Universidad de Valladolid

ESCUELA DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN

Simulador de múltiples arquitecturas segmentadas de computadores

---

Curso académico:
2020-2021

Autor:
Manuel de Castro Caballero

Tutores académicos:
Javier Bastida Ibáñez
Yuri Torres de la Sierra

A mi tío y padrino,
Juan Pablo.
Puede que no nos hagamos ricos,
pero estamos en esto para divertirnos.

Agradecimientos

Me gustaría agradecer a mis tutores, Javier Bastida Ibáñez y Yuri Torres de la Sierra, por haberme dado la oportunidad de trabajar en este proyecto que he encontrado tan interesante. También me gustaría agradecer a todos aquellos profesores y profesoras que se ofrecieron a prestarme su ayuda en algún punto de su desarrollo: Valentín Cardeñoso Payo, Yania Crespo González Carvajal, Francisco José Andújar Muñoz y Arturo González Escribano.

A mi familia, por apoyarme de manera incondicional durante toda mi trayectoria académica, y por financiármela; incluyendo aquellas asignaturas que he cursado de manera innecesaria, puramente por gusto, durante este último cuatrimestre.

A mis amigos del Grupo Universitario de Informática, en especial a Pablo Martínez López, Javier Gatón Herguedas y Hugo Prieto Tárrega, por estar ahí para ayudarme con las ramas de la informática que peor se me dan, siempre que lo he necesitado.

Por último pero no menos importante, a todos mis amigos que han tenido la paciencia suficiente para esperarme durante todo este tiempo que he estado trabajando demasiado. En especial, a Daniel López Martínez, Sofía Mara Rivas Cuevas y Lucía Alonso Losa.

Resumen

Los lenguajes ensambladores son comúnmente estudiados en asignaturas básicas sobre Arquitectura de Computadores para explicar el funcionamiento de los procesadores. Existe un conjunto significativo de lenguajes ensambladores surgidos de las distintas arquitecturas de computadores existentes. Dicho conjunto de lenguajes va en aumento conforme se desarrollan más arquitecturas hardware. Elegir qué lenguaje ensamblador estudiar y de qué modo es una decisión limitada a las tecnologías de desarrollo o simulación existentes para cada arquitectura.

Este trabajo describe la implementación de un prototipo de simulador de lenguajes ensambladores con propósito docente escrito en Java. Este simulador ha sido desarrollado para soportar un conjunto extensible de lenguajes ensambladores distintos, centrándose en aquellos de arquitecturas RISC. Actualmente, está implementado el *backend* para ARM LEGv8, arquitectura descrita en *Computer Organization and Design: ARM edition*. Este *backend* implementa funcionalidades de segmentación de instrucciones, tales como las descritas en *Computer Architecture: A Quantitative Approach*, incluyendo la simulación de unidades funcionales multiciclo. También se ha implementado el *backend* para el subconjunto de instrucciones RV64I de RISC-V, validando la capacidad de extensión del simulador.

El trabajo desarrollado en este proyecto ha dado lugar a dos publicaciones científicas que han sido aceptadas y serán presentadas en las XXXI Jornadas de Paralelismo 2020/2021 de la Sociedad de Arquitecturas de Computadores (SARTECO).

Consideramos que la herramienta desarrollada puede ser de gran utilidad tanto para docentes como para estudiantes de asignaturas básicas de Arquitectura de Computadores.

Abstract

Assembly languages are commonly studied in basic Computer Architecture courses to explain the inner workings of processors. A significantly large set of assembly languages has arisen from the various different computer architectures that exist. Said set is increasing as more hardware architectures are being developed. Choosing which assembly language to study in a subject and how to do so is a decision limited by the development and simulation tools available for each architecture.

This project describes the implementation of a prorotype of an education-oriented, Java-based assembly language simulator. This simulator has been developed to support an increasing set of different assembly languages, focusing on those of RISC architectures. Currently, the ARM LEGv8 backend is implemented, whose architecture is described in *Computer Organization and Design: ARM edition*. This backend implements instruction pipelining functionalities such as the ones described in *Computer Architecture: A Quantitative Approach*, including the simulation of multicycle functional units. The RV64I subset of instructions from the RISC-V architecture has also been implemented, proving the extension capabilities of the simulator.

The work developed in this project has led to the writing of two scientific articles that have been accepted and will be presented in the XXXI Jornadas de Paralelismo 2020/2021, organized by the Sociedad de Arquitectura de Computadores (SARTECO).

We consider that the developed tool might be really useful to both undergraduate computer science students and Computer Architecture professors.

Índice general

Índice de figuras

Índice de tablas

Índice de fragmentos de código

Capítulo 1

Introducción

En este capítulo se presentan los siguientes aspectos:

- Contexto en el que se sitúa el trabajo.
- Motivación y objetivos del trabajo.
- Estructura del documento.

1.1. Contexto

En esta sección se presenta el contexto en el que se desarrolla este proyecto. Se introducen los principales fundamentos de la Arquitectura de Computadores, de su estudio en ámbitos docentes, y de su importancia en el ámbito de la Computación de Alto Rendimiento.

1.1.1. Arquitectura de Computadores

La Arquitectura de Computadores es una rama de la Ingeniería Informática que estudia la funcionalidad, organización e implementación de sistemas de computadoras. En el ámbito de la Arquitectura de Computadores, es de especial interés el estudio de cómo la unidad de procesamiento central (CPU) funciona de forma interna y se comunica con la memoria. Entre los campos englobados por esta rama destacan:

- Diseño de los conjuntos de instrucciones de las computadoras.
- Diseño de microarquitecturas y organización de computadoras.
- Diseño lógico de computadoras.
- Implementación hardware de computadoras.

La Arquitectura de Computadores es una disciplina de gran relevancia en el contexto de la Ingeniería Informática y las Ciencias de la Computación. El software que se ejecuta a diario en millones de dispositivos depende del correcto funcionamiento de un hardware para funcionar. Comprender este hardware es una necesidad fundamental a la hora de desarrollar mejores programas, en cuanto a eficiencia computacional y energética. Por tanto, los avances en el desarrollo de computadores implican de forma directa mejoras en el desarrollo de software. No es sino gracias a la evolución de los computadores durante las últimas dos décadas que hoy en día podemos realizar tareas que antes tan solo teorizábamos: teléfonos inteligentes, redes neuronales y *deep learning* [1], e *Internet of Things* [2], por ejemplo.

Existen múltiples ámbitos informáticos en los que es necesario tener cierto conocimiento de las arquitecturas de computadores. Entre ellos destacamos:

- Desarrollo de compiladores: los compiladores traducen código fuente escrito en lenguajes "de alto nivel" a código máquina ejecutable por los computadores. El uso de compiladores es imprescindible en el proceso de desarrollo de software. Para desarrollar un compilador es necesario tener conocimientos de las arquitecturas de computadores para las que se pretende generar código ejecutable. Además, cuanto mejor exploten los compiladores las características particulares de estas arquitecturas, más eficientes serán los programas ejecutables generados, tanto en velocidad como en uso de recursos.
- Desarrollo de software para sistemas empotrados: los sistemas empotrados o embedidos son sistemas computacionales con una cantidad especialmente limitada de recursos (memoria, espacio de disco, potencia de cómputo). Estos sistemas se utilizan para realizar pocas tareas específicas. Los desarrolladores para sistemas empotrados deben conocer con exactitud las limitaciones de estos sistemas, así como las capacidades de sus arquitecturas, para lograr aprovechar su máximo rendimiento.
  Hoy en día, los sistemas empotrados son utilizados en gran cantidad de situaciones y contextos. Por ejemplo, en taxímetros, sistemas de control de acceso, máquinas expendedoras, sistemas de control de impresoras y fotocopiadoras, etc.

Existen diversos simuladores de arquitecturas de computadores con distintos propósitos. Algunos simuladores tienen como finalidad facilitar el desarrollo de software para ciertos microcontroladores. Estos son principalmente utilizados en los contextos de los sistemas empotrados. Otros simuladores tienen una finalidad didáctica, y son especialmente utilizados en los contextos docentes.

1.1.2. Estudio de la Arquitectura de Computadores en la universidad

Para dedicarse profesionalmente al sector de la informática es necesario tener cierta formación en Arquitectura de Computadores. Esta necesidad surge por la influencia significativa de las arquitecturas de computadores en cualquier actividad informática: todo software se ejecuta en un computador. Es importante conocer las características de los computadores para comprender el proceso de ejecución de cualquier software.

La formación básica en Arquitectura de Computadores es de importancia en toda clase de

profesiones: desde aquellas que consisten en desarrollar aplicaciones para ordenadores personales, hasta las que trabajen en las áreas de la ciencia de datos e inteligencia artificial. Por ello, cualquier clase de formación en ciencias o ingenierías relacionadas con la informática suele incluir en su plan de estudios asignaturas de carácter obligatorio dedicadas exclusivamente al estudio del diseño de computadores.

En el caso del grado en Ingeniería Informática de la Universidad de Valladolid [3], debemos destacar la presencia de dos asignaturas obligatorias: Fundamentos de Computadoras [4], y Arquitectura y Organización de Computadoras [5]. Estas asignaturas son estudiadas en el primer y segundo curso del grado respectivamente, y cubren la formación de todos los alumnos del grado con respecto a Arquitectura de Computadores. Otra asignatura, Arquitecturas de Computación Avanzadas [6], de carácter optativo para el tercer curso de la mención de Tecnologías de la Información, complementa la formación de las dos asignaturas anteriores mediante el estudio del diseño de los computadores más actuales. Estas asignaturas presentan conceptos que sirven como base para desarrollar otras asignaturas del grado, tales como Estructura de Sistemas Operativos [7], Lenguajes de Programación [8], Computación Paralela [9], Sistemas Empotrados [10], Hardware Empotrado [11], y Rendimiento y Evaluación de Computadores [12], entre otras.

En las asignaturas de Fundamentos de Computadores, Arquitectura y Organización de Computadores, y Arquitecturas de Computación Avanzadas de la Universidad de Valladolid, se utiliza *MARS* [13], un simulador de una arquitectura de computadores como herramienta docente de apoyo. Utilizando este simulador se desarrollan clases y actividades prácticas que sirven como complemento a las clases teóricas de las asignaturas. El uso de simuladores fomenta la adquisición de competencias básicas y avanzadas sobre Arquitectura de Computadores por parte de los alumnos, por lo que se trata de herramientas de gran utilidad.

1.1.3. Arquitecturas de Alto Rendimiento

La Computación de Alto Rendimiento (*High Performance Computing*, HPC) se refiere al desarrollo de sistemas computacionales de gran capacidad de cálculo, y al estudio de su aplicación para la resolución de problemas complejos de ciencia, ingeniería o gestión [14]. Los campos de estudio de HPC abarcan diversas disciplinas informáticas, como las de diseño de hardware, co-diseño de hardware-software, diseño de aplicaciones, modelos de cómputo, lenguajes de programación, optimización de código, técnicas de compilación, sistemas de ejecución, etc. Todas estas disciplinas comparten un punto común, y es que afectan al rendimiento obtenido por las unidades computacionales.

La Arquitectura de Computadores juega un papel fundamental en el ámbito de HPC. A lo largo de la historia ha habido numerosos intentos de aumentar la capacidad de cómputo ofrecida por las unidades computacionales. Estos intentos han dado lugar a arquitecturas de computadores específicas, como las siguientes:

- Arquitecturas superescalares: aquellas cuyas unidades computacionales ejecutan múltiples operaciones mediante la replicación de componentes hardware.
- Arquitecturas distribuidas: aquellas formadas por redes de unidades computacionales que se

comunican mediante el paso de mensajes entre ellas.

- Arquitecturas *multicore*: aquellas que combinan varias unidades computacionales en un mismo circuito, y permiten su uso simultáneo mediante mecanismos de sincronización.
- Arquitecturas vectoriales: aquellas que introducen paralelismo de datos mediante el uso de hardware que ejecuta la misma operación sobre una gran cantidad de datos, de manera simultánea.
- Arquitecturas heterogéneas: aquellas que utilizan múltiples unidades computacionales de distinta naturaleza de forma sincronizada para obtener un mayor rendimiento en diversos problemas de carácter específico.

Todas estas arquitecturas se basan en los mismos principios comunes. Sin embargo, presentan diferencias significativas entre ellas, convirtiéndolas en campos de estudio distintos.

Gran cantidad de los avances más significativos en tecnologías informáticas han sido debidos al desarrollo en el campo de la Arquitectura de Computadores. Muchos de estos avances provienen del ámbito de HPC. El afán por desarrollar computadores más potentes ha revolucionado la tecnología actual, presentando nuevas arquitecturas de computadores, con un mayor rendimiento, que utilizar en nuestro día a día.

En la sociedad actual, con la informatización de cada vez más sistemas y aspectos de la vida cotidiana, la Computación de Alto Rendimiento domina el paradigma de desarrollo de arquitecturas de computadores. El ritmo en el que avanza la tecnología actual está determinado por la potencia de cómputo de la que disponemos. Como consecuencia, cada vez se desarrollan arquitecturas más complejas en busca de obtener un mayor rendimiento computacional.

En estos contextos, en ocasiones se utilizan simuladores con diversos fines. Un posible fin es realizar estimaciones de la productividad de una arquitectura concreta, ahorrando los costes asociados a la construcción física de dicha arquitectura. Un ejemplo de esta clase de simuladores es el simulador Hiperion [15] Otro fin consiste en depurar y optimizar fragmentos de código de aplicaciones de HPC. Los simuladores pueden ofrecer estimaciones de parámetros difíciles de medir en sistemas reales, además de presentar facilidades de ejecución con las que ciertos *clústers* de supercomputadores no cuentan. De este modo, puede resultar más conveniente o sencillo optimizar un programa haciendo uso de simuladores, en vez de solamente realizando experimentos en máquinas reales.

1.2. Motivación

Es habitual que las asignaturas de Arquitectura de Computadores cuenten con clases prácticas. Estas clases prácticas agilizan el aprendizaje de los alumnos, ilustrando desde otro punto de vista los conceptos estudiados en las clase teóricas. Por tanto, estas clases prácticas son de gran utilidad para la formación de los alumnos.

En estas clases prácticas se suelen desarrollar actividades entorno al uso de algún simulador de una arquitectura y lenguaje ensamblador específico. Con estos simuladores, los alumnos experi-

mentan de primera mano el funcionamiento de las CPUs. Esta aproximación presenta numerosas ventajas:

- Bajo coste material y económico.
- Accesibilidad y facilidad de uso de los simuladores.
- Alta relación entre el tiempo y esfuerzo invertido en el estudio y los conocimientos adquiridos.

Idealmente, los simuladores utilizados tienen como arquitectura objetivo alguna arquitectura de actualidad, facilitando la contextualización e incrementando la utilidad de las actividades realizadas. Sin embargo, este no suele ser el caso, utilizándose simuladores para arquitecturas obsoletas por distintos motivos: familiaridad con el simulador, funcionalidades del simulador, o dificultad por encontrar simuladores para otras arquitecturas más actuales.

El desarrollo de nuevos simuladores es una tarea costosa en tiempo y esfuerzo. Este esfuerzo se incrementa si se desea utilizar todas las funcionalidades de los simuladores ya existentes, en vez de solo las más básicas, con una arquitectura más novedosa. La actualización de simuladores ya existentes a las arquitecturas más novedosas tampoco resulta particularmente sencillo, ya que los simuladores pueden haberse desarrollado sin tener en cuenta este posible deseo de actualización. Sumado a una escasa documentación del código fuente, puede resultar más difícil actualizar un simulador existente que desarrollar uno nuevo.

También se ha detectado una escasez de simuladores en el ámbito de la Computación de Alto Rendimiento. Los simuladores de arquitecturas de computadores pueden presentar múltiples utilidades en esos contextos, como podrían ser la optimización de código de aplicaciones específicas, evaluar la eficacia de compiladores optimizadores, o simular la funcionalidad de una propuesta de arquitectura.

1.3. Objetivos

En las siguientes secciones se presenta el objetivo que se pretende cumplir con este proyecto, así como los objetivos secundarios del mismo.

1.3.1. Objetivo del proyecto

En este proyecto se propone el desarrollo de un prototipo de simulador de arquitecturas de computadores con propósito principalmente didáctico. Este simulador debe cubrir las necesidades básicas de las asignaturas dedicadas al estudio de este campo. Asimismo, debe reflejar en la medida de lo posible el contexto actual de la Arquitectura de Computadores, presentando características de los computadores utilizados a día de hoy, al menos a nivel fundamental. Con esto se pretende maximizar la utilidad de los conceptos que se puedan estudiar con su uso.

El simulador debe ser fácilmente extensible y actualizable, de forma que incrementar sus funcionalidades o reacomodarlo a otras características más novedosas resulte sencillo. De este modo, si en un futuro se considerase necesario utilizar un simulador de una arquitectura de computadores con unas características distintas, no fuese necesario desarrollar otro simulador completamente nuevo desde cero. Esto reducirá considerablemente el esfuerzo y tiempo de actualización del software.

El simulador va a ser utilizado en las asignaturas del grado en Ingeniería Informática de la Universidad de Valladolid de Fundamentos de Computadoras [4] y Arquitectura y Organización de Computadoras [5]. Por tanto, debe cubrir las necesidades básicas de esas dos asignaturas, al menos al mismo nivel que el simulador utilizado hasta la fecha.

También, es utilizado en el grupo de Investigación Trasgo de la Universidad de Valladolid para ayudar a la optimización del código de aplicaciones de HPC. Por ejemplo, se ha utilizado para optimizar una aplicación de cómputos de tipo *stencil* [16] para ser ejecutada en dispositivos NVIDIA Jetson Nano [17] con arquitectura ARM. El simulador debe presentar funcionalidades de utilidad para la simulación de aplicaciones reales.

1.3.2. Objetivos secundarios

Los objetivos secundarios del proyecto tratan de asegurar unos niveles de calidad mínimos, contribuyendo indirectamente al objetivo principal. Estos objetivos son:

- Se debe realizar un análisis de las arquitecturas de computadores más utilizadas actualmente. Se deben identificar las características comunes a todas ellas para facilitar la extensibilidad del simulador.

- Se debe realizar un análisis exhaustivo de la arquitectura ARM LEGv8 para ser utilizada como arquitectura principal del simulador. Asimismo, se deben analizar sus diferencias con la arquitectura ARMv8 original y la viabilidad de utilizar esta última en el simulador.

- El código fuente del prototipo de simulador debe ser de fácil comprensión y modificación. Este objetivo deriva de la necesidad de que el simulador sea fácilmente extensible y actualizable. Esa necesidad conlleva implicaciones no solo estructurales en su diseño, sino también de estilo en su código fuente.

- Se deben analizar las tecnologías software actualmente disponibles que sean de utilidad para el desarrollo del simulador. Con este objetivo se pretenden tomar decisiones de implementación similares o mejores que las observadas en los simuladores ya existentes. Las tecnologías finalmente elegidas se deben evaluar durante el desarrollo del proyecto.

- El prototipo de simulador debe ser validado, analizando tanto sus capacidades como sus defectos y carencias para ser mejoradas en un futuro.

1.4. Estructura del documento

El resto de este documento se estructura de la siguiente forma. El capítulo 2 describe el proceso de planificación del proyecto, junto con la estimación de su presupuesto. El capítulo 3 introduce el estado del arte relevante al proyecto. El capítulo 4 presenta los conocimientos necesarios para comenzar el desarrollo del sistema. El capítulo 5 describe el proceso de análisis y diseño del simulador propuesto, y el capítulo 6 las características principales de su implementación. El capítulo 7 describe las pruebas realizadas para validar el simulador. Finalmente, el capítulo 8 presenta las conclusiones del proyecto y el posible trabajo futuro.

Capítulo 2

Planificación y presupuesto del proyecto

En este capítulo se detallan los siguientes aspectos:

- Análisis de tareas a realizar durante el desarrollo el proyecto.
- La metodología de desarrollo seguida.
- El análisis de riesgos y sus respectivas contingencias.
- El coste económico estimado para el proyecto.

2.1. Análisis de tareas

El desarrollo de un simulador es una tarea compleja, ya que tales sistemas constan de numerosas etapas y subsistemas para su correcto funcionamiento. Dependiendo de la amplitud del sistema a simular o el grado de detalle con el que se pretende simular, la cantidad de estas etapas y subsistemas puede incrementarse de forma considerable. Es necesario un exhaustivo conocimiento del sistema a simular, así como de los subsistemas o funciones en los que se puede dividir. En la planificación del desarrollo de un simulador se debe tener en cuenta el desarrollo de estos subsistemas como tareas independientes. En este proyecto se identifican correctamente estos subsistemas, asignando una tarea al desarrollo de cada uno.

El proyecto se planifica para ser desarrollado en aproximadamente 300 horas, a lo largo de un periodo aproximado de año y medio. Durante este año y medio, el desarrollador trabajará en el proyecto a media jornada. Esto se debe a que el desarrollador compaginará el desarrollo del proyecto con actividades académicas universitarias, por un valor de 90 créditos ECTS en total; y con labores de investigación en un grupo de investigación de la Universidad de Valladolid, cumpliendo una beca de colaboración de alrededor de 630 horas totales.

Para el desarrollo del prototipo de simulador planteado, se identifican las siguientes tareas a realizar:

T1.- Contextualización: estudio de las características principales de las arquitecturas de computadores que se desean simular.

T2.- Elección de la arquitectura concreta a simular.

T3.- Estudio exhaustivo de la arquitectura a simular.

T4.- Estudio de simuladores previos y sus subsistemas.

T5.- Diseño del sistema. Elección de la aproximación concreta al problema: tipo y subsistemas del simulador.

T6.- Elección y despliegue del entorno de trabajo y conjunto de herramientas de desarrollo de software a utilizar.

T7.- Desarrollo de la estructura básica del simulador.

- T7.1.- Desarrollo de las interfaces software.
- T7.2.- Desarrollo de la simulación del banco de registros.
- T7.3.- Desarrollo de la simulación de la memoria del computador.
- T7.4.- Desarrollo de los elementos y estructuras básicas de un programa.

T8.- Análisis y elección de las llamadas al sistema que soporte el simulador.

T9.- Desarrollo de la simulación de la ejecución de instrucciones ensamblador.

T10.- Desarrollo del compilador de ficheros fuente.

- T10.1.- Elección de la sintaxis de los programas.
- T10.2.- Análisis y elección de características adicionales permitidas en los programas.
- T10.3.- Análisis e implementación de las estructuras de compilación necesarias.

T11.- Desarrollo del flujo de ejecución de programas. Análisis e implementación del *pipeline* segmentado.

T12.- Desarrollo de la interfaz gráfica de usuario.

- T12.1.- Análisis y elección de las tecnologías software existentes para el desarrollo de interfaces gráficas.
- T12.2.- Análisis e implementación de los componentes gráficos necesarios en el simulador.

T13.- Pruebas de validación del simulador.

T14.- Desarrollo del manual de usuario.

T15.- Pruebas de usabilidad con usuarios reales.

- T15.1.- Diseño de las pruebas.

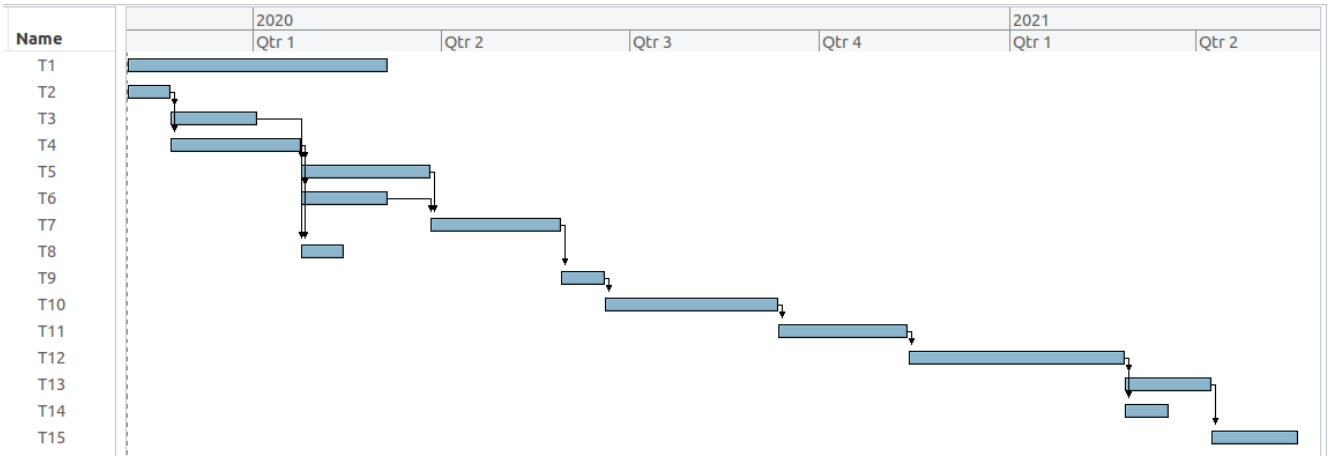

Figura 2.1: Diagrama de Gantt del proyecto.

T15.2.- Ejecución de las pruebas y recogida de retroalimentación.

T15.3.- Modificación del software en base a la retroalimentación obtenida.

El diagrama de Gantt [18] para la organización temporal tentativa del desarrollo de las tareas se muestra en la figura 2.1.

2.1.1. Camino crítico

El camino crítico de un proyecto es el conjunto de tareas en las que, si alguna de ellas sufre un retraso, se retrasa el proyecto total.

Partiendo del diagrama de Gantt del proyecto mostrado en la figura 2.1, el camino crítico de este proyecto se muestra en la figura 2.2.

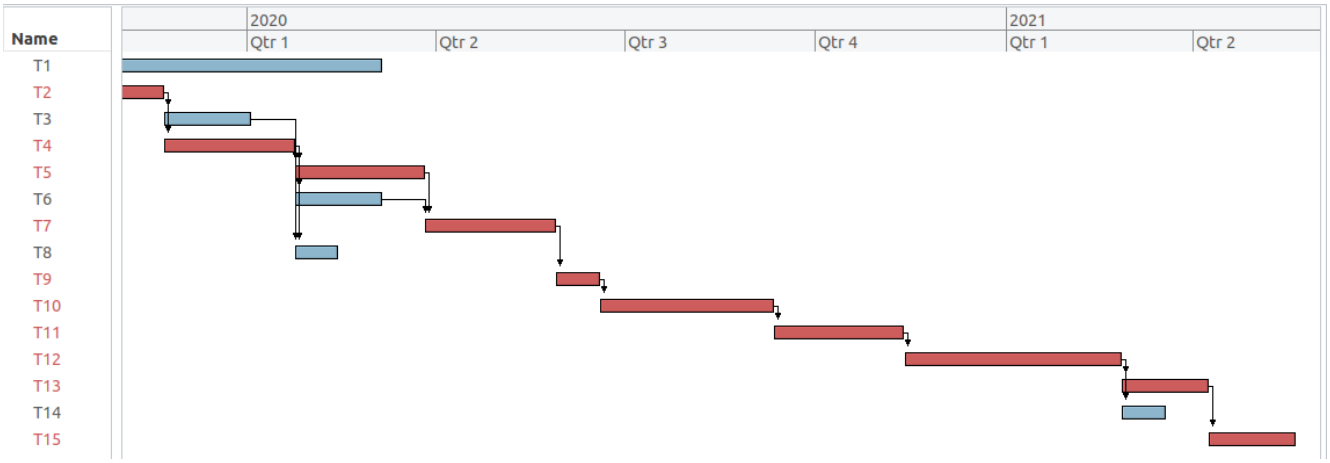

Figura 2.2: Diagrama de Gantt del proyecto, remarcando en rojo el camino crítico del mismo.

Dentro del camino crítico del proyecto se incluyen las siguientes tareas:

T2.- Elección de la arquitectura a simular.

T4.- Estudio de simuladores previos y sus subsistemas.

T5.- Diseño del sistema. Elección de la aproximación concreta al problema: tipo y subsistemas del simulador.

T7.- Desarrollo de la estructura básica del simulador.

T9.- Desarrollo de la simulación de la ejecución de instrucciones ensamblador.

T10.- Desarrollo del compilador de ficheros fuente.

T11.- Desarrollo del flujo de ejecución de programas. Análisis e implementación del *pipeline* segmentado.

T12.- Desarrollo de la interfaz gráfica de usuario.

T13.- Pruebas de validación del simulador.

T15.- Pruebas de usabilidad con usuarios reales.

La tarea T2, de elección de la arquitectura a simular, forma parte del camino crítico ya que debe decidirse qué se va a simular antes de comenzar el desarrollo del simulador. La tarea T4 también debe realizarse antes del comienzo del desarrollo del simulador, pues ayuda a sentar las bases de cómo podría organizarse el sistema. Una vez conocida la estructura de otros simuladores, ha de realizarse la tarea T5 de diseño de nuestro simulador. Las tareas T7, T9, T10, T11 y T12 comprenden todo el proceso de implementación incremental del simulador, por lo que también forman parte del camino crítico. Finalmente, las tareas T13 y T14 comprenden las fases de prueba y validación del software, formando las últimas tareas a realizar en el proyecto, y formando parte del camino crítico.

2.2. Metodología de desarrollo

Para el desarrollo del proyecto se sigue una metodología de tipo ágil [19]. Esta decisión se ha tomado debido a que el proyecto presenta una complejidad elevada y solo se cuenta con un desarrollador trabajando en el mismo.

Concretamente, durante el desarrollo del software del proyecto se sigue una metodología principalmente de programación extrema [20], pero con ciertas características más afines a metodologías de tipo *Scrum* [21]. De este modo, el proyecto se desarrolla por *sprints* de corta duración: una o dos semanas generalmente. Para cada *sprint* se definen unos objetivos tentativos. Al final de cada *sprint* se realiza una reunión con los tutores académicos para discutir el avance del proyecto. En estas reuniones se exponen las nuevas funcionalidades añadidas, así como los problemas surgidos, y se definen los objetivos para el siguiente *sprint*.

La utilización de metodologías ágiles permite la reacomodación de objetivos y tareas a realizar sobre la marcha. Esto resulta de gran utilidad a la hora de solventar los problemas y riesgos que surjan durante el desarrollo del proyecto.

2.3. Riesgos y contingencias

La planificación de un proyecto software es una tarea de gran relevancia y repercusión en el resto del desarrollo proyecto. Una planificación incorrecta que ignore los riesgos existentes en el proyecto puede retrasar los plazos de finalización del mismo, incrementando significativamente sus costes. Es necesario realizar un análisis de los riesgos que se pueden presentar durante el desarrollo del proyecto, junto con el desarrollo de planes de contingencia para mitigarlos y minimizar su impacto.

Para este proyecto, se ha realizado un análisis de riesgos basado en las tareas y el alcance del proyecto. La tabla 2.1 presenta los riesgos detectados. Se incluye una estimación de la probabilidad de que ocurran, el impacto que tendrían en el proyecto, y el riesgo total que suponen para el mismo, denominado índice de exposición del riesgo [22]. A las probabilidades se les asigna un valor entre 0 (imposible que ocurra) y 10 (se está seguro de que va a ocurrir). Al impacto se le asigna un valor entre 0 (no repercute en el proyecto) y 10 (el proyecto no puede continuar). El riesgo total se calcula como el producto de la probabilidad y el impacto del riesgo. Los riesgos con valores totales más altos son los más importantes, por su potencial influencia en el proyecto.

Descripción del riesgo	Probabilidad	Impacto	Riesgo
Desarrollo técnicamente demasiado complejo	5	4	20
Documentación del sistema a simular insuficiente	3	5	15
Estimación incorrecta del tiempo de aprendizaje de una tecnología software	6	5	30
Estimación incorrecta del tiempo de corrección de errores tras las pruebas de validación	5	4	20
Cambio en los requisitos durante el desarrollo del proyecto	3	6	18

Tabla 2.1: Riesgos del proyecto.

Junto a la detección de riesgos, se han desarrollado planes de contingencia para mitigar sus efectos en caso de materializarse. La tabla 2.2 resume los planes de contingencia desarrollados para cada riesgo detectado.

Como se ha mencionado en la sección 2.2, como contingencia general para los riesgos se ha adoptado una metodología de trabajo ágil. Esto permite la reacomodación de objetivos y tareas sobre la marcha, en caso de que se materializaran los riesgos o surgieran imprevistos durante el desarrollo.

Uno de los riesgos más importantes detectados es que el desarrollo sea técnicamente demasiado complejo. Como plan de contingencia, si durante el desarrollo del simulador se desconoce cómo debería implementarse cierta funcionalidad concreta, se consultará la implementación de otros simuladores para esa funcionalidad y considerará la viabilidad de utilizar alguna implementación similar.

La posible estimación incorrecta del tiempo de aprendizaje de una tecnología software es el

Descripción del riesgo	Plan de contingencia
Desarrollo técnicamente demasiado complejo	Consultar la implementación de otros simuladores como referencia
Documentación del sistema a simular insuficiente	Implementar el funcionamiento de las características no documentadas como decida el desarrollador, indicando y documentando esta decisión.
Estimación incorrecta del tiempo de aprendizaje de una tecnología software	Acotar los objetivos del proyecto, dividiendo el desarrollo del simulador en dos proyectos.
Estimación incorrecta del tiempo de corrección de errores tras las pruebas de validación	Limitar las características del simulador. Documentar los errores no corregidos, y considerar el sistema en estado de "beta".
Cambio en los requisitos durante el desarrollo del proyecto	Aumentar las horas de trabajo semanales para alcanzar los nuevos objetivos.

Tabla 2.2: Planes de contingencia concretos para los riesgos del proyecto detectados.

mayor riesgo del proyecto. Como plan de contingencia, para las tareas que requieren el aprendizaje de una tecnología nueva, si se detecta que este aprendizaje puede extender considerablemente el tiempo de desarrollo del proyecto, se plantea la posibilidad de dividir el desarrollo del simulador en dos partes: la primera parte constaría del desarrollo de las funcionalidades de simulación (*backend*), y la segunda del desarrollo de la interfaz (*frontend*), contando con las pruebas de usabilidad. Esta contingencia abre la posibilidad de que otro alumno del grado en Ingeniería Informática de la Universidad de Valladolid pueda realizar su Trabajo de Fin de Grado finalizando el desarrollo del simulador.

2.4. Desviación de la planificación original

En esta sección describimos la planificación ejecutada finalmente durante el desarrollo del proyecto, destacando las modificaciones realizadas sobre la planificación original. Esta sección contiene información sobre sucesos que han ocurrido cronológicamente después de la fase de planificación del proyecto. Se incluyen en este capítulo y no en ningún otro ya que guardan una relación más estrecha con los temas relativos a la planificación del proyecto que con cualquiera de los otros temas tratados en el resto del documento.

A la hora de desarrollar el proyecto algunos de los riesgos previstos se materializaron (ver tablas 2.1 y 2.2). En concreto, la documentación del sistema a simular resultó ser de difícil acceso e interpretación, lo que produjo que el tiempo de dedicación estimado para el estudio de la arquitectura elegida fuese insuficiente.[1]. También, se subestimaron los tiempos necesarios para aprender las tecnologías utilizadas durante el desarrollo del proyecto, especialmente las requeridas para desarrollar la interfaz gráfica del simulador.

[1]En el apéndice C se explica de manera más detallada el problema existente con la documentación del sistema a simular.

A los riesgos descritos en el párrafo anterior se añadió la materialización de un riesgo no previsto: la pandemia de la COVID-19 y sus consecuencias a nivel de condiciones de salud y trabajo. La pandemia de la COVID-19 afectó especialmente al desarrollo del proyecto, impidiendo realizar tantas reuniones de seguimiento con los tutores académicos como se habían planificado inicialmente. También dificultó las condiciones en las que estas se desarrollarían. La disminución en la cantidad y calidad de las reuniones con los tutores académicos agravó las consecuencias de la materialización de los otros riesgos, llevando a la necesidad de poner en marcha planes de contingencia más contundentes.

Como plan de acción ante los riesgos materializados descritos, se decide tomar las siguientes medidas:

- Desarrollar un prototipo de interfaz gráfica en vez de una completamente funcional. Solo se requeriría una interfaz lo suficientemente funcional para poder realizar una demostración de las capacidades del simulador. (Modificación de la tarea T14).
- Desarrollar un resumen de manual de usuario, acorde al prototipo de interfaz desarrollado. (Modificación de la tarea T14).
- Posponer las pruebas de usabilidad con usuarios reales. Estas podrían realizarse el primer cuatrimestre del curso académico 2021-2022, con alumnos de la asignatura Arquitectura y Organización de Computadores [5]. (Supresión de la tarea T15).

Con estas medidas, el proyecto podría desarrollarse hasta un estado estable, requiriendo tan solo de ligeras modificaciones y adiciones para poder comenzar la fase final de validación. Se estima que dichas modificaciones y adiciones podrían completarse durante la primera mitad del tercer cuatrimestre de 2021.

2.5. Presupuesto del proyecto

En este apartado se presenta un análisis del presupuesto estimado para el proyecto. El coste asociado al mismo se puede dividir en dos categorías principales: las horas de trabajo del desarrollador y los tutores, y la amortización de las máquinas de trabajo utilizadas para desarrollar el proyecto. No ha sido necesaria la adquisición de licencias de software para el desarrollo del proyecto.

Para realizar la estimación del presupuesto del proyecto, se han estimado los siguientes costes:

- Sueldo del desarrollador: se estima que el sueldo de un ingeniero informático junior es de unos 20 000 € brutos anuales. Suponiendo una jornada completa de 8 horas y diarias, y 250 días laborales al año aproximadamente, se obtiene un coste del desarrollador de 10 € la hora.
- Sueldo de los tutores académicos: se estima el sueldo de un profesor titular de escuela universitaria en unos 30 000 € brutos anuales, y el sueldo de un profesor ayudante doctor en

unos 27 000 € brutos anuales[2]. Suponiendo una jornada completa de 8 horas diarias, y 250 días laborales al año, se obtiene un coste de los tutores académicos de 15 € y 13.5 € la hora, respectivamente.

- Coste de las máquinas de desarrollo: durante el desarrollo del proyecto se han utilizado dos máquinas distintas:
  - El ordenador personal de sobremesa del desarrollador. Se trata de un ordenador hecho a piezas, encargado y personalizado por el desarrollador. Este ordenador ha sido utilizado por el desarrollador para trabajar en el proyecto desde casa. Fue comprado en junio de 2017 por un precio de 1144,65 € y una estimación de la vida útil de 5 años. El coste asociado a la amortización de este material de trabajo es de 0,16 € la hora.
  - El ordenador portátil del desarrollador. Se trata de un HP Pavilion comprado en diciembre de 2013 por un precio de 599,00 €. La estimación de su vida útil fue de 4 años. El coste asociado a la amortización de este material de trabajo es de 0,10 € la hora.

Las horas de utilización de cada recurso se proyectan de la siguiente forma:

- Desarrollador: 450 horas; 300 asociadas a los 12 créditos de la asignatura del Trabajo de Fin de Grado, y 150 a la realización de dos publicaciones de carácter investigador relacionadas con el trabajo desarrollado en el proyecto.
- Tutores académicos: 60 horas cada uno, asociadas a las reuniones que se realizarán semanalmente como mínimo, durante todo el desarrollo del proyecto. La duración de estas reuniones se estima en 45 minutos.
- Máquinas del desarrollador: se utilizan a partes iguales durante todo el proyecto. El ordenador de sobremesa se utiliza para trabajar desde casa, y el ordenador portátil para trabajar en la facultad. Por tanto, 225 horas cada uno.

Siguiendo estas condiciones, el coste del trabajo personal y de uso de las máquinas en el proyecto se describe en las tablas 2.3 y 2.4

Personal	Coste/hora (€)	Horas	Total
Alumno	10	450	4500
Tutor TEU	15	60	900
Tutor AYUDOC	13.5	60	810
Total			6210

Tabla 2.3: Coste de trabajo de personal del proyecto.

El coste total del proyecto, calculado a partir del coste de las máquinas y de las horas de trabajo del personal, se indica en la tabla 2.5

[2]El sueldo real depende de los trienios y quinquenios de los tutores académicos.

Máquina	Precio (€)	Amortización (años)	Coste/hora (€)	Horas	Total (€)
Ordenador de sobremesa	1144,65	5	0,16	225	36
Ordenador portátil	599,00	5	0,10	225	22,5
Total					58,5

Tabla 2.4: Coste de uso de las máquinas en el proyecto.

Actividad	Coste (€)
Horas de trabajo del personal	6210
Uso de las máquinas	58,5
Total	**6268,5**

Tabla 2.5: Coste total del proyecto.

2.6. Resumen

En este capítulo se ha detallado la planificación del proyecto. En esta se incluyen las tareas a realizar, los posibles riesgos y sus planes de contingencia, y la descripción del camino crítico del proyecto. También se incluye una sección dedicada a la descripción de los riesgos que finalmente se materializaron durante el desarrollo del proyecto, y cómo afectaron a la planificación inicial. Para terminar, se realiza una estimación del presupuesto del proyecto, basado en el tiempo de desarrollo del mismo.

Capítulo 3

Estado del arte

En este capítulo se detalla el estado del arte sobre las áreas de estudio en las que se centra el proyecto:

- El estado del arte de la arquitectura de computadores.
- La aproximación pedagógica actual al estudio de la arquitectura de computadores.
- Los simuladores de computadores didácticos actuales.

3.1. Arquitecturas de computadores

Existen multitud de arquitecturas de computadores distintas, remontándose las primeras arquitecturas modernas a la década de 1960. A día de hoy, existen dos tipos principales de arquitecturas:

- Arquitecturas **CISC** (*Complex Instruction Set Computer*, arquitectura de conjunto de instrucciones complejo). Estas arquitecturas incluyen instrucciones muy complejas que pueden ejecutar múltiples operaciones de bajo nivel de forma individual. Por ejemplo, carga de memoria, operación aritmética, y almacenamiento en memoria, todo en la misma instrucción.
  La filosofía de los computadores CISC se centra en diseñar una gran cantidad de instrucciones que realicen operaciones muy específicas. Estas instrucciones suelen tomar más de un ciclo de reloj en ser ejecutadas. Este diseño pretende acercar las operaciones realizables por las instrucciones ensamblador de los computadores a construcciones de lenguajes de programación de alto nivel (llamadas a procedimientos, control de bucles, acceso a *arrays* y estructuras...). El diseño hardware de los computadores CISC es relativamente denso y complejo.
- Arquitecturas **RISC** (*Reduced Instruction Set Computer*, computador de conjunto de instrucciones reducido). Estas arquitecturas trabajan con conjuntos de instrucciones pequeños. Surgieron como intento por simplificar los diseños de los computadores CISC.
  Las instrucciones de los computadores RISC realizan operaciones más generales que las de

los computadores CISC, requiriendo a menudo múltiples instrucciones RISC para realizar la operación equivalente de un computador CISC. No obstante, los computadores RISC suelen acercarse a la ejecución de una instrucción por ciclo de reloj.
El reducido tamaño del conjunto de instrucciones soportado por los computadores RISC simplifica su diseño hardware, así como la codificación de las instrucciones. Del mismo modo, los compiladores pueden generar código máquina que utiliza óptimamente todo un conjunto de instrucciones RISC. En el caso de los conjuntos de instrucciones CISC, es común que los compiladores no sepan generar código que aproveche todas las instrucciones disponibles.

Actualmente, dados los avances en el desarrollo de computadores RISC y CISC, así como los avances en el desarrollo de los compiladores, la distinción entre CISC y RISC en arquitecturas de uso comercial se ha vuelto algo difusa. De forma general, aunque con excepciones, la distinción se suele realizar en base a si en el conjunto de instrucciones de la arquitectura se incluyen instrucciones aritméticas que hacen accesos a memoria (CISC), o no (RISC). Hoy en día existen procesadores RISC más complejos que algunos considerados históricamente CISC. Se ha alcanzado un punto en el desarrollo de procesadores en el que ambos términos se suelen utilizar de forma subjetiva. En este documento se utilizan estos términos para clasificar las arquitecturas según la complejidad de su filosofía de diseño inicial, sin importar la evolución de sus características a lo largo de la historia. En el ámbito docente, esta distinción es adecuada para categorizar la dificultad de estudio de una arquitectura.

La arquitectura de computadores más utilizada en ordenadores personales hoy en día es x86 [23]. x86 fue desarrollada en 1978, inicialmente por Intel en exclusiva, aunque AMD también ha obteniendo licencias para desarrollar la arquitectura. Se trata de una arquitectura de tipo CISC, lo que la hace difícil de estudiar. Es por eso que no suele estudiarse en asignaturas básicas de Arquitectura de Computadores. No obstante, consideramos pertinente mencionarla por su amplio uso cotidiano.

ARM [24] es otra arquitectura de gran relevancia actualmente. Se utiliza de forma muy habitual en sistemas empotrados, aunque su uso se está extendiendo cada vez más a sistemas personales y de supercomputación. Por ejemplo, *Fugaku*, el supercomputador más potente a junio de 2021, utiliza la arquitectura ARM. En el ámbito cotidiano, algunas familias de dispositivos que utilizan habitualmente ARM son *smartphones*, *tablets*, videoconsolas portátiles, y similares. A fecha de 2021, más de 180 mil millones de chips ARM han sido producidos, siendo la arquitectura más utilizada y mayormente producida [25]. Se trata de una arquitectura de tipo RISC, por lo que es conceptualmente más simple que x86, y su estudio en una asignatura de fundamentos de computadores resulta más asequible.

También, destacamos MIPS [26] y RISC-V [27]. MIPS es una arquitectura RISC que tuvo gran relevancia en el pasado, sobre todo en la década de 1990, influenciando notablemente el diseño de arquitecturas RISC posteriores. Debido a dicha influencia, es muy común que universidades y escuelas técnicas elijan estudiar esta arquitectura en asignaturas de Arquitectura de Computadores. Sin embargo, hoy en día es una arquitectura obsoleta y su relevancia en el mercado tecnológico de los últimos años es muy baja. En marzo de 2021, la empresa MIPS anunció que el desarrollo de la arquitectura se había descontinuado, y que realizarían una transición al desarrollo de, precisamente, la arquitectura RISC-V.

RISC-V es una arquitectura RISC desarrollada por la Universidad de California, Berkeley.

RISC-V ha sido distribuido bajo licencias de código abierto, por lo que su uso en cualquier contexto no requiere el pago de ninguna tasa. Esta arquitectura fue diseñada originalmente en 2010 con propósitos de investigación y didácticos, relacionados con las arquitecturas RISC. Sin embargo, su uso en la actualidad se ha extendido hasta el ámbito comercial, y aunque no sean muchos, ya existen dispositivos de uso cotidiano basados en esta arquitectura.

3.2. Aproximación pedagógica a la Arquitectura de Computadores

Los doctores David A. Patterson y John L. Hennessy son dos de los autores más importantes en el campo de la docencia de arquitectura de computadores. Ambos son profesores científicos de computación y dos de los principales pioneros y contribuidores a las arquitecturas RISC. David A. Patterson acuñó el término "RISC", y ha sido uno de los principales desarrolladores de la arquitectura RISC-V. John L. Hennessy es el cofundador de *MIPS Computer Systems Inc.*. Ambos profesores son coautores de dos libros sobre arquitectura de computadores ampliamente utilizados en ámbitos docentes, desde la década de 1990 hasta hoy, como base para asignaturas que tratan esos temas.

El que se titula *Computer Organization and Design - The Hardware/Software Interface* [28], conocido comúnmente como *Patterson and Hennessy*, es especialmente relevante en cursos de iniciación a la arquitectura de computadores. Dicho libro, que actualmente cuenta con 9 ediciones, ha sido reeditado para tres arquitecturas distintas: MIPS, ARM y RISC-V [28–30], por orden de publicación. Por ello, estas tres arquitecturas se son algunas de las más estudiadas en universidades de todo el mundo, contando así con un importante respaldo académico.

Las tres versiones del libro *Patterson and Hennessy* son esencialmente la misma, excepto por el conjunto de instrucciones ensamblador que utilizan para construir los ejemplos descriptivos. El libro trata los conceptos más fundamentales de la Arquitectura de Computadores:

- Qué es y cómo funciona un computador.
- El lenguaje ensamblador.
- Cómo realizan los computadores operaciones aritméticas.
- Fundamentos de implementación de procesadores.
- Jerarquías de memoria.
- Procesadores paralelos.

El libro hace especial hincapié en la importancia de la eficiencia de los computadores; principalmente en la productividad (entendiéndose como operaciones realizadas por un computador por unidad de tiempo), aunque también en la eficiencia energética. En las secciones finales de cada capítulo, se suelen analizar varias arquitecturas reales y comercialmente utilizadas. En estas secciones se describe cómo las arquitecturas comerciales implementan las características que se han

tratado durante el capítulo. De esta forma, se detalla cómo los conceptos estudiados en el libro se reflejan en la realidad.

