Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

# Mejora y actualización de un simulador didáctico de arquitectura de computadores

*Improvement and update of an educational computer architecture simulator*

Francesco La spina

La Laguna, 24 de mayo de 2024

D. **Iván Castilla Rodríguez**, profesor Contratado Doctor adscrito al Departamento de Ingeniería de Sistemas y Automática de la Universidad de La Laguna, como tutor

**C E R T I F I C A**

Que la presente memoria titulada:

”*Mejora y actualización de un simulador didáctico de arquitectura de computadores*”

ha sido realizada bajo su dirección por D. **Francesco La spina**.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 24 de mayo de 2024

# Agradecimientos

A mi tutor, Iván Castilla, por su ayuda, mentoría, paciencia y su gran dedicación como profesor y tutor.

A Óscar Carrasco, por su mentoría, entusiasmo y gran esfuerzo que me ha contagiado con SIMDE.

A todos los contribuidores de SIMDE, por su dedicación.

A mis padres, por las oportunidades que me han brindado.

A todas las amistades que me he encontrado por el camino, por todos esos buenos momentos juntos.

# Licencia

**Resumen**

*SIMDE es un simulador didáctico de arquitecturas de computadores que nace hace 20 años dentro de esta universidad, el cual desde entonces ha pasado por varios autores y tecnologías diferentes.*

*El principal objetivo de este trabajo ha sido mejorar el código base existente, a través de un mayor marco de testeo y varias refactorizaciones, y la integración de nuevas funcionalidades, como son la generación de métricas para el análisis de las ejecuciones de programas y la introducción de un nuevo modelo de caché de mapeado directo.*

**Palabras clave:** Arquitectura de computadores, Tecnologías Web, Simulación, Mantenimiento de software, Estadísticas, Cache

**Abstract**

*SIMDE is a didactic simulator of computer architectures that was developed 20 years ago within this university. Since then, it has undergone several authorship transitions and employed different technologies.*

*The primary objective of this work has been to enhance the existing code base through the introduction of a more comprehensive testing framework and a series of refactorings, accompanied by the integration of novel functionalities, including the generation of metrics for the analysis of program executions and the introduction of a direct mapping cache model.*

**Keywords:** Computer Architecture, Web Technologies, Simulation, Software Maintenance, Statistics, Cache

# Índice general

# Índice de Figuras

# Índice de Tablas

# Capítulo 1

# Introducción

## 1.1. Introducción

Las actividades docentes en asignaturas relacionadas con el hardware de un computador se enfrentan a desafíos significativos al intentar transmitir conceptos complejos empleando solamente medios expositivos o clases magistrales.

Durante muchos años, el uso de simuladores de parte o la totalidad de diferentes computadores lleva siendo una herramienta fundamental para salvar esta dificultad. Sin embargo, muchas de estas herramientas se van quedando obsoletas o no aprovechan plenamente los recursos actuales, incluyendo las ventajas ofrecidas por aplicaciones basadas en tecnologías web.

En este contexto, el presente Trabajo de Fin de Grado (TFG) tiene como objetivo principal continuar con el desarrollo y mantenimiento de SIMDE [5], un simulador propio de esta universidad y actualmente usado en la asignatura de Arquitecturas de Computadores de tercero del Grado en Ingeniería Informática.

## 1.2. Motivación para el trabajo

Aunque SIMDE es un proyecto recurrente en los trabajos de fin de grado (TFG) y actualmente se utiliza en la docencia, presenta oportunidades para ampliar la mantenibilidad de su código y, como consecuencia, facilitar la implementación de nuevas ideas que han surgido durante los últimos 7 años desde que el proyecto renació como aplicación web.

Por ello, este proyecto es ideal para aplicar, de forma práctica, todas las competencias relacionadas con el mantenimiento del software, a menudo vistas únicamente de manera teórica en la carrera debido al alcance limitado de las prácticas. Además, una vez se tenga una base de código más sólida, este proyecto es ideal para implementar y experimentar con diversas ideas propias del itinerario de Ingeniería de Computadores, como la obtención de métricas de la ejecución de un programa, estrechamente relacionada con el análisis de rendimiento de software (“profiling”), o la implementación de un simulador de cachés sencillas.

Por lo tanto, en este trabajo se buscará utilizar varias metodologías de mantenimiento de software, como la refactorización y el testeo, e implementar nuevas funcionalidades ligadas con el itinerario de Ingeniería de Computadores.

## 1.3. Objetivos y fases

El presente TFG se centrará principalmente en tres áreas principales de mejora.

En primer lugar, interpretar los resultados de las simulaciones es fundamental para obtener conclusiones útiles. Para lograr esto, el simulador no solo debe recopilar y exportar datos, sino también presentarlos de manera clara mediante gráficos amigables para el usuario. En la actualidad, la capacidad de generar resultados visuales es limitada en el simulador. Por lo tanto, se busca incorporar funciones de visualización más robustas.

En segundo lugar, se busca ampliar el alcance del simulador para su utilización en otras asignaturas, como Estructura de Computadores o Principios de Computadores. Para lograr este objetivo, se están desarrollando varias ideas adicionales, incluyendo la integración de un simulador de máquinas secuenciales, diseñado para enseñar los conceptos básicos de una arquitectura Von Neumann, que forma parte de otro TFG actualmente activo. Asimismo, se contempla la incorporación de un simulador de cachés más detallado, que permitirá a los estudiantes experimentar de manera práctica conceptos relacionados con el mapeo y diferentes técnicas de reemplazo.

Por último, el proyecto requiere una profunda refactorización y pruebas exhaustivas, con la implementación de mejores prácticas de desarrollo con el fin de garantizar la mantenibilidad del código a largo plazo.

Con estos objetivos en mente, se puede trazar un plan de trabajo para llevarlos a cabo, proponiendo las siguientes fases:

1. **Refactorización y testeo**: En esta primera fase se realizará un análisis previo del código actual del simulador para identificar áreas problemáticas. Paralelamente, se crearán pruebas para garantizar la integridad del código durante el proceso de refactorización.

2. **Creación de visualizaciones**: En esta fase se estudiará la mejor manera de presentar los datos de las simulaciones y se implementará un sistema de recopilación y visualización de estadísticas.

3. **Testeo**: En esta fase se realizarán pruebas exhaustivas del simulador para identificar posibles errores y problemas, los cuales serán corregidos antes de proceder.

4. **Añadir nuevas funcionalidades**: En esta fase se implementarán otras nuevas funcionalidades planificadas, como la caché.

# Capítulo 2

# Antecedentes

En este capítulo se proporcionará una mayor contextualización al proyecto, dando una breve introducción del los conceptos subyacentes, al propio simulador y ahondando en su historia y en su situación actual.

## 2.1. Paralelismo a nivel de instrucción

El concepto subyacente a SIMDE es el paralelismo a nivel de instrucción, implementando una máquina superescalar y una máquina “Very Long Instruction Word” (VLIW). Uno de los primeros enfoques y más conocidos es la segmentación del cauce (“pipeline”) de instrucciones, que se limita a alcanzar solo un ciclo por instrucción. Sin embargo, al incorporar múltiples unidades que realizan la misma tarea, surge la idea de no solo segmentar la ejecución de la instrucción en múltiples etapas, sino también de emitir múltiples instrucciones al mismo tiempo para su ejecución en paralelo. Debido a que los recursos son limitados y las instrucciones presentan dependencias, se requiere una planificación cuidadosa del lanzamiento de estas instrucciones. De esta forma, las máquinas superescalares realizan una planificación dinámica, reordenando las instrucciones en el tiempo de ejecución. Mientras, las maquinas VLIW realizan una planificación estática, reordenando las instrucciones en tiempo de compilación.

## 2.2. Introducción a SIMDE

En la Figura 2.1 se muestra la pantalla principal del simulador de la máquina superescalar de SIMDE. Para realizar la planificación, utiliza el algoritmo de Tomasulo [27], por tanto, presenta un “Reorder Buffer”(ROB), como se puede observar en el centro debajo. Este contiene principalmente las instrucciones en ejecución, su orden, el registro y el valor que van a escribir. A la derecha de este, se encuentran las estaciones de reserva para los diferentes tipos de instrucciones. Estás, además de contener las instrucciones a la espera de ejecución, contiene los valores de sus operandos o una referencia a la instrucción de los que depende. Finalmente, están las unidades funcionales que llevan a cabo las diferentes etapas del cálculo requerido por cada instrucción. También se muestran las instrucciones en decodificación, prebúsqueda y las tablas de salto y mapeo de registros.

Por otro lado, en la Figura 2.2 se observa la máquina VLIW, donde aún están presentes las unidades funcionales, pero el resto de la pantalla está dedicado a los paquetes de instrucciones que se ejecutarán en cada ciclo.

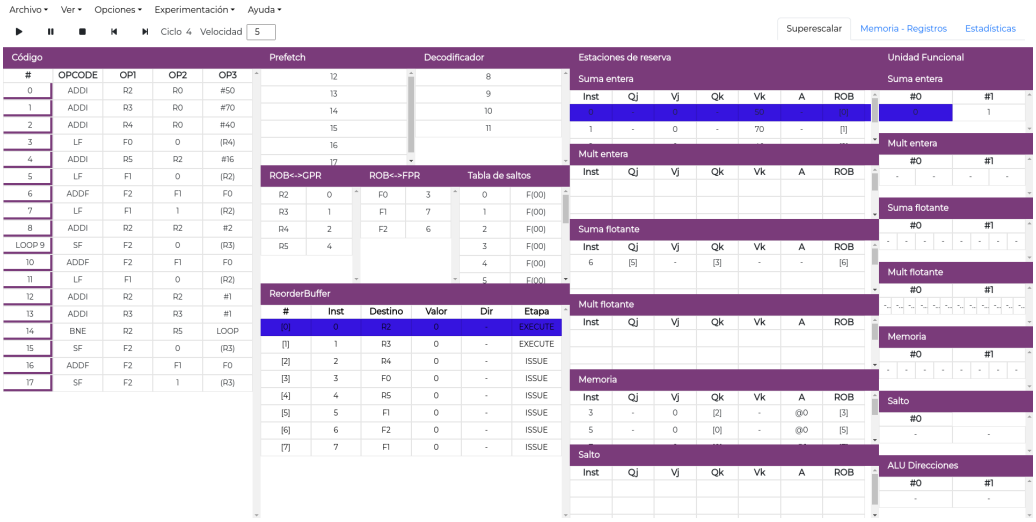



Figura 2.1: Pantalla principal de SIMDE superescalar ejecutando un programa de prueba.

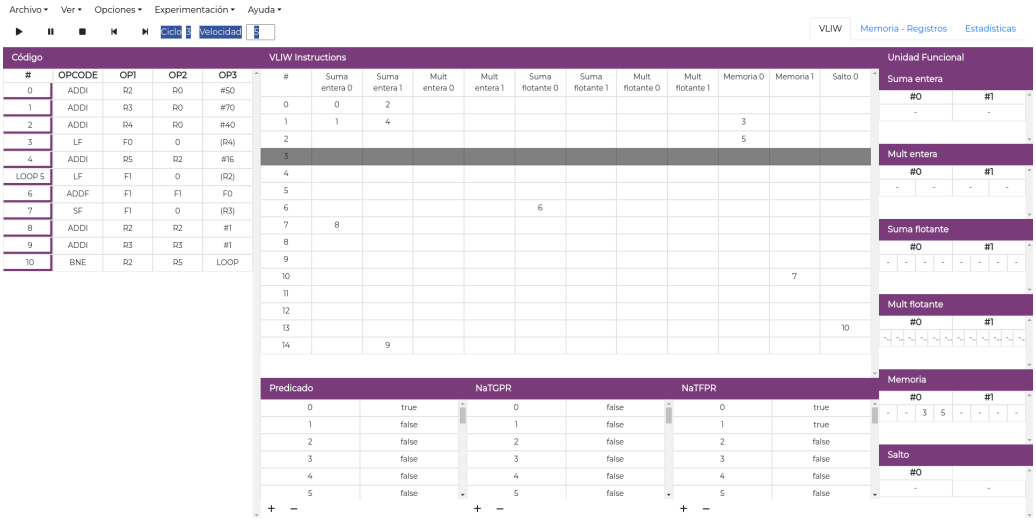



Figura 2.2: Pantalla principal de SIMDE VLIW ejecutando un programa de prueba.

## 2.3. Historia: SIMDE Legacy y SIMDE Web

SIMDE, en su versión Legacy, nace hace casi 20 años dentro de esta universidad con la intención de ser un simulador docente de arquitecturas.

