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

Índice

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.

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 [1]. Pero no sería hasta el año 2011 [2] 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 [3] 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. 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

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 [4].

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 ellas definiendo su funcionamiento interno particular, y constituyendo esas relaciones, conexiones entre los diferentes módulos de los que se componga el diseño global.

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

```
1  package mux4
2
3  import chisel3._
4  import chisel3.util._
5
6  class Mux4 extends Module {
7    val io = IO(new Bundle{
8      val i1 = Input(UInt(32.W))
9      val i2 = Input(UInt(32.W))
10     val i3 = Input(UInt(32.W))
11     val i4 = Input(UInt(32.W))
12     val sel = Input(UInt(2.W))
13
14     val o  = Output(UInt(32.W))
15   })
16
17   io.o := MuxLookup(io.sel, io.i1)(
18     Seq(
19       0.U -> io.i1,
20       1.U -> io.i2,
21       2.U -> io.i3,
22       3.U -> io.i4,
23     ))
24 }
```

Figura 3.1: Definición de un multiplexor sencillo en Chisel

definen como clases que extienden la clase *Module*, lo cual hace mandatorio que se cuente 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.

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.

Estas interfaces han ido cambiando conforme evolucionaba el propio HDL y, a pesar de no ser la forma de prueba mantenida en las últimas versiones de Chisel, que sería

```scala
package mux4

import chisel3._
import chiseltest._
import org.scalatest.flatspec.AnyFlatSpec
import mux4.Mux4

class Mux4Test extends AnyFlatSpec with ChiselScalatestTester {
  behavior.of("Mux4")
  it should "generate the right output given any valid selector value as input" in {
    test(new Mux4) { c =>
      c.io.i1.poke(1.U)
      c.io.i2.poke(2.U)
      c.io.i3.poke(3.U)
      c.io.i4.poke(4.U)
      var x = 0
      for (x <- 0 to 3) {
        c.io.sel.poke(x.U)
        c.io.o.expect(x + 1)
      }
    }
  }
}
```

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

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

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.

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[5].

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, previa elaboración de este.

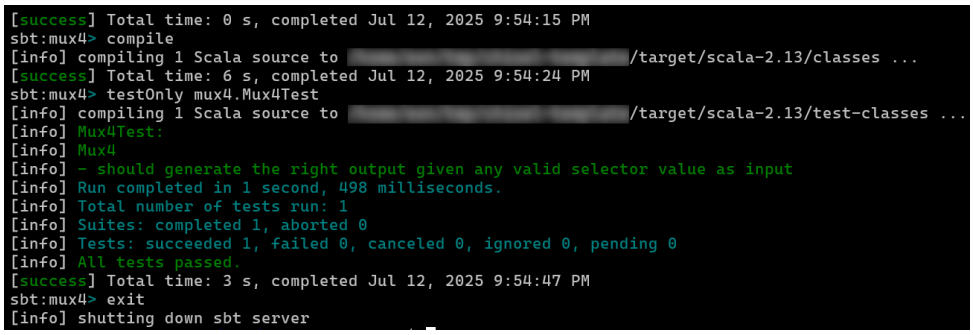

Figura 3.3: Elaboración y prueba del módulo descrito en la sección 3.1

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[6], 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[7] 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.