El libro escrito por PATTERSON y HENNESSY es ampliamente utilizado en todo el mundo como base para las asignaturas que traten los fundamentos de arquitectura de computadores. Probablemente se trate del libro más utilizado en tales contextos. La bibliografía de PATTERSON y HENNESSY es frecuentemente referenciada en el estado del arte centrado en este área de conocimiento. Otros libros pueden ser utilizados para tratar contextos más específicos de forma más detallada (por ejemplo, redes de computación o multiprocesadores), pero ninguno ha alcanzado la fama y repercusión de *Computer Organization and Design - The Hardware/Software Interface*.

En la Universidad de Valladolid, por ejemplo, la 5ª edición de *Computer Organization and Design - The Hardware/Software Interface* es utilizada como material de consulta básico en las asignaturas Fundamentos de Computadores, Arquitectura y Organización de Computadores, y Arquitecturas de Computación Avanzadas [4–6].

3.3. Computación de Alto Rendimiento

En los últimos años, en el contexto de HPC se ha impulsado la utilización y desarrollo de arquitecturas heterogéneas. Estas arquitecturas están compuestas por varios tipos distintos de computadores. Por una parte, están los procesadores de propósito general, las unidades centrales de procesamiento (CPUs), utilizadas en la mayoría de sistemas informáticos como ordenadores personales, portátiles y smartphones. Por otra parte, están los coprocesadores específicos utilizados como aceleradores hardware, como las unidades de procesamiento gráfico (GPUs), matrices de puertas lógicas programables en campo (FPGAs), o circuitos integrados para aplicaciones específicas (ASICs).

En oposición a las CPUs convencionales, que tratan de ser dispositivos versátiles con capacidad de resolver una amplia gama de problemas computacionales, los aceleradores hardware solo se centran en la resolución cierto tipo de problemas. Por ejemplo: las unidades de procesamiento gráfico (GPUs) están diseñadas para trabajar de manera eficiente con imágenes o *arrays* de píxeles. Limitando el rango de problemas que pretenden resolver, los aceleradores pueden obtener una muy alta eficiencia computacional a la hora de resolver tales problemas. De forma ilustrativa, la figura 3.1 muestra una comparación del rendimiento de una CPU y una GPU para el mismo problema.

La arquitectura de los aceleradores hardware puede distar considerablemente de la arquitectura tradicional de las CPUs, por lo que su programación resulta una tarea compleja. Algunos aceleradores hardware como las GPUs y FPGAs, requieren del uso sistemas de ejecución y compilación específicos, como *CUDA* [31] o *oneAPI* [32], respectivamente. Estos sistemas, además de proporcionar los medios de comunicación entre las CPUs y los aceleradores, proporcionan los métodos de compilación de código de alto nivel a código máquina específico compatible con las arquitecturas de los aceleradores. Los requisitos necesarios para utilizar estos aceleradores y sus respectivos entornos de ejecución a menudo convierten el proceso de desarrollo para los mismos en una tarea compleja y tediosa, desalentando su uso frecuente.

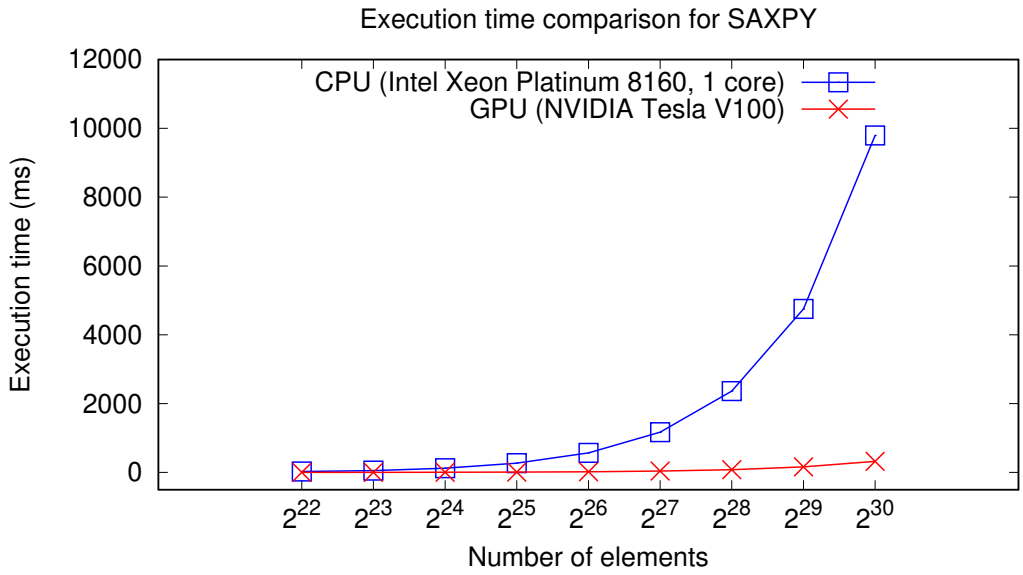

Figura 3.1: Comparación del tiempo de ejecución del algoritmo SAXPY en una CPU (azul) y una GPU (rojo). El eje X representa el número de elementos del vector sobre el que trabaja el algoritmo, como potencia de 2. El eje Y representa el tiempo en milisegundos que se tarda en ejecutar el algoritmo. Las pruebas se realizaron en la máquina *Manticore* del grupo investigación Trasgo de la Universidad de Valladolid.

Desde hace años, es habitual que los comerciantes de los aceleradores ofrezcan métodos de simulación o emulación para facilitar el proceso de desarrollo para estos dispositivos. Estos sistemas de simulación permiten validar los programas desarrollados para los aceleradores hardware sin la necesidad de utilizar o configurar todo el entorno de ejecución asociado, facilitando las tareas de desarrollo y depuración de software para los aceleradores.

3.4. Simuladores de código ensamblador existentes

Los fabricantes de microprocesadores suelen desarrollar simuladores que facilitan el desarrollo y la depuración de software para sus microprocesadores específicos. Estos simuladores se enfocan en chips específicos, en vez de en familias de arquitecturas a nivel general; y se enfocan principalmente en el desarrollo a nivel profesional o personal, y no en el ámbito didáctico o docente.

La arquitectura MIPS presenta la mayor cantidad de simuladores de propósito didáctico. Para ella existen, entre otros: *QtSPIM*, *MARS*, *QtMIPS*, *WebMIPS* y *WepSIM* [13, 33–35]. Los dos últimos son plataformas web interactivas, presentando gran portabilidad, aunque perdiendo cierta usabilidad. Por otra parte, algunos de ellos permiten la simulación del *pipeline* clásico RISC de 5 etapas, presentada por Hennessy y Patterson en el libro mencionado en la sección 3.2, añadiendo interés didáctico.

De entre todos los simuladores de MIPS existentes, *MARS* es probablemente el más popular en universidades de todo el mundo. *MARS-F*, por su parte, es una modificación de *MARS* desarrollada por el doctor Francisco José Andújar Muñoz, añadiendo funcionalidades de planificación estática y dinámica al simulador base. *MARS-F* es utilizado como material de apoyo en la asignatura Arquitecturas de Computación Avanzadas [6] del grado en Ingeniería Informática de la Universidad de Valladolid. En el apéndice D se tratan de forma más detallada las características de *MARS*.

La arquitectura ARM, pese a su popularidad de uso, no presenta gran cantidad de simuladores de uso didáctico. Destacamos el simulador *ARMSim#* [36], basado en un microprocesador ARM7, así como el simulador *QtARMSim* [37] para la arquitectura ARM Thumb 2. Estas arquitecturas, aunque aún utilizadas, no son las más actuales que ARM presenta. Los simuladores para arquitecturas más actuales, como ARMv8, o para alguna sub-arquitectura derivada, presentan varios defectos. La escasa cantidad de simuladores existentes para estas arquitecturas se presentan como pequeños proyectos de prueba de concepto, o no están preparados para una cómoda utilización por terceros.

Trabajos previos han tomado otra aproximación a la enseñanza de las arquitecturas ARMv8, utilizando las populares placas Raspberry Pi como entorno de ejecución de programas ensamblador, sustituyendo simuladores software [38]. Aunque factible, consideramos esta solución demasiado compleja como para ser adoptada de forma generalizada, y no pensamos que sea un sustituto eficiente de un buen simulador. Otras universidades han optado por realizar modificaciones al simulador *MARS*, haciéndolo compatible con una sub-arquitectura ARMv8 [39], aunque no hemos logrado localizar el software resultante en ninguna página web.

En el caso de RISC-V, sí que se pueden encontrar más simuladores funcionales de uso didáctico; quizás por su naturaleza inherentemente educativa. Aun así, no son tantos como los que existen para MIPS. Destacamos *BRISC-V Simulator* de ASCS Laboratory [40], y *Venus* [41], ambos simuladores web. Ninguno de los dos es especialmente sofisticado, ni alcanzan la facilidad de uso de los simuladores de MIPS. Existen otras alternativas para esta arquitectura, aunque se trata de intérpretes de binarios precompilados, y no de programas textuales en código ensamblador. Esto dificulta su uso didáctico y haciéndolos más apropiados para otros contextos.

Analizando estos simuladores mencionados, se han observado varias carencias que ninguno de ellos solventa por completo:

- Los simuladores más utilizados son de arquitecturas ya obsoletas.
- Los simuladores de arquitecturas más modernas son más limitados y difíciles de utilizar.
- Los simuladores más completos no son aptos para su uso didáctico.
- La aproximación de los simuladores a los *pipelines* de los procesadores es inexistente o carente de flexibilidad.
- El esfuerzo que tomaría modificar cualquiera de los simuladores para añadirle nuevas características es demasiado grande.

Algunas de estas carencias suponen una limitación importante a la hora de utilizar estos simuladores con propósito didáctico, sirviendo tan solo para representar los conceptos teóricos más básicos estudiados en las asignaturas.

3.5. Resumen

En este capítulo se ha estudiado el estado actual del campo de la Arquitectura de Computadores; así como la aproximación que se toma en las universidades para su estudio, y los simuladores disponibles a utilizar en las asignaturas que se dedican a tal estudio.

Las conclusiones del capítulo son las siguientes:

- Dentro de las arquitecturas de computadores existentes, las de tipo RISC son las más apropiadas para ser sujeto de estudio en asignaturas que traten los fundamentos de la arquitectura de computadoras. En ese campo, destaca especialmente el trabajo de los doctores David A. Patterson y John L. Hennessy, ambos coautores de libros pedagógicos que han sido ampliamente utilizados como base para asignaturas sobre Arquitectura de Computadores en universidades de todo el mundo.
- En muchas asignaturas sobre Arquitecturas de Computadores se imparten clases prácticas como apoyo a la docencia teórica. Para impartir estas clases prácticas se hace uso de simuladores de ciertas arquitecturas. Uno de los simuladores más populares es *MARS*, de la arquitectura MIPS. Esta arquitectura está ya obsoleta, por lo que convendría utilizar simuladores de arquitecturas RISC más relevantes en la actualidad. Ejemplos de estas arquitecturas son ARMv8 o RISC-V. A día de hoy, los simuladores existentes de estas últimas arquitecturas presentan limitaciones considerables de cara a su uso didáctico.

Capítulo 4

Conocimientos previos

La primera tarea del proyecto consiste en la familiarización con las características de las arquitecturas de computadores relevantes para el simulador que se desea desarrollar (tarea T1 detalladas en la sección 2.1), así como con la arquitectura específica a simular (tareas T2 y T3 detalladas en la sección 2.1). Esto implica realizar un análisis de las características de interés comunes a todas las arquitecturas consideradas para el simulador, así como un estudio exhaustivo de la arquitectura específica que se simulará, antes de comenzar el desarrollo del simulador en sí mismo.

Como consecuencia de dicha tarea de análisis se ha realizado un minucioso estudio de las materias de interés para definir las características del simulador. Algunas de los temas que conforman estas materias son lo suficientemente complejos como para requerir de un capítulo específico en el que se detallen, de forma que se pueda comprender de forma correcta el desarrollo del proyecto. En este capítulo se exponen los contenidos que conforman los conocimientos previos necesarios para entender el desarrollo del proyecto:

- Las arquitecturas de computadores de tipo RISC.
- La técnica de segmentación de instrucciones en procesadores.
- La arquitectura ARM LEGv8.

4.1. Arquitecturas RISC

RISC, del inglés *Reduced Instruction Set Computer* (computador con conjunto de instrucciones reducido) es un tipo de arquitectura de computadores que engloba a todas las arquitecturas que presentan un conjunto pequeño de instrucciones. Se presentan como alternativa a las arquitecturas de tipo CISC (*Complex Instruction Set Computer*, computador con conjunto de instrucciones complejas), que presentan una gran cantidad de instrucciones muy especializadas.

El término RISC data de 1980, acuñado por David A. Patterson, profesor de la Universidad de California, Berkeley [42]. Patterson lidero el proyecto *Berkeley RISC*, que trataba de

diseñar microprocesadores más rápidos a partir de la utilización de conjuntos de instrucciones más sencillos que los utilizados hasta entonces. Al mismo tiempo, se desarrolló el proyecto *MIPS* de la universidad de Stanford, con objetivos similares. Además de proponer una simplificación de los conjuntos de instrucciones y del diseño de los procesadores, a partir de ambos proyectos se determinó que para desarrollar procesadores más rápidos, estos debían contar con un mayor número de registros, cuya lectura o escritura resulta más rápida que la lectura o escritura de una memoria global.

Los resultados de los proyectos mencionados inspiraron el diseño de gran cantidad de arquitecturas de computadores posteriores: las arquitecturas RISC. Todas estas arquitecturas siguen las bases comunes de los computadores RISC:

- Arquitecturas de carga y almacenamiento. Estas arquitecturas dividen las instrucciones en dos tipos: las que acceden a la memoria, y las que realizan operaciones aritmético-lógicas sobre registros. Esto implica que los computadores no pueden trabajar directamente sobre valores almacenados en memoria, sino que deben ser previamente cargados en un registro, y almacenados después.

- Presencia de registros de propósito general que pueden utilizarse tanto de fuente como de destino en cualquier operación. Esta característica ayuda a simplificar el diseño de los compiladores.

    - Pueden existir ficheros de registros separados dedicados a los operandos de cierto subconjunto de instrucciones concreto. Suele ser el caso de los registros de coma flotante y los registros vectoriales.

- Formato de instrucciones simple, de tamaño fijo y uniforme, con códigos de operación breves y generalmente de tamaño fijo también. Esta característica simplifica la lógica de búsqueda, decodificación y preparación de los computadores.

- Distintos formatos de instrucciones, utilizados para diferenciar las instrucciones según las tareas que realizan: operaciones aritmético-lógicas, cargas y almacenamientos de memoria, control del flujo del programa, etc...

- Modos de direccionamiento de memoria simples. Los direccionamientos complejos se realizan mediante secuencias de múltiples instrucciones.

- Soporte hardware tan solo para tipos de datos simples: bytes, medias palabras, palabras, dobles palabras... El soporte para tipos de datos más complejos, como las cadenas de caracteres, se debe gestionar a través del software.

- Rendimiento de los computadores cercano a 1 instrucción por ciclo de reloj (en teoría).

Estas características centradas en la simplicidad de diseño, sobre todo en lo que al conjunto de instrucciones se refiere, convierten a las arquitecturas RISC en arquitecturas muy populares en los contextos docentes. Como se ha visto en el capítulo 3, en las asignaturas universitarias sobre Arquitectura de Computadores se suele estudiar el diseño de las arquitecturas RISC por su baja complejidad, y por su extenso uso en el mundo actual.

4.2. Segmentación de instrucciones

La segmentación de instrucciones o *pipelining* es una técnica utilizada en Arquitectura de Computadores que implementa paralelismo a nivel de instrucción en los procesadores. Consiste en dividir la ejecución de las instrucciones en un procesador en varias etapas secuenciales, conocidas en conjunto como *pipeline*. Las instrucciones avanzan de etapa en etapa para ser ejecutadas, teniendo cada etapa una función distinta. Cada ciclo de reloj del procesador se ejecuta una de las etapas de cada instrucción en el *pipeline*, avanzando a la siguiente. Como en cada etapa se utiliza solo una parte del hardware del procesador, pueden ejecutarse varias instrucciones a la vez, siempre y cuando estén en etapas distintas.

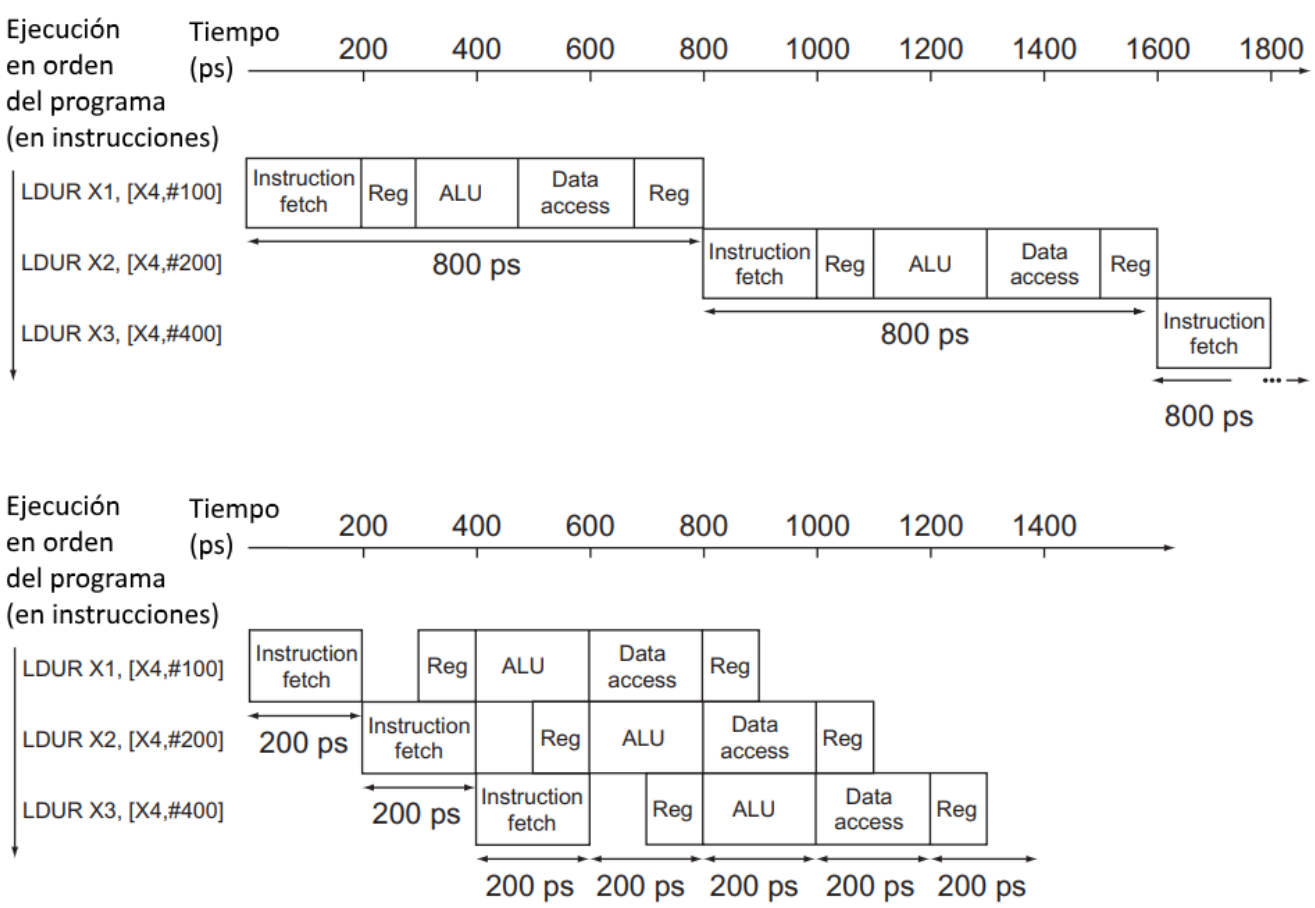

Figura 4.1: Incremento en la productividad de los procesadores a través de la segmentación de instrucciones: ejecución en un ciclo sin *pipeline* (arriba) versus ejecución con *pipeline* (abajo). Extraído de [28]. El procesador de arriba ejecuta una instrucción por ciclo de reloj, con un periodo de reloj de 800ps. El procesador de abajo ejecuta una instrucción en 5 ciclos de reloj, con un periodo de reloj de 200ps. Pese a que la ejecución de una única instrucción tarde más en finalizar en el procesador de abajo que en el de arriba, el procesador de abajo obtiene una productividad mayor al poder ejecutar múltiples instrucciones en paralelo.

Idealmente, todas las etapas del *pipeline* estarían ocupadas ejecutando alguna instrucción. De ser así, el procesador alcanzaría una productividad de una instrucción por ciclo. En teoría, cuantas más etapas tenga un *pipeline*, menor puede ser el periodo de reloj del procesador. Es decir, el procesador sería más rápido. Por tanto, la segmentación de instrucciones trata de incrementar la

productividad de los procesadores disminuyendo el periodo de reloj de los mismos y manteniendo una tasa de ejecución de instrucciones de una instrucción por ciclo. La figura 4.1 ilustra este concepto.

La segmentación de instrucciones es una técnica utilizada por la práctica totalidad de los procesadores actuales, debido a su relativa sencillez de implementación y considerable ganancia en productividad. Esta utilización masiva la convierte en una técnica fundamental en el ámbito de la Arquitectura de Computadores; tanto es así que se suele estudiar en las asignaturas que tratan los conceptos fundamentales de Arquitectura de Computadores de las universidades de todo el mundo.

Cada procesador puede implementar su *pipeline* de forma distinta, variando él número y la función de las etapas. La conocida como "*pipeline* clásico RISC", explicada en el libro de PATTERSON y HENNESSY e implementada por la arquitectura LEGv8, consta de 5 etapas:

1. Búsqueda de instrucciones (abreviada *IF* por *Instruction Fetch*): la siguiente instrucción a ejecutar se busca en la memoria de instrucciones y se lleva al hardware del procesador.

2. Decodificación de instrucciones (abreviada *ID* por *Instruction Decode*): la instrucción obtenida de la etapa anterior es decodificada. Esto activa las señales necesarias en el procesador para la ejecución de la instrucción. Además, se leen los operandos de la instrucción del banco de registros del procesador.

3. Ejecución de instrucciones (abreviada *EX* por *EXecution*): la Unidad Aritmético-Lógica (ALU a partir de ahora) realiza la operación indicada por la instrucción sobre los operandos de la misma, generando un resultado.

4. Acceso a memoria (abreviada *M* o *MEM*): las instrucciones que lo requieran, leen o escriben datos de memoria en esta fase.

5. Escritura de resultados (abreviada *WB* por *Write Back*): el resultado producido o el dato leído de memoria se escribe en el correspondiente registro del banco de registros.

La figura 4.2 ilustra la ejecución de instrucciones en esta *pipeline*.

el *pipeline* clásico RISC puede parecer muy simplista, pero es suficiente para ilustrar los conceptos básicos de la segmentación de instrucciones en gran detalle. Los procesadores reales pueden implementar *pipelines* más o menos extensos, dependiendo del compromiso que los fabricantes estén dispuestos a adoptar en cuanto a productividad y costes de producción.

4.2.1. Riesgos en *pipelines* y técnicas para resolverlos

La segmentación de instrucciones no está exenta de complicaciones. Existen situaciones en las que una instrucción no puede comenzar su ejecución hasta que otra anterior haya finalizado todas sus etapas. Estas situaciones son conocidas como "riesgos". En un procesador segmentado se pueden dar tres tipos distintos de riesgos:

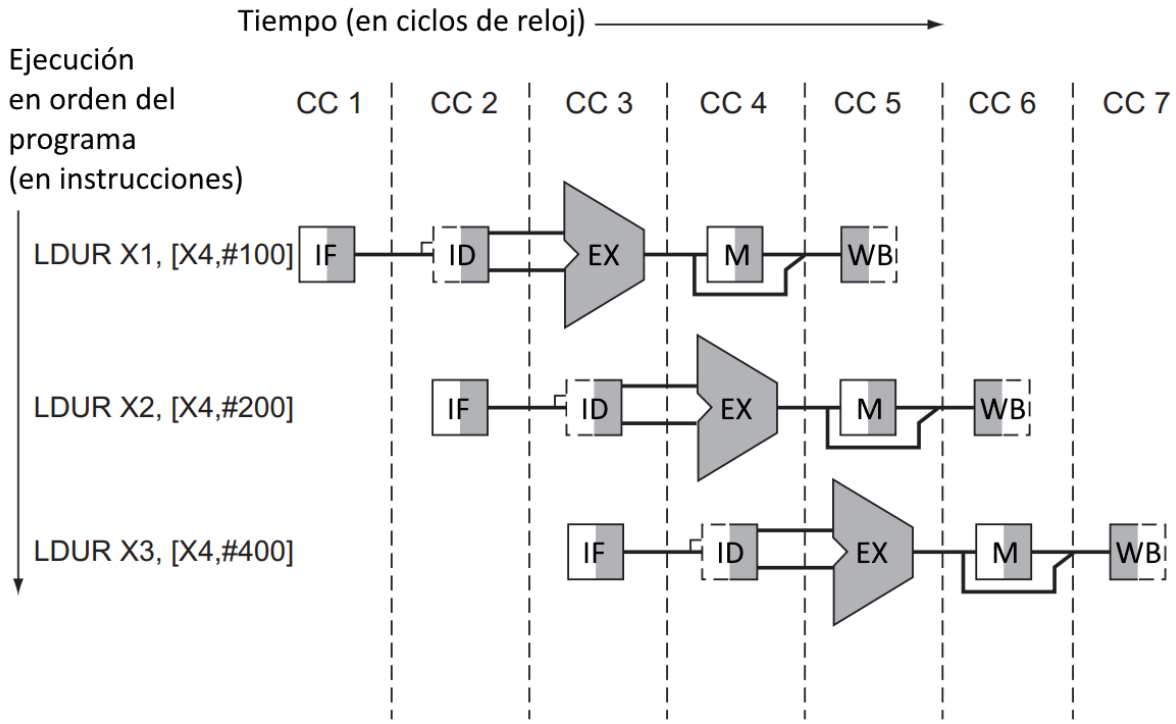

Figura 4.2: Ejecución de tres instrucciones LEGv8 en el *pipeline* clásico RISC. Extraído de [28]. En la figura se indica en qué etapa del *pipeline* se encuentra cada instrucción durante los distintos ciclos de reloj que toma su ejecución completa.

- Riesgos estructurales: se dan cuando no hay hardware suficiente para ejecutar las distintas etapas de dos o más instrucciones específicas en el mismo ciclo de reloj. Es decir, se producen cuando dos o más instrucciones requieren hacer uso del mismo recurso hardware en el mismo ciclo de reloj. En procesadores sencillos, estos riesgos no deberían producirse nunca. Sin embargo, la introducción de instrucciones complejas como las de multiplicación o división tienden a producir estos riesgos de manera inevitable.

- Riesgos de datos: estos se producen cuando una o varias instrucciones deben usar como operandos los resultados de una instrucción que todavía no ha finalizado su ejecución. Hay varios tipos de riesgos de datos. Los más comunes son los riesgos de tipo *RAW* (*read after write*): una instrucción debe leer un dato antes de que una instrucción anterior lo haya producido. Utilizando el *pipeline* clásico RISC como ejemplo: la ejecución de instrucciones se produce en la etapa 3 del *pipeline*, pero la escritura de resultados no se produce hasta la 5. Supongamos que tenemos una instrucción aritmética que hace uso del resultado producido por la instrucción inmediatamente anterior. Cuando se deba ejecutar dicha instrucción (etapa 3), la anterior todavía no habrá escrito su resultado en el banco de registros (etapa 5), ya que todavía se hallará en la etapa 4. Este clase de riesgos se ilustra en el ejemplo del código 4.1 y la figura 4.3.

- Riesgos de control: esta clase de riesgos se da con instrucciones que alteran el flujo de control del programa: saltos o bifurcaciones. Las bifurcaciones condicionales son aquellas en las que solo se produce la bifurcación (cambio de flujo) si se cumple una condición, generalmente basada en la comparación anterior de dos valores. Con las instrucciones de bifurcación condicional es posible que la instrucción inmediatamente después no sea la siguiente que deba ejecutarse. Además, esa información probablemente no se conozca en la primera etapa del

```
SUB X2, X1, X3        // Registro X2 escrito por SUB.
AND X12, X2, X5       // Primer operando (X2) depende de SUB.
OR X13, X6, X2        // Segundo operando (X2) depende de SUB.
ADD X14, X2, X2       // Primer y segundo operando dependen de SUB.
STUR X15, [X2, #100] // Base de la carga de memoria (X2) depende de SUB.
```

Fragmento de código 4.1: Fragmento de código ensamblador LEGv8 que ilustra los riesgos de datos en computadores segmentados. Todas la primera instrucción (SUB) genera un resultado (X2) del que dependen todas las demás instrucciones. Esta dependencia es causa de problemas en computadores segmentados, ya que algunas de las instrucciones podrían requerir leer el dato antes de que la primera instrucción escriba su valor en el registro de resultado. La figura 4.3 ilustra este problema en el *pipeline* clásico RISC.

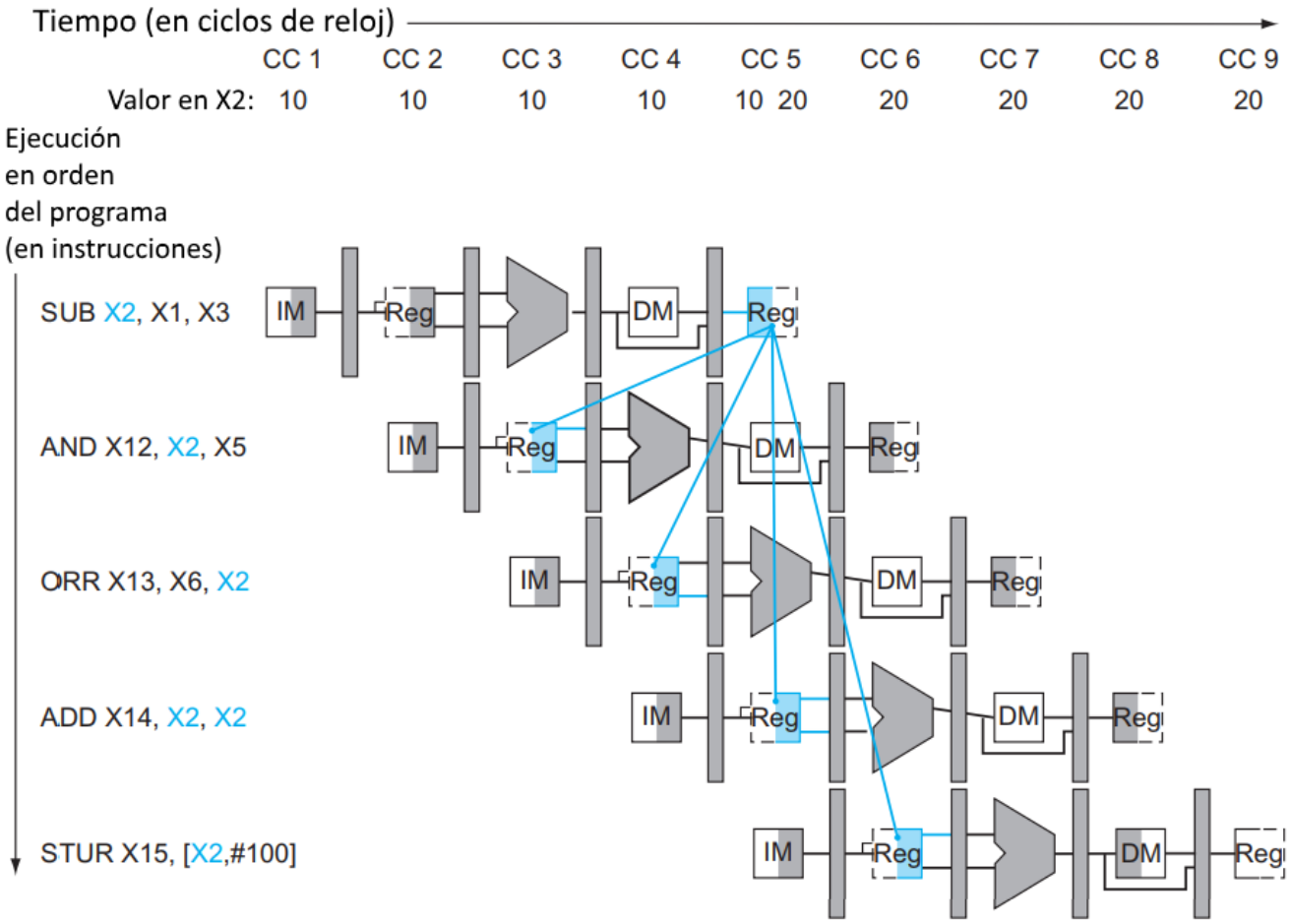

Figura 4.3: Ilustración de las dependencias de datos del fragmento de código LEGv8 presentado en el código 4.1 al ejecutarse en el *pipeline* clásico RISC de 5 etapas. Extraído de [28]. La figura muestra en orden cronológico la ejecución de las 5 instrucciones, destacando las dependencias de datos con color azul, tanto en las propias instrucciones como en el diagrama de etapas de ejecución de las instrucciones. "CC n" en la parte superior de la figura indica el n-ésimo ciclo de reloj de la ejecución. Puede observarse que el resultado de la primera instrucción (SUB) no se escribe hasta el ciclo de reloj 5, pero es requerido en los ciclos de reloj 3, 4, 5 y 6. Las dependencias en los ciclos 6 e incluso 5 no presentan ningún problema (los registros son escritos en la primera mitad de una etapa y leídos en la segunda mitad, pudiendo leerse un resultado el mismo ciclo que se ha producido). Las dependencias que van atrás en el tiempo son riesgos de datos en el *pipeline*.

pipeline. En el tiempo que se tarda en decidir si la bifurcación es tomada o no, más instrucciones podrían entrar en el *pipeline*, potencialmente de forma incorrecta.
Además, también es posible que se desconozca la dirección de memoria de la instrucción a la que se debe bifurcar hasta después de la primera etapa del *pipeline*. Es el caso, por ejemplo, en instrucciones que bifurcan a la dirección apuntada por un registro (bifurcaciones indirectas). Esto causa problemas en el caso de las bifurcaciones incondicionales también: aunque se sabe cuál es la dirección que debe tomar el flujo del programa, no es posible llevar la siguiente instrucción al procesador en el siguiente ciclo de reloj.

En general, los riesgos impiden continuar la ejecución de instrucciones en el procesador de forma correcta. Es necesario buscar soluciones a estos riesgos para que los procesadores segmentados puedan funcionar.

La solución más simple para los riesgos consiste en detener el flujo de instrucciones del procesador, dejando de iniciar la ejecución de instrucciones hasta que todas las anteriores hayan finalizado. Este fenómeno es conocido como *burbuja*. La lógica y circuitería necesarias para implementar esta solución son mínimas. Sin embargo, también es la solución más indeseable de todas. Teniendo en cuenta que se desea una productividad de 1 instrucción ejecutada por ciclo, esta solución es indeseable ya que impide la ejecución de instrucciones de forma temporal. Aun así, en algunas ocasiones es la única solución factible para asegurar la correcta ejecución de un programa en el procesador.

Para resolver los riesgos de datos se suele aplicar la técnica denominada *anticipación*. Esta técnica consiste en añadir circuitería al procesador para que los resultados producidos en cada etapa (si hubiera) estén disponibles en las etapas anteriores. De este modo, las siguientes instrucciones no tendrían que esperar a la escritura de los resultados para ser ejecutadas; solo a que se produzca el resultado. Dependiendo del *pipeline* del procesador y del conjunto de instrucciones utilizado por el mismo, esta técnica puede no ser aplicable a todas las secuencias de instrucciones. No obstante, es una técnica relativamente sencilla de implementar en los procesadores, e idealmente mantiene la productividad de 1 instrucción ejecutada por ciclo. Es por eso que se trata de una técnica ampliamente utilizada en los procesadores segmentados.

En el caso de los riesgos de control, numerosas técnicas han sido desarrolladas a lo largo de los años para paliar sus efectos. Las relacionadas con bifurcaciones condicionales generalmente implican el uso de algún predictor de saltos. Los predictores de saltos permiten continuar la ejecución del programa por alguna de las vías alternativas, esperando que la elegida sea la correcta. Cuando se confirma la dirección de la bifurcación, en una etapa del *pipeline* posterior a en la que se hizo la predicción, la predicción se resuelve como correcta o incorrecta. De ser la predicción incorrecta, se deben descartar todas las instrucciones que han entrado a el *pipeline* desde que se hizo la predicción, obteniendo la misma productividad que se habría obtenido deteniendo el *pipeline*. De ser la predicción correcta, se continua la ejecución tal y como se estaba haciendo, obteniendo la máxima productividad posible en esa bifurcación. Los predictores de saltos pueden ser más o menos complejos, siendo los más sencillos los predictores estáticos: predecir siempre que la bifurcación se toma o que la bifurcación no se toma. La discusión de otros tipos de predictores queda fuera del ámbito de este documento.

Otra técnica utilizada para paliar los riesgos de control se denomina “bifurcación retardada”. Consiste en continuar la ejecución de las instrucciones directamente después de la instrucción de

bifurcación hasta que se resuelva la propia bifurcación. Esta técnica confía en que el programador o el compilador reordene las instrucciones del programa de tal forma que se conserve la corrección del mismo. Las ventajas de esta técnica son que no requiere la modificación del hardware o la lógica del computador, y que la productividad obtenida es teóricamente la máxima alcanzable (al no producirse ninguna detención en ningún caso). La principal desventaja es la confianza en una entidad externa para reordenar correctamente las instrucciones del programa, tarea que en ocasiones resulta imposible. Es muy común que no se disponga de instrucciones útiles que se puedan reordenar para ser ejecutadas después de la instrucción de bifurcación, manteniendo la corrección del programa. En estos casos es necesario introducir instrucciones que no hacen nada (*nops*), produciendo resultados equivalentes a detener el flujo de instrucciones.

Los riesgos y sus respectivas soluciones expuestos en esta sección son esenciales a la hora de trabajar con procesadores segmentados. Comprenderlos es fundamental para entender las complicaciones que acarrea la segmentación de instrucciones. Es por ello que suelen conformar parte del temario relacionado con segmentación de instrucciones en todas las asignaturas universitarias que tratan conceptos fundamentales de Arquitectura de Computadores.

4.3. Arquitectura ARM LEGv8

LEGv8, (*Lessen Extrinsic Garrulity version 8*, "disminuir la garrulidad extrínseca versión 8"), es una micro-arquitectura derivada de la arquitectura AArch64 [43]. AArch64 a su vez es el modo de 64 bits de la arquitectura ARMv8. (Ver el apéndice C para más información sobre las versiones de ARM).

LEGv8 fue desarrollado por Hennessy y Patterson para ser utilizada con motivos pedagógicos en su libro *Computer Organization and Design ARM edition: The Hardware/Software Interface*. Dicho libro, en conjunto con la arquitectura LEGv8, fue publicado en 2016, y trataba de actualizar las ediciones anteriores del mismo, en las que se utilizaba la arquitectura MIPS. El enfoque que se tomó a la hora de diseñar la arquitectura fue, en esencia, elegir un subconjunto de instrucciones ensamblador de Aarch64 lo más similar posible a la arquitectura MIPS usada en las ediciones anteriores. Se tomó este enfoque para minimizar el número de cambios a realizar en el resto del libro. En definitiva, podría verse como una versión de MIPS de 64 bits, adaptada a la sintaxis e instrucciones de ARMv8.

La decisión de utilizar la versión de 64 bits de ARM y no alguna de las múltiples de 32 bits que existen se basó en una encuesta realizada a los profesores y alumnos de una facultad donde se utilizaba una edición anterior del libro como material docente: el 75 % de los encuestados o bien preferían cambiar a una versión con direcciones de memoria más grandes (que 32 bits), o bien les era indiferente.

Las características principales de la arquitectura LEGv8 son:

- Arquitectura de diseño RISC y de tipo registro-registro.
- 31 registros de 64 bits de propósito general, y un registro que almacena la constante cero. Además, puede contar con 32 registros opcionales para el almacenamiento de datos en

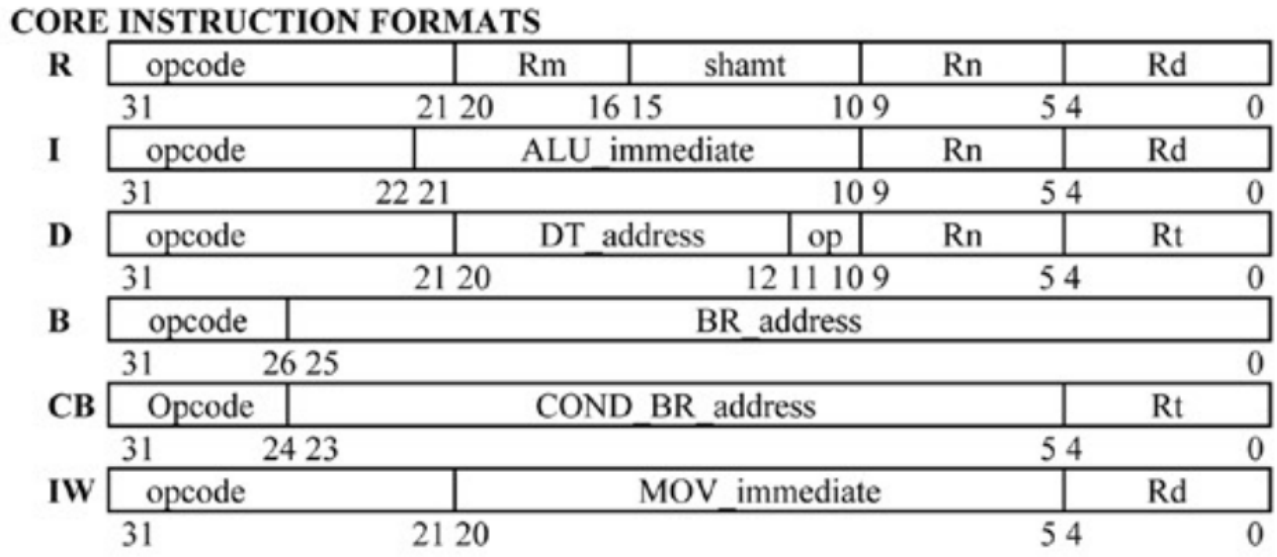

Figura 4.4: Codificación de los distintos formatos de instrucción presentes en la arquitectura LEGv8. Extraído de [28]. *Opcode* es el código de operación. *Rn*, *Rm*, *Rt* y *Rd* son los registros que participan en la ejecución de la instrucción, tanto los operandos como el destino. *shamt* se utiliza en conjunto con el código de operación para distinguir las instrucciones de tipo R. Los distintos campos que terminan en *address* son las direcciones de memoria, relativas o absolutas, sobre las que trabajan las distintas instrucciones. Los campos que terminan en *immediate* son los valores numéricos constantes sobre los que trabajan las instrucciones de tipo I o IW.

formato coma flotante de doble precisión.

- Espacio de direcciones de 64 bits, con palabras de 32 bits (4 bytes), permitiendo el direccionamiento de un total de 2^{62} palabras.
- Instrucciones ensamblador de 32 bits, con código de operación de longitud variable.
- Múltiples formatos de instrucción. Los formatos de instrucción de LEGv8 son los siguientes:
  - Formato R: para instrucciones aritmético-lógicas que trabajan exclusivamente sobre registros.
  - Formato I: para instrucciones aritmético-lógicas que trabajan sobre registros y valores numéricos constantes (inmediatos).
  - Formato D: para instrucciones de transferencia de datos entre registros y memoria.
  - Formato B: para instrucciones de bifurcación (salto) incondicionales.
  - Formato CB: para instrucciones de bifurcación (salto) condicionales.
  - Formato IW o IM: para instrucciones que trabajan con valores numéricos constantes (inmediatos) de gran tamaño.

La figura 4.4 muestra la codificación de los distintos formatos de instrucciones. La tabla 4.1 muestra todas las instrucciones básicas que conforman la arquitectura y su extensión aritmética.

Instrucción LEGv8	Mnemónico	Formato
Suma	ADD	R
Resta	SUB	R
Suma inmediato	ADDI	I
Resta inmediato	SUBI	I
Suma y cambia las banderas	ADDS	R
Resta y cambia las banderas	SUBS	R
Suma inmediato y cambia las banderas	ADDIS	R
Resta inmediato y cambia las banderas	SUBIS	R
Carga un registro (8 bytes)	LDUR	D
Almacena un registro (8 bytes)	STUR	D
Carga una palabra con signo (4 bytes)	LDURSW	D
Almacena una palabra (4 bytes)	STURW	D
Carga media palabra (2 bytes)	LDURH	D
Almacena media palabra (2 bytes)	STURH	D
Carga un byte	LDURB	D
Almacena un byte	STURB	D
Carga un registro, en exclusión	LDXR	D
Almacena un registro, en exclusión	STXR	D
Mover inmediato grande a registro, sobrescribiendo	MOVZ	IM
Mover inmediato grande a registro, manteniendo otros bits	MOVK	IM
Máscara and	AND	R
Máscara or inclusiva	ORR	R
Máscara or exclusiva	EOR	R
Máscara and con inmediato	ANDI	I
Máscara or inclusiva con inmediato	ORRI	I
Máscara or exclusiva con inmediato	EORI	I
Desplazamiento lógico a la izquierda	LSL	R
Desplazamiento lógico a la derecha	LSR	R

Comparar y bifurcar si igual a 0	CBZ	CB
Comparar y bifurcar si distinto a 0	CBNZ	CB
Bifurcar condicionalmente	B.cond	CB
Bifurcar	B	B
Bifurcar a registro	BR	R
Bifurcar con enlace	BL	B

Instrucción LEGv8 (extensión aritmética)	**Mnemónico**	**Formato**
Multiplicar	MUL	R
Multiplicar con signo, parte alta	SMULH	R
Multiplicar sin signo, parte alta	UMULH	R
Dividir con signo	SDIV	R
Dividir sin signo	UDIV	R
Suma de coma flotante, precisión simple	FADDS	R
Resta de coma flotante, precisión simple	FSUBS	R
Multiplicación de punto flotante, precisión simple	FMULS	R
División de coma flotante, precisión simple	FDIVS	R
Suma de coma flotante, precisión doble	FADDD	R
Resta de coma flotante, precisión doble	FSUBD	R
Multiplicación de punto flotante, precisión doble	FMULD	R
División de coma flotante, precisión doble	FDIVD	R
Comparación de punto flotante, precisión simple	FCMPS	R
Comparación de punto flotante, precisión doble	FCMPD	R
Carga de coma flotante, precisión simple	LDURS	D
Carga de coma flotante, precisión doble	LDURD	D
Almacenamiento de punto flotante, precisión simple	STURS	D

Almacenamiento de punto flotante, precisión doble	STURD	D

Tabla 4.1: Conjunto de instrucciones completo de LEGv8. Esta tabla incluye las instrucciones básicas y las instrucciones de la extensión aritmética.

Una característica de ARMv8 que no estaba presente en la arquitectura MIPS y que LEGv8 sí que incorpora son las banderas (*flags*) o códigos de condición. Se trata de 4 bits en la CPU cuyos valores se ponen a 1 o a 0 dependiendo de los resultados producidos por ciertas instrucciones específicas. Las 4 banderas de condición de la arquitectura son:

- Negativo (N): la instrucción ha producido como resultado un número negativo (es decir, con el bit más significativo igual a 1).
- Cero (Z): la instrucción ha producido como resultado el número cero.
- Desbordamiento (V): la instrucción ha producido desbordamiento en su resultado.
- Acarreo (C): la instrucción ha producido un acarreo por su bit más significativo.

Las instrucciones LEGv8 que modifican las banderas de condición son:

- Las instrucciones del subconjunto básico cuyo mnemónico termina en *S*: ADDS, ADDIS, SUBS y SUBIS.
- Las instrucciones de comparación de la extensión aritmética de la arquitectura: FCMPS y FCMPD.