El resultado de este trabajo quedó reflejado en el Proyecto de Fin de Carrera “Un Simulador para el Apoyo Docente en la Enseñanza de las Arquitecturas ILP con Planificación Dinámica y Estática” de Iván Castilla Rodríguez [22] en 2004. Desde entonces, ha sido objeto de diferentes TFG, que lo han ido actualizando y mejorando.

En 2017, se retoma su desarrollo, migrándolo a tecnologías web mediante los trabajos de Adrían Abreu González [1], “Simulador didáctico de arquitectura de computador”, migrando la parte de superescalar, y Melissa Díaz Arteaga [6], “Simulador didáctico de una Arquitectura de planificación estática”, migrando la parte VLIW.

En 2018, se intenta “ludificar” la experiencia del usuario al usar el simulador, a través del TFG “Plataforma de ludificación de un simulador didáctico de arquitectura de computadores” por Antonio Jesús López Garnier [14]. Pero, estas nuevas características nunca lograron integrarse en la versión actual del simulador, lo que evidenció la necesidad de mejorar la mantenibilidad del proyecto.

Finalmente, en 2022, gracias al trabajo de Óscar Carrasco Benítez [3], “Simulador didáctico de arquitectura de computadores: aplicación de metodologías de integración y mantenimiento”, se lograron dar los primeros pasos en la mejora tanto de la usabilidad como de la mantenibilidad de este proyecto.

Así, en el momento de comienzo de este TFG, el proyecto se encontraba en un estado adecuado, pero con amplias oportunidades de mantenimiento, pues el código no escalaba bien, el testeo presentaba carencias y existían oportunidades para obtener un código más comprensible y modificable. De esta forma, cualquier cambio en el código era difícil de realizar y solía resultar en redundancias o bugs desapercibidos. Estas problemáticas se analizarán más en detalle en la sección 5.1.2.

# Capítulo 3

# Estado del arte

Una vez introducido y comentado el estado actual del simulador de este trabajo, es interesante hacer un análisis del resto de simuladores similares que existen. Estos pueden ser una fuente de inspiración para mejorar SIMDE y entender cómo han abordado problemáticas similares. Además, permite justificar la existencia de este proyecto a través de las carencias de los demás simuladores.

En primer lugar, es destacable el repositorio de herramientas titulado “Computer Architecture Educational Tools”, realizado por Israel Koren [13] de la Universidad de Massachusetts Amherst. Aun no siendo exactamente un simulador, disponen de una variedad de herramientas ilustrativas similares a lo perseguido por SIMDE. Sin embargo, estas herramientas han permanecido desde el 2006 sin actualizaciones y se basan en la tecnología de Java Applets[16], la cual ya no está disponible en la actualidad. Aun así, siguen siendo una fuente de inspiración para este proyecto.

Por otro lado, se puede realizar un repaso rápido sobre los simuladores que se han analizado en los trabajos anteriores[1, 6, 22]. Tanto ReSIM, SATSim, VLIW-DLX, VEX y MIDAS han quedado en un estado de abandono, apenas quedando información disponible de estos proyectos. En cambio, de SESC existe un “fork” reciente mantenido por el grupo MASC [12], pero sigue manteniendo las carencias originales de una falta de interfaz gráfica y una alta complejidad.

Por último, se puede hacer una búsqueda de simuladores más actuales.

- RSM [2]: Es un simulador de una máquina superescalar, igual que SIMDE tiene una “Instruction Set Arquitecture”(ISA) propia y busca simplicidad. Este no puede ilustrar conceptos como hace SIMDE, pues no dispone de ninguna clase de interfaz gráfica. Además, lleva sin recibir actualizaciones 2 años. Como ventajas, dispone de una ISA con un lenguaje ensamblador bastante más sencillo para el programador. También dispone de una arquitectura de memoria virtual y multihilo, abordando más conceptos.

- CENOS [10]: También simula un procesador superescalar, pero, implementa menos características que SIMDE, ya que no dispone de interfaz gráfica y lleva sin recibir actualizaciones 2 años también.

Ripes [20] es otro simulador actual que merece un análisis más profundo debido a su alta popularidad, alcanzando las 2300 estrellas en GitHub y teniendo varias presentaciones en numerosas conferencias [18, 19].

Haciendo un primer análisis se puede observar que este presenta un alcance notablemente diferente a SIMDE, pues busca ser un simulador bastante realista de unos

procesadores sencillos, ya que a lo mucho solamente están segmentados, de arquitecturas reales como RISC-V. Por el contrario, el alcance de SIMDE se podría considerar un antónimo de este, pues busca ser un simulador didáctico, ergo sencillo y poco realista, de varias micro arquitecturas de procesadores complejas. De esta forma implementa su propia arquitectura ideal y sencilla. Por lo tanto, Ripes desde la primera pantalla (Figura 3.1) muestra un complejo caminos de datos interconectando una compleja estructura de unidades. Sus otras pantallas son más de lo mismo, muestra una visión de la memoria realista, permite añadir periféricos mapeados en memoria, muestra tanto el código en ensamblador como el código máquina, etc. No hace falta argumentar que SIMDE busca en sus interfaces todo lo contrario, mostrar el detalle lo más abstracto y conciso posible, para ilustrar el concepto que se busca enseñar.

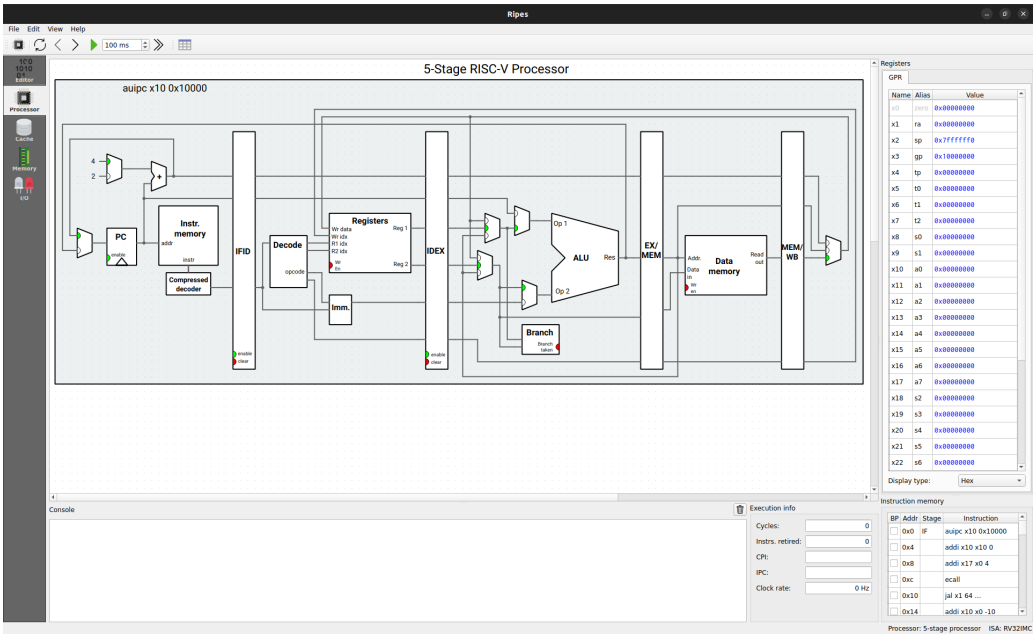



Figura 3.1: Pantalla inicial de Ripes.

Aun con estas diferencias, Ripes puede servir de inspiración para mejorar SIMDE. Aunque aporta una implementación e interfaz compleja de una caché, la forma en la que representa ésta es bastante didáctica y puede servir de inspiración y justificación para implementar un sistema de caché en SIMDE. Además, el hecho de que Ripes utilice una ISA real, plantea una serie de preguntas sobre si SIMDE debería hacer lo mismo. Pues este hecho le ha brindado dos puntos positivos. En primer lugar, le ha permitido tener un lenguaje ensamblador de más alto nivel, pues se puede aprovechar de ensambladores ya existentes, incluso, permitiendo cargar código en C. Esto permite al usuario crear más fácilmente programas para simular, además de visualizar cómo se traduce estos a ensamblador y como se comportan estas traducciones en el procesador. En segundo lugar, justamente gran parte de la popularidad de Ripes se debe a que es un simulador de RISC-V, por lo que queda claro que a las comunidades alrededor de una arquitectura como RISC-V les podría interesar simuladores como SIMDE.

Finalmente, en cuanto a la tecnología utilizada, Ripes está implementado en Qt[25] y C++, por lo que comparte unas tecnologías más similares a las que utilizaba SIMDE Legacy que el actual. Sin embargo, estas no le aportan ninguna ventaja significativa respecto a SIMDE. De hecho, podría argumentarse que presentan ciertas desventajas. Recientemente, Ripes ha lanzado una versión web que aún se encuentra en fase experi-

mental y está limitada por las restricciones de Qt sobre WebAssembly[28]. Al contrario, SIMDE es nativo web, y si en algún momento se considerara lanzar una versión de escritorio, existen herramientas ampliamente adoptadas como Electron[23].

# Capítulo 4

# Herramientas y tecnologías

Está claro que el mundo del desarrollo web se caracteriza por su constante evolución y cambio. Por ello, es importante que el proyecto se adapte a las nuevas tecnologías y herramientas que surgen en el mercado. En este sentido, este capítulo se enfoca en analizar las tecnologías seleccionadas en su momento para el proyecto, justificando su elección y explorando posibles alternativas más contemporáneas.

De esta manera, la mayoría de las tecnologías utilizadas actualmente fueron seleccionadas durante la migración del proyecto a web, en [1]. Si se recupera el análisis realizado en su momento, se puede concluir que fue acertado en gran medida. Por ejemplo, al elegir el lenguaje de lógica de aplicación, se debatió entre CoffeeScript, Dart y TypeScript, decidiéndose finalmente por este último. La justificación inicial, que se basaba en la baja adopción de CoffeeScript y Dart, sigue siendo relevante en la actualidad, donde TypeScript se ha consolidado como el lenguaje dominante en el desarrollo web [23].

## 4.1. Tecnologías descartadas y reemplazadas

Tanto durante el desarrollo de este trabajo, como en etapas previas o simultáneas, se han reemplazado diversas tecnologías que ya no resultaban adecuadas para el proyecto, ya sea debido a su obsolescencia o por la existencia de alternativas más adecuadas.

- AVA: Esta librería servía para ejecutar las pruebas del proyecto. Aunque aún goza de un desarrollo activo, se optó por su reemplazo debido a la disponibilidad de alternativas que se integraban mejor con el sistema de construcción del proyecto.
- Lex: Utilizado como analizador léxico para el código cargado en el simulador, esta librería había sido abandonada desde hace 9 años. Por lo tanto, se decidió sustituirla por una solución más actualizada y mantenida.
- Webpack: Servía para empaquetar el proyecto. Aunque sigue siendo ampliamente utilizada y útil, se utilizaba una versión antigua, y se identificaron alternativas más simples y rápidas que podrían mejorar el flujo de trabajo del proyecto.
- NPM: Aunque sigue siendo el gestor de paquetes más utilizado, su modo de instalación de dependencias a menudo generaba una serie de problemas que podrían evitarse. Para mejorar la eficiencia y la estabilidad del proyecto, se exploraron otras opciones de gestión de paquetes que ofrecieran una mejor experiencia.

## 4.2. Tecnologías que se continuarán usando

Igual que se han descartado o implementado nuevas tecnologías, también se han mantenido una serie de tecnologías en el proyecto, ya sea por ser bastante adecuadas y útiles o por ser difíciles de sustituir.

En primer lugar, queda claro que el proyecto seguirá basándose en TypeScript y React debido a su relevancia y utilidad para el desarrollo. Además, se continuará utilizando Redux en conjunto con estas tecnologías. Sin embargo, cabe destacar que la permanencia de éste no se debe a que sea la opción óptima, sino más bien a la complejidad que conlleva su eliminación. Quitar Redux requeriría una refactorización significativa, lo cual está fuera del alcance del presente trabajo.

Por otro lado, este proyecto cuenta con una serie de herramientas de integración continua, como son Renovate, que gestiona y actualiza automáticamente las dependencias, y GitHub Actions, que automatiza los flujos de trabajo del desarrollo de software, las cuales han demostrado ser altamente beneficiosas y están actualizadas, al haber sido implementadas recientemente en el proyecto [3]. Por ello, se mantendrán en uso debido a su eficacia y relevancia para el desarrollo continuo del proyecto.

## 4.3. Nuevas tecnologías usadas

En general, estas nuevas tecnologías han sido seleccionadas para sustituir a las herramientas previamente descartadas. Con la excepción de Vitest y Apache Echarts, todas han sido integradas en el proyecto antes del desarrollo de este TFG. Por lo tanto, a continuación solo se presenta un breve comentario sobre cada una de ellas.

