\_intercon*** , ***uart\_top*** , etc.). El módulo superior para el procesador SweRV EH1, ***swerv\_wrapper\_dmi*** , instancia los dos módulos principales del núcleo: ***mem*** y ***swerv*** . En el resto de este documento, enumeramos los submódulos y las señales principales de estos dos módulos. Tenga en cuenta que puede encontrar las señales restantes de cada módulo en la interfaz del módulo. En Labs 11-20 estudiamos estas señales al analizar el funcionamiento de las diferentes partes del procesador.



Figura 12 . Jerarquía del Sistema RVfpga

**MÓDULO: *memoria***

**FUNCIÓN:** Este módulo instancia las tres memorias internas disponibles en SweRV: ICCM, DCCM e I$. La Tabla 2 enumera los submódulos de *mem* y sus señales de interfaz.



Figura 13 . Módulo *mem* y sus submódulos

Tabla 2 . submódulos *mem y E/S*

|  |  |  |  |
| --- | --- | --- | --- |
| **Unidad** | **E/S** | **Nombre** | **Descripción** |
| ICCM: ***ifu\_iccm\_mem***  (Contiene el envoltorio del módulo ICCM) | Aporte | iccm\_wren | Habilitar escritura |
| iccm\_rden | Habilitar lectura |
| [`RV\_ICCM\_BITS-1:2] iccm\_rw\_dirección | Dirección de lectura/escritura |
| [77:0] iccm\_wr\_data | Escribir datos |
| Producción | [155:0] iccm\_rd\_data | Leer datos |
| yo$: ***ifu\_ic\_mem***  (Contiene el envoltorio del módulo de etiqueta y datos de caché de instrucciones) | Aporte | [3:0] ic\_wr\_es | Habilitar escritura |
| ic\_rd\_es | Habilitar lectura |
| [31:2] ic\_rw\_dirección | Dirección de lectura/escritura |
| [67:0] ic\_wr\_data | Datos para llenar al Icache. Con Paridad. |
| Producción | [135:0] ic\_rd\_datos | Lectura de datos de Icache. Etapa F2. Con Paridad. |
| [3:0] ic\_rd\_hit | Acertar/fallar en cada sentido |
| DCCM:  ***lsu\_dccm\_mem***  (Contiene el envoltorio del módulo DCCM) | Aporte | dccm\_wren | Habilitar escritura |
| dccm\_rden | Habilitar lectura |
| [`RV\_DCCM\_BITS-1:0] dccm\_wr\_addr | escribir dirección |
| [`RV\_DCCM\_BITS-1:0] dccm\_rd\_addr\_lo | Leer dirección |
| [`RV\_DCCM\_BITS-1:0] dccm\_rd\_addr\_hi | Leer la dirección del banco superior (alto) cuando el acceso está desalineado |
| [`RV\_DCCM\_FDATA\_WIDTH-1:0] dccm\_wr\_data | Escribir datos |
| Producción | [`RV\_DCCM\_FDATA\_WIDTH-1:0] dccm\_rd\_data\_lo | Leer banco bajo de datos |
| [`RV\_DCCM\_FDATA\_WIDTH-1:0] dccm\_rd\_data\_hi | Leer datos banco alto |

**MÓDULO: *desvío***

**FUNCIÓN:** como se muestra en la Figura 14 , *swerv* es el módulo de nivel superior para el núcleo SweRV EH1. Instancia los módulos principales del núcleo, los más importantes: *ifu* , *dec* , *exu* y *lsu* . Tabla 3 : la tabla 6 enumera cada uno de los submódulos y las señales de interfaz de estas unidades. El módulo *swerv* se comunica con el módulo *mem* a través del contenedor SweRV ( *swerv\_wrapper\_dmi* ).



