Generación de cores basados en RISC-V y exploración arquitectónica de las memorias caché usando Rocket Chip Generator

RISC-V based core generation and architectural exploration of caches using Rocket Chip Generator

María José Belda Beneyto

GRADO EN INGENIERÍA INFORMÁTICA
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin de Grado en Ingeniería Informática

Septiembre 2020

Directores:

Katzalin Olcoz Herrero
Fernando Castro Rodríguez

Resumen

En este trabajo se explora la arquitectura RISC-V, el core Rocket que la implementa y la herramienta Rocket-chip que nos permite diseñar microarquitecturas y generar un emulador de estas. Estos emuladores simulan la ejecución ciclo a ciclo de un programa en dichas microarquitecturas.

Posteriormente, se crean programas para explorar las propiedades de las cachés de datos de los diseños generados y se ejecutan dichos programas sobre los emuladores para realizar mediciones de rendimiento y sacar los resultados consecuentes.

Palabras clave

RISC-V, Rocket-chip, caché.

Abstract

This work explores the RISC-V architecture, the Rocket core that implements it and the Rocket-chip tool that allows us to design microarchitectures and generate an emulator of these. These emulators simulate the cycle-by-cycle execution of a program in said microarchitectures.

Subsequently, programs are created to explore the properties of the data caches of the generated designs and these programs are run on the emulators to perform performance measurements and obtain the consequent results.

Keywords

RISC-V, Rocket-chip, cache.

Índice general

Índice I

1. Introducción 1

2. Introduction 3

3. Conceptos previos 5
3.1. RISC-V . . . 5
3.1.1. El repertorio de instrucciones RISC-V . . . 5
3.1.2. Cores que implementan RISC-V . . . 6
3.1.3. Aceleradores o co-procesadores para RISC-V . . . 9
3.2. Chisel . . . 9
3.3. Rocket-tools . . . 10
3.3.1. Spike . . . 11
3.3.2. OpenOCD y gdb . . . 11
3.4. Rocket-chip . . . 12
3.4.1. Flujo de generación y testeo genérico de un emulador en Rocket-Chip 15
3.4.2. Ejecución de un test personalizado en el emulador . . . 17
3.4.3. Depuración de un programa ejecutado sobre el emulador . . . 20

4. Contribución 24
4.1. Diseño con caché asociativa de 2 vías y 4 conjuntos . . . 24
4.2. Diseño con caché directa . . . 25
4.3. Tests para la exploración de la caché . . . 26
4.4. Entorno experimental . . . 30

5. Resultados 33
5.1. Pruebas en el emulador con cachés asociativas de 4 conjuntos y 2 vías. . . . 34
5.1.1. Prueba: Resultados de los patrones puros. . . . 34
5.1.2. Prueba: Coherencia entre los patrones fijado un número de accesos a memoria por iteración. . . . 36
5.1.3. Prueba: Impacto del prefetcher de la microarquitectura. . . . 40
5.1.4. Prueba: Variaciones del tamaño del vector en el patrón “recency friendly”. 43
5.1.5. Prueba: Variaciones del tamaño del vector en el patrón “thrashing”. . 45
5.1.6. Prueba: Variaciones del tamaño de los vectores en el patrón “mixed”. 46
5.1.7. Prueba: Variaciones de la probabilidad en el patrón “mixed”. . . . 47
5.1.8. Prueba: Elementos de los vectores en distintos bloques de memoria. . 48

5.2. Pruebas en el emulador con caché directa de datos y caché asociativa de 4 conjuntos y 2 vías de instrucciones. 50
5.2.1. Prueba: Coherencia entre los patrones fijado un número de accesos a memoria por iteración. 50

6. Conclusiones **53**

7. Conclusions **55**

Bibliografía **58**

A. Instalación de las herramientas **59**

Capítulo 1

Introducción

La arquitectura RISC-V empezó a desarrollarse por el año 2010 en la universidad de California en Berkeley de la mano de los profesores David Patterson como director y Krste Asanović, y de dos estudiantes llamados Yunsup Lee y Andrew Waterman[20].

Eligieron que fuese código libre y abierto con la licencia de la universidad Berkeley Software Distribution (BSD) porque el repertorio de instrucciones no es algo novedoso, pues se basa en ideas ya existentes. Así, con esto y la madurez de las FPGAs que permite generar diseños de manera accesible y rápida, logran que todo el mundo tenga acceso a generar un hardware personalizado para el software que va a ejecutarse en él.

Más tarde, se creó la Fundación RISC-V con sede en California para encargarse tanto de la arquitectura RISC-V como de numerosos proyectos que nacen a su alrededor, como herramientas software para generar diseños de microarquitecturas. Y en la actualidad, han cambiado su nombre por RISC-V International y su sede se encuentra en Suiza.

Uno de los proyectos software creados en torno al RISC-V es el Rocket-Chip. Este entorno, también creado y mantenido por RISC-V International junto con ChipsAlliance, permite generar diseños de microarquitecturas que usen el core Rocket, que implementa la arquitectura RISC-V. Y con estos diseños, permite generar un emulador del hardware del diseño, generar código Verilog para volcarlo sobre una FPGA o generar Verilog para usarlo con las herramientas de diseño VLSI[2].

Dado el interés que genera este estándar de repertorio de instrucciones abierto, libre y personalizable, en este trabajo se persigue familiarizarse con la arquitectura RISC-V y con el entorno Rocket-Chip y sus herramientas. Para ello, se estudiará el core Rocket que implementa RISC-V y se generarán diseños de microarquitecturas que lo utilicen mediante el Rocket-Chip.

Posteriormente, se perseguirá explorar la jerarquía de la memoria caché. Para ello, se generarán algunos tests personalizados con patrones que permiten observar el efecto del

reuso en los fallos de caché y se ejecutarán sobre los diseños anteriormente creados en el Rocket-Chip.

A continuación, se recopilarán algunas medidas de rendimiento al ejecutar los tests para analizarlas y sacar resultados y conclusiones de la exploración que habremos realizado sobre la jerarquía de la caché.

En el siguiente capítulo se presentan los conceptos necesarios para entender el funcionamiento tanto del core Rocket y las herramientas que se necesitan para generar tanto los diseños de las microarquitecturas y sus emuladores como los tests que se ejecutarán en ellos.

Posteriormente, se verán en detalle los códigos asociados a cada test así como los objetivos que se persiguen con cada uno de ellos. Y también, la generación detallada de los emuladores sobre los que se ejecutarán los tests.

Para finalizar, se explicarán los resultados obtenidos en las mediciones de las ejecuciones de los tests y se presentarán las conclusiones a razón de esos resultados.

Capítulo 2

Introduction

The RISC-V architecture began to be developed in 2010 at the University of California at Berkeley by professors David Patterson as director and Krste Asanović, and two students named Yunsup Lee and Andrew Waterman[20].

They chose the architecture to be open source licensed by Berkeley Software Distribution (BSD) because the set of instructions is not something new, as it is based on existing ideas. Thus, with this and the maturity of the FPGAs that allow to generate designs in an accessible and fast way, they achieve that everyone has access to generate a customized hardware for the software that is going to run on it.

Later, the California-based RISC-V Foundation was created to take care of both the RISC-V architecture and the many projects that are born around it, such as software tools to generate microarchitecture designs. And today, they have changed their name to RISC-V International and their headquarters are in Switzerland.

One of the software projects created around the RISC-V is the Rocket-Chip. This environment, also created and maintained by RISC-V International and ChipsAlliance, allows the generation of microarchitecture designs that use the Rocket core, which implements the RISC-V architecture. And with these designs, it allows to generate an emulator of the design hardware, generate Verilog code to map it on an FPGA or generate Verilog to use it with the VLSI design tools[2].

Given the interest generated by this open, free and customizable ISA, this work seeks to familiarize yourself with the RISC-V architecture and with the Rocket-Chip environment and its tools. To do this, we will study the Rocket core that implements RISC-V and microarchitecture designs that use it will be generated using the Rocket-Chip Generator.

Later, we will pursue exploring the cache hierarchy. To do this, some customized tests will be generated with patterns that allow observing the effect of reuse in cache misses and will be executed on the designs previously created in the Rocket-Chip.

Next, some performance measures will be collected when executing the tests to analyze them and draw results and conclusions from the exploration that we will have carried out on the cache hierarchy.

The following chapter presents the necessary concepts to understand the functioning of both the Rocket core and the tools that are needed to generate both the designs of the microarchitectures and their emulators as well as the tests that will be executed on them.

Subsequently, the codes associated with each test as well as the objectives pursued with each of them will be seen in detail. And also, the detailed generation of the emulators on which the tests will be executed.

Finally, the results obtained in the measurements of the test executions will be explained and the conclusions will be presented based on these results.

Capítulo 3

Conceptos previos

En este capítulo se explican todos los conceptos sobre las herramientas que se usan que se deben conocer para poder entender y realizar el trabajo. Destacamos el RISC-V como la arquitectura que se va a usar y el entorno software Rocket-chip como herramienta principal para generar emuladores de diseños de microarquitecturas.

3.1. RISC-V

RISC-V es una arquitectura que se basa en un diseño de conjunto de instrucciones reducido (RISC) cuyo código es libre y abierto. Esto ha generado una comunidad a su alrededor que se encarga de crear tanto software para generar diseños de microarquitecturas para procesadores, también de código libre y abierto, como cores que lo implementen.

3.1.1. El repertorio de instrucciones RISC-V

En realidad, no hay un único repertorio de instrucciones RISC-V, sino que son cuatro repertorios de instrucciones básicos similares que se distinguen por el tamaño de los registros de enteros y la cantidad de ellos que hay y sobre estos se pueden añadir extensiones estándar y específicas para generar otros repertorios. Estas extensiones quedan fuera del ámbito del trabajo así que vamos a centrarnos en los repertorios de instrucciones básicos y las opciones que utilicemos.

Tenemos por una parte los repertorios de enteros básicos RV32I y RV64I donde el tamaño de las direcciones de memoria es de 32 bits y 64 bits respectivamente. Por otra parte, el RV32E es una variante del RV32I, con la mitad de registros, pensada especialmente para microcontroladores. Y finalmente, el RV128I que soporta un tamaño de registro de 128 bits [21].

Ahora, comentamos las opciones C y G, pues en este trabajo usamos el repertorio

RV64GC. La opción C denota que existen instrucciones comprimidas más cortas que las normales, por ejemplo con solo dos operandos, para generar códigos en ensamblador más cortos. La innovación de RV64C respecto otros repertorios que tienen también este tipo de instrucciones es que como se desarrollaron al mismo tiempo RV64 y RV32, no se necesitó crear RV64 a partir de RV32, por tanto, tiene suficientes códigos de operación vacíos para permitir que quepan las instrucciones que añade la opción C al repertorio base.[16].

Finalmente, la opción G denota que se combina el repertorio base RV64I con otro repertorio de carácter general, en particular, con el repertorio IMAFDZicsr Zifencei[21].

3.1.2. Cores que implementan RISC-V

En la actualidad, hay una gran variedad de cores que implementan RISC-V creados por multitud de empresas y universidades, como el XuanTie C910 de T-Head (Alibaba) o el Ariane de ETH Zurich, Università di Bologna[15]. Sin embargo, no todos son de código libre y abierto, por eso, nos vamos a restringir al Rocket y al Boom de SiFive. Veamos, a continuación, las características fundamentales de estos dos cores.

Rocket

El core Rocket cuenta con un pipeline de 6 etapas principales que podemos ver en la Figura 3.1. Estas etapas son: generación del PC; Fetch, donde lee la instrucción de la caché de instrucciones; Decode, donde lee qué tipo de instrucción es y qué registros necesita como operandos para determinar los posibles bloqueos; Execution, donde ejecuta la operación de la instrucción; Memory, donde se accede a la caché de datos; y Commit, donde actualiza el estado de la arquitectura. Además, también cuenta con unidades para punto flotante. Se pueden ver todas las etapas en el orden en el que se ejecutan con detalle en la Figura 3.2 que muestra la microarquitectura completa del Rocket.

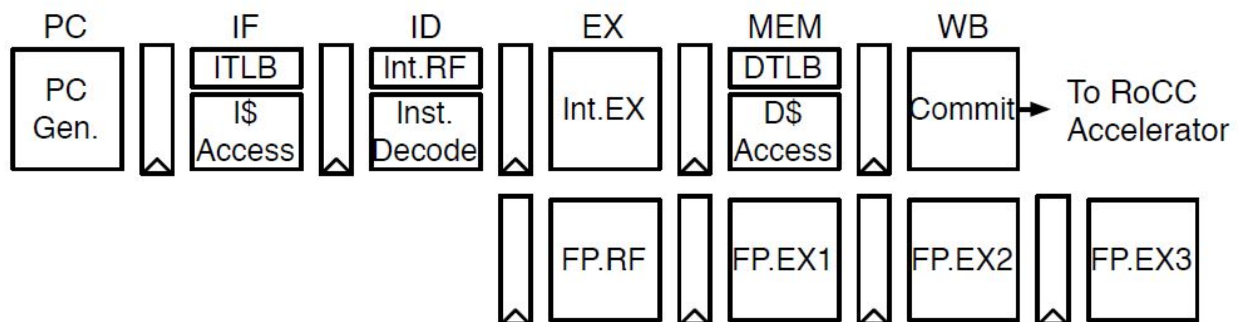


Figura 3.1: *Pipeline del core Rocket*[1].