- Vite: Es una herramienta de construcción de proyectos web que se caracteriza por su rapidez y sencillez. Se ha empleado para reemplazar a Webpack, y su integración se realizó en paralelo a la elaboración de este TFG.
- Ts-parsec: Es una librería de análisis léxico y sintáctico de gran potencia. Ha sido utilizada para reemplazar a Lex y reescribir enteramente los *Parsers* del proyecto.
- Pnpm: Es un gestor de paquetes que solucionar ciertas problemáticas presentes en NPM y Yarn, entre ellas su eficiencia en el uso de espacio en disco.
- Vitest: Desarrollada para integrarse con Vite, esta herramienta de “testing” ha reemplazado a AVA en el proyecto.
- Apache Echarts: Es una librería de generación de gráficas que se ha integrado en el proyecto para implementar la nueva funcionalidad de visualización de estadísticas.

# Capítulo 5

# Desarrollo del proyecto

En este capítulo se explicará el proceso por el cual se ha desarrollado las diferentes fases del proyecto.

## 5.1. Mantenimiento del codigo

En esta sección se engloban las diferentes tareas realizadas en pos de obtener un código base más mantenible y que presente una mayor facilidad de implementar nuevas características sobre este. De esta forma, en este apartado se presenta la ampliación y refactorización del marco de pruebas y la refactorización general sobre el núcleo del simulador que se realizó.

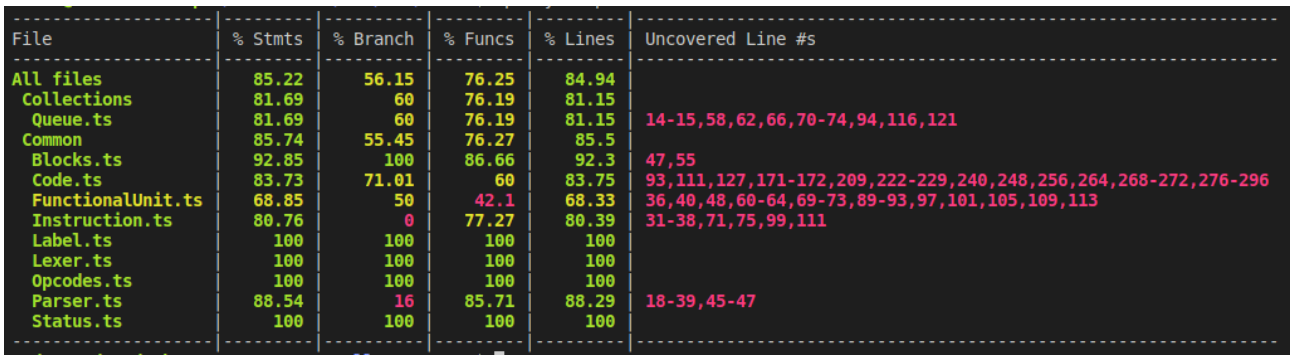
### 5.1.1. Refactorización de pruebas

Para comenzar este apartado, cabe realizar un diagnóstico del estado en el que se encontraban las pruebas en el proyecto al comienzo de este trabajo, el cual puede considerarse desfavorable.

En primer lugar, las pruebas, realizadas con el framework AVA, estaban desconfiguradas, lo que resultaba en la ejecución únicamente de los tests presentes en la carpeta raíz, ignorando los tests de las subcarpetas. Esto significaba que solo se ejecutaban 25 de las 40 pruebas implementadas en ese momento. Además, la estructuración de las carpetas no era óptima, ya que no se distinguían claramente los tests funcionales de los unitarios, y no seguían una estructura lógica basada en la propia organización del proyecto. Así, por ejemplo, dentro de la carpeta *VLIW* se encontraba el test *VLIWParser*, pero también se encontraba duplicado en la subcarpeta *Parser*.

Todo esto afectaba las métricas de cobertura del código, ya que solo se consideraba un conjunto limitado de archivos fuente (Figura 5.1). Además, el estado de algunas de las pruebas no las hacía aptas para conocer si el código se encontraba funcionando correctamente, pues no se verificaban los resultados obtenidos, o se hacía de manera errónea. Ejemplificando, la totalidad de las pruebas funcionales de la máquina VLIW ejecutaban programas de testeo originarios de SIMDE Legacy sin cargar los datos necesarios ni verificar los resultados obtenidos, limitándose a confirmar que la máquina podía cargar y ejecutar los programas.

Para abordar estas deficiencias, tras reestructurar las carpetas de los tests, el primer punto abordado fue la creación de un conjunto de pruebas funcionales para las máquinas. Aunque ya existía parcialmente un conjunto de pruebas de SIMDE Legacy, nunca se



| File | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s |
|---|---|---|---|---|---|
| All files | 85.22 | 56.15 | 76.25 | 84.94 | |
| Collections | 81.69 | 60 | 76.19 | 81.15 | |
| Queue.ts | 81.69 | 60 | 76.19 | 81.15 | 14-15,58,62,66,70-74,94,116,121 |
| Common | 85.74 | 55.45 | 76.27 | 85.5 | |
| Blocks.ts | 92.85 | 100 | 86.66 | 92.3 | 47,55 |
| Code.ts | 83.73 | 71.01 | 60 | 83.75 | 93,111,127,171-172,209,222-229,240,248,256,264,268-272,276-296 |
| FunctionalUnit.ts | 68.85 | 50 | 42.1 | 68.33 | 36,40,48,60-64,69-73,89-93,97,101,105,109,113 |
| Instruction.ts | 80.76 | 0 | 77.27 | 80.39 | 31-38,71,75,99,111 |
| Label.ts | 100 | 100 | 100 | 100 | |
| Lexer.ts | 100 | 100 | 100 | 100 | |
| Opcodes.ts | 100 | 100 | 100 | 100 | |
| Parser.ts | 88.54 | 16 | 85.71 | 88.29 | 18-39,45-47 |
| Status.ts | 100 | 100 | 100 | 100 | |

Figura 5.1: Tabla que muestra la cobertura de código inicial del proyecto, en donde se observa pocos ficheros testeados.

había utilizado de manera completa y correcta en el proyecto. Por lo tanto, tras llevar este conjunto a las subcarpeta *code*, se crearon datos de prueba para cada uno de los programas de testeo y se diseñó una serie de verificaciones sobre los resultados obtenidos. Esta tarea fue laboriosa, ya que entre las verificaciones se incluía la validación del número de ciclos de ejecución, lo que requería asegurar la correcta ejecución del programa de testeo en cada ciclo.

Con todo esto se procedió a la creación del código de prueba para los tests funcionales. Cada test sigue un esquema similar:

1. Inicializa la máquina.
2. Carga el código y los datos iniciales.
3. Ejecuta la máquina.
4. Realiza varias comprobaciones, como verificar que el programa llegue a la última instrucción, que el resultado obtenido sea el esperado y que el número de ciclos de ejecución sea correcto.

Además, se implementó un test extendido llamado *iteratelist*, que realiza comprobaciones adicionales durante la ejecución del código, verificando el estado de la máquina en ciclos clave. Para ello, la prueba verifica el contenido de las unidades funcionales, decodificador, unidad de prebúsqueda y la tabla de predicción de saltos.

Aun con todos estos tests funcionales, se propusieron 4 nuevos tests específicos para ciertas partes del funcionamiento de la máquina, para así evitar la introducción de bugs inadvertidos, creándose las siguientes pruebas:

1. Un test que ejecuta varias instrucciones destinadas a unidades funcionales distintas y mide los ciclos empleados. Con esto, se verifica que tanto la latencia de la unidad funcional asociada a cada instrucción y el resto de etapas es correcta.
2. Un test que intenta modificar el registro R0 y acceder a su valor. Este registro es el único que no se puede modificar, por lo que este test pretende verificar que esta restricción se cumple.
3. Un test que ejecuta varias instrucciones de almacenamiento y luego carga de datos en diferentes localizaciones (registros generales, flotantes y memoria) y verifica que se cargan los últimos datos almacenados. De esta forma, se verifica que no se producen problemas de dependencias de datos.

4. Un test que ejecuta instrucciones de manera especulativa que intentan modificar el estado de la máquina y luego se comprueba que este no se ha modificado, pues esa ejecución especulativa es descartada.

La implementación de estos tests resultó efectiva para detectar y corregir varios errores en el código de la máquina, como la posibilidad de mutar el registro R0 y problemas de latencia en las instrucciones de “Load” y “Store”.

En cuanto a los tests unitarios, se encontraban en un estado aceptable y solo requerían modificaciones mínimas. Las principales modificaciones se centraron en las pruebas de los *Parser*, que fueron reescritas y mejoradas para incluir pruebas adicionales de procesamiento de datos. Por ejemplo:

- Para las pruebas del *Parser* de contenido, se ha procedido a comprobar que se procesan correctamente los números hexadecimales y flotantes. También, que efectivamente se comprueben los límites de los registros y memoria y no se permita escribir en la posición de memoria 1025 o superior.

- Para las pruebas del *Parser* de instrucciones, también se ha procedido a comprobar los límites de registros y memoria. Además, se ha añadido un test que comprueba que no se permiten valores extraños en el inmediato de las instrucciones.

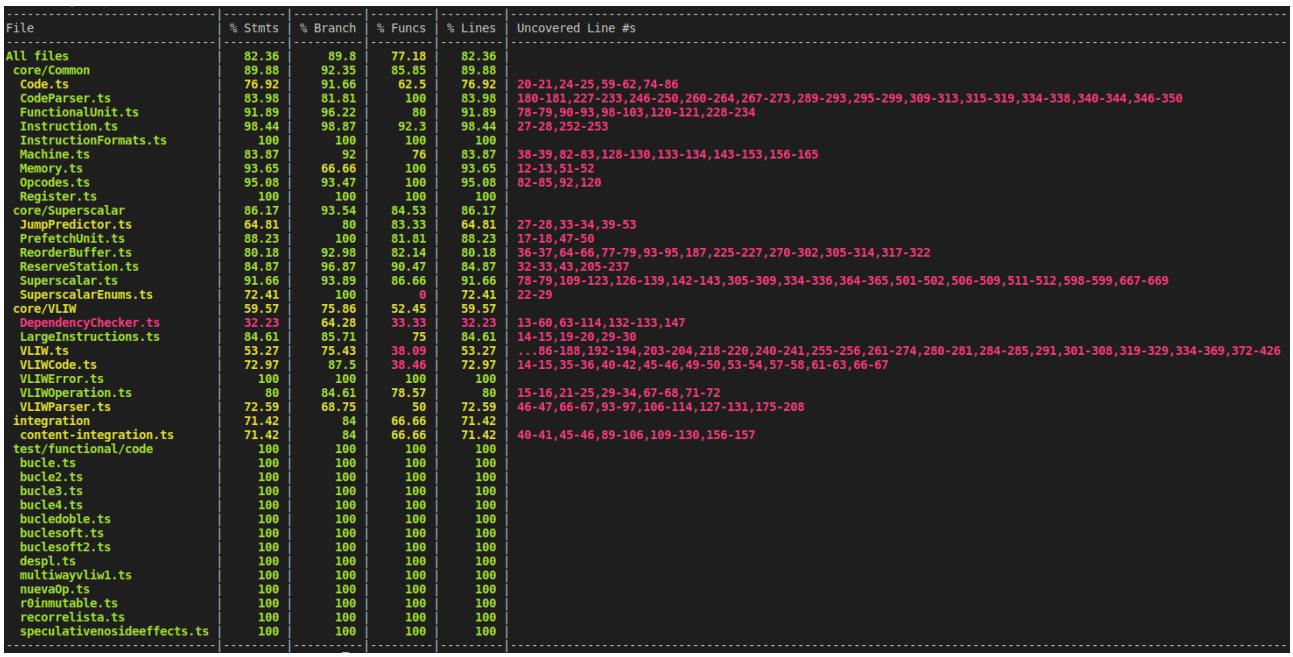
Finalmente, todos los tests fueron reescritos utilizando la librería Vitest en lugar del framework AVA, ya que el proyecto en ese momento se estaba migrando hacia Vite y Vitest ofrecía una mejor integración. Aunque la migración se realizó rápidamente debido a la similitud entre las “Application Programming Interface” (API) de ambas librerías, no se aprovecharon todas las ventajas de Vitest ni se escribieron los tests de manera idiomática para esta librería.

Realizando nuevamente un análisis de cobertura de código (Figura 5.2), se observó una mejora en los resultados, con la mayor parte del código ahora testeado. La excepción fue la máquina VLIW, pues contiene funciones sin usar y, por lo tanto, no testeadas.

