IMPLEMENTACIÓN Y EXTENSIÓN DE UN PROCESADOR RISCV-V SOBRE CHISEL Y FPGA

(Implementation and Extension of a RISC-V Processor based on Chisel and FPGA)

Trabajo de Fin de Máster
para acceder al

Máster Universitario en Ingeniería Informática

Autor: Noé Ruano Gutiérrez
Director: Pablo Prieto Torralbo
Julio - 2025

TODO:

Resumen

TODO:

Abstract

TODO:

Índice

TODO:

1 Introducción

1.1. Motivación

En la actualidad existen principalmente dos modelos de negocio en el ámbito del diseño y/o fabricación de procesadores. Por un lado estarían aquellas compañías cuyo beneficio proviene principalmente de la venta de licencias para el uso de sus diseños, como pudiera ser el caso de ARM, al tiempo que existen otras compañías que centran su actividad empresarial tanto en el diseño como en la fabricación y venta de hardware (casos de Intel o AMD), sin perjuicio de que pudieran obtener beneficios también del pago por el uso de su propiedad intelectual con fines comerciales [1].

Ambos modelos se basan en la siguiente premisa: que la especificación del conjunto de instrucciones que deberá ejecutar un computador constituye por sí mismo una entidad patentable. Esto obliga a todo aquel que quisiera obtener beneficio de la comercialización de productos derivados de esas especificaciones, a pagar regalías al tenedor de la propiedad intelectual, lo cual obstaculiza de manera notable el surgimiento de innovaciones que pudieran aprovechar las ya existentes en materia de conjuntos de instrucciones y arquitecturas de largo recorrido como son x86-64 o ARM, obligando a todos aquellos sujetos que no pudieran costearse el pago de las licencias a partir de cero y diseñar sus propios ISAs.

No obstante, en los últimos años del siglo XX y principios del XXI, surgieron corrientes que apostaban por la creación de arquitecturas libres, con proyectos como Sparc u OpenRISC, este último aún en desarrollo [2]. Pero no sería hasta el año 2011 [3] cuando, en la Universidad de Berkeley, comenzaría a gestarse el que es sin lugar a dudas el ISA abierto que mayor renombre ha conseguido tomar hasta la fecha: RISC-V. Tanto es así, que gigantes tecnológicos como Google, Intel, Microsoft o Nvidia [4] realizan importantes aportaciones monetarias de manera regular a la fundación que mantiene y desarrolla la especificación (RISC-V International), claro está, con objeto de obtener un beneficio futuro a partir de la comercialización de productos basados en esta nueva tecnología.

Además de compañías privadas, multitud de instituciones públicas contribuyen, por supuesto, a impulsar el desarrollo de la especificación. En el ámbito europeo, la EPI[1] es la iniciativa mediante la cual se canalizan los esfuerzos de la academia e industria europeas por hacer realidad el proyecto de una arquitectura abierta, robusta y capaz, que sitúe a la Unión Europea a la vanguardia de la computación a exaescala. Y es este uno de

[1] European Processor Initiative

los principales motivos que han impulsado el desarrollo de este trabajo: la posibilidad de conocer en profundidad y trabajar con una de las tecnologías clave para el futuro tecnológico de Europa, y de España.

1.2. Objetivos

El principal objetivo del proyecto es la implementación de un computador sencillo, mononúcleo, segmentado y en orden, que pueda ejecutar el conjunto de instrucciones definido en la especificación del ISA abierto RISC-V, en su versión de 32 bits. Para ello se partirá de un diseño ya existente, monociclo e incompleto, elaborado por el investigador de la Universidad Nacional Cheng Kung, Ching-Chun (Jim) Huang, quien lo emplea como material docente en una asignatura de arquitectura y tecnología de computadores impartida en esa misma universidad.

El diseño de partida se encuentra descrito en el lenguaje Chisel, que es un lenguaje de propósito específico o DSL (Domain-Specific Language) construido sobre el lenguaje de programación Scala, este último, un lenguaje de propósito general o GPL (General Purpose Language). Esto requerirá las adquisición, por parte del autor del presente trabajo, del conocimiento sobre ambos lenguajes que le permita, a posteriori, afrontar la tarea de completar el pipeline del mencionado diseño original para, más adelante, modificarlo de manera que este se encuentre segmentado en cinco etapas.

Por último, se pretende elaborar una prueba de concepto consistente en el despliegue del diseño final sobre una placa FPGA (Field Programmable Gate Array), de modo que pueda comprobarse sobre hardware real el buen funcionamiento y desempeño de este diseño.

1.3. Planificación y metodología seguidos

1.3.1. Formación previa

En primer lugar se realizó una formación relativa a los lenguajes Scala y Chisel[2]. Esta formación fue creada por diversos investigadores y colaboradores de la Universidad de Berkeley, y consiste en un cuaderno de Jupyter en el que se abordan, incialmente, aspectos introductorios relativos al desarrollo de software en el lenguaje Scala para, posteriormente, proporcionar al usuario, en lecciones de complejidad incremental, nociones fundamentales sobre el diseño de hardware empleando el lenguaje Chisel.

1.3.2. Construcción del computador monociclo

La segunda fase del proyecto consistió en la implementación de las partes incompletas del pipeline original, las cuales se encuentran debidamente indicadas en el código de partida. Esto implicó la puesta en práctica y consiguiente asentamiento de los conocimientos

[2]Disponible en: `www.github.com/freechipsproject/chisel-bootcamp`

adquiridos durante la realización del bootcamp sobre Scala y Chisel, además de que constituyó en sí mismo un proceso de descubrimiento de la arquitectura del núcleo monociclo y sus particularidades.

1.3.3. Construcción del procesador segmentado

A continuación, con el diseño base completado, se procedió a segmentar el pipeline en las cinco etapas siguientes:

1. Lectura de instrucción o fase de *fech*
2. Decodificación de la instrucción
3. Ejecución
4. Lectura/escritura de resultados en memoria
5. Escritura en el banco de registros, o fase de *writeback*

En un primer momento se optó por aplicar un enfoque incremental en lo que respecta a la inserción de los registros de etapa. No obstante, este enfoque se aplicó tan solo en la integración del primer registro de etapa para separar la fase de *fetch* del resto del pipeline. Esto fue así puesto que, de haber seguido integrando registros de etapa uno a uno, verificando el funcionamiento del núcleo con dos, tres y cuatro registros, el proceso de desarrollo se hubiera dilatado en el tiempo más allá de lo deseado inicalmente. Esto hubiera sido así puesto que, para cada una de las integraciones de un registro de segmentación a partir de aquel que separa las fases de *fetch* y decodificación, hubiera sido preciso diseñar una solución distinta al problema que generan las dependencias de datos y de control entre las instrucciones que atraviesan el pipeline.

Es por ello por lo que se consideró que resultaría más eficiente integrar los registros de segmentación siguientes al de *fetch*-decodificación a la vez, desarrollando una única solución al problema generado por las dependencias de datos y de control.

1.3.4. Despliegue del diseño final sobre un FPGA

Para concluir el proyecto se elaboraría una prueba de concepto consistente en el despliegue del diseño elaborado sobre una placa FPGA diseñando, también en lenguaje Chisel, una interfaz con la que poder comunicar el computador con los dispositivos presentes en la placa. Ello posibilitaría el envío, carga en memoria y ejecución de binarios compilados para la arquitectura RISC-V, gracias a un módulo UART desarrollado por el investigador del grupo de Ingeniería de Computadores y director de este trabajo, Pablo Prieto Torralbo.