Figura 14 . *swerv* y sus submódulos

Tabla 3 . *ifu* (Unidad de obtención de instrucciones) E/S y submódulos (incluidas sus E/S)

|  |  |  |  |
| --- | --- | --- | --- |
| Unidad | E/S | Nombre | Descripción |
| Unidad de obtención de instrucciones:  *si tu*  (Este es el módulo de nivel superior para la Obtención de las Instrucciones, la Predicción del Predictor de Rancho y el Alineador) | / Salida | Varias señales | Puertos ICCM hacia/desde el módulo de memoria |
| Varias señales | I$ puertos a/desde módulo mem |
| Varias señales | Puertos IFU AXI |
| Aporte | exu\_flush\_final | Limpiar la tubería |
| [31:1] exu\_flush\_path\_final | Vaciar la dirección de búsqueda |
| Producción | [31:0] ifu\_i0\_instr | Instrucción 0. De Alinear a Decodificar |
| [31:0] ifu\_i1\_instr | Instrucción 1. De Alinear a Decodificar |
| [31:1] ifu\_i0\_pc | Instrucción 0 PC (contador de programa). De alinear a decodificar |
| [31:1] ifu\_i1\_pc | Instrucción 1 PC. De alinear a decodificar |
| Control de búsqueda:  ***ifu\_ifc\_ctl***  (Este módulo implementa Fetch Pipe Control. Genera la siguiente dirección para obtener de la memoria de instrucciones). | Aporte | exu\_flush\_final | Limpiar la tubería |
| [31:1] ifu\_bp\_btb\_objetivo\_f2 | PC objetivo pronosticado |
| [31:1] exu\_flush\_path\_final | Camino al ras |
| Producción | lógica de salida [31:1] ifc\_fetch\_addr\_f1 | Obtener dirección en FC1 |
| Interno | lógica [31:1] fetch\_addr\_next | dirección secuencial |
| Control de memoria de instrucciones (I$ e ICCM):  ***ifu\_mem\_ctl***  (Control de Memoria de Instrucciones – Icache y ICCM –) | Aporte | [31:1] buscar\_dirección\_f1 | Obtener dirección en FC1 ( ifc\_fetch\_addr\_f1 renombrado ) |
| Producción | [127:0] ic\_datos\_f2 | Lectura de datos en FC2 desde I$ o ICCM hasta la etapa de alineación |
| Control de alineación:  ***ifu\_aln\_ctl***  (Alineador de instrucciones) | Aporte | [127:0] ifu\_fetch\_datos | Datos de recuperación de 128 bits de Fetch Stage |
| Interno | lógica [127:0] q2,q1,q0 | 3 tampones |
| Producción | [31:0] ifu\_i0\_instr | Modo de instrucción 0 |
| [31:0] ifu\_i1\_instr | Forma de instrucción 1 |
| [31:1] ifu\_i0\_pc | Modo de instrucción 0 PC |
| [31:1] ifu\_i1\_pc | Forma de instrucción 1 PC |
| Predictor de rama:  ***ifu\_bp\_ctl*** | Aporte | [31:1] ifc\_fetch\_addr\_f1 | Obtener dirección en FC1 |
| Producción | [31:1] ifu\_bp\_btb\_objetivo\_f2 | PC objetivo pronosticado |
| ifu\_bp\_kill\_next\_f2 | Rama tomada/no tomada |

Tabla 4 . *dec* (Unidad de decodificación) E/S y submódulos (incluidas sus E/S)