### 5.1.2. Refactorización general

A medida que se iba desarrollando el proyecto, se hizo evidente que el código presentaba bastantes oportunidades de mejora. Por ejemplo, existía código duplicado, acoplado y difícil de comprender, clases poco contenidas, con visibilidad total de sus atributos y responsabilidades poco definidas, y una estructura de carpetas desordenada. De facto, los problemas relacionados con el uso incorrecto del paradigma de programación orientada a objetos surgen, en parte, de la adaptación del código original de SIMDE Legacy. Este código utilizaba intensivamente estructuras en C y solo contaba con unas pocas clases básicas. La conversión realizada fue bastante simple, transformando estructuras en clases sin métodos y produciendo clases que no aprovechan adecuadamente las características del lenguaje. De hecho, citando el trabajo de Abreu [1]: “Por comodidad durante el desarrollo, se convirtieron las estructuras de C++ en clases, de tal forma que la gestión de las mismas fuera más sencilla”.

De esta forma, para abordar estas oportunidades de mejora, se realizó una refactorización dividida principalmente en dos fases. En esta primera fase se llevaron a cabo un análisis del código en busca de partes innecesarias o altamente acopladas para reducir su complejidad y mejorar su mantenibilidad. Para una mejor comprensión, es recomendable



| File | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s |
|---|---|---|---|---|---|
| All files | 82.36 | 89.8 | 77.18 | 82.36 | |
| core/Common | 89.88 | 92.35 | 85.85 | 89.88 | |
| Code.ts | 76.92 | 91.66 | 62.5 | 76.92 | 20-21,24-25,59-62,74-86 |
| CodeParser.ts | 83.98 | 81.81 | 100 | 83.98 | 180-181,227-233,246-250,260-264,267-273,289-293,295-299,309-313,315-319,334-338,340-344,346-350 |
| FunctionalUnit.ts | 91.89 | 96.22 | 80 | 91.89 | 78-79,90-93,98-103,120-121,228-234 |
| Instruction.ts | 98.44 | 98.87 | 92.3 | 98.44 | 27-28,252-253 |
| InstructionFormats.ts | 100 | 100 | 100 | 100 | |
| Machine.ts | 83.87 | 92 | 76 | 83.87 | 38-39,82-83,128-130,133-134,143-153,156-165 |
| Memory.ts | 93.65 | 66.66 | 100 | 93.65 | 12-13,51-52 |
| Opcodes.ts | 95.08 | 93.47 | 100 | 95.08 | 82-85,92,120 |
| Register.ts | 100 | 100 | 100 | 100 | |
| core/Superscalar | 86.17 | 93.54 | 84.53 | 86.17 | |
| JumpPredictor.ts | 64.81 | 80 | 83.33 | 64.81 | 27-28,33-34,39-53 |
| PrefetchUnit.ts | 88.23 | 100 | 81.81 | 88.23 | 17-18,47-50 |
| ReorderBuffer.ts | 80.18 | 92.98 | 82.14 | 80.18 | 36-37,64-66,77-79,93-95,187,225-227,270-302,305-314,317-322 |
| ReserveStation.ts | 84.87 | 96.87 | 90.47 | 84.87 | 32-33,43,205-237 |
| Superscalar.ts | 91.66 | 93.89 | 86.66 | 91.66 | 78-79,109-123,126-139,142-143,305-309,334-336,364-365,501-502,506-509,511-512,598-599,667-669 |
| SuperscalarEnums.ts | 72.41 | 100 | 0 | 72.41 | 22-29 |
| core/VLIW | 59.57 | 75.86 | 52.45 | 59.57 | |
| DependencyChecker.ts | 32.23 | 64.28 | 33.33 | 32.23 | 13-60,63-114,132-133,147 |
| LargeInstructions.ts | 84.61 | 85.71 | 75 | 84.61 | 14-15,19-20,29-30 |
| VLIW.ts | 53.27 | 75.43 | 38.09 | 53.27 | ...86-188,192-194,203-204,218-220,240-241,255-256,261-274,280-281,284-285,291,301-308,319-329,334-369,372-426 |
| VLIWCode.ts | 72.97 | 87.5 | 38.46 | 72.97 | 14-15,35-36,40-42,45-46,49-50,53-54,57-58,61-63,66-67 |
| VLIWError.ts | 100 | 100 | 100 | 100 | |
| VLIWOperation.ts | 80 | 84.61 | 78.57 | 80 | 15-16,21-25,29-34,67-68,71-72 |
| VLIWParser.ts | 72.59 | 68.75 | 50 | 72.59 | 46-47,66-67,93-97,106-114,127-131,175-208 |
| integration | 71.42 | 84 | 66.66 | 71.42 | |
| content-integration.ts | 71.42 | 84 | 66.66 | 71.42 | 40-41,45-46,89-106,109-130,156-157 |
| test/functional/code | 100 | 100 | 100 | 100 | |
| bucle.ts | 100 | 100 | 100 | 100 | |
| bucle2.ts | 100 | 100 | 100 | 100 | |
| bucle3.ts | 100 | 100 | 100 | 100 | |
| bucle4.ts | 100 | 100 | 100 | 100 | |
| bucledoble.ts | 100 | 100 | 100 | 100 | |
| buclesoft.ts | 100 | 100 | 100 | 100 | |
| buclesoft2.ts | 100 | 100 | 100 | 100 | |
| despl.ts | 100 | 100 | 100 | 100 | |
| multiwayvliw1.ts | 100 | 100 | 100 | 100 | |
| nuevaOp.ts | 100 | 100 | 100 | 100 | |
| r0inmutable.ts | 100 | 100 | 100 | 100 | |
| recorrelista.ts | 100 | 100 | 100 | 100 | |
| speculativenosideeffects.ts | 100 | 100 | 100 | 100 | |

Figura 5.2: Tabla que muestra la cobertura de código tras implementar las nuevas pruebas.

revisar la Figura 5.3, que muestra un esquema simplificado de clases antes de la refactorización principal. En la segunda fase, se analizaron y corrigieron otras problemáticas detectadas en la primera fase.

En primer lugar, destaca la clase *Queue*, que no aporta funcionalidad adicional a la que proporcionan los métodos de un *Array*. Después de una significativa refactorización en las áreas donde se usaba esta clase, se evidenció su prescindibilidad, lo que llevó a su eliminación como parte de este desarrollo. A continuación, se encuentran las clases *Label* y *BasicBlock*, utilizadas para almacenar etiquetas y bloques básicos del código durante el análisis. Fueron eliminadas debido a que no se empleaban en ninguna parte del proyecto, ya que existían estructuras de datos alternativas más efectivas. Además, estas clases carecían de documentación y su complejidad, con el uso de una doble lista enlazada, las volvía poco adecuadas para su mantenimiento.

Por otra parte, al estudiar la lógica de la máquina superescalar, destaca que estaba fuertemente acoplada en un solo archivo, lo que dificultaba cualquier cambio o mejora sin introducir errores. Así, se procedió a primero desacoplar la lógica del “Reorder Buffer” (ROB) a su propia clase, luego la de las estaciones de reserva, la del decodificador, la unidad de predicción de saltos y la unidad de prebúsqueda. De esta forma, en la clase superescalar solo quedó la lógica que conecta estas clases. Esta modularización siguió los principios de la programación orientada a objetos, aislando las responsabilidades de cada clase y reduciendo la visibilidad de sus atributos para garantizar su cohesión. Sin embargo, con esta acción la interfaz de la máquina superescalar dejó de funcionar, pues ya no se podía acceder directamente a los datos necesarios, requiriendo la adición de métodos a las nuevas clases para mantener la funcionalidad previa. Esta adaptación permitió una separación más clara entre los datos y su representación, mejorando la visualización de las estaciones de reserva y el ROB, como se muestra en las Figuras 5.4 y 5.5. En última instancia, estos cambios condujeron a una máquina superescalar más comprensible, mantenible y con una mejor interfaz.

A pesar de que las unidades funcionales ya estaban en su propia clase, tras analizarla,

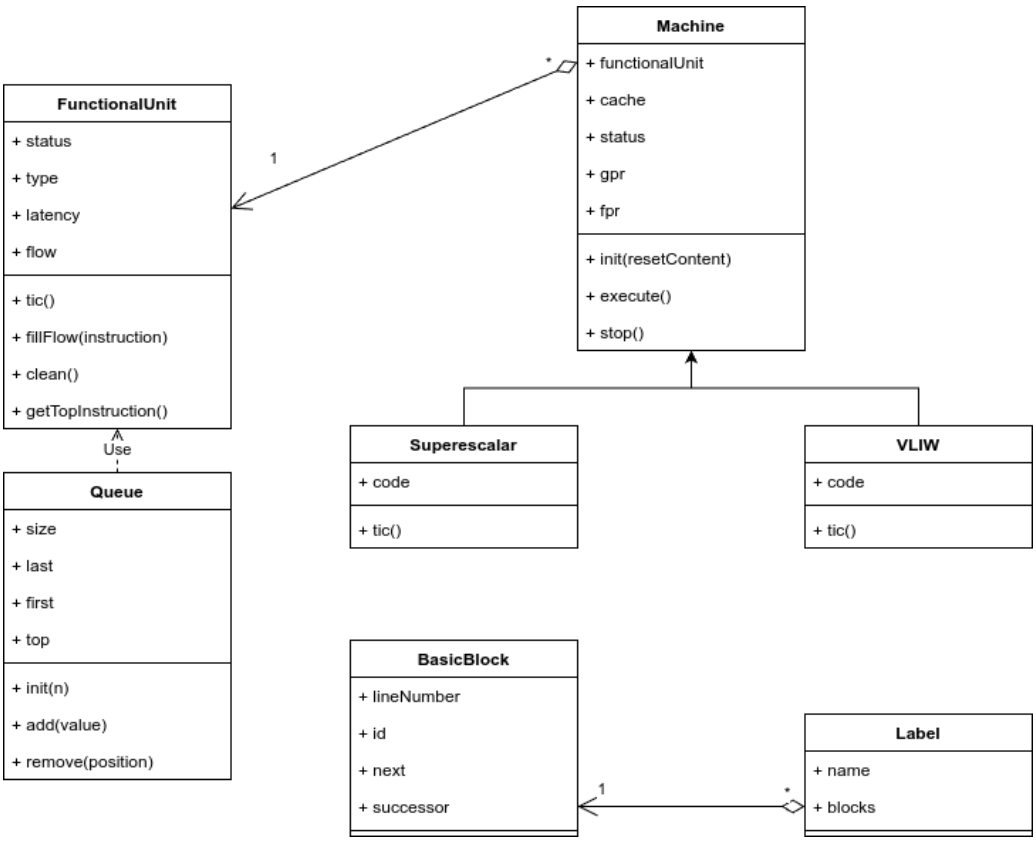



Figura 5.3: Diagrama de clases del núcleo simplificado antes de la refactorización.

fue evidente las diferencias significativas en su implementación en comparación con las nuevas clases creadas, pues no presentaba visibilidad en sus atributos y delegaba ciertas responsabilidades a la clase *Superscalar*, como la ejecución en sí de la instrucción y no estaba documentada. Por lo tanto, se reconsideró su implementación, redefiniendo los métodos necesarios, como la inserción y ejecución de instrucciones, avanzar un ciclo, y los necesarios para realizar comprobaciones sobre esta clase. Internamente, para almacenar las instrucciones, se utiliza un *Array*, que contiene las instrucciones en orden de llegada. Además, junto a la instrucción, se almacena un contador que indica el número de ciclos vacíos por delante de esta, de esta manera, no es necesario insertar nulos en el *Array* en ciclos donde no se inserta una instrucción. Además, igual que en las otras clases, se agregaron métodos para obtener la representación de esta unidad.

En la Figura 5.6, se representa el anterior esquema de clases después de este trabajo de refactorización.

En esta segunda fase, a pesar de las anteriores mejoras, seguían persistiendo tres problemas principales en el código:

1. Se realizaban múltiples comprobaciones del tipo de una instrucción en partes dispersas del código de las máquinas y unidades, a través del “opcode” de la instrucción, lo que generaba una necesidad de modificar estas comprobaciones al añadir nuevas instrucciones, aumentando el riesgo de errores. Por ejemplo, se comprobaba si era un salto, si tenía un registro de destino, si sus operandos eran flotantes, etc. Por ello se centralizó toda esta lógica en la clase *Instruction*, a través de métodos booleanos que permiten consultar el tipo de instrucción.