2 RISC-V

RISC son las siglas de 'Reduced Instruction Set Computer', es decir, un computador con un conjunto de instrucciones reducido. Una de las primeras menciones de este concepto la encontramos en la década de 1970. En aquel entonces, en IBM, tenían la necesidad de desarrollar un computador sencillo, pero capaz de ejecutar 12 millones de instrucciones por segundo, de modo que pudiera ser integrado en un sistema de enrutamiento de llamadas telefónicas [5].

Hasta la fecha los computadores incorporaban arquitecturas CISC[1] con las que eran capaces de llevar a cabo procesos complejos por medio de la ejecución de instrucciones igualmente complejas. Con frecuencia, esas instrucciones debían ser interpretadas a nivel arquitectural y traducidas a un microcódigo con el que se ejecutaban las operaciones necesarias para obtener los resultados esperados. No obstante, este enfoque requería contar, en primer lugar, con un intérprete de bajo nivel que tradujera las instrucciones en sus correspondientes microcódigos, además de una memoria dedicada en la que alojarlos.

Las arquitecturas RISC surgen como respuesta a la tendencia de los conjuntos de instrucciones a ganar complejidad con el tiempo, a pesar de que algunos estudios [6] apuntaban que la mayoría de los programas desarrollados por aquel entonces tan solo empleaban un subconjunto de las instrucciones disponibles. En estos estudios se proponía la creación de arquitecturas más sencillas, baratas de producir y eficientes, las cuales, aún siéndolo, fueran capaces de ejecutar programas complejos de igual modo que lo hacían las arquitecturas CISC.

2.1. Historia de RISC-V

El manual de la primera versión del ISA abierto RISC-V fue publicado en el año 2011. Por aquel entonces, los investigadores de la Universidad de California en Berkeley, Krste Asanović, Yunsup Lee y Andrew Waterman, bajo la dirección de David A. Patterson, integraban el equipo *ParLab*, trabajando en un proyecto de investigación sobre computación paralela, financiado por Intel y Microsoft, y del cual surgieron como derivados del trabajo de investigación tanto el ISA RISC-V, como el lenguaje de construcción de hardware (HDL) Chisel [7].

Más tarde, en el año 2015, se creó la RISC-V Foundation con el objetivo de organizar los esfuerzos de la comunidad global de desarrolladores de software y hardware,

[1] 'Complex Instruction Set Computer', un computador con un conjunto de instrucciones complejo

por impulsar la adopción del estándar, mantenerlo y hacerlo evolucionar de una manera organizada y eficaz. Además, en este mismo año, los integrantes del *ParLab* (a excepción de Patterson) fundarían SiFive, una compañía dedicada al diseño y venta (que no fabricación) de procesadores basados en RISC-V.

2.2. Especificaciones

La especificación la componen dos ISAs, uno estándar y otro privilegiado, siendo el primero aquel sobre el que se ha puesto el foco a la hora de proponer y desarrollar este trabajo. La especificación del ISA privilegiado recoge las instrucciones y modos de ejecución necesarios para ejecutar sistemas operativos y operar con periféricos [8], mientras que la especificación del ISA no privilegiado recoge las instrucciones necesarias para la construcción de arquitecturas RISC-V básicas pero completamente funcionales.

El ISA no privilegiado lo componen, a su vez, cuatro especificaciones básicas junto con sus respectivas extensiones. Esas especificaciones básicas deben estar presentes en toda implementación que quiera hacerse del ISA, es decir, que una implementación debería integrar una de las cuatro especificaciones base, junto con tantas extensiones como se requiera. Las especificaciones base recogen el conjunto de instrucciones básico que todo núcleo RISC-V debería ser capaz de ejecutar [9], y son las siguientes:

- **RV{32,64}I**: ISAs con un XLEN (longitud de registro) de 32 y 64 bits.
- **RV{32,64}E**: subconjuntos de RV32I y RV64I con la mitad de registros. Elaborados para propiciar la construcción de microcontroladores de tamaño reducido.

Las instrucciones de las especificaciones base posibilitan la ejecución de operaciones aritmético-lógicas (suma, resta, operaciones lógicas, *lui*[2] y *auipc*[3]), saltos condicionales y no condicionales, así como operaciones de lectura y escritura en memoria [10]. En el ISA base se definen, además, las operaciones de ordenación de memoria necesarias para garantizar la consistencia en sistemas con múltiples hilos de ejecución paralelos o (*harts*) [11].

Asimismo, se definen las instrucciones 'ECALL' y 'EBREAK' cuya utilidad es, respectivamente, realizar llamadas al sistema y pausar la ejecución de un *hart* para inspeccionar el estado del banco de registros y la memoria (útil en labores de depuración) [12].

Por último, la especificación indica el formato de otro tipo de instrucciones las cuales, aún siendo válidas, pueden no tener ningún efecto sobre el estado de la arquitectura. Estas instrucciones, denominadas 'HINTs', se reservan para casos en que se desee agregar una cierta funcionalidad al ISA, la cual no tenga por qué ser adoptada obligatoriamente por una arquitectura ya existente, pudiendo esta simplemente ignorar la instrucción [13].

[2] *load upper immediate*: carga en un registro un entero de 32 bits construido con los 20 bits más significativos de la instrucción (inmediato), a los cuales se aplica un shift lógico de 12 bits hacia la izda.

[3] *add upper immediate (to) program counter*: construye un inmediato de 32 bits exactamente igual que *lui*, lo suma al valor del contador de programa y lo guarda en un registro

2.3. El conjunto de instrucciones base RV32I

En el manual de la arquitectura se indica que en el conjunto RV32I las instrucciones deben poder direccionar a 32 registros de propósito general. No obstante, tan solo se restringe el uso del registro 'x0' para albergar una constante de valor 0, quedando recogido el uso del resto de registros en la ABI[4] de RISC-V [14].

En lo que respecta al formato de las instrucciones, existen cuatro formatos base [15]:

- **R**: operaciones aritmético-lógicas en las que los dos operandos son registros.
- **I**: operaciones aritmético-lógicas en las que uno de los operandos es un inmediato codificado en la propia instrucción. Este formato es el empleado, además, en la codificación de la instrucción de lectura de memoria *lw* y sus derivados, así como en la codificación de la instrucción de salto incondicional *jalr*[5].
- **S**: instrucción de escritura en memoria *sw* y sus derivados.
- **U**: es el formato empleado en la codificación de las instrucciones *lui* y *auipc*.

2.3.1. Formatos derivados

Formato B

El formato B es una variación del S y se emplea en la codificación de las instrucciones de salto condicional. La diferencia entre este formato y el formato S reside en la manera de interpretar los campos con que se construye el inmediato.

En las instrucciones de escritura en memoria, la dirección efectiva de acceso se forma concatenando las dos secciones del inmediato (la más significativa y la menos significativa) dentro de la instrucción, extendiendo el signo del resultado a 32 bits, y sumándolo al contenido del registro *rs1*. Por otro lado, en la decodificación de las instrucciones de salto condicional, el offset respecto del contador de programa se construye de una manera menos intuitiva, tal y como puede verse en la Figura 2.1.