|  |  |  |  |
| --- | --- | --- | --- |
| Unidad | E/S | Nombre | Descripción |
| Unidad de decodificación:  *dic*  (Este es el módulo de nivel superior para la Descodificación de las Instrucciones, el Marcador de Dependencias y el acceso al Archivo de Registro) | Aporte | exu\_flush\_final | enjuague la tubería cuando 1 |
| [31:0] ifu\_i0\_instr,  [31:1] ifu\_i1\_instr | Instrucciones de Align |
| [31:1] ifu\_i0\_pc  [31:1] ifu\_i1\_pc | PC de Align |
| Producción | alu\_pkt\_t i0\_ap  alu\_pkt\_t i1\_ap | Señales de control ALU |
| lsu\_pkt\_t lsu\_p | Señales de control LSU |
| mul\_pkt\_t mul\_p | Señales de control MUL |
| div\_pkt\_t div\_p | Señales de control DIV |
| predecir\_pkt\_t  i0\_predecir\_p\_d  i1\_predecir\_p\_d | señales de predicción a las ALU |
| [31:1] dec\_i0\_pc\_d  [31:1] dec\_i1\_pc\_d | Dirección de instrucciones en la etapa de decodificación |
| [31:0] gpr\_i0\_rs1\_d  [31:0] gpr\_i0\_rs2\_d  [31:0] gpr\_i1\_rs1\_d  [31:0] gpr\_i1\_rs2\_d | I0/I1 rs1/rs2 datos del archivo de registro |
| [31:0] dec\_i0\_immed\_d  [31:0] dec\_i1\_immed\_d | valor inmediato |
| [12:1] dec\_i0\_br\_immed\_d  [12:1] dec\_i1\_br\_immed\_d | Desplazamiento de rama |
| [31:0] i0\_rs1\_bypass\_data\_d  [31:0] i0\_rs2\_bypass\_data\_d  [31:0] i0\_rs1\_bypass\_data\_e2  [31:0] i0\_rs2\_bypass\_data\_e2  [31:0] i0\_rs1\_bypass\_data\_e3  [31:0] i0\_rs2\_bypass\_data\_e3 | I0 rs1/rs2 datos de derivación |
| [31:0] i1\_rs1\_bypass\_data\_d  [31:0] i1\_rs2\_bypass\_data\_d  [31:0] i1\_rs1\_bypass\_data\_e2  [31:0] i1\_rs2\_bypass\_data\_e2  [31:0] i1\_rs1\_bypass\_data\_e3  [31:0] i1\_rs2\_bypass\_data\_e3 | I1 rs1/rs2 datos de derivación |
| Interno | [31:0] dec\_i0\_instr\_d  [31:0] dec\_i1\_instr\_d | Instrucciones en la etapa de decodificación |
| [31:0] dec\_i0\_rs1\_d  [31:0] dec\_i0\_rs2\_d  [31:0] dec\_i1\_rs1\_d  [31:0] dec\_i1\_rs2\_d | datos rs1/rs2 |
| Instrucciones/PC para enviar desde Alinear a Decodificar:  *dec\_ib\_ctl*  (Búferes para propagar las instrucciones y PC desde el Alineador al Decodificador) | Aporte | [31:0] ifu\_i0\_instr  [31:0] ifu\_i1\_instr | Instrucción I0/I1 de Align |
| [31:1] ifu\_i0\_pc  [31:1] ifu\_i1\_pc | PC I0/I1 de Align |
| Producción | [31:0] dec\_i0\_instr\_d  [31:0] dec\_i1\_instr\_d | Instrucción I0/I1 en decodificación |
| [31:1] dec\_i0\_pc\_d  [31:1] dec\_i1\_pc\_d | PC I0/I1 en decodificación |
| Decodifique la instrucción y calcule los valores de omisión:  ***dec\_decode\_ctl***  (Decodifique las 2 instrucciones y calcule los valores de derivación) | Aporte | [31:1] dec\_i0\_pc\_d  [31:1] dec\_i1\_pc\_d  [31:0] exu\_i0\_resultado\_e1 | PC I0/I1 |
| [31:0] dec\_i0\_instr\_d, [31:0] dec\_i1\_instr\_d | instrucción en la etapa de decodificación |
| Producción | alu\_pkt\_t i0\_a  alu\_pkt\_t i1\_ap | Señales de control ALU |
| lsu\_pkt\_t lsu\_p | Señales de control LSU |
| mul\_pkt\_t mul\_p | Señales de control MUL |
| div\_pkt\_t div\_p | Señales de control DIV |
| predecir\_pkt\_t i0\_predecir\_p\_d  i1\_predecir\_p\_d | señales de predicción a ALU |
| [4:0] dec\_i0\_rs1\_d  [4:0] dec\_i0\_rs2\_d  [4:0] dec\_i1\_rs1\_d  [4:0] dec\_i1\_rs2\_d | Índice I0/I1 rs1/rs2 |
| [31:0] dec\_i0\_immed\_d  [31:0] dec\_i1\_immed\_d | valor inmediato |
| [12:1] dec\_i0\_br\_immed\_d  [12:1] dec\_i1\_br\_immed\_d | Desplazamiento de rama |
| [31:0] i0\_rs1\_bypass\_data\_d  [31:0] i0\_rs2\_bypass\_data\_d  [31:0] i0\_rs1\_bypass\_data\_e2  [31:0] i0\_rs2\_bypass\_data\_e2  [31:0] i0\_rs1\_bypass\_data\_e3  [31:0] i0\_rs2\_bypass\_data\_e3 | I0 rs1/rs2 datos de derivación |
| [31:0] i1\_rs1\_bypass\_data\_d  [31:0] i1\_rs2\_bypass\_data\_d  [31:0] i1\_rs1\_bypass\_data\_e2  [31:0] i1\_rs2\_bypass\_data\_e2  [31:0] i1\_rs1\_bypass\_data\_e3  [31:0] i1\_rs2\_bypass\_data\_e3 | I1 rs1/rs2 datos de derivación |
| Archivo de registro: *dec\_gpr\_ctl*  (Archivo de registro) | Aporte | [4:0] raddr0, raddr1  [4:0] raddr2, raddr3 | Leer direcciones |
| [4:0] waddr0, waddr1 [4:0] waddr2 | escribir direcciones |
| [31:0] wd0, wd1, wd2 | Escribir datos |
| rden0, rden1, rden2, rden3 | Habilitar lectura |
| wen0, wen1, wen2 | Habilitar escritura |
| Producción | [31:0] rd0, rd1, rd2, rd3 | Leer datos |