La instrucción de bifurcación condicional `B.cond` se vale de combinaciones de estos códigos de condición para determinar si una bifurcación se debe tomar o no. De esta forma, la arquitectura LEGv8 implementa más bifurcaciones condicionales de las que se incluyen en una arquitectura MIPS de forma nativa. Las posibles bifurcaciones condicionales derivadas de la instrucción `B.cond` junto con los valores de las banderas que las hacen bifurcar son:

- Bifurcar si igual (`B.EQ`). La bifurcación se toma cuando Z vale 1.
- Bifurcar si distinto (`B.NE`). La bifurcación se toma cuando Z vale 0.
- Bifurcar si menor, en aritmética con signo (`B.LT`). La bifurcación se toma cuando el valor de N es distinto del de V.
- Bifurcar si menor o igual, en aritmética con signo (`B.LE`). La bifurcación se toma cuando Z vale 1 o el valor de N es distinto del de V.
- Bifurcar si mayor, en aritmética con signo (`B.GT`). La bifurcación se toma cuando Z vale 0 o N vale lo mismo que V.

- Bifurcar si mayor o igual, en aritmética con signo (`B.GE`). La bifurcación se toma cuando N vale lo mismo que V.
- Bifurcar si menor, en aritmética sin signo (`B.LO` o `B.CC`). La bifurcación se toma cuando C vale 0.
- Bifurcar si menor o igual, en aritmética sin signo (`B.LS`). La bifurcación se toma cuando C vale 0 o Z vale 1.
- Bifurcar si mayor, en aritmética con signo (`B.HI`). La bifurcación se toma cuando C vale 1 y Z vale 0.
- Bifurcar si mayor o igual, en aritmética con signo (`B.HS` o `B.CS`). La bifurcación se toma cuando C vale 1.
- Bifurcar si negativo (`B.MI`). La bifurcación se toma cuando N vale 1.
- Bifurcar si positivo (`B.PL`). La bifurcación se toma cuando N vale 0.
- Bifurcar si ha habido desbordamiento (`B.VS`). La bifurcación se toma cuando V vale 1.
- Bifurcar si no ha habido desbordamiento (`B.VC`). La bifurcación se toma cuando V vale 0.
- Bifurcar siempre (`B.AL` o `B.NV`). Estas dos variantes de `B.cond` siempre bifurcan. Existen para dar una funcionalidad a todas las codificaciones posibles de la instrucción `B.cond`: La condición `cond` de la instrucción se codifica con 4 bits, dando lugar a 16 códigos de condición posibles. 14 de ellos se utilizan para las 14 variantes de la instrucción vistas en los anteriores puntos. Los dos restantes se agrupan en esta variante.

Haciendo uso de esta sencilla arquitectura y conjunto de instrucciones ensamblador, Hennessy y Patterson ilustran magistralmente los conceptos fundamentales subyacentes en la arquitectura de computadores. Principalmente, la arquitectura LEGv8 adquiere una mayor relevancia en los capítulos dedicados a la presentación del lenguaje ensamblador y el diseño de procesadores segmentados (capítulos 2 y 4 respectivamente). También presenta cierta importancia ilustrativa en los capítulos dedicados a la aritmética de los procesadores y las arquitecturas vectoriales (capítulos 3 y 5 respectivamente). A lo largo del libro son comunes las comparaciones de LEGv8 con MIPS, por ser la arquitectura utilizada hasta entonces para las distintas ediciones del libro. También son frecuentes las comparativas con ARMv8, dejando entrever la complejidad de la arquitectura completa de la que surgió LEGv8.

Queda claro de esta forma que la arquitectura LEGv8 es una buena elección para realizar una aproximación inicial a la arquitectura de computadores, ya que combina la sencillez de MIPS con la relevancia actual de ARM. Es por ello por lo que hemos decidido que el simulador desarrollado tenga como arquitectura objetivo LEGv8.

4.4. Resumen

En este capítulo se han estudiado tres conceptos fundamentales a la hora de entender el proceso de desarrollo del simulador: las bases sobre las que se fundamentan las arquitecturas RISC, la

técnica de segmentación de instrucciones, y la arquitectura LEGv8.

Las principales conclusiones del capítulo son las siguientes:

- Las arquitecturas RISC se caracterizan por la relativa simpleza de su diseño, especialmente de su conjunto de instrucciones. Estas instrucciones trabajan sobre registros, y se dividen en distintos tipos según su funcionalidad. Las instrucciones se dividen principalmente en aritmético-lógicas, de escritura-lectura de memoria, o de bifurcación. Debido a su simpleza, es común que se estudien las arquitecturas RISC en asignaturas de fundamentos de Arquitectura de Computadores.
- Los procesadores suelen implementar varias etapas de ejecución de instrucciones, denominadas en conjunto *pipeline*. Varias instrucciones se pueden ejecutar a la vez en la misma *pipeline*, pero hay que aplicar diversas técnicas de control para que su ejecución sea correcta.
- LEGv8 es una arquitectura de computadores RISC desarrollada por Hennessy y Patterson. Combina elementos de MIPS y ARMv8, y se desarrolló para su utilización con objetivo didáctico en el libro *Computer Organization and Design ARM edition: The Hardware/Software Interface*. Esta combinación da lugar a una arquitectura sencilla de estudiar pero con relevancia actual, lo que la convierte en una candidata perfecta a la hora de desarrollar un simulador didáctico.

Capítulo 5

Análisis y diseño del simulador

En este capítulo se presentan los distintos aspectos relacionados con el diseño del software del proyecto:

- Requisitos del sistema.
- Arquitectura del sistema.
- Diseño de los módulos del sistema
- Casos de uso

5.1. Análisis de requisitos

La primera etapa del desarrollo del software del simulador consiste en la especificación de requisitos del sistema. Para realizar esta etapa es necesario realizar previamente el análisis del estado del arte y el estudio de los conocimientos previos, detallados en los capítulos 3 y 4 respectivamente. El estudio de los conocimientos previos es necesario para definir requisitos funcionales del sistema: qué debe hacer el simulador, más concretamente, qué aspectos de una arquitectura debe simular. El análisis del estado del arte, en especial de los simuladores previamente existentes, es necesario para definir otros requisitos funcionales y algunos no funcionales: cómo se debe realizar la simulación, qué información de la simulación se presenta al usuario y cómo. Los requisitos derivados del análisis de otros simuladores principalmente tienen dos objetivos:

- Identificar aspectos de otros simuladores que se consideran de utilidad y quieren replicarse. En especial, funcionalidades que aportan una utilidad relevante al simulador y cuya implementación presenta una especial facilidad de uso, en los contextos de herramienta docente.
- Identificar aspectos de otros simuladores que se pretenden mejorar. En concreto, funcionalidades que se consideran de utilidad pero su implementación en otros simuladores no se considera correcta; o funcionalidades inexistentes en otros simuladores que se consideran de utilidad.

La especificación de requisitos en base a las funcionalidades observadas en otros simuladores tiene como riesgo asociado la posibilidad de definir más requisitos funcionales de los que se puedan cumplir durante el desarrollo previsto del proyecto. Como este proyecto trata de desarrollar un prototipo, es decir, realizar el desarrollo inicial de una herramienta de propósito docente, con la posibilidad de ampliarse y mejorarse según se vaya utilizando en los sucesivos cursos académicos, se plantearon una serie de requisitos tentativos deseados para el simulador, por orden de prioridad. Para los requisitos que no se pudieran llegar a cumplir, se facilitaría su futuro cumplimiento en la medida de lo posible y se dejarían como trabajo futuro. Para asegurar una calidad mínima en este proyecto, una serie de requisitos serían considerados esenciales, y no se daría por finalizado el mismo hasta que no se llegaran a cumplir. En definitiva, una versión definitiva del simulador debe cumplir todos los requisitos especificados en esta sección. El prototipo desarrollado en este proyecto tan solo debe cumplir los requisitos considerados esenciales, y facilitar el cumplimiento de todos los demás en la medida de lo posible.

5.1.1. Requisitos funcionales

A continuación se detalla la lista de requisitos funcionales planteados para el simulador:

RF1.- El sistema debe ser capaz de simular la ejecución de programas textuales escritos en ensamblador ARM LEGv8.

RF2.- El sistema debe permitir la simulación en "modo interactivo", proporcionando una interfaz gráfica al usuario en la que se presente toda la información de la simulación.

RF3.- El sistema debe ser capaz de proporcionar información al usuario sobre el contenido de los registros del computador simulado.

RF4.- El sistema debe ser capaz de proporcionar información al usuario sobre el contenido de la memoria del computador simulado.

RF5.- El sistema debe ser capaz de proporcionar información al usuario sobre las etiquetas de memoria especificadas en el código ensamblador.

RF6.- El sistema debe proporcionar información sobre la compilación/ensamblado de los programas ensamblador. Esta información debe incluir los fallos sintácticos hallados en el código, haciendo referencia a la línea de código fuente donde se encuentren. También debe incluir los resultados de las diferentes etapas de compilación:

a) Programa preprocesado que traduce pseudo-instrucciones, constantes, etiquetas y otros símbolos por sus valores reales.

b) Programa en código máquina.

RF7.- El sistema debe soportar el uso de directivas de preprocesador que permitan la cómoda especificación de los datos de los programas en su código fuente.

RF8.- El sistema debe ser capaz de simular pseudo-instrucciones ARM LEGv8 , tanto las indicadas en la especificación de la arquitectura en [28], como otras definibles por el usuario.

RF9.- El sistema debe ser capaz de simular la funcionalidad de "segmentación de instrucciones" de manera similar a como se describe en el capítulo cuatro de *Computer Organization and Design - The Hardware/Software Interface*.

RF10.- El sistema debe ser capaz de simular la funcionalidad de "bifurcación retardada" utilizada en múltiples computadores comerciales.

RF11.- El sistema debe ser capaz de simular *pipelines* genéricas con unidades funcionales multiciclo, similares a las que se describen en el *Computer Architecture: A Quantitative Approach* [44].

RF12.- El sistema debe permitir la simulación de distintas arquitecturas de computadores, eligiendo el usuario cuál utilizar.

RF13.- El sistema debe permitir la edición de código ensamblador cuando se ejecute en modo interactivo, proporcionando un editor capaz de cargar y guardar ficheros para ello.

RF14.- El sistema debe permitir la simulación de la ejecución completa de un programa ensamblador.

RF15.- El sistema debe permitir la simulación de la ejecución instrucción a instrucción de un programa ensamblador.

RF16.- El sistema debe permitir la simulación en "modo terminal", simulando la ejecución un programa especificado y proporcionando la información de la simulación antes de finalizar la ejecución.

RF17.- El sistema debe ser capaz de proporcionar información sobre las *flags* de estado del computador simulado.

RF18.- El sistema debe permitir la especificación de puntos de ruptura en el código que pausen la simulación de los programas.

RF19.- El sistema debe ser capaz de proporcionar información al usuario sobre el *pipeline* en tiempo de ejecución, incluyendo sus etapas y las instrucciones ejecutándose en cada una.

RF20.- El sistema debe ser capaz de simular la ejecución de programas textuales escritos en ensamblador RISC-V.

RF21.- El sistema debe ser capaz de simular la ejecución de programas textuales escritos en ensamblador de MIPS.

De estos requisitos se consideran requisitos esenciales RF1, RF2, RF3, RF4, RF5 y RF6. Esto implica que el software resultante de este proyecto debe cumplir por lo menos todos esos requisitos. Durante el proyecto también se trata de cumplir todos los requisitos funcionales restantes, pero estos se consideran secundarios y no estrictamente necesarios.

5.1.2. Requisitos no funcionales

A continuación se detalla la lista de requisitos no funcionales planteados para el simulador.

RNF1.- El sistema debe ser multiplataforma, pudiendo ejecutarse en entornos Windows y Unix.

RNF2.- El sistema debe presentar una estructura modular.

RNF3.- El sistema debe ser fácilmente extensible para soportar otras arquitecturas de computadores y lenguajes ensamblador.

RNF4.- El código fuente del sistema debe ser de fácil comprensión y lectura, con el objetivo de facilitar la posible modificación y expansión de sus funcionalidades.

RNF5.- El código fuente del sistema debe estar escrito en inglés; incluyendo nombres de estructuras, clases, variables locales y globales, comentarios, y cualquier otro tipo de elemento al que el programador deba dar nombre.

RNF6.- El modo interactivo del sistema debe permitir cambiar de arquitectura de ejecución sin la necesidad de reiniciar el sistema.

RNF7.- El sistema debe permitir presentar la información numérica sobre la simulación de distintas formas cuando se ejecute en modo interactivo, incluyendo en base decimal, en base hexadecimal, y como caracteres ASCII.

RNF8.- El sistema debe permitir presentar la información de memoria de la simulación en distintos tipos de *endianness*.

De estos requisitos se consideran requisitos esenciales RNF1, RNF2, RNF3 y RNF4.

5.2. Arquitectura del sistema

En esta sección se describe la arquitectura elegida para el sistema, así como las decisiones de diseño relacionadas con la misma.

El requisito RNF2 especifica que el simulador debe presentar una estructura modular, aunque no detalla cuántos ni cuáles deben ser los módulos del mismo. Uno de los principales motivos por los que se impone esta estructura modular como requisito no funcional es para contribuir al cumplimiento del requisito RNF4. El requisito RNF4 especifica que el simulador debe ser extensible para soportar más arquitecturas y lenguajes ensamblador, con el objetivo de no tener que desarrollar otro sistema completo nuevo si se quiere utilizar arquitectura. Basándonos en estos dos requisitos, se ha decidido que el sistema se dividirá en varios módulos aislados, definiendo cada uno una arquitectura y lenguaje ensamblador soportada por el simulador. A cada uno de estos módulos los denominamos "arquitecturas del simulador", o simplemente "arquitecturas". Por ejemplo, en este proyecto se desarrolla la arquitectura LEGv8, y se plantea la posibilidad de desarrollar las arquitecturas RISC-V y MIPS (requisitos RF1, RF20 y RF21, respectivamente).

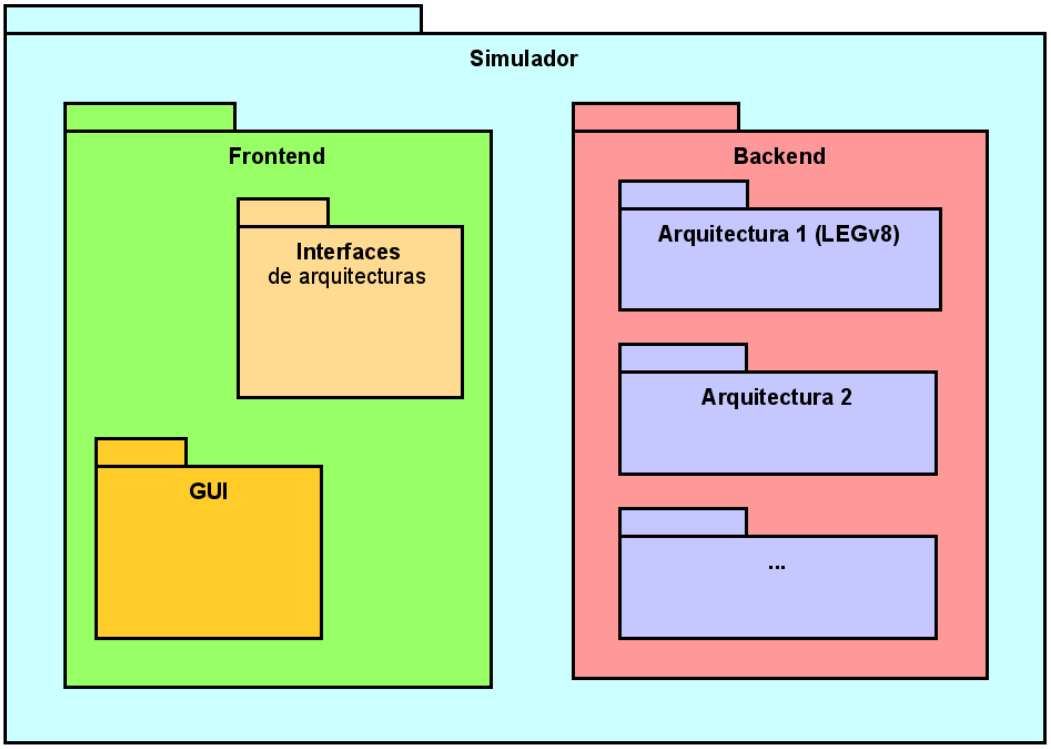

Figura 5.1: Diseño general básico de la arquitectura del simulador. El simulador puede implementar un número indefinido de arquitecturas.

Las arquitecturas por si solas, tal y como se han planteado, parecen ser simuladores individuales sin ninguna relación entre sí: necesitan una conexión común. Por tanto, el sistema debe presentar otro módulo adicional. Este módulo define todo el *frontend* del sistema, siendo las distintas arquitecturas del mismo, en su conjunto, el *backend*. Los propósitos del *frontend* serían dos:

1. Definir la interfaz gráfica del sistema, siendo el punto de conexión del usuario con el simulador.
2. Definir las interfaces que deben realizar las distintas arquitecturas para poder comunicarse correctamente con la interfaz gráfica del sistema.

La figura 5.1 ilustra la arquitectura general del sistema a nivel básico, tal y como se ha descrito hasta ahora.

Los módulos descritos hasta el momento son tan complejos que podrían tratarse de propios sistemas por sí mismos. Para definirlos con más detalle y facilitar su comprensión y utilización, se dividen a su vez en paquetes o submódulos. El *frontend* se divide en el módulo de interfaz gráfica, y el módulo de interfaces de arquitecturas. Estos dos módulos también se dividen en varios paquetes o submódulos. Las distintas arquitecturas del simulador se dividen en los submódulos análogos a los definidos en el módulo de interfaces de arquitecturas.

Para realizar la interfaz gráfica del simulador, se ha decidido utilizar el patrón de modelo-vista-controlador pasivo. De esta forma, el módulo de interfaz gráfica del *frontend* se dividirá en vistas

y controladores.

Decidir la subdivisión de las arquitecturas y el módulo de interfaces de arquitecturas no es una tarea sencilla. Se pretende que las interfaces que se definan sirvan como abstracción de cualquier arquitectura soportada por el simulador, ya que esa debería ser su función. Esto implica que se debe diseñar un módulo que sirva como interfaz genérica para cualquier arquitectura de computadores que se desee soportar. En un principio no se pretende imponer ninguna limitación sobre qué arquitecturas soporta el sistema. Sin embargo, dada la inmensa variedad de arquitecturas de computadores diferentes que se han desarrollado a lo largo de la historia, es cuestionable que se puedan definir interfaces que sirvan de abstracción para todas ellas al mismo tiempo.

Para reducir la complejidad de la tarea, se decidió que el simulador tan solo soportaría arquitecturas de computadores que pudieran ser consideradas RISC. Todas las arquitecturas RISC presentan ciertas características similares, lo que facilita la definición de interfaces comunes que las abstraigan.

Tras un análisis de las arquitecturas de tipo RISC, especialmente de las arquitecturas ARMv8, LEGv8, RISC-V y MIPS, tal y como se definen en las distintas ediciones de *Computer Organization and Design - The Hardware/Software Interface* [28–30], se identificaron las siguientes características de interés comunes:

- La manipulación de registros que almacenan un valor, incluyendo la realización de operaciones aritmético-lógicas sobre dichos valores.
- La utilización de una memoria para almacenar y cargar valores de distintos tipos, así como la división de esta memoria en varias "secciones" o rangos de direcciones con distinto propósito.
- El funcionamiento a base de instrucciones asociadas a un código máquina específico. Estas instrucciones presentan un nombre o mnemónico particular, y pueden agruparse o no en distintos tipos según su codificación, tal y como se indicaba en la sección 4.1. Las instrucciones son ejecutadas por el computador, y pueden producir un resultado sobre el que operen otras instrucciones.
- La presencia de un *pipeline* o secuencia de ejecución de instrucciones más o menos compleja, cuyo periodo de ejecución fundamental es el ciclo de reloj del procesador.
- La existencia de ciertos parámetros de estado mutables que determinan el comportamiento de ciertos componentes del procesador, o el resultado de la ejecución de ciertas instrucciones.

La identificación de estas características comunes facilitó la definición de los submódulos del módulo de interfaces. Como consecuencia directa, también facilitó la definición de submódulos de los módulos de arquitecturas del simulador. La división finalmente planteada, derivada directamente de las características comunes identificadas, es la que se indica a continuación:

- Un módulo para la gestión de los registros de la arquitectura.
- Un módulo para la gestión de la memoria de la arquitectura.

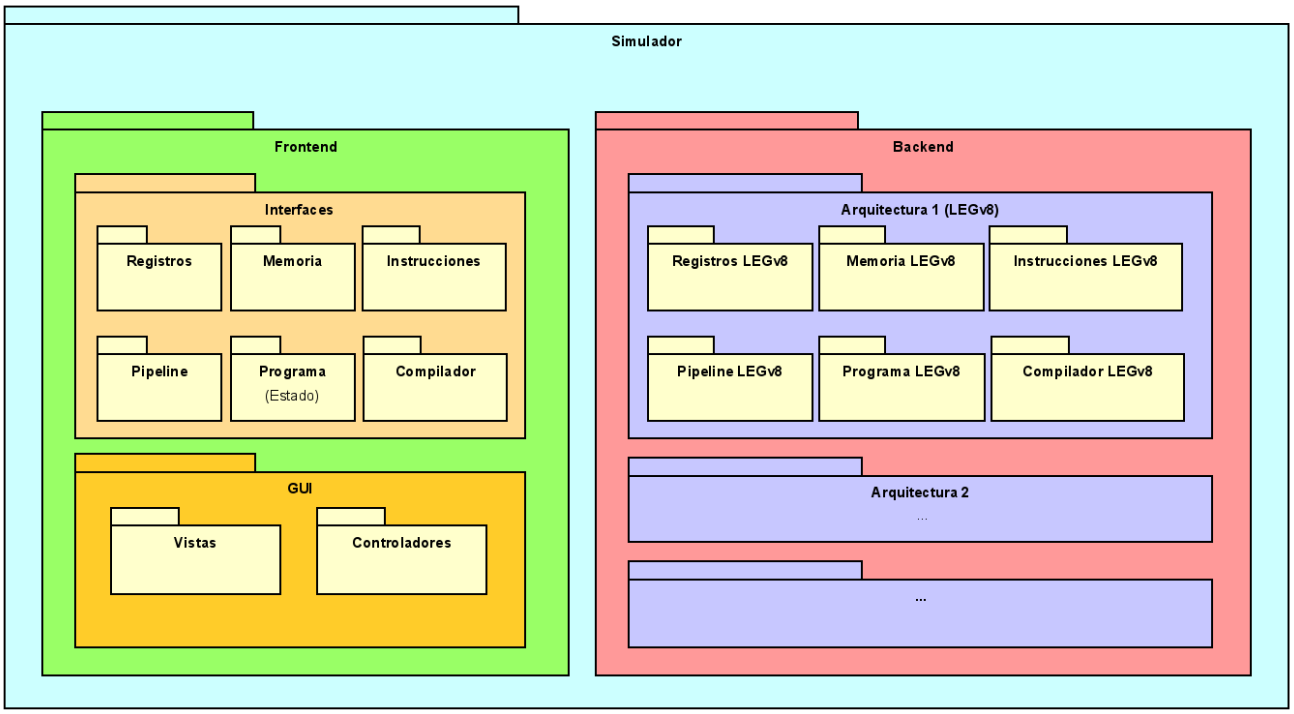

Figura 5.2: Diseño general detallado de la arquitectura del simulador. Todas las arquitecturas se subdividen en los mismos módulos.

- Un módulo para la gestión de las instrucciones ensamblador de la arquitectura. Las instrucciones son consideradas la "entidad de simulación" fundamentales del sistema. Este módulo contiene toda la funcionalidad asociada a las entidades de simulación.
- Un módulo para la gestión de los programas, incluyendo la información de estado de la CPU durante su ejecución.
- Un módulo para la gestión del flujo de simulación Este módulo implementa las funcionalidades de *pipeline* y de simulación de las distintas entidades de simulación.

A todos estos módulos descritos se debe añadir un módulo adicional de compilación/ensamblado de programas textuales a entidades de simulación ejecutables por el sistema. Este módulo se excluye de la lista anterior ya que no deriva de una característica intrínseca de las arquitecturas RISC, sino de una necesidad del simulador por establecer una traducción de las entradas del usuario a estructuras interpretables por el sistema. La figura 5.2 ilustra la estructura general del sistema de forma detallada, tal y como se ha descrito hasta ahora.

5.3. Diseño de los módulos del sistema

Una vez detallada la estructura general del sistema, se deben detallar las entidades en las que se descompondrá el simulador, así como sus relaciones y operaciones a través de las que se comunican. Para el diseño del sistema, las entidades en las que se descompone el simulador se han detallado como clases UML [45].

Tal y como se planteó la estructura en la sección anterior, los submódulos del sistema no requieren agrupar demasiadas clases. Esto se traduce en una alta cohesión dentro de los submódulos, ya que denota que la funcionalidad de cada submódulo puede delegarse en unas pocas clases.

Los requisitos de modularidad y extensibilidad del simulador (RNF2 y RNF3) tienen consecuencias significativas en el diseño del simulador. La más relevante es que todo el diseño del simulador gira fundamentalmente en torno al diseño de las interfaces de las arquitecturas. Desde el punto de vista del diseño, la descripción de las clases contendidas en el módulo de interfaces de las arquitecturas es equivalente a la descripción de las clases de cualquier implementación de una arquitectura. La única diferencia entre ambos tipos de clases es que las segundas son clases concretas que realizan las primeras. Así mismo. la definición de las clases contenidas en el módulo de interfaz gráfica depende de la definición de las clases del módulo de interfaces de las arquitecturas, pues la función del primer módulo es presentar al usuario información sobre las clases del segundo de forma interactiva. Se entiende por tanto que el módulo principal del simulador es el de interfaces de las arquitecturas. Su diseño es especialmente importante, pues influye en el diseño de los demás módulos.

En las siguientes secciones se expone de forma detallada el diseño del módulo de interfaces de arquitecturas, el diseño de la interfaz gráfica del simulador y el diseño de la ejecución en modo terminal del simulador. También se hace un inciso en el diseño de la arquitectura LEGv8 desarrollada para el simulador.

5.3.1. Módulo de interfaces de las arquitecturas

Como se ha detallado anteriormente, el módulo de interfaces de las arquitecturas juega un papel fundamental en el diseño del sistema, condicionando el diseño del resto de módulos del sistema. Esto se debe a las funciones que presenta el módulo en el sistema:

- Abstraer de forma genérica el concepto de "arquitectura de computadores" para permitir la extensibilidad modular del simulador (requisitos RNF2 y RNF3).
- Proporcionar información a la interfaz gráfica sobre la simulación de programas ensamblador.

Dada la especial importancia que presenta este módulo en el sistema, su diseño se explica en gran detalle en esta sección.

La figura 5.3.1 muestra el modelo de diseño del módulo de interfaces de arquitecturas del sistema, detallando las clases que lo conforman, sus operaciones y sus relaciones. El modelo de diseño se desarrolla partiendo de los requisitos del sistema y del posterior análisis del modelo de dominio del mismo. Acompañando al modelo de diseño, a continuación se explican de forma detallada todas las clases que conforman los submódulos del módulo de interfaces de las arquitecturas del sistema y las operaciones que realizan.

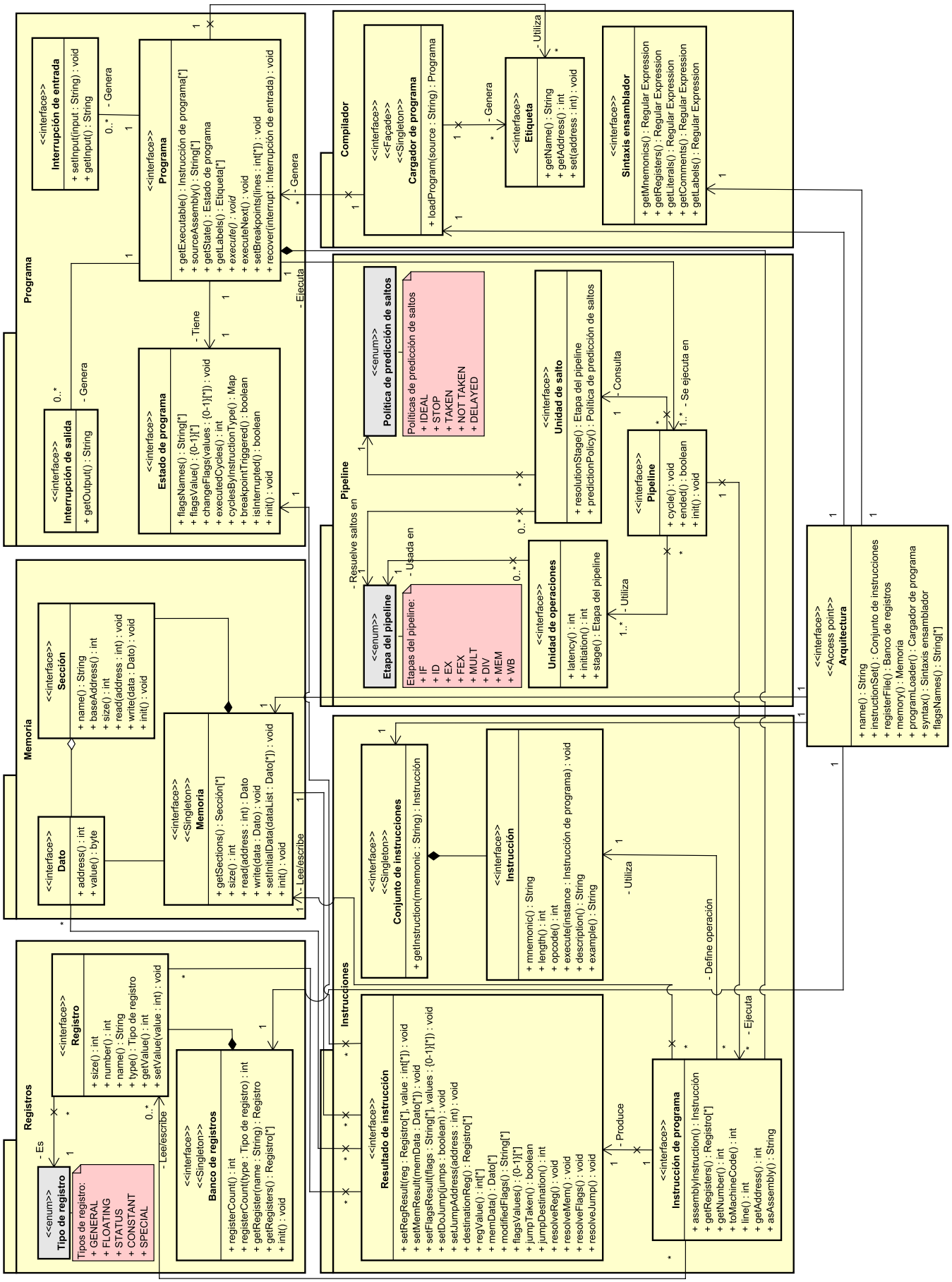

Figura 5.3: Modelo de diseño del módulo de interfaces de arquitecturas del sistema.

- **Módulo de Registros**.

  × Tipo de registro. Esta enumeración sirve para identificar los registros de la CPU según su propósito. Ciertas instrucciones ensamblador trabajan exclusivamente con cierto tipo de registros.
  Los distintos tipos de registros son:

  - *GENERAL* - registros de propósito general. Pueden ser leídos o escritos sin restricciones.
  - *FLOATING* - registros de datos de tipo coma flotante.
  - *STATUS* - registros de estado de la CPU. El valor almacenado en estos registros describe el estado de la CPU. Generalmente son registros de solo lectura.
  - *CONSTANT* - registros de almacenamiento de constantes. Estos registros son de solo lectura y su valor nunca cambia. El ejemplo habitual de registro constante es el registro *cero* de muchos computadores, que almacena el valor 0.
  - *VECTOR* - registros vectoriales. Estos almacenan conjuntos de valores en lugar de un valor único. Permiten que ciertas instrucciones ensamblador operen sobre múltiples valores a la vez.
  - *SPECIAL* - registros de propósito especial. Un ejemplo puede ser el contador de programa (*PC*), que almacena la dirección de memoria de la siguiente instrucción a ejecutar. El tratamiento de estos registros depende de la arquitectura concreta. Pueden o no ser registros de solo lectura. También pueden ser registros inaccesibles por el programador.

- Registro. Esta clase representa un registro genérico de una CPU. Los registros están definidos por un número y nombre únicos en la CPU y el valor que almacenan.
  Las operaciones que realiza esta clase son:

  - *size() : integer* - devuelve el tamaño en bits del valor contenido en el registro.
  - *number() : integer* - devuelve el número único del registro en la CPU. Este número sirve como identificador del registro.
  - *name() : String* - devuelve el nombre único del registro de la CPU. Este nombre puede coincidir con el número del registro. Por lo general, el nombre del registro almacena alguna clase de información sobre su función o uso convencional. Los programadores suelen referirse a los registros por su nombre en los programas ensamblador, aunque dependiendo del compilador también pueden hacerlo por su número.
  - *type() : Tipo de registro* - devuelve un valor (*enum*) identificativo del tipo del registro.
  - *getValue() : integer* - devuelve el valor almacenado en el registro.
  - *setValue(value : integer) : void* - modifica el valor almacenado en el registro.

- Banco de registros. Esta clase es una composición de todos los registros del computador. Su función es inicializar en el sistema todos los registros existentes en la arquitectura y proporcionar un punto de acceso único a ellos. La clase Banco de registros, por tanto, sigue un patrón de diseño *singleton* [46].
  Aunque una CPU real puede tener más de un banco de registros, trabajando distinto tipo de instrucciones ensamblador sobre distintos bancos de registros, el simulador solo trabaja con una única instancia de Banco de registros en tiempo de ejecución. Esta discrepancia entre el sistema real simulado y el simulador se debe a que, conceptualmente,

tener un mayor número de bancos de registros en una CPU no aporta ningún beneficio; es simplemente una limitación del hardware. Las reglas lógicas que rigen la accesibilidad a los registros para cada instrucción ensamblador concreta se pueden implementar con un único banco de registros.
Las operaciones que realiza esta clase son:

- *registerCount() : integer* - devuelve el número de registros totales que hay en la CPU simulada.
- *registerCount(type : Tipo de registro) : integer* - devuelve el número de registros del tipo especificado por parámetro que hay en la CPU simulada.
- *getRegister(name : String) : Registro* - devuelve la instancia del Registro cuyo nombre es el provisto como parámetro (*name*). También puede especificarse el número de registro como *String*, en lugar del nombre.
- *getRegisters() : Registro[*] {ordered}* - devuelve una lista ordenada que contiene todos los registros de la arquitectura. El orden exacto de los registros en la lista es específico para cada arquitectura. La única restricción de orden es que la lista debería considerarse ordenada a ojos de un programador de lenguaje ensamblador de la arquitectura.
- *init() : void* - inicializa el banco de registros de la CPU. Esta operación prepara el banco de registros para ejecutar un programa, almacenando en cada uno de los registros de la CPU su valor por defecto. Conceptualmente, realizaría la operación análoga a reiniciar el computador, a nivel de los registros.

- **Módulo de Memoria**

  - Dato. Esta clase sirve como contenedor o abstracción para el concepto de "dato almacenado en una memoria". Los datos de memoria están definidos por la dirección de memoria en la que se encuentran almacenados, así como por el valor que almacenan. Habitualmente, las memorias de los computadores almacenan datos de un byte de tamaño (equivalente a 8 bits). Por tanto, el tamaño del valor de un dato es de un byte.
    Las operaciones que realiza esta clase son:
    - *address() : integer* - devuelve la dirección en memoria correspondiente al dato.
    - *value() : byte* - devuelve el valor del dato.
  - Sección de memoria. Esta clase proporciona una abstracción para un conjunto contiguo de datos en memoria que comparten cierta finalidad lógica. Esta finalidad común puede venir definida por la implementación hardware del computador, o simplemente tratarse de una convención aceptada por la mayoría de programadores para la arquitectura. Un ejemplo de sección de memoria habitual es la pila (*stack*).
    Una sección viene definida por un nombre que la identifica, una dirección de memoria base desde la que comienza la sección, y un tamaño que define el número de datos de memoria contenidos en la sección.
    Las operaciones que realiza esta clase son:
    - *name() : String* - devuelve el nombre identificativo de la sección.
    - *baseAddress() : integer* - devuelve la dirección de memoria base de la sección.
    - *size() : integer* - devuelve el número de datos de memoria contenidos en la sección.
    - *read(address : integer) : Dato* - devuelve el Dato almacenado en la dirección de memoria proporcionada como parámetro. La dirección de memoria proporcionada debe estar contenida en la sección.

  - *write(data : Dato) : void)* - almacena en memoria el Dato proporcionado como parámetro. La dirección de memoria del dato proporcionado debe estar contenida en la sección.
  - *init() : void* - inicializa la sección de memoria. Esta operación prepara la sección de memoria para ejecutar un programa, poniendo todos los datos de memoria de la sección a su valor por defecto (habitualmente 0). Conceptualmente, realizaría la operación análoga a reiniciar el computador, a nivel de la sección de memoria concreta.

- Memoria. Esta clase representa una memoria genérica de una CPU. Las memorias de los computadores tienen como funcionalidad almacenar grandes cantidades de datos, accesibles para realizar cómputo sobre ellos. Los datos almacenados se indexan comenzando por 0, hasta la capacidad de la memoria menos uno. A estos índices se les llama "direcciones" de memoria. El acceso los datos almacenados en la memoria se realiza a través de sus respectivas direcciones de memoria. Por tanto, la clase Memoria proporciona un punto de acceso único para acceder y gestionar una gran cantidad de Datos de memoria a través de sus direcciones. Por tanto, la clase Memoria sigue un patrón de diseño *singleton*.
  Desde el punto de vista del sistema, la Memoria es una composición de todas las secciones de memoria de la CPU. Al igual que ocurría con el Banco de registros, aunque una CPU real puede tener más de una memoria, el simulador solo trabaja con una única instancia de Memoria. En este contexto, esta discrepancia se debe a que el acceso a memoria generalmente se realiza de forma transparente al computador. Otro componente hardware distinto suele ser el encargado de resolver cuál de las memorias físicas existentes corresponde a la dirección de memoria solicitada por el computador. De esta forma, desde el punto de vista del computador solo existe una memoria única que contiene los datos para todo el espacio de direcciones accesible. Esta función corresponde exactamente con la función de la clase Memoria.
  Las operaciones realizadas por esta clase son:
  - *getSections() : Section[*]* - devuelve una lista que contiene todas las secciones de memoria de la arquitectura.
  - *size() : integer* - devuelve el tamaño de la memoria. El tamaño indica la cantidad de datos (de tamaño byte) que pueden ser almacenados.
  - *read(address : integer) : Dato* - devuelve el Dato almacenado en la dirección de memoria proporcionada como parámetro.
  - *write(data : Dato) : void)* - almacena en memoria el Dato proporcionado como parámetro.
  - *setInitialData(dataList : Dato[*]) : void* - indica a la memoria de la CPU cómo debe inicializarse antes de ejecutar el programa actualmente cargado en el simulador. Esta indicación se hace a través de una lista de datos proporcionada como parámetro: esta lista contiene todos los Datos (direcciones y respectivos valores) especificados en el código fuente del programa y que deben ser cargados en memoria antes de su ejecución.
  - *init() : void* - inicializa la memoria de la CPU. Esta operación prepara la memoria para ejecutar el programa actualmente cargado en el simulador. Esto implica inicializar todas las secciones de memoria y almacenar los datos iniciales del programa, especificados con la operación *setInitialData(Data[*])* (ver la operación anterior). Conceptualmente, realizaría la operación análoga a reiniciar el computador y volver

a cargar el programa actual, a nivel de memoria.

- **Módulo de instrucciones**

  - Resultado de instrucción. Esta clase representa el resultado de la ejecución de una instrucción. Se utiliza para facilitar la anticipación de resultados al simular *pipelines* segmentados. Se contemplan cuatro formas en las que la ejecución de una instrucción altera el estado de una CPU: modificando los contenidos de los registros, modificando el contenido de la memoria, modificando las banderas de estado, o produciendo una bifurcación (modificando el contador de programa de forma directa). Estas cuatro alteraciones pueden manifestarse en la CPU en distintas etapas de ejecución de la instrucción que las causa. Al manifestarse en una etapa distinta a en la que se ejecuta la instrucción (EX), estas alteraciones deben almacenarse explícitamente en una clase para ser resueltas (es decir, para que su efecto se manifieste) en otra etapa.
    Las operaciones realizadas por esta clase son:

    - *setRegResult(reg : Registro[*], value : integer[*]) : void* - apunta que el resultado de instrucción modifica los registros especificados por parámetro, cambiando los valores que almacenan por los respectivos valores especificados por parámetro.
    - *setMemResult(memData : Dato[*]) : void* - apunta que el resultado de instrucción modifica los contenidos de la memoria de la CPU, almacenando los Datos especificados por parámetro.
    - *setFlagsResult(flags : String[*], values : {0, 1}[*]) : void* - apunta que el resultado de instrucción modifica las banderas de estado de la CPU indicadas por parámetro, cambiando su valor por los respectivos valores especificados por parámetro. Los valores de las banderas pueden ser 0 o 1 exclusivamente.
    - *setDoJump(jumps : boolean) : void* - apunta si la instrucción modifica el flujo del programa (es decir, bifurca) o no, dependiendo del parámetro provisto.
    - *setJumpAddress(address : integer) : void* - apunta la dirección de memoria de instrucción a la que debería bifurcar el programa si el resultado de instrucción causa una bifurcación. Este valor se utiliza para modificar el contador de programa al resolver el resultado de instrucción.
    - *destinationReg() : Register[*]* - devuelve los registros modificados por el resultado de instrucción.
    - *regValue() : integer[*]* - devuelve los valores que este resultado de instrucción almacena en los registros que modifica. El orden de los valores devueltos corresponde con el orden de los Registros devueltos al invocar la operación *destinationReg()*.
    - *memData() : Data[*]* - devuelve los datos de memoria modificados por el resultado de instrucción.
    - *modifiedFlags() : String[*]* - devuelve el nombre de las banderas de estado de la CPU modificadas por el resultado de instrucción.
    - *flagsValues() : {0, 1}[*]* - devuelve los valores con los que el resultado de instrucción modifica las banderas de estado de la CPU. El orden de los valores devueltos corresponde con el orden de los nombres de las banderas tal y como se devuelven al invocar la operación *modifiedFlags()*.
    - *jumpTaken() : boolean* - devuelve si el resultado de instrucción causa una bifurcación o no.

- *jumpDestination() : integer* - devuelve la dirección de memoria de la instrucción a la que el resultado de instrucción hace bifurcar el flujo del programa. En otras palabras, devuelve el valor con el que el resultado de instrucción modifica el contador de programa.
- *resolveReg() : void* - causa que las modificaciones que el resultado de instrucción realiza sobre los registros de la CPU tomen efecto.
- *resolveMem() : void* - causa que las modificaciones que el resultado de instrucción realiza sobre la memoria de la CPU tomen efecto.
- *resolveFlags() : void* - causa que las modificaciones que el resultado de instrucción realiza sobre las banderas de estado de la CPU tomen efecto.
- *resolveJump() : void* - causa que las modificaciones que el resultado de instrucción realiza sobre el flujo del programa actual tomen efecto, de existir.

- Instrucción. Esta clase representa una instrucción ensamblador del conjunto de instrucciones de la arquitectura. Las instrucciones ensamblador están definidas por un mnemónico, un código de operación y por la ejecución de una funcionalidad u operación concreta en la CPU.
  Las operaciones que realiza esta clase son:
  - *mnemonic() : String* - devuelve el mnemónico de la instrucción ensamblador.
  - *length() : integer* - devuelve el tamaño en bits necesarios para codificar la instrucción.
  - *opcode() : integer* - devuelve el código de operación asociado a la instrucción ensamblador y que la CPU usa internamente para decodificar su funcionalidad asociada.
  - *execute(instance : Instrucción de programa) : void* - ejecuta la operación asociada a la instrucción con la información o sobre los operandos especificados en la Instrucción de programa (clase detallada más adelante) provista por parámetro.
  - *description() : String* - provee una descripción explicativa de la utilización de la instrucción.
  - *example() : String* - provee un ejemplo de uso de la instrucción.
- Conjunto de instrucciones. Esta clase representa el conjunto de instrucciones ensamblador soportado por la arquitectura. Es, por tanto, una composición de todas las Instrucciones de la misma arquitectura en el sistema. Su función es inicializar todas las instancias de Instrucción de la arquitectura y proporcionar un punto de acceso único a ellas. La clase Conjunto de instrucciones, por tanto, sigue un patrón de diseño *singleton*.
  Esta clase realiza una única operación:
  - *getInstruction(mnemonic : String) : Instruction* - devuelve la Instrucción de la arquitectura cuyo mnemónico coincide con el provisto por parámetro.
- Instrucción de programa. Esta clase representa una sentencia concreta de un programa ensamblador específico. Las instrucciones de programa están definidas por una instrucción ensamblador, la información u operandos sobre los que trabaja, la línea del código fuente en la que está escrita la sentencia y la dirección de memoria en la que ha sido escrita. Se trata de la unidad de simulación fundamental en el sistema.
  Las operaciones que realiza esta clase son:
  - *assemblyInstruction() : Instrucción* - devuelve la Instrucción ensamblador asociada a la instrucción de programa.

- *getRegisters() : Registro[*]* - devuelve todos los registros de la CPU utilizados por la instrucción de programa, de haberlos. Esto incluye tanto registros que son leídos como registros que son escritos.
- *getNumber() : integer* - devuelve la información numérica utilizada por la instrucción de programa, de haberla. El significado de esta información numérica depende de la instrucción ensamblador específica asociada a la instrucción de programa. Ejemplos de esta información numérica son: valores numéricos utilizados por instrucciones que trabajan con inmediatos, o direcciones de memoria utilizadas por instrucciones que acceden a memoria o bifurcan.
- *toMachineCode() : integer* - devuelve el código máquina correspondiente a la instrucción de programa.
- *line() : integer* - devuelve el número de la línea del código fuente que ha generado la instrucción de programa.
- *getAddress() : integer* - devuelve la dirección de memoria en la que está escrito el código máquina de esta instrucción de programa.
- *asAssembly() : String* - devuelve el código ensamblador de más bajo nivel utilizado para generar la instrucción de programa. Este código ensamblador ya debe estar preprocesado, pudiendo convertirse de forma directa al código máquina asociado a la instrucción de programa.

Módulo de *pipeline*

× Etapa del *pipeline*. Esta enumeración sirve para identificar y gestionar las diferentes etapas del *pipeline* de la arquitectura.
Las distintas etapas del *pipeline* contempladas son:

- *IF* - etapa de búsqueda de instrucciones. En esta etapa una nueva instrucción entra en el *pipeline*.
- *ID* - etapa de decodificación de instrucciones. En esta etapa se ha identificado el tipo de instrucción ensamblador que ha entrado en el *pipeline*. También se detectan los posibles riesgos que genera con otras instrucciones en ejecución, decidiendo si la instrucción pasa a la siguiente etapa o se detiene el flujo de instrucciones un ciclo.
- *EX* - etapa de ejecución de instrucciones. En esta etapa se realiza la operación asociada a las instrucciones, generando su resultado.
- *FEX* - subetapa de ejecución de instrucciones de coma flotante. En esta etapa puede realizarse la operación asociada a instrucciones de coma flotante, generando su resultado.
- *MULT* - subetapa de ejecución de instrucciones de multiplicación. En esta etapa puede realizarse la operación asociada a instrucciones de multiplicación, generando su resultado.
- *DIV* - subetapa de ejecución de instrucciones de división. En esta etapa puede realizarse la operación asociada a instrucciones de división, generando su resultado.
- *MEM* - etapa de acceso a memoria. En esta etapa se realizan los accesos (lecturas o escrituras) a memoria indicados por las instrucciones.
- *WB* - etapa de escritura de resultados. En esta etapa los resultados generados por la ejecución de las instrucciones son escritos en los correspondientes registros de la CPU. La ejecución de una instrucción queda totalmente reflejada en la CPU tras esta etapa.