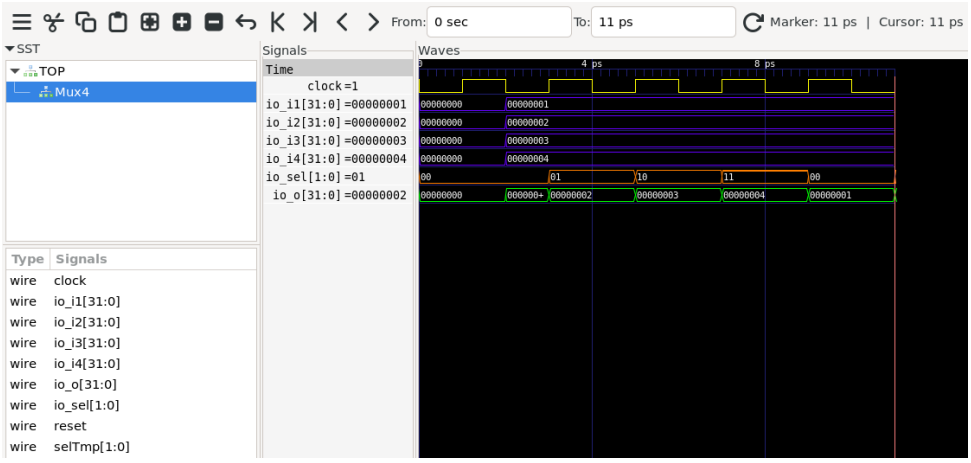

Figura 3.4: Visualización de la traza generada tras la simulación del test

3.5. FPGA Artix A7 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 Artix A7 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) Artix-100T, con 101440 celdas lógicas[8], superficie más que suficiente para albergar el núcleo segmentado.

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 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[9], 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[10]

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

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.

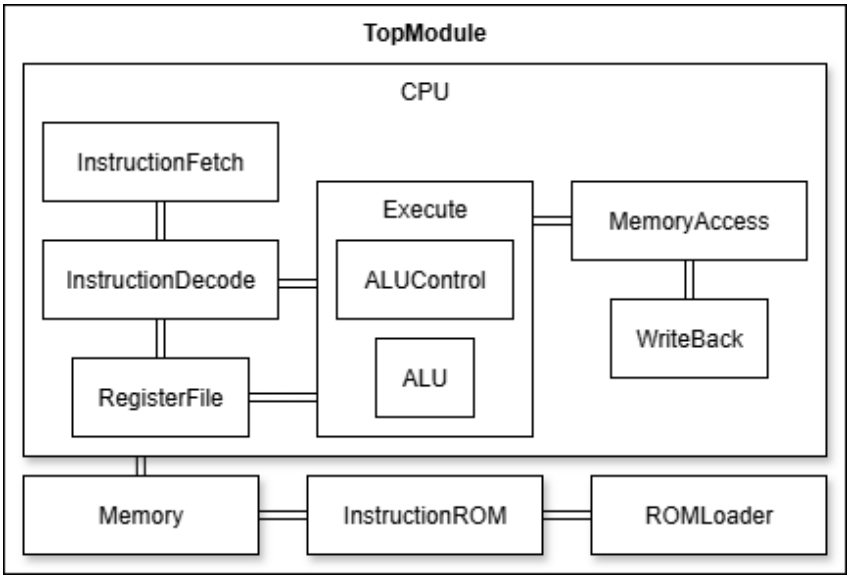

Figura 4.1: Diagrama de bloques de la arquitectura global del computador

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. 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

4.1.2. 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.

```
when(!rom_loader.io.load_finished) {
  rom_loader.io.bundle <> mem.io.bundle
  cpu.io.memory_bundle.read_data := 0.U
}.otherwise {
  rom_loader.io.bundle.read_data := 0.U
  cpu.io.memory_bundle <> mem.io.bundle
}
```

Figura 4.3: Implementación de la clase RAMBundle

4.1.3. InstructionFetch

En este módulo se describe el comportamiento de la fase de *fetch*, donde se avanza el contador de programa en 4 unidades siempre y cuando se de alguna de las dos circunstancias siguientes:

- Que el cargador no 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 que el cargador no comunique la finalización del proceso de volcado y, en el segundo caso, el contador de programa tomará el valor del destino del salto.

Algo a notar en el código es que dentro de este módulo no se define de ningún modo la lectura de la siguiente instrucción. Esto es así porque las operaciones sobre la memoria son llevadas a cabo por la lógica del módulo *Memory*, que cuenta con una instancia definida de manera interna en el módulo *TopModule*.

Señales de entrada

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

Bibliografía

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

[2] 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.

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

[4] 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.

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

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

[7] «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.

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

[9] 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.

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