UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Evaluación de procesadores OoO basados en la arquitectura RISC-V

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Derek A. Berger López
Tutor: Julio Sahuquillo Borrás
Tutor Externo: Alberto Ros Bardisa

Curso 2019-2020

Dedico este trabajo a mis padres, a mi hermano y a Andrea, por haber estado en todo momento a mi lado. Sin vuestra ayuda esto no hubiera sido posible.

Resum

A causa dels múltiples simuladors de RISC-V ISA i extensions obertes, es realitza un estudi on s'analitzen avantatges i inconvenients dels nuclis més importants (BOOM, Ariane, Rocket, RiskyOO, ...) fent especial èmfasi en les versions que modelen nuclis que suporten execució fora d'ordre, i que es puguen sintetitzar. Donades aquestes condicions es tria BOOM, encara que també es construeix Rocket Core per dur a terme les primeres proves i comparacions. Es realitzen simulacions modificant paràmetres de BOOM i finalment es fa ús dels comptadors *hardware*. En aquest últim pas s'implementa un nou esdeveniment que detecta si el ROB s'ha bloquejat i es fa ús d'un comptador per registrar el nombre de vegades que això passa, per arribar a les conclusions finals de la feina.

Paraules clau: RISC-V, microarquitectura, comptadors de prestacions

Resumen

Debido a los múltiples simuladores de RISC-V ISA y extensiones abiertas, se realiza un estudio donde se analizan ventajas e inconvenientes de los núcleos más importantes (BOOM, Ariane, Rocket, RiskyOO,...) haciendo especial énfasis en las versiones que modelen núcleos que soporten ejecución fuera de orden, y que se puedan sintetizar. Dadas esas condiciones se elige BOOM, aunque también se construye Rocket Core para llevar a cabo las primeras pruebas y comparaciones. Se realizan simulaciones modificando parámetros de BOOM y finalmente se hace uso de los contadores de rendimiento *hardware*. En este último paso se implementa un nuevo evento que detecta si el ROB se ha bloqueado y se hace uso de un contador para registrar el número de veces que esto ocurre, para llegar a las conclusiones finales del trabajo.

Palabras clave: RISC-V, microarquitectura, contadores de prestaciones

Abstract

Due to the multiple RISC-V ISA simulators and open extensions, a study is carried out where advantages and disadvantages of the most important cores (BOOM, Ariane, Rocket, RiskyOO,...) are analyzed with special emphasis on the versions that model cores that support execution out-of-order, and that can be synthesized. Given these conditions, BOOM is chosen, although Rocket Core is also built to carry out the first tests and comparisons. Simulations are carried out by modifying BOOM parameters and finally hardware performance counters are used. In this last step, a new event is implemented that detects if the ROB has been blocked and a counter is used to record the number of times this occurs, to reach the final conclusions of the work.

Key words: RISC-V, microarchitecture, performance counters

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

Los avances en la estandarización de *software* y *hardware* han acelerado el progreso técnico a una escala global sin precedentes. Esto se ha conseguido través de la colaboración global y el consenso, así como a través del desarrollo de código abierto. El lanzamiento de RISC-V por parte de los investigadores de Berkeley a la comunidad abierta para su estandarización y su mejora continúa de manera que muchas empresas y proyectos europeos han apostado por RISC-V como proyección de futuro. Esto aumentará la probabilidad de empleo en los próximos años.

El término de RISC aparece a principios de los años 80. Una década más tarde, algunos académicos crearon RISC DLX, un *ISA* para uso educativo, aunque no tuvo éxito comercial. Fue en 2010 cuando en la Universidad de California(Berkeley) se decide desarrollar y publicar un pequeño proyecto de corta duración con el objetivo de ayudar a usuarios académicos e industriales. Ahora, la Fundación RISC-V cuenta con la ayuda de organizaciones como AMD, Google, Huawei, IBM, Nvidia, Oracle, Qualcomm y Raspberry Pi, entre otras muchas[30].

1.1 Motivación

La elección de este trabajo está impulsada por el interés específico hacia las asignaturas de arquitecturas de computadores, cursadas a lo largo del grado y de una manera más profunda en la rama de ingeniería de computadores. Tras ello, una primera experiencia en este área en continua evolución gracias a la investigación, tanto académica como industrial, me ilustró las infinitas posibilidades de estudio de los procesadores basados en la arquitectura RISC-V.

La mayor motivación que hay detrás de este trabajo es la posibilidad de poder aportar soluciones a este nicho en específico, que constituye un área de investigación prometedora en la cual la comunidad de código abierto compite con las grandes empresas privadas. Esto supone una gran oportunidad para la sociedad actual, donde el desarrollo y el conocimiento podrán ser compartidos de manera libre de forma que las tecnologías actuales avancen sin la obstaculización que los grandes organismos privados puedan suponer.

A consecuencia de esta liberación tecnológica, se desarrollan diferentes herramientas para la simulación de procesadores basados en la arquitectura RISC-V que facilitan la reproducción e investigación sobre modificaciones en los mismos y, a su vez, hace posible este trabajo.

1.2 Objetivos

Este trabajo fin de grado tiene como objetivo la evaluación de distintos procesadores RISC-V OoO (out-of-order o fuera de orden).

Con ello, se trata de obtener resultados evidentes sobre el impacto que tienen distintas modificaciones de los parámetros que conforman el *core* o núcleo del procesador, detectando así posibles ”cuellos de botella” en las prestaciones, es decir, identificar los principales parámetros que determinan las prestaciones de un procesador determinado cuando ejecuta una determinada aplicación (carga de trabajo).

Esto se realiza mediante el aceso a los determinados ”performance counters” (o contadores de prestaciones) que capturan eventos hardware específicos como fallos de la cache de nivel uno. Se hace uso de varios contadores existentes y se declaran e implementan otros según se necesiten.

Todas las modificaciones e implementaciones dentro del procesador se llevan a cabo mediante la utilización de un lenguaje de diseño *hardware* llamado Scala.

1.3 Metodología

El presente proyecto sigue la metodología descrita en esta sección para la alcanzar los objetivos mencionados de una manera efectiva y eficiente.

Se evalúan con detalle los distintos *cores* basados en RISC-V que puedan ofrecer un estudio avanzado de esta arquitectura y su accesible modificación.

Se lleva a cabo el aprendizaje del lenguaje utilizado en la implementación del procesador para su posterior entendimiento y utilización a la hora de realizar modificaciones e implementaciones de código. La compilación y simulación del *core* a evaluar son necesarios para comprender el funcionamiento de la herramienta.

Se estudia el conjunto de *benchmarks* que se utilizan para la obtención de métricas sobre las prestaciones.

Una vez se conoce el entorno de simulación y sus opciones, se procede a la modificación del *core* y su posterior evaluación respecto a su versión original.

Por último, se añaden contadores *hardware* a través de su implementación en el lenguaje correspondiente con el fin de analizar los cuellos de botella que puedan existir.

1.4 Estructura de la memoria

El documento se compone de 9 capítulos resumidos a continuación:

La introducción se encuentra en el primer capítulo de la memoria, donde se describe la motivación, los objetivos y la metodología a seguir, además de la estructura y algunas convenciones.

En el Capítulo 2 se ilustra el estado del arte y cuál es la propuesta realizada en este trabajo. Además, se aclaran algunas ideas y conceptos sobre RISC-V y sus variantes.

En el Capítulo 3 se analiza cuáles son los posibles puntos débiles de los procesadores diseñados, así como las posibles soluciones a través de la evaluación de estos, y se explica la solución que se propone.

La tecnología utilizada se aborda en el Capítulo 4. En el mismo se presenta qué sistema operativo, herramientas, lenguajes y entornos se utilizan, además de la estimación del coste de aprendizaje que conlleva su uso.

El Capítulo 5 desarrolla la solución propuesta, qué dificultades se han encontrado hasta llegar a la solución y las decisiones que se han tomado.

En el Capítulo 6 se ve la implantación de la solución, es decir, la ejecución de la misma, y se muestran los resultados obtenidos.

En el Capítulo 7 se realizan pruebas para verificar que los resultados anteriores son válidos.

El Capítulo 8 es la conclusión, donde se expone qué objetivos se han alcanzado y en qué medida. Se explica también cómo se relaciona todo este trabajo con los estudios y la rama cursados.

Por último, en el capítulo 9 se presentan posibles ideas e implementaciones que podrían realizarse a partir de este trabajo.

1.5 Convenciones

En este documento existe la siguiente normativa de marcado:

- El código fuente se remarca en ”Times New Roman” sobre un fondo gris
- Las palabras extranjeras se enfatizan
- Se entrecomillan las citas textuales externas al mismo

CAPÍTULO 2
Estado del arte

Durante las últimas décadas, han sido aquellos con suficientes recursos los que han tenido acceso a las últimas y más desarrolladas tecnologías a causa de la privatización empresarial que impide a los más vulnerables poder contar con estas herramientas [17]. Además, la privatización de estos productos impide su uso en la educación , ya que no se dispone de la documentación técnica necesaria para su desarrollo[22].

Es por ello que, con el fin de reducir el escalón de poder entre ellos, surgen en contraposición las comunidades que trabajan para poder ofrecer de manera abierta y libre tecnología que compite con las grandes compañías[9][13]. Incluso la seguridad se ve afectada en este ámbito[22], pues se actúa de una forma mucho más rápida cuando surge cualquier tipo de problema con una comunidad de tal tamaño repartida por todo el mundo. Dos grandes ejemplos son el sistema operativo Linux [18] y la arquitectura de conjunto de instrucciones RISC-V[14].

En la siguiente figura se puede ver como el primer ejemplo ha evolucionado entre las grandes compañías hasta situarse en primera posición en cuanto a supercomputación, llegando a liderar en su totalidad las primeras 500 posiciones en el TOP500.

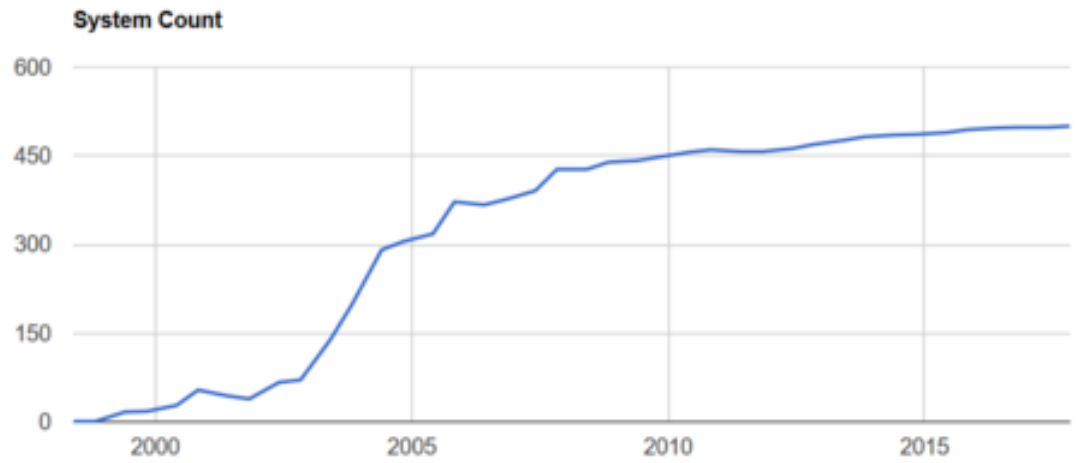

Figura 2.1: Evolución número de Supercomputadoras Linux a lo largo de los años. Obtenido en [48].

Con el foco sobre RISC-V, la acogida de esta arquitectura ha experimentado un crecimiento en los últimos años que deja evidencia de que la tecnología abierta al público

avanza de una forma vertiginosa. Tanto es así que cada vez más compañías hacen uso de ella, como se puede observar en la Figura 2.2.