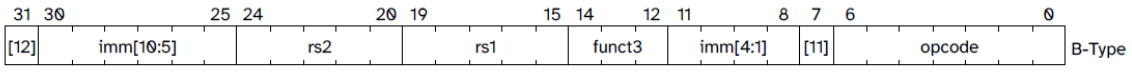

Figura 2.1: Esquema con el formato que deben seguir las instrucciones del tipo B

El offset siempre va a ser un múltiplo de 2, por lo que el último dígito binario del inmediato puede asumirse siempre nulo. Con esto, el inmediato en las instrucciones de

[4] *Application Binary Interface*: una especificación de la forma en que los programas deben ser llamados a nivel de lenguaje máquina (paso de parámetros, retorno de resultados, etc.), así como la forma en que deben construirse y enlazarse los ejecutables que los albergan

[5] *jump and link register*: avanza el contador de programa a una posición igual a su valor actual más el offset formado con la extensión de signo del inmediato a 32 bits. Además, guarda la dirección de retorno (instrucción siguiente al salto) en un registro de destino codificado en la instrucción

salto condicional podría codificar los 12 bits más significativos sigiuentes a ese primer dígito binario, quedando un offset de 13 bits que, posteriormente, habría que extender a 32 bits para sumarlo al contador de programa. En aras de favorecer la reutilización de la lógica de decodificación del inmediato del formato S, se decidió construir el formato B de forma que hubiera tanta superposición como fuera posible con el S, facilitando de este modo la decodificación. En resumidas cuentas, se sacrificó ligeramente la legibilidad del inmediato dentro del formato, favoreciendo la eficiencia en la implementación.

Formato J

El formato J debe emplearse en la codificación de las instrucciones de salto incondicional. Se trata de una variación del tipo U, que sigue la lógica del formato B en cuanto a la construcción del inmediato, tal y como puede observarse en la Figura 2.2 el offset se encuentra codificado en múltiplos de 2 por lo que el último bit va a ser siempre 0, con lo que puede codificarse dentro de la instrucción los 19 bits más significativos inmediatamente posteriores a ese último dígito binario.

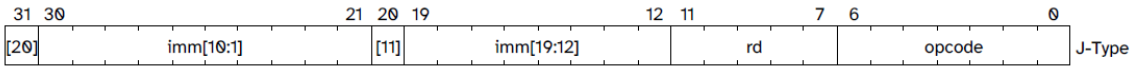

Figura 2.2: Esquema con el formato que deben seguir las instrucciones del tipo J

2.3.2. Extensiones

La arquitectura de la que parte el proyecto no implementaba ninguna extensión, y el alcance del proyecto tan solo abarcaba la segmentación del pipeline. No obstante, en este capítulo dedicado al ISA RISC-V cabe mencionar algunas de las principales extensiones que se contemplan en la especificación.

Extensión 'M'

Para simplificar la implementación del conjunto de instrucciones base se decidió incorporar las instrucciones de multiplicación y división de enteros a su propia extensión en la que se definen, de manera separada, el formato de las instrucciones de multiplicación (MUL, MULHU y MULHSU[6]) y el de las instrucciones de división (DIV[7], DIVU[8] y REM(U)[9]).

[6]Las instrucciones MULHU y MULHSU retornan los $XLEN$ bits más significativos en resultados cuya representación requiera de, a lo sumo, $2 * XLEN$ bits

[7]Cociente con signo

[8]Cociente sin signo

[9]Resto de los cocientes con y sin signo

Extensión 'Zicsr'

La especificación del conjunto de instrucciones base no contempla la presencia de un registro de estado y control o 'CSR'[10]. No obstante a ello, en la especificación del ISA no privilegiado sí se definen un conjunto de operaciones que son consideradas de utilidad en sistemas que gestionen múltiples *harts*. Estas operaciones incluyen el manejo de contadores y temporizadores, así como la notificación del estado de las operaciones de punto flotante [16].

Extensión 'F'

La extensión 'F' recoge las operaciones de punto flotante en precisión simple, y define sus formatos y comportamiento, así como la presencia de un conjunto de 32 registros dedicados, además de un registro de estado propio (*fcsr*) y la disposición de sus campos.

Extensión 'Zifencei'

En esta extensión se detallan el formato y comportamiento de la instrucción *FENCE.I*, empleada en la sincronización entre escrituras sobre la memoria de instrucciones, y las lecturas sobre esa misma memoria [17]. Esta instrucción toma especial relevancia en contextos donde el código de un *hart* que ejecuta en un sistema con arquitectura Harvard[11], puede hacer que se sobreescriba a sí mismo. Tras una escritura en un área de la memoria correspondiente al código del programa, este podría invocar la instrucción *FENCE.I* para garantizar la coherencia en la memoria de instrucciones con respecto a la de datos, que es donde realiza la escritura. La primera, en el momento de ejecutar la instrucción escrita previamente, podría leer en la dirección de la instrucción un dato inconsistente, en función de si alguna lectura de instrucciones anteriores hubiera forzado o no la actualización de esa posición en la memoria.

[10] Control-Status Register

[11] Arquitecturas donde la memoria se encuentra dividida, proporcionando buses dedicados e independientes para acceder a la memoria de datos y a la de instrucciones

3 Herramientas y tecnologías

3.1. Chisel

Chisel es un lenguaje de descripción de hardware de alto nivel gestado en el año 2012 en el seno del Departamento de Ingeniería Eléctrica y Ciencias de la Computación (abrev. EECS) de la Universidad de California, en Berkeley, y surgió como parte de un proyecto de investigación cuyo objetivo final era la implementación de una interfaz de lenguaje con la que elevar el nivel de abstracción ofrecido por HDL's convencionales como son VHDL o Verilog, ofreciendo la posibilidad de traducir los desarrollos elaborados, en entidades descritas en estos mismos lenguajes de bajo nivel, con objeto de correr procesos de síntesis o emulación que permitiesen evaluar su funcionamiento [18].

El hecho de que se trate de un HDL de alto nivel tiene un impacto directo y positivo sobre la celeridad en los procesos de desarrollo y prueba del hardware, permitiendo elaborar y probar diseños de una manera ágil sin perder las características de eficiencia y robustez que proporcionan lenguajes de más bajo nivel.

Es un lenguaje de propósito específico construido sobre Scala, un lenguaje de propósito general orientado a objetos, siendo esta la principal propiedad que permite a Chisel mostrar al usuario una interfaz de diseño de hardware de tan alto nivel y, por ende, amigable tanto para el desarrollador novel como para usuarios más avanzados, y es que resulta muy práctico el poder modelar un circuito lógico como un conjunto de clases relacionadas entre sí. Cada una de esas clases define el funcionamiento interno particular de un componente determinado y, las relaciones entre las clases de las que se compone el diseño global, constituyen las conexiones físicas entre los diferentes módulos de los que este se compone.

3.1.1. Desarrollo en Chisel

La unidad fundamental de diseño de hardware en Chisel son los módulos, que encapsulan la disposición interna y el comportamiento de un circuito lógico. Esta disposición interna pueden componerla otros módulos, cables internos o cables de entrada/salida, cuyos valores serán empleados en la evaluación de las expresiones lógicas que definan el comportamiento del módulo, y que pueden describirse por medio de operadores aritmético-lógicos convencionales, o bien, por medio de los tipos de datos y componentes definidos en las librerías de Chisel, tales como multiplexores, decodificadores o incluso memorias.

