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 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 considerables 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

TODO:

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 programación software 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].

Chisel es un lenguaje de propósito específico construido sobre Scala, que es, a su vez, 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.

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 TODO:, 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*

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 [.]

```
import chisel3.{RawModule, withClockAndReset}

class Foo extends Module {
  val io = IO(new Bundle{
    val a = Input(Bool())
    val b = Output(Bool())
  })
  io.b := !io.a
}
```

Figura 3.1: Definición de un multiplexor sencillo

```
  apt install -y zip unzip
  curl -s "https://get.sdkman.io" | bash

  sdk install java $(sdk list java | grep -o "\b8\.[0-9]*\.[0-9]*\-tem" |
      head -1)
  sdk install sbt

  sbt --version
```

TODO: indicar cómo se crea un proyecto, hablar del build.sbt, comentar algo del sbt test y el testOnly...

La forma más sencilla de obtener SBT en una distribución linux es mediante el gestor de SDK's, SDKMAN, ejecutando la secuencia de comandos indicada en la Figura TODO:. Se instalan las utilidades *zip* y *unzip*, empleadas por el script de instalación de SDKMAN, así como la versión del JDK considerada más adecuada para ejecutar *sbt* y, por último, el propio *sbt*.

3.3. RISC-V GNU Toolchain

TODO:

3.4. GTKWave

TODO:

3.5. AMD Vivado Design Suite

TODO:

3.6. Git y GitHub

TODO:

3.7. Herramientas de apoyo a la depuración

TODO:

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. 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.