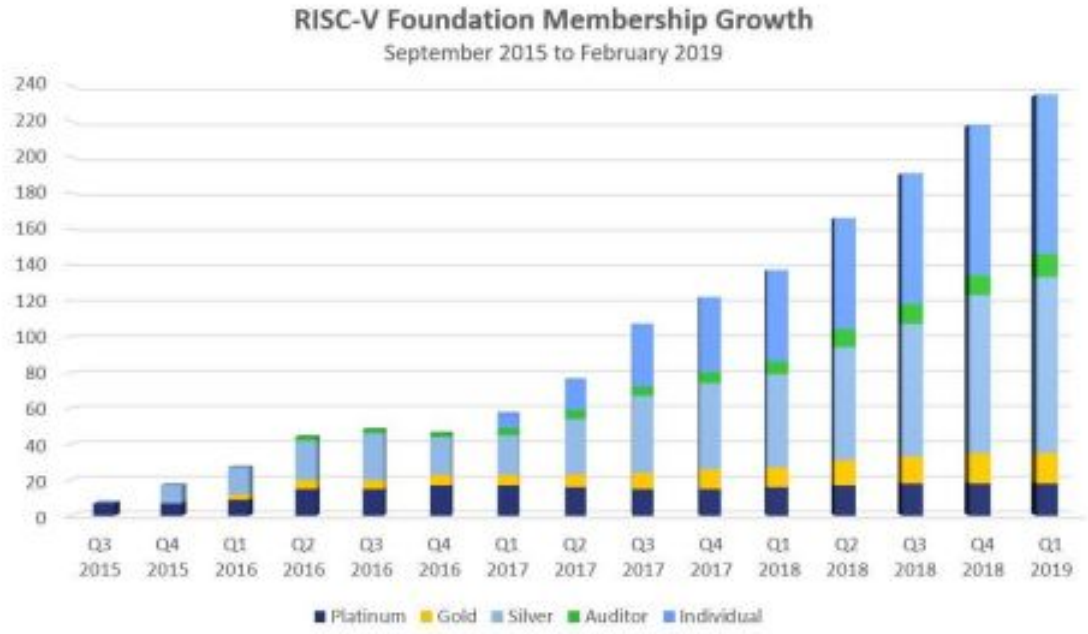

Figura 2.2: Crecimiento del número de membresías por trimestre. Obtenido en [49].

Esto se traduce en un cambio a la hora de innovar en las empresas [21], ya que estas toman ventaja de la flexibilidad, escalabilidad y extensibilidad que aporta RISC-V. De esta forma, adoptan esta arquitectura y además contribuyen activamente en su desarrollo aprovechando su futuro uso.

En la actualidad, numerosos procesadores han sido desarrollados en base a esta arquitectura para competir contra los procesadores comerciales más utilizados y fabricados sobre otras arquitecturas como ARM, sobre la que se basan la mayoría de microprocesadores que se encuentran en los dispositivos electrónicos como lo son los *smartphones* [24]. De esta manera, RISC-V consigue enfrentarse cara a cara con grandes nombre como Intel o AMD.

Algunos de los procesadores desarrollados con esta arquitectura *open-source* más conocidos son Rocket Core, BOOM, RiscyOO o Ariane[23]. Estos mismos son objeto de estudio de este trabajo, por lo que son evaluados y se trabaja con uno de ellos según la prioridad establecida.

Tanto Linux como RISC-V son explicados y descritos con detalle con sus respectivas secciones más adelante en otro capítulo.

2.1 Crítica al estado del arte

Como cualquier procesador existente, aún quedan posibles mejoras según el rendimiento obtenido para las cargas de trabajo a los que están destinados cada uno de ellos[2].

Esto se debe a que las distintos productos desarrollados no responden siempre a la mejor configuración ante todo tipo de exigencias.

A día de hoy, se han desarrollado trabajos dirigidos a la implementación física de procesadores basados en RISC-V como podemos ver en [3] o en [4]. En cambio, en ninguno de ellos existe una evaluación de los ya existentes ni se han implantado nuevos contadores *hardware* para verificar las prestaciones de éstos de una forma más detallada.

2.2 Propuesta

Por lo expuesto en la sección anterior y, dado que esta arquitectura en particular depende en gran parte del desarrollo y los avances que realiza la comunidad, se elige uno de los *cores* nombrados anteriormente con el objetivo de construirlo, simularlo y realizar mediciones en el mismo para distintas cargas o *benchmarks* con el objetivo de detectar posibles cuellos de botella.

Para poder detectar estos puntos críticos se debe hacer uso de distintos *HPE (Hardware Performance Counters)* de forma que se obtengan datos del rendimiento del procesador y con ello poder analizar la estructura del mismo, proponiendo posteriormente algunas modificaciones que puedan resultar en una mejor actuación ante la misma carga.

CAPÍTULO 3
Análisis del problema

La oferta de procesadores hoy en día es abrumadora, pues existen numerosas soluciones ofertadas por parte de las empresas que diseñan los procesadores utilizados en todo tipo de productos electrónicos. Y esto no es diferente en el caso de RISC-V. Como se ha comentado en el capítulo anterior, existen cada vez más soluciones en base a esta arquitectura, pero estas, naturalmente, no abordan todos los escenarios posibles bajo el mejor rendimiento.

Además de lo comentado previamente, algunas de estas soluciones proporcionan implementaciones de contadores en el código liberado. En cambio, en ocasiones estas implementaciones no existen, al menos para todos los eventos que se desean medir en particular.

Es por ello que, si se desea conocer con detalle el rendimiento de cada uno de los aspectos que sean de interés o el punto donde este se ve afectado de forma negativa, es necesario hacer uso de estos contadores e incluso de su implementación.

3.1 Identificación y análisis de soluciones posibles

Una vez conocido el problema, se podrían identificar los posibles puntos débiles de una de estas estructuras desarrolladas a través de la simulación de diferentes *benchmarks* o cargas de trabajo con el objetivo de proponer algunas modificaciones posteriormente con las que se podría obtener mejores resultados bajo estas actividades.

Dado que existen varias opciones disponibles se evalúan teniendo en cuenta el lenguaje o los lenguajes y el entorno utilizados, si es sintetizable, el orden en el que se ejecutan las instrucciones (en orden o fuera de orden) y sus posibles usos.

A continuación se presentan las opciones consideradas para este estudio.

3.1.1. Ariane

Ariane es un procesador con ejecución en *in-order*, que lanza lanza únicamente una instrucción a ejecución por ciclo o *single-issue* y con un pipeline que consta de 6 etapas e implementa el conjunto de instrucciones RISC-V de 64 bits cuyo repositorio y descripción completa se pueden encontrar en [25]. *In-order* quiere decir que las instrucciones se lanzarán y ejecutarán en el orden en que se encuentran dentro del código donde son capturadas. Con la ejecución en orden, las prestaciones se ven seriamente afectadas res-

pecto a la ejecución fuera de orden, pues por cada operando necesario para su ejecución no disponible se insertarán ciclos de parada que supone una demora en el tiempo total de ejecución. La estructura del mismo se muestra en la Figura 3.1. Este procesador está escrito en lenguaje Verilog.

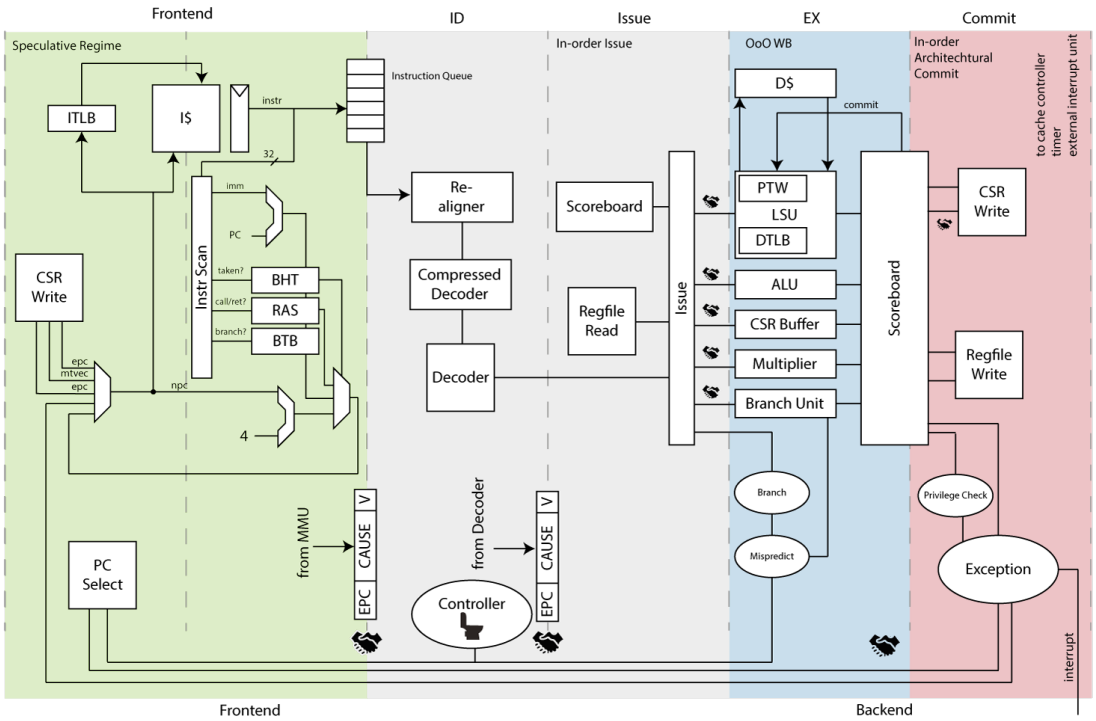

Figura 3.1: Estructura del procesador Ariane. Obtenido en [50].

Verilog es un lenguaje de descripción de *hardware* o *HDL*, aunque tiene un preprocesador como C ya que comparten una sintaxis similar, pues sus diseñadores tenían como objetivo que el lenguaje fuera sencillo para los usuarios que lo usaran, en este caso los ingenieros[26]. Además, se ha usado en los trabajos ya mencionados [3] y [4].

Este procesador es sintetizable, por lo que se podría implementar físicamente sobre una *FPGA (Field-programmable gate array)* lo que permitiría su uso y evaluación real ya que estaría construido físicamente como cualquier otro procesador[27].

3.1.2. BOOM

BOOM (Berkeley Out-of-Order Machine) es una de las apuestas de la Universidad de California en Berkeley. En este caso el procesador está implementado a través del lenguaje de diseño de hardware Chisel, el cual se presenta más adelante. También implementa el conjunto de instrucciones de 64 bits y es sintetizable para *FPGAs*. También es totalmente parametrizable, lo que daría la ventaja de poder personalizarlo al interés de este trabajo. La característica más destacable es la ejecución fuera de orden, que nos permite aumentar considerablemente el rendimiento de este[28].

Además de todo lo indicado anteriormente, su uso está enfocado a la investigación de nuevas arquitecturas.

3.1.3. RiscyOO

Para el procesador RiscyOO se encuentran una características muy similares al que posee BOOM en cuanto al tipo de ejecución y su posibilidad de sintetización, aunque este está escrito en Verilog.

Otra de sus diferencias sustanciales se encuentra en sus posibles usos, siendo la construcción de enclaves seguros sobre esta plataforma el destino más extendido[5].

3.1.4. Rocket Core

El último *core* que se toma como posible objeto de estudio es el procesador Rocket. Es un núcleo en orden de 5 etapas que implementa la ISA RV64G y la memoria virtual basada en páginas. Otras características también son la utilización del lenguaje Chisel y su posible sintetización sobre un *FPGA*.

El propósito original del diseño del núcleo Rocket era permitir la investigación arquitectónica de coprocesadores vectoriales al actuar como el procesador de control escalar. Algunos de esos trabajos se pueden encontrar en [29].

3.2 Solución propuesta

Una vez se han identificado los posibles núcleos a los que se podrían someter esta evaluación, se procede a realizar una comparativa entre sus características y se propone uno de ellos según los criterios definidos a continuación en orden de importancia:

- Se priorizan aquellos que consten de una ejecución fuera de orden.
- La sintetización sobre FPGA es un factor determinante para su posterior uso real.
- Se valora positivamente que el lenguaje proporcione más posibilidades de desarrollo.

Teniendo lo anteriormente descrito en cuenta, se realiza la Tabla 3.1 y se comparan los procesadores.

	Ariane	BOOM	RiscyOO	Rocket
Lenguaje	Verilog	Chisel	Verilog	Chisel
Orden	En orden	Fuera de orden	Fuera de orden	En orden
Sintetizable sobre FPGA	Sí	Sí	Sí	Sí

Tabla 3.1: Comparativa procesadores RISC-V

Después de realizar la comparación entre ellos y siguiendo los criterios de prioridad establecidos, se opta por elegir el procesador BOOM como objeto de estudio de este trabajo por ser un *core* cuya ejecución es fuera de orden, por su capacidad de ser sintetizado

posteriormente sobre un *FPGA* y por utilizar un lenguaje que brinda el mayor número de opciones al desarrollo.