A continuación, en la Figura 3.1, se proporciona un código sencillo con el que se describe un multiplexor de 4 entradas de 32 bits cada una. Nótese que los módulos se definen como clases que extienden la clase *Module*, lo cual obliga a contar con, al menos, una interfaz encapsulada en el método *IO*, por medio del cual se definirá esa interfaz como un elemento de entrada/salida. En el caso del código mostrado en la Figura 3.1 las interfaces de entrada/salida se encuentran definidas en las líneas 7 a 14 como parte de una instancia de la clase *Budle*, que es un constructo que permite en Chisel crear nuevos tipos de datos al estilo de los *struct* en el lenguaje C.

```
1  package mux4
2  import chisel3._
3  import chisel3.util._
4
5  class Mux4 extends Module {
6    val io = IO(new Bundle{
7      val i1 = Input(UInt(32.W))
8      val i2 = Input(UInt(32.W))
9      val i3 = Input(UInt(32.W))
10     val i4 = Input(UInt(32.W))
11     val sel = Input(UInt(2.W))
12
13     val o  = Output(UInt(32.W))
14   })
15
16   io.o := MuxLookup(io.sel, io.i1)(
17     Seq(
18       0.U -> io.i1,
19       1.U -> io.i2,
20       2.U -> io.i3,
21       3.U -> io.i4,
22     ))
23 }
```

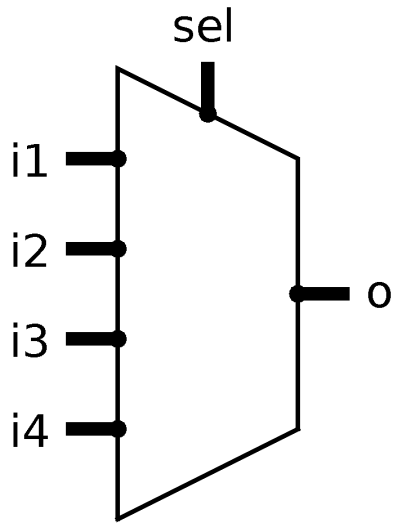

Figura 3.1: Código y diagrama del multiplexor de 4 entradas en Chisel

3.1.2. Pruebas en Chisel

Como se comentaba al comienzo de esta sección sobre Chisel, una de sus características más reseñables es la agilidad con la que puede probarse el comportamiento de los módulos, y ello es gracias a los tests, los cuales son definiciones de clases que implementan uno de los *traits* de Chisel en los que se definen a su vez los procedimientos que permiten evaluar el funcionamiento de los módulos. Estos *traits* son interfaces que definen una serie funciones básicas con las que comprobar y alterar el estado de las señales dentro de los módulos.

Se ha decidido utilizar *ChiselScalatestTester* como interfaz de pruebas en lugar de la más reciente *ChiselSim*, principalmente, porque *ChiselScalatestTester* es la forma de prueba utilizada en el proyecto base, además de que, desde el punto de vista de la implementación, no se encuentran grandes diferencias entre ellas.

Las interfaces de prueba de Chisel definen cuatro funciones básicas con las que evaluar el comportamiento de un módulo: *peek*, *poke*, *expect* y *step*. Estas funciones se invocan sobre las diferentes señales de las cuales se componga el módulo y permiten realizar las siguientes acciones:

- ***peek***: lectura del valor de una señal en el instante de invocación de la función.
- ***poke***: cambio en el valor de la señal.
- ***expect***: evaluación de un valor determinado en la señal, esto es, probar si el valor de la señal en el instante de invocación de la función es el esperado.
- ***step***: se invoca sobre una señal de reloj y permite avanzarlo un ciclo.

En la Figura 3.2 se muestra un ejemplo de test que evalúa el comportamiento del módulo definido en el código de la Figura 3.1. En el código del test se establece el valor de los cuatro puertos de entrada por medio del método *poke* y, luego, se itera con los posibles valores de la señal de selección, comprobando, por medio del método *expect*, que el valor en el puerto de salida es el esperado.

```
1  package mux4
2  import chisel3._
3  import chiseltest._
4  import org.scalatest.flatspec.AnyFlatSpec
5  import mux4.Mux4
6
7  class Mux4Test extends AnyFlatSpec with ChiselScalatestTester {
8    behavior.of("Mux4")
9    it should "output the right input, given any valid selector value" in {
10     test(new Mux4) { c =>
11       c.io.i1.poke(1.U)
12       c.io.i2.poke(2.U)
13       c.io.i3.poke(3.U)
14       c.io.i4.poke(4.U)
15       var x = 0
16       for (x <- 0 to 3) {
17         c.io.sel.poke(x.U)
18         c.io.o.expect(x + 1)
19       }
20     }
21   }
22 }
```

Figura 3.2: Implementación de un test para el multiplexor de 4 entradas

3.2. SBT

SBT es una herramienta de construcción de software semejante a Maven o Gradle, y es la más utilizada por los desarrolladores en lenguaje Scala[19].Su uso es por medio de una interfaz de línea de comandos, proporcionando una consola interactiva desde la que pueden realizarse todo tipo de tareas administrativas sobre los proyectos, tales como lanzar pruebas, ejecutar un programa principal, compilar ficheros de código fuente de manera individual o construir el proyecto, requirendo esto último la creación de un fichero de configuración *build.sbt* dentro del directorio del proyecto.

Escrito también en el lenguaje Scala, en él se definirán las propiedades del proyecto, como puedan ser su nombre y versión, la versión de Scala empleada en la implementación o las librerías de las que depende, siendo estas dos últimas propiedades imprescibdibles de cara al proceso de compilación.

En la Figura 3.3 se proporcionan ejemplos de uso de la consola de SBT, primero, invocando el proceso de elaboración del módulo descrito en la sección 3.1 mediante el comando *compile* y, acto seguido, lanzando el test desarrollado.

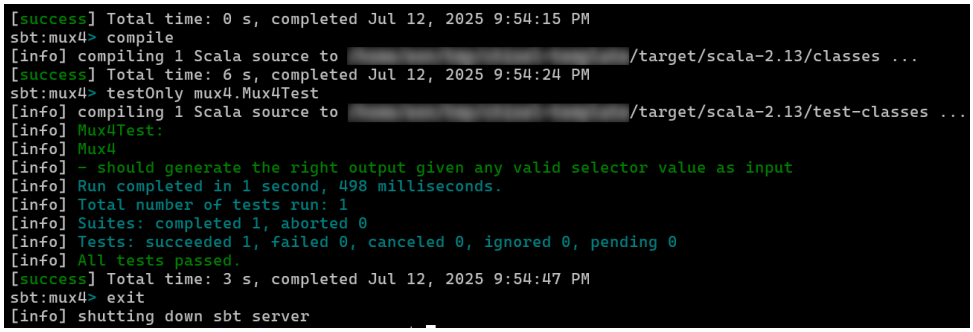

Figura 3.3: Elaboración y prueba del módulo descrito en la sección 3.1

3.3. RISC-V GNU Toolchain

Gran parte de las pruebas llevadas a cabo sobre la implementación del núcleo han consistido en la carga, ejecución y depuración de programas desarrollados en lenguaje C, lo cual ha implicado aplicar un proceso de compilación cruzada para obtener binarios que pudieran ejecutar sobre la arquitectura RISC-V.

Este proceso ha sido llevado a cabo por medio de la suite de herramientas de que se compone el toolchain de GNU para RISC-V[20], disponible en sistemas Linux basados en Debian por medio del gestor de paquetes *apt* (paquete *gcc-riscv64-unknown-elf*).

3.4. GTKWave

La depuración de componentes arquitecturales individuales tales como la ALU o el banco de registros, o incluso fases completas dentro del pipeline, es una tarea que puede acometerse de manera ágil por medio de tests, estableciendo primeramente los casos de prueba a implementar, y verificando que los cambios en las señales bajo supervisión son los esperados para el conjunto de entradas definido en cada caso de prueba. No obstante, este proceso de depuración del pipeline se torna más complejo cuando se desea verificar el comportamiento global de la arquitectura mediante la ejecución de un programa, lo cual requiere la implementación de tests que verifiquen el estado de un número mucho mayor de señales a cada ciclo, incrementando notablemente la complejidad de los tests desarrollados y del proceso de depuración.

Aún siendo esta una tarea ciertamente abordable, se optó por explorar alternativas que permitiesen depurar el estado de la arquitectura de una manera más rápida y visual, y se consideró que la herramienta GTKWave cumpliría perfectamente con este propósito.