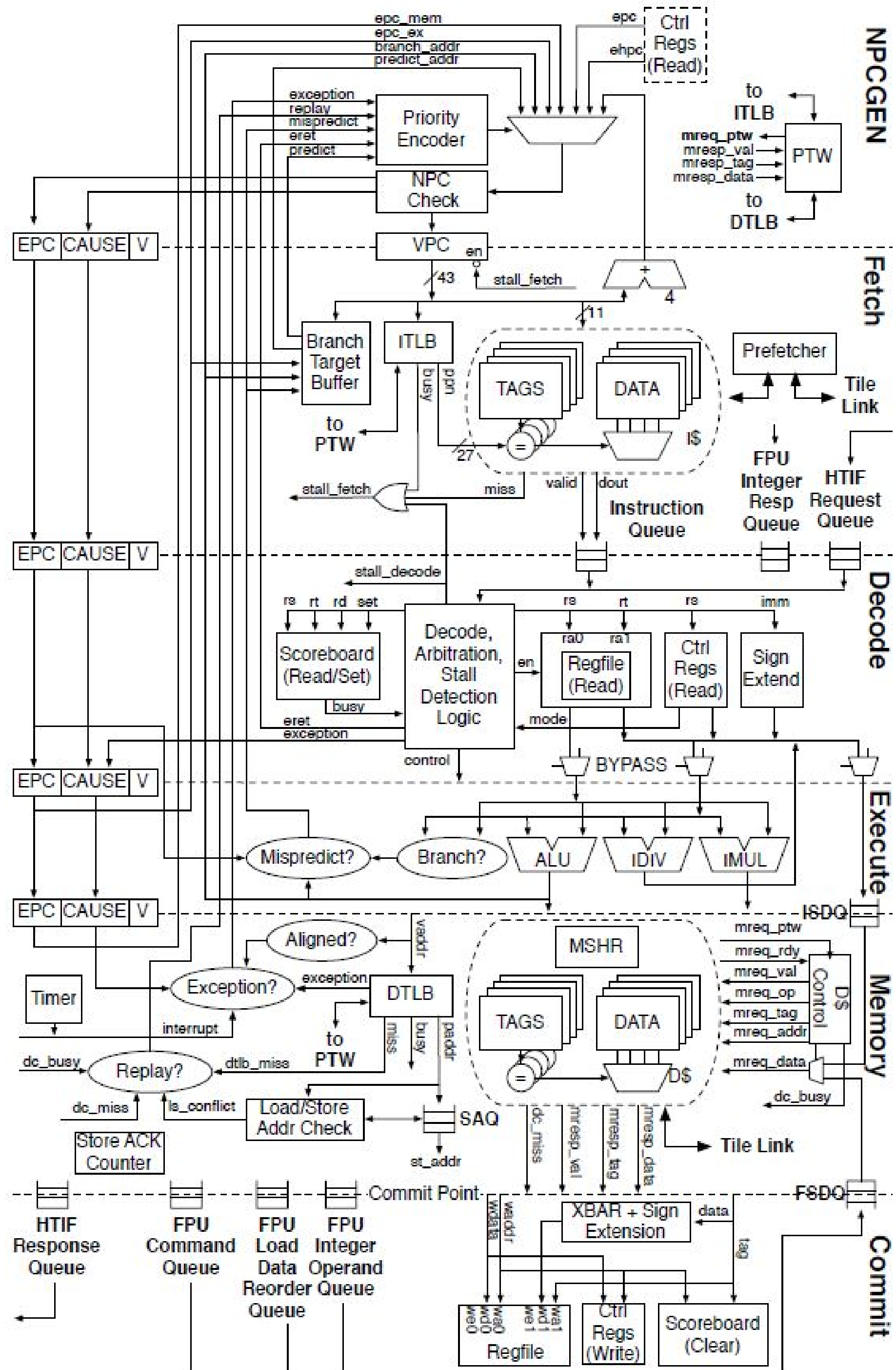


Figura 3.2: *Microarquitectura del core Rocket*[22].

Boom

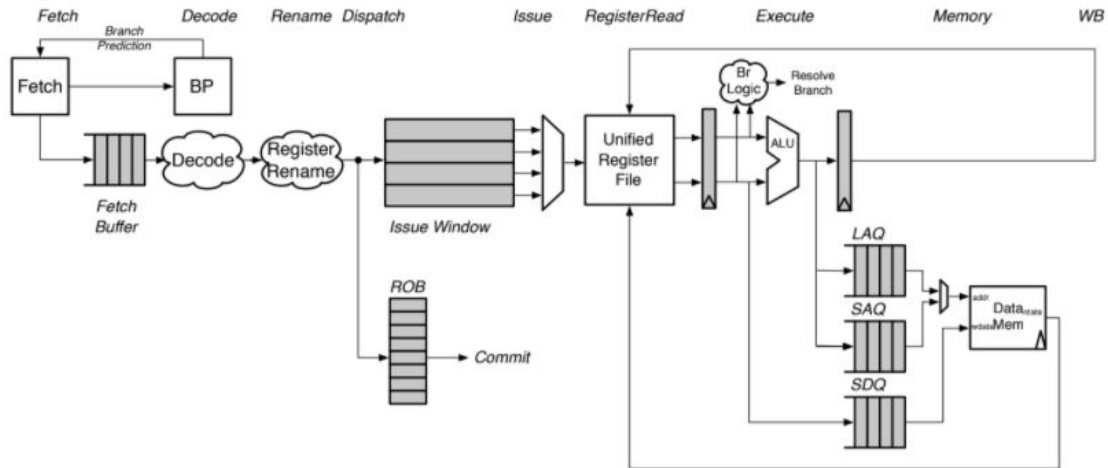
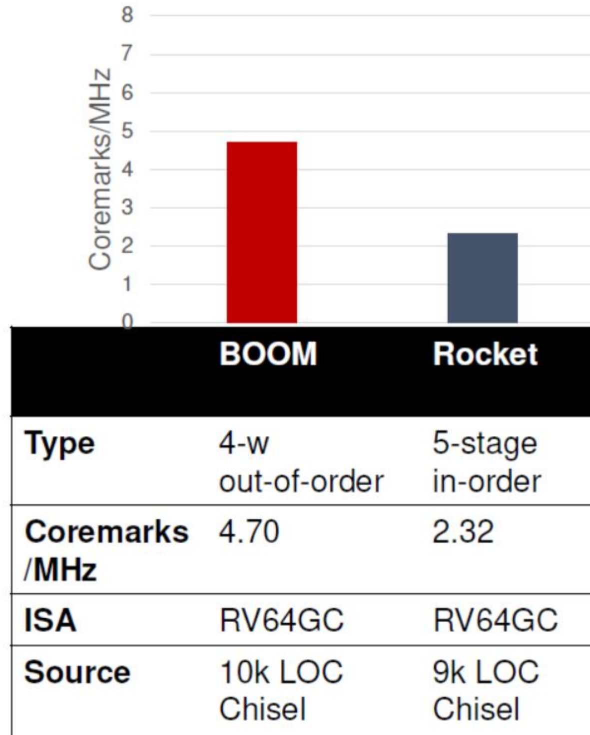


Figura 3.3: *Pipeline del core Boom*[3].

EL core BOOM es similar al Rocket pero ejecuta las instrucciones fuera de orden y tiene un pipeline más complejo con 10 etapas: fetch, decode, rename, dispatch, issue, register-read, execute, memory, wb y commit, que podemos ver en la Figura 3.3. Esto le permite tener un mayor rendimiento respecto al Rocket, como podemos ver en la Figura 3.4.

No obstante, en este trabajo elegiremos el core Rocket para generar diseños de microarquitecturas por ser posible utilizarlo con el entorno Rocket-chip que es sencillo, pues el Boom se tendría que usar con el entorno Chipyard[17]. Este entorno Chipyard es una extensión del Rocket-Chip también de código libre y abierto, con un generador fuertemente parametrizado que hace sencillo modificar los parámetros del diseño de un procesador, donde se pueden usar multitud de cores, entre ellos el Rocket y el Boom, así como aceleradores y co-procesadores. Es por esto, que el Chipyard es un entorno más complejo que el Rocket-Chip y queda fuera del alcance del trabajo.



	BOOM	Rocket
Type	4-w out-of-order	5-stage in-order
Coremarks /MHz	4.70	2.32
ISA	RV64GC	RV64GC
Source	10k LOC Chisel	9k LOC Chisel

Figura 3.4: *Rendimiento de los cores Rocket y Boom*[14].

3.1.3. Aceleradores o co-procesadores para RISC-V

También disponemos de aceleradores o co-procesadores especializados para RISC-V como el Gemini o el Hwacha. El primero de estos es un acelerador especializado en la multiplicación de matrices, inspirado en los recientes aceleradores para aprendizaje automático y enfocado para SoCs de móviles[6]. Sin embargo, el Hwacha es un co-procesador vectorial[12]. Aunque sean de lo más interesante, los aceleradores y co-procesadores quedan fuera del ámbito de este trabajo nuevamente por el entorno que necesitan para integrarlos en el diseño de la microarquitectura.

3.2. Chisel

Chisel (Constructing Hardware In a Scala Embedded Language)[5] es un lenguaje de diseño de hardware para la generación de circuitos y la reutilización del diseño para diseños lógicos digitales ASIC y FPGA basado en el lenguaje de programación Scala. Scala es un lenguaje de alto nivel creado por Martin Odersky, preparado para la programación orientada a objetos que se ejecuta sobre una JVM (Java Virtual Machine)[23].

Chisel fue creado para solucionar los problemas que presentan los otros lenguajes de programación de hardware que se utilizan actualmente, como Verilog o VHDL. Por eso, añade primitivas necesarias para la generación de código para hardware a Scala y se beneficia del alto nivel y la abstracción que le proporciona éste.

De esta forma, partiendo de código Chisel podemos llegar mediante el Rocket-chip a un emulador de una microarquitectura o a código para volcar sobre una FPGA o convertirlo mediante herramientas de diseño VLSI en un ASIC, como podemos ver en la Figura 3.5.

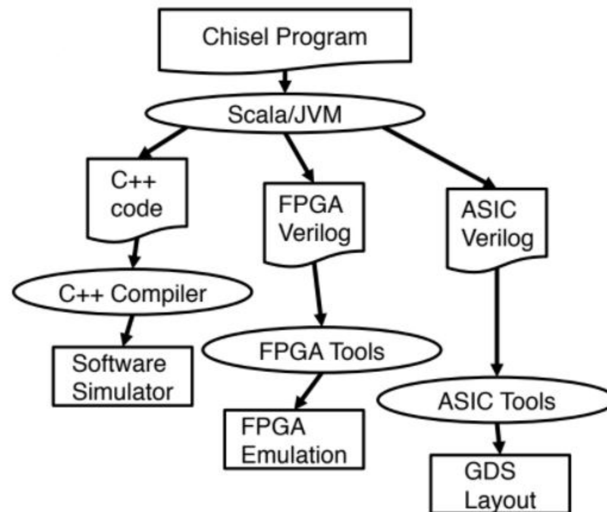


Figura 3.5: *Finalidades del código Chisel*[11].

3.3. Rocket-tools

El repositorio rocket-tools es un repositorio de Github creado por ChipsAlliance para proveer al entorno Rocket-chip de las herramientas software necesarias para generar o depurar los diseños de procesadores. Algunas de las carpetas más relevantes que contiene son las siguientes[4]:

- riscv-isa-sim: Es el repositorio del simulador Spike para RISC-V.
- riscv-gnu-toolchain: Es el repositorio del compilador cruzado de C y C++ para RISC-V.
- riscv-openocd: Es el repositorio de la herramienta OpenOCD adaptada para RISC-V.
- riscv-pk: Es el repositorio de la herramienta ProxyKernel adaptada para RISC-V.
- riscv-tests: Es un repositorio con tests adaptados para RISC-V.

Destacamos que el módulo "riscv-gnu-toolchain" es necesario para la generación de los diseños, pues contiene compiladores cruzados para generar los emuladores en C de los diseños de los procesadores. El módulo "riscv-tests" contiene distintos tests básicos para probar los diseños de los procesadores que generemos. Y el resto de módulos son programas adaptados para RISC-V necesarios para la depuración o simulación que vamos a ver con más detalle.

3.3.1. Spike

Spike es un simulador de software ampliamente usado en el desarrollo de procesadores. Se encarga de proporcionar un sistema base de hardware completo sobre el que ejecutar programas preparados para "bare metal", que son aquellos ejecutables que se ejecutan sin sistema operativo, o bien, se le puede especificar que use un proxy kernel[7] con protocolo de transmisión HTIF/FESVR para poder ejecutar programas que necesiten un kernel para manejar las llamadas. Podemos encontrar una versión adaptada en un repositorio de Github[19] para algunos repertorios RISC-V como el RV32I o el RV64GC que se usa en este trabajo.

3.3.2. OpenOCD y gdb

OpenOCD es un software que permite comunicarse con un dispositivo a través de una interfaz JTAG. Entre otras cosas nos permite la ejecución de programas paso a paso mediante el uso de un depurador como el gdb o la creación de breakpoints. La interfaz JTAG que utiliza nos servirá para comunicar el OpenOCD con el emulador. Además, en el otro extremo, las interfaces de red que interactúan con OpenOCD son: Telnet, GDB y TCL[18] como podemos ver en la Figura 3.6. Notamos que el OpenOCD que usaremos no es el estándar, sino la adaptación a RISC-V que se encuentra en el repositorio Rocket-tools.

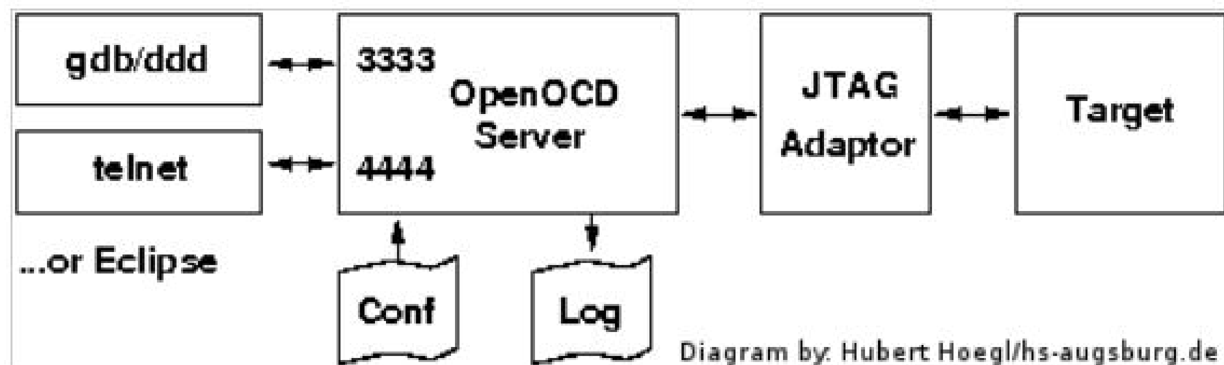


Figura 3.6: *Diagrama de conexiones de OpenOCD.*

En nuestro caso usaremos GDB que está incluido en el módulo "riscv-gnu-toolchain" y que nos permite observar que está ocurriendo en cada momento en el programa que estamos ejecutando, pudiendo ver los valores de las variables en momentos concretos y pausando el

programa si es necesario. Además soporta que los programas se ejecuten en nativo, remoto o simulador[8]. Luego nos sirve especialmente para depurar sobre el emulador.

3.4. Rocket-chip

Rocket-chip es un entorno de código libre y abierto, como la arquitectura RISC-V. Fue creado y continúa siendo mantenido por la universidad de Berkeley junto con ChipsAlliance y RISC-V International. Este entorno está especializado en el core Rocket y destaca por su generador parametrizado, pues tiene interfaces que permiten personalizar fácilmente muchos de los parámetros concretos de cada componente, como puede ser el tamaño de bloque de las cachés L1 de datos e instrucciones.