CAPÍTULO 4
Diseño de la solución

En este capítulo se extienden muchos de los conceptos y tecnologías ya mencionadas o brevemente descritas en capítulos anteriores, y se presentan además otras nuevas.

Después de haber realizado un análisis sobre las características más destacables de cada procesador, se elige BOOM como núcleo a construir y medir en base a su rendimiento. Para ello se hace uso del repositorio Chipyard (véase Subsección 4.1.2). Para ello se debe utilizar el sistema operativo Linux, aunque es posible trabajar también desde macOS. Dado que *Chipyard* incluye la opción de construir Rocket Core y Ariane y ambos son *in-order*, también desarrollaremos el core Rocket (pues este está escrito también en Chisel, véase Subsubsección 4.1.2 y Subsección 4.1.4). De esta manera se puede comparar la diferencia de rendimiento (aproximada, ya que tienen diferentes estructuras y la diferencia entre sus rendimientos bajo las mismas condiciones de escalaridad podría variar) entre la ejecución en orden y la fuera de orden.

Para construirlos es necesario hacer uso de uno de los tres simuladores que proporciona Chipyard, conocido como Verilator. Una vez hecho esto y comprobado su funcionamiento, se pasa a la ejecución de los *benchmarks* y otros programas que se desarrollen para observar los resultados que se obtienen.

Conocido el funcionamiento de las herramientas principales, se procede a profundizar en las distintas versiones y opciones que se ofrecen para el procesador BOOM, y se pasa a construir aquellas que sean de mayor interés para seguir desarrollando la evaluación.

Una vez determinadas las mejores versiones del procesador, se accede a manipular los ficheros escritos en Scala después de haber realizado un estudio general del lenguaje para su mejor comprensión y posible escritura básica. Se añade una nueva opción al fichero de configuraciones del simulador construido con las opciones disponibles tratando de obtener la mejor configuración y por ende los mejores resultados en cuanto al rendimiento.

Posteriormente, con el deseo de poseer más contadores que nos permitan averiguar la demora del procesador en situaciones de alta ocupación de los operadores en ejecución, se implementan uno o varios de ellos con el fin de obtener esos datos.

De esta manera se concluye este trabajo, en el que se muestran las conclusiones a las que se han llegado después de la implementación y evaluación a través de contadores de todas las versiones de los procesadores bajo las cargas de trabajo que se han utilizado.

4.1 Tecnología utilizada

Después de determinar el diseño de la solución se explica toda la tecnología utilizada, incluyendo sistemas operativos, entornos o *frameworks*, lenguajes de programación, simuladores, *benchmarks* y *cores*.

4.1.1. RISC-V

RISC-V (pronunciado 'Risk-Five') es una arquitectura de conjunto de instrucciones (ISA) de hardware libre basado en un diseño de tipo RISC (conjunto de instrucciones reducido)."[30]

Así es como está definido en Wikipedia y como se ha definido anteriormente. Este conjunto de instrucciones difiere del resto en que es libre y abierto, pues cualquiera puede diseñar, fabricar y vender sus propios chips y software basado en este[6].

El objetivo principal de esta tecnología novedosa (pues el proyecto se inició en 2010) es poder denominarse como una arquitectura de conjunto de instrucciones *universal*, y para ello debe:

- Acoplarse a todos los procesadores.
- Funcionar correctamente con un gran número de paquetes software y lenguajes de programación.
- Poder implementarse en todo tipo de tecnologías (*FPGAs*, chips personalizados, etc.)
- Ser eficiente para todo tipo de microarquitecturas
- permitir una alta especialización (para poder ser usado en aceleradores personalizados).
- Ser estable, es decir, que la arquitectura del conjunto de instrucciones no varíe.

"El objetivo de la Fundación RISC-V es mantener la estabilidad de RISC-V", de forma que este sólo evolucione de una forma precavida y suave si hay mejoras técnicas suficientemente sólidas detrás, pretendiendo así conseguir que se convierta en un éxito en el área del *hardaware* así como lo ha sido y es Linux en el área de los sistemas operativos.

Pero, ¿por qué se desarrolla un nuevo *ISA*? Además de las ventajas que ya se han comentado en la introducción de este trabajo y en esta última sección existen otras muchas razones por las cuales se llevó a iniciar este proyecto[19]:

- **Las *ISAs* comerciales son propietarias.** La mayoría de los propietarios de las *ISAs* comerciales protegen su propiedad intelectual y no son partícipes de las implementaciones competitivas disponibles gratuitamente. Esto es un problema menor para la investigación académica y la enseñanza, pero es un problema de una magnitud mucho mayor para los grupos cuyo objetivo es compartir implementaciones *RTL* reales.

- **Las *ISAs* comerciales sólo son populares en ciertos dominios de mercado.** Los ejemplos más claros son que la arquitectura ARM no está bien soportada en el área de los servidores, y la arquitectura Intel x86 no está bien soportada en el área de los móviles, aunque ambos están intentando invadir los segmentos de mercado del otro.

- **Las *ISAs* comerciales no son estables.** Existen arquitecturas de investigación anteriores que se han construido en base a *ISAs* comerciales que ya no son tan populares o que incluso ya no se siguen produciendo. Esto pierde el beneficio que ofrece un entorno de software activo. También puede ocurrir que una *ISA* abierta quede relegada a un segundo plano, pero cualquier parte interesada o que haga uso de ella puede seguir desarrollando ese entorno.

- **Las *ISAs* comerciales populares son muy complejas.** Las *ISAs* comerciales dominantes (como las dos mencionadas en párrafos anteriores) son muy complejas de implementar en *hardware* al nivel de apoyo de pilas software y sistemas operativos. Lo peor de todo esto es que se debe a decisiones de diseños no apropiados de *ISAs* más que características que puedan realmente mejorar la eficiencia.

- **Las *ISAs* comerciales solas no son suficientes para producir aplicaciones.** Incluso si dedicáramos nuestro esfuerzo en implementar una *ISA* comercial no sería suficiente para correr aplicaciones existentes en ella, dado que la mayoría de aplicaciones necesitan de interfaces completas y estas, a su vez, dependen del soporte de sistemas operativos. Esto se convierte en algo vastamente complejo de implementar.

- **Las *ISAs* comerciales no fueron diseñadas para su extensibilidad.** Las *ISAs* comerciales dominantes no fueron especialmente diseñadas para su extensibilidad, y como consecuencia ha añadido una considerable complejidad a la hora de codificar instrucciones ya que el conjunto de estas ha aumentado.

- **Una *ISA* comercial modificada es una nueva *ISA*.** Uno de los principales objetivos de RISC-V es apoyar la investigación de arquitecturas, incluyendo las principales extensiones de *ISAs*. De hecho las modificaciones pequeñas disminuyen el beneficio de usar una *ISA* estándar, ya que se debería modificar los compiladores y reconstruir las aplicaciones desde el código fuente para poder utilizar esa extensión.

Dadas estas razones, se opta por ofrecer esta tecnología libremente para cubrir necesidades en investigación y educación, y a día de hoy sigue con su cometido compitiendo con las *ISAs* de mayor renombre.

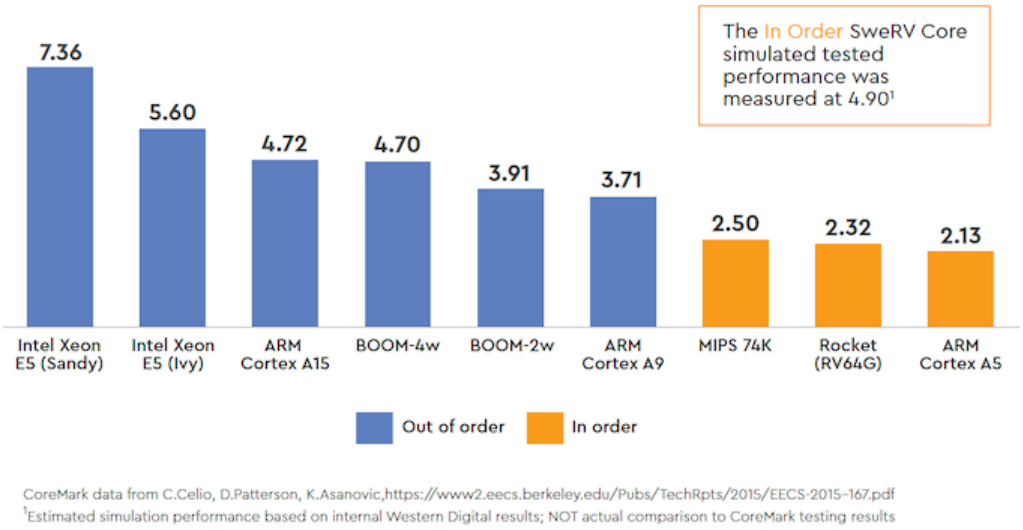

Figura 4.1: Comparativa del rendimiento de cores. Obtenida en [31].

4.1.2. Chipyard

Chipyard es un entorno integrado de diseño, simulación e implementación de *SoC* (*System on a Chip*) para sistemas informáticos especializados. Chipyard incluye bloques IP configurables de código abierto y basados en generadores que se pueden usar en múltiples etapas del flujo de desarrollo de hardware mientras se mantiene la intención del diseño y la coherencia de integración. A través de la simulación acelerada FPGA alojada en la nube y la implementación rápida de *ASIC*(circuito integrado para aplicaciones específicas), Chipyard permite la validación continua de sistemas personalizados físicamente realizables[7].

Es también un entorno de trabajo o *framework* de código libre y, dado que ha sido desarrollado por la Universidad de California en Berkeley, su objetivo principal es ofrecer apoyo a la hora de producir RISC-V *SoCs* a través de sus diferentes proyectos como lo son el ”Chisel HDL” y el generador de ”Rocket Chip SoC”.

Como se ha comentado anteriormente, Chipyard contiene los *cores* Rocket, BOOM y Ariane, y aceleradores como Hwacha o Gemmini. Incluye además los simuladores FireSim, VCS y Verilator, y varios *benchmarks*. Se utilizan los dos primeros procesadores para desarrollar este trabajo junto con los *benchmarks* además del simulador Verilator, y se comentan sus detalles en las secciones posteriores.

Este entorno está en continuo desarrollo por parte del Grupo de Investigación de Arquitecturas de Berkeley, en el Departamento de Ciencias de Computadores e Ingeniería Eléctrica de la Universidad de California, Berkeley.

Existen numerosos recursos para la instalación y uso de cada uno de sus componentes y funcionalidades, aunque se recomienda la documentación oficial proporcionada en el repositorio de Github donde se encuentra[35].

A continuación se presentan en profundidad en forma de subsecciones los componentes que usaremos que se encuentran en Chipyard.

BOOM

BOOM es el procesador principal de este estudio. Sobre él se desarrollan todas las evaluaciones y posteriores modificaciones que se lleven a cabo.

Como indican los propios desarrolladores de Berkeley[8], BOOM es un *core* sintetizable, parametrizable, fuera de orden y superescalar diseñado para servir como de procesador base prototipo para futuros estudios microarquitecturales de procesadores fuera de orden.

Este núcleo está escrito en 9.000 líneas de código *hardware* en el lenguaje Chisel. Como cabe esperar, implementa la arquitectura del conjunto de instrucciones de código libre RISC-V.

Desde su lanzamiento se han llevado a cabo modificaciones que han dado lugar a nuevas versiones del procesador, conocidas como BOOMv2 y BOOMv3 o SonicBOOM.