× Política de predicción de saltos. Esta enumeración sirve para configurar y determinar el comportamiento del *pipeline* con respecto a la predicción de saltos o bifurcaciones. En el sistema solo se contemplan políticas de predicción de salto estáticas.
Las políticas de predicción de saltos contempladas son:

- *IDEAL* - los resultados de los saltos se predicen correctamente siempre. Por tanto, durante la ejecución de programas, nunca se pierde productividad por mantener la CPU desocupada u ocupada con instrucciones incorrectas a causa de bifurcaciones. Evidentemente, esta política de predicción no se da en ningún computador real, ya que es imposible conocer el resultado de una instrucción de bifurcación antes de ejecutarla. No obstante, presenta un especial interés didáctico, por lo que se incluye en el simulador.
- *STOP* - los resultados de los saltos no se predicen. En su lugar, se producen burbujas en el *pipeline*, dejando de introducir nuevas instrucciones, hasta que se ejecute la instrucción de salto correspondiente y se conozca su resultado.
- *TAKEN* - se predice que los saltos siempre son tomados. Hasta que se conozca el resultado de la instrucción de bifurcación, se añadirán instrucciones a el *pipeline* asumiendo que ya se debe tomar el salto, modificando el contador de programa de manera correspondiente. Si la predicción resulta ser incorrecta, se descartan las instrucciones añadidas incorrectamente a el *pipeline* y se recupera el valor correcto del contador de programa.
- *NOT TAKEN* - se predice que los saltos nunca son tomados. Hasta que se conozca el resultado de la instrucción de bifurcación, se añadirán instrucciones a el *pipeline* asumiendo que no se va a modificar el flujo actual de instrucciones y que no debe modificarse el contador de programa de manera especial. Si la predicción resulta ser incorrecta, se descartan las instrucciones añadidas incorrectamente a el *pipeline* y se realiza el salto.
- *DELAYED* - se simula la funcionalidad de "bifurcación retardada". Esta política asume que las instrucciones del programa están reordenadas de forma que, al tomarse un salto, todas las instrucciones que se encuentren en el *pipeline* deben ejecutarse antes de ejecutar la instrucción a la que se salta. A nivel de funcionamiento, esta política es igual que la política *NOT TAKEN*, exceptuando que no se descarta ninguna instrucción en ningún caso.
  La necesidad de soportar este tipo de política de predicción de saltos surge de forma directa del requisito RF10.

• Unidad de salto. Esta clase sirve para gestionar la predicción de saltos en el *pipeline*. Es habitual que los procesadores, para minimizar la penalización de las predicciones de saltos incorrectas, adelanten la ejecución de los saltos lo máximo posible. Por ejemplo, en el *pipeline* clásico RISC, las bifurcaciones podrían ejecutarse en la etapa ID en vez de en EX, reduciendo la penalización de las predicciones incorrectas de 2 ciclos de reloj a 1 (tal y como se describe en la sección 4.8 de *Computer Organization and Design ARM edition: The Hardware/Software Interface* [28]). Esta clase es un contenedor tanto para la política de predicción de saltos utilizada como para la etapa del *pipeline* en la que se producen los saltos. Por tanto, almacena toda la información relativa a la ejecución y predicción de saltos en el *pipeline*.
El simulador permite resolver los saltos en las etapas EX, ID e IF. Aunque sea poco realista permitir la resolución de saltos en la etapa IF, se implementa la simulación de esa funcionalidad porque presenta cierto interés didáctico.

Las operaciones que realiza esta clase son:

- *resolutionStage() : Etapa del pipeline* - devuelve la etapa del *pipeline* en la que se producen las bifurcaciones.
- *predictionPolicy() : Política de predicción de saltos* - devuelve la política de predicción de saltos utilizada en el *pipeline*.

- Unidad de operaciones. Esta clase representa una unidad funcional del hardware del computador dedicada a realizar cálculos de operaciones aritmético-lógicas. Ejemplos de estas unidades serían las unidades aritmético-lógicas (ALUs), las unidades de coma flotante, las unidades de multiplicación o las unidades de división.
  Estas unidades funcionales son utilizadas en alguna de las subetapas de ejecución del *pipeline* (*EX*, *FEX*, *MUL*, *DIV*) para ejecutar la operación asociada a ciertas instrucciones, según corresponda. Cada una de estas unidades puede tardar un número de ciclos de reloj distinto en producir los resultados de sus respectivas instrucciones, dando lugar a etapas multiciclo en el *pipeline* si este número de ciclos es mayor que uno. Del mismo modo, cada una de estas unidades puede ejecutar de forma segmentada un número distinto de instrucciones a la vez. Estas dos características vienen determinadas por la latencia y el intervalo de iniciación de la unidad funcional, tal y como se describe en el apéndice C de *Computer Architecture: A Quantitative Approach* [44]. Varias unidades funcionales distintas pueden utilizarse al mismo tiempo, cada una en una subetapa paralela distinta dentro de la etapa de ejecución del *pipeline*. Sin embargo, esto introduce nuevos riesgos estructurales en el computador.
  La necesidad de diseñar una clase con estas características surge de forma directa del requisito RF11.
  Las operaciones que realiza esta clase son:
  - *latency() : integer* - devuelve la latencia de la unidad funcional. La latencia es el número de ciclos intermedios que se deben suceder en la CPU desde que se emite una instrucción que utilice la unidad funcional para que se pueda emitir otra instrucción que utilice el resultado de la primera. En otras palabras, es el número de ciclos que la unidad funcional tarda en procesar una instrucción hasta producir su resultado, menos uno.
  - *initiation() : integer* - devuelve el intervalo de iniciación de la unidad funcional. El intervalo de iniciación es el número de ciclos que se deben suceder en la CPU entre la emisión de dos instrucciones que utilicen la unidad funcional.
  - *stage() : Etapa del pipeline* - devuelve la subetapa del *pipeline* en la que se utiliza la unidad de operaciones.

- *Pipeline*. Esta clase representa el *pipeline* de ejecución de instrucciones de un computador. Múltiples instrucciones pueden encontrarse en ejecución al mismo tiempo en un mismo *pipeline*, siempre y cuando lo hagan en etapas o subetapas distintas.
  Esta clase se encarga de ejecutar todas las instrucciones de programa que conforman un Programa.
  Las operaciones que realiza esta clase son:
  - *cycle() : void* - hace pasar un ciclo de reloj en el *pipeline*. Esta operación posiblemente produzca el avance en el *pipeline* y la ejecución de varias instrucciones.
  - *ended() : boolean* - consulta si el *pipeline* se ha detenido tras la ejecución completa del programa actual. Si esta operación devuelve *true*, todas las instrucciones de programa del programa actual han sido ejecutadas. Si devuelve *false*, todavía quedan instrucciones de programa por ejecutar.

- *init() : void* - inicializa el *pipeline* de la CPU. Esta operación prepara el *pipeline* para ejecutar un programa.

- **Módulo de Programa**

  - Interrupción de salida. Esta clase representa una interrupción o llamada al sistema asociada a una acción de salida que puede generarse durante la ejecución de un programa. Las interrupciones de salida proveen información de forma explícita a un sistema externo; en este caso, a la interfaz gráfica del simulador. La interfaz gráfica del simulador, a su vez, presenta la información de estas interrupciones al usuario. Un ejemplo de interrupción de salida es la impresión por salida estándar de datos de distinto tipo.
    Esta clase realiza una única operación:
    - *getOutput() : String* - devuelve, en formato textual, la información de salida que se provee al sistema externo.
  - Interrupción de entrada. Esta clase representa una interrupción o llamada al sistema asociada a una acción de entrada que puede generarse durante la ejecución de un programa. Las interrupciones de entrada proveen información de forma explícita al computador desde un sistema externo; en este caso, desde el usuario, pasando por la interfaz gráfica. Un ejemplo de interrupción de salida es la lectura por entrada estándar de datos de distinto tipo.
    Las operaciones que realiza esta clase son:
    - *setInput(input : String) : void* - define, en formato textual, la información de entrada de la interrupción. Esta información será provista al computador.
    - *getInput() : String* - devuelve, en formato textual, la información de entrada que se provee al computador. Esta información debe ser previamente definida mediante la invocación de *setInput(String)* (ver la operación anterior).
  - Estado de programa. Esta clase engloba las características que gobiernan el estado de la CPU mientras ejecuta un programa. También engloba otras variables sobre la ejecución del programa de carácter estadístico y de interés informativo en las simulaciones.
    Las funciones de esta clase podrían combinarse con las de Programa (ver siguiente clase) en una única clase. Sin embargo, se ha preferido realizar esta división de las funcionalidades en dos clases para favorecer una mejor comprensión y organización conceptual.
    Las operaciones que realiza esta clase son:
    - *flagsNames() : String[*]* - devuelve el nombre de las banderas de estado que haya en la CPU.
    - *flagsValue() : {0, 1}[*]* - devuelve los valores actuales de las banderas de estado de la CPU. El orden de los valores devueltos corresponde con el orden de los nombres de las banderas tal y como se devuelven al invocar la operación *flagsNames()*.
    - *changeFlags(values : {0, 1}[*]) : void* - cambia el valor de las banderas de estado de la CPU por los provistos por parámetro. El orden de los valores provistos corresponde con el orden de los nombres de las banderas tal y como se devuelven al invocar la operación *flagsNames()*.
    - *executedCycles() : integer* - devuelve el número de ciclos de reloj que lleva ejecutándose el programa actual.
    - *cyclesByInstructionType() : (String, integer)[*]* - para cada tipo de instrucción distinto de la arquitectura, devuelve tanto el nombre del tipo de instrucción, como

el número de ciclos de reloj de dedicados hasta el momento en ejecutar instrucciones de ese tipo.
  - *breakpointTriggered() : boolean* - consulta si la ejecución del programa se ha detenido por el efecto de un punto de ruptura.
  - *isInterrupted() : boolean* - consulta si la ejecución del programa se ha detenido por el efecto de una interrupción o llamada al sistema que debe ser resuelta.
  - *init() : void* - inicializa el estado del programa. Esta operación inicializa todos los valores necesarios para comenzar la ejecución del programa asociado.

- Programa. Esta clase representa un programa ensamblador de la arquitectura. Se trata de una composición de las instrucciones de programa de un programa ensamblador específico, y es la unidad de ejecución más general del simulador.
  Las operaciones que realiza esta clase son:
  - *getExecutable() : Instrucción de programa[*]* - devuelve el conjunto de instrucciones de programa que componen el programa.
  - *sourceAssembly() : String[*]* - devuelve el código ensamblador textual que generó el programa, separado en las distintas líneas que lo componen.
  - *getState() : Estado de programa* - devuelve el Estado de programa asociado al programa.
  - *getLabels() : Etiqueta[*]* - devuelve las etiquetas de memoria (clase detallada en el punto dedicado al módulo de compilación) definidas en el programa. Cada etiqueta incluye el nombre de la etiqueta y su dirección de memoria correspondiente.
  - *execute() : void* - ejecuta el programa de principio a fin. Esto implica ejecutar todas las instrucciones de programa que componen el programa.
  - *executeNext() : void* - ejecuta el siguiente ciclo de reloj del programa.
  - *setBreakpoints(lines : integer[*])* - añade puntos de ruptura al programa. Los puntos de ruptura se añaden en las instrucciones correspondientes a los números de línea provistos por parámetro.
  - *recover(interrupt : Interrupción de entrada) : void* - recupera el programa de una interrupción de entrada que estaba pendiente por resolver. Esta operación resuelve la interupción de entrada provista por parámetro.

Módulo de compilador

- Cargador de programa. Esta clase sirve como fachada para el compilador de programas ensamblador de la arquitectura. Sigue, por tanto, el patrón de diseño *façade*. El resto de clases con las que se comunica esta fachada dependen de la implementación concreta de las arquitecturas.
  El Cargador de programa compila un programa en código ensamblador a Instrucciones de programa, produciendo un Programa simulable como resultado.
  Una única instancia de esta clase es capaz de compilar un número indefinido de programas independientes. Por tanto, esta clase sigue el patrón de diseño *singleton*.
  Esta clase realiza una única operación:
  - *loadProgram(source : String) : Program* - compila un programa textual de código fuente ensamblador de la arquitectura a un programa ejecutable por el simulador.

- Etiqueta. Esta clase representa una etiqueta de memoria de un programa ensamblador. Las etiquetas de memoria se utilizan en programas ensamblador para facilitar el acceso

a una dirección de memoria, ya sea de instrucciones o de datos, al programador. Las etiquetas de memoria son ampliamente usadas a la hora de realizar cargas y almacenamientos de datos en memoria, así como a la hora de realizar bifurcaciones y saltos a otras instrucciones del programa.
Las etiquetas de memoria están definidas por un nombre y una dirección de memoria. Permiten acceder a las direcciones de memoria correspondientes utilizando los nombres de las etiquetas, que generalmente son mucho más fáciles de recordar y gestionar. Las etiquetas de memoria son generadas, gestionadas y sustituidas por las respectivas direcciones de memoria en tiempo de compilación.
Las operaciones que realiza esta clase son:

  - *getName() : String* - devuelve el nombre de la etiqueta de memoria, que sirve para identificarla.
  - *getAddress() : integer* - devuelve la dirección de memoria absoluta asociada a la etiqueta. Todas las apariciones de la etiqueta en el código fuente ensamblador serán sustituidas por una referencia a esta dirección de memoria, ya sea por su valor absoluto o como dirección relativa, dependiendo del contexto.
  - *set(address : integer) : void* - asigna una dirección de memoria a la etiqueta. Es común que en los programas ensamblador haya apariciones de etiquetas en líneas previas a la definición de las mismas (por ejemplo, en saltos hacia delante o llamadas a funciones. En estos casos, la dirección de memoria asociada a las etiquetas ha de asignarse después de la creación de la clase.

- Sintaxis ensamblador. Esta clase agrupa el conjunto de reglas sintácticas, en forma de expresiones regulares, utilizadas para resaltar distintos elementos del código fuente de los programas en editores de código. Se utiliza para resaltar la sintaxis de los programas ensamblador en el editor de código del simulador.
Las operaciones que realiza esta clase son:

  - *getMnemonics() : Regular Expression* - devuelve las reglas sintácticas para resaltar los mnemónicos de las distintas instrucciones en editores de código ensamblador de la arquitectura.
  - *getRegisters() : Regular Expression* - devuelve las reglas sintácticas para resaltar los registros en editores de código ensamblador de la arquitectura.
  - *getLiterals() : Regular Expression* - devuelve las reglas sintácticas para resaltar los literales (valores numéricos) en editores de código ensamblador de la arquitectura.
  - *getComments() : Regular Expression* - devuelve las reglas sintácticas para resaltar los comentarios en editores de código ensamblador de la arquitectura.
  - *getLabels() : Regular Expression* - devuelve las reglas sintácticas para resaltar las etiquetas de memoria en editores de código ensamblador de la arquitectura.

Aparte de todas estas clases, el módulo de interfaces de las arquitecturas del sistema presenta una clase adicional que no se incluye en ninguno de los submódulos:

- Arquitectura. Esta clase es una fachada que permite el acceso por parte de la interfaz gráfica a las clases del sistema de las que debe consultar información o realizar operaciones. No sigue estrictamente el patrón de diseño *façade*, pero juega un papel especial en el sistema, comportándose de forma similar a una fachada para el módulo. Es el único punto de acceso

que conocen los demás módulos para acceder a las demás clases del módulo de interfaces de las arquitecturas.

- *name() : String* - devuelve el nombre identificativo de la arquitectura.
- *instructionSet() : Conjunto de instrucciones* - devuelve el conjunto de instrucciones de la arquitectura.
- *registerFile() : Banco de registros* - devuelve el banco de registros de la arquitectura.
- *memory() : Memoria* - devuelve la memoria de la arquitectura.
- *programLoader() : Cargador de programa* - devuelve el cargador de programa de la arquitectura.
- *syntax() : Sintaxis ensamblador* - devuelve las reglas de resaltado de sintaxis ensamblador de la arquitectura.
- *flagsNames() : String[*]* - devuelve los nombres de las banderas de estado de la arquitectura.

El diseño detallado es suficiente para comenzar la implementación lógica del submódulo de interfaces de arquitecturas del sistema. Antes de comenzar la implementación, sin embargo, es conveniente realizar un análisis del diseño propuesto en busca de puntos problemáticos y posibles mejoras. Puesto que el sistema a desarrollar es un prototipo que pretende expandirse y refinarse de forma iterativa durante los próximos cursos académicos, ha de evaluarse la facilidad de modificación de los módulos de diseño propuestos. Estas característica se evalúan analizando las conexiones existentes entre los distintos módulos del sistema.

Observando las relaciones entre las clases y módulos de diseño del sistema, se puede observar que el módulo de instrucciones es el que presenta un mayor número de relaciones con clases de otro módulo. Esto es comprensible, pues la ejecución de instrucciones consulta y modifica múltiples componentes del computador.

Por otra parte, hay otros módulos que, aunque presenten pocas relaciones, son bastante estrechas. Es el caso del módulo de programa con los módulos de compilador y *pipeline*. Pese a esta estrecha relación, la división de las clases en estos tres módulos esta justificada, aparte de conceptualmente, por las diferencias de complejidad de implementación de cada módulo. Los módulos de *pipeline* y compilador requieren una implementación más cuidada y extensa que el módulo de programa, como se detalla en el capítulo 6. Esta diferencia favorece la separación del módulo de programa tal y como se ha hecho. Al no existir ninguna relación entre las clases del módulo de *pipeline* y las clases del módulo de compilador, está justificado también que estas se encuentren en módulos separados.

La estabilidad [47] de un módulo es una medida de de la dificultad de cambio del módulo. Para evaluar el diseño del módulo de interfaces de las arquitecturas, se ha decidido medir la estabilidad de sus submódulos. La tabla 5.1 muestra los resultados obtenidos. Se puede observar que los submódulos que presentan menor estabilidad son el de *Pipeline* y el de Programa, teniendo ambos un valor de la métrica I de 0.5. El principio de desarrollo software de dependencias estables [48] establece que la métrica I de un paquete debe ser mayor que las métricas I de los paquetes de los que depende. En nuestro sistema, el submódulo de Programa depende del submódulo de *Pipeline*, ambos con el mismo valor de la métrica I. Esto puede indicar que no se está cumpliendo del todo el

Módulo	***Afferent Couplings***	***Efferent Couplings***	***I***
Registros	3	0	0
Memoria	3	0	0
Programa	1	1	0.5
Compilador	2	0	0
Pipeline	1	1	0.5
Instrucciones	3	2	0.4

Tabla 5.1: Estabilidad de los paquetes del módulo de interfaces de las arquitecturas. *Afferent Couplings* (C_a) es el número de clases fuera del paquete que dependen de clases dentro del paquete. *Efferent Couplings* (C_e) es el número de clases de dentro del paquete que dependen de clases de fuera del paquete. I es una medida de la estabilidad del paquete, definida como $I = \frac{C_e}{C_a+C_e}$. Un valor de I de 0 indica que la estabilidad del paquete es máxima. Un valor de I de 1 indica que la inestabilidad del paquete es máxima.

principio de dependencias estables. Sin embargo, el número de dependencias que presentan los dos módulos implicados no son demasiadas. El bajo número de dependencias de los dos módulos reduce la dificultad de cambio asociada a la relajación del cumplimiento del principio de dependencias estables entre ambos módulos. No obstante, podría tratarse de un posible punto de mejora en el diseño del sistema, y ha de tenerse en cuenta durante la implementación del mismo.

5.3.2. Interfaz gráfica

Para el diseño de la interfaz gráfica, se decidió tomar como modelo la interfaz gráfica de un simulador ya existente. En concreto, si usó como modelo la interfaz gráfica del simulador *MARS* [13], el simulador utilizado hasta el momento en las asignaturas de Arquitectura de Computadores de la Universidad de Valladolid. La interfaz de ese simulador presenta de manera eficiente e intuitiva una gran cantidad de información de utilidad didáctica sobre la simulación de programas ensamblador. Aun así, nuestro simulador realiza algunas mejoras y modificaciones sobre la interfaz de *MARS*. La influencia de *MARS* en el desarrollo y diseño del simulador se detalla en el apéndice D.

Al utilizar una interfaz gráfica ya existente como modelo para el diseño de la interfaz gráfica del simulador, la complejidad de ese proceso de diseño se reduce considerablemente. Entre otros, se reduce la necesidad de realizar prototipos iniciales de baja fidelidad [49] hasta eliminarse prácticamente. También se reduce el número de iteraciones necesarias para finalizar el diseño de la interfaz gráfica.

De acuerdo con el requisito RF12 (detallado en la sección 5.1.1, el simulador debe presentar un editor de texto al usuario. De acuerdo con el requisito RF2, el simulador también debe presentar información al usuario sobre el estado de la CPU durante la simulación de los programas. La información concreta que debe presentarse se detalla en los requisitos RF3, RF4, RF5, RF6 y RF17. Entonces, distinguimos dos modos dentro de la interfaz gráfica del simulador: el modo de edición y el modo de simulación, en el que se presenta la información de la CPU. Los modos se presentarán como pestañas entre las que el usuario podrá cambiar a voluntad, siendo la pestaña por defecto la de edición de texto.

En todo momento e independientemente del modo en uso, la parte superior de la ventana la interfaz gráfica presenta un conjunto de botones. Estos botones permiten:

- La creación de un nuevo programa en blanco.
- La apertura de un fichero de código fuente de programa ensamblador.
- El guardado del código fuente del programa actualmente en edición.
- El guardado del código fuente del programa actualmente en edición, en modo "guardar como" (es decir, preguntando siempre con qué nombre quiere guardarse el programa, en lugar de usando el nombre actual de manera automática).
- La compilación del programa actualmente en edición, para permitir su ejecución por el simulador.
- La ejecución completa del programa compilado (de estar el programa compilado).
- La ejecución del siguiente ciclo de reloj del programa compilado (de estar el programa compilado).
- El reinicio del programa compilado (de estar el programa compilado).

Al lado de estos botones se encuentra un panel desplegable que permite cambiar la arquitectura actual del simulador por cualquier otra que se encuentre disponible. El diseño de esta barra superior puede observarse en las figuras 5.4 y 5.7.

La figura 5.4 ilustra el diseño de la pestaña de edición de la interfaz gráfica. Esta pestaña contiene exclusivamente un editor de código ensamblador. El editor de código presenta características que favorecen la escritura de código ensamblador. Entre estas características destacamos la indicación de los números de línea en la parte izquierda del editor, así como el resaltado de sintaxis. El resaltado de sintaxis modifica el formato del texto escrito, incluyendo su color y tipo de letra, para destacar ciertos elementos del código: los mnemónicos de las instrucciones, los registros utilizados, las etiquetas de memoria, etc.

El editor también presenta un sistema de sugerencias que ayuda al aprendizaje del conjunto de instrucciones de la arquitectura. Este sistema da sugerencias de instrucciones al usuario según el mnemónico que esté escribiendo. Estas sugerencias cuentan con una breve descripción de la instrucción explicando su funcionamiento. Del mismo modo, una vez el usuario ha escrito un mnemónico por completo, el editor presenta un ejemplo de uso de la instrucción escrita. Estos conceptos se ilustran en las figuras 5.5 y 5.6, respectivamente.

Si se trata de compilar un código que presente fallos sintácticos, aparece un cuadro de texto en la parte derecha de la ventana indicando los fallos encontrados y se cancela la compilación. Los programas ensamblador se componen por sentencias generalmente cortas, por lo que el código ensamblador suele presentar una anchura muy reducida. Por otra parte, los programas ensamblador suelen ser relativamente extensos, puesto que cada línea solo puede codificar una única operación de bajo nivel. Es por estas dos características que se ha decidido que la localización del cuadro de texto de fallos de compilación aparezca a la derecha del editor de texto, en vez de debajo, como

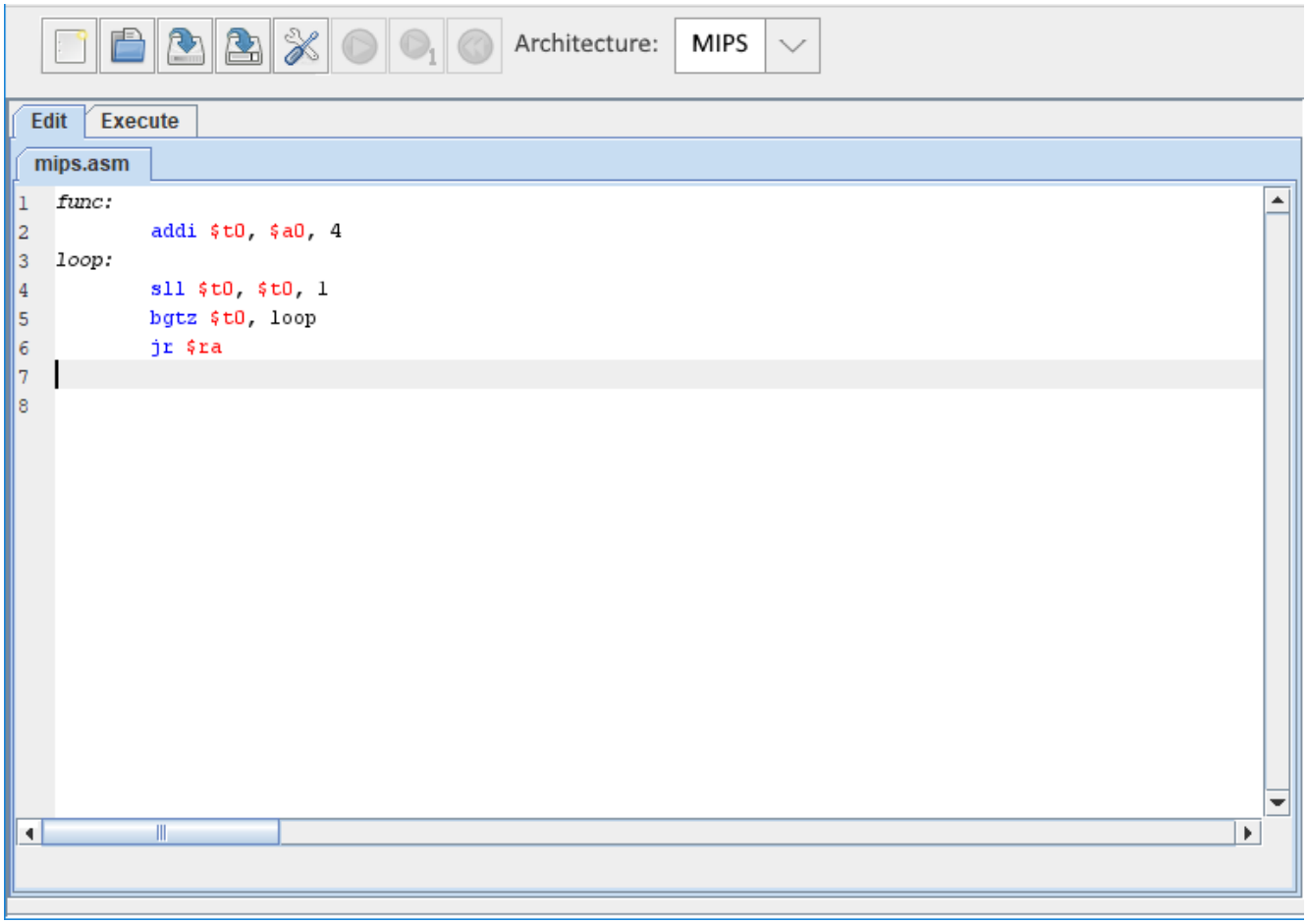

Figura 5.4: Prototipo de la pestaña de edición de texto de la interfaz gráfica del simulador. En esta pestaña el usuario puede editar sus programas ensamblador. Los botones de la parte superior de la ventana permiten, entre otros, abrir ficheros, guardar ficheros, y compilar el fichero en edición, para su simulación.
La figura muestra las capacidades de indicación de números de línea y de resaltado de sintaxis del editor. Para la demostración se utiliza código ensamblador MIPS.

podría parecer natural: la cantidad de información que queda oculta por la aparición del cuadro de texto es menor si este se sitúa verticalmente paralelo al código, en vez de horizontalmente paralelo.

La figura 5.7 ilustra el diseño de la pestaña de ejecución del simulador. Esta pestaña presenta información relativa a diversos aspectos de la simulación y el programa. En concreto, se presenta la información relativa a los siguientes aspectos:

- El contenido de los registros en cada momento.
- El contenido de la memoria en cada momento.
- Las banderas de estado de la CPU en cada momento.
- Las etiquetas de memoria definidas en el programa.
- El preprocesado del programa y la compilación del programa a código máquina.

Figura 5.5: Prototipo del sistema de sugerencias de instrucciones para el editor de código del simulador.

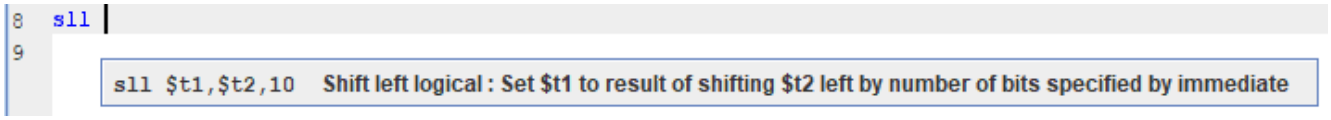

Figura 5.6: Prototipo del sistema de ejemplos de instrucciones para el editor de código del simulador.

- Los mensajes de salida del programa.

Como resumen de la interfaz gráfica, la figura 5.8 presenta el diagrama de componentes principales de la interfaz gráfica del sistema. Las clases derivadas de cada uno de estos componentes dependen de la tecnología concreta utilizada para implementar la interfaz gráfica. Por tanto, se considera que carecen de interés en diseño sino en implementación, por lo que no se detallan en este capítulo.

5.3.3. Diseño de la ejecución en modo terminal

Según el requisito RF16, el simulador debe poder ejecutarse en modo terminal. Este modo de ejecución prescinde de la interfaz gráfica, permitiendo solamente la ejecución de programas completos. El sistema utiliza este modo de ejecución automáticamente si detecta que se le ha proporcionado algún argumento de ejecución al ser invocado a través de una terminal. El primero de estos argumentos se interpreta como la ruta al fichero ensamblador que debe ejecutarse, por lo que el sistema trata de abrir el fichero asociado, compilarlo y ejecutarlo.

Para añadir cierta flexibilidad a este modo de ejecución, se permitirá la configuración de ciertos parámetros de ejecución a través de los argumentos de entrada que reciba el programa. Entre otros, se permite configurar:

- La arquitectura utilizada para compilar y ejecutar el programa. Esta será LEGv8 por defecto.
- La impresión por salida estándar del contenido de los registros al finalizar la ejecución.

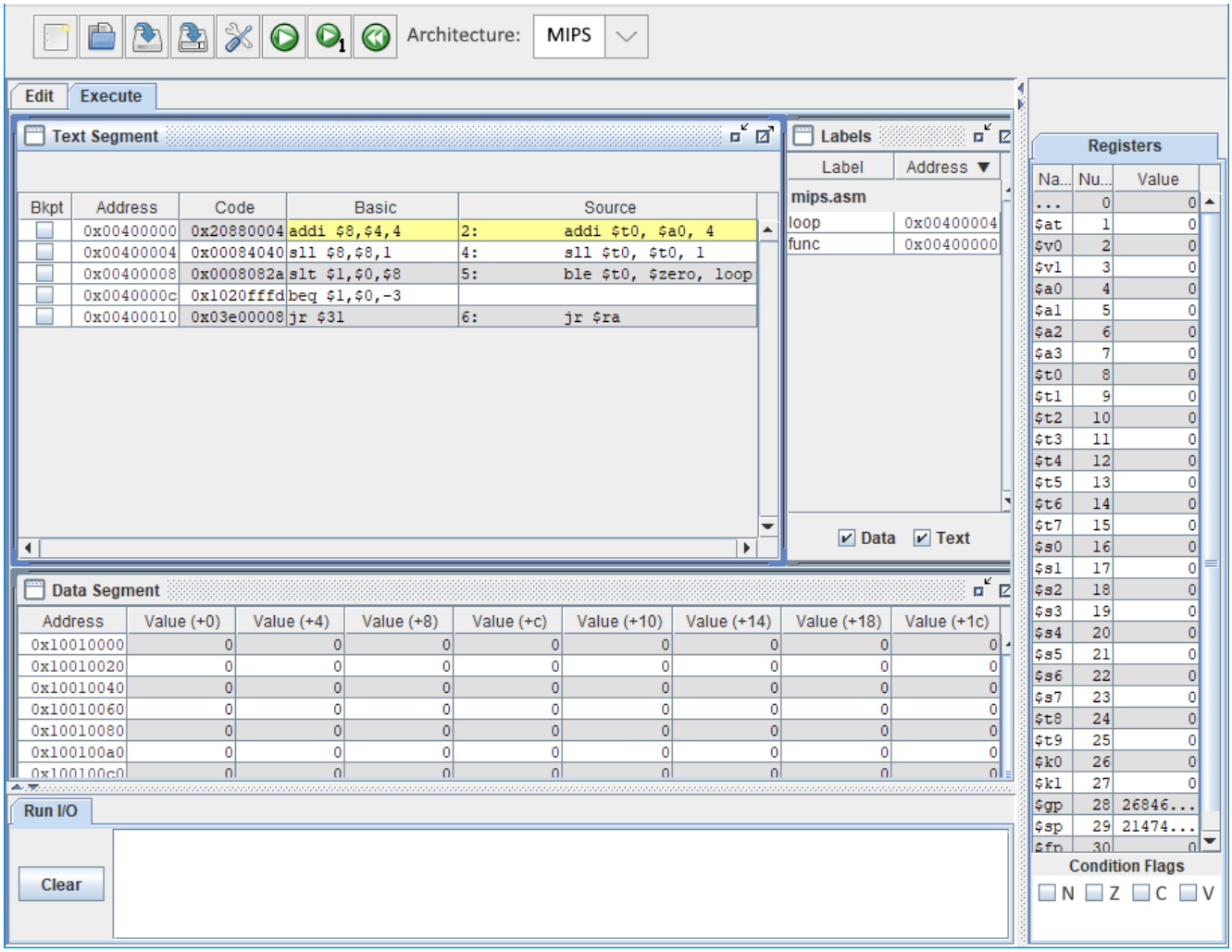

Figura 5.7: Prototipo de la pestaña de edición de texto de la interfaz gráfica del simulador. En esta pestaña el usuario puede consultar información relativa a: los registros, la memoria, las banderas de estado, las etiquetas de memoria, la compilación del programa y los mensajes de salida que genera el programa.

- La impresión por salida estándar del contenido de la memoria al finalizar la ejecución.
- La impresión por salida estándar de las etiquetas de memoria del programa al finalizar la ejecución.
- La impresión por salida estándar del código preprocesado del programa.
- La impresión por salida estándar del código máquina del programa.

De estos aspectos configurables, el que puede llevar asociada una mayor complejidad es el primero: cargar una arquitectura de las disponibles en el simulador. Para realizar esta tarea, se diseña una nueva clase en el sistema:

- Cargador de arquitecturas: esta clase realiza la gestión de las arquitecturas disponibles en el simulador. Es una clase puramente estática; la funcionalidad que ofrece no requiere su

Figura 5.8: Resumen de componentes principales de la interfaz gráfica del sistema.

instanciación.
Esta clase realiza dos operaciones:

- loadArch(nombre : String) : Arquitectura - devuelve una instancia de la arquitectura cuyo nombre se provee por parámetro. El nombre provisto por parámetro debe corresponder al de una arquitectura disponible en el simulador. En otro caso, la invocación de la función produce un error.
- getAvailableArchs() : String[*] - devuelve el nombre de todas las arquitecturas actualmente disponibles en el simulador. Los nombres devueltos son aquellos que se pueden proporcionar como parámetro al método *loadArch(String)* sin que produzca un fallo.

El modo terminal del simulador utiliza el método loadArch(String) para obtener una instancia de la Arquitectura deseada. Con esta instancia de Arquitectura, se genera una instancia de Programa, se ejecuta, y se imprimen por salida estándar la información especificada en los argumentos de ejecución del simulador. El nombre de arquitectura provisto como parámetro de loadArch(String) se obtiene también de los argumentos de ejecución del simulador, o se utiliza "LEGv8" por defecto.

El método getAvailableArchs() se define ya que resulta de utilidad a la hora de presentar información en la interfaz gráfica del sistema. Este método resulta de utilidad, concretamente en el panel desplegable de cambio de arquitectura. No es estrictamente necesario delegar la operación de obtención de nombres de arquitecturas a una clase particular, por lo que no se especificó

en el diseño de la interfaz gráfica. Sin embargo, es buena práctica hacerlo, sobre todo si se puede agrupar con otras funciones relacionadas con la carga de arquitecturas. Aprovechando la necesidad de definir una clase con el método loadArch(String) para el modo terminal del sistema, se delega esta funcionalidad de la interfaz gráfica a la misma clase.

La figura 5.9 resume en forma de diagrama el funcionamiento de la clase Cargador de arquitecturas.

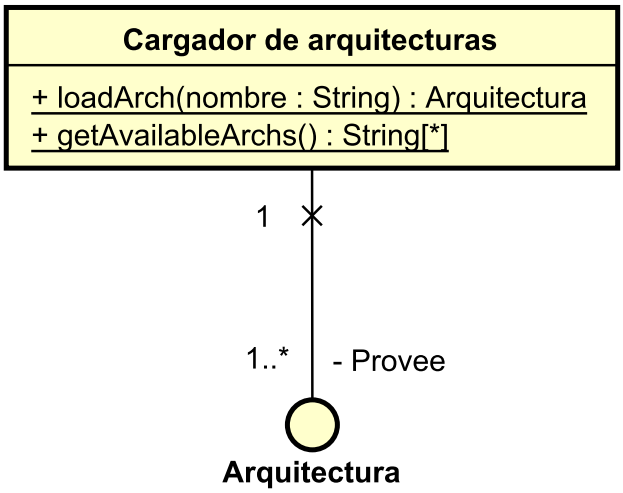

Figura 5.9: Diagrama de funcionamiento del cargador de arquitecturas. Esta clase estática es capaz de proveer los nombres de todas las Arquitecturas disponibles en el sistema, así como instancias de arquitecturas concretas a partir de sus nombres.

5.3.4. Módulos de arquitecturas: LEGv8

La función de los módulos de arquitecturas del sistema es la de realizar las interfaces definidas en el módulo de interfaces de las arquitecturas, proporcionando los sistemas necesarios para simular una arquitectura de computadores y lenguaje ensamblador concreto. A nivel de diseño, los distintos módulos de arquitecturas son similares al módulo de interfaces de las arquitecturas detallado en la sección 5.3.1. Los módulos de arquitecturas deben presentar clases similares que realicen, por lo menos, las mismas operaciones. En cierto sentido, la figura 5.3.1, que presenta el modelo de diseño del módulo de interfaces de arquitecturas del sistema, presenta también un diseño genérico para cualquier módulo de arquitecturas del sistema.

Evidentemente, el diseño de un módulo de arquitectura específico puede variar con respecto al diseño del módulo de interfaces de arquitecturas. Sin embargo, estos cambios no tienen por qué ser muy significativos. Los cambios principales son:

- Especificar los constructores y destructores de las clases que realizan las interfaces del módulo de interfaces de las arquitecturas.

- Añadir los métodos estáticos de obtención de instancias de las clases que siguen el patrón de diseño *singleton*.
- Opcionalmente, añadir los atributos de las clases que realizan las interfaces del módulo de interfaces de las arquitecturas. Este cambio es opcional debido a que, según el principio de acceso uniforme del desarrollo software [50], el acceso a una propiedad de una clase debe ser indistinguible independientemente de si se trata de un dato calculado o almacenado. En un módulo de arquitectura bien diseñado, el acceso a las propiedades de una clase deberían realizarse a través de sus métodos públicos, siendo sus atributos privados.
- Añadir las clases del módulo de compilador, accedidas por la fachada Cargador de programa. Sin embargo, es posible que la funcionalidad accedida por esta fachada sea generada de forma automática por una tecnología específica de desarrollo de compiladores. En este caso, la definición en diseño de estas clases carece de sentido, debido a que son dependientes de la implementación concreta. La fachada Cargador de programa actuaría como una caja negra que compila los programas textuales a programas ensamblador, accediendo de forma no detallada a las funcionalidades de la tecnología elegida.

En este proyecto se desarrolla el módulo de la arquitectura LEGv8, ya que, según el requisito RF1, el sistema debe permitir la simulación de programas ensamblador LEGv8. Para diseñar esta arquitectura, se parte del modelo de diseño del módulo de interfaces de las arquitecturas (figura 5.3.1), y se realizan los cambios que se han explicado anteriormente: especificación de constructores, destructores y de métodos de acceso a instancias *singleton*.

En muchos casos es posible derivar los constructores y destructores de las clases a partir de los métodos de acceso a propiedades de las mismas. Al constructor de una clase deben proveerse de manera explícita los valores de las propiedades no constantes de la clase que son devueltas por métodos de clase (*getters*), pero no son especificados mediante métodos de clase (*setters*). Estas son las propiedades no constantes que, en clases que no siguen el patrón de diseño *singleton*, tienen un método de acceso asociado (de tipo *getter*), pero no uno de escritura o modificación (de tipo *setter*). Las clases de tipo *singleton* no poseen constructores (públicos), generando en su instanciación las propiedades accesibles pero no modificables de manera automática.

Además de estos cambios, la arquitectura LEGv8 añade dos clases a las propuestas por el módulo de interfaces de las arquitecturas. Estas clases, localizadas en el módulo de instrucciones LEGv8, son:

- Código de ejecución. Se trata de una interfaz que sirve para definir las operaciones o funcionalidades de las instrucciones de la arquitectura. Se relaciona con la clase de Instrucción LEGv8, proveyendo la funcionalidad asociada a su operación *execute(Instrucción de programa)*.
  Esta clase no presenta utilidad en diseño, pero sí durante la implementación del simulador. Permite la definición de clases anónimas que especifiquen las funcionalidades de las instrucciones ensamblador de la arquitectura. Entre los beneficios de esta aproximación se encuentra una considerable reducción de la complejidad de implementación. La alternativa consistiría en definir el método *execute(Instrucción de programa)* de la clase Instrucción como abstracto, y forzar la definición de una clase que herede de Instrucción y concrete este método por cada instrucción distinta existente. Esta alternativa, aunque en el fondo resulte similar, es

bastante indeseable dada la complejidad conceptual y de implementación asociada a la gran cantidad de clases concretas que requiere definir.
Esta clase realiza una única operación:

- *execute(instancia : Instrucción de programa) : void* - ejecuta la instrucción de programa provista por parámetro. Este método requiere que la instrucción asociada al código de ejecución sea la misma que la instrucción asociada a la instrucción de programa provista por parámetro.

- Ejecución condicional. Se trata de una interfaz que sirve para definir la condición bajo la que se ejecuta una instrucción. Esta clase se relaciona con la clase de Instrucción LEGv8. Se utiliza en conjunto con la instrucción de bifurcación condicional `B.cond`. Esta instrucción puede producir una bifurcación en el flujo de instrucciones del programa en función de los valores de las banderas de estado de la CPU.
  De nuevo, esta clase no presenta utilidad en diseño, sino en implementación. Su uso permite la definición de clases anónimas para reducir la complejidad de implementación de todas las variantes de la instrucción `B.cond` de la arquitectura.
  Esta clase realiza una única operación:

  - *check() : void* - comprueba si la condición de ejecución de una instrucción se cumple o no.

La figura 5.10 muestra un diagrama UML ilustrando el funcionamiento de estas dos clases.

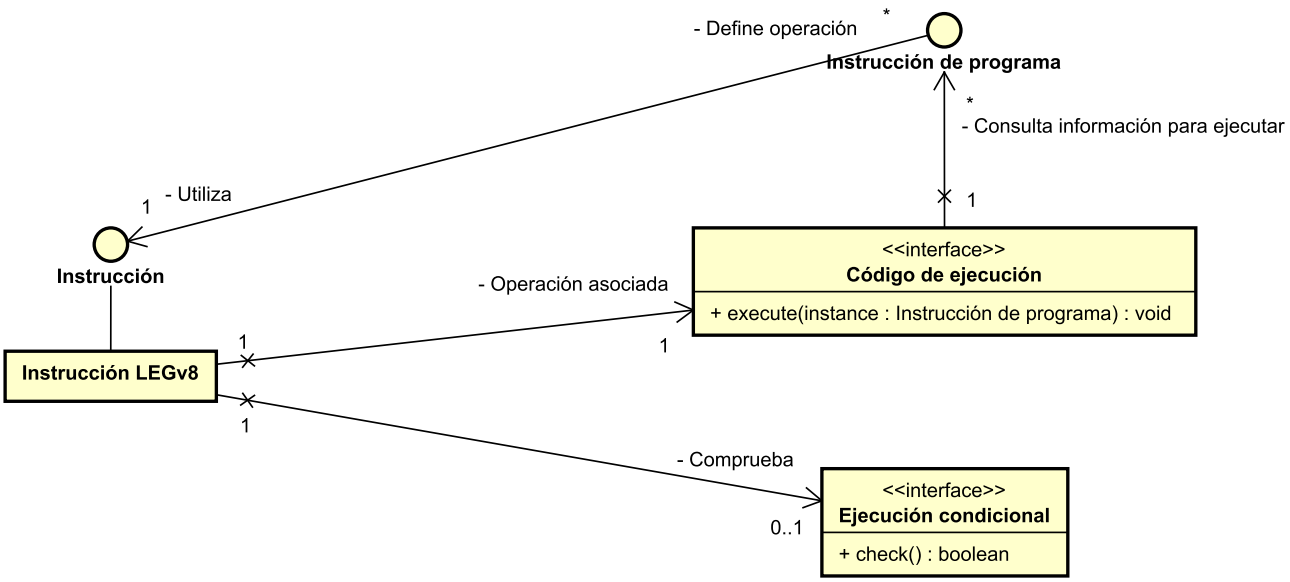

Figura 5.10: Diagrama de clases añadidas en el módulo de arquitectura LEGv8. El diagrama muestra también las relaciones de las clases Código de ejecución y Ejecución condicional con las clases Instrucción e Instrucción de programa, del módulo de interfaces de las arquitecturas.

Mas allá de los cambios ya especificados y las dos clases adicionales, no existen diferencias en diseño entre el módulo de la arquitectura LEGv8 y el módulo de interfaces de las arquitecturas. Es posible que en implementación se definan más operaciones en las clases del módulo de la arquitectura LEGv8. Estas operaciones adicionales pueden surgir principalmente para facilitar

las labores de programación, siendo cambios de "calidad de vida" (*Quality of Life*); o tratarse de métodos privados que ayudan a reorganizar el código para reducir su complejidad. En ambos casos, no se considera que estas operaciones surjan por necesidades de diseño del sistema, siendo en su lugar definidas específicamente para apoyar labores de implementación. Se ha decidido relajar el diseño del modelo para que no sea necesario reflejar estas características.

En general, el módulo de interfaces de las arquitecturas se ha diseñado de tal manera que se especifican las operaciones que deben realizar las clases de los módulos de arquitectura, dejando suficiente libertad para que estas operaciones se implementen sin grandes limitaciones. En otras palabras, sirve como guía para implementar las arquitecturas concretas del simulador, sin limitar en exceso cómo puedan implementarse. Su diseño puede servir como base para diseñar las arquitecturas concretas, sin requerir muchos cambios o añadidos.

5.4. Casos de uso

Para ejemplificar el uso del simulador, en esta sección se recogen e ilustran los principales casos de uso del mismo. La figura 5.11 muestra el diagrama de casos de uso del simulador.

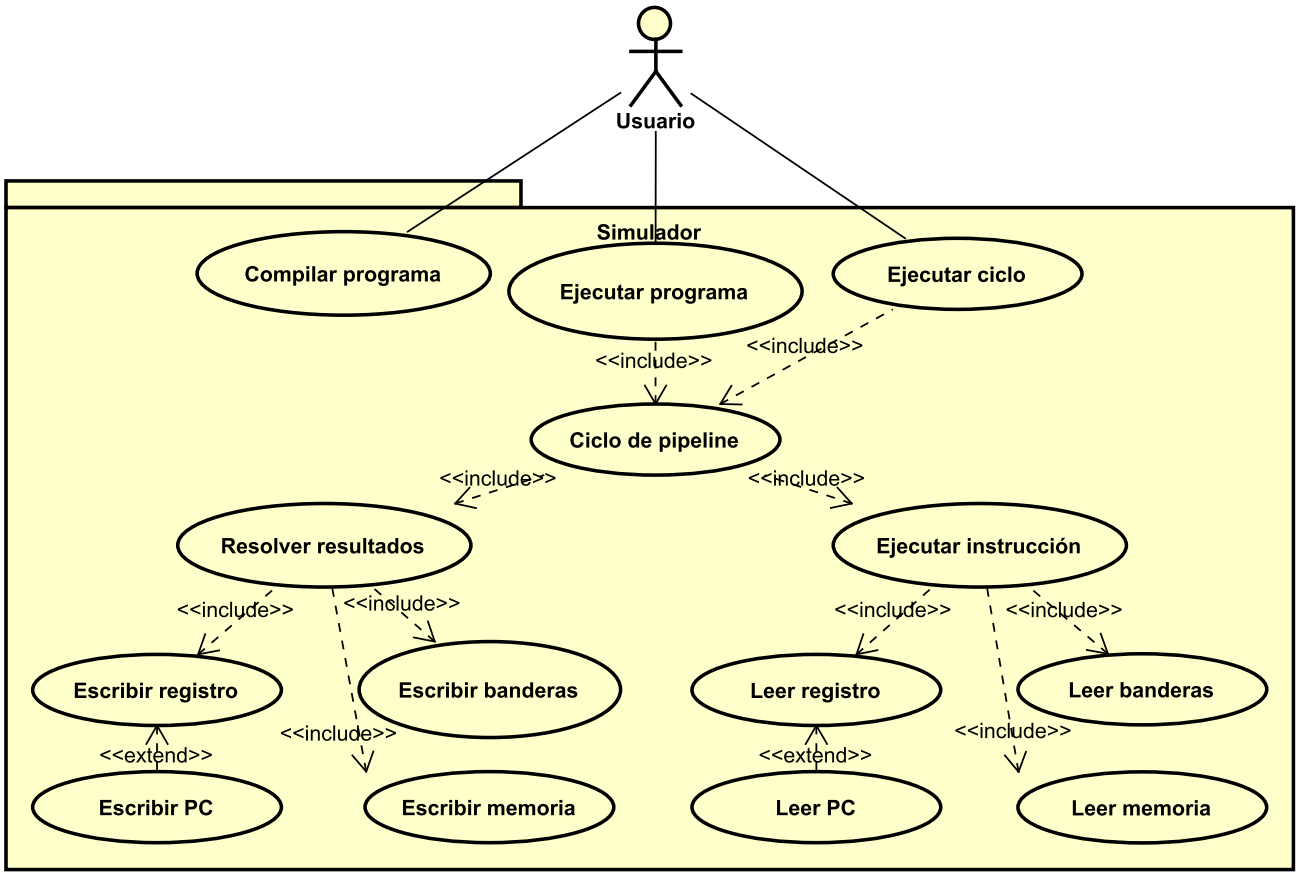

Figura 5.11: Diagrama de casos de uso del simulador.