El Rocket-Chip genera un diseño de una microarquitectura para un procesador basado en "tiles". En cada "tile" queda encapsulado el core con opcionalmente un co-procesador o un acelerador y sus cachés de datos e instrucciones de nivel uno. Es este "tile", la entidad que se relaciona con los buses y el resto de componentes externos, como podemos ver en la Figura 3.7. Y el protocolo que usa para comunicarse entre componentes es el HTIF que ya viene implementado en el Rocket, por tanto, no nos será necesario un proxy kernel al emular.

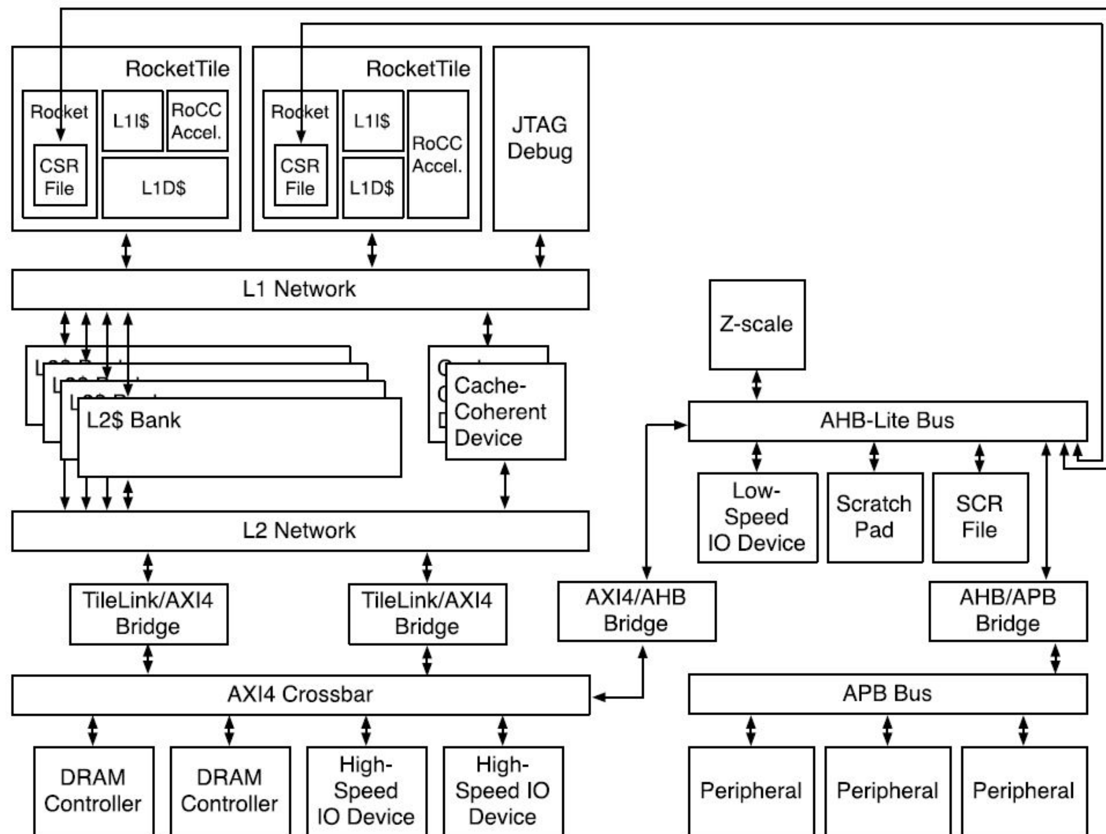


Figura 3.7: *Modelo de SoC generable con Rocket-chip*[13].

Además, Rocket-chip puede generar código Verilog para sintetizarse sobre una FPGA, Verilog para VLSI o código Verilator con precisión de ciclo, o su correspondiente privativo VCS, para simulación de software. En este trabajo vamos a centrarnos en esta última opción, que nos permite generar un emulador del diseño de la microarquitectura que reproduce el efecto que tendría ejecutar un programa instrucción por instrucción y ciclo a ciclo. El flujo para generar el emulador y los tests, lo podemos ver en la Figura 3.8.

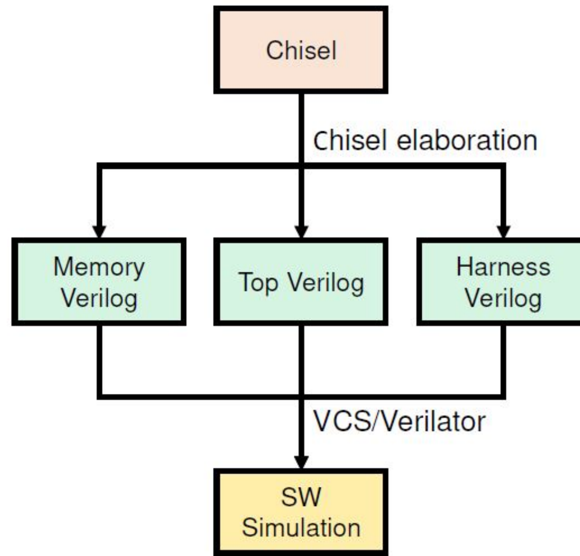


Figura 3.8: *Flujo de generación de Verilator para simulación de software*[14].

Al Rocket-chip se accede descargando su repositorio de Github que contiene todo el código necesario para generar y personalizar, como veremos posteriormente, un diseño de la microarquitectura de un procesador con el core Rocket. El repositorio cuenta con una ordenación por carpetas de los módulos y submódulos de código que se usarán para generar los diseños en la ruta `rocket-chip/src/main/scala`. A continuación se presenta un esquema de las carpetas más relevantes y de lo que contiene cada una de ellas[2].

- Amba: Genera implementaciones de buses con protocolos AMBA.
- Devices: Contiene implementaciones para periféricos.
- Diplomacy: Permite que algunos parámetros tengan valores dinámicos y cambien entre distintos módulos.
- Groundtest: Genera tests que acceden aleatoriamente a memoria para poner a prueba la jerarquía de memoria principal con el máximo uso.
- Jtag: Contiene definiciones necesarias para generar interfaces de bus JTAG.
- Regmapper: Genera dispositivos esclavos para permitir el acceso a sus registros mapeados en memoria.
- Rocket: Genera el core Rocket junto con sus caches de instrucciones y datos de nivel uno.
- Tile: Contiene componentes como aceleradores o FPUs que se pueden usar junto con cores de otros paquetes para generar tiles.

- Tilelink: Usa el paquete diplomacy para generar implementaciones de buses con el protocolo TileLink, necesario para unir distintos tiles.
- System: Contiene los ficheros de configuración generales donde se personalizan los parámetros de los componentes y se generan los diseños.
- Subsystem: Contiene los ficheros de configuración en lenguaje Chisel necesarios para generar los componentes secundarios de un diseño y sus tests generales asociados.
- Unittest: Genera programas de testeo para módulos individuales.
- Util: Contiene fragmentos Chisel y Scala que se reutilizan en los otros paquetes.

Para generar el diseño, además del Rocket-chip se necesitan otras herramientas básicas adaptadas a RISC-V, como compiladores, que se van a tener que obtener de otras fuentes. La mayoría de ellas se encuentran en otro repositorio de Github preparado para funcionar junto con el Rocket-chip llamado rocket-tools, el cual debemos descargar e integrar en el Rocket-chip. Por otra parte, el resto de herramientas que se necesitan, como Java, se descargan e instalan por separado. Para más detalles de la instalación de Rocket-chip y todas las herramientas que se necesitan se debería ver el Apéndice A: Instalación de las herramientas.

3.4.1. Flujo de generación y testeo genérico de un emulador en Rocket-Chip

El funcionamiento básico de Rocket-chip es el siguiente: se configura un diseño de una microarquitectura mediante una clase en lenguaje Scala, se genera el emulador correspondiente y se testea el emulador. Veamos en un ejemplo básico este flujo de generación.

Configuración de un diseño de una microarquitectura

En la ruta `/rocket-chip/src/main/scala` tenemos organizado en carpetas el código de todos los elementos necesarios para una microarquitectura, desde el core hasta las cachés de L2. En nuestro caso, nos centramos en crear una microarquitectura modificando ligeramente los parámetros de los componentes que ya hay programados en estas carpetas. Para ello, vamos directamente a la carpeta “System” que contiene el fichero “Configs.scala” con las definiciones de los diseños. Estos vienen dados por definiciones de clases que llaman a funciones para parametrizar cada componente de la microarquitectura. A continuación, vemos una configuración base para cualquier diseño que viene dada:

```
1 class BaseConfig extends Config(
2     new WithDefaultMemPort() ++
3     new WithDefaultMMIOPort() ++
4     new WithDefaultSlavePort() ++
```

```
5     new WithTimebase(BigInt(1000000)) ++ // 1 MHz
6     new WithDTS("freechips.rocketchip-unknown", Nil) ++
7     new WithNExtTopInterrupts(2) ++
8     new BaseSubsystemConfig()
9 )
```

Aquí podemos ver que esta configuración incluye entre otros los puertos de memoria para comunicar el core con la caché, el tiempo de ciclo de reloj y una llamada a una configuración base de los subsistemas. Sin embargo, no lo tiene todo, le falta el core. Este lo añadimos en la clase de la configuración del diseño, que sería la siguiente:

```
1 class MyNewConfig extends Config(
2     new WithNBigCores(1) ++
3     new BaseConfig
4 )
```

donde hemos añadido un core Rocket grande al diseño final. También se podría elegir un core pequeño o mediano con las parametrizaciones “WithNSmallCores(1)” y “WithNMedCores(1)”. La diferencia entre los cores es la disponibilidad de BTB (Brach Target Buffer) y FPU (Floating Point Unit) y los tamaños de las memorias caché L1 de datos e instrucciones. Además, existe una configuración “With1TinyCore” con un core con las mínimas prestaciones y de 32 bits exclusivamente.

Generación de un emulador para el diseño

Ahora, falta generar un emulador en C de este diseño. Para ello, vamos a la ruta `/rocket-chip/emulator` donde nos encontraremos con un “Makefile” que nos ayudará a generar el emulador. Este “Makefile” está muy parametrizado como el resto de Rocket-chip y por defecto genera el emulador para la configuración con nombre de clase “DefaultConfig”. Para elegir en base a qué configuración queremos que genere el emulador, debemos pasarle por parámetro el nombre de la clase de Scala que queremos usar como podemos ver a continuación:

```
$ make CONFIG=MyNewConfig
```

y esto nos generará un fichero ejecutable del emulador con nombre “emulator-freechips.rocketchip.system-MyNewConfig”.

Además, si queremos depurar los programas que se van a ejecutar en el emulador usando software externos como el openOCD y el gdb, debemos especificarle al comando make que nos genere el emulador apto para depurar de la siguiente forma:

```
$ make CONFIG=MyNewConfig debug
```

Hasta aquí llega la parte de generación del emulador. Todos los archivos derivados de este proceso junto con algunos tests que se generan por defecto para cada emulador aparecen en `/rocket-chip/emulator/generated-src`.

Testeo genérico del emulador

Ahora para comprobar que el emulador funcione correctamente debemos testearlo. Para ello, ejecutamos los tests que se han generado por defecto. Para ejecutarlos basta con el siguiente comando:

```
$ make CONFIG=MyNewConfig -jN run
```

donde N es el número de cores de los que dispone el sistema donde se está ejecutando el emulador. También se pueden ejecutar los tests en ensamblador y los benchmarks por separado como sigue:

```
$ make CONFIG=MyNewConfig -jN run-asm-tests
$ make CONFIG=MyNewConfig -jN run-bmark-tests
```

En el caso del emulador especial para depurar bastaría con añadir "-debug" a los comandos anteriores para que pase los tests. Los comandos serían así:

```
$ make CONFIG=MyNewConfig -jN run-debug
$ make CONFIG=MyNewConfig -jN run-asm-tests-debug
$ make CONFIG=MyNewConfig -jN run-bmark-tests-debug
```

Como podemos ver, siempre hay que especificarle en la variable "CONFIG" el diseño sobre el que queremos ejecutar los comandos. Por tanto, por comodidad se puede exportar el valor de esta variable, con cuidado de cambiarlo cuando cambiemos de diseño.

3.4.2. Ejecución de un test personalizado en el emulador

Compilación de un test personalizado para el emulador

Ahora que tenemos claro como generar un emulador y ejecutar los tests por defecto para comprobar que el emulador funciona correctamente, pasemos a ver como generar y ejecutar tests personalizados siguiendo el flujo que podemos ver en la Figura 3.9.

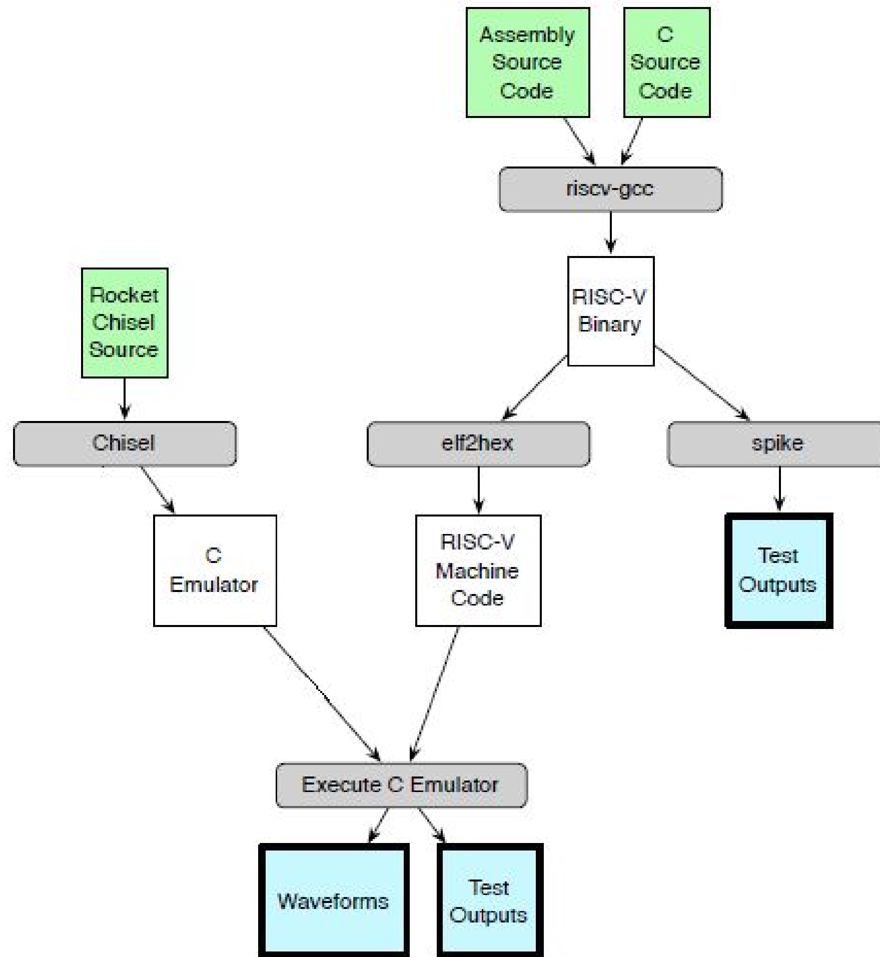


Figura 3.9: *Flujo de generación y ejecución de un test personalizado sobre un simulador y un emulador[10].*

Para empezar, vamos a realizar el procedimiento con el típico programa sencillo de "Hola mundo". Primero, lo escribimos en un fichero .c usual.

```
#include <stdio.h>
int main() {
   printf("Hola mundo!");
   return 0;
}
```

Ahora, para compilar los tests de forma que se puedan ejecutar sobre el emulador del procesador, debemos compilarlos con el gcc especial de RISC-V de 64 bits que es la arquitectura para la que hemos creado las herramientas. Este se encuentra en el repositorio de rocket-tools que ya hemos integrado anteriormente en el Rocket-chip y habremos añadido a la ruta del sistema la carpeta donde se encuentran los ejecutables. De esta forma, para

usarlo bastaría con escribir por consola riscv64-unknown-elf-gcc.