A continuación se puede ver una pequeña comparativa entre las diferentes versiones de la arquitectura empleada para cada uno:

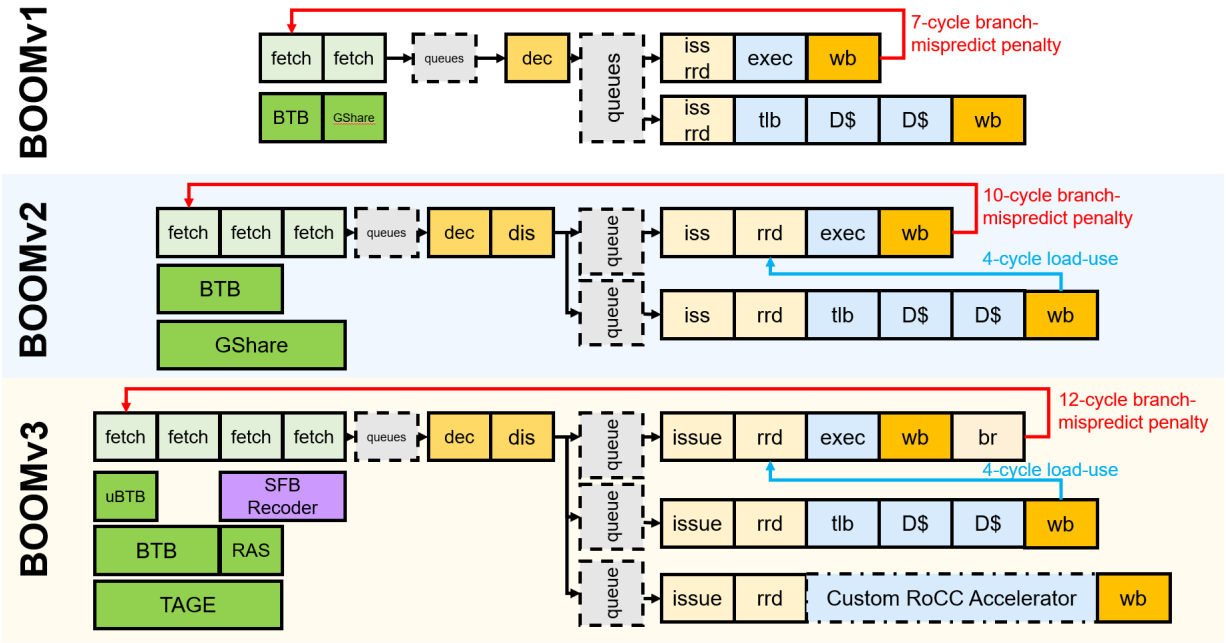

Figura 4.2: Comparativa de las estructuras utilizadas en cada versión de BOOM. Obtenido en [51].

En ella se puede apreciar que la principal diferencia entre las versiones es el tipo de tecnología y arquitectura utilizadas para la predicción de saltos. La penalización es mayor en las últimas versiones, pero se asume un porcentaje de aciertos mucho mayor que aumenta el rendimiento de manera considerable. Esto se puede observar en la Figura 4.3, donde se aprecia fácilmente esta mejora entre la segunda (rojo) y la tercera versión (verde) basándose en la puntuación obtenida para el *benchmark* CoreMark, además de otros *cores*.

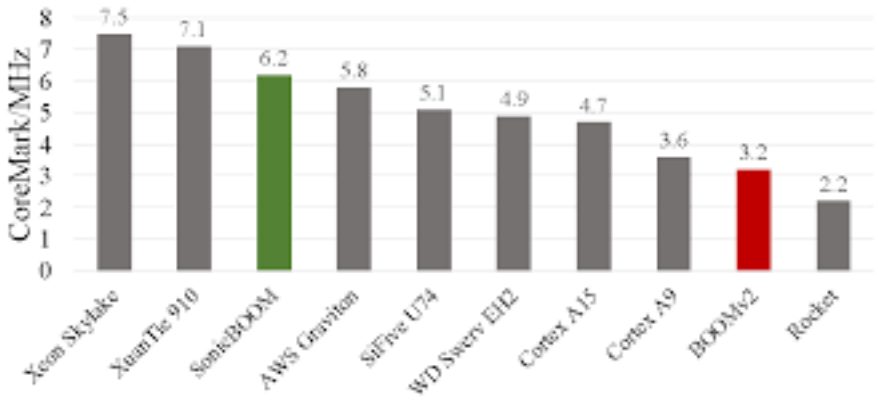

Figura 4.3: Comparativa BOOMv2 vs BOOMv3 y otros. Obtenido en [16].

Rocket Core

Rocket Core es el segundo procesador que usaremos para realizar una primera evaluación y comparación frente a BOOM.

Se trata de un núcleo sintetizable, parametizable, escalar y con ejecución en orden que fue originalmente desarrollado por la Universidad de California y que actualmente está respaldado por SiFive. Proporciona un *pipeline* de 5 etapas e implementa también la versión de 64-bits de la ISA RISC-V.

Además de lo indicado previamente, está escrito en el lenguaje de construcción *hardware* Chisel, al igual que BOOM.

Observando la Figura 4.3 se puede comprobar que el rendimiento de este era inferior a la segunda versión de BOOM, siendo la puntuación de la tercera versión casi 3 veces mayor.

No se entra en detalles para este núcleo, puesto que sirve simplemente para una primera puesta en marcha de la herramienta y una evaluación superficial comparativa sobre los *benchmarks* que se describen posteriormente.

Hardware Performance Counters

Los contadores de prestaciones de *hardware* (o *HPC* por sus siglas en inglés) son un conjunto de registros de propósito especial integrados en los microprocesadores modernos para almacenar los recuentos de eventos o actividades relacionadas con el *hardware* dentro de los sistemas informáticos. Los usuarios avanzados suelen hacer uso de estos contadores a la hora de realizar análisis cuando se da bajo rendimiento o simplemente para ajustes [36].

El número de contadores *hardware* en un procesador es limitado, aunque que cada modelo de procesador podría tener numerosos eventos diferentes que un desarrollador podría querer medir. Cada contador puede ser programado con el índice del tipo de evento que se quiere monitorizar, como lo podría ser un fallo de acceso a la caché L1 o un fallo de predicción de salto.

Estos contadores ofrecen acceso de bajo costo a una gran cantidad de información detallada de rendimiento relacionada con todo tipo de componentes que conforman el *hardware*. Aunque no se necesite modificar el código fuente para usarlos, los tipos y

significados de los contadores pueden variar de un tipo de arquitectura a otro debido a la forma en que estas se organizan.

Uno de los mayores problemas tiene que ver con el reducido número de registros disponibles para almacenar los contadores, lo que provoca que los usuarios realicen numerosas mediciones para obtener todas las métricas de rendimiento que se desea.

Para este trabajo se hace uso de los eventos implementados en cada uno de los procesadores que se construyen (BOOM y Rocket). En las Tablas 4.1, 4.2 y 4.3 se puede ver una descripción de los diferentes tipos de eventos que existen en el procesador BOOM y las máscaras de *bits* que deben usarse en cada contador para medirlos.

mhpeventX[7:0] = 0	
Bits	**Meaning**
8	Exception

Tabla 4.1: Eventos del commit de instrucciones

mhpeventX[7:0] = 1	
Bits	**Meaning**
8	I$ blocked
9	-
10	Branch misprediction
11	Control-flow misprediction
12	Flush
13	Branch resolved

Tabla 4.2: Eventos de la microarquitectura

mhpeventX[7:0] = 1	
Bits	**Meaning**
8	I$ miss
9	D$ miss
10	D$ release
11	ITLB miss
12	DTLB miss
13	L2 TLB miss

Tabla 4.3: Eventos del sistema de memoria

Como se puede observar, se puede hacer uso de varios de ellos dado que se proporcionan directamente, aunque para realizar un análisis más detallado se debe llevar a cabo la implementación de uno o varios de ellos, que es el objetivo de este trabajo.

Para el procesador Rocket, el número de eventos definidos es mucho mayor[37], aunque no se entra en mayor detalle con ellos.

Verilator

Verilator es una herramienta de software gratuita y de código abierto que convierte Verilog (un lenguaje de descripción de hardware) leyéndolo y realizando comprobaciones que se convierten en archivos, que serían el *"Verilated" code* o código de Verilator.

Una vez obtenidos estos archivos, el usuario escribe un pequeño archivo contenedor en C++/SystemC, que instancia el código traducido por Verilator. Es después de esto cuando el archivo o archivos contenedores se compilan (a través del compilador gcc, por ejemplo) resultando en un ejecutable que lleva a cabo la simulación del diseño.

En este trabajo se usa Verilator para descargar, construir y ejecutar simulaciones[38].

Además de lo mencionado anteriormente, Chipyard incluye herramientas para desarrollar cargas de trabajo de programas objetivo o *target software workloads*. Entre los *target* existentes, se utilizará Spike para la simulación de los *benchmarks* y los programas que se han escrito para este trabajo se simulan como programas RISC-V Baremetal:

Spike. Es el simulador de software funcional RISC-V ISA C++ por excelencia. Proporciona emulación de sistema completo o emulación de proxy con Host Target Interface(HTIF)/Frontend Server(FESVR). Sirve como punto de partida para ejecutar software en un objetivo RISC-V[40].

Programas RISC-V Baremetal. Para construir programas RISC-V Baremetal para que se ejecuten en simulación, usamos el compilador cruzado **riscv64-unknown-elf** y una bifurcación de la librería **libgloss**. Para profundizar más sobre cómo se debe compilar un programa Baremetal se debe consultar la documentación en [39].

Benchmarks

En Chipyard se incluye un conjunto de programas de prueba, comúnmente conocidas como *benchmarks*, las cuales utilizaremos para evaluar las diferentes construcciones de los procesadores seleccionados a través de Verilator. Estas se compilan produciendo un archivo ".riscv" que se utiliza en la simulación junto con Spike.

Para su mejor comprensión a la hora de analizar los resultados obtenidos posteriormente, se resume brevemente en qué consisten (aunque algunos de ellos quedan fuera de este estudio, pues no nos interesa ejecutar operaciones *multithreading*, entre otros factores):

- **dhrystone:** Este benchmark se utiliza para medir y comparar el rendimiento de diferentes computadoras o, en este caso, la eficiencia del código generado para la misma computadora por diferentes compiladores. La prueba informa el rendimiento general en Dhrystone por segundo.

  Como la mayoría de los programas de referencia, Dhrystone consta de código estándar y se concentra en el manejo de cadenas. No utiliza operaciones de punto flotante. Está muy influenciado por el diseño de hardware y software, las opciones de compilador y enlazador, la optimización de código, la memoria caché, los estados de espera y los tipos de datos enteros.

- **median:** En este *benchmark* se realiza un filtro de mediana de tres elementos en 1D.

- **multiply:** Este *benchmark* prueba la implementación de la multiplicación software.

- **qsort:** Esta prueba utiliza la ordenación rápida o algoritmo *QuickSort* para ordenar una matriz de números enteros. La implementación está adaptada en gran medida de ”Numerical Recipes for C”.

- **spmv:** Este algoritmo de multiplicación de doble precisión de matriz dispersa por vector (o *Sparse Matrix-Vector*, de ahí sus siglas) es una de las operaciones algebraicas más utilizadas en una gran cantidad de aplicaciones científicas actuales.

  Por ello, se utiliza como carga de prueba para medir el rendimiento en numerosos estudios o trabajos[10].

- **towers:** ”Las Torres de Hanoi” es un clásico problema de rompecabezas. El juego consta de tres clavijas y un juego de discos. Cada disco tiene un tamaño diferente e inicialmente todos los discos están en la clavija más a la izquierda con el disco más pequeño en la parte superior y el disco más grande en la parte inferior. El objetivo es mover todos los discos a la clavija más a la derecha. El problema es que solo se le permite mover un disco a la vez y nunca puede colocar un disco más grande encima de un disco más pequeño.

  Esta implementación comienza con un número de discos especificado y usa un algoritmo recursivo para resolver el rompecabezas.

- **vvadd:** Este último *benchmark* suma dos vectores y lo guarda en un tercero.

4.1.3. Lenguaje de programación C

C es un lenguaje de programación de propósito general originalmente diseñado e implementado en el sistema operativo UNIX.

C es muy valorado por la eficiencia del código que produce y es el lenguaje de programación más popular para crear software de sistema, aunque también se utiliza para crear aplicaciones.

También se dice que C no es un lenguaje de ”muy alto nivel”, aunque el lenguaje C está demostrando ser la herramienta de programación más valiosa para tareas en tiempo real e intensivas en computación[11]. Se trata de un lenguaje de tipos de datos estáticos de medio nivel, ya que dispone de las estructuras típicas de los lenguajes de alto nivel pero, a su vez, dispone de construcciones del lenguaje que permiten un control a muy bajo nivel.

Los compiladores suelen ofrecer extensiones al lenguaje que posibilitan mezclar código en ensamblador con código C o acceder directamente a memoria o dispositivos periféricos[41].