2. Se carecía de un identificador único para cada instancia de instrucción en ejecución,

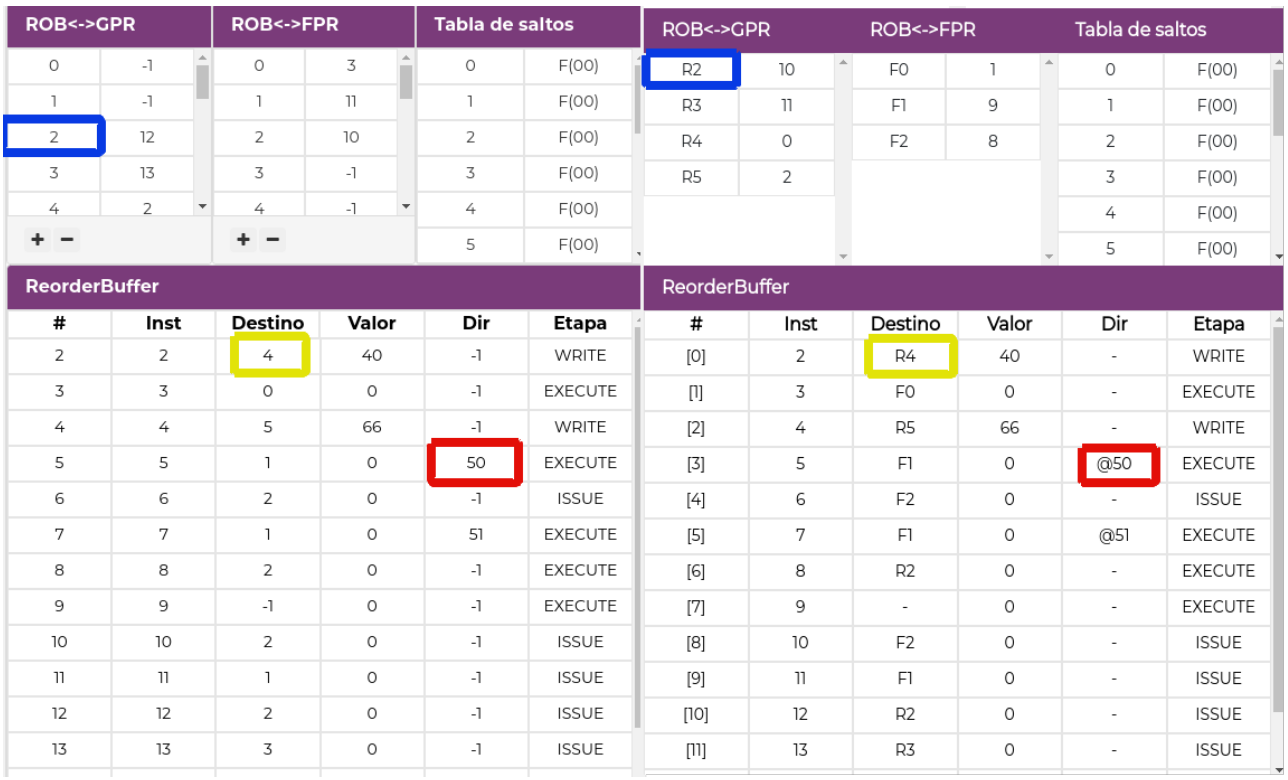



Figura 5.4: A la izquierda, interfaz anterior del ROB. Mientras, a la derecha, la nueva interfaz.

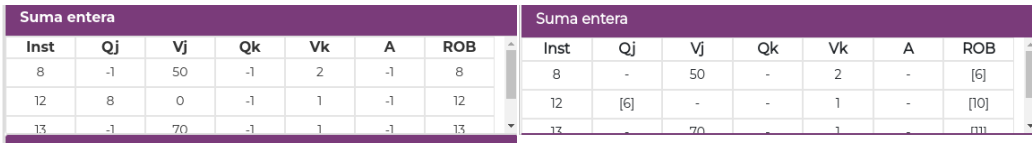



Figura 5.5: A la izquierda, interfaz anterior de las estaciones de reserva. Mientras, a la derecha, la nueva interfaz.

lo que llevaba a que se usasen referencias relativas a las entradas del ROB, estaciones de reserva o unidades funcionales. En agravante, estas referencias cambiaban al extraer instrucciones de estas estructuras. Para evitar estas complicaciones, se creó un identificador único para cada instancia, facilitando su seguimiento en todas las unidades. Al realizar esto, la implementación del sistema de coloreado, que permite colorear las celdas en donde se encuentra una instrucción para seguirla más fácilmente (Figura 5.7), se pudo realizar de una manera más sencilla, ya que antes dependía de actualizar el color en cada lugar donde se encontraba la instancia de la instrucción. Ahora, el coloreado se realiza directamente en la interfaz, asignando en el estado global a cada identificador único el color deseado y rescatando este color en cada elemento visual.

3. Se identificaron problemas en la gestión de los valores de configuración de las máquinas, que estaban dispersos en varias clases. Estos son tanto constantes como modificables por el usuario a través de la interfaz. Se planteó la posibilidad de utilizar Redux para centralizar estos valores y facilitar su acceso y modificación, aunque esto implicaría un cambio en el patrón de diseño existente del núcleo. Actualmente, el núcleo no está diseñado para acceder directamente al estado global, sino que el

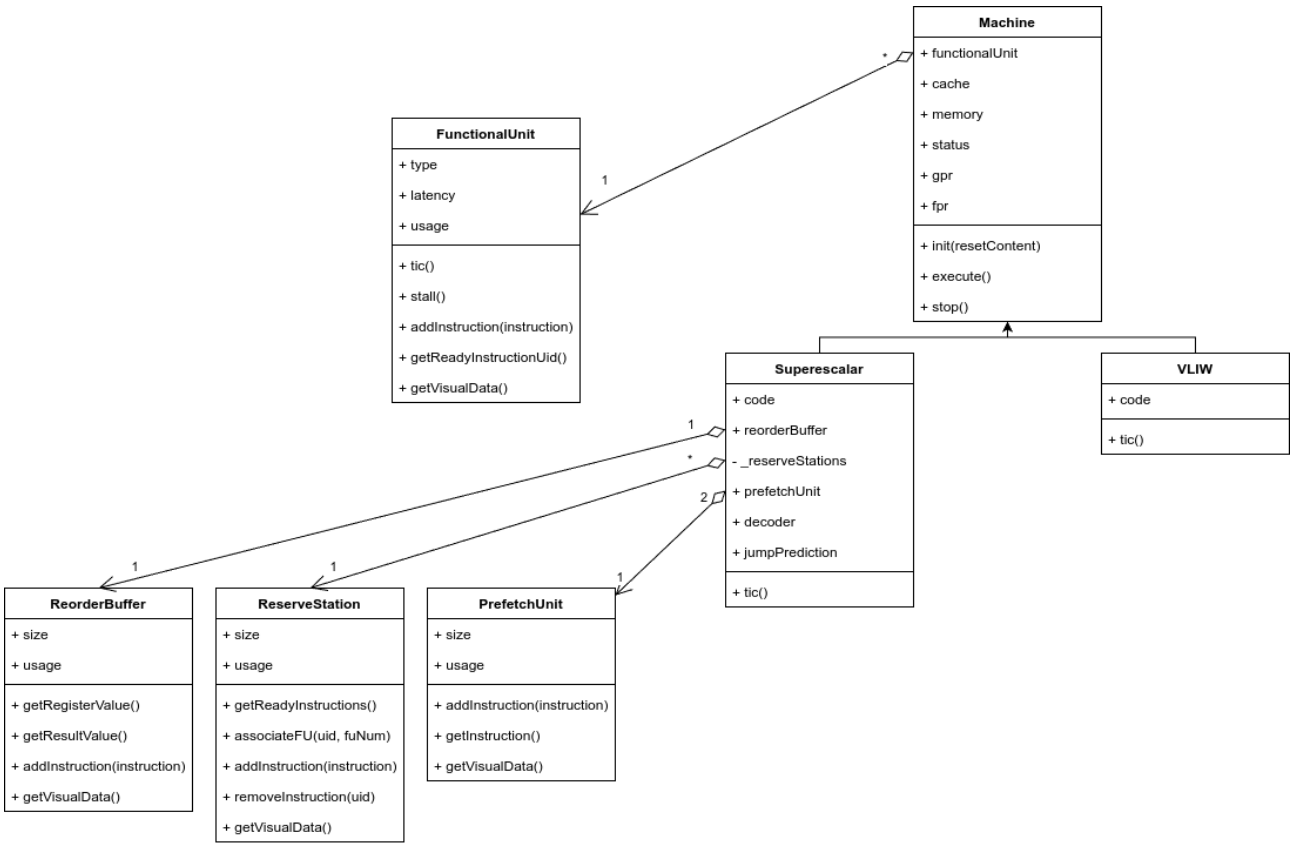



Figura 5.6: Diagrama de clases del núcleo simplificado después de la refactorización.

integrador de las máquinas se encarga de ello. Así, convendría hacer un análisis pormenorizado de cada valor, para identificar qué alternativa conviene seguir. En la Tabla 5.1 se realiza este análisis, quedando patente que no hay un problema de dependencia para propagar el cambio de los valores modificables. Por lo tanto, se decidió mejorar y estandarizar la implementación actual de la gestión de estos valores en lugar de adoptar Redux.

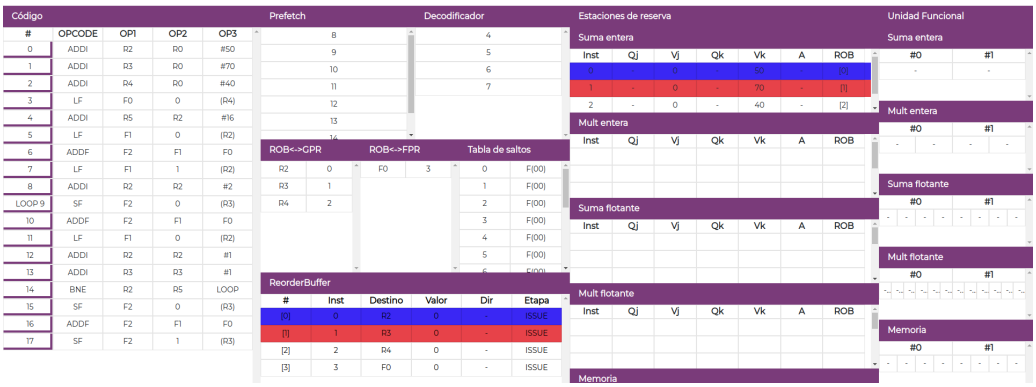



Figura 5.7: Ejecución de ejemplo de SIMDE superescalar con las instrucciones 0 y 1 coloreadas de azul y rojo.

Tabla 5.1: Análisis pormenorizado de los valores de configuración de las máquinas.

| **Nombre** | **Función** | **¿Constante?** | **Utilizado en** | **¿Se puede hacer modificable?** |
|---|---|---|---|---|
| PREDTABLESIZE | Tamaño de la tabla de saltos | Sí | Superescalar | Sí, como un parámetro de construcción de la clase |
| issue | Capacidad del decodificador y unidad de prebúsqueda | No | Superescalar | Sí, propagando el cambio donde es necesario |
| NPR | Tamaño de la tabla de predicado | Sí | VLIW | Sí, aunque podría presentar problemas de propagación con VLIWParser |
| memoryFailLatency | Latencia de fallo de caché | No | Machine (Superescalar y VLIW) | Sí, propagando el cambio donde es necesario |
| MEMORY_SIZE | Tamaño de memoria | Sí | Machine (Superescalar y VLIW). Code. Interfaz. | No, presenta problemas de propagación con la interfaz y clase Code |
| NGP | Tamaño registros generales | Sí | Machine (Superescalar y VLIW). Code. Interfaz. | No, presenta problemas de propagación con la interfaz y clase Code |
| NFP | Tamaño registros flotantes | Sí | Machine (Superescalar y VLIW). Code. Interfaz. | No, presenta problemas de propagación con la interfaz y clase Code |
| functionalUnitLantencies | Latencias de las unidades funcionales | No | Machine (Superescalar y VLIW). DependencyChecker. | Sí, propagando el cambio donde es necesario |
| functionalUnitNumbers | Número de unidades funcionales | No | VLIW. | Sí, aunque podría presentar problemas de propagación con VLIWParser |

## 5.2. Implementación de nuevas funcionalidades

Una vez se dispuso de un código más sólido, gracias a las pruebas y refactorizaciones de la anterior sección, fue el momento idóneo para ampliar las funcionalidades del proyecto.

### 5.2.1. Recolección y visualización de estadísticas

Desde hace tiempo se ha considerado la integración de herramientas de recolección y visualización de estadísticas, una propuesta que ya se mencionaba en la memoria original de SIMDE Legacy: “Incluir herramientas de verificación y de estadísticas que permitan comparar varias simulaciones.” [22]. Recoger y mostrar estas métricas jugaría un papel fundamental en el carácter educativo del proyecto, permitiendo al usuario comprender analíticamente el funcionamiento del programa en ejecución. Con esta nueva visión analítica, podría proceder a experimentar tanto modificando el programa como la máquina, por ejemplo en busca de un objetivo como puede ser la optimización del rendimiento, y consecuentemente afianzando los conocimientos bases de la arquitectura de computadores de las máquinas de SIMDE. De una forma más esquematizada, la Figura 5.8 ilustra los nuevos casos de uso que posibilitarían la recolección y visualización de estadísticas.