Tabla 5 . *exu* (Unidad de ejecución) E/S y submódulos (incluidas sus E/S)

|  |  |  |  |
| --- | --- | --- | --- |
| Unidad | E/S | Nombre | Descripción |
| Unidad Ejecutora: *exú*  (Este es el módulo de nivel superior para la Ejecución de las Instrucciones AL) | Aporte | alu\_pkt\_t i0\_ap, alu\_pkt\_t i1\_ap | mando ALU |
| mul\_pkt\_t mul\_p | control MUL |
| div\_pkt\_t div\_p | control DIV |
| [31:1] dec\_i0\_pc\_d, dec\_i1\_pc\_d | PC de Decode |
| [31:0] gpr\_i0\_rs1\_d  [31:0] gpr\_i0\_rs2\_d  [31:0] gpr\_i1\_rs1\_d  [31:0] gpr\_i1\_rs2\_d | I0/I1 rs1/rs2 |
| [31:0] dec\_i0\_immed\_d  [31:0] dec\_i1\_immed\_d | Valores inmediatos |
| [12:1] dec\_i0\_br\_immed\_d  [12:1] dec\_i1\_br\_immed\_d | Compensaciones de rama |
| [31:0] i0\_rs1\_bypass\_data\_d  [31:0] i0\_rs2\_bypass\_data\_d  [31:0] i0\_rs1\_bypass\_data\_e2  [31:0] i0\_rs2\_bypass\_data\_e2  [31:0] i0\_rs1\_bypass\_data\_e3  [31:0] i0\_rs2\_bypass\_data\_e3 | I0 rs1/rs2 datos de derivación |
| [31:0] i1\_rs1\_bypass\_data\_d  [31:0] i1\_rs2\_bypass\_data\_d  [31:0] i1\_rs1\_bypass\_data\_e2  [31:0] i1\_rs2\_bypass\_data\_e2  [31:0] i1\_rs1\_bypass\_data\_e3  [31:0] i1\_rs2\_bypass\_data\_e3 | I1 rs1/rs2 datos de derivación |
| Producción | exu\_flush\_final | enjuague la tubería cuando 1 |
| [31:0] exu\_i0\_resultado\_e1  [31:0] exu\_i1\_resultado\_e1 | resultado ALU primario |
| [31:0] exu\_i0\_result\_e4  [31:0] exu\_i1\_resultado\_e4 | resultado ALU secundario |
| [31:0] exu\_mul\_resultado\_e3 | resultado MUL |
| [31:0] exu\_div\_resultado | resultado DIV |
| [31:0] exu\_lsu\_rs1\_d | Dirección de carga/almacenamiento |
| [31:0] exu\_lsu\_rs2\_d | Almacenamiento de datos |
| ALU:  *exu\_alu\_ctl*  (Unidad Aritmética Lógica) | Aporte | [31:0] un | un operando |
| [31:0] segundo | operando B |
| [31:1] ordenador personal | para cálculos pcnext (es decir, pc+2 o pc+4) |
| [12:1] rebosante | desplazamiento de rama |
| alu\_pkt\_tap | mando ALU |
| Producción | [31:0] fuera | resultado ALU |
| al ras\_superior | rama al ras |
| [31:1] ruta\_descarga | PC de destino |
| [31:1] pc\_ff |  |
| Multiplicador:  *exu\_mul\_ctl* | Aporte | [31:0] un | un operando |
| [31:0] segundo | operando B |
| mul\_pkt\_t MP | control MUL |
| Producción | [31:0] fuera | resultado MUL |
| Divisor:  *exu\_div\_ctl* | Aporte | [31:0] dividendo | numerador |
| [31:0] divisor | denominador |
| div\_pkt\_t dp | control DIV |
| Producción | [31:0] fuera | resultado DIV |