A través de este lenguaje se construyen los programas *baremetal* necesarios para, a través de contadores *hardware*, evaluar los procesadores que sean de interés utilizando el simulador Verilator.

4.1.4. Chisel

Chisel es un nuevo lenguaje de construcción de hardware que admite el diseño de hardware avanzado utilizando generadores altamente parametrizados y lenguajes de hardware específicos de dominio en capas[8]. Al incorporar Chisel en el lenguaje de programación Scala, aumentamos el nivel de abstracción del diseño de hardware al proporcionar conceptos que incluyen orientación a objetos, programación funcional, tipos parametrizados e inferencia de tipos. Chisel puede generar un simulador de software de precisión de ciclo basado en C ++ de alta velocidad, o Verilog de bajo nivel diseñado para mapear ya sea en FPGA o en un flujo ASIC estándar para síntesis[12].

Chisel está destinado a ser una plataforma simple que proporciona características de lenguajes de programación modernos para especificar con precisión bloques de *hardware* de bajo nivel, pero que se puede ampliar fácilmente para capturar muchos patrones de diseño de *hardware* de alto nivel útiles.

También puede generar simuladores C++ rápidos con precisión de ciclo para un diseño o generar Verilog de bajo nivel adecuado para emulaciones sobre un *FPGA* o síntesis *ASIC* con herramientas estándar.

Uno de los puntos clave para la elección del procesador a estudiar se basa en el lenguaje en el que este se escribe. En este caso, se elige tanto BOOM como Rocket Core por utilizar Chisel como lenguaje de programación. Esta preferencia sobre otros lenguajes de descripción *hardware* clásicos radica en que, de manera resumida, Chisel ofrece un mayor número de posibilidades que el resto. Se llega a esta conclusión después de realizar un análisis sobre sus ventajas y desventajas[43][44].

Scala

Scala es un lenguaje muy poderoso con características que son importantes para construir generadores de circuitos. Está específicamente desarrollado como una base para lenguajes específicos de dominio y se compila en la *Java Virtual Machine (JVM)*. Además tiene un gran conjunto de herramientas de desarrollo e *IDE*, y tiene una comunidad de usuarios bastante grande y en crecimiento.

Chisel comprende un conjunto de bibliotecas Scala que definen nuevos tipos de datos de *hardware* y un conjunto de rutinas para convertir una estructura de datos de *hardware* en un simulador de C++ rápido o Verilog de bajo nivel para emulación o síntesis.

El mayor problema de este lenguaje es su curva de aprendizaje[45][46], donde la mayoría de usuarios y desarrolladores expertos coinciden.

Tanto es así, que el ex vicepresidente del grupo de ingeniería de plataformas en Twitter Raffi Krikorian , declaró que no habría elegido Scala en 2011 debido a su curva de aprendizaje . El mismo mes, el vicepresidente senior de LinkedIn, Kevin Scott, declaró su decisión de ”minimizar su dependencia de Scala”. En noviembre de 2011, Yammer se alejó de Scala por motivos que incluían la curva de aprendizaje para los nuevos miembros del equipo y la incompatibilidad de una versión del compilador de Scala a la siguiente[42].

En la Figura 4.4 se puede observar una sencilla representación de cómo se comparan las curvas de aprendizaje de algunos de los lenguajes más populares junto con Scala.

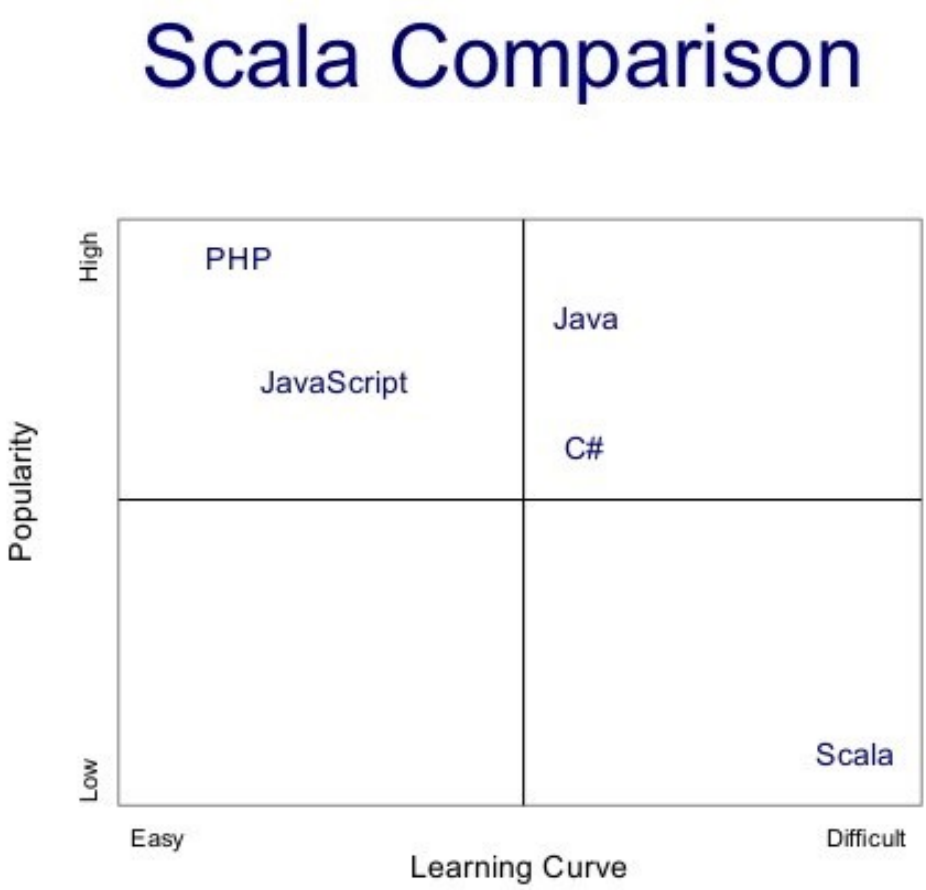

Figura 4.4: Comparativa Scala vs otros lenguajes. Obtenido en [52].

A pesar de ello, se estudia y se manipulan los ficheros fuente de los procesadores que están escritos en este lenguaje con el objetivo de modificar y ampliar algunas líneas de código, algo desafiante por el tiempo que se dispone para este proyecto.

CAPÍTULO 5
Desarrollo de la solución propuesta

En este capítulo se describe con detalle y paso a paso el procedimiento que se ha llevado a cabo hasta que se procede a implantar la solución. Se nombran algunos problemas y dificultades que se han encontrado durante este proceso y se muestran algunos análisis tras las primeras pruebas realizadas.

5.1 Instalación y ajustes de *WSL*

Para comenzar este trabajo, se realiza primero la puesta en marcha de *WSL*. Esto conlleva la descarga del entorno y la posterior actualización de todos los componentes que posee Linux.

Además, la instalación de Chipyard requiere de ciertos paquetes, librerías y otra dependencias para su correcto funcionamiento. También se instala el simulador Verilator antes de continuar.

5.2 Clonación y preparación de Chipyard

Una vez hecho esto, se procede a la clonación del repositorio de Chipyard desde Github. Dentro de la carpeta del entorno se encuentran los *scripts* necesarios para la instalación de las herramientas de RISC-V, que conlleva un largo proceso. Antes de su ejecución se instalan también algunos submódulos.

Una vez finalizado esto y, cada vez que se vaya a utilizar este *framework*, se debe llevar a cabo la exportación de las variables y directorios a usar para este entorno a través de un *script*. De lo contrario, cualquier proceso de ejecución podría fallar.

5.3 Construcción de los procesadores LargeBOOM y Rocket Core

Una vez se ha llegado a este punto, la herramienta está lista para su correcto funcionamiento, y es entonces cuando se procede a realizar la construcción de los procesadores a estudiar.

Para ello, situándose en el directorio del simulador Verilator, se ejecuta la orden *make*. Por defecto, se construye el *core* Rocket, ya que está establecida su configuración como *default* a menos que se especifique una configuración en la ejecución. Puesto que también es de interés, no se indica ninguna configuración.

Para construir el procesador BOOM se debe ir a su carpeta dentro del directorio de generadores, donde se encuentran todos los procesadores, aceleradores y componentes de sistemas. Allí, en el fichero de configuraciones escrito en Scala se encuentran implementadas numerosas configuraciones.

"LargeBoom" es una configuración que intenta simular el procesador ARM Cortex-A15. Dado que en la documentación de Chipyard se utiliza esta configuración para la construcción y simulación de BOOM, se sigue este modelo.

5.4 Primera simulación de los *benchmarks*

Construidos Rocket Core y BOOM, se realiza la primera simulación de los *benchmarks* que se incluyen en las herramientas de RISC-V.

Esto se consigue de forma parecida a la construcción, ejecutando el fichero *Makefile* e indicando tanto la configuración (es decir, el procesador construido) deseada como el juego de *benchmarks* que se desea ejecutar.

En las herramientas de RISC-V se encuentran dos posibles juegos de tests (uno contiene *benchmarks* y el otro posee *tests* en ensamblador) que se pueden utilizar en la simulación para medir el rendimiento de un procesador. El primero de ellos se invoca añadiendo el parámetro "run-asm-tests". Este juego está formado por pruebas de rendimiento escritas en ensamblador, y con ellas se puede medir el rendimiento. Por otro lado, se puede ejecutar el segundo juego añadiendo "run-bmark-tests", realizando así diferentes pruebas con las que se puede medir la actuación del procesador a través de varios tipos de carga.

Una vez tenemos los dos procesadores, se ejecutan los *benchmark* en cada uno de ellos, obteniendo los resultados de la Tabla 5.1. De esta forma, dividiendo los ciclos totales por las instrucciones retiradas totales obtenemos el CPI de las pruebas de trabajo para cada procesador.

Se puede observar en la Figura 5.1 cómo para la mayoría de las cargas LargeBOOM necesita menos ciclos por instrucción. A pesar de ser superior la diferencia de CPI es mínima en los *benchmarks* "median" y "qsort", en cambio en el resto de ellos la diferencia es sustancial, siendo el CPI casi tres veces menor para "spmv".

Una de las principales ventajas que poseen los procesadores con ejecución fuera de orden reside en que puede ejecutar las mismas instrucciones que uno con ejecución en orden pero en diferente orden, de forma que se elabora una "agenda" en principio mucho más eficiente. Esto se debe al uso de la ejecución especulativa y, sobretodo, a la capacidad de recuperación cuando se produce una predicción errónea [15]. Los factores que pueden influir para provocar un rendimiento más bajo que un procesador en orden podrían ser en tamaño del ROB, el *renombrado de registros* o las colas de instrucciones de carga y almacenamiento.

Analizados los resultados obtenidos, se procede a construir nuevos modelos del procesador BOOM.

		LargeBoom	Rocket
dhrystone	*Microseconds for one run through Dhrystone*	177	471
	Dhrystones per second	5632	2121
	Total cycles	88875	235709
	Total retired instructions	196029	196029
median	*Total cycles*	6554	5745
	Total retired instructions	4156	4156
multiply	*Total cycles*	15138	27351
	Total retired instructions	24101	24101
qsort	*Total cycles*	208663	181362
	Total retired instructions	127368	127368
spmv	*Total cycles*	20350	59346
	Total retired instructions	34851	34851
towers	*Total cycles*	3641	6372
	Total retired instructions	6170	6170
vvadd	*Total cycles*	1285	2519
	Total retired instructions	2417	2417

Tabla 5.1: Resultados obtenidos para LargeBoom y Rocket tras la simulación de los *benchmarks*

5.5 Construcción de los modelos Mega y Giga BOOM

Dado que los principales motivos de prestaciones inferiores por parte de un procesador fuera de orden respecto de uno en orden son las características de las estructuras del *buffer* de reordenación de instrucciones y las colas de éstas, se construyen los modelos MegaBOOM y GigaBOOM que se nombran también como *4-wide* y *5-wide* BOOM, respectivamente.

Las principales diferencias entre estos modelos residen en el número de entradas en el ROB, el ancho de la etapa de decodificación y el tamaño de las colas de lanzamiento, además del número de registros, entradas y *buffers* en general.

Una vez se han construido se continúa con la simulación de los *benchmarks*, cuyos resultados se pueden observar en la Tabla 5.2.

De la misma forma que se calculó el CPI para los procesadores LargeBoom y Rocket se obtiene para estos dos nuevos modelos. Este se puede analizar en la gráfica que se muestra en la Figura 5.2.