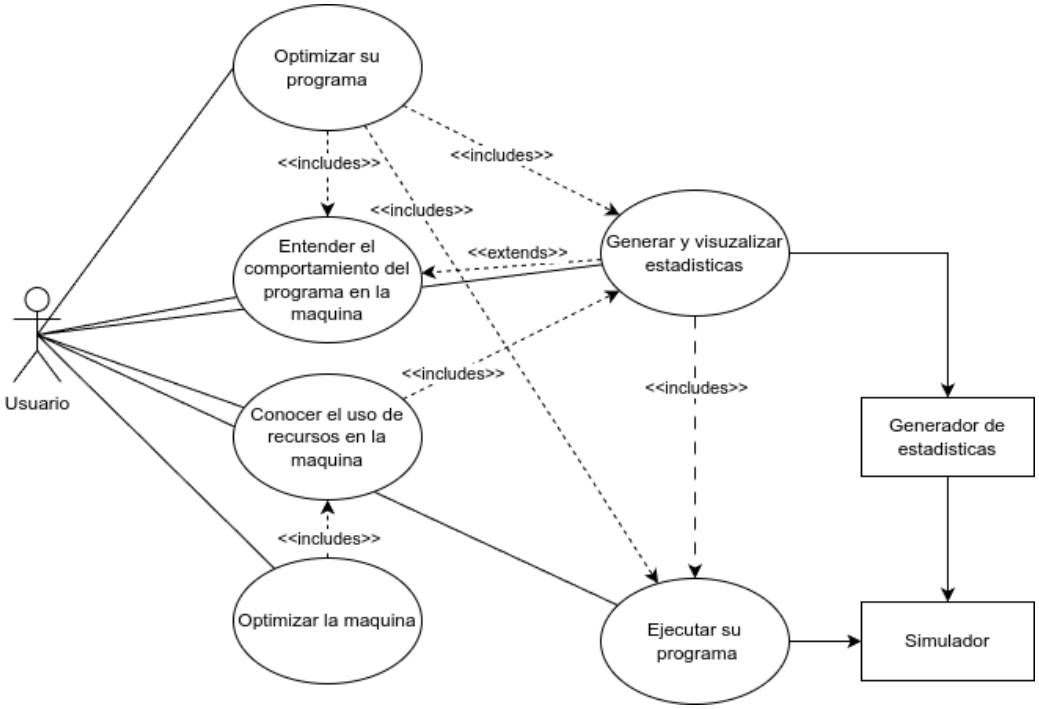



Figura 5.8: Casos de uso relacionados con la recolección y visualización de estadísticas.

Para comenzar, es importante considerar qué métricas proporcionarían información útil al usuario sin generar confusión. Tener demasiadas métricas puede sobrecargar y confundir al usuario, mientras que tener muy pocas podría no ofrecer una comprensión completa del funcionamiento del programa y la máquina.

- Tanto la máquina superescalar como la VLIW tienen una serie de unidades(por ejemplo, unidades funcionales), que, en determinados casos, pueden llegar a llenarse, por lo que puede llegar a ser interesante recoger estadísticas sobre estas unidades, como por ejemplo el **porcentaje de uso en cada ciclo**.

- Se pueden considerar métricas que sigan el flujo de ejecución de cada instrucción, lo que proporcionaría una percepción más clara de lo que está sucediendo en la máquina.

  En primer lugar, puede existir una métrica sobre **cuantas instrucciones hay en cada etapa**(ejecución, decodificación, etc.) de cada ciclo. Este dato tiene una fuerte relación con el anterior enfoque, pues normalmente cada etapa se relaciona con una unidad de la máquina. Igualmente, no es redundante, ya que no todas las etapas están vinculadas a una unidad específica, y los datos se expresan de manera absoluta en lugar de como un porcentaje.

  Siguiendo con este enfoque, se puede plantear una estadística sencilla que cuente el **número total de instrucciones que han realizado "commit" frente a descartadas**.

  Además, otra métrica de interés sería **el recuento total de fallos de predicción**. Sin embargo, resulta más valioso comprender el patrón de estos fallos, es decir, cuándo ocurren. Ambos datos son fácilmente derivables de otras métricas, por lo que se han omitido para evitar redundancias.

- Se pueden plantear métricas que sigan el cauce de ejecución de cada instancia de instrucción. De esta forma se podría mostrar **el número de ciclos que se pasa cada instancia de instrucción en cada etapa**. Dado que los programas suelen instanciar una gran cantidad de instrucciones, mostrar todas estas métricas podría resultar engorroso. Por lo tanto, es más práctico representarlas utilizando medidas estadísticas como la media, mediana, moda y desviación.

Una vez recopilados todos estos datos, es imprescindible contar con una manera clara de presentarlos al usuario. En este apartado, se discutirán las gráficas que se utilizarán para cada métrica.

- Para las estadísticas de **uso de cada unidad**, resulta apropiado emplear un gráfico de líneas (Figura 5.9). Este gráfico presenta una línea para cada unidad, con el eje horizontal representando el porcentaje y el eje vertical mostrando los ciclos. De forma predeterminada, se ocultan todas las unidades menos la tres principales por la que pasa cada instrucción (unidad de prebúsqueda, decodificador y ROB). Esto es debido a que hay una gran cantidad de unidades que generarían un gráfico difícil de leer, por lo que se deja al usuario la opción de mostrar las unidades que necesite.

- Para representar **el número de instrucciones en cada etapa**, un gráfico de barras puede resultar más efectivo, específicamente uno apilado, pues de esta forma se puede observar visualmente el total de instrucciones que se encuentran en el flujo de ejecución en cada ciclo (Figura 5.10). En este gráfico, el eje horizontal representa el ciclo, mientras que el eje vertical muestra el número de instrucciones. Por otra parte, hay 6 "pilas", o más bien categorías: Instrucciones en prebúsqueda, en decodificación, en emisión, en ejecución, en escritura y que realizan "commit" en ese ciclo. Cabe subrayar que, por defecto, la categoría de instrucciones que realizan "commit" se encuentra oculta, ya que en ese ciclo, la instrucción ya no se encuentra en ninguna etapa del flujo de ejecución.

- Para representar **el número de instrucciones que han realizado "commit" frente a las descartadas**, un gráfico circular es el más adecuado(Figura 5.11). En

este se muestran tanto los datos absolutos como en porcentaje de todos los ciclos que se han ejecutado hasta el momento.

- Para representar las métricas pormenorizadas por instrucción, no se ha utilizado un gráfico, pues resultaría difícil encontrar uno que pudiera mostrar toda esta información de manera clara. En su lugar, se presenta al usuario una tabla. Tal como se ve en la Figura 5.12, cada entrada de esta tabla representa una instrucción, y en los diferentes campos de cada columna se muestra la posición y mnemónico de la instrucción, así como **la media de ciclos que pasa en cada etapa** y el porcentaje de "commits" frente a descartes. Cabe resaltar, que se optó por mostrar únicamente la media por simplicidad, igualmente, se ha dejado oportunidad al usuario para obtener las otras medidas estadísticas.


Figura 5.9: Gráfico de ocupación de cada unidad por ciclo.

Por otro lado, es esencial tener en cuenta que las máquinas permiten un modo de ejecución en lote, en el cual cada réplica de ejecución no es determinista debido a la existencia de un modelo de caché aleatoria simple. Por lo tanto, es crucial integrar el sistema de estadísticas con este modo de ejecución. Para lograrlo, existen varias aproximaciones posibles:

- Mostrar simplemente las estadísticas de cada réplica de ejecución, permitiendo al usuario seleccionar cuál desea visualizar. Sin embargo, esta opción, además de requerir un cambio en la interfaz, implicaría que el usuario no tendría una visión completa de todas las réplicas, lo que dificultaría evaluar la tendencia en las métricas que sigue su programa en una ejecución en lote.

- Mostrar el agregado de todas las réplicas, utilizando nuevamente alguna medida estadística como la media.

- No integrar las estadísticas con este modo de ejecución, pero esto restaría totalmente la utilidad a este modo, ya que precisamente estas métricas proporcionan al usuario una comprensión de la ejecución de su programa en él.

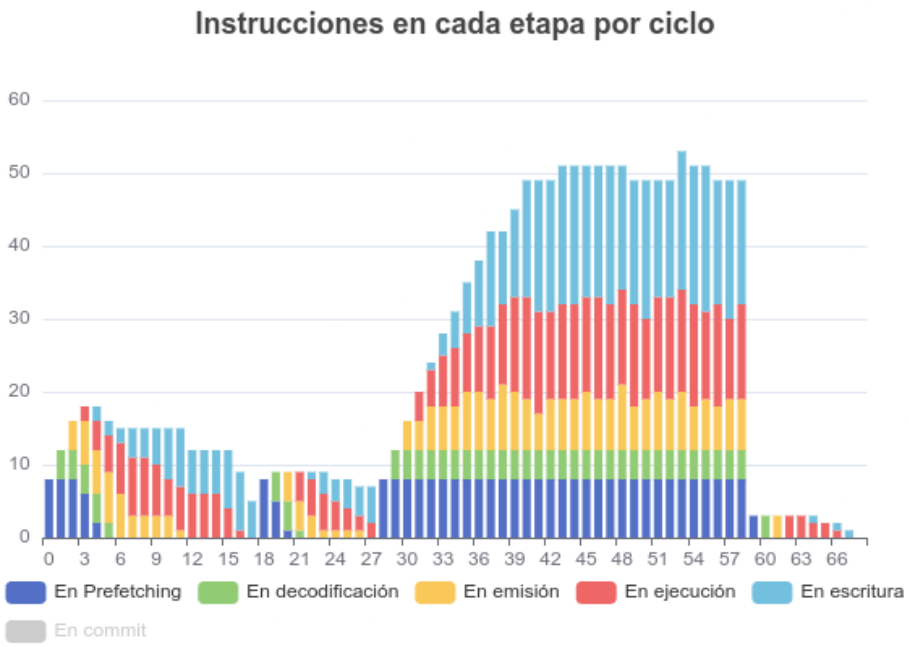



Figura 5.10: Gráfico de instrucciones en cada etapa por ciclo.

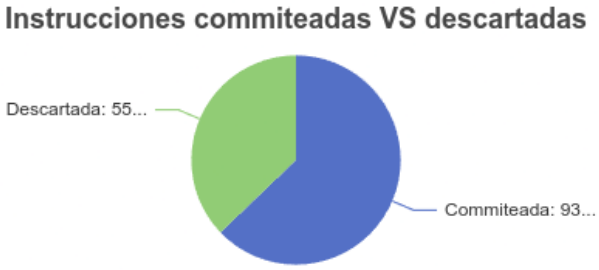



Figura 5.11: Gráfico de instrucciones que han realizado “commit” versus descartadas.

De esta forma, se ha decidido generar la media de los datos y mostrarlos como si fuesen de una ejecución normal. Así, el usuario puede ver la tendencia de las estadísticas según modifique la configuración de caché. Cabe destacar que, igual que antes, se ha dejado de lado mostrar otras medidas importantes, como puede ser la varianza, para evitar complicar la interfaz. No obstante, se ha dejado la puerta abierta a que el usuario pueda obtener estos datos si los necesita.

Es relevante señalar que, debido a la naturaleza no determinista de las réplicas, el número de ciclos, que normalmente es constante en un programa entre ejecuciones, varía. Por lo tanto, para este modo se ha incluido una nueva estadística que muestra **el número de ciclos en cada instancia.** Para ello, se ha implementado un sencillo gráfico de barras que se presenta después de llevar a cabo una ejecución en lote (Figura 5.13).

Finalmente, para concluir este apartado se comentará la manera en la que se ha implementado el sistema de estadísticas. En primer lugar, durante cada ciclo de ejecución, se recopilan datos de la máquina para calcular métricas. Esta recolección no se realiza en el núcleo ni modifica su arquitectura, sino que se realiza en la capa de integración, en donde se le pasa a la clase de estadísticas (*Stats*) los diferentes datos expuestos por la clase de la máquina y unidades. Internamente, esta clase almacena estos datos en estructuras de datos como *Maps*, para luego exponer métodos que permiten calcular

Media de ciclos de cada etapa por instrucción

| # | Código | En Prefetching | En decodificación | En emisión | En ejecución | En escritura | En commit |
|---|---|---|---|---|---|---|---|
| 0 | ADDI R2 R0 #50 | 1 | 1 | 1 | 1 | 1 | 100% |
| 1 | ADDI R3 R0 #70 | 1 | 1 | 1 | 1 | 1 | 100% |
| 2 | ADDI R4 R0 #40 | 1 | 1 | 2 | 1 | 1 | 100% |
| 3 | LF F0 0 (R4) | 1 | 1 | 3 | 6 | 1 | 100% |
| 4 | ADDI R5 R2 #16 | 2 | 1 | 1 | 1 | 7 | 100% |

Figura 5.12: Tabla de media de ciclos por instrucción.