También tendremos que darle las siguientes opciones especiales de compilación:

```
-DPREALLOCATE=1 -mcmodel=medany -static -std=gnu99 -O0 -ffast-math -fno-
    common -fno-builtin-printf
```

Así, no usará las librerías estándar, sino que enlazara estáticamente las adecuadas que se encuentran en el repositorio de herramientas. Además, especificamos "-O0" pensando en los futuros tests sencillos para explotar la caché para que el compilador no optimice el código y pueda modificar los accesos a memoria que deseamos realizar.

Finalmente, el enlazado también necesitará las siguientes opciones especiales:

```
-static -nostdlib -nostartfiles -lm -lgcc -T /rocket-chip/rocket-tools/
    riscv-tests/benchmarks/common/test.ld
```

Por comodidad, todas estas opciones de compilación y enlazado se encuentran agrupadas en el Makefile del directorio `/rocket-chip/rocket-tools/riscv-tests/benchmarks`. Así para compilar un test en C bastaría tener su código asociado con el mismo formato que el de los tests del directorio y añadir su nombre a la variable del fichero "Makefile" que contiene los benchmarks a compilar.

Los tests de este directorio tienen un formato sencillo estando contenido el código de cada uno de ellos en una carpeta con su nombre. Y su código está organizado en al menos un fichero .c llamado "nombreDelTest_main". Por tanto, para nuestro ejemplo de "Hola mundo", creamos en `/rocket-chip/rocket-tools/riscv-tests/benchmarks` la carpeta llamada "holaMundo" y le incluimos el fichero "holaMundo_main.c" que hemos creado anteriormente.

Ahora, editamos el nombre de los benchmarks que va a compilar el fichero "Makefile" que se encuentran en la variable "bmarks" y le incluimos el nombre de nuestra carpeta. Una vez hecho esto, ejecutamos el fichero "Makefile" simplemente introduciendo el comando "make" en la consola. Y esto nos habrá generado en esta misma carpeta un ejecutable "holaMundo.riscv" y un fichero "holaMundo.riscv.dump" con el desensamblado del ejecutable por si queremos comprobarlo con un editor de texto.

Corrección de un test personalizado

Una vez generado el test deberíamos comprobar que funcione correctamente y no de errores en ejecución que no hayamos contemplado. Para realizar la comprobación sin incumbir al emulador por si este no es correcto, podemos usar el simulador Spike.

Para ejecutar el test en el simulador Spike tenemos dos opciones dependiendo de si hemos compilado el test para “bare metal”, como es nuestro caso, o no. En el caso de que no lo hayamos compilado para “bare metal”, sirve con indicarle a Spike que use un proxy kernel para gestionar las llamadas al sistema. Los comandos necesarios para cada caso respectivamente serían los siguientes:

```
$ spike holaMundo
$ spike pk holaMundo.riscv
```

En nuestro caso para saber que el test ha funcionado correctamente, bastaría con comprobar que nos muestra la cadena “Hola mundo!” por consola. En otros casos, conviene incluir una cadena a mostrar al final o ir mostrando valores de variables por consola durante el test para comprobar que su funcionamiento es el esperado.

Ejecución en el emulador de un test personalizado

Finalmente, nos quedaría ejecutar el test personalizado en el emulador. Para ello tenemos que ir a la carpeta del emulador `rocket-chip/emulator` y ejecutar el siguiente comando por consola:

```
$ ./emulator-freechips.rocketchip.system-MyNewConfig $RISCV/riscv-tests/
   benchmarks/holaMundo.riscv
```

donde le indicamos el emulador y el ejecutable a utilizar.

Es habitual que esto nos muestre poca información por consola, en particular, solo muestra que está escuchando en un puerto y finaliza. Si queremos más información debemos añadir el parámetro adicional “+verbose” al comando anterior, de esta forma, nos muestra en cada ciclo información sobre la instrucción que está ejecutando como podemos ver en la Figura 3.10. Debido al volumen de información, convendría volcarlo a un fichero en vez de mostrarlo por consola.

```
C0:       9826 [0] pc=[0000000000000838] W[r 0=000000000000083c] R[r 0=0000000000000000] R[r 0=0000000000000003] inst=[30000067] DASM(30000067)
C0:       9827 [0] pc=[0000000000000838] W[r 0=000000000000083c] R[r 0=0000000000000000] R[r 0=0000000000000003] inst=[30000067] DASM(30000067)
C0:       9828 [1] pc=[0000000000000300] W[r 0=0000000000000304] R[r 0=0000000000000000] R[r24=0000000000000003] inst=[0380006f] DASM(0380006f)
C0:       9829 [0] pc=[0000000000000300] W[r 0=0000000000000304] R[r 0=0000000000000000] R[r24=0000000000000003] inst=[0380006f] DASM(0380006f)
C0:       9830 [0] pc=[0000000000000300] W[r 0=0000000000000304] R[r 0=0000000000000000] R[r24=0000000000000003] inst=[0380006f] DASM(0380006f)
```

Figura 3.10: *Información al ejecutar un test en el emulador con la opción +verbose.*

3.4.3. Depuración de un programa ejecutado sobre el emulador

Llegados aquí ya tenemos claro como generar tests adaptados para el repertorio de instrucciones RISC-V, como crear diseños de microarquitecturas, como generar emuladores para los diseños y como ejecutar los tests personalizados en los emuladores. Pero por si se produce algún fallo convendría poder depurar los programas una vez se están ejecutando en

los emuladores. Para ello disponemos del OpenOCD y el gdb que hemos explicado anteriormente.

Una vez tenemos un emulador capaz de depurar, si queremos que genere un fichero .vcd con las formas de onda resultantes de las señales implicadas en la ejecución de un test, habría que especificarselo con la opción --vcd=<nombre fichero> con el nombre del fichero donde queremos que nos guarde la información. De esta forma, ejecutar el programa "holaMundo.riscv" con esta opción quedaría así:

```
$ ./emulator-freechips.rocketchip.system-MyNewConfig-debug --vcd=
   outputHolaMundo.vcd $RISCV/riscv-tests/benchmarks/holaMundo.riscv
```

Generación de un emulador apto para conectarse con OpenOCD

Para conectar el OpenOCD con el emulador mediante la interfaz JTAG, debemos especificarle al emulador esta opción. Para ello, añadimos un parámetro en la configuración base que vimos anteriormente en la página 16, de forma que quedaría como sigue:

```
1 class MyNewConfig extends Config(
2     new WithJtagDTMSystem ++
3     new WithNBigCores(1) ++
4     new BaseConfig
5 )
```

Conexión del emulador con OpenOCD

Una vez generado el emulador con esta configuración, debemos seguir ciertos pasos para conectarlo con el OpenOCD y empezar el proceso de depuración. Primero, en una primera consola ejecutaremos el emulador pasándole por argumento el ejecutable, la opción para activar el protocolo JTAG y el puerto de conexión, mediante el siguiente comando:

```
./emulator-freechips.rocketchip.system-MyNewConfig +jtag_rbb_enable=1 --
   rbb-port=9823 /rocket-tools/riscv-tests/benchmarks/holaMundo.riscv
```

Una vez hecho esto, se nos mostrará por consola la siguiente información:

```
This emulator compiled with JTAG Remote Bitbang client. To enable, use
   +jtag_rbb_enable=1.
Listening on port 9823
Attempting to accept client socket
```

Ahora es momento de generar un fichero con la configuración necesaria para que el OpenOCD se comunique en el mismo puerto que el emulador. El fichero a generar sería un .cfg, por ejemplo con nombre cemulator.cfg, con el siguiente contenido:

```
interface remote_bitbang
remote_bitbang_host localhost
remote_bitbang_port 9823

set _CHIPNAME riscv
jtag newtap $_CHIPNAME cpu -irlen 5

set _TARGETNAME $_CHIPNAME.cpu
target create $_TARGETNAME riscv -chain-position $_TARGETNAME

gdb_report_data_abort enable

init
halt
```

A continuación, ya podemos ejecutar el OpenOCD con la anterior configuración mediante el comando:

```
openocd -f /path/to/config.cfg
```

Ahora, si todo se ha conectado correctamente, la primera consola nos mostrará un mensaje confirmando que ha aceptado la conexión. Y por esta segunda consola se nos mostrará entre otra información, los posibles puertos de conexión:

```
Info : Listening on port 3333 for gdb connections
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
```

A continuación, como queremos depurar con el gdb, tenemos que abrir una consola para conectar el gdb con el OpenOCD. Sin embargo, esto no es directo, habrá que seguir unos pasos. Primero, iniciamos el gdb con el fichero que queremos ejecutar sobre el emulador y depurar, con el siguiente comando:

```
$ riscv64-unknown-elf-gdb holaMundo.riscv
```

Con esto, se nos inicia el gdb y se queda esperando órdenes. Ahora bien, como el emulador que estamos usando es más lento que Spike y otros simuladores, el gdb encontrará numerosos excesos de tiempo entre mandarle un comando al emulador y que este le conteste, por eso, conviene subirle el tiempo de espera lanzando en gdb el siguiente comando:

```
$ set remotetimeout 2000
```

Una vez hecho esto, solo queda el paso final que es conectar el gdb con el OpenOCD. Para ello basta con ejecutar el siguiente comando:

```
$ target remote localhost:3333
```

Finalmente, ya podemos lanzarle el comando “load” al gdb para que cargue las secciones del programa que queremos depurar e ir usando comandos para avanzar en la ejecución del programa.

Así, concluimos el capítulo habiendo visto como generar emuladores y ejecutables para RISC-V. Además de como simular los ejecutables y finalmente depurarlos en el emulador. En el siguiente capítulo veremos los emuladores y los tests personalizados que vamos a generar usando los conceptos de este capítulo.

Capítulo 4

Contribución

En este capítulo vamos a ver la generación de los diseños que hemos creado y los tests personalizados que se ejecutan en ellos para analizar posteriormente sus tiempos de ejecución.

Respecto al tamaño de las cachés que vamos a generar en los diseños que siguen, cabe destacar que es siempre 256B. Este tamaño está elegido con cuidado, pues Rocket-chip no permite generar diseños con caches excesivamente pequeñas que no sean útiles en el mundo real.

4.1. Diseño con caché asociativa de 2 vías y 4 conjuntos

Empecemos con la generación de un diseño de una arquitectura básica de RISC-V con un core Rocket pequeño y dos cachés L1 asociativas de 2 vías y 4 conjuntos con un tamaño de bloque de 32B y un tamaño total de 256B, una de datos y otra de instrucciones. Para ello, vamos al fichero de configuración de la microarquitectura en `/rocket-chip/src/main/scala/system/Configs.scala` y añadimos la siguiente clase:

```
class MyConfig2Ways4Conj32BBloque extends Config(
  new WithL1ICacheSets(4) ++
  new WithL1DCacheSets(4) ++
  new WithL1ICacheWays(2) ++
  new WithL1DCacheWays(2) ++
  new WithCacheBlockBytes(32) ++
  new WithNSmallCores(1) ++
  new BaseConfig )
```

Aquí podemos identificar como la llamada a BaseConfig genera una configuración por defecto básica para el Rocket. Posteriormente, se hacen las llamadas para modificar el tamaño y la cantidad de los cores, de forma que el diseño tenga tan solo un core Rocket pequeño. Y finalmente, le modificamos los parámetros de la caché L1 para tener una de un tamaño y

organización adecuada para los experimentos.

Ahora generaríamos el emulador de C para el diseño de la microarquitectura. Este se genera en la ruta `/rocket-chip/emulator` usando el siguiente comando:

```
$ make CONFIG=MyConfig2Ways4Conj32BBloque
```

A continuación, ejecutamos los tests por defecto que se han generado junto con el emulador. Estos tests se encuentran en `/rocket-chip/emulator/generated-src`, pero para ejecutarlos basta con estar en el directorio `/rocket-chip/emulator` y ejecutar el siguiente comando:

```
$ make CONFIG=MyConfig2Ways4Conj32BBloque -jN run
```

Así, si se pasan todos los tests habremos comprobado que el diseño es correcto.

4.2. Diseño con caché directa

Ahora generamos un diseño de una arquitectura básica de RISC-V con un core Rocket pequeño, una caché de instrucciones asociativa de dos vías y cuatro conjuntos y una caché de datos directa, ambas con un tamaño de bloque de 32B y un tamaño total de 256B. Para ello, vamos al fichero de configuración del sistema en `/rocket-chip/src/main/scala/system/Configs.scala` y añadimos la siguiente clase:

```
1 class MyConfigCacheDatosDirecta extends Config(
2   new WithL1ICacheSets(4) ++
3   new WithL1DCacheSets(8) ++
4   new WithL1ICacheWays(2) ++
5   new WithL1DCacheWays(1) ++
6   new WithCacheBlockBytes(32) ++
7   new WithNSmallCores(1) ++
8   new BaseConfig )
```

Aquí podemos ver como lo único que cambia respecto a la configuración anterior son los valores que les damos a los procedimientos que parametrizan el número de conjuntos y de vías de las cachés L1 de datos e instrucciones. Por tanto, nos damos cuenta de que es muy sencillo modificar un diseño para generar otro con propiedades muy diferentes.