Se puede observar cómo el modelo LargeBoom obtiene unos resultados inferiores a los que obtienen los modelos Mega y Giga de BOOM, en vista al CPI. Además, parece que los modelos MegaBoom y GigaBoom poseen un rendimiento similar, pero es de interés conocer la verdadera diferencia entre estos, por lo que se normalizan los resultados (mediante el algoritmo MIN-MAX siendo el CPI del peor igual a ”1” y el mejor igual a ”0”, simplemente para resaltar cuál sale más perjudicado en la comparación) en base al CPI obtenido para LargeBoom y se obtienen los que muestra la Figura 5.3.

Se puede ver fácilmente cómo el modelo GigaBoom rinde mejor que el modelo MegaBoom, ya que la diferencia es mínima en las cargas donde necesita más ciclos por instrucción, siendo esta mucho mayor en las que necesita menos.

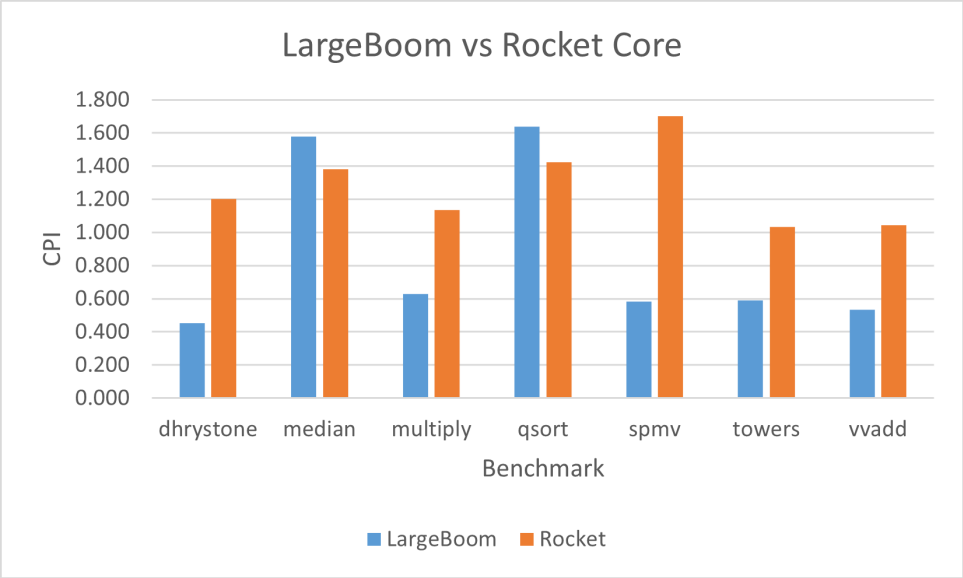

Figura 5.1: Comparativa de los ciclos por instrucción (CPI) de LargeBOOM y Rocket para cada *benchmark*

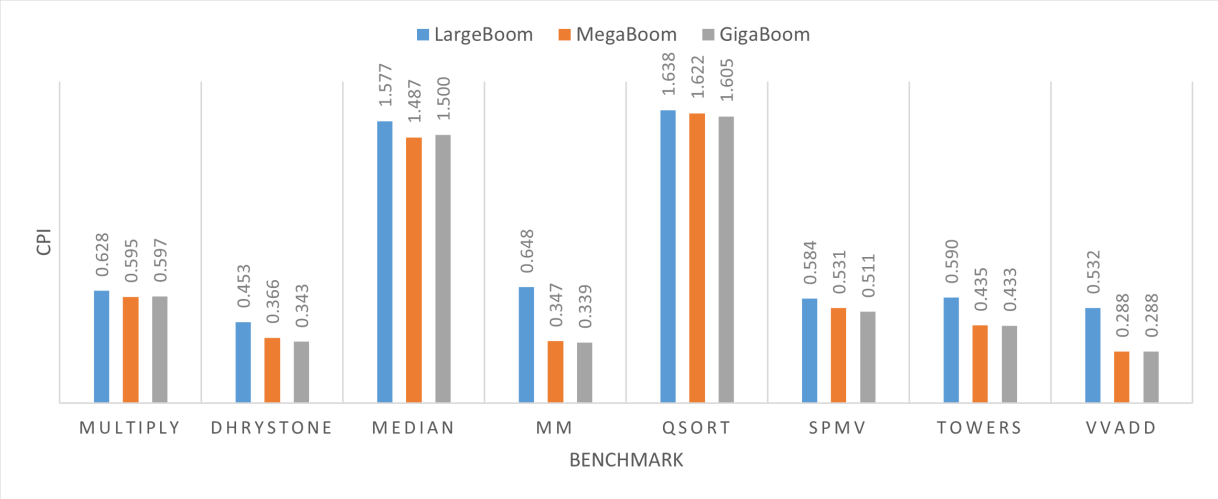

Figura 5.2: Comparativa del CPI obtenido para las versiones Large, Mega y Giga del procesador BOOM bajo las distintas cargas de trabajo

		MegaBoom	GigaBoom
dhrystone	*Microseconds for one run through Dhrystone*	143	134
	Dhrystones per second	6978	7450
	Total cycles	71758	67215
	Total retired instructions	196029	196029
median	*Total cycles*	6178	6235
	Total retired instructions	4156	4156
multiply	*Total cycles*	14332	14390
	Total retired instructions	24101	24101
qsort	*Total cycles*	206609	204385
	Total retired instructions	127368	127368
spmv	*Total cycles*	18521	17804
	Total retired instructions	34851	34851
towers	*Total cycles*	2683	2674
	Total retired instructions	6170	6170
vvadd	*Total cycles*	697	697
	Total retired instructions	2417	2417

Tabla 5.2: Resultados obtenidos para MegaBoom y GigaBoom tras la simulación de los *benchmarks*

En este punto del trabajo se propone trabajar con los *benchmarks* que se incluyen en el conjunto SPEC2017. En este conjunto de cargas se encuentran los *benchmarks* más utilizados para medir el rendimiento de los procesadores, obteniendo información detallada muy útil.

A pesar de su intento de uso, y después de un grande esfuerzo y tiempo dedicado para su inclusión en este estudio, se decide dejarlo de lado después de múltiples errores a la hora de simular los *benchmarks* con los procesadores desarrollados. Probablemente esto se debe a incompatibilidades ocasionadas por el uso de *WSL (Windows Subsystem for Linux)* en lugar de realizar las pruebas de forma nativa sobre el sistema operativo Linux (o incluso utilizar *WSL 2*, en el cual indican haber manejado posibles incompatibilidades en las llamadas al sistema), pero el tiempo no ha permitido cambiar el entorno de trabajo e intentar solucionarlo.

Después de haber construido tanto Rocket Core como BOOM, así como las distintas versiones de este último, se procede a implantar la solución propuesta, pues el rendimiento puede analizarse de una manera más detallada a través de un contador *hardware* que contabilice el número de veces que el evento ”ROB bloqueado” se produce.

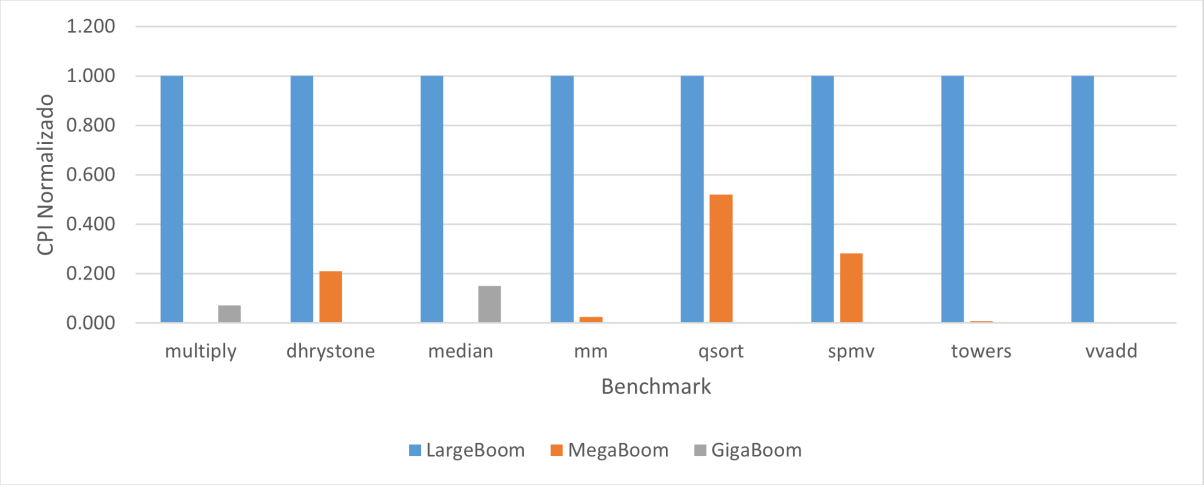

Figura 5.3: Comparativa del CPI (normalizado) obtenido para las versiones Large, Mega y Giga del procesador BOOM bajo las distintas cargas de trabajo

CAPÍTULO 6
Implantación

Finalizada la fase de construcción y simulación de los diferentes procesadores y cargas de trabajo, se avanza a la fase de implantación de la solución propuesta, ya que se necesita conocer con profundidad ciertos datos sobre el *hardware* que ayuden al análisis y la comprensión de los resultados obtenidos. Esto se consigue mediante el uso de contadores de rendimiento del *hardware*.

Para ello se accede al fichero del *core* de BOOM, que está escrito en el lenguaje Scala, y que es el archivo donde se encuentran definidos e implementadas todas las etapas y el funcionamiento del procesador en general, es decir, su estructura y circuitos.

Además de lo anterior, dentro de este fichero podemos encontrar algunos contadores de rendimiento *hardware*, los cuales se pueden observar en las Tablas 4.1, 4.2 y 4.3. Como se ha explicado previamente, se necesita poseer un contador donde se registren las veces que se ha bloqueado el ROB, ya que este indicador podría ayudar a determinar si el número de entradas del mismo ayudaría a mejorar el rendimiento o no, y que es el objetivo de este trabajo.

Observando las tablas no se encuentra un evento que describa lo que se necesita, por lo que se debe implementar un nuevo evento dentro de este fichero que nos permita contabilizar y analizar esos números. Para ello se cuenta con que previamente se haya realizado una lectura de la estructura del lenguaje y de los diferentes ficheros que componen el procesador para la comprensión del funcionamiento de este *core*.

Una vez hecho esto se procede a implementar el código de este evento, que queda resumido en las siguientes líneas de código:

```
1    //
        ------------------------------------------------------------
2    // Uarch Hardware Performance Events (HPEs)
3
4    val perfEvents = new freechips.rocketchip.rocket.EventSets(Seq
        (
5      new freechips.rocketchip.rocket.EventSet((mask, hits) => (
          mask & hits).orR, Seq(
6        ("exception", () => rob.io.com_xcpt.valid),
7        ("nop",       () => false.B),
8        ("nop",       () => false.B),
9        ("nop",       () => false.B))),
```

```
10
11      new freechips.rocketchip.rocket.EventSet((mask, hits) => (
           mask & hits).orR, Seq(
12  //       ("I$ blocked",                          () =>
        icache_blocked),
13         //("nop",                                 () => false.B),
14         // ("branch misprediction",                () => br_unit.
              brinfo.mispredict),
15         // ("control-flow target misprediction", () => br_unit.
              brinfo.mispredict &&
16         //                                          br_unit.
              brinfo.cfi_type === CFI_JALR),

              //("flush",                                () => rob.io.
              flush.valid)

              //("branch resolved",                      () => br_unit.
              brinfo.valid)
17         ("ROB blocked",                          () => !rob.io.ready)
18      )),
```

Simplemente se ha añadido la línea de código de la fila 17 del código que se muestra (realmente línea 260/1476 del archivo "core.scala"). Como se puede ver se ha añadido la descripción "ROB blocked" seguido del evento *hardware*, que simplemente comprueba en cada ciclo si las entradas y salidas (*io*) del ROB están preparadas (*ready*) o si por el contrario no lo están (añadiendo "!" al principio para negar el valor booleano recibido. De esta forma, si se recibe un valor *false* o "0" este se vuelve un *true* o "1" al estar negado y registrándose así un caso afirmativo del evento descrito.