El usuario tiene acceso a los casos de uso Compilar programa, Ejecutar ciclo y Ejecutar programa, a través de los botones de la interfaz gráfica del simulador. Los diagramas de secuencia para la ejecución general de estos tres casos de uso se muestran en las figuras 5.12, 5.13 y 5.14, respectivamente.

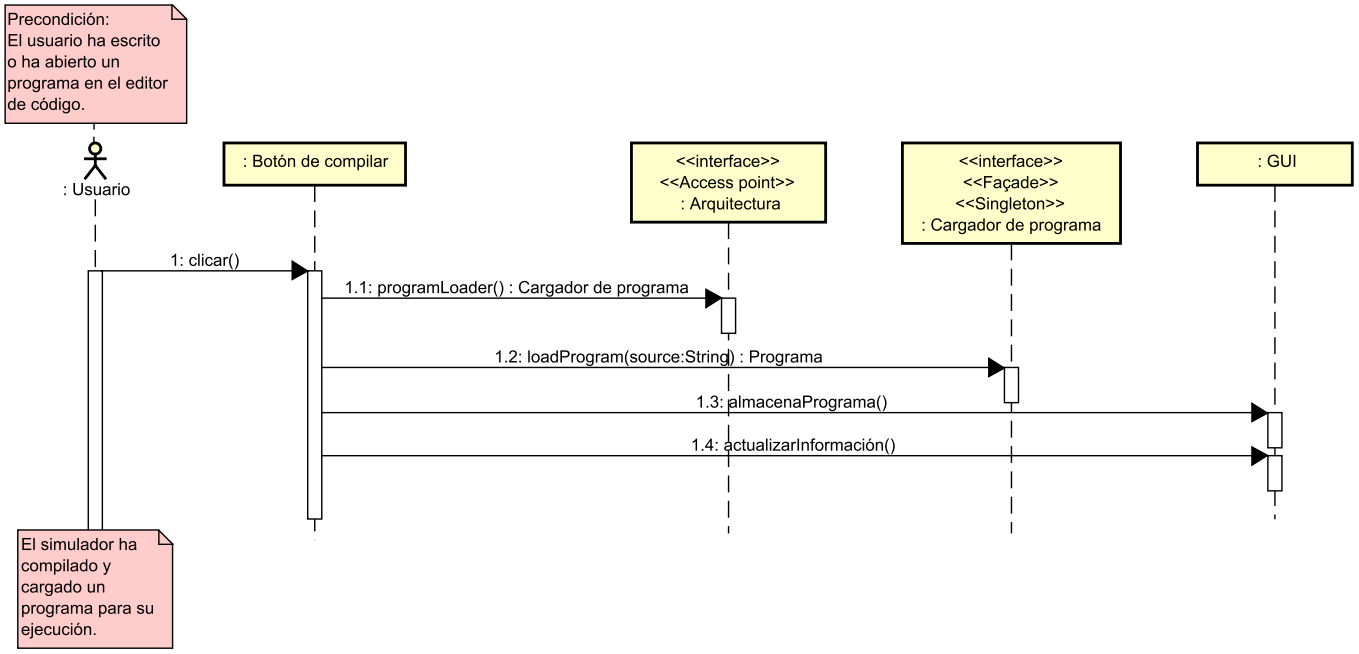

Figura 5.12: Diagrama de secuencia para el caso de uso Compilar.

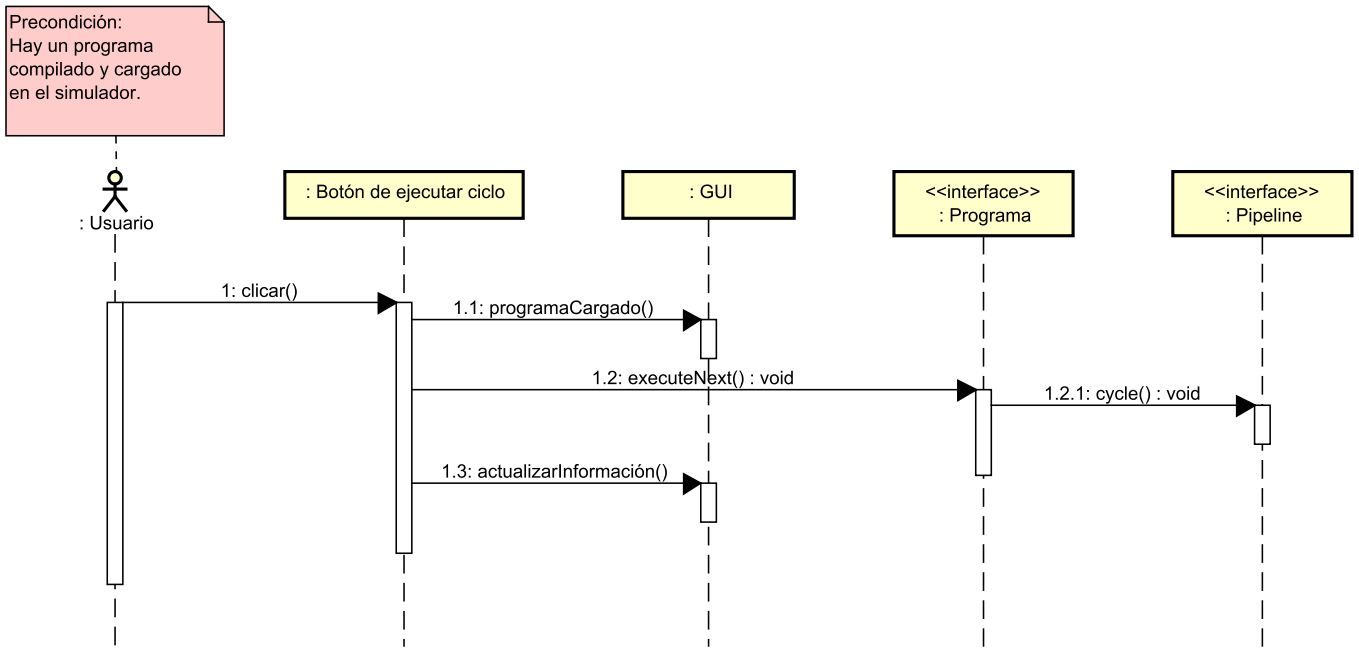

Figura 5.13: Diagrama de secuencia para el caso de uso Ejecutar ciclo.

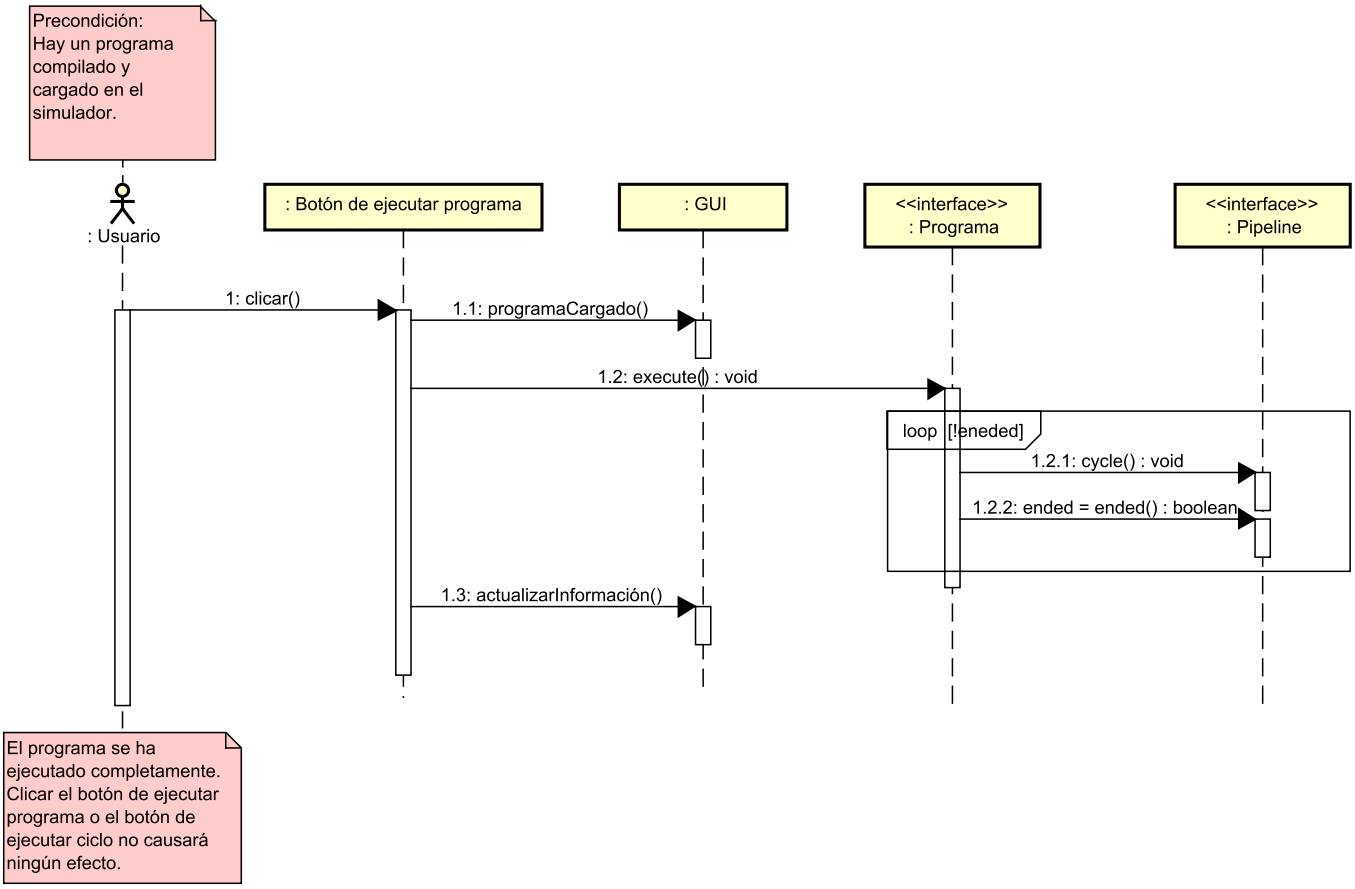

Figura 5.14: Diagrama de secuencia para el caso de uso Ejecutar programa.

Tanto el caso de uso Ejecutar ciclo como Ejecutar programa incluyen el caso de uso Ciclo de *pipeline*. Ese es uno de los casos de uso más importantes del simulador, pues en él se produce la ejecución de las instrucciones de un programa, y la resolución de sus resultados, modificando el estado de la CPU. El diagrama de secuencia de este caso de uso se puede observar en la figura 5.5.

5.5. Resumen

En este capítulo se ha detallado el proceso de diseño del simulador. Entre las actividades comprendidas en este proceso de diseño se encuentran las siguientes:

- La especificación de requisitos, tanto funcionales como no funcionales.
- El diseño de la arquitectura del sistema.
- El diseño de las clases de sistema contenidas en cada submódulo de la arquitectura del sistema.
- El diseño de la interfaz gráfica del sistema.
- El diseño del módulo de la arquitectura LEGv8.

- La descripción de los casos de uso del sistema.

De este capítulo se concluye lo siguiente:

- El simulador se divide en dos módulos, el *frontend* y el *backend*, que a su vez se dividen en submódulos.
- El *frontend* define la interfaz gráfica del sistema y las interfaces que deben realizar las arquitecturas concretas para ser soportadas por el simulador.
- El *backend* define las arquitecturas concretas soportadas por el simulador.
- El diseño de la interfaz gráfica del sistema parte de la interfaz gráfica del simulador *MARS*, sobre la que se realizan ciertos cambios.
- El sistema puede ejecutarse en modo terminal, prescindiendo de interfaz gráfica. Para ello, se le debe proporcionar la ruta de un programa ensamblador a ejecutar, como argumento al invocar el programa a través de una terminal. Otros argumentos pueden utilizarse para determinar la arquitectura a utilizar o la información que imprimir por salida estándar.
- El diseño de los módulos de arquitecturas concretas es similar al diseño del módulo de interfaces de las arquitecturas. Estos módulos se dividen en los siguientes submódulos:
  - Módulo de registros.
  - Módulo de memoria.
  - Módulo de instrucciones.
  - Módulo de *pipeline*.
  - Módulo de programa.
  - Módulo de compilador.

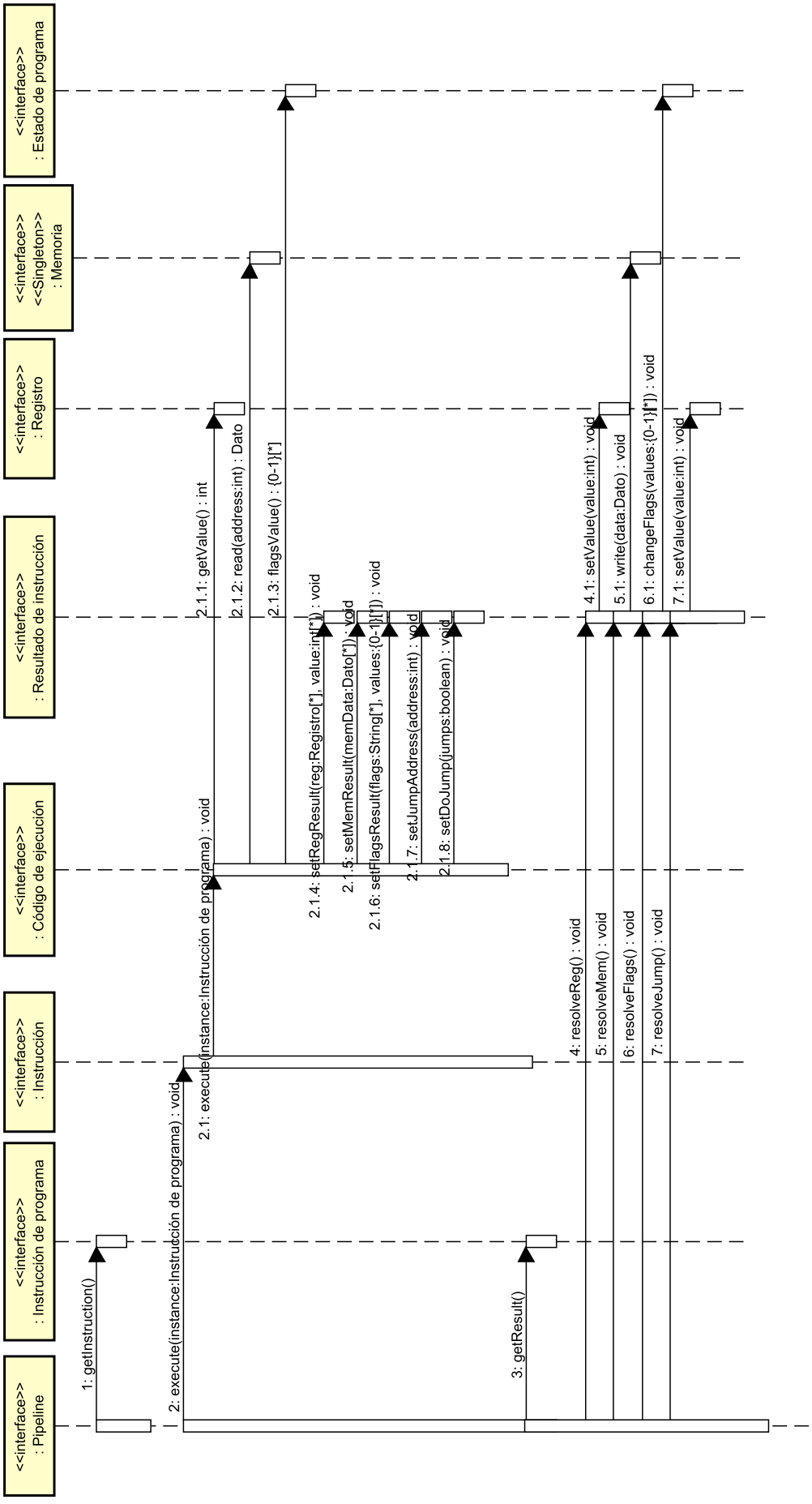

Figura 5.15: Diagrama de secuencia para el caso de uso Ciclo de *pipeline*.

Capítulo 6

Implementación del simulador

En este capítulo se presentan los siguientes aspectos relacionados con la implementación del simulador:

- La descripción de las tecnologías y herramientas utilizadas para desarrollar la implementación del simulador.
- La descripción de la implementación del simulador.

6.1. Descripción de tecnologías y herramientas utilizadas

El requisito RNF1 (detallado en la sección 5.1.2 de este documento) especifica que el simulador debe ser multiplataforma, por lo que se decidió desde un principio que el lenguaje de programación utilizado para implementar el sistema sería Java [51]. Java es un lenguaje de programación de alto nivel, tipado y fuertemente orientado a objetos. Es uno de los lenguajes de programación más populares a día de hoy, siendo comúnmente utilizado en asignaturas de fundamentos de programación y programación orientada a objetos. Es un lenguaje con el que el programador estaba familiarizado, por lo que no fue necesario dedicar tiempo a su aprendizaje.

Con respecto a la versión específica de Java que se utiliza, en un principio se trató de utilizar Java 8, ya que es la versión de Java más utilizada a fecha de 2021. Se pretendía utilizar esta versión para no crear dependencias de versiones posteriores, evitando la necesidad de actualizar Java en los sistemas que pretendiesen utilizar el software. Sin embargo, ciertos componentes del proyecto requieren el uso de Java 11 para funcionar, por lo que esa es la versión de Java finalmente utilizada.

El entorno de desarrollo (IDE) utilizado para desarrollar el software del proyecto es IntelliJ IDEA Community [52]. IntelliJ IDEA es un entorno de desarrollo que presenta una gran flexibilidad. Ofrece grandes facilidades para el desarrollo de proyectos de código Java, como sistemas de autocompletitud de código, navegación de código, refactorización de código, depuración de código,

una terminal de comandos propia e integración con *git*, entre otros. Además, admite la adición de extensiones que amplían las funcionalidades que ofrece el IDE.

Para gestionar las dependencias del proyecto software y automatizar las construcciones (*builds*) se utiliza Gradle [53]. Gradle es un sistema de automatización de construcción de código software, y se encuentra entre los sistemas de automatización de construcción de código soportados de manera nativa por IntelliJ IDEA.

Para el desarrollo del compilador del módulo de la arquitectura LEGv8 se utiliza ANTLR 4 [54]. ANTLR es un generador de analizadores sintácticos (*parsers*). Para ello, toma una gramática libre de contexto que especifica un lenguaje a reconocer, y genera el código fuente para reconocer ese lenguaje. ANTLR es capaz de generar código para varios lenguajes de programación, siendo notablemente usado para generar *parsers* en Java. ANTLR es capaz de generar analizadores léxicos (*lexers*), analizadores sintácticos (*parsers*) e incluso analizadores sintáctico-léxicos (*lexer-parsers*), a partir de un único fichero fuente. Ofrece una notación unificada para definir los *lexers* y *parsers*, lo que facilita considerablemente el desarrollo de compiladores.

Para la implementación de la interfaz gráfica del sistema se utiliza JavaFX [55]. JavaFX es una plataforma software para crear aplicaciones con entornos gráficos para gran variedad de sistemas operativos. Pretende ser una novedad sobre Java Swing [56] y AWT [57], ofreciendo un mayor número de características y una mayor facilidad y flexibilidad de desarrollo. Desde Java 11, se incluye como parte del proyecto OpenJDK de Oracle, por lo que actualmente está integrado en la mayoría de instalaciones de Java. Esta herramienta es la que obliga a que la versión de Java en la que se desarrolla el sistema sea Java 11.

Para favorecer la corrección y la autodocumentación del código fuente del sistema, se decidió que se programaría utilizando Contratos para Java (*cofoja*) [58]. Cofoja es una infraestructura de programación mediante contratos que utiliza anotaciones Java para proporcionar comprobaciones de contratos en tiempo de ejecución. Un contrato [59] software define unas precondiciones y postcondiciones para la invocación de funciones en un sistema. Una función asegura su correcta ejecución si se cumplen las precondiciones que establece. También, la función asegura que, tras su ejecución, se cumplen las postcondiciones que establece. Una anotación Java es una forma de añadir metadatos sintácticos a código fuente Java. Las anotaciones Java pueden ser tenidas en cuenta por el compilador a la hora de generar el ejecutable del programa, alterando su funcionamiento. Cofoja ofrece una forma simple de establecer contratos en programas Java, favoreciendo la corrección del programa, y facilitando la comprensión de su código fuente.

Además de Cofoja, también se utilizan las anotaciones de código provistas por el paquete `org.jetbrains.annotations`, que se incluye con el entorno de desarrollo de IntelliJ IDEA. En concreto, siempre que se considera apropiado se utilizan las anotaciones `@Nullable` y `@NotNull` para denotar que el valor de una variable puede o no puede ser `null`, respectivamente.

También, para asegurar una calidad mínima del código fuente del sistema, se impuso el cumplimiento de las convenciones de código Java [60], exceptuando los casos en los que estas entrasen en conflicto con el dominio de la aplicación[1]. Aunque actualmente no se trate de una convención

[1]Por ejemplo: las convenciones de Java establecen que los nombres de variables deben comenzar por minúscula; pero los registros en una instrucción de programa LEGv8 se llaman *Rd*/*Rt*/*Rn*/*Rm*. Preferimos utilizar los nombres de variables más parecidos al concepto que se está simulando, en vez de cumplir las convenciones de Java.

de código activamente promovida, también se estableció que no debían superarse los 80 caracteres por línea en todos los ficheros del código fuente (exceptuando los generados automáticamente por herramientas como ANTLR).

Por lo general, también se siguieron principios y buenas prácticas de desarrollo de software comúnmente promovidos, como los 5 principios SOLID [61]. En algunas ocasiones, estos principios se incumplieron de forma consciente para reducir la complejidad del código. Es el caso del principio de acceso uniforme, por ejemplo. No obstante, se minimizó el incumplimiento de estas prácticas y principios, permitiéndose exclusivamente en situaciones debidamente justificadas.

Para promover de forma activa el cumplimiento de otras convenciones de código y buenas prácticas que aseguraran un nivel de calidad mínimo en el código fuente, se utilizó la extensión de SonarLint [62] para IntellIJ IDEA. SonarLint es una extensión para diferentes IDEs que detecta de forma automática problemas de calidad de código mientras se escribe el propio código. Categoriza los problemas según su impacto en la calidad del código y ofrece alternativas y soluciones automatizadas para ellos. Es, a su vez, una extensión adaptable, permitiendo la desactivación de las reglas de calidad específicas que no se ajusten al proyecto.

6.2. Implementación de la propuesta

En esta sección se trata la implementación de los componentes principales que forman el simulador. La implementación del simulador parte del diseño detallado en la sección 5.3.

Debido a la complejidad del sistema desarrollado, no se puede describir con detalle todas las funciones que son utilizadas en el sistema. En su lugar, se describe de forma general las principales estructuras y funcionalidades del simulador, desde la perspectiva del modelo de dominio, y centrándose en la arquitectura LEGv8. Para una descripción más detallada del funcionamiento y la implementación de los componentes del sistema, se recomienda consultar el código fuente del simulador y su documentación *Javadoc*.

En esta sección se asume que el lector está familiarizado hasta cierto punto con la programación en Java, conociendo al menos los conceptos, la terminología, y las clases básicas del lenguaje.

6.2.1. Consideraciones iniciales de la implementación

El requisito RNF5 especifica que el código fuente del simulador debe estar escrito en inglés. Por ello, en la implementación del simulador, los nombres de los módulos y las clases descritas en la sección 5.3 se traducen al inglés. La traducción utilizada para los nombres de los módulos y sus clases del sistema es la siguiente:

- **Registros**: *registers*.
    - Tipo de Registro: *RegisterType*.

  - Registro: *Register*.
  - Banco de registros: *RegisterFile*.
- **Memoria**: *memory*.
  - Dato: *Data*.
  - Sección de memoria: *Section*.
  - Memoria: *Memory*.
- **Instrucciones**: *instructions*.
  - Resultado de instrucción: *InstructionResult*.
  - Instrucción: *Instruction*.
  - Conjunto de instrucciones: *InstructionSet*.
  - Instrucción de programa: *ProgramInstruction*.
  - Código de ejecución: *ExecutionCode*.
- ***Pipeline***: *pipeline*.
  - Etapa del *pipeline*: *PipelineStage*.
  - Política de predicción de saltos: *PredictionPolicy*.
  - Unidad de salto: *JumpUnit*.
  - Unidad de operaciones: *OperationUnit*.
  - *Pipeline*: *Pipeline*.
- **Programa**.
  - Interrupción de salida: *InterruptOut*.
  - Interrupción de entrada: *InterruptIn*.
  - Estado de programa: *ProgramState*.
  - Programa: *Program*.
- **Compilador**: *assembler*.
  - Cargador de programa: *ProgramLoader*.
  - Etiqueta: *Label*.
  - Sintaxis ensamblador: *AssemblerSyntax*.
- Arquitectura: *Architecture*.

Las clases de un paquete de arquitecturas específico que realizan las interfaces definidas en el paquete de interfaces de las arquitecturas tienen por nombre el nombre de la arquitectura seguido por el nombre de la interfaz que realizan. Por ejemplo, en el paquete de la arquitectura LEGv8, la clase que realiza la interfaz Register es LEGv8Register.

Un aspecto a tener en cuenta a la hora de implementar el sistema es que, al tratar de construir un *frontend* genérico que funcione con cualquier tipo de arquitectura, los tipos de datos también

deben generalizarse. Es decir, no se pueden hacer suposiciones sobre el tamaño de los datos con los que trabaja una arquitectura. Por ejemplo, MIPS trabaja con registros de 32 bits, mientras que los registros de LEGv8 son de 64 bits. El simulador debe soportar ambas arquitecturas. En diseño, no se especifican los tamaños de los datos numéricos, indicándose de forma genérica como de tipo *integer*. Como solución, siempre que se desconozca el tamaño que va a tener un dato numérico, se utiliza como tipo del dato el tipo más grande nativo en Java, `long`, de 64 bits.

Como consecuencia, la implementación actual del simulador no soporta arquitecturas que trabajen con tipos de datos más grandes que 64 bits. Aunque a día de hoy existen arquitecturas de este tipo (por ejemplo, la versión de 128 bits de RISC-V [27]), no son un conjunto tan significativo ni tan ampliamente utilizadas como para que su exclusión tenga un impacto relevante en el cumplimiento de los objetivos de extensibilidad del simulador.

Otra consideración a tener en cuenta es que la interfaz de diseño InstructionResult se implementa como cuatro interfaces distintas:

- InstructionResultR, que describe las operaciones que debe realizar un resultado de instrucción que escribe registros.
- InstructionResultM, que describe las operaciones que debe realizar un resultado de instrucción que escribe en memoria.
- InstructionResultS, que describe las operaciones que debe realizar un resultado de instrucción que modifique las banderas de estado de la CPU.
- InstructionResultJ, que describe las operaciones que debe realizar un resultado de instrucción que puede producir una bifurcación.

La división de la interfaz única en diseño en cuatro interfaces en implementación se debe a un intento de reducir la complejidad del código desarrollado: es raro que una instrucción produzca un resultado que deba realizar más de una de las cuatro interfaces a la vez. No obstante, se implementa una propuesta de clase que realiza las cuatro interfaces al mismo tiempo: InstructionResult. Esta clase puede ser, o no, utilizada por los módulos de las arquitecturas concretas.

Del mismo modo, se implementa la clase DataImpl. Esta clase es una propuesta de clase concreta que realiza la interfaz Data, y puede ser, o no, utilizada por los módulos de las arquitecturas concretas. Esta clase se incluye en el módulo de interfaces de las arquitecturas debido a su considerable sencillez y a que implementa funcionalidades comunes a un gran número de arquitecturas.

También se implementa la clase Label como clase concreta en vez de como interfaz. Esto se debe a que se consideró que su implementación no iba a variar de forma notable entre distintas arquitecturas.

6.2.2. Implementación de la capacidad de extensibilidad del simulador

El requisito RNF3, detallado en la sección 5.1.2 de este documento, establece que el sistema debe ser fácilmente extensible para soportar otras arquitecturas de computadores y lenguajes en-

samblador. Para implementar esta funcionalidad de forma que resulte fácil de utilidad, se utiliza el sistema de carga de servicios de Java. Este sistema es accesible mediante la clase ServiceLoader [63]. La documentación de la clase explica su funcionalidad utilizando los siguientes términos:

- *Servicio*: una interfaz o clase conocida para la cual existen cero, uno, o varios proveedores existen.
- *Proveedor de servicio* o *proveedor*: es una clase que implementa o extiende la interfaz o clase conocida.
- *Cargador de servicio*: objeto que localiza y carga proveedores de servicio desplegados en el entorno de ejecución, a petición de la aplicación. El cargador de servicio es capaz de distinguir entre múltiples proveedores.

En el prototipo desarrollado, la interfaz Architecture es un servicio. El módulo de la arquitectura LEGv8, por ejemplo, presenta un proveedor de ese servicio: LEGv8Architecture. Esto significa que, de hallarse el módulo LEGv8 compilado en una ruta localizable por el simulador, este sería capaz de cargar e instancializar una LEGv8Architecture y utilizarla como si se tratase de una instancia de Architecture. Este proceso no requiere la modificación ni la recompilación del simulador cuando se desee añadir un nuevo módulo de arquitectura; la extensión del simulador con más arquitecturas es de tipo *plug-and-play*.

Los módulos que contienen los proveedores deben ser compilados como ficheros jar de Java. Para que el simulador pueda localizar un proveedor de Architecture, el fichero jar producto de la compilación del proveedor debe especificar que provee ese servicio. Para ello, se debe incluir en el directorio de metainformación del jar (*META-INF*) un directorio llamados *services* que incluya un fichero por cada servicio provisto. El fichero debe llamarse como la ruta Java de la interfaz o clase que define el servicio que se provee, y debe contener la ruta Java de la clase que provee el servicio. Por ejemplo, para el módulo de la arquitectura LEGv8, el directorio *META-INF/services/* incluye el fichero:

```
es.uva.infor.andromeda.cpu.interfaces.Architecture
```

El contenido de este fichero es exclusivamente la línea:

```
es.uva.infor.andromeda.cpu.legv8.LEGv8Architecture
```

El fragmento de código 6.1 presenta el código fuente de los métodos de la clase ArchLoader, equivalente al Cargador de arquitecturas en diseño. Esta clase es la que utiliza el sistema de cargador de servicios de Java para proporcionar las instancias de Architecture al sistema.

6.2.3. Implementación del banco de registros LEGv8

El banco de registros de la arquitectura LEGv8 se implementa mediante las clases LEGv8RegisterFile y LEGv8Regiser. La primera, es una clase de tipo *singleton* que almacena todas las instancias existentes en el sistema de LEGv8Register.

```
public static List<String> getAvailableArchs() {
  List<String> archs = new ArrayList<>();

  for (Architecture arch : ServiceLoader.load(Architecture.class)) {
    archs.add(arch.name());
  }

  return archs;
}

public static @Nullable Architecture loadArch(String archName) {
  for (Architecture arch : ServiceLoader.load(Architecture.class)) {
    if (arch.name().equals(archName)) {
      return arch;
    }
  }

  return null;
}
```

Fragmento de código 6.1: Métodos de carga de arquitecturas de la clase archLoader. Estos métodos son capaces de decir qué arquitecturas hay disponibles en el entorno de ejecución del simulador, así como cargar una de ellas para su uso.

El conjunto de registros de la arquitectura se representa en la clase LEGv8RegisterFile utilizando una estructura de tipo mapa. Concretamente, se utiliza una instancia de la clase HashMap provista por Java. Las claves del mapa son los nombres de los registros, y los valores son las instancias de LEGv8Register cuyo nombre coincide con la clave. Cualquier fragmento de código que requiera acceder a una instancia específica de LEGv8Register debe hacerlo a través de este mapa.

En la inicialización de la instancia única de LEGv8RegisterFile se crean las instancias de todos los registros utilizados en la arquitectura LEGv8, añadiéndose al mapa. Los registros se añaden al mapa de forma que puedan ser accesibles a través de todos sus nombres. Por tanto, múltiples claves pueden hacer referencia a la misma instancia de LEGv8Register. La cantidad y los nombres de los registros que se crean se han tomado del libro de referencia de LEGv8 [28].

6.2.4. Implementación de la memoria LEGv8

La memoria de la arquitectura LEGv8 se implementa mediante las clases LEGv8Memory, LEGv8Section y DataImpl. La primera es una clase de tipo *singleton* que almacena todas las instancias existentes en el sistema de LEGv8Section. La segunda es una clase que almacena instancias de DataImpl, simulando el almacenamiento de datos en memoria.

Para la clase LEGv8Section, en un principio se planteó que las secciones de memoria podían implementarse como listas de Datos, utilizando la clase List de Java. Sin embargo, según la referencia de LEGv8, el espacio de memoria accesible en una arquitectura LEGv8 incluye todo el rango de direcciones desde 0x0 en hexadecimal (0 en decimal) hasta 0x7ffffffc en hexadecimal (549 755 813 884 en decimal). Cualquier implementación de este espacio de memoria basada en listas requeriría el uso de un mínimo de 550 Gigabytes de memoria RAM. Este requisito es insatis-

facible por la gran mayoría de ordenadores personales existentes. Para solucionar este problema, se plantearon dos alternativas:

- Limitar el tamaño máximo de las secciones de memoria LEGv8, impidiendo la representación de todo el espacio de direcciones especificado en la referencia.
- Buscar una representación alternativa a las listas que no requiriese el uso de tanta memoria.

La primera alternativa es factible, sencilla de implementar, y tampoco impone unas limitaciones tan importantes, ya que no se espera que la mayoría de programas ensamblador desarrollados para el simulador no utilizan más que unos Kilobytes de memoria de datos. Sin embargo, la segunda alternativa es la más versátil, permitiendo la simulación de cualquier dirección de memoria accesible por la arquitectura. Finalmente, fue la alternativa escogida.

Las secciones de memoria almacenan los datos de memoria utilizando mapa, mediante la clase HashMap de Java. Los datos almacenados en el mapa son instancias de la clase DataImpl. Las direcciones de memoria de los datos son las claves del mapa, mientras que las instancias de DataImpl son los valores. Cualquier fragmento de código que requiera acceder a un dato concreto de memoria, es decir, a una instancia de DataImpl, debe hacerlo a través del mapa correspondiente a la sección de memoria que abarca ese dato.

En el mapa se almacenan exclusivamente aquellos datos cuyos valores son distintos de cero. Como todas las direcciones de memoria se inicializan a 0, en el mapa solo se almacenan los datos modificados por el usuario mediante los programas. La lógica del método que implementa la funcionalidad de acceso a datos de memoria, es la que sigue:

1. Se comprueba que la dirección de memoria a la que se desea acceder es válida.
2. Se comprueba si la dirección de memoria se incluye entre las claves del mapa de datos.
3. Si se incluye, se devuelve el valor asociado al dato.
4. Si no se incluye, se devuelve 0.

La lógica del método que implementa la funcionalidad de escritura de datos en memoria, es la que sigue:

1. Se comprueba que la dirección de memoria en la que se desea escribir es válida.
2. Se comprueba si el valor que se desea escribir es 0.
3. Si lo es, se elimina del mapa de datos el par clave-valor asociado a la dirección de memoria.
4. Si no lo es, se comprueba si la dirección que se desea escribir se incluye en las claves del mapa de datos.
5. Si se encuentra, se modifica el valor del dato.

6. Si no se encuentra, se crea un nuevo dato con la dirección y el valor correspondiente, y se añade al mapa.

El fragmento de código 6.2 muestra la implementación de estas funciones.

```
@Override
public byte read(long address) {
  checkAddress(address);

  if (data.containsKey(address)) {
    return data.get(address).value();
  } else {
    return 0;
  }
}

@Override
public void write(long address, byte value) {
  checkAddress(address);

  if (value != 0) {
    if (data.containsKey(address)) {
      data.get(address).setValue(value);
    } else {
      data.put(address, new DataImpl(address, value));
    }
  } else {
    data.remove(address);
  }
}
```

Fragmento de código 6.2: Implementación de los métodos de escritura y lectura de datos en secciones de memoria LEGv8. Los datos se almacenan en un HashMap. Exclusivamente los datos cuyo valor es distinto de 0 se almacenan, ahorrando grandes cantidades de memoria.

La clase LEGv8Memory almacena todas las secciones de la arquitectura LEGv8 utilizando una clase de tipo mapa. Las claves del mapa son el nombre de las secciones, siendo los valores almacenados las propias instancias de las secciones.

Una característica particular de la arquitectura LEGv8 es que permite la lectura y escritura en forma exclusiva de direcciones de memoria. Esta característica se implementa siguiendo la referencia *Arm Architecture Reference Manual - Armv8, for Armv8-A architecture profile* [64]. Este manual especifica que ciertas direcciones de memoria pueden ser marcadas como "exclusivas", alterando el funcionamiento de la ejecución de ciertas instrucciones. Estas marcas se implementan en la clase como un conjunto, mediante la clase HashSet. Este conjunto almacena todas las direcciones de memoria que se marquen como exclusivas. Las instrucciones pueden comprobar si una dirección es exclusiva. La clase LEGv8Memory proporciona métodos para realizar estas comprobaciones, comprobando los contenidos del HashSet.

6.2.5. Implementación del conjunto de instrucciones LEGv8

El conjunto de instrucciones LEGv8 se implementa principalmente mediante las clases LEGv8InstructionSet, LEGv8Instruction y LEGv8ProgramInstruction. La primera es una clase de tipo *singleton* que almacena todas las instancias existentes en el sistema de LEGv8Instruction.

La clase LEGv8Instruction gestiona la información relativa a las instrucciones LEGv8. En esta clase, los códigos de operación de las instrucciones se generan utilizando una estructura de tipo *switch-case*. Esta estructura comprueba el mnemónico de la instrucción, generando correspondientemente sus campos *opcode* y *shamt*. Esta estructura *switch-case* también se utiliza para definir el tipo o formato de la instrucción, que se utiliza para generar el código máquina de las instrucciones de programa en LEGv8ProgramInstruction. La clase LEGv8Instruction también contiene un atributo de tipo ExecutionCode. Este atributo se define mediante una clase anónima y almacena la lógica asociada a la ejecución de la instrucción.

El conjunto de instrucciones LEGv8 se representa en la clase LEGv8InstructionSet utilizando una estructura de tipo mapa, mediante un HashMap. las claves del mapa son los mnemónicos de las instrucciones, y los valores son las instancias de LEGv8Instruction que corresponden a los mnemónicos. En total se incluyen 72 instrucciones en el HashMap. Cualquier fragmento de código que requiera acceder a una instancia específica de LEGv8Instruction debe hacerlo a través de este mapa.

En la inicialización de la instancia única de LEGv8InstructionSet se crean las instancias de todas las instrucciones de la arquitectura LEGv8, añadiéndose al mapa. Es en esta inicialización también en la que se crean las clases anónimas que realizan la interfaz ExecutionCode y definen la funcionalidad de la instrucción. Desde Java 8, Java ofrece una sintaxis concisa para definir las clases anónimas mediante funciones lambda. Esta sintaxis facilita la lectura y comprensión del código. Como ejemplo, el fragmento de código 6.3 muestra la creación de la instrucción LEGv8 ADD y su adición al mapa de instrucciones.

Para implementar la lógica de ejecución de las instrucciones LEGv8, se ha utilizado como referencia *Arm Architecture Reference Manual - Armv8 for Armv8-A architecture profile* [64]. Ese documento especifica la funcionalidad que ejecuta cada instrucción ARMv8 (conteniendo, por tanto, todas las instrucciones LEGv8). La operación que realiza cada una de las instrucciones se especifica utilizando pseudocódigo, por lo que en muchas ocasiones la implementación de la lógica de ejecución de las instrucciones en el sistema es directa. En otras ocasiones, las limitaciones de Java obligaban a buscar implementaciones alternativas. El fragmento de código 6.4 ilustra la implementación en pseudocódigo de la instrucción ARMv8 ADD, equivalentes a la instrucción LEGv8 ADD cuya implementación se muestra en el fragmento de código 6.3.

La clase LEGv8ProgramInstruction contiene los datos para ejecutar la funcionalidad de las instrucciones LEGv8: registros de fuente y destino, y valores numéricos constantes. Para ejecutar la funcionalidad asociada a una LEGv8Instruction, se debe proveer una instancia de LEGv8ProgramInstruction que contenga los datos sobre los que trabajar. El proceso por el cual se ejecuta una instrucción LEGv8 en el simulador es el siguiente:

1. Se invoca el método `public void execute()` de LEGv8ProgramInstruction.

```
set.put("ADD", new LEGv8Instruction(
    "ADD",
    "Add register: adds two register values, and writes " +
        "the result to the destination register.",
    "ADD X19, X20, X19",
    instance -> {
      List<Register> regList = instance.getRegisters();
      Register Rd = regList.get(0);
      Register Rn = regList.get(1);
      Register Rm = regList.get(2);

      long operand1 = legv8.currentProgram.valueIn(Rn);
      long operand2 = legv8.currentProgram.valueIn(Rm);
      long result = operand1 + operand2;

      ((LEGv8ProgramInstruction) instance).setResult(result, Rd);
    }));
```

Fragmento de código 6.3: Creación y adición al mapa de instrucciones de la instrucción LEGv8 ADD, en la clase LEGv8InstructionSet. La lógica de ejecución de la instrucción se implementa haciendo uso de funciones lambda de Java. Esta expresión lambda se le pasa al constructor de LEGv8Instruction como cuarto parámetro.

2. LEGv8ProgramInstruction invoca el método `public void execute(ProgramInstruction)` de LEGv8Instruction, pasando como parámetro la instancia de LEGv8ProgramInstruction que realiza la invocación.

3. LEGv8Instruction invoca el método `public void execute(ProgramInstruction)` de la interfaz ExecutionCode, pasando como parámetro la instancia de LEGv8ProgramInstruction que ha recibido.

4. La clase anónima asociada a la instancia de LEGv8Instruction y que realiza la interfaz ExecutionCode ejecuta su método `public void execute(LEGv8ProgramInstruction)`. Este método ejecuta la funcionalidad asociada a la instancia específica de LEGv8Instruction, utilizando para ello los datos contenidos en la instancia de LEGv8ProgramInstruction que se le pasa por parámetro. La funcionalidad concreta de este método ha sido definida mediante la clase anónima creada junto con la instancia de LEGv8Instruction que ha realizado la invocación del método. Ambas, tanto la definición de de la funcionalidad del método como la creación de la instancia de LEGv8Instruction que lo invoca, se realizan en la clase *singleton* LEGv8InstructionSet.

Las instancias de LEGv8ProgramInstruction que conforman un programa son generadas por el compilador de la arquitectura, y almacenadas en un LEGv8Program.