Ahora generaríamos el emulador de C para el diseño de la microarquitectura. Este se genera en la ruta `/rocket-chip/emulator` usando el siguiente comando:

```
$ make CONFIG=MyConfigCacheDirecta32BBloque
```

Y a continuación, ejecutamos los tests por defecto que se han generado junto con el emulador con el comando:

```
$ make CONFIG=MyConfigCacheDirecta32BBloque -jN run
```

Así, comprobamos la corrección del diseño.

4.3. Tests para la exploración de la caché

A continuación, seguimos con la generación de tests que pondrán a prueba los parámetros de la memoria caché L1 de nuestro diseño de la microarquitectura. Para ello nos vamos a basar en un artículo de Aamer Jaleel y Co.[9] donde se distinguen los siguientes cuatro tipos de patrones para explorar los fallos y aciertos de la caché.

1. Recency friendly.
   En este patrón se recorre un vector en cierto sentido y a continuación, al revés, de esta forma se espera que al hacer el segundo recorrido se encuentren en caché los últimos elementos del vector, que se reutilizan. El esquema del patrón sería el siguiente: $(a_1, \ldots, a_k, a_k, \ldots, a_1)^N$. Además, el tamaño del vector k es independiente del tamaño de la caché. De esta forma, si k es menor que el número de elementos del vector que caben en la caché, en el recorrido inverso del vector $a_k, \ldots, a_1$ tendremos todo aciertos de caché. Vemos que se repite N veces, de forma que entre una iteración y otra dividiremos dos casos posibles:

   - Si no se limpia la caché, los primeros elementos del vector ya estarán en la caché porque han sido los últimos accesos. Así, al empezar la siguiente iteración, habrá aciertos de caché. El código correspondiente que se ha generado para reproducir su comportamiento es el siguiente.

```
1        for (Run_Index = 1; Run_Index <= Number_Of_Runs; ++
    Run_Index)
2        {
3            /*Accedo a posiciones consecutivas del vector
4            desde 0 hasta k-1*/
5            long long int a = 0;
6            int i = 0;
7            while (i < k){
8              a = b[i];
9              i = i + 1;
10           }
11           i = i - 1;
12           /*Accedo a posiciones consecutivas del vector
13           desde k-1 hasta 0*/
14           while(i >= 0){
15             a = b[i];
16             i = i - 1;
17           }
18       }
19
```

- Si se limpia la caché entre iteraciones, al empezar cualquiera no habrá ningún elemento del vector en la caché y se darán fallos para todos los elementos la primera vez que se recorre el vector. El código correspondiente que se ha generado para reproducir su comportamiento es el siguiente.

```
        for (Run_Index = 1; Run_Index <= Number_Of_Runs; ++
    Run_Index)
        {
            /*Accedo a posiciones consecutivas del vector
            desde 0 hasta k-1*/
            long long int a = 0;
            int i = 0;
            while (i < k){
              a = b[i];
              i = i + 1;
            }
            i = i - 1;
            /*Accedo a posiciones consecutivas del vector
            desde k-1 hasta 0*/
            while(i >= 0){
              a = b[i];
              i = i - 1;
            }

            //Limpia la cache: cargando otro vector
            int numlonglongintEnBloque =tamBloque/tamLongLongInt;
            int j = 0;
            int k = tamCache/tamLongLongInt;
            while(j < k){
              long long int x = vectorLimpiar[j];
              j = j + numlonglongintEnBloque;
            }
        }

```

2. Thrashing.
   En este patrón simplemente se recorre un vector de inicio a fin y se repite el proceso N veces. El esquema del patrón sería el siguiente: $(a_1, \ldots, a_k)^N$. Al igual que en el caso anterior, dependiendo de si se limpia la caché entre iteraciones tendremos más fallos de caché o menos. Además, se pide que el tamaño del vector k sea mayor que el número de elementos que caben en la caché, para que haya fallos de caché entre iteraciones. Sino no sería un caso interesante, ya que solo la primera iteración daría fallos de caché, luego serían todo aciertos. El código correspondiente que se ha generado para reproducir su comportamiento es el siguiente.

```
    for (Run_Index = 1; Run_Index <= Number_Of_Runs; ++Run_Index)
    {
```

```
        /*Accedo a posiciones consecutivas del vector
        desde 0 hasta k-1*/
        long long int a = 0;
        int i = 0;
        while (i < k){
          a = b[i];
          i = i + 1;
        }
    }

```

3. Streaming.
   En este patrón se persigue que no haya nada de reuso de bloques de caché. Por tanto, se recorre un vector eligiendo elementos de distintos bloques. El esquema del patrón sería el siguiente: $(a_1, a_{j+1}, \ldots, a_k)^N$, con j lo suficientemente grande para que cada elemento esté en un bloque distinto. El código correspondiente que se ha generado para reproducir su comportamiento es el siguiente.

```
    for (Run_Index = 1; Run_Index <= Number_Of_Runs; ++Run_Index)
    {
        /*Accedo a posiciones no consecutivas del vector
        desde 0 hasta k-1, de forma, que solo se coja
        un dato de cada bloque*/
        long long int a = 0;
        int i = 0;
        while (i < k){
          a = b[i];
          i = i + datosPorBloque;
        }
    }

```

4. Mixed.
   Este patrón como bien indica su nombre es una combinación de lo que hemos visto en los anteriores. El esquema sería el siguiente: $((a_1, \ldots, a_k, a_k, \ldots, a_1)^A P_\varepsilon[b_1, \ldots, b_n])^N$. Primero se recorrería un vector en ambos sentidos, como en el patrón "recency friendly", hasta cumplir A iteraciones. A continuación, dependiendo de cierta probabilidad ε se decide si se recorre de forma simple otro vector. Además, se pide que el tamaño del primer vector k sea menor que la cantidad de elementos que caben en la caché y el tamaño del segundo n sea mayor que la cantidad de elementos que caben en la caché. Así este patrón simula más o menos la versión del patrón "recency friendly" con limpieza de la caché entre iteraciones si tomamos unos valores de k ajustados para ello en ambos patrones, pues en los casos que se recorra el segundo vector lo que estaremos haciendo es eliminar el primer vector de la caché. El código correspondiente que se ha generado para reproducir su comportamiento es el siguiente.

```
    for (Run_Index = 1; Run_Index <= Number_Of_Runs; ++Run_Index)
    {
        /*Accedo a posiciones consecutivas del vector
        desde 0 hasta k-1*/
        long long int a = 0;
        int i = 0;
        while (i < k){
          a = b[i];
          i = i + 1;
        }
        i = i - 1;
        /*Accedo a posiciones consecutivas del vector
        desde k-1 hasta 0*/
        while (i >= 0){
          a = b[i];
          i = i - 1;
        }

        /*La primera de cada 10 veces cargamos el
        otro vector en cache*/
        if (Run_Index % 10 == 0) {
            int j = 0;
            while (j < n){
                a = otro[j];
                j = j+ 1;
            }
        }
    }

```

Como se puede ver, la parte de la probabilidad ε del patrón se ha cambiado por exactamente un décimo, así sabemos con seguridad cuantas veces se ha limpiado el primer vector de la caché y se pueden ajustar mejor los tiempos para comparar los patrones entre ellos.

Notamos que la modalidad del patrón "recency friendly" con limpieza de la caché entre iteraciones es un caso particular de este patrón con probabilidad $\varepsilon = 1$. Pero lo vemos aparte porque la probabilidad ε está pensada para que sea un valor pequeño próximo a cero.

Hemos podido ver que los tests están hechos con vectores de datos del tipo long long int. Esto es porque en el RISC-V de 64 bits que estamos usando, los enteros ocupan tan solo 4B, mientras que los long long int ocupan 8B. Así, llenamos la caché con menos elementos y necesitamos menos número de elementos en cada vector.

4.4. Entorno experimental

Una vez tenemos el código de los tests y los emuladores listos, nos queda ejecutarlos y medir algunos parámetros relevantes para poder compararlos entre ellos y extraer algunos resultados y conclusiones de los experimentos.

En nuestro caso, cada test ha sido preparado para medir los ciclos de reloj y las instrucciones retiradas durante la ejecución del código que hemos presentado para cada uno. De forma que, todo el código relativo a las inicializaciones de los vectores y las variables necesarias queda fuera de las mediciones. Podemos ver un esquema de la organización del código a continuación.

```
1  int main (int argc, char** argv)
2  {
3    /* Declaracion de registros para las variables */
4    REG   int             Run_Index;
5    REG   int             Number_Of_Runs;
6    REG   int             tamLongLongInt;
7    REG   int             tamBloque;
8    REG   int             numVias;
9    REG   int             numConjuntos;
10   REG   int             tamCache;
11
12   /* Inicializacion de las variables */
13   Number_Of_Runs = 100;
14   tamLongLongInt = 8;
15   tamBloque = 32;
16   numVias = 2;
17   numConjuntos = 4;
18   tamCache = tamBloque*numVias*numConjuntos;
19
20   const int numlonglongintEnBloque = tamBloque/tamLongLongInt;
21   const int k = 50; //Tamano del vector
22
23   /* Declaracion e inicializacion del vector */
24   int cont = 0;
25   long long int numAux = 1;
26   long long int b[k];
27   while(cont < k){
28     b[cont] = numAux;
29     numAux = numAux + 1;
30     cont = cont + 1;
31   }
32
33   /* Empieza a medir el tiempo */
34   setStats(1);
35
36   //Codigo a ejecutar dependiendo del patron
37   .
```

```
38   .
39   .
40
41   /* Finaliza la medicion del tiempo */
42   setStats(0);
43
44 }
```

Para medir estos parámetros, hemos usado los registros CSR que contienen información sobre el sistema en tiempo real. En particular, hay treinta y dos especializados para medición: el mcycle, que cuenta el número de ciclos; el minstret, que cuenta el número de instrucciones retiradas; y los treinta restantes con nombre del tipo "mhpmcounter3" son parametrizables para que el usuario elija las ocurrencias de qué evento quiere medir. Para esto debe guardar en el registro con nombre del tipo "mhpmevent3" el nombre del evento que se quiere medir[16]. En nuestro caso, nos ha bastado con utilizar los registros mcycle y minstret.

Para acceder al valor de estos registros, existen las siguientes funciones especializadas que podemos encontrar en `/rocket-tools/riscv-opcodes/encoding.h`.

```
1    #define read_csr(reg) ({ unsigned long __tmp; \
2      asm volatile ("csrr %0, " #reg : "=r"(__tmp)); \
3      __tmp; })
4
5    #define write_csr(reg, val) ({ \
6      asm volatile ("csrw " #reg ", %0" :: "rK"(val)); })
7
8    #define swap_csr(reg, val) ({ unsigned long __tmp; \
9      asm volatile ("csrrw %0, " #reg ", %1" : "=r"(__tmp) : "rK"(val)); \
10     __tmp; })
11
12   #define set_csr(reg, bit) ({ unsigned long __tmp; \
13     asm volatile ("csrrs %0, " #reg ", %1" : "=r"(__tmp) : "rK"(bit)); \
14     __tmp; })
15
16   #define clear_csr(reg, bit) ({ unsigned long __tmp; \
17     asm volatile ("csrrc %0, " #reg ", %1" : "=r"(__tmp) : "rK"(bit)); \
18     __tmp; })
19
20   #define rdtime() read_csr(time)
21   #define rdcycle() read_csr(cycle)
22   #define rdinstret() read_csr(instret)
```

Finalmente, para realizar los cálculos del número de ciclos que se han usado y el número de instrucciones que se han retirado durante la ejecución de la parte de código que queremos, hemos usado la función "setStats" que viene como declaración externa en el fichero "util.h" que se encuentra en la siguiente ruta `/rocket-tools/riscv-tests/benchmarks/common` y cuyo comportamiento es activar y desactivar la función que presentamos a continuación.

```
#define stats(code, iter) do { \
unsigned long _c = -read_csr(mcycle), _i = -read_csr(minstret); \
code; \
_c += read_csr(mcycle), _i += read_csr(minstret); \
if (cid == 0) \
  printf("\n%s: %ld cycles, %ld.%ld cycles/iter, %ld.%ld CPI\n", \
         stringify(code), _c, _c/iter, 10*_c/iter%10, _c/_i, 10*_c/_i
%10); \
} while(0)
```

Esta función se encuentra en el mismo fichero que la anterior y vemos que su comportamiento es medir la diferencia de los valores en los registros mcycle y minstret.

Ahora que hemos visto como generar los emuladores y los programas de prueba de la cache, pasamos al capítulo siguiente donde se muestran los resultados de dichas pruebas.

Capítulo 5

Resultados

En este capítulo vamos a presentar los resultados que se han obtenido al medir los tiempos de ejecución de los diferentes tests con distintas parametrizaciones de las iteraciones y tamaño de los vectores. Notamos que las mediciones son deterministas, pues tras realizar varias ejecuciones de los mismos tests igual parametrizados, los valores medidos eran exactamente los mismos en todas las ejecuciones. Por tanto, se presenta solo un resultado por cada parametrización de los tests.

Primeramente, para poder comparar los tiempos de los distintos patrones debemos homogeneizarlos. Para ello, vamos a fijar un número de accesos a memoria por iteración y ajustaremos los valores del tamaño de los vectores para cada patrón de forma que todos se adecuen al número de accesos a memoria.

Estudiemos el número de accesos a memoria que hace cada patrón en una iteración para conseguir una fórmula general que nos de los valores de los tamaños de los vectores en función de los accesos a memoria que fijemos.

1. Recency friendly: $(a_1, \ldots, a_k, a_k, \ldots, a_1)$
   En cada iteración recorre el vector hacia delante y hacia atrás, luego accede $2k$ veces a memoria.

2. Thrashing: $(a_1, \ldots, a_k)$
   En cada iteración recorre el vector solo hacia delante, luego accede k veces a memoria.

3. Streaming: $(a_1, a_{j+1}, a_{2j+1}, \ldots, a_k)$
   En cada iteración recorre el vector hacia delante saltando los elementos de j en j. Este j se elige en función del tamaño de bloque de la caché que en nuestro caso es 32B. Por tanto, en cada bloque caben 4 elementos del tipo long long int y debemos fijar $j = 4$. Así, accede $k/4$ veces a memoria.

4. Mixed: $((a_1, \ldots, a_k, a_k, \ldots, a_1)^A P_\varepsilon[b_1, \ldots, b_m])$
   Por simplicidad y similitud a los otros patrones para poder compararlos adecuadamente vamos a fijar $A = 1$ para todos los casos. Hecho esto, el patrón recorre el vector

hacia delante y hacia atrás en todas las iteraciones, lo cual, son $2k$ accesos a memoria. Además, como fijamos $\varepsilon = 1/10$, en una décima parte de las iteraciones se hacen m accesos más. Por tanto, siendo N el número total de iteraciones del patrón, se accede $N(m/10 + 2k)$ veces a memoria en total.