Todo esto se ha añadido en el segundo bloque de eventos, como se puede ver en el código provisto. Esto se traduce en que, para acceder al evento a través de un contador, la máscara de los primeros 8 bits debe ser igual a "1" (ver Tabla 4.2). Además, como es el primer evento que se implementa (pues los demás se han comentado para facilitar su uso, ya que el resto no se utiliza) se debe definir el contador activando también el primer bit de los siguientes bits de la máscara[47], quedando así la línea de código:

```
1  write_csr(mhpmevent3, 0x101); // read ROB Blocked event
```

De esta forma se habilita el contador "mhpmevent3" para la lectura del número de bloqueos del ROB. En este trabajo, en lugar de establecer la máscara para ese contador en un fichero del procesador, se establece la máscara de *bits* directamente en el código del programa. Esto se realiza a través de una función incluida en el fichero de codificación del procesador que permite escribir en los registros de control. Se define para ello una "macro" donde se escriben los bits de la máscara de nuestro evento (0x101). Para leer este registro se debe incluir la definición de la siguiente función en el programa que se vaya a ejecutar:

```
1  #define read_csr_safe(reg) ({ register long __tmp asm("a0"); \
2          asm volatile ("csrr %0, " #reg : "=r"(__tmp)); \
```

```
3           __tmp; })
```

Esta función permite leer de forma segura el registro del contador que indiquemos. Para usarla correctamente primero se debe inicializar una variable de tipo entero sin signo que recibe el primer valor de este registro. Es por ello que los contadores que se vayan a usar se deben colocar en las líneas previas a la ejecución del programa a medir, y después de este se vuelve a medir de forma que la diferencia entre el valor que se obtiene después de la ejecución menos el valor que se obtuvo antes representa el valor neto que se quiere medir. En el siguiente fragmento de código puede verse un ejemplo:

```
1 // read initial value of HPMC's in user mode
2 uint64_t start_hpmc3  = read_csr_safe(hpmcounter3);
3 ...
4 uint64_t start_hpmc31 = read_csr_safe(hpmcounter31);
5
6 // program to monitor
7
8 // read final value of HPMC's and substract initial in user mode
9 printf("Value of Event (zero'd): %d\n", read_csr_safe(
     hpmcounter3) - start_hpmc3);
```

Siguiendo estos pasos se implanta la solución y se procede a realizar pruebas, así como a analizar sus resultados.

CAPÍTULO 7
Pruebas

Para realizar las pruebas finales de la implantación de la solución se crea un programa sencillo en lenguaje C. Este programa ejecuta una función que consiste en un bucle sencillo: se establece un contador a ”0”, y se le suma ”x” veces un número ”y”. En este caso se determina aleatoriamente que ”x” es igual a 200 e ”y” igual a 1, de forma que sea un bucle ligero para la simulación y únicamente se realicen 200 sumas.

Para poner en uso nuestro contador de bloqueos del ROB se incluye la función de lectura segura que se indica en el capítulo anterior, así como la escritura de la máscara en el registro de control. Antes del bucle se lee el contador, y después se imprime por pantalla el resultado a través de la función *print*, utilizando como parámetro a imprimir la diferencia entre una nueva lectura del contador con la que se guardó previamente en una variable, como se indica en [47].

Además, se pone a prueba también un evento ya existente a través de otro contador, que es el fallo de acceso a la caché de datos. Esto se lleva a cabo de igual forma que la implantación del otro contador, con la única diferencia de que se usa la máscara de bits ”0x202” dado que se encuentra en el tercer conjunto de eventos, en segunda posición.

Una vez escrito el programa se realiza la simulación con el procesador GigaBOOM. Para ello no se utiliza Spike, ya que se ha compilado como programa *baremetal*. Se obtiene de esa simulación los siguientes resultados:

Número de fallos de acceso a la cache (D$)	14
Número de bloqueos del ROB	87
Número de instrucciones retiradas	5757
Número de ciclos totales	4509

Tabla 7.1: Resultados obtenidos de la ejecución de un programa básico con GigaBOOM. Primera comprobación de los contadores *hardware*

.

Seguidamente se modifican los *benchmark* de forma que estos impriman también el número de ciclos que el ROB se encuentra bloqueado. Puesto que se desea analizar el impacto de una modificación en el tamaño del ROB se modifica el número de entradas que la versión GigaBOOM posee (se asume que la versión Giga inicial establece 130 entradas para el ROB), partiendo de 30 entradas hasta 150 (parece ser que el propio entorno limita

el número máximo de entradas a este número), sumando cada vez 30 entradas. Así, se obtiene la siguiente gráfica:

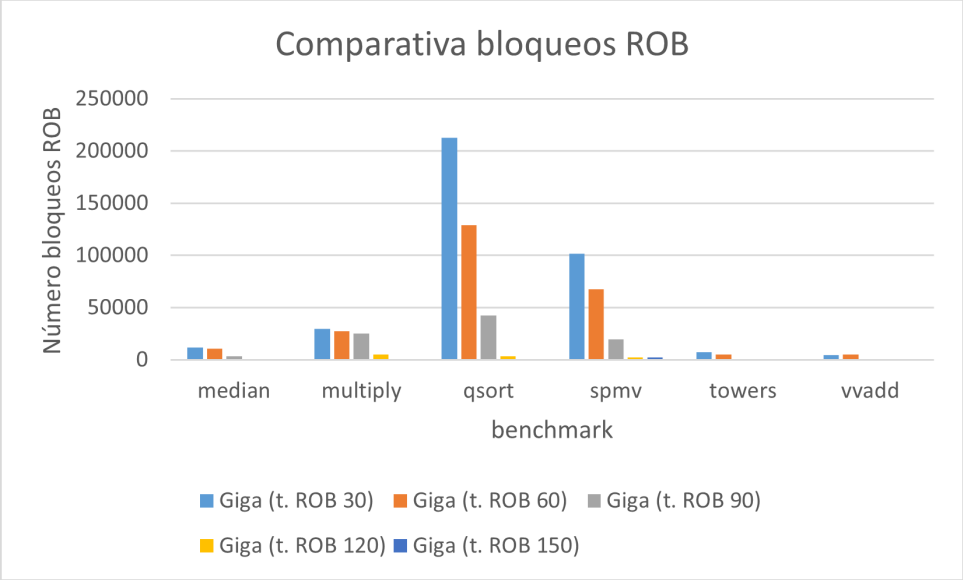

Figura 7.1: Comparativa del número de bloqueos del ROB entre los distintos números de entradas para GigaBOOM.

De esta manera se observa fácilmente cómo el número de bloqueos disminuye conforme el número de entradas del ROB aumenta.

Para conocer además el porcentaje tiempo de ejecución que el ROB permanece bloqueado se puede observar la Figura 7.2.

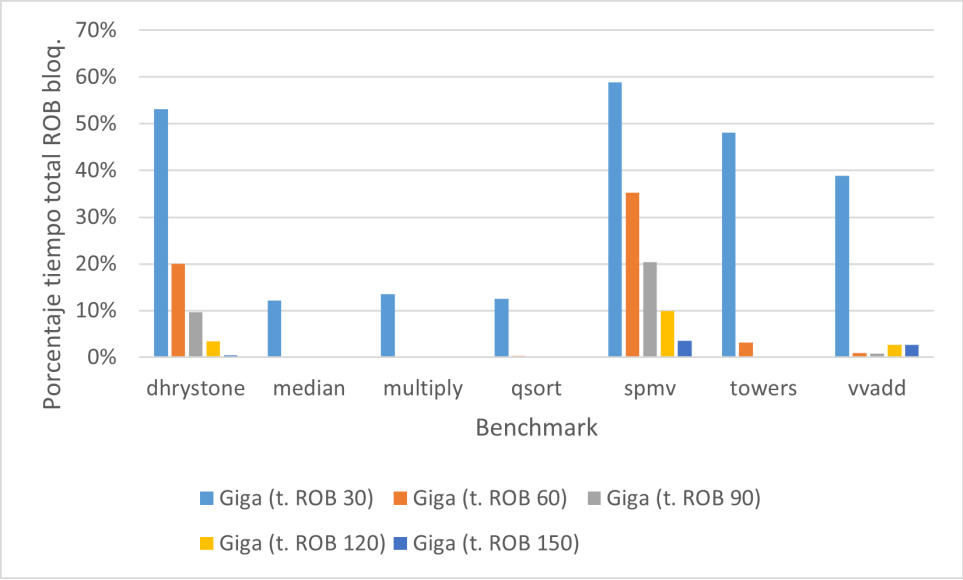

Figura 7.2: Comparativa del porcentaje de tiempo que el ROB pasa bloqueado del tiempo de ejecución total en GigaBoom con diferentes números de entrada.

Se observa así cómo disminuye generalmente el porcentaje de tiempo que pasa bloqueado el ROB conforme aumenta el tamaño del mismo.

Se desea también conocer cuál es el número de bloqueos del ROB que se obtendría para las versiones Large y Mega, por lo que se realizan las simulaciones de igual forma con estos modelos y los resultados se pueden consultar en la Tabla 7.2.

	LargeBoom	MegaBoom	GigaBoom
median	6	6	6
multiply	6	6	2755
qsort	178	561	1717
spmv	1820	1925	1876
towers	6	6	6
vvadd	6	6	6

Tabla 7.2: Número de bloqueos del ROB obtenidos para Large, Mega y Giga BOOM en cada *benchmark*

Se representan los datos en una gráfica (Figura 7.3)para poder observar de una forma más sencilla que las versiones que alcanzaban un rendimiento inferior a la versión Giga consiguen un menor número de bloqueos del ROB. Esto puede deberse a que la relación del número de instrucciones lanzadas y el número de estructuras operacionales disponibles es menor en GigaBOOM que en el resto, pero sigue obteniendo mejores resultados en cuanto al rendimiento dado que tanto el número de registros, colas y operadores sigue siendo mayor, además de que se utiliza un algoritmo de predicción de saltos y una caché de datos mejor.

Para comprobar la validez de los resultados se incluyen también los contadores de ciclos e instrucciones ya que estos no cambian, y se comparan con los ya implementados por el propio Chipyard. Se observa que son prácticamente iguales (a expensas de un pequeño margen de error que depende del lugar donde se empiezan a contar los mismos), confirmando así la validez de nuestro contador.

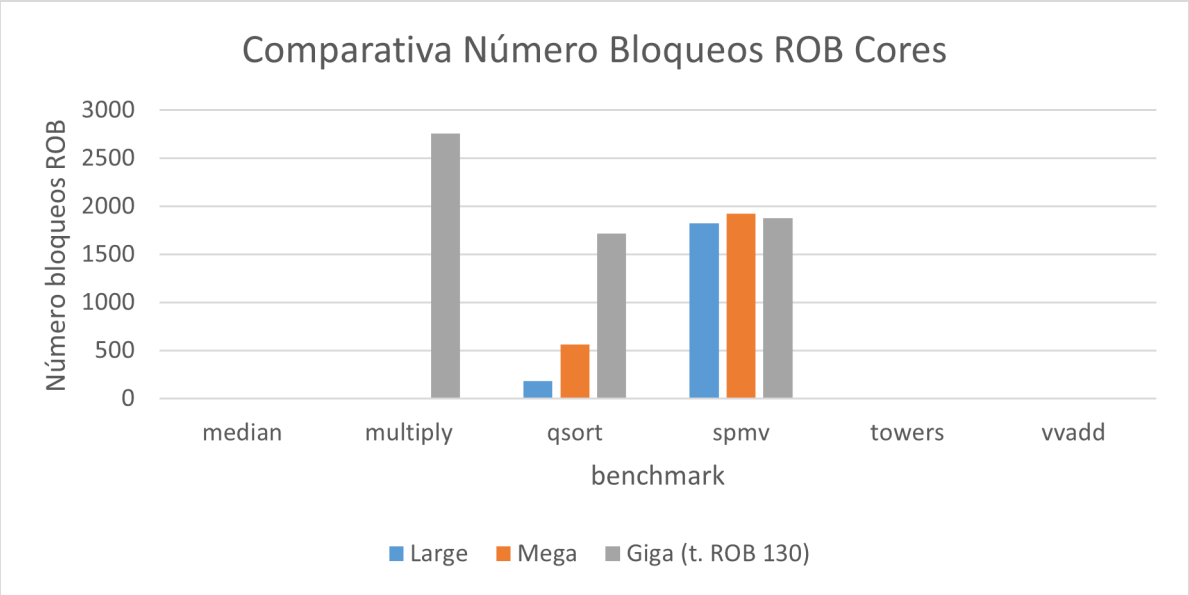

Figura 7.3: Comparativa del número de bloqueos del ROB entre las versiones Large, Mega y Giga de BOOM

CAPÍTULO 8
Conclusiones

En este proyecto se han evaluado y comparado de forma exitosa diferentes procesadores y se ha profundizado en varias versiones de uno de ellos. Además, se ha cumplido el objetivo de implementar un nuevo evento y registrarlo en un contador, para analizar así desde otra perspectiva el rendimiento obtenido en cada procesador bajo diferentes cargas de trabajo. También se ha modificado la versión GigaBOOM obteniendo mejores resultados en el rendimiento de esta forma.