Tabla 6 . *lsu* (Unidad de carga/almacenamiento) E/S y submódulos (incluidas sus E/S)

|  |  |  |  |
| --- | --- | --- | --- |
| Unidad | E/S | Nombre | Descripción |
| Unidad de carga/almacenamiento:  *lsu*  (Este es el módulo de nivel superior para la Unidad de Carga/Almacenamiento de las Instrucciones) | De entrada y salida | Varias señales | Puertos DCCM hacia/desde el módulo de memoria |
| Varias señales | esclavo DMA |
| Varias señales | Puertos LSU AXI |
| Aporte | [31:0] exu\_lsu\_rs1\_d | Dirección de carga/almacenamiento |
| [31:0] exu\_lsu\_rs2\_d | Almacenamiento de datos |
| [11:0] dec\_lsu\_offset\_d | compensación de dirección |
| lsu\_pkt\_t lsu\_p | Control LSU |
| Producción | [31:0] lsu\_resultado\_dc3 | LSU leer datos |
| Cálculo de direcciones:  *lsu\_lsc\_ctl*  (Control de LSU y calcular dirección de carga/almacenamiento) | Aporte | [31:0] exu\_lsu\_rs1\_d | Dirección de carga/almacenamiento |
| [31:0] exu\_lsu\_rs2\_d | Almacenamiento de datos |
| [11:0] dec\_lsu\_offset\_d | compensación de dirección |
| lsu\_pkt\_t lsu\_p | Control LSU |
| Producción | [31:0] lsu\_addr\_dc1  [31:0] end\_addr\_dc1 | Dirección inicial/final |
| Control DCCM:  ***lsu\_dccm\_ctl***  (Control DCCM) | Aporte | [`RV\_DCCM\_FDATA\_WIDTH-1:0]  dccm\_rd\_data\_lo | leer datos (banco bajo) |
| [`RV\_DCCM\_FDATA\_WIDTH-1:0]  dccm\_rd\_data\_hi | leer datos (hola banco) |
| Producción | dccm\_wren | habilitar escritura |
| dccm\_rden | habilitar lectura |
| [`RV\_DCCM\_BITS-1:0]  dccm\_wr\_addr | escribir dirección |
| [`RV\_DCCM\_BITS-1:0]  dccm\_rd\_addr\_lo | leer dirección (lo) |
| [`RV\_DCCM\_BITS-1:0]  dccm\_rd\_addr\_hi | leer dirección (hi): necesaria para cargas desalineadas |
| [`RV\_DCCM\_FDATA\_WIDTH-1:0]  dccm\_wr\_data | escribir datos |
| Almacenar búfer:  *lsu\_stbuf*  (Almacenar búfer) | Aporte | lsu\_addr\_dc3 | dirección |
| [`RV\_DCCM\_DATA\_WIDTH-1:0] store\_ecc\_datafn\_hi\_dc3 | escribir datos (hola) |
| [`RV\_DCCM\_DATA\_WIDTH-1:0] store\_ecc\_datafn\_lo\_dc3 | escribir datos (lo) |
| Producción | [`RV\_LSU\_SB\_BITS-1:0] stbuf\_addr\_any | almacenar la dirección del búfer |
| [`RV\_DCCM\_DATA\_WIDTH-1:0] stbuf\_data\_any | almacenar datos de búfer |

# 4 . ESTRUCTURAS Y TIPOS PARA AGRUPAMIENTO DE BROCAS DE CONTROL

A continuación se muestra un resumen de los principales tipos de estructuras definidos en el archivo *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv\_types.sv* y utilizados en el procesador SweRV EH1 para agrupar las señales de control.