GTKWave es una herramienta de análisis empleada en la depuración de trazas generadas en los procesos de simulación de modelos elaborados con Verilog o VHDL. Estas trazas, generadas con la ejecución de los tests de Chisel, se almacenan en ficheros con un formato definido originalmente en el estándard IEEE 1364-1995[21] y constituyen una descripción de todos los cambios experimentados por las señales de los módulos durante la ejecución de las simulaciones.

La herramienta permite explorar esos cambios de una manera visual por medio de una interfaz gráfica en la que puede explorarse el histórico de cambios de la totalidad de las señales que conformen cada módulo, tal y como se muestra en la Figura 3.4, donde se presenta una captura de pantalla de la interfaz gráfica de GTKWave, mostrando una visualización de la traza generada tras la ejecución de un test semejante al mostrado en la Figura 3.2, adaptado para que los cambios en las señales se reflejen en la traza, algo que no se conseguía con la implementación original.

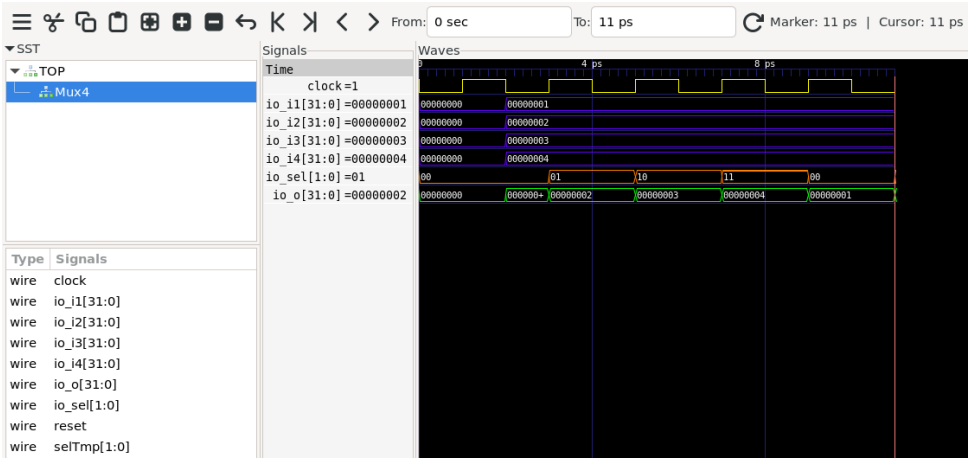

Figura 3.4: Visualización de la traza generada tras la simulación del test

3.5. FPGA Arty A7-100T y AMD Vivado Design Suite

AMD Vivado es un software de diseño hardware para SoC[1] y FPGAs desarrollado y mantenido por la empresa AMD que es, además, el fabricante de la placa de desarrollo Artit A7-100T empleada en las pruebas llevadas a cabo como parte del proceso de verificación final de la arquitectura diseñada.

La placa alberga el FPGA AMD (anteriormente Xilinx) Artix7, con 101440 celdas lógicas[22], superficie más que suficiente para albergar el núcleo segmentado. Además, tal y como puede verse en la imagen de la placa presentada en la Figura 3.5[2], esta cuenta con numerosos elementos de entrada/salida tales como switches de pulsación (señalado como número 10 en la imagen) y lineales (número 9), diodos LED (números 7 y 8), interfaces Ethernet (3) y UART (bajo USB; número 2), así como conectores para modulos de expansión de Diligent (*pmod*[3]; número 16) y shields de Arduino (11).

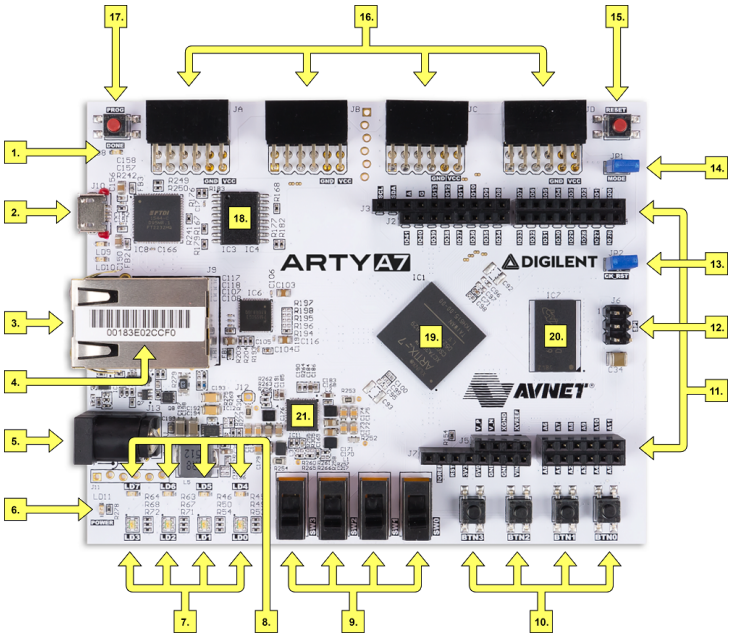

Figura 3.5: Placa de desarrollo Arty A7-100T

3.6. Logisim Evolution

Se trata de un software CAD de código abierto utilizado para el diseño y simulación de circuitos lógicos, y fue la herramienta empleada para la creación de los diagramas de

[1] System-on-Chip: circuitos integrados que incorporan todos o la gran mayoría de los componentes que constituyen un computador

[2] Diligent. (2023). Arty A7 - Diligent Reference [png]. Arty A7 Reference Manual. Disponible aquí

[3] *Peripheral module interface*

la arquitectura que se mostrarán en capítulos posteriores de esta memoria.

3.7. Git

Git fue concebido originalmente por Linus Torvalds como una alternativa de código abierto a BitKeeper, el software de control de versiones propietario empleado en el desarrollo del kernel de Linux entre los años 2002 y 2005[23], y es actualmente el software de control de versiones más utilizado por los desarrolladores, según los datos reflejados en el *Stack Overflow Annual Developer Survey* del año 2022[24]

4 El procesador monociclo

4.1. Descripción de la arquitectura de partida

El código que describe la composición y funcionamiento internos del pipeline original se encuentra distribuido en cinco módulos, cada uno de los cuales modela una de las cinco etapas en las que se ha decidido segmentarlo. Además, existe un módulo superior dentro del cual se establece el conexionado entre las diferentes etapas y, por encima de este, un módulo en un nivel adicional en el que se definen las conexiones entre el agregado de las unidades funcionales del pipeline y la memoria, unificada, la cual alberga datos e instrucciones.

Se proporciona a continuación, en la Figura 4.1, un diagrama con el que se espera poder ayudar a comprender las relaciones entre los distintos componentes que conforman el computador.

