

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

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.

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 EPI¹ 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

¹European 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 la 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 Chisel². 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, inicialmente, 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

²Disponible 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 *fetch*
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 inicialmente. 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


```

1 package mux4
2
3 import chisel3._
4 import chiseltest._
5 import org.scalatest.flatspec.AnyFlatSpec
6 import mux4.Mux4
7
8 class Mux4Test extends AnyFlatSpec with ChiselScalatestTester {
9   behavior.of("Mux4")
10  it should "generate the right output given any valid selector value as
      input" in {
11    test(new Mux4) { c =>
12      c.io.i1.poke(1.U)
13      c.io.i2.poke(2.U)
14      c.io.i3.poke(3.U)
15      c.io.i4.poke(4.U)
16      var x = 0
17      for (x <- 0 to 3) {
18        c.io.sel.poke(x.U)
19        c.io.o.expect(x + 1)
20      }
21    }
22  }
23 }

```

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, requiriendo 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 imprescindibles 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.

```
[success] Total time: 0 s, completed Jul 12, 2025 9:54:15 PM
sbt:mux4> compile
[info] compiling 1 Scala source to /target/scala-2.13/classes ...
[success] Total time: 6 s, completed Jul 12, 2025 9:54:24 PM
sbt:mux4> testOnly mux4.Mux4Test
[info] compiling 1 Scala source to /target/scala-2.13/test-classes ...
[info] Mux4Test:
[info] Mux4
[info] - should generate the right output given any valid selector value as input
[info] Run completed in 1 second, 498 milliseconds.
[info] Total number of tests run: 1
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 1, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[success] Total time: 3 s, completed Jul 12, 2025 9:54:47 PM
sbt:mux4> exit
[info] shutting down sbt server
```

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 estándar 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.

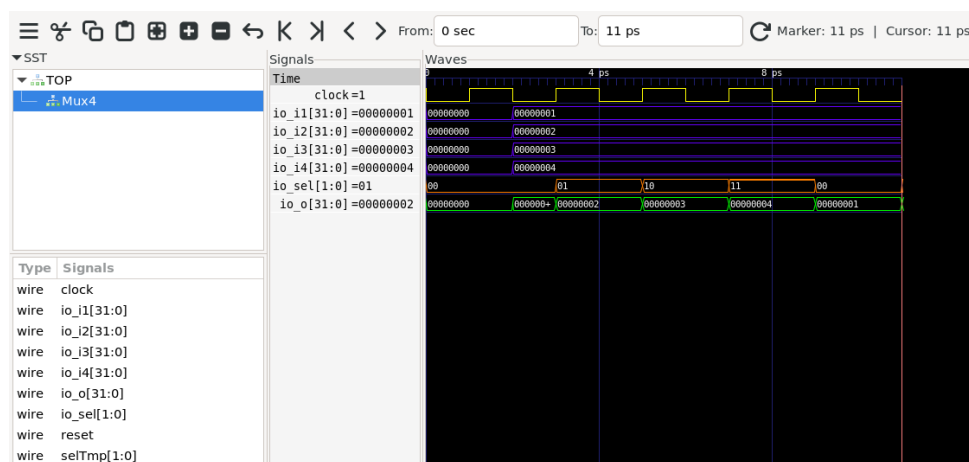


Figura 3.4: Visualización de la traza generada tras la simulación del test

3.5. FPGA Artix A7 y AMD Vivado Design Suite

AMD Vivado es un software de diseño hardware para SoC¹ 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 los 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]

¹System-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

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>.