* **dec\_pkt\_t :** Este es el tipo de estructura de control principal y contiene las principales señales de control del procesador, como alu (1 si se ejecuta una instrucción aritmético-lógica, 0 en caso contrario), carga (1 si se ejecuta una instrucción de carga , 0 en caso contrario), legal (1 si la instrucción es legal, 0 si no lo es), rs1 (1 si la instrucción obtiene el primer operando de entrada del archivo de registro, 0 de lo contrario), imm12 (1 si la instrucción usa un inmediato de 12 bits como un operando de entrada, 0 en caso contrario), etc.

Este tipo de estructura se usa dentro del módulo **dec\_decode\_ctl** para generar muchas otras señales de control. Se declaran cuatro señales de este tipo (Way-0: i0\_dp\_raw , i0\_dp . Way-1: i1\_dp\_raw , i1\_dp ) y se utilizan para generar los bits de control de otras estructuras definidas en el archivo *swerv\_types.sv* .

Estos bits se asignan dentro del módulo **dec\_dec\_ctl** , un módulo que se genera automáticamente utilizando herramientas de código abierto ( *coredecode* y *espresso* ) y que se puede encontrar al final del archivo *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec\_decode\_ctl. sv* .

* **alu\_pkt\_t :** Este tipo de estructura contiene las señales de control relacionadas con la operación de la ALU, como valid (1 si se ejecuta una instrucción aritmético-lógica, 0 en caso contrario), add (1 si se ejecuta una instrucción de suma , 0 en caso contrario), beq (1 si se ejecuta una instrucción beq , 0 en caso contrario), etc. Dos señales de este tipo, denominadas i0\_ap e i1\_ap , se definen dentro del módulo **dec\_decode\_ctl** .

Estos bits se asignan dentro del módulo **dec\_decode\_ctl** (implementado en: *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec\_decode\_ctl.sv* ), según los bits de la estructura dec\_pkt\_t (consulte las líneas 711-770 de **dec\_decode\_ctl** ).

* **reg\_pkt\_ t :** este tipo de estructura contiene los identificadores de los dos registros de origen (campos rs 1 y rs 2 ) y el registro de destino (campo rd ) . Dos señales de este tipo, llamadas i0 r y i1 r , se definen dentro del módulo **dec\_decode\_ct l** . Estas señales se asignan desde los campos correspondientes del Registro de Instrucciones dentro del módulo **dec\_decode\_ctl** (ver líneas 1121-1127 de este módulo).
* **dest\_pkt\_t :** este tipo de estructura contiene bits de control utilizados en la etapa Write-Back, que analizaremos en una próxima sección. Una señal de este tipo, llamada dd , se define dentro del módulo **dec\_decode\_ctl** .
* **rets\_pkt\_t , br\_pkt\_t , br\_tlu\_pkt\_t y predict\_pkt\_t :** estos tipos de estructura están relacionados con las instrucciones de bifurcación y el predictor de bifurcación.
* **lsu\_pkt\_t :** este tipo de estructura contiene las señales de control relacionadas con la unidad de carga/almacenamiento, como mitad (1 si se lee/escribe media palabra, 0 en caso contrario), carga (1 si se ejecuta una instrucción de carga , 0 en caso contrario), válida (1 si la instrucción es válida, 0 en caso contrario), etc. Una señal de este tipo, llamada lsu\_p , se define dentro del módulo **dec\_decode\_ctl** .
* **mul\_pkt\_t :** Este tipo de estructura contiene las señales de control relacionadas con la Unidad de Multiplicación, como rs1\_sign y rs2\_sign (que determinan si los operandos de entrada se tratan como con signo o sin signo), válido (1 si la instrucción es válida, 0 en caso contrario), etc. Una señal de este tipo, llamada mul\_p , se define dentro del módulo **dec\_decode\_ctl** .
* **div\_pkt\_t :** Este tipo de estructura contiene las señales de control relacionadas con la Unidad de División, como unsign (1 si la operación no está firmada, 0 en caso contrario), valid (1 si la instrucción es válida, 0 en caso contrario), etc. Una señal de este tipo , llamado div\_p , se define dentro del módulo **dec\_decode\_ctl** .

**TAREA :** Abra el archivo *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv\_types.sv* y analícelo durante las siguientes descripciones de los tipos de estructuras que agrupan los bits de control .