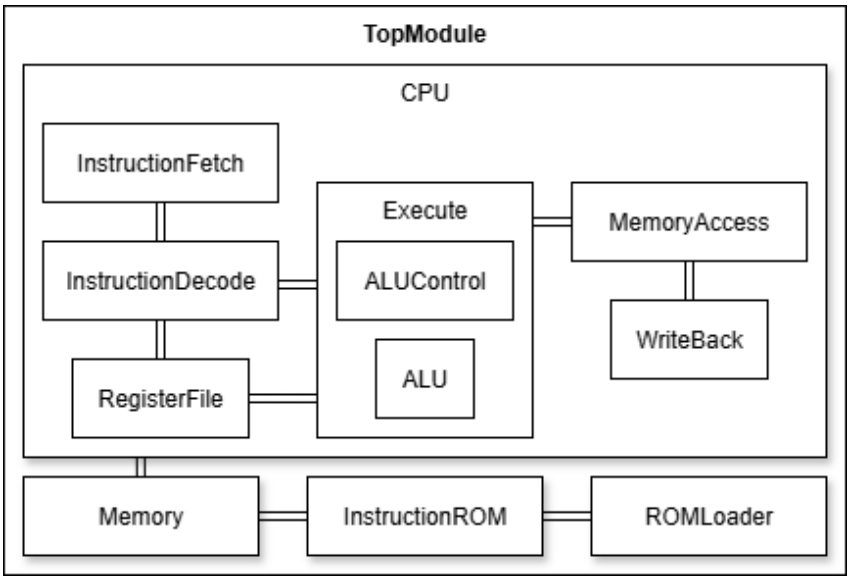

Figura 4.1: Diagrama de bloques de la arquitectura global del computador

A continuación se indicarán los módulos que conforman el diseño de partida junto con una descripción tanto de la funcionalidad que modelan, como de las señales de entrada y salida que reciben y generan respectivamente. Además, cabe indicar que los ficheros de

código fuente donde se encuentra definido cada módulo siguen la siguiente nomenclatura: *«NombreModulo».scala*.

4.1.1. TopModule

TopModule es el módulo en el que se describe el conexionado de los grandes bloques que conforman el computador, es decir, el núcleo, la memoria, el cargador y la ROM de instrucciones, cuyo funcionamiento interno e interoperabilidad serán descritos en los subapartados siguientes a este.

El módulo cuenta con una interfaz de entrada-salida cuyos puertos permiten observar el estado del banco de registros y de la memoria en cualquier momento, lo cual resulta de gran utilidad cuando se necesita depurar la ejecución de programas complejos que puedan sacar a la luz defectos en el diseño del pipeline, los cuales no hubieran podido ser detectados por medio de los tests convencionales que prueban el comportamiento individual de cada uno de los componentes que conforman el diseño global.

4.1.2. Memory

Tal y como se indicaba al comienzo de este capítulo sobre la arquitectura interna del computador, la memoria se encuentra unificada, albergando tanto instrucciones como datos, lo cual contribuye a simplificar el diseño global, haciendo posible que todas las operaciones sobre la memoria sean realizadas por medio de una misma interfaz, definida dentro de un único módulo (*Memory*) que cuenta, además, con una instancia definida dentro del módulo *TopModule*.

Sobre el código del módulo *Memory*, en él se define la clase *Memory* con la que se construyen tanto la memoria unificada del computador, como la interfaz a la que se conecta el módulo *CPU* para habilitar la comunicación entre la memoria y los módulos que compomen el núcleo, más concretamente, *InstructionFetch* y *MemoryAccess*, que emplean los puertos definidos en la interfaz de la memoria para leer de manera independiente, y en paralelo, los datos e instrucciones que en ella se encuentran alojados.

A continuación se detallará el propósito de los puertos de entrada-salida del módulo. No obstante, dentro de la entrada-salida se define también un *bundle* como una instancia de la clase *RAMBundle*, definida dentro del mismo fichero donde se especifica la clase *Memory*. Al instanciar esta clase se crea un *Bundle* de Chisel, es decir, una agrupación de señales de entrada-salida, en este caso, relacionadas con la interacción con la memoria.

Señales de entrada

- ***instruction_address***: es un bus de 32 bits por donde la memoria recibe la dirección de la siguiente instrucción generada en el *fetch*.
- ***debug_read_address***: se trata de otro bus de 32 bits por el que la memoria puede recibir direcciones arbitrarias en cualquier instante de tiempo con fines de depuración.

- ***address@bundle***: la utilidad de este puerto, definido dentro del *bundle*, es indicar una dirección de memoria, cualquiera, para leerla o escribir en ella
- ***write_data@bundle***: es un bus de 32 bits mediante el cual se indica un dato a escribir en memoria
- ***write_enable@bundle***: una señal con la que habilitar o deshabilitar la escritura en memoria
- ***write_strobe@bundle***: tal y como puede verse en la Figura 4.2, donde se muestra el código con el que se implementa la clase RAMBundle, el tipo de dato del puerto de entrada *write_strobe* es un vector. Cada elemento de este vector es un booleano que indica si la sección n-ésima de la palabra a escribir en memoria, donde $0 \leq n < Parameters.DataWidth$, con $Parameters.DataWidth = 4$, deberá o no serlo. Por ejemplo, en las instrucciones de escritura de bytes individuales, el vector con el que se define el 'strobe' deberá tener la siguiente forma para indicar que deberá escribirse el último de los 4 bytes que conforman el dato: {0001}.

Señales de salida

- ***instruction***: la instrucción leída en la dirección *instruction_address*.
- ***debug_read_data***: el dato leído en la dirección *debug_read_address*.
- ***read_data@bundle***: el dato leído en la dirección *address@bundle* con la máscara indicada en el *strobe*.

```
1 class RAMBundle extends Bundle {
2   val address     = Input(UInt(Parameters.AddrWidth))
3   val write_data  = Input(UInt(Parameters.DataWidth))
4   val write_enable = Input(Bool())
5   val write_strobe = Input(Vec(Parameters.WordSize, Bool()))
6   val read_data   = Output(UInt(Parameters.DataWidth))
7 }
```

Figura 4.2: Implementación de la clase RAMBundle

4.1.3. InstructionROM y ROMLoader

Puede verse en el diagrama de la Figura 4.1 que el computador lo componen, además del núcleo y la memoria, dos piezas adicionales: *InstructionROM* y *ROMLoader*. Estos son dos módulos involucrados en la carga de binarios en memoria. El primero define internamente una rutina con la que se traslada el contenido del fichero binario resultante de un proceso de compilación cruzada, a un fichero de texto codificado en ASCII con el que puede inicializarse una memoria de Chisel, definida también a nivel interno dentro del módulo. Posteriormente, la lógica que define el comportamiento del módulo *ROMLoader*

se encarga de trasladar, palabra a palabra, los contenidos de la ROM de instrucciones a la memoria de la CPU, a partir de un punto de entrada recibido como parámetro.

Por último, el módulo *ROMLloader* cuenta con una señal de salida con la que indica al módulo superior *TopModule* que la carga del programa en memoria ha finalizado, con lo que la ejecución podría dar comienzo. Mientras la carga del programa sigue en curso, el *bundle* de la memoria (*Memory*) permanece conectado a un bundle definido dentro de *ROMLoader*, de modo que pueda hacerse el volcado del contenido de la memoria definida dentro de *InstructionROM*, a la memoria que empleará más adelante el core. Cuando el cargador finaliza su trabajo, se desacopla de la memoria y esta se conecta al core ya con el programa cargado y listo para ejecutar.

En la Figura 4.3 se proporciona el fragmento del código de la clase *TOPModule* con el que se construye la lógica que define el comportamiento descrito en este último párrafo. Puede verse que el acoplamiento entre los *bundles* del cargador y la memoria, o entre esta y el núcleo, se realiza por medio del operador ´<>´, con el que se realiza el conexionado de señales con el mismo nombre.

```
1 when(!rom_loader.io.load_finished) {
2   rom_loader.io.bundle <> mem.io.bundle
3   cpu.io.memory_bundle.read_data := 0.U
4 }.otherwise {
5   rom_loader.io.bundle.read_data := 0.U
6   cpu.io.memory_bundle <> mem.io.bundle
7 }
```

Figura 4.3: Implementación de la clase RAMBundle

4.1.4. CPU

Este módulo constituye uno de los grandes bloques que conforma el computador, el núcleo, y en su interior se describe el conexionado entre los distintos bloques que modelan cada una de las etapas por las que pasa una instrucción cuando es leída de la memoria. El funcionamiento de estas etapas se describirá en los subapartados siguientes a este.