En el caso del "recency friendly" que limpia, además de los $2k$ accesos al vector, le deberíamos sumar los accesos que se realizan simplemente para limpiar la caché. En vez de esto, vamos a generar un test con solo la parte de limpiar la caché, para restarle estas mediciones a dicho "recency friendly". Notamos que la función de limpiar la caché no depende del tamaño k del vector, pues simplemente depende del tamaño de la caché, que como hemos dicho siempre será el mismo. Por tanto, nos bastará con realizar las mediciones del test que limpia la caché solo una vez.

Ahora, fijemos un número de accesos a memoria y realicemos pruebas para los distintos patrones con distintos números de iteraciones. En estas pruebas se van a medir las instrucciones retiradas y los ciclos de reloj que se usan para ejecutar solamente la parte central de los tests que vimos en el Capítulo 4, el resto de preparación no se ha incluido en la medición.

Los resultados de ciclos de reloj e instrucciones obtenidos junto con los parámetros que se han elegido para los patrones se presentan a continuación. Estos datos se van a presentar en tablas con las filas "Ciclos" e "Instrucciones retiradas" donde se tienen los valores numéricos sacados de las ejecuciones de los patrones que se representan en las columnas. Además, tenemos la fila "Normalizado" que tiene los valores de los ciclos normalizados con base uno de los patrones, habitualmente, el patrón "recency friendly" que no usa la función de limpiar. Finalmente, la fila "Ciclos por accesos a memoria" expresa de media cuantos ciclos se necesitan por cada acceso a memoria.

5.1. Pruebas en el emulador con cachés asociativas de 4 conjuntos y 2 vías.

5.1.1. Prueba: Resultados de los patrones puros.

En esta primera prueba, vamos a ejecutar una sola iteración de cada patrón con 50 accesos a memoria, para ver cómo se comportan en su estado puro. La prueba se va a realizar con todos los patrones excepto el "mixed", pues en su caso, necesitamos que haya un mínimo de ejecuciones para que tenga sentido la parte de la probabilidad.

	Recency friendly (a1...ak,ak...a1)	Streaming a1, a2, a3, ...	Thrashing (a1...ak) k > cache
Ciclos	1983	2764	1989
Instr retiradas	750	843	793
Ciclos normalizados	1	1,393847705	1,003025719
Ciclos por accesos a memoria	39,66	55,28	39,78
k	25	200	50

Figura 5.1: *Resultados de los patrones con una iteración.*

Podemos ver los resultados obtenidos en la Figura 5.1. Con este bajo número de iteraciones es muy importante que el código de todos los patrones sea lo más parecido posible para no incluir ruido en las mediciones. Podemos ver como el patrón "streaming" es el más lento, como esperábamos. Ahora bien, los patrones "recency friendly" y "thrashing" tienen casi el mismo número de ciclos por accesos a memoria, pero debería haber más diferencia, pues el "recency friendly" reutiliza todos los bloques del vector cuando lo recorre hacia atrás, mientras que el "thrashing" no tiene nada de reuso.

Pensándolo bien, esto se debe al código de cada patrón, pues si nos acordamos de cómo están implementados en el capítulo 4, el "recency friendly" tiene dos bucles y una instrucción de resta entre ellos, mientras que el "thrashing" solo tiene un bucle. Luego, el "thrashing" tiene menos ruido de instrucciones que no sean exactamente del patrón. Por esto, vamos a igualar las condiciones de los códigos y repetir las medidas.

Los códigos de los patrones "thrashing" y "streaming" los modificamos de forma que tengan dos bucles para recorrer el vector. Vemos a continuación el código del "thrashing", el del "streaming" sería análogo.

```
1        for (Run_Index = 1; Run_Index <= Number_Of_Runs; ++Run_Index)
2        {
3            /*Accedo a posiciones consecutivas del vector
4            desde 0 hasta k/2*/
5            long long int a = 0;
6            int i = 0;
7            while (i < k/2){
8              a = b[i];
9              i = i + 1;
```

```
10              }
11              /*Accedo a posiciones consecutivas del vector
12              desde k/2 hasta k*/
13              while(i < k){
14                a = b[i];
15                i = i + 1;
16              }
17          }
18
```

Con estos cambios, los resultados obtenidos son los presentados en la Figura 5.2. Observamos que el "streaming" es el que tarda más ciclos por cada acceso a memoria, seguido por el "thrashing". Finalmente, el "recency friendly" es el que menos ciclos por acceso requiere. Esto es lo esperable, pues el patrón "recency friendly" se caracteriza por el reuso de los elementos de dentro de cada bloque más el reuso de todos sus bloques al leer el vector de atrás hacia delante, pues con $k = 25$ todos los bloques caben en la caché. Sin embargo, el patrón "thrashing" con una iteración solo reutilizará los elementos de dentro de los bloques. Y finalmente, el "streaming" se caracteriza por no tener nada de reuso.

	Recency friendly (a1...ak,ak...a1)	Streaming a1, a2, a3, ...	Thrashing (a1...ak) k > cache
Ciclos	1983	3057	2058
Instr retiradas	750	1115	877
Ciclos normalizados	1	1,541603631	1,037821483
Ciclos por accesos a memoria	39,66	61,14	41,16
k	25	200	50

Figura 5.2: *Resultados de los patrones con una iteración con el código adaptado.*

5.1.2. Prueba: Coherencia entre los patrones fijado un número de accesos a memoria por iteración.

Primero, fijamos $N = 100$ el número total de iteraciones y vamos a comparar todos los patrones entre sí. Para ello fijamos el tamaño de cada vector en cada patrón según el análisis de accesos a memoria realizado anteriormente para que se realicen exactamente 50 accesos a memoria por iteración o 5000 accesos totales en el caso del patrón "mixed" que no accede

las mismas veces a memoria en todas las iteraciones.

La primera tabla que podemos ver en la Figura 5.3 está dedicada al patrón "recency friendly" que se divide en dos modalidades. Además, tenemos la columna "Limpiar" que se refiere exclusivamente a la parte del procedimiento de limpiar la caché. Así, a la modalidad del patrón que en cada iteración además de recorrer el vector ejecuta el procedimiento, podemos restarle los resultados de la parte del procedimiento limpiar. Y con esto conseguimos que para las medidas se tengan en cuenta solo los accesos al vector principal para cumplir el número de accesos fijado.

En esta ocasión hemos fijado el valor del tamaño del vector en $k = 25$ de forma que tiene menos elementos de los que caben en la caché, pues cada elemento es un long long int que ocupa 8B y la caché es de 256B, luego caben 32 elementos. Al ser más grande la caché que el vector, se da la situación de que una vez acabada la primera iteración tendremos todo el vector cargado en la caché, y si esta no se limpia todo serán aciertos en las siguientes.

	Recency friendly (a1...ak,ak...a1)			
	Limpiando			Sin limpiar
	Total	Limpiar	Menos limpiar	
Ciclos	266213	43891	222322	146506
Instr retiradas	84386	14363	70023	68728
Ciclos normalizados	-	-	1,517494164	1
Ciclos por accesos a memoria	-	-	44,4644	29,3012
k	25 (<cache)			

Figura 5.3: *Resultados de las variaciones del patrón "recency friendly" con 50 accesos a memoria por iteración.*

En la fila normalizada podemos ver para este primer patrón que su versión estándar sin ejecutar la función limpiar es la más rápida. Esto es coherente, pues, como no se limpia la caché después de la primera iteración ya tendremos el vector en caché para las siguientes. Mientras que en la modalidad que limpia la caché entre iteraciones, todas las iteraciones van a tener fallos de caché al cargar el vector, luego serán más lentas. Por eso, aunque le quitemos los ciclos que se usan para limpiar la caché a la versión que limpia entre iteraciones, continúa siendo más lenta.

Ahora vamos a ver en la Figura 5.4 los resultados para el resto de patrones en estas mismas condiciones. Empecemos con el patrón "streaming". Este patrón, por definición, no

tiene nada de reuso de caché, por tanto, podemos ver que es el más lento con diferencia como cabía esperar.

En cuanto al patrón “mixed”, vemos que es más lento que el “recency friendly” estándar pero más rápido que el “recency friendly” que limpia. Esto es porque este patrón reutiliza en 9 de cada 10 iteraciones los datos de la caché, mientras que el “recency friendly” estándar lo hace en todas y el que limpia, en ninguna.

	Streaming a1, a2, a3, ...	Thrashing (a1...ak) k > cache	Mixed (a1...ak,ak...a1, P[b1,...bn]) k < cache, n > cache
Ciclos	323106	180108	169108
Instr retiradas	80278	73216	69886
Ciclos normalizados	2,205411382	1,229355794	1,154273545
Ciclos por accesos a memoria	64,6212	36,0216	33,8216
k	200	50	k = 23, n = 40

Figura 5.4: *Resultados de los patrones “streaming”, “thrashing” y “mixed” con 50 accesos a memoria por iteración.*

Finalmente, tenemos el patrón “thrashing”. Podemos ver que es más lento que el “mixed”. Esto tiene sentido, pues el patrón “mixed” siempre reusa para recorrer el vector hacia atrás, pero 9 de cada 10 veces reusa entre iteraciones. Sin embargo, el patrón “thrashing” no reutiliza tanto los bloques de caché. Esto se debe al $k = 50$ que hemos elegido, pues al caber en la caché 32 elementos y ser esta asociativa con 2 vías y 4 conjuntos, tenemos la siguiente organización de la caché en la primera iteración, que podemos ver en la Figura 5.5.

Conjunto	Via	Elementos
Conjunto 1	Via 1	a1,...,a4
	Via 2	a17,...,a20
Conjunto 2	Via 1	a5,...,a8
	Via 2	a21,...,a24
Conjunto 3	Via 1	a9,...,a12
	Via 2	a25,...,a28
Conjunto 4	Via 1	a13,...,a16
	Via 2	a29,...,a32

Conjunto	Via	Elementos
Conjunto 1	Via 1	a33,...,a36
	Via 2	a49,a50
Conjunto 2	Via 1	a37,...,a40
	Via 2	a21,...,a24
Conjunto 3	Via 1	a41,...,a44
	Via 2	a25,...,a28
Conjunto 4	Via 1	a45,...,a48
	Via 2	a29,...,a32

Figura 5.5: *Organización de la caché a la mitad y al final de la primera iteración del patrón "thrashing", a la izquierda y a la derecha, respectivamente.*

La primera tabla es la organización de la caché al cargar hasta el elemento número treinta y dos. A continuación, como la caché está llena, los nuevos elementos reemplazan a los existentes siguiendo la organización de asociatividad y la política LRU (least recently used). Entonces obtenemos al final de la primera iteración la segunda tabla, donde vemos que siguen estando los elementos $\{a_{21} - a_{32}\}$.

Conjunto	Via	Elementos
Conjunto 1	Via 1	a1,...,a4
	Via 2	a17,...,a20
Conjunto 2	Via 1	a21,...,a24
	Via 2	a5,...,a8
Conjunto 3	Via 1	a25,...,a28
	Via 2	a9,...,a12
Conjunto 4	Via 1	a29,...,a32
	Via 2	a13,...,a16

Figura 5.6: *Organización de la caché en la segunda iteración del patrón "thrashing".*

Sin embargo, en la segunda iteración, al cargar los $\{a_5 - a_8\}$, $\{a_9 - a_{12}\}$, $\{a_{13} - a_{16}\}$ por la política de reemplazo, se colocarán en los bloques correspondientes a los elementos $\{a_{21} - a_{24}\}$, $\{a_{25} - a_{28}\}$, $\{a_{29} - a_{32}\}$. Por tanto, cuando tengamos que cargar estos últimos elementos ya no se encontrarán en caché y el primer elemento de cada bloque nos supondrá un fallo de caché y la organización de la memoria quedaría como podemos ver en la Figura 5.6 a la mitad de la segunda iteración.

Por tanto, el patrón "thrashing" no tendrá aciertos de caché por reuso de bloques con este $k = 50$ tan elevado. Sin embargo, eligiendo un valor de k más ajustado al tamaño de la caché, sí que habría bloques que se conservan entre las iteraciones y, por tanto, un mayor reuso.

5.1.3. Prueba: Impacto del prefetcher de la microarquitectura.

Al hacer pruebas hemos encontrado resultados más rápidos de lo esperado para patrones grandes y revisando la documentación hemos visto que existe prebúsqueda de instrucciones. Por tanto, vamos a hacer experimentos para intentar averiguar si hay prebúsqueda en la cache de datos.

Para ver este comportamiento, vamos a generar dos códigos similares al del patrón "thrashing" de forma que en uno recorramos cuatro vectores distintos que no están almacenados en memoria consecutivamente y en el otro, recorremos un único vector con longitud la suma de los cuatro anteriores.

A continuación, vemos el primero de los códigos donde definimos cuatro vectores distintos no consecutivos en memoria y los recorremos.

```
//Declaracion de los vectores y variables
const int k = 25;
int cont = 0;
long long int numAux = 1;
long long int b[k];
while(cont < k){
b[cont] = numAux;
numAux = numAux + 1;
cont = cont + 1;
}

//Para que no esten consecutivos en memoria
long long int x = 3;
int y = 5;
long long int z = x*y;

cont = 0;
numAux = 100;
long long int c[k];
while(cont < k){
c[cont] = numAux;
numAux = numAux + 1;
cont = cont + 1;
}

long long int xx=3000;
long long int h=3;

cont = 0;
numAux = 200;
long long int d[k];
while(cont < k){
d[cont] = numAux;
numAux = numAux + 1;
```

```
35      cont = cont + 1;
36      }
37
38      long long int yy=5697;
39      int f=50;
40
41      cont = 0;
42      numAux = 300;
43      long long int e[k];
44      while(cont < k){
45      e[cont] = numAux;
46      numAux = numAux + 1;
47      cont = cont + 1;
48      }
49
50      //Empezamos a medir
51
52      long long int a = 0;
53      int i = 0;
54      while (i < k){
55        a = b[i];
56        i = i + 1;
57      }
58      i=0;
59      while (i < k){
60        a = c[i];
61        i = i + 1;
62      }
63
64      i=0;
65      while (i < k){
66        a = d[i];
67        i = i + 1;
68      }
69
70      i=0;
71      while (i < k){
72        a = e[i];
73        i = i + 1;
74      }
75
76      //Finalizamos la medicion
77
```

En el segundo de los códigos que presentamos a continuación, se declara solo un vector de tamaño 100, pero se usan cuatro bucles para recorrerlo, de forma que, tengamos casi el mismo ruido de instrucciones que no son accesos a memoria en ambos códigos. La única diferencia será que el otro código tiene cuatro instrucciones aritméticas sin dependencias (i=0;) de más.

```
1       //Declaracion de los vectores y variables
```

```
    const int k = 100;
    int cont = 0;
    long long int numAux = 1;
    long long int b[k];
    while(cont < k){
    b[cont] = numAux;
    numAux = numAux + 1;
    cont = cont + 1;
    }

    //Empezamos a medir

    long long int a = 0;
    int i = 0;
    while (i < 25){
      a = b[i];
      i = i + 1;
    }

    while (i < 50){
      a = b[i];
      i = i + 1;
    }

    while (i < 75){
      a = b[i];
      i = i + 1;
    }

    while (i < k){
      a = b[i];
      i = i + 1;
    }

    //Finalizamos la medicion

```

Ahora, en la Figura 5.7 podemos ver los ciclos y el número de instrucciones retiradas que se han empleado para ejecutar ambos códigos. Notamos que el caso con cuatro vectores usa muchos más ciclos que el otro. Por la generación de código anterior, el código de los cuatro vectores en C solo tiene 4 instrucciones más, luego el ensamblador debería tener 81 instrucciones más. Como su CPI es de 3.17 aproximadamente, estas instrucciones deberían suponerle unos 250 ciclos extra. Pero la diferencia de ciclos es de 1100. Por tanto, siguen habiendo alrededor de 850 ciclos sin explicación.