**TAREA :** Eche un vistazo rápido a los módulos **dec\_decode\_ctl** y **dec\_dec\_ctl** para ver cómo se asignan los campos de las señales de control en función de los 32 bits de la instrucción . Estos dos módulos son muy extensos y bastante complejos, por lo que la idea no es analizarlos en detalle. Además, vea que el módulo **dec\_dec\_ctl** se crea automáticamente como se explica en las líneas 2482-2495 de *dec\_decode\_ctl.sv* .

# 5 . INSTRUCCIONES COMPRIMIDAS

Si bien en la mayoría de los experimentos que incluimos en los laboratorios deshabilitamos el uso de instrucciones comprimidas por simplicidad, en esta sección describimos y analizamos la extensión de instrucciones comprimidas (RVC) de RISC-V y la ejecución de instrucciones comprimidas en SweRV. EH1. Obviamente, usted es libre de habilitar el uso de instrucciones comprimidas en los experimentos y ampliar el análisis por su cuenta.

**NOTA:** Antes de comenzar con este laboratorio, recomendamos leer la Sección 6.6.5 del libro de S. Harris y D. Harris, “ *Diseño digital y arquitectura de computadoras: Edición RISC-V* ”, Morgan Kaufmann [DDCARV]. Algunos de los contenidos de esta sección están inspirados en ese libro.

La extensión RVC reduce el tamaño de las instrucciones de punto flotante y entero común a 16 bits al reducir el tamaño de los campos de control, inmediato y de registro y al aprovechar los registros redundantes o implícitos. Este tamaño de instrucción reducido reduce el costo, la potencia y la memoria requerida, todo lo cual puede ser crucial para las aplicaciones portátiles y móviles. Nuestros programas de ensamblaje pueden usar una combinación de instrucciones comprimidas y de 32 bits, dado que SweRV EH1 incluye el RVC.

En SweRV EH1 hay un módulo específicamente dedicado a descomprimir instrucciones: **ifu\_compress\_ctl** . Este módulo recibe una instrucción comprimida de 16 bits y genera la correspondiente instrucción de 32 bits sin comprimir. En la Figura 15 mostramos la Etapa de alineación con un poco más de detalle que en el Laboratorio 11 (todavía dejamos algunos recuadros negros que puede analizar usted mismo). Se crean instancias de tres módulos **ifu\_compress\_ctl** dentro del módulo **ifu\_aln\_ctl** , que reciben una instrucción comprimida de la señal aligndata[63:0] y devuelven la instrucción sin comprimir correspondiente en las señales uncompress0[31:0] , uncompress1[31:0] y uncompress2[31:0] . Si las instrucciones ya están en su formato sin comprimir, se proporcionan directamente desde la señal aligndata[63:0] .



Figura 15 . Alinear escenario

El código ilustrado en la parte superior de la Figura 16 muestra el programa C simple del Capítulo 6 – Ejemplo de código 31 – DDCARV. El código ilustrado en la parte inferior de la Figura 16 muestra el código ensamblador generado cuando el programa C se compila en PlatformIO con la extensión RVC habilitada (tenga en cuenta que el código ensamblador es ligeramente diferente al que se muestra en [DDCARV]). Resaltamos en rojo las instrucciones que componen el cuerpo del bucle, que son una combinación de instrucciones de 16 y 32 bits.

|  |
| --- |
| puntajes int[200]; int principal (vacío) { int i; for (i = 0; i < 200; i = i + 1){ puntuaciones[i] = puntuaciones[i] + 10; } retorno(0);  } |
| 00000088 <principal>:  88: 6789 luia5,0x2  8a: 12078793 addia5,a5,288 # 2120 <partituras>  8e: 32078693 addia3,a5,800  92: **4398 lw a4,0(a5)**  94: **0791 añadir a5,a5,4**  96: **0729 añadir a4,a4,10**  98: **fee7ae23 swa4,-4(a5)**  9c: **fed79be3 bnea5,a3,92 <principal+0xa>**  a0: 4501 lia0,0  a2: 8082 ret | | |

Figura 16 . Ejemplo de instrucciones comprimidas

La Figura 17 muestra la simulación de Verilator de una iteración completa del ciclo de la Figura 16 . Tenga en cuenta que cuando la instrucción addi a5,a5,4 está en la etapa de alineación (resaltada en rojo en el primer ciclo de la figura), la instrucción se extrae del paquete de 64 bits ( aligndata[63:0] ) y se descomprime de un instrucción de 16 bits ( **0x0791** ) en una instrucción de 32 bits ( **0x00478793** ). ( El código se proporciona en *[RVfpgaPath]/RVfpga/Labs /Lab11/Compressed\_C-Example* para que pueda ejecutar su propia simulación de Verilator ) .