Además de este conexionado, se define una interfaz de entrada-salida con los siguientes puertos:

Puertos de entrada

- ***instruction***: la instrucción leída de la memoria, en la dirección que genera el módulo *InstructionFetch*.
- ***instruction_valid***: una señal que controla el flujo de ejecución de nuevas instrucciones, en tanto que puede detener su lectura, paralizando el pipeline. Esta entrada se conecta al puerto de salida *load_finished* del cargador, con el que se indica que

la carga de instrucciones en la memoria principal del computador ha finalizado, puediendo dar comienzo a la ejecución del programa.

- ***debug_read_address***: un bus de 5 bits por el cual puede indicarse el índice de un registro cuyo valor quiera conocerse. Esta dirección es recibida por medio de su correspondiente puerto de depuración en el módulo superior (*TopModule*).

Puertos de salida

- ***debug_read_data***: el puerto de depuración donde el núcleo deposita el contenido del registro solicitado.
- ***instruction_address***: la dirección de la siguiente instrucción generada en el *fetch*.

- ***device_select***: el computador puede tener conectados hasta 8 dispositivos mapeados en memoria, cada uno de los cuales se identifica por medio de una etiqueta de 3 bits ($\log_2 8 = 3bits$), y es este el puerto por medio del cual se indica el dispositivo con cuyo espacio de memoria se quiere interactuar. En la práctica este puerto carece de utilidad, puesto que el único dispositivo con cuya memoria puede trabajarse desde el núcleo es la propia memoria principal. No obstante, queda claro que este mecanismo resultaría muy conveniente si se quisiera dotar al computador de comunicación de entrada-salida con una mayor variedad de dispositivos.

4.1.5. InstructionFetch

En este módulo se describe el comportamiento de la fase de *fetch*, donde se avanza el contador de programa a la siguiente instrucción sumándole 4 unidades[1], siempre y cuando se de alguna de las dos circunstancias siguientes:

- Que el cargador haya terminado de volcar el contenido de la ROM de instrucciones en la memoria principal de la CPU.
- Que el decodificador no haya determinado en el ciclo anterior que la instrucción en ese ciclo era un salto que debía tomarse.

En el primer caso el contador de programa permanecerá inmutable hasta el momento en que el cargador comunique la finalización del proceso de volcado y, en el segundo caso, el contador de programa tomará el valor del destino del salto.

Señales de entrada

- ***jump_flag_id***: es un bit que indica si el valor del contador de programa deberá ser igual al valor actual más cuatro unidades o, por el contrario, la dirección de destino de un salto que debe ser tomado.

[1] Recordemos que se trata de un sistema 32 bits direccionable a byte, por lo que la siguiente instrucción se encontrará alejada 32 bits (4 bytes) respecto de la actual

- ***jump_address_id***: una señal de 32 bits por donde se recibe la dirección de destino calculada en la ejecución de las instrucciones de salto.
- ***instruction_read_data***: la instrucción leída de la memoria en la posición del contador de memoria.
- ***instruction_valid***: la señal con la que el cargador indica que su trabajo ha finalizado.

Señales de salida

- ***instruction_address***: la dirección de la instrucción a leer de la memoria en el siguiente ciclo.
- ***instruction***: la instrucción leída de la memoria en el ciclo actual. Se hace un *passthrough*, es decir, una asignación directa, entre este puerto y el puerto *instruction_read_data*.

Originalmente el código de este módulo se encontraba incompleto, puesto que carecía de la implementación de la lógica que establece el siguiente valor del contador de programa, es decir, el destino de un salto o la dirección de la siguiente instrucción en el programa. Esta lógica, mostrada en el código de la Figura 4.4, consiste en un multiplexor cuyo selector se conecta a la señal *jump_flag_id*, y cuyas dos entradas son, por un lado, el destino de un salto recibido por el puerto *jump_address_id* y, por otro lado, el valor del contador de programa más cuatro unidades (recordemos que se trata de una arquitectura de 32 bits).

```
1 pc := Mux(io.jump_flag_id, io.jump_address_id, pc + 4.U)
```

Figura 4.4: Conexionado del contador de programa a la salida del multiplexor que determina su valor a cada ciclo en función de los saltos que se producen en el flujo de ejecución de los programas cargados en el computador

4.1.6. InstructionDecode

Como su nombre indica, esta es la clase que modela la lógica encargada de realizar la decodificación de las instrucciones recibidas desde el módulo de *fetch* para elaborar el conjunto de señales que componen la palabra de control, por medio de las cuales se gobernará el comportamiento del resto de unidades funcionales en el pipeline, de modo que estas actúen conforme al significado de las instrucciones decodificadas.

Se trata quizás del módulo con mayor complejidad de entre todos de los que se compone el pipeline, y esto se debe al elevado número de casuísticas que deben contemplarse en la generación de la palabra de control, que son el resultado directo de la variabilidad en cuanto a formatos de instrucciones dentro del ISA, así como de la variabilidad en el conjunto de valores posibles que pueden tomar los campos definidos en estos formatos.

Señales de entrada

- ***instruction***: la instrucción leída en la fase de *fetch*.

Señales de salida

- ***regs_reg{1,2}_read_address***: las direcciones de los registros cuyos valores se emplearán en el cómputo de los resultados de instrucciones aritmético-lógicas, o en la determinación del cumplimiento de la condición de salto en las instrucciones de salto condicional.
- ***ex_immediate***: inmediato son signo, extendido a 32 bits.
- ***ex_aluop{1,2}_source***: señales de control mediante las cuales se selecciona la procedencia de los operandos de la ALU.
- ***memory_{read,write}_enable***: señales de control mediante las cuales se habilita la lectura o escritura en la memoria.
- ***wb_reg_write_source***: señal de control con la que se selecciona el origen del dato a escribir en el banco de registros en la fase de *writeback*.
- ***reg_write_enable***: señal de control mediante la cual se habilita la escritura en el banco de registros.
- ***reg_write_address***: la dirección del registro en el que debe escribirse un resultado en la fase de *writeback*.

Dentro de este módulo tan solo fue preciso implementar la lógica correspondiente a la habilitación de la lectura o escritura en memoria, así como de la selección del origen del resultado a escribir en la fase de *writeback*, tal y como puede verse en el código expuesto en la Figura 4.5. Esta lógica se materializa por medio de un par de multiplexores: el primero de ellos gobierna la habilitación de la lectura en memoria y, el segundo, la escritura en memoria.

```
1  io.memory_read_enable := Mux(
2    opcode === InstructionTypes.L,
3    true.B,
4    false.B
5  )
6  io.memory_write_enable := Mux(
7    opcode === InstructionTypes.S,
8    true.B,
9    false.B
10 )
```

Figura 4.5: Generación de las señales de habilitación de lectura/escritura en memoria

4.1.7. InstructionExecute

La clase *Execute* modela el comportamiento del hardware implicado en las operaciones que se llevan a cabo en la fase de ejecución y que son, por un lado, el cómputo de resultados de operaciones aritmético-lógicas en la ALU[2] y, por otro lado, la evaluación de las condiciones de salto junto con el cómputo del siguiente contador de programa en caso de resultar positivas esas evaluaciones.

La ALU se construye dentro del módulo *Execute* por medio de una instancia de la clase *ALU*, la cual cuenta con tres puertos de entrada (operación y primer y segundo operandos), y uno de salida (resultado). Además, su comportamiento es gobernado en base a las salidas generadas por una instancia de la clase *ALUControl*, por medio de cuya lógica se infiere el tipo de operación que deberá llevarse a cabo, en función de los valores que tomen los campos *opcode*, *funct3* y *funct7* en la instrucción.