Figura 5.13: Gráfico de ciclos por réplica.

los datos estadísticos que se le pasaran a la interfaz. Debido a esta forma de funcionar, esta clase actúa como un “logger” detallado de datos, que almacena más información que la que se muestra al usuario. Por ello, existe la posibilidad de exportar los datos que contiene esta clase a un fichero JSON[7], para un análisis más profundo. En cambio, para mostrar las métricas en la interfaz en tiempo real, se utilizan acciones de Redux para almacenar los datos en el estado global de la aplicación en cada ciclo de ejecución, permitiendo la actualización instantánea gracias a React. Para visualizar las métricas, se emplea la librería Apache Echarts por su versatilidad y popularidad, integrada mediante React-echarts. Por último, para integrar las estadísticas con el modo de ejecución en lote existe la clase *Aggregator*, que como su nombre indica, proporciona métodos similares a *Stats* para obtener las métricas agregadas en forma de media. Su funcionamiento se basa en que internamente almacena las clases *Stats* de cada réplica, y luego, cuando se calcula una métrica, recorre todas estas clases y calcula la media de los datos que contienen. También permite exportar todos los datos de todas las replicas a un fichero JSON, de forma similar a *Stats*, para permitir un análisis más avanzado al usuario. Por otro lado, para mostrar el número de ciclos en cada réplica, este dato se recolecta y envía al estado global directamente en la capa de integración, y al dejar de estar vacío en el estado global, la interfaz muestra el gráfico de este.

### 5.2.2. Implementación de una caché sencilla

La implementación presente al comienzo de este trabajo de SIMDE contaba con una caché ficticia, que falla aleatoriamente al obtener un dato. Sin embargo, esta implemen-

tación presentaba limitaciones, como su integración directa sobre la clase de memoria, lo que resulta en un acoplamiento de código. Además, solo se activaba en el modo de ejecución por lotes y no permitía modularización para implementar otros modelos de caché. Por lo tanto, para introducir un nuevo modelo de caché más realista, es esencial refactorizar esta parte.

Inicialmente, el código de memoria devolvía una estructura de datos denominada *Datum*, que contenía el dato leído y una bandera que indicaba si se había producido un fallo de caché. Dado que la responsabilidad de la caché recaía sobre la clase *Memory*, sería preferible que esta clase sea totalmente agnóstica a la existencia de una caché. En cambio, las responsabilidades de la caché deberían encontrarse delegadas a su propia clase *Cache*, la cual debería ser gestionada por las máquinas, idealmente en la clase genérica *Machine*.

Para lograr esto de manera elegante, se ha utilizado *Proxy* de JavaScript [15]. Esta herramienta permite interceptar llamadas a funciones, por ejemplo las de lectura y escritura a memoria. Por lo tanto, se ha implementado una clase abstracta padre *Cache*, que expone dos estructuras de datos de tipo *ProxyHandler* que *Machine* utiliza para generar los *Proxy* que interceptan los métodos *getData* y *setData* de *Memory*. De esta forma, la caché intercepta las llamadas a memoria, pudiendo ejecutar su lógica y modifica el atributo booleano *success*, indicando a *Machine* si la última lectura de memoria ha sido exitosa o ha resultado en un fallo de caché (Figura 5.14). La anterior caché aleatoria se ha reimplementado como *RandomCache*, siguiendo esta nueva manera.

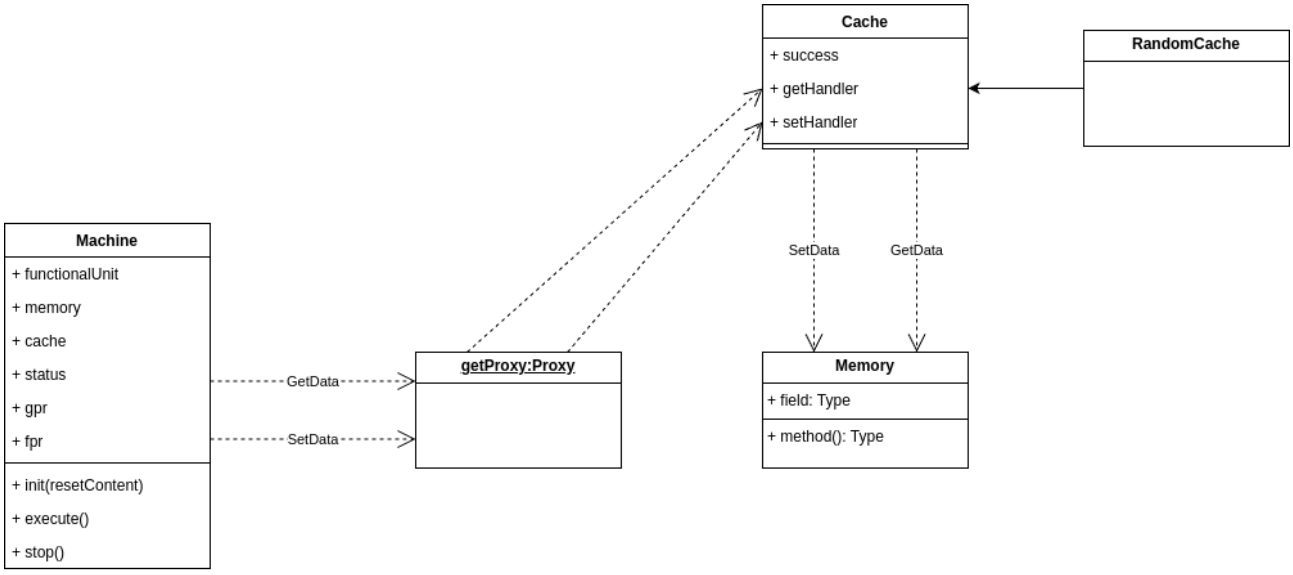



Figura 5.14: Esquema ilustrativo del funcionamiento del sistema de caché con los proxies.

Por otra parte, a nivel de interfaz e integración, se han trasladado las opciones de caché del modal de ejecución en lote al de configuración de la máquina, añadiendo también un menú desplegable que permite seleccionar el modelo de caché o desactivar esta (Figura 5.15). Para integrar esto, se ha empleado un “factory method”[21] para producir la clase de caché correspondiente seleccionada.

A continuación, con un código más modular, es posible examinar los diversos modelos de caché para determinar cuál sería más adecuado de implementar. En este caso, con modelo se hace referencia principalmente a las políticas de ubicación de bloques (también conocidas como “Cache placement policies”) y las políticas de remplazo de bloques. En el primer caso, existen tres políticas principales: Mapeo directo, asociativo y asociativo por

conjuntos. Es subrayable que el mapeo asociativo por conjuntos es un modelo generalista que permite derivar los otros dos mapeos con una determinada configuración. Además, tanto este como el mapeo asociativo, requieren establecer unas políticas de remplazo de bloques. Existen una gran variedad de políticas de remplazo [11, 29], lo cual complica la decisión de cuáles implementar en SIMDE. Por último, también existen las políticas de escritura en caché, generalmente las más conocidas son “Write Through” y “Write Back”.

Debido a que se pretende implantar un modelo sencillo, solamente se integrará el modelo de caché de mapeado directo, evitando así la incorporación de algoritmos de reemplazo de bloques. Por otra parte, se ha decidido usar de forma fija una política de escritura “Write Through” en ambas caches implementadas en SIMDE. De esta forma, el dato se actualiza tanto en la caché como en la memoria en escrituras.


Figura 5.15: Comparación de los diferentes nuevos parámetros de caché mostrados en el modal de configuración de la máquina.

# Capítulo 6

# Conclusiones y líneas futuras

En este apartado se concluirá el trabajo realizado, analizando los resultados obtenidos en forma de conclusiones y proponiendo líneas futuras para este proyecto, tras una revisión de las propuestas previas.

## 6.1. Conclusión

En este trabajo se han implementado pruebas de código funcional, actualizado el marco de pruebas existentes, refactorizado el código con un especial enfoque en el núcleo e implementado nuevas funcionalidades sobre SIMDE.

De esta forma, con este trabajo se ha conseguido una nueva versión de SIMDE con una base de código más estable, limpia y testeada. Además, se ha implementado la generación de estadísticas de ejecución y se ha integrado un nuevo modelo de caché de mapeado directo.

Durante este trabajo se han puesto en práctica aptitudes de desarrollo y mantenimiento de código, además de adquirir conocimientos sobre nuevos lenguajes, herramientas y librerías como Typescript.

## 6.2. Líneas futuras

### 6.2.1. Análisis de las ideas previas

Antes de proponer nuevas líneas de trabajo, sería interesante hacer un análisis de las ideas propuestas por los anteriores TFG.

- **Mayor testeo del código**: Propuesta por [1]. En gran medida esta idea se ha llevado a cabo tanto en [3] como en este trabajo.
- **Mejora en la gestión del estado global**: En [1] ya se comentaba que el actual sistema de gestión de estado no cumple con las expectativas y se sugería rehacerlo. En [14] se sugiere rediseñarlo usando otra librería. También en este propio trabajo se ha comentado el descontento con la actual implementación, por lo que queda claro que esta es una línea de trabajo necesaria para el futuro desarrollo del proyecto.
- **Creación de tutoriales de uso**: Tanto [1] y [6] sugieren la necesidad de crear contenido informativo para facilitar a los usuarios el primer contacto con SIMDE. La necesidad de contenido informativo para facilitar la adopción de SIMDE por parte

de los usuarios es evidente y con el avance y la expansión del simulador, se vuelve cada vez más esencial. De hecho, a día de hoy, la única documentación de uso de SIMDE proviene de la primera versión de este, es decir, de [22].

- **Ludificación del simulador y sistema de ejercicios**: Propuesta por la mayoría de TFG [1, 6, 3] y realizada como parte de [14]. Aunque esta idea fue implementada, no se encuentra integrada en la versión actual del simulador, debido en gran parte a la carga extra de mantenimiento que añadía al proyecto. Sin duda, añadir la necesidad de un Backend y de gestionar usuarios y contenido cargado por estos supone afrontar una cantidad de obstáculos que no parecen razonables en la actualidad ni en un futuro cercano.
- **Desarrollo colaborativo**: Propuesto por [1, 6, 14] y aunque nunca se ha llegado a implementar, la complejidad técnica de implementar y mantener un sistema de sincronización de estado en red plantea dudas sobre su viabilidad y utilidad. Además de que posiblemente se encuentren las mismas dificultades que con la ludificación. Por ello, no parece una línea futura viable.
- **Implementación de más simuladores**: Sin duda esta idea es de vital importancia para seguir expandiendo SIMDE y en perfecta línea con el objetivo perseguido. De hecho, ha sido propuesta por todos los TFG anteriores [1, 6, 14, 3], aunque aún no se han realizado avances notables en esta dirección.
- **Editor de planificación VLIW**: Fue propuesta por [6] y aunque ha habido un pequeño avance en este parte, la creación y edición de la planificación VLIW en SIMDE Web sigue siendo carente y genera una problemática al necesitar depender de SIMDE Legacy para generar éstas.
- **Mejora de la interfaz**: Propuesta por [6] en forma de poder extraer las actuales pestañas en ventanas emergentes y en [3] en forma de mejorar la usabilidad del simulador.
- **Generación de estadísticas de ejecución**: Propuesta por [3] y realizado como parte de este trabajo.
- **Simulación de una memoria caché**: Propuesta por [3] y en este trabajo se ha realizado un avance en esta parte.
- **Modularización de las máquinas**: Citando la propia propuesta original “conllevaría un trabajo considerable y algo desafiante” [3] y, por lo tanto, viendo los actuales retos a los que se enfrenta SIMDE, no parece una idea interesante a perseguir.

### 6.2.2. Líneas futuras propuestas

Visto todas la ideas anteriormente propuestas, ahora se presentan sugerencias para futuras líneas de trabajo, incluyendo algunas no previamente discutidas.