* En RISC-V, el código de operación para la instrucción c.addi de 16 bits es (consulte el Apéndice B de [DDCARV]):

000 | mm(1 bit) | rd/rs1 | imm(5-bits) | 01

Entonces puedes verificar fácilmente que **0x0791** ( 000 0 01111 00100 01 ) corresponde a: c.addi a5,4 (recuerda que a5=x15).

* + - mm = 000100
    - rd = rs1 = 01111 ( x15 )
* addi de 32 bits es (consulte el Apéndice B de [DDCARV]):

imm (12 bits) | rs1 | 000 | rd | 0010011

Entonces puedes verificar fácilmente que **0x00478793** ( 000000000100 01111 000 01111 0010011 ) corresponde a: addi a5,a5,4 (recuerda que a5 = x15).

* + - mm = 000000000100
    - rs1 = 01111 ( x15 )
    - rd = 01111 ( x15 )

En el segundo ciclo que se muestra en la Figura 17 , la instrucción sw está alineada. Dado que esta instrucción carece de la versión comprimida correspondiente en la arquitectura RISC-V, no necesita descomprimirse y se selecciona y propaga a la etapa de decodificación directamente desde la señal aligndata[63:0] .

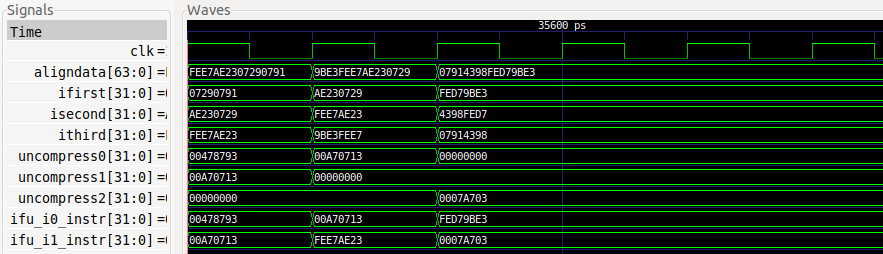


Figura 17 . Simulación del código que se muestra en la Figura 16

**TAREA :** Analice las instrucciones restantes del cuerpo del ciclo en términos de instrucciones comprimidas/sin comprimir.

**TAREA :** Eche un vistazo al interior del módulo **ifu\_compress\_ctl** e intente hacerse una idea de cómo funciona.

# 6. REFERENCIAS REALES

En la carpeta *[RVfpgaPath]/RVfpga/Labs /Lab20/RealBenchmarks* proporcionamos tres aplicaciones reales que usará en Lab 20 para probar las diferentes funciones de nuestro procesador SweRV EH1. En ese laboratorio, puede encontrar una descripción más detallada sobre estos tres puntos de referencia y las diferentes versiones que ofrecemos para cada uno de ellos.

* **CoreMark** : en la carpeta *[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/CoreMark\_HwCounters* puede encontrar un proyecto PlatformIO que contiene el punto de referencia CoreMark para ejecutar en RVfpgaNexys. Usamos las fuentes proporcionadas en <https://github.com/chipsalliance/Cores-SweRV>y las adaptamos a nuestro sistema RVfpga.
* **Dhrystone** : en la carpeta *[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/Dhrystone\_HwCounters* puede encontrar un proyecto PlatformIO que contiene el punto de referencia de Dhrystone para ejecutar en RVfpgaNexys. Usamos las fuentes proporcionadas en <https://github.com/chipsalliance/Cores-SweRV>y las adaptamos a nuestro sistema RVfpga.
* **Procesamiento de imágenes** : en la carpeta *[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/ImageProcessing\_HwCounters* puede encontrar un proyecto PlatformIO que contiene la aplicación que usamos en el laboratorio 5 para transformar una imagen RGB en escala de grises.