Por tanto, hay algún mecanismo que hace que los accesos a vectores largos situados de manera consecutiva en memoria sean más rápidos de lo esperado. Por eso, conjeturamos que podría ser que hubiera prebúsqueda, pero no hemos podido confirmarlo con la documentación a nuestro alcance.

	4 bucles	1 bucle
Ciclos	4801	3693
Instr retiradas	1514	1433
Ciclos normalizados	1	0,769214747
Ciclos por accesos a memoria	48,01	36,93
k	25	100

Figura 5.7: *Resultados del recorrido de cuatro vectores de 25 elementos en contraposición a un vector de 100 elementos.*

5.1.4. Prueba: Variaciones del tamaño del vector en el patrón "recency friendly".

Para esta prueba, haremos solo 1 iteración y modificamos el número de accesos a caché, de forma que tenemos tres casos distintos. Primero, el caso visto en la prueba 5.1.2, donde el número de elementos k del vector es menor que el número de elementos que caben en la caché (32). A continuación, que el número de elementos k del vector sea mayor que el número de elementos que caben en la caché pero menor que dos veces este. Y finalmente, que k sea mayor que dos veces el número de elementos que caben en la caché.

Vemos en los resultados de las mediciones presentados en la Figura 5.8 que la tercera opción necesita bastantes más ciclos para ejecutarse, de hecho, un poco más del triple que la primera versión, pero sin llegar a 4 veces los que necesitaba con $k = 25$. La relación del tamaño del vector y el número de elementos que caben en la caché es importante, pues en la primera versión, el vector cabe entero en la caché, por tanto, solo habrá fallos de caché la primera vez que se cargue en la primera iteración. En las siguientes iteraciones ya tendremos el vector en caché y todo serán aciertos.

	Recency friendly (a1...ak,ak...a1)	Recency friendly (a1...ak,ak...a1)	Recency friendly (a1...ak,ak...a1)
Ciclos	2137	3573	6610
Instr retiradas	750	1400	2700
Ciclos normalizados	1	1,671970051	3,093121198
Ciclos por accesos a memoria	42,74	35,73	33,05
k	25	50	100

Figura 5.8: *Resultados de las variaciones del patrón "recency friendly" con una iteración.*

Sin embargo, siguiendo esta lógica, en el tercer caso con $k = 100$ debería darse lo contrario. Como el vector es mayor que dos veces el número de elementos que caben en la caché, de una iteración a la otra no tenemos bloques para reutilizar. Sin embargo, aún teniendo en cuenta esto y que la tercera versión carga un vector 4 veces más grande, el número de ciclos es menor que 4 veces el que necesita la primera versión. Esto se debe a la prebúsqueda que hemos visto en la prueba 5.1.3, pues como el vector es suficientemente grande, se detecta que se está accediendo muchas veces a posiciones de memoria consecutivas y se activa el mecanismo. De esta forma, estamos teniendo más aciertos de los que teóricamente se debería. Por eso, se explican los números de ciclos por accesos a memoria más bajos tanto en la opción de $k = 100$ como en la de $k = 50$.

Finalmente, el segundo caso es un punto intermedio, en el que solo algunos bloques se quedan en la caché entre iteraciones y, por tanto, se reutilizan. El esquema que tendría la caché una vez recorridos los primeros 32 elementos y al final del primer recorrido hacia delante del vector sería el mismo que el del patrón "thrashing" que explicábamos anteriormente en la Figura 5.5. Sin embargo, desde esta segunda organización, ahora recorremos el vector hacia atrás, por tanto, vemos que tendremos bastantes aciertos, de hecho, hasta el elemento a_{21} incluido serán todo aciertos. Ahora bien, cuando queramos cargar desde el elemento a_{20} al a_1 tendremos fallos de caché por cada bloque que se deba traer a memoria. Y esto nos lleva a la situación que se presenta en la Figura 5.9 cuando llegamos al final de la iteración.

En particular, vemos que los bloques $\{a_{21} - a_{24}\}$, $\{a_{25} - a_{28}\}$, $\{a_{29} - a_{32}\}$ se mantienen siempre en la caché.

Conjunto 1	Via 1	a1,...,a4
	Via 2	a17,...,a20
Conjunto 2	Via 1	a5,...,a8
	Via 2	a21,...,a24
Conjunto 3	Via 1	a9,...,a12
	Via 2	a25,...,a28
Conjunto 4	Via 1	a13,...,a16
	Via 2	a29,...,a32

Figura 5.9: *Organización de la caché al final de una iteración del patrón "recency friendly" con* $k = 50$.

5.1.5. Prueba: Variaciones del tamaño del vector en el patrón "thrashing".

Para esta prueba, mantenemos las 100 iteraciones pero modificamos el número de accesos a caché, de forma que tenemos dos casos distintos. Por una parte, el caso donde el número de elementos k del vector es mayor que el número de elementos que caben en la caché pero menor que dos veces este. Y por otro lado, que k sea mayor que dos veces el número de elementos que caben en la caché. El caso en que el número de elementos k del vector es menor que el número de elementos que caben en la caché, que veíamos en la prueba 5.1.4 anterior para el patrón "recency friendly" aquí no es posible por definición del patrón.

Vemos en la Figura 5.10 como prácticamente tarda los mismos ciclos proporcionalmente, pues hace el doble de accesos y usa casi el doble de ciclos. Además, los ciclos por accesos a memoria son muy similares porque como ya hemos comentado, con $k = 50$ no hay un gran reuso de bloques de memoria. Además, cuantas más iteraciones se hacen los resultados en ciclos son más bajos debido al mecanismo de prebúsqueda que hemos visto anteriormente.

	Thrashing (a1...ak) k > cache	
Ciclos	180108	354823
Instr retiradas	73216	144948
Ciclos normalizados	1	1,970056855
Ciclos por accesos a memoria	36,0216	35,4823
k	50	100

Figura 5.10: *Resultados de las variaciones del patrón "thrashing" con 100 iteraciones.*

5.1.6. Prueba: Variaciones del tamaño de los vectores en el patrón "mixed".

En esta prueba, vamos a comprobar que el patrón "mixed" escale bien. Pero esta vez respecto a sí mismo, variando su parametrización interna del tamaño k y n de los vectores. Los resultados se presentan en la Figura 5.11.

	Mixed (a1...ak,ak...a1, P[b1,...bn]) k < cache, n > cache	
Ciclos	131344	268417
Instr retiradas	57090	111431
Ciclos normalizados	1	2,043618285
Ciclos por accesos a memoria	32,836	33,552125
k	k=15, n=100	k=30, n=200

Figura 5.11: *Resultados del patrón "mixed" con 100 iteraciones totales y distinto tamaño de sus vectores.*

Vemos como al doblar el tamaño de los vectores, y por tanto, el número de accesos a memoria totales, también se doblan los ciclos utilizados. Además, el número de ciclos por accesos a memoria queda muy similar en ambos casos. Notamos que se han elegido ambos k de forma que el vector quepa en la caché, pues lo piden las especificaciones del patrón.

5.1.7. Prueba: Variaciones de la probabilidad en el patrón "mixed".

En esta prueba, vamos a mantener el número de iteraciones en $N = 100$ pero modificamos la probabilidad del patrón "mixed" para ver cómo se comporta. Hasta ahora, en todas las pruebas que se han realizado con este patrón, se ha usado la probabilidad $\varepsilon = 0{,}1$, pero ahora consideraremos también el valor $\varepsilon = 0{,}2$.

Al variar la probabilidad, cambiamos el número de accesos a memoria, pues cuanta más probabilidad más veces se cargará el segundo vector. Por tanto, para tener en todos los casos los mismos accesos a memoria necesitaremos adaptar el tamaño de los vectores. Vamos a fijar en $n = 40$ el tamaño del segundo vector para todas las probabilidades. Estudiando el patrón, tenemos que hace $2kN + \varepsilon Nn$ accesos a memoria en cada iteración. Por tanto, para que hagan los mismos accesos y sea justa la comparación elegimos tomar para $\varepsilon = 0{,}1$ el valor $k = 30$ y para $\varepsilon = 0{,}2$ el valor $k = 28$.

	Mixed (a1...ak,ak...a1, P[b1,...bn]) k < cache, n > cache	
	P=0.1	P=0.2
Ciclos	220000	225273
Instr retiradas	88452	89135
Ciclos normalizados	1	1,023968182
Ciclos por accesos a memoria	34,375	35,19890625
k	30	28

Figura 5.12: *Resultados del patrón "mixed" variando la probabilidad ε.*

Cuanta más probabilidad le demos, se carga más veces el segundo vector en la memoria, reemplazando los bloques del vector "a". De esta forma, cada vez que se ejecuta la carga de "b" en memoria, en la siguiente iteración, al cargar el vector "a" volveremos a tener fallos de caché. Por tanto, a mayor probabilidad, más fallos de caché hay.

5.1.8. Prueba: Elementos de los vectores en distintos bloques de memoria.

En esta prueba, vamos a suponer que cada elemento del vector está en un bloque distinto al del resto de los elementos. De esta forma, todo elemento dará fallo de caché a menos que se reutilice su bloque para acceder a él mismo. Nos quitamos, por tanto, todos los aciertos de memoria para elementos contiguos que estaban en el mismo bloque de las otras pruebas.

	Recency friendly (a1...ak,ak...a1)			
	Limpiando			Sin limpiar
	Total	Limpiar	Menos limpiar	
Ciclos	355619	43891	311728	234838
Instr retiradas	90228	14363	75865	74595
Ciclos normalizados	-	-	1,327417198	1
Ciclos por accesos a memoria	-	-	62,3456	46,9676
k	25 (<cache)			

Figura 5.13: *Resultados del patrón "recency friendly" con los elementos de los vectores en distintos bloques de memoria.*

Tomamos el número de iteraciones $N = 100$ y 50 accesos a memoria por iteración, como en la prueba 5.1.2. Notemos antes de saber los resultados, que el patrón "streaming" ya tenía este comportamiento por definición. Por tanto, no cambiarán sus medidas. Veámoslo junto con el resto de patrones en las Figuras 5.13 y 5.14. Apreciamos en las tablas como claramente el número de ciclos por accesos a memoria ha aumentado en todos los patrones menos en el "streaming" como esperábamos. Y las relaciones entre los patrones se mantienen, en general, como en las pruebas con los elementos del vector compartiendo bloque.

	Streaming a1, a2, a3, ...	Thrashing (a1...ak) k > cache	Mixed (a1...ak,ak...a1, P[b1,...bn]) k < cache, n > cache
Ciclos	323106	253908	247142
Instr retiradas	80278	78988	75758
Ciclos normalizados	1,375867619	1,081204916	1,052393565
Ciclos por accesos a memoria	64,6212	50,7816	49,4284
k	200	50	k = 23, n = 40

Figura 5.14: *Resultados de los patrones patrones "streaming", "thrashing" y "mixed" con los elementos de los vectores en distintos bloques de memoria.*

	Streaming a1, a2, a3, ...	Thrashing (a1...ak) k > cache
Ciclos	178735	122770
Instr retiradas	54404	50435
Ciclos normalizados	1,455852407	1
Ciclos por accesos a memoria	52,56911765	36,10882353
k	136	34

Figura 5.15: *Comparación del patrón "streaming" con el patrón "thrashing" para* $k = 136$ *y* $k = 34$ *respectivamente.*

Hagamos una pequeña comparación entre el "streaming" y el "thrashing" por separado para un $k = 34$ en el "thrashing" que se corresponde con un $k = 136$ en el "streaming". Vemos así en la Figura 5.15 como al acercar el número de elementos k del vector del "thrashing" al número de elementos que caben en la caché, se aumenta el reuso de bloques y, por tanto, el "thrashing" es mucho más rápido que el "streaming".

5.2. Pruebas en el emulador con caché directa de datos y caché asociativa de 4 conjuntos y 2 vías de instrucciones.

5.2.1. Prueba: Coherencia entre los patrones fijado un número de accesos a memoria por iteración.

Esta prueba es análoga a la Prueba 5.1.2, pero usando una caché de datos directa. Por tanto, fijamos el número de iteraciones en $N = 100$ y para cada patrón ajustamos el tamaño de sus vectores de forma que en total haya 50 accesos a memoria por iteración, excepto en el "mixed" que en lugar de por iteración, serán 5000 totales.

	Recency friendly (a1...ak,ak...a1)			
	Limpiando			Sin limpiar
	Total	Limpiar	Menos limpiar	
Ciclos	305586	43166	262420	172325
Instr retiradas	84762	14363	70399	68911
Ciclos normalizados	-	-	1,522820252	1
Ciclos por accesos a memoria	-	-	52,484	34,465
k	25 (<cache)			

Figura 5.16: *Resultados del patrón "recency friendly" con 50 accesos a memoria por iteración.*

	Streaming a1, a2, a3, ...	Thrashing (a1...ak) k > cache	Mixed (a1...ak,ak...a1, P[b1,...bn]) k < cache, n > cache
Ciclos	316637	198416	204584
Instr retiradas	80080	73409	70277
Ciclos normalizados	1,837440882	1,151405774	1,187198607
Ciclos por accesos a memoria	63,3274	39,6832	40,9168
k	200	50	k = 23, n = 40

Figura 5.17: *Resultados de los patrones "streaming", "thrashing" y "mixed" con 50 accesos a memoria por iteración.*

Podemos ver en las Figuras 5.16 y 5.17 como el número de ciclos totales respecto a la prueba 5.1.2 ha aumentado. Con lo cual, vemos el aumento reflejado en el número de ciclos por accesos a memoria. Además, las relaciones entre patrones se mantienen. Luego deducimos que la caché asociativa es de gran ayuda para solucionar conflictos de bloques en memoria y ayuda a que haya un mayor reuso de bloques de memoria.