- **Eliminación del estado global**: Esta propuesta disruptiva con respecto a los anteriores autores, proponemos eliminar el estado global completamente, argumentando que actualmente añade una capa innecesaria entre la integración y la interfaz. Además, si se analiza la propia documentación de Redux respecto a esto, se observa

que SIMDE no es un caso de uso adecuado, citándola “Not all apps need Redux. It’s important to understand the kind of application you’re building, the kinds of problems that you need to solve, and what tools can best solve the problems you’re facing.” [26]

- **Implementación de más simuladores**: Este trabajo he pretendido allanar el camino para que futuros contribuidores de este proyecto puedan implementar nuevas máquinas fácilmente. Son numerosas los tipos de máquinas que se han sugerido, pero actualmente implementar una máquina segmentada sería lo más adecuado como siguiente paso, tanto por su sencillez como por su inmediata utilidad en otras asignaturas de la universidad, en este caso Estructuras de computadores.

- **Editor de planificación VLIW**: Debido a que la planificación VLIW es una tarea tediosa y propensa a errores, es de suma importancia para el futuro de esta máquina realizar un estudio de UX e implementar una interfaz cómoda y extensa para la creación y edición de planificaciones.

- **Mejora de la interfaz**: Aunque en este trabajo y anteriores [3] se ha realizado un amplio mantenimiento a este proyecto, la interfaz presenta múltiples apartados de mejora, pues los esfuerzos se han centrado en otros apartados. Por lo que como futura línea en cuanto a mantenibilidad sería interesante realizar una refactorización sobre la interfaz. Por ejemplo, implementando los componentes de React en forma de “Function Components” en vez de “Class Components” [8], integrando de mejor manera la librería de estilo Bootstrap [24] e intentando usar un sistema de gestión de ventanas como Golden Layout [9].

- **SIMDE CLI**: Una línea futura interesante sería la creación de una interfaz de línea de comandos para SIMDE. Actualmente, SIMDE se encuentra bien dividido entre interfaz y núcleo, por lo que no supondría una gran dificultad crear una interfaz de línea de comandos que permita ejecutar las mismas funcionalidades que la interfaz web, devolviendo los resultados en formato JSON o similar. Esto permitiría usar de manera sencilla SIMDE en scripts y automatizaciones, lo que abriría un nuevo abanico de posibilidades de uso. De hecho, para suplir ciertas de estas, existe SIMDELite [4], que proporciona ciertas funcionalidades de SIMDE en forma de herramienta de terminal, pero no se encuentra integrada en el proyecto.

- **Soportar arquitectura RISC-V**: Esta nueva arquitectura está ganando una notable popularidad en la actualidad, sobre todo en el ámbito académico y de enseñanza, de hecho libros de texto de suma importancia como “Computer Organization and Design RISC-V Edition” [17] ya están empezando a usarla y dan una buena justificación para utilizarla en su prólogo. También, se comentó al analizar Ripes, la comunidad detrás de RISC-V podría mostrar un interés por SIMDE si se implementara esta arquitectura. Además, esto permitiría experimentar con código generado por compiladores sobre SIMDE.

# Capítulo 7

# Summary and Conclusions

This section will conclude the work performed by presenting the results obtained in the form of conclusions and proposing future lines for this project, after a review of previous proposals.

## 7.1. Conclusions

In this work, we have implemented functional code tests, updated the existing testing framework, refactored the code with a special focus on the core, and implemented new functionalities on SIMDE. As a result, a new version of SIMDE with a more stable, clean, and tested code base has been produced. Furthermore, the generation of execution statistics has been implemented, and a new direct mapping cache model has been integrated.

During this undertaking, the abilities to develop and maintain code, as well as to acquire knowledge of new languages, tools and libraries such as Typescript, have been put into practice.

## 7.2. Future lines of work

### 7.2.1. Previous proposals

Prior to proposing new lines of work, it would be beneficial to conduct an analysis of the ideas proposed by previous works.

- **More code testing**: Proposed by [1]. To a large extent this idea has been carried out both in [3] and in this work.
- **Better global state**: In [1] it was already commented that the current global state does not meet the expectations, and it was suggested to redo it. In [**?** ] it is suggested to redesign it using another library. Also in this own work the dissatisfaction with the current implementation has been commented, so it is clear that this is a necessary line of work for the future development of the project.
- **Adding tutorials**: Both [1] and [6] suggest the need to create informative content to facilitate users' first contact with SIMDE. The need for tutorials to facilitate user adoption of SIMDE is evident and with the advancement and expansion of the simu-

lator, it becomes increasingly essential. In fact, as of today, the only documentation on the use of SIMDE comes from the first version of SIMDE, i.e., from [22].

- **Simulator gamification and exercise system**: Proposed by the majority of previous works [1, 6, 3] and realized as part of [14]. Although this idea was implemented, it is not integrated into the current version of the simulator, largely due to the extra maintenance burden it added to the project. Undoubtedly, adding the need for a backend and managing users and user-loaded content means facing a number of hurdles that do not seem reasonable at present or in the near future.
- **Collaborative tools**: Proposed by [1, 6, 14] and although never actually implemented, the technical complexity of implementing and maintaining a networked state synchronization system raises doubts about its feasibility and usefulness. In addition to possibly encountering the same difficulties as with gamification. Therefore, it does not appear to be a viable future line.
- **More simulators**: Undoubtedly this idea is of vital importance to continue expanding SIMDE and in perfect line with the objective pursued. In fact, it has been proposed by all previous works [1, 6, 14, 3], although no notable progress has yet been made in this direction.
- **VLIW Planning Editor**: It was proposed by [6] and although there has been a small advance in this part, the creation and edition of the VLIW planning in SIMDE Web is still lacking and generates a problem by needing to depend on SIMDE Legacy to generate them.
- **Better interface**: Proposed by [6] in the form of being able to extract the current tabs in popup windows and by [3] in the form of improving the usability of the simulator.
- **Generation of execution statistics**: Proposed by [3] and carried out as part of this work.
- **Simulation of a cache memory**: Proposed by [3] and in this work an advance in this part has been made.
- **Modularization of machines**: To quote the original proposal itself “it would entail considerable and somewhat challenging work” [3] and, therefore, looking at the current challenges faced by SIMDE, it does not seem an interesting idea to pursue.

### 7.2.2. Our proposals

In light of the preceding ideas, suggestions for future lines of work, including some that have not previously been discussed, are now presented.

- **Suppression of the global state**: In this disruptive proposal with respect to the previous authors, we propose to remove the global state completely, arguing that it currently adds an unnecessary layer between the integration and the interface. Furthermore, looking at Redux’s own documentation with respect to this, we note that SIMDE is not a suitable use case, citing it “Not all apps need Redux. It’s important to understand the kind of application you’re building, the kinds of problems that you need to solve, and what tools can best solve the problems you’re facing.” [26]

- **More simulators**: This work has finally paved the way for future contributors to this project to easily implement new machines. Numerous types of machines have been suggested, but currently implementing a segmented machine would be the most appropriate next step, both for its simplicity and its immediate usefulness in other university courses, i.e. Computer Structures.
- **VLIW Planning Editor**: Because VLIW planning is a tedious and error-prone task, it is of utmost importance for the future of this machine to perform a UX study and implement a convenient and extensive interface for creating and editing the planning.
- **Better interface**: Although in this work and previous [3] a wide maintenance has been made to this project, the interface presents multiple sections of improvement, since the efforts have been centered in other sections. Therefore, as a future line in terms of maintainability, it would be interesting to perform a refactoring on the interface. In this way, implementing the React components in the form of "Function Components" instead of "Class Components" [8], integrating in a better way the Bootstrap style library [24] and trying to use a window management system such as Golden Layout [9].
- **SIMDE CLI**: An interesting future line would be the creation of a command line interface for SIMDE. Currently, SIMDE is well divided between interface and core, so it would not be a great difficulty to create a command line interface that would allow to execute the same functionalities as the web interface, returning the results in JSON or similar format. This would allow the easy use of SIMDE in scripts and automations, which would open a new range of possibilities of use. In fact, to supply some of these possibilities, there is SIMDELite [4], which provides some SIMDE functionalities in the form of a terminal tool, but it is not integrated in the project.
- **RISC-V support**: This new architecture is gaining remarkable popularity nowadays, especially in academia and teaching, in fact textbooks of utmost importance such as "Computer Organization and Design RISC-V Edition" [17] are already starting to use it and give a good justification for using it in their foreword. Also, it was commented when discussing Ripes, the community behind RISC-V might show an interest in SIMDE if this architecture were implemented. In addition, this would allow experimentation with compiler-generated code on SIMDE.

# Capítulo 8

# Presupuesto

A continuación se presenta, en la Tabla 8.1, el presupuesto estimado para la realización del proyecto. Es destacable que el precio por hora incluye la amortización del ordenador de trabajo y la cuota de autónomo.

| Horas de trabajo | Precio por hora | Total |
|---|---|---|
| 200 | 40 € | 8000 € |

Tabla 8.1: Presupuesto general.

# Bibliografía


[1] Abreu González, A. Simulador didáctico de arquitectura de computadores. *Repositorio institucional de la universidad de La Laguna* (2017).

[2] Andersson, R. RSM. `https://github.com/rsms/rsm`, 2022.

[3] Carrasco Benítez, Ó. Simulador didáctico de arquitectura de computadores: aplicación de metodologías de integración y mantenimiento. *Repositorio institucional de la universidad de La Laguna* (2022).

[4] Castilla Rodríguez, I. Simde-vliw-lite. `https://github.com/icasrod/SIMDE-VLIW-Lite`, 2021.

[5] Castilla Rodríguez, I., Abreu González, A., Díaz Arteaga, M., Carrasco Benítez, Ó., and La Spina, F. Simulador para planificación dinámica y estática. `https://github.com/SIMDE-ULL/SIMDE`, 2017.

[6] Díaz Arteaga, M. Simulador didáctico de una arquitectura de planificación estática. *Repositorio institucional de la universidad de La Laguna* (2019).

[7] ECMA. 404: The json data interchange syntax. *ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr,* (2017).

[8] GeeksforGeeks. Differences between functional components and class components. `https://www.geeksforgeeks.org/differences-between-functional-components-and-class-components/`, 2023.

[9] Golden Layout community. Golden Layout. `https://github.com/golden-layout/golden-layout`, 2024.

[10] Gupta, S., Shukla, S., Kumar, A., et al. CENOS: The Modern CPU Simulator. `https://github.com/0mega28/CPU-Simulator`, 2022.

[11] Jain, A., and Lin, C. *Cache replacement policies*. Synthesis lectures on computer architecture. Springer International Publishing, Cham, Switzerland, June 2019.

[12] K. Ardestani, E., and Renau, J. ESESC: A Fast Multicore Simulator Using Time-Based Sampling. In *International Symposium on High Performance Computer Architecture* (2013), HPCA'19.

[13] Koren, I., et al. Computer architecture educational tools. `https://www.ecs.umass.edu/ece/koren/architecture/`, 1998.

[14] López Garnier, A. J. Plataforma de ludificación de un simulador didáctico de arquitectura de computadores. *Repositorio institucional de la universidad de La Laguna* (2018).

[15] MDN contributors. Proxy - javascript. https://developer.mozilla.org/enUS/docs/Web/JavaScript/Reference/Global_Objects/Proxy, 2023.

[16] Oracle Corporation. Applets. *Java Technical Details* (1995).

[17] Patterson, D. A., and Hennessy, J. L. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2017.

[18] Petersen, M. B. Ripes: Teaching computer architecture through visual and interactive simulators. In *RISC-V Summit 2020* (2020), RISC-V International.

[19] Petersen, M. B. Ripes: A visual computer architecture simulator. In *2021 ACM/IEEE Workshop on Computer Architecture Education (WCAE)* (2021), IEEE, pp. 1–8.

[20] Petersen, M. B., et al. Ripes. https://github.com/mortbopet/Ripes, 2018.

[21] Refactoring.Guru. Factory method en typescript. https://refactoring.guru/es/design-patterns/factory-method/typescript/example/, 2024.

[22] Rodríguez, I. C., Ruiz, L. M., Saavedra, J. F. S., González, C., and González, E. J. Simde: un simulador para el apoyo docente en la enseñanza de las arquitecturas ilp con planificación dinámica y estática. In *JENUI 2004* (2004).

[23] Stack Exchange, inc. Stack Overflow Developer Survey 2023. https://survey.stackoverflow.co/2023/#most-popular-technologies-language, 2023.

[24] The Bootstrap Authors. Bootstrap. https://github.com/twbs/bootstrap, 2011.

[25] The Qt Company. Qt Framework. https://www.qt.io/product/framework, 1995.

[26] The Redux documentation authors. Redux faq: When should i use redux? https://redux.js.org/faq/general#when-should-i-use-redux, Nov 2023.

[27] Tomasulo, R. M. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development 11*, 1 (1967), 25–33.

[28] W3C WebAssembly Working Group. WebAssembly. https://webassembly.org/, 2017.

[29] Wikipedia contributors. Cache replacement policies — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Cache_replacement_policies&oldid=1202544636, 2024.