A continuación se detallan las señales que conforman la entrada-salida del módulo, así como la función que desempeñan dentro de este.

Señales de entrada

- ***instruction***: la instrucción completa, de la cual se extraerán los campos *opcode*, *funct3* y *funct7*.
- ***instruction_address***: la dirección de la instrucción en ejecución. Este valor se empleará en el cómputo de direcciones efectivas de salto, así como del valor resultante de la ejecución de la instrucción *auipc*.
- ***reg1_data***: el valor leído en la dirección del primer registro fuente.
- ***reg2_data***: el valor leído en la dirección del segundo registro fuente.
- ***immediate***: el inmediato codificado en la instrucción, previamente extendido a 32 bits en la fase de decodificación.
- ***aluop1_source***: una señal de un bit con la que se indica cuál deberá ser el valor empleado como primer operando en la ALU. Esta señal controla la salida de un multiplexor cuyas entradas son la dirección de la instrucción y el valor leído en la dirección del primer registro fuente.
- ***aluop2_source***: una señal de un bit con la que se indica cuál deberá ser el valor empleado como segundo operando en la ALU. Esta señal controla la salida de un multiplexor cuyas entradas son el inmediato y el valor leído en la dirección del segundo registro fuente.

[2] Arithmetic-Logic Unit: Unidad Aritmético-Lógica

Señales de salida

- **mem_alu_result**: el valor calculado en la ALU.
- **if_jump_flag**: una señal de un bit con la que se indica el resultado de la evaluación de la condición de salto.
- **if_jump_address**: en caso de resultar positiva la evaluación de la condición de salto, la dirección efectiva de este.

Bibliografía

[1] Krste Asanović y David A Patterson. «Instruction sets should be free: The case for risc-v». En: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146* (2014). Consultado el 30 de agosto de 2025. URL: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.pdf.

[2] OpenRISC Community. *Architecture*. Consultado el 28 de mayo de 2025. 2025. URL: https://openrisc.io/architecture.

[3] Tony Chen y David A. Patterson. *RISC-V Geneology*. Inf. téc. UCB/EECS-2016-6. Accedido el 28 de mayo de 2025. University of California at Berkeley, 2016, pág. 2. URL: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-6.pdf.

[4] RISC-V International. *RISC-V Landscape: Members*. Consultado el 28 de mayo de 2025. 2025. URL: https://riscv.landscape2.io/?group=members.

[5] John Cocke y Victoria Markstein. «The evolution of RISC technology at IBM». En: *IBM Journal of research and development* 34.1 (1990). Consultado el 23 de agosto de 2025, págs. 4-11. URL: https://ieeexplore.ieee.org/document/5389855.

[6] Andrew S Tanenbaum. «Implications of structured programming for machine architecture». En: *Communications of the ACM* 21.3 (1978). Consultado el 30 de agosto de 2025, págs. 237-246. URL: https://dl.acm.org/doi/abs/10.1145/359361.359454.

[7] RISC-V International. *About RISC-V International*. Consultado el 30 de agosto de 2025. 2025. URL: https://riscv.org/about/#history.

[8] Andrew Waterman et al. *The RISC-V Instruction Set Manual, Volume I: Unprivileged Architecture (User-Level ISA)*. Versión 20250508, Consultado el 31 de agosto de 2025. RISC-V International, Unprivileged ISA Committee. Mayo de 2025, pág. 9. URL: https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view.

[9] Andrew Waterman et al. *The RISC-V Instruction Set Manual, Volume I: Unprivileged Architecture (User-Level ISA)*. Versión 20250508, Consultado el 31 de agosto de 2025. RISC-V International, Unprivileged ISA Committee. Mayo de 2025, págs. 15-17. URL: https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view.

[10] Andrew Waterman et al. *The RISC-V Instruction Set Manual, Volume I: Unprivileged Architecture (User-Level ISA)*. Versión 20250508, Consultado el 31 de agosto de 2025. RISC-V International, Unprivileged ISA Committee. Mayo de 2025, págs. 27-35. URL: https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view.

[11] Andrew Waterman et al. *The RISC-V Instruction Set Manual, Volume I: Unprivileged Architecture (User-Level ISA)*. Versión 20250508, Consultado el 31 de agosto de 2025. RISC-V International, Unprivileged ISA Committee. Mayo de 2025, págs. 35-37. URL: https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view.

[12] Andrew Waterman et al. *The RISC-V Instruction Set Manual, Volume I: Unprivileged Architecture (User-Level ISA)*. Versión 20250508, Consultado el 31 de agosto de 2025. RISC-V International, Unprivileged ISA Committee. Mayo de 2025, págs. 37-38. URL: https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view.

[13] Andrew Waterman et al. *The RISC-V Instruction Set Manual, Volume I: Unprivileged Architecture (User-Level ISA)*. Versión 20250508, Consultado el 31 de agosto de 2025. RISC-V International, Unprivileged ISA Committee. Mayo de 2025, págs. 38-40. URL: https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view.

[14] Kito Cheng y Jessica Clarke. *RISC-V ABIs Specification*. Ver. August 12, 2025-draft. Consultado el 31 de agosto de 2025. RISC-V International. 2025, pág. 6. URL: https://github.com/riscv-non-isa/riscv-elf-psabi-doc.

[15] Andrew Waterman et al. *The RISC-V Instruction Set Manual, Volume I: Unprivileged Architecture (User-Level ISA)*. Versión 20250508, Consultado el 31 de agosto de 2025. RISC-V International, Unprivileged ISA Committee. Mayo de 2025, págs. 25-26. URL: https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view.

[16] Andrew Waterman et al. *The RISC-V Instruction Set Manual, Volume I: Unprivileged Architecture (User-Level ISA)*. Versión 20250508, Consultado el 31 de agosto de 2025. RISC-V International, Unprivileged ISA Committee. Mayo de 2025, págs. 49-52. URL: https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view.

[17] Andrew Waterman et al. *The RISC-V Instruction Set Manual, Volume I: Unprivileged Architecture (User-Level ISA)*. Versión 20250508, Consultado el 31 de agosto de 2025. RISC-V International, Unprivileged ISA Committee. Mayo de 2025, págs. 47-48. URL: https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view.

[18] Jonathan Bachrach et al. «Chisel: constructing hardware in a scala embedded language». En: *Proceedings of the 49th annual design automation conference*. Consultado el 11 de julio de 2025. 2012, págs. 1216-1225. URL: https://dl.acm.org/doi/abs/10.1145/2228360.2228584.

[19] VirtusLab. *Scala Survey Results 2023*. Consultado el 11 de julio de 2025. 2023. URL: https://scalasurvey2023.virtuslab.com/.

[20] RISC-V International. *riscv-gnu-toolchain*. 2020. URL: https://github.com/riscv-collab/riscv-gnu-toolchain.

[21] «IEEE Standard Hardware Description Language Based on the Verilog(R) Hardware Description Language». En: *IEEE Std 1364-1995* (1996). Consultado el 13 de julio de 2025, págs. 207-218. DOI: 10.1109/IEEESTD.1996.81542.

[22] AMD. *7 Series FPGAs Data Sheet: Overview*. 2020. URL: https://docs.amd.com/v/u/en-US/ds180_7Series_Overview.

[23] Scott Chacon y Ben Straub. *Pro Git*. 2.ª ed. Consultado el 15 de julio de 2025. Berkeley, CA: Apress, 2014. ISBN: 978-1-4842-0076-6. URL: https://git-scm.com/book/en/v2.

[24] Stack Exchange Inc. *2022 Developer Survey*. Consultado el 15 de julio de 2025. 2022. URL: https://survey.stackoverflow.co/2022/#technology-version-control.