Finalmente, a continuación pasamos a hacer una recapitulación de todos los conceptos y procedimientos que hemos visto en capítulos anteriores, así como los resultados de los

tests que hemos explicado en este capítulo. Asimismo, presentaremos las conclusiones que se obtienen de este trabajo.

Capítulo 6

Conclusiones

Una vez hemos realizado todo el proceso de documentación de las herramientas utilizadas, se han generado emuladores y los correspondientes tests personalizados para ellos, y hemos obtenido resultados basados en las medidas de rendimiento que hemos conseguido al ejecutar los distintos tests, pasamos a sacar las conclusiones del trabajo.

En primer lugar, se ha conseguido el primero de los objetivos que era familiarizarse con la arquitectura RISC-V, pues la hemos estado utilizando en los emuladores que hemos generado. Esto nos hace cumplir con el siguiente objetivo, el de aprender a usar las herramientas asociadas a RISC-V. En particular, hemos aprendido a usar compiladores cruzados cuando compilamos los tests de forma adecuada para ejecutarlos sobre el emulador; también, el OpenOCD y el gdb, que nos han permitido depurar los tests ejecutados sobre un emulador. Esto ha sido un punto clave, pues al principio del desarrollo de los tests tuvimos que usar la depuración sobre el emulador para ver que se cargase el ejecutable del test correctamente para verificar que estaba compilado de acuerdo a las necesidades del emulador.

En segundo lugar, hemos aprendido a crear diseños de microarquitecturas con el Rocket-chip de forma fácil y sencilla. Este proceso ha sido arduo por la falta de documentación al respecto, pero con esfuerzo y constancia se ha conseguido tener un gran manejo de esta herramienta tan potente y útil.

También, hemos aprendido el lenguaje Chisel que permite parametrizar la generación de los diseños aprovechando la capacidad de abstracción de Scala y así que sea sencillo modificar parámetros de los diseños de la microarquitectura.

Posteriormente, hemos reafirmado los conocimientos sobre la jerarquía de la caché, cuando hemos tenido que ajustar los parámetros de los tests y el tamaño y organización de las cachés de los emuladores para conseguir unos resultados de los tests coherentes. Además, los tests nos han enseñado a cómo generar distintos patrones de memoria con y sin reuso.

A continuación, hemos sacado unos resultados en base a las mediciones de rendimiento

sobre los tests. Las mediciones han sido realizadas con los contadores hardware para rendimiento que incorpora el core Rocket y que son de gran utilidad. Estos resultados nos han confirmado que un programa ejecutado sobre una caché de datos directa necesitará más ciclos que el mismo programa ejecutado sobre una caché asociativa, pues esta última nos resuelve algunos problemas de colocación en memoria, consiguiendo un mayor reuso de datos.

Finalmente, hemos podido ver el comportamiento de los patrones gracias a los resultados de los tests. Como cabía esperar se puede ver en las pruebas con iteraciones y en los patrones puros como el patrón "streaming" destaca por ser el más lento por su nulo reuso de datos. Sin embargo, el "recency friendly" destaca por ser el de mayor reuso, ya que dependiendo del tamaño del vector puede reutilizar todo el contenido de la memoria. Vemos que el patrón "thrashing" tiene poco reuso en general, aunque si ajustamos el tamaño del vector a un tamaño cercano al número de elementos que caben en la caché, conseguimos que tenga bastante reuso entre iteraciones. Y finalmente, el patrón "mixed" a mayor probabilidad menos reuso tiene.

Con esto damos por alcanzados todos los objetivos y concluimos el trabajo realizado, aunque una futura extensión podría ser explorar otros componentes de una microarquitectura basada en RISC-V. Por ejemplo, modificar parámetros del core Rocket, usar el core BOOM que ejecuta instrucciones fuera de orden o añadir un acelerador a la microarquitectura.

Capítulo 7

Conclusions

Once we have made the whole process of documentation of the tools used, emulators have been generated and the corresponding tests customized for them, and we have obtained results based on the performance measures that we have achieved when executing the different tests, we move on to draw the conclusions from the work.

In the first place, the first of the objectives has been achieved, which was to become familiar with the RISC-V architecture, since we have been using it in the emulators that we have generated. This makes us fulfill the following objective, that of learning to use the tools associated with RISC-V. In particular, we have learned to use cross compilers when compiling the tests properly to run on the emulator; also, OpenOCD and gdb, which have allowed us to debug the tests executed on an emulator. This has been a key point, because at the beginning of the development of the tests we had to use debugging on the emulator to see that the executable of the test was loaded correctly to verify that it was compiled according to the needs of the emulator.

Second, we have learned how to create microarchitecture designs with the Rocket-chip easily and simply. This process has been arduous due to the lack of documentation in this regard, but with effort and perseverance it has been possible to have a great use of this powerful and useful tool.

Also, we have learned the Chisel language that allows to parameterize the generation of the designs taking advantage of Scala's capacity for abstraction and thus make it easy to modify parameters of the microarchitecture designs.

Subsequently, we have reaffirmed our knowledge about the cache hierarchy, when we had to adjust the test parameters and the size and organization of the emulator caches to achieve consistent test results. In addition, the tests have taught us how to generate different memory patterns with and without reuse.

Next, we have obtained some results based on the performance measurements on the

tests. The measurements have been made with the hardware performance counters that the Rocket core incorporates and that are very useful. These results have confirmed that a program executed on a direct data cache will need more cycles than the same program executed on an associative cache, since the latter solves some memory placement problems, achieving greater data reuse.

Finally, we have been able to see the behavior of the patterns thanks to the test results. As expected, it can be seen in the tests with iterations and in the pure patterns how the " streaming " pattern stands out for being the slowest due to its zero data reuse. However, the " recency friendly " stands out for being the one with the highest reuse, since depending on the size of the vector it can reuse all the memory content. We see that the " thrashing " pattern has little reuse in general, although if we adjust the size of the vector to a size close to the number of elements that can fit in the cache, we get it to have a lot of reuse between iterations. And finally, the pattern " mixed " the higher the probability, the less reuse it has.

With this, we consider all the objectives achieved and we conclude the work carried out, although a future extension could be to explore other components of a microarchitecture based on RISC-V. For example, modifying parameters of the Rocket core, using the BOOM core that executes instructions out of order or adding an accelerator to the microarchitecture.

Bibliografía

[1] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, and John Koenig. *The Rocket Chip Generator.*, University of California, Berkeley, April 15, 2016.

[2] Krste Asanović, Rimas Avižienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Palmer Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Benjamin Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. *The Rocket Chip Generator, Technical Report UCB/EECS-2016-17.* EECS Department, University of California, Berkeley, April 2016.

[3] Christopher Celio, David Patterson, and Krste Asanovic. *The Berkeley Out-of-Order Machine (BOOM) Design Specification*, University of California, Berkeley, California 94720-1770, December 4, 2016.

[4] Chipsalliance. *Repositorio de Rocket-tools.* Se encuentra en `https://github.com/chipsalliance/rocket-tools`, 2020.

[5] Chisel. *Chisel/FIRRTL Hardware Compiler Framework.* Se encuentra en `https://github.com/freechipsproject/chisel3`.

[6] Hasan Genc, Ameer Haj-Ali, Vighnesh Iyer, Alon Amid, Howard Mao, John Wright, Colin Schmidt, Jerry Zhao, Albert Ou, Max Banister, Yakun Sophia Shao, Borivoje Nikolic, Ion Stoica, and Krste Asanovic. Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures. arXiv preprint arXiv:1911.09925, 2019.

[7] RISC-V International. *Repositorio del proxy kernel pk.* Se encuentra en `https://github.com/riscv/riscv-pk`.

[8] RISC-V International. *Readme de GDB.* Se encuentra en `https://github.com/riscv/riscv-binutils-gdb`, 2020.

[9] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel Emer. *High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP).* ISCA'10, 06-19–23, 2010, Saint-Malo, Francia.

[10] Ben Keller. *RISC-V, Spike, and the Rocket Core. CS250 Laboratory 2 (Version 091713)*, University of California, Berkeley, Fall, 2013.

[11] Yunsup Lee. *RISC-V "Rocket Chip" SoC Generator in Chisel*, UC Berkeley.

[12] Yunsup Lee, Colin Schmidt, Albert Ou, Andrew Waterman, and Krste Asanović. *The Hwacha Vector-Fetch Architecture Manual, Version 3.8.1. Technical Report, UCB/EECS-2015-262.*, University of California, Berkeley, December 2015.

[13] Yunsup Lee, Brian Zimmer, Andrew Waterman, Alberto Puggelli, Jaehwa Kwak, Ruzica Jevtic, Ben Keller, Stevo Bailey, Milovan Blagojevic, Pi-Feng Chiu, Henry Cook, Rimas Avizienis, Brian Richards, Elad Alon, Borivoje Nikolic, and Krste Asanovic. *Raven: A 28nm RISC-V Vector Processor with Integrated Switched-Capacitor DC-DC Converters and Adaptive Clocking.*, University of California, Berkeley.

[14] Howard Mao and Jerry Zhao. *Generating Rocket/BOOM SoCs with Rocket Chip*, UC Berkeley Architecture Research, Hot Chips 2019.

[15] Michael Gielda (mgielda). *RISC-V Cores and SoC Overview.* Se encuentra en `https://github.com/riscv/riscv-cores-list`.

[16] David Patterson and Andrew Waterman. Traducido por Alí Lemus y Eduardo Corpeño. *Guía Práctica de RISC-V: El Atlas de una Arquitectura Abierta. Primera Edición, 1.0.5*, 11 de Julio de 2018.

[17] Berkeley Architecture Research. *Chipyard Documentation*, Sep 06, 2020.

[18] RISC-V Universal. *Readme de OpenOCD para RISC-V.* Se encuentra en `https://github.com/riscv/riscv-openocd`, 2020.

[19] RISC-V Universal. *Repositorio de Spike.* Se encuentra en `https://spike.readme.io/docs/getting-started`, 2020.

[20] RISC-V Universal. *RISC-V Origin.* Se encuentra en `https://riscv.org/about/history/`, 2020.

[21] Editors Andrew Waterman, Krste Asanović, and RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-draft*, December 2019.

[22] John Wawrzynek, Krste Asanovic, John Lazzaro, and Brian Zimmer. *CS250 VLSI Systems Design. Lecture 11: Patterns for Communication Links, Rocket μArchitecture, Testing*, UC Berkeley, Fall 2011.

[23] Scala y colaboradores (mlachkar y alvinj). *SCALA BOOK. PRELUDE: A TASTE OF SCALA.* Se encuentra en `https://docs.scala-lang.org/overviews/scala-book/prelude-taste-of-scala.html`.

Apéndice A

Instalación de las herramientas

En este apéndice se detallan los pasos a seguir para instalar las herramientas necesarias para poder usar Rocket-chip con todas sus funcionalidades en un sistema con Ubuntu.

1. Descargamos el repositorio de Rocket-Chip.

   Usamos el comando de github que nos permite clonar el repositorio.

   ```
   $ git clone https://github.com/chipsalliance/rocket-chip
   ```

   Si no tenemos instalado git para poder manejar repositorios de github se instala con el siguiente comando:

   ```
    $ sudo apt-get install git
   ```

   Ahora actualizamos los submódulos del proyecto como sigue:

   ```
   $ cd rocket-chip
   $ git submodule update --init
   ```

2. Descargamos el repositorio de rocket-tools.

   Vamos a necesitar el repositorio de rocket-tools para las herramientas especiales que necesita RISC-V. Así que lo descargamos también y lo actualizamos. Podemos descargarlo dentro de rocket-chip o en cualquier otro lugar, pues después simplemente lo añadiremos a las rutas del sistema. Para ello ejecutamos los siguientes comandos:

   ```
   $ git clone https://github.com/freechipsproject/rocket-tools
   $ cd rocket-tools
   $ git submodule update --init --recursive
   ```

   Ahora, exportamos las variables globales que necesita el Makefile del repositorio de rocket-tools.

```
$ export RISCV=/ruta/donde/se/instalaran/las/herramientas
$ export MAKEFLAGS="$MAKEFLAGS -jN"
```

donde N es el número de cores que tiene el sistema donde está el repositorio.

3. Instalamos las dependencias para generar las herramientas de rocket-tools.

   El siguiente comando las instala todas de una vez.

   ```
   sudo apt-get install autoconf automake autotools-dev curl libmpc-
   dev libmpfr-dev libgmp-dev libusb-1.0-0-dev gawk build-essential
   bison flex texinfo gperf libtool patchutils bc zlib1g-dev device-
   tree-compiler pkg-config libexpat-dev libfl-dev openjdk-8-jdk
   ```

4. Generamos las herramientas de rocket-tools. (∼30min)

   Debemos ejecutar el Makefile del repositorio para que nos genere las herramientas adecuadas. Usaremos uno de los siguientes comandos en función de nuestras necesidades: el primero, si vamos a generar emuladores de RISC-V de 64 bits que es el por defecto; o el segundo, para generar emuladores con RISC-V de 32 bits.

   ```
   $ ./build.sh
   $ ./build-rv32ima.sh
   ```

   Y añadimos a la ruta del equipo el directorio `/rocket-tools/bin`.

5. Reiniciamos el ordenador.
   Si las variables anteriores de RISCV y PATH se habían añadido permanentemente al entorno de inicio del sistema no habrá problema. En caso contrario, habrá que volverlas a exportar.

6. Generar un emulador en C para que se acabe de configurar el Rocket-chip. Para generar el emulador por defecto basta con ir a `/rocket-chip/emulator` y ejecutar el comando make.

   Esto generará una carpeta llamada generated-src y un ejecutable llamado emulator-freechips.rocketchip.system-freechips.rocketchip.system.DefaultConfig.

Ahora por seguridad, solo quedaría generar los tests por defecto y comprobar que el emulador funciona con uno de esos tests para asegurarnos de que todo ha quedado bien instalado. Este procedimiento ya se detalla en la sección 2.Conceptos previos en la memoria.