La clase LEGv8ProgramInstruction también define la lógica que calcula el código máquina correspondiente a una instrucción de programa LEGv8. El cálculo del código máquina depende del formato de la LEGv8Instruction asociada a la LEGv8ProgramInstruction. Para realizar la implementación de esa lógica, se utiliza una estructura de tipo *switch-case*. Los casos contemplados por la estructura son los distintos formatos de instrucción válidos para las instrucciones LEGv8. Dentro de cada caso, se calcula la codificación de todos los campos de la instrucción, se obtiene el *opcode* (y opcionalmente el *shamt*) de la LEGv8Instruction asociada, y se genera el código

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_tyoe, shift_amount);

(result, -) = AddWithCarry(operand1, operand2, '0')
X[d] = result;

// AddWithCarry()
// ==============
// Integer addition with carry input, returning result and NZCV flags
(bits(N), bits(4)) AddWithCarry(bits(N) x, bits(N) y, bit carry_in)
  integer unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
  integer signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
  bits(N) result = unsigned_sum<N-1:0>; // same value as signed_sum<N-1:0>
  bit n = result<N-1>;
  bit z = if IsZero(result) then '1' else '0';
  bit c = if UInt(result) == unsigned_sum then '0' else '1';
  bit v = if SInt(result) == signed_sum then '0' else '1';
  return (result, n:z:c:v)
```

Fragmento de código 6.4: Pseudocódigo para la instrucción A64 (ARMv8) ADD. Extraído de [64], página C6.2.5. El valor *datasize* es el tamaño de los valores almacenados en los registros (64 en LEGv8). Se incluye también el pseudocódigo de la operación `AddWithCarry()`, para que el fragmento de código sea autocontenido. Esta operación también se utiliza en otras instrucciones, como ADDS, ADDI o ADDIS, por lo que también calcula el resultado a almacenar en las banderas de condición. La instrucción ADD ignora este segundo resultado.

máquina tal y como se indica en la referencia de LEGv8 (ver figura 4.4 en el capítulo 4 de este documento).

Por último, cada instancia de LEGv8ProgramInstruction almacena en un atributo de tipo String la línea textual de código que la generó. Esta información se almacena para poder reconstruir el código fuente preprocesado de un programa.

6.2.6. Implementación de la estructura de programas LEGv8

Los programas LEGv8 se implementan mediante las clases LEGv8Program y LEGv8ProgramState. La primera clase almacena como atributos información de interés sobre un programa LEGv8:

- La lista de instancias de LEGv8ProgramInstruction que componen el programa. A partir de esta lista se puede obtener el código máquina equivalente al programa, así como el código fuente textual del programa, preprocesado.
- La lista de líneas de código textuales (no preprocesadas) que han generado el programa.
- Los datos de memoria con los que se debe inicializar la memoria del simulador antes de ejecutar el programa. Estos datos incluyen todos los datos especificados con directivas de

preprocesador en el código fuente, así como los datos equivalentes a las instrucciones del programa.

- Una instancia de LEGv8ProgramState, que almacena el estado del programa.
- Una instancia de LEGv8Pipeline, que representa el *pipeline* en que se ejecuta el programa.

La clase LEGv8ProgramState almacena información sobre el estado de la ejecución programa. Principalmente, almacena los valores de las banderas de condición, como un atributo de tipo int. Solo los 4 bits menos significativos de la variable, que representan las 4 banderas de condición, son modificados.

La clase LEGv8Program incluye dos métodos de para ejecutar el programa:

- `public void execute()`, que invoca en bucle el método `cycle()` de la instancia de LEGv8Pipeline almacenada como atributo. El bucle se detiene cuando la instancia de LEGv8Pipeline indica que ha finalizado la ejecución, mediante la invocación del método `ended()`.
- `public void executeNext()`, que invoca una sola vez el método `cycle()` de la instancia de LEGv8Pipeline almacenada como atributo.

Como se puede observar, ambos métodos simplemente delegan sus funcionalidades en la clase LEGv8Pipeline.

La inicialización de la CPU para ejecutar un programa también se realiza desde la clase LEGv8Program. Esta clase incluye un método, `public void initCPU()`, que invoca los métodos `init()` de las clases que hay que inicializar: LEGv8RegisterFile, LEGv8Memory, LEGv8ProgramState y LEGv8Pipeline. También inicializa estáticamente la clase InstructionResult, definiendo para todas sus instancias:

- La instancia de Memory en la que se deben escribir los resultados de escrituras a memoria.
- La instancia de ProgramState en la que se deben consultar y modificar los valores de las banderas de condición.
- La instancia de Registro que representa el contador de programa de la arquitectura.

6.2.7. Implementación del *pipeline* segmentado LEGv8

El *pipeline* LEGv8 se implementa mediante la clase LEGv8Pipeline.

La clase LEGv8Pipeline presenta los siguientes atributos, entre otros:

- Un mapa de tipo EnumMap que representa las instrucciones dentro del *pipeline*. Las claves del mapa son las distintas etapas del *pipeline*, y los valores son arrays que incluyen las instrucciones que las ocupan (o *null*, si no las ocupa ninguna instrucción). Se utiliza un *array* en vez de instancias directamente debido a que, en ciertas etapas, puede haber más de una instrucción en ejecución a la vez. Estos arrays, en caso de tener un tamaño mayor que uno, representarían los "*subpipelines*" de las etapas del *pipeline*.
- Las instancias de OperationUnit que representan las unidades funcionales que calculan, respectivamente:
  - las operaciones simples con números enteros,
  - las operaciones simples con números de coma flotante,
  - las multiplicaciones de cualquier tipo de número y
  - las divisiones de cualquier tipo de número.
- La instancia de JumpUnit utilizada para predecir y resolver las bifurcaciones.
- Atributos de tipo booleano que señalan el estado del *pipeline*. Entre ellos, se encuentra `ended`, que marca si la ejecución del programa en el *pipeline* ha finalizado.

La clase LEGv8Pipeline realiza la interfaz Pipeline. Esta interfaz declara tres métodos: `void cycle()`, `boolean ended()` y `void init()`. En LEGv8Pipeline, el método `init()` simplemente

- inicializa el atributo `ended` a `false`,
- inicializa todos los valores de los arrays del mapa de etapas a *null*,
- e introduce la primera instrucción del programa en la primera etapa del mapa.

El método `ended()` simplemente es un *getter* del atributo del mismo nombre.

El método `cycle()`, sin embargo, realiza muchas subtareas de gran complejidad. Estas son las asociadas a cada etapa. Para organizar mejor el código de la clase, cada etapa se divide en un método privado independiente. Entre las tareas que se realizan en cada una de estas funciones se encuentran:

- Etapa WB:
  - Escribir los resultados de las instrucciones en los registros.
  - Actualizar el atributo `ended` a `true` si se ha ejecutado la última instrucción del programa.
- Etapa MEM:
  - Ejecutar las instrucciones de carga desde memoria.
  - Escribir en memoria los resultados de las instrucciones de almacenamiento en memoria.

- Actualizar el atributo `ended` a `true` si se ha ejecutado la última instrucción del programa, y esta no utiliza la etapa WB del *pipeline*.
- Avanzar la instrucción a la etapa WB, si la instrucción la utiliza.

- Etapa EX:
  - Se ejecutan las subetapas de ejecución asociadas a las 4 instancias de OperationUnit almacenadas como atributo. En estas subetapas se realizan las siguientes tareas:
    - Avanzar las instrucciones en el *subpipeline* de la etapa correspondiente, tal y como marquen la latencia y el intervalo de iniciación de la unidad de operaciones de la etapa.
    - Comprobar si alguna instrucción se encuentra en la etapa final del *subpipeline*, ejecutándola.
    - Avanzar las instrucciones salientes a la etapa MEM o WB, según corresponda.
  - Se resuelven las bifurcaciones, si así lo dice la política de saltos de la instancia de JumpUnit almacenada como atributo.

- Etapa ID:
  - Comprobar todos los posibles riesgos que se generan en el *pipeline* por la introducción de la instrucción que ocupa la etapa, incluyendo:
    - Riesgos de datos de tipo *RAW*, entre instrucciones de tipo R y cargas desde memoria.
    - Riesgos estructurales causados por la ocupación de las unidades de operaciones multiciclo.
    - Riesgos estructurales causados por la utilización de la etapa WB de dos instrucciones que utilizan diferentes unidades de operaciones.
    - Riesgos de datos de tipo *RAW*, entre instrucciones que utilizan distintas unidades de operaciones.
  - Detener el avance de las instrucciones en el *pipeline* durante un ciclo si se detecta alguno de los riesgos especificados en el punto anterior.
  - Resolver las bifurcaciones, si así lo dice la política de saltos de la instancia de JumpUnit almacenada como atributo.
  - Avanzar la instrucción a la subetapa EX que le corresponda, si no se ha producido una detención ese ciclo.

- Etapa IF:
  - Realizar la predicción de saltos siguiendo la política de predicción de la instancia de JumpUnit almacenada como atributo.
  - Resolver las bifurcaciones, si así lo dice la política de saltos de la instancia de JumpUnit almacenada como atributo.
  - Modifica el contador de programa, teniendo en cuenta si se ha bifurcado ese ciclo o no.
  - Avanzar la instrucción a la etapa ID, si no se ha producido una detención ese ciclo.
  - Añadir en la etapa IF la próxima instrucción a ejecutar, si no se ha producido una detención ese ciclo.

La lógica de detección de riesgos y de gestión de saltos, tanto predicción como resolución, presentan una gran complejidad. Hay que conocer detalladamente el dominio del sistema simulado, especialmente de los *pipelines* segmentados, para comprender la implementación.

Hay que tener en cuenta que las etapas del *pipeline* se ejecutan de última (WB) a primera (IF). Esto es así ya que una etapa debe liberarse antes de que la anterior haga avanzar su instrucción; si no, se sobrescribiría la instrucción de la etapa. Esto incrementa aun más la lógica de la clase, pues hay que tener en cuenta que, dependiendo de en qué etapa se esté trabajando, algunas instrucciones ya habrán avanzado a la etapa siguiente, mientras que otras todavía no. En otras palabras, conceptualmente, todas las instrucciones de cada etapa se ejecutan al mismo tiempo; pero en la implementación no es así. Esta discrepancia dificulta el desarrollo y la comprensión del código del *pipeline*, haciendo que sea común confundir la etapa en la que realmente se encuentra una instrucción.

Como en los *pipelines* segmentados reales, la clase LEGv8Pipeline simula la anticipación de resultados. Esto se hace invocando el método `public long valueIn(Register)` de LEGv8Pipeline cada vez que se desee obtener el valor de un registro, en vez de obteniendo el valor directamente del registro invocando su método `public long getValue()`. El método `valueIn(Register)` realiza ciertas comprobaciones para detectar si el registro que se desea leer es modificado por una instrucción que todavía no ha ejecutado su etapa WB. De ser así, se accede a la instancia de InstructionResult asociada a esa instrucción y se devuelve el valor que se va a escribir en el registro. De no ser así, se devuelve el valor ya almacenado en el registro. El fragmento de código 6.5 muestra la implementación del método `valueIn(Register)`. Todas las clases anónimas que definen la funcionalidad de las instrucciones LEGv8 utilizan el método `valueIn(Register)` cuando quieren acceder a un registro, como se puede observar en el fragmento de código 6.3.

```
public long valueIn(Register reg) {
  // Instrucciones que este ciclo empezaron en la etapa MEM:
  @Nullable LEGv8ProgramInstruction inWB = line.get(WB)[0];
  if (inWB != null && inWB.writesReg(reg)
      // Las instrucciones que no utilizan la ALU de enteros
      // no utilizan la fase MEM, asi que ya han escrito sus
      // resultados.
      && getOperationUnit(inWB).equals(ALU)) {
    return inWB.result().regValue(reg);
  }

  // Resto de instrucciones (ya han ejecutado WB):
  return LEGv8Architecture.registerFile.getRegister(reg.name())
      .getValue();
}
```

Fragmento de código 6.5: Función de anticipación de resultados de LEGv8Pipeline. Hay que tener en cuenta que este método es invocado exclusivamente en la etapa EX del *pipeline*, por lo que las instrucciones que ese ciclo comenzaron en las etapas WB o MEM ya han finalizado su ejecución o avanzado a WB, respectivamente.

Como se ha dicho anteriormente, la predicción de saltos se realiza en la etapa IF. El fragmento de código 6.6 muestra la lógica asociada a estas predicciones. Para resolver los saltos y corregir las predicciones incorrectas, se utiliza un método privado:
`private boolean processJump(PipelineStage, LEGv8ProgramInstruction)`. Este método pue-

```
if (ins != null && ins.isBranch()) {
  if (jmp.resolutionStage() == IF) {
    ... // Logica de resolucion de saltos en la etapa IF.
  } else if (jmp.predictionPolicy() == STOP || ins.is("BR")) {
    // Las instrucciones BR no pueden predecirse sin ser ejecutadas.
    stopped = true;
  } else if (ins.type() == LEGv8Instruction.InstructionType.B
      || jmp.predictionPolicy() == TAKEN) {
    long branchAddress =
        ins.getAddress() + 4 * ins.getNumber();
    PC.setValue(branchAddress);
    jumped = true;
  }
  // Las predicciones DELAYED y NOT_TAKEN no toman el salto.
  // La prediccion IDEAL resuelve los saltos en la etapa IF,
  //  por lo que no necesita predecirlos.
}
```

Fragmento de código 6.6: Lógica de predicción de saltos en el método de la etapa IF de LEGv8Pipeline. La instrucción de bifurcación BR no puede predecirse, por lo que siempre detiene el flujo de instrucciones en el *pipeline*. La política de predicción IDEAL equivale a resolver los saltos en la etapa IF.

de invocarse en las etapas EX, IF o ID. La invocación en la etapa EX no debería generar ningún problema ya que los resultados de instrucciones anteriores pueden anticiparse a esa etapa. Este no es el caso para las etapas IF o ID. Para solucionar los problemas asociados a las predicciones en esas etapas, se ha implementado el método de tal forma que la resolución de los saltos es siempre ideal, independientemente de la etapa en la que se realice tal predicción. Para realizar esta resolución ideal de saltos, todas las instrucciones del *pipeline* anteriores al salto se ejecutan de forma forzosa antes de realizar el salto. Esto asegura que, si el salto depende de una instrucción anterior en el *pipeline*, cuando se ejecute, lo hará teniendo acceso a los valores correctos de los registros y las banderas de condición.

La ejecución forzosa de instrucciones para resolver correctamente saltos acarrea el evidente problema de que las instrucciones finalizan su ejecución antes de tiempo, volviendo incorrecta la simulación. Para solucionar este problema, se ha implementado un sistema para revertir los efectos de las instrucciones, de forma transparente al resto del simulador. Este sistema se basa en el guardado y la recuperación de "estados":

- Las clases susceptibles a modificar su estado debido a la ejecución de una instrucción implementan los métodos `public void savestate()` y `public void loadstate()`, además de un atributo de copia de estado. Estas clases son: LEGv8RegisterFile, LEGv8Memory y LEGv8ProgramState.
- El método `savestate()` guarda el estado actual de la clase en el atributo de copia de estado.
- El método `loadstate()` recupera el estado de la clase guardado en el atributo de estado.

A la hora de resolver un salto, se procede como sigue:

1. Se guardan los estados de la memoria LEGv8, el banco de registros LEGv8, y el estado del programa. Para ello, se utilizan los respectivos métodos `savestate()`.
2. Se ejecutan todas las instrucciones del *pipeline* anteriores al salto.
3. Se ejecuta el salto, generando su resultado.
4. Se recupera el estado de la memoria LEGv8, el banco de registros LEGv8, y el estado del programa. Para ello, se utilizan los respectivos métodos *loadstate()*.
5. Si el salto es tomado, se modifica el registro LEGv8 del contador de programa como corresponda.
6. Si la predicción realizada en la etapa IF es incorrecta, se eliminan del *pipeline* todas las instrucciones posteriores al salto. El fragmento de código 6.7 muestra el código encargado de la gestión de las predicciones incorrectas de saltos.

De esta forma, se resuelven los saltos de forma correcta sin causar efectos secundarios en el resto del sistema.

```
if (result.jumpTaken()) {
  // Corregir las predicciones que no toman el salto:
  switch (jmp.predictionPolicy()) {
    case NOT_TAKEN:
      flushPipeline(stg);
    case STOP:
      jumped = true;
      result.resolveJump();
    default:
      break;
  }
} else if (jmp.predictionPolicy() == TAKEN) {
  // Corregir las predicciones que toman el salto:
  flushPipeline(stg);
  PC.setValue(ins.getAddress());  // Arreglar el valor del PC.
}
```

Fragmento de código 6.7: Código de gestión de las predicciones incorrectas de saltos en la clase LEGv8Pipeline.

Para eliminar las instrucciones del *pipeline* incorrectamente añadidas debido a la predicción incorrecta de un salto, se implementa el método privado `flushPipeline(PipelineStage)`. Este método convierte en *nulls* todas las instrucciones en etapas anteriores a la provista por parámetro. Este método se muestra en el fragmento de código 6.8.

Para detectar la finalización de la ejecución de un programa en el *pipeline*, se utiliza el atributo `lastInstruction`. Este atributo de tipo LEGv8ProgramInstruciton almacena la instancia de la que se cree que será la última instrucción en ejecutarse. El valor de este atributo es modificado cada vez que, en la etapa IF, se detecta que el contador de programa apunta fuera de la lista de instrucciones de programa por ejecutar. Cuando se da este caso, el valor del atributo pasa a ser el de la última instrucción no nula en el *pipeline*. Cuando esa instrucción de programa llega a su etapa final, ya sea MEM o WB, el valor del atributo `ended` pasa a ser `true`, ya que se ha

```
private void flushPipeline(@NotNull PipelineStage stg) {
  for (PipelineStage s : line.keySet()) { // Se itera sobre las etapas del
   Pipeline
    if (s == stg) {
      return;
    }

    LEGv8ProgramInstruction[] pipeStage = line.get(s);
    for (int i = 0; i < pipeStage.length; i++) {
      pipeStage[i] = null;
    }
  }
}
```

Fragmento de código 6.8: Método de descarte de instrucciones incorrectamente añadidas al *pipeline* debido a una predicción de saltos incorrecta. Este método se incluye en la clase LEGv8Pipeline.

finalizado la ejecución de la última instrucción del programa. Si antes de que esa instrucción llegue a su etapa final alguna instrucción produjese que se tomara un salto en el *pipeline*, saltando a una instrucción anterior, el valor del atributo `lastInstruction` volvería a ser *null*.

Para finalizar, cabe destacar que el constructor de LEGv8Pipeline toma como parámetros de entrada:

- el programa a ejecutar,
- las cuatro unidades operacionales asociadas a las cuatro subetapas EX y
- la unidad de salto a utilizar.

Esto permite la creación de distintas *pipelines* con unidades de operaciones y unidades de salto que presenten distintas configuraciones. En el caso de las unidades de operaciones, se puede variar su latencia y su intervalo de iniciación. El constructor generará automáticamente los *subpipelines* asociados a cada unidad de operaciones con el número de etapas adecuado. En el caso de la unidad de salto, se puede variar la política de predicción de saltos utilizada, además de la etapa de resolución de los saltos.

Como alternativa a los constructores, la clase presenta un método estático, `idealPipeline(LEGv8Program)`, que devuelve una instancia de LEGv8Pipeline que presenta una configuración ideal: la política de resolución de saltos es IDEAL, y todas las unidades de operaciones tienen una latencia de 0 e intervalo de iniciación de 1.

6.2.8. Implementación del compilador de ficheros ensamblador LEGv8

El compilador de LEGv8 se implementa utilizando ANTLR 4. ANTLR 4 permite la definición de analizadores léxicos y analizadores sintácticos en un mismo fichero único, lo que significa que todo un compilador puede escribirse en un único fichero ANTLR.

A primera vista, la especificación de un programa ensamblador no parece presentar gran dificultad. La estructura de las sentencias ensamblador es muy parecida al código de tres direcciones descrito en *Compilers: Principles, Techniques and Tools* [65]. El código de tres direcciones se suele utilizar dentro de los compiladores como representación intermedia de los programas de alto nivel, ya que presenta una gran sencillez estructural que facilita su traducción al objeto final de la compilación. Cada instrucción ensamblador, así como cada instrucción de código de tres direcciones, presenta:

- Un operador, generalmente binario.
- Generalmente una asignación.
- Como mucho tres operandos, generalmente dos sobre los que se opera y uno al que se asigna el resultado.

Por ejemplo, la instrucción de código de tres direcciones

```
x2 := x0 + x1
```

equivaldría a la instrucción LEGv8

```
ADD X2, X0, X1.
```

Para reconocer una sentencia ensamblador, sería tan fácil como:

1. Identificar la operación (instrucción) de la sentencia mediante su mnemónico. Para ello, se lee la primera palabra del texto. En el caso del ejemplo anterior, esta sería `ADD`.
2. Identificar el número de operandos sobre los que trabaja la instrucción identificada. En el caso del ejemplo anterior, este sería 3.
3. Leer tantas palabras, separadas por comas, como operandos utilice la instrucción identificada (ya sean registros o constantes numéricas, según corresponda). En el caso del ejemplo anterior, se leería `X2`, `X0` y `X1`.
4. Con las palabras (llamadas *tokens* en el contexto de los analizadores léxicos) identificadas, se construye la correspondiente LEGv8ProgramInstruction.

Para generar un programa ensamblador completo (LEGv8Program) a partir de su código fuente, parecería tan sencillo como realizar el proceso anterior sobre todas las sentencias del código fuente, añadiendo las LEGv8ProgramInstruction resultantes a una lista sobre la que construir el respectivo LEGv8Program.

Sin embargo, todo el planteamiento del compilador se complica en cuanto se añade un preprocesador. El requisito RF6 (detallado en la sección 5.1.1 de este documento) especifica que el simulador debe proporcionar información sobre el preprocesado del programa. Este preprocesado

debe traducir pseudo-instrucciones a instrucciones reales, constantes numéricas y etiquetas[2] por sus valores, y macros por su código equivalente.

Implementar un preprocesador implica que la compilación de los ficheros ensamblador se debe realizar en dos pasadas por el código fuente, como mínimo. Cada una de estas pasadas realiza lo siguiente:

1. Procesar las directivas de preprocesador (macros, definición de símbolos, etc.) y procesar las etiquetas de memoria.
2. Procesar las instrucciones ensamblador, generando las instancias de LEGv8ProgramInstruction correspondientes.

La cantidad final de pasadas del compilador es tres: dos para preprocesado, y una para compilación a instancias de LEGv8ProgramInstruction. El preprocesado requiere dos pasadas debido a que, en muchas ocasiones, es común utilizar en los programas ensamblador las etiquetas de memoria en instrucciones de salto que se encuentran en líneas anteriores a la definición estas etiquetas. Es decir, las instrucciones de salto que realizan un salto hacia delante en el código deben utilizar una etiqueta de memoria cuya dirección asociada todavía no se conoce. Como una pasada de un compilador procesa un fichero de arriba a abajo, sin poder dar saltos hacia atrás o hacia adelante, se deben realizar dos pasadas para poder sustituir las etiquetas de memoria y otros símbolos por sus respectivos valores.

La función final de cada pasada del compilador es:

1. Procesar las directivas de preprocesador, rellenando las tablas de símbolos, etiquetas de memoria y macros.
2. Realizar la sustitución de los símbolos, etiquetas de memoria y macros definidos en sus respectivas tablas.
3. Procesar las instrucciones ensamblador, generando las instancias de LEGv8ProgramInstruction correspondientes.

Las pasadas 1 y 3 se definen utilizando ANTLR 4, en dos ficheros distintos: el de preprocesador y el de compilador, respectivamente. La pasada 2 se define utilizando código Java en el fichero ANTLR 4 de preprocesado. ANTLR 4 permite la definición de funcionalidades con código Java en los ficheros de definición de gramáticas, insertando este código Java en las clases resultantes de las gramáticas. De esta forma, se definen dos ficheros de gramáticas:

- LEGv8Preprocessor.g4, encargado de las dos pasadas de preprocesado.
- LEGv8.g4, encargado de la pasada de compilación de código preprocesado.

[2]En ensambladores reales, la traducción de etiquetas por sus valores se realiza en compilación y no en preprocesado, simplificando la lógica del ensamblador. El simulador debe realizarlo en preprocesado, pese a que requiera una lógica más compleja, ya que es un requisito funcional.

El fichero LEGv8Preprocessor.g4 es el que presenta la lógica más compleja: define la lógica de dos pasadas, siendo la lógica de ambas no trivial. El fichero LEGv8.g4 define la lógica de una pasada, siendo esta lógica trivial, como se ha visto anteriormente.

La mayor complejidad del fichero LEGv8.g4 reside en la gran cantidad de palabras clave que hay que definir como *tokens* léxicos. Los lenguajes ensamblador definen muchas más palabras clave que los lenguajes de alto nivel, ya que cada mnemónico de una instrucción y cada nombre de un registro debe ser una palabra clave. Todas estas palabras clave deben definirse como *tokens* léxicos de la gramática. En definitiva, la gramática definida en el fichero LEGv8.g4 no presenta gran complejidad estructural, pero sí una gran cantidad de *tokens*. El fragmento de código 6.9 muestra la definición de los *tokens* léxicos asociados a instrucciones LEGv8 de formato R. El fragmento de código 6.10 muestra las reglas gramaticales que utilizan estos *tokens* para generar las correspondientes instancias de LEGv8ProgramInstruction.

```
OP3R
  // Instrucciones R que utilizan 3 registros.
  : (A D D)
  | (A D D S)
  | (A N D)
  | (A N D S)
  | (E O R)
  | (O R R)
  | (S U B)
  | (S U B S)
  ;

OP2R
  // Instrucciones R que utilizan 2 registros.
  : (L S L)
  | (L S R)
  ;
```

Fragmento de código 6.9: Definición de los *tokens* léxicos asociados a los mnemónicos de instrucciones LEGv8 de formato R que utilizan 3 y 2 registros. Estos *tokens* se utilizan en el fichero LEGv8.g4 para compilar programas ensamblador LEGv8 a objetos ejecutables por el simulador. Las reglas gramaticales que utilizan estos *tokens* se muestran en el fragmento de código 6.10.

En el fichero LEGv8Preprocessor.g4 se definen los siguientes métodos públicos:

- `public String preprocessedOutput()`, que devuelve el código fuente del programa preprocesado.
- `public Collection<Label> getLabels()`, que devuelve la lista de etiquetas de memoria del sistema.
- `public Collection<Data> getData()`, que devuelve la lista de datos de memoria especificados mediante directivas de preprocesador. Esta lista es pasada al constructor de LEGv8Program como parámetro, encargándose esta clase de escribirlos en la memoria del sistema.

```
instruction
  : OP3R rd=REG SEP rn=REG SEP rm=REG {
      program.add(LEGv8ProgramInstruction.programInstructionR(
        instructions.get($OP3R.text), $rd.text, $rn.text, $rm.text,
        $OP3R.text + " " + $rd.text + ", " + $rn.text + ", " +
          $rm.text,
        $OP3R.line, programLine, getAddress())
      );
  }
  | OP2R rd=REG SEP rn=REG SEP IMM_PREFIX number {
      program.add(LEGv8ProgramInstruction.programInstructionShift(
        instructions.get($OP2R.text), $rd.text, $rn.text, $number.val,
        $OP2R.text + " " + $rd.text + ", " + $rn.text + ", #0x" +
            Integer.toHexString($number.val),
        $OP2R.line, programLine, getAddress())
      );
  }
  ...
```

Fragmento de código 6.10: Definición de las reglas gramaticales asociadas a las instrucciones LEGv8 de formato R que utilizan 3 y 2 registros. Estas reglas generan instancias de LEGv8ProgramInstruction, que son añadidas a la lista de instrucciones de programa `program`. *REG* es el conjunto de *tokens* léxicos de los registros. *SEP* es el *token* de separador de registros, que equivale a una coma.

En el fichero LEGv8.g4 se define el siguiente método público:

- `public List<LEGv8ProgramInstruction> outputProgram()`, que devuelve la lista de las instancias de LEGv8ProgramInstruction que conforman el programa.

Estos métodos son invocados desde la clase LEGv8ProgramLoader, la clase que hace de fachada con el compilador generado por ANTLR 4. El fragmento de código 6.11 muestra el código simplificado del método de generación de programas LEGv8. Este método construye instancias de las clases generadas por las gramáticas ANTLR 4, invocando los métodos que se han definido en ellas para generar el LEGv8Program.

El preprocesador implementado soporta una variante de las directivas de preprocesador de *gas*, el ensamblador de GNU [66]. En el apéndice F se detallan las directivas de preprocesador soportadas por el preprocesador. Entre estas directivas se encuentran las directivas de creación de macros `.macro` y `.endm`.

Las macros sirven para sustituir una breve expresión en el código por un fragmento de código predefinido. El preprocesador soporta le definición de macros parametrizadas, tal y como se definen en *gas*. Este tipo de macros permiten la definición de pseudo-instrucciones ensamblador en el propio código de los programas. De esta forma, se está cumpliendo el requisito RF8 (detallado en la sección 5.1.1 de este documento). El fragmento de código 6.12 muestra la definición de la pseudo-instrucciones LEGv8 definidas en el libro de referencia, haciendo uso de macros.

La lógica para implementar macros parametrizadas es relativamente compleja. Por eso, se delega en una clase propia: Macro. Esta clase implementa todos los métodos necesarios para

```
@Override
public Program loadProgram(String source) {
  /* Preprocesado */
  LEGv8PreprocessorLexer preLexer =
      new LEGv8PreprocessorLexer(CharStreams.fromString(source));
  CommonTokenStream preTokens = new CommonTokenStream(preLexer);
  LEGv8PreprocessorParser preParser =
      new LEGv8PreprocessorParser(preTokens);
  preParser.preProgram();

  String preprocessedSource = preParser.preprocessedOutput();
  Collection<Label> programLabels = preParser.getLabels();
  Collection<Data> programData = preParser.getData();

  /* Compilacion */
  LEGv8Lexer lexer =
        new LEGv8Lexer(CharStreams.fromString(preprocessedSource));
  CommonTokenStream tokens = new CommonTokenStream(lexer);
  LEGv8Parser parser = new LEGv8Parser(tokens);
  parser.program();

  List<LEGv8ProgramInstruction> program = parser.outputProgram();

  LEGv8Program loadedProgram = new LEGv8Program(program,
      source.split("\n"), programLabels, programData);

  LEGv8Architecture.legv8.setProgram(loadedProgram);
  loadedProgram.initCPU();
  return loadedProgram;
}
```

Fragmento de código 6.11: Código simplificado del método de la clase LEGv8ProgramLoader que genera las instancias de LEGv8Program a ser ejecutas. Este método es el único que accede a las clases generadas por las gramáticas ANTLR 4. El método recibe como único parámetro la String que contiene todo el código fuente del programa.

gestionar las macros del preprocesador, incluyendo:

- la creación de macros a partir de su nombre, la lista de sus parámetros, y su definición, y
- la sustitución de las macro por el código al que expanden, a partir de la lista de valores con los que sustituir sus parámetros (`public String expand(List<String> values)`).

El fragmento de código 6.13 muestra el código ANTLR 4 encargado de la creación de macros en la primera pasada del proceso de compilación, utilizando la clase Macro. El fragmento de código 6.14 muestra el código Java encargado de la expansión de macros en la segunda pasada del proceso de compilación.

```
.macro CMP p1, p2   // Compare
SUBS XZR, \p1, \p2
.endm

.macro CMPI p, imm  // Compare immediate
SUBIS XZR, \p, \imm
.endm

.macro LDA p, label // Load address
ADDI \p, XZR, \label
.endm

.macro MOV p1, p2   // Move
ADD, \p1, XZR, \p2
.endm
```

Fragmento de código 6.12: Pseudo-instrucciones LEGv8 implementadas como macros de preprocesador.

6.2.9. Implementación de la interfaz gráfica del simulador

Como se explica en la sección 2.4 de este documento, por limitaciones de tiempo, el sistema implementa un prototipo preliminar de interfaz gráfica, en vez de una versión definitiva. Este prototipo trata de demostrar las características principales que presentará la versión definitiva de la interfaz gráfica, sin implementarlas todas ni llegar a estar preparada para su uso con usuarios reales todavía. También sienta las bases para comenzar el desarrollo de la versión definitiva de la interfaz gráfica.

El prototipo implementado sigue el diseño propuesto en la sección 5.3.2 de este documento. Se ha implementado haciendo uso de la biblioteca JavaFX, diseñada para desarrollar interfaces gráficas en Java, de forma fácil y flexible. JavaFX utiliza una estructura de árbol para organizar sus componentes. Cada componente de la interfaz es un nodo, y cada nodo se añade como hijo de otro de forma jerárquica. La raíz del árbol, un nodo instancia de Stage, representa toda la interfaz gráfica.

La biblioteca contiene clases que implementan los siguientes componentes de la interfaz gráfica:

- Botones, mediante la clase Button.
- Paneles desplegables, mediante la clase ComboBox.
- Barras de herramientas, mediante la clase ToolBar.
- Tablas, mediante las clases TableCell, TableColumn y TableView.
- Pestañas, mediante la clase Tab.
- Áreas de texto, mediante la clase TextArea.

La biblioteca utiliza un sistema de eventos para asignar funcionalidad a los componentes; como, por ejemplo, a los botones, los paneles desplegables, o las celdas de las tablas.

```
|MACRO name (args+=arg (SEP args+=arg)*)? EOL lines+=preLine* ENDM EOL {
  $txt = "\n";
  String name = $name.text;
  List<String> args = new ArrayList<>();
  for (ArgContext arg : $args) {
    args.add(arg.text);
  }
  StringBuilder definition = new StringBuilder();
  for (PreLineContext line : $lines) {
    definition.append(line.txt).append("\n");
    $txt += "\n";
  }

  macroTable.put("\\b" + name + "\\b",
    new Macro(name, args, definition.toString()));
}
    \item Este error causa que algunas l neas de c digo no se resalten en el
   editor de c digo del simulador.\\El error no ha podido ser identificado.
   Probablemente se trate de un error de l gica al utilizar las bibliotecas
   RichTextFX y JavaFX.\\Se ha tratado de corregir el error reescribiendo la
   l gica de resaltado de texto del editor de c digo , pero sin resultados.
```

Fragmento de código 6.13: Código ANTLR 4 de la directiva de preprocesador para crear macros.

La información presentada en la interfaz gráfica se implementa principalmente mediante tablas (instancias de la clase TableView). Esto incluye:

- La información relativa a los registros.
- La información relativa a la memoria.
- La información relativa a las etiquetas de memoria.
- La información relativa al programa preprocesado y el código máquina de las instrucciones (información de compilación).

La información relativa a los errores de compilaciones se implementa utilizando una instancia de TextArea. A esta instancia se le modifica la visibilidad para hacerla aparecer si se detecta algún error en la compilación, y desaparecer si el usuario pulsa la tecla de escape.

El editor de código se implementa utilizando RichTextFX [67], una biblioteca derivada de JavaFX que se especializa en el desarrollo de áreas de texto para interfaces gráficas. Una de las características que es fácilmente implementable usando RichTextFX es la de resaltado de sintaxis en editores de código. Las palabras a resaltar se pueden especificar mediante expresiones regulares con grupos de captura nombrados, como se muestra en el fragmento de código 6.15. Los colores y la forma con que resaltarlas se puede indicar mediante una hoja de estilo CSS, como la que se muestra en el fragmento de código 6.16.

Las tablas de información, instancias de TableView, utilizan características llamadas "propiedades" (Property) para determinar el valor de sus celdas de forma automática. Para ello hay que

```
/* Procesado de macros: */
for (String macroName : macroTable.keySet()) {
  Macro macro = macroTable.get(macroName);
  Pattern paramPattern = Pattern.compile("((# )?[_a-zA-Z0-9]*? , *)" +
    "{" + (macro.numParams() - 1) +  "}(# )?[_a-zA-Z0-9]*");

  Matcher macroMatcher = Pattern.compile(macroName).matcher(program);

  StringBuilder expandingProgram = new StringBuilder();
  int prevIndex = 0;
  while (macroMatcher.find()) {
    String toProcess = program.substring(macroMatcher.start());
    Matcher paramMatcher = paramPattern.matcher(toProcess);

    paramMatcher.find();
    ArrayList<String> macroParams =
      new ArrayList<>(Arrays.asList(
        paramMatcher.group().split(", *")));

    String expanded = macro.expand(macroParams);

    expandingProgram.append(program.substring(prevIndex,
      macroMatcher.start())).append(expanded);
    prevIndex = macroMatcher.start() + paramMatcher.end();
  }
  program = expandingProgram.toString() +
    program.substring(prevIndex);
}
```

Fragmento de código 6.14: Código Java de la expansión de macros del preprocesador. Las macros se procesan haciendo uso de las clases que ofrece Java para trabajar con expresiones regulares: Pattern y Matcher.

indicar a cada columna o fila de la tabla el nombre (como String) de la propiedad que debe observar. Por ejemplo, la tabla de información de los registros tiene tres columnas. Cada una de las columnas observa una propiedad distinta de los registros:

- El nombre del registro, propiedad ‘‘`name`’’.
- El número del registro, propiedad ‘‘`number`’’.
- El valor almacenado en el registro, propiedad ‘‘`value`’’.

Para que la tabla pueda observar estas propiedades de las instancias de Register, en la interfaz Register debe indicarse de forma explícita que todas las instancias presentan estas tres propiedades, y se debe proveer una forma de obtener sus valores. En otras palabras, es necesario implementar métodos *getter* de las propiedades observadas por las tablas. Estos métodos deben ser públicos, llamarse `[nombre de la propiedad]Property()` y deben devolver una instancia de Property<String> que contenga el texto con el que rellenar la tabla. Para más información sobre propiedades y tablas de JavaFX, puede leerse la documentación de la biblioteca [68]

Como consecuencia de lo expuesto en el párrafo anterior, para poder implementar las tablas de información en la interfaz gráfica, ciertas interfaces y clases del sistema deben modificarse. En

```
/* Resaltado de sintaxis */
AssemblerSyntax syn = arch.syntax();
SYNTAX = Pattern.compile(
\captionsetup{labelfont=bf}

  "(?<MNEMONIC>" + syn.getMnemonics() + ")" +
      "|(?<REG>" + syn.getRegisters() + ")" +
      "|(?<IMM>" + syn.getLiterals() + ")" +
      "|(?<COMMENT>" + syn.getComments() + ")" +
      "|(?<LABEL>" + syn.getLabels() + ")"
);
```

Fragmento de código 6.15: Código Java que genera la expresión regular que indica las palabras a resaltar en el editor de código del simulador. La expresión regular utiliza los siguientes grupos de captura nombrados para identificar los distintos tipos de palabras: MNEMONIC, REG, IMM, COMMENT y LABEL.

concreto, deben modificarse para devolver sus propiedades como instancias de Property<String>. Estas son la interfaz Register, y la clase Label. El fragmento de código 6.17 muestra la implementación de estos métodos en la interfaz Register, como métodos por defecto de Java.

Para las tablas de memoria y de información de compilación del programa no se modifica ninguna clase del sistema. En su lugar, se utilizan clases *wrapper* intermedias que incluyen estos métodos *getter* de propiedades: MemoryFragment y ProgramCompilationInfo, respectivamente. Estas clases intermedias, aparte de implementar estos métodos, procesan la información contenidas en las correspondientes instancias de Memory y Program para presentarla en las tablas. Las clases Memory y Program no están preparadas para presentar esta información de forma directa, como sí lo están Register y Label.

6.2.10. Implementación de las llamadas al sistema en el simulador

Las llamadas al sistema son la única funcionalidad del simulador que requiere la participación activa de componentes tanto de las arquitecturas concretas como de la interfaz gráfica. Las llamadas al sistema son instrucciones ensamblador que generan una operación de entrada o salida en el sistema. Concretamente, en el simulador, generan una impresión por consola o una petición de lectura de datos por consola.

Para realizar llamadas al sistema en un programa ensamblador LEGv8 se añade una nueva instrucción al conjunto de instrucciones LEGv8 soportadas por la arquitectura: SVC (*SuperVisor Call*, llamada al supervisor), del conjunto original de instrucciones ARMv8. Para distinguirla de todas las demás instrucciones LEGv8, se le asigna un formato de instrucción nuevo[3], SVC, exclusivo de esa instrucción. A nivel de funcionalidad, la instrucción SVC toma como operando un único valor inmediato, el código de la operación de entrada o salida que se solicita.

[3]ARMv8 en realidad no especifica distintos formatos estructurales para sus instrucciones. Los formatos de instrucción de LEGv8 fueron inventados por los autores Patterson y Hennessy, para ilustrar ciertos conceptos de interés didáctico de los lenguajes ensamblador. Por tanto, no se le puede asignar un formato de instrucción real a la instrucción SVC, ya que en realidad no presenta ningún formato estructural definido.

```
.mnemonic {
  -fx-fill: blue;
  -fx-font-weight:bold;
}
.reg {
  -fx-fill: red;
}
.comment {
  -fx-fill: grey;
}
.immediate {
  -fx-fill: blueviolet;
}
.label {
  -fx-fill: darkgreen;
  -fx-font-style: italic;
}
```

Fragmento de código 6.16: Ejemplo de hoja de estilo CSS con las instrucciones para resaltar las distintas palabras de los programas ensamblador en el editor del sistema. Los mnemónicos de instrucciones se resaltan con color azul y en negrita; los nombres de registros se resaltan con rojo; los comentarios se resaltan con gris; las constantes numéricas se resaltan con violeta; las etiquetas de memoria se resaltan con verde oscuro y cursiva.

Las llamadas al sistema se implementan utilizando el sistema de excepciones Java. Las clases InterruptIn e InterruptOut se hacen heredar de Throwable, de forma que puedan ejecutar abruptamente la ejecución normal de un programa. Se ha seguido esta aproximación ya que es la más similar al funcionamiento de una interrupción real. El código de ejecución de las instrucciones SVC lanza siempre una excepción de tipo InterruptIn o InterruptOut, dependiendo del código de llamada al sistema especificado en la instrucción de programa. Estas excepciones se capturan en la clase de la interfaz gráfica del simulador y son tratadas correctamente:

- Para las interrupciones de tipo InterruptOut,

    1. se recupera el mensaje que almacenan,

```
default Property<String> nameProperty() {
  return new SimpleStringProperty(name());
}

default Property<String> numberProperty() {
  return new SimpleStringProperty(Integer.toString(number()));
}

default Property<String> valueProperty() {
  return new SimpleStringProperty("0x" +
      String.format("%016x", getValue()));
}
```

Fragmento de código 6.17: Implementación de los métodos *getter* de las propiedades de Registro observadas por la tabla de registros de la interfaz gráfica del simulador.

2. se imprime este mensaje en el área de texto de la pestaña de ejecución de la interfaz gráfica (o por terminal si se ejecuta el simulador en modo terminal), y
3. se continúa la ejecución normal del programa.

- Para las interrupciones de tipo InterruptIn,

1. se activa la inserción de texto por parte del usuario en el área de texto de la pestaña de ejecución de la interfaz gráfica (si el simulador se ejecuta en modo interactivo),
2. se espera a que el usuario introduzca un dato de manera textual y pulse la tecla Enter,
3. se recupera el mensaje introducido por el usuario y se almacena en la instancia de InterruptIn capturada,
4. se "recupera" la CPU, invocando el método `recover(InterruptIn)` de la instancia de Program, pasando como parámetro la instancia de InterruptIn capturada,
5. la CPU recupera el mensaje de entrada,
6. la CPU hace un casting del mensaje al tipo de datos correspondiente (indicado por el código de llamada al sistema especificado),
7. la CPU almacena el resultado en el registro X0 o D0, dependiendo de si el tipo de datos es de tipo entero o coma flotante, respectivamente, y
8. se continúa la ejecución normal del programa.

Como llamadas al sistema concretas de LEGv8, se implementa un subconjunto de las llamadas al sistema que ofrece el simulador de la arquitectura MIPS, MARS [13]. Las llamadas al sistema soportadas se presentan en la tabla 6.1.

Servicio proporcionado	**Código**	**Argumentos**	**Resultado**
Imprimir un entero	1	X0 = entero	
Imprimir un float	2	D0 = valor en FP32	
Imprimir un double	3	D0 = valor en FP64	
Imprimir String	4	X0 = puntero a la String	
Leer un entero	5		Entero leído en X0
Leer un float	6		FP32 leído en D0
Leer un double	7		FP64 leído en D0
Leer String	8	X0 = puntero al *buffer*, X1 = longitud de la String	Se almacena la cadena leída en memoria, a partir de la dirección apuntada por X0. Se lee el número de caracteres indicados por X1.
Imprimir un carácter	11	X0 = carácter	
Leer carácter	12		Carácter leído en X0.

Tabla 6.1: Llamadas al sistema implementadas en la arquitectura LEGv8. Las llamadas al sistema son invocadas mediante la instrucción SVC, con el código de llamada correspondiente.

Como las llamadas al sistema se implementan como excepciones de Java, se deben modificar varios métodos para declarar que pueden lanzar excepciones de tipo InterruptIn e InterruptOut. Estos métodos son:

- `execute(ProgramInstruction)` en ExecutionCode.
- `execute(ProgramInstruction)` en Instruction.
- `cycle()` en Pipeline.
- `execute()` en Program.
- `executeNext()` en Program.

Las excepciones de Java, y por tanto las llamadas al sistema implementadas, se deben capturar con estructuras de código de tipo *try-catch*. Consideramos que estas estructuras son algo incómodas de utilizar, ya que fuerzan una organización del código específica a la que el programador se debe adaptar. Es por ello que consideramos que podría resultar interesante realizar, en el futuro, una revisión de la implementación de las llamadas al sistema y buscar otras alternativas a la implementación actual.

6.3. Resumen

En este capítulo se ha detallado de forma general el proceso de implementación de las principales características del simulador. Como conclusiones del capítulo podemos destacar:

- El simulador se codifica principalmente en Java 11.
- La implementación del simulador parte del diseño propuesto en el capítulo 5, pero realizando pequeñas modificaciones que facilitan la programación.
- La mayoría de las principales estructuras de datos de la arquitectura LEGv8 (banco de registros, memoria, conjunto de instrucciones, etc.) se implementan utilizando una estructura de mapa.
- El *pipeline* de la arquitectura LEGv8 es segmentado. Implementa la lógica completa de detección de riesgos, anticipación de resultados, ejecución de instrucciones en distintas unidades funcionales multiciclo, y predicción y resolución de saltos en distintas etapas y siguiendo distintas políticas.
- El compilador LEGv8 es un compilador de tres pasadas implementado mediante ANTLR 4. Incluye soporte para casi todas las directivas de preprocesador del ensamblador de GNU, *gas*.
- La interfaz gráfica del simulador utiliza la biblioteca JavaFX. El editor de texto de la interfaz gráfica utiliza la biblioteca RichTextFX, que se basa en JavaFX. La interfaz gráfica actual es un prototipo y no la versión definitiva que utilizará el simulador.

Para ampliar los conocimientos sobre la implementación del simulador, se recomienda leer su código fuente y su documentación *Javadoc* asociada.

Capítulo 7

Pruebas de validación

En este capítulo se detallan las pruebas desarrolladas para validar las características y funcionalidades del prototipo de simulador desarrollado. Las pruebas de validación desarrolladas se dividen en tres tipos:

- Pruebas de extensibilidad.
- Pruebas de caja negra.
- Pruebas de caja blanca.

Para finalizar, se detallan los errores detectados con las pruebas y su corrección.

7.1. Pruebas de extensibilidad

Una de las principales características que ha regido el diseño e implementación del simulador es la capacidad de ser extendido a otras arquitecturas de computadores y lenguajes ensamblador. Por ello, se considera oportuno realizar pruebas para determinar la facilidad de extensión del simulador a nuevas arquitecturas.

Se plantean dos pruebas que evalúan la facilidad de extensión del simulador. La facilidad de extensión del simulador se evalúa a través de dos métricas: el tiempo de desarrollo del módulo de la arquitectura a la que se añade soporte en el simulador, y la experiencia subjetiva del programador que realiza tal desarrollo.

La primera prueba planteada evalúa el proceso de extensión del simulador a una arquitectura similar a LEGv8. Las arquitecturas de este tipo presentan un gran interés didáctico. Por tanto, podría surgir la necesidad real de extender el simulador para soportar un mayor número de arquitecturas con características similares a LEGv8.

La primera prueba consiste en implementar la arquitectura RISC-V, especificada en el libro *Computer Organization and Design RISC-V edition: The Hardware/Software Interface* [30], con ciertas limitaciones. Estas limitaciones son las siguientes:

- Implementar exclusivamente el subconjunto de instrucciones RV64I, que incluye las instrucciones básicas de 64 y 32 bits de la arquitectura.
- Implementar un *pipeline* no segmentado, para reducir de forma considerable la complejidad de la arquitectura.

En el apéndice E se tratan en más detalle las características de la arquitectura RISC-V y de su subconjunto de instrucciones RV64I.

La implementación parte del módulo de la arquitectura LEGv8. Sobre ese módulo, se realizan las modificaciones de las clases pertinentes para soportar RISC-V. Las principales clases que se modifican son las siguientes:

- RISCVInstruction (derivada de LEGv8Instruction).
- RISCVProgramInstruction (derivada de LEGv8ProgramInstruction).
- RISCVInstructionSet (derivada de LEGv8InstructionSet).
- RISCVPipeline (derivada de LEGv8Pipeline).

El preprocesador y el compilador ANTLR, RISCVPreprocessor.g4 y RISCV.g4 (derivados de LEGv8Preprocessor.g4 y LEGv8.g4, respectivamente), también se modifican para soportar el lenguaje ensamblador RISC-V. El preprocesador se modifica ligeramente. El compilador se modifica de forma más extensa: principalmente se cambian los mnemónicos aceptados, aunque también se realizan cambios en otras reglas sintácticas de construcción de instrucciones.

La segunda prueba planteada evalúa el proceso de extensión del simulador a una arquitectura diferente a LEGv8. La prueba consiste en implementar la arquitectura PTX especificada en la documentación de NVIDIA [69], con ciertas limitaciones. Estas limitaciones son:

- Implementar exclusivamente un subconjunto de instrucciones reducido, similar en tamaño y capacidades al de LEGv8.
- Implementar un *pipeline* no segmentado, para reducir de forma considerable la complejidad de la arquitectura.

Esta implementación se realiza sin utilizar otro módulo de arquitectura como base.

A continuación se detallan las pruebas de extensibilidad planteadas y sus resultados:

P-Ext1.- **Extensión a la arquitectura RISC-V**.

- Resultado: satisfactorio.
- Conclusiones: El tiempo final de extensión del simulador a la arquitectura fue de aproximadamente 25 horas. El programador considera que la experiencia de desarrollo resultó sencillo.
  Consideramos que el esfuerzo dedicado a implementar la arquitectura es reducido. Deducimos, por tanto, que el esfuerzo requerido para implementar arquitecturas similares a LEGv8 en el simulador es bajo.

P-Ext2.- **Extensión a la arquitectura PTX**.

- Resultado: no realizada.
- Justificación: Se estima un tiempo necesario para desarrollar la prueba superior a 50 horas. Este tiempo supera el tiempo de planificación asignado a realizar pruebas de extensibilidad. Se prefiere invertir ese tiempo en realizar pruebas de otro tipo.
  La prueba queda planteada, pendiente de ser realizada en un futuro.

7.2. Pruebas de caja negra

Las pruebas de caja negra [70] son un método de validación de software que evalúa la funcionalidad de un sistema ignorando su implementación concreta. Se centran en la comprobación de que el componente evaluado produzca los resultados esperados, al realizar casos de uso considerados comunes.

Para validar el funcionamiento esperado del simulador en casos de uso comunes, se llevan a cabo pruebas de caja negra para el sistema completo. Las pruebas desarrolladas son de dos tipos:

- Pruebas de casos base: se valida el correcto funcionamiento del simulador ante programas correctos.
- Pruebas de casos con entradas erróneas: se valida el correcto funcionamiento del simulador ante programas erróneos.

7.2.1. Pruebas de casos base

Las pruebas de casos base consisten en la implementación y ejecución a través del simulador de programas ensamblador LEGv8 y RISC-V. Estos programas son de dos tipos:

- Programas reales de distinta naturaleza. Con estos programas se busca validar casos de uso reales de forma completa. Sirven para evaluar el funcionamiento del simulador desde el punto de vista de los usuario reales.

- Programas minimalistas que evalúan instrucciones individuales. Con estos programas se busca validar exhaustivamente el correcto funcionamiento de todas las instrucciones de las arquitecturas del simulador. Sirven para evaluar de forma sencilla una gran cantidad de casos de uso complejos, que se pueden interpretar como combinaciones de estos programas.

Para las pruebas con programas reales, se utilizan variaciones de programas descritos en los libros de referencia. En los libros de referencia estos programas se utilizan con propósito didáctico, para explicar conceptos básicos sobre los lenguajes ensamblador. Por ello, estos programas conforman una buena base de pruebas de casos de uso reales. Un ejemplo de tales programas es el algoritmo DAXPY, ilustrado en el fragmento de código 7.1.