Se ha comprobado la influencia que el tamaño del ROB tiene sobre el rendimiento de un procesador, actuando inversamente su tamaño sobre el número de bloqueos del mismo.

Para poder llevar este trabajo a cabo se ha debido aprender a usar distintas herramientas y tecnologías que no se han visto durante la carrera. Entre las herramientas o tecnologías que no se conocían se encuentran el entorno de Chipyard junto con todas las herramientas que lo componen(simuladores, contadores *hardware*, etc.), así como un lenguaje de diseño *hardware* nuevo cuya curva de aprendizaje y dificultad son consideradas como de las más severas dentro de los lenguajes más utilizados. Para poder llegar a desenvolverse de una manera adecuada con todas estas herramientas se ha tenido que hacer un esfuerzo de trabajo y tiempo añadido, pues ha resultado ser un proceso difícil a causa de las distintas restricciones y metodologías que existen dentro de este *framework* y que no se detallan de forma explícita en la documentación, resultando en numerosos errores desconocidos y mucho tiempo dedicado a la resolución de estos.

También se han encontrado problemas a la hora de usar el conjunto de *benchmarks* SPEC2017, debido a incompatibilidades entre el entorno usado (*WSL*) y las llamadas al sistema que se realizan en estos programas. Se podría haber evitado usando desde un principio el SO Linux de forma nativa, pero esto se desconocía hasta el final del trabajo y esto ha impedido poder solucionarlo por no disponer del tiempo suficiente.

Además de todo lo mencionado anteriormente, la situación actual ha supuesto un punto extra en la dificultad de este proyecto, teniéndose que llevar a cabo de forma remota y utilizando las tecnologías disponibles para la comunicación con los tutores.

A pesar de los errores encontrados, se han solucionado aquellos que eran de máxima importancia y ello ha permitido alcanzar los objetivos propuestos.

Gracias a los conocimientos y el grado de dominio que se han adquirido de todas las herramientas utilizadas se podría volver a realizar el mismo proyecto en muy poco tiempo. Esto permitiría también seguir avanzando para realizar un análisis mucho más profundo de

estos procesadores y herramientas, así como nuevas funcionalidades e implementaciones que permitirían ampliar los conocimientos en este área a través de la investigación.

Se ha logrado desarrollar una mejor capacidad de organización, resolución de problemas y aplicación de los conocimientos adquiridos durante la carrera.

8.1 Relación del trabajo desarrollado con los estudios cursados

Este trabajo está relacionado estrechamente con la rama de Ingeniería de Computadores, además de otras asignaturas relacionadas del resto de los estudios.

Se han utilizado lenguajes que se han estudiado durante la carrera, como lo es el lenguaje C. Se han manejado ficheros de librerías y *scripts*. Se ha puesto en práctica el dominio adquirido en el sistema operativo Linux, utilizando una gran cantidad de comandos y funciones para llevar a cabo de una manera efectiva las tareas del trabajo. Para realizar el análisis de los procesadores se ha puesto en práctica todo lo estudiado sobre estructuras y arquitecturas de computadores.

Se han alcanzado un alto grado en numerosas competencias transversales, destacando aquellas relacionadas con la resolución de problemas, el pensamiento práctico, el aprendizaje permanente y la instrumentación específica.

CAPÍTULO 9
Trabajos futuros

A partir de este trabajo se pueden desarrollar otras funcionalidades para el procesador estudiado y otros que no se han visto. Esto se podría llevar a cabo a través de la implementación de nuevos eventos y contadores que amplíen el nivel de detalle de los análisis que se realizan para medir el rendimiento de los procesadores.

Además, se recomienda seguir este proyecto a través de Linux de forma nativa para evitar incompatibilidades que dificulten el desarrollo del mismo. De esta forma se podría experimentar y medir el rendimiento con las SPEC.

Una ampliación del mismo sería la utilización del simulador FireSim para la posterior sintetización del procesador en un *FPGA* de forma física.

Bibliografía

[1] Delgado, Jesús Delgado. La explosión de un paradigma. *Vivat Academia* 22-28, 2001.

[2] Li, Peijie. Reduce Static Code Size and Improve RISC-V Compression. Diss. Master's thesis. *EECS Department, University of California, Berkeley*, 2019

[3] Zomeño Tortajada, Alejandro. Verificación física de implementaciones del microprocesador RISC-V mediante plataforma embebida basada en FPGA. Diss. 2020.

[4] Segarra Górriz, Izan. Desarrollo de un módulo IP de procesador RISC-V en System Verilog. Diss. 2019.

[5] Bourgeat, Thomas, et al. Mi6: Secure enclaves in a speculative out-of-order processor. Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture. 2019.

[6] Suárez Santamaría, Elena Zaira. RISC-V un ISA de código abierto. TFM. 2019.

[7] A. Amid et al. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. IEEE Micro, vol. 40, no. 4, pp. 10-21, 1 July-Aug. 2020, doi: 10.1109/MM.2020.2996616.

[8] Asanovic, Krste, David A. Patterson, and Christopher Celio. The Berkeley Out-of-Order Machine (BOOM): An industry-competitive, synthesizable, parameterized risc-v processor. University of California at Berkeley Berkeley United States, 2015.

[9] García Aristegui, David, and César Rendueles. Abierto, libre... y público. Los desafíos políticos de la ciencia abierta. Argumentos de Razón Técnica, 17, 45-64. (2014).

[10] Valiente, Waldo, et al. Medición de rendimiento del algoritmo spmv utilizando contadores de hardware para GP GPU en arquitecturas no homogéneas. (2019).

[11] Embree, Paul M., Bruce Kimble, and James F. Bartram. C language algorithms for digital signal processing. (1991): 618-618.

[12] Bachrach, Jonathan, et al. Chisel: constructing hardware in a scala embedded language. DAC Design Automation Conference 2012. IEEE, 2012.

[13] Lerner, Josh, and Jean Tirole. The economics of technology sharing: Open source and beyond. Journal of Economic Perspectives 19.2 (2005): 99-120.

[14] Waterman, Andrew Shell. Design of the RISC-V instruction set architecture. Diss. UC Berkeley, 2016.

[15] McFarlin, Daniel S., Charles Tucker, and Craig Zilles. Discerning the dominant out-of-order performance advantage: Is it speculation or dynamism?. ACM SIGARCH Computer Architecture News 41.1 (2013): 241-252.

[16] Zhao, Jerry, et al. Sonicboom: The 3rd generation berkeley out-of-order machine. Technical report, EECS Department, University of California, Berkeley, 2020.

[17] Deek, Fadi P., y James A. M. McHugh. *Open Source: Technology and Policy*. Cambridge University Press, 2007.

[18] Bokhari, S. N. *The Linux operating system*. Computer 28.8, 1995.

[19] Waterman, Andrew, et al. *The risc-v instruction set manual. volume 1: User-level isa, version 2.0*. CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES, 2014.

[20] Why GNU/Linux. Consultado en http://www.hopelchen.tecnm.mx/principal/sylabus/fpdb/recursos/r129287.PDF

[21] RISC-V Foundation Membership Exceeds 100 Percent Growth Over The Past Year. Consultado en https://riscv.org/announcements/2019/02/risc-v-foundation-membership-exceeds-100-percent-growth-over-the-pa

[22] RISC-V: La necesidad de una arquitectura abierta. Consultado en https://openexpoeurope.com/es/risc-v-la-necesidad-de-una-arquitectura-abierta/

[23] RISC-V Exchange: Cores & SoCs. Consultado en https://riscv.org/exchange/cores-socs/

[24] RISC-V Grows Globally as an Alternative to Arm and Its License Fees. Consultado en https://venturebeat.com/2019/12/11/risc-v-grows-globally-as-an-alternative-to-arm-and-its-license-fees

[25] Ariane RISC-V CPU. Consultado en https://github.com/lowRISC/ariane

[26] Verilog. Consultado en https://es.wikipedia.org/wiki/Verilog

[27] Field-programmable gate array (FPGA). Consultado en https://en.wikipedia.org/wiki/Field-programmable_gate_array

[28] RISC-V BOOM. Consultado en https://boom-core.org/

[29] The Hwacha Project. Consultado en http://hwacha.org

[30] RISC-V. Consultado en https://es.wikipedia.org/wiki/RISC-V

[31] Shilov, A. (2018). *Western Digital Reveals SweRV RISC-V Core, Cache Coherency over Ethernet Initiative*. [Figura]. Recuperado de https://www.anandtech.com/show/13678/western-digital-reveals-swerv-risc-v-core-and-omnixtend-coherency-t

[32] Windows Subsystem for Linux. Consultado en https://es.wikipedia.org/wiki/Windows_Subsystem_for_Linux

[33] SPEC CPU® 2017. Consultado en https://www.spec.org/cpu2017/

[34] Comparing WSL 1 and WSL 2. Consultado en https://docs.microsoft.com/en-us/windows/wsl/compare-versions

[35] Repositorio oficial de Chipyard en Github. Consultado en https://github.com/ucb-bar/chipyard#building-the-tools

[36] Hardware performance counter. Consultado en https://en.wikipedia.org/wiki/Hardware_performance_counter

[37] SiFive U54-MC Core Complex Manual v1p0. Consultado en https://static.dev.sifive.com/U54-MC-RVCoreIP.pdf

[38] Software RTL Simulation - Verilator(Open Source). Consultado en https://chipyard.readthedocs.io/en/latest/Simulation/Software-RTL-Simulation.html#verilator-open-source

[39] Baremetal RISC-V Programs. Consultado en https://chipyard.readthedocs.io/en/latest/Software/Baremetal.html

[40] The RISC-V ISA Simulator (Spike). Consultado en https://chipyard.readthedocs.io/en/latest/Software/Spike.html

[41] C (Lenguaje de programación). Consultado en https://es.wikipedia.org/wiki/C_(lenguaje_de_programaci%C3%B3n)

[42] Scala (lenguaje de programación) - Scala (programming language). Consultado en https://es.qaz.wiki/wiki/Scala_(programming_language)

[43] What exactly is the point of developing RISC-V based CPUs in Chisel rather than Verilog or VHDL? What are the pros and cons?. Consultado en https://www.quora.com/What-exactly-is-the-point-of-developing-RISC-V-based-CPUs-in-Chisel

[44] What benefits does Chisel offer over classic Hardware Description Languages?. Consultado en https://stackoverflow.com/questions/53007782/what-benefits-does-chisel-offer-over-classic-hardware-description-l

[45] How to start learning Scala?. Consultado en https://medium.com/packlinkeng/how-to-start-learning-scala-7ed48eca8fe0

[46] Scala has a really steep learning curve when you encounter ... Consultado en https://news.ycombinator.com/item?id=23424565

[47] Micro-architectural Event Tracking Consultado en https://docs.boom-core.org/en/latest/sections/uarch-counters.html

[48] Linux es líder absoluto en supercomputación, ¿por qué Windows o macOS no?. Consultado en https://www.xataka.com/especiales/linux-es-lider-absoluto-en-supercomputacion-por-que-windows-o-macos

[49] RISC-V Foundation Membership Exceeds 100 Percent Growth Over The Past Year Consultado en https://riscv.org/announcements/2019/02/risc-v-foundation-membership-exceeds-100-percent-growth-over-the-pa

[50] Ariane RISC-V CPU Consultado en https://github.com/lowRISC/ariane

[51] The Berkeley Out-of-Order RISC-V Processor Consultado en https://github.com/riscv-boom/riscv-boom

[52] Scala Jump Start Consultado en https://www.slideshare.net/lifemichael/scala-jump-start