```
// DAXPY loop in LEGv8 assembly.
.equ n_bytes, 64
.data
X:
    .space n_bytes
Y:
    .space n_bytes
Z:
    .space n_bytes
.equ k, 16

.text
...

// DAXPY init:
    MOVZ X4, #X
    MOVZ X5, #Y
    MOVZ X6, #Z
    ADDI X3, X6, #n_bytes
    MOVZ X2, #k

// DAXPY loop:
daxpy:
    LDUR X0, [X4, #0]
    MUL X0, X0, X2
    LDUR X1, [X5, #0]
    ADD X1, X1, X0
    STUR X1, [X6, #0]
    ADDI X4, X4, #8
    ADDI X5, X5, #8
    ADDI X6, X6, #8
    SUBS XZR, X6, X3
    B.NE daxpy
```

```
// DAXPY loop in RISC-V assembly.
.equ n_bytes, 64
.data
X:
    .space n_bytes
Y:
    .space n_bytes
Z:
    .space n_bytes
.equ k, 16

.text
...

// DAXPY init:
    addi s4, s4, X
    addi s5, s5, Y
    addi s6, s6, Z
    addi s3, s6, n_bytes
    addi s2, s2, k

// DAXPY loop:
daxpy:
    ld s0, 0(s4)
    mul s0, s0, s2
    ld s1, 0(s5)
    add s1, s1, s0
    sd s1, 0(s6)
    addi s4, s4, 8
    addi s5, s5, 8
    addi s6, s6, 8
    bne s6, s3, daxpy
```

Fragmento de código 7.1: Programas ensamblador que implementan el algoritmo DAXPY. A la izquierda se presenta una implementación en LEGv8. A la derecha se presenta una implementación en RISC-V.

También se utilizan otros programas ensamblador simples, desarrollados a partir de programas básicos que pretenden ilustrar conceptos fundamentales sobre programación. Ejemplos de estos programas son una implementación recursiva de una función que calcula la sucesión de Fibonacci, o una implementación de un algoritmo de multiplicación de matrices. El fragmento de código 7.2 muestra la implementación del programa que calcula toda la sucesión de Fibonacci hasta un

número dado.

Para las pruebas con programas minimalistas, se desarrollaron programas de una sola instrucción para todas las instrucciones ensamblador implementadas en las arquitecturas de LEGv8 y RISC-V.

A continuación se detallan las pruebas de casos base realizadas y sus resultados:

P-CB1.- **DAXPY en LEGv8**.

- Entrada: programa ensamblador LEGv8 del algortimo DAXPY.
- Salida esperada: ejecución correcta del programa.
  Vector resultante almacenado en memoria.
- Resultado: erróneo.
- Descripción de errores:

ERR-1.- Error de compilación al utilizar directivas de preprocesador. Error identificado y corregido.

ERR-2.- Fallo del sistema irrecuperable al tratar de ejecutar. Excepción de Java de tipo *NullPointerException*. Error identificado y corregido.

ERR-3.- Lectura incorrecta de datos de memoria. Error identificado y corregido.

P-CB2.- **DAXPY en RISC-V**

- Entrada: programa ensamblador RISC-V del algoritmo DAXPY.
- Salida esperada: ejecución correcta del programa.
  Vector resultante almacenado en memoria.
- Resultado: erróneo.
- Descripción de errores:

ERR-4.- Suma de registros con inmediatos incorrecta. Error identificado y corregido.

P-CB3.- **Programa p86 de LEGv8**

- Entrada: programa LEGv8 simple descrito en la página 86 del libro de referencia de LEGv8 [28].
- Salida esperada: ejecución correcta del programa.
  Contenidos de memoria modificados correctamente.
- Resultado: correcto.

P-CB4.- **Programa p85 de RISC-V**

- Entrada: programa RISC-V simple descrito en la página 85 del libro de referencia de RISC-V [30].
- Salida esperada: ejecución correcta del programa.
  Contenidos de memoria modificados correctamente.
- Resultado: correcto.

P-CB5.- **Programa de la sucesión de Fibonacci en LEGv8**

- Entrada: programa LEGv8 que imprime los 11 primeros elementos de la sucesión de Fibonacci utilizando una función recursiva.
- Salida esperada: Ejecución correcta del programa.
  Impresión en el simulador de los 11 primeros números de la sucesión de Fibonacci.
- Resultado: erróneo.
- Descripción de errores:

ERR-5.- Error de compilación en la definición de macros. Error identificado.

ERR-6.- Error de escritura en memoria al tratar de escribir un dato en la pila. Error identificado y corregido.

ERR-7.- Comportamiento incorrecto de las instrucciones de salto. Error identificado y corregido.

ERR-8.- Retorno de funciones incorrecto. Error identificado y corregido.

ERR-9.- Líneas de código fuente incorrectamente detectadas al utilizar macros. Error identificado y corregido.

P-CB6.- **Programa de la sucesión de Fibonacci en RISC-V**

- Entrada: programa RISC-V que imprime los 11 primeros elementos de la sucesión de Fibonacci utilizando una función recursiva.
- Salida esperada: ejecución correcta del programa.
  Impresión en el simulador de los 11 primeros números de la sucesión de Fibonacci.
- Resultado: correcto.

P-CB7.- **Programa de cálculo de un número factorial en LEGv8**

- Entrada: programa LEGv8 para el cálculo recursivo de un número factorial descrito en la página 105 del libro de referencia de LEGv8. Valor 5 en el registro X0.
- Salida esperada: ejecución correcta del programa.
  Valor 120 en el registro X1.
- Resultado: correcto.

P-CB8.- **Programa de cálculo de un número factorial en RISC-V**

- Entrada: programa RISC-V para el cálculo recursivo de un número factorial descrito en la página 103 del libro de referencia de RISC-V. Valor 5 en el registro x10.
- Salida esperada: ejecución correcta del programa.
  Valor 120 en el registro x10.
- Resultado: correcto.

P-CB9.- **Programa de multiplicación de matrices en LEGv8**

- Entrada: programa LEGv8 para el cálculo de multiplicación de matrices. Matriz identidad como primer factor, matriz arbitraria como segundo.

- Salida esperada: ejecución correcta del programa.
  Matriz arbitraria replicada en memoria.
- Resultado: correcto.

P-CB10.- **Programa de multiplicación de matrices en RISC-V**

- Entrada: programa RISC-V, para el cálculo de multiplicación de matrices. Matriz identidad como primer factor, matriz arbitraria como segundo.
- Salida esperada: ejecución correcta del programa.
  Matriz arbitraria replicada en memoria.
- Resultado: correcto.

P-CB11.- **Verificación de instrucciones LEGv8 individuales**

- Entrada: programas LEGv8 con una única instrucción. Un programa por instrucción.
- Salida esperada: ejecución correcta de todos los programas. Modificación apropiada de los componentes de la CPU simulada.
- Resultado: correcto.

P-CB12.- **Verificación de instrucciones RISC-V individuales**

- Entrada: programas RISC-V con una única instrucción. Un programa por instrucción.
- Salida esperada: ejecución correcta de todos los programas. Modificación apropiada de los componentes de la CPU simulada.
- Resultado: correcto.

7.2.2. Pruebas de casos con entradas erróneas

Para las pruebas de casos con entradas erróneas, se desarrollaron programas LEGv8 y RISC-V que presentaban errores sintácticos y semánticos. Estos incluían programas completamente erróneos, como cadenas de texto aleatorias, o programas que cometían ligeros fallos en la sintaxis de las instrucciones o directivas de preprocesador.

A continuación se detallan las pruebas de casos con entradas erróneas realizadas y sus resultados:

P-CErr1.- **Compilación de un programa erróneo**

- Entrada: texto aleatorio como programa.
- Salida esperada: error de compilación.
- Resultado: correcto.

```
.macro CMPI p, imm
    SUBIS XZR, \p, \imm
.endm
.equ n_elements, 11
    ADDI X0, X0, #0
    SUBI SP, SP, #4
loop:
    STURW X0, [SP, #0]
    BL fib
    BL print
    LDURSW X0, [SP, #0]
    CMPI X0, #n_elements
    ADDI X0, X0, #1
    B.NE loop
    B end
fib:
    SUBI SP, SP, #8
    STURW X30, [SP, #0]
    STURW X0, [SP, #4]
    CMPI X0, #1
    B.EQ fib_1
    CMPI X0, #0
    B.LE fib_0
    SUBI X0, X0, #2
    BL fib
    LDURSW X1, [SP, #4]
    STURW X0, [SP, #4]
    SUBI X0, X1, #1
    BL fib
    LDURSW X1, [SP, #4]
    ADD X0, X0, X1
    LDURSW X30, [SP, #0]
    ADDI SP, SP, #8
    BR X30
fib_1:
    ADDI X0, XZR, #1
    ADDI SP, SP, #8
    BR X30
fib_0:
    ADDI X0, XZR, #0
    ADDI SP, SP, #8
    BR X30
print:
    SVC #1
    ADDI X1, X0, #0
    ADDI X0, XZR, #32
    SVC #11
    ADDI X0, X1, #0
    BR X30
end:
```

```
.equ n_elements, 11
    addi a0, zero, 0
    addi s0, zero, n_elements
    addi sp, sp, -8
loop:
    sw a0, 0(sp)
    sw s0, 4(sp)
    jal ra, fib
    jal ra, print
    lw s0, 4(sp)
    lw a0, 0(sp)
    addi a0, a0, 1
    blt a0, s0, loop
    jal zero, end
fib:
    addi sp, sp, -8
    sw ra, 0(sp)
    sw a0, 4(sp)
    addi s1, a0, -1
    beq s1, zero, fib_1
    blt s1, zero, fib_0
    addi a0, a0, -2
    jal ra, fib
    lw s0, 4(sp)
    sw a0, 4(sp)
    addi a0, s0, -1
    jal ra, fib
    lw s0, 4(sp)
    add a0, a0, s0
    lw ra, 0(sp)
    addi sp, sp, 8
    jalr zero, ra, 0
fib_1:
    addi a0, zero, 1
    addi sp, sp, 8
    jalr zero, ra, 0
fib_0:
addi a0, zero, 0
    addi sp, sp, 8
    jalr zero, ra, 0
print:
    addi a7, zero, 1
    ecall
    addi s0, a0, 0
    addi a0, zero, 32
    addi a7, zero, 11
    ecall
    addi a0, s0, 0
    jalr zero, ra, 0
end:
```

Fragmento de código 7.2: Programas ensamblador que implementan una función de cálculo de la sucesión de Fibonacci. El cálculo se realiza de manera recursiva. A la izquierda se presenta una implementación en LEGv8. A la derecha se presenta una implementación en RISC-V.

P-CErr2.- **Error en mnemónicos**

- Entrada: programa con un mnemónico incorrecto.
- Salida esperada: error de compilación, indicando la línea del mnemónico incorrecto.
- Resultado: parcialmente correcto.
- Descripción de errores:

ERR-10.- Los errores sintácticos son detectados en la línea correcta. Los mensajes de error de compilación son poco descriptivos. Error identificado.

ERR-11.- Un único error se detecta como dos o más errores. Error identificado.

P-CErr3.- **Error en un operando**

- Entrada: programa con una instrucción con un operando del tipo incorrecto.
- Salida esperada: error de compilación, indicando la línea de la instrucción incorrecta.
- Resultado: correcto.

P-CErr4.- **Error en una etiqueta de memoria**

- Entrada: programa con una instrucción de salto a una etiqueta de memoria inexistente.
- Salida esperada: error de compilación, indicando la línea con la etiqueta errónea.
- Resultado: parcialmente correcto (error ERR-10).

P-CErr5.- **Error en el número de un registro**

- Entrada: programa LEGv8 con una instrucción que utiliza un registro inexistente, X33.
- Salida esperada: error de compilación, indicando la línea con el registro erróneo.
- Resultado: erróneo.
- Descripción de errores:

ERR-12.- Fallo del sistema irrecuperable al tratar de compilar. Error identificado.

P-CErr6.- **Escritura en una dirección de memoria errónea**

- Entrada: programa LEGv8 con una instrucción de escritura en memoria que escribe en una dirección incorrecta.
- Salida esperada: error de ejecución, notificando al usuario del error.
- Resultado: correcto.

P-CErr7.- **Salto a una dirección de memoria errónea**

- Entrada: programa LEGv8 con una instrucción de salto a una dirección fuera del rango de direcciones de memoria para instrucciones.
- Salida esperada: error de ejecución, notificando al usuario del error.
- Resultado: correcto.

P-CErr8.- **Múltiples errores en un programa**

- Entrada: programa LEGv8 con múltiples errores sintácticos de distinto tipo.
- Salida esperada: detección de todos los errores, indicando las líneas de los errores.
- Resultado: parcialmente correcto (errores ERR-10 y ERR-11).

P-CErr9.- **Error en una directiva de preprocesador**

- Entrada: programa con una directiva de preprocesador inexistente.
- Salida esperada: error de compilación, indicando el número de línea de la directiva errónea.
- Resultado: parcialmente correcto (error ERR-10).

7.3. Pruebas de caja blanca

Las pruebas de caja blanca [70] son un método de validación de software que evalúa la el funcionamiento interno de un sistema teniendo en cuenta la implementación concreta del mismo. Se centran en la comprobación del comportamiento que presenta el componente evaluado ante casos de uso generalmente poco habituales y de carácter límite.

Para validar la robustez del simulador ante situaciones inesperadas de carácter límite, se han llevado a cabo pruebas de caja blanca para el sistema completo. Las pruebas desarrolladas son de dos tipos:

- Pruebas de casos límite para el sistema simulado: se valida la robustez del simulador ante entradas de carácter límite para el sistema que se simula.
- Pruebas de casos límite para el simulador: se valida la robustez del simulador ante entradas que demandan un uso extremo de alguno de sus componentes software.

7.3.1. Pruebas de casos límite para el sistema simulado

Las pruebas de casos límite para el sistema simulado tratan de evaluar el comportamiento del simulador en casos de carácter límite para el sistema que se simula. El comportamiento del simulador y el del sistema simulado debería ser el mismo.

El planteamiento de estas pruebas es complicado, ya que en muchas ocasiones los casos límite del sistema que se simula no están especificados. Por tanto, estos casos límite son acotados por el desarrollador del simulador, basándose en la experiencia de estudio del sistema simulado.

Los principales casos límite planteados surgen de las limitaciones de codificación de las instrucciones de un programa a código máquina. La cantidad de bits dedicados a codificar las constantes numéricas de las instrucciones limita el valor de estas constantes.

Otros casos límite incluyen la escritura en las direcciones extremas de memoria, las instrucciones que producen desbordamiento, la escritura de resultados en registros especiales, y las bifurcaciones al principio y final de un programa.

A continuación se detallan las pruebas de casos límite para el sistema simulado realizadas y sus resultados:

P-CLSS1.- **Valor límite superior del campo *shamt***

- Entrada: programa LEGv8 que contiene una instrucción de desplazamiento lógico con un desplazamiento mayor igual al valor hexadecimal 0x40.
- Salida esperada: codificación de la instrucción con un 0 en el campo *shamt*.
- Resultado: correcto.

P-CLSS2.- **Valor límite superior del campo *ALU_immediate***

- Entrada: programa que contiene una instrucción de tipo I que escribe el valor hexadecimal 0x1000 en un registro.
- Salida esperada: codificación de la instrucción con un 0 en el campo *ALU_Immediate*. Tras ejecutar el programa, registro destino contiene el valor 0.
- Resultado: correcto.

P-CLSS3.- **Valor límite superior del campo *DT_address***

- Entrada: programa LEGv8 que contiene una instrucción de tipo D con un *offset* con un valor hexadecimal de 0x200.
- Salida esperada: codificación de la instrucción con un 0 en el campo *DT_address*. Tras ejecutar el programa, lectura/escritura de memoria en la dirección con un desplazamiento de 0.
- Resultado: correcto.

P-CLSS4.- **Valor límite superior del campo *BR_address***

- Entrada: programa LEGv8 que contiene una instrucción de tipo B que hace un salto a la dirección relativa 0x4000000.
- Salida esperada: codificación de la instrucción con un 0 en el campo *BR_address*. Tras ejecutar el programa, la instrucción bifurca a sí misma y entra en bucle infinito.
- Resultado:

P-CLSS5.- **Valor límite superior del campo *COND_BR_address***

- Entrada: programa LEGv8 que contiene una instrucción de tipo CB que hace un salto condicional a la dirección relativa 0x80000. El programa debe tomar la bifurcación.
- Salida esperada: codificación de la instrucción con un 0 en el campo *COND_BR_address*. Tras ejecutar, la instrucción bifurca a sí misma y entra en bucle infinito.

- Resultado: correcto.

P-CLSS6.- **Valor límite superior del campo *MOV_immediate***

- Entrada: programa que contiene una instrucción de tipo IM que escribe el valor hexadecimal 0x10000 en un registro.
- Salida esperada: codificación de la instrucción con un 0 en el campo *MOV_Immediate*. Tras ejecutar el programa, registro destino contiene el valor 0.
- Resultado: correcto.

P-CLSS7.- **Desbordamiento en registros**

- Entrada: programa que realiza una suma de los valores hexadecimales 0xffffffffffffffff ($2^{32} - 1$, el valor máximo almacenable en un registro) y 0x1
- Salida esperada: se almacena el valor 0 en el registro destino.
- Resultado: correcto.

P-CLSS8.- **Escritura en registro cero de LEGv8**

- Entrada: programa LEGv8 con una instrucción que escribe un valor distinto de cero en el registro XZR.
- Salida esperada: el registro XZR contiene el valor 0.
- Resultado: correcto.

P-CLSS9.- **Escritura en registro cero de RISC-V**

- Entrada: programa RISC-V con una instrucción que escribe un valor distinto de cero en el registro *zero*.
- Salida esperada: el registro *zero* contiene el valor 0.
- Resultado: correcto.

P-CLSS10.- **Escritura en el contador de programa**

- Entrada: programa con una instrucción que escribe en el registro del contador de programa.
- Salida esperada: error de compilación.
- Resultado: correcto.

P-CLSS11.- **Escritura en la primera dirección de memoria permitida**

- Entrada: programa con una instrucción que escribe un valor en la dirección hexadecimal de memoria 0x10000000.
- Salida esperada: escritura del valor en memoria.
- Resultado: correcto.

P-CLSS12.- **Escritura en la última dirección de memoria permitida**

- Entrada: programa con una instrucción que escribe un valor en la dirección hexadecimal de memoria 0x7ffffffc.

- Salida esperada: escritura del valor en memoria.
- Resultado: correcto.

P-CLSS13.- **Escritura en la dirección de memoria posterior a la última dirección de memoria permitida**

- Entrada: programa con una instrucción que escribe un valor en la dirección hexadecimal de memoria 0x7ffffffffd.
- Salida esperada: error de ejecución, notificando al usuario del error.
- Resultado: correcto.

P-CLSS14.- **Escritura en una dirección de memoria reservada**

- Entrada: programa con una instrucción que escribe un valor en la dirección hexadecimal de memoria 0x0.
- Salida esperada: error de ejecución, notificando al usuario del error.
- Resultado: erróneo.
- Descripción de errores:

ERR-13 Escribir en la sección de memoria reservada no causa errores de ejecución, cuando debería. Error identificado y corregido.

P-CLSS15.- **Escritura en la sección de memoria de texto**

- Entrada: programa con una instrucción que escribe un valor en la dirección hexadecimal de memoria 0x400000.
- Salida esperada: error de ejecución, notificando al usuario del error.
- Resultado: erróneo.
- Descripción de errores:

ERR-14 Escribir en la sección de memoria dedicada a las instrucciones del programa no causa errores de ejecución ni modifica las propias instrucciones del programa. Debería suceder una de las dos. Error identificado y corregido.

P-CLSS16.- **Bifurcación al inicio del programa**

- Entrada: programa con una instrucción de bifurcación que bifurca a la primera instrucción del programa.
- Salida esperada: ejecución correcta.
  El programa bifurca a la primera instrucción.
- Resultado: correcto.

P-CLSS17.- **Bifurcación al final del programa**

- Entrada: programa con una instrucción de bifurcación que bifurca más allá de la última instrucción del programa.
- Salida esperada: ejecución correcta.
  El programa bifurca al final del programa, finalizando su ejecución.
- Resultado: correcto.

7.3.2. Pruebas de casos límite para el sistema

Las pruebas de casos límite para el sistema tratan de evaluar el comportamiento del simulador en casos de uso que utilizan de forma exhaustiva alguno de sus componentes.

A continuación se detallan las pruebas de casos límite para el simulador realizadas y sus resultados:

P-CL1.- **Almacenamiento masivo en la memoria**

- Entrada: programa que escribe en bucle infinito datos en memoria, en direcciones de memoria distintas.
- Salida esperada: comportamiento correcto de la memoria.
  No se produce ningún error de ejecución debido a un fallo la estructura de datos que modela la memoria, durante la simulación de un número razonablemente alto de ciclos de reloj.
- Resultado: correcto.

P-CL2.- **Programa con un gran número de instrucciones**

- Entrada: programa que contiene un gran número de instrucciones y, por tanto un gran número de líneas de código.
- Salida esperada: compilación correcta del programa.
  No se produce ningún error de ejecución debido a un fallo de la estructura de datos que almacena las instrucciones de un programa.
  El editor de código del simulador funciona correctamente.
- Resultado: parcialmente correcto.
- Descripción de errores:

ERR-15 La funcionalidad de resaltado de sintaxis del editor de código de la interfaz gráfica falla en ocasiones.

P-CL3.- **Programa con un gran número de llamadas recursivas**

- Entrada: programa que implementa una función recursiva cualquiera con un gran número de llamadas recursivas. (Por ejemplo, un programa que calcule recursivamente el 100º número de la sucesión de Fibonacci).
- Salida esperada: ejecución correcta del programa.
  La ejecución del programa parece detenerse o entrar en bucle infinito por el gran número de llamadas recursivas, pero no se produce ningún error de ejecución.
- Resultado: correcto.

7.4. Errores detectados y correcciones

Gracias al desarrollo de las pruebas descritas en este capítulo, se detectaron múltiples errores en el funcionamiento del simulador. La mayoría de estos errores se detectaron en las primeras

pruebas de validación.

Muchos de los errores detectados han sido corregidos. Otros errores no han podido ser corregidos, principalmente por limitaciones de tiempo. Estos errores sin corregir podrían corregirse en algún momento del futuro antes de comenzar las pruebas con usuarios reales.

Los errores detectados con las pruebas de validación son los siguientes:

ERR-1.- Este error causa que no se pudieran utilizar directivas de preprocesador en los programas ensamblador ejecutados.
El error se ha identificado como un fallo en las reglas de producción de la gramática ANTLR 4 de los preprocesadores.
El error se corrige modificando ciertas reglas del preprocesador y cambiando el orden de otras.

ERR-2.- Este error produce una excepción Java de tipo *NullPointerException* al tratar de ejecutar un programa.
El error se ha identificado como un problema en el orden de inicialización de los componentes del sistema: el banco de registros y la memoria se inicializan después que el conjunto de instrucciones, por lo que todas las instrucciones se ejecutan con instancias de ambas que son nulas.
El error se corrige modificando el orden de inicialización de los componentes, haciendo que las instrucciones sean lo último en inicializarse.

ERR-3.- Este error causa que algunos datos de memoria con tamaño superior a un byte se lean incorrectamente, cargando un valor incorrecto en los registros
El error se ha identificado como un fallo al interpretar los bytes individuales que forman un dato de tamaño superior a un byte, tratándolos como bytes con signo al recomponer el valor del dato.
El error se corrige haciendo una máscara *and* de los bytes y el valor hexadecimal 0xff al recomponer el valor de datos de tamaño superior a un byte.

ERR-4.- Este error causa que las instrucciones *addi* y *addiw* de RISC-V generaran resultados incorrectos.
El error se ha identificado como un error en la lógica de ambas instrucciones, estando su lógica intercambiada.
El error se corrige intercambiando la lógica de ambas instrucciones.

ERR-5.- Este error causa un error de compilación al utilizar macros con parámetros en los programas.
El error se ha identificado como un error en el preprocesador, que identifica la declaración de los parámetros de las macros incorrectamente en ciertas circunstancias. Este error en el preprocesador surge de un conflicto con la regla léxica que reconoce caracteres escapados (`\n`, `\r`, ...) en cadenas de texto.
El error no pudo corregirse debido a que implica una modificación no trivial de una gran cantidad de reglas de la gramática del preprocesador. El tiempo estimado para realizar un cambio de tal magnitud excede el tiempo del proyecto asignado a la corrección de errores, según la planificación. Su corrección afectaría al camino crítico,

retrasando la finalización del proyecto. Como contingencia, se identificaron las circunstancias bajo las que se produce el error, pudiendo evitarlas en futuras pruebas. Se asumen las consecuencias de no corregir error.

ERR-6.- Este error causa una excepción de escritura inválida en la memoria de la CPU simulada cuando se intenta almacenar un valor en la primera dirección válida de la pila.
El error se ha identificado como un error aritmético al calcular la primera dirección válida de la pila.
El error se corrige modificando la expresión aritmética de cálculo de la dirección correctamente, de tal forma que devuelva un valor mayor en una unidad.

ERR-7.- Este error causa que las instrucciones de bifurcación condicional no bifurquen en circunstancias en las que deben hacerlo.
El error se ha identificado como un error en la lógica de la ejecución forzosa de instrucciones en el *pipeline* para procesar correctamente las bifurcaciones.
El error se corrige rediseñando y reescribiendo el método de procesamiento de bifurcaciones en el *pipeline*.

ERR-8.- Este error causa que ciertas instrucciones de bifurcación produzcan bifurcaciones incorrectas, a zonas del programa distintas de a las que deberían bifurcar.
El error se ha identificado como un error de ejecución de la instrucción LEGv8 BL, *Branch and Link* (bifurcación y enlace). Esta instrucción nunca llega a la etapa WB del *pipeline*, debido a que es eliminada en cuanto se procesa la bifurcación. Al no llegar nunca a la etapa WB, no se escribe el resultado del enlace en el registro correspondiente. Esto causa que la dirección a la que bifurcan otras instrucciones que utilizan el mismo registro sea incorrecta.
El error se corrige haciendo que las instrucciones de bifurcación no se eliminen del *pipeline* tras procesar sus bifurcaciones asociadas.

ERR-9.- Este error causa que el uso de macros en un programa ensamblador altere los números de línea que se asocian a cada instrucción del programa, haciendo que la asociación sea incorrecta. Esto se manifiesta proporcionando información errónea en la tabla de información de compilación de la interfaz gráfica del sistema.
El error se ha identificado como un error en la lógica asociada a la definición de macros en el preprocesador. Esta lógica omite saltos de línea del código fuente original.
El error se corrige modificando la lógica de definición de macros en el preprocesador para añadir los saltos de línea omitidos.

ERR-10.- Este error causa que los mensajes de información sobre errores léxicos y sintácticos en los programas ensamblador sean poco descriptivos de cara al usuario final.
El error se ha identificado como una característica de los analizadores sintácticos generados por ANTLR 4. Los mensajes de error generados por estos analizadores proporcionan información de utilidad al desarrollador del compilador, pero no al usuario que quiere compilar un programa.
El error no pudo corregirse porque requiere añadir una etapa de procesado de errores del compilador antes de proporcionar la información de los errores del programa al usuario. El tiempo estimado para añadir este sistema de procesado excede el tiempo del proyecto asignado a la corrección de errores, según la planificación. Su corrección afectaría al camino crítico, retrasando la finalización del proyecto. De todas formas, el error no se considera un error crítico que requiera una corrección inmediata.

ERR-11.- Este error causa que los errores léxicos y sintácticos en los programas ensamblador se detecten como múltiples errores.
El error se ha identificado como una característica de los analizadores sintácticos generados por ANTLR 4.
Este error está fuertemente relacionado con el error ERR-10. Como consecuencia, no pudo corregirse por los mismos motivos.

ERR-12.- Este error causa un fallo irrecuperable en el sistema al tratar de compilar programas con instrucciones que utilizan registros inexistentes.
El error se identificó como un error del compilador. ANTLR 4 no presenta ningún mecanismo de captura de errores de ejecución causados al procesar las reglas de una gramática. Como consecuencia, no presenta ningún mecanismo de gestión de los errores semánticos del compilador. Esto causa que el error producido en el banco de registros al tratar de acceder a un registro inexistente no se capture y sea devuelto como un error de compilación.
El error no pudo corregirse porque requiere añadir una etapa de captura y gestión de errores semánticos. El tiempo estimado para añadir este sistema de tratamiento de errores semánticos excede el tiempo del proyecto asignado a la corrección de errores, según la planificación. Su corrección afectaría al camino crítico, retrasando la finalización del proyecto. Se asumen las consecuencias de no corregir el error.

ERR-13.- Este error causa que la escritura y lectura de datos en la sección de memoria reservada esté permitida, cuando no debería estarlo.
El error se ha identificado como un error en la simulación de la memoria. La sección de memoria reservada no presenta ningún sistema de seguridad contra lectura o escritura.
El error se corrige añadiendo un sistema de seguridad a la sección de memoria reservada, de forma que se lance un error cada vez que se accede a ella.

ERR-14.- Este error causa que la escritura de datos en la sección de memoria de texto (dedicada a las instrucciones del programa) esté permitida sin consecuencias. La lectura en esta sección de memoria no debería estar permitida, o debería generar la edición en tiempo de ejecución de las instrucciones del programa. Este error está fuertemente relacionado con el error ERR-13.
El error se ha identificado como un error en la simulación de la memoria. La sección de memoria de texto no presenta ningún sistema de seguridad contra lectura o escritura.
El error se corrigió añadiendo un sistema de seguridad a la sección de memoria de texto, de forma que se lance un error cada vez que se accede a ella.

ERR-15.- Este error causa que algunas líneas de código no se resalten en el editor de código del simulador.
El error no ha podido ser identificado. Probablemente se trate de un error de lógica al utilizar las bibliotecas RichTextFX y JavaFX.
Se ha tratado de corregir el error reescribiendo la lógica de resaltado de texto del editor de código, pero sin resultados.

Durante el proceso de identificación y corrección de los errores anteriores se identificaron más errores en el código fuente del sistema. Estos errores son los siguientes:

ERR-16.- Este error causa que, para programas LEGv8, las distintas políticas de resolución de saltos

presenten comportamientos inconsistentes, dependiendo de la etapa del *pipeline* en que se resuelve el salto.
El error se ha identificado como un error conceptual en el simulador. La resolución de saltos en las etapas IF e ID siempre se realiza de forma ideal, mientras que la resolución de saltos en la etapa EX siempre se realiza de forma realista.
El error no pudo corregirse porque requiere realizar un proceso relativamente extenso de análisis, diseño y elección en el que participan múltiples soluciones posibles. El tiempo estimado para realizar este proceso excede el tiempo del proyecto asignado a la corrección de errores, según la planificación. Su corrección afectaría al camino crítico, retrasando la finalización del proyecto. De todas formas, el error no se considera un error crítico que requiera una corrección inmediata. Se asumen las consecuencias de no corregir el error.

ERR-17.- Este error potencialmente puede causar que los resultados de algunas instrucciones sean incorrectos bajo determinadas circunstancias.
Este error se ha identificado como un error en la simulación de la ejecución de instrucciones LEGv8 de carga desde memoria. Estas instrucciones se ejecutan en la etapa EX del *pipeline*, cuando en el libro de referencia se especifica que este tipo de instrucciones se ejecutan en la etapa MEM.
El error se corrige añadiendo lógica a la simulación del *pipeline* para ejecutar las cargas desde memoria en la etapa MEM en vez de en IF.

7.5. Resumen

En este capítulo se detallan las pruebas realizadas para validar las distintas funcionalidades del simulador. Se realizaron pruebas de los siguientes tipos:

- Pruebas de extensibilidad. Con estas pruebas se ha evaluado la capacidad del simulador de ser fácilmente extendido a otras arquitecturas.
- Pruebas de caja negra. Estas pruebas han sido a su vez de dos tipos:
  - Pruebas de casos base. Con estas pruebas se ha validado la funcionalidad del simulador ante casos de uso comunes. Han consistido en la ejecución de programas ensamblador completos especificados en el libro de referencia o derivados de programas didácticos sobre programación. También se han ejecutado programas minimalistas de una instrucción para validar de forma exhaustiva el funcionamiento de todas las instrucciones del simulador
  - Pruebas de casos con entradas erróneas. Con estas pruebas se ha validado el funcionamiento del simulador ante casos de uso con errores. Han consistido en la compilación y ejecución de programas ensamblador erróneos. Por ejemplo, programas con caracteres aleatorios, programas con ligeros errores léxicos y sintácticos, programas con múltiples errores léxicos y sintácticos, programas con errores semánticos, y programas que hacen un uso incorrecto de los componentes de la CPU simulados.
- Pruebas de caja blanca. Estas pruebas han sido a su vez de dos tipos:

- Pruebas de caso límite para el sistema simulado. Con estas pruebas se ha validado el funcionamiento correcto del simulador ante casos considerados límite para el sistema simulado. Han consistido en la ejecución en el simulador de programas ensamblador que presentan comportamientos especiales en CPUs reales. Por ejemplo, programas que codifican instrucciones con constantes numéŕicas más grandes de lo permitido, o programas que escriben en registros de solo lectura.
- Pruebas de caso límite para el simulador. Con estas pruebas se ha validado el funcionamiento del simulador ante casos de uso considerados límite. Han consistido en la ejecución de programas que hacen un uso intensivo de los componentes simulados. Por ejemplo, programas con un gran número de almacenamientos en memoria, programas con un gran número de instrucciones, y programas con un gran número de llamadas recursivas.

Gracias a las pruebas de validación realizadas, un gran número de errores en el simulador han sido identificados. La mayoría de estos errores han sido corregidos. Algunos errores no han podido ser corregidos debido a limitaciones de tiempo.

Capítulo 8

Conclusiones

En este capítulo se detallan las conclusiones del proyecto. Se exponen los siguientes aspectos:

- Los objetivos cumplidos.
- Las conclusiones principales.
- Las líneas de trabajo futuro.
- La valoración personal del proyecto.

8.1. Objetivos cumplidos

A lo largo de este proyecto se ha desarrollado un prototipo de simulador de arquitecturas de computadores y lenguajes ensamblador. El prototipo cuenta con características de interés de cara a la docencia, por lo que puede resultar de utilidad como herramienta de apoyo en asignaturas de Arquitectura de Computadores.

A continuación se incluyen las principales características del prototipo desarrollado:

- Capacidad de simulación de forma interactiva, ofreciendo un editor de texto para la edición de programas ensamblador, y una pestaña de información sobre el estado de la CPU simulada en tiempo de ejecución de los programas.
- Software multiplataforma que puede ejecutarse tanto en entornos Linux como en entornos Windows, facilitando la portabilidad.
- Soporte para múltiples arquitecturas de forma extensible, sin necesidad de recompilar el software.
- Código fuente que cumple unos criterios mínimos de calidad, mediante el seguimiento de convenciones de código y buenas prácticas, facilitando su comprensión y expansión en futuros cursos académicos.

Durante el proyecto se ha implementado en el simulador soporte para las arquitecturas ARM LEGv8 y RISC-V, dos de las arquitecturas RISC más utilizadas en contextos docentes a día de hoy. La arquitectura LEGv8 cuenta con un *pipeline* segmentado configurable y múltiples técnicas de predicción estática de saltos. Ambas características presentan un gran interés didáctico, y no suelen implementarse en los simuladores de arquitecturas de computadores. Por tanto, el prototipo desarrollado presenta funcionalidades innovadoras con respecto a los simuladores didácticos más ampliamente utilizados a día de hoy.

El prototipo ha alcanzado un punto de desarrollo estable. Cumple todos los requisitos considerados "esenciales" y la mayoría de requisitos tentativos. Presenta las funcionalidades suficientes para realizar una demostración de uso y funcionamiento del sistema. Para la finalización de la primera versión del sistema, tan solo sería necesario desarrollar una versión definitiva de la interfaz gráfica y solucionar fallos menores del sistema de compilación.

El simulador va a utilizarse como herramienta de apoyo en las asignaturas de Fundamentos de Computadoras y Arquitectura y Organización de Computadoras de la Universidad de Valladolid a partir del segundo cuatrimestre del curso 2021-2022.

El simulador se utiliza en el grupo de Investigación Trasgo de la Universidad de Valladolid para ayudar a la optimización del código de aplicaciones de HPC para dispositivos con arquitectura ARM, como NVIDIA Jetson Nano. En concreto, se ha utilizado para optimizar una aplicación de cálculo de cómputos de tipo *stencil* en sistemas heterogéneos distribuidos.

El trabajo desarrollado en este proyecto ha dado lugar a dos publicaciones de carácter científico que han sido aceptadas y serán presentadas en las XXXI Jornadas de Paralelismo 2020/2021 de la Sociedad de Arquitecturas y Tecnologías de Computadores (SARTECO), durante los días del 22 al 24 de septiembre:

- **Simulador didáctico de múltiples arquitecturas segmentadas de computadores**, presentado en la sección de Docencia en Arquitectura y Tecnología de Computadores.
- **Esqueleto paralelo genérico para computaciones stencil en sistemas heterogéneos distribuidos**, presentado en las secciones de Aplicaciones de la computación de altas prestaciones, y Arquitecturas heterogéneas y modelos de programación para GPU, FPGA y aceleradores de IA.

Actualmente, se está desarrollando una publicación científica que continúa el trabajo presentado en esas dos publicaciones. Ese trabajo pretende ser presentado en un congreso internacional cuya clasificación GII-GRIN-SCIE-C [71] es de tipo 2 relevante (core A).

8.2. Conclusiones del desarrollo

Durante el desarrollo de este proyecto se han estudiado diferentes aspectos relacionados con Arquitectura de Computadores, lenguajes ensamblador, simulación y desarrollo software. Tras

todo el tiempo dedicado al estudio de esos temas, y con la experiencia ganada por el desarrollo del simulador, se ha llegado a diversas conclusiones. De entre ellas destacamos las siguientes:

- Los simuladores son sistemas muy complejos. Su desarrollo requiere de un conocimiento exhaustivo y detallado del sistema simulado. Por ello, la documentación del sistema simulado es un material de consulta frecuente durante todo el proceso de desarrollo. Es necesario una buena documentación de un sistema para poder desarrollar un buen simulador del mismo.

- La arquitectura LEGv8 surge como intento de utilizar ARMv8 en contextos de docencia. No se trata de una arquitectura real, utilizada en procesadores de forma comercial. Tampoco se ha diseñado con propósito de ser simulada. Por tanto, en muchas ocasiones presenta incoherencias o falta de documentación. Estos defectos podrían tener justificación en un contexto didáctico en el que solo pretende ser una arquitectura para ilustrar ejemplos básicos, pero se agravan en contextos que pretenden darle un uso real a la arquitectura.

- La arquitectura RISC-V surge en una universidad, con fines didácticos y de investigación. Por tanto, está mucho mejor diseñada de cara a ser utilizada tanto en contextos de uso real como en contextos de docencia. La documentación que presenta es más extensa y accesible.

- La arquitectura completa de ARMv8 expande de manera considerable las funcionalidades presentes en LEGv8. La posibilidad de extender el simulador desarrollado para soportarla parece factible, pero requeriría un tiempo de desarrollo considerable. Asimismo, es posible que la arquitectura no presente un interés didáctico suficiente como para que realizar tal implementación sea una prioridad.

- La convención de código de no superar los 80 caracteres por línea de código fuente tenía sentido hace años, cuando los entornos de desarrollo presentaban más limitaciones (como pantallas más estrechas). Sin embargo, a día de hoy, es una práctica que puede dificultar el desarrollo y la lectura de código, sin aportar grandes beneficios de portabilidad. Estos inconvenientes son especialmente notables en código Java, donde las líneas de código pueden llegar a ser muy extensas por múltiples motivos derivados de su sintaxis.
  Actualmente, los monitores de los ordenadores son más anchos, y los editores de código más versátiles. Es habitual que, en proyectos software desarrollados en equipo, se permita un mayor número de caracteres por línea de código, como 90, 100 o 120.
  En cualquier caso, esta limitación debería ser evaluada por el equipo de desarrollo y acomodada a las necesidades del proyecto, en vez de adoptada y cumplida de forma estricta, solo por haber sido una práctica común hace años.

- Java es un lenguaje de programación que presenta características de gran utilidad en muchos contextos. Con cada actualización nueva añaden funcionalidades que facilitan el proceso de programación y expanden las posibilidades del lenguaje. Sin embargo, podría resultar interesante utilizar lenguajes más actuales para el desarrollo de proyectos similares.
  Un ejemplo ejemplo de estos lenguajes podría ser Kotlin. Kotlin es un lenguaje de programación surgido en 2016, completamente intercompatible con Java, que trata de presentar las mismas características, actualizadas al contexto de desarrollo de software actual. Podría resultar de interés continuar el desarrollo del simulador en este lenguaje, evaluando su impacto en la facilidad de comprensión, mantenimiento y extensión del código.

- Cofoja, la biblioteca que permite la programación por contratos en Java, no ofrece unas características lo suficientemente beneficiosas como para justificar su uso de forma continuada

en proyectos software. Presenta características interesantes y de gran utilidad, especialmente la definición de contratos en interfaces Java que son heredados por todas las clases que realizan las interfaces. No obstante, también presenta inconvenientes notorios, como sintaxis no muy cómoda de escribir, errores de generación de documentación, y dificultades de instalación y uso.
Asimismo, los entornos de desarrollo y el propio lenguaje Java ofrecen alternativas a los contratos, como los asertos de Java y las anotaciones de código de contratos y nulidad de IntelliJ IDEA. Estas alternativas son menos flexibles pero mucho más fáciles de utilizar, con mayor integración en los entornos de desarrollo, y con un número mucho más reducido de inconvenientes.
Cofoja lleva sin recibir actualizaciones oficiales desde agosto de 2017. La última versión disponible presenta problemas de compatibilidad con las versiones más actuales de Java. Si existiera un esfuerzo por arreglar, actualizar y expandir la biblioteca, su uso podría estar justificado en proyectos de gran extensión en el tiempo. Sin embargo, a día de hoy tal esfuerzo no existe, y la biblioteca parece haber llegado a su última versión oficial. Dadas las características e inconvenientes que presenta esa versión, no es una herramienta que podamos recomendar, y no pretendemos seguir utilizándola en el futuro para el desarrollo del simulador.

- IntelliJ IDEA es un entorno de desarrollo muy flexible, que ofrece múltiples herramientas y utilidades que facilitan notablemente el desarrollo de proyectos Java o Kotlin. Este proyecto supuso la primera toma de contacto del alumno con este entorno de desarrollo, quien considera que es el mejor entorno de desarrollo Java que ha utilizado hasta la fecha. Es una herramienta que recomendamos, y pretendemos seguir utilizándola para el desarrollo del simulador.

8.3. Trabajo futuro

El desarrollo de este trabajo abre numerosas líneas de trabajo futuro. Esas líneas de trabajo futuro pueden dividirse según el beneficio que tratan de aportar al simulador desarrollado.

El trabajo futuro relacionado con la finalización de la versión actual del simulador incluye las siguientes tareas:

- Corregir los errores no corregidos del sistema, de entre todos los errores identificados en la sección 7.4 de este documento. Estos errores incluyen aquellos relacionados con la compilación de los programas ensamblador.
- Desarrollo de la interfaz gráfica definitiva para el sistema.
- Realización de las pruebas de usabilidad con usuarios reales.
- Finalizar las pruebas de extensibilidad del simulador, añadiendo soporte para la arquitectura PTX de NVIDIA.

Además, se plantean líneas de trabajo futuro que pretenden mejorar las características actuales del sistema. Tales líneas de trabajo futuro incluyen las siguientes tareas:

- Añadir soporte para los subconjuntos de instrucciones de RISC-V de multiplicación y división (RV64M), de coma flotante (RV64F y RV64D) y atómicas (RV64A).
- Ampliar los aspectos configurables de los *pipelines* segmentados, incluyendo un mayor número de predictores estáticos de saltos, y un mayor número de unidades funcionales multiciclo.
- Añadir la capacidad de modificar, a través de la interfaz gráfica, los contenidos de los registros y la memoria de la CPU simulada, mientras se ejecuta un programa.
- Añadir la capacidad de visualizar los contenidos del *pipeline* en la interfaz gráfica del sistema.

Por último, se plantean un gran número de líneas de trabajo futuro que pretenden extender las capacidades del simulador, ofreciendo funcionalidades de gran interés didáctico. Tales líneas de trabajo futuro incluyen las siguientes tareas:

- Extender el simulador para soportar la arquitectura MIPS.
- Añadir soporte para *pipelines* genéricas, utilizables con cualquier arquitectura, y con capacidad extensible similar a la que presentan actualmente las arquitecturas del simulador.
- Implementar la capacidad de extender el simulador mediante *plug-ins*, como el *plug-in* "*x-ray*" del simulador MARS.
- Implementar soporte para simular características de interés en asignaturas sobre técnicas avanzadas de arquitectura de computadores. Por ejemplo:
  - Planificación dinámica de instrucciones, siguiendo el algoritmo de Tomasulo.
  - Ejecución especulativa y fuera de orden de instrucciones.
  - Predicción dinámica de saltos y direcciones de salto.
  - Conjuntos de instrucciones vectoriales
- Añadir la capacidad de ir hacia atrás en la ejecución de un programa, de manera similar a como permite el simulador MARS.
- Añadir la capacidad de ejecutar instrucciones de forma temporizada, de manera similar a como permite el simulador MARS.
- Añadir soporte para distintos temas estéticos del simulador y el editor de código
- Añadir funcionalidad de comprobación automática de sintaxis mientras se está programando con el editor de código del simulador, como la que incluyen una gran cantidad de entornos de desarrollo actuales.

8.4. Valoración personal del proyecto

Este proyecto ha sido mi primer contacto profesional con el área de Arquitectura de Computadores, un campo de estudio que me ha apasionado desde el momento en que cursé Fundamentos

de Computadoras en primero de carrera. Gracias a él, he podido expandir considerablemente mis conocimientos sobre Arquitectura de Computadores, aprendiendo conceptos utilizados en el desarrollo actual de microprocesadores, que presentan una gran utilidad de cara a mi futuro profesional. Además, este proyecto me ha motivado expandir aun más mis conocimientos sobre Arquitectura de Computadores y Diseño Hardware, llevándome a cursar cuatro asignaturas adicionales de forma voluntaria durante mi último cuatrimestre de carrera: Arquitecturas de Computación Avanzadas, Rendimiento y Evaluación de Computadores, Sistemas Empotrados y Hardware Empotrado. También me ha servido como introducción al Diseño de Compiladores, una rama de la informática que deseaba estudiar.

También ha supuesto mi primer proyecto software a gran escala. Me ha servido para reafirmar y expandir mis conocimientos sobre Diseño, Tecnologías e Ingeniería de Software. He aprendido técnicas de gran utilidad de cara a realizar proyectos software en entornos grupales profesionales. A nivel académico, me ha servido para acercarme a un área de la Ingeniería Informática que solo había podido ver a distancia, dado que he cursado la mención en Computación, por lo cual estoy muy agradecido.

Este proyecto me ha abierto las puertas al grupo de investigación Trasgo de la Universidad de Valladolid, cuyas labores de investigación se centran en Arquitectura y Tecnología de Computadores. He tenido el privilegio de poder realizar mis prácticas de empresa en ese grupo de investigación, y de extender mis labores en el mismo incluso más a través de la concesión de una beca de colaboración otorgada por el Ministerio de Educación y Formación Profesional. Mi trabajo en el grupo de investigación Trasgo ha dado lugar a mis dos primeras publicaciones científicas, y me ha introducido al mundo de la Computación de Alto Rendimiento, en el que quiero seguir investigando. Asimismo, me ha ayudado a decidir que quiero cursar un máster sobre Computación de Alto Rendimiento durante el próximo curso.

En general, este proyecto ha supuesto una experiencia muy enriquecedora para mí. Me ha servido para complementar mi formación en la mención de Computación con conocimientos sobre las otras dos menciones ofertadas en el grado en Ingeniería Informática de la Universidad de Valladolid: Ingeniería de Software, y Tecnologías de la Información. Me ha ayudado a expandir mis objetivos, tanto profesionales como personales, y ha reafirmado mi pasión por la informática y la Arquitectura de Computadores.

Apéndice A

Contenidos del fichero ZIP

El sistema desarrollado en este Trabajo de Fin de Grado ha sido entregado en un fichero ZIP que se incluye junto a esta memoria. El contenido del fichero ZIP es el siguiente:

- Directorio `andromeda/`: módulo del *frontend* del simulador, incluyendo la interfaz gráfica y el módulo de interfaces de las arquitecturas. El código fuente se incluye en el directorio `src/`, y los artefactos de la compilación en el directorio *build/*.
- Directorio `LEGv8/`: módulo del *backend* de LEGv8 del simulador. El código fuente se incluye en el directorio `src/`, y los artefactos de la compilación en el directorio *build/*.
- Directorio `RISC-V/`: módulo del *backend* de RISC-V del simulador. El código fuente se incluye en el directorio `src/`, y los artefactos de la compilación en el directorio *build/*.
- Directorio `examples/`: incluye ejemplos de programas ensamblador LEGv8 y RISC-V ejecutables por el simulador.

La información referente a cómo compilar, ejecutar y utilizar el simulador se detalla en el apéndice B.

Apéndice B

Manual de usuario del simulador

En este apéndice se describe el manual de usuario del prototipo de simulador desarrollado. Se tratan los siguientes aspectos:

- La compilación y ejecución del simulador.
- El uso del simulador en modo gráfico.
- La guía de contacto para la resolución de problemas.

B.1. Compilación y ejecución

B.1.1. Compilación

El simulador presenta una serie de dependencias que han de instalarse antes de poder compilarse. Las dependencias del simulador son las siguientes:

- Java 11.
- Gradle.

Para compilar el simulador o alguno de los módulos de arquitectura a partir de su código fuente, se debe abrir una terminal de comandos en el directorio raíz del módulo a compilar. En Linux, se debe ejecutar en la terminal:

```
$ gradle build
```

En Windows, se debe ejecutar en la terminal:

```
$ gradlew build
```

Actualmente, se pueden compilar los módulos localizados en las siguientes carpetas del proyecto:

- `andromeda`: módulo principal del simulador. Incluye la interfaz gráfica y el módulo de interfaces de las arquitecturas.
- `LEGv8`: módulo de la arquitectura LEGv8.
- `RISC-V`: módulo de la arquitectura RISC-V.

La compilación de los módulos generará una serie de ficheros. Entre ellos, se encontrará el fichero `jar` del módulo, localizado en la ruta:

`<directorio_raíz_del_proyecto>/<módulo>/build/libs/`

B.1.2. Ejecución

Para ejecutar el simulador con soporte para determinadas arquitecturas, es necesario que los ficheros `jar` producidos por la compilación de los módulos de esas arquitecturas se copien a la ruta:

`<directorio_raíz_del_proyecto>/andromeda/archs/`

Es necesario que el `jar` del módulo LEGv8 se encuentre incluido en dicho directorio para que el simulador funcione.

Para ejecutar el simulador, se debe abrir una terminal en el directorio `andromeda` del proyecto. En Linux, se debe ejecutar en la terminal:

```
$ gradle run
```

En Windows, se debe ejecutar en la terminal:

```
$ gradlew run
```

B.2. Uso del simulador en modo gráfico

Una vez se ha ejecutado el simulador, se abrirá una ventana como la mostrada en la figura B.1. La ventana presenta un editor de código en el que el usuario puede escribir un programa ensamblador. Un ejemplo de programa ensamblador escrito en el editor de código del simulador se puede observar en la figura B.2.

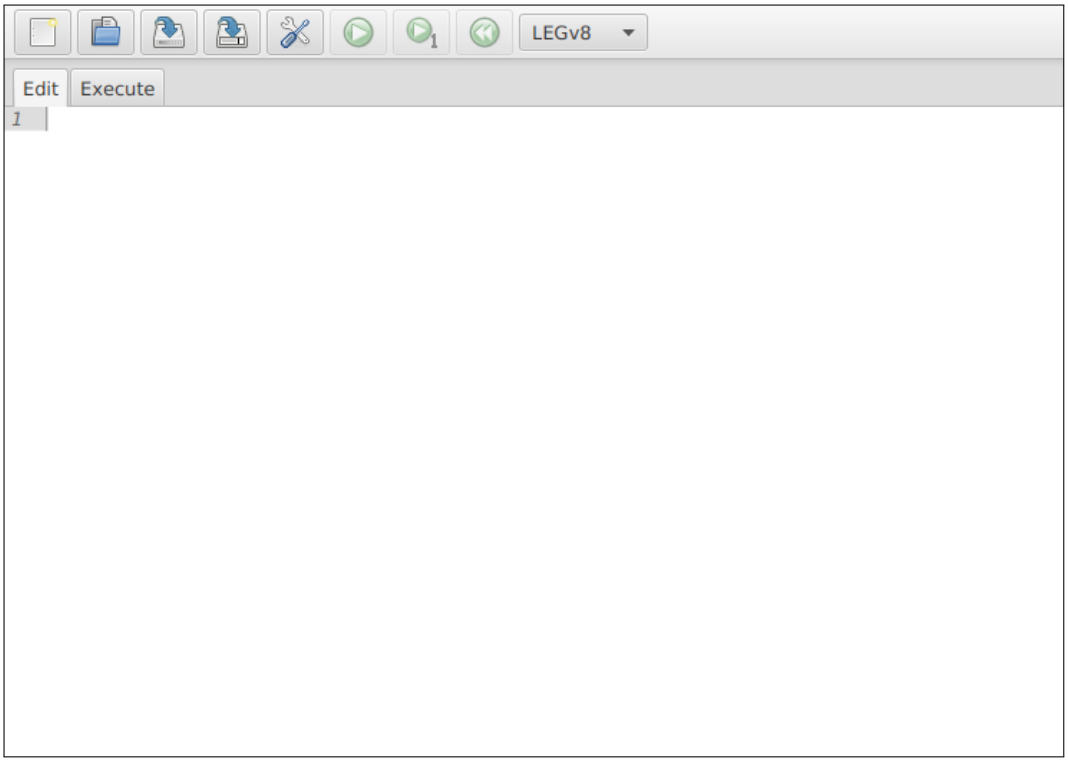

Figura B.1: Pestaña de edición del simulador, con el editor de código en blanco.

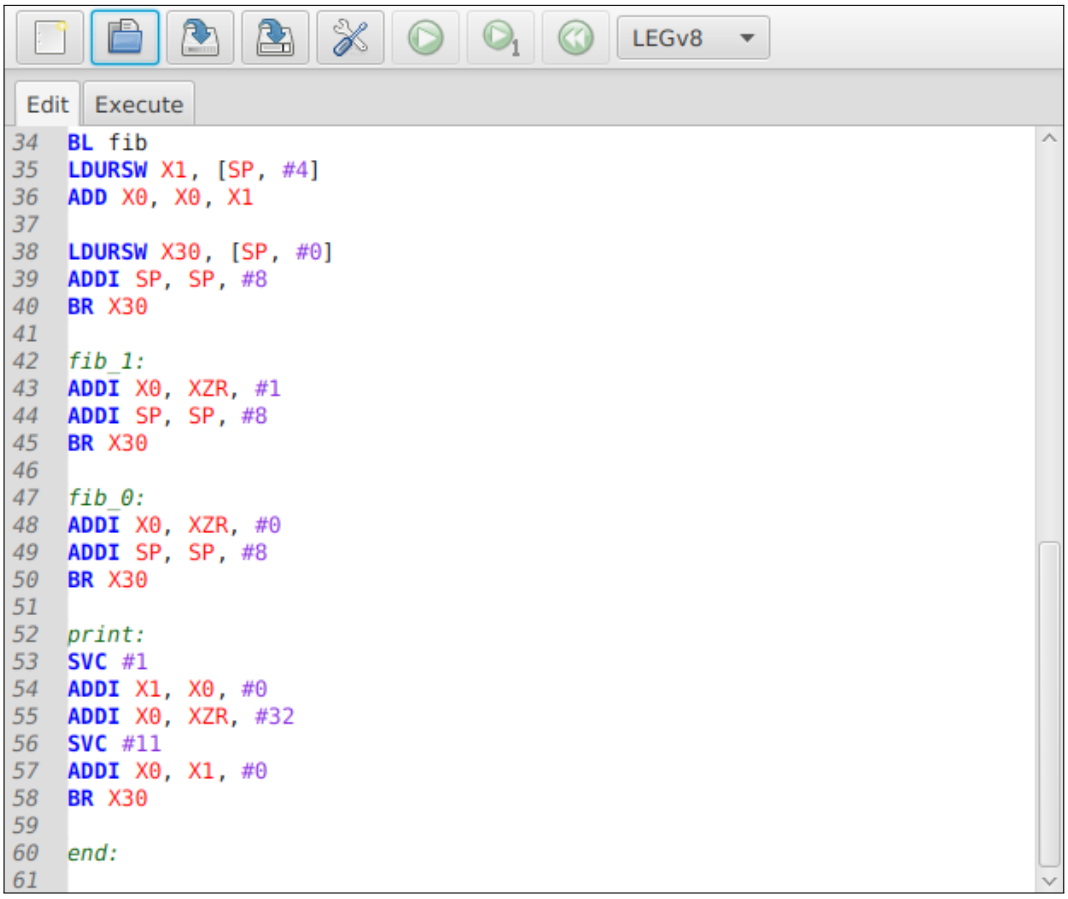

Figura B.2: Pestaña de edición del simulador, con un programa escrito en el editor de código.

La ventana gráfica del simulador presenta una barra de botones en la parte superior. La funcionalidad de cada botón de la barra, de izquierda a derecha, es la siguiente:

- Botón para limpiar el editor de código: borra todo el texto del editor de código del simulador.
- Botón de apertura de fichero: permite la apertura de un fichero ensamblador para su edición, compilación y ejecución.
- Botón de guardado de fichero: permite guardar el texto actual del editor de código en un fichero. Si no se ha abierto un fichero ensamblador existente y es la primera vez que se guarda el texto del editor de código, se abrirá una ventana pera elegir la ruta en la que guardar el fichero.
- Botón de "guardar como": permite guardar el texto actual del editor de código en un fichero. Este botón siempre abre una ventana para elegir la ruta en la que guardar el fichero.
- Botón de compilación: permite compilar el programa ensamblador escrito en el editor de código del simulador. Una vez compilado el programa, la ventana cambiará a la pestaña de ejecución, y se activará la funcionalidad de los tres botones restantes (los botones de ejecución, ejecución de un ciclo, y reinicio).
- Botón de ejecución: permite la ejecución completa del programa ensamblador compilado. La funcionalidad de este botón no se activa hasta que se compila un programa en el simulador. Si se modifica el programa en el editor de código, la funcionalidad de este botón se desactiva hasta que se vuelva a compilar el programa. Si se produce un error de ejecución en el programa, la funcionalidad de este botón se desactiva hasta que se vuelva a compilar el programa.
- Botón de ejecución de un ciclo: permite la ejecución del siguiente ciclo de reloj del programa ensamblador compilado. La funcionalidad de este botón no se activa hasta que se compila un programa en el simulador. Si se modifica el programa en el editor de código, la funcionalidad de este botón se desactiva hasta que se vuelva a compilar el programa. Si se produce un error de ejecución en el programa, la funcionalidad de este botón se desactiva hasta que se vuelva a compilar el programa.
- Botón de reinicio: permite reiniciar la ejecución del programa ensamblador compilado. La funcionalidad de este botón no se activa hasta que se compila un programa en el simulador. Si se modifica el programa en el editor de código, la funcionalidad de este botón se desactiva hasta que se vuelva a compilar el programa. Si se produce un error de ejecución en el programa, la funcionalidad de este botón se desactiva hasta que se vuelva a compilar el programa.

A la derecha del botón de reinicio se encuentra la barra desplegable de selección de arquitectura. Al clicar en el desplegable, se muestra la lista de arquitecturas disponibles en el simulador. Para cambiar la arquitectura ensamblador del simulador, se debe clicar en el nombre de la arquitectura deseada. Al cambiarse la arquitectura del simulador, se borra todo el texto del editor de código.

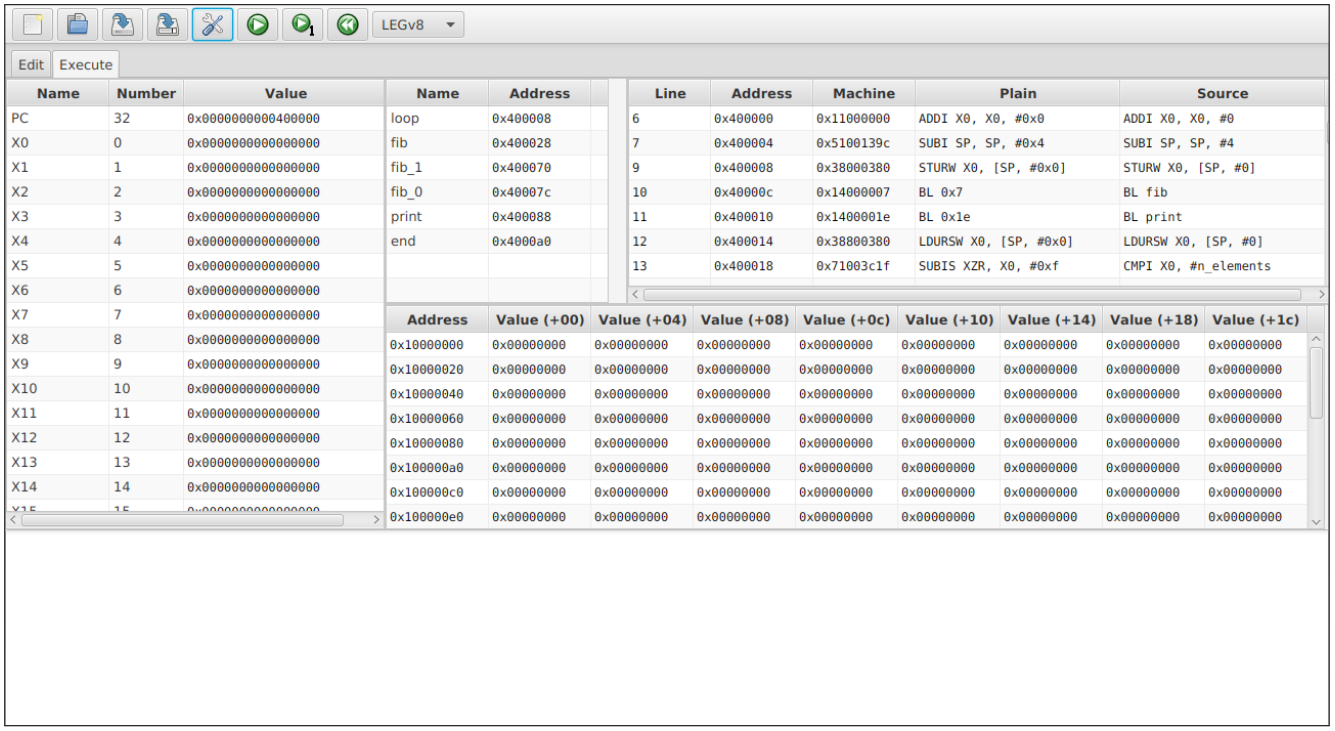

Figura B.3: Pestaña de ejecución del simulador.

En la pestaña de ejecución (*Execute*) puede observarse la información relativa al estado de CPU durante la ejecución. Esta pestaña se muestra en la figura B.3. En esta pestaña se incluye la siguiente información del estado de la CPU:

- El nombre, número y contenido de los registros.
- El nombre y la dirección de las etiquetas de memoria del programa.
- La equivalencia entre las líneas del código fuente del programa, las líneas del programa preprocesado, el código máquina de cada instrucción del programa, y la dirección de memoria de las instrucciones del programa.
- Los contenidos de un pequeño rango de direcciones de la sección de datos de la memoria.

En la parte de abajo de la pestaña de ejecución hay un cuadro de texto. Este cuadro de texto muestra los mensajes de las interrupciones de salida de los programas ensamblador. También permite la entrada de texto por parte del usuario cuando se producen interrupciones de entrada en los programas.

B.3. Ayuda y resolución de problemas

Actualmente, el simulador se encuentra en estado de prototipo y ciertas características pueden no funcionar correctamente.

Para recibir ayuda sobre la instalación y el uso del simulador, o información sobre actualizaciones, se puede escribir un correo electrónico al desarrollador:

- Manuel de Castro Caballero: `manuel.castro@alumnos.uva.es`.

Apéndice C

ARM y sus múltiples versiones

En este apéndice se trata de forma resumida las múltiples versiones existentes de la arquitectura ARM y su impacto en el desarrollo del proyecto.

Uno de los principales retos a la hora de trabajar con la arquitectura ARM LEGv8 es la dificultad por encontrar documentación útil de la misma. En parte, esto es debido a las distintas versiones existentes de la arquitectura ARM y a la dificultad por diferenciarlas.

Las distintas versiones de las arquitecturas ARM principales[1] que han existido a lo largo de la historia se pueden resumir como se indica a continuación:

- Arquitecturas antiguas (*legacy*). Estas soportan el mismo conjunto de instrucciones de 32 bits, y utilizan el mismo lenguaje ensamblador, en cuanto a sintaxis y elementos léxicos se refiere (con ciertas diferencias dependiendo de las capacidades del microprocesador específico). Entre estas arquitecturas se incluyen las versiones ARMv2, ARMv3, ARMv4T, ARMv5 y ARMv6.
- Arquitecturas actuales. A su vez, en estas arquitecturas se distingue:
  - ARMv7. Esta es una arquitectura de 32 bits. Soporta el mismo conjunto de instrucciones y lenguaje ensamblador que las arquitecturas *legacy*. Sin embargo, es la única arquitectura de 32 bits con la que la arquitectura ARMv8 presenta compatibilidad. Por tanto, desde que existe la arquitectura ARMv8, a la arquitectura ARMv7 también se la conoce como AArch32. También, se renombró el conjunto de instrucciones soportado por las arquitecturas *legacy* a A32. En algunos contextos, a esta arquitectura también se la conoce como ARM32
  - ARMv8. Esta arquitectura supuso un cambio fundamental en las arquitecturas ARM. Además de poder soportar la arquitectura AArch32, ARMv8 es la primera arquitectura en añadir soporte opcional para una arquitectura de 64 bits, denominada AArch64.

[1] Los microprocesadores ARM suelen presentar soporte para un conjunto de instrucciones secundario, denominado Thumb. También existen otros conjuntos de instrucciones secundarios, como Thumb-2 o Jazelle. La discusión de estos conjuntos secundarios está fuera del ámbito de este proyecto. En este apéndice nos centramos exclusivamente en las versiones principales de ARM.

> En algunos contextos, a esta arquitectura también se la conoce como ARM64. El conjunto de instrucciones de AArch64 se conoce como A64. AArch64 es completamente compatible con AArch32. Por tanto, podría decirse que ARMv8 no es una única arquitectura, sino dos.

Además, los microprocesadores ARM que se comercializan suelen tener nombres muy parecidos a las arquitecturas ARM, lo que puede llevar a confusiones. Ejemplo de esto son los microprocesadores ARM7, ARM8, ARM9, ARM10, ARM11, etc.

Los autores Patterson y Hennessy, diseñadores de LEGv8, especifican en [28] que LEGv8está basado en ARMv8. En la sección 2.3 del libro de referencia, página 73, detallan que LEGv8 es un subconjunto de A64. Sin embargo, durante todo el resto del libro, tratan LEGv8 como "un subconjunto de de instrucciones de ARMv8". En la sección 2.1 del libro, página 62, en la que se introduce LEGv8, los autores dicen:

> Usaremos el término ARMv8 cuando hablemos del conjunto de instrucciones completo original, y LEGv8 cuando nos refiramos al subconjunto didáctico, que está evidentemente basado en el conjunto de instrucciones ARMv8.

La forma en la que los autores utilizan los términos ARMv8 y LEGv8 en el libro de referencia puede ser causa de confusiones a la hora de buscar documentación sobre el subconjunto original (AArch64). Su aproximación a LEGv8 es principalmente didáctica, no estando interesados en diseñar una arquitectura real que se pueda implementar en microprocesadores comerciales. Es por ello que se suelen omitir detalles de definición e implementación del subconjunto.

Añadido a esto, ARM tampoco expone de forma directa y explícita esta distinción entre todas sus versiones. Además, al no tratarse de una arquitectura oficial, ARM no ofrece ninguna documentación sobre LEGv8. ARM provee documentación *online* para todas las versiones descritas anteriormente. Esta documentación se ciñe de forma exclusiva a su respectiva versión de la arquitectura, por lo que en cada documento no se hace referencia a las demás versiones ni a las distinciones que puedan existir entre ellas. Es el desarrollador el que debe buscar y encontrar la documentación correcta entre todas las documentaciones que ofrece ARM.

Para la realización de este proyecto, la documentación ARM que se utilizó finalmente, de entre todas las encontradas, es *Arm Architecture Reference Manual - Armv8, for Armv8-A architecture profile* [64]. Este manual detalla la implementación de las arquitecturas AArch64 y AArch32 en los microprocesadores con arquitectura ARMv8-A.

Apéndice D

MARS

En este apéndice se trata de forma resumida el simulador *MARS* y su impacto en el desarrollo del proyecto.

MARS [13] es un simulador del lenguaje ensamblador MIPS con propósito didáctico. Fue desarrollado por los doctores Pete Sanderson (Universidad de Otterbein, Ohio, Estados Unidos) y Ken Vollmar (Universidad estatal de Missouri, Missouri, Estados Unidos). Su desarrollo comenzó en 2003, y su última versión fue liberada en agosto de 2014.

MARS fue diseñado específicamente para cubrir las necesidades de estudiantes de grado y sus profesores. Los conceptos que implementa son ampliamente tratados en el libro de Patterson y Hennessy sobre MIPS [29], que durante muchos años ha servido como base para asignaturas sobre Arquitectura de Computadores.

MARS ha sido una gran inspiración de cara a desarrollar este proyecto. El motivo principal es que es el simulador de lenguajes ensamblador actualmente utilizado en todas las asignaturas sobre Arquitectura de Computadores del grado en Ingeniería Informática de la Universidad de Valladolid [4–6]. Tanto el alumno que ha realizado este proyecto como sus tutores académicos están ampliamente familiarizados con *MARS*. Es por ello que una gran cantidad de características del simulador desarrollado son similares a las que presenta *MARS*:

- Dos pestañas en la interfaz gráfica: una de edición de código y otra de ejecución de programas.
- Se provee información sobre los registros de la CPU, la memoria de la CPU, las etiquetas de memoria del programa, y la compilación del programa (instrucciones preprocesadas y código máquina equivalente).
- Se permite la ejecución de programas completos, así como la ejecución ciclo a ciclo de programas.
- Las llamadas al sistema implementadas en las arquitecturas del simulador desarrollado son un subconjunto de las llamadas al sistema de *MARS*.

Durante el desarrollo del proyecto, se ha tratado de mejorar el diseño de *MARS*, prescindiendo

de características muy complejas o de poca utilidad, y añadiendo otras que consideran mejores. Algunas de las características de *MARS* que el prototipo de simulador desarrollado no presenta (todavía) son las siguientes:

- Control de la velocidad de ejecución de los programas, pudiendo configurar un tiempo de retardo entre la ejecución de cada instrucción.
- Opción a deshacer la ejecución de instrucciones, instrucción a instrucción.
- Edición de los valores de los registros y la memoria de la CPU por parte del usuario, en tiempo de ejecución.
- Extensiones que añaden funcionalidad al simulador (como el*x-ray*, que permite la visualización de los componentes hardware de la CPU simulada).

Algunas de las características del prototipo de simulador desarrollado que *MARS* no presenta son:

- Capacidad extensible a nuevas arquitecturas.
- Soporte para *pipelines* segmentados.
- Clara diferenciación entre la pestaña de edición y la pestaña de ejecución.

Cabe destacar que, para el desarrollo del prototipo de simulador, se ha consultado el código fuente de MARS. Sin embargo, no se utiliza nada de su código fuente en el código del prototipo desarrollado.

D.1. *MARS-F*

MARS-F es una versión actualizada de *MARS* desarrollada por el doctor Francisco José Andújar Muñoz. Esta versión se utiliza en la asignatura Arquitecturas de Computación Avanzadas [6] del grado en Ingeniería Informática de la Universidad de Valladolid.

Esta versión implementa *pipelines* segmentados con unidades funcionales multiciclo en *MARS*, junto con técnicas de planificación dinámica de instrucciones. Sin embargo, estas nuevas características solo están disponibles en el modo de ejecución en terminal, y el modo de ejecución gráfico no presenta ninguna actualización notable.

MARS-F ha servido de inspiración a este proyecto de cara a implementar los *pipelines* segmentados con unidades multiciclo. En ese aspecto, el prototipo de simulador desarrollado trata de comenzar el proceso de integración de los *pipelines* segmentados en un simulador gráfico. Con esto se busca hacer más accesible el aprendizaje de características avanzadas de los computadores, como lo son los *pipelines* segmentados con unidades multiciclo, mediante el uso de simuladores.

Entre las líneas de trabajo futuro propuestas en la sección 8.3 de este documento se incluye la extensión de las funcionalidades del simulador con las añadidas a *MARS* en *MARS-F*.

Apéndice E

Arquitectura RISC-V

En este apéndice se explica de forma resumida la arquitectura RISC-V y sus características.

RISC-V es una arquitectura de computadores de hardware libre. Su conjunto de instrucciones es libre y abierto, pudiendo usarse sin royalties para cualquier propósito. La arquitectura define variantes en 32, 64 y 128 bits; aunque la variante de 128 bits permanece indefinida de forma intencionada debido a la poca experiencia práctica existente con sistemas de esas características.

El proyecto surgió en 2010 en la universidad de California, Berkeley, y contó con la ayuda de David A. Patterson, el coautor del libro de referencia de LEGv8 [28]. El proyecto tenía como propósito diseñar un conjunto de instrucciones libre, con interés didáctico y en investigación. Para ello, parte de los conceptos originales de las arquitecturas RISC.

En cuanto a especificaciones técnicas, la versión de 64 bits de RISC-V es muy similar a LEGv8:

- Arquitectura de diseño RISC y de tipo registro-registro.
- 31 registros de 64 bits de propósito general, y un registro que almacena la constante cero.
- Espacio de direcciones de 64 bits con palabras de 32 bits.
- Instrucciones ensamblador de 32 bits. A diferencia de LEGv8, RISC-V presenta códigos de operación de tamaño fijo.
- Múltiples formatos de instrucción. Los formatos de instrucción de RISC-V son los siguientes:
  - Formato R: para instrucciones aritmético-lógicas que trabajan sobre registros.
  - Formato I: para instrucciones aritmético-lógicas que trabajan sobre registros y valores numéricos constantes (inmediatos), y para instrucciones de carga desde memoria.
  - Formato S: para instrucciones de almacenamiento en memoria.
  - Formato SB: para instrucciones de bifurcación (salto) condicionales.
  - Formato U: para instrucciones que trabajan con valores numéricos constantes (inmediatos) de gran tamaño.
  - Formato UJ: para instrucciones de salto incondicionales.

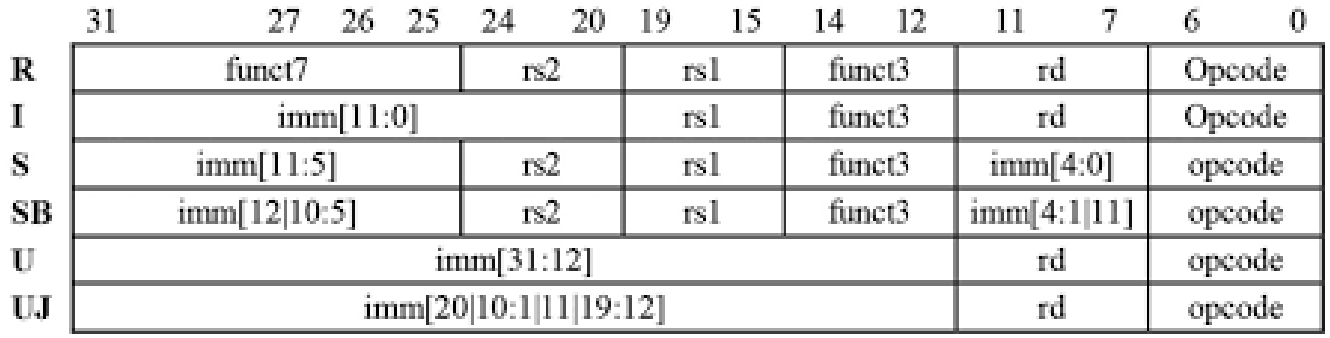

Figura E.1: Codificación de los distintos formatos de instrucción presentes en la arquitectura RISC-V. Extraído de [30]. *Opcode* es el código de operación. *rs1*, *rs2* y *Rd* son los registros que participan en la ejecución de la instrucción, tanto los operandos como el destino. *funct7* y *funct3* se utilizan en conjunto con el código de operación para distinguir los distintos tipos de instrucciones. Los distintos campos *imm* codifican los distintos bits de los valores numéricos constantes sobre los que trabajan las instrucciones.

La figura E.1 muestra la codificación de los distintos formatos de instrucciones RISC-V. La tabla E.1 muestra las instrucciones RISC-V que se han implementado en el respectivo *backend* del simulador desarrollado. Estas instrucciones forman parte del del núcleo RV64I (instrucciones de enteros básicas) de RISC-V.

Instrucción RISC-V	**Mnemónico**	**Formato**
Suma	add, addw	R
Suma inmediato	addi, addiw	I
Máscara and	and	R
Máscara and con inmediato	andi	I
Suma inmediato grande al contador de programa	auipc	U
Bifurca si igual	beq	SB
Bifurca si mayor o igual	bge	SB
Bifurca si mayor o igual, sin signo	bgeu	SB
Bifurca si menor que	blt	SB
Bifurca si menor que, sin signo	bltu	SB
Bifurca si distinto de	bne	SB
Llamada al sistema	ecall	I
Salta y enlaza	jal	UJ
Salta a registro y enlaza	jalr	I
Carga un byte	lb	I
Carga un byte sin signo	lbu	I
Carga una palabra doble (8 bytes)	ld	I
Carga media palabra (2 bytes)	lh	I

Carga media palabra sin signo (2 bytes)	lhu	I
Carga inmediato grande	lui	I
Carga palabra (4 bytes)	lw	I
Carga palabra sin signo (4 bytes)	lwu	I
Máscara or	or	I
Máscara or con inmediato	ori	I
Almacena un byte	sb	S
Almacena una palabra doble (8 bytes)	sd	S
Almacena media palabra (2 bytes)	sh	S
Desplazamiento lógico a la izquierda	sll, slw	R
Desplazamiento lógico a la izquierda con inmediato	slli, slliw	I
Compara si menor	slt	R
Compara si menor con inmediato	slti	I
Compara si menor con inmediato sin signo	sltiu	I
Compara si menor sin signo	sltu	I
Desplazamiento aritmético a la derecha	sra	R
Desplazamiento aritmético a la derecha con inmediato	srai	I
Desplazamiento lógico a la derecha	srl	R
Desplazamiento lógico a la derecha con inmediato	srli	I
Resta	sub, subw	R
Almacena una palabra (4 bytes)	sw	S
Máscara or exclusiva	xor	R
Máscara or exclusiva con inmediato	xori	I

Tabla E.1: Conjunto de instrucciones RV64I implementadas en el *backend* de RISC-V del simulador. Las instrucciones con dos mnemónicos pueden trabajar con registros de 64 bits (si el mnemónico no termina en w) o con registros de 32 bits (si el mnemónico termina en w).

A diferencia de LEGv8, RISC-V no tiene códigos ni banderas de condición.

Además del núcleo RV64I, la implementación de RISC-V detallada en [30] implementa las siguientes extensiones:

- Extensión de multiplicación (RV64M): añade instrucciones de multiplicación y división, similares a las incluidas en la extensión aritmética de LEGv8.
- Extensiones de coma flotante (RV64F y RV64D): añade instrucciones de tratamiento de datos en formato coma flotante, como las incluidas en la extensión aritmética de LEGv8.
- Extensión de instrucciones atómicas (RV64A).

Entre las líneas de trabajo futuro propuestas en la sección 8.3 de este documento se incluye la extensión del *backend* de RISC-V para incluir estas extensiones.

Apéndice F

Directivas de preprocesador del ensamblador

En este apéndice se incluye un resumen de las directivas de preprocesador implementadas en los preprocesadores de los *backends* del simulador desarrollado.

Las directivas de preprocesador implementadas se muestran en la tabla F.1. Estas directivas son variaciones de las directivas del ensamblador de GNU para ARM [72].

Directiva de preprocesador	**Descripción**
`.ascii "<string>"`	Inserta la cadena de texto como datos en la memoria. Similar a `DCB` en el ensamblador de ARM.
`.asciiz "<string>"`	Similar a `.ascii`, pero termina la cadena de texto con un byte de 0.
`.balign <potencia_de_2> {, <valor> {, <máximo>}}`	Alinea la dirección a <potencia_de_2>. La alineación se produce añadiendo bytes con el valor <valor>, o 0 por defecto. La alineación no ocurrirá si hay que insertar más de <máximo> bytes. Similar a `ALIGN` en el ensamblador de ARM.
`.byte <byte1> {, <byte2>} ...`	Inserta una lista de bytes como datos en la memoria. Similar a `DCB` en el ensamblador de ARM.
`.else`	Se usa con `.if` y `.endif`. Similar a `ELSE` en el ensamblador de ARM.
`.end`	Marca el final del programa ensamblador. Normalmente omitido.*
`.endif`	Marca el final de un bloque de compilación condicional. Ver `.ifm`, `.ifdef`, `.ifndef`. Similar a `ENDIF` en el ensamblador de ARM.
`.endm`	Marca el final de la definición de una macro. Ver `.macro`. Similar a `MEND` en el ensamblador de ARM.
`.endr`	Finaliza un bloque de repetición. Ver `.rept` y `.irp`. Similar a `WEND` en el ensamblador de ARM.

`.equ <símbolo>, <valor>`	Apunta el valor de un símbolo. Similar a `EQU` en el ensamblador de ARM.
`.err`	Causa que el proceso de compilación finalice con un error.*
`.hword <short1> {, <short2>} ...`	Inserta una lista de medias palabras como datos en la memoria. Similar a `DCW` en el ensamblador de ARM.
`.if <expresión_lógica>`	Inicia un bloque de código de compilación condicional. Ver `.endif` y `.else`. Similar a `IF` en el ensamblador de ARM.
`.ifdef <símbolo>`	Inicia un bloque de código que se compila si <símbolo> está definido. Ver `.endif`.
`.ifndef <símbolo>`	Inicia un bloque de código que se compila si <símbolo> no está definido. Ver `.endif`.
`.irp <parámetro> {, <val_1>} {, <val_2>}...`	Repite un bloque de código una vez por cada valor en la lista. Ver `.ednr`. En el bloque de código repetido, se sustituye \<parámetro> por el respectivo valor de la lista.*
`.macro <nombre> {<arg_1>} {, <arg_2>}... {, <arg_N>}`	Inicia la definición de una macro ensamblador llamada <nombre> con N parámetros. En la definición de la macro, los parámetros a sustituir deben indicarse precedidos por \. Ver `.endm`. Similar a `MACRO` en el ensamblador de ARM.
`.rept <número_de_veces>`	Repite un bloque de código el número de veces indicado. Ver `.endr`.
`.req <símbolo>, <texto>`	Define un símbolo que será sustituido por un texto. Intencionado para renombrar registros.
`.data`	Inicia una sección de datos.
`.text`	Inicia una sección de código.
`.set <símbolo>, <valor>`	Define el valor de un símbolo. Parecido a `.req` y `.equ`. Similar a `SETA` en el ensamblador de ARM.
`.space <num_bytes> {, <valor>}`	Reserva el número de bytes indicado. Los bytes se llenan con 0 o con <valor>, si se especifica. Similar a `SPACE` en el ensamblador de ARM.
`.word <word1> {, <word2>}...`	Inserta una lista de palabras (4 bytes) como datos en la memoria. Similar a `DCD` en el ensamblador de ARM.

Tabla F.1: Directivas de preprocesador soportadas por los preprocesadores del simulador desarrollado. También se incluyen las directivas del ensamblador de ARM equivalentes. Las directivas cuya descripción termina en asterisco (*) se encuentran parcialmente implementadas en el simulador. por el momento.

Bibliografía

[1] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, Jan 2015.

[2] Alexander S Gillis. internet of things (iot). Página de acceso: `https://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT`, último acceso: Julio 2021.

[3] Universidad de Valladolid. Plan de estudios del grado en ingeniería informática, 2020/2021. Página de acceso: `http://www.uva.es/resources/docencia/_ficheros/2020/545/asignaturas.pdf`, último acceso: Junio 2021.

[4] Universidad de Valladolid. Guía docente de la asignatura fundamentos de computadoras, 2020/2021. Página de acceso: `https://albergueweb1.uva.es/guia_docente/uploads/2020/545/46907/1/Documento.pdf`, último acceso: Junio 2021.

[5] Universidad de Valladolid. Guía docente de la asignatura arquitectura y organización de computadoras, 2020/2021. Página de acceso: `https://albergueweb1.uva.es/guia_docente/uploads/2020/545/46911/1/Documento.pdf`, último acceso: Junio 2021.

[6] Universidad de Valladolid. Guía docente de la asignatura arquitecturas de computación avanzadas, 2020/2021. Página de acceso: `https://albergueweb1.uva.es/guia_docente/uploads/2020/545/46943/1/Documento.pdf`, último acceso: Junio 2021.

[7] Universidad de Valladolid. Guía docente de la asignatura estructura de sistemas operativos, 2020/2021. Página de acceso: `https://albergueweb1.uva.es/guia_docente/uploads/2020/545/46915/1/Documento.pdf`, último acceso: Julio 2021.

[8] Universidad de Valladolid. Guía docente de la asignatura lenguajes de programación, 2020/2021. Página de acceso: `https://albergueweb1.uva.es/guia_docente/uploads/2020/545/46926/1/Documento.pdf`, último acceso: Julio 2021.

[9] Universidad de Valladolid. Guía docente de la asignatura computación paralela, 2020/2021. Página de acceso: `https://albergueweb1.uva.es/guia_docente/uploads/2020/545/46929/1/Documento.pdf`, último acceso: Julio 2021.

[10] Universidad de Valladolid. Guía docente de la asignatura sistemas empotrados, 2020/2021. Página de acceso: `https://albergueweb1.uva.es/guia_docente/uploads/2021/545/46941/1/Documento.pdf`, último acceso: Julio 2021.

[11] Universidad de Valladolid. Guía docente de la asignatura rendimiento y evaluación de computadores, 2020/2021. Página de acceso: `https://albergueweb1.uva.es/guia_docente/uploads/2021/545/46968/1/Documento.pdf`, último acceso: Julio 2021.

[12] Universidad de Valladolid. Guía docente de la asignatura hardware empotrado, 2020/2021. Página de acceso: `https://albergueweb1.uva.es/guia_docente/uploads/2021/545/46967/1/Documento.pdf`, último acceso: Junio 2021.

[13] Kenneth Vollmar and Pete Sanderson. Mars: an education-oriented mips assembly language simulator. volume 38, pages 239–243, 03 2006.

[14] Priati Assiroj, April Lia Hananto, Ahmad Fauzi, and Harco Leslie Hendric Spits Warnars. High performance computing (hpc) implementation: A survey. In *2018 Indonesian Association for Pattern Recognition International Conference (INAPR)*, pages 213–217, 2018.

[15] Francisco José Andújar and Javier Cano-Cano. Hiperion. Página principal: `https://gitraap.i3a.info/fandujar/hiperion`, último acceso: Julio 2021.

[16] M. Sourouri, J. Langguth, F. Spiga, S. B. Baden, and X. Cai. Cpu+gpu programming of stencil computations for resource-efficient use of gpu clusters. In *2015 IEEE 18th International Conference on Computational Science and Engineering*, pages 17–26, Oct 2015.

[17] NVIDIA. Jetson nano developer kit. Página principal: `https://developer.nvidia.com/embedded/jetson-nano-developer-kit`, último acceso: Julio 2021.

[18] Joana Geraldi and Thomas Lechter. Gantt charts revisited: A critical analysis of its roots and implications to the management of projects today. *International Journal of Managing Projects in Business*, 5, 09 2012.

[19] Agile Alliance. What is agile? Página de acceso: `https://www.agilealliance.org/agile101/`, último acceso: Julio 2021.

[20] Terence Parr. Extreme programming. Página de acceso: `https://www.cs.usfca.edu/~parrt/course/601/lectures/xp.html`, último acceso: Julio 2021.

[21] Scrum Guides. The 2020 scrum guide. Página de acceso: `https://scrumguides.org/scrum-guide.html`, último acceso: Julio 2021.

[22] Bob Hughes and Mike Cotterel. Software project management. pages 162–188, May 2009.

[23] Tom Bartenstein. x86: An historical perspective. Página de acceso: `http://www.cs.binghamton.edu/~tbartens/CS220_Spring_2019/lectures/L13_X86_History.pdf`, último acceso: Julio 2021.

[24] ARM. Learn the architecture: Introducing the arm architecture. Página de acceso: `https://developer.arm.com/documentation/102404/latest`, último acceso: Junio 2021.

[25] Architecting a smart world and powering artificial intelligence: Arm. *Silicon Review*, october 2019. Página de acceso: `https://thesiliconreview.com/magazine/profile/architecting-a-smart-world-and-powering-artificial-intelligence-arm`, último acceso: Mayo 2021.

[26] MIPS Technologies. Mips32 architecture. Página de acceso: `https://www.mips.com/products/architectures/mips32-2/`, último acceso: Julio 2021.

[27] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovic. The risc-v instruction set manual, volume i: Base user-level isa, 2011.

[28] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware Software Interface ARM Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2016.

[29] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[30] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. ISSN. Elsevier Science, 2017.

[31] NVIDIA Developer. Cuda zone. Página principal: `https://developer.nvidia.com/cuda-zone`, último acceso: Julio 2021.

[32] Intel. Intel oneapi toolits. Página principal: `https://software.intel.com/content/www/us/en/develop/tools/oneapi.html#gs.65k1lw`, último acceso: Julio 2021.

[33] James Larus. Spim: A mips32 simulator. Página principal: `http://spimsimulator.sourceforge.net/`, último acceso: Julio 2021.

[34] Pavel Pisa and Fanda Vacek. Qtmips. Página principal: `https://github.com/cvut/QtMips`, último acceso: Julio 2021.

[35] Irina Branovic, Roberto Giorgi, and Enrico Martinelli. Webmips: A new web-based mips simulation environment for computer architecture education. In *Proceedings of the 2004 Workshop on Computer Architecture Education: Held in Conjunction with the 31st International Symposium on Computer Architecture*, WCAE '04, page 19–es, New York, NY, USA, 2004. Association for Computing Machinery. Página de acceso: `https://doi.org/10.1145/1275571.1275596`, último acceso: Junio 2021.

[36] R. Nigel Horspool, W. Dale Lyons, and Micaela Serra. Armsim#: A simulator for the arm architecture. Página de acceso: `https://connex.csc.uvic.ca/access/content/group/ARMSim/SIMWeb/index.html`, último acceso: Junio 2021.

[37] Sergio Barrachina Mir, Germán Fabregat Llueca, Juan Carlos Fernández Fernández, and Germán León Navarro. Armsim y qtarmsim: simulador de arm para docencia, 2015.

[38] Pablo Ferreyra, Agustín Laprovitta, Delfina Velez Ibarra, Gonzalo Vodanovic, and Nicolás Wolovick. Legv8, raspberry pi 3 y una vieja fórmula. *VI Workshop de Innovación en Educación en Informática (WIEI). available at: http://sedici.unlp.edu.ar/handle/10915/63921 [accessed: 12/05/2017]*, 2017.

[39] Josiah Jeremiah White and Ryan Meuth. Legv8 runtime simulator, december 2017. Página de acceso: `https://repository.asu.edu/items/45923`, último acceso: Junio 2021.

[40] Boston University. Brisc-v simulator. Página de acceso: `https://ascslab.org/research/briscv/simulator/manual.html`, último acceso: Junio 2021.

[41] Keyhan Vakil. Venus, a risc-v instruction set simulator built for education. Página de acceso: `https://github.com/kvakil/venus`, último acceso: Junio 2021.

[42] D. Patterson and D. Ditzel. The case for the reduced instruction set computer. *SIGARCH Comput. Archit. News*, 8:25–33, 1980.

[43] ARM Developer. Learn the architecture: Aarch64 instruction set architecture. Página de acceso: `https://developer.arm.com/documentation/102374/latest/`, último acceso: Julio 2021.

[44] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[45] Object Management Group. An introduction to omg's unified modeling language (uml). Página de acceso: `https://www.uml.org/what-is-uml.htm`, último acceso: Julio 2021.

[46] M. Fowler. Patterns of enterprise application architecture. 2002.

[47] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, USA, 2003.

[48] Robert Martin. Oo design quality metrics: An analysis of dependencies. Accesible en: `https://linux.ime.usp.br/~joaomm/mac499/arquivos/referencias/oodmetrics.pdf`, último acceso: Julio 2021.

[49] usability.gov. Prototyping. Página de acceso: `https://www.usability.gov/how-to-and-tools/methods/prototyping.html`, último acceso: Julio 2021.

[50] Bertrand Meyer. *Object-Oriented Software Construction (2nd Ed.)*. Prentice-Hall, Inc., USA, 1997.

[51] Oracle. Oracle java. Página principal: `https://www.oracle.com/java/`, último acceso: Julio 2021.

[52] JetBrains. Intellij idea. Página principal: `https://www.jetbrains.com/idea/`, último acceso: Junio 2021.

[53] Gradle Inc. Gradle build tool. Página principal: `https://gradle.org/`, último acceso: Julio 2021.

[54] Terence Parr. Antlrv4. Página principal: `https://www.antlr.org/`, último acceso: Junio 2021.

[55] Oracle Corporation. Javafx. Página de acceso: `https://wiki.openjdk.java.net/display/OpenJFX/Main`, último acceso: Junio 2021.

[56] Oracle Corporation. Packacke javax.swing. Página de acceso: `https://docs.oracle.com/javase/8/docs/api/javax/swing/package-summary.html`, último acceso: Junio 2021.

[57] Oracle Corporation. Package java.awt. Página de acceso: `https://docs.oracle.com/javase/8/docs/api/java/awt/package-summary.html`, último acceso: Junio 2021.

[58] Nhat Minh Lê. Contracts for java. Página principal: `https://github.com/nhatminhle/cofoja`, último acceso: Julio 2021.

[59] Richard Mitchell, Jim McKim, and Bertrand Meyer. *Design by Contract, by Example*. Addison Wesley Longman Publishing Co., Inc., USA, 2001.

[60] Sun Microsystems Inc. *Java Code Conventions*. 2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A, 1997. Accesible en: `https://github.com/nhatminhle/cofoja`, último acceso: Julio 2021.

[61] Uncle Bob. Getting a solid start. Página de acceso: `https://sites.google.com/site/unclebobconsultingllc/getting-a-solid-start`, último acceso: Julio 2021.

[62] SonarSource. sonarlint. Página principal: `https://www.sonarlint.org/`, último acceso: Julio 2021.

[63] Oracle. Java platform, standard edition java development kit version 11 api specification. Página principal: `https://docs.oracle.com/en/java/javase/11/docs/api/index.html`, último acceso: Julio 2021.

[64] arm Developer. Arm architecture reference manual: Armv8, for armv8-a arcgutectyre profile. Página de acceso: `https://developer.arm.com/documentation/ddi0487/latest/`, último acceso: Julio 2021.

[65] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[66] sourceware.org. Using as. Página de acceso: `https://www.sourceware.org/binutils/docs-2.12/as.info/index.html#Top`, último acceso: Julio 2021.

[67] Tomas Mikula and Jordan Martinex. Richtextfx, a rich-text area for javafx. Página de acceso: `https://github.com/FXmisc/RichTextFX`, último acceso: Junio 2021.

[68] Oracle. Javafx javadoc. Página de acceso: `https://openjfx.io/javadoc/11/`, último acceso: Julio 2021.

[69] NVIDIA. Parallel thread execution isa version 7.4. Página de acceso: `https://docs.nvidia.com/cuda/parallel-thread-execution/index.html`, último acceso: Julio 2021.

[70] Jerry Gao, Jacob Tsao, and Ye Wu. Testing and quality assurance for component-based software. 01 2003.

[71] http://scie.lcc.uma.es/. The gii-grin-scie conference raing. Página de acceso: `http://scie.lcc.uma.es/gii-grin-scie-rating/`, último acceso: Julio 2021.

[72] Gnu arm assembler quick reference. Página de acceso: `https://www.ic.unicamp.br/~celio/mc404-2014/docs/gnu-arm-directives.pdf`, último acceso: Julio 2021.